A Component Model for High Level and Efficient Parallel Programming on Distributed Architectures

January 3, 2005

ABSTRACT

The computer science community has claimed for parallel languages and models with a higher level of abstraction and modularity, without performance penalties, that could be used in conjunction with advanced software engineering techniques, and that are suitable to work with large-scale programs. This paper presents general aspects about the #1 parallel programming model and its associated programming environment, designed to address the above issues.

1 INTRODUCTION

Research initiatives through the last two decades made possible the use of distributed architectures for high performance computing (HPC), in alternative to vector systems. Today, HPC distributed architectures may be split into three classes: capability computing (MPP’s), cluster computing [7] and grid computing [16]. Essentially, they differ on their communication interface between processing nodes in decreasing coupling order. In consequence, they are used in different application contexts, but not necessarily disjoint [15].

High end distributed parallel architectures have brought new challenges for HPC users, regarding their programmability. They can support deep memory and parallelism source hierarchies. Grids, for example, may have clusters and MPP’s as processing nodes, which may be formed by multiprocessors (constellations). Individual processors may implement vector and super-scalar processing. Intel Pentium III, a typical example of a commodity super-scalar processor, implements SSE (Streaming SIMD Extensions) for supporting vector processing in multimedia applications.

Efficient high performance parallel programming is inherently a non-trivial task, mainly using low-level tools of today. Despite having to specify computations, like in sequential programming, programmers must partition the application functionality and/or data, according to the features of the target architecture, and implement process synchronization. There are no common models for programming the diversity of parallel architectures. Depending on the nesting level of the hierarchical design supported, where parallelism may be exploited at multiprocessor and distributed levels, a complicated combination of openMP and MPI, and possibly grid enabling tools, such as Globus Toolkit, should be employed.

The search for models and languages for programming high end distributed high performance architectures that reconcile requirements of generality (G), high level of abstraction (A), portability (P) and efficiency (E), allowing to apply advanced software engineering concepts into the development of HPC software, is still one of the most important challenges in parallel computing [15, 28]. We argue that evolution of parallel programming technology is still entering its third phase2. The first phase

1Hash.

2There is a curious analogy with the evolution of conventional (sequential) programming languages. In the first phase, the main concern was to achieve efficiency on the use of the scarce computational resources, by employing low-level machine codes or assembly languages. The second phase, marked by the advent of FORTRAN (1954), looks for portability, motivated by appearance of several computer architectures and vendors. The third phase searches for higher levels of abstraction and modularity. It begins with software crisis by the end of 1960’s [14]. In those years, several
was marked by the use of low level architecture-specific message passing interfaces. The frontier between the first and second phase is marked by the creation of CRPC (Center for Research on Parallel Computation), in 1989. From that milestone on research efforts started to be coordinated, culminating with the development of several efficient and portable tools, including libraries for message passing (MPI [22] and PVM [17]), multiprocessor programming (openMP [23]) and specific-purpose scientific computing (PETSc [3], ScalAPACK [5], and many others [15]). But in the recent years, the expansion in scale of HPC applications, due to cluster and grid computing, has increased the motivation of HPC community to search for parallel programming approaches that reconcile higher level of abstraction and modularity with generality, portability and efficiency (GAPE), allowing that advanced software engineering concepts to be applied into the development of HPC applications [28].

But to lift abstraction level and modularity of parallel programming languages is not trivial. In HPC, performance concerns are absolute and may not be neglected, since it is, by nature, the most important concern to be addressed. This assumption was not valid for conventional programming technology in its third phase, where performance concerns are not absolute, but relative to the status of hardware performance. Indeed, the rapid increasing in performance and capacity of the hardware allowed to make transparent the performance penalties imposed by higher level mechanisms.

This paper introduces the # parallel programming model and discusses aspects and trends related to the design of its integrated programming environment. It provides a new way to work with explicit message passing programming, where concerns about parallelism and computations are separate in different dimensions and programmed using orthogonal languages. The # parallel programming environment supports analysis of large scale parallel programs by using Petri nets, including “debugging” and simulation facilities, proof of formal properties, and performance evaluation. The idea behind the # environment is to provide a “glue” for integrating existing high performance computing programming technologies in a common component-based framework, where advanced software engineering techniques may be successfully applied.

This paper is organized as follows: Section 2 presents details on the # model. Section 3 discusses the features of an environment based on # model for design, development and analysis of large scale parallel programs. Section 4 provides some examples of # programming. Section 5 presents some performance figures of programs developed in Haskell#, a # implementation where sequential components are implemented in Haskell, a pure, non-strict, functional language. Section 6 concludes this paper and draws lines for further developments.

2 THE # MODEL FOR PARALLEL PROGRAMMING

The # parallel programming model takes lessons from the evolution of conventional programming and applies them in order to meet the demand of HPC community for higher level parallel programming models. It can be viewed as a structured way to work with message passing programming. It relates to conventional approaches for message passing like structured programming relates to unstructured programming [13].

In # programming, the concerns about specification of computations and parallelism configuration are divided in two orthogonal dimensions: computation (“the world of functions”) and coordination (“the world of processes”) media, using disjoint languages. From this perspective, # model may be viewed as a process oriented coordination model [18].

In conventional message-passing, processes are the programming building blocks. Programs are defined by a composition of processes, using low level message passing primitives as a glue. The # model moves upwards the level of abstraction of message passing programming. Now, components are the building blocks, and composition of components are used in order to build programs. While processes only address raw computations, components implement functional or non-functional concerns, possibly cross-cutting ones. The component view of a parallel program employed by # programming new large scale applications emerged specially in the commercial scenario. For this kind of applications, low level and unstructured programming techniques were shown to be inadequate. Structured programming [13] and modularization mechanisms [25, 20] emerged as alternatives for increasing the ability of programming languages for working with large scale programs.
Figure 1: Component Perspective versus Process Perspective

may be easily translated into a corresponding process view. This is opposite to the conventional programming approach, where the composition of processes implicitly specifies the composition of concerns of the application, presented in a non-modular way. The focus on component programming makes possible to apply advanced engineering techniques in the development of implementations of large-scale HPC applications.

Components may either be composed or simple. The set of composed components forms the coordination medium of a # program, while its set of simple components forms its computation medium. Composed components are programmed using the # configuration language (HCL), from the composition of other components. HCL differs from other composition and configuration languages because it supports overlapped composition of components, making possible to interlace the addressed concerns. Simple components, or functional modules, are implemented using a host language, supposed to be any general purpose sequential language bound to the # environment. Functional modules may implement only functional concerns, the atoms of functionality of # programs.

The # model of parallel programming supports the notion of skeletons [12]. Partial topological skeletons (PTS) may be used to expose topological patterns of interaction amongst processes and/or to make transparent use of parallelism strategies and load balancing algorithms. The # compiler may use skeletons in order to produce efficient code targeted at specific architectures and execution environments [10]. PTS’s are implemented as composed components parameterized by the concerns addressed.

Figure 2: Configuring a Unit
component SystolicMesh<N> with
use Skeletons.Common.PipeLine

Iterator i, j range [1,N]
[/ unit v_pipe[i]]; assign PipeLine<N> to v_pipeline[i] [/]
[/ unit h_pipe[j]]; assign PipeLine<N> to h_pipeline[j] [/]

interface IMesh t where
ports: <Pipe t> & top → bottom |
protocol: repeat seq {par {bottom;right} ;
par {top;left}}
until < top & bottom & right & left >
[/ unify v_pipe[i], h_pipeline[j] to mesh.cell[i][j] ]

where ports: IMesh [/]

Figure 3: HCL Code for the Systolic Mesh Skeleton

2.1 PROGRAMMING COMPOSED COMPONENTS

HCL describes a collection of units, defined as entities that carry specific tasks. Units are instantiated from interfaces that describe a collection of input and output ports and a protocol that specifies the order in which these ports must be activated. The activation of a port enables it to complete a communication. Interfaces may be declared separately allowing their reuse. Reusable interfaces are called interface classes. The interface of a unit describes how it interacts with the coordination medium. An interface may be composed from other interfaces. The interfaces that comprise a top-level interface are called their interface slices. The task carried by the unit is described by assigning a component to it. The concern addressed by the assigned component defines the role of the unit. The input and output ports of the unit should be (partially) mapped onto the arguments and return points of the component.

A unit may replicate a port. Groups of ports may be of two kinds: any or all. The activation of groups of ports of kind any is inspired by the semantics of the ALT constructor of OCCAM [19]. The activation of groups of ports of kind all enables all the ports belonging to the group. Wire functions may be specified in the boundary between ports, or groups of ports, and arguments/return points for forcing their compatibility.

Ports must be connected through point-to-point, unidirectional and typed communication channels, with three possible modes: synchronous, bound buffered and ready.

Units may be unified or factorized, using the slices of their interfaces as decomposition units. It is also possible to replicate sub-networks of a unit topology. Behavioral and topological preserving restrictions exist for ensuring the coherence of the network topology of a unit in the presence of these operations.

2.1.1 Virtual Units and Skeletons

Units for which no component was assigned are called virtual. They allow to parameterize the concern addressed by a component. A partial topological skeleton (PTS), or abstract component, is a composed component formed by at least one virtual unit. Due to this fact, it is possible to compose skeletons from other skeletons. In Figure 3, it is illustrated the composition of a systolic mesh of processes by means of pipe-line skeletons. The use of skeletons neatly allows parallel programming to take advantage of specific architectural features tuning code without losing the overall program portability.
2.2 Programming Simple Components

Functional modules are programmed in a host language. To bind a host language to the # programming environment, it is necessary to map the host language constructors to the arguments and return points of functional modules. No extension to the host language is needed for this purpose, keeping the orthogonality between coordination and computation medium. Our goal is to implement a really multi-lingual approach for parallel programming either, maybe on top of CCA (Common Component Architecture) [1], or interoperable implementations of MPI [29].

The way like functional modules are programmed depends on the kind of host language employed. In lazy functional languages, such as Haskell, arguments and return points correspond to the arguments and elements of a tuple returned by a function main in a module [9]. Lazy lists implement streams, allowing the overlapping between communication and computation [8].

In procedural languages, such as C and Fortran, functional modules are abstract data types. The encapsulated data structure represents the state of the process. The functions that access the data structure correspond to the return points of the component. The value produced by a function is transmitted through the port associated with the return point. There is a correspondence between the parameters of the return functions and the arguments of the component. The same argument may be referred by more than one parameter of a return function. Special functions, called guard checkers, are needed to implement choices in the alt constructor of the unit protocol. The use of the par combinator allows to execute more than one return function simultaneously. We intend to use this approach to implement multiprocessor parallelism by using openMP. It has been suggested to enrich the set of protocol combinators in order to support loop parallelization and other techniques implemented by openMP. The # programming environment may be able to exploit multiple levels of the hierarchy of parallelism [4].

In object oriented languages, such as JAVA and C++, functional modules are objects, in such a way that return points are implemented as methods and arguments as method parameters.

Procedural and object oriented functional modules may be overlapped without any need to join source codes. Using C and Fortran, for example, it is possible to use external clauses. Aspect programming [20] may be used in object oriented functional modules.

3 THE # PROGRAMMING ENVIRONMENT

An environment was designed for providing cleaner support for # programming. VHT (Visual # Tool) was prototyped using Eclipse and Java. Its most important features are emphasized in the following four items:

1. Visual programming: HCL has a corresponding visual language (HVL) that helps programmers on building # programs;

2. Libraries of reusable components and interface classes: The # programming environment provides a comprehensive way to store and retrieve reusable (composed and simple) com-
ponents, including the support for programming with skeletons. Components are stored in XML format and may be translated onto HCL or HVL according to the needs of the programmer. In Figure 3, it is exemplified the topology of a systolic implementation of matrix multiplication built from composition of skeletons FARM and TORUS;

3. **Simulation, formal properties analysis and performance evaluation using Petri nets:** A # program may be translated onto a Petri net that models the interaction of processes on its coordination medium [11]. Using Petri net tools like INA [27] and TimeNET [31], it is possible to simulate, prove and analyze formal properties and to perform performance evaluation of # programs. The Figure 5 exemplifies the Petri nets generated by two implementations of the Dining Philosophers problem. The generated Petri net may require large computational resources (time and space) for precise formal analysis. The next item provides an alternative to overcome these difficulties. A further topic to be addressed is to design a higher level environment for formal analysis of # programs on top of INA and TimeNET, treated as transparent engines in analyzing computations. At present, the raw Petri net is generated and submitted to Petri net tools. For relative simple programs, this low level approach results in hard work;

4. **Programming as a collaborative task on the grid infra-structure:** The concept of virtual organizations (VO’s) in grids could make possible a collaborative environment where # programmers could share reusable components and interface classes (item 2) and perform complex computations, whenever analyzing large scale # programs using Petri net tools (item 3), using the grid resources. Other forms of collaboration could be suggested.

4 **EXAMPLE: CLIMATE SYSTEM MODEL (CSM)**

The Climate System Model (CSM) program has been originally developed at NCAR (National Center for Atmosphere Research), in USA, using Fortran/MPI [6]. In CSM, four processes implement climate models for atmosphere, biosphere, hydrosphere, and cryosphere. They interact by exchanging messages
configuration CSM<\text{cpl},\text{peer[1]}> with

-- single components below were originally implemented in FORTRAN
use Applications.CSM_Functional.Modules.(\text{cpl}, \text{atm}, \text{ico}, \text{ocn}, \text{w})

interface IModel where
  ports: \{ir, r\} → \{is, s\}
protocol: seq \{as, ir\}, repeat seq \{is, s\} until \(<r k s>\}

interface ICoupler where
  ports: IModel ∘ (ir, ir) → \{is, is\} ∘ IModel → \{is, is\}
protocol: seq \{as, ir\}, repeat seq \{is, s\} until \(<r k s>\}

unify \{is, s\} \rightarrow \{as, ir\}

IOWriteOnly

IModel

Ice

Atm

Ocn

Lnd

Cpl

CSM_Debugger

peer[1]

with an the coupler process. The # configuration code of CSM component is shown in Figure 6. Its topology appears in Figure 7(d).

In Figure 8(a), it is presented the # configuration of a component that implements a debugger for CSM, named CSM_Debugger. It is formed from the composition of two other components: IOWriteOnly and BCast. The latter implements output of programming traces in screen terminals, while the former allows the coupler to broadcast information about its state to climate models. The simple components assigned to the climate model units perform verifications on the local state of the climate models and the coupler, contained in their data structures, in order to check state consistency.

CSM_Debugger is overlapped with CSM in order to build a new component, named CSM_With_Debugger. Let * be one of the climate model unigs or the coupler. The unification of units \text{funct.}* with \text{debug.}* imposes the unification of the simple components assigned to them in its respective components. Assuming that the simple components are written in standard FORTRAN, COMMON declarations may be used to allow them to share the data structures that implement climate model state. It is not necessary to join their source codes, which are kept separate.

It is important to attempt modularizing the approach presented. Global debugging in a parallel program may be viewed as a cross-cutting concern, whose implementation is scattered in the implementation of each individual process. In MPMD programs, like CSM, this may be problematic, imposing a high level of complexity to the debugging task. The # model allows to encapsulate parallel
programming cross-cutting concerns in components. Several other examples of cross-cutting concerns can be used to illustrate this feature, such as placement of processes onto processors, collective I/O operations on disk devices, etc. With respect to collective I/O, we are working on defining components for encapsulating MPI/I/O functionalities.

Another point worth noticing is the possibility of using multi-processor technologies for allowing climate models to execute in parallel.

## 5 HASKELL\#: IMPLEMENTING THE # MODEL

In the current implementation of the # programming model, named Haskell\#, functional modules are assumed to be written in Haskell, a non-strict functional language [30]. Non-strict functional languages have advantageous features, in relation to languages from other paradigms, for making transparent the interface “glue” between computation and coordination media, due to the possibility of implementing streams as lazy lists. It also makes possible to use tools for proving formal properties about computation medium.

The Haskell\# compiler is entirely implemented in Haskell. It generates Haskell code that accesses C/mpi functionalities through a wrapper interface library, specially designed for Haskell\#. It treats issues such as marshalling of Haskell data structures onto contiguous buffers in memory at C side. Functional modules are compiled using GHC (Glasgow Haskell Compiler), a state-of-the-art compiler for Haskell [26]. There is no need to modify GHC compiler to use it in conjunction with the HCL compiler, due to the orthogonality between coordination and computation media. This is an advantage of Haskell\# in relation to other parallel functional languages. In general, they require to modify the underlying Haskell compiler. In some cases, such as Eden, it is even necessary to disable some of the optimizations of the compiler [24].

Figure 10 presents some benchmarks in Haskell\# executed on a cluster comprising 16 dual nodes (Intel Xeon, 1.2GHz, 1GB RAM). The performance figures on the left are for Haskell\# versions of NAS Parallel Benchmarks (NPB) kernels (EP, IS, and CG) [2], originally written in C, while the performance figures presented on the right are Haskell\# versions of three applications recently used for comparing performance of implementations of parallel functional languages Eden, GpH, and PMLS [21]: LinSolv (LS), Matrix Multiplication (MM) and Ray Tracer (RT). Haskell\# versions offered similar or better
Figure 9: Performance Figures of Haskell
Figure 10: Performance Figures of Haskell
results than other parallel functional implementations. The profiling of these benchmarks allows to conclude that the performance penalties paid by the # model to obtain a higher level of abstraction are not relevant.

6 CONCLUSIONS AND LINES FOR FURTHER WORKS

This paper presents the # parallel programming model and its associated environment, proposed as an efficient, portable and high level alternative for general purpose message passing programming. It meets the modern software engineering concepts, as it moves parallel programming from a process-based perspective to a component-based outlook. It supports skeleton-based programming, modularization of cross-cutting functional and non-functional concerns, and integration with current specific purpose libraries for scientific computing, by encapsulating their functionalities into components.

The development of the # programming environment is on progress. The implementation of the # configuration language and its Petri net compiler have just been finished. Now, we are concentrating efforts on: (i) prototyping a new version of the # programming environment (VHT) on top of Eclipse/Java, supporting the multi-lingual approach and using the concept of grid collaborative environments; (ii) defining an interface for scientific specific purpose libraries (PETSc and ScaLAPACK) as components in the # environment; (iii) providing a higher level approach for analysis and performance evaluation of # programs, on top of existing Petri nets tools like INA, PEP and TimeNET; (iv) specifying a categorical semantics for the # parallel programming model; and (v) defining a cost model for # skeletons, with support to overlapping skeletons.

References
