XTPML – SIMPLIFYING THE DEVELOPMENT OF PARALLEL PROGRAMS FOR IMPLEMENTATION ON VARIOUS TRANSPUTER ARCHITECTURES

J.K. Hartley, A. Bargiela
Real Time Telemetry Systems - Simulation and Modelling Group
Department of Computing , The Nottingham Trent University
Burton St., Nottingham NG1 4BU, U.K.
Email: jkh@doc.ntu.ac.uk

KEYWORDS

Distributed architectures, Parallel programming, Graphical user interface.

ABSTRACT

Parallel processing has always held a great promise of the high efficiency and scalability of computations. However, it must be noticed that, to date, parallel processing has been used to a lesser degree than initially expected. One reason for such a situation is the inherent complexity, stemming from the hardware dependent nature of programming parallel computers (Pazzini and Navaux 1992). Another, possibly more important, reason for the limited use of parallel hardware is the lack of portability of applications between various parallel computers. The program XTPML has been developed which offers a generic tool for programming transputer platforms. XTPML simplifies the development of parallel programs by offering the facility of a logical specification of inter-task communications that abstracts from the physical detail of processor connectivity and task placement. A graphical user interface has been implemented as a front end – allowing the user (at the press of a button) to distribute the tasks throughout the hardware, to compile the set of programs and to execute the application. Any alteration to the processor connectivity is handled internally by XTPML.

XTPML has been applied to the parallelisation of a simulation program in the context of large-scale water distribution networks (Hartley and Bargiela, 1997).

INTRODUCTION

The advantages of developing parallel algorithms for the simulation of large-scale networks has been proved in many different disciplines, such as power, gas and water (Hartley and Bargiela 1995). The increasing complexity of such networks, together with the requirement for smaller discretisation intervals means that the task of simulating large-scale networks frequently outgrows the computational power of economically viable single processor systems. However, the diversity of parallel hardware designs means that it is exceedingly difficult to develop and test parallel applications on hardware that is different from the target computer. A logical model of parallel computations, which would act as a reference model for both the parallel hardware designers and the software developers, in much the same way as the von Neumann model acts as a reference for sequential processing, would help to overcome such difficulties.

The challenge is that the model needs to be sufficiently general to encompass a full spectrum of parallel processing architectures, from shared memory multiprocessors through to local area networks. In particular, the model needs to encompass the message passing and the shared memory programming paradigms of parallel systems (Argile and Bargiela 1993). A recently proposed candidate to such a role is a Bulk Synchronous Parallel (BSP) processing model (Valiant 1990). The parallel metalanguage TPML (Hartley and Bargiela 1994), developed at The Nottingham Trent University, represents an implementation of the BSP processing model. It has been developed and implemented as an extension to the parallel (3L) Fortran for a transputer platform.

To aid those scientists and engineers who want to concentrate on solving problems from their application domains (rather than being distracted by low level programming considerations), a graphical user interface (XTPML) has been developed to work alongside TPML. XTPML generates an application file from the set of TPML-augmented tasks, in response to the user's request by interaction with the graphical user interface. This development can be easily extended to any MIMD (multiple instruction, multiple data streams) computer.

PARALLEL METALANGUAGE (TPML)

The parallel metalanguage enables a logical specification of parallel tasks and inter-task communications. This operation is performed while hiding the message transport complexities inherent to a specific physical processor connectivity and/or processor-memory arrangements. The execution of the metalanguage statements has the effect of generating application code in 3L Parallel Fortran and generating all the hardware specific information that is necessary to support the execution of parallel tasks. Since the major TPML design objective was to facilitate easy transition from hardware specific to hardware independent parallel software development, the design decisions favour simplicity in preference of optimality.

This TPML implementation assumes a coarse grained model of parallel processing with individual tasks typically allocated to separate processing nodes - although processors may possess a number of tasks, or indeed be redundant. Communication between these tasks is performed by sending messages. Within this model of computation each processor is assigned an automatically generated auxiliary message routing task which is an agent facilitating inter-task data communication. This task implements a virtual channel of communication between its associated computational tasks and every other computational task. The routing task is individualised by the inclusion of information about the physical processor connectivity, which is optimised so as to provide the most efficient links between the computing nodes.

If the number of processors is less than the number of computational tasks, more than one task must be placed on the same processor. On such a processor, TPML generates just one router. While, conceptually, any computational task could be placed on any processing node, one needs to ensure that the I/O statements are contained only within the tasks placed on the root processor. This is because the I/O statements must have access to the afserver process via the filter task, which is positioned on the root processor. If the number of processors exceeds the number of tasks, the tasks can be remotely placed on these processors.

Figure 1 illustrates the logical view of a parallel program as specified with the TPML. In this example, the master task APPLICATION_M is in communication with every other application task (APPLICATION_1 through to APPLICATION_6). There is no communication between these slave tasks, thus the logical communication network has a star topology.

TPML derives appropriate switching vectors for each router, which ensures that the optimal route is followed from one task to another, given the placement of these tasks on the processors. The connectivity between the routers mirrors the physical connectivity of the processing nodes, while the logical connectivity of tasks is modelled by the appropriate values of the

switching vectors. Figure 2 and Figure 3 illustrate the configuration of the routers. It is worth noting, that in order to map the star topology of the logical connectivity of tasks, onto the physical connectivity of processors, only a subset of entries in the switching vectors is used, as indicated in Figure 2.

Figure 1. Logical inter-task communications

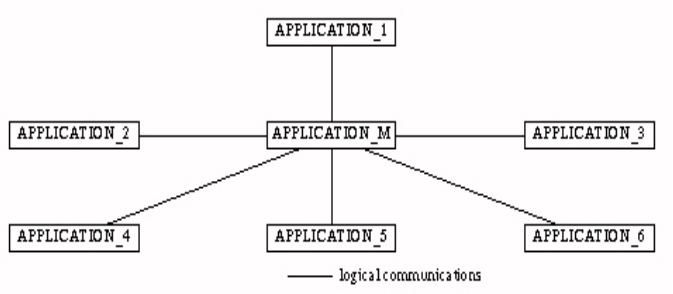


Figure 3. Physical system configuration

TPML SYNTAX

The syntax of the TPML, provided for use within the application code includes three sets of commands for the respective management of: tasks, inter-task communication and superstep synchronisation.

Task management:

Each task is bracketed by a pair:

c>>tpml_task_begin(task_id)
and
c>>tpml_task_end

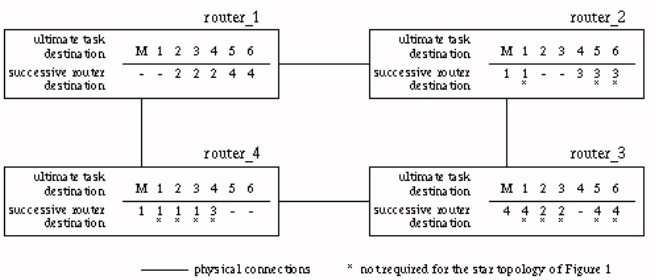
The user includes the code for the task named 'task_id' within these expressions. The code must include all declarations of variables found within the task, along with any subroutines or functions that might be accessed during the execution of the program.

Inter-task communication:

c>>tpml_send(destination_task_id, data, length)
and
c>>tpml_receive(data)

The tpml_send command specifies to which task a data packet is to be sent. In order to keep the implementation of the metalanguage simple, we have allowed that the data packet is of an integer array type only. It has been left to the programmer to pack or unpack the data of other types into this array. By explicitly specifying the number of elements in that array, we minimise the volume of data on the communication links. To receive a data packet, TPML requires the programmer to specify only the name of the array to which the incoming data packet is to be assigned. This is so, because the data packets always arrive from the same source – the router.

Superstep synchronisation:



c>>tpml_step_begin(step_id)
and
c>>tpml_step_end

The superstep facility serves to ensure that the access to the shared data by the parallel tasks is properly synchronised. The 'step_id' parameter, identifies a superstep within each parallel task. Execution of parallel tasks proceeds through all the supersteps in the same order. This implies that the feasible interleaving sequences of the concurrent tasks are well controlled, but the application programmer is required to ensure that all the tasks proceed through the supersteps in the same order. Within the brackets: **c>>tpml_step_begin** and **c>>tpml_step _end,** each task may perform different computations but they must all reach the end of their current supersteps before any one can proceed to the next. The superstep facility provides a barrier synchronisation for the participating tasks.

The superstep has been implemented as a separate task that holds information concerning the amount of supersteps of each identity, positioned within the application code. The superstep task receives messages from the application tasks, indicating that that particular task's individual superstep, with identity 'step_id', has been completed. These messages are queued internally and counted by the superstep task and if they balance out with the number of expected supersteps of type 'step_id', the permissions to enter the next superstep are granted.

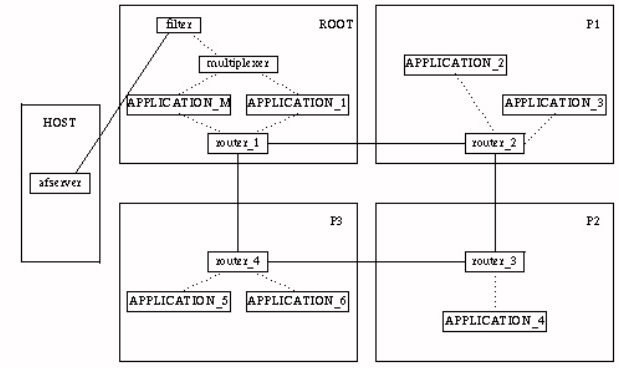
The command

c>>tpml_kill

has been introduced to ensure that the execution of continual loops in the application tasks is discontinued if required.

METALANGUAGE COMPILATION

The generation of the application code is evoked by the command **xtpml**.



The compilation of a parallel application code written using the TPML syntax includes a number of effects:

- extraction of the code,
- extraction of information on processor topology,
- generation of router tasks and their customisation,
- augmentation of the user code by the communication commands,
- generation of a configuration file.

These may all be executed by interaction with the graphical user interface (see Figure 4) – implemented using TCL.

Generation of tasks

The user has the option to create **New** files or **Open** existing files - the user is asked to provide the path name of the **Project Directory** where the files are either to be created or are already listed. According to the **Number of Tasks** (requested by the user or already in existence), a **Tasks List** is generated. By highlighting a particular task, in this box, and then selecting **Edit**, the individual code for the application tasks may be edited. This code will include the TPML syntax. Once all the tasks have been created and the editing is complete, the user may choose where the tasks are to be placed on the transputer architecture.

The selection of **Save** results in the generation of the application tasks, the router tasks and the configuration file. The application tasks are generated by a single pass along the parallel code, specified in TPML. The beginning and end of each task's code are indicated by the inclusion of the TPML commands **c>>tpml_task_begin** and **c>>tpml_task_end**. The code contained between these brackets is then written to a file headed by the declarations required by the TPML commands for communication and superstep implementation. Subroutines which take care of the execution of communications between the tasks are included at the end of the file. A library of subroutines is available, and specific versions are selected according to the location of the task on a particular processor (Hartley and Bargiela 1994).

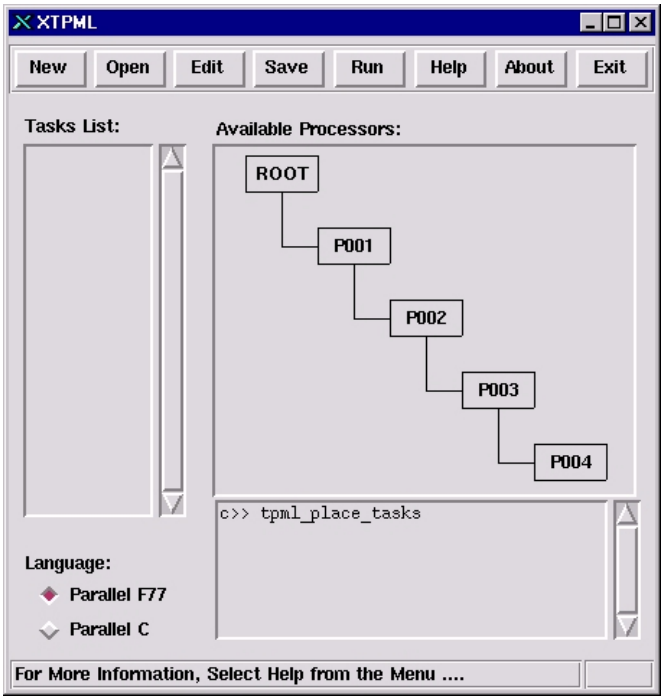


Figure 4. The graphical user interface of XTPML

The router tasks are automatically constructed by adding details of a physical hardware to the logical specification of inter-task communication (as implicitly programmed in TPML). The additional information is retrieved automatically by the XTPML compiler and consists of: the number of available processors, the processor connectivity and the number of communication ports needed. Based on this, XTPML provides automatic generation of routing paths from one task to any other, ensuring that the minimum length path between tasks is followed. These paths are generated by building reachability trees from each task to all other tasks, thus eliminating the possibility of loops within the designated paths. An individual array, indicating the output port to be selected given the destination task of the message, is included within the code for each router (see Figure 2).

The creation of the superstep task is implicit in the reference to the **c>>tpml_step_begin/end** commands. The structure of this task is determined by analysing the TPML source code and counting the number of times a superstep of identity 'step_id' is encountered in the corresponding tasks.

Augmentation of code by communication commands

The XTPML model of parallel processing is that of the communicating sequential processes. All data is exchanged between the application tasks by means of explicit unsynchronised send and receive calls. The sending and receiving tasks are decoupled by means of the communication routers that are responsible for this message delivery. All of these router tasks are optimised to be structurally deadlock free, ensuring the completion of program executions.

This logical model of communication is mapped, within the transputer systems environment onto the physical communications over channels. A channel connects exactly one process to exactly one other process, carrying messages in one direction only. If a bi-directional communication between two processes is required, two channels must be used. Each process can have any number of input and output channels, but the channels in a system are fixed - new channels cannot be created dynamically. An important property of transputer channel communications is that they are synchronised. A process wanting to send a message over a channel is always forced to wait until the receiving process reads the message. So, the XTPML must ascertain that the parallel program is structurally free from deadlock. This problem is resolved by the inclusion of a buffer space in each router so that the sending and receiving tasks are de-synchronised.

The programmer must supply the array to be communicated ('data'), along with its size ('length') and logical destination ('destination_task_id'), within the command

c>>tpml_send(destination_task_id, data, length).

No specification of physical addresses or routing information is required. Similarly, within the code of the destination task, the programmer must indicate to which array (`data') the received message must be written -

c>>tpml_receive(data).

The commands **c>>tpml_send** and **c>>tpml_receive** are substituted by appropriate calls to the corresponding subroutines, included at the end of each application task's code:

```
        subroutine tpml_send(task,data,length)
c --- Send integer array 'data' to 'task' via a router
        CALL F77_CHAN_OUT_WORD(length,outport1)
        CALL F77_CHAN_OUT_MESSAGE(length,data,outport1)
        CALL F77_CHAN_OUT_WORD(task,outport1)
        return
        end

        subroutine tpml_receive(data,length)
c --- Send message to router to indicate that the task is ready
c --- to receive a packet
        CALL F77_CHAN_OUT_WORD(continue,outport1)
c --- Task receives a packet in the form of an integer array 'data'
        CALL F77_CHAN_IN_WORD(length,inport1)
        CALL F77_CHAN_IN_MESSAGE(length,data,inport1)
        CALL F77_CHAN_IN_WORD(task,inport1)
        return
        end
```

The indication of the task being ready to receive a packet in the tpml_receive subroutine goes part way to solving the problem of deadlocks.

The command **c>>tpml_step_end** also indicates a call to the tpml_send subroutine, with 'task' being denoted by the superstep task.

Generation of the configuration file

The configuration file is read by the afserver program to determine which computing nodes of the transputer platform should be used by the specific application tasks.

The configuration file specifies:
- the processors and the wires connecting them
- the names of the files containing the component tasks of the application
- the connections between the various tasks' ports
- the placement of particular tasks on particular processors in the network.

The XTPML is capable of generating any appropriate configuration file

according to the present architecture. Arbitrary networks of transputers can be constructed simply by wiring together the transputers into a selected topology. The current topology of the network of transputers is displayed as part of the graphical user interface (see Figure 4). The graphical user interface allows the user to place the application tasks on the individual processors without the need of any extra coding or hardware considerations. The application task is placed on a processor by highlighting the task and then selecting the appropriate processor in the **Available Processors** display. This selection will be shown as text in the window below. All application tasks communicate with each other via the router tasks - transparently to the user. A router may have any number of channels connecting itself to application tasks on the same processor. However the routers are placed on different processors, so can only be connected where physical wires connect the processors' links. Each connection, between two routers placed on different processors, is assigned exclusive use of one of the physical link channels connecting these processors. The number of interconnections between routers on different processors is therefore limited by the number of hardware links each one has (four). There are two channels assigned to communicate along each of these links, one in each direction.

Communication between application tasks is not limited in this way. If the number of processors is less than the number of computational tasks, more than one task must be placed on the same processor. On such a processor, XTPML generates just one router, which may connect to any number of local application tasks. While, conceptually, any computational task could be placed on any processing node, one needs to ensure that the I/O statements are contained only within the tasks placed on the root processor. This is because the I/O statements must have access to the afserver process via the filter task, which is positioned on the root processor. If the number of processors exceeds the number of tasks, the tasks can be remotely placed on these processors.

An example of the XTPML-generated configuration file, tpml.cfg is given below:

```
processor HOST
processor ROOT
wire ? ROOT[0] HOST[0]          ! physical wire connection between the
processor P001                  root and host
:
wire ? P003[1] P002[2]
!
Task tafserver  Ins=1  Outs=1
Task filter  Ins=2  Outs=2  Data=50k
Task multiplexer  File=filemux  Ins=3  Outs=3  Data=8k
Task tpml_router1  File=router_1  Ins=4  Outs=4        ! tpml_router1 has
Task tpml_router2  File=router_2  Ins=4  Outs=4        task image file
Task tpml_router3  File=router_3  Ins=3  Outs=3        router_1.b4
Task tpml_router4  File=router_4  Ins=4  Outs=4
Task application_m  File=application_m  Ins=3  Outs=3  Data=500k
Task application_1  File=application_1  Ins=3  Outs=3  Data=500k
:                               ! application_1 has 3 input + output ports
Task application_6  File=application_6  Ins=1  Outs=1  Data=500k
!
Place tafserver  Host
Place filter  Root
Place application_m  Root       ! application_m  is placed on the root
Place application_1  p001       processor
:
Place tpml_router4 p003
!
Connect ? tafserver[0]  filter[0]
Connect ? filter[0]  tafserver[0]
Connect ? filter[1]  multiplexer[0]
Connect ? multiplexer[0]  filter[1]
Connect ? multiplexer[1] application_m[1]  ! logical connection from
Connect ? application_m[1] multiplexer[1]  application_m to multiplexer
:
Connect ? tpml_router1[2] tpml_router2[0]          ! tpml_router1 has
:                                                  output port 2
Connect ? application_6[0]  tpml_router4[0]
Connect ? tpml_router4[0]  application_6[0]
```

The information about the number of transputers and the interconnections between them is extracted by the XTPML compiler automatically by means of the `tworm' utility (Beam 1991). According to how the processors are connected, the TPML compiler generates an appropriate configuration file. For example, when the transputer network is connected into a pipeline, each processor - other than the root and the last processor in the line - has associated two pairs of communication ports. However, when the transputer network is connected into a tree structure, the processors may use up to four pairs of communication ports. This requires customisation of the sending and receiving procedures that map the logical task connectivity onto the variants of physical processor connectivity.

COMPILATION OF THE XTPML PROGRAM

Under **xtpml**, when the user selects the button **Save**, a list of system commands are automatically executed. This results in the generation of the configuration file and the application, router and step tasks - all coded in Parallel Fortran. These tasks are compiled and then linked ready for execution across the distributed transputer network.

```
tworm –c> tpml.cfg    # generates processor interconnection information
trun t_parse.app      # instigates compilation of the programmer's code
tff –t application_m.f          # compiles as a task to be later configured
tff –t –s application_1.f       # compiles with the stand-alone library
:                               as a task to be later configured
tff –t –s application_6.f
tff –t –s router_1.f
:
tff – t –s router_4.f
tconfig tpml.cfg tpml.app       # generates tpml.app from the
                                interconnected task images
```

As the name suggests, selection of the **Run** button invokes the run command.

```
trun tpml.app         # invokes execution across the transputer network
```

The graphical user interface allows the user to configure, compile and execute any set of tasks (coded in Parallel Fortran using TPML) at the press of a button.

PARALLEL SIMULATION USING XTPML

XTPML has been used as an aid to parallelising a simulation algorithm for water distribution systems (Hartley and Bargiela, 1997). Simulation involves fitting a set of measurements to the corresponding values calculated from the mathematical model of the system. In general, simulation procedures are approximately quadratically dependent on the size of the physical network. This quasi-quadratic numerical complexity of the simulation algorithm suggests a need for parallel implementation of the simulator, so that the real-time performance may be maintained also for large-scale systems.

Recent advances in computing hardware point towards the development of distributed, parallel programs. In the case of the simulation of water distribution systems, the physical structure of the problem can be utilised when decomposing it into a number of smaller subproblems making it particularly suitable for use with distributed processors. Each of these subproblems are solved individually and the solutions are recombined to achieve the solution of the overall problem. To take into account the interactions between the subproblems, the solution usually proceeds iteratively with information exchange between the subproblems and a coordinating master problem.

XTPML requires the programmer to produce code for the coordinating master problem and the smaller identical subproblems only. XTPML then

generates tasks in accordance to their placement on the distributed transputer network. The subproblems can be duplicated as many times as required for individual large-scale networks.

The simulation algorithm only needs the master problem to communicate with the subproblems, so the logical inter-task communications are in the form of a star topology (see Figure 1). The communication between all of the tasks will be dealt with by XTPML – an increased number of processors and tasks will not require any extra coding by the programmer. If the programmer wishes to alter the configuration of the transputer hardware, XTPML will alter the tasks accordingly along with the configuration file.

The results of the parallel algorithm described in (Hartley and Bargiela, 1997) show that, with appropriately sized subnetworks, the simulation time of the entire problem can be reduced:

Table 1. Convergence times of the algorithm applied to a 130-node network partitioned into two, three and four subnetworks.

Number of nodes	Total subsystems solution time (s)	Coordination time (s)	Total solution time (s)
65, 65	10.7	0.03	11.3
43, 43, 44	5.4	0.08	6.1
32, 32, 33, 33	5.0	0.19	5.7

XTPML has allowed the parallel simulation algorithm to be tested with different numbers of subsystems (and consequently varying numbers of tasks and processors in this example) without the programmer needing to alter any of the original code.

CONCLUDING REMARKS

The paper describes a parallel computers simulation environment that is an implementation of the BSP model of parallel computations, on the transputer platforms. The design of the language, however, is general and it is eminently implementable on other parallel computers. XTPML affords the programmer a logical specification of inter-task communications and itself elaborates on the detail of physical processor connectivity and task placement. Any alteration to the processor connectivity is handled internally by XTPML, so that the specification of communications in the application programs remains unchanged. The graphical user interface enables the user to determine the optimal placement of the application tasks by allowing the XTPML-generated code to be executed on any number of transputer configurations, without the need of reviewing the user-written code. So XTPML allows scientists and engineers to concentrate on determining solutions to complex simulation problems without becoming involved in the complexities of parallel programming.

XTPML has been successfully used to generate parallel code for the simulation of large-scale water networks. The ensuing parallel algorithm was tested on a 130-node network partitioned into a varying number of subnetworks and consequently distributed on a varying number of processors. This implementation was transparent to the message transport complexities inherent to parallel programming and illustrates the improved portability of applications between different transputer architectures.

REFERENCES

Argile, A., Bargiela, A., 1993. "Using X11 Windows to Provide Shared Task-Memory in Distributed Systems", Integrated Computer Applications in Water Supply, Vol. 1 (Sept.), edited by B. Coulbeck, Research Studies Press, John Wiley, Leicester, 305-16.

Beam Ltd., 1991. 'TRANSX RELEASE NOTES'.

Hartley, J.K., Bargiela, A., 1994. "TPML: Parallel Meta Language for Scientific and Engineering Computations using Transputers", *Proc. SMS TPE 94* (Sept.), 22-31.

Hartley, J.K., Bargiela, A., 1995. "Parallel Simulation of Large-Scale Water Distribution Systems", *Proc. of ESM '95* (June), 723-727.

Hartley, J.K., Bargiela, A., 1997. "Parallel State Estimation with Confidence Limit Analysis", *Parallel Algorithms and Applications*, Vol. 11, No. 1-2, 155-167.

Pazzini, M., Navaux, P., 1992. "TRIX, A Multiprocessor Transputer-Based Operating System", Parallel Computing and Transputer Applications, edited by M.Valero et al, IOS Press/CIMNE, Barcelona.

Valiant, L.G., 1990. "A Bridging Model for Parallel Computation", *Communications of the ACM*, Vol. 33, No. 8 (Aug.).