Coordinating Functional Processes with Haskell#

F. H. Carvalho Jr.*
Centro de Informática
Universidade Federal de Pernambuco
R. Prof. Luiz Freire, s/n
Recife, Brazil
fhcj@cin.ufpe.br

R. M. F. Lima
Associação de Ensino Superior de Olinda
Av. Transamazônica, 405
Olinda, Brazil
ricardo@cs.chalmers.se

R. D. Lins†
Depto. de Eletrônica e Sistemas
Universidade Federal de Pernambuco
R. Acad. Hélio Ramos, s/n
Recife, Brazil
rdl@ee.ufpe.br

ABSTRACT

This paper presents Haskell#, a parallel functional language based on coordination. Haskell# supports *lazy stream communication* and facilities, at coordination level, to the specification of *data parallel programs*. Haskell# supports a clean and complete, semantic and syntactic, separation between *coordination* and *computation* levels of programming, with several benefits to parallel program engineering. The implementation of some well-known applications in Haskell# is presented, demonstrating its expressiveness, allowing for elegant, simple, and concise specification of any static pattern of parallel, concurrent or distributed computation.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Concurrent Programming—*Distributed programming, Parallel Programming*; D.3.2 [**Programming Languages**]: Languages Classifications—*Concurrent, distributed and parallel languages, Applicative (functional) languages*

General Terms

Languages

Keywords

Parallel Functional Programming, Coordination, Haskell, Parallel Software Engineering

1. INTRODUCTION

Haskell[27] is a general purpose, pure functional programming language incorporating recent innovations in programming language design. It has now become *de facto* standard for the non-strict (or lazy) functional programming community, with several compilers available.

The idea of parallel functional programming dates back to 1975 [23, 12] when Burge [5] suggested the technique of evaluating function arguments in parallel, with the possibility of functions absorbing unevaluated arguments and perhaps also exploiting speculative evaluation. In general, parallelism obtained from referential transparency in pure functional languages is of too fine granularity, not yielding good performance. The search for ways of controlling the degree of parallelism of functional programs by means of automatic mechanisms, either static or dynamic, had little success [15, 26, 18]. Compilers that exploit implicit parallelism have been facing difficulty to promote good load balancing amongst processors and to keep communication costs low. On the other hand, explicit parallelism with annotations to control the demand of evaluation of expressions, creation/termination of processes, sequential and parallel composition of tasks, and mapping of these tasks onto specific processors have been proposed by many authors [6, 16, 28, 32] with good performance results. But, in general, in those approaches *computation* and *communication* are interwinded, not allowing reasoning about these elements in isolation.

The coordination paradigm [11] is an attempt to separate concurrent programs in two components: *coordination* and *computation*. There is a consensus that the abstraction of concurrent programming in these two levels provides higher degrees of *modularity, composability, generality, portability, possibility of formal analysis of parallel programs*, and *support for heterogeneous computing* [11, 10]. The relation of this new paradigm to parallel functional programming can be seen from two perspectives. In the first one, coordination is an important tool for facing the main difficulties of parallel functional programming [3, 19]. In the second one, higher-order and non-strictness turn functional languages adequate to the specification of coordination of tasks [8, 24].

Haskell#[7, 21] is a general concurrent extension to *Haskell* based on *coordination* aimed at distributed memory parallel architectures[1]. Haskell# offers a *clean* and *complete* sepa-

*Also at Departamento de Estatística e Informática, Universidade Católica de Pernambuco, Recife, Brazil.

†Sponsored by CNPq grants 463858/00-0 and 523974/96-5.

2002 ACM Symposium on Applied Computation Madrid, Spain

SAC 2002, Madrid, Spain

[1]At present, Haskell# has been implemented for SP2 and CoP's (clusters of PC's) architectures.

ration between sequential computation and coordination, at semantic and syntactic levels, by the use of an orthogonal configuration language (HCL) for coordination of *Haskell* sequential processes. The non-strict nature of *Haskell* is essential to allow generality and total transparency between coordination and computation, a property known as *process hierarchy*. Many benefits of Haskell$_{\#}$ parallel programming engineering are due to this property. Process hierarchy allows for independent development of program components, reusing of existing and tested *Haskell* modules, formal analysis of parallel programs based on Petri net formalism, efficient implementation over virtually any distributed memory parallel architecture, etc. Haskell$_{\#}$ model of concurrency is inspired by Occam [17], a language based on Hoare's CSP (*Calculus of Sequential Processes*)[14]. The decision for following the Occam computational model had as goal to make possible the automatic analysis of formal properties and, thus, to help the programmer to reason about the application under development. An environment to analyze formal properties of Haskell$_{\#}$ applications using Petri nets is described in [21].

The structure of this paper comprises six sections. Section 1 is this introduction. In Section 2, the Haskell$_{\#}$ parallel programming environment is described. In Section 3, Haskell$_{\#}$ is compared to other parallel functional languages based on coordination. In Section 4, some Haskell$_{\#}$ programs are presented to demonstrate its expressiveness. Conclusions and lines for further work are discussed in Section 5. Bibliography is presented in the Section 6.

2. THE HASKELL$_{\#}$ LANGUAGE

Haskell$_{\#}$ provides an integrated coordination environment for developing, simulating and analyzing formal properties of generic concurrent systems. Applications are structured in two layers. The sequential one relates to *functional modules*, sequential *Haskell* programs. The communication level (HCL, or Haskell$_{\#}$ Coordination Language) "glues" functional modules together forming a network of processes, later mapped onto processors of a distributed parallel architecture. Figure 1 depicts the Haskell$_{\#}$ programming environment. After writing a Haskell$_{\#}$ program, there are two possibilities: either to generate its executable code, or to translate it into Petri nets. In the former case, the system executes in a distributed memory environment. In the later case, it is possible to analyze formal properties of the topology of the network structure using INA [29], a Petri net simulator and analyser, helping programmers to reason about the system.

In what follows, we show how Haskell$_{\#}$ programs are specified at coordination level, presenting informally the syntax and semantics of the HCL constructors. At the computational level, standard *Haskell* programming is used.

2.1 The Structure of HCL Configurations

A HCL program is composed by four sections, which respectively declares *module interfaces*, *channels*, *distribution of processes onto processors*, and *activations of functional processes*. Each kind of HCL declaration is described below. An application has also a header in which its *name* and *interface* are declared. The declaration below shows an example of a header:

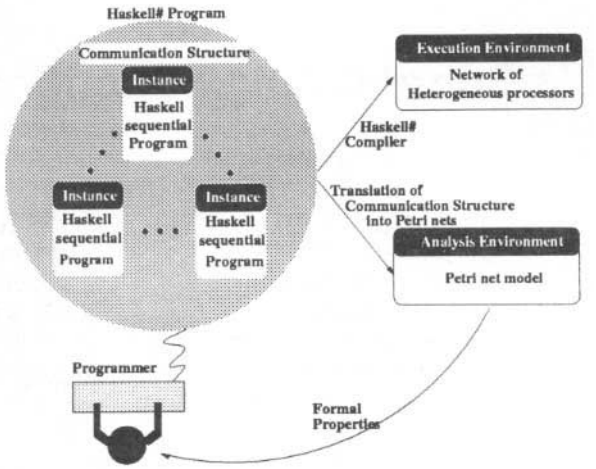

Figure 1: Haskell$_{\#}$ Programming Environment

$$\textbf{application App} < p_1, \cdots, p_k > (t_1, t_2, \cdots, t_n) \rightarrow (u_1, u_2, \cdots, u_m)$$

It specifies that an application, called App, has n dynamic arguments (t_1 to t_n) and m outputs (u_1 to u_m). Application interfaces will permit, in the near future, to include in Haskell$_{\#}$ support for hierarchical composition of Haskell$_{\#}$ programs. The idea is to generalize the notion of instantiation of processes, allowing them to be instantiated from configurations, similarly to the one implemented in K2[1], providing new opportunities for reuse, now at coordination level. Static parameters (p_1 to p_n) allow support for parameterized applications. The compiler must replace actual values, provided at compile time, for the formal parameters in the HCL code.

2.2 Instantiating Functional Processes

In Haskell$_{\#}$, functional processes are instances of functional modules. *Module abstractions* must configure the interface of processes being instantiated from a functional module. Input and output ports enable processes to communicate amongst themselves. Ports are strongly typed and of ground type, either basic (integer, floating points etc.) or structured over basic ones (lists of integers, trees of booleans, etc.). A module abstraction is defined as following:

module interface I **from** M
  input $ip_1 :: t_1$
  input $ip_2, ip_3[1..n] :: t_2$
  input $f\ (ip_4, ip_5, ip_6) :: t_3$
  output $op_1 :: t_4$
  output $op_2, op_3 :: t_5$
  output $g\ (op_4[1..n]) :: t'_6$
  instances $m[1..8]$

Here, the module interface I defines an interface for functional module M. This module declares a function *main* with the following interface:

$$main :: t_1 \rightarrow t_2 \rightarrow t'_3 \rightarrow \mathbf{IO}(t_4, t_5, t_6)$$

Note that t_1, t_2, t'_3, t_4, t_5, t_6, and t'_6 are *Haskell* types. Each process $m[i]$, for i from 1 to 8, instantiated from M using the interface I, has $n+5$ input ports, associated with the three arguments of its *main* function (one input port declaration for each argument), and $n+3$ output ports associated with the three elements of the tuple returned by *main* (one output port declaration for each tuple element). The input port ip_1 is associated with the first argument. The input ports ip_2 and $ip_3[i]$, for i from 1 to n, where n is a parameter, associated with the second argument, yields a non-deterministic choice. The actual value of the argument comes from the first port to be ready to receive a value[2]. Function f receives a list of values of type t_3 ($[t_3]$), whose elements are received from ports ip_4, ip_5 and ip_6, and transforms it into a single value of type t'_3, which is passed to the third argument of *main*. The *output port* op_1 sends the value of the first tuple element. Ports op_2 and op_3 model non-deterministic output ports. The value is sent through one of the ports, chosen non-deterministically, that are connected to an input port of another process that is ready to receive a value. Ports $op_4[1]$ to $op_4[n]$ send the value of the third element of the tuple. The function g receives as argument this value and the number of ports through which the value must be *distributed*. Its result type is a list of values of type $[t'_6]$. Each element of this list is then sent through a port. During process execution, the lazy semantics of functional processes guarantees that an input port is read only when its associated argument value is required for computation of some expression. A process remains blocked until a value is available on the channel (*synchronous communication*).

2.3 Communication Channels

Similarly to Occam[17], Haskell$_\#$ channels are *point-to-point*, *unidirectional* and *synchronous*. *Strict communication semantics* restricts values exchanged between processes to those already evaluated. Using the HCL **connect** constructor, a channel is statically declared through the connection of two ports from different processes, of the same type and opposite direction. For instance, observe the following piece of HCL code:

```
connect p0.a to p1.b
connect master.c[i] to slave[j].c stream 100,
        for i=1..n, j=1..m such that i <= j
```

There are two channel declarations. The first one defines a channel that connects output port a of process p_0 to input port b of the process p_1. The channel type is the type of the involved ports. The second one uses an indexed form to declare $m \times n$ channels which connect output ports c[i] of process *master* to input ports c of the processes *slaves*[j], for i from 1 to n and j from 1 to m, such that i is less than j. These ports are connected through a *lazy stream* channel, which may transmit elements of a list one at a time, whenever it is evaluated. *Lazy stream* channels are implemented using *Haskell* lazy lists and allow transparent process interaction during computation. The number 100, in the declaration above, specifies that the list must be sent in blocks of that size, controlling communication *overheads* and granularity. If the block size is omitted, size one is assumed by default.

Haskell$_\#$ neither allows dynamic channel creation nor full-duplex communication. One could argue that this is too restrictive. However, our emphasis is to provide a model of channel that makes possible to statically analyze formal properties of the process network. Besides that, strict rules force programmers to have a better understanding of the system and to specify precisely what they want to do.

Non-determinism and Streams

If stream channels connect n ports (either input or output) involved in a non-deterministic communication, it is important to note that a non-deterministic choice is made for each element in the list to be transmitted or received non-deterministically. This allows the merge of several streams into a single list on input side, or the split of a list in several streams on the output side. In the program described in Section 4, we use this feature to model management of demands and responses in a client/server application.

Indexing and Parameterization

The parameterization and support for indexed referencing of processes and ports, possibly using variables in conjunction with **for** and **such that** clauses, allow the management of HCL code of a large number of processes and the easier representation of complex network topologies.

2.4 Initialization, Execution, and Termination

Haskell$_\#$ programs start by the explicit activation of all of its processes in **start** declarations. Execution is demand driven by the arguments on input ports. Values may be passed explicitly to processes. Examples of **start** declarations follow:

```
start fp1 ? ?
start fp2 [1,2,3,4] ? b ? ?
start repetitive fp2 ? f, where f in  even_values
```

The first declaration activates process $fp1$, which will receive the value from its two input ports. The first and third ports of process fp_2 receive values explicitly. They are not connected to a channel. The third **start** declaration is an example of a repeated activation. Process $fp2$ will be activated repeatedly until the hole program ends. On each activation, it demands arguments on its first port, while, on the second one, values are explicitly given from an infinite list, generated by function *even_values*. In the **where** clause of repetitive processes, it is a necessary condition that the list or lists declared be infinite. The use of the symbol ? as a list element means that the corresponding value must be obtained through communication, i.e., not explicitly. If an element of the list is ?, it must be received from the port. This feature makes possible to avoid deadlocks on cyclic networks by explicit activation of some processes in the cycle. The solution of the *dining philosophers problem* presented in Section 4.1 use this approach.

Haskell code can be written inside HCL code using #'s delimiters. For instance, the function *even_values* is declared in the following way:

[2]The semantic of input ports non-deterministic choice is inspired in the PRI ALT constructor semantic in OCCAM.

```
#
even_values :: [Int]
even_values = [1,3,..]
#
```

Stream communication and Repetitive Processes

In Haskell$_{\#}$, a port of type $[t]$ of a *non repetitive* process can be connected to a port of type t of a *repetitive* process. On each activation, the repetitive process will consume an element of the *stream*.

Termination Condition

A Haskell$_{\#}$ application terminates whenever all of its non repetitive processes terminate. Thus, a program with only repetitive processes never terminate.

2.5 Placement of Functional Processes

The execution environment of Haskell$_{\#}$ applications is a network of processing *nodes*, onto which functional processes are statically mapped. Programmers allocate processes in groups, in which processes execute concurrently. Before task allocation, nodes are classified according to their *features*, such as *node processor speed* (*fast* or *slow*), *amount of memory available* (*large* or *thin*) or *communication speed* (*high* or *low*), in file `node.classes`. In the `node.id` file, each node must be assigned to at most one feature of each class. The main goal of classification is to model heterogeneity of the execution environment, allowing the HCL Compiler to determine the best allocation of processes to nodes. Functional processes remain mapped onto a node during all their life. The **alloc** constructor used to define the mapping scheme is exemplified below:

```
alloc (wide, fast) p0, p1, p2
alloc (slow) p3[i], for i = 1..n
alloc (slow) (p4[i], for i=1..n)
```

Processes p_0, p_1, and p_2 are allocated on a processor with features *fast* and *wide* from classes *speed* and *memory*, respectively, while n processes $p_3[i]$ are mapped onto distinct nodes with feature *slow* of class *speed*. There is no reference to the class *memory*, allowing a *wide* or *thin* node to be allocated for p_3. In the last declaration, the n processes $p_4[i]$ are mapped onto the same node.

2.6 A Brief History of Haskell$_{\#}$ Evolution

Haskell$_{\#}$ is still an evolving language. In its first version[22], message passing primitives, *send* (!) and *receive* (? =), were defined in *Haskell*, on top of the IO monad. In order to allow the analysis of the *Haskell#* programs using Petri nets, message passing primitives were eliminated from the computation language [7, 21]. This kept *process hierarchy* but decreased the expressive power of Haskell$_{\#}$ in specifying some common static patterns of parallel and concurrent computation. The use of *stream* communication made possible for Haskell$_{\#}$ to be as expressive as in its original version without any loss of process hierarchy. The Haskell$_{\#}$ version presented here is the first to include *streams*, and also new facilities to build data parallel programs and a revised notion of *initialization* and *finalization* of programs.

3. RELATED WORK

The relation between coordination and parallel functional programming can be seen from two perspectives. In the first one, coordination is considered an important tool for parallel functional languages and models, because of its ability to abstract parallel concerns from specification of computation. Eden[2] and Caliban[19, 31] are examples of languages focused on these ideas. In the last one, higher-order and non-strict style of functional programming has been seen as an powerfull way to specify coordination amongst tasks. SCL[8] and Delirium[24] are examples of languages that use the functional paradigm at coordination level. Haskell$_{\#}$ belongs to the first category. Likewise Eden and Caliban, it uses *Haskell* for specifying computation, assuming a static network where functional processes communicate through point-to-point and unidirectional channels. Below, we discuss the most important points which distinguish Haskell$_{\#}$ from the other known parallel functional languages based on coordination:

- *The adoption of a configuration based*[3] *[20] coordination language (HCL)* provides complete separation on the construction of computational code (pure *Haskell*) from coordination one (HCL). In *Eden* and *Caliban*, examples of embedded coordination languages, primitives extend *Haskell* syntax for "gluing" processes to the coordination medium, while, in Haskell$_{\#}$, HCL is orthogonal to *Haskell*. This Haskell$_{\#}$ feature allows the parallelization of sequential pre-existent *Haskell* programs and yields independent specification and development of computational and coordination code, reducing development costs by code reuse. The ability for composing programs from parts also makes Haskell$_{\#}$ more suitable for larger scale parallel programming than *Eden* and *Caliban* [10, 9] and forces programmers to adopt a *coarse grain* view of parallelism, useful in *high-latency* distributed architectures, such as *clusters* of PC's.
- *The modelling of parallel architectures.* It is a widely acknowledged fact that generic *mapping* of processes to processors is a difficult problem to be treated automatically. The mechanisms for this purpose, either dynamic or static, are not efficient at all instances. We decided to follow a *static* and *explicit* approach in Haskell$_{\#}$ for allocation of processes, similarly to Caliban. The only difference is that Haskell$_{\#}$ makes possible to model both processes needs for optimal execution and architecture characteristics. The programmer is then responsible to find explicitly the best mapping between functional processes and processors using these information.
- *The analysis of formal properties using Petri nets.* When developing Haskell$_{\#}$, one of our main concerns was to support the analysis of formal properties of programs. A compiler that translates HCL into INA[29], a Petri net analysis tool, was developed by Lima[21].
- *The easy and efficient implementations.* Unlike *Eden* and *Caliban*, Haskell$_{\#}$ needs no run-time system support. It can be easily implemented by gluing a fast

[3]The configuration paradigm was developed in the context of specification of distributed systems[20].

message passing library to a state-of-the-art sequential *Haskell* functional compiler[4]. Assuming that Haskell$_\#$ applications are coarse grain, one can take advantage of the best technology for compilation of sequential functional programs.

- *Generality.* Haskell$_\#$ was developed to give appropriate support to the specification of general *concurrent* systems in a unified way. *Caliban*, *Delirium* e *SCL* are well suited for parallel applications, but turn difficult to specify some kinds of *assynchronous* applications, such as *client/server* ones, for example. Eden has features for implementation of *reactive systems*, like non-deterministic operations and *dynamic reply channels*, but has no explicit concerns about distribution.

4. HASKELL$_\#$ EXAMPLES

Following the terminology defined in [30], we distinguish between two kinds of concurrent systems: *transformational* and *reactive*. The first one relates to systems that receive some input and yield an output at its end, while the second one relates to systems where the central task is not to compute a result, but to maintain some interaction with its environment. In general, reactive systems never terminate. Operating systems and some kinds of control applications are examples of them. Parallel systems can be seen as concurrent systems with transformational behaviour. In general, applications belonging to this subset of concurrent systems have requirements of efficiency, because of their time constraints, while the others have requirements of structuring software or distribution. Distributed systems are concurrent systems where processes are distributed across a network of computers. Recently, the emerging of *cluster computing* technology has inspired a new distributed view of parallelism. This is one of the important facts which let us to believe that Haskell$_\#$ is a useful tool for programming on *clusters*.

For building parallel programs, Haskell$_\#$ offers a general functional view of parallelism. Using this approach, *data parallelism*[5] can be easily implemented by instantiation of several processes that perform the same task for processing of parts from some large data structure. The data can be distributed amongst processes and joined after parallel computation using special user-defined data parallel operators, programmed in *Haskell*, specified at configuration level (HCL) (see the syntax of declaration of ports in Section 2.2). We intend to define, in the near future, a set of pre-defined data parallel operators to be used in HCL programs. Indexing notation allows referencing several processes concisely, a requirement of data parallel programming environments.

The explicit and static notion of processes communicating through a network is well suited to specification of general concurrent systems. Several languages, notably based on configuration, use this approach to deal with distributed systems[20], but not dealing with requirements of parallel ones. In this section, we present the implementation of some common concurrent applications in Haskell$_\#$. Our goal here is to demonstrate how expressive is Haskell$_\#$ to express well-known patterns of general static *concurrent* systems.

The Haskell$_\#$ code for the examples presented in this paper can be found at `www.cin.ufpe.br\~ fhcj\sac2002_sources`.

4.1 The Dining Philosophers Problem

The *Dining Philosophers* is a common synchronization problem from concurrency theory, stated as follows: a number of *philosophers* are seated around a table forming a circle, each one with a plate of spaghetti and two forks to eat it. Each fork is shared by two adjacent philosophers. The philosopher can only eat after getting the two forks in a certain order (left to right). Thus, if a philosopher is eating, the two adjacent ones must be thinking. If all the philosophers get their left forks at the same time, they will wait forever for the right one. This situation models *deadlock*, a state where all processes belong to a communicating group in a system are waiting for another process in the group.

The Haskell$_\#$ solution is very simple. Assume n philosopher processes connected in a ring, each one with two input ports for receiving the left and right forks and two output ones to give the forks to its left and right neighbours (See Figure 2). Philosophers are thinking when they have no forks or eating when they have both left and right forks. In `start` declaration, we distribute the forks amongst philosophers, such that the maximal number of philosophers will initiate eating. If there is an even number of philosophers, the remaining fork is given to the last one. The other philosophers will initiate thinking and will try to receive forks from their neighbours. It is not difficult to verify that this solution thereafter never deadlocks.

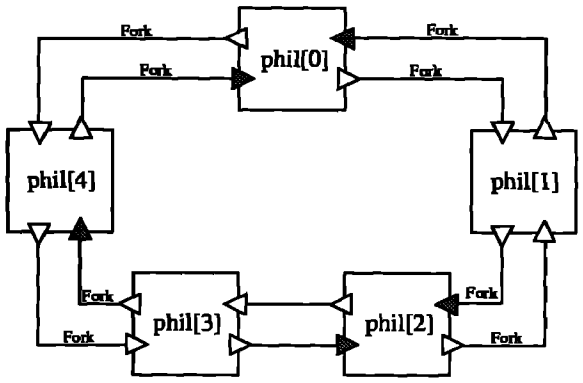

Figure 2: Dining Philosophers Haskell$_\#$ Network

The code for the functional module `Philosopher.hs` is:

```
-- MODULE FILE: Philosopher.hs
module Philosopher(main) where
```

[4]We have successfully used MPI and GHC, respectively, in our implementations

[5]*Data Parallelism* is considered the most common and efficient form of parallelism, providing high *scalability*, because the amount of parallelism exploited depends on the amount of data to be processed. However, it is less general than *functional parallelism*, in which it can be easily simulated.

```
data Fork = Fork
main ::  Fork → Fork → IO (Fork, Fork)
main rf lf = eat rf lf >> think >> (rf,lf)
```

The HCL configuration code to the network is:

```
application DiningPhilosophers<n>
module Philosopher
       input rforkin ::Fork
       input lforkin ::Fork
       output rforkout ::Fork
       output lforkout ::Fork
       instance phil[0..n-1]
for i=0..n-1
    connect phil[i].lfork to phil[(i+1) mod n]
    connect phil[(i+1) mod n].rfork to phil[i]
alloc phil[i], for i=0..n-1
start repetitive phil[i] ?  ?,
      for i=0..n-2 & (i mod 2 = 1)
start repetitive phil[i] f f,
      for i=0..n-2 & (i mod 2 = 0),
          where f in Fork:[?,?..]
start repetitive phil[n-1] f ?,
      for n mod 2=0, where f in Fork:[?,?..]
```

4.2 The Alternating Bit Protocol

The *Alternating Bit Protocol* (ABP) is a simple yet effective protocol for managing retransmission of lost messages on low-level implementations of messaging-passing model. Considering a receiver process A and a sender process B connected by two *stream channels*, the protocol ensures that whenever a message transmitted from B to A is lost, it is retransmitted.

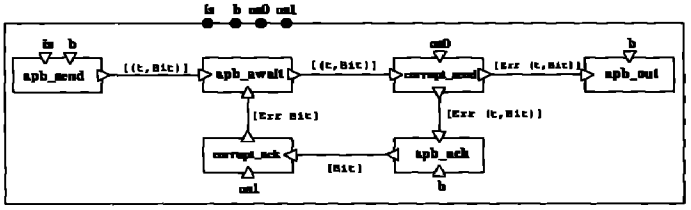

Figure 3: Alternating Bit Protocol Network

Figure 3 shows the process network of the ABP application specified in Haskell$_{\#}$. Processes *abp_send* and *abp_await* model the *sender* while processes *apb_out* and *apb_ack* model the *receiver*. Processes *corrupt_send* and *corrupt_ack* model two unreliable virtual channels that link the *sender* and *receiver*. The *apb_send* process sends a stream of messages to *apb_out* through its stream port *as*. Process *apb_await* guarantees that each message, received from stream port *as*, will be transmitted correctly, through stream port *as'*, assuming an unreliable medium for communication, modelled by *corrupt_send* process, which receives each message, from stream port *as*, and verifies if it is corrupted or not using an oracle stream of bits (os_0). Non corrupted messages are sent from *corrupt_send*, from stream ports bs_1 and bs_2, to *apb_out* and to *apb_ack* processes respectively, which receives messages from its stream ports named *bs*. Process *apb_ack* sends back an acknowledgement bit to *abp_await* for each message using its stream of bits port *cs*, through an unreliable medium, modelled by *corrupt_ack*. The order of this stream of bits is used by *apb_await* for verifying if the message was correctly sent or not. If the message was incorrectly transmitted, they are retransmitted by *apb_await*. We prove the correctness and formal properties of this program using Petri nets.

4.3 Generic Client/Server System

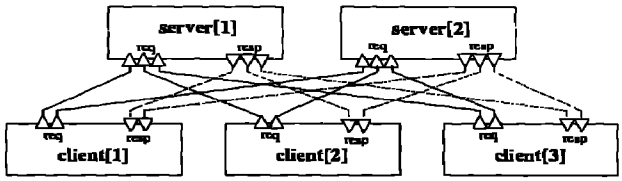

Figure 4: Client/Server System Haskell$_{\#}$ Network

Now, a typical example of a reactive system is presented: a Client/Server application, inspired by the generic one described in [4]. Here, *servers* process demands from *clients* processes (requisitions). Our purpose is to show how expressive is Haskell$_{\#}$ for representing reactive systems and to demonstrate use of non-determinism. Clients make an automatic choice amongst free servers when making a demand from their *req*[*i*] ports, each one associated to a server. They automatically receive responses from servers, through their *resp*[*i*] ports, one for each server too. All queries must identify the source client. Observe that there is no need for a manager process to decide in which server to process each query, as needed in the solution presented for Eden[4]. We model this using only the support for non-determinism of Haskell$_{\#}$.

4.4 Matrix Multiplication on the Mesh

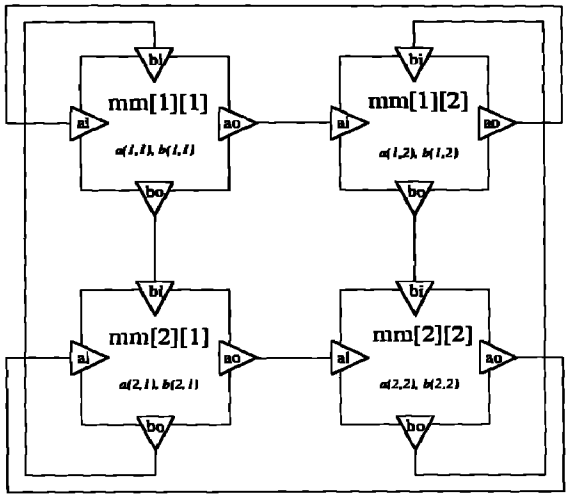

Figure 5: Matrix Multiplication on the Mesh: Haskell$_{\#}$ Network for a 2 X 2 Grid

The matrix multiplication on the mesh problem was extracted from [25]. This problem is stated as follows: *given*

two n X n matrices A and B, such that initially A[i, j] and B[i, j] reside in processor P[i, j], compute C = A.B, such that C[i, j] resides in processor P[i, j]. The solution consists of shifting of values between processes on the grid at each step until the final result is obtained. In its simple form, this solution has too fine granularity for obtaining good speedup, but it is very interesting for demonstrating how expressive Haskell$_{\#}$ is to specify a *systolic* pattern of parallel computation on a grid. Many parallel algorithms use this scheme. Figure 5 presents part of the network of processes of this application, emphasizing the grid of processes.

4.5 Photon Transport Simulation

MCP-Haskell$_{\#}$ is a parallel version of MCP-*Haskell*[13], a program that implements a simplified form of the Monte Carlo Particle Transport Problem, which involves simulating statistical behavior of particles (photons, neutrons, electrons, etc.) while they travel through objects of specified shapes and materials. MCP-*Haskell* is based on a Fortran code developed at Los Alamos over many years, called MCNP (Monte Carlo N-Particle)[33].

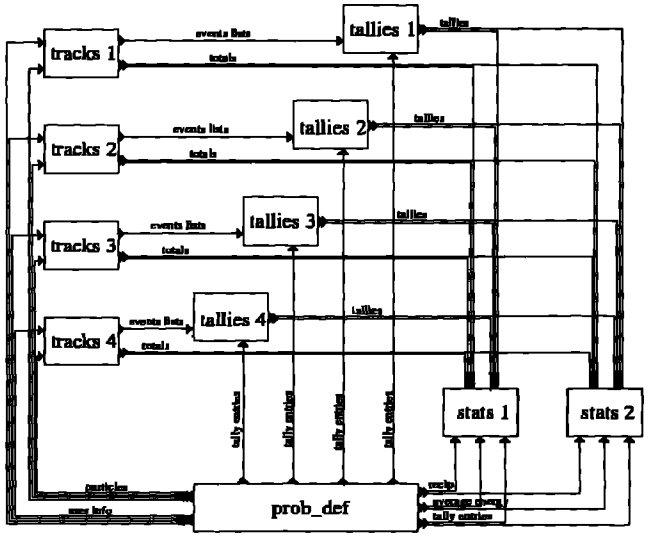

Figure 6: MCP-Haskell$_{\#}$ Process Network

MCP-Haskell$_{\#}$ makes use of the Haskell$_{\#}$ way to implement *data parallel* and *pipeline* parallelism. The experience with MCP-Haskell$_{\#}$ specification demonstrates how easy is to parallelize pre-existent *Haskell* code with Haskell$_{\#}$. There was no need for code rewritting or modification. It was only necessary some restructuring of the original programs in functional modules.

Data parallelism allows processing of particles in parallel, because each photon is independent and can be *tracked* and *tallied* independently. Each *track* and *tally* process composition computes a disjoint set of photons. The process *statistics* collects information yielded from tallying and computes statistical information about the simulation. A *pipeline* connects processes *prob_defs*, *tracks* and *tallies* for allowing them to operate in parallel. Stream communication is essential for this purpose, but the performance benchmark has shown that the gain in performance obtained from use of the pipe line is very poor, because almost all computation is performed by the *track* processes.

Figure 7 compares the *speedup* obtained for a problem instance of this application (solid line) to the linear (optimal) speedup (dashed line), when executing over a cluster of Linux PC's (6 Pentium II MMX 350MHz and 2 Pentium III 550MHz interconnected by Fast Ethernet (100Mbs) network interface). Here, Haskell$_{\#}$ uses MPI for process communication and GHC for compilation of functional processes.

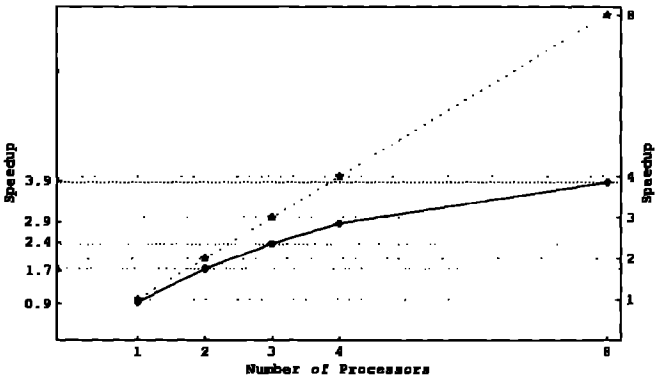

Figure 7: Speedup for MCP-Haskell$_{\#}$ executing on a Cluster of Linux PC's

5. CONCLUSIONS

In this paper, we presented Haskell$_{\#}$, a parallel extension to *Haskell* based on coordination. Its ability to compose parallel programs from sequential parts in a transparent way makes it suitable for the definition of the most important patterns of parallel, concurrent and distributed computation in a unified way. We showed the specification of some applications using Haskell$_{\#}$ to demonstrate its expressiveness. Our goal now is to implement large-scale applications of practical interest, for making performance benchmarking on clusters, and for improving Haskell$_{\#}$ model of coordination and its environment for program construction. At present, we started to develop an integrated environment for graphical specification of parallel and general concurrent systems, to be used in education (teaching of parallelism and concurrency in undergraduate courses) and also for practical development and management of complex applications. We also continue to investigate the applicability of Petri net formalism to the analysis of formal properties of Haskell$_{\#}$ parallel programs. At present, a new specification for translation of Haskell$_{\#}$ applications to Petri nets has been produced, now dealing with stream communication.

6. REFERENCES

[1] C. Abmann. Coordinating Functional Processes Using Petri Nets. *Implementation of Functional Languages, Springer-Verlag, LNCS 1268*, pages 162–183, Sept. 1997.

[2] S. Breitinger, R. Loogen, Y. Ortega Mallén, and R. Peña. The Eden Coordination Model for

Distributed Memory Systems. In *High-Level Parallel Programming Models and Supportive Environments (HIPS)*, 1997.

[3] S. Breitinger, R. Loogen, Y. Ortega Malln, and R. Peña. High-level Parallel and Concurrent Programming in Eden. In *Proceedings of APPIA-GULP-PRODE Joint Conference on Declarative Programming*, pages 213–224, June 1997.

[4] S. Briesmeister, R. Loogen, Y. Ortega Mallén, and R. Pe na. Eden: Language Definition and Operational Semantics. Technical report, FB Mathematik, Universität Marburg, 1998.

[5] W. H. Burge. Recursive Programming Techniques. *Addison-Wesley Publishers Ltd.*, 1975.

[6] F. Burton. Functional Programming for Concurrent and Distributed Computing. *Computer Journal*, 30(5):437–450, 1987.

[7] F. H. Carvalho Jr. *Haskell$_{\#}$*: Uma Extensão Paralela para Haskell. Master's thesis, Centro de Informática, Universidade Federal de Pernambuco, Jan. 2000.

[8] J. Darlington, Y. Guo, H. To, and J. Yang. Functional Skeletons for Parallel Coordination. *Lecture Notes in Computer Science*, 966:55–68, 1995.

[9] F. DeRemer and H. H. Kron. Programming-in-the-Large versus Programming-in-the-small. *IEEE Transactions on Software Engineering*, pages 80–86, June 1976.

[10] I. Foster. Compositional Parallel Programming Languages. *ACM Transactions on Programming Languages and Systems*, 18(4):454–476, 1985.

[11] D. Gelernter and N. Carriero. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, Feb. 1992.

[12] K. Hammond and G. Michaelson. Research Directions in Parallel Functional Programming. *Springer-Verlag*, 1999.

[13] J. Hicks, D. Chiou, B. S. Ang, and Arvind. Performance Studies of Id on the Monsoon Dataflow System. *Journal of Parallel and Distributing Computing*, 18:273–300, 1993.

[14] C. A. R. Hoare. Communicating Sequential Processes. *Prentice-Hall, C.A.R. Hoare Series Editor*, 1985.

[15] P. Hudak. Serial Combinators: "Optimal" Grains of Parallelism. *FPCA'85*, pages 382–399, Sept. 1985.

[16] P. Hudak. Para-Functional Programming in Haskell. *Parallel Functional Languages and Compilers, B. K. Szymanski, Ed. ACM Press, New York*, pages 159–196, 1991.

[17] Inmos. Occam 2 Reference Manual. *Prentice-Hall, C.A.R. Hoare Series Editor*, 1988.

[18] O. Kaser, C. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Equals - A Fast Parallel Implementation of a Lazy Language. *Journal of Functional Programming*, 7(2):183–217, Mar. 1997.

[19] P. Kelly. Functional Programming for Loosely-coupled Multiprocessors. *Research Monographs in Parallel and Distributed Computing, MIT Press*, 1989.

[20] J. Krammer. Distributed Software Engineering. In IEEE Computer Society Press, editor, *Proc. 16th IEEE International Conference on Software*, 1994.

[21] R. M. F. Lima. *Haskell$_{\#}$ - Uma Linguagem Funcional Paralela - Ambiente de Progrmação, Implementação e Otimização*. PhD thesis, Centro de Informática, UFPE, July 2000.

[22] R. M. F. Lima, F. H. Carvalho Jr., and R. D. Lins. *Haskell$_{\#}$*: A Message Passing Extension to Haskell. *CLAPF'99 - 3rd Latin American Conference on Functional Programming*, pages 93–108, Mar. 1999.

[23] R. D. Lins. Functional Programming and Parallel Processing. *2nd International Conference on Vector and Parallel Processing - VECPAR'96 - LNCS 1215 Springer-Verlag*, pages 429–457, Sept. 1996.

[24] S. Lucco and O. Sharp. Delirium: An Embedding Coordination Language. In ACM Press, editor, *Proceedings of Supoercomputing'90*, 1990.

[25] U. Manber. *Introduction to Algorithms: A Creative Approach*, chapter 12, pages 375–409. Addison-Wesley, Reading, Massachusetts, Oct. 1999.

[26] S. L. Peyton Jones, C. Clack, and J. Salkild. GRIP - A High-Performance Architecture for Parallel Graph Reduction. *FPCA'87: Conference on Functional Programming Languages and Computer Architecture - Springer-Verlag LNCS 274*, pages 98–112, 1987.

[27] S. L. Peyton Jones and J. Hughes. Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language. Feb. 1999.

[28] M. J. Plasmeijer and M. van Eekelen. Functional Programming and Parallel Graph Rewriting. *Addison-Wesley Publishers Ltd.*, 1993.

[29] S. Roch and P. Starke. Manual: Integrated Net Analyzer Version 2.2. *Humboldt-Universität zu Berlin, Institut für Informatik, Lehrstuhl für Automaten- und Systemtheorie*, 1999.

[30] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21:413–510, 1989.

[31] F. Taylor. Parallel Functional Programming by Partitioning. *PhD Thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London*, Jan. 1997.

[32] P. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. P. L. Jones. GUM: A Portable Parallel Implementation of Haskell. *PLDI'96 - Programming Languages Design and Implementation*, pages 79–88, 1996.

[33] D. J. Whalen, D. E. Hollowell, and J. S. Hendriks. MCNP: Photon Benchmark Problems. Technical Report LA-12196, Los Alamos National Laboratory, 1991.