A Type System for Contextual Dynamic Binding of Parallel Components in High Performance Computing Platforms

Francisco Heron de Carvalho-Junior
Mestrado e Doutorado em Ciência da Computação
Universidade Federal do Ceará
Fortaleza, Brazil
Email: heron@lia.ufc.br

Rafael Dueire Lins
Departamento de Eletrônica e Sistemas
Universidade Federal de Pernambuco
Recife, Brazil
Email: rdl@ufpe.br

Resumo—The Hash (#) component model attempts to improve the practice of parallel programming. This paper introduces a type system for #-programming systems whose general objective is to increase the abstraction and safety of programming with components for parallel computing platforms. In particular, it provides an approach for safe dynamic discovering, fetching and binding of parallel components based on statically specified execution contexts.

I. INTRODUCTION

The dissemination of cost-effective and internet-based parallel computing platforms have led to new possibilities regarding cooperative work in multidisciplinary environments aimed to solve important challenges from sciences and engineering, both in academia and industry. As a consequence, new classes of large scale applications have emerged in these fields, which pose new requirements to development platforms, regarding productivity, complexity, and performance [1].

Parallel programming is the key to achieve high performance computing (HPC). Unfortunately, parallel programming is still hard to be incorporated into usual software development platforms [2]. The situation is worse in the level of programming-in-the-large. Unlike programming-in-the-small, the experiences with parallel programming tools and techniques that prioritize programming-in-the-large requirements, where software engineering/architecture techniques are applied with stronger emphasis, are rare. This is because, historically, software from sciences and engineering is structurally simple, compared to corporative software, being mostly programmed using Fortran and C by groups of one to three programmers at research institutions.

In the last years, component models and frameworks for HPC applications have been proposed [3], such as CCA and its compliant frameworks [4], Fractal/ProActive [5], and GCM [6]. However, the HPC community still looks for a general notion of parallel component and better connectors for efficient parallel synchronization.

The # component model was proposed to meet the aims of parallel software in HPC domain. It provides (#)-components with the ability to be deployed in a pool of computing nodes of a parallel execution platform and to address non-functional concerns. Based on a framework architecture recently proposed [7], a #-programming system based on the notion of #-components was designed and prototyped, called HPE (The Hash Programming Environment). This paper presents HTS (Hash Type System), a type system for #-programming systems that support a suitable notion of abstract component for enabling safe dynamic discovering, fetching and binding of parallel components based on statically specified execution contexts. Moreover, issues about the implementation of HTS in HPE are discussed.

Section II introduces the # component model and HPE. Section III presents a motivating example for providing intuitions about HTS. Section IV presents the formal specification of HTS, whose implementation in HPE is discussed in Section V. Section VI concludes this paper, outlining ongoing and further works.

II. HASH: A MODEL OF PARALLEL COMPONENTS

Models of parallel components have been proposed for computational frameworks aimed at developing HPC applications [3]. In general, they lack the level of expressiveness and efficiency of message passing libraries such as MPI [8]. For this reason, the search for more expressive ways to express parallelism with components is still an important concern of people that work with platforms of components that comply to CCA, Fractal, and GCM.

The # component model proposes a notion of components that are intrinsical parallel and how they can be combined to form new components and applications. It has origins in the coordination model of Haskell #, a parallel extension to Haskell [9]. It moves from the usual style of parallel programming focused on processes as units of software composition to a style focused on concerns [10], which is closer to the modern software engineering practices [11].

Unlike in usual component models, #-components may be deployed in a set of nodes of a parallel computing platform. For that, a #-component is composed by a set of units, each one placed on a node. A unit defines the role of a process in the
concern addressed by the #-component. In usual component models, a component is like a #-component with only one unit, since it must execute in a single address space.

A set of #-components, called inner components, may be combined to form a new #-component by overlapping composition, a kind of hierarchical composition where a unit of the new #-component is formed by importing units from its inner components. If a unit u, of an inner component, is imported into a unit v, of the new #-component, v is a slice of u. Each unit of an inner component must be a slice of some unit of the new #-component. Figure 1 illustrates the overlapping composition of inner components B, C, and D, to form a #-component A. #-components are represented by ellipses and their units by rectangles. E and F are direct inner components of B, as well as transitive inner components of A. The inner component E is shared between inner components B and C. The possibility of resource sharing, for resources represented as #-components, is an important feature for ensuring better performance in component interactions.

A. #-programming systems

A #-programming system is a component-oriented programming environment that complies to the # component model. For that, it must support distributed deployment and execution of a set of component parts, called units; hierarchical composition by overlapping; and a finite set of component kinds.

Component kinds group #-components that have similar component, connection and deployment models [12]. Composition restrictions may be imposed to components according to their kinds. In practice, the component kinds of a #-programming system will specify how #-components are defined in terms of usual software parts. Such flexibility makes possible the coexistence of #-components that address functional and non-functional concerns, as well as the interoperability between components of different models. In fact, kinds place the # component model at a higher level of abstraction when compared to other component models, like Fractal, CCA, and GCM, which define only one form of component kind and a fixed pre-defined set of connectors, making more difficult to deal with the heterogeneity of parallel computing platforms. Moreover, they often deal with non-functional concerns by introducing new abstractions orthogonal to components, such as membranes of Fractal, bindings and cohorts of CCA, collective ports of GCM, etc. A #-programming system may incorporate such new abstractions by adding support for new component kinds, without relying on extensions to the basic component model. For example, and also due to the distributed nature of #-components, any kind of component connector may be implemented using #-components [13].

The ability of programmers to develop libraries of connectors tuned for specific parallel computing platforms, taking advantage of abstraction mechanisms of high level programming languages, like polymorphism and abstract data types, is an important feature of #-programming systems, making possible to support skeletal programming [14], a promising parallel programming technique, as exemplified in Section III.

B. HPE: A Platform of Parallel Components for Clusters

The Hash Programming Environment (HPE) is a general purpose #-programming system targeted at clusters, whose design is based on an architecture aimed at developing frameworks from which #-programming systems targeted at specific application domains may be instantiated [7]. For that, it now supports nine component kinds: computations, data structures, synchronizers, enumerators, architectures, environments, applications, services, and qualifiers.

The HPE platform is composed by three services: the FRONT-END, where programmers build configurations of #-components and control their life cycle; the CORE, which manages a library of #-components distributed across a set of locations; and the BACK-END, which manages the components infrastructure where #-components are deployed and the execution platforms where they execute.

The interface between these services have been implemented as Web Services for promoting their independence regarding localization and development platforms. Thus, the FRONT-END can connect, as a client, to any CORE and/or BACK-END of interest. The BACK-END of HPE has been implemented by extending the CLI/Mono platform, while the FRONT-END and the CORE has been implemented in Java.

HPE introduces the concept of abstract component, representing sets of #-components that implement a given concern by supposing different contexts. Contexts are defined by the so called formal context parameters of an abstract component, each one defined by a context variable and a context bound. A context bound is an abstract component that defines a subtyping restriction on the abstract component, i.e. actual context parameter, that will replace its corresponding context variable in a given instantiation of abstract component. Therefore, instantiations are contextualized abstract components, those whose context parameters are supplied, for which #-components must be assigned in the environment of deployed #-component managed by the BACK-END. The example in Section III will elucidate intuitively the meaning and using of abstract components, while Section IV will formalize them as polymorphic types of #-components.

HPE is an open-source project hosted at http://hash-programming-environment.googlecode.com. It deserves special

Figura 1. Overlapping Composition
MPI environment of the Front-End of HPE.

implemented and validated.

input abstract components. For instance, the inner component possible to make direct references to context parameters of context bound configuration and C is its context signature.

Algorithmic skeletons have been widely investigated as a technique for increasing abstraction in parallel programming [14]. The farm skeleton captures the most used way to exploit parallelism in a distributed platform, having processes of two kinds: one manager, which distribute jobs among workers and combine their results, and many workers, which compute the results of lists of jobs. Well-known parallel programming techniques for grid computing, like bag-of-tasks techniques for grid computing, like pReduce framework [15], are variants of farm.

Figura 2. Configuration of the Farm Skeleton

attention in this paper, because it is actually the main test bed environment for # programming, where HTS has been implemented and validated.

III. MOTIVATING EXAMPLE: MULTIDIMENSIONAL INTEGRATION USING SKELETONS

Algorithmic skeletons have been widely investigated as a technique for increasing abstraction in parallel programming [14]. The farm skeleton captures the most used way to exploit parallelism in a distributed platform, having processes of two kinds: one manager, which distribute jobs among workers and combine their results, and many workers, which compute the results of lists of jobs. Well-known parallel programming techniques for grid computing, like pReduce framework [15], are variants of farm.

The inner components are annotated with their types. Except for mpi, simply typed by MPI, all the other inner components are typed by X : C, where X is a context variable of the configuration and C is its context bound. As the reader can see in the context signature of Farm, all context parameters of a configuration are also associated to a name, making possible to make direct references to context parameters of abstract components. For instance, the inner component input, representing input data structure to be partitioned in jobs is typed by the context variable I, defined by the context parameter named input type, whose context bound is DATA. Analogously, the inner components job[I], where I is the enumerator of workers, representing the jobs, are typed by J, defined by the context parameter job type, also bounded by DATA. The inner component scatter, responsible to partition input in jobs, and send them to the workers, is typed by the context variable S, defined by the context parameter scatter strategy, bounded by DISTRIBUTE with their context parameters data source and data target supplied with the context variables I and J, respectively. The configuration of the abstract component DISTRIBUTE is presented in Figure 3.

By supplying S with an abstract component that specializes DISTRIBUTE (actual context parameter), one may build a #-component implementing FARM specifically designed for a context defined by some specific strategy of partition and distribution of jobs to the workers. Analogously, the FARM may be specifically tuned to work with a specific strategy to collect results (inner component gather, typed by G) and to process the jobs (inner components work, typed by W). Finally, it is also possible to specialize FARM for the data structures that define the data to be partitioned by the manager (context variable J), data structure representing a job (J), data structure representing a result of job processing (R), and data structure representing the final result (O).

The units of FARM are manager and worker[I], for each element of the enumerator I, denoting the manager and worker process slices of an application that uses a farm. The configuration elements IManager and IWorker denote interfaces of units. They specify the common features of a set of units with the same types of slices, including slice names and the protocol. In the case of FARM, only one unit is attached to each interface. In the figure, the slices are annotated with ⟨slice_name⟩ : ⟨inner_name⟩.(unit_name), where ⟨slice_name⟩ denote their identifiers and ⟨inner_name⟩.(unit_name) denote the unit from which they have origin.

Figure 5 presents the configuration of a #-component implementing the FARM, called FarmImpl. By inspecting their supplied context parameters, one may notice that it may work

scatter strategy, bounded by DISTRIBUTE with their context parameters data source and data target supplied with the context variables I and J, respectively. The configuration of the abstract component DISTRIBUTE is presented in Figure 3.

By supplying S with an abstract component that specializes DISTRIBUTE (actual context parameter), one may build a #-component implementing FARM specifically designed for a context defined by some specific strategy of partition and distribution of jobs to the workers. Analogously, the FARM may be specifically tuned to work with a specific strategy to collect results (inner component gather, typed by G) and to process the jobs (inner components work, typed by W). Finally, it is also possible to specialize FARM for the data structures that define the data to be partitioned by the manager (context variable J), data structure representing a job (J), data structure representing a result of job processing (R), and data structure representing the final result (O).

The units of FARM are manager and worker[I], for each element of the enumerator I, denoting the manager and worker process slices of an application that uses a farm. The configuration elements IManager and IWorker denote interfaces of units. They specify the common features of a set of units with the same types of slices, including slice names and the protocol. In the case of FARM, only one unit is attached to each interface. In the figure, the slices are annotated with ⟨slice_name⟩ : ⟨inner_name⟩.(unit_name), where ⟨slice_name⟩ denote their identifiers and ⟨inner_name⟩.(unit_name) denote the unit from which they have origin.

Figure 5 presents the configuration of a #-component implementing the FARM, called FarmImpl. By inspecting their supplied context parameters, one may notice that it may work

scatter strategy, bounded by DISTRIBUTE with their context parameters data source and data target supplied with the context variables I and J, respectively. The configuration of the abstract component DISTRIBUTE is presented in Figure 3.
with any implementation of message passing library, kind of data structures, gather/scatter strategy and work. To test the farm, an application was developed, implementing a approach to parallelize the calculation of the integral of a function over a specified interval using the Romberg’s method.

The configuration of the abstract component RombergIntegrator is depicted in Figure 4. The root process partitions the interval in n subintervals, representing jobs. Then, a list of jobs are sent to each peer process, which apply the Romberg’s integration procedure over the subintervals, sending the results back to the root, where they are added. It abstracts from the integration procedure over the subintervals, sending the results back to the root, where they are added. It abstracts from the type of the function by defining a context parameter function, bounded by FUNCTION, that must be supplied by their #-components, such as RombergIntegratorImpl in Figure 5.

The configuration of RombergIntegrator involves the following specialized abstract components, with their respective #-components:

- INTEGRALCase, for supplying the context parameters input type and job type, representing the type of jobs, carrying an interval and an integrating function.
- APPROXIMATEINTEGRAL, for supplying W, representing the approximation of the integral of a function over a given interval using the Romberg’s method;
- DISTRIBUTEINTERVAL, for supplying S, representing the strategy to partition the integration interval and distribute the partitions to the workers;
- SUMINTEGRALS, for supplying G, representing the strategy to collect and sum the partial integrals calculated by each worker.

SUMINTEGRALS reuses REDUCE[operator = SUM] from a library of #-components for collective communication. DATA, DOUBLE, and FUNCTION are also retrieved from the library. All the source codes related to this adaptative quadrature example are available at http://hash-programming-environment.googlecode.com/svn/trunk/Exam-

IV. HTS: A TYPE SYSTEM OF #-COMPONENTS

In this section, the Hash Type System (HTS) will be defined over a calculus of overlapping composition of #-components called HOCC (Hash Overlapping Composition Calculus). The designs of HTS and HOCC started from the following premises: (1) to focus on the common aspects of #-programming systems which define the # component model, abstracting away from their specific features; (2) to separate composition, coordination and computational concerns of #-components; (3) to abstract away from computational concerns, which is a concern of #-programming systems, by concentration on composition and coordination ones; (4) to be built on top of standard formalisms, in order to inherit their known properties, as well as adopting standard notation as much as possible, to facilitate derivation of results.

Aiming at attending the first three premises, a two-layer approach is adopted. In the top layer, there are terms and types of a calculus whose semantics define how #-components are combined. In the bottom layer, it is assumed the existence of a set of process slices S, a set of types of process slices Σ, a typing relation type_of : S × Σ that associates process slices with their types, and a subtype relation subtype_of : Σ × Σ. Process slices are the building blocks of #-components, since they define their units. A set of process slices may define the set of units of a #-component if together they define the implementation of an application concern in a parallel program. The quadruple (S, Σ, type_of, subtype_of) is a characteristic of the #-programming system.

Finally, for attending the fourth premise, the λ-calculus and some of their extensions has been chosen, since it is widely applied to study the properties of type systems.

The syntax of terms and types of HOCC that define overlapping composition are presented in Figure 6. Figure 9 adds the terms and types related to the concepts of contexts and fetching, introduced in Section IV-B. The semantics of HOCC terms, specified by the reduction rules of Figure 7, defines the semantics of overlapping composition, showing how #-components are combined hierarchically. Typing and subtyping rules of HOCC are presented in Figures 8 and 10, respectively, defining HTS.

The metavariable t denotes terms, while the metavariable T denotes their types. The terms and types of HOCC are
closely related to the notions of \#-components and abstract components of HPE, whose meaning were intuitively introduced in Section III. A term may be viewed as a configuration, whose evaluation yields a \#-component. Abstract components are types of configurations.

A. The Basic Terms and Overlapping Composition

A term \#-component is defined as a list of units \([l_i: u_i \overset{i=1\ldots n}\rightarrow]\), where the label \(l_i\) represents the name of the unit \(u_i\). A unit may be either a qualified variable \((x:l)\), a process slice \((s)\), or a composition of units using a folding operator \(\otimes\). In a qualified variable, \(l\) is a unit name. A \#-component term is a value if all their units are process slices.

The terms variable, abstraction, and application borrow their syntax and semantics from the \(\lambda\)-calculus, giving full computability power for the configuration language to describe compositions in some \#-programming system. In order to improve the reader understanding on how overlapping composition takes place using such kind of terms, let \(t_1\ t_2\) be an application term of the following form:

\[
\lambda_{\text{chan}}:\begin{pmatrix}
\text{recv} : \sigma_r \\
\text{send} : \sigma_s \\
\end{pmatrix}
\]

\[
\text{peer}_{11} : \begin{pmatrix}
\text{chan} \cdot \text{send} \otimes_{\otimes_1} s_1 \\
\text{send} : u_s \\
\text{recv} : u_r \\
\end{pmatrix}
\]

\[
\text{peer}_{12} : \begin{pmatrix}
\text{chan} \cdot \text{send} \otimes_{\otimes_2} s_3 \\
\text{send} : u_s \\
\text{recv} : u_r \\
\end{pmatrix}
\]

\[
\text{peer}_{1} : \begin{pmatrix}
\text{send} : s_1' \\
\text{recv} : s_2' \\
\end{pmatrix}
\]

\[
\text{peer}_{2} : \begin{pmatrix}
\text{send} : s_1' \\
\text{recv} : s_2' \\
\end{pmatrix}
\]

\[
\text{peer}_{3} : \begin{pmatrix}
\text{send} : s_3' \\
\text{recv} : s_3' \\
\end{pmatrix}
\]

The term \(t_1\) models the configuration of a \#-component with three units, named \(\text{peer}_1,\ \text{peer}_2,\) and \(\text{peer}_3\), that cooperate to implement some parallel computation. The variable \(\text{chan}\) models an inner component of \(t_1\), representing a communication channel from which unit \(\text{peer}_1\) may send a message to the unit \(\text{peer}_3\). This interpretation conforms to the view of components as functors, whose arguments represent their constituent components provided at runtime.

The type of \(\text{chan}\) specifies that the required \#-component must have two units named \(\text{send}\) and \(\text{recv}\), whose invocation causes communication of data from the process that has the unit \(\text{send}\) to the process that has the unit \(\text{recv}\) as one of their slices. Since the types of \(\text{send}\) and \(\text{recv}\) in the host language are \(\sigma_s\) and \(\sigma_r\), \(t_1\) expects that the units \(u_r\) and \(u_s\) of \(t_2\) be typed as \(\sigma_s\) and \(\sigma_r\), respectively.

The order of the units in the type of \(\text{chan}\) is irrelevant with respect to the semantics of the application of \(t_2\) to \(t_1\), since, the label \(l\) in an abstract slice \(x:l\), such as \(\text{chan}.\text{send}\) or \(\text{chan}.\text{recv}\), indicates which unit of the \#-component abstracted in \(x\) is slice of the resultant unit. We say that a variable \(x\) in an abstraction term \(\lambda x.t\) is sliced in \(t\) if it is referred in \(t\) by their slice variables, for distinct units of \(t\).

The unit \(\text{peer}_1\) is defined by an application of the folding combinator \(\otimes\) to the unit abstracted by the slice variable \(\text{chan}.\text{send}\), which represents the sending slice of the channel, and the unit \(s_1\), which may represent the local computation performed by \(\text{peer}_3\). The unit \(\text{peer}_3\) may be defined in an analogous way. The folding combinators \(\otimes_1\) and \(\otimes_2\) are closed under \(\Sigma\), and they are defined by the \#-programming system, as well as \(\Sigma\) and \(\Sigma\).

The application of the reduction rules of Figure 7 to the term \(t_1\ t_2\) yields a \#-component like

\[
\begin{pmatrix}
\text{peer}_1 : s_1' \\
\text{peer}_2 : s_2' \\
\text{peer}_3 : s_3' \\
\end{pmatrix}
\]

, where \(s_1'\) and \(s_3'\) are the results of \(u_s \otimes_{\otimes_1} s_1\) and \(u_r \otimes_{\otimes_2} s_3\), respectively. Thus, the type of \(t_1\), according to the type rules of Figure 8 is

\[
\begin{pmatrix}
\text{send} : \sigma_s \\
\text{recv} : \sigma_r \\
\end{pmatrix} \rightarrow
\begin{pmatrix}
\text{peer}_1 : s_1'' \\
\text{peer}_2 : s_2'' \\
\text{peer}_3 : s_3'' \\
\end{pmatrix}
\]

, where the unit types \(\sigma_s\), \(\sigma_s\), and \(\sigma_r\) are the types of \(s_1',\ s_2',\) and \(s_3',\) respectively. In the functor type of \(t_1\), the hole, denoted by the symbol \(\cdot\), and the reordering of unit types, makes possible to define the correspondence of slices to the units of the target type in an abstraction term. Holes and the order of units do not affect affect the meaning of types. Types that differ by the use of holes and/or by ordering of units are equivalent.

The reduction rules are based on the applicative evaluation of \(\lambda\)-calculus. Moreover, the meaning of the projection operator \(\cdot\), which occurs when a variable \(x\) in a unit term \(x.l\) is
a unit type definition is analogous in the case when type variables $X$ component functor $x$. Thus, it is not possible to apply a component functor to a

performance needs and abstraction levels in HPC. We argue that a combination of existential to hide architectural details of execution platforms, does not fit applications for sequential and distributed computing, trying computationally intensive parts of parallel programs. So, the which largely influence the choice of algorithms to implement computation. $\Gamma \vdash t : S \quad \Gamma \vdash S < T$ (T-Sub)

Figure 9 presents terms and types that define contexts and fetching. A context abstraction term $[X <: T_3] \triangleright t_2$ denotes the configuration of a #-component that executes in the context defined by the formal context parameter $X$. The type of $[X <: T_3] \triangleright t_2$ is $[X <: T_3] \triangleright T_2$, where $T_2$ is the type of $t_2$ by supposing that $X <: T_3$. In a configuration, the actual context may be specified by a context application $t_1 < [T_3]'$, where $T_3' <: T_3$. We can type $t_1 < [T_3]'$ as $[[X <: T_3] \triangleright T_2] \triangleright [T_3]'$, where $T_3'$. A term like $[[X <: T_3] \triangleright T_2] \triangleright [T_3]'$, which applies an abstract component to a context, is called an instantiation. It is semantically equivalent to $[X \mapsto T_3'] T_2$.

Example #1: Suppose $T_1$ is the abstract component FARM of Section III by ignoring all context parameters except $E <: \text{MPIBasic}$. MPIBasic denotes the type of #-components that implement an interface for message-passing using the five basic subroutines of MPI [8]. Thus, #-components of FARM may be tuned for any message passing library that implement the basic subset of MPI, by supplying $E$ with the abstract component (type) of the library. For example, FarmMPI supplies $E$ with MPIFULL, for any library that implement the complete specification of MPI. For that, MPIFULL $<: \text{MPIBasic}$. You can use the same approach to tune FARM #-components according to the other context parameters.

Example #2: Let CHANNELMODE be the type of #-components that define communication modes of channels in some library of components for channel-based distributed synchronization. Let SYNCHRONOUS, BUFFERED, and READY be subtypes of CHANNELMODE that define possible choices of communication modes. A component type $\text{CHANNEL} \equiv [[X <: \text{CHANNELMODE}] \triangleright \text{CHANNELType}]$, may denote the type of #-components implementing channels. A user may use a fetch term $?ch : \text{CHANNEL} \triangleright ['S\text{YNCHRONOUS}']$, where

 replacing a term $t$ in the reduction of a substitution clause $[x \mapsto t] t'$, is:

$$[l_i : u_i = 1 \ldots k] T = u_{i_1}, \text{ where } l = l_i$$

Thus, it is not possible to apply a component functor to a component functor $\lambda x : T. t$ where $x$ is sliced in $t$. The definition is analogous in the case when type variables $X$ in a unit type $X.l$ is replaced by a type $T$ in the reduction of a substitution clause $[X \mapsto T] t'$.

B. Context Abstractions

In HPC, the heterogeneous nature of execution platforms leads to the needs of providing the ability to tune implementation of components according to their architectural features, which largely influence the choice of algorithms to implement computationally intensive parts of parallel programs. So, the traditional universal polymorphism employed in corporative applications for sequential and distributed computing, trying to hide architectural details of execution platforms, does not fit the needs of HPC. We argue that a combination of existential and universal polymorphism is the better way to reconcile performance needs and abstraction levels in HPC.

Figures 8 and 9 present the types of contexts and fetching. A context abstraction term $[X <: T_3] \triangleright t_2$ denotes the configuration of a #-component that executes in the context defined by the formal context parameter $X$. The type of $[X <: T_3] \triangleright t_2$ is $[X <: T_3] \triangleright T_2$, where $T_2$ is the type of $t_2$ by supposing that $X <: T_3$. In a configuration, the actual context may be specified by a context application $t_1 < [T_3]'$, where $T_3' <: T_3$. We can type $t_1 < [T_3]'$ as $[[X <: T_3] \triangleright T_2] \triangleright [T_3]'$. A term like $[[X <: T_3] \triangleright T_2] \triangleright [T_3]'$, which applies an abstract component to a context, is called an instantiation. It is semantically equivalent to $[X \mapsto T_3'] T_2$.

Example #1: Suppose $T_1$ is the abstract component FARM of Section III by ignoring all context parameters except $E <: \text{MPIBasic}$. MPIBasic denotes the type of #-components that implement an interface for message-passing using the five basic subroutines of MPI [8]. Thus, #-components of FARM may be tuned for any message passing library that implement the basic subset of MPI, by supplying $E$ with the abstract component (type) of the library. For example, FarmMPI supplies $E$ with MPIFULL, for any library that implement the complete specification of MPI. For that, MPIFULL $<: \text{MPIBasic}$. You can use the same approach to tune FARM #-components according to the other context parameters.

Example #2: Let CHANNELMODE be the type of #-components that define communication modes of channels in some library of components for channel-based distributed synchronization. Let SYNCHRONOUS, BUFFERED, and READY be subtypes of CHANNELMODE that define possible choices of communication modes. A component type $\text{CHANNEL} \equiv [[X <: \text{CHANNELMODE}] \triangleright \text{CHANNELType}]$, may denote the type of #-components implementing channels. A user may use a fetch term $?ch : \text{CHANNEL} \triangleright ['S\text{YNCHRONOUS}']$, where

1Components in # programming environments may not be defined by concrete software artifacts, such as source codes or configuration files. The example with component types CHANNELMODE, SYNCHRONOUS, BUFFERED, and READY illustrates this fact. They are examples of qualifier components, a kind of #-component introduced in HPC which aims to specify non-functional properties of #-components.

Figures 8 and 9 present terms and types that define contexts and fetching. A context abstraction term $[X <: T_3] \triangleright t_2$ denotes the configuration of a #-component that executes in the context defined by the formal context parameter $X$. The type of $[X <: T_3] \triangleright t_2$ is $[X <: T_3] \triangleright T_2$, where $T_2$ is the type of $t_2$ by supposing that $X <: T_3$. In a configuration, the actual context may be specified by a context application $t_1 < [T_3]'$, where $T_3' <: T_3$. We can type $t_1 < [T_3]'$ as $[[X <: T_3] \triangleright T_2] \triangleright [T_3]'$. A term like $[[X <: T_3] \triangleright T_2] \triangleright [T_3]'$, which applies an abstract component to a context, is called an instantiation. It is semantically equivalent to $[X \mapsto T_3'] T_2$.

Example #1: Suppose $T_1$ is the abstract component FARM of Section III by ignoring all context parameters except $E <: \text{MPIBasic}$. MPIBasic denotes the type of #-components that implement an interface for message-passing using the five basic subroutines of MPI [8]. Thus, #-components of FARM may be tuned for any message passing library that implement the basic subset of MPI, by supplying $E$ with the abstract component (type) of the library. For example, FarmMPI supplies $E$ with MPIFULL, for any library that implement the complete specification of MPI. For that, MPIFULL $<: \text{MPIBasic}$. You can use the same approach to tune FARM #-components according to the other context parameters.

Example #2: Let CHANNELMODE be the type of #-components that define communication modes of channels in some library of components for channel-based distributed synchronization. Let SYNCHRONOUS, BUFFERED, and READY be subtypes of CHANNELMODE that define possible choices of communication modes. A component type $\text{CHANNEL} \equiv [[X <: \text{CHANNELMODE}] \triangleright \text{CHANNELType}]$, may denote the type of #-components implementing channels. A user may use a fetch term $?ch : \text{CHANNEL} \triangleright ['S\text{YNCHRONOUS}']$, where

1Components in # programming environments may not be defined by concrete software artifacts, such as source codes or configuration files. The example with component types CHANNELMODE, SYNCHRONOUS, BUFFERED, and READY illustrates this fact. They are examples of qualifier components, a kind of #-component introduced in HPC which aims to specify non-functional properties of #-components.
t is the code that uses the channel ch, in order to find an implementation of a synchronous channel in the environment Σ of deployed #-components.

C. Fetch Terms

In order to support contexts, a distinguishing feature of HOCC are fetch terms, denoted by ?x : T1. t1. As the reader can see in the reduction rules E→Fetch1 and E→Fetch2, in Figure 7, a configuration t2 associated to the context type T1 is searched in an environment Σ, using the function fetch. E→Fetch1 and E→Fetch2 differ on the shape of the context type T1. For E→Fetch1, T1 is an instantiation of the form T11 ≪ [T12], where T11 denotes an abstract component, of the form [X ≪ T3] ⊃ T4, and T12 denotes the context where it is being applied, which serves to restrict the possible choices for t2 among the #-components that implement the abstract component T11. Thus, we say that the chosen configuration t2, assigned to the instantiation T11 ≪ [T12] in Σ, is an implementation of the component type T11 that is tuned to the actual context T12. This is only possible if 0 ⊢ t2 ≪ [T12] : [X ⊃ T12] T4, requiring that T12 ≪: T3. In such case, the configuration t2 ≪ [T12] replaces all occurrences of the variable x in t1. The passing of T12 to t2 informs t2 about the actual context where it will execute. For E→Fetch2, there is no context for restricting the abstract component denoted by T11 itself. Thus, it is not necessary to pass the context to t2.

Example #3: Let Farm ∼ [[MPIBASIC]] and Farm ∼[[MPIFULL]] be well-formed instantiations. In the environment Σ, now suppose that Farmmpl is mapped to Farm ∼ [[MPIFULL]], but there is no #-component mapped to Farm ∼ [[MPIFULL]]. In the evaluation of a fetch term ?x : Farm ∼ [[MPIFULL]]. t1 in the environment Σ, that may occur in the configuration of the abstract component ROMBERGINTEGRATION, which has a Farm farm as an inner component, the function fetch will not find a term in Σ to supply x in its first attempt, but it will generalize the context trying to find an appropriate supertype instantiation that fits the context. Such generalization process will cause Farmmpl to supply x in t1. This is safe because the instantiation Farm ∼ [[MPIBASIC]] is a subtype of the instantiation Farm ∼ [[MPIFULL]] according to ≪ relation (Figure 10). Intuitively, a farm that uses only basic routines of MPI will operate safely in a context where a complete implementation of MPI is provided. Finally, Farmmpl will be applied to MPIFULL, defining its actual execution context.

D. The Environment Σ - Formal Definition

In a parallel execution platform, the environment Σ denotes the set of deployed configurations with two relevant relations between them: single inheritance (nominal subtyping), between pairs of component types, and assignments, between instantiations and configurations. The function fetch denotes a resolution algorithm that searches Σ to find the most specific configuration that can supply a given instantiation. Therefore, we formally define Σ by a pair of directed acyclic graphs (DAG) Σ = ⟨G1, G2⟩, where:

- G1 = (V1, E1) is a tree, where V1 contains a set of component types and E1 : V1 ⊃ V1 contains a set of nominal subtyping relations between component types in E1 (single inheritance). The root of G1 is Top;

- G2 = (V0 ∪ V1, E2) is a bipartite graph, where V0 is a set of configurations and V1 is set of component types. E2 : V1 ⊃ V0 is a partial function that denotes assignments, mappings from instantiations to configurations, such that

\[ T1 \triangleleft [[T2]] \in V1 \quad t \in V0 \quad \emptyset \vdash t \triangleleft [[T2]] : T1 \triangleleft [[T2]] \]

and

\[ T \text{ does not match } T1 \triangleleft [[T2]] \quad T \in V2 \quad t \in V0 \quad \emptyset \vdash t : T \]

\[ T \Rightarrow t \in E2 \]

An assignment T ⊃ t ∈ E2 specify that t is the configuration that represent the best #-component for the instantiation T in Σ. We are interested in the first kind of assignment, where T ≡ T1 ≲ [[T2]] and T1 is an abstract component instantiated by t for the context T2.

E. The Resolution Algorithm

The main use of Σ is to support the specification of the semantics of fetch terms, like ?x : T1. t1. The function fetch implements a resolution algorithm that traverse the structure of

\[ \text{fun fetch}(T, \Sigma : (\langle x \rangle, \langle x \rangle, E2)) \text{ returns } h \]

if (\exists h'. U \equiv h' \in E2) then

\[ h = h' \]

else if T matches T1 ≲ [[T2]] and \exists T2\sup, T2 ⇒ T3 ∈ E1

\[ h = \text{fetch}(T1 \triangleleft [[T2\sup]], \Sigma) \]

else

\[ h = \text{none} \]

end

Figura 10. Subtyping Rules

Figura 11. Resolution Algorithm
Noting an arbitrary one step derivation of and fetching. The codification of such terms and types in us concentrate on the terms and types implementing contexts lapsing composition terms are easily codified in \texttt{Top}. Thus, it is sufficient to take \( T \) that \( \emptyset \vdash t < \{T_2\} \), it is required that \( \emptyset \vdash t < \{T_2\} : T_1 \vdash t < \{T_2\} \). By application of the typing rule \( T \Rightarrow T_{\text{Abs}} \),

\[
\emptyset \vdash t : \{X_1 : T_{11}\} \triangleright T_1 \quad \Gamma \vdash T_2 : \{T_2\}
\]

. Thus, it is sufficient to take \( T_{11} = T_2^{\text{sup}} \). □

Termination and Determinism: The resolution algorithm always stop because there is a \texttt{Top} type that is an upper bound for the subtype relation, and it will be reached after a finite steps of generalization. Also, the algorithm is deterministic, due to the single inheritance in \( \Sigma \) and due to the fact that there is only one assignment for each instantiation.

### F. Formal Basis

The terms and types of HOCC can be codified in kernel System F_<:, a type system with known properties. Since overlapping composition terms are easily codified in \( \lambda \)-calculus, let us concentrate on the terms and types implementing contexts and fetching. The codification of such terms and types in System F_<: is presented in the table below:

<table>
<thead>
<tr>
<th>Context abstraction term</th>
<th>( [X &lt; : T'] \triangleright t \equiv \lambda X &lt; : T' {X, t} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context application term</td>
<td>( t &lt; [T] \equiv t {T} )</td>
</tr>
<tr>
<td>Fetch term</td>
<td>( \Sigma \vdash \exists x : U. t_1 \equiv (\lambda x. t_1) t_2 ) where ( t_1 = f\text{etch}(U, \Sigma) )</td>
</tr>
<tr>
<td>Context abstraction type</td>
<td>( {X_1 : T_{11}} \triangleright T_2 \equiv \forall X_1 &lt; : T_{11}. \exists X_2 &lt; : X_1, T_2 ) where ( T_2' = [X_1 \mapsto X_2, T_2] )</td>
</tr>
</tbody>
</table>

Now, consider the following typing proof in \( \tau \)-HOCC, denoting an arbitrary one step derivation of \( T \Rightarrow T_{\text{Abs}} \):

\[
\Gamma, X_1 < : T_1 \vdash t : T_2
\]

. By the above codification, one may perform the same proof in System F_<:.

By the above codification, one may perform the same proof in System F_<:

<table>
<thead>
<tr>
<th>( \Gamma, X_1 &lt; : T_1 \vdash t : T_2 )</th>
<th>( \Gamma, X_1 &lt; : T_1 \vdash t : T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma, X_1 &lt; : T_1 \vdash t : T_2 )</td>
<td>( \Gamma, X_1 &lt; : T_1 \vdash t : T_2 )</td>
</tr>
<tr>
<td>( \Gamma, X_1 &lt; : T_1 \vdash t : T_2 )</td>
<td>( \Gamma, X_1 &lt; : T_1 \vdash t : T_2 )</td>
</tr>
<tr>
<td>( \Gamma, X_1 &lt; : T_1 \vdash t : T_2 )</td>
<td>( \Gamma, X_1 &lt; : T_1 \vdash t : T_2 )</td>
</tr>
<tr>
<td>( \Gamma, X_1 &lt; : T_1 \vdash t : T_2 )</td>
<td>( \Gamma, X_1 &lt; : T_1 \vdash t : T_2 )</td>
</tr>
<tr>
<td>( \Gamma, X_1 &lt; : T_1 \vdash t : T_2 )</td>
<td>( \Gamma, X_1 &lt; : T_1 \vdash t : T_2 )</td>
</tr>
</tbody>
</table>

. showing that a one-step derivation by \( T \Rightarrow T_{\text{Abs}} \) may be interpreted as a derivation in System F_<:. Notice that the final goals are equivalent.

Using the same approach, one can prove the equivalence of the other typing rules of HOCC with typing derivations in System F_<:.

The mapping to kernel System F_<: ensures that typing and subtyping in HOCC are decidable. In a \( \# \)-programming system, the knowledge of algorithms to check both safety of assignment of configurations to instantiations in deployment and inheritance relations between deployed abstract components is required to implement \( \Sigma \).

### V. Implementation of HTS in HPE

The \textsc{Back-End} of HPE is implemented on top of the CLI/Mono platform [16], a free source implementation of the CLI standard [17]. Mono provides an infrastructure for deployment and execution of components that are independent on languages and architectures. Components are managed as \textit{assemblies}, libraries compiled as single files of extension “dll”. In a CLI platform, the information about deployed assemblies is managed by the module GAC (Global Assembly Cache). The choice of a CLI compliant platform to implement HPE is not arbitrary. In fact, CLI platforms introduced some features that are not supported by other popular virtual machines, such as JVM, but that are relevant in HPC domain [18], like true multidimensional arrays, side-by-side control of component versions, programming languages interoperability, a strongly typed intermediary language, safe memory pointers, and a high performance virtual execution engine that makes possible to execute native (unmanaged) code in critical execution points.

A unit assembly contains a package where one or more modules are declared. As depicted in Figure 12, the modules in unit assemblies of the abstract components \textsc{Distribute} and \textsc{DistributeInterval}, from the case study of Section III, are interfaces, so called \textit{unit interfaces}, whose type variables and their bounds correspond to the context variables and context bounds of context parameters of abstract components. Since \textsc{DistributeInterval} is an specialization of \textsc{Distribute} (\textsc{DistributeInterval} <: \textsc{Distribute}), their unit interfaces are defined by extending the corresponding unit interfaces of \textsc{Distribute}. Analogously, the modules in the unit assembly of the \#-component \textsc{DistributeIntervalImpl}, which implement \textsc{DistributeInterval} for some context, are \textit{unit classes}. Therefore, the unit classes of a \#-component implement the corresponding unit interfaces of the abstract component it implements. Unit classes must be also generic on the context parameters of the interfaces they implement.

As one can see in Figure 12, the package inside a unit assembly has a \textit{base} and a \textit{user} module (class or interface). The base modules are automatically generated by the Front-End, from the configuration of the units, defining properties for accessing the slices of the unit, called \textit{slices ports}. The code of base modules should not be modified by the programmer. A user module is defined by inheritance from a base module,
defining the user-defined parts of the unit. For that, base classes must be abstract.

For each kind \( k \), there is an abstract component \( \text{Top}_k \), and a \#-component \( \text{TopImpl}_k \). Formally, abstract components and \#-components of kind \( k \) are implicitly defined by specialization from \( \text{Top}_k \) and \( \text{TopImpl}_k \), respectively. As a consequence, user classes of a \#-components of kind \( k \) must implement the interface imposed by the unit interfaces of \( \text{Top}_k \), which is abstract or has a default implementation in the user classes of \( \text{TopImpl}_k \). For example, a user class of a computation unit must implement the method \textit{compute}, defining the computation logic that implement the role of the unit in the computation, in addition to all methods and properties required by the corresponding user unit interface.

### A. Dynamic Composition and Resolution

To execute a program, a user must submit an instantiation to the method \textit{runApplication} of the BACK-END service, by selecting an abstract component of kind application and a set of abstract components to supply their context parameters. The BACK-END will search a \#-component that is assigned to the instantiation in the environment, by using an extended version of the resolution algorithm of Figure 11. If it is found, the main method of each unit of the application is called, which execute the following procedure (from \textit{AdapativeQuadratureImpl}):

```java
static void Main(string [] args) {
    IPeerImpl<ITestingFunction> peer = new IPeerImpl<ITestingFunction>();
    DGAC.Init(ID, "peer", peer, args);
    peer.compute();
    DGAC.Finalize();
}
```

The call to the method \textit{compute} of \textit{peer} initiates the application logic. Such method must be implemented by unit user classes of applications and computations.

In \textit{DGAC.Init}, the method \textit{createSlices} of \textit{peer}, implemented by all unit base classes, is invoked, causing calls to the method \textit{DGAC.createSlice} for each slice of the unit. For illustrating, the \textit{createSlices} of the \textit{worker} unit of FarmImpl, declared in its base module (class \textit{BaselWorkerImpl}), is presented below:

```java
override public void createSlices() {
    base.createSlices();
    Receive = (S) DGAC.createSlice(this, UID, "scatter", "receive");
    "execut
```

\textit{DGAC.createSlice} is a parallel method, invoked synchronously for all the units of an inner component in order to instantiate it. It is responsible to call the resolution procedure to find \#-components for the given instantiations. After that, it invokes the method \textit{DGAC.loadImpl} to instantiate the slice (unit object), by passing the actual context parameters. Finally, it calls the \textit{createSlices} method of the instantiated slice. The code of \textit{DGAC.loadImpl} is outlined below:

```java
public static HUnit loadImpl(IInstantiation instInfo, Type[] typeParams) {
    // FIND THE #-COMPONENT AND RETURN ITS UNIT
    IUnitClassInfo u = Loader.resolveImpl(instInfo);
    // LOAD ASSEMBLY USING THE ASSEMBLY STRING
    Assembly a = Assembly.Load(u.Assembly_string);
    // FETCH THE UNIT CLASS FROM THE ASSEMBLY
    Type t = a.GetType(u.Class_name);
    // SUPPLY FORMAL PARAMETERS OF THE UNIT CLASS
    Type cf = typeParams.Length > 0 ?
    t.MakeGenericType(typeParams) : t;
    // CREATE AN INSTANCE OF THE UNIT CLASS
    HUnit unit = (hpe.basic.IUnit) Activator.CreateInstance(cf);
    return unit;
}
```

The mutual recursive calls between \textit{createSlices} and \textit{DGAC.createSlice} loads all the inner components.

The instantiation that must be loaded, which is necessary to be passed to the resolution algorithm implemented by the method \textit{Loader.resolveImpl}, is given by the \textit{instInfo} argument. The class \textit{UnitClassInfo} stores the information that is necessary to load a class: the assembly string and the class identification of the form \((Class)^n\), where \(Class\) is the name of the class and \(n\) is the number of generic type arguments. An object of such class is returned by the resolution algorithm. Finally, the \textit{typeParams} argument carries the unit interface types, of the current context, that must be passed to the loaded unit class. The object in the variable unit is the unit object, returned by \textit{loadImpl}. The polymorphism of CTS (Common Type System), the type system of CLI-compliant virtual execution platforms, allows dynamic inspection and instantiation of the actual type of a context variable. This is not supported by JVM (Java Virtual Machine), due to \textit{type erasure}, one of our reasons to adopt a CLI platform.

### B. Binding Time

In Table I, the time to load all inner components in an application, so called \textit{binding time} (column \textit{bind}), is compared to the effective execution time (column \textit{exec}), using the application of Section III. The experiment aims to evidence that binding time is constant with the problem size and increases sublinearly with the number of processors, mostly depending on the complexity of collective operations, by supposing that the DGAC database is replicated among all nodes of the
cluster. In fact, the reader can observe that the observer binding
times for different values of \( N \), the number of dimensions of
the integrand function that defines the problem size, remains
constant, whereas it varies slightly for \( P \) varying from 1 to
4, where the database is replicated on all nodes. Due to a
limitation of MySQL-cluster 5.1, the database is distributed in
two sets of nodes for \( P = 8 \), increasing binding time.

The results shows that binding time can be easily amorti-
zed in realistic cluster computing scenarios of today, where
effective execution time is high and there are only dozens
of multi-core processing nodes. It is important to notice that
our first prototype could be yet more optimized to minimize
binding time. An interesting optimization is to execute binding
procedures (resolution and loading of inner components) con-
currently with the computation, taking advantage of multi-core
and multiprocessor nodes of the cluster. Another optimization
is to store DGAC in-memory, using an appropriate data
structure, avoiding costly accesses to the database.

VI. CONCLUSIONS AND FURTHER WORKS

HPE has been developed as a prototyping and proof-of-
concept environment to validate the ideas behind \# programming
systems, such as the design of HTS.

There are many extensions to HTS that we have omitted
from this paper to make easy the understanding their basic
concepts, such as contexts and fetching. Two further extensions
deserves special attention: recursive component types, which
makes possible to configure recursive process topologies like
pipe-lines and tree-based networks; and dependent component
types, allowing to tune implementations of abstract component
for a given context (\#-components) according to another di-

mension besides context parameters, the number of processors
where the \#-component will execute, which is known only
at execution time. Both are motivated by intrinsic issues
only found in parallel programming, respectively the recursive
nature of process topologies and the different scalability
properties of parallel algorithms.

VII. ACKNOWLEDGMENTS

This work is sponsored by CNPq, grants 475826/2006-0 and
480307/2009-1.

REFERÊNCIAS


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), Madrid, Spain. IEEE Computer Society, 2004,
pp. 76–84.

141–150, 2006.

M. Sottile, T. Epperly, and Dahlgreen Tamara, “The CCA Component
Model For High-Performance Scientific Computing,” Concurrency and
Computation: Practice and Experience, vol. 18, no. 2, pp. 215–229,
2006.

Software Composition with Sharing,” in European Conference on Object

[6] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, W. Getov, L. Henrio,
and C. Prez, “GCM: A Grid Extension to Fractal for Autonomous
Distributed Components,” Annals of Telecommunications, vol. 64, no. 1,
pp. 5–24, 2009.

an Architecture for Component-Oriented Parallel Programming,”
Concurrency and Computation: Practice and Experience, vol. 19, no. 5,

Standard for MPP and Workstation,” Communications of ACM, vol. 39,
n. 7, pp. 84–90, 1996.

Made Simple and Efficient,” Journal of Universal Computer Science,

Concerns,” in Workshop on Early Aspects - Aspect Oriented Software

Improving Practice of Parallel Programming,” INFORMATION, An

Interscience, 2005.

Performance Computing Component Models,” in Joint Conference on
HPC Grid programming Environments and Components (HPC-GEco),
and on Components and Frameworks for High Performance Computing
(HPC"CompFrame), 2007.

for Skeletal Parallel Programming,” Parallel Computing, vol. 30, no. 3,


[18] W. Vogels, “HPC.NET - Are CLI-based Virtual Machines Suitable for
conference on Supercomputing. IEEE Computer Society, Nov. 2003,
pp. 36–46.

<table>
<thead>
<tr>
<th>( P )</th>
<th>( \text{bind} )</th>
<th>( \text{exec} )</th>
<th>( \text{over} )</th>
<th>( \text{bind} )</th>
<th>( \text{exec} )</th>
<th>( \text{over} )</th>
<th>( \text{bind} )</th>
<th>( \text{exec} )</th>
<th>( \text{over} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.4s</td>
<td>0.4s</td>
<td>84%</td>
<td>2.4s</td>
<td>6.9s</td>
<td>26%</td>
<td>2.4s</td>
<td>161.3s</td>
<td>1.5%</td>
</tr>
<tr>
<td>2</td>
<td>2.6s</td>
<td>0.3s</td>
<td>90%</td>
<td>2.6s</td>
<td>3.6s</td>
<td>42%</td>
<td>2.6s</td>
<td>80.7s</td>
<td>3.2%</td>
</tr>
<tr>
<td>4</td>
<td>2.9s</td>
<td>0.2s</td>
<td>93%</td>
<td>2.9s</td>
<td>1.9s</td>
<td>61%</td>
<td>2.9s</td>
<td>41.6s</td>
<td>6.6%</td>
</tr>
<tr>
<td>8</td>
<td>4.7s</td>
<td>0.2s</td>
<td>96%</td>
<td>4.7s</td>
<td>1.1s</td>
<td>81%</td>
<td>4.7s</td>
<td>21.3s</td>
<td>18.0%</td>
</tr>
</tbody>
</table>

Tabela I
EVALUATION (% OF THE BINDING TIME (bind) IN RELATION TO THE TOTAL EXECUTION TIME (bind + exec)