Haskell#*: Coordinating Functional Processes

Francisco Heron de Carvalho-Junior
Departamento de Computação
Universidade Federal do Ceará (UFC)
Fortaleza, Brazil

Rafael Dueire Lins
Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil

June 2004

Abstract

This paper presents Haskell#, a coordination language targeted at the efficient implementation of parallel scientific applications on loosely coupled parallel architectures, using the functional language Haskell. Its programming environment encompasses an editor, a compiler into Petri nets, a Petri net animator and proof tool, and a skeleton library. Examples of applications, their implementation details and performance figures are presented.

1 Introduction

The peak performance of parallel architectures is growing at a faster pace than predicted by Moore’s law, that states that at each 18 months computer hardware becomes twice as fast and halves its sale price. However, parallel programming tools have not being able to reconcile portability, scalability and a higher level of abstraction without imposing severe performance penalties to applications [28].
The emerging technologies in the 1990s gave birth to new challenges in high-performance computing. The advent of clusters [8], low cost supercomputers built on top of networks of workstations and personal computers, disseminated supercomputing among academic institutions, industries and companies [11, 4, 19, 20]. More recently, advances in wide area network interconnection technologies have made possible to use their infra-structure to build distributed supercomputers of virtually infinite scale, the grids, which are particularly suitable for addressing very coarse grained scientific computing applications. Great efforts to make these technologies viable are been promoted, with promising results [34].

Clusters and grids sparked a myriad of new applications in supercomputing for scientific computation. Most of them are not addressed adequately by contemporary tools, yielding inefficient distribution of parallel programs [83]. In [10], some parallel programming approaches used in scientific computing are compared in relation to scalability (efficiency), generality and abstraction dimensions. MPI (Message Passing Interface) [63], the most widespread message passing library, provides scalability, generality, but is less abstract than TCE (Tensor Contraction Engine) [7], PETSc [5], GA (Global Array) [66], openMP [67], auto-parallelized C/Fortran90 and HPF (High Performance Fortran) [32]. PETSc and TCE are specific purpose libraries for scientific computing, providing a high level of abstraction and scalability. Implicit approaches, such as C/Fortran90, present low scalability, high level of abstraction and high generality. These observations illustrate that, despite the efforts conducted on the last decade, the need for new parallel programming environments that reconcile a high-level of abstraction, modularity, and generality with scalability and peak performance is still a challenge [28, 77, 82].

This paper presents Haskell\#\textsuperscript{2}, a process-oriented coordination language [35] where Haskell [75], a language considered de facto a standard in lazy functional programming, is used for programming at computation level. Haskell\#\textsuperscript{2} aims to provide high-level programming mechanisms without sacrificing performance significantly, by minimizing the overheads of the management of parallelism. One of the most important concerns in Haskell\#\textsuperscript{2} is to make easier to prove correctness of programs. For that, a divide-and-conquer approach was adopted to increase the chances of formally analyzing programs: the process network is completely orthogonal to the sequential blocks of code (process functionality). Haskell allows sequential programs to be proved correct in a simpler fashion than their equivalent written in languages which belong to other programming paradigms. The communication primitives
were designed in such a way as to allow their translation into Petri nets [72], a well reputed formalism for the specification of concurrent systems, with several analysis and verification tools [80, 12] available.

Haskell\# emphasizes compositional programming and provides support for skeletons [25]. Skeletons are used to expose topological information that can guide the Haskell\# compiler in the generation of more efficient code. MPI (Message Passing Interface) [29] is used to manage parallelism without claiming for any run-time support. Due to the recent development of interoperable [84] and grid enabled [49] versions of MPI Haskell\# programs may be executable on grids without any extra burden.

Examples of benchmark programs and their performance figures are provided, elucidating the most important aspects of programming in Haskell\#.

This paper comprises five other sections. Section 2 gives background for programming in Haskell\#, focusing on programming abstractions. Section 3 presents motivating application examples of Haskell\# programming. Section 4 presents details about current implementation of Haskell\# for clusters. Section 5 presents performance figures about applications presented in Section 3 running on implementation described in Section 4. Section 6 concludes this paper outlining the work in progress with Haskell\#.

2 Programming in Haskell\#  

Haskell\# programs are composed from a set of components, each one describing an application concern. Concerns may be functional or non-functional. Examples of functional concerns are the calculation of an exact solution for a system of linear equations and the calculation of a finite-difference approximation for a system of partial differential equations. An example of non-functional concern is the allocation of processes to processors. Components may be reused among Haskell\# programs.

In Haskell\# programming, the process of composing components is inductive. Simple components, functional modules implemented in Haskell, are basic building blocks. Given a collection of components, simple or composed ones, it is possible to define a new composed component by specifying their composition through Haskell\# Configuration Language (HCL). The result of this process is a hierarchy of components, where the main component, describing the application functionality, is at the root. Components at the leaves are simple components (always addressing functional concerns) and
intermediate nodes are composed components.

Under perspective of process-oriented coordination models [35], the collection of functional modules of a Haskell program forms a computation medium, while the collection of composed components forms a coordination medium. The concerns on the parallel composition of Haskell functional computations are sufficiently and necessarily resolved at the coordination level. The use of Haskell for programming the computation medium allows that coordination and computation languages be really orthogonal. Lazy lists allow the overlapping of communication and computation in process execution, without to need to embed coordination extensions in the code of the functional modules.

The idea of hierarchical compositional languages implemented using configuration languages is not a recent idea [13, 1]. Haskell difference from its predecessor languages resides in its support for skeletons, by allowing to partially parameterize the concern addressed by components, and its ability for overlapping them, making possible to encapsulate cross-cutting concerns [21]. The use of skeletons has gained attention of parallel programming community since last decade [25] and now it is supported by many languages and paradigms [79]. The problem of modularizing cross-cutting concerns have gained attention in software engineering research community, particularly for programming large scale object-oriented systems. An example of cross-cutting concern is validation procedures executed by processes for accessing computational resources in a grid environment. With respect to this feature, Haskell may implement the notions of AOP (Aspect Oriented Programming) [52] and Hyperspaces [68] using an unified set of language constructors. Skeletons may be overlapped, forming more complex ones.

Haskell programs may be translated into Petri nets. This allows to prove formal properties and to evaluate the performance of parallel programs using automatic tools. Some previous work have addressed the problem of translating Haskell programs into Petri nets [56, 23]. The expressive power of HCL for describing patterns of interaction among processes is equivalent to descriptive power of labelled Petri nets [71].

Now, relevant details about how Haskell programs are implemented are presented. HCL abstractions for programming at coordination medium are informally introduced and it is shown how simple components are programmed in Haskell. Motivating examples of Haskell are presented in Section 3, illustrating the use of Haskell programming abstractions. Appendix B formalizes an algebra for describing semantics of Haskell programming
abstractions. The informal description points at the corresponding Haskell\# algebraic constructions.

2.1 Programming Composed Components

Composed components, which form coordination medium of Haskell\# programs, are programmed in HCL configurations. HCL programming corresponds to the inductive step in Haskell\# programming task described in last section. In what follows, the constructors used at coordination level for programming Haskell\# applications are informally introduced. Their for-
A parallel version of MCP-Haskell [22] is used for exemplifying the syntax of HCL. MCP-Haskell [39] is a simplified sequential version of MCNP, a code developed at Los Alamos during many years for simulating the statistical behaviour of particles (photons, neutrons, electrons, etc.) while they travel through objects of specified shapes and materials [15]. HCL code of MCP-Haskell is shown in Figure 2. The corresponding network topology is presented in Figure 1. The parallelism is obtained from three sources. Firstly, tracking and tallying procedures must be executed concurrently using

\[ \text{MCP-Haskell} \]

Figure 2: HCL Code for MCP-Haskell

The parallelism is obtained from three sources. Firstly, tracking and tallying procedures must be executed concurrently using natural syntax is presented in Appendix A. Appendix B brings their algebraic semantics.

\[ \text{MCP-Haskell} \]

A parallel version of MCP-Haskell [22] is used for exemplifying the syntax of HCL. MCP-Haskell [39] is a simplified sequential version of MCNP, a code developed at Los Alamos during many years for simulating the statistical behaviour of particles (photons, neutrons, electrons, etc.) while they travel through objects of specified shapes and materials [15]. HCL code of MCP-Haskell is shown in Figure 2. The corresponding network topology is presented in Figure 1. The parallelism is obtained from three sources. Firstly, tracking and tallying procedures must be executed concurrently using
a pipe-line. The main source of parallelism is the second. It comes from the fact that particles may be tracked and tallied independently. To take advantage of this problem feature, a work pool pattern of interaction is employed, where a manager process distributes jobs (particles) to worker processes, on demand controlled by their availability, and collects the results from each job. A third source of parallelism comes from the fact that the statistics of different tallies may be computed in parallel. Thus, each statistical process in the network is responsible for computing a specified set of tallies. In the following sections, it is explained how a HCL configuration may implement this network topology.

A HCL configuration starts with a header, declaring the name of the composed component, its static formal parameters and its arguments and return points. MCP-Haskell\# has two static parameters, \( m \) and \( n \), which controls the number of parallel tasks, but no argument or return point is defined. In general, arguments and return points are not defined for the main component of an application. They are normally used in the configuration of the encapsulated functional concerns.

Figure 3: Diagrammatic Notation for Haskell\# Abstractions
2.1.1 The Basic Abstractions: Units and Channels

A Haskell\# configuration is specified by a collection of units, which are abstractions for agents that execute a particular task. Units synchronize using communication channels. The task performed by a unit is defined statically, by assigning a component for it. Units may be viewed as a “glue” for composing components. Units have interfaces, comprising collections of input and output ports. Interfaces are necessary for allowing units to be connected through communication channels. An interface also describes a partial order for the activation of ports during execution, characterizing the behavior of a unit. A communication channel is defined by linking two ports from opposite directions through a communication mode: synchronous, buffered and ready. Communication modes of Haskell\# channels have direct correspondence to MPI primitives, ensuring their efficient implementation, and have semantic equivalence with OCCAM [46] and CSP [43]. Ports linked through a communication channel are said to be communication pairs.

In Figure 2, lines 20 to 26 have declarations of units, whose identifiers are placed after the keyword unit. The assign declarations bind components to units. The interface of a unit is declared after the clause “#”. In the example, an interface class identifier is employed but it is possible to declare an interface directly. This topic is discussed further in the next section.

The low level of abstraction provided by units, ports and channels is not appropriate for programming large-scale and complex distributed parallel programs. Next sections introduce additional abstractions intended to raise the level of abstraction in HCL programming, simplifying the specification of large-scale and complex process topologies. Essentially, they provide support for partial topological skeletons.

2.1.2 Interface Classes

Haskell\# incorporates the notion of interface class for representing interfaces of units that present equivalent behavior. Examples of interface class declarations are shown in lines 07 to 18 of Figure 2. The identifier of an interface class is configured after the interface keyword. The notation $(i_1, i_2, \ldots, i_n) \rightarrow (o_1, o_2, \ldots, o_m)$ sets up $n$ input ports and $m$ output ports, with the respective identifiers. In a where clause, an interface composition operator (#) allows defining how an interface is obtained from the composition of existing ones. The semantics of the # operator is formalized in
Appendix B.

Units that declare the same interface name after “#” clause in unit declarations inherit the same behavior, specified in the corresponding interface declaration.

A small language is embedded in behavior clause of interface declarations, intended to describe partial orders in the activation of ports. Its combinators have semantic equivalence to operators of regular expressions controlled by semaphores [47], which are regular expressions enriched with an interleaving operator, represented in HCL by the combinator par, and counter semaphores primitives, represented by the primitives wait and signal. This feature ensures that the HCL descriptive power is equivalent to the power of terminal labelled Petri nets in describing the interaction patterns between processes.

2.1.3 Wire Functions

In an assignment declaration, it is necessary to map input and output ports of the unit to arguments and return points of the assigned component, respectively. The notation \((i_1, i_2, \ldots, i_n) \rightarrow (o_1, o_2, \ldots, o_m)\) may be used whenever the order of ports does not match the order of corresponding arguments/return points.

In fact, the association between the input and output ports and the arguments and return points of components in assign declarations defines how Haskell# glues coordination and computation media. Whenever an argument is not bound to an input port, an explicit value must be provided to it. Also, whenever a return point is not associated with an output port, it is not evaluated.

In wire clauses of unit declarations, HCL allows to define a wire function that maps a value received through an input port onto a value that is passed to an argument. Analogously, it is possible to define a wire function that receives a value produced at a return point yielding another value that is sent through the associated output port. This increases the chances that a component be reused without changing its internal implementation, in such cases where there is some type incompatibility between the type or meaning of arguments and the return points and the expected input and output ports types and meaning at coordination level.
2.1.4 Groups of Ports

Another useful feature of HCL is the replication of interface ports of a unit, forming groups of ports where individual members are referenced using enu-
meration indexes. A group of ports is treated as an individual entity from the local perspective of the unit. Thus, they are bound to a unique argu-
ment/return point and must be activated atomically. However, from a global view, individual ports of the group are treated in separate, being possible to connect them through different channels.

Groups of ports may be of two kinds: *any* or *all*. When a group of input ports of kind *all* is activated, each port member must receive a value. The array of values received is mapped to a unique value by using a wire function. Then, the value is passed to the argument mapped onto the group of ports. When an output group of ports is activated, the value yielded at the return point mapped to it is transformed, using an wire function, into an array of values that are sent through port members of the group. In activation of groups of ports of kind *any*, one port belonging to the group is chosen among ports whose communication pair is activated. Once the port is chosen, communication occur like in individual ports. Notice that wire functions are necessary for configuring groups of ports of kind *all*. Because of that, groups of ports are configured in clause **wire** of unit declarations, like exemplified in Figure 2 with *tally_entries* group of ports. For configuring a group of ports of kind *any*, use **any** keyword instead of **all** keyword, as illustrated in the example. Figure 4 illustrate semantics of wire functions.

![Figure 4: Wire Functions and Groups of Ports](image)
and groups of ports.

2.1.5 Stream Ports

Stream ports allow to transmit sequences of values (streams) terminated by an end marker. Haskell streams may be nested (streams of streams) at arbitrary nesting levels, which must be statically configured. Stream ports of units for which it is assigned a simple component must be mapped to argument and return points of lazy lists types in the functional module. Nested streams are associated to nested lazy lists of at least the same nesting level.

In interface declarations in lines 11 and 14, stream ports may be identified by the occurrence of sequences of symbols “*” after the identifier of the port. The number of *’s indicates its nesting level. For instance, stream ports particles and events of interface ITracking have nesting level equal to one. In Figure 9, where Haskell code of the functional module Tracking is presented, arguments and return points associated to particles and events ports of track unit are lazy lists of nesting level greater than or equal to one. Stream ports are essential for Haskell expressiveness, once it allows overlapping communication operations and computations during the execution of processes. The same approach is used by other parallel functional languages, such as Eden [14].

2.1.6 Configuring Arguments and Return Points of Composed Components

Arguments and return points of composed components are, respectively, input and output ports of units that are not connected through any communication channel. For specifying ports that must be connected to arguments and return points, HCL supports bind declarations.

2.1.7 The Distinction Between Processes and Clusters

It is convenient to distinguish between units associated to simple and composed components. The former are called processes, while the latter are called clusters. Processes are concrete entities and may be viewed as agents that perform sequential computations programmed in Haskell. Clusters are abstract entities and must be viewed as a parallel composition of processes. The abstraction of clusters is essential for expressing hierarchical parallelism.
For example, in a constellation architecture \(^1\), a cluster must be associated with a multiprocessing node, in such a way that its comprising processes are allocated to processors for shared memory parallel execution. Instead of generating MPI code, the Haskell\(_\#\) compiler could generate openMP \([67]\) code for implementing communications among processes inside the cluster, more appropriate for multiprocessors. The support for multiple hierarchies of parallelism is essential for grid computing architectures \([48]\) and is recognized as an important challenge for parallel programming languages designers \([9]\).

In MCP-Haskell\(_\#\) specification, \(pp\) and \(wp\) are clusters, units respectively associated to composed components PIPELINE and WORKPOOL, which represent skeletons. Units \(prob\_\text{def}, tally, \text{track}\) and \(\text{statistics}\) are declared as processes. The components assigned to these units are functional modules, written in Haskell.

### 2.1.8 Termination of Haskell\(_\#\) Programs

Units may be declared as repetitive or non-repetitive. Non-repetitive units perform a task and go to their final state, while repetitive ones always go back to their initial state, for executing its task once more. In HCL, a unit is declared repetitive by placing a symbol \(\ast\) after the keyword \(\text{unit}\) in its declaration. For declaring a cluster as repetitive, all units belonging to the composed component assigned to it must be repetitive. Otherwise, an error is detected and informed by HCL compiler.

A Haskell\(_\#\) program terminates whenever all non-repetitive units belonging to its main component terminates. If it has only repetitive units, it does not terminate. Repetitive units may be used to model reactive applications.

A non-stream port of a repetitive unit may be connected to a stream port of a non-repetitive unit. Each value produced in the stream is consumed in an execution of the repetitive process.

### 2.1.9 Virtual Units and The Support for Skeletons

A skeletons was defined above as a composed component where its addressed concern is partially defined or totally undefined. Now that the structure of composed components was scrutinized, it is possible to define Haskell\(_\#\) skeletons in more precise terms. In fact, the concern addressed by a composed

---

\(^1\)Constellations have been defined as clusters of multiprocessor nodes with at least sixteen processors per node \([9, 28]\).
Component is defined by the composition of concerns addressed by components assigned to its comprising units. If some unit of a component does not have a component assigned to it, it is said that the component is partially parameterized by its addressed concern. This kind of component is called a partial topological skeleton. Units not assigned to a component are called virtual units. In other skeleton-based languages, skeletons are usually total, in the sense that all units are virtual. After instantiating a partial topological skeleton, or simply a skeleton, by assigning it to a unit comprising a configuration of a component, it is possible to assign components to the virtual units of the skeleton, configuring its addressed concern.

The components Farm and Workpool are examples of total skeletons. They are used for structuring the topology of the MCP-Haskell program. They are instantiated by assigning them to units pp and wp, respectively. The replaces declaration, exemplified in lines 30 to 33 of Figure 2, takes a virtual unit from a skeleton and replaces it by another unit, such that there is an homomorphism relation from interface of the original unit to the interface of the new unit. This is formalized in Appendix B by the pair of relations \( \subseteq \) and \( \supseteq \) between interfaces. Indeed, replacing declarations are syntactic sugaring of HCL. The same effect could be obtained by creating a new unit, unifying it with the skeleton unit and assigning the appropriate component to the resulting unit. For that reason, replacing declarations are not formalized in Appendix B. This topic is revisited in the next section, where unification is introduced.

2.1.10 Operations over Virtual Units and Overlapping of Skeletons

Two operations are useful for the specification of complex topologies through the composition of skeletons: unification and factorization. Unification sub-
stitutes a collection of virtual units by a new virtual unit, obeying the network connectivity and behavioral preserving restrictions formalized in the Appendix B. In this process, ports, individual or groups, may be grouped. To group groups of ports involves to merge their sets of ports. Factorization performs inverse operation of unification. It takes a unit and splits it in a collection of units, also respecting behavioral and networking connectivity preserving restrictions. It may be needed to replicate communication pairs of interface ports of a factorized unit for preserving network connectivity. Thus, it is also necessary to configure wire functions whenever a new group of ports is resulted from a factorization. For that, HCL provides clause `adjust wire` in unification and factorization declarations.

In Figure 6, illustrative abstract examples of unification are presented, illustrating duality between these operations. A more concrete example of factorization is presented in line 28 of Figure 2, where `manager` unit from `WORKPOOL` skeleton is split up into units `dispatcher` and `collector`, dividing tasks realized by the manager. Unification does not appear directly in example of Figure 2. But replacing declarations, like discussed in the last section, is a syntactic sugaring of HCL that may be defined using unification. For instance, consider replacing declaration in line 31. It can be rewritten using the following equivalent code:

```hcl
unit track' # ITracking

: unify pp.stage[1] # particle → events, track' # (user_info, particles) → (events, totals)
    to track # ITracking (user_info, particles) → (events, totals)

: assign Tracking to track
```

Figure 6: An Illustrative Example of Unification/Factorization
Unification, and consequently replacing declarations, allows for overlapping skeletons. In this sense, units from distinct skeletons may be unified forming a new unit. Overlapping of skeletons is not supported by other skeleton-based languages. In general, only nesting composition has been addressed and cost models have been defined incorporating this feature [38]. A further step is to work on defining new cost models that incorporate the overlapping of skeletons.

### 2.1.11 Replicating Units

Another useful feature of HCL is to support replicate sub-networks from the overall network of the units described by the configuration. For that, a collection of units to be replicated and a natural number greater than one are provided. Network preserving restrictions must be observed, making necessary to replicate communication pairs of interface ports of a replicated unit, like in factorization. Wire functions must be provided to resulted groups of ports using the `adjust wire` clause.

Replication is exemplified in line 35 of Figure 2. The unit `pp` is replicated into `n` units (`pp[i]`, `0 ≤ i ≤ n − 1`), which replace worker units of WORKPOOL skeleton. Figure 7 presents an illustrative example.

### 2.1.12 Indexed Notation

The `#` configuration language supports a special kind of syntactic sugaring for allowing to declare briefly large collections of entities. The `iterator` declaration employs one or more indexes and their ranging values. Syntactic elements that appear enclosed in `[` and `]` delimiters (variation scopes) are unfolded, according to range of indexes that appears on its scope. The `#` compiler incorporates a pre-processor for unfolding indexed notation.

In Figure 2, an iterator `i` is declared, varying from 1 to `n`. The replacing
2.2 Programming Simple Components

Simple components are programmed using standard Haskell. No extensions are necessary to Haskell for gluing functional modules in the coordination medium. They are connected to units at the coordination medium by assignment declarations, where a mapping between ports of the unit interface and argument/return points of the component is defined. Arguments of a functional module are represented by the collection of arguments of its function named main, while return points are represented by the elements of the returned tuple. The general signature of main is shown in Figure 8.

The main function may return values in the IO monad [90], but the I/O concerns may be resolved at coordination level using a skeleton that implements an I/O aspect, an example of cross-cutting non-functional concern.

Figure 9 presents the Haskell code for the functional module Tracking of MCP-Haskell#. Notice the correspondence of the arguments and return points with the ports of the unit track. Functional modules are programmed in pure Haskell. There is no reference in the computation code for any
module Tracking(main) where

import Track
import Tallies
import Mcp.types

main :: User.spec.info → [(Particle,Seed)] → ([[Event]],[Int])
main user_info particle_list = let events's = map f particle_list in (events's, tally_bal event_lists)
  where
    f (particle@(x,y, e, _), sd) = (Create_source e):(track user_info particle [] sd)

Figure 9: A Functional Module from MCP-Haskell

2.3 Haskell# in the Parallel Functional Languages Context

Some authors have written papers on the evolution of parallel functional languages [57, 41, 87]. It is convenient to analyze the evolution of parallel functional programming by dividing it into two periods [57]. In first one, the decades of 1970 and 1980, parallelism was viewed as possibility to make functional programs run faster. After that period, functional programming techniques have been viewed as a promising alternative to promote higher-level parallel programming, mainly motivated by the use of skeletons implemented using higher order functions [25].

The first attempts to embed the support for parallelism in functional languages suggested the technique of evaluating function arguments in parallel, with the possibility of functions absorbing unevaluated arguments and perhaps also exploiting speculative evaluation [16]. However, the granularity of the parallelism obtained from referential transparency in pure functional languages is too fine, not yielding good performance on distributed architectures. Techniques for controlling granularity, either statically or dynamically, produced little success in practice [44, 73, 50]. Implicit parallelizing compilers face difficulties to promote good load balancing amongst processors and to keep the communication costs low. On the other hand, explicit parallelism
with annotations to control the demand of the evaluation of expressions, the
creation/termination of processes, the sequential and parallel composition of
tasks, and the mapping of these tasks onto processors yielded better results
[18, 51, 45, 76, 14, 86]. GpH adopts a semi-explicit approach, where program-
mers may annotate the code, but responsibility to decide when to evaluate
expressions in parallel is left to the compiler. Explicit approaches have the
disadvantage of cross-cutting the computation and the communication code,
not allowing to reason about these elements in isolation. Skeleton-based ap-
proaches have obtained a relative success in parallel functional programming
[26, 42, 64, 40].

The coordination paradigm [35] influenced the design of parallel func-
tional languages in 1990s, being exploited from two perspectives. In the first
one, it is used for abstracting parallel concerns from specification of computa-
tions. Eden [14], Caliban [85], and Haskell$_#$ focuses on these ideas. In the
second one, a higher-order and non-strict style of functional programming
has been seen as a convenient way for specifying the coordination amongst
tasks. SCL [26] and Delirium [61] are examples of languages that employ
the functional paradigm at coordination level, describing computations us-
ing languages from other paradigms. Haskell$_#$ have other similarities with
Eden and Caliban besides adopting the coordination paradigm and Haskell
for describing computations. They all use constructors for explicit specifi-
cation of network topologies where processes communicate through point-to-
point and unidirectional channels. Like Eden, Haskell$_#$ employs lazy lists for
interleaving computation and computation and is strict in communication.
Higher order values can not be transmitted through channels. Eden includes
functionalities for specifying dynamic topologies, contrary to Caliban and
Haskell$_#$. Static parallelism is an important premise of Haskell$_#$ design, since
it is intended to analyze Haskell$_#$ programs by translating them into Petri
nets. Also, Haskell$_#$ is oriented for high performance computing, where static
parallelism is a reasonable assumption, and not for general concurrency. In
the next paragraphs, some important distinguishable features Haskell$_#$ are
discussed.

The Adoption of a configuration based approach for coordination.
Configuration languages [53], integrated to a lazy functional language like
Haskell, allows a complete separation between parallelism and computational
programming dimensions. No extensions are required to Haskell for program-
ming at computational level. Haskell and the HCL are orthogonal. *Eden* and *Caliban*, examples of embedded coordination languages, extend Haskell syntax with primitives for “gluing” processes to the coordination medium. GpH tries to separate parallel coordination code by using *evaluation strategies* [88]. Evaluation strategies is an interesting idea, but after inspecting some GpH programs that uses them, we noticed that a complete and transparent separation of the parallelism and the computation is very difficult to obtain. This is even worse when programmers want to reach peak performance of applications at any cost. The experience with Haskell#, and other parallel functional languages, has shown that a really transparent separation makes easier to parallelize existing Haskell programs. This increases opportunities for the reuse of code and allows independent specification and development of functional modules and coordination code, reducing programming efforts and costs. The ability of composing programs from parts using the configuration approach also makes Haskell# more suitable for programming large scale high-performance applications than other parallel functional languages [33, 27]. Programmers are forced to adopt a *coarse grained* view of parallelism that is convenient for clusters and grids.

**The Modelling of parallel architectures.** Developing general techniques for freeing programmers from making decisions on the allocation of processes to processing nodes of a parallel architecture is an old challenge to the parallel programming community. However, this problem is hard to be treated in general. Existing mechanisms for this purpose, either dynamic or static, apply efficiently to restricted instances of the general problem and some of them are based on heuristics. With the advent of grids, cluster of heterogeneous nodes, constellations, etc, it is not expected that a unified approach, covering all realistic cases, may appear. Because of that, Haskell# follows a *static and explicit* approach for process allocation, as in Caliban. Eden and GpH, on the other hand, let allocation decisions to the compiler. In Haskell#, it is possible to model both processes needs for optimal execution and architecture characteristics by using partial topological skeletons for treating allocation as an *aspect*. Each skeleton may be implemented using specific allocation policies convenient for different architectures.

**The analysis of formal properties using Petri nets.** The support for proving and analysing of formal properties of parallel programs by using
Petri nets is one of the most important premisses that guided the design of Haskell#. A compiler that translates HCL configurations into INA [80], a Petri net analysis tool, was developed [55]. In [23], a new translation schema incorporating some extensions to the original HCL was presented. Recently, a new translation schema has appeared and we are working on a new compiler for translating Haskell# programs into PNML [91], a format supported by many Petri net analysis tools, and SPNL [36], for analysing the performance of Haskell# programs by using stochastic Petri nets. TimeNET [92] will be used for this purpose. Other parallel functional languages do not support formal analysis of parallel programs.

**Simple and portable implementations**. Unlike other parallel functional languages, it was not necessary to modify or extend the run-time system of GHC for implementing Haskell#. Indeed, any Haskell compiler could be used in alternative to GHC, with all optimizations enabled. Haskell# programs take advantage of the evolution of compilation techniques with little efforts. Eden, for example, modifies GHC compiler and disables some of its optimizations [69]. Modifications to the run-time system of the Haskell compiler makes difficult to adapt the parallel language extension to new versions of the compiler. In Haskell#, internal changes to the GHC run-time system do not require modifications to the code generated by the Haskell# compiler. Only if the interface of some used library is changed, minor modifications are necessary. GpH and Eden developers should also carefully analyze the effects of modifications to their parallel run-time system.

**Efficiency**. Potentially, Haskell# compiler may generate efficient MPI code without using advanced compilation techniques for parallel code. This is due to the direct correspondence of HCL constructors to MPI primitives and the use of skeletons to abstract specific interaction patterns. Languages that use higher-level constructors, in the sense that parallelism is transparent or implicit, have difficulties on promoting the generation of MPI code able to take advantage of peak performance in cluster architectures and, mainly, in grid computing environments.
3 Motivating Examples

This section presents Haskell\# implementations for three applications recently used for benchmarking the parallel functional languages Eden, GpH and PMLS: Matrix Multiplication, LinSolv and Ray Tracer [59]. A Haskell\# implementation for a sub-set of NPB (NAS Parallel Benchmarks) [2] is also presented. These applications will be used in Section 5 for performance evaluation of the current Haskell\# implementation, presented in Section 4.

3.1 Matrix Multiplication

Given two square matrices $A, B \in Z^{n \times n}$, $n \in N$, a matrix $C \in Z^{n \times n}$ is calculated, such that $C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j}$.

A trivial, fine-grained, parallel solution requires $n \times n$ processors. Each processor computes an element $C_{i,j}$, from scalar product of row $i$ of $A$ and column $j$ of $B$. This solution is obviously impractical, since large matrices are common in real applications, requiring a number of processors not supplied by contemporary parallel architectures. Three approaches are commonly used in order to aggregate computations for increasing granularity [59]:

- **Row Clustering**: each process computes a set of rows of $C$. For that, the process needs the corresponding set of rows of $A$ and all matrix $B$;
Figure 11: Haskell\# Configuration of Block Clustering using WORKPOOL and FARM

- **Block Clustering**: each process computes a block of the resulting matrix $C$. For that, the corresponding rows of $A$ and columns of $B$ are needed;

- **Gentleman’s algorithm**: the processes are organized in a torus (circular mesh) for performing a systolic computation \[78\]. Each process computes a block in $C$. At initial state, the corresponding blocks in $A$ and $B$ are arranged across processes. Then, they execute $k$ steps, where $k$ is the number of rows and columns of processes. At each step, a process sends the blocks from $A$ and $B$ that it contains to its left and down neighbors, and receive new blocks from right and top neighbors. A local computation is performed and the resulting matrix is accumulated.

The above solutions differ on the number and size of messages exchanged. In Haskell\# programs, composition of skeletons may be used to describe topologies for the solutions. Firstly, consider implementations of row and block clustering using WORKPOOL skeleton, where a manager process distributes rows or blocks, respectively, as jobs to a collection of worker processes, on demand of their availability. Once a worker finishes a job, it sends
{- In FILE MatMultTorus.hcl -}
configuration MatMult\<N\> where

iterator i range [1,N*N]

use Skeletons (Torus, Farm)
use ReadMatrix, MatrixMult, WriteMatrix
import MatrixMult,WPs(splitM,combineM)

unit farm\_a; assign Farm\<N\^2,splitM,combineM\> to farm\_a
unit farm\_b; assign Farm\<N\^2,splitM,combineM\> to farm\_b
unit torus; assign Torus\<N\> to torus

// unify farm\_a.worker[i] # a -> c,
// farm\_b.worker[i] # b -> c,
// torus.cell[i/N][i%N] # (as\_l, bs\_l) -> (as\_r, bs\_r)
// to cell[i/N][i%N] # (a::Matrix, b::Matrix, as\_l:: [Matrix], bs\_l:: [Matrix])
// -> (c::Matrix, as\_r:: [Matrix], bs\_r:: [Matrix]) /=

unify farm\_a.collector # c -> (), farm\_b.collector # c -> () to collector # c::Matrix -> ()

assign ReadMatrix \<N\> to farm\_a.distributor # () -> a
assign ReadMatrix \<N\> to b to farm\_b.distributor # () -> b

// assign MatrixMult to cell[i/N][i%N] /
assign WriteMatrix to collector

Figure 12: Systolic Matrix Multiplication using a Torus (HCL Code)

its result back to the manager and stay available for receiving another job. This technique is suitable when the number of jobs exceed the number of processors available. Load balancing is automatically achieved in architectures where processor workload or performance may vary. Because of that, it has been widely used in grid computations [34]. The unit manager in the WORKPOOL skeleton in Haskell\# has two groups of ports of kind any: one for sending jobs to workers and another for receiving results from them. Workers receive jobs from their input ports and send results through their output ports.

Row and block clustering may also be implemented using Farm skeleton. Now, a master process sends a job to each slave process. Ideally, jobs have similar workload. After completing a job, slaves send the result to their master and finish. The master combines the solutions received from all slaves. This approach may reduce significantly the number of messages exchanged and minimizes the communication overheads by using underlying collective communication primitives. In fact, the Farm skeleton is defined by overlapping of Gather and Scatter skeletons. Farm employs wire functions for distributing and combining values sent to and received from slave processes. For achieving better load balancing, processors must be homogeneous. This is a reasonable assumption to be made in cluster architectures, but not in grid ones.
module MatMult_Toroidal where

import MatrixTypes
import List

main = mult'

mult' nc nr sm1 sm2 sm1s sm2s = (result, toRight, toDown)
  where
toRight = take (nc-1) (sm1:sm1s)
toDown = take (nr-1) (sm2':sm2s)
    sm2' = transpose sm2
    sms = zipWith multMatricesTr (sm1:sm1s) (sm2':sm2s)
result = foldl1' addMatrices sms

addMatrices :: Matrix -> Matrix -> Matrix
addMatrices m1 m2 = zipWith addVectors m1 m2

where addVectors :: Vector -> Vector -> Vector
    addVectors v1 v2 = zipWith (+) v1 v2

multMatricesTr :: Matrix -> Matrix -> Matrix
multMatricesTr m1 m2 = [[prodEscalar row col | col <- m2] | row <- m1]

foldl1' :: (a->a->a) -> [a] -> a
foldl1' f (x:xs) = foldl1 f x xs

foldl' :: (a->a) -> a -> [a] -> a
foldl' f a [] = a
foldl' f a (x:xs) = foldl' f (f a x) xs

prodEscalar :: Vector -> Vector -> MyInteger
prodEscalar v1 v2 = sum (zipWith (*) v1 v2)

module LS_homSolv(main) where

import Matrix
import LUDecompMatrix (det, homsolve)
import qualified Matrix (determinant)
import ModArithm

main :: (SqMatrix Integer, Vector Integer) -> [Integer] -> [[Integer]]
main (a,b) = gen xList
  where
gen xList :: [Integer] -> [[Integer]]
gen xList ps = map get homSolv ps
get homSolv = Integer -> [Integer]
    get homSolv p = 
      let b0 = vecHom p b
          a0 = matHom p a
          modDet = modDet p a0
          pmx = homsolve0 p a0 b0
          ((iLo,jLo),(iHi,jHi)) = matBounds a
      in 
        (p : modDet : if modDet == 0 then [] else pmx)
slow determinant :: SqMatrix Integer -> Integer
slow determinant = Matrix.determinant

determinant :: Integer -> SqMatrix Integer -> Integer
    determinant = det

homsolve0 :: Integer -> SqMatrix Integer -> Vector Integer
    homsolve0 p a0 b0 = vecCont v
    where
      v = homsolve p a0 b0

Figure 13: Functional Modules of Matrix Multiplication and LinSolv

Figure 11 presents the Haskell configuration codes for block clustering using WORKPOOL and FARM skeletons. Two matrices are distributed, thus it is necessary to overlap two instances of both skeletons, as illustrated in Figure 10. The units readA, readB and writeC are clustered to implement the manager process. The implementation of row clustering makes use of identical topological description. Differences are on port types and implementation of computations. This evidences the importance of reuse and composition in Haskell programming.

The Gentleman’s algorithm is implemented by overlapping two instances of the FARM skeleton, one for each input matrix, with a TORNUS skeleton, as in Figure 10. The TORNUS describes the interaction pattern among slave processes from the overlapped FARMs. The HCL code for this arrangement is presented in Figure 12.

Haskell components that implement the solutions above have the same names and interfaces. Only internal details, concerning the parallelism strat-
egy adopted, varies. Thus, they can be used interchangeably in an application by nesting composition. The Haskell# visual programming environment allows several component versions to co-exist. The programmer may choose the appropriate version, depending on the target parallel architecture. For instance, implementing matrix multiplication using FARM may be more efficient in clusters. In grids, a WORKPOOL may prove more suitable. In supercomputers where processors are organized in a torus, the toroidal solution may be the best choice.

### 3.2 LinSolv

Given a matrix $A \in \mathbb{Z}^{n \times n}$ and a vector $b \in \mathbb{Z}^n$, $n \in \mathbb{N}$, find an exact solution to the linear system of equations of the form $Ax = b$.

The solution described here is exact and operates over arbitrary precision integers. A multiple homomorphic image approach is adopted [54], consisting of three stages [59]:

1. map the input data into several homomorphic images. The domain of homomorphic images is $\mathbb{Z}$ modulo $p (\mathbb{Z}_p)$, where $p$ is a prime number;
2. compute the solution in each of these images, using LU-decomposition followed by forward and backward substitution;

3. combine the results of all images into a result in the original domain, using a fold-based CRA (Chinese Remainder Algorithm) \[58\].

The parallel strategy implemented in Haskell\# is based on Eden and GpH versions \[60\]. A manager process distributes computations of homomorphic solutions as jobs to a collection of worker processes. The skeleton WORKPOOL was adopted to distribute prime numbers to workers and to collect computed homomorphic solutions. The BCAST collective communication skeleton is used for distributing working data (A and b) to the workers. The Haskell\# configuration code that implements this arrangement is presented in Figure 15. A composed component LS_MANAGER is configured for aggregating computations of functional modules LS_INPUT (obtains input data A and b), LS_PRIMES (computes the list of primes for calculating homomorphic solutions), LS_CRA (aggregates homomorphic solutions using Chinese Remainder Algorithm), and LS_OUTPUT (outputs result x). In
composed component LINSOLV, the main component, a cluster is created by assigning LS_MANAGER to unit ls_manager, which is configured in such a way that it makes the role of root unit in BCAST skeleton and manager of WORKPOOL skeleton. The functional module LS_HOMSOL implements computation of a homomorphic solution for a given prime number. It is assigned to units ls_worker[i], for 0 ≤ i ≤ N − 1, obtained by unification of worker units of WORKPOOL and peer units of BCAST. Notice that these skeletons are overlapped. The cluster ls_manager might be placed onto a multiprocessor node, in such a way that processes input, primes, cra and output could execute concurrently. Figure 14 illustrates topological specification of LinSolv. Figure 13 shows examples of functional modules of Matrix Multiplication and LinSolv.

### 3.3 Ray Tracer

Given a collections of objects in the three dimensional space, calculate the corresponding two dimensional image. All rays in a window (for each pixel in the grid) are traced and their intersections with objects are computed. The colour of an intersection point is computed based on the strength of the ray
A data parallel solution is trivial, since rays can be traced independently for each pixel. In Haskell implementation, a direct mapping of the image lines to \( N \) parallel processes, assuming one at each processor, is employed. Each process receives the same number of lines to compute. This solution yields load balancing in homogeneous clusters. The HCL for ray tracer is presented in Figure 17 and its topology is described in Figure 16. It is implemented by overlapping three skeletons: \textsc{Bcast}, \textsc{Gather} and \textsc{Scatter}. The root units of these skeletons are unified to form the manager unit, responsible for distributing and collecting work among worker units, obtained by overlapping their peer units. The manager also acts as a worker. Distribution and collection are specified by wire functions. The \textsc{Bcast} skeleton disseminates the world scene to workers. \textsc{Scatter} and \textsc{Gather} are used to distribute jobs and collect the results from the workers.
3.4 NAS Parallel Benchmarks

This section presents the Haskell implementations for a sub-set of NPB (NAS Parallel Benchmarks) [2], a package comprising eight programs, specified in NASA Research Center at Ames, USA, intended to benchmark the performance of parallel computing architectures for execution of the NAS (Numerical Aerodynamic Simulation) programs. NPB programs implemented in Haskell are:

- **EP (Embarrassingly Parallel)** generates pairs of Gaussian deviates according to a specified scheme and tabulates the number of pairs in successive square annuli. It was developed to estimate the upper achievable limit for floating point performance in a parallel architecture;
- **IS (Integer Sorting)** performs parallel sorting of $N$ keys using bucket sort algorithm. Keys are generated using a sequential algorithm described in [3] and must be uniformly distributed;
- **CG (Conjugate Gradient)** implements a solution to an unstructured sparse linear system, based on conjugate gradient method. The inverse power method is used to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of non zeros;
- **LU (LU factorization)** uses symmetric successive over-relaxation (SSOR) procedure to solve a block lower triangular-block upper triangular system of equations resulting from an unfactored implicit finite-difference discretization of the Navier-Stokes equations in three dimensions;

NPB programs exercise the expressiveness of HCL for describing SPMD programs and for translating MPI programs into Haskell. LU gave us an important insight on how to facilitate programming of applications where processes have a large number of input and output ports. CG and IS help on evaluating the performance of collective communication skeletons.

3.4.1 The Embarassingly Parallel (EP) Kernel

The HCL code of EP is presented in Appendix C.1. It declares $n$ units, named $ep\_unit[i]$, for $1 \leq i \leq n$. The interface class that describes the behavior of these units, $IEP$, is formed by the composition of three instances
of \textit{IAllReduce} interface class, called \textit{sx}, \textit{sy} and \textit{q}. The definition of channels is specified by overlapping three instances of the \texttt{AllReduce} skeleton. For that, clusters \texttt{sx\_comm}, \texttt{sy\_comm}, and \texttt{q\_comm} are associated with \texttt{AllReduce} component and their virtual units are unified. The HCL compiler uses the topological information provided by \texttt{AllReduce} skeleton and generates code that uses the \texttt{MPI\_AllReduce} primitive of MPI.

### 3.4.2 The Integer Sort (IS) Kernel

The HCL code of IS is shown in Appendix C.2. It declares a network of \textit{n} units, named \textit{is\_unit[i]}, for \(1 \leq i \leq n\). The interface class for describing the behavior of IS units, called \textit{IIS}, is a composition of interfaces \textit{IAllReduce}.
duce, IAllToAllv and IRShif. A cyclic pattern of communication (repeat combinator) now appears, due to presence of stream ports on specification of IIS.

IS network topology is defined by overlapping skeletons ALLREDUCE and ALLTOALIV, for collective communication, and RSHIFT, which performs a data shift right amongst processes. Cluster units bs_comm, kb_comm and k_shift are assigned to them, respectively, and their virtual units are unified. The interface components bs, kb, rshift of IIS indicate which ports of IS units participate in the skeleton instances, respectively.

### 3.4.3 The Conjugate Gradient (CG Kernel)

The original topology of CG, specified in FORTRAN/MPI, imposes that the number of processes, organized in a rectangular mesh, is a power of two. The version of CG in Haskell is less restrictive. The programmer must provide parameters dim (the number of mesh rows), and col_factor (the number of mesh columns is obtained by multiplying it to dim). Any number of units may be configured using this approach, but different configurations may result in different performance. The programmer should adequate the parameters values to the features of the execution environment. CG units cg_unit[i][j], for 1 ≤ i ≤ dim and 1 ≤ j ≤ dim * col_factor. The HCL code of CG is presented in Appendix C.3.

The interface class that describes the behavior of CG units, ICG, is a composition of interface classes IAllReduce (rho, aux, rnorm, norm_temp_1 and norm_temp_2) and ITranspose (q and r). CG topology is defined by overlapping ALLREDUCE and TRANPOSE skeletons. The former is used for data exchange during parallel scalar products at mesh rows, and the latter for data exchange in parallel matrix multiplications, whenever a transpose operation is performed on data stored in processors. In MPI original code,
several calls to MPI\texttt{Irecv} primitive are needed to perform these operations, making difficult to understand the structure of the topology without a careful analysis of the parameters of the problem.

Five clusters are needed for each row of processes: $\rho_{\text{comm}}[i]$, $\alpha_{\text{comm}}[i]$, $\rho_{\text{norm\_comm}}[i]$, $\alpha_{\text{norm\_temp\_1}}[i]$, and $\alpha_{\text{norm\_temp\_2}}[i]$, $1 \leq i \leq \text{rows}$. The \texttt{ALLREDUCE} component is assigned to them. The \texttt{TRANSPOSE} component is assigned to the other two clusters, $q_{\text{comm}}$ and $r_{\text{comm}}$, encompassing all processes in the network. Their units are unified producing the final Haskell\# topology of CG.

The Haskell\# configuration code of \texttt{TRANSPOSE} is presented in Appendix C.3.1. It organizes virtual units according to parameters $\text{dim}$ and $\text{col\_factor}$, supplied by CG configuration. Firstly, a square mesh of units with dimension $\text{dim}$ is assembled. The ports are connected to transpose data amongst processors using appropriate wire functions applied on groups of ports. These units are factorized in $\text{col\_factor}$ units, resulting in a square mesh with $\text{dim}$ rows and $\text{dim} \times \text{col\_factor}$ columns. The diagram in Figure 20 illustrates the factorization process involved in \texttt{TRANSPOSE} specification. In order to make it easier to understand, only channels connected to $u[1][1]$ ports are shown. They are replicated according to factorization rules.

### 3.4.4 The LU Factorization (LU Simulated Application)

The HCL code of LU is presented in Appendix C.4. LU organizes $n$ processes, where $n$ is a power of two, in a grid. It employs the wavefront method [6] in parallel computation. It differs from other NPB programs because communication is performed by small messages of approximately 40 bytes. Another particularity of LU is the great number of communication ports in units (thirty input ports and thirty output ports). Skeletons \texttt{EXCHANGE\_1B}, \texttt{EXCHANGE\_3B}, \texttt{EXCHANGE\_4}, \texttt{EXCHANGE\_5}, and \texttt{EXCHANGE\_6} describes communication topologies in several communication phases during execution, using the wavefront method. The same nomenclature employed in the original LU versions are used here to make easier to compare the two approaches. In these skeletons, there are several interfaces for virtual units that comprise them. Their specification vary according to their position in the grid. Interface generalization is useful in such cases, avoiding classes of units to be treated individually in the configuration.
4 Implementation

Haskell\# may be implemented on top of a message passing library and a sequential Haskell compiler, without any modifications or extensions to any of them. MPI 1.1 and GHC (Glasgow Haskell Compiler) are currently used, respectively. MPI is now considered the most efficient message passing library for clusters, providing standard bindings for C and Fortran. Recently, MPI versions for grid computing have appeared [49]. GHC is now considered state-of-the-art techniques for the compilation of lazy functional programs. It supports FFI (Foreign Function Interface) [24] to make direct calls to MPI routines from Haskell programs. The use of an efficient sequential Haskell compiler has important impact on performance of Haskell\# programs, since Haskell\# programs assumes medium and coarse grained parallelism, where most of time is spent in sequential mode of execution. Haskell\# implementations are easily portable to new MPI and GHC versions. Indeed, it is possible to replace GHC with any Haskell compiler that supports FFI. All optimizations and extensions provided by the Haskell compiler may be enabled. This is an important feature of Haskell\#, since other parallel functional languages built on top of GHC need to modify its run-time system. The current Haskell\# implementation has already been tested on top of LAM-MPI 6.5.9 [17], MPICH 1.2.5.2 [37] and GHC versions 6.01 and 6.2 in clusters equipped with RedHat Linux 8.0 and 9.0.
4.1 An Overview of the Haskell\# Compilation Process

The Haskell\# compiler has been entirely programmed in Haskell, using Alex 2.0 [30] and Happy 1.13 [62] for parsing. It is divided into two modules: front-end and back-end. The compilation process is illustrated in Figure 21. The front-end module parses all components of a Haskell\# program, by traversing its tree of components, from application component to simple components. A flat representation of the processes network is generated. Relevant topological information, obtained from the use of skeletons, that could guide the back-end for the generation of optimized code is stored. The flat code is currently represented as an algebraic data type in Haskell, but it is intended to implement it in XML (Extended Markup Language), allowing to use it as an intermediate language for interfacing tools for the analysis performance and formal properties in the programming environment under development.

The back-end uses flat code and topological information for generating a wrapper module for each process and for inferring the mapping of processes onto processors of the target architecture. A wrapper module is a Haskell program that controls the execution of a process. The wrapper modules and the functional modules are compiled using GHC. The mapper is a program that copies executable files onto the target machine where it will execute, based on the mapping of processes onto processors inferred by the back-end.

4.2 Wrapper Modules

In Figure 22, the structure of a wrapper module is illustrated. A wrapper makes a call to the main function of the functional module associated to with the process. The values produced at return points \((r_i, 1 \leq i \leq k)\) are copied concurrently to channel variables\(^2\) \((chan_{r_j})\), using functions send\_stream and send\_atom, depending on the nature of the associated output port. The arguments provided to the main function \((a_j, 1 \leq j \leq n)\) may also be obtained from channel variables (ON\_DEMAND \(chan_{r_j}\)) or directly (FORCED action\(_j\)), on demand of evaluation of return points. The function perform\_actions controls the completion of the communication operations, according to a guide automaton that recognizes the behavior specified in the process interface. Whenever an output port must be activated, perform\_actions evaluates perform\_communication, which reads a value from the

\(^2\)Type Chan \(t\) from Concurrent Haskell [74].
module Main(main) where

import System(getArgs)
import Concurrent(forkIO, Chan, newChan, newQSemN, waitQSemN, signalQSemN)
import HHashSupport

main :: IO ()
main =
do argv <- getArgs
   argc <- (return.length) argv
   give_args [] argv
   -- MPI initialization
   args [] argv (mpi init BUFFER_SIZE argc)
a1 chan :: Chan (Comm [Channel Type]) <- newChan
   -- Initializing channel variables for arguments
   a2 chan :: Chan (Comm [Channel Type]) <- newChan
   ...
a_n chan :: Chan (Comm [Channel Type]) <- newChan
   ...
r1 chan :: Chan (Comm [Channel Type]) <- newChan
   -- Initializing channel variables for return points
   r2 chan :: Chan (Comm [Channel Type]) <- newChan
   ...
r_k chan :: Chan (Comm [Channel Type]) <- newChan

for each p, an individual port or group of ports involved in a collective operation:
   p <- [mpi register _port . . . | mpi register _peer . . .]
   let comm p = [SingleIPort | SingleOPort | GroupIPort | GroupOPort | Bcast | Gather | Scatter |
                  Scatterv | Allgather | Allgatherv | Allreduce | Alltoall | Alltoallv | ReduceScatter | Scan] p . . .

   caut <- [code to setup guide automata]

   control _automata _init cast
   sync <- newQSemN 0

   forkIO (perform_actions _ | signalQSemN sem)
   let a1 = [recv_stream | recv_atom | [ON_DEMAND action1 | FORCED chan_a1]
             a2 = [recv_stream | recv_atom | [ON_DEMAND action2 | FORCED chan_a2]
                 ...
             a_n = [recv_stream | recv_atom | [ON_DEMAND action_n | FORCED chan_a_n]

   (r1, r2, . . . r_k) = _Functional Module (main a1 a2 . . . a_n)

   forkIO ([send_stream | send_atom] r1 chan r1 _ | signalQSemN sync)
   forkIO ([send_stream | send_atom] r2 chan r2 _ | signalQSemN sync)
   ...

   forkIO ([send_stream | send_atom] r_k chan r_k _ | signalQSemN sync)

   waitQSemN (k+1) sem

   mpi finalize
corresponding return point and sends it through the active port. For input ports, perform_communication may be called inside recv_stream or recv_atom functions, when an argument value is demanded. In this case, the operation is validated by the guide automaton and a channel variable is not necessary. However, in some collective communication operations, when a process sends and then receives a value (the root process in a broadcast, for example), it is needed to write and read, in a single call to perform_communication, channel variables associated to a return point and to an argument, respectively. This is a situation where a channel variable is necessary for an argument. The Haskell compiler forces evaluation of the input ports inside perform_actions whenever it may infer that an input port must be strictly activated before the activation of some output port. This is typical when the alt (choice) constructor does not occur in process behavior specification. Figure 23 illustrates the use of channel variables.

Since processes spend some time with synchronization, concurrent evaluation of perform_action and exit points, using send_stream and send_atom, allow the overlapping of computations when a process is executing perform_communication. In multiprocessors and super scalar processors, which may execute instructions in parallel and speculate about their execution, performance might be improved.

4.3 Guide automaton: Controlling Activation of Ports

A guide automaton is an abstract data type, implemented in C, used for controlling and validating the activation order of ports in execution of Haskell programs. It might be algebraically described by a tuple of the following form:

\[ C = (\Pi, Q, T, \varphi_0, \varphi_1, \rho, F, S, \sigma, \pi, \gamma, \kappa) \]
where:

- $\Pi$ is a set of port identifiers that forms the alphabet of the guide automata;
- $Q$ is a finite set of states;
- $T$ is a finite set of transitions;
- $\varphi_0 : T \to Q$ maps each transition to its origin state;
- $\varphi_1 : T \to Q$ maps each transition to its target state;
- $\rho : T \to \Pi$ labels each transition with a port identifier;
- $F \subseteq Q$ is a set of final states;
- $S$ is a finite set of symbols, representing semaphores;
- $\sigma : Q \to 2^{S \times \mathbb{N}}$ associates states to semaphore updates. For instance, consider a semaphore $s \in S$. If $(s, n) \in \sigma(q)$ then the value of $s$ must be incremented by $n$ when entering state $q$;
- $\pi : Q \to \{\text{forward, choice, fork, join}\}$ gives the kinds of the states;
- $\gamma : Q \to \{\text{True, False}\}$ maps choice states to an expression (termination condition of a repeat combinator) that evaluates to True or False;
- $\kappa : Q \to 2^Q \times 2^Q$ associates a state $q$, to a pair of set of states $(Q', Q'')$, whose meaning depends on $\pi(q)$ (see the next paragraph).
States and transitions are represented as natural numbers. The initial state is 0 (zero). Let \( q \) be the current state of a guide automaton. The function \( \text{perform\_actions} \) looks up \( \kappa(q) \) in order to choose the next communication operation to be performed. For instance, consider \( \kappa(q) = (Q^l, Q^r) \). There must be a path from state \( q \) to each state in \( Q^l \cup Q^r \). If \( \pi(q) = \text{forward} \), \( Q^r = \emptyset \) and \( Q^l \) determines the forward states of \( q \). Among them, the goal states are chosen. For that, let us consider a set of transitions \( T^r = \{ t \mid \varphi_1(t) = q' \land q' \in Q^l \land t \text{ is in a path from } q \text{ to } q' \} \). Port \( p \) is chosen from ports \( \{ p \mid t \in T^r \land \rho(t) = p \} \), among those whose communication pairs are active at that instant (ready for communication). Forward states \( q \), such that, for some \( t \in T^r \), \( \varphi_1(t) = q \) and \( \rho(t) = p \), are goal states. Choices appear only in the implementation of occurrences of the \texttt{alt} constructor. The port \( p \) is activated. If \( p \) is an output port (default case), it may cause the implicit activation of input ports, in \texttt{recv\_stream} or \texttt{recv\_atom} function calls, before completing communication. After any port activation in \texttt{perform\_communication}, the \texttt{advance\_automata} function is called for updating the current automata state, validating the operation, by raising an error whenever there is no transition from the current state labelled with the activated port, and updated semaphores. After the activation of \( p \), the guide automaton must be in one of the goal states. Otherwise, the operation is invalid. If \( \pi(q) = \text{choice} \), \( \gamma(q) \) must be evaluated (termination condition of a repetition). If \( \gamma(q) \) is true, the set of forward states of \( q \) is \( Q^l \), otherwise it is \( Q^r \). Choice states are used in the implementation of occurrences of \texttt{repeat} and \texttt{if} combinators. If \( \pi(q) = \text{fork} \), \( Q^r = \emptyset \) and \( \forall t : \varphi_0(t) = q : \varphi_1(t) \in Q^l \land \rho(t) = \bot \). When a fork state is reached, threads are forked for executing communication actions starting from the states in \( Q^l \). All threads must reach the same join state, where they finalize and resume execution from that state. If \( \pi(q) = \text{join} \), \( Q^l = \emptyset \) and \( Q^r = \emptyset \). Fork and join states are used to implement occurrences of \texttt{par} combinator. If there is no forward state from current state and it is a final state, \texttt{perform\_actions} finalizes.

Semaphores are updated in calls to \texttt{advance\_automata}. The function \( \sigma \) is used to update their values according to the new current state. A semaphore must have more than one value at a time. During execution, it must be guaranteed that all semaphores must be at least one positive value. Otherwise, an error is informed. Negative values are discarded. Semaphores only exist for validating non-regular patterns of communication that may be described by labelled Petri nets [89]. However, in general, regular patterns of communication are sufficient to describe behavior of most of high-performance
parallel programs [65, 70]. Thus, overhead due to semaphore updating might be avoided for parallel programs where peak performance is critical.

### 4.4 Implementing Communication Operations

There are two kinds of communication operations in Haskell\#: point-to-point and collective. The former is implemented through simultaneous activation of channel’s communication pairs. MPI tags, in message envelopes, represent communication channels in calls to point-to-point primitives. The later is implemented using MPI support for dynamic configuration of communication groups and contexts and MPI collective communication primitives. Groups of ports involved in a collective communication are called communication peers. Each communication pair is configured using the function \texttt{mpi\_register\_pair}, while communication peers are configured in a single call to \texttt{mpi\_register\_peers}. These functions are implemented in C, being called from Haskell code through FFI. Their arguments, detailed in Table 1, set up parameters for completion of communication operations over involved ports during execution. A communication handle, an integer number, is returned and bound to a variable for allowing to access pair/peers information whenever necessary.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>pair</th>
<th>peer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>⋆</td>
<td>⋆</td>
<td>Specifies if a port is for input or output</td>
</tr>
<tr>
<td>Source/Target rank</td>
<td>⋆</td>
<td></td>
<td>Rank of the process that owns its comm. pair</td>
</tr>
<tr>
<td>Channel tag</td>
<td>⋆</td>
<td>⋆</td>
<td>A number that identifies individually a channel</td>
</tr>
<tr>
<td>Collective Op. Type</td>
<td>⋆</td>
<td>⋆</td>
<td>Kind of the collective communication operation</td>
</tr>
<tr>
<td>Number of Processes</td>
<td>⋆</td>
<td></td>
<td>Number of processes in the collective operation</td>
</tr>
<tr>
<td>Processes in group</td>
<td>⋆</td>
<td></td>
<td>Ranks of processes in the collective operation</td>
</tr>
<tr>
<td>Buffer Size</td>
<td>⋆</td>
<td>⋆</td>
<td>Buffer used for storing data to be transmitted</td>
</tr>
<tr>
<td>Data Type</td>
<td>⋆</td>
<td></td>
<td>MPI data type (used in a reduce operations)</td>
</tr>
<tr>
<td>Reduce Operation</td>
<td>⋆</td>
<td></td>
<td>MPI operation (used in a reduce operations)</td>
</tr>
<tr>
<td>Is Probed Flag</td>
<td>⋆</td>
<td></td>
<td>Flag indicating if a port belongs to a choice group</td>
</tr>
<tr>
<td>Pair is Probed Flag</td>
<td>⋆</td>
<td></td>
<td>Flag indicating if the communication pair of a port belongs to a choice group.</td>
</tr>
</tbody>
</table>

The polymorphic and higher-order function \texttt{perform\_communication} has one argument, a value from the algebraic data type \texttt{PortInfo t u v}, whose constructors identifies the kind of communication operation to be performed: \texttt{SingleIPort, SingleOPort, GroupIPort, GroupOPort} (point-to-point
communication), Bcast, Gather, Scatter, Scatterv, Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Reduce_Scatter, Scan (collective communication). The PortInfo’s fields encapsulate necessary information for completion of communication operations: communication handle, port type (choice or combine), wire functions, and channel variables. The type variables \( t, u \) and \( v \) are used for generalization of channel variables and wire functions types.

The MPI point-to-point communication primitive used for completion of communication over an output individual port (SingleOPort) depends on the communication mode of the channel where it is linked: buffered (MPI_Bsend), synchronous (MPI_Ssend) or ready (MPI_Rsend). For groups of output ports of kind ALL, the corresponding asynchronous MPI sending primitives (MPI_Ibsend, MPI_Issend and MPI_Irsend) are used for initiating the communication on each port belonging to the group. Then, a call to MPI_Waitall waits for the completion of all the returned request. Similarly, a call to MPI_Recv implements the communication on individual input ports, while MPI_Irecv (asynchronous) and MPI_Waitall, implements groups of input ports of kind ALL. Groups of ports of kind ANY are implemented using the channel probing protocol, which allows the verification of the status of activation of communication pairs.

Transmitting streams and atom values. In Haskell\#, a value of type \( t \) is transmitted as a value of algebraic type \( \text{Comm}\ t \), whose Haskell representation is depicted below:

\[
\text{data}\ \text{Comm}\ t = \text{ATOM}\ \{\text{data} :: t\} \mid \text{MID}\ \{\text{data} :: t\} \mid \text{END}\ \{\text{depth} :: \text{Int}\}
\]

The \text{ATOM} constructor encapsulates atomically transmitted values, while streamed ones are encapsulated using \text{MID} and \text{END} constructors. The integer value in the END field represents the depth of a finalized stream. For instance, consider a stream port \( p \) of type (Int,Int) and nesting factor 2 \((p^{**}::(\text{Int,Int}))\). The lazy list associated to the port must be of type \([[(\text{Int,Int})]]\). Consider the lazy list \([[(\text{1,2}),(\text{3,4})],[\text{5,6}]]\), \([[(\text{7,8}),(\text{9,0}),(\text{1,2})]],\ [[\text{3,4}]],\ [[\text{5,6}],[\text{7,8}]]\]. The list of values effectively transmitted through the stream port \( p \) at each activation is \([\text{MID}\ (\text{1,2}),\text{MID}\ (\text{3,4}),\ \text{END}\ 3,\ \text{MID}\ (\text{5,6}),\ \text{END}\ 3,\ \text{END}\ 2,\ \text{MID}\ (\text{7,8}),\ \text{MID}\ (\text{9,0}),\text{MID}\ (\text{1,2}),\ \text{END}\ 3,\ \text{END}\ 2,\ \text{END}\ 3,\ \text{MID}\ (\text{3,4}),\ \text{END}\ 3,\ \text{MID}\ (\text{5,6}),\ \text{MID}\ (\text{7,8}),\ \text{END}\ 3,\ \text{END}\ 2,\ \text{END}\ 1]\). Whenever possible, stream communication is implemented using MPI persistent
communication objects, for minimizing communication overhead.

**Marshalling Haskell Values to C Buffers.** In order to transmit Haskell values using MPI primitives, they must be marshalled onto C contiguous buffers. For that, the *Storable* class, from FFI, is employed. Default *Storable* instances are provided for basic data types. User defined data types should be instantiated for this class. The Haskell*#* compiler traverses Haskell modules of the Haskell*#* program for finding user defined type values that must be instantiated for the *Storable* class. Structured data types, such as lists, arrays, tuples and algebraic data types must be packed and unpacked element by element. This could result in a considerable source of inefficiency when number of elements is very large. The benchmarks presented in Section 5.1 evidence this fact. GHC provides unboxed arrays, whose values are stored in contiguous memory areas and can be directly marshalled to MPI buffers. Since most high performance computing applications operate over arrays, and not using lists, unboxed arrays may be used in order to avoid this source of inefficiency.

### 5 Performance Evaluation

This section presents some performance figures for Haskell*#* programs presented in Section 3. The architecture used is a Beowulf cluster comprising 16 dual Intel Xeon processors (clock: 2 GHz, RAM: 1GB), connected through a Fast Ethernet (100MBs). Measures with 32 nodes were performed in dual multiprocessing mode. MPICH 1.2.3 on top of TCP/IP was used for communication between processes.

#### 5.1 Benchmarking Haskell*#* with NPB

The benchmark results of Haskell*#* versions of NPB kernels (EP, IS and CG) are presented in Figure 25. The plots to left hand side present their respective running times, while the plots at right hand side presents their corresponding absolute speedups, comparing them to linear speedup, always represented by a solid line.

Two problem instances were used for measuring performance of Haskell*#* kernel versions (Table 2). In the second one, processes demand about twice
Figure 25: Performance Figures of NPB kernels in Haskell
Table 2: Instances of Problem Sizes Used to Run Each Kernel

<table>
<thead>
<tr>
<th>KERNEL</th>
<th>1st Problem Size</th>
<th>2nd Problem Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP</td>
<td>$m = 25$</td>
<td>$m = 28$</td>
</tr>
<tr>
<td>IS</td>
<td>$total_keys_{\log 2} = 20$</td>
<td>$total_keys_{\log 2} = 21$</td>
</tr>
<tr>
<td></td>
<td>$max_key_{\log 2} = 16$</td>
<td>$max_key_{\log 2} = 17$</td>
</tr>
<tr>
<td>CG</td>
<td>$na = 14000$</td>
<td>$na = 18000$</td>
</tr>
<tr>
<td></td>
<td>$nonzer = 11$</td>
<td>$nonzer = 12$</td>
</tr>
<tr>
<td></td>
<td>$niter = 45$</td>
<td>$niter = 45$</td>
</tr>
</tbody>
</table>

as much memory space as the first one, without exhausting physical memory resources of a single node of the cluster. The default problem classes of NPB (S,W,A,B,C) were not used because they were tuned for use with C/FORTRAN + MPI original versions. Due to laziness and the use of immutable arrays, sequential performance of Haskell versions are about an order of magnitude worse than the performance of the original versions of NPB kernels, both considering time and space. Because of that, some default problem sizes exhaust physical memory resources of cluster nodes, causing virtual memory overheads that must be avoided in measures. The use of mutable arrays could minimize this source of inefficiency, but they require the encapsulation of computations inside the IO monad, preventing arrays of being transmitted through lazy lists.

Also due to performance differences in sequential mode of execution, granularity of Haskell processes is coarser than the granularity of processes in original NPB versions. While Haskell computations execute slower than C/FORTRAN computations, the amount of data transmitted is about the same. The original speedup measures of NPB kernels serve only to establish the lower bounds of the performance of the cluster. One should not use that to make assumptions and claims about relative efficiency of Haskell implementation.

Using GHC profiling tools [81], five main cost centres were identified in CG and IS Haskell implementations. Table 3 presents the impact of each of them in parallel execution. The impact of cost centres in speedup is evaluated on Table 4. By analyzing the data obtained, one may be conclude that:

1. If only time spent in computation is considered, the speedup is linear;
Table 3: Cost Centre Analysis of IS and CG (% of total execution time)

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>ii</th>
<th>iii</th>
<th>iv</th>
<th>v</th>
<th>i</th>
<th>ii</th>
<th>iii</th>
<th>iv</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQ</td>
<td>45.9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>54.1</td>
<td>90.2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9.8</td>
</tr>
<tr>
<td>2</td>
<td>35.4</td>
<td>3.0</td>
<td>7.4</td>
<td>4.6</td>
<td>45.3</td>
<td>79.1</td>
<td>1.5</td>
<td>2.1</td>
<td>4.7</td>
<td>7.7</td>
</tr>
<tr>
<td>4 IS-1</td>
<td>37.6</td>
<td>3.0</td>
<td>7.2</td>
<td>11.0</td>
<td>35.7</td>
<td>CG-1</td>
<td>70.9</td>
<td>1.8</td>
<td>3.2</td>
<td>11.6</td>
</tr>
<tr>
<td>8</td>
<td>36.0</td>
<td>2.7</td>
<td>7.2</td>
<td>20.8</td>
<td>28.7</td>
<td></td>
<td>57.5</td>
<td>3.5</td>
<td>5.6</td>
<td>24.0</td>
</tr>
<tr>
<td>16</td>
<td>34.0</td>
<td>2.5</td>
<td>7.1</td>
<td>27.8</td>
<td>24.4</td>
<td></td>
<td>50.5</td>
<td>4.1</td>
<td>7.3</td>
<td>32.7</td>
</tr>
<tr>
<td>SEQ</td>
<td>34.5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>65.5</td>
<td>84.5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>15.5</td>
</tr>
<tr>
<td>2</td>
<td>38.6</td>
<td>2.8</td>
<td>6.7</td>
<td>5.4</td>
<td>49.3</td>
<td></td>
<td>68.5</td>
<td>1.2</td>
<td>1.7</td>
<td>10.8</td>
</tr>
<tr>
<td>4 IS-2</td>
<td>35.3</td>
<td>2.8</td>
<td>6.8</td>
<td>11.7</td>
<td>38.9</td>
<td>CG-2</td>
<td>70.2</td>
<td>1.5</td>
<td>2.5</td>
<td>12.9</td>
</tr>
<tr>
<td>8</td>
<td>32.8</td>
<td>2.7</td>
<td>7.0</td>
<td>21.1</td>
<td>32.6</td>
<td></td>
<td>61.1</td>
<td>3.2</td>
<td>5.1</td>
<td>19.5</td>
</tr>
<tr>
<td>16</td>
<td>30.1</td>
<td>2.7</td>
<td>7.2</td>
<td>27.8</td>
<td>28.7</td>
<td></td>
<td>58.5</td>
<td>3.5</td>
<td>5.6</td>
<td>25.0</td>
</tr>
</tbody>
</table>

i: Raw computation time, ii: Evaluation of wire functions, iii: Marshalling, iv: Communication and synchronization, v: Garbage Collection

2. The marshalling cost centre is the unique source of overhead inherent to Haskell implementation. The other ones are inherent to parallelism. In some cases, marshalling overhead increases with the number of processors (CG-1 and CG-2). Marshalling could be avoided if GHC allows to copy immutable arrays to contiguous buffers in constant time. But this feature could not be provided yet;

3. The garbage collection overhead decreases by increasing the number of processors used in parallel computation. This fact is attributed to less use of heap when the problem size is split among more processors and the enforcement of data locality. Cache behavior effects are also being investigated. It is worthwhile to remember that garbage collector parameters were tuned before execution. The results obtained here do not guarantee that every Haskell program presents the same behavior;

4. In CG, whenever number of processors increases, the gains in performance due to the minimization of the garbage collection overhead appears to compensate losses due to the marshalling overhead. Thus, in some cases, Haskell overhead may be considered null. Indeed, assuming that arrays are copied directly and in constant time, the minimization of the garbage collection overhead could compensate their sources of overhead that are inherent to parallelization;
Table 4: Influence of Cost Centres in Speedup

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.1</td>
<td>1.9</td>
<td>1.6</td>
<td>1.5</td>
<td>1.2</td>
<td>2.0</td>
<td>2.0</td>
<td>1.9</td>
<td>1.8</td>
<td>1.9</td>
</tr>
<tr>
<td>4</td>
<td>4.1</td>
<td>3.8</td>
<td>3.2</td>
<td>2.6</td>
<td>2.5</td>
<td>3.9</td>
<td>3.8</td>
<td>3.6</td>
<td>3.2</td>
<td>3.2</td>
</tr>
<tr>
<td>8</td>
<td>IS-1</td>
<td>7.5</td>
<td>7.0</td>
<td>5.9</td>
<td>4.0</td>
<td>4.4</td>
<td>CG-1</td>
<td>7.9</td>
<td>7.4</td>
<td>6.7</td>
</tr>
<tr>
<td>16</td>
<td>15.1</td>
<td>13.8</td>
<td>11.8</td>
<td>8.3</td>
<td>8.5</td>
<td>15.9</td>
<td>13.4</td>
<td>12.8</td>
<td>10.5</td>
<td>10.9</td>
</tr>
<tr>
<td>2</td>
<td>2.1</td>
<td>2.0</td>
<td>1.9</td>
<td>-</td>
<td>-</td>
<td>2.0</td>
<td>2.0</td>
<td>1.7</td>
<td>1.8</td>
<td>3.3</td>
</tr>
<tr>
<td>4</td>
<td>4.1</td>
<td>3.8</td>
<td>3.2</td>
<td>2.5</td>
<td>2.5</td>
<td>4.0</td>
<td>3.9</td>
<td>3.7</td>
<td>3.2</td>
<td>3.4</td>
</tr>
<tr>
<td>8</td>
<td>IS-2</td>
<td>8.0</td>
<td>7.3</td>
<td>6.1</td>
<td>4.1</td>
<td>4.5</td>
<td>CG-2</td>
<td>8.0</td>
<td>7.6</td>
<td>7.0</td>
</tr>
<tr>
<td>16</td>
<td>16.1</td>
<td>14.7</td>
<td>12.0</td>
<td>8.0</td>
<td>8.2</td>
<td>16.0</td>
<td>14.4</td>
<td>13.4</td>
<td>10.9</td>
<td>11.2</td>
</tr>
</tbody>
</table>

a: i, b: i/ii, c: i/ii/iii, d: i/ii/iii/iv, e: i/ii/iii/iv/v

The observations above are evidences that Haskell\# programs are an efficient approach for parallelizing functional computations. The fact observed that splitting of problems among processors may reduce the garbage collection overheads is another motivation for using Haskell\# for parallelizing scientific high-performance applications written in Haskell, in addition to the gains in execution time of computations, since this kind of application normally processes large data structures stored in memory. The benchmarks presented in the next section compare Haskell\# to other parallel functional languages.

5.2 Benchmarking Haskell\# with Loild’s Benchmark Suite

The benchmarking results of Haskell\# implementations of Matrix Multiplication (MM), LinSolv (LS), and Ray Tracer (RT), based on Eden and GpH versions presented in [59], are shown in Figure 26. The parameters are described on Table 5. Since the cluster used has nodes about three times as fast as than nodes of the cluster used in Loild’s measures, the size of the problem instance of MM and RT used in this paper are larger. This attempts to approximate the sequential run-time of original measures and the increase of granularity of computations. For LS, however, the same problem size is used since its scalability is less sensitive to variations in problem size.

The speedup curves of LS and RT are nearly linear, while the speedup
Figure 26: Performance Figures of MM, LS and RT in Haskell
Table 5: Problem Instance Parameters for Loïd’s Benchmark Suite

<table>
<thead>
<tr>
<th>Problem Instance</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Matrix Multiplication</strong></td>
<td>$960 \times 960$ matrices of integers with maximum value of 65536.</td>
</tr>
<tr>
<td><strong>LinSolv</strong></td>
<td>Dense $62 \times 62$ matrix of arbitrary precision integers with maximum value of $2^{16} - 1$.</td>
</tr>
<tr>
<td><strong>Ray Tracer</strong></td>
<td>An $1000 \times 1000$ image (in pixels) with a scene comprising 640 spheres.</td>
</tr>
</tbody>
</table>

Curve of MM is negatively affected due to the overhead caused by marshalling large nested lists of integers. For row and block clustering of MM, the times measured in 16 processors were little worse than those obtained for 8 and 9 processors, respectively. The marshalling overhead could be minimized by use of Haskell arrays instead of lists to represent matrices. The Haskell implementations for NPB kernels, where the amount of exchanged data is far larger, evidence this hypothesis. The toroidal version of MM yields a better performance scalability in comparison to row and block clustering, once the amount of data transmitted is comparatively smaller.

It is important to observe that measures of LS and RT for 32 processors were obtained on dual processing mode across 16 nodes of the cluster. Unexpected additional overhead was observed when executing MPI programs using the dual mode processing capabilities. This effect was more easily observed when measuring the run time of Haskell versions of NPB kernels CG and IS, probably due to the large amount of data exchanged between processors in collective communication. Because of that, the results for NPB with 32 processors was not presented. Best speedup for LS and RT were expected for 32 non-dual processors. For that reason, in following discussion, the measures with 16 processors is used as a reference for comparing benchmarks of Haskell to benchmarks of GpH, Eden, and PMLS.

Comparing Haskell results to the best ones obtained for Eden, GpH, and PNML described in [59], one may observe that Haskell results are slightly better in all cases. For example, MM using toroidal solution obtains a speedup of 11.0 on 16 processors, while a speedup of approximately 5.0 was the best obtained in Eden toroidal solution. For LS, the speedup obtained in
Haskell\# is 14.3 on 16 processors, while the best speedup obtained in Eden version was 14.0. For RT, a speedup of 15.6 was obtained by Haskell\# on 16 processors, while 15.1 was the best speedup obtained in PMLS version.

The results presented herein are not yet sufficient to conclude that Haskell\# programs are always more efficient than their GpH, Eden and PMLS versions. The two compared benchmarks were obtained for distinct architectures and using different problem sizes. However, the results presented in this paper evidence that Haskell\# implementation presents comparable behavior to well-known and mature implementations of parallel functional languages, such as GpH, Eden and PMLS. The results obtained are not surprising, since Haskell\# run-time system is very light in relation to the complex parallel run-time systems of GpH, Eden, and PMLS, which try to hide some parallel management details from programmers at different degrees. Decades of experience in parallel languages design have shown that as explicit as it is a general parallel language, assuming that it has an efficient implementation, best scalability is obtained using a simpler run-time system. The combination of the results obtained in this paper and in [59] only confirm this hypothesis. In this sense, Haskell\# is the most explicit of all, followed by Eden, PMLS and GpH, respectively.

6 Conclusions and Lines for Further Work

This paper introduces Haskell\#, a coordination language for describing parallel execution of functional computations in Haskell. Haskell\# intends to raise the level of abstraction in explicit message-passing parallel programming on distributed architectures, such as clusters, for the development of large scale parallel scientific computing applications. Motivating examples, implementation issues and performance figures of Haskell\# benchmarks are also presented.

After some years of design, implementation and evaluation, Haskell\# has reached some maturity. Several works unfoldings are on progress. Firstly, a parallel programming environment based on Haskell\# have been prototyped in JAVA, including the support for programming with visual abstractions, integration to Petri net tools for animation, proving of formal properties, and performance evaluation of programs. It is also under development the use of network simulators, such as Network Simulator (NS) [31] for simulating the performance of parallel programs. Such tool will allow to study the effect of
modifications to network characteristics on performance of parallel programs. This work has important impact on studying behavior of Haskell programs whenever executing on grids.

Since HCL is a coordination language orthogonal to Haskell, it is conceptually possible to use other languages, in alternative to Haskell, for programming functional modules. The parallel programming environment under development assumes that Haskell is the ideal language for specifying, prototyping and evaluating the formal properties of parallel programs. Once parallel composition is proved be safe, programmers may implement the functional modules using a language more appropriate for implementing the functionality of the simple components. For example, numerical intensive functional modules could be implemented in Fortran, while sorting of large amount of numbers in parallel may be implemented in C. JAVA can be used for programming functional modules that make access to some database. This kind of multi-lingual compositional approach is a further development. One important design difficult with multilingual approach is to maintain the orthogonality between languages used at coordination and computations levels in absence of lazy and higher-order functional programming. Imperative languages, for example, do not allow to hide the control flow. It is intended to use techniques from aspect oriented programming (AOP) for addressing this matter. In this direction, parallel composition could be treated as an aspect of programming. The recent appearance of heterogeneous versions of MPI [84] is important for making feasible a multi-lingual approach for Haskell.

An even more relevant important topic to be addressed is to develop cost models for Haskell skeletons, incorporating the possibility of overlapping them, and to use it for allowing Haskell compiler to make automatic decisions, such as better allocation of processes to processors, use of special primitives, and special restrictions on communication modes, such as the size of buffers. However, a recent idea is to design a meta-language for programmers to teach explicitly Haskell compiler on how to generate the appropriate code for a given skeleton or a combination of skeletons. The latter approach is more in tune with Haskell design premises. However, it is not difficult to see that the two lines could be combined.

Further developments will address grid enabled implementations of Haskell. A grid enabled version of MPI, such as the recently proposed MPICH-G2 [49], might be used.
References


A  The Formal Syntax of HCL

In what follows, it is described a context-free grammar for HCL, the Haskell Configuration Language, whose syntax and programming abstractions were informally presented in Section 2. Examples of HCL configurations and their meanings were presented in Sections 2 and 3. The notation employed here is similar to that used for describing syntax of Haskell 98 [75]. Indeed, some non-terminals from that grammar are reused here, once some Haskell code appears in HCL configurations. They are faced italic and bold. A minor difference on notation resides on the use of $(...)$, instead of $[...$, for describing optional terms. For simplicity, notation for indexed notation is ignored from the description of formal syntax of HCL. It may be resolver by a preprocessor, before parsing.

A.1  Top-Level Definitions

\[
configuration \rightarrow header \  declaration_1 \ldots \ declaration_n \ (n \geq 0) \\
header \rightarrow component \ ID \ static\_parameter\_list? \ component\_interface? \\
static\_parameter\_list \rightarrow < ID_1 \ldots ID_n > \ (n \geq 0) \\
component\_interface \rightarrow \ ports\_naming \\
declaration \rightarrow import\_decl \ | use\_decl \ | iterator\_decl \ | interface\_decl \\
| unit\_decl \ | assign\_decl \ | replace\_decl \ | channel\_decl \\
| unify\_decl \ | factorize\_decl \ | replicate\_decl \ | bind\_decl \\
| haskell\_code
\]

A.2  Use Declaration

\[
use\_decl \rightarrow use \ use\_spec \\
use\_spec \rightarrow id \ | \ id.\ use\_spec \ | \ id.\{ \ use\_spec_1 \ , \ldots \ , \ use\_spec_n \ \} \ (n \geq 1)
\]

A.3  Import Declaration

\[
import\_decl \rightarrow imp\_decl
\]

A.4  Iterator Declaration

\[
iterator\_decl \rightarrow iterator \ id_1, \ldots, id_n \ range \ [ \ numeric\_exp \ , \ numeric\_exp \ ] \ (n \geq 1)
\]
A.5 Interface Declaration

\[
\text{interface}\_\text{decl} \rightarrow \text{interface} (\text{context} \Rightarrow)^2 \text{ID} \ \text{tyvar}_1 \ldots \text{tyvar}_k \ \text{interface\_spec}
\]

\[
\text{interface\_spec} \rightarrow \text{interface\_ports\_spec}
\]

\[
(\text{where: interface\_inheritance})^2 \ (\text{behavior: behavior\_expression})^2
\]

A.5.1 Interface Ports Description

\[
\text{interface\_ports\_spec} \rightarrow \text{port\_spec\_list} \Rightarrow \text{port\_spec\_list}
\]

\[
\text{port\_spec\_list} \rightarrow \text{port\_spec} \mid (\text{port\_spec}_1 \ldots \text{port\_spec}_n) \ (n \geq 2)
\]

\[
\text{port\_spec} \rightarrow \text{id} (*)^2 \ (\text{:: atype})^2 \mid \text{id}
\]

A.5.2 Interface Composition

\[
\text{interface\_inheritance} \rightarrow \text{interface\_slice}_1 \# \ldots \# \text{interface\_slice}_k \ (k \geq 1)
\]

\[
\text{interface\_slice} \rightarrow \text{id @ ID} \mid \text{ID} \ \text{ports\_naming\_composition}
\]

\[
\text{ports\_naming\_composition} \rightarrow \text{ports\_naming}
\]

\[
\mid (\text{ports\_naming}_1 \# \ldots \# \text{ports\_naming}_n) \ (n \geq 1)
\]

\[
\text{ports\_naming} \rightarrow \text{port\_naming\_list} \Rightarrow \text{port\_naming\_list}
\]

\[
\text{port\_naming\_list} \rightarrow \text{id} \mid (\text{id}_1 \ldots \text{id}_n) \ (n \geq 1)
\]

A.5.3 Interface Behavior

\[
\text{behavior\_expression} \rightarrow (\text{sem id}_1 \ldots \text{id}_n)^2 : \text{action} \ (n \geq 1)
\]

\[
\text{action} \rightarrow \text{par} \{ \text{action}_1 ; \ldots ; \text{action}_n \} \mid \text{seq} \{ \text{action}_1 ; \ldots ; \text{action}_n \}
\]

\[
\mid \text{alt} \{ \text{action}_1 ; \ldots ; \text{action}_n \} \mid \text{repeat action condition}^2
\]

\[
\mid \text{if condition then action else action}
\]

\[
\mid \text{id} ! \mid \text{id} ? \mid \text{signal id} \mid \text{wait id} \ (n \geq 2)
\]

\[
\text{condition} \rightarrow \text{until disjunction} \mid \text{counter numeric\_exp}
\]

\[
\text{disjunction} \rightarrow \text{conjunction}_1 \mid \ldots \mid \text{conjunction}_n \ (n \geq 1)
\]

\[
\text{sync\_conjunction} \rightarrow \{ \text{simple\_conjunction} \} \mid \text{simple\_conjunction}
\]

\[
\text{simple\_conjunction} \rightarrow \text{id} \mid (\text{id}_1 \& \ldots \& \text{id}_n) \ (n \geq 1)
\]

A.6 Unit Declaration

\[
\text{unit\_decl} \rightarrow \text{unit unit\_spec}
\]

\[
\text{unit\_spec} \rightarrow (*)^2 \text{id} (\# \text{unit\_interface})^2 (\text{wire wsetup}_1 \ldots \text{wsetup}_n)^2
\]

\[
\text{unit\_interface} \rightarrow \text{ID ports\_naming\_composition}^2 \mid \text{interface\_spec}
\]
\( \text{wf\_setup} \rightarrow \text{id (group\_type group\_spec)}? (\text{wire\_function})? \)

\( \text{group\_spec} \rightarrow \{ \text{id}_1, \ldots, \text{id}_n \} | ^* \text{numeric\_exp} \)

\( \text{group\_type} \rightarrow \text{any | all} \)

\( \text{wire\_function} \rightarrow ? | \text{exp} \)

### A.7 Assignment Declaration

\( \text{assign\_decl} \rightarrow \text{assign assigned\_component to assigned\_unit} \)

\( \text{assigned\_component} \rightarrow \text{ID actual\_parameter\_list}\? \text{ports\_naming\_composition}\? \)

\( \text{actual\_parameter\_list} \rightarrow < \text{numeric\_exp}_1, \ldots, \text{numeric\_exp}_n > (n \geq 1) \)

\( \text{assigned\_unit} \rightarrow \text{qid ports\_naming\_composition}\? \)

### A.8 Replace Declaration

\( \text{replace\_decl} \rightarrow \text{replace qid ports\_naming\_composition}\? \text{by operand\_unit} \)

### A.9 Channel Declaration

\( \text{channel\_decl} \rightarrow \text{connect qid -> qid to qid <- qid , comm\_mode} \)

\( \text{comm\_mode} \rightarrow \text{synchronous | buffered numeric\_exp | ready} \)

### A.10 Unification Declaration

\( \text{unify\_decl} \rightarrow \text{unify operand\_unit}_1, \ldots, \text{operand\_unit}_n \text{ to unitt\_spec} \)

\( \text{adjust wire} \ \text{wf\_setup}_1, \ldots, \text{wf\_setup}_k \ (n \geq 2, k \geq 1) \)

\( \text{operand\_unit} \rightarrow \text{qid \# interface\_pattern}_1 \ldots \# interface\_pattern}_n \ (n \geq 1) \)

\( \text{interface\_pattern} \rightarrow \text{port\_pattern\_list} \rightarrow \text{port\_pattern\_list} | \text{id} \)

\( \text{port\_pattern\_list} \rightarrow \text{pattern} | (\text{pattern}_1, \ldots, \text{pattern}_n) \)

\( \text{pattern} \rightarrow \text{id | @ qid | _ | _} \)

### A.11 Factorization Declaration

\( \text{factorize\_decl} \rightarrow \text{factorize operand\_unit to unit\_spec}_1 \ldots \text{unit\_spec}_n \)

\( \text{adjust wire} \ \text{wf\_setup}_1, \ldots, \text{wf\_setup}_k \ (n \geq 2, k \geq 1) \)
A.12 Replication Declaration

\[ \text{replicate}_\text{decl} \rightarrow \text{replicate} \ operand\_unit_1, \ldots, operand\_unit_n \ \text{into} \ numeric\_exp \ \text{adjust} \ wire \ wf\_setup_1, \ldots, wf\_setup_k \ (n \geq 2, \ k \geq 1) \]

A.13 Bind Declaration

\[ \text{bind}\_\text{declaration} \rightarrow \text{bind} \ qid \to \ qid \to \ id \ | \ \text{bind} \ qid \leftarrow \ qid \to \leftarrow \ id \]

A.14 Miscellaneous

\[ \text{haskell}\_\text{code} \rightarrow \text{topdecls} \]

\[ qid \rightarrow \text{id}_1 \ '. ' \ldots ' . ' \text{id}_n \ (n \leq 2) \]

\[ qID \rightarrow \text{ID}_1 \ '. ' \ldots ' . ' \text{ID}_n \ (n \leq 2) \]

B An Algebraic Semantics for Haskell# Components

This appendix presents an algebra intending to formalize semantics of Haskell# programming abstractions at coordination level. A Haskell# component \( H \) may be defined by an algebra with the following elements:

\[ H = \langle G, R, C \rangle \]

where \( G \) is a set of generators, \( R \) is a set of relations on generators, and \( C \) is a set of restrictions on relations, defined as following:

\[ G = \left\{ C, \ \text{composed components} \right\} \]

\[ \left\{ S, \ \text{simple components} \right\} \]

\[ \left\{ U, \ \text{units} \right\} \]

\[ \left\{ G, \ \text{ports groupings} \right\} \]

\[ \left\{ P, \ \text{individual ports} \right\} \]

\[ \left\{ R, \ \text{kinds of processes: repetitive or non-repetitive} \right\} \]

\[ \left\{ D, \ \text{port directions: input or output} \right\} \]

\[ \left\{ T, \ \text{port type: any or all} \right\} \]

\[ \left\{ M, \ \text{communication modes: synchronous, buffered or ready} \right\} \]
A Haskell# program is a component that may execute. Essentially, it does not have virtual units in its composition (it is not a partial skeleton). All units are assigned to a component (\( \forall u : u \in U : (\exists c : c \in C \cup S : \gamma(u) = c) \)).

In what follows, the restrictions from \( R_1 \) to \( R_{12} \) are described. They are formulas in predicate logic (predicate) of the following form (\( \forall b_1, b_2, \ldots, b_n : R : P \)) or (\( \exists b_1, b_2, \ldots, b_n : R : P \)), where \( \forall \) and \( \exists \) are the usual existential quantifier, \( b_i, 1 \leq i \leq n \), are bound variables, \( R \) is a formula that specifies the set of values of bound variables, and \( P \) is a logical predicate.

The restriction \( R_1 \) states that component \( \omega \) (main component) is the only component that is not assigned to any unit:

\[
R_1 \vdash \forall u : u \in U : \gamma(u) \neq \omega \tag{1}
\]

\( R_2 \) states that cyclic dependencies may not occur in component hierarchy:

\[
R_2 \vdash \forall u : u \in U \land \gamma(u) \neq \bot : u \notin (\delta \circ \gamma)(u)
\]

where:
\[
\begin{align*}
\delta(s) &= \emptyset & s &\in S \\
\delta(c) &= \bigcup_{u \in \delta(c)} (\delta \circ \gamma)(u) & c &\in C
\end{align*}
\]

\( R_3 \) states that a cluster is repetitive whenever all units belonging to its assigned component are repetitive:

\[
R_3 \vdash \forall u : u \in U \land (\exists c : c \in C : \gamma(u) = c \land \delta(c) \neq \emptyset) : \\
\rho(u) = \text{Repetitive} \iff (\forall u : u \subseteq (\delta \circ \gamma)(u)) : \rho(u) = \text{Repetitive} \tag{3}
\]
\textbf{R}_4 \text{ state that groups of ports are disjoint, } \textbf{R}_5 \text{ states that all individual ports belong to a group of ports, and } \textbf{R}_6 \text{ states that groups of ports must not be empty:}

\begin{equation}
\begin{aligned}
\textbf{R}_4 & \forall g, g' : g, g' \in G : \pi(g) \cap \pi(g') = \emptyset \\
\textbf{R}_5 & \forall p \exists g : p \in P \land g \in G : p \in \tau(g) \\
\textbf{R}_6 & \forall g : g \in G : \tau(g) \neq \emptyset
\end{aligned}
\end{equation}

(4)

In the algebra, all ports are treated as non-empty groups. Thus, an individual port in a Haskell program is represented as a group containing an unique port. The restrictions above makes possible to define a “inverse” relation $\tau^3$, such that $\tau(p)$ returns the group $g$ that $p$ belongs. It is useful for simplifying next formulations.

Restrictions $\textbf{R}_7$, $\textbf{R}_8$, and $\textbf{R}_9$ specifies rules for formation of channels. Respectively, they say that channels are point-to-point, unidirectional and have the same nesting factors:

\begin{equation}
\begin{aligned}
\textbf{R}_7 & \vdash (p^0_1, p^1_1, m) \in \nu \land (p^2_1, p^2_1, m) \in \nu \Rightarrow p^0_1 = p^2_1 \Leftrightarrow p^1_1 = p^2_1 \\
\textbf{R}_8 & \vdash (p^o, p^i, m) \in \nu \Rightarrow (\exists u, u' : u, u' \in U : (\pi \circ \tau)(p^o) = (u, \text{Output}) \land (\pi \circ \tau)(p^i) = (u', \text{Input})) \\
\textbf{R}_9 & \vdash (p^o, p^i, m) \in \nu \Rightarrow (\lambda \circ \tau)(p^o) = (\lambda \circ \tau)(p^i)
\end{aligned}
\end{equation}

(5)

Let $u$ be a cluster ($\gamma(u) \subseteq C$) and $p$ be an individual port belonging to group $g$, such that $\pi(g) = (u, d)$, for $d \in D$ ($p$ belongs to interface of $u$). The restriction $\textbf{R}_{10}$ ensures that $\psi(p)$ (argument or exit point of $\gamma(u)$) is a port with the same direction of $p$ belonging to interface of unit $u'$, such that $u'$ is a unit belonging to the component $\gamma(u)$ ($u' \in (\delta \circ \sigma)(u)$):

\begin{equation}
\begin{aligned}
\textbf{R}_{10} & \vdash (\pi \circ \tau)(p) = (u, d) \land \gamma(u) \subseteq C : ((\pi \circ \psi)(p) = (u', d) \land u' \in (\delta \circ \gamma)(u))
\end{aligned}
\end{equation}

(6)

The restriction $\textbf{R}_{11}$ defines the relation $\iota$, which describes the interface of a unit:

\begin{equation}
\begin{aligned}
\textbf{R}_{11} & \vdash \iota(u) = < \{ g \mid \pi(g) = (u, d) \}, \beta(u) >
\end{aligned}
\end{equation}

(7)

This is not the strict mathematical notion of inverse function, from set theory.
\( R_{12} \) says that ports belonging to the same group whose communication pairs also belong to the same groups are essentially the same port.

\[
R_{12} \vdash (p_0^1, m_1) \in \nu \land (p_0^2, m_2) \land \tau(p_0^1) = \tau(p_0^2) \land \tau(p_1^1) = \tau(p_1^2) \Rightarrow p_0^1 = p_0^2 \land p_1^1 = p_1^2
\]  

(8)

B.1 Formalizing Interfaces

This section formalizes homomorphism relations between interfaces, which are essential for formalizing unification and factorization operations in the next section.

B.1.1 The \# Operator

The \# operator allows for combining to interfaces, generating a new interface that inherits characteristics from original ones. It is defined as following:

\[
I_1 \# I_2 = <Q_1 \cup Q_2, B_1 \hat{\cup} B_2>, \text{ where } I_1 = <Q_1, B_1 > \text{ and } I_2 = <Q_2, B_2 >
\]  

(9)

The sets of ports from operand interfaces may overlap. The operator \( \hat{\cup} \) generates a new formal language describing a behavior for interface \( I_1 \# I_2 \), which is compatible with original behavior of \( I_1 \) and \( I_2 \), in separate. Given an interleaving operator \( \odot \), from concurrent expressions [47] and \( \ell \) a function that returns the language generated by a concurrent expression, formal definition of \( \hat{\cup} \) is:

\[
B_1 \hat{\cup} B_2 = \ell [(w_1 \odot u_1) \ s \ (w_2 \odot u_2) \ s \ldots \ s \ (w_n \odot u_n)], n \geq 1
\]

where

\[
s \in Q_1 \cap Q_2 \\
\begin{align*}
w_1 \ s \ \ldots \ s \ w_n & \in B_1 \ \\
\end{align*}
\]

\[
\begin{align*}
u_1 \ s \ \ldots \ s \ u_n & \in B_2 \ \\
\end{align*}
\]

\[
\begin{align*}
w_1 \ w_2 \ldots w_n & \in (Q_1 - \{s\})^* \ \\
u_1 \ u_2 \ldots u_n & \in (Q_2 - \{s\})^*
\end{align*}
\]

(10)

65
If operand interfaces do not overlap ports, $B_1 \hat{\cup} B_2$ corresponds to interleaving of their original behaviors ($B_1 \odot B_2$). Overlapping ports may be interpreted as synchronization points when combining formal languages $B_1$ and $B_2$.

### B.1.2 Homomorphisms Between Interfaces

Let $I_1 = < Q_1, B_1 >$ and $I_2 = < Q_2, B_2 >$ be interface classes. Let $H$ be a pair $< h : Q_1 \rightarrow Q_2, \overline{h} : B_1 \rightarrow Q_2^* >$, where $\overline{h}$ is defined as following:

$$\begin{align*}
\overline{h}(\epsilon) &= \epsilon \\
\overline{h}(aw) &= h(a)\overline{h}(w)
\end{align*}$$

(11)

With respect to $H = < h, \overline{h} >$, the following interface relations are defined:

$$
\begin{align*}
I_1 & \subseteq^H I_2 \iff \text{Im}(\overline{h}) \subseteq B_2 \\
I_1 & \supseteq^H I_2 \iff \text{Im}(\overline{h}) \supseteq B_2 \\
I_1 & \equiv^H I_2 \iff \text{Im}(\overline{h}) = B_2
\end{align*}
$$

(12)

Relations $\subseteq$ and $\supseteq$ characterize homomorphisms between interfaces, while $\equiv$ characterize isomorphisms between them.

### B.2 An Algebra for Haskell\# Programming

Now, it is defined an algebra to formalize Haskell\# programming task. Operations over units are defined here: unification, factorization, replication and assignment. They may be used to overlap and nest components that comprise a Haskell\# component. An algebra for Haskell\# programming is defined as:

$$< \{ H \}, \{ u : H \times H, f : H \times H, a : H \times H, r : H \times H, i : H \times H \}, \emptyset >$$

where generator $H$ contains all well-formed Haskell\# components. The relations $u$, $f$, $a$ and $r$ represents sets of pairs $(h_1, h_2)$, $h_1 \in H$ and $h_2 \in H$, where $h_2$ is a Haskell\# component obtained from Haskell\# component $h_1$ from an application of unification, factorization, assignment or replication operations, respectively, defined further. The relation $i$ is a identity relation containing pairs $(h, h)$, $\forall h \in H$. 
In what follows, assignment, unification, factorization, and replication operations, homomorphisms between Haskell\# components, are defined. Since all Haskell\# components may be described using HCL configurations that should be generated using a context-free grammar, the set $H$ is recursively enumerable. Thus, in what follows, the $i$-th Haskell\# program, $i \geq 0$, is denoted by $\#_i = (G_i, R_i, C_i)$, where $G_i = \{C_i, S_i, U_i, G_i, P_i, R_i, D_i, T_i, M_i\}$, $G_i = \{\omega_i, \delta_i, \psi_i, \gamma_i, \pi_i, \beta_i, \tau_i, \rho_i, \nu_i, \lambda_i\}$.

### B.2.1 Unification and Factorization

Unification and factorization, informally introduced in Section 2.1.10, are formalized here as mutually reversible relations in the algebra of Haskell\# programming. For instance, consider two Haskell\# components and their algebraic description, denoted by $\#_k$ and $\#_j$, for some $k, j \geq 0$. Consider $\hat{V} = \langle v_1, v_2, \ldots, v_n \rangle$ an ordered sub-set of virtual units in $U_k$, and their respective interfaces $\hat{I} = \langle I_1, I_2, \ldots, I_n \rangle$, such that $I_i = \iota(v_i) = (Q_i, B_i)$, for $1 \leq i \leq n$. Also, consider a virtual unit $v \in U_j$ and its interface $I = \iota(v) = (Q, B)$. A set of interface mappings $\hat{H} = \langle H_1, H_2, \ldots, H_n \rangle$, where $H_i = (h_i, \overline{h_i})$ maps interface $I_i$ to interface $I$ is defined. Suppose that $\#_j$ is obtained from $\#_k$ by unification of virtual units in $\hat{V}$ to a unique virtual unit $v$. It is also supposed correct to say that $\#_k$ is obtained from $\#_j$ by factorization of the virtual unit $v$ onto the set of virtual units $\hat{V}$.

Two restrictions may be ensured in a correct application of unification and factorization operations. The first one imposes behavior preserving restrictions for units, stating that $v$ is a proper unification of virtual units in set $\hat{V}$ if $I_i \xleftarrow{H_i} I$, for $1 \leq i \leq n$. Analogously, units in $\hat{V}$ constitute a proper factorization of $v$ if $I_i \xrightarrow{H_i} I$, for $1 \leq i \leq n$. The second one establishes restrictions for preservation of network connectivity. But before to talk about them, it is necessary to define relation $\hat{\tau} : Q \rightarrow 2^{I_\hat{V}_j}$. It makes possible to formalize partitioning of groups of ports, which must be configured explicitly in factorizations. In unifications, it is not necessary to configure $\hat{\tau}$ explicitly using HCL, since the inverse of partitioning of groups of ports is the union of them, which is resolved by merging the groups. Ports $b$ and $e$ in Figure 6 are examples of partitioning (right to left) and union (left to right) of ports. The relation $\hat{\tau}$ must satisfy the restriction defined in Equation 13, which relates it with interface mapping $\hat{H}$.
∀q': q' ∈ G_j \land (\exists q : q \in Q : \hat{H}(q) = q') : 
\left( \bigcup_{q \in R} \hat{\tau}(q) \right) = \tau(q'), \ R = \{ q \mid q \in Q \land H(q) = q' \}

(13)

In this paragraph, restrictions for ensuring preservation of network connectivity with respect to unification/factorization are discussed. In the trivial case, where overlapping of ports does not occur (\( \hat{H}(q_1) = \hat{H}(q_2) \Rightarrow \hat{\tau}(q_1) \cap \hat{\tau}(q_2) = \emptyset \)), all ports and channels are preserved (\( P_j = P_k \) and \( \nu_j = \nu_k \)) after applying unification/factorization. Essentially, only the sets of units (\( U_j - U_k = \{ v \} \land U_k - U_j = \hat{V} \)), ownership of ports (relations \( \pi_k \) and \( \pi_j \)), and grouping of ports (relations \( \tau_k \) and \( \tau_j \)) differs between \#_j and \#_k. Ownership and grouping of ports is affected by interface mappings \( \hat{H} \). If overlapping of port occurs, some adjustment of ports and channels may be necessary in order to ensure obedience to restrictions for channel formation. For instance, consider a port \( p \), such that \( \exists Q : Q \subseteq Q : (\forall q : q \in Q : p \in \hat{\tau}(q)) \land |Q| \geq 2 \).

From the perspective of factorization, \( p \) is interpreted as a port of unit \( v \), in component \#_j, that have more than one port in \( P_k \) associated to it, possibly all belonging to distinct units in the set \( \hat{V} \) of component \#_k. For ensuring point-to-point nature of channels (R7), the communication pair of \( p, \overline{p} \ ((p, \overline{p}, m) \in \nu_j \lor (\overline{p}, p, m) \in \nu_j) \), must be replicated in \(|Q|\) copies as consequence of factorization. They are connected to the ports belonging to groups in \( Q \) that have association to \( p \). From the perspective of unification, \( p \) is a port of \#_j that comes from unification of a set of ports \( Q = \{ p' \mid p' \in P_k \land p \in \hat{\tau}(p') \} \) of \#_k. The communication pairs of ports in \( Q, \overline{Q} \), are members of the same group of ports (\( \exists g : g \in G_k : Q \subseteq \tau_k(g) \)). In such case, in order to satisfy restriction R12, ports in \( P \) are unified in a single port \( \overline{p} \) in \#_j, the communication pair of \( p \).

B.2.2 Assignment

In an executable Haskell program, application component must not contain virtual units. Thus, it is necessary to define an operation for associating components to virtual units (nesting composition). Let \#_k and \#_i be Haskell programs, \( v \in V_k \) be a virtual unit in program \#_k, and \( \overline{\psi} \) a mapping from ports of interface of \( v \) to arguments and exit points of \( \omega_i \) (main component of \#_i).
Assignment of main component of $\#_i(\omega_i)$ to virtual unit $v$ of $\#_k$, produces a new program $\#_k$, the union of generators and relations from two programs, where $v$ is associated to $\omega_i$ through $\gamma_k$. Arguments and exit points of $\omega_i$ are associated to $v$ ports through $\psi_k$, using $\psi$.

B.2.3 Replication

Let $\#_k$ be a Haskell program. Given a positive integer $r > 1$ and a collection of units $U \subseteq U_k$, $U = \{u_1, u_2, \ldots, u_n\}$, it is possible to replicate the sub-network induced by units in $U$ in $r$ copies, forming a new program $\#_j$. In order to maintain network connectivity and attendance to Haskell algebra restrictions, when defining $\#_j$ from $\#_k$, it is necessary to replicate ports from units that are not in $U$ but are connected to any port of some unit in $U$. HCL allows for specifying wire functions for new groups. Channels connecting unit ports between units in $U$ are also replicated in $n$ copies, one connecting each pair of ports from the $n$ units copies.

C  HCL Code for NPB Benchmarks EP, IS, CG, and LU

C.1 EP

```
component EP<NO_NODES, MK, NN, NK, SQ, EPSILON, A, S> with
#define PARAMETERS (EP, Params, NO_NODES, MK, NN, NK, SQ, EPSILON, A, S)

iterator i range [1..NO_NODES]
use Skeletors.Collective.AllReduce
use EP_FM -- EP Functional Module

interface IEP (sx, sy, q) where: sx@AllReduce Double # sy@AllReduce Double # q@AllReduce UDVector

behaviour: seq (do sx; do sy; do q)

unit sx_comm; assign AllReduce<NO_NODES, MK, NN, NK, SQ, EPSILON, A, S> to sx_comm
unit sy_comm; assign AllReduce<NO_NODES, MK, NN, NK, SQ, EPSILON, A, S> to sy_comm
unit q_comm; assign AllReduce<NO_NODES, MK, NN, NK, SQ, EPSILON, A, S> to q_comm

[// unify sx_comm.p[i] # sx, sy_comm.p[i] # sy, q_comm.p[i] # q to ep_unit[i] # IEP
assign EP_FM (PARAMETERS, sx, sy, q) -> (sx, sy, q) to ep_unit[i] # sx # sy # q //]
```

C.2 IS

```
component IS<PROBLEM_CLASS, NUM_PROCS, MAX_RX, LOG2, NUM_BUCKETS, LOG2, TOTAL_RX, LOG2, MAX_ITERATIONS, MAX_PROCS, TEST_ARRAY_SIZE> with
#define PARAMETERS (IS, Parameters, PROBLEM_CLASS, NUM_PROCS, MAX_RX, LOG2, NUM_BUCKETS, LOG2, TOTAL_RX, LOG2, MAX_ITERATIONS, MAX_PROCS, TEST_ARRAY_SIZE)

iterator i range [1..NUM_PROCS]
```
C.3 O kernel CG

C.3.1 Esqueleto Transpose

component Transpose<\textit{dim}, \textit{col\_factor}>

\begin{itemize}
  \item \textbf{iterator} \textit{i}, \textbf{range} \{[1..\textit{dim}]
  \item \textbf{iterator} \textit{k}, \textbf{range} \{[1..\textit{col\_factor}]
\end{itemize}

\textbf{interface} \textit{ITranspose} ((\textit{x}: UDVector) \rightarrow (\textit{w}: UDVector)) \textbf{behaviour}: \texttt{seq} \{ \textit{w}!; \textit{x}? \}

\begin{itemize}
  \item \texttt{unit \textit{trans}[\textit{i}][\textit{j}] \# \textit{w} \rightarrow \textit{x}[\textit{j}]} \rightarrow \textit{w}[\textit{i}]
  \item \texttt{factorize \textit{trans}[\textit{i}][\textit{j}] \# \textit{x} \rightarrow \textit{w} \rightarrow \textit{x}[\textit{j}]}
\end{itemize}

C.3.2 Componente CG

component CG<\textit{dim}, \textit{col\_factor}, \textit{bs}, \textit{kb}, \textit{shift}, \textit{nter}, \textit{ncv}, \textit{zvv}> \# () \rightarrow (\textit{zeta}, \textit{x}) \textbf{with}

\#\textbf{define} \textbf{PARAMETERS} \{CG\_Params \textit{bs} \textit{dim} \textit{col\_factor} \textit{shift} \textit{nter} \textit{ncv} \textit{zvv}\}

use Skeletons MPI Collective ALL\textit{reduce}

use CG::FM \# \textit{CG} \textbf{Functional Module}

\textbf{index} \textit{i}, \textbf{range} \{[1..\textit{dim}]
\textbf{index} \textit{j}, \textbf{range} \{[1..\textit{col\_factor}]

\textbf{interface} \textit{ICG} ((\textit{r*, q**, rho**, aux**, rnorm*, norm}) \rightarrow ((\textit{r*, q**, rho**, aux**, rnorm*, norm}) \texttt{\_transpose} (\textit{x}: UDVector)) \textbf{behaviour}: \texttt{repeat seq} \{ \texttt{do rho: \texttt{repeat seq} \{ \texttt{do q: \texttt{do aux: \texttt{if rho then do rho else skip}} \}} \}\texttt{until \textit{q} < \textit{aux} \& \textit{rho}};

\begin{itemize}
  \item \texttt{do \textit{r} \rightarrow \textit{rho} \& \texttt{do \textit{norm\_temp1} \& \textit{rho} \& \textit{aux} \& \textit{norm\_temp2} \& \textit{rho} \& \textit{norm\_temp2}}
  \item \texttt{until \textit{r} < \textit{rho} \& \textit{norm\_temp1} \& \textit{rho} \& \textit{aux} \& \textit{rho} \& \textit{norm\_temp2} \& \textit{rho} \& \textit{norm\_temp2}}
\end{itemize}

\begin{itemize}
  \item \texttt{unit \textit{q\_comm}}; \texttt{assign \textit{Transp\_\textit{col\_factor}}} \{ \textit{dim} \* \textit{col\_factor}\} \rightarrow \textit{q\_comm}
  \item \texttt{unit \textit{r\_comm}}; \texttt{assign \textit{Transp\_\textit{col\_factor}}} \{ \textit{dim} \* \textit{col\_factor}\} \rightarrow \textit{r\_comm}
  \item \texttt{unit \textit{rho\_comm}[\textit{i}]}; \texttt{assign \textit{All\textit{reduce}}} \{ \textit{col\_factor}, \textit{REduce\_\textit{func}} \} \rightarrow \textit{rho\_comm}[\textit{i}]
  \item \texttt{unit \textit{aux\_comm}[\textit{i}]}; \texttt{assign \textit{All\textit{reduce}}} \{ \textit{col\_factor}, \textit{REduce\_\textit{func}} \} \rightarrow \textit{aux\_comm}[\textit{i}]
  \item \texttt{unit \textit{norm\_temp1\_comm}[\textit{i}]}; \texttt{assign \textit{All\textit{reduce}}} \{ \textit{col\_factor}, \textit{REduce\_\textit{func}} \} \rightarrow \textit{norm\_temp1\_comm}[\textit{i}]
  \item \texttt{unit \textit{norm\_temp2\_comm}[\textit{i}]}; \texttt{assign \textit{All\textit{reduce}}} \{ \textit{col\_factor}, \textit{REduce\_\textit{func}} \} \rightarrow \textit{norm\_temp2\_comm}[\textit{i}]
\end{itemize}

\begin{itemize}
  \item \texttt{unit \textit{rho\_comm}[\textit{i}][\textit{j}]} \# \textit{q}, \texttt{rho\_comm}[\textit{i}][\textit{j}]; \texttt{rho\_comm}[\textit{i}][\textit{j}]
  \item \texttt{unit \textit{aux\_comm}[\textit{i}][\textit{j}]} \# \textit{q}, \texttt{aux\_comm}[\textit{i}][\textit{j}]; \texttt{aux\_comm}[\textit{i}][\textit{j}]
  \item \texttt{unit \textit{norm\_temp1\_comm}[\textit{i}][\textit{j}]} \# \textit{q}, \texttt{norm\_temp1\_comm}[\textit{i}][\textit{j}]; \texttt{norm\_temp1\_comm}[\textit{i}][\textit{j}]
  \item \texttt{unit \textit{norm\_temp2\_comm}[\textit{i}][\textit{j}]} \# \textit{q}, \texttt{norm\_temp2\_comm}[\textit{i}][\textit{j}]; \texttt{norm\_temp2\_comm}[\textit{i}][\textit{j}]
\end{itemize}

\begin{itemize}
  \item \texttt{assign \textit{CG\_FM} (CG\_\textit{parameters}, \textit{q}, \textit{rho}, \textit{aux}, \texttt{norm\_temp1}, \texttt{norm\_temp2}) \rightarrow (\textit{q}, \textit{rho}, \textit{aux}, \textit{norm\_temp1}, \texttt{norm\_temp2})
  \item \texttt{assign \textit{CG\_FM} (CG\_\textit{parameters}, \textit{q}, \textit{rho}, \textit{aux}, \texttt{norm\_temp1}, \texttt{norm\_temp2}) \rightarrow (\textit{q}, \textit{rho}, \textit{aux}, \textit{norm\_temp1}, \texttt{norm\_temp2})
\end{itemize}
C.4 A Aplicação Simulada LU

C.4.1 Esqueleto Exchange

component Exchange_1b < xdiv , ydiv > with

iterator m range [0..(ydiv -1)]
iterator n range [0..(xdiv -1)]

interface Exchange_1b # (from_north**, from_west**, from_south**, from_east** :: UArray (Int,Int) Double)
→ (to_north**, to_east**, to_south** to_west** :: UArray (Int,Int) Double)

behaviour: repeat (seq (seq {from_north?, from_west?} until <from_north & from_west &
to_south & to_east>,
repeat seq {from_south?, from_east?} until <from_south & from_east &
to_north & to_west>,
)
) until itmax

C.4.2 Esqueleto Exchange_3b

component Exchange_3b < xdiv , ydiv > with

iterator m range [0..(ydiv -1)]
iterator n range [0..(xdiv -1)]

interface Exchange_3b # (from_north*, from_south*, from_east*, from_west* :: UArray Int Double)
→ (to_north*, to_south*, to_east*, to_west* :: UArray Int Double)

behaviour: repeat (seq {from_north?, from_west?} until <from_north & from_west &
to_south & to_east>,
repeat seq {from_south?, from_east?} until <from_south & from_east &
to_north & to_west>,
)
until itmax

C.4.3 Esqueleto Exchange_4

component Exchange_4 < xdiv , ydiv > with

iterator m range [0..(xdiv -2)]
iterator n range [0..(ydiv -2)]

interface Exchange_4

interface Exchange_4_Null specializes Exchange_4

interface Exchange_4_Border # (in::UArray Int Double) → (out::UArray Int Double)
behaviour: seq {out!in?} specializes Exchange_4

interface Exchange_4_Corner_NW # (in1, in2::UArray Int Double) → ()
behaviour: seq {in1?in2?} specializes Exchange_4

71
interface IExchange<Corner_SE> # () → (out1, out2::UArray Int Double)
    behaviour: seq (out1, out2) specializes IExchange_f

/° unit h0[i][j] # IExchange<Null />
unit h0[0][0] # IExchange<Corner_NW
unit h0 XRDV -1][YRDV -1] # IExchange<Corner_SE

/° unit h0[i][j] # IExchange<Border />
/° unit h0[i][j] # IExchange<Border />
/° unit h0[i][j] # IExchange<Border />

/° connect h0[i][j] → out to h0[i][j] ← in /
/° connect h0 XRDV -1][i] → out to h0 XRDV -1][j] ← in /
/° connect h0[i][j] → out to h0[i][j] ← in /

C.4.4 Esqueleto Exchange_5

component Exchange_5 < XRDV, YRDV > with

  iterator m range [0..(YRDV -1)]
  iterator n range [0..(XRDV -1)]
  iterator i range [1..(YRDV -2)]
  iterator j range [1..(XRDV -2)]

interface generalization IExchange_5

interface IExchange<Null> specializes IExchange_5
interface IExchange<Top> # (in::UArray Int Double) → () behaviour: in? specializes IExchange_5
interface IExchange<Bottom> # () → (out::UArray Int Double) behaviour: out! specializes IExchange_5
interface IExchange<Side> # (in::UArray Int Double) → (out::UArray Int Double)
    behaviour: seq (out! ? in?) specializes IExchange_5

/° unit h1[i][j] # IExchange<Null />
unit h1[0][0] # IExchange<Top>
unit h1[0] YRDV -1] # IExchange<Top>
unit h1[i][j] XRDV -1][YRDV -1] # IExchange<Bottom>
unit h1[i][j] XRDV -1][XRDV -1] # IExchange<Bottom>

/° unit h1[i][j] # IExchange<Side>
unit h1[i][j] YRDV -1] # IExchange<Side>

/° connect h1[i][j] → out to h1[i][j] ← in /
/° connect h1[i][j] YRDV -1] → out to h1[i][j] YRDV -1] ← in /

C.4.5 Esqueleto Exchange_6

component Exchange_6 < XRDV, YRDV > with

  iterator m range [0..(YRDV -1)]
  iterator n range [0..(XRDV -1)]
  iterator i range [1..(YRDV -2)]
  iterator j range [1..(XRDV -2)]

interface generalization IExchange_6

interface IExchange<Null> specializes IExchange_6
interface IExchange<Left> # (in::UArray Int Double) → () behaviour: in? specializes IExchange_6
interface IExchange<Right> # (out::UArray Int Double) behaviour: out! specializes IExchange_6
interface IExchange<Side> # (in::UArray Int Double) → (out::UArray Int Double)
    behaviour: seq (out! ? in?) specializes IExchange_6

/° unit h1[i][j] # IExchange<Null />
unit h1[0][0] # IExchange<Left>

72
C.4.6 Componente LU (Esqueleto de Aplicação)

Componente LU <nprocs, problem_size, dt, default, itmax> with

# define # log2(nprocs)/2
# define nV (ipow2(int(2^2 == log2(nprocs)), d, d + 1))
# define MPI_LEFT

use Skeletons.MPI (AllReduce, Bcast)
use Exchange, lb, Exchange, lb, Exchange, 4, Exchange, 5, Exchange, 6
use LU FM ↔ LU Functional Module

interface LU (ipr, inorm, itmax, nx0, ny0, nz0, dt, omega, tolsd, rsnm*, errnm, frc1, frc2, frc3, rsd1, rsd0, u1, phis, phiver, phivor)

where: ipr, inorm, itmax, nx0, ny0, nz0 @ Int #
tolrd @ Bcast MyArray1d, #
tolrd, errnm, tolsd, rsnm* @ AllReduce MyArray1d #
frc1, frc2, frc3 @ Bcast Double #
rsdnm; tolsd, rsdnm; tolsd, tolsd, rsnm* @ AllReduce Double #

behavior seq { do ipr; do inorm; do itmax; do nx0; do ny0; do nz0; do dt; do omega; do tolsd; do rsdnm; do rsnm; do errnm; do frc1; do frc2; do phis; do phiver; do phivor }

unit ipr, comm ; assign BCAST < XDIV * YDIV > to ipr, comm
unit inorm, comm ; assign BCAST < XDIV * YDIV > to inorm, comm
unit itmax, comm ; assign BCAST < XDIV * YDIV > to itmax, comm
unit nx0, comm ; assign BCAST < XDIV * YDIV > to nx0, comm
unit ny0, comm ; assign BCAST < XDIV * YDIV > to ny0, comm
unit nz0, comm ; assign BCAST < XDIV * YDIV > to nz0, comm
unit dt, comm ; assign BCAST < XDIV * YDIV > to dt, comm
unit omega, comm ; assign BCAST < XDIV * YDIV > to omega, comm
unit tolsd, comm ; assign BCAST < XDIV * YDIV > to tolsd, comm
unit rsnm, comm ; assign BCAST < XDIV * YDIV > to rsnm, comm
unit rsdnm, comm ; assign ALLREDUCE < XDIV * YDIV, MPI_DOUBLE, MPI_DOUBLE #
comm rsnm, comm
unit errnm, comm ; assign ALLREDUCE < XDIV * YDIV, MPI_DOUBLE, MPI_DOUBLE #
comm errnm, comm
unit frc1, comm ; assign ALLREDUCE < XDIV * YDIV, MPI_DOUBLE, MPI_DOUBLE #
comm frc1, comm
unit phiver, comm ; assign Exchange, { do xdiv, do ydiv > to phiver, comm
unit phivor, comm ; assign Exchange, { do xdiv, do ydiv > to phivor, comm
unit phis, comm ; assign Exchange, { do xdiv, do ydiv > to phis, comm
unit frc2, comm ; assign ALLREDUCE < XDIV * YDIV, MPI_DOUBLE, MPI_DOUBLE #
comm frc2, comm
unit frc3, comm ; assign ALLREDUCE < XDIV * YDIV, MPI_DOUBLE, MPI_DOUBLE #
comm frc3, comm

// unify ipr, comm p[n][m] # ipr ,
inorm, comm p[n][m] # inorm ,

itmax, comm p[n][m] # itmax ,
dt, comm p[n][m] # dt ,
tolrd, comm p[n][m] # tolr ,
nz0, comm p[n][m] # nz0 ,
ny0, comm p[n][m] # ny0 ,
rsdnm, comm g1[n][m] # rsnm ,
u1, comm g1[n][m] # u1 ,
rsdnm_comm.p[n][m]  # rdnm
ssor_comm.bigLoop[n][m]  # rsl1
errnm_comm.p[n][m]  # errnm
phis_comm.h0[n][m]  # phis
frc1_comm.p[n][m]  # frc1
phiver_comm.h1[n][m]  # phiver
frc2_comm.p[n][m]  # frc2
phihor_comm.h2[n][m]  # phihor
frc3_comm.p[n][m]  # frc3

to lu[n][m]  # IPR # INORM # ITMAX # MUX # NUA # DT # OMEGA # TOLRSN # RSDNM # ERRNM
   # FREC1 # FREC2 # FREC3 # RS1 # RS0 # A1 # PHIS # PHIVER # PHIVOR /

assign.1. FM(PARMETERS: ipr, inorm, itmax, mua, dta, omega, tolrsd, rdsn, errnm, 
   frc1, frc2, frc3, rsd1, rsd0, a1, phis, phiver, phivor) 
    → (ipr, inorm, itmax, mua, dta, omega, tolrsd, rdsn, errnm, 
    frc1, frc2, frc3, rsd1, rsd0, a1, phis, phiver, phivor)