Chapter 1

Semi-Explicit Parallel Programming in a Purely Functional Style: GpH

Declarative programming languages can play an important role in the process of designing and implementing parallel systems. They bridge the gap between a high-level specification, with proven properties of the overall system, and the execution of the system on real hardware. Efficiently exploiting parallelism on a wide range of architectures is a challenging task and should in our view be handled by a sophisticated runtime environment. Based on this design philosophy we have developed and formalised Glasgow parallel Haskell (GpH), and implemented it as a conservative extension of the Glasgow Haskell Compiler.

The high-level nature of declarative languages eases the task of mapping an algebraic specification down to executable code. In fact, the operational components of the specification can already be considered an implementation, with the associated properties acting as assertions in the program. Based on a formal model of the declarative language, the validity of these properties can be established by manual proof, which works on a level of detail similar to the specification language itself. Many operational aspects, usually complicating a proof of an implementation, do not come into the picture at this level. Most importantly, unnecessary sequentialisation of the code is avoided.

However, the goal of implicit parallelism has proven an elusive one. Often the automatically generated parallelism is too fine-grained to be efficient. In other cases the data-dependencies between expressions prohibit the generation of a sufficient amount of parallelism. Thus, we employ an approach of semi-explicit parallelism, where only potential parallelism has to be annotated in a program, and all aspects of coordination are delegated to the runtime environment. A corresponding formal model, in the form of a structured operational semantics, handling pools of both realised and potential parallelism, is used to establish the correctness of programs employing semi-explicit parallelism. The runtime environment itself is capable of synchronising parallelism, using automatic blocking on data under evaluation, and by simulating virtual shared memory across networks of machines. Being embedded into the optimised runtime environment for a sequential language, we achieve efficient execution of a high-level language, close to the original specification language, while minimising the programmer effort in parallelising the code and being scalable to large-scale applications that can be executed on heterogeneous networks and computational Grids.

0-8493-0052-5/00/$0.00+$.50
© 2001 by CRC Press LLC

This chapter summarises research performed over more than a decade, covering language design [28], semantics [6] and implementation [29, 21, 22]. In particular this chapter elaborates on the semi-explicit programming model (Section 1.1), reflects on the mapping from specification to executable code (Section 1.2), presents a structural operational semantics (Section 1.3) for reasoning about these parallel programs (Section 1.4), discusses the main characteristics of the graph-reduction-based implementation (Section 1.5), underlines the usability of the system by assessing the performance of existing applications (Section 1.6), and concludes by recapitulating the role of our language and methodology as a tool for specification, transformation and efficient parallel execution (Section 1.7).

1.1 Introduction

One of the key problems of parallel programming is to identify work that may be suitable for parallel execution. Because of their side-effect-free character, it is relatively easy to identify independent expressions in *purely functional languages*, such as Haskell [24], and to then construct independent *threads* to evaluate these expressions. However, the costs of creating and synchronising even ultra-lightweight threads can be extremely high relative to their runtime, and even projected multi-core architectures such as Intel's 80-core research testbed [13] will be unable to extract effective performance from more than a few hundred simultaneously running threads.

Many approaches have therefore been proposed to introduce parallelism for functional languages [16], ranging from purely implicit parallelism to explicit, manual thread creation, communication and load management. For the GpH variant of Haskell that is described here [29], and which targets a *multithreaded* parallel implementation, we have chosen a level of abstraction that hides most of the coordination aspects of the parallel execution, but that still enables the programmer to influence key aspects of thread creation. In this way it is possible to guide the granularity of GpH threads to avoid, for example, the creation of excessively fine-grained threads. The resulting *semi-explicit parallel* model of computation will be elaborated below.

Semi-Explicit Parallelism

Semi-explicit parallel languages [16] form an important class of notations, between explicit parallel languages where all coordination, communication and control is made explicit (e.g. C extended with the MPI communications library or Concurrent ML [26]), and purely implicit notations where no parallelism control at all is provided (e.g. Sisal [27], Id [23] or NESL [9]). While aiming to

`par` ::	$a \to b \to b$	— *parallel composition*
`seq` ::	$a \to b \to b$	— *sequential composition*

FIGURE 1.1: Types of the Basic Coordination Constructs in GpH

provide high levels of abstraction and automatically managing many aspects of parallelism, as with purely implicit approaches, semi-explicit approaches require the programmer to include some directives to specify important aspects of the parallel coordination. Examples of such languages include Skil [10], Concurrent Clean [25], MultiLisp [14] and our own GpH notation.

While semi-explicit approaches were often historically based around the use of annotations, more recent approaches, such as Caliban [18], provide compositional language constructs. This supports a more powerful, flexible, and often programmable, parallel programming methodology where (parallel) coordination is separated from (sequential) computation. Complete parallel programs are then *orchestrated* from lower-level components, which may themselves be broken down into parallel or sequential sub-computations, as required to implement the program.

GpH [29] is a modest extension of Haskell with parallel and sequential composition as its two basic coordination primitives (Figure 1.1). Denotationally, both the `par` and `seq` constructs are *projections* onto the second operand, that is, the value of the first operand is not returned as the result of the construct, though it will usually be shared as part of some sub-computation in the second operand. Operationally, `seq` indicates that the first operand should be evaluated before the second operand, and `par` indicates that the first operand *may* be evaluated in parallel with the evaluation of the second. The latter operation is termed "sparking". Unlike most parallel notations, the act of sparking an expression in GpH does not immediately force thread creation: rather the runtime environment determines which sparks are chosen to become parallel threads based on load and other information. It follows that programmers *simply need to expose* expressions in the program that they believe can usefully be evaluated in parallel. The runtime environment manages the details of parallel execution including thread creation, communication, workload balancing etc., as described in detail in Section 1.5. Parallel implementations of GpH are publicly available from `http://www.macs.hw.ac.uk/~dsg/gph/`.

These two basic operations are then used to build higher-level constructs that help simplify parallel programming. Our early experience of implementing non-trivial programs in GpH showed that unstructured use of `par` and `seq` operators could lead to unnecessarily obscure programs. This problem can be overcome by using *evaluation strategies* [28]: non-strict, polymorphic, higher-order functions that influence both the degree of evaluation of and the parallelism to be found in a GpH expression. Evaluation strategies provide a

```
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
```

FIGURE 1.2: Basic Evaluation Strategy Operations in GPH

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 1.2 shows the basic operations that we provide for managing evaluation strategies. Strategies are defined as functions of type `Strategy` that take a polymorphic argument and return a unit value. The `using` construct then applies an evaluation strategy to a Haskell expression. We define two strategies for handling reduction of Haskell expressions to normal forms. The basic evaluation strategy `rwhnf` specifies that the associated expression is to *weak head normal form* (WHNF), in which no reduction is performed either under $\lambda-$expressions or within data structures. This corresponds to the default non-strict evaluation order used by Haskell. The overloaded `rnf` strategy correspondingly specifies that the associated expression is to be reduced to full normal form (NF), representing a maximal evaluation degree. We provide instantiations of this strategy for all major standard Haskell types.

1.2 From Algebraic Specification to Executable Code

Functional programming languages are designed to provide a high level of abstraction, focusing on what should be computed without committing the machine as to how the computation should be organised. This design principle has made functional languages a popular choice as execution languages for algebraic specifications. The main language features that contribute to this high level of abstraction are higher-order functions, polymorphism and advanced mechanisms for modularisation, such as ML's functors or Haskell's type classes. Tight links between algebraic specification languages and execution languages can be identified in Extended ML (EML) [17], which maps down to ML, or in SPECTRUM [11], which uses advanced features such as type classes as found in Haskell.

In our work we exploit the proximity between algebraic specification lan-

$$
\begin{array}{rcl}
m \mid n & = & \exists k \in \mathbb{N}.\ m \times k = n \\
m \perp n & = & \neg\, \exists k \in \mathbb{N}.\ 1 < k \ \wedge\ k \mid m \ \wedge\ k \mid n \\
\varphi(i) & = & |\, \{m \in \mathbb{N} \mid m < i \ \wedge\ m \perp i\}\, | \\
sumEuler(n) & = & \Sigma_{i=1}^{n}\ \varphi(i)
\end{array}
$$

FIGURE 1.3: Specification of the Euler Totient Function φ

guages and modern functional programming languages by attempting a direct mapping of the specification into Haskell code. In many cases the algebraic specification can already be considered an executable specification without the need of further refinement steps. For more complex specifications the process of mapping it to executable code may yield a number of proof obligations, stated in the axioms of the specification language. The proof obligations may be discharged using tools, including general theorem provers such as Isabelle or Coq, or specialised provers, such as Sparkle. Our model of software development does not tie the development to any particular tool, and the proof obligations may even be discharged manually.

Some advanced language features provided by Haskell facilitate the mapping of a specification into a program. In particular, the sophisticated typing mechanism in Haskell, including type classes and functional dependencies, provides a powerful tool. Recent developments into the direction of dependent types [5] offer even more power to encode value-dependent properties into the type of a function, and move the programming language even closer to a specification language. These latest research directions aside, our current practice in mapping specification to code is, however, one of manually refining the specification to bring it into a rule-based format, suitable for a Haskell-like language. The proof obligations imposed by the axioms in the specification are proven either manually or with the help of a specialised theorem prover.

With respect to the parallel execution of the resulting code, a functional language like GpH avoids the redundant sequentialisation found in imperative languages. Nor do we mandate the specification of a parallelism structure in early stages of the refinement. As demonstrated by the running example used in this chapter, we generate rule-based code, which does not commit to a particular evaluation order. We then identify possible sources of parallelism and specify the parallel coordination using evaluation strategies [28]. Finally, we can exploit equational reasoning at this level to improve the parallel performance of the resulting program. It should be emphasised that this entire process is architecture-independent and Section 1.6 demonstrates good parallel performance of GpH applications on varying architectures from shared-memory machines to wide-area Grid networks.

As a running example, we will use a simple algorithm that computes the sum of the values of the *Euler totient function* for integers in the range from

```
mkList ::  Int → [Int]
mkList n = [1..(n-1)]

gcd ::  Int → Int → Int
gcd x 0 = x
gcd x y = gcd y (rem x y)

relprime ::  Int → Int → Bool
relprime x y = gcd x y == 1

euler ::  Int → Int
euler n = length (filter (relprime n) (mkList n))

sumEuler ::  Int → Int
sumEuler = sum .  (map euler) .  mkList
```

FIGURE 1.4: Sequential Haskell Definition of `sumEuler`

```
sumEulerPar1 n = sum ((map euler (mkList n)) `using` parList rnf)

sumEulerPar2 ::  Int → Int → Int
sumEulerPar2 c n =
  sum ([sum (map euler x) | x ← splitAtN c (mkList n)]
                    `using` parList rnf)
```

FIGURE 1.5: Two Parallel Versions of `sumEuler` in GpH

1 to n, for some given n. The Euler totient function of a given integer i is the number of integers less than i that are relatively prime to i, as shown in Figure 1.3. Figure 1.4 shows the sequential Haskell code that implements this example. It directly maps the specifications in Figure 1.3 to the functions `relprime`, `euler` and `sumEuler`. In the code for `relprime` the following property is exploited: $\forall\ m\ n.\ \text{gcd}\ m\ n = 1 \implies m \perp n$. Each function is preceded by a type signature, for example `mkList ::  Int → [Int]` specifies that `mkList` is a function that takes a fixed precision integer, of type *Int*, and returns a list of integers, of type *[Int]*. The notation `[1..(n-1)]` defines the list of integers between 1 and $n-1$. In the definition of the greatest common divisor (`gcd`) we use the remainder function on integers (`rem`). The higher-order function `filter` selects all elements of a list that fulfil a given predicate, and `map` applies a function to all elements of a list.

Figure 1.5 shows two simple parallel implementations of `sumEuler` that use evaluation strategies. The first parallel version, `sumEulerPar1`, simply applies the `parList rnf` strategy to the elements of the result list. The

```
parList ::  Strategy a → Strategy [a]
parList strat [] = ()
parList strat (x:xs) = strat x `par` (parList strat xs)
```

FIGURE 1.6: Definition of `parList` in GpH

`parList` strategy is a parameterised strategy for lists that applies its argument (another strategy) to each element of the list in parallel. In this case, we will evaluate each element to full normal form using a different thread. The `parList` strategy can easily be defined using the GpH `par` construct, as shown in Figure 1.6. The definition applies the argument strategy, `strat`, to each element of the list `x:xs` in parallel, returning the unit value `()` once the entire list has been traversed (`:` is list cons).

The second parallel version, `sumEulerPar2`, is more sophisticated. It first splits the list elements into groups of size `c` using the `splitAtN` function. A *list comprehension* is used to bind each of the groups to the variable `x` in turn. Here, the list comprehension syntax `[ e | v ← l ]` builds a list of expressions whose value is `e`. The expression `e` depends on the value of `v`, which is drawn from each element of the list `l` in turn. The inner sum of the Euler totients for each group is then calculated, before all these results are summed. The definition thus exploits associativity of the underlying + operator. The reason for splitting the indexes in this way is so that each *group* of `c` list elements may be evaluated in parallel by its own individual thread, thereby increasing granularity. We will revisit this example in Section 1.4, developing versions with improved parallel performance.

1.3 The Operational Semantics of GpH

In this section, we will present a parallel operational semantics for GpH, defined in terms of the *call-by-need* evaluation of a parallel extension to the λ-calculus, GpH-core. By making our semantics explicit in describing the way threads are managed and stored, we are able to reason accurately about the behaviour of GpH programs in terms of both *coordination* and *computation*.

1.3.1 Operational Semantics Overview

The GpH operational semantics is a two-level transition semantics. At the lower level, there are single-thread transitions for performing the ordinary evaluation of expressions through, e.g., β-reduction. All candidate single-thread steps are performed simultaneously and in lock-step. They are then

combined into a parallel computation super-step using coordination relations defined at the upper level. We follow Launchbury's seminal work [19] by using a heap to allow sharing and by requiring all closures to be constructed by some let-binding. Where Launchbury uses a big-step natural semantics, in order to properly model coordination issues, we have chosen to use a small-step computational semantics.

GpH-core is a simple subset of GpH, comprising the *untyped* λ*-calculus* extended with numbers, recursive **let**s, sequential composition, **seq**, and parallel composition, **par**. Expressions are normalised so that all variables are distinct, and the second argument to application and the first argument to **par** must be variables.

$$\begin{aligned} x, y, z &\in \text{Variable} \\ n &\in \text{Number} \\ e &\in \text{Expression} \\ e &::= n \mid x \mid e\ x \mid \lambda x.e \mid \mathbf{let}\ \{x_i = e_i\}_{i=1}^{n}\ \mathbf{in}\ e \\ &\quad \mid e_1\ \mathbf{seq}\ e_2 \mid x\ \mathbf{par}\ e \end{aligned}$$

1.3.2 Heaps and Labelled Bindings

Following [19], we use a heap of bindings of expressions to variables. To deal with parallelism, each binding also carries a label to indicate its state. Thus, heaps are partial functions from variables to expression/thread-state pairs:

$$\begin{aligned} H, K &\in \text{Heap} = \text{Variable} \circ\!\!\rightarrow (\text{Expression}, \text{State}) \\ \alpha, \beta &\in \text{State} \\ \alpha &::= \mathit{Inactive} \mid \mathit{Runnable} \mid \mathit{Active} \mid \mathit{Blocked} \end{aligned}$$

We write individual bindings with the thread state appearing as an annotation on the binding arrow, thus:

$$x \overset{\alpha}{\mapsto} e$$

A binding is *Active* (A) if it is currently being evaluated; it is *Blocked* (B) if it is waiting for another binding before it can continue its own evaluation; and it is *Runnable* (R) if it could be evaluated, but there are currently insufficient resources to evaluate it. All other bindings are *Inactive* (I). Bindings therefore correspond to heap closures and labelled bindings correspond to parallel threads. This is a rather simplified model of parallelism compared with the actual GUM implementation (Section 1.5): we assume idealised parallelism, with no communication costs; and unlike the actual implementation, threads are created instantly and may be migrated at no cost. However, it serves to provide limits on possible parallelism in the actual implementation.

The computational semantics is specified as a relation on heaps, $H \implies H'$. This is, in turn, defined in terms of a notion of single thread transitions (Section 1.3.3) and a scheduling relation (Section 1.3.4). The parallel operational

$$H : z \overset{A}{\mapsto} \mathbf{let}\ \{x_i = e_i\}_{i=1}^{n}\ \mathbf{in}\ e \ \longrightarrow\ (\{x_i \overset{I}{\mapsto} e_i\}_{i=1}^{n}, z \overset{A}{\mapsto} e) \qquad (let)$$

$$(H, x \overset{I}{\mapsto} v) : z \overset{A}{\mapsto} x \ \longrightarrow\ (z \overset{A}{\mapsto} \hat{v}) \qquad (var)$$

$$(H, x \overset{I}{\mapsto} e) : z \overset{A}{\mapsto} x \ \longrightarrow\ (x \overset{R}{\mapsto} e, z \overset{B}{\mapsto} x) \qquad (block_1)$$

$$(H, x \overset{RAB}{\mapsto} e) : z \overset{A}{\mapsto} x \ \longrightarrow\ (z \overset{B}{\mapsto} x) \qquad (block_2)$$

$$H : z \overset{A}{\mapsto} (\lambda y.e)\ x \ \longrightarrow\ (z \overset{A}{\mapsto} e[x/y]) \qquad (subst)$$

$$\frac{H : z \overset{A}{\mapsto} e \ \longrightarrow\ (K, z \overset{\alpha}{\mapsto} e')}{H : z \overset{A}{\mapsto} e\ x \ \longrightarrow\ (K, z \overset{\alpha}{\mapsto} e'\ x)} \qquad (app)$$

$$H : z \overset{A}{\mapsto} v\ \mathbf{seq}\ e \ \longrightarrow\ (z \overset{A}{\mapsto} e) \qquad (seq\text{-}elim)$$

$$\frac{H : z \overset{A}{\mapsto} e_1 \ \longrightarrow\ (K, z \overset{\alpha}{\mapsto} e_1')}{H : z \overset{A}{\mapsto} e_1\ \mathbf{seq}\ e_2 \ \longrightarrow\ (K, z \overset{\alpha}{\mapsto} e_1'\ \mathbf{seq}\ e_2)} \qquad (seq)$$

$$(H, x \overset{RAB}{\mapsto} e_1) : z \overset{A}{\mapsto} x\ \mathbf{par}\ e_2 \ \longrightarrow\ (z \overset{A}{\mapsto} e_2) \qquad (par\text{-}elim)$$

$$(H, x \overset{I}{\mapsto} e_1) : z \overset{A}{\mapsto} x\ \mathbf{par}\ e_2 \ \longrightarrow\ (x \overset{R}{\mapsto} e_1, z \overset{A}{\mapsto} e_2) \qquad (par)$$

FIGURE 1.7: Single Thread Transition Rules

semantics then builds on this to describe a reduction sequence from an initial global configuration to a final global configuration:

$$(H, main \overset{A}{\mapsto} e) \Longrightarrow \ldots \Longrightarrow (H', main \overset{I}{\mapsto} v)$$

where *main* identifies the root expression for the program. Values, v, are in weak head normal form, that is:

$$v ::= n \ \mid\ \lambda x.e$$

1.3.3 Single Thread Transitions

The transition relation $\longrightarrow$ of Figure 1.7 describes the computational step taken by each active binding in the heap. The left hand side in each rule represents a heap with the active binding distinguished by $H : z \overset{A}{\mapsto} e$. Multi-label bindings, such as $x \overset{RAB}{\mapsto} e$ in the *block*$_2$ rule mean that the state is one of

$$\frac{H^A = \{\, x_i \overset{A}{\mapsto} e_i \,\}_{i=1}^n \qquad \{H : x_i \overset{A}{\mapsto} e_i \;\longrightarrow\; K_i\}_{i=1}^n}{H \overset{p}{\Longrightarrow} H[\bigcup_{i=1}^n K_i]} \qquad (\textit{parallel})$$

FIGURE 1.8: Combining Multiple Thread Transitions

R, A or B but not I. The right hand sides of the rules are heaps comprising *only those bindings changed or created by that computation step.*

Let: The *let* rule populates the heap with new bindings. These bindings are inactive since under call-by-need they may not necessarily be evaluated.

Variables and blocking: In the *var* and $block_i$ rules, z is a pointer to another closure (called x). If x has already been evaluated to WHNF (the *var* rule), then z simply receives that value. The notation $\hat{v}$ indicates that all bound variables in v are replaced with fresh variable names. If x is inactive and has not yet been evaluated (the $block_1$ rule), then z blocks at this point and x joins the pool of runnable bindings. Finally, if x is not inactive (the $block_2$ rule), then z blocks but x is unaffected.

Application: Evaluating $e\,x$ involves reducing e to a function abstraction using *app* and then substituting x for the bound variable y using *subst*.

Seq: The *seq* rule initially evaluates e_1 without considering e_2. When and if e_1 is reduced to WHNF, its value (but not any changes to the heap) is discarded by the *seq-elim* rule and evaluation then proceeds to e_2.

Par: The *par* rule potentially introduces parallelism, i.e. suggests that an inactive binding could be made active by putting it into a *Runnable* state which may be promoted to *Active* later if sufficient resources are available. Nothing needs to be done if the binding is not inactive (*par-elim*).

1.3.4 Multi-thread Transitions and Scheduling

The changes required for all active bindings are combined by the *parallel* rule (Figure 1.8) to create a full heap-to-heap transition. This is the key point in the semantics where reductions are carried out in parallel. We write H^A to represent all the active bindings in H, i.e., $H^A = \{x \overset{A}{\mapsto} e \in H\}$. Hence in Figure 1.8 there are precisely n active bindings in H. The notation $H[K]$ updates heap H with all new or changed bindings given by K. A more precise definition, together with a proof that conflicts do not arise between bindings can be found in [6].

The scheduling actions for individual threads are defined in Figure 1.9 as follows: i) any binding that is immediately blocked on a completing thread is made runnable (*unblock*); ii) any active or runnable binding that is in WHNF is made inactive (*deactivate*); iii) as many runnable bindings as resources will allow are made active (*activate*). In the *unblock* rule, the notation e^x

$$(H, x \overset{RA}{\mapsto} v, z \overset{B}{\mapsto} e^x) \xrightarrow{u} (H, x \overset{RA}{\mapsto} v, z \overset{R}{\mapsto} e^x) \qquad (\textit{unblock})$$

$$(H, x \overset{RA}{\mapsto} v) \xrightarrow{d} (H, x \overset{I}{\mapsto} v) \qquad (\textit{deactivate})$$

$$\frac{|H^A| < \mathsf{N}}{(H, x \overset{R}{\mapsto} e) \xrightarrow{a} (H, x \overset{A}{\mapsto} e)} \qquad (\textit{activate})$$

FIGURE 1.9: Single Thread Scheduling Rules

$H \overset{\dagger}{\Longrightarrow} H'$ if:
i) $H \xrightarrow{\dagger}{}^{*} H'$ and ii) there is no H'' such that $H' \xrightarrow{\dagger} H''$.
($\dagger$ is u, d or a.)

FIGURE 1.10: Component Scheduling Relations

represents an expression that is immediately blocked on x, i.e., one of the three forms:

$$e^x ::= x \mid x\ y \mid x\ \mathbf{seq}\ e'$$

Note that in the *activate* rule, N is a parameter to the semantics, indicating the total number of processors. This ensures that no more than N bindings are activated in any step. These rules do not, however, specify *which* bindings are activated: bindings are chosen *non-deterministically* during the activation phase. Since, however, this choice is at the coordination level, it does not change the actual values that are computed.

The rules of Figure 1.10 extend the scheduling rules to multiple bindings. To achieve the maximum possible parallelism with respect to the available processors, it is necessary that all candidate threads are unblocked *before* deactivation and that deactivation takes place *before* activation. This sequence of actions is captured by the *schedule* relation of Figure 1.11.

1.3.5 The Computation Relation

Finally, our full semantic computation relation, *compute*, is defined as a parallel transition $\overset{p}{\Longrightarrow}$ followed by a scheduling of bindings $\overset{s}{\Longrightarrow}$ (Figure 1.11). This ordering ensures that the heaps that appear in a reduction sequence are always fully scheduled. Since our semantics is parameterised on the number of processors, we decorate the computation relation with the number of processors where necessary: $\underset{\mathsf{N}}{\Longrightarrow}$, so $\underset{1}{\Longrightarrow}$ indicates the single-processor case.

$$\stackrel{s}{\Longrightarrow} \quad = \quad \stackrel{a}{\Longrightarrow} \circ \stackrel{d}{\Longrightarrow} \circ \stackrel{u}{\Longrightarrow} \qquad (schedule)$$

$$\Longrightarrow \quad = \quad \stackrel{s}{\Longrightarrow} \circ \stackrel{p}{\Longrightarrow} \qquad (compute)$$

FIGURE 1.11: Overall Scheduling and Computation Relations

1.3.6 Properties of the Operational Semantics

Abramsky's denotational semantics of lazy evaluation [1] models functions by a lifted function space, thus distinguishing between a term Ω (a non-terminating computation) and $\lambda x.\Omega$ to reflect the fact that reduction is to weak head normal form rather than head normal form. This is a widely-used, simple and abstract semantics. The properties and results developed in this section are expressed relative to this denotational semantics.

Launchbury [19] shows a number of results relating his natural semantics of lazy evaluation to Abramsky's denotational semantics. We borrow much of his notation and several of our proofs are inspired by his. Previously we showed that the 1-processor case of our semantics corresponds to Launchbury's.

There are three main properties that we expect of our semantics: *soundness*: the computation relation preserves the meanings of terms; *adequacy*: evaluations terminate if and only if their denotation is not $\perp$; *determinacy*: the same result is always obtained, irrespective of the number of processors and irrespective of which runnable threads are chosen for activation during the computation.

The denotational semantics of our language is given in Figure 1.12. The *Val* domain is assumed to contain a lifted version of its own function space. The lifting injection is *lift* and the corresponding projection is *drop*.

The semantic function:

$$[\![\ldots]\!] : Exp \to Env \to Val$$

naturally extends to operate on heaps, the operational counterpart of environments:

$$\{\!\{\ldots\}\!\} : Heap \to Env \to Env$$

The recursive nature of heaps is reflected by a recursively defined environment:

$$\{\!\{x_1 \mapsto e_1 \ldots x_n \mapsto e_n\}\!\}\rho = \mu\rho'.\rho[x_1 \mapsto [\![e_1]\!]_{\rho'} \ldots x_n \mapsto [\![e_n]\!]_{\rho'}]$$

We also require an ordering on environments: if $\rho \leq \rho'$ then ρ' may bind more variables than ρ but they are otherwise equal. That is:

$$\forall x \,.\, \rho(x) \neq \perp \Rightarrow \rho(x) = \rho'(x)$$

$$
\begin{aligned}
\rho \in Env &= Var \to Val \\
[\![\lambda x.e]\!]_\rho &= lift\ \lambda\epsilon.[\![e]\!]_{\rho[x\mapsto\epsilon]} \\
[\![e\ x]\!]_\rho &= drop([\![e]\!]_\rho)([\![x]\!]_\rho) \\
[\![x]\!]_\rho &= \rho(x) \\
[\![\mathbf{let}\ \{x_i = e_i\}_{i=1}^n\ \mathbf{in}\ e]\!]_\rho &= [\![e]\!]_{\{\!|x_1\mapsto e_1 \ldots x_n\mapsto e_n|\!\}\rho} \\
[\![e_1\ \mathbf{seq}\ e_2]\!]_\rho &= \begin{cases} \bot & \text{if } [\![e_1]\!]_\rho = \bot \\ [\![e_2]\!]_\rho & \text{otherwise} \end{cases} \\
[\![x\ \mathbf{par}\ e]\!]_\rho &= [\![e]\!]_\rho
\end{aligned}
$$

FIGURE 1.12: Denotational Semantics

The arid environment ρ_0 takes all variables to $\bot$.

Soundness. Our computational relation $H \implies H'$ can be considered sound with respect to the denotational semantics in Figure 1.12 if the denotations of all the bindings in H are unchanged in H'. The $\leq$ ordering on environments neatly captures this notion.

PROPOSITION 1.1
If $H \implies H'$ *then for all* ρ, $\{\!|H|\!\}\rho \leq \{\!|H'|\!\}\rho$.

PROOF Induction on the size of H and the structure of expressions. □

Adequacy. We wish to characterise the termination properties of our semantics and Propositions 1.2 and 1.3 show an agreement with the denotational definition. The proofs are modelled on the corresponding ones in [19].

PROPOSITION 1.2
If $(H, z \overset{A}{\mapsto} e) \implies^* (H', z \overset{I}{\mapsto} v)$ *then* $[\![e]\!]_{\{\!|H|\!\}\rho} \neq \bot$.

PROOF For all values v, $[\![v]\!]_{\{\!|H'|\!\}\rho} \neq \bot$ so by Prop.1.1 $[\![e]\!]_{\{\!|H|\!\}\rho} \neq \bot$. □

PROPOSITION 1.3
If $[\![e]\!]_{\{\!|H|\!\}\rho} \neq \bot$, *there exists* H', z, v *such that* $(H, z \overset{A}{\mapsto} e) \implies^* (H', z \overset{I}{\mapsto} v)$.

A proof of Proposition 1.3 is outlined in [6]. It is closely based on the corresponding proof in [19], working with a variant of the denotational semantics

which is explicit about finite approximations.

Determinacy. We now turn to the question of obtaining the same result irrespective of the number of processors and irrespective of which runnable threads are chosen for activation during the computation. Clearly, since the results above hold for any number of processors it follows that *if* an evaluation with N processors gives *main* a value then, depending on which threads are activated, an evaluation with M processors *can* give the same result in the sense of Proposition 1.1.

However, a consequence of the definition of $\stackrel{a}{\Longrightarrow}$ is that the main thread may be left runnable but never progress. It is possible that the main thread could be delayed or suspended indefinitely, if there is a constant supply of unneeded speculative threads being generated and scheduled in place of the main thread. This corresponds to the implementation of GpH, with the management of speculative evaluation the programmer's responsibility [29]. It is possible to define an alternative activation relation $\stackrel{a'}{\Longrightarrow}$ that requires that a runnable thread on which *main* is blocked (in a transitive sense) will be activated in preference to other runnable threads. We can be sure that there will always be a free processor in this circumstance because the blocking action has made one available.

$$H \stackrel{a'}{\Longrightarrow} H' \quad \text{if:}$$

1. $H \stackrel{a}{\longrightarrow}{}^{*} H'$;
2. there is no H'' such that $H' \stackrel{a}{\longrightarrow} H''$ and
3. $req(main, H')$ is active in H'.

$$req(x, K) = \begin{cases} x, & \text{if } x \stackrel{RA}{\mapsto} e \in K \\ req(y, K), & \text{if } x \stackrel{B}{\mapsto} e^y \in K \end{cases}$$

FIGURE 1.13: Stronger Activation Relation

With this version of the activation relation, we can show that if *any* evaluation gives an answer for *main* then they all do, irrespective of the number of processors. For the 1-processor case, it is clear that the definition of $\stackrel{a'}{\Longrightarrow}$ in Figure 1.13 ensures that there is always exactly one active binding and that the blocked bindings form a chain from *main* to that active binding.

The following proposition demonstrates that all the closures activated in the one processor case will also be activated in the multi-processor case. Recall that $\stackrel{}{\underset{\mathsf{N}}{\Longrightarrow}}$ is the computation relation assuming a maximum of N processors.

PROPOSITION 1.4
Given $\mathsf{N} \geq 1$ *processors, suppose*
$(H, main \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow} H_1 \underset{1}{\Longrightarrow} H_2 \ldots$ *and*
$(H, main \overset{A}{\mapsto} e) \underset{\mathsf{N}}{\Longrightarrow} K_1 \underset{\mathsf{N}}{\Longrightarrow} K_2 \ldots$
If x *is active in some* H_i *then there is a* j *such that* x *is active in* K_j.

PROOF Suppose z_k is active in some H_i. By $\overset{a'}{\Longrightarrow}$ there is a chain $main \overset{B}{\mapsto} e^{z_1}, z_1 \overset{B}{\mapsto} e^{z_2}, z_2 \overset{B}{\mapsto} e^{z_3}, \ldots z_k \overset{A}{\mapsto} e$ in H_i.
By induction on the length k of this chain we can show that there must be some K_j where z_k is active in K_j. □

Finally we can bring all these results to bear to prove that evaluation is deterministic in the sense that we get the same answer every time, for any number of processors, assuming the $\overset{a'}{\Longrightarrow}$ activation relation.

COROLLARY 1.1
For any number of processors $\mathsf{N} \geq 1$, *if* $(H, main \overset{A}{\mapsto} e) \underset{1}{\Longrightarrow}^* (H', main \overset{I}{\mapsto} v)$ *and*
$(H, main \overset{A}{\mapsto} e) \underset{\mathsf{N}}{\Longrightarrow} K_1 \underset{\mathsf{N}}{\Longrightarrow} K_2 \ldots$ *then:*

1. *there is some* $i \geq 1$ *such that* $K_i = (K_i', main \overset{I}{\mapsto} v')$;
2. $[\![v']\!]_{\{\!\{K_i'\}\!\}\rho_0} = [\![v]\!]_{\{\!\{H'\}\!\}\rho_0}$

PROOF

1. If there is no such K_i then *main* must remain active or blocked forever. In either case there must be some binding $z \overset{A}{\mapsto} e$ that remains active and does not terminate. In that case the denotation of e in the context of the corresponding heap must be $\perp$ by Prop.1.3. But by Prop.1.4 at some stage in the 1-processor evaluation z will be active and *main* will be (transitively) blocked on z. By Prop.1.2 e will not reach a WHNF so *main* will remain blocked. (Unless $main = z$ in which case the result follows immediately.)

2. $\{\!\{H, main \overset{A}{\mapsto} e\}\!\}\rho_0 \leq \{\!\{H', main \overset{I}{\mapsto} v\}\!\}\rho_0$ by Prop.1.1, so in particular $[\![v]\!]_{\{\!\{H'\}\!\}\rho_0} = [\![e]\!]_{\{\!\{H\}\!\}\rho_0}$.
   Similarly, $[\![v']\!]_{\{\!\{K_i'\}\!\}\rho_0} = [\![e]\!]_{\{\!\{H\}\!\}\rho_0}$.

□

1.4 Program Equivalences and Reasoning

We will now demonstrate how we can perform program transformations on parallel programs in order to improve performance. We will use the semantic properties of the GpH `par` and `seq` constructs, defined in the previous section, to derive versions of the program that expose improved parallel behaviour. Our overall goal is to reduce the total execution time on some specific parallel platform. In working towards this goal, we will increase the degree of parallelism that can be found in the program by judicious introduction of evaluation strategies. However, since unlike the ideal situation considered by our operational semantics, in the real-world the costs and overheads of parallel execution and communication mean that maximal parallelism does not automatically lead to a minimal runtime.

We continue with the sequential version of `sumEuler` below. Note that the three main worker functions (`sum`, `map euler`, and `mkList`) have been composed into a three-stage sequential pipeline using the function composition operator (`.`).

```
sumEuler ::  Int → Int
sumEuler = sum .  map euler .  mkList
```

Despite the simplicity of this program, it is a good example because it exhibits a typical structure of symbolic applications: it uses fairly small auxiliary functions that are combined with function composition and higher-order functions. The overall program structure is a fold-of-map (where the `sum` function is a fold). Operationally, this structure suggests two possible ways of parallelisation: producer-consumer (or pipeline) parallelism, and data parallelism.

1.4.1 Pipeline Parallelism

The function compositions give rise to *producer-consumer (or pipeline) parallelism,* where the initial producer (`mkList`) runs in parallel with its consumer (`map euler`), and this runs in parallel with the final consumer of the list (`sum`). We can specify pipeline parallelism by using a parallel variant of the function composition operator (`.||`), which computes both the producer and the consumer function in parallel. The `sumEuler` function can then be written as:

```
sumEuler =    sum .|| map euler .|| mkList
```

where the pipeline operator is defined as:

```
(f .|| g) x = let { x' = g x ; y = f x' } in x' `par` y
```

While pipeline parallelism can be easily expressed, in this case it is not very efficient, since the tight connection between producer and consumer leads to frequent synchronisation between the generated threads, which can consequently result in poor performance.

This behaviour can be improved by attaching a strategy to the `.||` combinator, ensuring that the producer generates whole blocks of list elements and thus reduces the amount of synchronisation. In order to further increase parallelism, we can use a parallel evaluation strategy to not only specify evaluation degree, but also parallelism on the result of the producer. Here we use parallel, strategic function application `$||` which applies a strategy to the argument of a function: `sumEuler n =`

```
sum $|| (parList rnf) $ map euler $|| (parList rnf) $ mkList n
```

This is a mixture of pipeline and data parallelism, which we study in more detail in the following section. Notably, this pipeline construct makes it possible to specify parallelism on the top level, when combining sequential code: although `mkList n` is a sequential function, the application of `parList rnf` triggers the parallel evaluation of all list elements. For large-scale programming this approach of specifying parallelism when *combining* functions, rather than when defining them, reduces and localises the amount of necessary code changes (see [22] for a more detailed discussion).

1.4.2 Data Parallelism

More promising in this example is to use *data parallelism,* where the same operation is applied to multiple components of a data structure in parallel. In this case, the `euler` function, which is mapped over the list returned by `mkList`, is a good candidate for data parallelism. We can define a parallel variant of the `map` operation that builds on the `parList` strategy, and which abstracts over the pattern of parallelism defined in `sumEulerPar1`, as follows:

`parMap ::` *Strategy b → (a → b) → [a] → [b]*
```
parMap strat f xs = map f xs `using` parList strat
```

This clearly shows how higher level parallel constructs can be constructed in a layered manner, in order to maintain clear separation between coordination and computation: it is obvious from the definition that the value returned from a call to `parMap` is identical to a sequential `map`. We can use this equivalence to simply replace the `map euler` expression by `parMap s euler`, where `s` describes the evaluation degree on the elements of the result list. In this case we choose `rnf` to increase the granularity of the generated tasks.

Examining the behaviour of this data-parallel code we find a large number of very fine-grained threads, since every list element gives rise to a new thread. In order to improve performance further we need to combine computations on neighbouring list elements into one, coarse-grained thread, as in the definition of `parSumEuler2`. We call this process *clustering* of evaluation and demonstrate how it can be derived from this code, and how it improves parallel performance.

```
parListChunk ::  Int → Strategy a → Strategy [a]
parListChunk c strat [] = ()
parListChunk c strat xs = seqList strat (take c xs) `par`
                          parListChunk c strat (drop c xs)
```

FIGURE 1.14: A Clustering Strategy used in `sumEuler`

1.4.3 Strategic Clustering

Clustering is easily introduced into functions that return a list or other collection. For example, it is possible to construct a strategy that captures the splitting technique used in the definition of `sumEulerPar2` above. The `parListChunk` strategy of Figure 1.14 introduces a number of parallel threads to operate on subsequences, or *chunks*, of an input list. Here, each thread evaluates a sub-list of length `c`. The `seqList` strategy is a sequential analogue of `parList`, applying its argument strategy to every element of the list in sequence. Using a `parListChunk c s xs` strategy will therefore generate $\lfloor \frac{k}{c} \rfloor$ potential threads, where k is the length of the list `xs`. The expression `map f xs` can now be clustered as follows.

```
parMapChunk c strat f xs = map f xs `using` parListChunk c strat
```

As before, the advantage of a purely strategic approach is that the clustered coordination operations can be captured entirely by the evaluation strategy and isolated from the computation.

1.4.4 Cluster Class

As defined above, clustering works on the results of a function. However, many common functions do not return a list or other collection as their result. One common example is a `fold` function, that collapses a collection to a single value. In order to cluster such functions, we introduce a generic *Cluster* type class with functions to *cluster* the input data, *lift* the function to operate on the clustered data, and perhaps *decluster* the result. Although clustering changes the computation component of a parallel program, equivalence to the sequential program is maintained by introducing clustering systematically using semantics-preserving identities. We will discuss such transformations in the following sections.

To be more precise, we require that every collection type c that is to be clustered is an instance of the Haskell `MMonad` class, as shown by the subclass dependency for *Cluster* below. We use a formulation of monads based on the functions *munit*, *mjoin* and *mmap* [32], which is more suitable for our

purposes than the usual Kleisli category (see Section 10.4 of [7]) with *return* and *bind* operations that is used in many of the standard Haskell libraries.

```
class  MMonad  c   where
     munit ::  a → c a
     mjoin ::  c (c a) → c a
     mmap  ::  (a → b) → (c a → c b)
```

We introduce a new Haskell class, *Cluster*, parametrised by the collection type c with four operations: *singleton* turns a collection into a 1 element collection of collections; the generalised variant *cluster* n maps a collection into a collection of sub-collections each of size n; *decluster* flattens a collection of collections into a single collection; and *lift* takes a function on $c\ a$ and applies it to a collection of collections. For *singleton*, *decluster* and *lift* we can use existing definitions in the *MMonad* class to provide default definitions.

```
class (MMonad c) => Cluster c where
   singleton ::  c a → c (c a)
   cluster   ::  Int → c a → c (c a)
   decluster ::  c (c a) → c a
   lift      ::  (c a → b) → (c (c a) → c b)

   singleton = munit
   decluster = mjoin
   lift = mmap
```

All instances of the monad class come with proof obligations for the monad identities (see Rules (I)–(III),(i)–(iv) of [32]). From these identities we obtain the following equations relating *lift*, *decluster*, and *singleton*.

$$\begin{aligned} decluster \circ singleton &= id && \text{(M I)}\\ decluster \circ lift\ singleton &= id && \text{(M II)}\\ decluster \circ decluster &= decluster \circ lift\ decluster && \text{(M III)} \end{aligned}$$

$$\begin{aligned} lift\ id &= id && \text{(M i)}\\ lift\ (f \circ g) &= (lift\ f) \circ (lift\ g) && \text{(M ii)}\\ lift\ f \circ singleton &= singleton \circ f && \text{(M iii)}\\ lift\ f \circ decluster &= decluster \circ lift\ (lift\ f) && \text{(M iv)} \end{aligned}$$

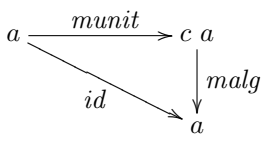
We further require for each n that *cluster* n is a one-sided inverse of *decluster*.

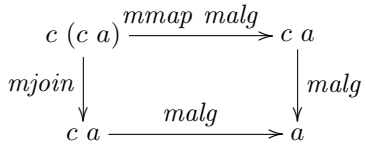
$$decluster \circ cluster\ n = id \qquad \text{(C I)}$$

We now examine the properties of functions that modify the structure of the base domain. We call a function $malg : : c\ a \to a$ an (Eilenberg-Moore) algebra for the monad c if the following two identities hold

$$malg \circ munit = id \tag{A I}$$
$$malg \circ mmap\ malg = malg \circ mjoin \tag{A II}$$

The identities for an algebra can be shown as the two commuting diagrams.





Given two algebras $\alpha : : c\ a \to a$ and $\beta : : c\ b \to b$, a *homomorphism* between them is a function $f : : a \to b$ such that $f \circ \alpha = \beta \circ mmap\ f$.

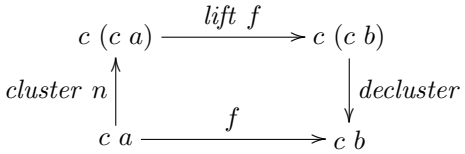
1.4.5 Transforming Clustered Programs

The categorical identities on collections are useful for transforming clustered programs. This section discusses the two main identities we use (lift1 and lift2), and shows that they follow from those of monads and algebras stated in the previous section, together with the identity for *cluster*.

We note that, for every a, the function $mjoin : : c\ (c\ a) \to c\ a$ is an algebra for the monad c (called the *free* algebra on a), and that, for every $f : : a \to b$, $mmap\ f$ is an algebra homomorphism between these free algebras.

Clustering can be introduced into a program by the following identity that holds for all algebra homomorphisms, $f : : c\ a \to c\ b$. Algorithmically the right-hand side splits the input into clusters, applies f to every cluster and flattens the result.

$$f = decluster \circ lift\ f \circ cluster\ n \tag{lift1}$$



Recall that $mmap = lift$ and $mjoin = decluster$. Since f is an algebra homomorphism we know $decluster \circ lift\ f = f \circ decluster$, and (lift1) follows from (C I). The (A II) identity can be used as the following rewrite rule in order to apply the function *malg* to all clusters before combining the results using *malg* again.

$$f = f \circ lift\ f \circ cluster\ n \tag{lift2}$$

```
sumEuler ::  Int → Int → Int
sumEuler c n = sum (map euler (mkList n)
                   `using`
                   parListChunk c rnf )
```

FIGURE 1.15: Strategic Clustering Version of `sumEuler`

Again, using *mmap* = *lift* and *mjoin* = *decluster*, this identity has an easy proof. Since *malg* is an algebra for the monad, the identity *malg* ∘ *lift malg* = *malg* ∘ *decluster* holds by (A II), and (lift2) follows again from (C I).

1.4.6 A Strategic Clustering Version of `sumEuler`

We will now exploit this generic clustering mechanism to provide an improved parallel definition of `sumEuler`. We first summarise the performance analysis of the (unclustered) data parallel version (see also Section 1.6). In the map-phase of `sumEuler` it is easy to exploit data parallelism, by computing every `euler` function in parallel. However, an unclustered version such as this will generally yield a large number of very fine-grained threads, resulting in a speedup that may be close to one.

We have already seen how to improve the granularity of the algorithm by arranging for a whole list chunk to be computed by just one thread, using a `parListChunk` strategy. We now apply this strategy to the `sumEulerPar1` code in Figure 1.5 and arrive at a *"strategic clustering version"* of `sumEuler`, shown in Figure 1.15. This takes the cluster size `c` as an additional parameter and applies the `parListChunk` evaluation strategy to the inner expression. This then generates the list of result values that should be summed. Unfortunately, measurements in [21] show that the parallel performance of the resulting algorithm is still rather unsatisfactory on typical tightly-connected networks. While sufficient parallelism is generated early on in the program, the sequential *fold* at the end of the computation becomes a serious bottleneck, giving typical speedups of only around a factor of four on a sixteen-processor network.

We can identify two reasons for this poor performance. Firstly, the `sum` computation is sequential and this will therefore inevitably generate a sequential tail to the computation, even if it is partially overlapped by the parallel `map`. Secondly, both the argument to and the result of each parallel thread is a list, and these may require a relatively large amount of time to communicate where large chunks are concerned.

```
sumEuler ::  Int → Int → Int
sumEuler z n = sum ((lift worker) (cluster z (mkList n))
                    `using` parList rnf)
               where worker = sum .  map euler
```

FIGURE 1.16: Clustered Version of `sumEuler`

1.4.7 A Generic Clustering Version of `sumEuler`

The derivation of a generic clustering version of `sumEuler` proceeds as follows. First we observe that a list is a monad with list-map as the *mmap* and list-append as the *mjoin* function. Since `sum` is defined as `fold (+) 0` in the Haskell prelude, it is an algebra over lists. Finally, `map euler` is an algebra homomorphism between free algebras.

$$\begin{aligned} sumEuler &= sum \circ map\ euler && \text{(unfold)} \\ &= sum \circ decluster \circ lift\ (map\ euler) \circ cluster\ z && \text{(lift1)} \\ &= sum \circ lift\ (sum) \circ lift\ (map\ euler) \circ cluster\ z && \text{(A II)} \\ &= sum \circ lift\ (sum \circ map\ euler) \circ cluster\ z && \text{(M ii)} \end{aligned}$$

The transformed `sumEuler` code in Figure 1.16 retains separation between the algorithmic and coordination code. In particular, the sequential code can be used as a `worker` function, and the clustering operations are "wrapped" around the (lifted) worker in order to achieve an efficient parallel version.

This version exhibits a much improved average parallelism on the same modern sixteen-processor network as before. The sequential tail in the computation has been vastly reduced, to about 15% of the total runtime compared with about 80% of the total runtime in the strategic clustering version. As a consequence of this improved behaviour we achieve a relative speed-up of 14.3 in a sixteen processor configuration. A detailed performance comparison can be found in [21]. On larger parallel machines it might be advantageous to use a different clustering scheme, which collects every z-th element of the list into one block in order to achieve a better load balance. Notably, such a change would effect only the definition of `(de)cluster` but not the code of `sumEuler` itself.

In summary, we have seen how a (manual) program transformation process, controlled by basic monad and algebra laws, was able to significantly improve the performance of a data parallel algorithm. The developed clustering is generic and can be applied to arbitrary algebras and monads. The parallel performance obtained for the unclustered, and both strategic and generic clustered, versions of the program is reported in Section 1.6.

1.5 The Implementation of GpH

GpH is implemented by the GUM runtime environment [29], a multi-threaded parallel implementation that has been steadily evolved since the early 1990s, and which targets a variety of parallel and distributed architectures ranging from multi-core systems through shared-memory machines and distributed clusters to wide-area computational Grids [30, 4].

1.5.1 Implementation Overview

GUM uses a virtual shared memory model of parallelism, implementing a parallel *graph reduction* mechanism to handle non-strict parallel evaluation of GpH programs. In this model, the program constructs a graph structure during execution. Each node in the graph represents a possibly shared (sub)-expression that may need to be evaluated. Nodes are evaluated if (and only if) they contribute to the result of the program, realising *lazy evaluation*. Following evaluation, the graph node is updated with the value of the subexpression and thus sharing of results is preserved.

Parallelism is introduced by creating threads whose purpose is to evaluate nodes that have been marked using the `par` construct. This may cause other nodes to be evaluated if they are linked as subexpressions of the main node that is evaluated by the thread. Following evaluation, each graph node that has been evaluated by a thread is updated with the result of the evaluation. If one or more threads depend on this result and have therefore been *blocked* waiting for the node to be evaluated, they may now be notified of the value, and unblocked. We will elaborate on this below.

Since we use a virtual shared graph model, communication can be handled *implicitly* through accesses to globally shared nodes rather than through explicit system calls. When a thread needs the value of a globally shared node, and this is not available on the local processor, the processor which owns the master copy will be contacted. In the simplest case, when the master copy has already been evaluated, the value of the result is returned immediately, and recorded as a locally cached copy. Similarly, if the master copy has not yet been evaluated, it is returned in an unevaluated form to the requesting processor. The local node will now become the master copy, and the thread will continue by evaluating this node. If the master copy is currently under evaluation, the thread that requested the value becomes blocked until the result is produced, at which point it will be notified of the result.

An important feature of the GUM implementation is the use of a local heap in addition to a global heap. A local heap is used to hold cached copies of global values that will not change in future (the use of a purely functional language ensures there are no cache coherence issues — once produced, the value of a graph node is fixed and will never change), and to hold a graph that is not

reachable globally. Since the majority of a graph that is produced falls into the latter category, this is a major advantage. The dual-level approach allows fast independent local collection, integrated with a slower, but much less frequently used global collection mechanism, currently based around distributed reference counting.

1.5.2 Thread Management

A *thread* is a virtual processor that executes a task to evaluate a given graph node. GUM threads are extremely lightweight compared to operating system constructs such as *pthreads*. They are implemented entirely within the language's runtime environment, and contain minimal state: a set of thread-specific registers plus a pointer to a dedicated stack object.

Threads are allocated to processing elements (PEs), which usually correspond to the cores or CPUs that are available on the target system. Each PE has a pool of runnable threads. At each scheduling step, the runtime scheduler selects one of these threads for execution. This thread then runs until either it completes, it blocks, or the system terminates as the result of an error condition (such as insufficient memory). This *unfair* scheduling approach has the advantage of tending to decrease both the space usage and the overall execution time [12], which is beneficial for parallel execution. However, it is not suitable for *concurrency*, or for handling *speculative threads*, since in both cases it is necessary to interleave thread execution.

In GpH, parallelism is introduced using `par` constructs in the source program. When the expression `e1 ‘par‘ e2` is evaluated, `e1` is *sparked* for possible future evaluation, and then `e2` is evaluated. Sparking involves recording the graph node associated with `e1` in the current PE's *spark pool*. At a future point, if there is insufficient workload, sparks may be selected from the spark pool and used to construct threads to actually evaluate the closure. Sparking a thunk (an unevaluated expression) is thus a cheap and lightweight operation compared with thread creation, usually involving adding only a pointer to the spark pool.

1.5.3 Spark Stealing

If a PE becomes idle, it will extract a previously saved spark from its local spark pool if it can, and use this to create a new thread. If the spark is no longer useful (eg., because the node it refers to has already been evaluated by another thread), it will be discarded, and another one chosen for execution.

If the process fails and there are no useful local sparks, then the PE will attempt to find work from another PE. It does this by generating a "FISH" message that is passed at random from PE to PE until either some work is found, or it has visited a preset number of PEs. If no work is found, then the message is returned to the originating PE, and after a short, tunable delay, a new FISH message is generated.

If the PE that receives a FISH has a useful spark it sends a "SCHEDULE" message to the PE that originated the FISH, containing the corresponding graph node packaged with a tunable amount of nearby graph. The spark is added to the local spark pool, and an ACK message is sent to the PE that donated the spark.

1.5.4 Memory Management

Parallel graph reduction proceeds on a shared program/data graph, and a primary function of the runtime environment of a parallel functional language is to manage the virtual shared memory in which the graph resides.

In GUM, most sequential execution is exactly as in the standard sequential Glasgow Haskell Compiler (GHC) implementation. This allows the GUM implementation to take advantage of all the sequential optimisations that have been built into GHC. Each PE has its own local heap memory that is used to allocate graph nodes and on which it performs local garbage collections, independently of other PEs. Local heap addresses are normal pointers into this local heap.

Global addresses (GAs) are used to refer to remote objects. A global address consists of a (PE identifier, local identifier) pair. Each PE then maintains a table of *in-pointers* mapping these local identifiers to specific local addresses. The advantage of this approach over simply using local addresses as part of a global address is that it allows the use of a copying garbage collector, such as the generational collector used in GHC. The garbage collector treats the table of *in-pointers* as additional roots to the garbage collection. The corresponding local address is then updated to reflect the new location of the object following garbage collection. An important part of this design is that it is easy to turn *any* local address into a global address (so that it can be exported to another PE), simply by creating a new in-pointer.

In order to allow in-pointers to be garbage collected, we use a *weighted reference counting* scheme [8], where global addresses accumulate some weight, which is returned to the owner node if the remote copy of the node is garbage collected. If all weight is returned, the in-pointer becomes garbage, and the local node may be a candidate for recovery during garbage collection.

The only garbage not collected by this scheme consists of cycles that are spread across PEs. We plan ultimately to recover these cycles, too, by halting all PEs and performing a global collective garbage collection, but we have not yet found the need for this in practice, even in very long-lived applications.

1.6 Assessment of Parallel Performance

Taking `sumEuler` as our running example, Figure 1.17 compares the relative speedups of the unclustered version and both strategic and generic clustering versions developed in Section 1.4. The figure reports results on a 16-node Beowulf cluster of Linux RedHat 6.2 workstations with 533MHz Celeron processors and 128MB of DRAM connected through a 100Mb/s fast Ethernet switch. It shows that the unclustered version produces hardly any speedup at all, due to the extreme fine granularity of the generated parallelism. A naive strategic version, which combines the execution of neighbouring elements, produces only speedups up to 3.7, mainly owing to the sequential sum operation at the end. The improved generic clustering version avoids this sequential bottleneck at the end and shows a good speedup of 14.3 on 16 processors.

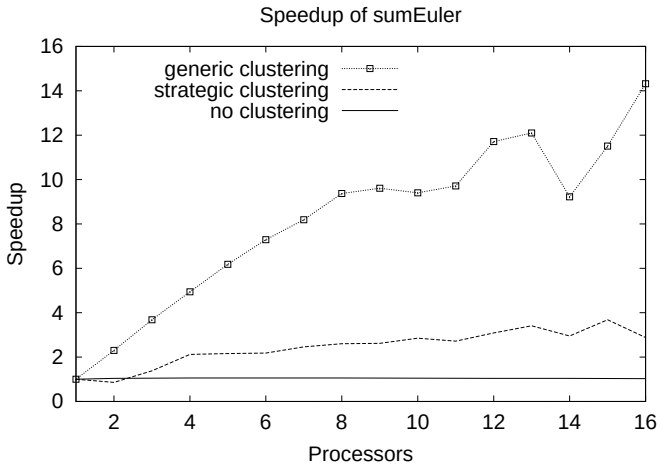


FIGURE 1.17: Speedups of Different Versions of `sumEuler` [21]

We have used our high-level program development approach on applications from different application areas, and Table 1.1 summarises key aspects of some example programs. The first column shows the parallel paradigm. The second and third columns show the application domain and program name, respectively. The fourth and fifth columns present the total size of the program

and size of the code specifying the parallel coordination, both measured in Source Lines of Code (SLOC). The last two columns report the maximal parallel performance and give a reference to detailed performance measurements. Parallel performance is measured as maximum relative speedup, i.e., speedup over the runtime of the *parallel* program executed on a single processor, on a given number of Processing Elements (PEs).

TABLE 1.1: Program Characteristics and Performance Summary

Parallel	Applications		size		Max Speed	Ref
Paradigm	Domain	Name	Code	Coord.	Spdup/PEs	
Data Parallel	Symbolic	`linSolv`	121	22	11.9/16	[20]
	Computation	`smallGroup`	52	8	26.9/28	[2]
	Graphics	`raytracer`	80	10	8.9/16	[20]
	Numeric	`sumEuler`	31	5	9.1/10	[31]
		`matMult`	43	9	3.4/4	[20]
Divide&Conquer	AI	`Queens`	21	5	7.9/16	[20]
Nested Parallelism	Natural Lang.- Processing	`Lolita`	47k	13	1.4/2	[22]
		`Naira`	5k	6	2.5/5	[22]

A key aspect of GpH programming reflected by the fifth column of Table 1.1 is that the parallelisation requires only small code changes, localised in only a few modules. This is in stark contrast to the pervasive changes required by a lower-level parallel programming models. The applications require a range of parallelism paradigms. The majority of the applications are symbolic in nature and exhibit irregular parallelism, i.e., both varying numbers and sizes of tasks. Hence the parallelism results should not be directly compared with more regular problems where near-optimal parallelism is achievable. The results show that, despite their very different computational structures, GpH delivers respectable parallel performance for all of the programs and on widely-varying architectures, from modern multi-cores to very high latency computational Grids [30, 22, 4]. The performance results show that we can obtain good parallelism on local networks, such as Beowulf clusters, e.g., 26.9 on 28PEs [22]. More recent results exhibit almost linear speedup on an 8-core machine [15], and acceptable performance on a wide-area network composed of three local networks: we achieve speedups of 7 on 7 nodes, and up to 16 on 41 nodes for a challenging graphics application [4, 3]. Thus, the parallel performance of GpH scales well even on heterogeneous high-latency architectures. We attribute the largely architecture-independent nature of the achieved speedup to the high-level of abstraction provided by our parallel programming model. Unlike most programming models it does not tie a concrete parallel implementation to the specifics of the underlying architecture. Rather, a parallel runtime environment is in charge of coordinating the parallelism by adjusting the coordination to dynamic properties such as the current workload.

1.7 Conclusion

On the one hand, the inherent complexity of designing parallel systems calls for a high-level approach of specifying its behaviour. On the other hand, one of the main goals for using parallelism is to improve performance through the coordinated use of many processors on the same application. Thus, an efficient implementation of the specified parallelism is almost as crucial as the correctness of the implementation itself. To reconcile the tension between high-level design and low-level performance, we use GpH, a parallel extension of Haskell, as both specification and as programming language. As the former, it is a useful intermediate step from a high-level algebraic specification to tuned executable code. In this role it exploits the formal foundations of the purely functional programming language Haskell, and enables the programmer to extensively use code transformations to modify the degree of parallelism in the code. As the latter, it profits from an efficient sequential implementation (based on the highly optimising Glasgow Haskell Compiler), augmented with advanced concepts such as a virtual shared heap, yielding a system of efficient, largely architecture-independent parallelism.

We have demonstrated our high-level program development approach on applications from a range of application areas: e.g., numerical analysis, symbolic computation, and natural language processing. These applications use a range of parallelism paradigms and deliver good parallel performance on a range of widely-varying architectures, from modern multi-cores to very high latency computational Grids [30, 22, 4]. We have shown that GpH parallelisation requires minimal, local refactoring rather than the pervasive changes required by lower-level approaches. We attribute the largely architecture-independent performance of GpH to its high-level parallel programming model. In [20] we have outlined how GpH's performance compares favourably with both conventional parallel technologies and with other parallel functional languages.

Our experience in realising numerous parallel applications in various application areas underlines that efficient parallel code can be developed through a transformation-based programming methodology. In this process a set of tools for controlling and examining the parallel behaviour has proven indispensable. Our measurement results indicate good performance both on shared-memory machines and clusters of workstations. The emerging architectures of large-scale, computational Grids on the one hand, and multi-core, parallel machines on the other hand, are a hard stress-test for the underlying parallel runtime environment. Even though we already achieve acceptable performance on both of these new kinds of architectures, we are currently enhancing the features of the runtime environment to better deal with these new architectures, by adding support for dynamically adapting the chosen policies for work distribution and scheduling.

References

[1] S. Abramsky. The Lazy Lambda Calculus. In *Research Topics in Functional Programming*, pages 65–117. Addison Wesley, 1990.

[2] A. Al Zain, K. Hammond, P. Trinder, S. Linton, H.-W. Loidl, and M. Costanti. SymGrid-Par: Designing a Framework for Executing Computational Algebra Systems on Computational Grids. In *International Conference on Computational Science (2)*, LNCS 4488, pages 617–624. Springer, 2007.

[3] A. Al Zain, P. Trinder, H-W. Loidl, and G. Michaelson. Supporting High-Level Grid Parallel Programming: the Design and Implementation of Grid-GUM2. In *UK e-Science All Hands Meeting*, pages 182–189. EPSRC, September 2007.

[4] A. Al Zain, P. Trinder, G. Michaelson, and H-W. Loidl. Evaluating a High-Level Parallel Language (GpH) for Computational Grids. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):219–233, 2008.

[5] D. Aspinall and M. Hofmann. Dependent Types. In *Advanced Topics in Types and Programming Languages*, pages 45–86. MIT Press, 2005.

[6] C. Baker-Finch, D. King, and P. Trinder. An Operational Semantics for Parallel Lazy Evaluation. In *ICFP'00 — International Conference on Functional Programming*, pages 162–173, Montreal, Canada, September 2000. ACM Press.

[7] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice-Hall, 1995. ISBN 0133238091.

[8] D.I. Bevan. Distributed Garbage Collection Using Reference Counting. In *PARLE'87 — Parallel Architectures and Languages Europe*, LNCS 259, pages 176–187, Eindhoven, June 12–16, 1987. Springer.

[9] G.E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[10] G.H. Botorog and H. Kuchen. Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In *HPDC'96 — International Symposium on High Performance Distributed Computing*, pages 243–252. IEEE Computer Society Press, 1996.

0-8493-0052-5/00/$0.00+$.50
© 2001 by CRC Press LLC

[11] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, and K. Stølen. The Requirement and Design Specification Language SPECTRUM: an Informal Introduction. Technical report TUM 19311/2, Institut für Informatik, TU München, 1993.

[12] F.W. Burton and V.J. Rayward Smith. Worst Case Scheduling for Parallel Functional Programming. *Journal of Functional Programming*, 4(1):65–75, January 1994.

[13] A.A. Chien. Parallelism Drives Computing. In *Manycore Computing Workshop*, Seattle, USA, June 2007.

[14] R. Halstead. Multilisp: a Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):106–117, October 1985.

[15] K. Hammond, A. Al Zain, G. Cooperman, D. Petcu, and P. Trinder. SymGrid: a Framework for Symbolic Computation on the Grid. In *EuroPar'07 — European Conference on Parallel Processing*, LNCS 4641, pages 457–466, Rennes, France, August 2007. Springer.

[16] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer, 1999. ISBN 1-85233-092-9.

[17] S. Kahrs, D. Sannella, and A. Tarlecki. The Definition of Extended ML: a Gentle Introduction. *Theoretical Computer Science*, 173:445–484, 1997.

[18] P.H.J. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989. ISBN 0262610574.

[19] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *POPL'93 — Principles of Programming Languages*, pages 144–154, Charleston, USA, 1993. ACM Press.

[20] H-W. Loidl, F. Rubio Diez, N. Scaife, K. Hammond, U. Klusik, R. Loogen, G. Michaelson, S. Horiguchi, R. Pena Mari, S. Priebe, A. Rebon Portillo, and P. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-order and Symbolic Computation*, 16(3):203–251, 2003.

[21] H-W. Loidl, P. Trinder, and C. Butz. Tuning Task Granularity and Data Locality of Data Parallel GpH Programs. *Parallel Processing Letters*, 11(4):471–486, December 2001.

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

[23] R.S. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. In *Workshop on Massive Parallelism: Hardware, Programming and Applications*, Amalfi, Italy, 1989. CSG Memo 313.

[24] S.L. Peyton Jones, editor. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003. ISBN 0521826144. Also: http://www.haskell.org/.

[25] M.J. Plasmeijr, M.C.J.D. van Eekelen, E. Nöcker, and J.E.W. Smesters. The Concurrent Clean System — Functional Programming on the MacIntosh. In *International Conference of the Apple European University Consortium*, pages 14–24, Paris, 1991.

[26] J.H. Reppy. Concurrent ML: Design, Application and Semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, LNCS 693, pages 165–198. Springer, 1993.

[27] S. Skedzielewski. Sisal. In *Parallel Functional Languages and Compilers*, Frontier Series, pages 105–158. ACM Press, 1991.

[28] P. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[29] P. Trinder, K. Hammond, J.S. Mattson Jr., A.S Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96 — Programming Languages Design and Implementation*, pages 79–88, Philadelphia, USA, May 1996.

[30] P. Trinder, H-W. Loidl, E. Barry Jr., K. Hammond, U. Klusik, S.L. Peyton Jones, and A. Rebon Portillo. The Multi-Architecture Performance of the Parallel Functional Language GpH. In *Euro-Par 2000 — Parallel Processing*, LNCS 1900, pages 739–743, Munich, 2000. Springer.

[31] P. Trinder, H-W. Loidl, and R. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming*, 12(4&5):469–510, July 2002. Special Issue on Haskell.

[32] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.