**Proceedings of the IASTED International Conference
Parallel and Distributed Systems - Euro-PDS '97
June 9-11, 1997 - Barcelona, Spain**

Derivation of Efficient Parallel Algorithms on a Ring of Processors

Ali E. Abdallah

Department of Computer Science
The University of Reading
Reading, RG6 6AY, UK
email: A.Abdallah@reading.ac.uk

Theoharis Theoharis

Department of Informatics
The University of Athens
Panepistimioupolis, 15771 Athens, Greece
email: theotheo@di.uoa.gr

Abstract

A binary operator which takes two lists as arguments is called multiscan if every element of the first list must be considered in conjunction with every element of the second list in order to produce the result. Several problems such as the relational database operators join, intersection, and difference can be expressed as specific instances of multiscan. In this paper we consider a generic functional definition of multiscan and show how it can be implemented as a network of communicating sequential processes (CSP) with a ring configuration. We examine issues which affect the performance of the parallel implementation and identify two properties which, if possessed by a multiscan operator, allow the derivation of an efficient scalable parallel implementation on a ring of processors. A practical illustration from the field of relational data bases is given.

Keywords: Functional programming; Generic parallel algorithms; Communicating sequential processes; Ring networks.

1 Introduction

Over the past decade, a fair amount of research efforts have been directed at the development of parallel algorithms for a variety of application domains. Parallelisation usually involves the introduction of communication between parallel tasks. The communication cost is usually greater the more dependent the tasks are. The communication structure of an algorithm is directly dependent on the topology of the underlying parallel hardware and thus algorithms are associated with pipelines, rings, trees, hypercubes etc.

Although individual applications are useful and interesting per se, the generalisation and formal treatment of each topology should help us to rapidly answer questions such as:

- Is application A suitable for parallel implementation on topology T?
- How much do we expect to gain from the parallel implementation of application A on topology T?
- Can a generic algorithm be constructed for computation on topology T?

A formal treatment of the derivation of efficient parallel algorithms from high-level functional specifications on pipeline topology, has been given in [1, 2, 3]. The current paper addresses the ring topology. This is not a simple extension of the pipe, as the addition of a single link between the end elements might suggest, for several reasons. Firstly, the symmetry of the ring imposes symmetry on the parallel algorithms if they are to be efficient. Secondly, the ring introduces the possibility of deadlock. Thirdly, the result of the pipeline computation is normally output by the final stage of the pipeline. In contrast, the result of a ring computation is distributed among its processors, each storing in its local memory a part of the result.

This paper examines ring computation by modeling the ring data elements as lists of a fixed length p. It considers the parallel computation of a generic functional form called *multiscan* (see Section 3) and identifes two properties under which the functional form can be efficiently implemented as a ring network of communicating sequential processes. A case study from the field of relational databases illustrates the approach.

2 Notation and Preliminaries

Throughout this paper, we use the functional notation and calculus developed by Bird and Meertens

[5, 6, 9] for specifying algorithmics and reasoning about them and will use the CSP notation and its calculus developed by Hoare [18] for specifying processes and reasoning about them. We give a brief summary of the notation and conventions used in this paper. The reader is advised to consult the above references for details.

Lists are finite sequences of values of the same type. The list concatenation operator is denoted by $+\!\!+$ and the list construction operator is denoted by :. The elements of a list are displayed between square brackets and separated by comas. Functions are usually defined using higher order functions or by sets of recursive equations. The operator $*$ (pronounced "map") takes a function on the left and a list on the right and applies the function to each element of the list. Informally, we have:

$$f * [a_1, a_2, \cdots, a_2] \quad = \quad [f(a_1), f(a_2), \cdots, f(a_n)]$$

The operator / (pronounced "reduce") takes an associative binary operator on the left and a list on the right. It can be informally described as follows

$$(\oplus)/[a_1, a_2, \cdots, a_n] \quad = \quad a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

The left reduction operator $(\oplus \not\rightarrow e)$ is a special case of reduction in which the computation of a list starts with e as an initial value and gradually accumulates the result by traversing the list from left to right. Informally, $(\oplus \not\rightarrow e)[a_1, a_2, \cdots, a_n]$ is:

$$(\cdots((e \oplus a_1) \oplus a_2) \oplus \cdots) \oplus a_n$$

Given a binary operator $\oplus$, the zipwith operator $\nabla_\oplus$ takes two lists of the same length $[a_1, a_2, \cdots, a_p]$ and $[b_1, b_2, \cdots, b_p]$ and returns the following list:

$$[a_1 \oplus b_1, \ a_2 \oplus b_2, \ \cdots, \ a_p \oplus b_p]$$

3 Efficient Implementation of Multi-scan Operators on a Ring

3.1 The problem

A binary operator $\otimes$ which takes two lists xs and ys as arguments will be called *multiscan* if every element in the list xs must be considered in conjunction with every element of list ys in order to produce the result. This term was coined in the relational database field [13]. A multiscan operator $\otimes$ is usually computed using a simpler operator $\oplus$ as follows:

$$[x_1, x_2, \cdots, x_p] \otimes ys = [x_1 \oplus ys, \ x_2 \oplus ys, \ \cdots, \ x_p \oplus ys]$$

Formally, we have

$$xs \otimes ys = xs \ \nabla_\oplus \ [ys \mid i \leftarrow [1..\#xs]\ ]$$

The elements of the result on the right hand side of the above expression can be computed in parallel on p processors by assigning the computation of $x_i \oplus ys$ to processor i.

Many important operators such as those used in relational databases (join, intersection, difference) and matrix manipulation (multiplication, closure) belong to this class.

3.2 The issues

In order to efficiently perform the above parallel computation we have to provide appropriate treatments to the following issues:

1. *Communication.* The value of ys is required by every processor and thus needs to be communicated. Assuming the size of ys is n, a total of np values need to be communicated to the processing elements. If the communication is not handled efficiently this could cost $O(np)$ time steps.

2. *Computation/Communication Scheduling.* The communication and computation must be arranged so that a high processor utilisation is achieved. Typically, the total sequential computation effort involved in executing $\otimes$ is $O(n^2)$. If communication is not properly interwined with computation there is danger of low processor utilisation due to processors being idle while waiting for data to be communicated. In the worst case the parallel algorithm could take $O(n^2)$ time steps to execute.

3. *Space.* In many applications, such as data bases, the size of xs and ys can be very large. Space then becomes an important practical consideration for the choice of algorithm. In scalable parallel systems it is not reasonable to assume that the space available on each element is enough to accommodate all the data for the algorithm.

3.3 The Basic Solution

The basic solution is a parallel composition of p parallel processes with a ring configuration. It assumes that the lists xs and ys have the same length, say p, and the i^{th} elements of the lists xs and ys are stored in the i^{th} processor in the ring. The algorithm consists of two phases. The first phase aims at communicating the list ys to all the processors.

The second phase actually computes the result. To communicate the list ys, we can take advantage of the ring topology by utilizing all of the p ring links in parallel so that the p^2 values are communicated in the ring in p time steps. At each step the list ys is rotated clockwise around the ring of processors. Care needs to be taken because the elements of ys will be arriving in a different order at each processor. This order is a circular shift of the original list. After $p-1$ such rotations the whole of the list ys will have passed through and been stored by each processor. At this point each processing element has all the relevant data for the computation of the required part of the result (i.e. x_i and ys). Therefore, all the processors can now proceed in parallel with the the i^{th} processor performing the computation of $x_i \oplus ys$.

The i^{th} ring process R_i holds two values x and y, being the i^{th} elements of the lists xs and ys respectively. Its main purpose is to compute r, the value of the i^{th} element of the resulting list. The behaviour of R_i can be viewed as a sequence of two phases, C_i representing the communication phase and P_i representing the processing phase. The behaviour of these processes is formally captured as follows:

$$\begin{array}{lcl} R_i(x,y) & = & C_i(y,[]); P_i(x) \\ C_i(y,ys) & = & \text{seq}_{j=1}^{p} \; (\;\; ((in?y' \rightarrow SKIP) \\ & & \qquad\qquad \parallel (out!y \rightarrow SKIP)); \\ & & \qquad\quad y := y'; \;\; ys.(i \ominus j) := y') \\ P_i(x) & = & r := x \oplus ys \end{array}$$

where the operator $\ominus$ is subtraction modulo p and the notation $s.i$ denotes the i^{th} element of the list s. The behaviour of the ring is just a parallel composition of the p ring processes.

The above computation will be completed in $O(p)$ time steps by all processors. Thus the total time required by the algorithm is $O(pc+p)$ where c represents the cost of a communication step. It is clear that the speedup of this algorithm is p.

This algorithm provides good solutions to the first two issues mentioned above. However, there are two important drawbacks. First, each processor must have sufficient space to store the whole of the list ys. Second, there is high *latency* before starting the generation of the result because this cannot commence until the expensive communication phase has completely finished.

3.4 An Improved Solution

In the previous algorithm the behaviour of each ring process was split into two separate phases. The first is dedicated to communication and the second is dedicated to computation. Often ring computation can be improved by carefully interwining computation and communication so that each ring station makes some computational progress after each communication. This is only possible if the multiscan operator $\oplus$ satisfies certain conditions. To determine these conditions let us carefully consider the behaviour of each ring process i.e. the list of values it receives and the result it computes.

Clearly the i^{th} ring process R_i receives as input the elements of the list ys in a certain order[1], see Figure 1. This order is captured by the function $rot\ i$ defined as follows:

$$rot\ i\ s = rev\ (take\ i\ s) + \!\! + \ rev\ (drop\ i\ s)$$

$rot\ i$ is the function which gives the i^{th} rotation (clockwise circular shift) of its list argument and can be informally defined as $rot\ i\ [s_1, s_2 ... s_p] = [s_i, s_{i-1} ... s_1, s_p, s_{p-1} ... s_{i+1}]$.

The result which should be computed by ring station i is $x_i \oplus ys$. In order to avoid the storage of the whole list ys in station i the result should be accumulated on the fly. Station i receives a rotation of the list ys and computes:

$$x_i \ \tilde{\oplus}_i \ rot\ i\ ys$$

The oparator $\tilde{\oplus}_i$ should be defined so that

$$x_i \ \tilde{\oplus}_i \ rot\ i\ ys = x_i \oplus ys \qquad\qquad (C1)$$

A binary operator $\oplus$ is called *rotation invariant* if its result is independent of the particular rotation of its right hand side list argument. Formally:

$$x \oplus ys = x \oplus rot\ i\ ys, \qquad \forall i \in \{1...p\}$$

If $\oplus$ is rotation invariant then condition $C1$ can easily be satisfied by simply taking $\tilde{\oplus}_i = \oplus$.

It is highly desirable that the operator $\oplus$ is *incrementally computable* ($C2$) i.e. part of the result can be computed as soon as an element of ys becomes available. Typically, although not exclusively, an incrementally computable operator $\oplus$ can be described as a left reduction on ys as follows:

$$x \oplus ys = (\odot_x \not\rightarrow e_x)\ ys$$

where $\odot_x$ is a binary operator which depends on the value x and e_x is the initial value of the computation. This property has two advantages. Firstly, storage for only one element of ys per ring station is required and secondly, the result is incrementally built-up and

[1] As can be seen in the CSP description below, this can be optimised as a station does not need to receive the element of the list which resides in itself.

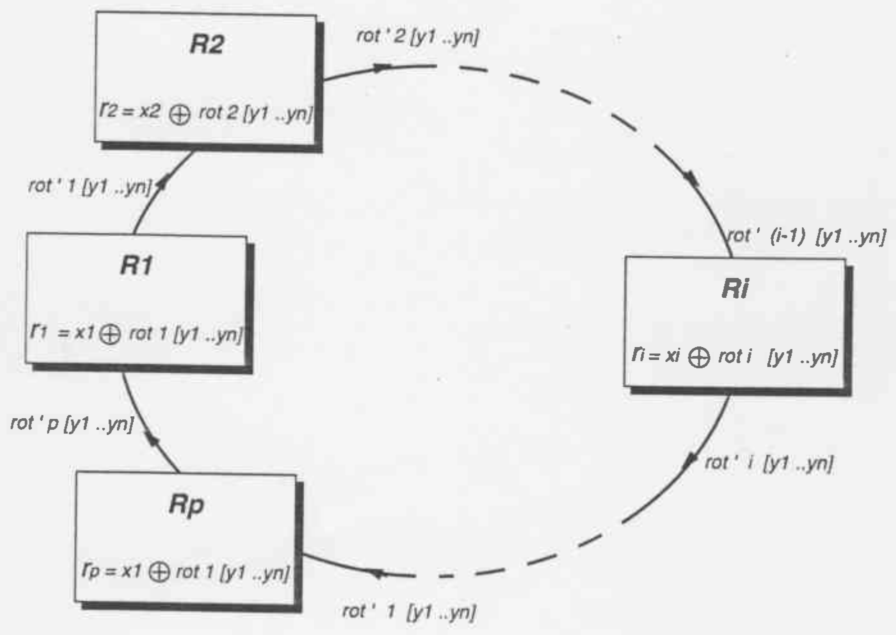

Figure 1: Computation and Communication on a Ring (the list *rot' i s* is *rot i s* without the i^{th} element).

can thus be incrementally extracted; this is attractive in systems where multiple I/O operations can proceed in parallel or where I/O and computation can be overlapped (e.g. Transputer systems).

Ring processing, as defined above, requires $p-1$ parallel communication operations, reducing the total communication cost to $O(p)$.

Let us now describe the functionality of a ring station. A ring station performs alternately:

- one step of ring communication (input from left and output to right)
- one computation step which updates the cumulative result for the new input

The above steps are repeated $p-1$ times i.e. as many as are necessary to correctly compute the final result.

A typical ring station R holding a binary operator $\odot$ for the reduction computation, an initial value r for accumulation of the result, and a value y for the most recent input can be depicted as in Figure 2.

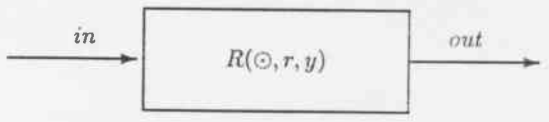

Figure 2: A Ring Station

Its behaviour can be captured in CSP as follows:

$$\alpha R \quad = \quad \{in, out\}$$

$$R(\odot, r, y) \quad = \quad r := r \odot y; \mathrm{seq}_{i=1}^{p-1}\ STEP$$

$$STEP \quad = \quad (in?y' \rightarrow SKIP) \parallel (out!y \rightarrow SKIP)\ ; \quad y := y';\ r := r \odot y$$

The final value of r residing in the ring station is:

$$(\odot\!\not\rightarrow\! r)\ (rot\ i\ ys)$$

The whole ring $RING(xs, ys, \odot, e)$ with p stations can be described as

$$\|_{i=1}^{p}\ R(\odot_{xi}, e_{xi}, yi)[c_i/in, c_{(i+1) \bmod p}/out]$$

where xi and yi are the i^{th} elements of the lists xs and ys respectively, $\odot_x$ is a binary operator which may depend on x, and e_x is the initial value for computing the reduction. Figure 3 gives an illustration of the sequence of steps performed in ring processing.

Deadlock is avoided by not ordering the input and output phases of each ring station i.e. executing input and output in parallel using double-buffering. The parallel functioning of communication links is increasingly available on modern processors used for building parallel systems.

4 Applications

In a parallel relational database system using p processors, each relation r is normally partitioned into p pairwise disjoint parts which physically reside in the local storage units of p processor modules. This is refered to as a horizontal partitioning and is extensively used in concrete parallel databases such as [12, 25]. At an abstract level, we can view this as changing the representation of relations from a single set of tuples to a fixed number p of such sets. For this representation, the definitions of several relational operators such as *intersection*, *difference*, and *join* can be naturally expressed as multiscan operators as defined in Section 3. For example, the intersection operator over two horizontally partitioned relations, say $r = [r_1, r_2, ..r_{p-1}, r_p]$ and $s = [s_1, s_2, ..s_{p-1}, s_p]$, is defined as:

$$r \cap s = [r_1 \oplus s, r_2 \oplus s, \cdots, r_p \oplus s]$$

where the operator $\oplus$ is described as:

$$x \oplus [s_1, s_2, .., s_p] = x \cap \bigcup_{i=1}^{p} s_i$$

It is clear that relational intersection $\cap$ is a multiscan operator such that

$$r \cap s = r \nabla_{\oplus} \underbrace{[s, s, .., s]}_{p \text{ times}}$$

The operator $\oplus$ can be easily described as a left reduction and is thus, incrementally computable. We have:

$$a \oplus s = (\odot_a \not\rightarrow \{\})\ s$$

where $x \odot_a b = x \cup (a \cap b)$. Finally, $\oplus$ is rotation invariant due to the commutativity of $\cap$. Therefore, a highly parallel algorithm for this problem can be derived by simply instantiating the generic solution on the ring with the specific values of the parameters $\odot_x$ and e. Figure 3 depicts the execution of the intersection algorithm for two horizontally partitioned sets over three processors, $x = [[2,5],[1,4],[0,3]]$ and $y = [[5,3],[9,8],[2,0]]$, with the final result being stored in r.

The database operators *difference* and *join* can be defined as different instances of the ring pattern which satisfy the two required conditions. Hence, efficient parallel ring algorithms can be synthesized.

Many other problems can be described as instances of the ring pattern including the cartesian product of two sets, matrix multiplication and matrix closure algorithms.

5 Related Work

This paper has been profoundly influenced by the work of Bird and Meertens [5, 6, 7, 8] on developing a calculus for program synthesis and the work of Hoare [18] on developing a calculus for communicating sequential processes. It fits extremly well with ideas advocated by several other researchers such as Cole [10], Skillicorn [23], Darlington [11], Lenguaer and Gorlatch [15] and Misra [21] which attempt at describing parallel algorithms at a high level of abstraction and use program transformation techniques to derive efficeint parallel implementations for specific architectures. Skillicorn [22] proposed Bird-Meertens Formalism (BMF) as a coherent approach to the development of data parallel algorithms. Gorlatch [16] showed how BMF can be used to derive efficient parallel implementation schema of *distributed homomorphisms* on a hypercube of processors. Misra [21] introduced a data structure called *powerlist* together with its algebra, and used it within a functional setting to elegantly describe several divide-and-conquer data-parallel algorithms. Darlington, Kelly [11, 19], Mou and Huddak [20], Harrison and Guzman [17] have used functional notations and algebraic laws to develop parallel functional programs. Our work goes a step further in refining the final functional version into networks of CSP processes.

6 Conclusion

We have given a generic functional definition of *multiscan* operators and showed how it can be implemented as a ring network of communicating processes in CSP. We have examined communication issues in the network which can drastically reduce its performance. We have identified two conditions on the parameters of the functional form which guarantee its efficient and scalable parallel implementation

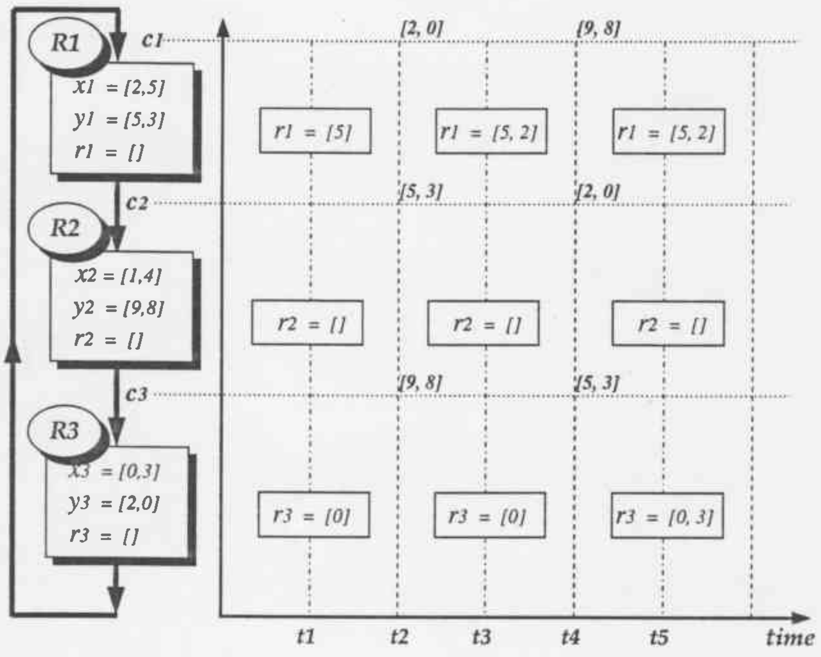

Figure 3: Timing diagram for set intersection

on a ring of processors. The parameters of the functional form can be instantiated to efficiently implement a number of multiscan operators provided that they are *rotation invariant* and *incrementally computable.* Several useful applications can be defined in terms of such operators.

Acknowledgements

The authors would like to thank *The British Council* for financially supporting their joint research proramme between The University of Reading and Athens University. The second author is supported by The University of Athens research grant 70/4/2401.

References

[1] A. E. Abdallah, Derivation of Parallel Algorithms from Functional Specifications to *CSP* Processes, in: Bernhard Möller, ed., *Mathematics of Program Construction*, LNCS **947**, (Springer Verlag, 1995) 67-96.

[2] A. E. Abdallah, Synthesis of Massively Pipelined Algorithms for List Manipulation, in L. Bouge and P. Fraigniaud and A. Mignotte and Y. Robert (eds), Proceedings of the *European Conference on Parallel Processing, EuroPar'96*, LNCS **1024**, (Springer Verlag, 1996), pp 911-920.

[3] A. E. Abdallah, and T. Theoharis, Synthesis of Massively Pipelined Algorithms from Recursive Functional Programs, in: K. Li, T.S. Abdelrahman, E. Luque, eds., Proceedings of the Eighth IASTED International Conference on Parallel and Distributed Computing and Systems, Chicago, USA, (IASTED/ACTA press, October 1996), 500-504.

[4] G.S. Almasi and A. Gottlieb, *Highly Parallel Computing*, 2nd Edition, (Benjamin/Cummings Publishing Company, 1994).

[5] R. S. Bird, An Introduction to the Theory of Lists, in M. Broy, ed., *Logic of Programming and Calculi of Discreet Design*, (Springer, Berlin, 1987) 3-42.

[6] R. S. Bird, Functional Algorithm Design, in: Bernhard Möller, ed., *Mathematics of Program Construction*, LNCS **947**, (Springer Verlag, 1995) 2-17.

[7] R. S. Bird, and L. G. L. T. Meertens, Two Exercices Found in a Book on Algorithmics. in L. G.

L. T. Meertens, ed., *Program Specification and Transformation.* (North Holland, 1986)

[8] R. S. Bird, and O. de Moor, *The Algebra of Programming,* (Prentice-Hall, 1996).

[9] R. S. Bird, and P. Wadler, *Introduction to Functional Programming,* (Prentice-Hall, 1988).

[10] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation* (Pitman, 1989).

[11] J. Darlington, A. J. Field, P.G. Harrison, and P.H.J. Kelly, Q. Wu, and R.L. While, Parallel Programming Using Skeletons Functions, in *PARLE93, Parallel Architectures and Languages Europe* LNCS , (Springer-Verlag, 93).

[12] D. DeWitt, et al., Tha Gamma Database Machine Project, *IEEE Trans. on Knowledge and Data Engineering,* Vol. 2, no. 1, March 1990, pp.44-61.

[13] O. Frieder, Multiprocessor Algorithms for Relational-Database Operators on Hypercube Systems *IEEE Computer*, November 1990, pp.13-28.

[14] I. Foster, *Designing and Building Parallel Programs* (Addison Wesley, 1995).

[15] S. Gorlatch and C. Lengauer, Parallelization of Divide-and-Conquer in the Bird-Meertens Formalism, *Formal Aspects of Computing* **7** (6) (1995) 663-682.

[16] S. Gorlatch Systematic Efficient Parallelization of Scan and Other List Homomorphisms, in L. Bouge and P. Fraigniaud and A. Mignotte and Y. Robert (eds), Proceedings of the *European Conference on Parallel Processing, Euro-Par'96*, LNCS **1124**, (Springer-Verlag, 96) 401-408.

[17] I. P. de Guzman, P.G. Harrison, and E. Medina, Pipelines for Divide-and-Conquer Functions, *The Computer Journal,* **36** (3) (1993).

[18] C. A. R. Hoare, *Communicating Sequential Processes.* (Prentice-Hall, 1985).

[19] Paul Kelly, *Functional Programming for Loosley-Coupled Multiprocessors.* Research Monographs in Parallel and Distributed Computing, (Pitman, 1989).

[20] Z.G. Mou, and M. Hudak, An Algebraic Model for Divide-and-Conquer Algorithms and its Parallelism, *Journal of Supercomputing,* **2** (3) (1988).

[21] J. Misra, Powerlist: A Structure for Parallel Recursion, *ACM TOPLAS* **16** (6) (1994).

[22] D. B. Skillicorn, Models for Practical Parallel Computation, *International Journal of Parallel Programming* **20** (2) (1991) 133-158.

[23] D. B. Skillicorn, *Foundations of Parallel Programming,* (Cambridge University Press, 1994).

[24] M. Stonebraker, The Case for Shared Nothing, *IEEE Database Engineering,* March 1986, pp. 4-9.

[25] T. Theoharis and A. Paraskevopoulos, PARDB: Design, Algorithms and Performance of a Transputer Based Parallel Relational DBMS, *submitted to Transputer Communications,*