Functional Programming and Parallelism

John Mitchell

Reading
A Tutorial on Parallel and Concurrent Programming in Haskell
Skip Section 5 on STM

Thanks to Simon Peyton Jones, Satnam Singh, and Don Stewart for various slides.

Additional programming models

- Multilisp “futures”
  - Historical concept
  - Simple and influential

- Candidate models in Haskell
  - Explicit threads
  - Semi-implicit parallelism
  - Data parallelism

Multilisp future

- Functional definition of mergesort
  \[
  \text{(define (split x) (...))}
  \text{(define (merge x y) (... (car x) ...))}
  \text{(define (mergesort x)}
  \text{  (let ((y,z) (split x))}
  \text{    (merge (mergesort y) (mergesort z))))}
  \]

- How to rewrite this as concurrent algorithm?
  Slide example: Michael Hicks

Future: value to exist in the future

- Fork thread to compute value [Halstead 85]
  - (future e) evaluate e concurrently with parent
  - Parent thread may block when value is needed

- Benefits
  - Notationally lightweight
  - Sequential algorithm still expressed in code
  - Concurrency determined by the run-time system
  - Can be based on system resources
  - Simple coordination between threads

Where to invest in the future?

- Ineffective - result is used immediately in the following call

Where to invest in the future?

- Good - recursive calls can operate in parallel
Implementation

- `(future e)`
  - fork a new thread `T` to evaluate `e`
  - return a proxy `p` to the parent
  - called a future or promise

- **Producer**
  - Thread `T` stores result of `e` into proxy `p`

- **Consumer**
  - Run-time system gets result from `p` as needed by parent
  - Called a touch or claim
  - Consumer thread may block if "the future has not arrived"

Implementing Touches

- `(define (merge x y) ... (car x) ...)`

  - Futurized implementation of `(car x)`
    - `(if (pair? (touch x)) (get first elem of x) (error))`

  - Where `(touch x)` is
    - `(if (future? x) (get x) x)`

Optimization I

- Forking a thread per future could be expensive and without advantage
  - Particularly if not many CPUs

- Idea: only use as many threads as there are processors [Mohr et al 91]
  - At a future call, use idle thread, if any
  - Otherwise, continue using current thread
  - Save continuation on a separate queue
  - When a thread would block, save the current continuation and grab one from the queue

Optimization II

- Once a future computation completes, its result is immutable
  - Proxy and further touches redundant

- Thus
  - Use garbage collector to throw away the proxy and replace with the result [Halstead 85]
  - Avoid touching at all if static analysis can prove it's unnecessary [Flanagan & Felleisen 95]

“Futures” in Java, using join

- Join waits for thread to terminate

```
class Future extends Thread {  
  private int result;  
  public void run() { result = f(...); }  
  public int getResult() { return result; }  
}  
Future t = new future;  
t.start(); // start new thread  
t.join(); x = t.getResult(); // wait and get result  
```

Additional programming models

- MultiLisp “futures”
  - Historical concept
  - Simple and influential

- Candidate models in Haskell
  - Explicit threads
  - Semi-implicit parallelism
  - Data parallelism
Concurrency models in Haskell

- Explicit threads
  - Non-deterministic by design
  - Monadic: `forkIO` and `STM`
- Semi-implicit parallelism
  - Deterministic
  - Pure: `pseq` and `pwaq`
- Data parallelism
  - Deterministic
  - Pure: `parallel arrays`

```haskell
main :: IO ()
do {
  ch <- newChan;
  forkIO (ioManager ch);
  forkIO (worker 1 ch);
  ...
  ...
}
```

Concurrency vs Parallelism

- A concurrent program models independent agents that can communicate and synchronize.
  - Meaningful on a machine with one processor
  - Non-deterministic
- A parallel program exploits real parallel computing resources to run faster while computing the same answer.
  - Expectation of genuinely simultaneous execution
  - Deterministic

Haskell Execution Model

```
```

Functional Programming to the Rescue?

- No side effects makes parallelism easy, right?
  - It is always safe to speculate on pure code.
  - Execute each sub-expression in its own thread?
- Alas, the 80s dream does not work.
  - Far too many parallel tasks, many of which are too small to be worth the overhead of forking them.
  - Difficult/impossible for compiler to guess which are worth forking.

```
```

The par combinator

```
```

Concurrency Hierarchy

```
```
The GHC Runtime

• Multiple virtual CPUs
  – Each virtual CPU has a pool of OS threads.
  – CPU-local spark pools for additional work.
  – Work-stealing queue to run sparks.
• Lightweight Haskell threads map many-to-one onto OS threads.
  – Automatic thread migration and load balancing.
  – Parallel, generational garbage collection.

The meaning of par

• par does not guarantee a new Haskell thread
• It hints that it would be good to evaluate the first argument in parallel
• The runtime decides whether to convert spark
  – Depending on current workload
• This allows par to be very cheap
  – Programmers can use it almost anywhere
  – Safely over-approximate program parallelism

Example: One processor

Example: Two Processors

Model: Two Processors

Model: One Processor

• No extra resources, so spark for f fizzes
No parallelism?

- Main thread demands f, so spark fizzles.

Lucky parallelism

- f `par` (f + e)

A second combinator: pseq

- pseq: Evaluate x in the current thread, then return y.
- Operationally,

  \[ \text{x `pseq` y} = \begin{cases} \text{"do not terminate" if x does not terminate} \\ \text{y} \text{ otherwise.} \end{cases} \]

- With pseq, we can control evaluation order

  x `par` f `pseq` y `par` e

Using pseq

- f `par` (e `pseq` (f + e))

ThreadScope

- ThreadScope (in Beta) displays event logs generated by GHC to track spark behavior:

Sample Program

- The fib and sumEuler functions are unchanged.

- The fib and sumEuler functions are unchanged.
**Summary:**

**Semi-implicit parallelism**
- Deterministic:
  - Same results with parallel and sequential programs.
  - No races, no errors.
  - Good for reasoning: Erase the par combinator and get the original program.
- Relies on purity.
- Cheap: Sprinkle par as you like, then measure with ThreadScope and refine.
- Takes practice to learn where to put par and pseq.
- Often good speed-ups with little effort.

---

**Candidate Models in Haskell**

- **Explicit threads**
  - Non-deterministic by design
  - Monadic: `forkIO` and `STM`
- **Semi-implicit**
  - Deterministic
  - Pure: `par` and `pseq`
- **Data parallelism**
  - Deterministic
  - Pure: parallel arrays
  - Shared memory initially; distributed memory eventually; possibly even GPUs...

---

**Road map**

- **Multicore**
- **Parallel programming**
- **Task parallelism**
  - Each thread is different (MIMD)
  - Explicit threads, MVars, STM
  - Implicit: `par` & `pseq`
- **Data parallelism**
  - Operate simultaneously on bulk data (SIMD)
  - Modest parallelism
  - Hard to program
  - Massive parallelism
  - Easy to program
  - Implicit synchronisation

---

**Flat data parallel**

- Widely used, well understood, well supported
  - Established approach: Fortran, C, MPI, map/reduce
  - Limited applicability (dense matrix, map/reduce)
  - Well developed
  - Limited new opportunities
- Developed in 90’s
- Much wider applicability (sparse matrix, graph algorithms, games etc)
- Practically un-developed
- Huge opportunity

- **BUT:** Single point of concurrency
  - Individual tasks are sequential
- **Easy to implement:** use “chunking”
- **Good cost model**

---

**Flat data parallel**

```
foreach i in 1..N {
    ...do something to A[i]...
}
```

- 1,000,000’s of (small) work items
Nested data parallel

- Main idea: Allow nested tasks to be parallel

- Parallelism structure is recursive, potentially un-balanced
- Still good cost model
- Harder to implement!

Nested DP is great for programmers

- Fundamentally more modular.
- Opens up a much wider range of applications:
  - Divide and conquer algorithms (e.g. sort)
  - Graph algorithms (e.g. shortest path, spanning trees)
  - Sparse arrays, variable grid adaptive methods (e.g. Barnes-Hut)
  - Physics engines for games, computational graphics (e.g. Delauny triangulation)
  - Machine learning, optimization, constraint solving

Data Parallel Haskell

- Substantial improvement in:
  - Expressiveness
  - Performance

- Shared memory initially
- Distributed memory eventually
- GPUs anyone?

Not a special purpose data-parallel compiler! Most support is either useful for other things, or is in the form of library code.

Array comprehensions

- \[ \text{vecMul} :: [\text{Float}] \rightarrow [\text{Float}] \rightarrow \text{Float} \]
- \[ \text{vecMul } v_1 v_2 = \text{sumP } [f_1 \cdot f_2 | f_1 < v_1 \land f_2 < v_2] \]
- \[ \text{sumP} :: [\text{Float}] \rightarrow \text{Float} \]

Operations over parallel array are computed in parallel: that is the only way the programmer says “do parallel stuff.”

Sparse vector multiplication

- Sparse vector represented as a vector of (index, value) pairs:
  \[ \{(0,3),(2,10)\} \text{ instead of } [3,0,10,0]. \]
- \[ \text{sDotP} :: [(\text{Int},\text{Float})] \rightarrow [\text{Float}] \rightarrow \text{Float} \]
- \[ \text{sDotP } v = \text{sumP } [f \cdot (v!i) | (i,f) < sv] \]

Parallelism is proportional to length of sparse vector.
**Sparse matrix multiplication**

A sparse matrix is a vector of sparse vectors:

\[
\{(1,3),(4,10)\}, \{(0,2),(1,12),(4,6)\}
\]

\[
\text{smMul} :: [\{\text{Int,Float}\}] \to [\text{Float}] \to \text{Float}
\]

\[
\text{smMul} \text{ sm } v = \text{sumP} \left[ s \text{DotP} \text{ sv } v \mid \text{sv} < - \text{sm} \right]
\]

Nested data parallelism: We are calling a parallel operation, sDotP, on every element of a parallel array, sm.

**Example: Data-parallel Quicksort**

\[
\text{sort} :: [\text{Float}] \to [\text{Float}]
\]

\[
\text{sort a} = \begin{cases} 
\text{a} & \text{if } (\text{lengthP } a \leq 1) \\
\text{else } \text{sal}0 \maplus \text{eq} \maplus \text{sal}1 
\end{cases}
\]

where

\[
\begin{align*}
\text{p} &= \text{a}0 \\
\text{lt} &= [f \mid f < - \text{a}, f \neq \text{p}] \\
\text{eq} &= [f \mid f < - \text{a}, f = \text{p}] \\
\text{gr} &= [f \mid f < - \text{a}, f > \text{p}] \\
\text{sa} &= [\text{sort a} \mid a < [\text{lt}, \text{gr}]]
\end{align*}
\]

2-way nested data parallelism

**How it works**

Step 1

Step 2

Step 3

...etc...

- All sub-sorts at the same level are done in parallel.
- Segment vectors track which chunk belongs to which sub problem.
- Difficult to simulate easily by hand

**Example: Parallel Search**

\[
\begin{align*}
\text{type } \text{Doc} &= [\text{String}] \\
\text{type } \text{Corpus} &= [\text{Doc}] \\
\text{search} :: \text{Corpus} \to [\text{String}] \to [\{\text{Doc,Int}\}]
\end{align*}
\]

\[
\text{wordOccs} :: \text{Doc} \to [\text{String}] \to [\text{Int}]
\]

Find all Docs that mention the string, along with the places where it is mentioned (e.g. word 45 and 99)

---

**Example: Parallel Search**

\[
\begin{align*}
\text{type } \text{Doc} &= [\text{String}] \\
\text{type } \text{Corpus} &= [\text{Doc}] \\
\text{search} :: \text{Corpus} \to [\text{String}] \to [\{\text{Doc,Int}\}]
\end{align*}
\]

\[
\text{wordOccs} :: \text{Doc} \to [\text{String}] \to [\text{Int}]
\]

Find all the places where a string is mentioned in a document (e.g. word 45 and 99).
Example: Parallel Search

```haskell
type Doc = [: String :) type Corpus = [: Doc :]

search :: Corpus -> String -> [: (Doc [: Int :]) :]

wordOccs :: Doc -> String -> [: Int :]
wordOccs d s = [: i | (i,si) <- zipP positions d , s == si :]
where
  positions :: [: Int :]
  positions = [: 1..lengthP d :]
```

Hard to implement well!

Evenly chunking at top level might be *ill-balanced*
Top level alone might not be very parallel

The flattening transformation

Concatenate sub-arrays into one big, flat array.
Operate in parallel on the big array.
Segment vector tracks extent of sub-arrays.

```
<example diagram>
```

Lots of tricky book-keeping!
Possible to do by hand (and done in practice),
but very hard to get right.
Blelloch showed it could be done systematically.

Implementation Techniques

Four key pieces of technology:

- Vectorization
  - Specific to parallel arrays
- Non-parametric data (array) representation
  - A generically useful new feature in GHC
- Distribution
  - Divide up the work evenly between processors
- Aggressive fusion
  - Eliminate synchronizations
    - Uses "rewrite rules," an old feature of GHC

Haskell advance: an optimizing data-parallel compiler implemented
by modest enhancements to a full-scale functional language implementation.

Purity pays off

- Two key transformations:
  - Flattening
  - Fusion
- Both rely on purely-functional semantics:
  - No assignments.
  - Every operation is pure.

Prediction: The data-parallel languages of the future will be functional languages

Promising performance

1-processor version goes only 30% slower than C
Perf win with 2 processors
Pinch of salt

```
<example diagram>
```
Data Parallel Summary

- Data parallelism is the most promising way to harness 100's of cores.
- Nested DP is great for programmers: far, far more flexible than flat DP.
- Nested DP is tough to implement, but progress on how to do it.
- Functional programming is a massive win in this space.
- Work in progress: starting to be available in GHC 6.10 and 6.12.

http://haskell.org/haskellwiki/GHC/Data_Parallel_Haskell

Programming models

- Multilisp “futures”
  - Historical concept
  - Simple and influential
- Candidate models in Haskell
  - Explicit threads
  - Semi-implicit parallelism
  - Data parallelism

Concurrency Models in Haskell

- Explicit threads
  - Non-deterministic by design
  - Monadic: forkIO and STM
- Semi-implicit
  - Deterministic
  - Pure: par and pseq
- Data parallelism
  - Deterministic
  - Pure: parallel arrays
  - Shared memory initially; distributed memory eventually; possibly even GPUs...

Main

```haskell
main :: IO ()
    = do (ch <- newChan)
         ; forkIO (ioManager ch)
         ; forkIO (worker 1 ch)
         ...
    ...
```

F : Int -> Int

```haskell
f x = a `par` b `pseq` a + b
    where
        a = f1 (x-1)
        b = f2 (x-2)
```

Grand Challenge

- Making effective use of multicore hardware is leading challenge for programming languages
- Hardware is getting increasingly complicated:
  - Nested memory hierarchies
  - Hybrid processors: GPU + CPU, Cell, FPGA...
  - Massive compute power sitting mostly idle.
- We need new programming models to program new commodity machines effectively.
- Language researchers are working hard to answer this challenge...

Note: last year’s slides have many more implementation details