Towards a Generalised Runtime Environment for Parallel Haskells*

Jost Berthold

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans-Meerwein-Straße, D-35032 Marburg, Germany
berthold@informatik.uni-marburg.de

Abstract. Implementations of parallel dialects (or: coordination languages) on a functional base (or: computation) language always have to extend complex runtime environments by the even more complex parallelism to maintain a high level of abstraction. Starting from two parallel dialects of the purely functional language Haskell and their implementations, we generalise the characteristics of Haskell-based parallel language implementations, abstracting over low-level details. This generalisation is the basis for a shared runtime environment which can support different coordination concepts and alleviate the implementation of new constructs by a well-defined API and a layered structure.

1 Introduction

The area of parallel functional programming exhibits a variety of approaches with the common basis of referential transparency of functional programs and the ability to evaluate subexpressions independently. Some approaches pursue the target of (half-)automatic parallelisation of operations on list-like data structures (i.e. *data parallelism*). Other dialects are more explicit in their handling of parallelism and allow what we call *general-purpose parallelism*, able to capture schemes of parallelism which are not data-oriented. Whereas machine-specific optimisation is easier with specialised structures and operations, these more general approaches present a considerable advantage in language design: It is generally accepted [1, 2] that functional languages allow a clean distinction between a computation (or "base") language and independent coordination constructs for parallelism control. The more special-purpose data structures enter the language, the more vague this important distinction will become.

Implementations of general-purpose languages often resemble each other, differing in syntactic sugar or providing special domain-specific features. Thus, in the implementation, many concepts can be reduced to a common infrastructure enriched with a small amount of special features depending on the concrete, language-specific coordination concept. In this paper, we present an overview of the two main-stream general-purpose parallel extensions to the functional language Haskell, focusing on implementation aspects. Our aim is to bring together

* Work supported by ARC/DAAD Grant No. D/03/20257

related implementations in a generalised runtime environment (RTE), capable of supporting different language concepts from the same configurable base system.

The paper is organised as follows: Section 2 presents the two parallel languages based on Haskell, the starting point of our work. In Section 3, we systemise the functionality a common base system must provide to implement both languages, or even a combination of their features. A short example in Section 4 indicates the potential use of such an API. Section 5 concludes.

2 General-Purpose Parallelism in Haskell

The Haskell Community Report [3] mentions two major approaches to parallel computation based on Haskell: *Glasgow parallel Haskell* [4] and *Eden* [5]. These two languages show major differences in their coordination concept and, in consequence, in their implementation, while on the other hand, they both capture the main idea of evaluating independent subexpressions in parallel. Another common point is that, in contrast to other parallel Haskells [2], both GpH and Eden are designed as general-purpose coordination languages. Neither of them is dedicated to a certain paradigm such as task- or data-parallelism or pure skeleton-based programming, though the respective coordination schemes can be expressed by both. Both language extensions are implemented using GHC [6] as a platform. The implementation is even sharing infrastructure code, e.g. for basic communication and system setup, but diverging whenever the different languages require specific features.

2.1 Glasgow Parallel Haskell

Glasgow Parallel Haskell (GpH) [4] is a well-known parallel dialect of Haskell investigated since the 90's. The overall paradigm of GpH is semi-implicit data and task parallelism, following annotations in the source program. In every definition, subexpressions can be marked as "suitable for parallel evaluation" by a `par`-expression in the overall result. The coordination construct `par` takes 2 arguments and returns the second one after recording the first one as a "spark", to evaluate in parallel. An idle processor can fetch a spark and evaluate it. The built-in `seq` is the sequential analogon, which forces evaluation of the first argument before returning the second one.

```
par,seq :: a -> b -> b
```

These coordination atoms can be combined in higher-order functions to control the evaluation degree and its parallelism without mixing coordination and computation in the code. This technique of *evaluation strategies* described in [7] offers sufficient evaluation control to define constructs similar to skeleton-based programming. However, as opposed to usual skeletons, parallelism always remains semi-implicit in GpH, since the runtime environment (RTE) can either ignore any spark or eventually activate it.

The implementation of GpH, GUM [4], relies essentially on the administration of a distributed shared heap and on the described two-stage task creation

mechanism where potential parallel subtasks first become local sparks before they may get activated. The only access point to the system is the spark creation primitive; parallel computations and their administrative requirements are completely left to the RTE and mainly concern spark retrieval and synchronisation of a distributed heap. Once a spark gets activated, the data which is evaluated in parallel could subsequently reside on a different processor and thus must get a global address, so it can be sent back on request.

The main advantage of the implicit GpH concept is that it dynamically adapts the parallel computation to the state and load of nodes in the parallel system. The GpH implementation would even allow to introduce certain heuristics to reconfigure the parallel machine at runtime. However, parallel evaluation on this dynamic basis is hardly predictable and accessible only by simulation and tracing tools like GranSim [8].

2.2 Eden

The parallel Haskell dialect Eden [5] allows to define *process abstractions* by a constructing function `process` and to explicitly *instantiate* (i.e. run) them on remote processors using the operator `( # )`. Processes are distinguished from functions by their operational property to be executed remotely, while their denotational meaning remains unchanged as compared to the underlying function.

```
process :: (Trans a, Trans b) => (a -> b) -> Process a b
( # )   :: (Trans a, Trans b) => Process a b -> a -> b
```

For a given function `f`, evaluation of the expression `(process f) # arg` leads to the creation of a new (remote) process which evaluates the application of the function `f` to the argument `arg`. The argument is evaluated locally and sent to the new process. The implementation of Eden [9] uses implicit inter-process connections (*channels*) and data transmission in two possible ways: single-value channels and stream channels. Eden processes are encapsulated units of computation which only communicate via these channels. This concept avoids global memory management and its costs, but it can sometimes duplicate work on the evaluation of shared data structures.

Communication between processes is automatically managed by the system and hidden from the programmer, but additional language constructs allow to create and access communication channels explicitly and to create arbitrary process networks. The task of parallel programming is further simplified by a library of predefined skeletons [10]. Skeletons are higher-order functions defining parallel interaction patterns which are shared in many parallel applications. The programmer can use such known schemes from the library to achieve an instant parallelisation of a program.

The implementation of Eden extends the runtime environment of GHC by a small set of primitive operations for process creation and data transmission between processes (including synchronisation and special communication modes). While the RTE shares basic communication facilities with the GpH implementation, the implementation concept is essentially different in that the needed

primitives only provide simple basic actions, while more complex operations are encoded by a superimposed functional module. This module, which encodes process creation and communication explicitly, can abstract over the administrative issues, profiting from Haskell's support in genericity and code reuse [9], moreover, it protects the basic primitives from being misused.

3 Generalising the Parallel Runtime Environment

3.1 Layered Implementation

To systemise the common parts of parallel Haskell implementations, we follow the approach of Eden's layered implementation, i.e. thick layers of functionality exploited strictly level-to-level to avoid dependencies across abstraction levels. Apart from maintenance of only one system, the concept of layered implementation is promising for the implementation of other coordination languages based on Haskell, since it facilitates maintenance and experimental development. With one flexible basic layer, different language concepts can be easily implemented by a top-layer module (making use of the underlying RTE support) where the appropriate coordination constructs are defined. As shown in Fig. 1, the *Evaluation Strategy* module for GpH [7] is just an example of these high-level parallelism libraries. As well as Eden, one could implement explicit parallel coordination e.g. for some data-parallel language by adding an appropriate module for all parallel operations to their sequential implementation.

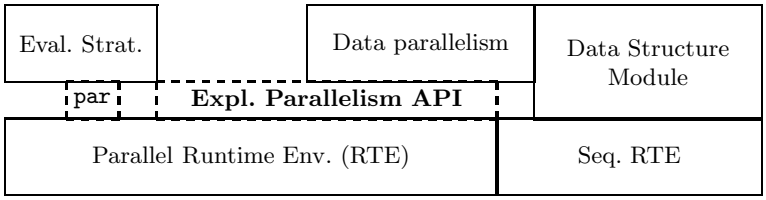

Fig. 1. Layered implementation of parallel coordination (example)

These top-layer modules use parts of a common ParallelAPI provided by the underlying runtime layer, which we will describe now in terms of the supported operations. A more basic concept is the overall coordination in Haskell-based parallel languages, which relies on parallel graph reduction with synchronisation nodes representing remote data (not described further due to space limitations, see [1]). In order to support implicit parallelism, a generalised RTE will also have to support virtual shared memory and implicit, load-dependent task creation. Concepts for implicit coordination use this basic RTE support, which is not accessible from language level and thus not part of the API.

3.2 ParallelAPI of Primitive Operations

Functionality to coordinate a parallel computation will be implemented in a set of primitive operations (i.e. the ParallelAPI) which exposes functionality to

Table 1. API Operations

Primitive :: Type	Description
`rTask :: PE -> a -> ()`	actively spawn a process on a remote PE
`fork :: a -> b -> b`	spawn new thread (same process)
`par :: a -> ()`	passively mark data as "potentially parallel"
`createDC :: ChanMode -> Int -> ( ChanName a, a)`	create new channel and sync. nodes in the local heap (type `a` expected)
`connectC :: ChanName a -> ()`	connect to a channel to write data
`dataSend :: SMode-> a -> ()`	send subgraph from one PE to another
`conSend :: a -> ChanName b -> ()`	send top-level constructor and reply channel for destination of its components
`noPE :: Int`	number of available nodes in parallel system
`myPE :: PE`	node ID where caller is located (unsafe)

define high-level coordination, intended for language developers. It should be complete in that it allows a variety of different coordination possibilities and ideally orthogonal in that it provides only one way to accomplish a certain task. Functionality exposed by the API falls into three categories: task management, explicit communication and system information, shown in Tab.1.

Task Management. The obvious issue in task management is to start remote computations, which requires a primitive `rTask` to actively transfer a subgraph to a remote processor for evaluation. The receiver of a transferred task creates a thread to evaluate the included subgraph, i.e. solve the task. The primitive in itself does not imply any communication between parent and child process, it is due to the caller and the transferred task to establish this communication. Tasks can be explicitly placed on certain nodes of the parallel system when required, otherwise, the RTE uses configurable placement patterns or random placement. The conceptual entities of computation in this description follow the style of Eden: The primitive `rTask` sends a subgraph (task) to a remote node, which creates a thread for its evaluation. A *task* (subgraph to be evaluated) thereby turns into a *process* containing initially one evaluation *thread*. This first thread can then `fork` to create other threads, which all reside conceptually in the same process.

An implicit variant of active task creation is the two-stage concept of GpH, which only records subgraphs as "sparks" suitable for remote evaluation. The API supports spark creation by an annotation primitive `par`, but the action of turning a spark into a thread cannot be exposed to language level, as well as all necessary data must be shared via global memory, since it cannot be explicitly transferred.

Communication. The API will expose communication functionality for (completely or partially) transferring subgraphs from one process to another via Eden-style *channels*, linking to placeholders in the evaluation graph. Channels are created using the primitive `createDC`, and configured for 1:1 or n:1-communication upon their creation. Once created, channels have a representation at the language level and can be transferred to other threads just as normal data, enabling to build arbitrary process networks. Threads in different processes can then use `connectC` to connect to the channel and send data.

Values (i.e. subgraphs) are sent through the created channels using a primitive `dataSend`, which can have several modes: Either a *single value* is sent, which implies that the channel becomes invalid (since the receiver replaces the respective placeholder by received data), or the data is an element in a *stream* and leaves the channel open for further elements, recomposed to a list in the receiver's heap. As a general rule, data transmitted by `dataSend` should be in *normal form*, i.e. sent only after complete evaluation by the sender. Fully evaluated data can be duplicated without risk, whereas data shared via global memory synchronisation inside the RTE can have any state of evaluation, and unevaluated data should be *moved* instead of copied to keep global references synchronised.

Another, more complex, variant of data transmission is to send only the top-level constructor using `conSend`, which requires the receiver to open and send back new channels for the arguments of this constructor. This rather complex implementation is the way to overrule the principle of normal form evaluation prior to channel communication, and we can imagine useful applications.

System Information. In order to profit from explicit process placement, the runtime setup must be accessible by the API to determine how many nodes are available and, for every process, on which node it is executed. While the primitive for the former, `noPE`, is at least constant during the whole runtime, the latter information, determined by `myPE`, ultimately destroys referential transparency (unsafe). On the other hand, this feature is useful to place processes on appropriate nodes; it should be used only for coordination purposes.

3.3 Functionality of the Basic System Layer

As mentioned, implicit, load-driven task creation and synchronisation must enable all threads to share data, and use a two-stage task creation in order to allow decent evaluation control by the programmer. The global memory management, as well as other system tasks such as basic communication and system management, reside in the RTE alone and are not exposed to the programmer. We will briefly outline these parts as well.

Heap (Memory) Management. Our approach inherently needs a virtual shared heap for transferred data, since implicit parallel evaluation is not within reach of the API, but managed by the RTE. However, not all data must be globalised, but only the parts which are required for remote evaluation and not addressed by explicit channel communication.

This distinction becomes manifest in the placeholder nodes in the graph heap. Nodes for globalised data, which the RTE fetches *actively* from other nodes when a thread needs it, are opposed to nodes for channel communication, which passively wait for data to arrive via the connected channel. For the latter, the RTE only needs a table for the respective channels, whereas management of the former needs more attention: besides the mapping from nodes to global addresses and vice-versa, the RTE must prevent duplication of data in one node's local heap. Furthermore, data fetching can be implemented with different strategies

(bulk or incremental fetching), which yields platform-dependant results, according to [11]. Fetching strategy and data size, as well as rescheduling and spark retrieval strategy, are good examples for configuration options of the RTE.

Communication and System Management. Communication inside the RTE is needed to synchronise the global heap and to exchange data between threads. We do not need a particular standard or product; any middleware capable of exchanging raw data between nodes in a network can be used (even raw sockets). However, targeting a standard such as MPI [12] makes the system far more portable, since implementations for various platforms are freely available. We intend to build a modular communication subsystem and exchangeable adapters to common message passing middleware. Necessary startup actions and node synchronisation are another reason why existing middleware is used instead of a customised solution.

4 Prospected Coordination Languages

The variety of possible coordination languages using the described API would fill a considerable amount of pages, so we set it aside and only give a small example. The Eden implementation described in [9] is a more complex instance.

Example: We define `rFork` as an explicit variant of the `par`-construct: It spawns a process for a `task` on the indicated `node` and continues with the `cont`. The RTE assigns a global address to `task` before sending, and it is evaluated by a new thread on `node`. If results are needed somewhere else, data will be fetched from there. This construct can be used in Haskell code to define higher-level constructs, e.g. a parallel fold-like operation shown here:[1]

```
rFork :: PE -> a -> b -> b
rFork node task cont = (rTask# node task) `seq` cont
foldSpread :: ( [a] -> b ) -> [a] -> [b]
foldSpread f xs = let size = ..--arbitrarily fixed, or by input length
                      tasks = [ f sublist | sublist <- splitList size xs ]
                      peList= drop (toInt myPE#) (cycle [ 1..noPE# ])
                  in zipWith rFork peList tasks
```

The effect of `foldSpread` is to distribute its input list (in portions of a certain size) over several nodes, which, in turn, evaluate a sub-result for each sublist. Sublists and results will definitely reside on the nodes we indicated: we can do further processing on these same nodes explicitly. ◁

The described ParallelAPI allows to express parallel coordination in easy, declarative terms and gives much expressive power to language designers. The most ambitious and interesting area to explore with it is to combine both *explicit* coordination and *implicit* parallelism, with freedom for load-balancing by the RTE. This can be applied when computations are big enough for wide-area distribution, but in a heterogeneous setup where dynamic load-balancing is required; found e.g. in scientific Grid-Computing with high communication latency and a rapidly changing environment.

[1] The symbol `#` indicates primitive operations and data types.

5 Conclusions

We have described a future implementation for parallel Haskells, which is based on a generalised runtime environment (RTE) and a layered implementation concept. The described API and RTE can express general-purpose parallelism declaratively and is based on experiences with two existing parallel Haskell systems. Since the approach only refers to runtime support, it can be freely combined with static analysis techniques for automatic parallelisation, and skeleton programming. Hence, the outcome of the prospected work is a standardised implementation for Haskell-based coordination concepts, for which this paper gives the guideline and points at similarities in existing parallel Haskells.

The concepts described in this paper are *work in progress*, i.e. details of the generalised RTE may change with further research. Our prospects for future work are to implement and use the described API for an integration of Eden and GpH in a common language, to express high-level coordination for large-scale parallel computations with the possibility of dynamic load-balancing and coordination control by the programmer. For this purpose, the RTE will be extended with more task placement policies, adaptive behaviour and runtime reconfiguration.

References

1. Hammond, K., Michaelson, G., eds.: Research Directions in Parallel Functional Programming. Springer (1999)
2. Trinder, P., Loidl, H.W., Pointon, R.: Parallel and Distributed Haskells. J. of Functional Programming **12** (2002)
3. Claus Reinke (ed.): Haskell Communities and Activities Report. Fifth Edition (2003) *www.haskell.org/communities*.
4. Trinder, P., Hammond, K., Mattson Jr., J., Partridge, A., Peyton Jones, S.: GUM: a Portable Parallel Implementation of Haskell. In: PLDI'96, ACM Press (1996)
5. Breitinger, S., Loogen, R., Ortega-Mallén, Y., Peña, R.: The Eden Coordination Model for Distributed Memory Systems. In: HIPS. LNCS 1123, IEEE Press (1997)
6. Peyton Jones, S., Hall, C., Hammond, K., Partain, W., Wadler, P.: The Glasgow Haskell Compiler: a Technical Overview. In: JFIT'93. (1993)
7. Trinder, P., Hammond, K., Loidl, H.W., Peyton Jones, S.: Algorithm + Strategy = Parallelism. J. of Functional Programming **8** (1998)
8. Loidl, H.W.: Granularity in Large-Scale Parallel Functional Programming. PhD thesis, Department of Computing Science, University of Glasgow (1998)
9. Berthold, J., Klusik, U., Loogen, R., Priebe, S., Weskamp, N.: High-level Process Control in Eden. In: EuroPar 2003 – Parallel Processing. LNCS 2790, Klagenfurt, Austria, Springer (2003)
10. Loogen, R., Ortega-Mallén, Y., Peña, R., Priebe, S., Rubio, F.: Parallelism Abstractions in Eden. In Rabhi, F.A., Gorlatch, S., eds.: Patterns and Skeletons for Parallel and Distr. Computing. LNCS 2011, Springer (2002)
11. Loidl, H.W., Hammond, K.: Making a Packet: Cost-Effective Comm. for a Parallel Graph Reducer. In: IFL'96. LNCS 1268, Springer (1996)
12. MPI Forum: MPI 2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville (1997)