TPML: Parallel Meta Language for Scientific and Engineering Computations using Transputers

J.K. Hartley, A. Bargiela

Department of Computing
The Nottingham Trent University
Burton St, Nottingham NG1 4BU, U.K.
jkh@uk.ac.ntu.doc, andre@uk.ac.ntu.doc

Summary

The experience with the transputer implementation of parallel processing algorithms, in the field of real-time process control, has led to the development of a parallel meta language (TPML) which offers a generic tool for programming transputer platforms. The meta language complies with the Bulk Synchronous Parallel (BSP) processing model proposed by Valiant [1], and it is seen as a prototype for implementations on other distributed hardware architectures. This paper argues that the TPML offers a simple, yet general, solution to the problems associated with programming parallel hardware architectures.

The TPML 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. Any alteration to the processor connectivity is handled internally by the TPML, so that the specification of communications in the application programs remains unchanged. The complete specification of the parallel processing is possible through a small set of TPML commands.

A prototype of the TPML has been implemented in 3L Parallel Fortran 77 targetted at a transputer platform. In order to assess the overheads associated with the use of TPML an existing application code has been converted to TPML.

Keywords

Distributed architectures, Parallel programming, Bulk Synchronous Parallel processing model.

1. Introduction

Parallel processing has always held a great promise of the high efficiency and scalability of computations. This is particularly attractive in scientific and engineering application domains. Most real-time systems, for example, require a fast response time alongwith high computational power - parallel processing is seen as a means for the achievement of these goals [2,3,4]. 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 [5]. Scientists and engineers want to concentrate on solving problems from their application domains - and do not wish to be distracted by any low level parallel programming considerations [6].

Another, possibly more important, reason for the limited use of parallel hardware is the lack of portability of applications between various parallel computers. 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. What appears to be a missing ingredient, in parallel systems development, is 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.

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 [7]. A recently proposed candidate to such a role is a Bulk Synchronous Parallel (BSP) processing model [1]. Although it has been originally intended to be a tool for the evaluation of computational complexity of parallel algorithms, it seems to be well suited for a more general role of a logical model of parallel computation. In essence, the BSP model assumes that any parallel processing can be expressed as an alternating sequence of parallel computations and communications. In order to maintain the integrity of the data communicated between tasks, all parallel computational processes synchronise before the communication begins - thus, the Bulk Synchronous Processing name. The BSP model is general in that it does not specify the nature of interprocess communications which can be implemented either by means of sending/receiving messages or reading/writing global variables.

A parallel meta language - TPML, developed at The Nottingham Trent University, represents an implementation of a BSP processing model. It has been originally developed and implemented as an extension to the parallel (3L) Fortran for a transputer platform. This development can be easily extended to any MIMD (multiple instruction, multiple data streams) computer and to other programming languages.

A meta language approach to the implementation of the BSP processing model was adopted in order to make full use of the available software development tools. However, it is envisaged that in future the TPML functionality will be available through a dedicated parallel compiler [8].

2. Parallel Meta Language (TPML)

The parallel meta language enables a logical specification of parallel tasks and intertask 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 meta language 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 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, in the case of 3L Fortran, 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.

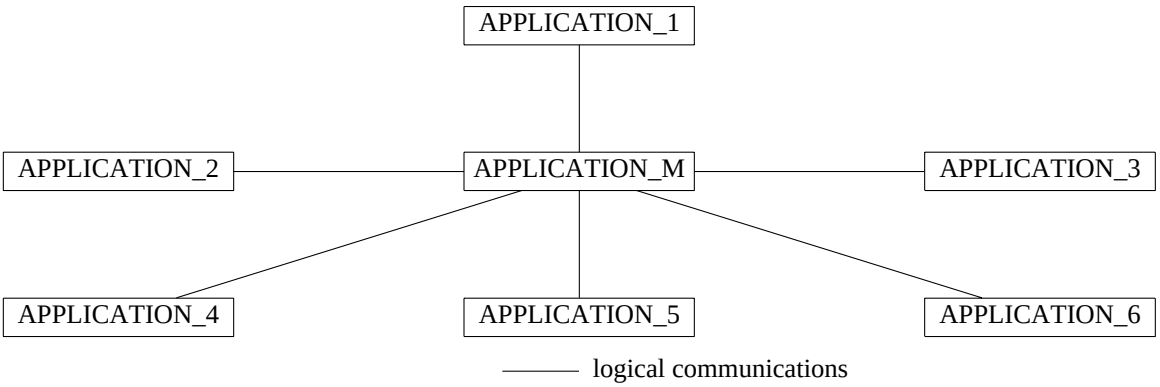

Figure 1a Logical inter-task communications

Figure 1a 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 commmunication 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 1b illustrates 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 1b.

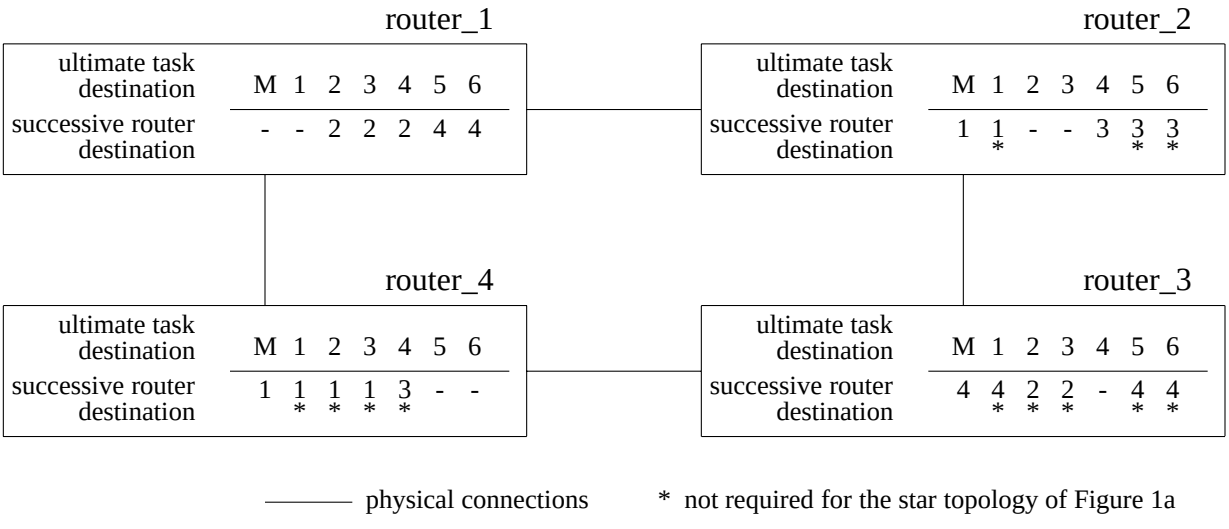

Figure 1b Routing tasks configuration

The configuration of the physical system, in terms of task placement and the processor connectivity is illustrated in Figure 1c. Clearly this configuration is dependent on the number of available processors and the links between them. The advantage of using the TPML is that it enables the application programmer to abstract from the hardware dependent considerations (Figures 1b and 1c) and to concentrate on the hardware independent logical view (Figure 1a).

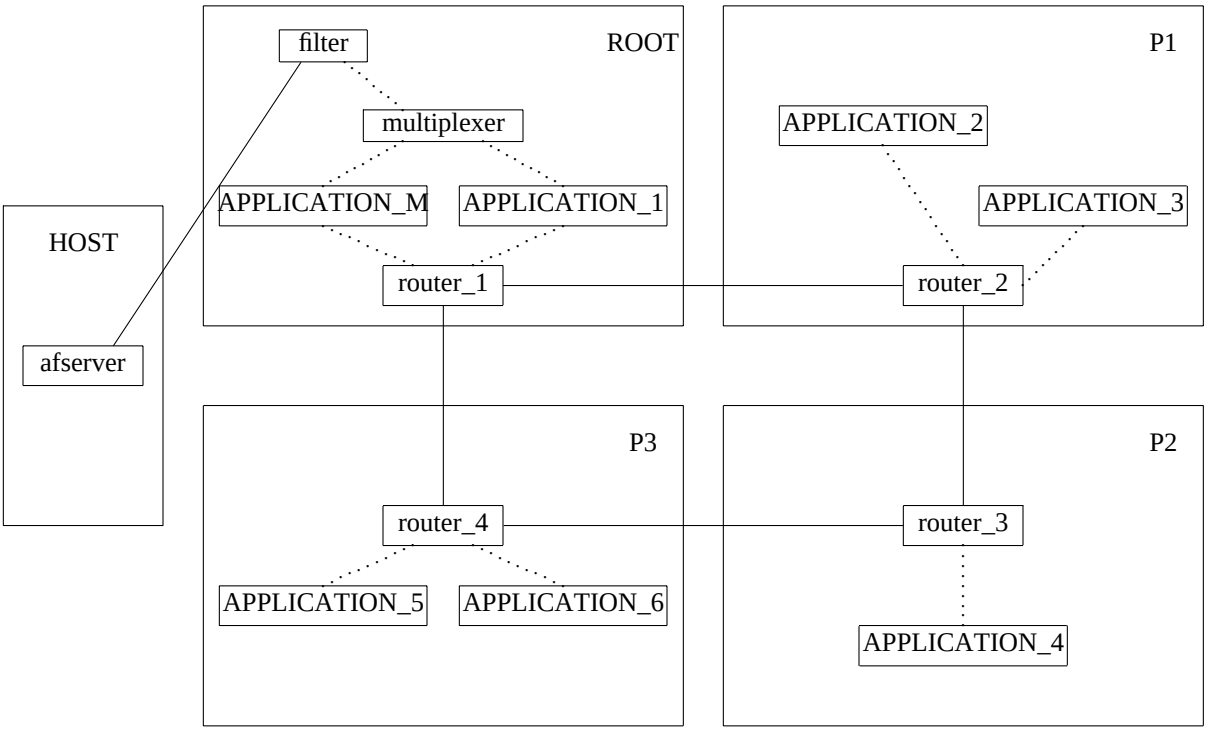

Figure 1c. Physical system configuration

3. 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, intertask communication and superstep synchronisation.

Task management:

Each task is bracketted by a pair

c>>tpml_task_begin(task_id) or **c>>tpml_task_begin(task_id, same_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, alongwith any subroutines or functions that might be accessed during the execution of the program. In the case of a number of identical tasks being created, it is not necessary to duplicate the code for each task. Instead, the initialising command is given two arguments - the first naming the task, and the second defining the task to be copied.

Intertask 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 meta language simple, we have allowed that the data packet is of an integer array type only. It has been left to the programmer to pack/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 synchronization:

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 which 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.

For programming reasons, an auxiliary facility of an explicit label to mark the end of the application code has been included within the TPML. Its syntax is:

c>>tpml_end.

Also, the command

c>>tpml_kill

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

4. Meta Language Compilation

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

- extraction of the code (duplication if necessary),
- 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.

4.1 Generation of tasks

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 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.

The router tasks are constructed by adding details of a physical hardware to the logical specification of inter-task communication (as programmed in TPML). The additional information is retrieved automatically by the TPML compiler and consists of: the number of available processors, the processor connectivity and the number of communication ports needed. Based on this, the TPML 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 1b).

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.

4.2 Augmentation of code by communication commands

The TPML 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 which are responsible for this message delivery. The implicit synchronisation between the application and the communication router task, makes it possible to raise more than one communication request in a single BSP superstep. 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 channnels, 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 TPML 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'), alongwith 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, specified below. These subroutines are included at the end of each application task's code:

```
      subroutine tpml_send(task,data,length)
c     ==============================
c --- Send integer array 'data' to 'task' via a router
      CALL F77_CHAN_OUT_WORD(4*length, outport1)
      CALL F77_CHAN_OUT_MESSAGE(4*length, data, outport1)
      CALL F77_CHAN_OUT_WORD(task, outport1)
      return
      end
c
      subroutine tpml_receive(data,length)
c     ============================
c --- Send message to router to indicate that the task is ready 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*4, inport1)
      CALL F77_CHAN_IN_MESSAGE(length*4, data, inport1)
      CALL F77_CHAN_IN_WORD(task, inport1)
      return
      end
```

The formal parameter 'length' in the tpml_receive subroutine is private to TPML and is used whilst unpacking the data array.

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

4.3 Generation of the configuration file

The configuration file is a file read by the afserver program determining 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 onto particular processors in the physical network.

The TPML 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. Tasks placed on the same processor can have any number of interconnecting channels. Tasks placed on different processors can only be connected where physical wires connect the processors' links. Each logical connection, between two tasks placed on different processors, is assigned exclusive use of one of the physical link channels connecting these processors. The number of interconnections between tasks 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. The implication of this

limitation for the TPML application is that all tasks requiring access to the afserver, to enable data to be sent along the I/O channels, must lie on the root processor. An example of the TPML-generated configuration file, tpml.cfg, is given below:

```
processor HOST
processor ROOT
wire ? ROOT[0] HOST[0]                          ! physical wire connection between the root and host
processor P001
wire ? P001[1] ROOT[2]
processor P002
wire ? P002[1] P001[2]
processor P002
wire ? P003[1] ROOT[2]
processor P003
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 image file router_1.b4
Task tpml_router2   File=router_2   Ins=4  Outs=4     ! tpml_router2 has 4 input and output ports
Task tpml_router3   File=router_3   Ins=3  Outs=3
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
:
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 processor
Place application_1   Root
:
Place application_6   p003
Place tpml_router1    Root
Place tpml_router2    p001
Place tpml_router3    p002
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]
Connect ? application_m[1]   multiplexer[1]      ! logical connection from application_m to task multiplexer
Connect ? multiplexer[2]     application_1[1]
Connect ? application_1[1]   multiplexer[2]
Connect ? tpml_router1[0]    application_m[2]
Connect ? application_m[2]   tpml_router1[0]
Connect ? tpml_router1[1]    application_1[2]
Connect ? application_[2]    tpml_router1[1]
Connect ? tpml_router1[2]    tpml_router2[0]     ! tpml_router1 has output port 2, tpml_router2 has input port 0
Connect ? tpml_router2[0]    tpml_router1[2]
:
Connect ? application_6[0]   tpml_router4[0]
Connect ? tpml_router4[0]    application_6[0]
```

The information about the number and the interconnections between the transputers is extracted by the TPML compiler automatically by means of the 'tworm' utility. 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 which map the logical task connectivity onto the variants of physical processor connectivity.

The TPML assigns a data value of 50k to the filter task and 500k to each of the application tasks. These values can be altered by the programmer if these values are not appropriate. The router tasks use the remaining memory available to the corresponding processor. If required, the multiplexer task is placed on the root processor, allowing more than one task to use the standard file I/O, by merging a number of request streams into a single stream. The multiplexer task is typically allocated less than 10k of memory.

5. Compilation of the TPML Program

When the user types in the command line **tpml**, a list of system commands are automatically executed.

```
tworm -c> tpml.cfg          # generates processor interconnection information and writes it to the file tpml.cfg
trun t_parse.app            # invokes the afserver process to execute t_parse.app instigating compilation of
                              the programmer's code
tff -t -g application_m.f   # compiles application_m.f as a task to be later configured
tff -t -g application_1.f
tff -t -s -g application_2.f # compiles application_2.f with the stand-alone library as a task to be later
:                             configured
tff -t -s -g application_6.f
tff -t -s -g router_1.f
tff -t -s -g router_2.f
tff -t -s -g router_3.f
tff -t -s -g router_4.f
tconfig tpml.cfg tpml.app   # generates tpml.app from the interconnected task images specified in the file
                              tpml.cfg
trun tpml.app               # invokes the afserver process to execute tpml.app across the transputer network
```

6. Concluding Remarks

The parallel meta language (TPML) 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. The TPML 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 the TPML, so that the specification of communications in the application programs remains unchanged. The TPML has been applied to an existing distributed state estimator [9] - the development of an earlier diakoptical simulation algorithm implemented for use with nonlinear networks [10,11]. In implementing this task, TPML has proven itself to be a convenient means of portable parallel programming.

References

1. VALIANT, L.G. "A Bridging Model for Parallel Computation" *Communications of the ACM*, Vol. 33, No. 8, August 1990.

2. CHAJAKIS, E.D., ZENIOS, S.A. "Synchronous and Asynchronous Implementations of Relaxation Algorithms for Nonlinear Network Optimisation" *Parallel Computing*, Vol. 17, pp 873-94, 1991.

3. FU-CHAW YU, YAN-PEI WU "A Multilevel Diakoptics Algorithm for Large Scale Networks" *Journal of the Chinese Institute of Engineers*, Vol. 11, Iss. 6, pp 667-79, Nov. 1988.

4. LIU, Z., THORELLI, L-E. "HSIM: A Distributed Simulation Environment Based on Trollius and Transputers" Parallel Computing and Transputer Applications, edited by M.Valero et al, IOS Press/CIMNE, Barcelona, 1992.

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

6. GAJDAROV, D. "LYNX - A New Language for Parallel Programming" Parallel Computing and Transputer Applications, edited by M.Valero et al, IOS Press/CIMNE, Barcelona, 1992.

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

8. ANDERSON, J., LAM, M. "Compiler Optimisations for Scalable Parallel Machines" *Parallel Update*, Vol. 17, Jan. 1994.

9. BARGIELA, A., HARTLEY, J.K. "Diakoptical State Estimation of Nonlinear Networks" in preparation *(preprints available from the authors).*

10. HOSSEINZAMAN, A., BARGIELA, A. "Parallel Simulation of Nonlinear Networks using Diakoptics" *Proc. PACTA '92*, Sept. 1992.

11. BARGIELA, A. "Nonlinear Network Tearing Algorithm for Transputer System Implementation" *Proc. TAPA '92*, pp 19-24, Nov. 1992.