P-RIO: An Environment for Modular Parallel Programming

Orlando Loques  
Julius Leite  
{loques, julius}@caa.uff.br  
CAA - UFF  
Rua Passo da Pátria, 156  
Niterói - RJ, 24210-240  
Brazil

ABSTRACT

This report presents the P-RIO\(^1\) environment which offers high-level, but straightforward, concepts for parallel and distributed programming. A simple object-based software construction methodology facilitates modularity and code reuse. This methodology promotes a clear separation of the individual sequential computation components from the interconnection structure used for the interaction between these components. The mapping from concepts associated with the software construction methodology to their graphical representations is immediate. P-RIO includes a graphical programming tool, has a modular construction, is highly portable, and provides run-time support mechanisms for parallel programs, in architectures composed of heterogeneous computing nodes.

INTRODUCTION

P-RIO\(^1\) tries to offer high-level, concepts for parallel and distributed programming. A simple software construction methodology makes most of the object-based technology properties available, facilitating modularity and code reuse. This methodology promotes a clear separation of the individual sequential computation components from the interconnection structure used for the interaction between these components. In addition to the language used for the programming of the sequential components, a configuration language is used to describe program composition and its interconnection structure. This separation into two languages makes the data and control interactions explicit, simplifying program visualization and understanding. The mapping from concepts associated with the software construction methodology to their graphical representations is immediate. Hence, P-RIO includes a graphical tool that provides system visualization features that help to configure, monitor and debug a parallel program.

The current implementation is based on message passing, which has strongly influenced the presentation of this report. However, we also provide some insights on application of this methodology in a distributed shared memory environment.

P-RIO CONCEPTS

The P-RIO methodology hinges on the configuration paradigm\([2,3]\) whereby a system or program can be assembled by the external interconnection of modules obtained from a library. The concept distinguishes the program modules used to achieve the functions of the application program from the connections required for the interaction

\(^1\) P-RIO stands for Parallel - Reconfigurable Interconnectable Objects

595
between these modules. A special (textual or graphical) configuration or architectural description language is used to specify the system module composition and the interconnection structure (fig. 1), used for the interactions between modules. In particular, different interaction mechanisms can be selected for use in distinct connections. The scheme caters for interconnection changes as the program evolves and helps the re-use of program modules in other systems with different interconnections and interaction mechanisms.

The points of interaction and interconnection between class instances are defined by named ports, which are associated with their parent classes. Ports themselves are defined as port classes that can be reused in different module classes to define their particular communication interfaces. A port instance is named and referenced for communication in the context of its owner module class. This provides configuration-level port-naming independence helping the re-use of module classes in different configurations. The configuration description language provides a link construct that is used to specify the connection of ports belonging to different modules.

Figure 1. An Example of Composite Module Class

TRANSACTION STYLES

We use the term transaction to name a specific control and data interaction rule used for communication performed through ports. Closely associated with each port is a distinct transaction style, that is specified at the configuration description level. Besides the interaction rule, the style implicitly defines the implementation mechanisms required for the support of each particular transaction.

Our basic transaction set includes bi-directional synchronous (rpc-like) and unidirectional asynchronous (datagram-like) transaction styles. Informally, we say that transactions are performed through ports. This basic transaction set can be extended or modified in order to fulfill different requirements of parallel programs. The first version of P-RIO has been implemented using the standard Parallel Virtual Machine (PVM) library. Currently, we are considering an implementation based on the Message Passing Interface (MPI) library. Other transaction styles, as for example an highly optimized remote procedure call mechanism based on active messages, could be included in our environment and selected for specific port-to-port connections.

GROUP TRANSACTIONS

Group communication abstractions fit well into the configuration model. A group can be seen as an abstract module that receives inputs and distributes them according to a particular transaction style. In practice, we use a simplified representation for group configurations and the implementation of the associated transaction style will depend on the particular communication substrate available in the support environment.

Our current implementation supports two multicast-group transaction styles: unidirectional and bidirectional, for use with the asynchronous and synchronous
transactions, respectively. The reliability semantics of these transactions is that provided by PVM, that is, they are unreliable providing no guarantee of atomicity or ordering of message receiving. Reliable group transaction semantics could be available if an appropriate communication system, such as Horus, which provides this style of group support, is incorporated into the environment. It is noteworthy that the popular scatter-gather, all-gather and all-to-all collective data moves can be implemented using the standard features of P-RIO.

**SHARED MEMORY TRANSACTIONS**

In this section, we present one scheme that we adopted to introduce the distributed shared memory (DSM) paradigm in the context of P-RIO. This provides a hybrid environment, helping the programmer to mix the two paradigms in order to implement the best solution for a specific problem.

In figure 2(a), let us consider that module S can perform intensive processing of a data structure on behalf of one of the modules C[i]. The latter modules concurrently compete to access these processing services through procedure call transactions associated with ports visible at the S interface (for simplicity, only one port is shown in figure 2). Thus, as an implementation option, S and its encapsulated data structures can be replicated at each hardware node where there is a potential user of its services. Thus, the procedure code can be executed locally, providing a simple computation and data migration policy. A procedure that performs read-only data operations does not need to execute any synchronization code. However, if the procedure performs updates to data held in S, they must be implemented using a shared memory coherence protocol. Configuration-level analysis, similar to compile-level analysis used in Orca, could determine that module S should be replicated. This could also be done by the programmer, using knowledge regarding program behavior and structure.

![Figure 2. Distributed Shared Memory Transaction](image)

According to our programming model, each module instance acts as a monitor; a port defines a monitor entry point. Thus, each procedure call transaction can be associated with a mutual exclusion region and, in consequence, is equivalent to the acquire primitive used in DSM libraries, such as Munin or TreadMarks. In our model, a special code associated with each port can be used to trigger the required underlying coherence protocol, as represented in figure 2(b). All accesses to the module data structures would then be synchronized. Alternatively, the shared module data structures can be explicitly defined in order to optimise the synchronization cost. It is also possible to use explicit primitives, such as acquire and release, within the module code. This allows increased parallelism at the expense of program structuring. It is interesting to note that in our proposal several different data consistency techniques and mappings between modules
and processors can be specified at the configuration level, allowing flexibility for system tuning.

Currently, for evaluation purposes, we are implementing a shared-memory coherency algorithm using Unix sockets for communication. In the near future we intend to integrate special DSM support in our environment. One point we want to investigate is the interaction of the DSM and message-passing paradigms, in terms of programming and run-time support mechanisms.

**GRAPHICAL SYSTEM COMPOSITION**

P-RIO includes a graphical interface that helps to configure, visualize, monitor and debug parallel applications. The graphical representation of each class (icon), including their ports, and of a complete system is automatically created from its textual configuration description. Alternatively, using the graphical tool, a system can be created by selecting classes from a library, specifying instances from them and interconnecting their communication ports. Module encapsulation is supported, permitting the graphical composition of new module classes from already existing classes. Class compression and decompression features allow the designer to inspect the internals of composite classes and simplify the visual display of large programs. With this tool, it is also possible to control the physical allocation, instantiation, execution and removal of modules with simple mouse commands. For example, using these features, one can reconfigure the application on the fly by stopping modules, reallocating them, or changing other parameters. Program configuration changes at the graphical interface level are automatically reflected at the textual representation, which can be saved for reuse. The system does not attempt to optimize the representation of repetitive program structures. However, the user can obtain this effect by editing the automatically generated textual representation.

**COMMENTS**

The P-RIO environment allows configuration changes to be performed on an running system; this speeds up experimentation with different versions of algorithms. A simple process-migration policy based on redirection of messages is supported. The system does not embed support for dynamic load balancing or fault-tolerance policies. However, policies not requiring state preservation could easily be supported. It is possible to embed the configuration-control constructs in the code of the modules. This provides flexibility for the control of dynamic configurations.

P-RIO adopts the configuration paradigm first proposed in Conic [2], for constructing distributed systems, and extends the configuration framework with abstractions and mechanisms for parallel programming. Its key characteristic is the use of a configuration or architecture description language [3] that facilitates the association of specific features to modules and data interaction channels. In particular, diverse group and shared memory transaction styles can be selected and used to configure concurrent programs.

The use of a system description language has other attractions: it makes the software architecture amenable to formal verification techniques and also to program flow-analysis techniques; the text is a portable and compact representation of the software that can be mapped to graphical representations (and vice-versa); the use of encapsulation,
indexes and flow control constructs simplifies the specification of large static and
dynamic program structures.

Like other researchers, we believe that object-based program structures are best
dealt with using an architectural description language. According to this view,
configuration programming is complementary to pure object-oriented software
programming. The configuration paradigm offers composition (as an alternative to
inheritance) as a means of constructing a new object, or even a complete system, from
previously existing objects. The combination of both approaches, pure object-orientation
and configuration programming, can take advantage of the best of both worlds and can be
valuable for the development of complex applications.

CONCLUSIONS

P-RIO promotes a software architecture view for application development. It is
centered on the configuration paradigm that provides high-level and flexible abstraction
mechanisms for program construction. This strategy has allowed us to pick up and put
together several useful concepts (and associated support mechanisms) for parallel
computing, providing a "middleware" based programming and execution environment.
In particular, different group and shared memory transaction styles can be quickly
selected and used to configure parallel and distributed programs. As mentioned in the
text, several extensions in the available set of transaction styles can be incorporated into
the environment.

The current version exploits the portability and interoperability between hardware
architectures provided by PVM. Performance evaluation tests have shown that P-RIO
does not introduce measurable overheads to message communication, when compared to
a pure PVM system. However, we should mention that PVM itself introduces
significant overheads when compared with basic communication over LANs. More
efficient communication support (hardware and software) can be transparently introduced
into P-RIO, without requiring changes in the programming of already existing software
modules. This would allow P-RIO to evolve over time, incorporating new
communication technologies as they become available.

Another earlier version of our environment maps each computational node onto a
lightweight process (thread) and is intended for distributed computing. It supports group
communication protocols with different failure semantics and a set of fault-tolerance
techniques based on module replication. We also used the configuration approach to
customise distributed applications based on CORBA objects. This experience has helped
to convince us of the flexibility provided by the configuration approach.

The current version of P-RIO, including the graphical tool, has been operational
since November 1994 and has been used to build several experimental applications. Its
code, manuals and additional documentation can be obtained at

ACKNOWLEDGMENTS

This project was developed under the Brazilian High Performance Computing
Program financed by Finep (Studies and Projects Funding Agency). Grants from the
CNPq (Brazilian National Research Council) and Faperj (Rio de Janeiro State Research
Support Foundation) have also partially supported the research described here.
REFERENCES

