in Proceedings of ZEUS 95 (ed. P. Fritzon and L. Finmo) IOS Press

# A Methodology for Initiating Arbitrary Structured Programs in PARIX by Interpreting Graphs

J.Y. Cotronis
Dept. of Informatics, Univ. of Athens, Panepistimiopolis, 157 71 Athens, GREECE.
tel.: +30 1 7230172 fax: +30 1 7219 561 e-mail: cotronis@di.uoa.ariadne-t.gr

**Abstract.** We present a distributed program design methodology by which we overcome the programming overhead effort required for creating arbitrary structured parallel applications in PARIX. We develop general program module creation and interconnection techniques. We propose program module programming principles permitting the reusability of program modules, in any distributed application, as library components. Parallel applications are virtually specified by Process Communication Graphs (PCGs) which are interpreted by a generator which automatically creates the appropriate processes and establishes their communication dependencies initiating PARIX applications. We have demonstrated the general principles of the methodology for parallel query optimization and execution. The methodology has been applied to the PVM programming environment.

## 1. Introduction

Distributed programming in PARIX [7] allows for the most general form of MIMD parallel computation, as application programs may possess arbitrary control and dependency structures. At any point in the execution of a concurrent application, the processes in existence may have arbitrary relationships between each other and in addition, any process may communicate and/or synchronize with any other. As in all programming environments there are application types that are most suited to the PARIX characteristics, making them easy to program and application types that are not so well suited and are thus much more difficult to program. Let us briefly present the PARIX characteristics:

1. Throughout an application program execution a set of processors, a partition or a network of processors, are exclusively reserved and managed as a private resource of the application program.

2. All processors in the partition are loaded with the same initial main program module, which is an executable residing on the host's file space.

3. A set of global data kept at each processor allows identification of the "own" position within the network. Depending on this position it is possible to execute different sections of code or execute identical instructions on different data.

4. A program may create light-weight processes, called also threads, handling asynchronous services. Threads are running concurrently in the same context, i.e. an environment with its own code and data, and share all global variables defined in the program. Variable protection and synchronization between threads in the same context may be achieved by using semaphore operations. Contexts cannot migrate to another processor.

5. It is possible to load and run additional code at any time on a processor by a thread issuing an Execute call. This call loads an executable program module on the processor the

calling threads runs on and creates a new context separated from the calling thread which waits for termination of the new context. More than one contexts may run on the same processor.

6. Communication between threads may be synchronous (S) or asynchronous (A). Communication may be based on virtual links (L) which build point-to-point connections between threads. A set of virtual links can be combined to build a virtual topology (T). Communication could be random (R), that is not requiring the definition of virtual links or topologies. There are routines for sending and receiving messages implementing the above communication types, as shown on the following table:

| communication types | **virtual links (L)** | **topology (T)** | **random (R)** |
|---|---|---|---|
| **synchronous (S)** | SendLink, RecvLink | Send, Recv | SendNode, RecvNode |
| **asynchronous (A)** | | ASend, ARecv | PutMessage, GetMessage |

There are five communication possibilities, denoted hereafter by SL, ST, SR, AT, AR from the corresponding row (S or A) and column (L, T, R). Note that there is no AL communication routines, that is asynchronous communication over links is not supported.

All communication types define, explicitely or implicitly, communication channels between threads. All programming notations based on message passing between processes provide channels of some form and primitives for sending to and receiving from them [1]. A channel is an abstraction of the physical communication network in that it provides a communication path between threads. When virtual links are used the channels are explicitely defined by point-to-point links between threads. A link between two threads is established when each calls a connection routine (e.g. ConnectLink) giving as parameters each other's processor identifier on which they are running and the same unique Request Identifier integer. Topologies group links under a topology name and give them unique symbolic names within a topology, thus defining channels explicitly and abstractly. In random communication the channels are implicitly specified by referring directly to the processor identifiers the two threads are running on and a unique Request Identifier .

Consequently, in all communication types between two threads the primary information are the processor identifiers the participating threads are running on and a unique Request Identifier. As threads do not migrate the processor identifier specifies the position of the thread and the request identifier the particular channel of communication as there may be more than one thread running on a processor, or the same thread may communicate with two or more threads. Random communication directly uses the primary information, communication over links needs it to define the links and topology communication needs already defined links to define the topology.

Applications of the SPMD paradigm are easy to program in PARIX. All processors are loaded with the same program module and each, depending on its position, operates on a different set of data. Also well-structured applications which consist of the same program module loaded in all processors, each communicating with others is some regular way (e.g. ring buffer, grid, complete binary tree, hypercube, etc.) is well suited to the PARIX model. For these cases library routines have been defined establishing virtual topologies relieving the programmer from having to built them.

However, programming arbitrarily structured application programs is not, in general, an easy task in PARIX. For example, when the communication dependencies of program modules does not form complete binary trees. In this case the same program module is loaded on all processors but as the communication dependencies are not regular the programmer has to program the particular communication dependencies. It is even more difficult when different types of program modules are required to be loaded and the communication dependencies between them form arbitrary graphs. As links and topologies

are more efficient in PARIX it is desirable to be able to establish links and topologies for arbitrary structured applications.

Establishing arbitrary structured graph-like process communication dependencies *requires a substantial programming effort*. The effort is twofold:

(i) Programming the loading of the various program modules. In PARIX the same program has to be loaded initially on all processors in the partition. A "main" program has therefore to be loaded which, depending on the processor's position, loads the appropriate program module. The programmer has to write a different "main" program for each distributed application, which *burdens the design of the distributed application*.

(ii) Programming of the communication dependencies between program modules directly in the source program, which *limit the reusability of the program components*. Their reusability is limited since such components rely on the specific dependencies they are involved in and they cannot be used without modification to establish a different communication dependency structure.

In this paper we address the problem of automatically initializing distributed applications in PARIX exhibiting any static process communication dependencies between their program modules. We develop generic program module loading and interconnecting techniques and suggest program module programming principles which permit their reusability as library components. If an application requires a reusable program component it only needs to initiate a copy of its executable file passing connectivity parameters. A reusable program component may have many instantiations in the same application, each with its own communication dependencies. A reusable program module may also be used in other applications, where its instantiations may communicate with altogether different modules, copies of other library components, again without modification. The effort is in three directions:

1. We have used Process Communication Graphs (PCG), as a general structure for specifying the modules and their communication dependencies. Nodes on a PCG denote program modules and arcs the communication dependencies between them. PCGs have been used in modeling [1], in dynamic analysis and simulation [9,10], in mapping techniques [2,5], etc. We shall use Process Communication Graphs as interpretable specifications for initiating distributed programs. The PCGs annotated with appropriate information may be interpreted by a PARIX generator which initializes the applications specified by the PCGs.

2. We have developed one generator for all PARIX applications, which reads an annotated PCG and loads the appropriate program module on each processor passing to it its communication parameters. Thus, a programmer does not have to write a different "main" program for each application.

3. The generator requires that the program modules are reusable without any modification. Program modules should not assume any specific communication structure in which they are involved, but only generally specify the number and type of communication channels. We define general channel descriptions of program modules which upon their creation, build the required communication structure.

The structure of the paper is as follows: in section 2 we present the Process Communication Graphs and their annotation; in section 3 we present the application generator which interprets annotated PCGs; in section 4 we present the design principles of reusable program modules in PARIX; in section 5 we outline a parallel query optimization and execution application we have developed as a case study of our methodology. In section 6 we present our conclusions and plans for future work.

## 2. The PCGs and their Annotation

Process Communication Graphs (PCG) are used as a general structure for specifying the modules and their communication dependencies. Let us define the elements of the PCGs. We shall refer to fig. 1 depicting a PCG to give examples of the PCG elements. Each node on a PCG, depicted as a circle, denotes a copy of a program module to be loaded; the names of the program modules appear within the circles. In our example PCG there are seven copies of program modules: one of program module A and two of each of program components B, C and D. Each program module has some communication ports, which are indexed by integers. In our example, program module A has four communication ports, B has five and C and D four; their index appears within the circles near the circumference. Ports correspond to the interface of the communication channels of program modules. Communication dependencies are depicted by arcs joining module ports. For example, ports 1 and 2 of module copy of A interchanges messages with port 1 of each of the two copies of B, respectively.

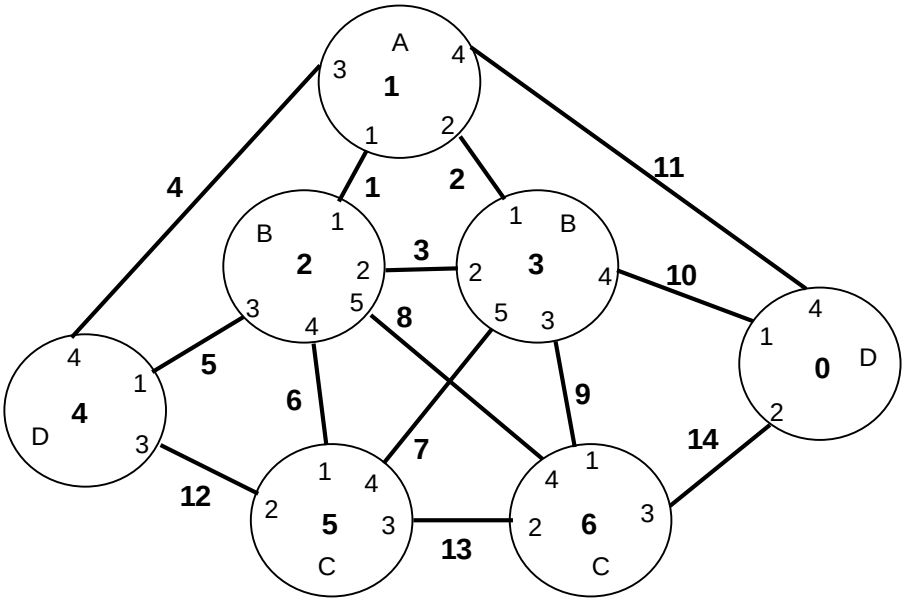


figure 1. An example of PCG

For the a PCG to specify a complete application it needs to be further annotated. We need to specify on which processor each copy will be loaded. Essentially, this is the mapping of the application as code is assigned to processors. The PCG may be annotated by the programmer, or automatically if there exists a mapping algorithm for the application. The nodes are annotated by consecutive integers starting from zero, which are the processor identifiers in a PARIX partition. In the PCG representation the integers appear in bold near the centers of the circles. For example the only copy of A is to be loaded on processor 1 and the two copies of B on processors 2 and 3 respectively.

Arcs on the PCGs represent communication channels. For a complete communication channel specification we need request identifiers, which are used by both sending and receiving threads. The request identifiers annotate the arcs of the PCG. The annotation may be performed automatically by a program giving unique integers to arcs.

The annotated PCG depicts all the characteristics of the application: the number of copies of each module type, and their interconnection; the processors to be loaded on; the request identifiers uniquely specifying messages to the same module copy. The annotated PCGs may now be interpreted by a generator program, common for all applications.

## 3. The Application Generator

The generator is the program that is responsible for initializing the distributed application. It is the general "main" program which is loaded on all processors on the partition with a specific PCG as argument. The applications will be initialized with the run command issued from the PARIX host in the form:

run -a "partition" generator "specific PCG"

The generator is loaded on each of the processors in the partition. Its action is simple. It first obtains the Processor Identifier it is running on, say MyProcID. It then visits each of PCG's nodes once and if a node is allocated to MyProcID it loads a copy of the module-type denoted on the node passing it appropriate parameters. The parameters are a list of value pairs providing the connectivity information for each of its ports. For each port two values must be provided (ProcID, RequestID). For example when creating the module type A on processor 1 the port connectivity information passed as parameters is

| port connectivity information | explanation |
|---|---|
| 1, 2, 1 | your port 1 is connected to Processor 2; ReqID is 1 |
| 2, 3, 2 | your port 2 is connected to Processor 3; ReqID is 2 |
| 3, 4, 4 | your port 3 is connected to Processor 4; ReqID is 4 |
| 4, 0, 11 | your port 4 is connected to Processor 0; ReqID is 11 |

The generator calls the Execute routine, having as first parameter the name of the executable file to be loaded and as a second its connectivity parameter list. In C programming under PARIX the parameter list has to be given as an environment pointer. The copies of the modules are now appropriately loaded. They have now to be interconnected as specified by the parameter list.

## 4. The Design of Reusable Program Modules

Reusability of program components means that they are compiled in executable files and, if required by an application, any number of copies of them can be started as processes. It should also be possible to dynamically establish upon their creation the appropriate communication channels between them. As the number of actual processes and their communication dependencies is not fixed, the program components should only specify the number and type of communication channels in a general way. A program module should provide the means for the establishment of actual communication between any module-copy of it with any other module-copies.

A general channel specification is defined as a data structure, called port, storing (ProcID, ReqID) pairs. The two elements in the port structure (ProcID; ReqID) indicate the primary connectivity information of the communication channel associated with a port which would be available at the time of creation. A program module has as many ports as channels, organized in a list or array of ports, which we call the interface. Each port is identified by its index to the interface. For the time being we shall limit our discussion to interfaces having a fixed number of ports. Without loss of generality communication and virtual link creation routines have to use as their parameters expressions of the form port[i].ProcID and port[i].ReqID.

The establishment of the communication structures may be achieved by setting values to ports from the value pairs (ProcID, ReqID) passed as parameters in the Execute call of the generator at the time of process creation; the interface of the process on processor 2 will have the following values:

**Interface**

| port ID | ProcID | ReqID |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 3 | 3 |
| 3 | 4 | 5 |
| 4 | 5 | 6 |
| 5 | 6 | 8 |

Having set up the interface the complete application has been initialised. The program modules are now completely reusable as they only specify general communication specification and do not assume communication with any particular module nor on which processor they should be loaded.

If however, the program modules use links (or topologies) the modules may not be reusable; as a matter of fact the program may not be initialized at all, coming to a deadlock. The reason is that for the establishment of virtual links, symmetric calls have to be made by the participating threads. Each thread calls the ConnectLink routine, which in order to create the link requires synchronous communication between the threads. This means that both threads should be able to reach their corresponding ConnectLink calls. The general problem is exemplified in the PARIX manual [7], chapter 10, pg. 90-97, where a ring topology is created. There, N processes are to be connected in a ring; each has two links, one left and one right. If all processes try first to connect to their left processor and then to their right, a deadlock occurs. The solution suggested is for the processes residing on a even ProcID to connect to their right first and then to their left, and for processes residing on an odd ProcID to connect first to their left and then to their right.

This example demonstrates that for the simplest of topologies care should be given to the order of calling ConnectLink routines. Significant design effort is required for more complex topologies. For the program modules to be completely reusable they cannot depend on such orderings, as they depend on other modules, which have not been created. Therefore, we organize program modules establishing virtual links by using parallel threads, one for each communication channel, as the alternative suggested in [7]. A module will therefore consist of a thread implementing the application domain computations and a number of threads responsible for establishing the virtual links and performing the communication with the other communication threads of other modules.

Modules in this structure are completely reusable as the virtual links are established by asynchronous processes the activation of which is performed by PARIX. No extra effort is required by the programmer.The methodology for application development we have introduced is depicted in fig. 2.

## 5. Parallel Optimization and Execution of RDBMS Queries in PARIX

We have applied this methodology on two applications for the parallel optimization and execution of RDBMS queries. We first describe the basic model for query execution and then present the two applications.

### *5.1 The Query Execution Model: Tree Pipelining*

In the Tree Pipelining query execution Model (TPM) [8,14] a query is transformed into a query tree representation, which is then optimized for parallel execution [11,16]. A cost model for the estimation of communication and I/O costs in parallel execution spaces according to TPM [15] has been used in exhaustive parallel optimization [13,14] and in parallelized enhanced iterative improvement [12,13].

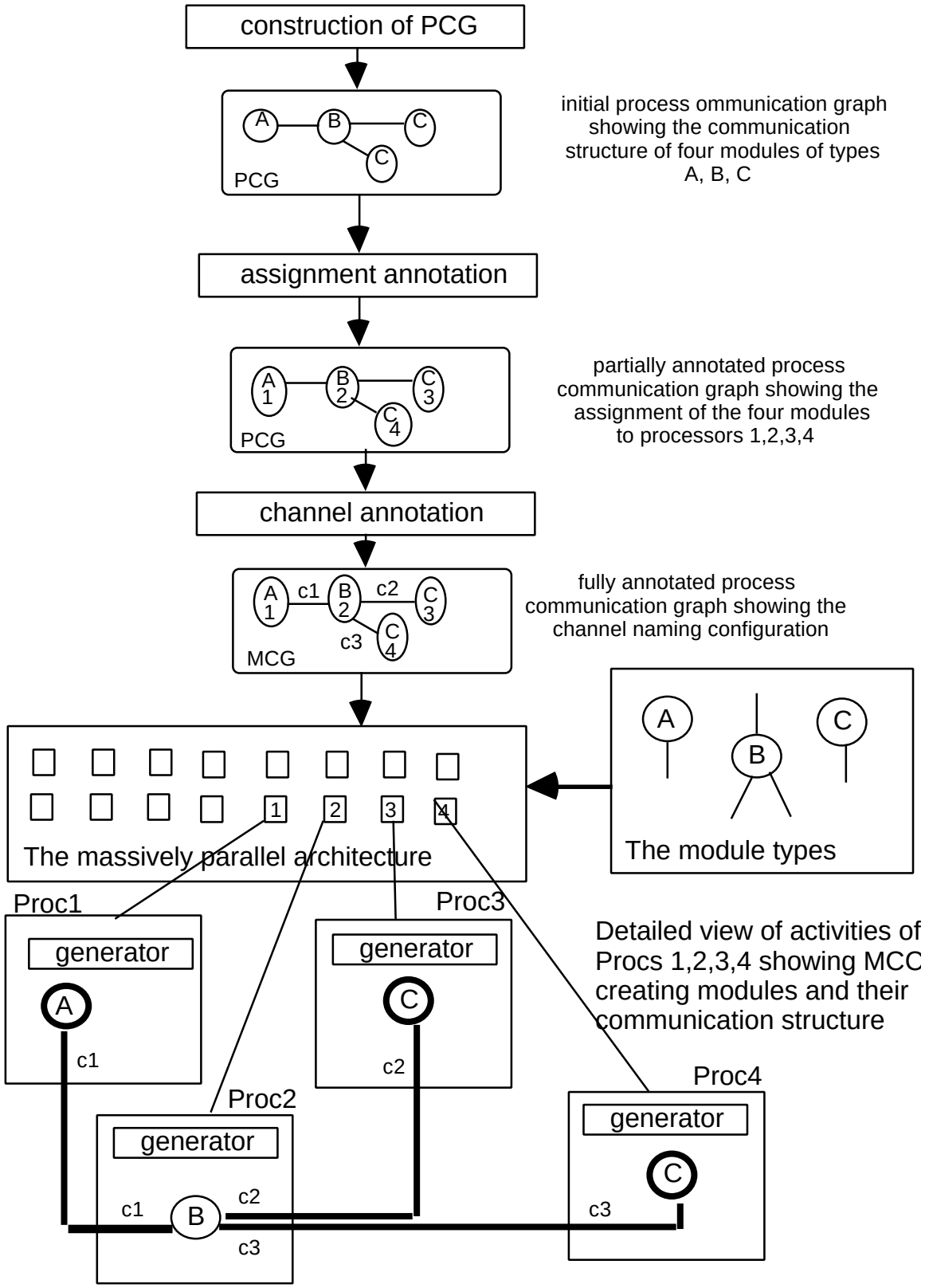


fig.2: A general overiew of the methodology

An optimized query tree is the process communication graph of the query execution. The nodes of the tree represent relational algebra operators and the arcs represent communication of results from children to parent nodes. The intrinsic parallelism of the tree representation is thus fully utilized: (i) processes on all leaf-processors may start execution immediately, as they have the base relations available, producing tuples of intermediate relations and propagating them to their parent nodes and (ii) processes on inner processor-nodes start execution in pipeline mode as soon as they have sufficient tuples to operate on.

*5.2 Parallel Optimization of Queries*

The optimization program, consists of a master module which distributes common data (the tree representation and dictionary information) to N (where N is number of join nodes of tree) similar optimization worker-modules running on p processors. The master module maintains a clock measuring device, collects the results (local minima) from the N processes and selects the best result (optimal query tree), which is used in the execution phase.

The initial process graph (master and N workers) is constructed. This process graph is annotated with channel and assignment information (N modules on p processors identified by integers), and given as input to the p+1 generator creating modules, creating one master and the N workers executing the optimization on p processors.

The optimal query tree produced by the optimization is the process graph specifying the distributed program for the parallel execution of the query. Thus, the optimization program produces the process graph, a tree, for the execution of the query.

*5.3 Parallel Execution of Queries*

The optimized query tree, the process graph of the execution, is now annotated and given as input to each generator running on the processors. Each generator creates copies of relational algebra modules on the processors passing as parameters the connectivity information. Each module connects with its parent and children modules. This is achieved by defining the join modules, for example, as a collection of four threads, one main and three satellite, communicating via local shared memory: the main thread executes the actual join algorithm on relations stored to the shared memory, one satellite establishing a virtual link with the parent processor and propagating the results to it by asynchronous communication and two more satellite threads establishing virtual links with the children processors, asynchronously receiving results from them and putting them to the shared memory for processing, (fig. 3).

## 6. Conclusions and Future Work

We have presented a distributed program design methodology by which we overcome the programming overhead effort required for creating arbitrary structured parallel applications in PARIX. In this methodology parallel applications are virtually specified by Process Communication Graphs (PCGs) which are interpreted by a general PARIX-specific generator which automatically creates the appropriate PARIX module copies establishing their communication dependencies. We have proposed program component programming principles permitting their reusability as library components.

We have applied the methodology on parallel query optimization and execution. An interesting aspect of these applications is that the PCGs are automatically produced by other programs and not by the programmer and the distributed programs corresponding to

the PCGs are created and executed according to our methodology. The SQL query is transformed to an unoptimized query tree. The optimization PCG, a master/slave configuration is set up, created and executed producing an optimized query tree, the PCG of the query execution. The distributed program is generated and executed producing the query result. The methodology opens up a new possibility of automatic synthesis of parallel programs embedded in larger applications, such as parallel transaction systems. We shall explore this type of applications in the future.

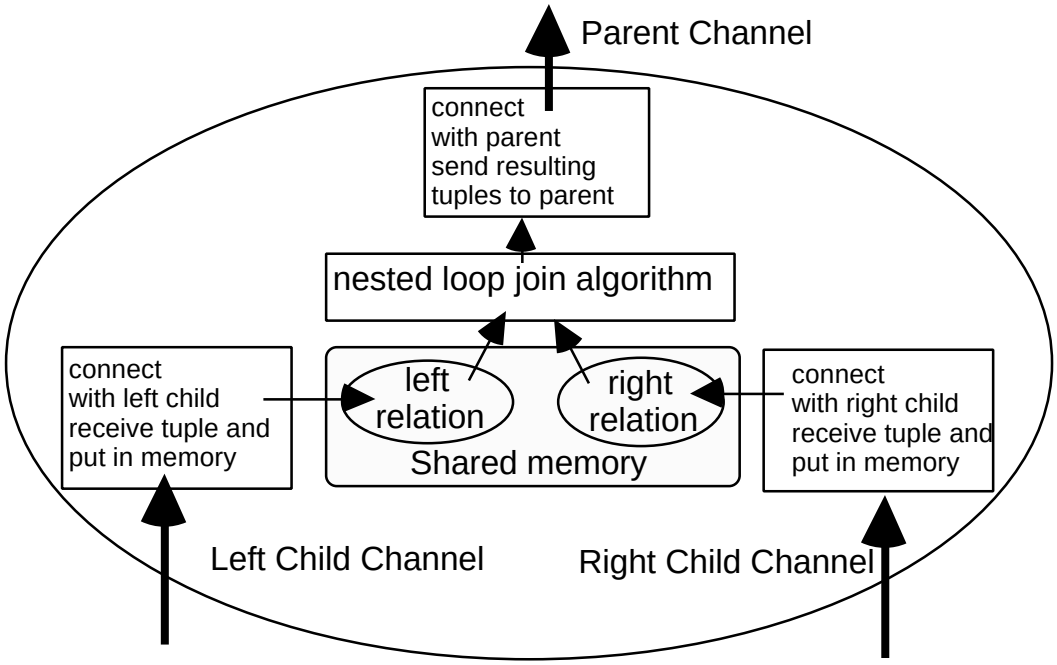


Fig. 3: Schematic presentation of a module performing nested loop joi
implemented as four concurrent processes using shared memory

The methodology may be applied to other message passing parallel environments by developing other system specific generators. We have applied it to the PVM system [3,4]. Although, the PVM environment imposes an altogether different program initialization model than PARIX, based on parent-child creation of processes, it was possible to apply the same methodology in principle and achieve reusability of program modules. We shall compare implementations of the methodology under PVM and Parix in a future report.

Our methodology is related to the reusability in Object Oriented systems by using objects and scripts [6] encouraging a component oriented approach to application development. The PCGs may be viewed as the scripts that bind modules together. We shall pursue this comparison further in the future.

**References**

[ 1] G.R.Andrews: ‘Paradigms for Process Interaction in Distributed Programs’, ACM Computing Surveys, Vol. 23, No.1, March 91.

[ 2] F. Berman, L.Snyder, ‘On mapping parallel algorithms into parallel architectures’, J. Parall. Distrib. Comput. 4, 5, 439-458.

[ 3] J.Y.Cotronis: “EASY-SPAWN: A Methodology for Initiating Unstructured distributed Programs in PVM by Interpreting Graphs”, internal report.

[ 4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, Vaidy Sunderam, ‘PVM 3 User’s guide and Reference Manual’, ORNL/TM-12187, May 1994.

[ 5] M.G.Norman, P.Thanisch, ‘Mapping in Multicomputers’, ACM Computing Surveys, Vol25, No.3, September 93.

[ 6] O. Nierstratz, D. Tsichritzis, V. de Mey, M. Stadelmann, ‘Objects+Scripts=Applications’, in Poceedings, Esprit 1991 Conference, Kluwer Academic Publishers, 1991, pp. 534-552.

[ 7] Parix1.2, Manual.

[ 8] G. Philokyprou, C. Halatsis, M. Hatzopoulos, J. Cotronis, D. Nikolos, M. Spiliopoulou, N. Flessas, D. Koutoulas, T. Kalentzos and G. Platanakis. ‘Implementation and evaluation of database management systems in parallel environment’. Report SPAN-WP14-6 for 1588-ESPRIT, Univ. of Athens, Dept. of Informatics, Dec. 1989.

[ 9] P.Pouzet, J.Paris, V.Jorrand, ‘Parallel Application Design: The Simulation Approach with HASTE’, Proc. High Performance Computing and Networking, Munich, April 18-20, 1994, Vol II, pp. 379-393.

[10] C. Scheidler, L.Schaefers, ‘TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications’, PARLE Conf., Munich, 403-413, 1993.

[ 11] M. Spiliopoulou, M. Hatzopoulos. translation of SQL Queries into a graph structure: query transformations and pre-optimization issues in a pipelined multiprocessor environment. Information Systems 17,2 (1992)

[12] M. Spiliopoulou, J. Cotronis, M. Hatzopoulos, Parallel Optimisation of Join Queries using an Enhanced Iterative Improvement Technique, PARLE Conf., 716-719, Munich, 1993.

[13] M. Spiliopoulou, M. Hatzopoulos, J. Cotronis, ‘Parallel Optimisation of Large Join Queries with Set Operators and Aggregates in a Parallel Environment Supporting Pipeline’. Submitted to IEEE-TKDE.

[14] M. Spiliopoulou, M. Hatzopoulos, C. Vassilakis, ‘Using parallelism and pipeline for the optimisation of join queries’ Proc. 1992 PARLE Conf., Paris, France, 279-294, 1992.

[15] M. Spiliopoulou, M. Hatzopoulos, C. Vassilakis, ‘A Cost Model for the Estimation of Query Execution Time in a Parallel Environment Supporting Pipeline. Submitted to Computers and Artificial Intelligence.

[16] A. Swami, A. Gupta, ‘Optimization of Large Join Queries’, Proc. SIGMOD Conf., ACM, Chicago, Illinois, 8-17, 1988.