LOCALITY-BASED PROGRAMMING ON A RECONFIGURABLE TRANSPUTER NETWORK

Internal Report C94-08

April 1994

R. Perego

TRANSPUTER COMMUNICATIONS, VOL. 0(0), 1–16 (SUBMITTED PAPER, 1993)

Locality–Based Programming on a Reconfigurable Transputer Network

R. Perego

Istituto CNUCE – Consiglio Nazionale delle Ricerche, Via S. Maria, 36 – 56126 Pisa – Italy, e-mail: perego@vm.cnuce.cnr.it, tel.: +39-50-593253, fax: +39-50-904052

SUMMARY

The importance of communication locality and the suitability of reconfigurable transputer networks for testing locality–based programming styles is investigated. To make the management of a reconfigurable transputer network easier, an extension to the Occam II parallel language is presented. The extension, named OCTOPUS, provides the programmer with a set of constructions and facilities for structuring and mapping parallel programs, and for flexible run–time management of the transputer interconnections. An OCTOPUS-to-Occam source translator automatically generates the code dealing with network configuration and placement of the Occam channels onto transputer links. The capability of altering at run-time the transputer interconnections allows the programmer to devise distinct network topologies that result more suitable to mapping the various phases of a computation in order to exploit solely local–communications. The approach sensibly decreases the programming costs and turned out to be effective for exploiting locality–based parallelism.

KEYWORDS: Occam, Transputer, programming tools, locality, run-time network configuration.

1 Introduction

Transputer networks and the *Occam* programming language [1,2] have gained widespread use in recent years as a low cost parallel solution to many problems. The clear semantics of Occam, directly derived by the CSP model [3], together with its efficient implementation on the transputer architecture, have attracted many researchers interested to message-passing computational paradigms.

Unfortunately, parallel program design with Occam requires a hard refinement

????–????/93/000001–16$13.00 *Received*
©1993 by John Wiley & Sons, Ltd. *Revised*

work, greatly influenced by the experience and skill of the programmer. First, the most promising decomposition strategy aimed at structuring the program as a bare set of cooperating processes that can be efficiently executed in parallel has to be chosen. Later, all the processes have to be correctly coded and mapped on the transputers of the network in a way that minimizes the total execution time of the program while maximizing the system utilization. The mapping cannot generally preserve the neighbourhood relations of all the processes and therefore both channel multiplexing/de-multiplexing functions and a deadlock-free routing algorithm have to be provided by the programmer in order to support the process communications by routing the messages through intermediate nodes. Parallel programming languages like Occam assume in fact a logical connection between all the communicating processes and do not force the programmer to structure the program according to the architectural characteristics of the target multicomputer. On the contrary, for well–known technological reasons, transputer systems as all other multicomputers, are built over networks in which each processing element (*pe*) is directly connected to only a fixed number of other neighbouring *pe*'s. Even if the interconnection network is reconfigurable, its connectivity is however limited by the number of *pe* links and its topology is usually modifiable only before the loading of the program takes place.

In this article, OCTOPUS (OCcam TOols for Programming Utility Software), an extension to Occam II concurrent language together with its corresponding tools is presented. OCTOPUS provides a set of constructions and facilities devoted to simplifying the development of efficient transputer applications. With OCTOPUS the programmer can easily configure at run–time the transputer network and project, when possible, the logical structure of the program onto the physical network. The prolix Occam code for configuring the interconnect and for declaring and mapping the processes on the transputers and the topology–dependent communications channels on the links is automatically generated by a OCTOPUS-to-Occam source translator. Moreover many topologies can be exploited in distinct execution stages of the same parallel application, as different communication patterns may be required during each of these stages.

The paper is organized as follows. Section 2 discusses in some depth the im-

portance of exploiting communications locality to efficiently program transputer networks and surveys some of the relevant literature. Section 3 describes the OCTOPUS language and the functionalities of the associated tools. Results obtained on a parallel matrix multiplication algorithm that exploits the dynamic reconfiguration capabilities of OCTOPUS are also reported. Finally, concluding remarks are presented in section 4.

2 Locality Exploitation

To design an efficient multicomputer application many aspects have to be considered. Some depend strictly on the architectural characteristics of the system (e.g. topology and communication bandwidth of the interconnection network, routing services, power of the *pe*); many others depend on the target application and, in particular, on the parallelism model chosen to exploit concurrency. As a matter of fact, the ability to structure the program giving priority to low cost local communications (between directly connected or neighbouring *pe*'s) over more expensive non–local ones (routed through a number of intermediate *pe*'s) plays a fundamental role in the achievement of satisfactory performances. This is particularly true for transputer systems with *Store and Forward* routing, but it holds also for last generation multicomputers designed upon low–latency communication networks with *wormhole* routing [4]. In fact, local communications, not only have lower latencies due to the fewer nodes crossed, but also consume a smaller fraction of the network bandwidth and reduce the probability of temporal conflicts for the use of the multiplexed communication links [5].

The enhancements from using message locality were recently considered in many proposals of new parallel languages. In ref. [6] the basis of an interesting architecture–independent programming language based on the Bird–Meertens formalism is presented. The proposal aims to restrict the parallel computational model by reducing the distance over which communications can take place. In ref. [7] a new parallel language, named P^3L, is presented. The language provides constructors that allow easy exploitation of locality-based massive parallelism according to common optimized algorithmic skeletons. Moreover, also the compilers for *data parallel* languages such as Fortran-D [8] and High Performance Fortran [9], exploit locality by splitting and mapping the arrays in a way that minimizes the need for

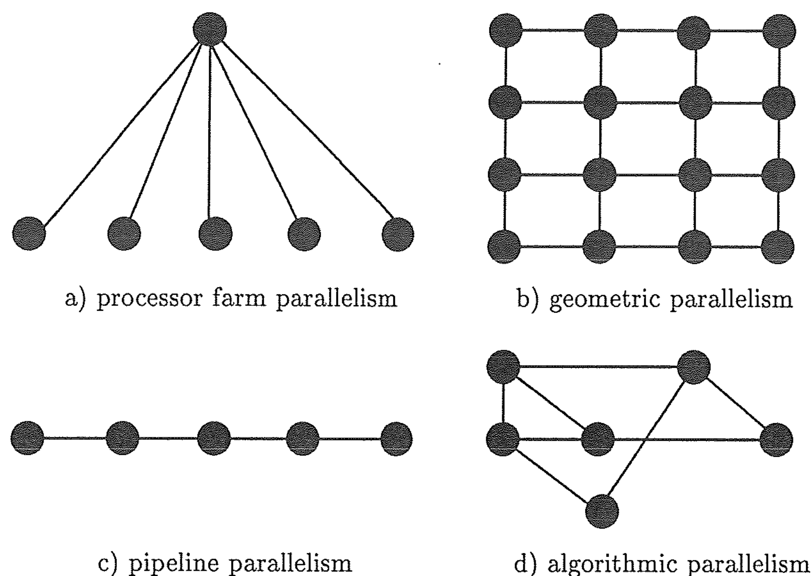

Figure 1. Sketches of some general parallelism models.

non-local data exchanges.

Reconfigurable multicomputers [10] are clearly good candidates for experimenting with a locality-based programming style. The interconnection topology of these systems can be optimized for mapping each specific application according to its communications requirements.

Other works deal with the alteration of the topology of transputer networks during run-time under the control of executing user code. In particular the work by Jones and Murta [11] is relevant for the complete analysis of the reconfiguration costs. They propose a communication scheme in which every transputer issuing a message on the network requires firstly to a centralized resource manager the direct connection with the destination *pe*. Once the request is granted, messages are injected on the physical connection that has after to be relinquished by the owner process. The approach, implemented on a *ParSiFal T-RACK*, realizes a virtual fully connected network, but its cost is very high.

In the following subsections the suitability and flexibility of reconfigurable multicomputers in exploiting locality in classes of applications involving regular and unregular communication patterns is analyzed.

2.1 Message locality on regular problems

A large class of scientific problems have an intrinsically regular underlying structure. For these applications a natural parallelization strategy based on the geometric decomposition of the data domain is generally followed. In the literature this parallelism model (see fig. 1.b) is referred to as geometric parallelism [12], or domain partition [13], model. The model provides for data decomposed and evenly distributed over all the *pe*'s that run the same code. Each *pe* solves the problem on its local data by cooperating with the nodes holding adjacent partitions.
The paradigm is not always easy to exploit and care has to be taken to determine the degree of parallelism which grants an acceptable compromise between computation and communication times on each node. Moreover effective scalability of this kind of application is unlikely to be reached by simply increasing the number of nodes to deal with a given problem with fixed dimensions. In other words, only when the dimensions of a problem increase, can a proportional increase in the exploited parallelism be reached by leaving the dimensions of the data subset assigned to each *pe* unchanged.
In designing a parallel program to solve an intrinsically regular problem, it is generally possible to find a perfect embedding which preserves the adjacency relations deriving from the data decomposition scheme. When this is achieved messages are exchanged only between physically connected *pe*'s. Some difficulties can arise when the natural geometry of the problem does not map the topology of the network (e.g. a 3-D data domain on a 2-D mesh or a tree architecture). However, especially on highly connected and reconfigurable multicomputers, a nearest–neighbour algorithm can be designed in most cases.
Many other problems lead to natural parallel solutions involving well–defined regular communication patterns. For some of these it is simple to derive a strictly local implementation (e.g. the pipeline model sketched in fig. 1.c); for others, communications locality is more difficult to exploit. Consider, for instance, the divide and conquer model [13] or the processor farm model [12] (see fig. 1.a).

2.2 Message locality on unregular problems

Analyzing the control flow of any sequential program one can devise different features that may be executed concurrently or in a pipeline fashion. A similar paral-

lelization strategy based on the decomposition of program functions requires the programmer to design a number of specialized cooperating processes. This leads to a process graph whose mutable structure strictly depends on the particular algorithm implemented. Hey's algorithmic parallelism model [12], based on the splitting and converging of multiple pipes of processes (see fig. 1.d), reflects a general instance of this control parallel approach that seldom gives origin to geometrically regular communication patterns. Although difficult to design and generally not scalable, the adoption of an algorithmic parallelization strategy is often compulsory at some level. In fact the limitations on *pe*'s local memory make often necessary partitioning and distributing both program functions and data. The approach generally prevents the exploiting of message locality on static networks. On reconfigurable multicomputers, on the contrary, it is often possible to configure the physical network with the same connections existing on the algorithmic network.

3 OCTOPUS

OCTOPUS has been designed as a research testbed to simplify the development of transputer parallel applications and to investigate the suitability of run–time reconfiguration as a strategy for enhancing program efficiency through the exploitation of message locality on each communications pattern involved in the computation. It provides the programmer with a small set of powerful and syntactically well–integrated extensions to the Occam II language that increase program modularity and programming ease with respect to the standard Occam II programming environment.

Using OCTOPUS the programmer no longer needs to know the physical addresses of the connections between all the transputers in the network nor how to deal with the explicit declaration and placement of the external communication channels onto the links (all Occam programmers know what this means in terms of annoyance and error proneness). The Occam code for these operations is, in fact, automatically generated by an OCTOPUS-to-Occam source translator (about 2000 lines of Pascal), as a function of the topology selected for the network and of the particular architecture of the target transputer network. The user needs only to link the codes produced by the translator with the OCTOPUS libraries when compiling. The target system of the OCTOPUS prototype is composed of an *INMOS ITEM* with 21

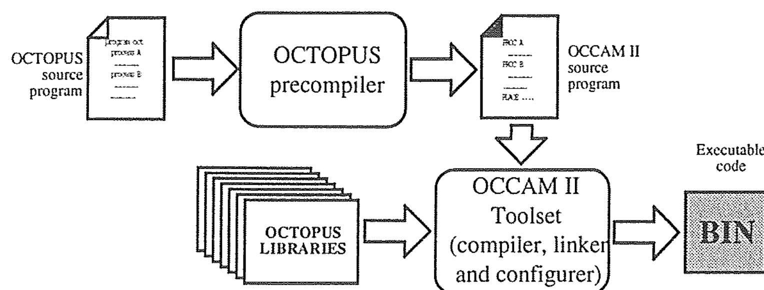

Figure 2. OCTOPUS system organization.

T800 transputer and four *C004* crossbar switches, hosted by a PC.

OCTOPUS programs may be divided into two categories: *(a)* static programs that do not involve multiple topologies, and *(b)* dynamic programs that change at run-time the network interconnect. Although the high level differences between static and dynamic programs are light, for clarity purposes we separate their discussion.

3.1 OCTOPUS static programs

Figure 3 sketches an OCTOPUS static program with a complete three-level ternary tree structure. The algorithmic description of the program, as well as the network configuration and the mapping information of all the parallelism units making up the parallel program, are included within a single construction. The name of the program (e.g. *Ternary.Tree.Computation*) and the physical topology selected for

```
PARALLEL Ternary.Tree.Computation ON TTREE[3]
    PROCESS Root.Module POSITION ROOT
        ... Occam code for the root processor
    PROCESS Node.Module POSITION NODE
        .... Occam code for node processors
    PROCESS Leaf.Module POSITION LEAF
        .... Occam code for leaf processors
```

Figure 3. An OCTOPUS static program with a complete three-level ternary tree structure.

running it (e.g. *TTREE[3]*, a three-levels ternary tree) are indicated in the program prologue after the keyword PARALLEL.
After the prologue, from one to N, with N equal to the number of *pe*'s of the selected topology, *parallelism units* (i.e. the Occam modules executed by each transputer) have to be defined. The code describing each parallelism unit is preceded by a heading composed of the keyword PROCESS, the user provided name of the unit (e.g. *Root.Module, Node.Module, Leaf.Module*) and the keyword POSITION followed by a *placer*. The *placer* defines the mapping of the parallelism unit on the topology. It can be a tuple with the coordinates of the *pe* on which the unit will run (e.g. the keyword *ROOT* that stands for *[1][1]*, indicates the first transputer at the first level of the ternary tree, i.e. the root), or a *replicator* (e.g. *NODE, LEAF*), whose syntax and semantics depend on the selected topology, used to place the same unit onto more *pe*'s.
Table 1 informally shows, for a subset of the topologies managed by OCTOPUS, the syntax of both single and multiple placers. The set of OCTOPUS predefined topologies includes the most common regular topologies that can be built with *pe*'s having four links such as rings, grids, binary and ternary trees, three and four dimensional hypercubes. Other irregular topologies can be added by the user to the OCTOPUS environment providing the physical description of their interconnection scheme in the standard INMOS MMS2 configuration language.

OCTOPUS programming style recalls the Single Program Multiple Data (SPMD) model where all the *pe*'s execute the same code, but their behaviour can be different on the basis, for example, of their position inside the network. This analogy is strengthened by the availability of the OCTOPUS library function *WhoAmI* that returns the position of the caller *pe* in the configured topology.
As an example, the simple OCTOPUS program:

```
PARALLEL grid.program ON MESH[5,5]
    PROCESS Node POSITION [FROM 1 FOR 5, FROM 1 FOR 5]
        ... Occam code
```

configures a two–dimensional square mesh on the network and specifies that the same Occam code has to be loaded and executed on all its 25 *pe*'s. On the basis of the position of the transputer on the mesh, returned by a call to *WhoAmI*, each

simplicity of performing the same operation with OCTOPUS, Figure 4 reports an Occam program equivalent to that of Figure 3. It is worth mentioning that the code reported in Figure 4 (a slightly modified version of that proposed in ref. [14]) assumes a homogeneous ternary tree topology on the transputer network with link 0 of each transputer connected to the parent *pe*, and links 1, 2 and 3 connected to the left, middle and right children *pe*'s respectively. In many cases this homogeneity in the hardware does not exist due to some architectural limitations in the transputer connectivity. Therefore the actual code would be much more complex and, especially, not portable across different transputer networks. Eventual architectural non-homogeneity and differences are instead hidden by OCTOPUS and do not affect program writing. Moreover, the code the programmer has to write to adapt his parallel algorithm to the transputer network is strongly reduced, resulting in significant time savings, time that can be instead more profitably spent in fine-tuning and optimizing the parallel algorithm.

3.2 OCTOPUS dynamic programs

It has been observed that many parallel applications use various communication patterns in different phases of execution [15]. Moreover, each distinct communication pattern often turns out to be highly regular. Consider, for example, geometric applications in which almost three different communications patterns can be devised if the data are not already distributed on the *pe*'s. In a first phase, the data must in fact be loaded and distributed over the nodes; in a second phase, the *pe*'s cooperate to compute the results; lastly, the distributed results have to be assembled and returned. Often these three communication patterns cannot be efficiently embedded into the same topology.

Only small advantages over a static configuration capability can be therefore gained because the network topology can be optimized for a single communication pattern only. A greater flexibility is instead theoretically offered by dynamically reconfigurable systems. The possibility of reconfiguring the network during execution allows the programmer to structure the computation so that the more suitable topology is chosen for each communication pattern involved. As a consequence, it is easier to map the processes, and locality exploitation is enhanced while reducing, and frequently totally eliminating, the need for implementing routing algorithms.

```
... SC root(CHAN OF ANY from.l.son,to.l.son, from.m.son,to.m.son,from.r.son,to.r.son)
... SC node(CHAN OF ANY from.father,to.father,from.l.son,to.l.son,from.m.son,to.m.son,
    from.r.son,to.r.son)
... SC leaf(CHAN OF ANY from.father, to.father)
[13]CHAN OF ANY a,b:
PLACED PAR
  PROCESSOR 0 T8
    PLACE a[1] AT 5:
    PLACE b[1] AT 1:
    PLACE a[2] AT 6:
    PLACE b[2] AT 2:
    PLACE a[3] AT 7:
    PLACE b[3] AT 3:
    root(a[1],b[1],a[2],b[2],a[3],b[3])
  PLACED PAR k2 = 0 FOR 3
    VAL p3 IS (3*k2)+1:
    PLACED PAR
      VAL p IS k2+1:
      VAL n IS p3:
      PROCESSOR p T8
        PLACE b[p] AT 4:
        PLACE a[p] AT 0:
        PLACE a[n] AT 5:
        PLACE b[n] AT 1:
        PLACE a[n+1] AT 6:
        PLACE b[n+1] AT 2:
        PLACE a[n+2] AT 7:
        PLACE b[n+2] AT 3:
        node(b[p],a[p],a[n],b[n],a[n+1],b[n+1],a[n+2],b[n+2])
      PLACED PAR k3 = 0 FOR 3
        VAL p IS (p3+k3):
        PROCESSOR p T8
          PLACE b[p] AT 4:
          PLACE a[p] AT 0:
          leaf(b[p],a[p])
```

Figure 4. Occam configuration code for a ternary tree network.

the prologue, OCTOPUS dynamic programs are characterized by the presence of more *SET* commands within the parallelism units. Each *SET* command is translated into a call to the OCTOPUS run-time support dealing with processor synchronization and links configuration. In writing dynamic programs few rules must be followed:

1. all parallelism units must select the same topologies in the same order to ensure mapping consistency;

2. topologies with different number of *pe*'s can be used in the same program, but the number of transputers used in each distinct configuration must be the same;

3. at every reconfiguration the external communication channels are redefined. The scope of the OCTOPUS names of the channels placed onto transputer links is restricted at the phase in which each topology is physically configured.

Reconfiguring the network at run–time requires the synchronization of all the *pe*'s in order to prevent processors from using the communication links while switching is in progress. Moreover the reconfiguration cost on the testbed architecture is proportional to the number of links that have to be switched, making similar dynamic reconfiguration approaches well suited only for small networks.

The time T_{conf}, needed to perform an alteration of the interconnections during run-time on the OCTOPUS prototype is expressed by the following equation:
$T_{conf} = T_{sync} + T_{startup} + (k * T_{switch})$
where T_{sync} is the time needed to synchronize all the *pe*'s, $T_{startup}$ is a constant time (about 25 microseconds), k is the number of links that have to be switched, and T_{switch} (about 20 microseconds), is the time required to switch one link.

On the other end, the reconfiguration cost is generally not weighed down by load unbalance because the computational phases one can devise in a parallel application are generally synchronous: in most cases in fact, program semantics forces each *pe* to wait for the others at the end of a phase before it can successfully begin the next one.
For its high cost, reconfiguration has therefore to be used carefully, only when the advantages, in terms both of locality exploitation and of programming ease, are

Table 2. Matrix multiplication times (in seconds) on sixteen *pe*'s.

matrix dimensions	static version	dynamic version
20 X 20	0.019	0.017
100 X 100	0.856	0.770
150 X 150	2.383	2.195
200 X 200	5.275	4.928

large.

To test the OCTOPUS dynamic reconfiguration capabilities, a dynamic version of the matrix multiplication algorithm with square subblock decomposition proposed in ref. [16] was implemented. The program uses an incomplete ternary tree both in the first phase (partitioning the matrices into subblocks and loading them on the nodes) and in the third phase (assembling the resulting matrix). Matrix multiplication is instead performed after having configured a mesh.
In table 2, execution times (in seconds) on sixteen T800 transputers of the OCTOPUS dynamic version of the program are compared with those measured with an Occam II version of the same algorithm in which the loading and unloading of the *pe*'s are also performed on a mesh adopting the efficient double-buffer pipelined algorithm proposed in ref. [14]. The results obtained are comparable although the times of the dynamic version are about 10% lower in the average than those of the standard Occam implementation. The main advantage is instead the easier implementation that together with the sensibly shorter OCTOPUS source code, results in sensible time savings for the programmer.

4 Concluding remarks

The role of message locality on multicomputer architectures and the suitability of reconfigurable multicomputer to exploit locality on the more common parallelism models was analyzed. Use of dynamic reconfiguration to more flexibly exploit locality was proposed, and the main aims of OCTOPUS, a new proposal for programming reconfigurable transputer networks, were outlined. The tool simplifies the programming and the mapping of parallel programs on transputer networks. The availability of a complete set of predefined topologies and communication contexts

that can be installed on the transputer network by a simple OCTOPUS command, makes program design easier. The exploitation of communication locality is possible also when multiple communication pattern are involved because OCTOPUS allows the alteration of the topology of the network during run-time under the control of executing user code. In this way, more networks can be used by the same OCTOPUS program, resulting in a dynamic evolution of the topology of the interconnect during execution.

An OCTOPUS-to-Occam translator and a library that supports the proposed extensions to Occam II have been implemented. The prototype, tested in every day use, works satisfactorily and provides meaningful error reporting. It checks the consistency of both single and multiple network settings and process placements.

Some results showing the gains achieved through the dynamic reconfiguration on a parallel matrix multiplication algorithm were given.

Acknowledgements

The author would like to thank Salvatore Orlando for valuable suggestions and encouragements and Paolo Pedelini for his important contribute in the design and development of the OCTOPUS prototype.

The work described in this paper has been partially funded by the Progetto Finalizzato "Sistemi Informatici e Calcolo Parallelo" of the Italian National Research Council.

REFERENCES

[1] D. May, R. Shepherd, and C. Keane, 'Communicating Process Architecture: Transputer and Occam', in *Future Parallel Computers: an Advanced Course*, pp. 35–81, Pisa, Italy, (June 1986). LNCS 272 Spinger-Verlag.

[2] INMOS, Prentice Hall, *Occam 2 Reference Manual*, 1986.

[3] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[4] W. Dally, 'Network and Processors Architecture for Message-Driven Computers', in *VLSI and Parallel Computation*, eds., R. Suaya and G. Bithwistle, chapter 3, 140–222, Morgan Kaufmann Publisher, Inc. - San Mateo, California, (1991).

[5] A. Agarwal, 'Limits on interconnection network performance', *IEEE Transactions on Parallel and Distributed Systems*, **2**(4), 398–412, (October 1991).

[6] D.B. Skillicorn, 'Architecture-Independent Parallel Computation', *IEEE Computer*,

38–50, (December 1990).

[7] F. Baiardi et al., 'Architectural models and design methodologies for general-purpose highly parallel computers', in *Proc. of IEEE 5th Int. Conf. COMP EURO 91*, pp. 18–25, (1991).

[8] C.-W Tseng, 'An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines', Technical Report TR-93-199, Rice Technical Report, (1993).

[9] 'High performance Fortran language specification'. Available by ftp from `titan.rice.cs.edu`, January 1993.

[10] A.J.G. Hey, 'Reconfigurable Transputer networks: Practical Concurrent Computation', in *Scientific Applications of Multiprocessors*, eds., R.J. Elliot and C.A.R. Hoare, 39–54, Prentice Hall, (1989).

[11] P Jones and A. Murta, 'Practical experience of run-time link reconfiguration in a multi-transputer machine', *Concurrency: Practice and Experience*, 1(2), 171–189, (December 1989).

[12] A.J.G. Hey, 'Experiment in MIMD Parallelism', in *Proc. of Int. Conf. PARLE '89*, pp. 28–42, Eindhoven, The Netherlands, (June 1989). LNCS 366 Spinger-Verlag.

[13] H.T. Kung, 'Computational Models for Parallel Computers', in *Scientific Applications of Multiprocessors*, eds., R.J. Elliot and C.A.R. Hoare, 1–15, Prentice Hall, (1989).

[14] R. S. Cok, *Parallel Programs for the Transputer*, Prentice Hall, 1991.

[15] W. Lo et al., 'Oregami: Software Tools for Mapping Parallel Computations to Parallel Architectures', in *Proc. of IEEE Int. Conf. on Parallel Processing*, pp. 88–92, (1990).

[16] G.C. Fox et al., *Solving Problem on Concurrent Processors*, Prentice Hall, 1988.