

The # Model for Parallel Programming: From Processes To Components with Insignificant Performance Overheads

Francisco Heron de Carvalho Junior¹, Rafael Dueire Lins²

¹Departamento de Computação – Universidade Federal do Ceará
Campus do Pici, Bloco 910 – Fortaleza, Brazil

²Departamento de Eletrônica e Sistemas – Universidade Federal de Pernambuco
Av. Acadêmico Hélio Ramos s/n – Recife, Brazil

heron@lia.ufc.br, rdl@ufpe.br

Abstract. *The computer science community has claimed for parallel programming 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 deal with large-scale programs. This paper presents the # component model for parallel programming, intended to meet these issues.*

1. Introduction

The last two decades of research initiatives made possible the use of distributed architectures for high performance computing (HPC). Parallel systems are now classified as: *capability computing*, the high-end computing architectures; *cluster computing* [4], parallel computers in a local area network built from commodity components; and *grid computing* [9], a computation infra-structure designed on top of a wide area network such as the *Internet*. High-end computing architectures support nested parallelism levels and several memory hierarchies, making impossible, even for computer scientists, to exploit all their potential performance using general programming models. On the other hand, clusters and grids have created new application niches for high performance computing, with increasing levels of complexity and scale. In consequence, there is no consensual model for high-performance computing that may be able to reconcile *efficiency* and *portability* with *abstraction* and *generality* in all application contexts [3]. Low level message passing libraries, such as MPI and PVM, offer poor abstraction and high generality. They are not able to deal with large scale HPC applications. Scientific computing libraries provide high abstraction, but they offer restricted applicability. Besides that, it is difficult to integrate them in multidisciplinary simulation environments on grids. Parallel programming technologies that attempt to hide details of parallelism from programmers, such as in parallel implementations of functional languages, have failed to ensure efficiency in the general case, due to their requirements of a heavy run-time system. Comparing evolution steps of conventional programming technologies with the ones of high-performance programming technologies, one may conclude that the latter is entering in its third phase, where programming artifacts show to be inappropriate to deal with complexity of the development of emerging large scale applications. In conventional programming, this phase corresponds to the “software crisis” from the late 1960s. The # programming model, introduced in this paper, tries to learn from the evolution of conventional programming and to apply those lessons to answer the claims of high performance community.

2. The # Component Model

The # programming model promotes a change of axis in the practice of programming for high performance computing, by moving parallel programming from a *process-based perspective* to an orthogonal *concern-oriented perspective*. From the process-based perspective, a program is viewed as a collection of processes that interact through communication primitives. All previous attempts described in the literature to lift the level of abstraction of process-based parallel

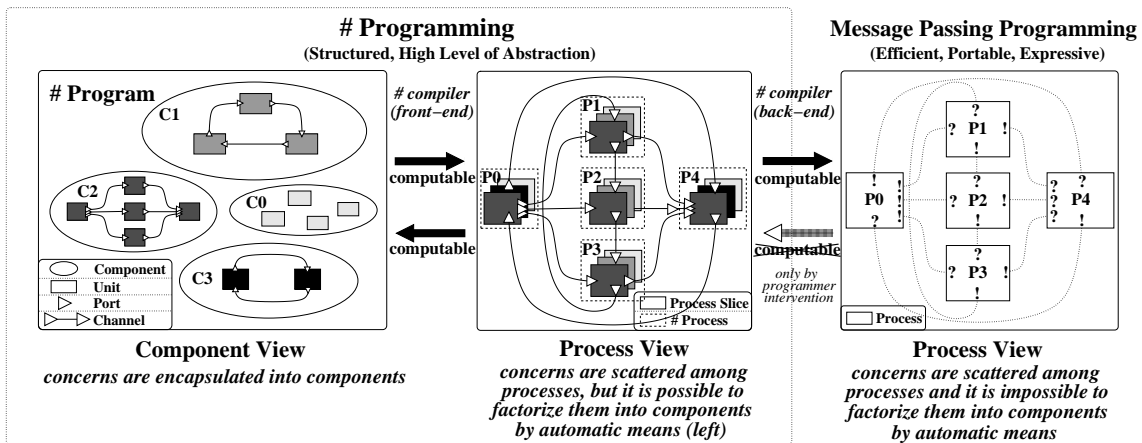


Figure 1: Components versus Processes

programming resulted in efficiency losses. Also, *concerns* [11] are found scattered along implementation of processes, since concerns are orthogonal to processes. In fact, a process may be viewed as a collection of *slices*, each one describing the role of the process with respect to a given concern. In this context, concerns are the *decomposition criterion* for *slicing* processes [15]. Thus, they may be viewed as collections of related slices, probably from distinct processes. Functional and non-functional concerns in the # model are addressed by *components*. Concerns are one of the fundamental concepts in modern software engineering, thus it is reasonable to suppose that a concern-oriented perspective of parallel programming fits contemporary advanced software engineering artifacts better than a process-based outlook.

In # programming, the slices that comprise a component are called *units*. They are integrated in a communication topology, formed by one-direction, point-to-point, and typed channels. A unit has a set of input and output ports, whose activation order is dictated by a protocol, specified using a formalism equivalent to labeled Petri nets. In # programming, parallelism and computation concerns are separated into composed and simple components, respectively. Composed components comprise the *coordination medium* of # programs. They are specified in terms of units and channels, possibly by composition of existing components, by using some language that supports the coordination level abstractions of the # model. The # programming environment encompasses two languages: HCL (a textual configuration language) and HVL (a graphical configuration language). Simple components are specified using Turing-computable languages, comprising the *computation medium* of # programs. They are the atoms of functionality in # programs. Simple components may be assigned onto units of composed components in order to configure computations performed by slices. Skeletons [7] are supported by allowing units with no component assigned, called *virtual units*, allowing a high level of abstraction without losses either in efficiency or portability. Nested composition of components is possible by allowing to assign composed components to units of other composed components. Besides providing support to non-functional concerns and skeletons, another important distinguishing feature of the # component model in relation to other component models [1, 2] is its ability to combine components by overlapping them. It is possible to unify units from different composed components. Component models of today allow only nesting composition. Components are black-boxes addressing functional concerns. Whenever supported, non-functional concerns are introduced by means of orthogonal language extensions or by using tangling code cross-cutting component modules, like in sequential programming. However, cross-cutting concerns are not exceptions in parallel programming. The ability to overlap components, provided by the # model, allows to treat cross-cutting concerns as first-class citizens in the canonical parallel decomposition of applications.

The # programming model has a strong formal basis built on top of Petri nets [12] and Category Theory [17]. The translation schemes of # programs onto Petri nets, focused on the concerns at the coordination level, have been proposed in earlier works [6]. Its categorical semantics was recently specified, but has not been published yet. Category theory allows to formalize the

compositional features of # programming, while Petri nets model the dynamic aspects in process interaction. Further research plans to investigate the use of category theory in comparing the expressiveness of the # model with other component models, concerning compositional features. An ongoing research topic is investigating the use of stochastic Petri nets for evaluating performance of # programs. For the same purpose, it has been proposed the use of network simulators, such as NS [8]. For that, it is necessary to generate NS simulations from configurations of # programs. NS allows to vary network parameters and to change protocols intending to perform sensibility analysis of performance metrics with respect to architectural features and to analyze the granularity of programs. For such purposes, the probabilistic behavior of synchronization and computation times of parallel programs have been studied.

The work with the # programming model has its origins in Haskell_# [5], a parallel extension to the functional language Haskell [14]. Today, Haskell_# is viewed as a # model *binding* where simple components are *functional modules* written in Haskell. Non-strict functional languages have nice properties for prototyping simple components. Besides that there is its well-known gain in programming practice, including the absence of side-effects and referential transparency. These features make possible to prove formal properties about the computation medium of # programs. It also makes possible a neat separation between coordination and computation media, without any need for intermediate languages or tangling combinators. In fact, simple components are pure Haskell modules. Recently, it has been designed a binding of the # model to *procedural* or *object-oriented* languages, where simple components are *imperative modules*. However, concepts from aspect-oriented programming [10] have been essential for gluing imperative modules at computation medium to coordination medium. Benchmarks were performed for comparing the performance of # versions of some kernels of the NAS Parallel Benchmarks [16] with the performance of the original C/Fortran MPI versions. Minimal differences have been observed in the measures. Functional and imperative modules could co-exist in the same application, but a multilingual version of the # programming environment have not been designed yet.

3. The # Programming Environment

The concept of *framework* occupies a central position in the design of an environment for # programming. *Frameworks* are defined as collections of components and rules that guide how to compose them in order to build applications. *Plug-in's* may be added to the basic compilation system of the # environment for driving the compiler to generate code for a # program that use a *framework* according to its rules.

The # component model supposes that any current and further parallel programming technology sits on top of low-level message-passing. Other works have investigated this assumption in the past [13]. On top of low-level message-passing abstractions, which have an efficient implementation on top of message-passing libraries, the # model provides higher-level combinators that make possible to design new components by abstracting from low-level details of message-passing. For this purpose, skeletons play a central role. By supposing that the # model may model any other model for parallel programming, the # programming environment intends to be a fertile substratum for integrating existing parallel programming technologies by using frameworks. At present, frameworks for scientific computing libraries, such as PETSc and ScaLAPACK, collective communications in message-passing programming, and service-oriented parallel programming have been designed. The latter has shown to have a high degree of applicability in grid programming. For instance, it has been planned to design bindings of CCA frameworks to the # programming environment. Concerning integration ideas, the most important challenge is to integrate components from different frameworks. It has been designed a distributed library of components, where programmers may share reusable components and perform complex analysis.

4. Conclusions

This paper provides an introduction to the # component model, focusing on its main features and current challenges concerning its implementation. The features of the # programming model are

discussed and compared with existing component models. It is advocated that the # model is a promising alternative for component-based parallel programming, intending to meet the HPC community claims for parallel programming models and languages that reconcile *generality*, *abstraction*, *portability*, and *efficiency*. The work in the implementation of the environment for supporting # programming is still in its infancy. A fully functional version of the # programming environment should be released in the near future. This version should support the visual composition of components, the management and sharing of distributed component libraries across grids and the integration with tools for the analysis of Petri nets and network simulators.

References

- [1] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Towards a Common Component Architecture for High-Performance Scientific Computing. In *The Eighth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society, 1999.
- [2] F. Baude, D. Caromel, and M. Morel. From Distributed Objects to Hierarchical Grid Components. In *International Symposium on Distributed Objects and Applications*. Springer-Verlag, 2003.
- [3] Bernholdt D. E. Raising Level of Programming Abstraction in Scalable Programming Models. In *IEEE International Conference on High Performance Computer Architecture (HPCA), Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, pages 76–84. Madrid, Spain, 2004.
- [4] R. Buyya (ed.). *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall, 1999.
- [5] F. H. Carvalho Junior and R. D. Lins. Haskell_#: Parallel Programming Made Simple and Efficient. *Journal of Universal Computer Science*, 9(8):776–794, August 2003.
- [6] F. H. Carvalho Junior, R. D. Lins, and R. M. F. Lima. Translating Haskell_# Programs into Petri Nets. *Lecture Notes in Computer Science (VECPAR'2002)*, 2565:635–649, 2002.
- [7] M. Cole. *Algorithm Skeletons: Structured Management of Paralell Computation*. Pitman, 1989.
- [8] K. Fall and K. Varadhan. The NS Manual (formerly NS Notes and Documentation). Technical report, The VINT Project, A Collaboration between researchers at UC Berkeley, LBL, USC/ISI, and Xerox PARC, April 2002.
- [9] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. M. Kauffman, 2004.
- [10] G. Kiczales, J. Lamping, Menhdhekar A., Maeda C., C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Lecture Notes in Computer Science (Object-Oriented Programming 11th European Conference – ECOOP '97)*, pages 220–242. Springer-Verlag, November 1997.
- [11] H. Milli, A. Elkharraz, and H. Mcheick. Understanding Separation of Concerns. In *Workshop on Early Aspects - Aspect Oriented Software Development (AOSD'04)*, pages 411–428, March 2004.
- [12] T. Murata. Petri Nets: Properties Analysis and Applications. *Proceedings of IEEE*, 77(4):541–580, April 1989.
- [13] D. B. Skillicorn. A Cost-Calculus for Parallel Functional Programming. *Journal of Paralell and Distributed Computing*, 28(1):65–83, April 1995.
- [14] Simon Thompson. Formulating Haskell. Technical Report 29-92*, University of Kent, Computing Laboratory, University of Kent, Canterbury, UK, November 1992.
- [15] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.
- [16] R. F. Van Der Wijngaart. The NAS Parallel Benchmarks 2.4. Technical Report NAS-02-007, NASA Ames Research Center, October 2002.
- [17] S. Wells. Innovative Computing Laboratory Annual Report. Technical report, University of Tennessee, 2002. <http://icl.cs.utk.edu/files/2002Report.pdf>.