Erlang and the Parallella

- Edward Tate, Erlang Solutions (3 years)
- ParaPhrase Research Group (1 year)
- My job: Finding new ways of expressing problems which contain parallelism on the Erlang VM
What this talk is about

- Erlang and its concurrency model
- Explorations into parallel programming in general (warning, crazy stuff ahead)
- Why the Parallella is of great interest to us
- Our experience in using Erlang in conjunction with the Parallella
- Briefly OpenCL (not going into detail here)
What is Erlang?

- Erlang is a language developed initially at Ericsson and was designed to be distributed and fault tolerant.
- Distributed in the sense that an Erlang node can communicate with any other Erlang node on the network without having to do any work fiddling with protocols.
- Ericsson needed a language to program telephone switches, they wanted 0 down time.
What is Erlang?

- Erlang ended up being a concurrent language where processes and message passing are core primitives.
- Processes are useful representations of entities which can receive and send messages to one another.
- Processes operate concurrently, and are assigned to run queues on manycore machines, giving us parallelism.
What does Erlang look like?

- Like Prolog
  
  \[
  \text{square}(X) \rightarrow X \ast X \text{ end.}
  \]

- Erlang has pattern matching, which is useful in dealing with protocols

- Matching network packets is cool
What does Erlang look like?

%%%% inevitable fib example
fib(N) when N <= 1 ->
    N;
fib(N) ->
    fib(N - 1) + fib(N - 2).
%% bit syntax
parse_message(M) ->
    <<X:2, Y:2>> = M, M.

parse_message(<<X:2, Y:2>> = M)->
    M.

%% list comprehensions
[X || X ← [1,2,3]].
Erlangs concurrency primitives

- **Spawn**
  
  Pid = spawn(fun () → ok end).

- **Send**
  
  Pid ! Message.

- **Receive**
  
  receive Message → Message end.
Erlangs Concurrency Abstractions

- Erlang OTP contains behaviours which are useful in developing concurrent code
- A behaviour is a generic abstraction which provides functions to be implemented
- They are like typeclasses in Haskell without any type enforcement (Erlang has no type system!)
- gen_server defines a generic server pattern, can be implemented for web servers or any other type of server
Explicit Concurrency in Erlang

- When the problem is process oriented, Erlang's explicit concurrency primitives fit the bill
- Web servers / Game servers
- Messaging systems
- Distributed systems
- What about matrix mult?
Explicit Parallelism in Erlang

- Erlangs primitives can be used to obtain parallelism
- Some code
Explicit Parallelism in Erlang

map(F, [], Lim) →
  receive_n(Lim);
map(F, [X|Xs], Lim) →
  P = self(),
  Q = spawn(fun () → P ! F(X) end),
  map(F, Xs, Lim).
Explicit Parallelism in Erlang

map

Root

F(X1)  F(X2)  F(X3)  F(Xn)
Implicit Parallelism

- We can also express parallelism implicitly, though not in Erlang.
- Regardless of how careful you are, debugging race conditions is hard.
- We realised there was another way early on in the project and wanted to have a go at it.
- Do not panic in the code to follow :)
Three forms of parallelism: Gentle Introduction

- Functional Parallelism
- Functions which depend on independent data

```latex
var X = fby.time 1 (f X)
var Y = fby.time 0 (g Y)
var Z = X + Y
```

- If Y here depended on X, (\texttt{var Y = g X}) our functions would execute sequentially
Three forms of parallelism: Gentle Introduction

- Functional Parallelism
- Functions which depend on independent data

```plaintext
var X = fby.time 1 (f X)
var Y = fby.time 0 (g Y)
var Z = X + Y
```
- If Y here depended on X, (var Y = g X) our functions would execute sequentially.
Three forms of parallelism

- Pipeline parallelism
  
  ```
  var X = fby.time 1 (X+1)
  var Y = f X
  var Z = g Y
  ```

- Whilst Y is computing `f X @ [t ← 2]`, Z can commence computing `g Y @ [t ← 1]`
Three forms of parallelism

- Spatial parallelism, also known as data parallelism

```
Z
where
  dim i ← 0
  dim j ← 0
  var Z = X + Y
end

Z @ [i ← 1..n, j ← 1..m]
```
In case you hadn't noticed

- The language used in coding the implicit parallelism examples is Lucid
- Lucid was developed in the 70s, but has since been mostly forgotten
- It was known as a dataflow language at the time, but is now known as an 'intensional' language
- Functional languages can be written to be completely parallel too!
Wouldn't this be better?

fun increment X = X + 1

fun xs.d f n X =
  (f X) @ [d ← 1..n]

fun map.d f n X = Y
where
  dim t ← 0
  var Y = fby.t X (xs.d f n Y)
end
Intensional Parallelism #1

• You can then ask for 10 values to be incremented across a spatial dimension at the same time

    dim space ← 0
    var X = 1

    map.space increment 5 X @ [t ← 1]

• Result:
    X @ [t ← 1] = 5
    X @ [t ← 2] = 5 etc

• Where the values of X were computed in parallel across the dimension 'space'
Intensional Parallelism #2

map.d increment 5 1 @ [t ← 1]
How does the program work?

- Very similar to explicit map example
- Difference is mainly that in the intensional example we can choose how many values can be computed across an arbitrary space dimension
- The functional example is list oriented
How does that program actually work?

- Every variable represents an infinite stream which may vary across one or more dimensions.
- In this particular example, we are defining a map function which varies across the t or time dimension, and asking for, in parallel, all of n values applied to f simultaneously in the space dimension.
- The parallelism here is implicit, no spawns!
What the heck did that mean?

- ESL has been developing a language for expressing parallelism
- We want to get small expressive programs which run fast onto the Erlang VM
- So that we can all benefit from having fast numeric operations, and a scheduler designed for high throughput
Why another new language?

- We only figured out in 2013 how to use multidimensionality to solve parallel problems
- Lucid has existed for ages
- GLU (Granular Lucid) was created in the 90s but implementation was lost
- GLU was massively parallel but based on old Lucid semantics
What about Erlang vanilla?

- Erlang's behaviours do not help us since they do not target the parallel programming domain.
- E.g.: There is no generic behaviour for matrix mult.
- Erlang has no floating point numbers.
- Erlang's primitives allow us to write programs which operate in parallel, but do not help us with solving the problem at hand.
How does Erlang help us where Parallelism is concerned?

- Explicit concurrency primitives can be thought of as the assembly language of parallel programs.
- The aspect of Erlang which is specifically useful for parallelism is the work stealing aspect of the scheduler.
- The Erlang scheduler knows about the cores on your system and will steal work items from the run queues at distinct cores.
About that... Erlangs Scheduler?
About that... Erlang's Scheduler?

- The Erlang scheduler was designed for low latency, not raw throughput.
- A scheduler which is tailored solely for throughput will look different than one designed for low latency (Cilk / Haskell vs Erlang).
- Some devices do not benefit from software level schedulers at all (those that can take advantage of the hardware directly, i.e., the GPU).
About that... Erlangs Scheduler?

- Fairness helps attain low latency by causing many small context switches, which is great.
- The way this works is that every process is given a set number of reductions, when these reductions are up the process is put back in the run queue and the next process is put to work.
- Load balancing abstractions, process migration slows us down.
What about offloading processes?

- Offloading Erlang processes to the Parallella is interesting and will be researched as part of ParaPhrase in the near future.
- We're not there yet :)
- We'd rather map Lucid expressions to a purely work stealing scheduler first, since it is simpler to express parallel problems this way.
- Then offer an alternate scheduler for Erlang which may be a better fit for the Parallella.
ParaPhrase

• The ParaPhrase group came up with a few ideas too, to make parallel programming easier
  - Pskel (pattern based skeleton framework)
  - Refactoring to pskel patterns
ParaPhrase: PSkel

- Pskel is a pattern / skeleton framework for Erlang which allows developers to express problems as a set of provided patterns.
  - Map (split / execute / merge)
  - Farm (execution of diff tasks by same farm)
  - Pipe
  - D&C
  - Etc
ParaPhrase: PSkel - Map

• A map consists of applying a function F in parallel over some data (split / merge / execute)
  pskel:map(F, Data).

• A farm applies a function F over the same data in parallel
  pskel:farm(F, Data).
ParaPhrase: PSkel

• PSkel is useful if one wishes to use generic parallel patterns in order to solve one's problems.
• Refactoring helps to expose latent parallelism in the program.
Why Intensional and not functional / parallel patterns?

- Intensional languages have primitives which operate upon dimensions, making it much easier to deal with parallelism.
- You can just say that at a specific point in time, N number of things should be done across another dimension: voila, parallelism.
- Multi dimensionality is present in OpenCL too (well, across 3 dimensions anyhow).
Functional vs Intensional

- You can have both in one language!
- We didn't know how to implement functions in Lucid until 2012
- Functions in Lucid do not close over any kind of environment, the values of variables are disjoint from the environment
Where does OpenCL fit in

- The first offloading demo we built was an audio compression server
- Received data from N nodes
- Executed a Haar wavelet transform on the data
- Sent result to output nodes
OpenCL

- We used OpenCL to offload heavy duty matrix computations
- OpenCL is great where GPUs are concerned since the API maps directly to the hardware
- With the Parallela, the OpenCL runtime must be responsible for subdividing (or scheduling) work across cores somehow
- Writing OpenCL kernels is hard :(
The Parallela

• We had 1 month to come up with something that ran on the Parallela machine that was kindly provided to us
• Since we had used OpenCL before, we wanted to continue using it on the Parallela in order to research its capabilities
What was the demo about?

- The demo was to capture video frames from Raspberry Pi machines which were hooked up to cameras and process them using a Parallela OpenCL node.
- We decided to do sobel edge detection in order to get real-time output.
Demo
Issues Encountered

• The biggest issue we encountered whilst developing on the Parallela was the lack of atomic primitives in OpenCL

• Most of our existing image processing kernels used atomic operations pervasively

• Mutexes were mentioned by devs but I had no clue as to how to get them to work in OpenCL
Inevitable Pic
Conclusions

- Next time we will do our own scheduling, cos' its more fun ;)
- We want to map our Lucid dialect to the Parallela and the Tilera in the near future to gain the ability to express parallelism in a declarative way
Questions?