Parallelizing MCP-Haskell for Evaluating Haskell$_{\#}$ Parallel Programming Environment

F. H. Carvalho Jr.[1], R. D. Lins[2], R. M. F. Lima[3 4]
[1] Centro de Informática, Universidade Federal de Pernambuco, Brasil
{fhcj@cin.ufpe.br}
[2] Departmento de Eletrônica e Sistemas, Universidade Federal de Pernambuco, Brasil
{rdl@ee.ufpe.br}
[3] Department of Computer Science, Chalmers University of Technology, Sweden
{ricardo@cs.chalmers.se}
[4] CESBAM, Associação de Ensino Superior de Olinda, Brasil
{ricardo@cs.chalmers.se}

Abstract—

In this paper, we present the parallelization of a sequential functional implementation of a Monte Carlo Transport Problem, called MCP-*Haskell*[Hammes *et al.*, 1995], using *Haskell*$_{\#}$. This experiment gave us important feedback for evaluating *Haskell*$_{\#}$ features, helping us to answer some questions, like how expressive is *Haskell*$_{\#}$ for representing known parallel computational patterns, how easy it is to build large scale parallel programs in an elegant and concise way, and how efficient are *Haskell*$_{\#}$ programs. Based on our conclusions, we suggest new features to be incorporated in *Haskell*$_{\#}$ to improve its *expressiveness* and *performance*. We also present the performance figures for the MCP-*Haskell*$_{\#}$ benchmark.

Keywords— **Parallel Processing, Parallel Software Engineering.**

I. Introduction

Haskell[Peyton Jones & Hughes, 1999] 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 [Lins, 1996, Hammond & Michaelson, 1999] when Burge [Burge, 1975] 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[Hudak, 1985, Peyton Jones *et al.*, 1987, Kaser *et al.*, 1997]. 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 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 [Burton, 1987, Hudak, 1991, Plasmeijer & van Eekelen, 1993, Trinder *et al.*, 1996]. In general, in this approach *computation* and *communication* are interwinded, not allowing reasoning about these elements in isolation.

Haskell$_{\#}$[Lima *et al.*, 1999] is a parallel extension to *Haskell*, based on coordination approach [Gelernter & Carriero, 1992] and aimed at distributed memory parallel architectures[1]. It offers process modularity on top of communication structure of sequential computation components written in standard Haskell. The structure of the communication network is defined by *Haskell*$_{\#}$ *coordination language (HCL)*, also used for task-to-processor allocation. On these two levels (*coordination* and *computation*), programming can be performed at independent stages of development process, a characteristic that tends to reduce development costs and to increase reliability. Reusing of existing and previously tested (or formally) verified Haskell modules is also possible. *Haskell*$_{\#}$ is inspired by Occam [Inmos, 1988], a language based on Hoare's CSP (*Calculus of Sequential Processes*)[Hoare, 1985]. The decision for following the Occam computational model had as its 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 [Lima, 2000].

The structure of this paper comprises seven sections. Section I is this introduction. Section II describes *Haskell*$_{\#}$ parallel programming environment. Section IV briefly presents Monte Carlo photon transport problem. Section V, shows parallelization process of MCP-*Haskell*. Section VI, dis-

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

cusses proposals for improving $Haskell_{\#}$ environment, using MCP-$Haskell_{\#}$ to evaluate them. Section VII presents the performance figure for some versions of the MCP-$Haskell_{\#}$ incorporating ideas described in Section VI. Conclusions and lines for further work are presented in Section VIII.

II. $Haskell_{\#}$ Programming Environment and Applications

$Haskell_{\#}$ provides an integrated environment for developing, simulating and analyzing formal properties of parallel systems. Applications are structured in two levels (*process hierarchy*). The sequential one relates to *functional modules*, sequential $Haskell$ programs. The communication level "glues" together functional modules forming a network of processes, later mapped onto physical nodes of a parallel architecture. The coordination language called HCL ($Haskell_{\#}$ Coordination Language) is used for that. Figure 1 depicts the $Haskell_{\#}$ programming environment. After writing the program, there are two possibilities: either to generate the executable code, or to translate HCL programs into Petri nets. In the former case, the system executes in an environment composed by distributed memory processors (nodes). In the latter case, it is possible to analyze formal properties of the topology of the network structure, helping programmers to reasoning about the system.. The Petri net model is simulated and analyzed through the computational tool INA [Roch & Starke, 1999].

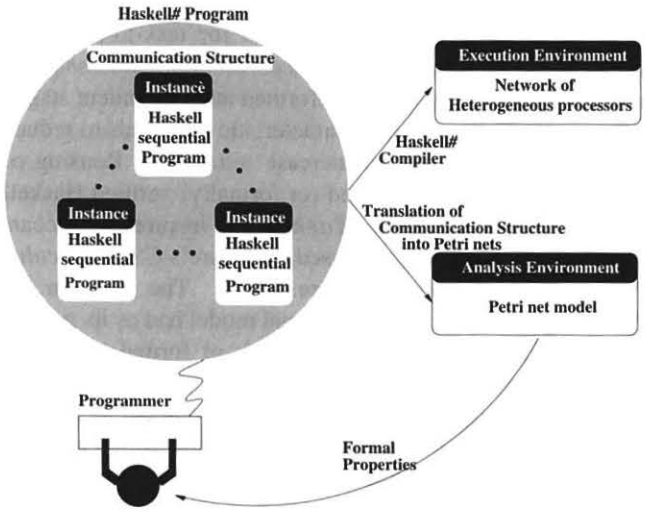

Fig. 1. $Haskell_{\#}$ Programming Environment

In the following sections, we describe how applications are specified on communication level, presenting the syntax and semantics of the HCL constructors.

A. Instantiating Functional Processes

In $Haskell_{\#}$, functional processes are instances of functional modules. Each one has its own interface, linked by communication ports allowing processes communicate with each other. *Direction* attributes specify if a port is used for *input* or *output*. Ports are strongly typed and of ground types, such as basic (integer, floating points etc.) and structured over basic ones (lists of integers, trees of booleans, etc.). In HCL, functional processes are defined by *module* declarations. For instance:

module M
  input $a :: t_1$, $\{b,\ c\} :: t_2$, $d :: t_3$
  output $e :: t_4$, $f :: t_5$
  instances m_1, m_2, m_3

define the interface of functional processes m_1, m_2 and m_3, instantiated from functional module M. There is a direct correspondence between order in which ports appear in **input/output** declarations and type of *main* function of functional modules. For instance, the type of *main* function of M is given by: $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow IO(t_4, t_5)$. In this case, the types t_1, t_2 e t_3 are mapped, respectively, onto input ports a, (b, c), and d of a functional processes instantiated from M. Ports b and c are associated with the same parameter (t_2), indicating that both ports will wait for the second argument non-deterministically.

B. Communication Channels

Similarly to Occam[Inmos, 1988], $Haskell_{\#}$ 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,

connect $p_0.a$ **to** $p_1.b$

defines a channel connecting output port a of process p_0 to input port b of the process p_1. The channel type is the type of the connected ports.

$Haskell_{\#}$ 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 analyze statically 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.

C. Initialization, Execution, and Termination

When running a $Haskell_{\#}$ application, we define its state at time t as a set which contains the states of each functional process at that time. A functional process can be in one of four states[Carvalho Jr., 2000]:

- **WI**: waiting for input in port i;
- **RN**: running;
- **WO**: waiting for output in port j;
- **FN**: finished;

In Figure 2, it is presented a state diagram which shows the possible transitions between states of functional processes. They are detailed in Table C.

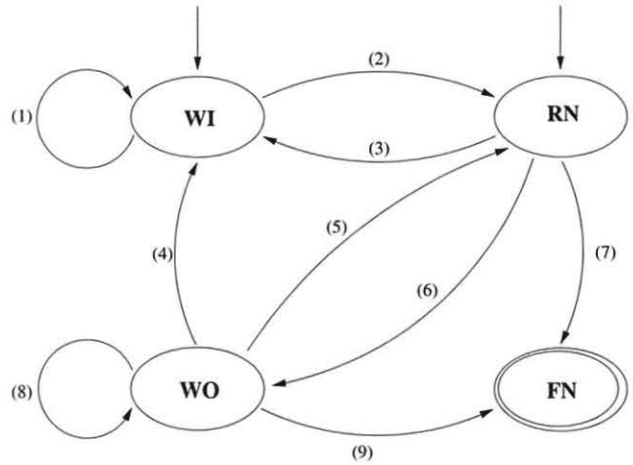

Fig. 2. State Diagram for Functional Processes

T	CS	NS	nIP	nOP	T?	Action
1	WI	WI	≥ 2	-	-	to read next input port
2	WI	RN	≥ 1	-	-	to evaluate *main*
3	RN	WI	-	$= 0$	yes	to read first input port
4	WO	WI	≥ 1	≥ 1	yes	to read first input port
5	WO	RN	$= 0$	-	yes	to evaluate *main*
6	RN	WO	-	≥ 1	-	to write first output port
7	RN	FN	-	$= 0$	no	to finish
8	WO	WO	-	≥ 2	-	to write next output port
9	WO	FN	-	≥ 1	no	to finish

T = Transition
CS = Current State
NS = Next State
nIP = Number of input ports
nOP = Number of output ports
T? = Is it repetitive ?

TABLE I
INTERPRETATION OF THE STATE DIAGRAM IN FIGURE 2

In the diagram, you can observe that a process can be in **RN** or **WI** states at the start. In the former case, called *explicit initialization*, processes are declared using the **start** declaration and start executing independently on arguments demanding from their input ports. In the latter case (processes not declared in some **start** declaration), processes only computes on demanding of arguments from their input ports. If a process declared in a **start** declaration has arguments, they can be explicitly passed using the same syntax of Haskell. For instance, consider the following declarations:

.
start $fp1$
start $fp2$ [1,2,3,4] (square 20) 2.0

In the example, $fp1$ and $fp2$ are processes that must be initialized in the **RN** state. The main function of process $fp1$ has no argument ($main :: IO()$), whereas the main function of $fp2$ expect three parameters ($main :: [Int] \rightarrow Int \rightarrow Double \rightarrow IO(t_1, t_2, \cdots, t_n)$).

Haskell code can be written inside HCL code using the character #. For instance the function $square$ must be declared in the following way inside HCL Code:

```
#
    square :: Int → Int
    square x = x * x
#
```

The only exception is when declaring port types and arguments for functional processes in a **start** declaration. Thus, the following declarations are invalid, because it is not necessary to put #'s delimiting $Haskell$ code:

```
⋮   input teste #::[[Int]]#
⋮   start fp1 #[1,2,3,4] (square)#
```

$Haskell_{\#}$ applications terminate when all processes have reached finished state (**FN**). Yet, due to their synchronous semantics, processes terminate when all their output values are consumed by other processes in the network. There are two types of processes: *nonrepetitive* and *repetitive*. The former starts, executes some activities and terminates, while the later starts, executes some activities and returns to the initial state, never reaching the termination state. *Repetitive* processes are useful for development of *reactive systems* [Shapiro, 1989], applications that does terminate, such as monitoring and controlling ones. Operating systems are classical examples.

Because termination is a global property, inherited by all processes in application, applications can contain either *repetitive* or *nonrepetitive processes*, but never both simultaneously. Therefore, termination behavior is declared in the header of the HCL program. Observe the code fragments which define two different termination properties for the application $AppExample$:

application $AppExample$ **repetitive**
application $AppExample$ **nonrepetitive**

The former declares it to be *repetitive*, while the later declares it to be *nonrepetitive*.

D. Mapping of Functional Processes

The execution environment of $Haskell_{\#}$ applications is a network of processing *nodes*, onto which functional processes are statically mapped. Programmers should allocate processes in groups. Processes belonging to a given group will execute concurrently onto the same node

Before task allocation, environment nodes should be classified based on relevant *features*, such as *node processor speed* (*fast* or *slow*), *amount of memory* (*large* or *thin*) or *communication speed* (*high* or *low*), in the 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 HCL Compiler to determine 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 $(wide, fast)\ p_0, p_1, p_2$
alloc $(slow)\ p_3$

Processes p_0, p_1, and p_2 are allocated to a processor with features $fast$ and $wide$ from classes *speed* and *memory*, respectively, while process p_3 is mapped onto a node 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.

D.1 Performance Tuning

Performance tuning optimises a particular $Haskell_{\#}$ application on a parallel environment. Load balancing is performed statically by increasing *data locality*, allocating processes connected by channels that communicate a large amount of data on the same processor, whenever possible. The use of node classification mechanism allows modelling of the parallel environment features and processes' needs. We identify two strategies for *node classification*, affecting *performance tuning*:

1. **Parallel Execution Environment Oriented.** Node classification is performed based on the characteristics of the parallel environment, increasing *efficiency* but sacrificing portability, because *performance tuning* should be performed again whenever an application is ported;
2. **Application Oriented.** Node classification is performed based on features which have a significant impact on the application performance, independently of the execution environment. Thus, when an application is ported to a new environment, it is only necessary to classify its nodes based on these features. This approach favors *portability* with little impact on efficiency, because only significant features are considered for performance tuning purposes.

We consider the second strategy more suitable than the first one in most of cases, because it provides portability without significant loss of efficiency. Whenever performance requirements are crucial and there is no possibility to port the application to a new environment or to change environment features, first strategy can be adequate. In some cases, adoption of a hybrid strategy is useful. For example, when the programmer ports an application and verifies that its performance on the new environment can be improved by the introduction of some intrinsic features not predicted in the original node classification.

III. $Haskell_{\#}$ Compared to Others Parallel Functional Languages Based on Coordination

In the context of parallel functional languages, $Haskell_{\#}$ belongs to the class of coordination based ones. Other important examples of this class are $Eden$[Breitinger *et al.*, 1997] and $Caliban$[Kelly, 1989, Taylor, 1997]. Like $Haskell_{\#}$, Both uses $Haskell$ for specifying computation and assume a static network where functional processes communicate via point-to-point and unidirectional channels. $Caliban$ also provides annotations for specifying placement of processes, while $Eden$ has facilities for specification of *reactive systems*[Shapiro, 1989]. Following, we discuss the most important points that distinguish $Haskell_{\#}$ from other known parallel functional languages based on coordination:

- *Adoption of a configuration based*[2] *[Krammer, 1994] coordination language (HCL)*, separating completely construction of computation code (pure $Haskell$) from coordination one (HCL). In $Eden$ and $Caliban$, computation and coordination code co-exist in the same program text. The existent separation is essentially semantic: management of parallelism are defined by some special annotations. Caliban try to make a syntactic separation by putting every annotation after a special clause, called **moreover**. This $Haskell_{\#}$ feature facilitates parallelization of sequential pre-existent $Haskell$ programs and induces independent specification and development of functional modules and coordination code, reducing development costs and favoring reuse of parts. This ability for composing programs from parts also turns $Haskell_{\#}$ more suitable for large scale parallel applications than $Eden$ and $Caliban$, following the

[2]The configuration paradigm was developed in the context of specification of distributed systems, offering good support for parallel and distributed software engineering[Krammer, 1994].

ideas discussed in [DeRemer & Kron, 1976]. Another consequence is to stimulate *coarse grain parallelism*, considered more efficient on distributed memory architectures, such as *clusters*.

- *Modelling of parallel architectures.* It is known that generic optimal *placement* of processes over processors is an NP-complete problem. Thus, automatic mechanisms for this purpose, dynamic or static, are not efficient in all instances. We decided to follow a *static* and *explicit* approach in $Haskell_{\#}$, as in 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 a best mapping between functional processes and processors using these information. Existing automatic tools are wellcome, but decisions should be left to programmer, as discussed in Section II).
- *Formal property analysis using Petri nets.* When developing $Haskell_{\#}$, one of our main concerns has been the possibility analysis of formal properties of applications. A compiler that translates HCL into INA, a Petri net analysis tool, was developed by Lima[Lima, 2000]
- *Easy and efficient implementation.* Unlike Eden and Caliban, $Haskell_{\#}$ does not need any run-time system support or dynamic mechanism. It can be easily implemented by gluing a message passing library to a sequential $Haskell$ functional compiler[3]. Assuming that $Haskell_{\#}$ applications are coarse grain, we can take advantage of the best technology for compilation of sequential functional programs. Also, absence of dynamic mechanisms and explicit nature of communication topology specification tends to reduces unnecessary communication.

Despite its argued advantages, $Haskell_{\#}$ computational model in its initial form is very restrictive in relation to $Eden$ and $Caliban$ for expression of some parallel computational patterns. The reason is the inability of $Haskell_{\#}$ for specifying interaction (communication) of functional processes during their computation[4]. This restriction is due to our initial concern of making $Haskell_{\#}$ communication synchronous and to facilitate translation of $Haskell_{\#}$ applications into Petri nets. $Caliban$ and $Eden$ use the concept of *streams* to support interaction between functional processes. One of our goal in this paper is to show how this feature can be incorporated to $Haskell_{\#}$. Support for explicit data parallelism and facilities for specification of large scale applications are also presented. MCP-$Haskell$ will be used to exemplify and to motivate our proposals.

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

[4]Synchronization of functional processes are performed only before or after functional process execution. Thus, communication and computation cannot be interwinded.

IV. Monte Carlo Particle Transport Simulation

MCNP(Monte Carlo N-Particle) is a Fortran code developed over many years at Los Alamos to solve the particle transport simulation with Monte Carlo methods [Whalen *et al.*, 1991]. It involves simulating statistical behavior of particles (photons, neutrons, electrons, etc.) while they travel through objects of specified shapes and materials. In [Hammes *et al.*, 1995], two simplified functional implementations of MCNP are described, using Haskell[Hudak *et al.*, 1992](MCP-Haskell) and Id[Hicks *et al.*, 1993](MCP-Id) respectively. In this paper, we have developed a parallel version of MCP-$Haskell$ and call it MCP-$Haskell_{\#}$

Besides its importance, the intrinsic parallel nature of MCNP and the availability of a simplified sequential Haskell implementation (MCP-$Haskell$) were our main motivations for choosing it in evaluating $Haskell_{\#}$. We have reused MCP-$Haskell$ code for reducing development costs and for allowing us to concentrate in parallelization process.

V. Parallelizing MCP-$Haskell$ with $Haskell_{\#}$

To describe the parallelization of MCP-Haskell[5], we will follow the modular methodology described in [Carvalho Jr., 2000], which takes advantage of the modular nature of $Haskell_{\#}$ applications. The following sections describe each stage involved in the parallelization process.

A. Identification of Functional Modules (Functional Decomposition)

We have identified in MCP-$Haskell_{\#}$ four functional modules::

- *Problem Definition.* It reads the input files, where problem parameters are specified, and sends them as output;
- *Tracks.* Given the problem parameters and a list of photons, it tracks a photon through a series of movements and collisions with nuclei. Its output is a list of *events* for each photon and a *tally* of all photon creates and losses, called *totals*;
- *Tallies.* It takes lists of events for each photon, extracts each *tally*, and reduces tally information to average and standard deviations so that error estimates can be given;
- *Statistics.* It receives as input the *average energy* of photons, a list of tally entries, the *totals*, and the *tallies* generated by photons tracks. Statistics operations are processed and the result of the simulation is presented.

[5]In this paper we do not present details about implementation of MCP-$Haskell$. For better understanding, the reader can consult [Hammes *et al.*, 1995].

The functional modules described above reuse, without modification, the original sequential code of MCP-$Haskell$. This evidences the high degree of potential reuse provided by $Haskell_{\#}$ programming platform.

B. Integrating Functional Modules

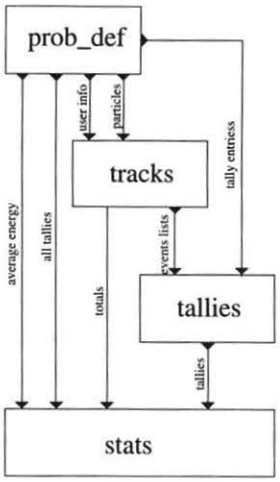

Fig. 3. MCP-$Haskell_{\#}$ Preliminary Network

In Figure 3, it is shown the *preliminary network* which defines semantics of MCP-$Haskell_{\#}$, by integrating functional modules identified in the *functional decomposition*. In that, no pattern of parallel computation can be exploited, due to the synchronous semantics of channels, which demands that *track* process sends the list, for each photon, of list of events to the tally process only when complete list is built. Thus, using the preliminary network as a *final network*, processes never run in parallel. The *final network* of processes (Figure 4) is defined by introducing *data parallelism*, using the fact that each particle can be tracked and tallied independently.

We divided composition of *track* and *tally* functional processes in n. Each resulting composition must process one of n groups of photons. *Statistics* process is divided into m processes which calculate, each one, statistical information about m different tallies. In Figure 4, $n = 4$ and $m = 2$.

C. Implementation of Functional Modules

Functional modules reuse original sequential code of MCP-$Haskell$ without any need for modification.

D. Mapping of Functional Processes and Performance Tuning

In the final network (Figure 4), processes form four groups:

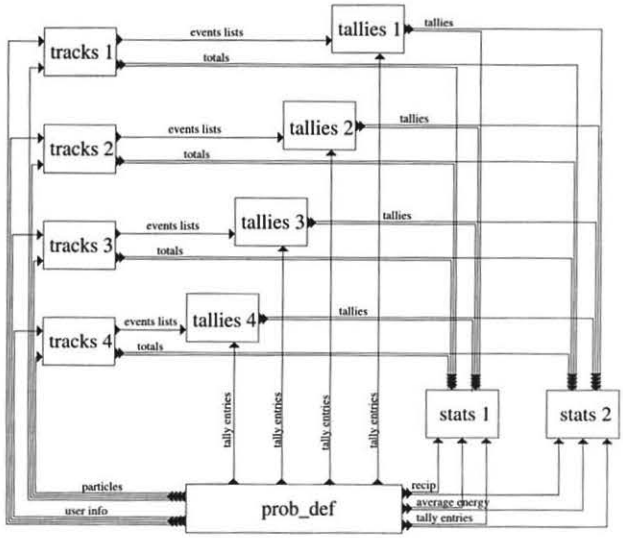

Fig. 4. MCP-$Haskell_{\#}$ Final Network

1. $prob\_def$, $tracks$ 1, $tallies$ 1;
2. $tracks$ 2, $tallies$ 2;
3. $tracks$ 3, $tallies$ 3, $stats$ 1;
4. $tracks$ 4, $tallies$ 4, $stats$ 2;

One can deduce that functional processes inside a group never run in parallel (data dependency). Thus, the groups can be allocated onto separate nodes, executing in parallel and having the same workload. For best *load balancing*, a homogeneous environment is suitable for running this application. The grouping of processes presented here can be extended to other configurations, modifying parameters m and n.

E. HCL code

In the original HCL code for the version of MCP-$Haskell_{\#}$ presented in figure 4, 76 ports from 11 functional processes are declared individually. Besides this, 38 **connect** declarations are necessary to create channels for connecting them. The code cannot be parameterized, which means that if a programmer wants to increase m and n parameters, a new code must be written. These limitations, a consequence of the simplicity of HCL, turns difficult to express large scale parallel applications. In Section VI, we will introduce to HCL some syntax sugaring to overcome these difficult. The new HCL code of MCP-$Haskell_{\#}$ incorporating these changes is shown in figure 5.

```
1.  application MCP<m,n,bl> nonrepetitive
2.  module MCPProbDef
3.      output <avg_e[1..m]> #broadcast# ::Double
4.      output <recip[1..m]> #broadcast# ::Double
5.      output <user_info[1..n]> #broadcast# ::User_spec_info
6.      output <all_tallies_[1..m]> #shuffle# ::[Tally_entry]
7.      output <all_tallies[1..n]> #shuffle# ::[Tally_entry]
8.      output <particles[1..n]> #shuffle# ::[(Particle,Seed)]
9.      instances probdef
10. module Tracks
11.     input user_info ::User_spec_info
12.     input particles ::[(Particle,Seed)]
13.     output events ::[[Event]]
14.     output <totals[1..m]> #shuffle# ::[Int]
15.     instances tracks[1..n]
16. module Tallies
17.     input events ::[[Event]]
18.     input all_tallies ::[Tally_entry]
19.     output <tallies[1..m]> #(\m → \a →
          ((shuffle m).transpose) a)# ::[[Array (Int,Int) Double]]
20.     instances tallies[1..n]
21. module Show_statistics
22.     input avg_e ::Double
23.     input recip ::Double
24.     input all_tallies ::[Tally_entry]
25.     input >totals[1..n]< #(map.sum).transpose# ::[Int]
26.     input >tallies[1..n]< #concat# ::[[Array (Int,Int) Double]]
27.     instances stats[1..m]
28. connect probdef.particles[i] to tracks[i].particles
                                        stream bl for i = 1..n
29. connect probdef.user_info[i] to tracks[i].user_info for i = 1..n
30. connect probdef.all_tallies[i] to tallies[i].all_tallies for i = 1..n
31. connect tracks[i].events to tallies[i].events stream bl for i = 1..n
32. connect tallies[i].tallies[j] to stats[j].tallies[i] for i = 1..n, j = 1..m
33. connect tracks[i].totals[j] to stats[j].totals[i] for i = 1..n, j = 1..m
34. connect probdef.all_tallies_[j] to stats[j].all_tallies for j = 1..m
35. connect probdef.recip[j] to stats[j].recip for j = 1..m
36. connect probdef.avg_e[j] to stats[j].avg_e for j = 1..m
37. start probdef
38. alloc probdef
39. alloc tracks[i] for i = 1..n
40. alloc tallies[i] for i = 1..n
41. alloc stats[i] for i=1..n
```

Fig. 5. HCL Code for MCP-$Haskell_{\#}$

VI. Proposals for Improving $Haskell_{\#}$

$Haskell_{\#}$ is still under development. Its original proposal [Carvalho Jr., 2000, Lima, 2000] focused on efficiency and formal properties analysis using Petri nets [Lima, 2000]. Therefore, a *static*, *explicit* and *synchronous* parallel computational model was adopted, inspired in OCCAM. We argue that this approach makes difficult to exploit some parallel computation patterns, like *pipeline* and *systolic computation* [Manber, 1999]. In general, it is hard to exploit patterns which involve interaction between processes during computation, because functional processes only communicate before and/or after computation. We have made $Haskell_{\#}$ process semantics too restrictive to guarantee modularity of *process hierarchy*, necessary to allow the translation of applications into Petri nets. Also, due to the simplicity of HCL, it is difficult to express in a concise and elegant way large scale parallel programs, which involves many processes and interconnections.

In what follows, we present some proposals to make $Haskell_{\#}$ more expressive, simpler, cleaner, and more efficient, as required for turning it more suitable for *general purpose parallel computing*. Modifications proposed do not affect neither *process hierarchy* nor translation into Petri nets.

A. Modifications to HCL syntax and structure

HCL is a simple coordination language, making some programs, notably large scale ones which involve a large number of processes and/or interconnections, difficult to express in an elegant and concise way. Proposals shown in this section overcome this difficult by introducing some syntactic sugaring to HCL.

A.1 Data Parallel Constructors

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.

$Haskell_{\#}$ essentially exploits functional parallelism. *Data parallelism* is simulated by instantiating a number of functional processes (*data parallel tasks*) from a functional module. Another one (*distributor*) is used to distribute data over them. In MCP-$Haskell_{\#}$, observe that the *problem definition* process (*prob_def*), when distributing particles over *track* processes (*data parallel tasks*), is a *distributor*.

The functional module from which the *distributor* process was instantiated must have a number of output ports corresponding to the number of *data parallel tasks*. Whenever the programmer wants to increase them, functional module code of the *distributor* process has to be modified. This is undesirable because it destroys the independence of paral-

lel and sequential parts of applications (*process hierarchy*). Our proposal is to add explicit *data parallel constructors* to HCL syntax, allowing total abstraction of data parallelism concerns from sequential programming. The constructors are:

- **Distribute constructor:** associates a number of output ports corresponding to the parallel tasks to a component of the tuple given as result of the `main` function of the functional module. A function is specified to distribute component data over the output ports.
- **Group constructor:** associates a number of input ports corresponding to the parallel tasks to an argument of the `main` function of the functional module. A function is specified to group data received from the input ports, allowing it to be passed to the argument;

The syntax is similar to that for non-deterministic ports and it is the same for *group* and *distribute* constructors. In figure 5 (HCL code for MCP-$Haskell_{\#}$), examples of their use can be seen. *Distribute* constructors are applied in the context of output port declarations (lines 3, 4, 5, 6, 7, 8, 14, 19), while *group* constructors are applied in the context of input ports (lines 25, 26). For instance, in line 19, *Tallies* module declares m ports ($tallies[1]$ to $tallies[m]$) associated with a component of the IO tuple returned by its `main` function. The value of this component is divided into m groups using the function ($\backslash m \rightarrow \backslash a \rightarrow ((shuffle\ m).transpose)\ a$) and sent through the m ports. *Statistic* module, in line 26, declares n ports ($tallies[1]$ to $tallies[n]$) associated with one of its `main` arguments. Data received from ports is grouped using $concat$ function and passed as argument.

For simulating a *broadcast*, it is only necessary to use an appropriate distribution function to repeat output value of the process over output ports, as shown in lines 3, 4, 5.

A.2 Indexing and Parameterizing

In his work, Turner[Turner, 1982] presents some considerations about explicit parallel programming environments:

"*The potential performance of this kind of architecture is enormous, but how can they be programmed ? An idea that can be dismissed more or less straight away is that we should take some conventional language and add facilities for explicitly creating and co-ordinating processes. This may work where the number of processes is small, but when we are talking about thousands and thousands of independent processes, this cannot possibly be under the conscious control of the programmer*"

Gelernter and Carriero[Gelernter & Carriero, 1992] disagree of Turner, attributing his fallacy to his supposition that each process in a coordination language must be created and dealt with individually. In practice, in large scale parallel programs, most processes are identical and have the function of computing a piece of a large data structure. Turner did not take these aspects into consideration. There are forms of managing several processes at the same time without referencing them individually (quoting Gelernter:"*to specify explicitly does not mean to specify individually*").

Turner's argument holds when talking about the original $Haskell_{\#}$ proposal, because it requires each process, port and channel to be referenced individually. Our proposal here is to enrich HCL syntax to allow indexed referencing of a parameterized number of processes, ports and channels. Observe the declaration of *Track* functional module of MCP-$Haskell_{\#}$ in Figure 5, where indexes are used to declare m *totals* ports and n *tracks* functional processes, allowing them to be reffered as $totals[1]$ to $totals[m]$ ports and as $tracks[1]$ to $tracks[n]$ functional processes. The general format to declare integer indexed identifiers is $id\_name[i..j]$, where id_name is the identifier name, i is the lower index, and j is the upper index. Another format, used for named ones, is $id\_name[i_1, i_2, \cdots, i_n]$, where i_1,..., i_n are index names.

To reference concisely a large number of processes, ports and channels, we propose the introduction of the **for** constructor, as exemplified in the **connect** section of MCP-$Haskell_{\#}$ (Figure 5, lines 28 to 36). It can be used in any HCL declaration, except in a *header* one, declared only once.

Observe that HCL code of MCP-$Haskell_{\#}$uses three variables (m, n, bl) to specify, respectively, the number of compositions of *tracks* and *tallies* processes, the number of *statistics* processes, and the length of buffer used in *stream* communication (explained in the next section). The possibility of parameterizing applications is another feature proposed in this paper to be introduced in $Haskell_{\#}$. Using it, the programmer provides to HCL compiler the values of parameters m, n and bl, which must be declared in the header of the application, as shown in line 1 of Figure 5.

B. Stream Channels: Interacting Functional Processes without breaking Process Hierarchy

In Section V(B), no parallel computational pattern could be exploited in *preliminary network* of MCP-$Haskell_{\#}$, due to data dependency amongst functional processes. This limitation is intrinsic to $Haskell_{\#}$, because we know that it is possible to exploit *pipeline* pattern in MCP-Haskell if interaction of processes during computation was supported by $Haskell_{\#}$, but *process hierarchy property* does not allow it.

A naive solution to overcome these problems is to allow processes to exchange messages during computation, by providing explicit *send* and *receive* primitives to be used in functional modules. This approach is undesirable, because it destroys the process hierarchy property. We propose a mechanism to allow processes to interact during computation taking

advantage of *lazy evaluation* in $Haskell$, which allows a list to be constructed on demand. This property turns *lazy functional languages* capable to work with *streams* (*infinite lists*). This approach does not destroy process hierarchy property.

In MCP-$Haskell_{\#}$, a *pipeline* can be exploited in the preliminary network, because the list of events is computed independently for each particle. For example, the channel that communicates *tracks* and *tallies* functional processes for transmitting the list of list of events for each particle can be declared as a *stream* channel, allowing each list of events of the stream to be sent whenever it is available, after its corresponding particle was tracked. The idea is that *tracks* process systematically produces lists of events for each particle and the *tallies* process consumes them, simultaneously.

Communication via streams requires to turn send operation asynchronous. It can be proved that this does not affect $Haskell_{\#}$ application semantics. Actually, we can simulate asynchronous send by using *buffer* processes. Asynchronous send is also a way to increase parallelism of applications, by reducing data dependency between processes.

The interleaving of *computation* and *communication* with the adoption of lazy streams forces the programmer to take more care about control of granularity of parallelism, avoiding excessive *communication overheads*. Control of granularity is one of the most important original concerns of $Haskell_{\#}$. In MCP-$Haskell_{\#}$, for example, the communication overhead can increase drastically if we consider transmission of each individual element of the stream whenever it is available. We intend to provide the programmer the possibility of configuring the number of elements of the stream that can be transmitted at the same time, trying to reduce number of communication operations and to increase the amount of data communicated in a single operation. With that, the unity of stream communication becomes a block of elements, instead of a single element.

The only visible modification to the HCL program needed to allow stream communication is to declare a channel to be a *stream*, as shown in lines 28 and 32 of Figure 5. The channel declaration in line 32 creates a *stream channel* connecting *events* ports of *tracks* and *tallies* functional processes. Events are transmitted in blocks of length *bl* (a parameter of the HCL program). If a number was not declared after **stream** in connect *declaration*, the compiler assumes that stream elements should be sent one at a time.

B.1 Effects of the Adoption of Stream Channels on the MCP-$Haskell_{\#}$ Allocation of Processors

In Section IV(B), mapping of processes of MCP-$Haskell_{\#}$ over a network of processors was presented. For a network with eleven processes (Figure 4), *four* processors were allocated.

When introducing *stream channels*, a *pipeline* is introduced amongst processes in the network, allowing that every process run in parallel. The eleven processes now can be allocated individually to processors. This evidences an increase of the degree of parallelism in MCP-$Haskell_{\#}$ with the adoption of *stream channels*.

VII. Performance Results

We have implemented synchronous versions of MCP-$Haskell_{\#}$ for running optimally over *one* (preliminary network as final network), *two*, *three*, *four* and *eight* processors. For each one, we have also implemented an associated *stream* version for making performance comparison. The *stream* versions use respectively *four*, *seven*, *eight*, *eleven* and *nineteen* processors for optimal execution. The parallel environment used is a cluster of 8 Linux PC's (6 Pentium II MMX 350MHz and 2 Pentium III 550MHz), configured as described in [Spector, 2000], connected through a 100Mbs fast ethernet network. Because of our limitation to 8 processors, it was only possible to compare the *four*, *seven* and *eight* processors version with, respectively, the synchronous versions for *one*, *two* and *three* processors.

Our implementation of $Haskell_{\#}$ over cluster of PC's uses GHC (*Glasgow Haskell Compiler*) [GHC Team, 1998], for compilation of functional modules, and MPI(*Message Passing Library*)[Dongarra *et al.*, n.d.], for management of parallelism.

In Figures 6, 7, 8, graphs for respectively *running time*, *speedup*, and *efficiency*[6] for synchronous $Haskell_{\#}$ version of MCP-$Haskell$ (solid lines) are compared with ideal values obtained from a perfect parallelization of this application (dashed lines). For example, in Figure 7, the synchronous version with eight processors has a speedup of 3.9, while a hypothetic perfect parallel implementation could obtain a speedup of 8.0. The graphs show satisfactory gains in performance of parallel $Haskell_{\#}$ synchronous version of MCP-$Haskell$ if compared to the performance of sequential one.

Table II compares running time values of *stream* version to *synchronous* one. The gains in performance with adoption of streams has been very poor for MCP-$Haskell_{\#}$, despite the increase of parallelism obtained. We have investigated this result and we have concluded that almost all running time of the application is spent with *track*'s functional processes, an initially unexpected characteristic that degenerates our pipeline parallelism. However, adoption of stream communication has not been invalidated, because we have reached our most important goal: to allow that processes interact without losing *process hierarchy*, increasing potential parallelism of the application. If computational costs of *tallies* functional processes were of the same order of magnitude of that in *tracks* ones, certainly we had obtained a better

[6]percentage of processor's time that is not wasted, if compared to the sequential algorithm [Manber, 1999].

gain in performance.

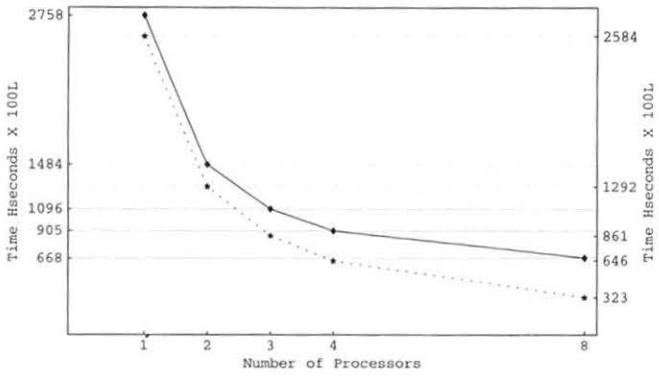

Fig. 6. Running time

A. *Disappointment with unboxed arrays in GHC 4.08.1*

We have tried to increase performance of the MCP-$Haskell_{\#}$ application by using GHC unboxed array (UArray), avoiding element by element copying of Array elements to the MPI buffer. Haskell unboxed arrays are stored in a contiguous buffer, which can be copied directly, using primitive operations, into a MPI buffer. But we have verified that the performance of the sequential version of MCP-$Haskell$ using *unboxed arrays* executes nearly two times slower than the simple array version. This result was a surprise because unboxed arrays were created for improving performance of arrays of primitive types. We have abandoned use of unboxed arrays until a new version of GHC fix this problem.

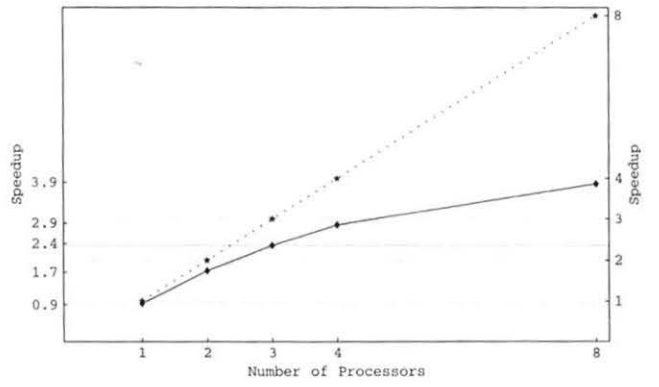

Fig. 7. Speedup

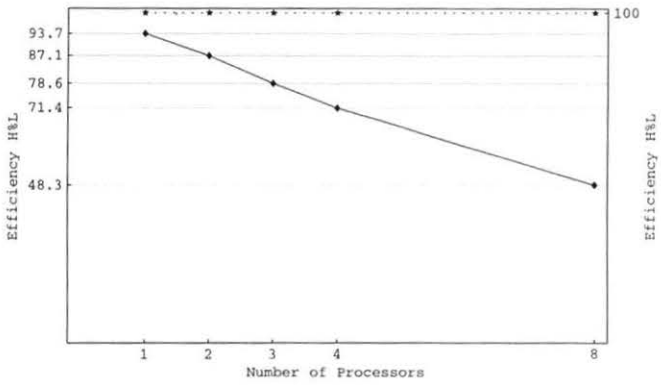

Fig. 8. Efficiency

TABLE II

SYNCHRONOUS X STREAM

P*	Synchronous	P	Stream
1	2757.5	4	2728.5
2	1484.1	7	1475.5
3	1095.6	9	1039.1

* - Number of processors needed for the best execution.

VIII. CONCLUSIONS AND LINES FOR FURTHER WORKS

This paper describes the parallelization of MCP-$Haskell$[Hammes *et al.*, 1995], a simplified functional version of Monte Carlo N-Particle transport application (MCNP[Whalen *et al.*, 1991]), for evaluating $Haskell_{\#}$ programming environment. This work gave us some important experimental results in exposing $Haskell_{\#}$ *virtues* and *limitations*. Changes to $Haskell_{\#}$ to overcome these limitations were proposed. It is not intended to alter $Haskell_{\#}$ original properties, but to improve its expressiveness for representing parallel computational patterns and to allow it to support concise and elegant implementation of large scale parallel applications. *Simplicity*, *efficiency*, *high-degree of modularity* (sequential MCP-$Haskell$ was reused without modifications), and potential for *formal analysis* are the most important virtues of $Haskell_{\#}$ shown in this paper. Performance benchmarking for MCP-$Haskell_{\#}$ is also presented.

We continue building applications to evaluate $Haskell_{\#}$ and to propose new improvements to it. A new $Haskell_{\#}$ specification should be produced in the near future, incorporating the ideas presented in this paper and others to come.

REFERENCES

[Breitinger *et al.*, 1997] Breitinger, S., Loogen, R., Ortega Mallén, Y., & Peña, R. (1997). The Eden Coordination Model for Distributed Mem-

ory Systems. *High-Level Parallel Programming Models and Supportive Environments (HIPS).*

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

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

[Carvalho Jr., 2000] Carvalho Jr., F. H. 2000 (Jan.). *Haskell$_{\#}$: Uma Extensão Paralela para Haskell.* M.Phil. thesis, Centro de Informática, Universidade Federal de Pernambuco.

[DeRemer & Kron, 1976] DeRemer, F., & Kron, H. H. (1976). Programming-in-the-Large versus Programming-in-the-small. *Ieee transactions on software engineering*, June, 80–86.

[Dongarra *et al.*, n.d.] Dongarra, J., Otto, S. W., Snir, M., & Walker, D. *Technical Report CS-95-274, University of Tennessee*, Jan.

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

[GHC Team, 1998] GHC Team. (1998). The Glasgow Haskell Compiler User's Guide, Version 4.01. *http://www.dcs.gla.ac.uk/fp/software/ghc/4.01/users_guide/users_guide.html.*

[Hammes *et al.*, 1995] Hammes, J., Lubeck, O., & Böhm, W. (1995). Comparing Id and Haskell in a Monte Carlo Photon Transport Code. *Journal of functional programming*, July, 283–316.

[Hammond & Michaelson, 1999] Hammond, K., & Michaelson, G. (1999). Research Directions in Parallel Functional Programming. *Springer-Verlag.*

[Hicks *et al.*, 1993] Hicks, J., Chiou, D., Ang, B. S., & Arvind. (1993). Performance Studies of Id on the Monsoon Dataflow System. *Journal of parallel and distributing computing*, **18**, 273–300.

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

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

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

[Hudak *et al.*, 1992] Hudak, P., Jones, S. P. L., & Wadler, P. L. (1992). Report on Programming Language Haskell: a Non-Strict, Purely Functional Languages. *Special Issue of SIGPLAN Notices*, **16**(5).

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

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

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

[Krammer, 1994] Krammer, J. (1994). Distributed Software Engineering. IEEE Computer Society Press (ed), *Proc. 16th IEEE International Conference on Software.*

[Lima, 2000] Lima, R. M. F. 2000 (July). *Haskell$_{\#}$ - uma linguagem funcional paralela - ambiente de progrmação, implementação e otimização.* Ph.D. thesis, Centro de Informáica, UFPE.

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

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

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

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

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

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

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

[Shapiro, 1989] Shapiro, E. (1989). The family of concurrent logic programming languages. *Acm computing surveys*, **21**, 413–510.

[Spector, 2000] Spector, H. M. (2000). *Building Linux cluster.* O'Reilly.

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

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

[Turner, 1982] Turner, D. A. (1982). Recursion Equations as a Programming Language. *Functional programming and its applications*, 1–28.

[Whalen *et al.*, 1991] Whalen, D. J., Hollowell, D. E., & Hendriks, J. S. (1991). *MCNP: Photon Benchmark Problems.* Tech. rept. LA-12196. Los Alamos National Laboratory.