The Torus:
An Exercise in Constructing a Processing Surface

by

Alain J. Martin
Computer Science Department
California Institute of Technology

5047:TR:82

Proceedings of the Second Caltech Conference
on VLSI, January 1981

THE TORUS: AN EXERCISE IN CONSTRUCTING A PROCESSING SURFACE

Alain J. Martin
Philips Research Laboratories
5600 MD Eindhoven
The Netherlands

Abstract. A "Processing Surface" is defined as a large, dense, and regular arrangement of processor and storage modules on a two-dimensional surface, e.g. a VLSI chip. A general method is described for distributing parallel recursive computations over such a surface. Scope rules enforcing the "locality" of variables and procedure parameters are introduced in the programming language. These rules and a particular interconnection of the modules on the surface make it possible to transmit parameter and variable values between modules without using extraneous communication actions.

The choice of the Processing Surface topology for binary recursive computations is discussed and a torus-like topology is chosen.

0. INTRODUCTION

Let us call a "Processing Surface" a large, dense, regular arrangement of processor and storage modules on a two-dimensional surface, e.g. a VLSI chip. How can a computation be distributed over such a surface? What are the arrangements of the modules on the surface best suited for a certain class of computations?

We propose to explore this problem in the following direction. In such an environment, an action on a variable differs in complexity (in terms of the number of elementary steps necessary to perform the action) depending on the distance between the processor module performing the action and the storage module containing the variable. We want to reflect this issue at the programming level by introducing scope rules defining the distance between the program component where a variable is declared, and the program components where the variable can be used.

Since we expect intense communications between the program components, we expect assignments of the form x:=y where x and y belong to two adjacent components (this assignment can take the form of a procedure call or a pair of matching communication actions) to occur as frequently as assignments between variables of the same component. In most distributed systems, the first type of assignment is an order of magnitude more complex than the second one. We consider this hidden discrepancy between equivalent actions unacceptable. We will show that it is possible to define some locality rule for the program variables, and to organize the processor and storage modules on the surface such that no discrepancy of this sort appears. In such a case, the Processing Surface is said to be "continuous".

Furthermore, since for instance inverting a 2*2 matrix does not require as much parallelism as inverting a 1000*1000 matrix, the potential parallelism of an algorithm should not be fixed beforehand (e.g. by the number of available processors) but should be determined dynamically according to the needs of the computation. The component actions of a computation should be created and destroyed as the computation proceeds, and should be automatically distributed over the available modules.

1. THE GENERAL METHOD

The general method we use has been described in [1]. We shall recall it briefly.

The component actions of a computation - the "nodes" - are regarded as the vertices of a graph - the "computation graph" - which grows and shrinks during the computation. An edge - a "channel" - between two nodes means that one of the two, say node A , has created the other, say node B , by a procedure call, and that A and B communicate directly with each other. A is the "father" of B , and B is a "son" of A . Thanks to a parallel procedure call, a father may create several sons simultaneously. The father/son relation defines a partial ordering of the nodes, and all nodes that are not relatively ordered can be performed in parallel.

A computation graph grows and shrinks through a given finite "implementation graph", whose vertices - the "cells" - represent the available modules, and the edges - the "links" - the communication possibilities between modules. Each node is mapped on a cell, and each channel on a link.

Hence, each cell may have to accommodate an unbounded number of nodes. Since a cell represents a very small number of sequential automata (in most cases, one!), the activities of all nodes simultaneously present in a cell have to be sequentialized in some way. But such a sequentialization may introduce deadlock. The main result of [1] is to prove that the nodes of a cell can be interleaved without introducing deadlock provided that the grain of interleaving be correctly chosen. The solution is very simple in that it does not require any particular knowledge about the nodes or the implementation graph nor complicated scheduling.

In this paper we shall consider a special class of computations, namely recursive computations. For this class of computations we shall describe how to implement a continuous Processing Surface, and we shall propose a torus-like topology for the implementation graph.

2. RECURSION

Much has been said about the use of recursion for parallel programming. The reader is referred to the abundant literature on this subject.
For the sake of simplicity, we shall restrict ourselves to one of the most usual recursive methods, namely "divide-and-conquer" (also called "recursive doubling"). Divide-and-conquer algorithms are particularly interesting in that they produce binary trees as computation graphs. Binary trees are regular structures and each node has an outdegree of two, which is interesting in view of their mapping onto a two-dimensional surface.

Parallelism is introduced only by calling two procedures "in parallel". The possibility of further increasing parallelism by pipelining the parameters will not be mentioned although it can easily be added. Nevertheless this class of algorithms is large enough (in particular numerical algorithms) for the exercise to be realistic.

3. THE LOCALITY OF VARIABLES AND PARAMETERS

Since a node is created by a procedure call, a node is a procedure instance with its own program counter, and its set of variables and parameters. The following rules define the "locality" of variables and parameters.

> . The unit of locality is the node: a variable declared inside a node is local to that node.
> . A variable local to a node A is a neighbour for all son nodes of A .

Since the father/son relation between nodes is not transitive, the locality or neighbourhood of a variable with respect to a node is not transitive either: if a node P1 calls a node P2 which calls a node P3 , a variable local to P1 is neighbour for P2 , but not for P3 .

Three types of parameters are used:
. An input parameter is used to "import" a parameter value into a son node, by an assignment of the actual parameter value to the formal parameter variable.
. An output parameter is used to "export" a value from a son node to its father by an assignment of the formal parameter value to the actual parameter variable.
. A reference parameter is used both to import and to export, but by a process of substitution, or "aliasing": the formal parameter replaces the actual parameter in the son node (it is another name for the same variable).

> In the case of the input and the output parameters, the formal parameter is local to the son.
> In the case of the reference parameter, the formal parameter has the same locality as the actual parameter. The formal parameter is thus not local to the son.

Assume that the value x of a variable is to be imported from a father node P1 into a son node P2 . Either an input or a reference mechanism can be used. Assume now that x is to be passed again from P2 to a son node P3 . If x was passed from P1 to P2 as an input parameter, x will be local to P3 if it is passed as an input parameter from P2 to P3 , and neighbour to P3 if it is passed as a reference parameter from P2 to P3 . But if x was passed from P1 to P2 as a reference parameter, x will neither be local nor neighbour to P3 , whether it be passed as an input or as a reference parameter from P2 to P3 .
(In the case where a value is to be exported from a son node to a father node, exactly the same differences hold according to whether it is passed as an output or a reference parameter.)

Hence, the locality or neighbourhood of a reference parameter with respect to a node is not transitive whereas that of an input or output parameter is. But when a value x is passed as an input or an output parameter from node P to node Q, by definition x is copied from the storage area of P into the storage area of Q. No copying is necessary when x is passed as a reference parameter.

The repetitive transport of values via global variables and reference parameters could be used in its full generality, but we propose to restrict its use by the following "locality rule".

Locality rule: An action of a node involves only variables and parameters that are local and/or neighbour for the node.

(Whether global variables should be used at all is doubtful. They have been included for the sake of completeness.) We shall see that this locality rule permits the implementation of a continuous Processing Surface.

4. IMPLEMENTATION OF A CONTINUOUS PROCESSING SURFACE

Definition: A Processing Surface is said to be "continuous" when any action performed on the surface involves only variables that are directly accessible to the processor performing the action, i.e. accessible by elementary read or write operations.

Hence, if we succeed in implementing a continuous surface, we shall have suppressed any form of extraneous communication action for accessing variables.

According to the general method, we know that if node N1 is mapped on cell C1, a son node N2 of N1 is mapped on a neighbour cell C2 of C1. For node N1 to be mapped on C1 means that the local variables and parameters of N1 must be allocated in the storage module associated with C1, and the same for N2 relative to C2. Let M1 and M2 be the storage modules associated with C1 and C2, respectively. According to the locality rule, any action of N2 may involve variables located in M1 and M2. The set $\{M1, M2\}$ is called the "locality area" of N2. In the case where the computation graph is a tree, the locality area of a node consists of at most two elements.

As a direct consequence of the locality rule and of the definition of a continuous Processing Surface, the Processing Surface is continuous if the property C(N) holds for any node N.

C(N) : any action of N is performed by a processor directly connected to the two storage modules of the locality area of N.

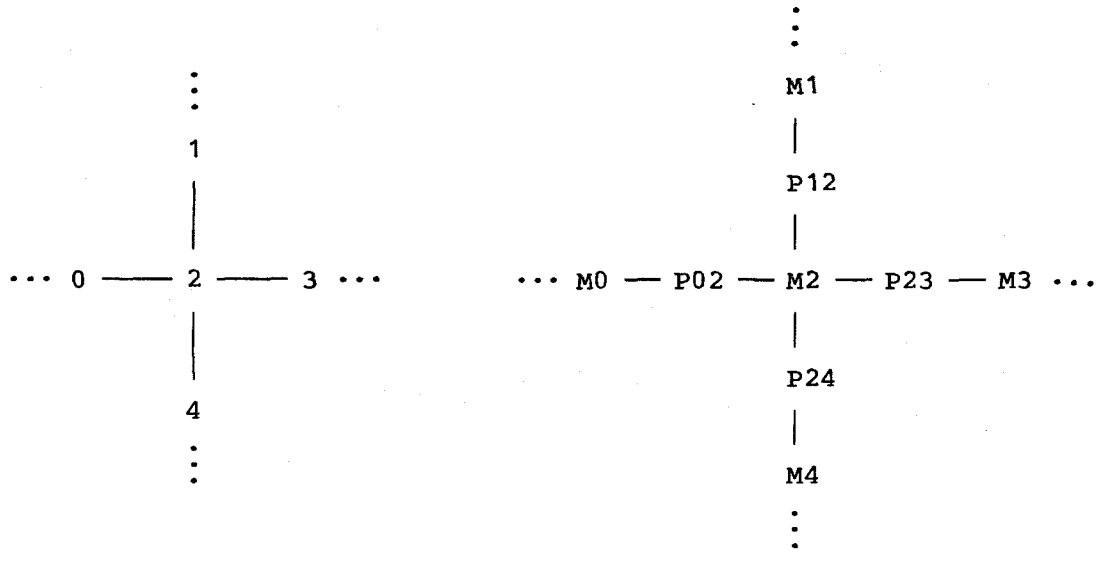
We shall describe a strategy for placing the processor and storage modules on the implementation graph, and for distributing the actions and the variables of the nodes over the processor and storage modules, such that C(N) holds for any node N.

This strategy thus implements a continuous Processing Surface.

1) The placing strategy is directly suggested by the property C(N) .

A storage module is placed at each vertex, and a processor module at each edge of the implementation graph.

(See an example on fig. 1.) Hence, each processor has direct access to two storage modules, and each storage module is shared by as many processors as the degree of the vertex where it is placed.
2) Assume that C(F) holds for a node F . For instance, F has been created in cell 2 of fig. 1(a); its local variables are in M2 , its neighbour variables in M1 , and its actions are processed by P12 (see fig. 2).



(a) implementation graph (b) processor and storage placement

Fig. 1.

Assume that at some stage in the computation of F two son nodes R and D (for right and down) of F are to be created in cells 3 and 4, respectively. The locality areas of R and D must then be (M2, M3) and (M2, M4) , respectively (see fig. 2).
This means that C(R) and C(D) will hold if and only if R and D are processed by P23 and P24 , respectively. Upon reaching the procedure calls of R and D in the procedure body of F , P12 must transmit the creation of R and D to P23 and P24 .
Since, by construction P12 , P23 , and P24 share a common store, namely M2 , the transmission of procedure calls is a simple and local action: P12 adds the names of R and D to the lists - located in M2 - of nodes to be processed by P23 and P24 , respectively.

A processor switches from one node to the other upon a procedure call in the same way as in a multiprogramming system a processor switches from one process to another upon a P-operation on a zero semaphore. We shall not describe the implementation in more detail.

Hence, if C(F) holds for a node F , C(R) and C(D) hold for the two son nodes of F . Observe that the above strategy is independent of the topologies of the implementation graph and of the computation tree.
The root node P of the computation tree is created by the "environment" of the computation. At least one cell of the implementation graph - a root cell - is connected to the environment. It is easy to map P onto a root cell in such a way that C(P) holds.

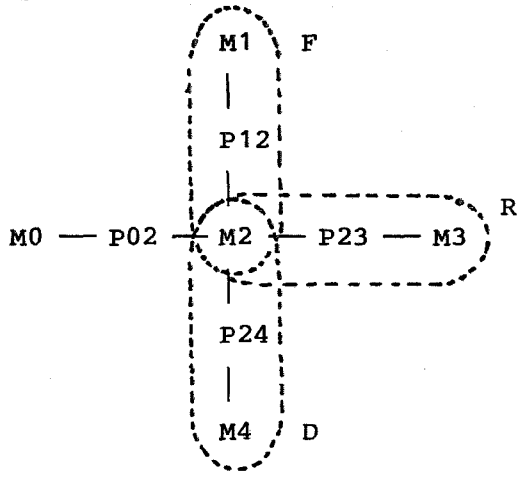


Fig. 2. locality areas

5. THE CHOICE OF THE IMPLEMENTATION GRAPH

We look for a finite implementation graph such that 1) an arbitrary binary tree can be mapped onto it without knowing the sizes of the tree and of the graph, 2) the nodes of the tree are optimally spread over the cells of the graph.

Because of 1), we aim at "simulating" an infinite graph on a finite one. Let us assume that we could indeed construct an infinite implementation graph, which graph would we choose? Since we are looking for graphs that can be represented in the plane by regular and dense structures, we are bound to choose between the three regular tessellations of the plane, which are the square, the triangular, and the hexagonal tessellations. (Although the infinite binary tree is regular, it is not dense, because it grows exponentially and therefore cannot be represented with minimal constant edge lengths.)

We have chosen the square tessellation, although the hexagonal is also interesting. We shall first discuss the problems of mapping a binary tree onto an infinite grid. We shall then simulate the infinite grid on a finite grid.

6. THE INFINITE GRID AS AN IMPLEMENTATION GRAPH

An infinite grid is a graph such that: for $i \geq 0$ and $j \geq 0$, vertex (i, j) is connected with vertex $(i+1, j)$ and vertex $(i, j+1)$.

The mapping of a binary tree on the grid is obvious. The root of the tree is mapped onto vertex $(0, 0)$. If a node is mapped on vertex (i, j) , then its right son R is mapped on vertex $(i, j+1)$, and its down son D