Parallel and Distributed Haskells

P.W.TRINDER, H-W. LOIDL∗, R.F. POINTON†

Dept. of Computing and Electrical Engineering,
Heriot-Watt University, Edinburgh, EH14 4AS.
(*e-mail:* {Trinder,HWLoidl,RPointon}@cee.hw.ac.uk)

Abstract

Parallel and distributed languages specify computations on multiple processors and have a computation language to describe the algorithm, i.e. *what* to compute, and a coordination language to describe *how* to organise the computations across the processors. Haskell has been used as the computation language for a wide variety of parallel and distributed languages, and this paper is a comprehensive survey of implemented languages. We outline parallel and distributed language concepts and classify Haskell extensions using them. Similar example programs are used to illustrate and contrast the coordination languages, and the comparison is facilitated by the common computation language. A lazy language is not an obvious choice for parallel or distributed computation, and we address the question of why Haskell is a common functional computation language.

1 Introduction

Parallel languages utilise additional processors to reduce program runtime. Distributed languages use state-transforming threads to manipulate global state, i.e. the resources of several processors. A typical distributed application is a multiuser game or learning environment where users on multiple machines interact with each other in a common virtual world. The combination of hardware redundancy and stateful computation in a distributed language facilitates the construction of reliable, i.e. fault tolerant, systems.

The potential of functional languages for parallelism has been recognised for over thirty years, long before Haskell existed e.g. (Wegner, 1971). Functional languages offer good abstraction mechanisms, a sophisticated type system, high-level computation language and high-level coordination. However, the key advantage of a pure functional paradigm is that referential transparency guarantees the implementation considerable freedom of execution order without changing program semantics. This is evident to the parallelism community and many parallel languages use pure computation languages, some of which are subsets of impure languages, e.g. (Scholz, 1996; Michaelson *et al.*, 2001).

In comparison, the benefits of the functional paradigm for distribution have been

∗ Supported by APART fellowship 624 from the Austrian Academy of Sciences.
† Supported by research grant GR/M 55633 from UK's EPSRC.

realised only recently. Like their parallel counterparts, distributed functional languages benefit from good abstraction mechanisms, a high-level computation language, and sophisticated type system, but most of all because large and identifiable parts of the program are referentially transparent. Referential transparency grants freedom of execution order, e.g. facilitating lazy communication of data between processors and parallel execution of parts of the program; moreover the pure components are amenable to reasoning, e.g. optimisation or compilation by transformation (Peyton Jones *et al.*, 1993). Even with its limited referential transparency, Erlang has been used successfully to demonstrate that a functional paradigm significantly aids engineering large distributed systems. One such example is the 525K-line AXD301 ATM Switching System distributed over up to 32 processors (Blau & Rooth, 1998).

Most parallel and distributed languages have a computation language and a coordination (sub)language. The computation language is used to specify the algorithm, i.e. to define *what* value is to be computed, and may be a sequential language like C, SML or Haskell98. The coordination language describes *how* the computations are to be arranged on the virtual machine, including aspects such as thread creation, placement, and synchronisation. In the parallelism literature the term *coordination language* usually refers to a language distinct from the computation language, e.g. PCN coordinates Fortran or C computations (Foster *et al.*, 1992). In contrast, functional languages typically extend the computation language with a few high-level coordination constructs, and it is these that are rather loosely termed the coordination language in this paper. The constructs support some coordination paradigm, and a wide range of paradigms and constructs have been used. Examples include data-parallelism supported by Data Field Haskell (Holmerin & Lisper, 2000), or skeleton-based parallelism supported by parallel map, fold and other skeletons (Herrmann, 2000).

Consistent with their high-level computation language, most parallel and distributed functional languages support high-level coordination with automatic management of many coordination aspects. As with computation, the great advantage of high-level coordination is that it frees the programmer from specifying low-level coordination details. The disadvantages are that automatic coordination management complicates the operational semantics, makes the performance of programs opaque, is hard to implement, and is frequently less effective than hand-crafted coordination. Explicit coordination constructs encourage programmers to construct static, simple or regular coordination, whereas more implicit constructs encourage more dynamic and irregular coordination.

Low-level coordination may be managed solely by the compiler as in PMLS (Michaelson *et al.*, 2001), solely by the runtime system as in GpH (Trinder *et al.*, 1996), or by both as in Eden (Breitinger *et al.*, 1997). Whichever mechanism is chosen, the implementation of sophisticated automatic coordination management is arduous, and there have been many more parallel and distributed language designs than well-engineered implementations. Haskell is a standard lazy functional research language with a sophisticated type and class system (Peyton Jones *et al.*, 1999). It has a relatively mature development environment including compilers, interpreters, libraries and profiling tools. This paper surveys all *implemented* parallel and distributed languages with Haskell as computation language.

We start by addressing the question of why Haskell is a suitable computation language, and basis for a variety of coordination languages, in Section 2. We define parallel and distributed language concepts, and classify Haskell extensions using them in Section 3. Parallel Haskells are surveyed and related to other parallel functional languages in Section 4, likewise distributed Haskells in Section 5. The same simple parallel or distributed program is expressed in each language to illustrate and enable comparison of coordination constructs. Section 6 summarises the coordination constructs in the languages and concludes by discussing open problems.

2 Why Haskell?

It is perhaps surprising to find a *lazy* language like Haskell as a popular functional computation language, indeed Hains argues cogently that parallel functional languages should be strict (Hains, 1994). The problem has the following two aspects. Firstly lazy evaluation is sequential and performs minimum work, with reduction ceasing when the expression is in weak head normal form. In contrast parallel and distributed programs arrange computations on multiple processors and hence require some eager evaluation. Secondly, while in a strict language the computational behaviour of an expression is independent of the way the result is used — it depends only on the operand values. In a lazy language the amount and order of evaluation is often under the control of the consumer of the result. This confers extra expressive power — but makes it very hard to construct cost models, and means that the programmer must specify the *evaluation degree* of an expression: namely how much evaluation should be performed (Klusik *et al.*, 2000a; Trinder *et al.*, 1998).

Properties of Haskell that make it attractive as a computation language and a basis for a coordination language are as follows. The individual properties are not unique to Haskell: many are properties of other lazy functional languages, or pure subsets of strict functional languages.

Referential Transparency. A key advantage of a pure computation language is that it can be easily married to many different coordination languages because referential transparency guarantees that execution order is immaterial. The range of coordination languages is amply illustrated by the languages outlined in Sections 4 and 5. A pure computation language conveys a number of immediate practical benefits. Parallel semantics are relatively easily developed, e.g. the operational semantics for GpH and Eden (Baker-Finch *et al.*, 2000; Hidalgo Herrero & Ortega Mallén, 2000). The language is amenable to analyses, e.g. the non-determinism analysis in Eden (Pena & Segura, 2000). Pure languages are amenable to program derivation, compilation by transformation, and transformations for optimising coordination are easily introduced: the Eden compiler is a good example (Pareja *et al.*, 2000).

Laziness. A computation language with non-strict evaluation naturally supports highly-dynamic coordination where evaluation is performed and data is communicated on demand. Assuming that the execution cost of the coordination is small compared with the computation, the primary cost of non-strict coordination is additional communication. For example where an eager language simply sends data from producer to consumer, a lazy language requires an additional message from the consumer to request the data. The benefit gained by the additional communication in a lazy language is a natural throttling of both communication and computation. An example of communication throttling is a remote thread consuming a small part of a large data structure, where only that small part is communicated. Where both strict functional and dataflow languages often suffer from the eager creation of excess parallelism, a lazy language ameliorates these problems at the cost of specifying how much evaluation should be performed. Finally laziness facilitates the separation of concerns, e.g. evaluation strategies in GpH make essential use of laziness to separate computation and coordination (Trinder *et al.*, 1998).

Abstraction Mechanisms. High-level modular coordination facilities are produced using Haskell's data and control abstractions including classes, modules, higher-order functions, polymorphism and abstract data types. Since non-strict languages separate the definition of a value from its evaluation, the programmer has the additional flexibility to decide where to specify the coordination. For example it is possible to specify coordination when composing functions, by attaching a coordination construct to the value passed between functions, without breaking the function abstraction. In the same way that the demand on the result of a function controls the evaluation degree from outside, coordination constructs can control the parallelism from outside. More important for large systems, this abstraction scales to expressing coordination only at module interfaces (Loidl *et al.*, 1999).

Polymorphic Strong Typing. The benefits of typing in computation languages are well-established, but the benefits of a typed coordination language are less so. Strong typing ensures that coordination expressions are well-formed and reduces runtime errors, and typed coordination constructs include process types in Eden (Breitinger *et al.*, 1997), and placement directives in Caliban (Taylor, 1997). Polymorphic types enable the construction of generic coordination constructs. Examples include skeletons in Eden (Klusik *et al.*, 2000b) and polymorphic data fields in Data Field Haskell (Holmerin & Lisper, 2000). Open distributed languages require *dynamic* typing to enforce type correct interfaces to new programs, e.g. to a new client or applet. Unusually, some Haskell-based languages are closed, e.g. Brisk and GdH, and hence can be *statically* typed.

Implementation Benefits. Due to the coroutine-like evaluation in lazy languages, their implementations already have many of the mechanisms required by parallel and distributed languages. For example Haskell implementations have mechanisms for encapsulating suspended computations for subsequent evaluation, and it is convenient to transfer work from processor to processor as a suspension. Similarly, many lazy language implementations are based on graph reduction, and the graph is a convenient and uniform structure for communicating both code and data.

Pragmatic Factors. There are many practical reasons for selecting Haskell as a computation language: the language is standardised and compilers are well-developed, with good sequential optimisation and support important practical features like useful libraries and a foreign language interface. The implementations are both open source and modular, and hence relatively easily adapted. Moreover there are tools like profilers available, and there is an active and supportive community.

Properties of Haskell that make it unattractive as a computation language and a basis for a coordination language are as follows.

Lazy Evaluation. As outlined above, lazy evaluation must frequently be overcome to obtain sufficient parallelism or distribution. Moreover, it is much harder to develop time and space cost models for non-strict languages (Sands, 1990; Loidl, 1998).

Limited Module and Class Systems. More sophisticated systems than the Haskell98 module and class system would facilitate the encapsulation and derivation of coordination constructs. In GpH for example, it would be beneficial to be able to derive basic evaluation strategies for new abstract data types, e.g. an `rnf` strategy that reduces values of the new type to root normal form (Trinder *et al.*, 1998).

Cumbersome State Manipulation. Distributed programs necessarily manipulate state on multiple processors. However, describing stateful computations in Haskell's monadic constructs is relatively verbose and hard to reason about.

Broadly speaking the properties that make Haskell a suitable computation language are braodly similar to the properties that make it a good sequential language: namely its referential transparency, sophisticated type system and good abstraction mechanisms as well as a number of pragmatic factors. These attractions are sufficient to overcome the additional coordination required to subvert the default lazy evaluation.

3 Coordination Language Concepts

Computer hardware may be arranged in a large variety of ways, ranging from single processors, shared-memory and distributed-memory multiprocessors to networks of machines. Parallel and distributed languages reflect some of the underlying architecture, while other languages abstract over it. In this section we define a number of concepts to facilitate parallel and distributed language classification. Because of the large number of concepts that distributed languages may or may not support it is very hard to construct a simple yet accurate classification, although a number have been given, e.g. (Skillicorn & Talia, 1998). Our definitions and classification are neither new nor unusual, but are suitable for defining and classifying the coordination in parallel and distributed functional languages. The classification is intended for small-scale systems composed of programs written in the same language. In contrast, large-scale distributed systems are supported by standard interfaces like CORBA (Siegel, 1997) or Microsoft DCOM (Merrick, 1996) and may have components written in multiple languages, supplied by several vendors, be executed on a heterogeneous collection of platforms, and have elaborate fault tolerance.

Processing Element (PE). A physical device that performs computation, typically a processor with memory and associated physical resources such as disk, screen, etc.

Thread. An independent sequence of executing instructions. Sometimes also known as a lightweight process to indicate that a thread has minimal private resources. Threads may be *explicit* with constructs for creation and termination; *semi-explicit* being managed by directives or annotations; or entirely *implicit* e.g. being managed by a data-parallel or skeleton compiler. A (semi-)explicit approach is typically taken by distributed languages such as Facile Antigua (Thomsen *et al.*, 1993) and GdH (Pointon *et al.*, 2000), whereas parallel languages tend to favour a more implicit approach, e.g. HDC (Herrmann, 2000) and High Performance Fortran (HPF, 1993). An important distinction is between pure threads that only return a value, and state-transforming threads that perform operations on external state.

Thread Interaction. The term used to describe both communication and synchronisation between threads. Communication is the exchange of data and synchronisation is the coordination of control. The two concepts are closely related and typically intertwined together, e.g. communication requires synchronisation to safely pass data to another thread, and some form of communication is necessary to indicate that synchronisation has occured. In languages with *implicit interactions* threads typically interact using shared data, freeing the programmer from specifying the interactions. For example GpH threads interact via shared variables, and Java threads interact via shared objects using synchronised methods (Daconta *et al.*, 1998). In languages with *explicit interactions* threads in the same location typically interact using shared location resources, e.g. a semaphore. If the threads are in different locations then interactions occur through some global resource, e.g. they may address a channel or the mailbox of a thread.

Location. A named bounded space containing resources, like memory and I/O capabilities, and usually threads. A location may reside on a PE or a group of PEs. A location is an abstraction of the familiar process concept, but is more general because a location's threads may be executing different programs, or it may contain no threads. A language is *location independent* if locations are implicit, e.g. enabling a file to be accessed regardless of its location. A language is *location aware* if locations are explicit, enabling the programmer to utilise the resources of a location, e.g. forking a new thread onto a PE. Examples of abstractions for location include Facile Antigua (Thomsen *et al.*, 1993) which provides `nodeid` to identify a particular PE and GdH (Pointon *et al.*, 2000) with `PEId` to name a location.

Open/Closed Systems. There is no reason why communicating threads must belong to the same program, and often large systems consist of many co-operating programs. In a *closed system* there is a static set of programs being executed and all modes of inter-thread interaction are known. Hence the interactions can be statically checked, e.g. for type safety, deadlock etc. An *open system* comprises multiple executing programs interacting using a predefined protocol, for example in a client-server model. This requires some language support to initialise communication to connect to other programs. Such languages support a dynamic model that is open in that it can be extended to include new programs. However, the interactions between such a dynamic set of programs cannot be statically checked.

Fault Tolerance. The ability of a program to detect, recover and continue after encountering faults. Faults may either be internal to the process, e.g. divide by zero, or external, e.g. disk failure, user interrupt.

3.1 Language Classification

Languages can be classified by the coordination concepts they support as follows. *Sequential languages* support a single thread and are very common, examples include Haskell98 (Peyton Jones *et al.*, 1999) and SML (Milner *et al.*, 1997). *Concurrent languages* support explicit interactions between multiple threads, and examples include Concurrent Haskell (Peyton Jones *et al.*, 1996) and CML (Reppy, 1992). *Parallel languages* support multiple PEs hosting multiple threads usually with implicit interactions and location independence. They aim to reduce program execution time. Parallel extensions of Haskell include Eden (Breitinger *et al.*, 1997), Nepal (Chakravarty *et al.*, 2001), and many others covered in Section 4. *Distributed languages* support multiple PEs hosting multiple threads with explicit interactions and location awareness. Distributed languages are also more likely to support open systems and more sophisticated fault tolerance. Distributed Haskells include Haskell with Ports (Huch & Norbisrath, 2000), GdH (Pointon *et al.*, 2000), and the others covered in Section 5.

The remainder of the paper focusses on parallel and distributed functional languages, concurrent languages are omitted because most execute either at a single location or on low-latency shared-memory architectures where location is relatively unimportant. Figure 1 classifies parallel and distributed Haskells, together with a few well-known languages, using thread interaction, location independence/awareness and open/closed properties.

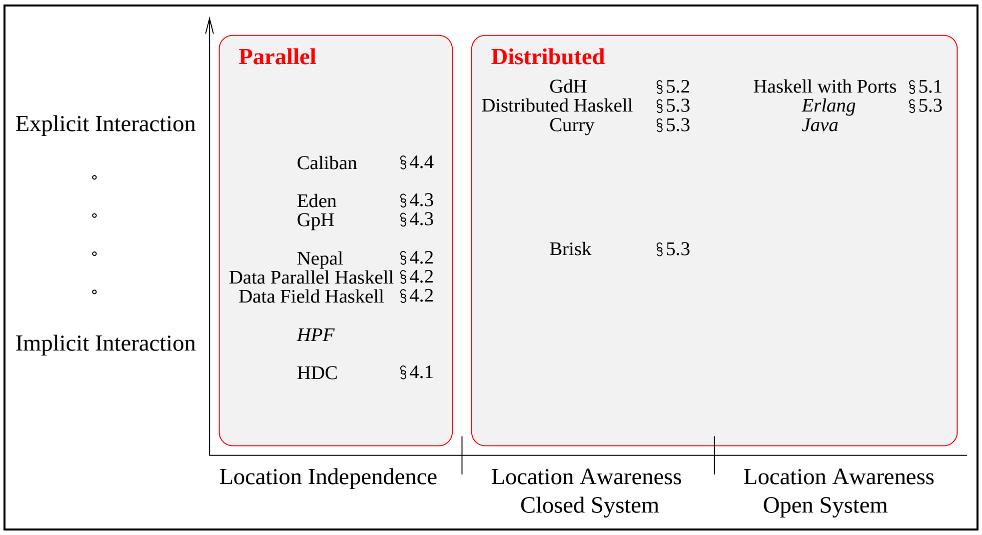

Fig. 1. Parallel and Distributed Haskell Classification.

4 Parallel Haskells

The goal of parallel programming is to achieve higher performance, thereby reducing runtime or increasing the tractable problem size. This section focusses on parallel coordination language constructs and paradigms: it is not possible to give meaningful performance comparisons of all the languages surveyed because the languages are implemented on a wide variety of parallel architectures, and few are available on multiple platforms. We do, however, give directly comparable measurements for Eden and GpH in Section 4.3.3. For the other languages we provide qualitative performance measures and reference more detailed discussion. Substantial performance comparisons of several programs implemented in Eden, GpH and PMLS, a strict parallel functional language, are reported in (Loidl *et al.*, 2001a).

Adding coordination to a computation language does not change its expressive power. Because performance is intensional, i.e. not exposed in a standard semantics of the language, many parallel Haskells make coordination substantially implicit. Typically parallel languages are closed, provide little or no fault tolerance, and limited location awareness. In a parallel language location is only indirectly important because it may enable performance improvements to the program, e.g. improved data locality.

Parallel Haskells are illustrated and compared using the `sumEuler` program shown in Figure 2. The program computes the sum of a list of Euler totient values produced by the `euler` function, i.e. the number of integers that are relatively prime to a given integer. It is also an instance of a common computational structure, namely a fold-of-map. More interestingly, `sumEuler` exposes several coordination issues. Firstly, it is inherently data parallel because of the independence of the `euler` computations. Secondly, good performance can only be obtained by selecting a good thread granularity. This is because individual calls to `euler` are very cheap and hence several must be combined into a single parallel thread to offset thread management costs. This is achieved by the `splitAtN` function, shown in Figure 3, that partitions the input list into "chunks". Thirdly, the sum (fold) must be effectively parallelised and this is done by computing the sum of the chunks of totient values, before computing the overall sum.

The remainder of this section is structured by coordination paradigm. We loosely follow the classification given in (Loogen, 1999), which also contains a more detailed discussion of non-Haskell parallel functional languages. We relate the parallel Haskells discussed in Sections 4.1, 4.2, 4.3, and 4.4 with other parallel functional languages in Section 4.5, and summarise by discussing parallel language pragmatics in Section 4.6.

4.1 A Skeleton-based Haskell

Skeletons are a popular parallel coordination construct. Typically, a language has a small set of predefined skeletons, where each skeleton is a higher-order function describing a common coordination pattern with an efficient parallel implementation (Cole, 1999). Rather than managing an unstructured set of parallel threads, the programmer need only use the higher-order functions appropriately to introduce parallelism. Often these higher-order functions work over compound data structures and consequently the resulting parallel code often resembles data parallel code as discussed in Section 4.2.

```
-- Top level function:
sumEuler    :: Int -> Int
sumEuler n = sum [ euler i | i <- [n,n-1..0] ]

-- Euler totient function:
euler    :: Int -> Int
euler n = length (filter (relprime n) [1..(n-1)])

-- Auxiliary functions:
-- Determine whether x and y are relatively prime
relprime      :: Int -> Int -> Bool
relprime x y = hcf x y == 1

-- Find the highest common factor of x and y
hcf      :: Int -> Int -> Int
hcf x 0 = x
hcf x y = hcf y (rem x y)
```

Fig. 2. Sequential Haskell98 version of `sumEuler`

```
splitAtN       :: Int -> [a] -> [[a]]
splitAtN n [] = []
splitAtN n xs = ys : splitAtN n zs
                where (ys,zs) = splitAt n xs
```

Fig. 3. A Clustering Function

4.1.1 HDC

HDC (Herrmann, 2000) is a strictly-evaluated subset of Haskell with skeleton-based coordination. HDC programs are compiled using a set of skeletons for common higher-order functions, like fold and map, and several forms of divide-and-conquer. The current implementation supports two divide-and-conquer skeletons and a parallel map, and the system relies on the use of these higher-order functions to generate parallel code. Unlike Haskell, HDC does not implement type classes, and has strict semantics to facilitate static thread placement. Language-level lists are implemented as arrays internally. List comprehensions are compiled to `map` and `filter`, where map operates in parallel over these arrays. In summary, HDC has purely implicit threads with implicit interaction. It is location independent, since parallelism is not explicit in the program at all.

In HDC it is possible to achieve parallel execution of the code in Figure 2 without any code changes. In tuning the performance of the parallel program, however, it is often necessary to modify the code, so as to weaken data dependencies or to increase granularity. In this case we can increase the granularity, i.e. the computation costs, of the individual threads by combining several `euler` computations into a single thread and computing the `sum` inside each thread. The additional argument `c` specifies the size of these chunks of input data, and `splitAtN` is used to generate the chunks. The function `seqmap` produces the same result as `map` but enforces sequential evaluation of the `euler` functions.

```
sumEuler     :: Int → Int → Int
sumEuler c n = sum [ (sum . seqmap euler) x | x ← splitAtN c [n,n-1..0] ]
```

Fig. 4. HDC Version of `sumEuler`

A particular focus of the HDC system is the time and space efficient static thread placement. The compiler uses a library of skeletons to decompose a program into parallel threads and place the threads on the available PEs. In contrast languages such as GpH and Eden, use more flexible, but also more expensive, dynamic resource management.

Reflecting the fact that the HDC compiler is still a prototype, only a set of fairly small example programs has been compiled to efficient code: a Karatsuba algorithm for polynomial multiplication, n-queens, maximum independent sets and convex hull computations. On a 1024-processor Transputer-based Parsytec GCel-1024 machine scalable three-digit speedups are reported for these programs in (Herrmann, 2000).

4.2 Data Parallel Haskells

Data parallel languages (O'Donnell, 1999) focus on the efficient implementation of the parallel evaluation of every element in a collection. The focus on defining parallelism over large data structures makes this approach very appealing for the parallelisation of data-intensive applications. Haskell's powerful constructs for bulk data types, in particular lists, provide a very useful basis for defining data-parallel constructs. Indeed, all of the languages discussed here use some parallel extension of list comprehensions and implicitly parallel higher-order functions such as `map`. Compared to other approaches to parallelism, the data parallel approach makes it easier to develop good cost models, although, it is notoriously difficult to develop cost models for languages with a non-strict semantics. Typically data parallel languages use a closed system model and implicit parallelism. Location awareness is not required at the program level because it is implicit in the data parallel execution.

4.2.1 Data Field Haskell

Data Field Haskell (Holmerin & Lisper, 2000) extends Haskell with the new notion of data fields: generalisations of arrays, with parallel bulk data (collection-oriented) operations defined over them, as shown in Figure 5. In general, a data field defines a partial function from index domain to value domain. Data fields may specify various multidimensional shapes, sparse or dense contents, and finite or infinite

```
class Pord a ...
type (Pord a, Ix a) ⇒ Datafield a b = ...
type (Pord a, Ix a) ⇒ Bounds a = ...

– operations over datafields: construction and selection
datafield  :: (Pord a,Ix a) ⇒ (a → b) → Bounds a → Datafield a b
(!)        :: Datafield a b → a → b
– operations over bounds of a datafield
bounds     :: (Pord a,Ix a) :: Datafield a b → Bounds a
(<:>)      :: (Ix a, Pord a) ⇒ a → a → Bounds a

– hyperstrict evaluation
hstrictTab :: (Pord a, Ix a, Eval a) ⇒ Datafield a b → Datafield a b

– forall abstraction (language construct)
forall apat1 ... apatn → exp
```

Fig. 5. Basic Coordination Constructs in Data Field Haskell

size. A rich set of functions for manipulating bounds are defined, e.g. `1<:>n` defines a dense index domain of all integers between 1 and n. The computation over a datafield is defined either as a `forall`-abstraction, i.e. a function applied to the index domain, or via a set of predefined higher-order functions over datafields, e.g. a fold-like `foldlDf`. To express the extent to which an expression should be evaluated Data Field Haskell introduces functions for strict and hyperstrict evaluation of Haskell expressions, e.g. `hstrictTab`. Data Field Haskell has been implemented as an extension for Haskell 1.3 on top of the nhc13 compiler (Röjemo, 1995). However, no parallel implementation is available, yet.

```
sumEuler     :: Int → Int → Int
sumEuler c n sumDf ( forall i →
                sumDf ( forall j → euler (xs!i)!j))
             where xs = mkField c n

mkField     :: Int → Int → Datafield Int (Datafield Int Int)
mkField c n = datafield ( λ i →
                datafield ( λ j → min c*i+j n ) (0<:>c-1)
                (0<:>n+c-1 ‘div‘ c - 1)

sumDf :: (Pord a, Ix a, Num b) ⇒ Datafield a b → b
sumDf = foldlDf (+) 0
```

Fig. 6. Data Field Haskell Version of `sumEuler`

The example code in Figure 6 demonstrates how to implement `sumEuler` in Data

Field Haskell. In `mkField` an explicit datafield constructor is used to build a nested datafield. The outer field ranges over the chunks that are mapped onto processors. The inner field ranges over the values passed to the `euler` function on one processor. The current, sequential, implementation does not distinguish between the two `forall` constructs, but in the parallel implementation it is planned to enable parallel execution by choosing an appropriate, parallel, index domain. To avoid high overhead when converting lists into datafields, most operations are performed on the datafields, even if there is little parallelism in the execution of the code.

So far only a small set of sequential programs has been implemented in Data Field Haskell. The largest applications include a particle simulation, a neural network relaxation model, and an LU-factorisation algorithm.

4.2.2 Nepal

The extension of Haskell developed in the Nepal project (Chakravarty *et al.*, 2001), here called *Nepal* for short, adds *parallel arrays* to Haskell. It provides special syntax such as array comprehensions and parallel implementations of basic functions over these arrays. Similar in spirit to the NESL language (see Section 4.5) data parallelism can be nested, achieving a high degree of flexibility. A special flattening transformation is used to transform nested into flat data parallelism (Chakravarty & Keller, 2000).

Using the new language constructs for arrays the implementation of `sumEuler` in Nepal is straightforward and shown in Figure 7. All standard operations on lists, such as `length`, `filter`, etc, have corresponding versions over parallel arrays. The `euler` function is modified to use arrays as well, to make better use of the flattening transformation thereby achieving a better data distribution. Array comprehensions of the form `[: ... :]` are analogous to Haskell's list comprehensions and are translated into calls to the functions `mapP` and `filterP`, which are in turn implemented as calls to parallel code. Nepal's flattening transformation in combination with a type system that distinguishes local from global values enables the compiler to automatically transform from the code in Figure 7 into a clustered version with better granularity (Keller & Chakravarty, 1999). On the positive side, this approach of implicit parallelism is convenient for the programmer and leads to concise programs. However, the downside is that clustering is outside the programmer's control, which implies that it cannot be easily modified nor adapted dynamically.

```
sumEuler   :: Int → Int
sumEuler n = sumP [: euler i | i ← [:n, n-1 .. 0:] :]

euler   :: Int → Int
euler n = lengthP (filterP (relprime n) [:1 .. n-1:])
```

Fig. 7. Nepal Version of `sumEuler`

Nepal is currently being implemented as an extension of GHC with the following main steps. The flattening transformation maps nested array computations to flat array computations. Then the data parallel primitives are unfolded by decomposing them into local components, with optimisations such as array loop fusion to improve granularity, and global components, introducing communication. For parallel execution library routines of a structured communication library are used to provide a high level of portability. Performance measurements of the current sequential implementation show a high efficiency of the array code, significantly outperforming both standard Haskell arrays and list-based implementations of test programs such as a Barnes-Hut algorithm. Parallel performance measurements of a hand-translated Barnes-Hut algorithm achieved promising speedups on up to 24 processors on a Cray T3E multi-processor (Chakravarty & Keller, 2000).

4.2.3 Data Parallel Haskell

An older system that used such a data parallel approach was *Data Parallel Haskell* (Hill, 1994). The central idea of Data Parallel Haskell was to replace the single "aim" of sequential computation, namely computing the result value, by a series of aims of evaluation and to force evaluation on all of them. Parallel performance could be improved by manipulating the aim, which becomes a central component of Data Parallel Haskell's operational semantics, but remains hidden from the programmer. The goal of this design was to achieve data-parallel execution without forcing strict evaluation and thereby sacrificing the advantages of a language with non-strict semantics.

As new language constructs Data Parallel Haskell defined special arrays called PODs (parallel data structures), represented as one-dimensional sparse and potentially infinite index/value pairs. POD comprehensions were used to define data parallelism. These comprehensions were compiled to parallel implementations of the well-known map, fold and scan functions. The implementation used program transformation to vectorise a functional program. Data Parallel Haskell has been implemented as a parallel extension of Haskell 1.2 on the GHC 0.16 compiler, but there is no current development.

4.3 Semi-Explicit Parallel Haskells

Semi-explicit parallel languages provide a few high-level constructs for controlling key coordination aspects, while automatically managing most coordination aspects statically or dynamically. Historically, annotations were commonly used for semi-explicit coordination, but more recent languages provide compositional language constructs. As a result, the distinction between semi-explicit coordination and coordination languages is now rather blurred, but the key difference in the approach is that semi-explicit languages aim for *minimal* explicit coordination.

4.3.1 GpH

GpH (Trinder *et al.*, 1998) is a modest extension of Haskell with parallel (`par`) and sequential (`seq`) composition as coordination primitives (see Figure 8). Denotationally, both constructs are projections onto the second argument. Operationally `seq` causes the first argument to be evaluated before the second and `par` indicates that the first argument may be executed in parallel. The latter operation is called the "sparking" of parallelism and is used in different variants in many parallel languages. The runtime-system, however, is free to ignore any available parallelism. In this model the programmer only has to expose expressions in the program that can usefully be evaluated in parallel. The runtime-system manages the details of the parallel execution such as thread creation, communication etc. In summary, GpH has a closed system model with semi-explicit parallelism and implicit communication, based on a virtual shared heap, and location independence. GpH is publicly available from (GPH, 2001).

```
par :: a → b → b                  – parallel composition
seq :: a → b → b                  – sequential composition

type Strategy a = a → ()          – type of evaluation strategy
using :: a → Strategy a → a       – strategy application

rwhnf :: Strategy a               – reduction to weak head normal form
class NFData a where              – class of reducible types
      rnf :: Strategy a           – reduction to normal form
```

Fig. 8. Basic Coordination Constructs in GpH

Experience of implementing non-trivial programs in GpH shows that the unstructured use of `par` and `seq` operators can lead to rather obscure programs. This problem can be overcome with *evaluation strategies*: lazy, polymorphic, higher-order functions controlling the evaluation degree and the parallelism of a Haskell expression. They provide a clean separation between coordination and computation. The driving philosophy behind evaluation strategies is that it should be possible to understand the computation specified by a function without considering its coordination. Figure 8 shows the basic operations over strategies. The `using` construct applies a strategy to a Haskell expression. The basic strategy `rwhnf` reduces an expression to weak head normal form (WHNF), the default in Haskell. The overloaded strategy `rnf` reduces an expression to normal form (NF), and is instantiated for all major types.

In GpH it is possible to specify block-wise evaluation over the input list with chunk size `c` applying the `parListChunk c rnf` strategy to the list comprehension. However, without changing the computational code it is not possible to compute the sum of each chunk. A version that does so is given in Figure 9. Again the

```
sumEuler       :: Int → Int → Int
sumEuler c n   = sum ([ (sum . map euler) x | x ← splitAtN c [n,n-1..0] ]
                      `using` parList rnf)
```

Fig. 9. GpH Version of `sumEuler`

`splitAtN` function is used to split the list into chunks of size `c` for granularity control. The strategy `parList` defines data parallelism over these segments. Each thread generated by this strategy computes the function `sum . map euler`. This clustering technique can be generalised to arbitrary data structures as discussed in (Loidl *et al.*, 2001b). In summary, the programmer has the choice working purely on strategy level, leaving the computational code of the program unchanged, or to perform some simple transformations of the computational code to further tune parallel performance.

GpH has been used to engineer several large programs, four of which are discussed in (Loidl *et al.*, 1999). The largest program is Lolita, a natural language processor comprising tens of thousands of lines of code, that has been parallelised for a shared memory machine. Naira is a parallelising compiler for a subset of Haskell, based on the dataflow model of computation. Blackspots is a data-intensive real-world application to find blackspots in a database of traffic accident records. LinSolv is an exact linear system solver. Performance results for all programs on workstation networks and a Sun SMP machine are reported in (Loidl *et al.*, 1999), and performance comparisons with Eden are reported in Section 4.3.3.

4.3.2 Eden

Eden (Breitinger *et al.*, 1997) coordinates parallel computations using explicit process creation and interconnection, enabling the programmer to define arbitrary process networks. Thread interaction can be either implicit, via shared variables and function parameters on process creation time, or explicit via communicating parameters to processes during process life time. The language uses a closed system model with location independence. A prototype of the Eden system is available on request.

Figure 10 summarises the basic coordination constructs available in Eden. Process abstractions with type *Process a b* define the behaviour of processes with input of type a and output of type b analogous to functions of type $a \to b$ defined by λ-abstractions. A process abstraction specifies the mapping of data input $\mathtt{in}_1 \ldots \mathtt{in}_m$ via inports to data output $\mathtt{out}_1 \ldots \mathtt{out}_m$ via outports. Inports and outports connect (unidirectional) communication channels to processes. Communication channels are not autonomous objects, but tightly coupled with processes. Processes and their interconnecting channels are created by the evaluation of *process instantiations* of the form `p # x` which applies the process abstraction `p` to the expression `x`, representing the input tuple. The result of a process instantiation is the tuple of outgoing data of the newly created process. Eden processes use independent threads to pro-

```
newtype Process a b = ...
– process abstraction (language construct)
process (in1,...,inm) → (out1,...,outn) :: Process (a1,...,am) (b1,...,bn)

– process instantiation
(#) ::  (Transmissible a, Transmissible b) ⇒ Process a b → a → b

– non-deterministic merge process
merge :: Process [[a]] [a]
```

Fig. 10. Basic Coordination Constructs in Eden

duce their outputs. For each output a separate thread is created which evaluates the output expression to normal form and sends the result value via the corresponding outport. Lists are transmitted as streams, i.e. element-wise. A predefined non-deterministic process `merge` is provided for many-to-one communication, which is useful for specifying reactive systems. It takes a list of input streams and merges the values in the order in which they arrive.

In Eden the programmer typically starts with a specific process network in mind and models this network using explicit processes. Evaluation strategies may also be required. This may amount to a higher effort in implementing a parallel algorithm, compared to GpH or HDC, especially when it is not possible to use one of a set of predefined Eden skeletons for parallel execution (Klusik *et al.*, 2000b). It offers, however, more possibilities for tuning the parallel performance.

```
sumEuler     :: Int → Int → Int
sumEuler c n = sum ([ (process z → (sum . map euler) z) # x
                    | x ← splitAtN c [n,n-1..0] ]
                    `using` seqList r0)
```

Fig. 11. Eden Version of `sumEuler`

Figure 11 shows an Eden version of the `sumEuler` program. The list comprehension defines parallelism over the chunks of input data by applying a process abstraction to all chunks generated by `splitAtN`. The body of the process abstraction specifies the sequential computation performed by each thread. The strategy `seqList r0` starts off the evaluation of the parallel threads by enforcing a spine strict evaluation of the list.

The largest programs implemented in Eden are a ray tracer of several hundred lines of code, a linear systems solver and a checkers program. Detailed measurements of these programs can be found in (Klusik *et al.*, 2001).

4.3.3 Eden/GpH Performance Comparisons

Eden and GpH are available on the same platform and hence we are able to summarise the following direct performance comparisons. The measurements have been performed on a 32-node Beowulf cluster (Ridge *et al.*, 1997) consisting of Linux RedHat 6.2 workstations with a 533MHz Celeron processor, 128kB cache, 128MB of DRAM, 5.7GB of IDE disk, connected through a 100Mb/s fast Ethernet switch with a latency of 142μs, measured under PVM 3.4.2. For the `sumEuler` program with a list length of 8000 and a cluster size of 100, the relative speedups on 16 processors are 13.1 for GpH and 12.4 for Eden.

Other programs that have been compared include a raytracer based on an Id program in the Impala benchmark suite (Impala, 2001). For this simple data parallel program a static mapping of threads to processors proves to be most efficient, with GpH's dynamic resource management generating additional overhead. Overall, for an input of 640 spheres and a 350×350 grid, and using clusters of 10 lines, Eden achieves a relative speedup of 13.3 on 16 processors, compared to 5.2 for GpH. An exact linear system solver, originally developed in GpH and ported to Eden, achieved relative speedups of 6.9 (GpH) and 13.2 (Eden) for a sparse 14×14 matrix with arbitrary precision integers as input. A detailed discussion of these results is presented in (Loidl *et al.*, 2001a) and the program sources are available online.

4.4 Haskell with a Coordination Language

Parallel coordination languages (Kelly & Taylor, 1999) are separate from the computation language and thereby provide a clean distinction between coordination and computation. Historically, Linda (Carriero & Gelernter, 1989) and PCN (Foster *et al.*, 1992) have been the most influential coordination languages, and often a coordination language can be combined with many different computation languages, typically Fortran or C. Other systems such as SCL (Darlington *et al.*, 1996) and P3L (Bacci *et al.*, 1995) focus on a skeleton approach for introducing parallelism and employ sophisticated compilation technology to achieve good resource management.

4.4.1 Caliban

The latest implementation of the *Caliban* coordination language uses Haskell$^-$ as computation language (Kelly, 1989; Taylor, 1997). Haskell$^-$ is a subset of Haskell, mainly omitting modules and type classes. Caliban has constructs for explicit partitioning of the computation into threads, and for assigning threads to (abstract) processors in a static process network. Communication between processors works on streams, i.e. eagerly evaluated lists, similar to Eden. In summary, Caliban uses a closed system model with coordination via semi-explicit threads, communication via implicitly defined data dependencies, and location independence.

Figure 12 summarises the basic coordination constructs in Caliban. Each entry represents a component in the data structure *Placement* controlling the evaluation of a Haskell$^-$ expression. Since Caliban's coordination constructs are integrated

`NoPlace`	– *null assertion*
`Bundle [x, y]`	– *place* x *and* y *on the same processor*
`Annot x`	– *extract location of* x
`Arc a b`	– *document a data dependency between* `a` *and* `b`
`a And b`	– *execute subnets* `a` *and* `b` *on different processors*
`a With b`	– *execute subnets* `a` *with* `b` *on the same group of processors*

Fig. 12. Basic Coordination Constructs in Caliban

into the host language, functions producing placement structures, so called network forming operators (NFOs), can be defined exploiting the full power of the host language. These NFOs are evaluated at compile-time using partial evaluation techniques. The variables `x` and `y` are Haskell$^-$ variables of type *Stream* representing computations, whereas `a` and `b` represent process networks of type *Placement*. The `Bundle` assertion produces a process network of co-located computations with threads being generated for each argument. The `Annot` directive extracts placement information from a computation. `Arc` is an assertion of a data dependency between two process networks, which is checked by the compiler. Two composition directives for process networks are available. The `And` directive indicates that the networks execute in parallel, whereas the `With` directive indicates that two networks should be executed on the same group of processors.

```
sumEuler       :: Int → Int → Int
sumEuler c n = res moreover fan res ress
               where res = sum ress
                     ress = map (sum . map euler) chunks
                     chunks = splitAtN c [n,n-1..0]

fan            :: Stream → [Stream] → Placement
fan s []       = NoPlace
fan s (x:xs) = (Bundle [x]) And (Arc x s) And (fan s xs)
```

Fig. 13. Caliban Version of `sumEuler`

Figure 13 shows the implementation of `sumEuler` in Caliban. In the body of `sumEuler` the coordination expression `fan res ress` is applied to `res` by using the `moreover` clause, similar to GpH's `using`. The definition of `fan` itself specifies the parallel execution of every list element in its second argument by using `And` for composition. It corresponds to GpH's `parList`. Overall, this code is similar to the code used in semi-explicit languages such as GpH. However, since Caliban describes static process networks it may employ more efficient, though less flexible, resource management.

A prototype implementation of Caliban with Haskell$^-$ as host language is available (Taylor, 1997). The largest applications implemented in Caliban are a Jacobi relaxation algorithm and a ray tracer, introduced in (Kelly, 1989). Although the overall structure of this ray tracer is similar to the one used in the comparison of GpH with Eden, it should be noted that they are based on different sequential versions and that the input size as well as parallel architecture differ. For an input modelling a scene with 20 cubes and a grid size of 100×100 rays, and using blocks of 40 rays for granularity control in a task farm architecture, relative speedups of up to 24 were achieved on 35 processors of a 128 processor Fujitsu AP1000 based on 25MHz Sparc processors (Taylor, 1997).

4.5 Other Parallel Functional Languages

Other Parallel Haskells. Para-functional programming (Hudak, 1986) is the general approach of adding control directives to a functional program to specify parallel execution. These control directives allow the programmer to describe detailed execution schedules as well as the mapping of threads to processors. A Haskell-based implementation of para-functional programming on an SGI Challenge shared-memory machine is described in (Mirani & Hudak, 1995). This implementation fully integrates the directives into Haskell by defining first-class schedules with a monadic type. These schedules are used in a similar way to evaluation strategies in GpH and `moreover` clauses in Caliban.

Haskell-Linda (Peterson *et al.*, 2000) is an extension of Haskell providing a binding to basic operations defined in the Linda model (Carriero & Gelernter, 1989) for describing parallel execution. It is an open system model with explicit parallelism and implicit synchronisation. In the Linda model communication between parallel threads is based on operations on a shared tuple space. The basic operations on this tuple space, which is split into several regions, are read, write, and in (for read and remove). Parallel threads, represented as process tuples in the tuple space, communicate by reading and writing tuples from/to the tuple space. In reading from the tuple space a pattern can be specified. If several tuples match the pattern the result is non-deterministic. Haskell-Linda is currently used to specify parallel functional reactive programs (Parallel-FRP) such as a web-based online auctioning system.

Finally, several bindings of explicit message passing libraries, such as PVM (PVM, 1993) and MPI (MPI, 1997), for Haskell have been developed (Breitinger *et al.*, 1998; Weber, 2000; Winstanley & O'Donnell, 1997). These languages use an open system model of explicit parallelism with explicit thread interaction. Since the coordination language is basically a stateful (imperative) language, monadic code is used on the coordination level. Although the high availability and portability of these systems are appealing, the language models suffer from the rigid separation between the stateful and purely functional levels.

Other Non-strict Languages. The late 80s saw an increasing interest in the parallel implementation of non-strict functional languages, which is reflected in the implementation of several such systems. The $<\nu, G>$-machine (Augustsson & Johnsson, 1989) used LML with annotations for sparking and was implemented on a Sequent Symmetry. The extension of Haskell with sparking annotations used on the parallel GRIP machine (Peyton Jones *et al.*, 1987) was a direct precursor of the GpH language covered in Section 4.3.1. The LML-like, lazy, implicitly-parallel functional language ALFL has been implemented on a distributed-memory Intel Hypercube as well as on a shared-memory Encore machine (Goldberg, 1988), with near-linear speedups for small programs such as nqueens on the latter architecture.

The HDG machine (Kingdon *et al.*, 1991) implemented a Miranda-like, implicitly-parallel, lazy language on a Transputer network, by using the evaluation transformer model (Burn, 1991) to extract parallelism. The PAM machine (Loogen *et al.*, 1989) implemented a simple non-strict, higher-order language with an explicit parallel let construct, in addition to the evaluation transformer model, on a Transputer network.

Concurrent Clean (Plasmeijr *et al.*, 1999; Nöcker *et al.*, 1991) is a language with close similarity to Haskell, in particular due to its non-strict semantics. Coordination is specified using annotations, i.e. compiler directives in comments, similar to, but more sophisticated than the directives in GpH. Concurrent Clean has been implemented on the Transputer-based ZAPP machine (Goldsmith *et al.*, 1993), which focusses on divide-and-conquer parallelism. Another implementation of Concurrent Clean on a Transputer network achieved good absolute performance results (Kesseler, 1996).

The Dutch Parallel Reduction machine project (Barendregt *et al.*, 1987; Hartel *et al.*, 1995) used a Miranda-like, lazy language with a special “sandwich” annotation for describing fork-and-join parallelism. Although this annotation favours divide-and-conquer parallelism, other paradigms such as data parallelism can be expressed by using program transformations. The largest application is a tidal prediction program on a small distributed-memory machine.

Other Strict Languages. Parallel extensions to Lisp have a long history: QLisp (Goldman *et al.*, 1989), Paralation Lisp (Di Napoli *et al.*, 1996), based on the general Paralation model (Sabot, 1988), EuLisp (Padget *et al.*, 1993), *Lisp (Thinking Machine Corporation, 1990), FX (Gifford *et al.*, 1992), PaiLisp (Kawamoto, 1999), BaLinda Lisp (Feng *et al.*, 1995), TS/Scheme (Jagannathan, 1993). Some of the most prominent and most influential systems are Multilisp (Halstead, 1985) and its successor MulT (Kranz *et al.*, 1989). The thread creation construct in these two languages is a future, which hides the synchronisation between parallel threads behind ordinary access to variables in a shared address space. In essence, it acts like a `par` operator in GpH. To reduce the overhead imposed by a huge number of parallel threads, lazy task creation was invented by Mohr *et al.* (1991). This technique allows the computation of a potential child thread to be subsumed by the parent thread.

SAC (Single Assignment C) (Scholz, 1996) is a strict, first-order functional language with implicit parallelism and implicit thread interaction, optimised for array processing. Its main application area is scientific computing with its focus on array structures, which can be abstracted over shape and dimensionality, and rather regular parallelism. Good performance results for a Jacobi relaxation algorithm are reported on a shared-memory Sun Enterprise (Grelck, 1998).

The UFO-Lite language (Sargeant, 1993) represents a first-order, hybrid functional object-oriented language with implicit parallelism and implicit thread interaction. Its prototype implementation on an SGI Origin focusses on the efficient handling of fine grained parallelism.

Skeleton-based Languages. A well-engineered skeleton-based language is the implicitly-parallel, strict functional language PMLS (Michaelson *et al.*, 2001). It is an automatically parallelising compiler for a pure subset of SML. The execution costs of functions are profiled by executing a structural operational semantics. Based on this information a cost model for the available skeletons, possibly nested, is used to select a decomposition and mapping of parallel threads. Measurements on a range of parallel machines including a Beowulf cluster, a Fujitsu AP3000, an IBM SP/2, and a Sun Enterprise SMP exhibit good speedups for programs such as matrix multiplication, a ray tracer and a linear system solver (Scaife *et al.*, 2001).

Other well-developed systems using a skeleton-based approach for parallelism are SCL (Darlington *et al.*, 1996) and P3L (Bacci *et al.*, 1995). Both systems define a coordination language that can be freely combined with an arbitrary computation language. In practice these systems often use C or Fortran as computation languages. As a crucial technique for the development of larger applications these languages allow the specification of data re-distribution to compose skeletons with conflicting data distributions.

Data Parallel Languages. One of the most successful parallel functional languages is NESL (Blelloch, 1996). NESL is a strict, strongly-typed, data-parallel language with implicit parallelism and implicit thread interaction. It has been implemented on a range of parallel architectures, including several vector computers. A wide range of algorithms have been parallelised in NESL, including a Delaunay algorithm for triangularisation (Blelloch & Narlikar, 1997), several algorithms for the n-body problem (Blelloch *et al.*, 1996), and several graph algorithms.

Fish (Jay & Steckler, 1998) is a higher-order polymorphic language with strict semantics. Its main innovation is the introduction of shapely types that encode information about the bounds of array-like objects in the type system of the language. This extended type system enables shape analysis and provides additional information to the compiler, which generates very efficient sequential code. The data-parallel variant of this language, GoldFish, is still under development.

Dataflow Languages. SISAL (Cann, 1992) is a first-order, strict functional language with implicit parallelism and implicit thread interaction. Its implementation is based on a dataflow model and it has been ported to a range of parallel architectures. Comparisons of SISAL code with parallel Fortran code show that its performance is competitive with Fortran, without adding the additional complexity of explicit coordination (LANL, 2001).

The pHLuid system (Flanagan & Nikhil, 1996) is a parallel implementation of Id on networks of workstations. It uses a dataflow model of computation to achieve implicit parallelism. The Id language is, despite many syntactic differences, closely related to Haskell. In (Hammes *et al.*, 1995) a good language and performance comparison of Id with Haskell on a realistic benchmark program is given. Id is polymorphic, higher-order and has a non-strict semantics, implemented via lenient or parallel eager evaluation. Indeed, a fusion of Id and Haskell has been proposed (Nikhil *et al.*, 1995).

Derivational Approaches. The referentially transparent semantics of Haskell makes it an attractive language for deriving parallel programs. In such an approach Haskell, or often BMF notation, is used as specification language, and the program is transformed, usually by hand, into a parallel program. The target language is often C with MPI or PVM, but in some cases intermediate points of the transformation are already executable, e.g. as Haskell+MPI programs. The most prominent of these approaches are abstract parallel machines (O'Donnell & Rünger, 2000), the TwoL system (Rauber & Rünger, 1996), systems using BSP (Valiant, 1990) as parallel programming model e.g. (Loulergue, 2000), and several systems for deriving skeleton-based parallel code out of Haskell or BMF specifications (Pepper, 1993; Bacci *et al.*, 1999).

4.6 Parallel Haskell Pragmatics

Tools and Environment. A common feature of the languages discussed in this section is their high-level and often dynamic coordination. Sometimes the programmer only has to identify expressions suitable for parallel execution (GpH) in other cases it suffices to give a high-level description of a process network (Eden, Caliban). In contrast to detailed static coordination, the parallel behaviour induced in a program by high-level, dynamic coordination is far from obvious. This opacity is unfortunate because the programmer must have a clear understanding of parallel behaviour to tune performance. Therefore a set of dynamic profiling and visualisation tools is very important for many parallel functional languages.

The best developed set of parallel profiling and visualisation tools exists for GpH. It consists of a highly-tunable simulator for parallel execution (GranSim) and several parallel profilers including GranCC and GranSP. The latter are post-mortem tools operating on a log file, and visualising multiple aspects of parallel execution, e.g. overall activity of the machine, per-processor activity or per-thread activity. For example, Figure 14 shows an overall activity profile of the `sumEuler` program from Section 4.3.1 executing on a 20 processor Beowulf, with execution time on the x-axis and the number of tasks on the y-axis. The tasks are separated into four classes, depending on their state: running if they are executing; runnable if they could be executed if a processor were available; blocked if they await data under evaluation; and fetching if they are retrieving data from another processor. These tools have been crucial in the parallelisation of a set of large GpH programs (Loidl *et al.*, 1999). The Eden system supports Paradise, a GranSim-like simulator (Hernandez *et al.*, 1999), and Caliban provides similar but less sophisticated visualisation tools for analysing parallel performance (Taylor, 1997).

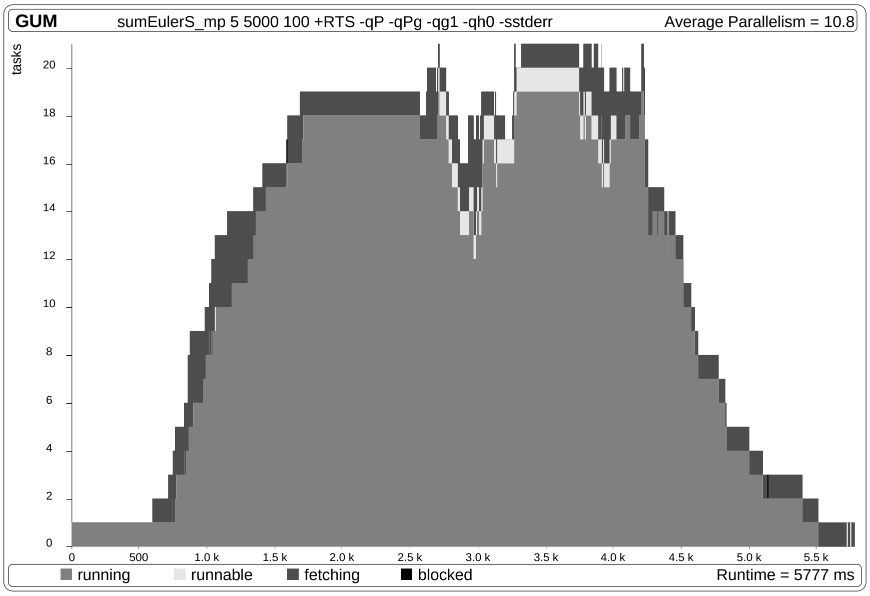

Fig. 14. A GpH Activity Profile

Programming Methodology. Implicit parallelism, often promised in the context of functional languages, offers the enticing vision of parallel execution without changes to the program. In reality, however, the program must be designed with parallelism in mind to avoid unnecessary sequentialisation. In theory, program analyses such as granularity, sharing, and usage analysis can be used to automatically generate parallelism. In practice, however, almost all current systems rely on some level of programmer control. The path from powerful and useful analyses, over the automatic extraction of the right amount of usable parallelism, to the dynamic resource management required for these systems is a long one, and poses many research problems, especially in the middle stage.

Current development methodologies have several interesting features. The combination of languages with minimal explicit coordination and good profiling tools facilitates the prototyping of alternative parallelisations. Obtaining good coordination at an early stage of parallel software development avoids expensive re-designs. In later development stages, detailed control over small but crucial parts of the program may be required, and profiling tools can help locate expensive parallel computations. During performance tuning the high level of abstraction may become a burden, hiding low level features that could be usefully controlled by the programmer. Specific coordination aspects, such as evaluation degree and data placement, often have to be carefully specified in some parts of a program, but they can only be indirectly controlled in languages like HDC, GpH, and Eden.

Implementation Issues. Coordination constructs can be added to an existing computation language such as Haskell in two ways: they may be *built-in* to the language,

`newPort`	::	*IO (Port a)*
`writePort`	::	*Port a* $\rightarrow$ *a* $\rightarrow$ *IO ()*
`readPort`	::	*Port a* $\rightarrow$ *IO a*
`mergePort`	::	*Port a* $\rightarrow$ *Port b* $\rightarrow$ *IO (Port (Either a b))*
`registerPort`	::	*Port a* $\rightarrow$ *PortName* $\rightarrow$ *IO ()*
`unregisterPort`	::	*Port a* $\rightarrow$ *IO ()*
`lookupPort`	::	*PortHost* $\rightarrow$ *PortName* $\rightarrow$ *IO (Port a)*
`link`	::	*Port a* $\rightarrow$ *IO ()* $\rightarrow$ *IO Link*
`unlink`	::	*Link* $\rightarrow$ *IO ()*

Fig. 15. Haskell with Ports Constructs

as in GpH and Eden, or *built-on* the language as a library, as in Haskell+MPI. The main advantage of integrating parallelism into the language is that it facilitates analysis and transformations of the program. Moreover, a tight coupling of parallelism with the runtime-system facilitates dynamic resource management.

On the other hand, providing a separate library for parallelism is in general easier to implement, and achieves a more modular design. It is no coincidence that there are several systems extending Haskell with some form of standard communication library. However, the main problem of this approach is the mismatch between the declarative computation language and a library of imperative coordination constructs. In practice this means that monadic, and therefore serialised, code must be used extensively, which both hampers the design of parallel algorithms and eliminates many of the benefits of a purely functional computation language.

5 Distributed Haskells

Many programs are naturally distributed in nature, that is they comprise multiple threads interacting explicitly on multiple PEs. Examples include CASE tools, multi-user simulations, multi-user distance learning environments. The following sections describe the two recent distributed Haskell implementations, and their relationship to other distributed functional languages.

5.1 Haskell with Ports

Haskell with Ports (Huch & Norbisrath, 2000) is a library for Concurrent Haskell that takes an imperative approach to distribution: adding additional monadic commands for communication between PEs. The design of the library is influenced by the ERLANG language which provides communication via message passing with a mailbox for every process (Armstrong *et al.*, 1996), and by concurrent constraint programming which introduces the notion of a port with a single reader (Janson *et al.*, 1993).

Haskell with Ports has an open system model and threads interact using ports.

```
– Interface –
data ServerMsg = Ping (Port ClientMsg)
data ClientMsg = Pong String

– Server –
main = do
     serverPort ← newPort
     registerPort serverPort "PingServer"
     let pingServer = do
          (Ping clientPort) ← readPort serverPort
          name ← getEnv "HOST"
          writePort clientPort (Pong name)
          pingServer
     pingServer

– Client –
main = do
     putStr "Host of ping server? "
     host ← getLine
     serverPort ← lookupPort host "PingServer"
     clientPort ← newPort
     let
          timePing p = do
               putStr ("Pinging "++show p++" ... ")
              (name,ms) ← timeit (pingPong p)
               putStrLn ("at "++name++" time="++show ms++"ms")
          pingPong p = do
               writePort p (Ping clientPort)
               (Pong name) ← readPort clientPort
               return name
     mapM timePing (replicate 4 serverPort)
     return ()
```

```
Host of ping server? ushas
Pinging (pHost="137.195.52.186") ... at ushas time=60ms
Pinging (pHost="137.195.52.186") ... at ushas time=79ms
Pinging (pHost="137.195.52.186") ... at ushas time=40ms
Pinging (pHost="137.195.52.186") ... at ushas time=68ms
```

Fig. 16. Haskell with Ports Ping

Ports allow explicit and dynamically-typed communication of first order values including ports. Within a location communication is lazy, but between locations communication is strict, i.e. messages sent to local threads are not strictly evaluated, but any message to a remote thread is strict because it is converted to text using `show`. A port may have multiple writers but only one reader, and Figure 15 lists the distribution constructs. A port is created by `newPort`, and the reader must be the port's creator and thus both port and reading thread reside on the same

PE. Data is requested from the port by a blocking `readPort`. The `mergePort` operation enables reading from multiple ports. Values are written to a port with a non-blocking `writePort`, and hence ports are essentially FIFO queues.

A port is *registered* to make it visible to other PEs, using `registerPort` and `unregisterPort`. Once a port is registered it can be retrieved using the PE and port names via the `lookupPort` operation. A separate process, the *postoffice*, executes on each PE and stores information about registered ports as well as performing inter-PE communication. Linking is the eager detection of errors in a port, i.e. actively watching for errors, rather than handling them. By using `link` and `unlink` an operation can be associated with port failure, e.g. a cleanup routine can be specified when a port disconnects. Linking together with exception handling on the read and write operations provide a sound basis for fault tolerant programming.

Ping is an example program that performs a lookup on an environment variable on a remote PE and then returns the resulting value to the original PE. The goal is to determine the overall time for the round trip where communication is the dominating cost. For comparison, in our network the UNIX ping utility returns a time of the order of 0.5ms. Figure 16 shows a pair of Haskell with Ports programs that implement ping, together with their output. The server program creates and registers a port `PingServer`, before looping waiting for messages and responding to them. The client program performs a lookup on the specified server for the port `PingServer`, then a monadic map, `mapM`, is used to call `timePing` four times. Within `timePing`, the `timeit` function times the `pingPong` call which sends a message to the server and waits for a reply. The client program reports relatively long times, which is unsurprising for several reasons. Communication proceeds with a message passing from client, to local postoffice, to remote postoffice, to server, and then back through this chain. All these components are implemented in Haskell and the communication is relatively high level, using sockets and the data is serialised, i.e. manipulated as text using `read` and `show` functions. The implementation is currently being optimised.

Other Haskell with Ports applications include a chat program where users communicate in a client server mode, and a database where users communicate through a client to a central database server to manipulate the data.

In summary, Haskell with Ports provides dynamically typed explicit communication of first order values and `Ports` using a new `Ports` construct. Communication is often strict with no sharing of data and therefore no synchronisation is necessary within the communicated data. An open system model allows programs to leave and join, while registering ports allows a connecting program locate specific resources. Location independence can be implemented but would require a major reimplementation of existing libraries. Exceptions and linking support robust fault tolerance. The library is publicly available from (Haskell+Ports, 2001).

5.2 GdH

GdH (Pointon *et al.*, 2000) is a modest conservative extension of Haskell98 and is a strict superset of both Concurrent Haskell (Peyton Jones *et al.*, 1996) and GpH.

```
forkIO        :: IO () → IO ThreadId
myThreadId    :: IO ThreadId

newEmptyMVar  :: IO (MVar a)
takeMVar      :: MVar a → IO a
putMVar       :: MVar a → a → IO ()
isEmptyMVar   :: MVar a → IO Bool

raiseInThread :: ThreadId → Exception → a
throw         :: Exception → a
catchAllIO    :: IO a → (Exception → IO a) → IO a
```

Fig. 17. Concurrent Haskell Constructs

It supports two classes of thread: pure threads and side-effecting I/O threads. Pure threads are inherited from GpH and intended for parallelism, interacting via shared variables, as described in Section 4.3.1. Evaluation strategies are used in GdH to coordinate pure threads, exactly as in GpH. The remaining discussion focusses on the I/O threads inherited from Concurrent Haskell.

Concurrent Haskell supports explicit interleaved concurrency with named I/O threads created by a monadic `forkIO` command (Peyton Jones *et al.*, 1999), and the constructs are summarised in Figure 17. I/O threads may interact implicitly, like pure threads, or explicitly within the I/O monad using polymorphic semaphore primitives, termed `MVar`s. Multiple threads may share an `MVar`, giving rise to non-deterministic semantics. I/O Threads and `MVar`s can be abstracted over to give buffers, FIFO channels, merging, etc. Concurrent Haskell supports both synchronous and asynchronous exceptions to allow the flexible handling of exceptional or error situations.

```
myPEId  :: IO PEId
allPEId :: IO [PEId]

class Immobile a where
     owningPE :: a → IO PEId
     revalIO  :: IO a → a → IO a

instance Immobile PEId
```

Fig. 18. GdH Constructs

GdH supports distributed programming by extending the semantics of Concurrent Haskell constructs to multiple PEs and adding the new language constructs for

```
- local thread creation -
forkIO          :: IO () → IO ThreadId

- remote thread creation -
rforkIO         :: IO () → PEId → IO ThreadId
rforkIO job p = revalIO (forkIO job) p
```

Fig. 19. Remote Thread Placement Using `revalIO`

location awareness given in Figure 18. A GdH program is a closed system and executes on a set of locations, each labelled with a `PEId`. A thread's current location is obtained by `myPEId`, and the list of all available locations is returned by `allPEId`. Stateful objects, such as `MVars`, threads or files, are unique and fixed at a location, although references to them are freely copied to other locations. Stateful objects are instances of the new `Immobile` class and are located by the `owningPE` method.

While GdH supports location-awareness, significant parts of a GdH program may be location independent. Pure and I/O threads interact implicitly regardless of location as in GpH. Location independent manipulation of stateful objects is supported by rewriting the relevant libraries, like those for MVars, to encapsulate and hide the use of `owningPE` to determine the object's location.

Distributed state is manipulated using a remote evaluation function in the IO monad: `revalIO job p` which blocks the calling thread until the execution of `job` at location `p` completes. That is, `revalIO` temporarily relocates the thread, rather like Java RMI (Daconta *et al.*, 1998). Location independent properties of the remote thread created by `revalIO` are preserved, e.g. error handling remains unaffected so that an exception raised in the remote thread may propagate back to a handler in another location. Stateful object placement can also be accomplished by `revalIO`, for example Figure 19 shows its use to create a distributed version of the Concurrent Haskell `forkIO` command that places a thread on a specified PE.

Partial distributed fault tolerance is supported in GdH by distributed exceptions without requiring any new language concepts. The synchronous and asynchronous exceptions supported by Concurrent Haskell are extended in a location independent manner, e.g. an exception may be raised in a named I/O thread irrespective of whether it is local or remote. The fault tolerance is limited because it is not easy to detect important failures including the failure of a PE and of a thread on a PE. Handling these failure modes is critical for the construction of robust systems and an initial study has been conducted but not yet implemented (Trinder *et al.*, 2000).

A GdH ping program is shown in Figure 20. A destination PE, `dest`, is selected from the list of PEs, and a monadic map, `mapM` calls `timePing` four times. Within `timePing`, `timeit` times the `pingPong` function which uses `revalIO` to perform a trivial operation on the remote `dest` PE. Compared with the pair of Haskell with Ports programs that communicate using explicit ports, the GdH program is a single, relatively compact program with all the communication occurring within the `revalIO` operation. Moreover, the GdH ping is relatively fast, returning values

```
main = do
     (_:dest:_) ← allPEId
     let
          timePing p = do
               putStr ("Pinging "++show p++" ... ")
               (name,ms) ← timeit (pingPong p)
               putStrLn ("at "++name++" time="++show ms++"ms")
          pingPong p = revalIO remote p
          remote = getEnv "HOST"
     mapM timePing (replicate 4 dest)
     return ()
```

```
Pinging PE:524305 ... at ncc1708 time=3ms
Pinging PE:524305 ... at ncc1708 time=1ms
Pinging PE:524305 ... at ncc1708 time=1ms
Pinging PE:524305 ... at ncc1708 time=1ms
```

Fig. 20. GdH Ping

of the same order of magnitude as UNIX ping on our network. This is unsurprising because the GdH runtime system uses PVM with UDP as the underlying protocol, and C code to serialise and pack the data.

Other GdH applications include the following. A cooperative editor allows multiple users on remote machines to edit the same file (Pointon *et al.*, 2000). A Factory Chatroom allows multiple remote clients to interact via a TclHaskell interface to a central server that maintains user profile and a shared factory simulation (Pointon *et al.*, 2001). A distributed file server and a multiuser geographical game have also been constructed (Pointon *et al.*, 2001).

In summary, GdH provides statically typed explicit communication of higher order and stateful objects, e.g. functions, suspensions, MVars. The `Immobile` class allows remote resources to be manipulated and shared in a location independent manner. Furthermore implicit thread interaction occurs through shared data, with communication occurring at the consumer's demand. Implicit thread interaction substantially lifts the burden of managing the communication of, and synchronisation on, data structures from the programmer (Pointon *et al.*, 2001). Additionally GdH is unusual in simultaneously supporting parallelism through pure threads, and distribution through I/O threads. GdH is a closed system, and capitalises on this by making all PEs visible so a program can manipulate any resource in the distributed state. Distributed exception handling is provided to support limited fault-tolerance. A more complete description of the design and implementation of GdH can be found in (Pointon *et al.*, 2000), and the implementation is bundled with publicly available Glasgow Haskell Compiler, version 5.00 onwards (GHC, 2001).

5.3 Other Distributed Functional Languages

Although the benefits of the functional paradigm for distribution have been realised only recently, compared with parallelism, distributed functional languages have already achieved greater commercial success in the form of ERLANG (Armstrong *et al.*, 1996; Blau & Rooth, 1998). This section briefly relates the distributed Haskells above to other distributed functional languages, including some Haskell-based designs. Broadly speaking distributed coordination may be declarative, imperative, or process algebra-based, and the languages discussed below are classified by coordination paradigm.

Before discussing languages by paradigm, it is worth noting that numerous recent language implementations compile to generic platforms like the Java Virtual Machine (JVM) and Microsoft .NET. Despite problems mapping functional virtual machines onto the platforms various classes of functional language have taken this route, including sequential, concurrent, parallel, distributed and mobile languages. An early JVM-based sequential Haskell was produced by Wakeling (1997) and he has since produced a mobile Haskell (Wakeling, 1998). A JVM-based parallel Haskell similar to GpH has been implemented by Rauber du Bois (2001). There is also a JVM-based implementation of the Curry language discussed below.

Declarative Coordination. Several recent distributed Haskell designs use declarative coordination: Distributed Haskell (Chakravarty *et al.*, 1998b) and Curry (Hanus, 1999) use logic-based coordination languages, while Brisk uses annotations, and an elaborated semantics (Holyer *et al.*, 1998). Distributed Haskell coordinates distribution with a constraint programming language. It evolved from the Goffin parallel programming language (Chakravarty *et al.*, 1998b), and a full implementation has not been constructed (Chakravarty *et al.*, 1998a). Concurrently executing processes are called agents, and Distributed Haskell adds language constructs for agent placement and introduces temporal constraints to the language to deal with timeouts and potentially provide fault tolerance. External ports are introduced for communication between applications and dynamic typing ensures the type safety of the messages.

Curry is similar to Goffin in that it is a functional-logic programming language in which communication is a constraint to be solved. To support distribution named ports are added in the I/O monad similar to Haskell with Ports.

Brisk introduces deterministic concurrency using multiple threads within the same shared heap, with implicit synchronisation on shared graph. The limitations of deterministic concurrency are weakened by allowing communication based on merging with hierarchical timestamps (Spiliopoulou, 1999), but the coordination language remains more restrictive than others, e.g. inherently non-deterministic programs like the dining philosophers cannot be described. Brisk allows the communication of higher-order values between PEs in a lazy and dynamic manner, it also supports the communication of code for the mobility of running computations, using a `remote` annotation. As Brisk is currently only partially implemented it is not clear the extend to which distribution will be explicit or implicit.

A major advantage of declarative coordination is that it facilitates reasoning about coordination and computation in a unified framework. Languages with declarative coordination typically have a closed systems model, and preserve referential transparency by making many coordination aspects implicit in the semantics. For example in Brisk the independent sources of output, e.g. different windows, correspond to independent sources of demand within the program. In consequence the implementations of these languages are often extremely elaborate, moreover declarative coordination languages often lack expressive power, as illustrated for Brisk above.

Imperative Coordination. Some coordination languages comprise explicit commands to create processes, communicate etc. ERLANG is probably the most commercially successful functional language, and was developed in the telecommunications industry for constructing distributed, real-time fault tolerant systems (Armstrong *et al.*, 1996; Wikstrom, 1994; Wikstrom, 1996). It has been used by a number of telecommunications companies including One-2-One, Ericsson and NorTel to construct a wide range of telecommunications utilities (Tillman, 2000; Fritchie, 2000; Hinde, 2000), including some large multiprocessor applications like the AXD301 switch (Blau & Rooth, 1998): 525K lines of code on 32 processors. Compared with Haskell, ERLANG is strict, impure, weakly typed and relatively simple: omitting features such as currying and higher-order functions. However the language has a number of extremely useful features, including the OTP libraries, hot loading of new code into running applications, explicit time manipulation to support soft real time systems, and message authentication. ERLANG systems are open, location-aware with explicit mailbox-based communication. Sophisticated fault tolerance is provided by timeouts, exception handlers with exceptions as values, and a mechanism where a process can monitor the termination of other processes.

Distributed Poly/ML and Facile Antigua both extend ML with imperative coordination constructs (Matthews, 1989; Matthews, 1991; Thomsen *et al.*, 1993). A Distributed Poly/ML program creates processes using `fork` and `rfork` primitives, and is location-aware as a PE can be specified. Communication is over channels, using `send` and `receive` primitives. Unusually Distributed Poly/ML provides a nondeterministic `choice` primitive that selects the first of two processes to terminate. In addition to primitives similar to those in Distributed Poly/ML, Facile Antigua provides a `ping` to ascertain the liveness of a PE and `kill` to reset a PE. Both languages have a closed system model and are location-aware, with explicit thread interaction, and some support for fault tolerance.

OZ, the language of the MOZART system, is a multi-paradigm distributed language combining functional, object-oriented, and logic paradigms (Haridi *et al.*, 1997). It provides a variety of primitives for distribution and fault tolerance and supports the communication of higher order values including variables. It uses exceptions for robust fault tolerance and distinguishes between lazy error detection by *handlers* for synchronous exceptions, and eager error detection by *watchers* for the management of asynchronous exceptions which may be generated by remote objects.

Concurrent Clean (Nöcker *et al.*, 1991), introduced in section 4.5, supports distribution using explicit message passing (Serrarens, 2001). It has Channels that allow lazy normal form copying of data structures. Moreover it provides primitives for creating, sharing, and type-checking channels between programs enabling the construction of open systems. Exception-based fault tolerance is also provided.

Imperative approaches are almost always explicit and location aware. Compared with process algebra and declarative coordination languages, it is relatively easy to construct a sophisticated imperative coordination model. The downside is that while it is still easy to reason about the computation parts of a program, it is hard to reason about the entire program because the imperative coordination restricts referential transparency. However, experience with ERLANG suggests that making even part of a large distributed system declarative is of considerable benefit.

Process Algebra Coordination. The imperative coordination model for some languages is based on process algebras like CCS (Milner, 1989) or CSP (Hoare, 1986). Pict is a concurrent language based on asynchronous π-calculus (Turner, 1995), and Nomadic Pict is an extension (Wojciechowski, 2000). The language has explicit coordination commands, e.g. processes synchronise to send and receive. Nomadic Pict programs are location aware: it is possible to `migrate` a process to a PE.

Process algebra languages make coordination explicit, and have the great advantage having a ready-made algebra for reasoning about coordination, timing etc. However, such algebras are very different from the equational approach used for reasoning about the computational parts of a program.

6 Discussion

To facilitate direct comparison, Table 1 summarises the coordination constructs of parallel and distributed Haskells using the concepts from Section 3. Some of the distributed language implementations are not yet mature enough to allow complete definitive classification: these are marked as 'Undef' in the table. Parallel Haskells cover all the major parallelism paradigms, and coordination ranges from fully implicit like HDC, to relatively explicit like Caliban. In comparison to other parallel language paradigms, all of the functional languages are relatively implicit. In comparison to other distributed languages paradigms, many distributed Haskells are closed and do not have well-developed fault tolerance. Coordination of state-transforming threads in distributed Haskells is almost always explicit, and the amount of implicit coordination possible in real distributed applications remains an open question.

Parallel and distributed functional programming the following wide range of challenges, and Haskell-based research languages are likely to be suitable vehicles for investigating many of them.

Reasoning about Coordination. A major challenge is to develop high-level equivalences between expressions in the coordination language, especially for extensible languages describing dynamic coordination. Potentially coordination equivalences will aid the derivation and transformation of parallel and distributed programs,

Language	Threads[a]	Location	Interaction[a]	System model	Fault-tolerance
Sequential:					
Haskell98	N/A	N/A	N/A	N/A	No
Concurrent:					
Concurrent Haskell	Exp.	N/A	Imp. & Exp.	N/A	Yes
Parallel:					
HDC	Imp.	Indep.	Imp.	Closed	No
Data Parallel Haskell	Imp.	Indep.	Imp.	Closed	No
Data Field Haskell	Imp.	Indep.	Imp.	Closed	No
Nepal	Imp.	Indep.	Imp.	Closed	No
GpH	Semi-Exp.	Indep.	Imp.	Closed	No
Eden	Semi-Exp.	Indep.	Imp. & Exp.[b]	Closed	No
Caliban	Semi-Exp.	Indep.	Exp.	Closed	No
Distributed:					
Haskell with Ports	Exp.	Aware	Imp. & Exp.[b]	Open	Yes
GdH	Exp.	Aware	Imp. & Exp.	Closed	Partial
Brisk	Exp.	Aware	Imp.	Closed	Undef.
Distributed Haskell	Exp.	Aware	Imp. & Exp.	Undef.	Undef.
Curry	Exp.	Aware	Imp. & Exp.[b]	Undef.	Undef.

[a] Imp - Implicit, Exp - Explicit.
[b] Restrictions exist on interactions between locations.

Table 1. *Haskell Coordination Language Summary*

and may be incorporated into compilers. The functional programming community has well-developed equational techniques for reasoning about the computation language, but reasoning about coordination is far less developed. Parallel cost models statically predict the time and space required to evaluate an expression, and parallel cost models add a model identifying the expressions simultaneously under evaluation to model coordination aspects such as average parallelism, runtime, and total space usage. Good parallel cost models exist for some skeleton languages, e.g. (Skillicorn, 1990; Bacci *et al.*, 1995), and some data parallel languages, e.g. (Blelloch, 1996). However, there are few models for more dynamic and extensible coordination, and most are low-level, e.g. parallel operational semantics (Blelloch & Greiner, 1996; Roe, 1991; Baker-Finch *et al.*, 2000; Hidalgo Herrero & Ortega Mallén, 2000). The challenge is greater for Haskell because time and space cost models are far harder to develop for lazy languages than for strict (Sands, 1990; Loidl, 1998).

Higher-level Coordination. A major challenge is to develop language constructs, static analyses and dynamic techniques to automatically introduce and control coordination. Many parallel and distributed functional language designers agree that coordination should be as high-level, i.e. implicit, as possible. Current substantially-implicit languages like skeleton-based, data-parallel or distributed languages with declarative coordination, have restricted coordination models as discussed above. The key problem for parallel languages is that functional programs have massive amounts of fine-grained parallelism. In lazy languages like Haskell expressions that can safely be evaluated in parallel can be identified by strictness analyses. Identifying expressions that are worthwhile evaluating in parallel requires accurate parallel cost models. It may also help the programmer if a visualisation of the coordination, e.g. a process network, can be produced statically.

Improved dynamic coordination control mechanisms reduce the explicit coordination control required in the language. This is especially important for non-strict parallel Haskells that naturally support highly-dynamic coordination, and challenges include the following. An important new parallelism concept is architecture independence: i.e. a program can be easily and systematically ported between architectures while preserving good parallel performance. Runtime systems must make good use of emerging architecture independent concepts. For example a runtime system may be parameterised by important architecture characteristics to facilitate good performance on a variety of architectures. Alternately a runtime system may measure key architecture characteristics and adapt itself. The massive fine-grained parallelism in functional programs facilitates adaptation to multiple architectures, but better mechanisms are required to aggregate small tasks into larger tasks and to manage threads cheaply. There is also a need for improved load management strategies to effectively utilise all PEs, and alleviate heavily loaded PEs.

Language constructs with appropriate semantics enable high-level coordination. Languages like Eden and Brisk attempt to capture many coordination aspects in the language semantics. Currently the coordination in these languages is limited, and the high-level constructs are augmented with additional coordination primitives, e.g. Eden uses evaluation strategies in addition to the process constructs. The challenge is to develop a small set of adequately expressive high-level coordination constructs. Just as skeletons abstract over common parallel coordination patterns, it may be possible to construct distributed skeletons to abstract over common distributed coordination patterns, like client-server.

Pragmatic Challenges. An ongoing challenge for parallel and distributed language implementors is to make the best of new technologies. Developing and maintaining the elaborate implementations required by parallel and distributed Haskells is a real issue for research groups. Development is aided by new architecture independent parallel middleware, like the PVM and MPI libraries (PVM, 1993; MPI, 1997), and it is not unusual to find a language available on half-a-dozen architectures. Similarly, the languages gain from improvements in functional compilation technology (Peyton Jones *et al.*, 1993; SML, 1993; Leroy, 1996). Lastly, implementations must adapt to new technologies, e.g. generic platforms like the JVM and .NET, or to make effective use of the increasingly cheap and popular clusters of commodity processors (Ridge *et al.*, 1997).

Programming Methodology. The finest programming language is useless without an established methodology for developing programs systematically. Emerging parallel functional programming methodologies have been discussed in Section 4.6.

Distributed functional programming is far newer, and few systematic development techniques have been used, an exception being (Karlsen, 1999). Specific issues are as follows. Better tools are required to support parallel and distributed program development, including improved profilers with better visualisation. Functional languages currently lack dynamic tools to visualise or control parallel and distributed programs during execution. A standard suite of benchmarks, analogous to the nofib suite (Partain, 1992) would facilitate direct language and implementation comparison. In principle languages like Haskell are a good basis for architecture-independent programming with their massive parallelism and dynamic high-level coordination, but further investigation is required to establish or refute this proposition.

References

Armstrong, J.L., Virding, S.R., Williams, M.C., & Wikstrom, C. (1996). *Concurrent Programming in Erlang*. 2nd edn. Prentice-Hall. (pp 24, 30, 31)

Augustsson, L., & Johnsson, T. (1989). Parallel Graph Reduction with the $\langle v, G\rangle$-machine. *Pages 202–213 of: FPCA'89 — Conference on Functional Programming Languages and Computer Architecture*. Imperial College, London, UK: ACM Press. (p 19)

Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., & Vanneschi, M. (1995). P^3L: A Structured High Level Programming Language and its Structured Support. *Concurrency — Practice and Experience*, **7**(3), 225–255. (pp 17, 21, 33)

Bacci, B., Gorlatch, S., Lengauer, C., & Pelagatti, S. (1999). Skeletons and Transformations in an Integrated Parallel Programming Environment. *Pages 13–27 of: PACT'99 — International Conference on Parallel Architecture and Compilations Techniques*. LNCS, vol. 1662. Springer-Verlag. (p 22)

Baker-Finch, C., King, D.J., Hall, J.G., & Trinder, P.W. 2000 (Sept.). An Operational Semantics for Parallel Lazy Evaluation. *Pages 162–173 of: ICFP'00 — International Conference on Functional Programming*. (pp 3, 33)

Barendregt, H.P., van Eekelen, M.C.J.D., Hartel, P.H., Hertzberger, L.O., Plasmeijer, M.J., & Vree, W.G. (1987). The Dutch Parallel Reduction Machine Project. *Future Generation Computer Systems*, **3**(Dec.), 261–270. (p 20)

Blau, S., & Rooth, J. (1998). AXD301 - A New Generation ATM Switching System. *Ericsson Review*, **75**(1), 10–17. (pp 2, 30, 31)

Blelloch, G.E. (1996). Programming Parallel Algorithms. *Communications of the ACM*, **39**(3), 85–97. (pp 21, 33)

Blelloch, G.E., & Greiner, J. 1996 (May). A Provable Time and Space Efficient Implementation of NESL. *Pages 213–225 of: ICFP'96 — International Conference on Functional Programming*. (p 33)

Blelloch, G.E., & Narlikar, G. (1997). A Practical Comparison of N-Body Algorithms. *Parallel Algorithms*. Series in Discrete Mathematics and Theoretical Computer Science, vol. 30. American Mathematical Society. (p 21)

Blelloch, G.E., Miller, G.L., & Talmor, D. 1996 (May). Developing a Practical Projection-Based Parallel Delaunay Algorithm. *Symposium on Computational Geometry*. ACM. (p 21)

Breitinger, S., Loogen, R., Ortega Mallén, Y., & Peña Mari, R. (1997). Eden — The Paradise of Functional Concurrent Programming. *Glasgow Workshop on Functional Programming*. (pp 2, 4, 7, 15)

Breitinger, S., Loogen, R., & Priebe, S. 1998 (Sept.). Parallel Programming with Haskell and MPI. *Pages 135–154 of: IFL'98 — International Workshop on the Implementation of Functional Languages.* Draft proceedings. (p 19)

Burn, G.L. (1991). Implementing the Evaluation Transformer Model of Reduction on Parallel Machines. *Journal of Functional Programming*, **1**(3), 329–366. (p 20)

Cann, D. (1992). Retire Fortran? A Debate Rekindled. *Communications of the ACM*, **35**(8), 81–89. (p 21)

Carriero, N., & Gelernter, D. (1989). How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, **21**(3), 323–357. (pp 17, 19)

Chakravarty, M., Guo, Y., & Köhler, M. (1998a). Distributed Haskell: Goffin on the Internet. *Pages 80–97 of: Fuji International Symposium on Functional and Logic Programming.* (p 30)

Chakravarty, M.M.T., & Keller, G. 2000 (Sept.). More Types for Nested Data Parallel Programming. *Pages 94–105 of: ICFP'00 — International Conference on Functional Programming.* (pp 12, 13)

Chakravarty, M.M.T., Yike Guo, Köhler, M., & Lock, H.C.R. (1998b). Goffin: Higher-Order Functions Meet Concurrent Constraints. *Science of Computer Programming*, **30**(1–2), 157–199. (p 30)

Chakravarty, M.M.T., Keller, G., Lechtchinsky, R., & Pfannenstiel, W. (2001). Nepal — Nested Data-Parallelism in Haskell. *EuroPar'01 — European Conference on Parallel Processing.* August 28–31, Manchester, U.K.: Springer-Verlag. (pp 7, 12)

Cole, M. (1999). Algorithmic Skeletons. *Pages 289–303 of:* Hammond, K., & Michaelson, G. (eds), *Research Directions in Parallel Functional Programming.* Springer-Verlag. (p 8)

Daconta, M.C., Saganich, A., Monk, E., & Snyder, M. (1998). *Java 1.2 and JavaScript for C and C++ Programmers.* John Wiley & Sons. (pp 6, 28)

Darlington, J., Guo, Y., & To, H.W. (1996). Structured Parallel Programming: Theory meets Practice. *Research Directions in Computer Science.* Cambridge University Press. (pp 17, 21)

Di Napoli, C., Giordano, M., & Furnari, M.M. (1996). A Paralation Model Implementation Based on a Concurrent Lisp Interpreter Community. *Pages 429–436 of: PDP'96 — Euromicro Workshop on Parallel and Distributed Processing.* Braga, Portugal, 24-26 January: IEEE Computer Society Press. (p 20)

Feng, M.D., Wong, W.F., & Yuen, C.K. (1995). Compiling Parallel Lisp for a Shared Memory Multiprocessor. *Pages 487–490 of: International Conference on Parallel and Distributed Computing and Systems.* (p 20)

Flanagan, C., & Nikhil, R.S. (1996). pHluid: The Design of a Parallel Functional Language Implementation on Workstations. *Pages 169–179 of: ICFP'96 — International Conference on Functional Programming.* Philadelphia, Pennsylvania: ACM Press. (p 21)

Foster, I., Olson, R., & Tuecke, S. (1992). Productive Parallel Programming: The PCN Approach. *Journal of Scientific Programming*, **1**(1), 51–66. (pp 2, 17)

Fritchie, S.L. 2000 (Oct.). Sendmail Meets Erlang: Experiences Using Erlang for Email Applications. *International Erlang/OTP User Conference.* (p 31)

GHC. (2001). *Glasgow Haskell Compiler.* WWW page. <URL:`http://www.haskell.org/ghc/`>. (p 29)

Gifford, D.K., Jouvelot, P., Sheldon, M.A., & O'Toole, J.W. 1992 (Feb.). *Report on the FX-91 Programming Language.* Tech. rept. INRIA. (p 20)

Goldberg, B. (1988). Multiprocessor Execution of Functional Programs. *International Journal of Parallel Programming*, **17**(5), 425–473. (p 20)

Goldman, R., Gabriel, R., & Sexton, C. (1989). Qlisp: Parallel Processing in Lisp. *Work-*

shop on Parallel Lisp. LNCS, vol. 441. June 5–7, 1989, Tohoku University, Sendai, Japan: SpringerVerlag. (p 20)

Goldsmith, R., McBurney, D.L., & Sleep, M.R. (1993). *Term Graph Rewriting: Theory and Practice*. John Wiley & Sons. Chap. Parallel Execution of Concurrent Clean on ZAPP. (p 20)

GPH. 2001 (Jan.). *Glasgow Parallel Haskell*. WWW page. <URL:`http://www.cee.hw.ac.uk/~dsg/gph/`>. (p 14)

Grelck, C. (1998). Shared Memory Multiprocessor Support for SAC. *Pages 38–54 of: IFL'98 — International Workshop on the Implementation of Functional Languages*. LNCS, vol. 1595. September 9–11, University College London, UK: Springer-Verlag. (p 20)

Hains, G. (1994). Parallel Functional Languages should be Strict. *Pages 527–532 of: Workshop on GPPP — World Computer Congress*. Hamburg, Germany: North-Holland. (p 3)

Halstead, R. (1985). Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, **7**(4), 106–117. (p 20)

Hammes, J., Lubeck, O., & Böhm, W. (1995). Comparing Id and Haskell in a Monte Carlo Photon Transport Code. *Journal of Functional Programming*, **5**(3), 283–316. (p 22)

Hanus, M. 1999 (Sept.). Distributed Programming in a Multi-Paradigm Declarative Language. *PPDP'99 — Principles and Practice of Declarative Programming*. (p 30)

Haridi, S., van Roy, P., & Smolka, G. (1997). An Overview of the Design of Distributed Oz. *PASCO 97 — International Symposium on Parallel Symbolic Computation*. (p 31)

Hartel, P.H., Hofman, R.F.H., Langendoen, K.G., Muller, H.L., Vree, W.G., & Hertzberger, L.O. (1995). A Toolkit for Parallel Functional Programming. *Concurrency — Practice and Experience*, **7**(8), 765–793. (p 20)

Haskell+Ports. (2001). *Haskell with Ports*. WWW page. <URL:`http://www-i2.informatik.rwth-aachen.de/hutch/distributedHaskell`>. (p 26)

Hernandez, F., Peña, R., & Rubio, F. (1999). From GranSim to Paradise. *Pages 11–19 of: SFP'99 — Scottish Functional Programming Workshop*. Trends in Functional Programming, vol. 1. Stirling, Scotland: Intellect. (p 22)

Herrmann, C. 2000 (June). *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions*. Ph.D. thesis, University of Passau. (pp 2, 6, 9, 10)

Hidalgo Herrero, M., & Ortega Mallén, Y. 2000 (July). A Distributed Operational Semantics for a Parallel Functional Language. *SFP'00 — Scottish Functional Programming Workshop*. Trends in Functional Programming, vol. 2. (pp 3, 33)

Hill, J. 1994 (Sept.). *Data-Parallel Lazy Functional Programming*. Ph.D. thesis, Queen Mary and Westfield College, University of London. (p 13)

Hinde, S. 2000 (Oct.). Use of Erlang/OTP as a Service Creation Tool for INS Services. *International Erlang/OTP User Conference*. (p 31)

Hoare, C. A. R. (1986). *Communicating Sequential Processes*. Prentice Hall. (p 32)

Holmerin, J., & Lisper, B. (2000). Development of Parallel Algorithms in Data Field Haskell. *Pages 762–766 of: EuroPar'00 — European Conference on Parallel Processing*. LNCS, no. 1900. Munich, Germany: Springer-Verlag. (pp 2, 4, 10)

Holyer, I., Davies, N., & Spiliopoulou, E. (1998). Distribution in a Demand Driven Style. *International Workshop on Component-based Software Development in Computational Logic*. (p 30)

HPF. (1993). *High Performance Fortran Language Specification*. Tech. rept. Rice University, Texas, USA. (p 6)

Huch, F., & Norbisrath, U. 2000 (Sept.). Distributed Programming in Haskell with Ports. *Pages 87–100 of: IFL'00 — International Workshop on Implementation of Functional Languages*. LNCS, vol. 2011. (pp 7, 24)

Hudak, P. (1986). Para-Functional Programming. *IEEE Computer*, **19**(8), 60–70. (p 19)

Impala. 2001 (July). *Impala – (IMplicitly PArallel LAnguage Application Suite)*. <URL:`http://www.csg.lcs.mit.edu/impala/`>. (p 17)

Jagannathan, S. (1993). TS/Scheme: Distributed Data Structures in Lisp. *Pages 260–267 of:* Ito, T., & Halstead, R.H.Jr. (eds), *Parallel Symbolic Computing: Languages, Systems, and Applications*. LNCS, vol. 748. October 14-17, 1992, Cambridge, USA: Springer-Verlag. (p 20)

Janson, S., Montelius, J., & Haridi, S. (1993). Ports for Objects in Concurrent Logic Programs. *Pages 211–231 of:* Agha, G., Wegner, P., & Yonezawa, A. (eds), *Research Directions in Concurrent Object-Oriented Programming*. London: MIT Press. (p 24)

Jay, C.B., & Steckler, P.A. (1998). The Functional Imperative: Shape! *Pages 139–53 of:* Hankin, Chris (ed), *ESOP'98 — European Symposium on Programming*. LNCS, vol. 1381. Lisbon, Portugal: Springer-Verlag. (p 21)

Karlsen, E.W. (1999). *Tool Integration in a Functional Programming Language*. Ph.D. thesis, Universität Bremen, Germany. (p 35)

Kawamoto, S. 1999 (Aug.). A Study on Parallel a LISP System with Multiple Evaluation Strategies. *Pages 158–160 of: Record of electrical & communication engineering conversazione*, vol. 68. (p 20)

Keller, G., & Chakravarty, M.M.T. (1999). On the Distributed Implementation of Aggregate Data Structures by Program Transformation. *Pages 108–122 of: HIPS'99 — Intl. Workshop on High-Level Parallel Programming Models and Supportive Environments*. LNCS, vol. 1586. April 12–16, 1999, San Juan, Puerto Rico: Springer-Verlag. (p 12)

Kelly, P., & Taylor, F. (1999). Coordination Languages. *Pages 305–321 of:* Hammond, K., & Michaelson, G. (eds), *Research Directions in Parallel Functional Programming*. Springer-Verlag. (p 17)

Kelly, P.H.J. (1989). *Functional Programming for Loosely-Coupled Multiprocessors*. Research Monographs in Parallel and Distributed Computing. MIT Press. (pp 17, 19)

Kesseler, M. (1996). *The Implementation of Functional Languages on Parallel Machines with Distributed Memory*. Ph.D. thesis, University of Nijmegen. (p 20)

Kingdon, H., Lester, D.R., & Burn, G. (1991). The HDG-machine: a Highly Distributed Graph-Reducer for a Transputer Network. *Computer Journal*, **34**(4), 290–301. (p 20)

Klusik, U., Loogen, R., & Priebe, S. 2000a (July). Controlling Parallelism and Data Distribution in Eden. *Pages 53–64 of: SFP'00 — Scottish Functional Programming Workshop*. Trends in Functional Programming, vol. 2. (p 3)

Klusik, U., Loogen, R., Priebe, S., & Rubio, F. 2000b (Sept.). Process Schemes in Eden — Efficient Parallel Programming with Low Effort. *Pages 133–152 of: IFL'00 — International Workshop on the Implementation of Functional Languages*. LNCS, vol. 2011. (pp 4, 16)

Klusik, U., Peña Mari, R., & Rubio Diez, F. (2001). Replicated Workers in Eden. *CMPP'00 — Constructive Methods for Parallel Programming*. Nova Science. To appear. (p 16)

Kranz, D.A., Halstead Jr., R.H., & Mohr, E. (1989). Mul-T: A High-Performance Parallel Lisp. *Pages 81–90 of: PLDI'91 — Programming Languages Design and Implementation*. SIGPLAN Notices, vol. 24(7). (p 20)

LANL. 2001 (Jan.). *Sisal Performance Data*. WWW page. <URL:`http://www.llnl.gov/sisal/PerformanceData.html`>. (p 21)

Leroy, X. (1996). *The Objective Caml System*. Tech. rept. INRIA. (p 34)

Loidl, H-W. 1998 (Mar.). *Granularity in Large-Scale Parallel Functional Programming.* Ph.D. thesis, Department of Computing Science, University of Glasgow. (pp 5, 33)

Loidl, H-W., Trinder, P.W., Hammond, K., Junaidu, S.B., Morgan, R.G., & Peyton Jones, S.L. (1999). Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, **11**(12), 701–752. (pp 4, 15, 22)

Loidl, H-W., Rubio Diez, F., Scaife, N., Hammond, K., Klusik, U., Loogen, R., G.J., Michaelson., Peña Marí, R., Priebe, S., Rebón Portillo, A., & Trinder, P.W. (2001a). A Comparison of High Level Parallel Functional Languages. . In preparation. (pp 8, 17)

Loidl, H-W., Trinder, P.W., & Butz, C. (2001b). Tuning Task Granularity and Data Locality of Data Parallel GpH Programs. *Parallel Processing Letters*. Selected papers from "HLPP'01 — Intl. Workshop on High-level Parallel Programming and Applications". To appear. (p 15)

Loogen, R. (1999). Programming Language Constructs. *Pages 63–91 of:* Hammond, K., & Michaelson, G. (eds), *Research Directions in Parallel Functional Programming.* Springer-Verlag. (p 8)

Loogen, R., Kuchen, H., & Indermark, K. (1989). Distributed Implementation of Programmed Graph Reduction. *Pages 136–157 of: PARLE 89 — Conference on Parallel Architectures and Languages Europe.* LNCS, vol. 365. Springer-Verlag. (p 20)

Loulergue, F. 2000 (July). Parallel Composition and Bulk Synchronous Parallel Functional Programming. *Pages 77–88 of: SFP'00 — Scottish Functional Programming Workshop.* Trends in Functional Programming, vol. 2. (p 22)

Matthews, D.C.J. (1989). *Papers on Poly/ML.* Tech. rept. University of Cambridge Computer Laboratory. (p 31)

Matthews, D.C.J. (1991). *A Distributed Concurrent Implementation of Standard ML.* Tech. rept. University of Edinburgh. (p 31)

Merrick, L. (1996). *DCOM Technical Overview.* Tech. rept. Microsoft White Paper. (p 5)

Michaelson, G., Scaife, N., Bristow, P., & King, P. (2001). Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Applications.* To appear Spring 2001. (pp 1, 2, 21)

Milner, R. (1989). *Communication and Concurrency.* Prentice Hall. (p 32)

Milner, R., Tofte, M., Harper, R., & MacQueen, D. (1997). *The Definition of Standard ML (Revised).* MIT Press. (p 7)

Mirani, R., & Hudak, P. (1995). First-Class Schedules and Virtual Maps. *Pages 78–85 of: FPCA'95 — Conference on Functional Programming Languages and Computer Architecture.* La Jolla, California: ACM Press. (p 19)

Mohr, E., Kranz, D.A., & Halstead Jr., R.H. (1991). Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, **2**(3), 264–280.

MPI. 1997 (July). *MPI-2: Extensions to the Message-Passing Interface.* Tech. rept. University of Tennessee, Knoxville. (pp 19, 34)

Nikhil, R.S., Arvind, Hicks, J., Aditya, S., Augustsson, L., Maessen, J.-W., & Zhou, Y. 1995 (Jan.). *pH Language Reference Manual.* Tech. rept. CSG Memo 369. Laboratory for Computer Science, M.I.T. (p 22)

Nöcker, E.G.J.M.H., Smetsers, J.E.W., van Eekelen, M.C.J.D., & Plasmeijer, M.J. (1991). Concurrent Clean. *Pages 202–219 of: PARLE'91 — Parallel Architectures and Languages Europe.* LNCS, no. 505. Veldhoven, The Netherlands: Springer-Verlag. (pp 20, 32)

O'Donnell, J. (1999). Data Parallelism. *Pages 191–206 of:* Hammond, K., & Michaelson,

G. (eds), *Research Directions in Parallel Functional Programming*. Springer-Verlag. (p 10)

O'Donnell, J., & Rünger, G. (2000). Abstract Parallel Machines. *Computers and Artificial Intelligence*, **19**, 105–129. (p 22)

Padget, J., Bretthauer, H., & Nuyens, G. (1993). An Overview of EuLisp. *Lisp and Symbolic Computation*, **6**(1/2), 9–98. (p 20)

Pareja, C., Pena, R., Rubio, F., & Segura, C. 2000 (July). Optimising Eden by Transformation. *SFP'00 — Scottish Functional Programming Workshop*. Trends in Functional Programming, vol. 2. (p 3)

Partain, W.D. (1992). The nofib Benchmark Suite of Haskell Programs. *Pages 195–202 of: Glasgow Workshop on Functional Programming*. Workshops in Computing. Ayr, Scotland: Springer-Verlag. (p 35)

Pena, R., & Segura, C. 2000 (Sept.). Non-determinism Analysis in a Parallel-Functional Language. *IFL'00 — International Workshop on Implementation of Functional Languages*. LNCS, vol. 2011. (p 3)

Pepper, P. (1993). Deductive Derivation of Parallel Programs. *Chap. 1, pages 1–53 of: Parallel Algorithm Derivation and Program Transformation*. Kluwer Academic Publishers. (p 22)

Peterson, J., Trifonov, V., & Serjantov, A. (2000). Parallel Functional Reactive Programming. *PADL'00 — Practical Aspects of Declarative Languages*. (p 19)

Peyton Jones, S.L., Clack, C., Salkild, J., & Hardie, M. (1987). GRIP — a High-Performance Architecture for Parallel Graph Reduction. *Pages 98–112 of: FPCA'87 — Conference on Functional Programming Languages and Computer Architecture*. LNCS, vol. 274. Portland, Oregon: Springer-Verlag. (p 20)

Peyton Jones, S.L., Hall, C., Hammond, K., Partain, W., & Wadler, P.L. (1993). The Glasgow Haskell Compiler: a Technical Overview. *Pages 249–257 of: UK Joint Framework for Information Technology, Technical Conference*. (pp 2, 34)

Peyton Jones, S.L., Gordon, A., & Finne, S. (1996). Concurrent Haskell. *Pages 295–308 of: POPL'96 — Symposium on Principles of Programming Languages*. St Petersburg, Florida: ACM. (pp 7, 26)

Peyton Jones, S.L., Hughes, J., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C., & Wadler, P. 1999 (Feb.). *Haskell 98: A Non-strict, Purely Functional Language*. Electronic document available on-line at http://www.haskell.org/. (pp 2, 7, 27)

Plasmeijr, R., van Eekelen, M., Pil, M., & Serrarens, P. (1999). Parallel and Distributed Programming in Concurrent Clean. *Pages 323–338 of:* Hammond, K., & Michaelson, G. (eds), *Research Directions in Parallel Functional Programming*. Springer-Verlag. (p 20)

Pointon, R.F., Trinder, P.W., & Loidl, H-W. 2000 (Sept.). The Design and Implementation of Glasgow distributed Haskell. *Pages 101–116 of: IFL'00 — International Workshop on the Implementation of Functional Languages*. LNCS, vol. 2011. (pp 6, 7, 26, 29)

Pointon, R.F., S., Priebe., Loidl, H-W., Loogen, R., & Trinder, P.W. 2001 (Feb.). Functional vs Object-Oriented Distributed Languages. *Eurocast'01 — International Conference on Computer Aided Systems Theory*. To appear. (p 29)

PVM. 1993 (Aug.). *Parallel Virtual Machine Reference Manual*. University of Tennessee. (pp 19, 34)

Rauber, T., & Rünger, G. (1996). The Compiler TwoL for the Design of Parallel Implementations. *Pages 292–301 of: PACT'96 — International Conference on Parallel Architecture and Compilations Techniques*. Boston, USA: IEEE Computer Society Press. (p 22)

Rauber Du Bois, A.R. 2001 (Feb.). Distributed Execution of Functional Programs on the JVM. *EUROCAST'01 — 8th International Conference on Computer Aided Systems Theory and Technology*. To appear.

Reppy, J.H. 1992 (Jan.). *Higher-order Concurrency*. Ph.D. thesis, Department of Computer Science, Cornell University. Also: Technical Report 92-1285. (p 7)

Ridge, D., Becker, D., Merkey, P., & Sterling, T. (1997). Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. *IEEE Aerospace*. (pp 17, 34)

Roe, P. (1991). *Parallel Programming using Functional Languages*. Ph.D. thesis, Department of Computing Science, University of Glasgow, Glasgow, Scotland. (p 33)

Röjemo, N. (1995). *Garbage Collection, and Memory Efficiency, in Lazy Functional Languages*. Ph.D. thesis, Department of Computing Science, Chalmers University of Technology. (p 11)

Sabot, G.W. (1988). *The Paralation Model: Architecture Independent Programming*. MIT Press. (p 20)

Sands, D. 1990 (Sep). *Calculi for Time Analysis of Functional Programs*. Ph.D. thesis, Imperial College, University of London. (pp 5, 33)

Sargeant, J. 1993 (Nov.). Uniting Functional and Object-Oriented Programming. *Pages 1–26 of: International Symposium on Object Technologies for Advanced Software*. LNCS, vol. 742. (p 21)

Scaife, N., Michaelson, G., & Horiguchi, S. 2001 (Sept.). Comparative Cross-Platform Performance Results from a Parallelizing SML Compiler. *IFL'01 — International Workshop on the Implementation of Functional Languages*. In preparation. (p 21)

Scholz, S-B. 1996 (Oct.). *Single Assignment C – Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen (in German)*. Ph.D. thesis, Institut für Informatik und praktische Mathematik, Universität Kiel. (pp 1, 20)

Serrarens, P. (2001). *Communication Issues in Distributed Functional Programming*. Ph.D. thesis. (p 32)

Siegel, J. (1997). *CORBA Fundamentals and Programming*. New York: John Wiley & Sons. (p 5)

Skillicorn, D.B. (1990). Architecture-Independent Parallel Computation. *IEEE Computer*, **23**(12), 38–50. (p 33)

Skillicorn, D.B., & Talia, D. (1998). Models and Languages for Parallel Computation. *ACM Computer Surveys*, **30**(2), 125–169. (p 5)

SML. 1993 (Feb.). *Standard ML of New Jersey: User's Guide, Version 0.93*. Tech. rept. AT&T Bell Laboratories. (p 34)

Spiliopoulou, E. (1999). *Concurrent and Distributed Functional Systems*. Ph.D. thesis, Department of Computer Science, University of Bristol. (p 30)

Taylor, F.S. (1997). *Parallel Functional Programming by Partitioning*. Ph.D. thesis, Department of Computing, Imperial College, London. (pp 4, 17, 18, 19, 22)

Thinking Machine Corporation. 1990 (Nov.). *Programming in ∗lisp*. 6.0 edn. (p 20)

Thomsen, B., Leth, L., Prasad, S., Kuo, T-M., Krammer, A., Knabe, F., & Giacalone, A. (1993). *Facile Antigua Release Programming Guide*. Tech. rept. European Computer-Industry Centre , Germany. (pp 6, 31)

Tillman, B. 2000 (Oct.). NETSim - Six Years with Erlang. *International Erlang/OTP User Conference*. (p 31)

Trinder, P.W., Hammond, K., Mattson Jr., J.S., Partridge, A.S., & Peyton Jones, S.L. 1996 (May). GUM: a portable implementation of Haskell. *PLDI'96 — Programming Language Design and Implementation*. (p 2)

Trinder, P.W., Hammond, K., Loidl, H-W., & Peyton Jones, S.L. (1998). Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, **8**(1), 23–60. (pp 3, 4, 5, 14)

Trinder, P.W., Pointon, R.F., & Loidl, H-W. (2000). Towards Runtime System Level Fault Tolerance for a Distributed Functional Language. *Pages 103–113 of: SFP'00 — Scottish Functional Programming Workshop.* Trends in Functional Programming, vol. 2. University of St Andrews, Scotland: Intellect. (p 28)

Turner, D.N. (1995). *The Polymorphic Pi-calculus: Theory and Implementation.* Ph.D. thesis, University of Edinburgh. (p 32)

Valiant, L.G. (1990). A Bridging Model for Parallel Computation. *Communications of the acm*, **33**(8). (p 22)

Wakeling, D. 1997 (Sept.). A Haskell to Java Virtual Machine Code Compiler. *Pages 39–52 of: IFL'97 — International Workshop on the Implementation of Functional Languages.* LNCS, no. 1467.

Wakeling, D. 1998 (Sept.). Mobile Haskell: Compiling Lazy Functional Programs for the Java Virtual Machine. *Pages 335–352 of: PLILP — International Symposium on Programming Languages, Implementations, Logics and Programs.* (p 30)

Weber, M. 2000 (July). *hMPI — Haskell with MPI.* WWW page. <URL:`http://www-i2.informatik.rwth-aachen.de/~michaelw/hmpi.html`>. (p 19)

Wegner, P. (1971). *Programming Languages, Information Structures and Machine Organisation.* New York: McGraw-Hill. (p 1)

Wikstrom, C. 1994 (Sept.). Distributed Programming in Erlang. *PASCO'94 — International Symposium on Parallel Symbolic Computation.* (p 31)

Wikstrom, C. 1996 (Apr.). Implementing Distributed Real-time Control Systems in a Functional Language. *IEEE Workshop on Parallel and Distributed Real-Time Systems.* (p 31)

Winstanley, N., & O'Donnell, J. 1997 (Aug.). Parallel Distributed Programming with Haskell+PVM. *Pages 670–677 of: EuroPar'97 — European Conference on Parallel Processing.* LNCS, vol. 1300. (p 19)

Wojciechowski, P. (2000). *Nomadic Pict: Language and Infrastructure Design for Mobile Computation.* Ph.D. thesis, University of Cambridge. (p 32)