On the Implementation of SPMD Applications using Haskell

Francisco Heron de Carvalho Junior\(^1\), Rafael Dueire Lins\(^2\)\(^\dagger\), Nívia Cruz Quental\(^1\)

\(^1\) Escola Politécnica de Pernambuco, Universidade de Pernambuco
Rua Benfica 455, Recife, Brazil
\{fhcj@upe.poli.br, nivia@upe.poli.br\}

\(^2\) Departamento de Eletrônica e Sistemas, Universidade Federal de Pernambuco
Rua Acadêmico Hélio Ramos, s/n, Recife, Brazil
\{rdl@ee.ufpe.br\}

Abstract—Commodities-built clusters, a low cost alternative for distributed parallel processing, brought high-performance computing to a wide range of users. Most of them are interested in parallelising scientific applications. Message passing parallel programming using lower level mechanisms, such as MPI and PVM libraries, has become popular, because these tools are free, easy to obtain, and efficient. However, for large-scale applications, parallel software engineering using low level tools is a very hard task, due to its poor abstraction and modularity. This makes difficult the widespread use of cluster computing, by mathematicians, biologists, engineers, physicists, \textit{et al.} Haskell\(^\#\) is an attempt to bring together higher-level parallel programming and cluster-based architectures, without sacrificing speedup and scalability. This paper compares speedup, scalability, and modularity of Haskell\(^\#\) and MPI implementations of some representative NAS benchmarks (EP, IS and CG).

Keywords—Parallel programming, clusters, benchmarking.

I. INTRODUCTION

In the years before the emerging of cluster computing [1], super-computing was restricted to a relatively small number of highly specialized users, in general at rich academic or government institutions or industries in developed countries. Efficient parallel processing was achieved by using specialized and architecture-dependent parallel languages or libraries. In general, they sacrifice portability in favor of performance, resulting in great efforts to the development of complex parallel programs.

The advent of commodity-based clusters architectures widened the range of users of high-performance computing and parallel programming. To achieve a good speedup and scalability in clusters, which have high communication costs, low-level explicit parallel programming libraries, such as MPI[2] and PVM[3] are employed. These tools became popular also due to the fact that they can be easily obtained free of charge. Parallelizing and tuning programs using these tools are not easy tasks for most of programmers. Today, biologists, mathematicians, physicists, amongst many other researchers, are interested in scientific applications that demand peak performance. The lack of parallel programming specialists and the inherent difficulty in the development of scientific applications, due to the required mathematical background for programmers, slows down the widespread use of cluster computing.

Parallel programs are growing fast in size and complexity due to the new application areas that emerged with low cost supercomputers. Thus, there is a great demand for higher-level programming tools. However, programming language designers know how difficult is to reconcile good performance and a higher level of abstraction in a parallel programming environment. In general, programming abstraction is sacrificed in favour of performance [4, 5]. One could say that it is easy to build high-performance computers using a cluster, but it is difficult to find programmers that can use them efficiently. It is now necessary to make parallel programming more accessible without significant losses to performance.

Haskell\(^\#\) is a parallel distributed extension to Haskell[6], a language considered a \textit{de facto} standard in functional programming. Haskell\(^\#\) intends to provide a high-level abstraction mechanism without sacrificing performance, by minimizing the overheads of parallelism management in clusters. Haskell\(^\#\) allows for property analysis of parallel programs by using Petri nets[7], a well reputed formalism for the specification of concurrent systems, with several analysis tools[8, 9] available. Haskell\(^\#\) makes the coordination [10] and functional [11] paradigms to meet at different levels [12, 13]. Parallelism control is described using Haskell\(^\#\) Configuration Language(HCL) and computations are programmed in plain Haskell, without any syntactic extension or additional library. Haskell\(^\#\) supports compositional and skeleton programming[14]. MPI [2] is used to manage parallelism without claiming for any run-time support.

The main motivations to use Haskell to describe sequential components are:

- To make possible a clean and total separation between coordination (parallelism) and and computation media of programming, by taking advantage of lazy evalu-
tion provided by non-strict semantics of Haskell for this purpose;

- To improve potential for analyzes of program properties at the computational level, taking advantage of formal analysis techniques adapted for pure non-strict functional programs. At coordination level, however, Petri nets have been used to prove properties about topology and communication behavior of the process network.

This paper presents a benchmark of SPMD scientific programs developed with the most recent version of Haskell\#. The set of benchmarks were taken from NAS Parallel Benchmarks (NPB)\[16\], a collection of eight programs intended to evaluate performance of parallel architectures when executing applications in the area of dynamic fluid simulation. Three of the five kernel benchmarks of NPB were implemented in Haskell\#, and compared to their Fortran/MPI versions\[18\]. The chosen benchmarks are: EP (Embarassingly Parallel), IS (Integer Sorting), and CG (Conjugate Gradient).

Besides this introduction, this paper comprises four sections. Section II describes the structure of Haskell\# parallel programs. Section III describes the Haskell\# implementation of the NPB kernels CG, IS and EP, discussing aspects relative to parallel software engineering with Haskell\#. Section IV presents the speedup and efficiency figures of Haskell\# and MPI implementations of the kernels, comparing the results obtained. Finally, the conclusions and lines for further work are presented.

II. HASKELL\# PROGRAMMING

Haskell\# is a distributed parallel extension to Haskell, aimed at distributed architectures and based on coordination paradigm\[10\]. The specification of computations is separated from the management of parallelism by a process hierarchy. Functional processes are thus instantiated from a collection of Haskell functional modules, which implement computations. Processes have their ports connected by channels, using a configuration language, called HCL (Haskell\# Configuration Language). Haskell\# goes beyond other coordination parallel languages, by making coordination and computation media (HCL and Haskell) orthogonal. No extensions are needed to the Haskell code at computation level for interfacing computation and coordination media, avoiding any kind of overlapping between parallel and computational code. Thus, it is possible to program functional modules without making any assumption on how the processes instantiated are integrated in the process network topology. Haskell\# intends to provide a higher modular, abstract, efficient and scalable component based framework for developing complex and large scale parallel applications on clusters.

The basic building blocks of Haskell\# programs are components, each one implementing a certain functionality. They are essentially described by a collection of entry and exit points. Components may be simple or compound. Simple components are Haskell modules, while compound components are HCL configurations. The entry points of simple components are the arguments of the main function of the functional module, while exit points are the elements of the tuple returned by it in an I/O action. In compound components, each entry point is plugged onto a free input port of some unit in the configuration, while exit points are plugged onto output ports. Simple components, or functional modules, are sequential components, while compound components, or configurations, are parallel components. A Haskell\# program is instantiated from a component, called application component.

A HCL program (configuration) declares a number of units. Units are executable and have an interface. A component must be assigned to a unit. The interface describes how a unit interacts with other units, by defining a collection of input and output ports, an expression that control the order in which they are activated (communication behaviour), and behavior restrictions that relate interface communication behavior to the communication behavior of other existing interfaces. The assigned component describes the unit functionality. This orthogonality of concepts makes possible to
model unit communication behaviour as 0-counter synchronized concurrent expressions [17], a formalism equivalent to Petri nets. Using this approach, Haskell₉ allows to analyze the structure and dynamic behaviour of the network of parallel processes without any assumption on what they do and how they are implemented (sequentially or in parallel).

Whenever declaring the interface and component of a unit, an injective mapping from entry and exit points of the component onto input and output ports of the interface must be defined. A value is provided explicitly to the entry points that are not associated with an input port, while the value produced by exit points not connected to an output port is ignored. A wire function should be specified to transform values between entry and exit points and input and output ports. If not specified, it is assumed as the identity function. The wire function maps a value of type \( t \) to a type \( u \).

In a unit declaration, after the mapping process, it is possible to replicate an interface port, forming a group of ports. Each port is associated with an entry or exit point. The default wire function for groups of ports is choice, which chooses one of the ports in the group to perform communication. The wire function for input and output ports, respectively must have the following types:

\[
\text{wire_in} :: \text{Int} \rightarrow (\text{Int} \rightarrow t) \rightarrow u \\
\text{wire_out} :: \text{Int} \rightarrow u \rightarrow (\text{Int} \rightarrow t)
\]

Units are synchronized by connecting their ports to form channels. Channels define a mapping between output ports and input ports of the units of the component. Ports that do not belong to a channel must be plugged onto the entry and exit points of the component. Channels are unidirectional, point-to-point, synchronous, and typed. Buffers may be defined to give asynchronous semantics for communication.

It is possible to define a unit without assigning a component to it. This kind of unit is called a virtual unit. A component that possess at least one virtual unit is called an abstract component. Application components cannot be abstract. Virtual units are used to model partial topological skeletons [14]. Thus, virtual components are also referred in this work as skeletons. A unit can have a skeleton as corresponding component. It is possible to assign an existing unit (virtual or not) to a virtual unit, instantiating it on the network of functional processes. In this case, the communication behaviour of the two units must be “compatible”. There is an homomorphism from the virtual unit communication behavior to the assigned one.

Virtual units can be unified or factorized to defined new virtual units. Unification is an operation that takes a collection of virtual units and defines one new unit that assumes their role in the network. Factorization allows to split a virtual unit into many units. These operations must respect behaviour preserving restrictions. Virtual units from different components can be unified. This allows to compose new skeletons from existing ones, by overlapping or nesting them.

A Haskell₉ program is finally defined by an application module, a configuration where one unit (normally, a cluster) named main is specified.

III. IMPLEMENTING SPMD PROGRAMS WITH HASKELL₉

SPMD (Single Program, Multiple Data) is an important model for developing parallel programs. Several high-performance scientific applications fit this model, yielding efficient parallel implementations. SPMD model is particularly suitable for data parallel applications over distributed architectures.

The Haskell₉ model of parallelism allows implementing a collection of heterogeneous processes that synchronize in a network, as a “MPMD (Multiple Program, Multiple Data)” model, a more general and functional approach. This section shows that SPMD applications can be implemented natural and efficiently using Haskell₉. For that, a sub-set of the NAS Parallel benchmarks (NPB), a well-known benchmark for parallel architectures, was chosen. Besides the large dissemination amongst the scientific community, another important motivation for the use of NPB is its availability in several languages, allowing to make performance comparisons.

A. The NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) [19, 16] is a collection of eight applications, specified by the NASA Research Center at Ames, for evaluating the throughput of high-performance computing architectures in the execution of the NAS program (Numerical Aerodynamic Simulation), a great effort conducted at that time to advance the state-of-the-art of computational aerodynamics. The applications are:

- **EP** (Embarrassingly Parallel): Generation of pairs of Gaussian deviates according to a specified scheme and tabulation of the number of pairs in successive square anuli;
- **IS** (Integer Sorting): Parallel sort over small integers;
- **CG** (Conjugate Gradient): Solving an unstructured sparse linear system by the conjugate gradient method.
- **FT** (Fast Fourier Transform): A 3-D fast fourier transform partial differential equation benchmark;
- **MG** (Multigrid): A simple 3D multigrid benchmark;
- **LU** (LU solver);
- **BT** (Block Tridiagonal Solve);
- **SP** (Pentadiagonal Solver);

The EP, IS, CG, FT, and MG are kernel benchmarks, whereas LU, BT, and SP are application benchmarks [19].

The version 1.0 of NPB [19] presented “paper and pencil” specification of NPB programs, abstracting from imple-
mentation concerns. This is due to the variety and intrinsic characteristics of parallel architectures, which should be considered only by NPB implementors. To standardize measurements, classes of problems (A and B) were defined, according to the amount of input data processed by each benchmark. Several restrictions and conventions, such as which languages and libraries are allowed, were defined to make valid the measurements conducted by implementors, making comparisons reliable.

The consolidation of distributed architectures, mainly after advent of clusters, motivated version 2.0 of NPB to appear. Now, a collection of standard SPMD portable implementations of the kernels, sequential and distributed, were introduced, written in FORTRAN and C, and using MPI for parallel execution. Also, a new class (C) of problem was introduced, reflecting the increase in the performance of architectures since the introduction of NPB 1.0. More recently, version 3.0 of NPB presented JAVA\textsuperscript{T M} and HPF versions of the benchmarks. An OpenMP implementation is now being implemented.

This work presents the implementation of three of the five NPB kernels (EP, IS, and CG) in Haskell\textsubscript{\$}, for evaluating the effectiveness of use of Haskell\textsubscript{\$} in the development of SPMD scientific applications, in relation to parallel software engineering and performance speedup and efficiency. Our implementation is based on version 2.3 of NPB, which contains C or Fortran implementations of the NPB benchmarks, using MPI to manage parallel execution of processes.

In what follows, Haskell\textsubscript{\$} implementations are described.

### A.1 The Embarassinly Parallel (EP) Kernel

The EP (Embarassinly Parallel) kernel benchmarks generate pairs of Gaussian random deviates according to a specific scheme\cite{19} and tabulate the number of pairs in successive square annuli. It was developed to estimate the upper achievable limit for floating point performance (performance without significant inter-processor communication) in a parallel architecture.

The HCL configuration of EP code is shown in Figure 3.

The implementation of EP in Haskell\textsubscript{\$} is the most simple presented in this paper. The network of functional processes comprises \( n \) homogeneous units, named \( \text{ep\_unit}[i] \), \( i \) varying from 1 to \( n \). They are instanced from the same interface and associated to the same component. Their interface describes three input ports (\( \text{sx\_total}, \text{sy\_total}, \) and \( q\_\text{total} \)) and three output ports (\( \text{sx}, \text{sy}, \) and \( q \)). The component describes the EP functionality. It has four entry and three exit points. One of the entry points is used to provide parameters for EP functional module, which define the applied problem instance. The other ones are mapped onto the interface ports of the unit.

The connectivity of the network (definition of channels) configuration:\[ \text{EP}<\text{NO\_NODES,MK, MM, NN, NK, NQ, EPSILON, A, S}> \]

with:

\[ \text{index \_ \_ \_ range \ [1..\text{NO\_NODES}] \} } \]

```
use Skeletons.MPIAllReduce
use EP

interface EP (sx\_total, sy\_total::Double; q\_total::UDVector) \( \rightarrow \) (sx, sy, q)
    behaving as AllReduce \# sx\_total \( \rightarrow \) sx
    behaving as AllReduce \# sy\_total \( \rightarrow \) sy
    behaving as AllReduce \# q\_total \( \rightarrow \) q
    behaving as: seq \{ sx!; sx\_total?!; sy!; sy\_total?!; q!; q\_total?! \}

unit sx\_comm as AllReduce<\text{NO\_NODES}>
unit sy\_comm as AllReduce<\text{NO\_NODES}>
unit q\_comm as AllReduce<\text{NO\_NODES}>
[/ unify sx\_comm.p[i] \# sx\_total \( \rightarrow \) sx,
    sy\_comm.p[i] \# sy\_total \( \rightarrow \) sy,
    q\_comm.p[i] \# q\_total \( \rightarrow \) q
    to ep\_unit[i] \# EP (sum\_pums sx\_total,
    sum\_piums sy\_total, sum\_vems q\_total) \( \rightarrow \) (broadcast sx
    broadcast sy, broadcast q)
    as EP \{\# Params \%1 \%2 \%3 \%4 \%5
    \%6 \%7 \%8 \%9 \%10 \} \text{\_NO\_NODES}
    MK MM NN NK NQ EPSILON A S,
    sx\_total, sy\_total, q\_total) \( \rightarrow \) (sx, sy, q)
```

Fig. 3. HCL Code of EP Kernel

![Fig. 4. EP Benchmark (Speedup Curve)](image-url)
is specified using three instances of the AllReduce skeleton, each one involving three disjoint sub-sets of ports: $sx$ prefixed ports, $sy$ prefixed ports, and $q$ prefixed ports. This reflects the occurrence of three calls to MPI_AllReduce primitives in the original Fortran code of EP. The cluster units $sx_{comm}$, $sy_{comm}$, and $q_{comm}$ are the skeleton instances. For that, they have no interface, but an AllReduce virtual component (skeleton) is assigned to them. The virtual units that comprise the cluster units $sx_{comm}$, $sy_{comm}$, and $q_{comm}$ are then unified to define the virtual topology of EP Haskell program. Then, $ep_{unit}$'s process are assigned to the unification resulting collection of virtual units.

Besides providing a higher-degree of modularity, the use of the MPISkeleton allows to provide topological information to the Haskell compiler in order to generate a more suitable use of MPI collective communication primitives, optimizing performance. The same approach is used in the implementation of the other benchmarks.

A.2 The Integer Sort (IS) Kernel

The IS kernel performs parallel sorting of $N$ keys using the bucket sort algorithm. The keys are generated by a sequential algorithm specified in [19] and must be uniformly distributed in memory.

The code for HCL configuration that describe process network topology of IS kernel is presented in Figure 5.

The SPMD network of functional processes of IS comprises $n$ processes, named $is_{unit}[i]$, with $i$ varying from 1 to $n$. As in the kernel EP, the interfaces of IS units are described by three input ports and three output ports. There is a cyclic pattern of communication (repeat alt), because there are stream ports on the interface. They model the fact that, in the original code, the information corresponding to these ports are communicated, using MPI primitives, in the context of an iteration. Each element in the stream corresponds to a value communicated at an iteration (call to rank procedure in the FORTRAN code). The component assigned to the $is_{unit}$'s processes, which defines its functionality, has four entry points and three exit points. The first entry point is used to give problem instance parameters to the functional module, while the others are mapped onto corresponding interface ports.

As in EP, MPI collective communication skeletons are employed to define the IS network topology. But now two distinct ones are needed: AllReduce and AllToAllv. The ports prefixed by $bs$ are connected using the AllReduce pattern, while that prefixed by $kb$ are connected using AllToAllv. The cluster units $bs_{comm}$ and $kb_{comm}$ are used to instantiate the MPI unit skeletons.

Besides them, another skeleton is used: $RShift$. It models a data right shift amongst a collection of $m$ processes. For that, virtual units indexed from 2 to $m - 1$ inside a $k_{shift}$-configuration IS<PROBLEM_CLASS, NUM_PROCS, MAX_KEY_LOG2, NUM_BUCKETS_LOG2, TOTAL_KEYS_LOG2, MAX_ITERATIONS, MAX_PROCS, TEST_ARRAY_SIZE> with

index i range [1, NUM_PROCS]

use IS
use Skeletons.MPI.AllReduce
use Skeletons.MPI.AllToAllv
use Skeletons.Misc.Shift

interface IS (bst*::UArray Int Int, kb2*::(Int, Ptr Int), kl::Int) → (bs*:UArray Int Int, kb1*::(Int, Ptr Int), kr::Int)

behaving as AllReduce # bst → bs,
behaving as AllToAllv # kb2 → kb1,
behaving as Shift # kr → kl
behaving as: seq { repeat alt bs! → seq{ bst?, kb1! kb2? }; }; kr!; kl? ;

unit bs_comm as MPI_AllReduce<NUM_PROCS>
unit kb_comm as MPI_AllToAllv<NUM_PROCS>
unit k_shift as shift<NUM_PROCS> 0 → .

[/ unify bs_comm.p[i] # bucket_size_totals's → bucket_size's,
 kb_comm.p[i] # key_buff2's → key_buff1's,
 k_comm.p[i] # k_left → k_right
to is_unit[i] # IS (sum_vec's bucket_size_totals's,
 combine_keys key_buff2's, k_left) → (broadcast_bucket_size's, distribute_keys key_buff1's, k_right)
as IS (Params PROBLEM_CLASS NUM_PROCS MAX_KEY_LOG2 NUM_BUCKETS_LOG2 TOTAL(keys LOG2 MAX_ITERATIONS MAX_PROCS TEST_ARRAY_SIZE, bucket_size_totals's, key_buff2's, k_left) → (bucket_size's, key_buff1's, k's_right)
]

Fig. 5. HCL Code of IS Kernel

Fig. 6. IS Benchmark (Speedup Curve)
cluster unit instantiate the Shift skeleton. Each of them has one input and one output port. The virtual unit indexed by 1 has only one output port, while the virtual unit indexed by \( n \) has only one input port.

The virtual units of \( bs_{\text{comm}}, kb_{\text{comm}}, \) and \( k_{\text{shift}} \) units are unified to form the virtual network topology of IS. Then, \( is_{\text{unit}}' \)s are assigned to the resulting collection of virtual units, buiding the application topology.

A.3 The Conjugate Gradient (CG) Kernel

The CG kernel benchmark implements a solution to an unstructured sparse linear system, based on the conjugate gradient method. The inverse power method is used to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of nonzeros.

The original CG network topology, implemented in Fortran/\( \text{MPI} \), comprises \( n \) process, where \( n \) is a power of two, organized in a rectangular mesh. The sparse matrix elements are uniformly distributed amongst processors, according to their topological organization. Assume that \( r \) and \( c \) denote the number of rows and columns of the mesh \( (r \times c = n) \), respectively. If \( n \) is a perfect square then \( r = c \), else \( c = 2 \times r \).

In order to demonstrate the higher expressivity of \( \text{HCL} \) for defining concisely intricated process topologies, a generalization of the original CG process topological organization was used in the Haskell\( \# \) implementation, but without additional programming effort. Now, it is also organized in a rectangular mesh, but now there is no restriction to the value of \( n \). The programmer must provide parameters \( dim \), which value correspond to the number of mesh rows, and \( \text{col} \_\text{factor} \), which, when multiplied to \( dim \), gives the number of mesh columns. Any number of units (\( n \)) can be configured using this approach, but notice that more than one configuration of \( dim \) and \( \text{col} \_\text{factor} \) can result in the same value for \( n \), but with different performance characteristics. In a real application, the programmer could adequate the configuration of these parameters to better execution on a certain environment, assuming problem instance characteristics. The units(processes) are named \( cg_{\text{unit}}[i][j] \), where \( i \) is its row position and \( j \) is its column position. The HCL code for CG process topology in Haskell\( \# \) is presented in Figure 7.

The interface that describes \( cg_{\text{unit}}' \)s ports and communication behaviour defines five input ports, named \( q\_i, \ rho\_i, \ aux\_i, r\_i', \) and \( \text{rnorm}_i \), and seven output ones, named \( q\_o, \ rho\_o, \ aux\_o, r\_o', \text{rnorm}_o, x, \) and \( \text{zeta} \). The CG functional module has six entry points and seven exit points. The first entry point receives the parameters that configure problem instance, while the others are mapped onto the interface ports.

Two skeletons are used to define CG communication topology: \( \text{AllReduce} \) and \( \text{Transpose} \). The first one is used in data exchange performed in parallel scalar products at pro-

configuration

\[
\text{CG}<\text{DIM}, \text{COL} \_\text{FACTOR}, \text{NA}, \text{NONZER}, \text{SHIFT}, \text{NITER}, \text{RCOND} > \text{ZETA}_{\text{VERIFY} \_\text{VALUE}> (Q) \rightarrow (zeta, x) \text{ with }}
\]

use \( \text{Skeletons.MPI.MPI}\_\text{AllReduce} \)
use \( \text{CG}\_\text{Transpose} \)
use \( \text{CG} \)

index i range [1..DIM]
index j range [1..\text{COL} \_\text{FACTOR}]

interface \( \text{CG} \) # (q\_i*, r\_i* :: UDVector;
\]
\( \text{rho} \_i**, \text{aux} \_i**, \text{rnorm} \_i** :: Double) \rightarrow
\( \text{q} \_o**, r\_o*, x :: UDVector;
\]
\( \text{rho} \_o**, \text{aux} \_o**, \text{rnorm}\_o*, \text{zeta} :: Double;
\]
\( \text{x} :: UDVector) \)

behaving as:
\[
\text{seq \{ q\_o!; q\_j!; \}}
\]
\[
\text{repeat alt \{ q\_o! → seq \{ repeat alt \{ q\_i? → seq \{ aux\_o!; aux\_j?; rho\_o!; rho\_j?; q\_o!; q\_i?; aux\_o!; aux\_j?; rho\_o!; rho\_j; r\_o!; r\_j!; \text{rnorm}_o!; \text{rnorm}_j!; \text{rho}_o!; \text{rho}_j \}
\}
\}
\)
\[
\text{q\_j};
\]
\[
\text{r\_o!}; \text{r\_j!};
\]
\[
\text{rnorm}_o!; \text{rnorm}_j!
\}
\]

unit q\_comm as CG\_\text{Transpose}<\text{DIM, DIM} \_* \text{DIM} \_* \text{COL} \_\text{FACTOR} >

unit r\_comm as CG\_\text{Transpose}<\text{DIM, DIM} \_* \text{DIM} \_* \text{COL} \_\text{FACTOR} >

[/ unit \( \text{rho}_{\text{comm}}[i] \) as AllReduce<\text{DIM} \_* \text{COL} \_\text{FACTOR}> /

[/ unit \( \text{aux}_{\text{comm}}[i] \) as AllReduce<\text{DIM} \_* \text{COL} \_\text{FACTOR}> /

[/ unit \( \text{norm}_{\text{comm}}[i] \) as AllReduce<\text{DIM} \_* \text{COL} \_\text{FACTOR}> /

[/ \text{unify} q_{\text{comm}}.u[i][j] \# q\_j → q\_o,
\]
\( r\_{\text{comm}}.u[i][j] \# r\_j → r\_o, \)
\( \text{rho}_{\text{comm}}[i].p[j] \# r\_j → r\_o, \)
\( \text{aux}_{\text{comm}}[i].p[j] \# \text{aux}_j → \text{aux}_o, \)
\( \text{norm}_{\text{comm}}[i].p[j] \# \text{norm}_j → \text{norm}_o \)
to \( \text{cg}[i][j] \) # CG (q\_i, r\_i*, rho\_i, aux\_i, norm\_i) →
\( \text{(q}\_o, r\_o, z, rho\_o, aux\_o, norm\_o, x, \text{zeta}) \)
as CG (Parameters DIM (DIM*COL\_FACTOR) NA NONZER SHIFT NITER RCOND ZETA\_VERIFY\_VALUE>
\]
\( \text{q}\_i, \text{r}\_i*, \text{rho}_i, \text{aux}_i, \text{norm}_i) →
\( (\text{q}_o, \text{r}_o, z, \text{rho}_o, \text{aux}_o, \text{norm}_o, \text{zeta}, \text{x}) \)
[/]

Fig. 7. HCL Code of kernel CG
To each row of processes, three AllReduce cluster units are necessary to define CG topology, named rho_comm[i], aux_comm[i], and rnorm_comm[i], where i varies from 1 to rows. They connect ports prefixed by rho, aux and rnorm, respectively. Only two cluster units define Transpose based patterns of communication, involving all processes in the network. They are called q_comm and aux_comm, connecting processes prefixed by q and aux, respectively. As in the other kernels, the virtual units in the context of the comm-suffixed cluster units are unified to form the overall network topology. The processes are then assigned to the resulting set of virtual units, configuring computation performed by the component.

The code for Transpose skeleton is presented in Figure 9. This skeleton organize its virtual units according to the values of parameters dim and col_factor, provided in the CG configuration. Firstly, a squared mesh of units with dimension dim is organized. The ports are connected in order to transpose data among processors using appropriate wire functions over groups of ports. Each one of these units are then factorized in col_factor units, resulting in a squared mesh with dim rows and dim*col_factor columns. The diagram in Figure 10 illustrate construction of Transpose skeleton. Notice that, after factorization, nested groups of ports appear connecting Transpose units.

IV. Benchmark Results and Discussion

In Figures 4, 6, and 8, the speedup curves for Haskell# versions of IS, EP and CG, respectively, are presented, comparing them to that obtained for their respective lower-level MPI versions.

The benchmark measures were performed in a cluster composed by 16 Pentium IV 2GHz nodes, with 512MB of RAM, interconnected by a 100 Mbs ethernet. Because technical problems, at most 13 machines were available on measuring time, which let us to make measures for 2, 4, and 8 processors.

In order to avoid influence of excessive use of virtual memory in our performance measures, different problem sizes were chosen for Haskell# and MPI versions of the kernels, assuming the different space behavior of the lazy functional language Haskell and the imperative languages C and Fortran. For example, for EP kernel, the standard problem sizes S and B were respectively used to measure performance of its Haskell# and Fortran/MPI versions. The Haskell sequential version finished about 350 seconds to run the class rows, and the latter is used in parallel matrix multiplications, whenever a transpose operation is performed on data stored in processors. In MPI original code, several calls to the MPI_Irecv primitive are needed to perform these operations, making difficult to understand topology structure without careful analysis of problem instance parameters.
resented as immutable arrays, a source of inefficiency that require to use a Haskell function to sum a list of vectors, re-
presentive to Haskell. The programmer could then choose the ap-
propriate language to program a certain functionality present in
the language Haskell, despite the last decade improvements in functional languages compilation technology, are still very poor to attend performance requirements of real applications in high performance computing. This fact could be demonstrated by comparing execution times and space behavior of sequential versions of NAS benchmarks implemented in Haskell and Fortran (EP) or C (IS). Because that, different problem sizes were used to implement EP, IS and CG in each compared language.

An important further advance in Haskell$_\#$ environment is to design a framework intended to support the functionalities of sequential or parallel widespread scientific libraries as reusable black-box components, in the Haskell$_\#$ sense. A multi-lingual approach, with integration of Haskell$_\#$ to other existing languages, like JAVA, C or FORTRAN is also been considered in further developments, allowing for programming sequential modules using other languages, in alternative to Haskell. The programmer could then choose the appropriate language to program a certain functionality present in a program. In this framework, Haskell could be viewed as a prototype language, used to guide programmers of sequential (functional) modules in the development and validation phases. However, it is necessary to investigate carefully how to maintain syntactic orthogonality between the configuration and the host language using a language that does not support lazy evaluation and non-strict semantics.

The work in comparing Haskell$_\#$ implementations of some known programs to their versions written in other parallel, high-performance, languages is still beginning. Besides to implement the other five NPB benchmarks and applications, it is intended to build implementations of some real applications, such as climate modeling and parallel oil reservoir simulation, in order to make better analysis of Haskell$_\#$ approach for parallel programming.

V. Conclusions

This paper presented Haskell$_\#$ implementations for a subset of the NAS parallel benchmarks, including EP, IS, and CG, having two main goals:

- To advocate the advantages of using Haskell$_\#$ to de-
velop SPMD programs, an important paradigm used to implement most of the high-performance parallel applications, in general supported by widespread message passing libraries such as MPI. Most of the advantages are extensible to more general parallel programming approaches;

- To compare performance scalability of Haskell$_\#$ and lower-level MPI programs, in order to conclude that the increasing in parallel programming abstraction and modularity due to the use of Haskell$_\#$ does not means significative overhead on their performance.

In fact, the performance figures for a poorly optimized version of Haskell$_\#$ have shown evidences that it is possible to reconcile higher level of abstraction, modularity, simplicity and higher performance scalability in the development of parallel programs.

However, the results presented herein have also shown that the language Haskell, despite the last decade improvements in functional languages compilation technology, are still very poor to attend performance requirements of real applications in high performance computing. This fact could be demonstrated by comparing execution times and space behavior of sequential versions of NAS benchmarks implemented in Haskell and Fortran (EP) or C (IS). Because that, different problem sizes were used to implement EP, IS and CG in each compared language.

An important further advance in Haskell$_\#$ environment is to design a framework intended to support the functionalities of sequential or parallel widespread scientific libraries as reusable black-box components, in the Haskell$_\#$ sense. A multi-lingual approach, with integration of Haskell$_\#$ to other existing languages, like JAVA, C or FORTRAN is also been considered in further developments, allowing for programming sequential modules using other languages, in alternative to Haskell. The programmer could then choose the appropriate language to program a certain functionality present in a program. In this framework, Haskell could be viewed as a prototype language, used to guide programmers of sequential (functional) modules in the development and validation phases. However, it is necessary to investigate carefully how to maintain syntactic orthogonality between the configuration and the host language using a language that does not support lazy evaluation and non-strict semantics.

The work in comparing Haskell$_\#$ implementations of some known programs to their versions written in other parallel, high-performance, languages is still beginning. Besides to implement the other five NPB benchmarks and applications, it is intended to build implementations of some real applications, such as climate modeling and parallel oil reservoir simulation, in order to make better analysis of Haskell$_\#$ approach for parallel programming.

REFERENCES


