The Parallel GHC Project

Parallel functional programming for the real world

Andres Löh

Well-Typed LLP

30 May 2011

About Well-Typed LLP

- Founded 2008 in Oxford.
- Three partners: Duncan Coutts, Ian Lynagh, me.
- Plus currently five contractors working on various projects (but not all full-time).

About Well-Typed LLP

- Founded 2008 in Oxford.
- Three partners: Duncan Coutts, Ian Lynagh, me.
- Plus currently five contractors working on various projects (but not all full-time).
- Distributed over the world (Australia, France, Germany, UK, US).
- We are “the Haskell Consultants”, offering planning, support, development and training around the Haskell programming language.
- We created the Industrial Haskell Group which has a collaborative development scheme to benefit the Haskell community, with different membership options (full, associate, academic).

Services we offer

- Planning and designing (Haskell) projects.
- Developing or improving Haskell programs.
- Code review.
- Support for companies using Haskell.
- Training on various topics surrounding Haskell and functional programming.

- We are trying to give back to the community by making as much of our work as possible open-source.

About Haskell

- A purely functional-programming language.
- A strong type system (lots of static guarantees, easy to use, type inference).
- Many compilers, one of them industrial-strength and strongly optimizing (GHC).
- Open-source (most compilers and libraries are BSD-licensed).
- Lots of libraries, Hackage repository.
- Easy to learn (lots of online tutorials).
- A very active community (mailing lists, IRC, StackOverflow, Hackathons).

The Parallel GHC Project

- The currently largest project we have at Well-Typed LLP.
- Funded by Microsoft Research in Cambridge (also GHC HQ).
- Runs for two years.

The Parallel GHC Project

Goals

- polish GHC's support for parallel programming,
- demonstrate the parallel programming in Haskell works and scales,
- develop and improve tools that support the programming task,
- develop tutorials and information material.

The Parallel GHC Project

Participating organizations

- **Los Alamos National Labs** (USA)
  Monte Carlo algorithms for particle and radiation simulation

The Parallel GHC Project

Participating organizations

- **Los Alamos National Labs** (USA)
  Monte Carlo algorithms for particle and radiation simulation
- **Dragonfly** (New Zealand)
  Implementation of a fast Bayesian model fitter

The Parallel GHC Project

Participating organizations

- **Los Alamos National Labs** (USA)
  Monte Carlo algorithms for particle and radiation simulation
- **Dragonfly** (New Zealand)
  Implementation of a fast Bayesian model fitter
- **Willow Garage** (USA)
  High-level distributed robot simulation

The Parallel GHC Project

Participating organizations

- **Los Alamos National Labs** (USA)
  Monte Carlo algorithms for particle and radiation simulation
- **Dragonfly** (New Zealand)
  Implementation of a fast Bayesian model fitter
- **Willow Garage** (USA)
  High-level distributed robot simulation
- **Internet Initiative Japan** (Japan)
  High-performance network servers

The Parallel GHC Project

Participating organizations

- **Los Alamos National Labs** (USA)
  Monte Carlo algorithms for particle and radiation simulation
- **Dragonfly** (New Zealand)
  Implementation of a fast Bayesian model fitter
- **Willow Garage** (USA)
  High-level distributed robot simulation
- **Internet Initiative Japan** (Japan)
  High-performance network servers

Each partner organization has a Haskell project involving parallelism they want to implement.

The Parallel GHC Project

Workflow

- Organizations discuss their project plans with us.
- We jointly develop implementation goals and the design of the programs.
- The organizations develop the programs, with our assistance.
- We identify potential problems and stumbling blocks.
- We spark off separate mini-projects in order to fix such problems.
- We communicate ideas for further improvements to the GHC developers.
- We collect results and experiences and extract it into regular project digests, and later into new tutorial material.

Mini-projects so far

- A web portal for parallel programming in Haskell.
- A monthly newsletter on parallel programming in Haskell.
- Fixing hidden limits in the GHC IO manager.
- A Haskell binding for MPI.
- Better visualizations in ThreadScope.
- Parallel PRNGs in Haskell.
- ...

Why parallel programming in Haskell?

Parallel vs. concurrent

Parallel programming

The goal is to run a program in parallel, on multiple cores, in order to speed up execution time.

Parallel vs. concurrent

Parallel programming

The goal is to run a program in parallel, on multiple cores, in order to speed up execution time.

Concurrent programming

The goal is to describe and manage several mostly independent processes (threads) within a program that appear to be interleaved.

Parallel vs. concurrent

The confusion

- Concurrent programs **might** be run in parallel, but concurrency is mainly a way to structure a program, and makes equally much sense on a single CPU.
- Concurrency leads to a whole range of potential tricky problems (such as deadlocks, race conditions, fairness concerns, etc.)
- Parallelism **can** be achieved using concurrency, but there are several other techniques that can be used.
- It is definitely worth investigating alternatives to concurrency for functional programming.

Parallel vs. concurrent

Haskell

Haskell supports classic concurrency:

- “classic” concurrency interface that allows creation and management of threads as well as communicating between threads;
- higher-level libraries such as Software Transactional Memory that allow the lock-free description of atomic transactions;

Parallel vs. concurrent

Haskell

Haskell supports classic concurrency:

- “classic” concurrency interface that allows creation and management of threads as well as communicating between threads;
- higher-level libraries such as Software Transactional Memory that allow the lock-free description of atomic transactions;

But Haskell supports separate approaches to parallelism:

- semi-explicit parallelism by annotating expressions that should be parallelized;
- higher-level libraries (strategies, skeletons) built on top of the basic interface;
- data parallelism

Parallel vs. concurrent

(Non)Determinism

- Models of concurrency are necessarily nondeterministic. Processes perform IO, events can occur at arbitrary times.
- In contrast, the approaches to Haskell parallelism are **deterministic** – number of processors and scheduling has no influence on the result.
- The programmer need not be concerned about mundane aspects such as scheduling, order of events, maintaining locks, communicating results, synchronization, . . .

A little bit about Haskell

More about Haskell

Purity

In Haskell, functions do not have side effects:

- A function `f :: Int → Int` is a pure function (in the mathematical sense) from integers to integers: there is no IO, no nondeterminism, no randomness involved.
- In particular, applying `f` to the same integer will always result in the same result.

More about Haskell

Purity

In Haskell, functions do not have side effects:

- A function `f :: Int → Int` is a pure function (in the mathematical sense) from integers to integers: there is no IO, no nondeterminism, no randomness involved.
- In particular, applying `f` to the same integer will always result in the same result.
- A function `g :: Int → IO Int` is a function from integers to an IO action yielding an integer. It is **still** pure! Applying `g` to the same integer will always result in the **same action**.
- Evaluating a value of type `IO Int` will itself not cause IO. Only the run-time system can “run” IO actions.

More about Haskell

Benefits of purity

What do we gain?

- It doesn't matter how often a value if evaluated. Might have an impact on efficiency, but not on the result of a program.

More about Haskell

Benefits of purity

What do we gain?

- It doesn't matter how often a value if evaluated. Might have an impact on efficiency, but not on the result of a program.
- We can see how evil a piece of code is by looking at its type. There is no escape from `IO`. Once a piece of code involves IO, it appears in the type.

More about Haskell

Benefits of purity

What do we gain?

- It doesn't matter how often a value if evaluated. Might have an impact on efficiency, but not on the result of a program.
- We can see how evil a piece of code is by looking at its type. There is no escape from `IO`. Once a piece of code involves IO, it appears in the type.
- We can thus in principle treat IO code differently from non-IO code (and similar for other kinds of effects).

More about Haskell

Laziness

Haskell uses non-strict (in practice: lazy) evaluation:
expressions are evaluated **on demand**.

More about Haskell

Laziness

Haskell uses non-strict (in practice: lazy) evaluation:
expressions are evaluated **on demand**.

Examples:

```
const x y = x
test = const 1 (error "foo")
```

```
cond b t e = if b then t else e
```

More about Haskell

Laziness

Haskell uses non-strict (in practice: lazy) evaluation: expressions are evaluated **on demand**.

Examples:

```
const x y = x
test = const 1 (error "foo")
```

```
cond b t e = if b then t else e
```

While laziness is not essential for parallel programming, the ability to define **control operators** is quite useful.

More about Haskell

Laziness

Haskell uses non-strict (in practice: lazy) evaluation: expressions are evaluated **on demand**.

Examples:

```
const x y = x
test = const 1 (error "foo")
```

```
cond b t e = if b then t else e
```

While laziness is not essential for parallel programming, the ability to define **control operators** is quite useful.

Speculative evaluation is allowed.

Deterministic parallel programming using annotations

Basic idea

- We mark parts of the program that we consider suitable for parallel evaluation.
- We let the runtime system decide about all the details.

Annotated programs

- If a program has type `a`, then an annotated program has type `Eval a`.
- Once we've built an annotated program, we can "run" it using `runEval :: Eval a → a`.

Annotation primitives

The core of the annotation mechanism are the following two primitives:

```
rseq :: a → Eval a
rpar :: a → Eval a
```

Annotation primitives

The core of the annotation mechanism are the following two primitives:

```
rseq :: a → Eval a
rpar :: a → Eval a
```

- The function `rseq` marks its argument as a computation that should be evaluated (to head-normal form) before continuing.
- The function `rpar` marks its argument as a computation that might be beneficial to be evaluated in parallel.

About head-normal form

- All Haskell datatypes define some form of tree structure.
- Depending on the type, a node can be one of several **constructors**.
- Depending on the constructor, there are different subtrees of various types.
- Quite comparable to parse trees and context-free grammars.

About head-normal form

- All Haskell datatypes define some form of tree structure.
- Depending on the type, a node can be one of several **constructors**.
- Depending on the constructor, there are different subtrees of various types.
- Quite comparable to parse trees and context-free grammars.

Evaluation:

- If a computation is evaluated far enough so that the root constructor is known, then it is in **head normal form**.
- If a computation is completely evaluated, i.e., if the complete tree is known, then it is in **normal form**.

Composing parallel computations

The datype `Eval` is abstract, but supports more operations:

```
return :: a → Eval a
(>>=)  :: Eval a → (a → Eval b) → Eval b
```

Composing parallel computations

The datype `Eval` is abstract, but supports more operations:

```
return :: a → Eval a
(>>=)  :: Eval a → (a → Eval b) → Eval b
```

- Here, `return` lifts any expression into a computation, without forcing any evaluation.
- The operator `(>>=)` is for combining two computations, where the second can use the result of the first.
- Once a computation is composed, we can run it and get at the final value.

Example

$$
\begin{aligned}
&\text{rpar expensive}_1 \gg\!= \lambda r_1 \rightarrow \\
&\text{rpar expensive}_2 \gg\!= \lambda r_2 \rightarrow \\
&\text{rseq } r_1 \gg\!= \lambda s_1 \rightarrow \\
&\text{rseq } r_2 \gg\!= \lambda s_2 \rightarrow \\
&\text{return } (s_1, s_2)
\end{aligned}
$$

Example

$$
\begin{aligned}
&\text{rpar expensive}_1 \gg\!= \lambda r_1 \rightarrow \\
&\text{rpar expensive}_2 \gg\!= \lambda r_2 \rightarrow \\
&\text{rseq } r_1 \gg\!= \lambda s_1 \rightarrow \\
&\text{rseq } r_2 \gg\!= \lambda s_2 \rightarrow \\
&\text{return } (s_1, s_2)
\end{aligned}
$$

Better syntax:

$$
\begin{aligned}
&\textbf{do} \\
&\quad r_1 \leftarrow \text{rpar expensive}_1 \\
&\quad r_2 \leftarrow \text{rpar expensive}_2 \\
&\quad s_1 \leftarrow \text{rseq } r_1 \\
&\quad s_2 \leftarrow \text{rseq } r_2 \\
&\quad \text{return } (s_1, s_2)
\end{aligned}
$$

Sparks

What really happens on `rpar`:

- The run-time system manages multiple **capabilities** (usually one per CPU core).
- For run-time maintains a **spark pool**.
- An expression marked for parallel execution in **rpar** creates a **spark** in the spark pool.
- If a capability is idle, it can **steal** a spark from the pool.
- Spark evaluation is speculative.

Annotating individual expressions feels low-level – can’t we abstract?

Strategies

type Strategy a $=$ a $\rightarrow$ Eval a

Strategies

```
type Strategy a = a → Eval a
```

Note:

```
rpar   :: Strategy a
rseq   :: Strategy a
return :: Strategy a
```

Applying strategies

```
using :: a → Strategy a → a
x `using` s = runEval (s x)
```

Applying strategies

```
using :: a → Strategy a → a
x `using` s = runEval (s x)
```

This concept of attaching strategies to values allows us to describe general evaluation strategies based on the **type**. And we can do so **separately** from the algorithm.

Composing strategies

```
dot :: Strategy a → Strategy a → Strategy a
(s1 `dot` s2) x = s2 (x `using` s1)
```

More control about evaluation

```
rnf :: NFData a ⇒ Strategy a
```

Evaluates completely (i.e., to normal form, not just head-normal form).

More control about evaluation

```
rnf :: NFData a ⇒ Strategy a
```

Evaluates completely (i.e., to normal form, not just head-normal form).

Note that `rseq` is enough to implement `rnf` for concrete datatypes (thus `rnf` is just a library function):

```
rnf_List :: NFData a ⇒ Strategy [a]
rnf_List []       = return []
rnf_List (x : xs) = do
                      x'  ← rnf x
                      xs' ← rnf_List xs
                      return (x' : xs')
```

Abstracting over list strategies

We make the element strategy an argument:

$$
\begin{aligned}
&\mathrm{eval}_{\mathrm{List}} :: \mathrm{Strategy\ a} \to \mathrm{Strategy\ [a]} \\
&\mathrm{eval}_{\mathrm{List}}\ \mathrm{s}\ [\,] \quad\quad = \mathrm{return}\ [\,] \\
&\mathrm{eval}_{\mathrm{List}}\ \mathrm{s}\ (\mathrm{x} : \mathrm{xs}) = \mathbf{do} \\
&\qquad\qquad\qquad\qquad\quad \mathrm{x}' \ \leftarrow \mathrm{s\ x} \\
&\qquad\qquad\qquad\qquad\quad \mathrm{xs}' \leftarrow \mathrm{eval}_{\mathrm{List}}\ \mathrm{s\ xs} \\
&\qquad\qquad\qquad\qquad\quad \mathrm{return}\ (\mathrm{x}' : \mathrm{xs}')
\end{aligned}
$$

Abstracting over list strategies

We make the element strategy an argument:

```
eval_List :: Strategy a → Strategy [a]
eval_List s []       = return []
eval_List s (x : xs) = do
                         x'  ← s x
                         xs' ← eval_List s xs
                         return (x' : xs')
```

Now:

```
rnf_List :: NFData a ⇒ Strategy [a]
rnf_List = eval_List rnf

par_List :: Strategy a → Strategy [a]
par_List s = eval_List (rpar `dot` s)
```

Common patterns

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

Common patterns

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

We can easily define similar strategies for other data types!

Laziness – good or bad?

Consider `parMap` again:

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

Laziness – good or bad?

Consider `parMap` again:

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

Such a function would be useless in a strict language, as the list would be evaluated prior to executing the function.

Laziness – good or bad?

Consider `parMap` again:

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

Such a function would be useless in a strict language, as the list would be evaluated prior to executing the function.

However . . .

Some pitfalls

Too much laziness

Sparking an expression suggests it for evaluation, but only to weak-head normal form.

Too much laziness

Sparking an expression suggests it for evaluation, but only to weak-head normal form.

If we want to spark computations that return complex datastructures, we have to prevent them from returning immediately with a thunk.

Too much laziness

Sparking an expression suggests it for evaluation, but only to weak-head normal form.

If we want to spark computations that return complex datastructures, we have to prevent them from returning immediately with a thunk.

It's a frequent mistake that parallelism does not arise because that sparked structures are only forced much later, in the main thread.

Too much laziness

Sparking an expression suggests it for evaluation, but only to weak-head normal form.

If we want to spark computations that return complex datastructures, we have to prevent them from returning immediately with a thunk.

It's a frequent mistake that parallelism does not arise because that sparked structures are only forced much later, in the main thread.

Strategies give us an easy way out. One should always think about how far a data structure should evaluate an expression.

Too little laziness

It may be tempting to say

```
reallyLongList ‘using‘ parList rnf
```

for a long list of potentially parallel computations.

Too little laziness

It may be tempting to say

```
reallyLongList `using` parList rnf
```

for a long list of potentially parallel computations.

However, regardless of its argument strategy `parList` always forces the spine of the entire list (in order to generate the sparks).

Too little laziness

It may be tempting to say

```
reallyLongList `using` parList rnf
```

for a long list of potentially parallel computations.

However, regardless of its argument strategy `parList` always forces the spine of the entire list (in order to generate the sparks).

Another strategy can help:

```
parBuffer :: Int → Strategy a → Strategy [a]
```

Maintains a rolling buffer of parallel computations. Only sparks the next if the first is consumed.

Granularity

There is little, but still some overhead to spark creation and computation:

- creating too many, too small sparks will save less work than it creates;
- creating too few, too large sparks will mean that some CPUs are idle while waiting for other required parts of the computation to finish.

Why not automatic?

This also answers why we cannot (yet) do fully automatic parallelisation:

- it is hard to judge the cost of an expression automatically, and therefore generate parallelism with the right granularity;
- Haskell's default laziness can play tricks on us, and making a program more strict automatically can lead to bad results.

Debugging parallel programs

The correctness of a parallel program in Haskell is no harder to achieve than that of a sequential program.

Debugging parallel programs

The correctness of a parallel program in Haskell is no harder to achieve than that of a sequential program.

However, the efficiency can still be tricky to get right. It is a common problem to get less speedup than expected.

GHC events

- The runtime system of GHC can emit statistics about how much each capability has worked, how much time has been spent on garbage collection etc.
- If requested, we can generate an extremely detailed stream of events from the runtime system.
- After execution, the event log can be analyzed and visualized using a tool called **ThreadScope**.

ThreadScope

- Using ThreadScope, we can get a time-based graphical view of CPU activity and garbage collection.
- We can spot granularity problems if we see CPUs starting and stopping on sparks continuously, and hardly doing any real work.
- We can also spot CPUs waiting for other CPUs.
- We can spot heap and garbage collection problems.
- We can spot evaluation order problems if computations take far longer, far shorter than expected or happen at strange times during the program.

ThreadScope

- Using ThreadScope, we can get a time-based graphical view of CPU activity and garbage collection.
- We can spot granularity problems if we see CPUs starting and stopping on sparks continuously, and hardly doing any real work.
- We can also spot CPUs waiting for other CPUs.
- We can spot heap and garbage collection problems.
- We can spot evaluation order problems if computations take far longer, far shorter than expected or happen at strange times during the program.
- Much can still be improved: we are working hard on new and better visualizations of even more information, and on automatic problem detection.

Experience in practice

For the LANL particle simulator, which in essence runs a very large number of simultaneous computations, parallelization using strategies was easy:

- we can chunk the input list into suitably large sets,
- we can buffer the input list to avoid creating too many sparks,
- we can then spark the chunks, and gradually accumulate the staticstics.

Experience in practice

For the LANL particle simulator, which in essence runs a very large number of simultaneous computations, parallelization using strategies was easy:

- we can chunk the input list into suitably large sets,
- we can buffer the input list to avoid creating too many sparks,
- we can then spark the chunks, and gradually accumulate the staticstics.

The strategy can be kept almost completely external to the rest of the code, and it is easy to play with many subtle variations.

Experience in practice

For the LANL particle simulator, which in essence runs a very large number of simultaneous computations, parallelization using strategies was easy:

- we can chunk the input list into suitably large sets,
- we can buffer the input list to avoid creating too many sparks,
- we can then spark the chunks, and gradually accumulate the staticstics.

The strategy can be kept almost completely external to the rest of the code, and it is easy to play with many subtle variations.

Programming patterns (such as map-reduce) can be extracted into reusable libraries.

Expectations for the future

In the Dragonfly project, we expect to get a long way using strategies as well, but we are also planning to use **data parallelism** by means of the **repa** library:

- shape-polymorphic arrays;
- index function specified independently of the data;
- algorithms on the arrays are automatically parallelized;
- library built on top of lower-level primitives.

Conclusions

- When you want to do parallel programming, try to avoid low-level concurrency.
- A pure functional programming language such as Haskell is particularly useful for high-level descriptions of parallel programs.
- Lots of different abstractions can be built on top of just a few primitives.
- We are working on the Parallel GHC Project to improve tools support and show that the current approaches can scale.

Recent news

Strategies and data parallelism are already there right now, but there are new technologies currently in development as well:

- Yet more to come: the **par-monad** package allows a deterministic approach to multithreading using write-once result variables.
- **Cloud Haskell** is a strongly typed Erlang-like approach to distributed programming in Haskell.

Recent news

Strategies and data parallelism are already there right now, but there are new technologies currently in development as well:

- Yet more to come: the **par-monad** package allows a deterministic approach to multithreading using write-once result variables.
- **Cloud Haskell** is a strongly typed Erlang-like approach to distributed programming in Haskell.

Simon Marlow has written an excellent an up-to-date tutorial to parallel programming in Haskell:
`http://community.haskell.org/~simonmar/par-tutorial.pdf`