A brief survey of functional parallelism

Matt Might
matt.might.net
@mattmight
What is functional programming?
Functional programming

• First-class functions
• Lexical closures
• Immutability / purity
• Equational reasoning
• Lazy evaluation
\lambda\text{-calculus}
\( \lambda \text{-calculus (Church, 1928)} \)
λ-calculus (Church, 1928)

- Minimalist, universal language

Alonzo Church
\(\lambda\)-calculus (Church, 1928)

- Minimalist, universal language
- Three expression types:
  
  - \(v\) [variable]
\( \lambda \text{-calculus (Church, 1928)} \)

- Minimalist, universal language
- Three expression types:
  - \( v \) [variable]
  - \( e_1(e_2) \) [function application]
\(\lambda\)-calculus (Church, 1928)

- Minimalist, universal language
- Three expression types:
  
  \(\nu\) [variable]

  \(e_1(e_2)\) [function application]

  \(\lambda \nu. e\) [anonymous function]
Lisp and Scheme

- \( v \equiv v \)

- \( f(e) \equiv (f \ e) \)

- \( \lambda v. e \equiv (\text{lambda} \ (v) \ e) \)
Python

- $v \equiv v$
- $f(e) \equiv f(e)$
- $\lambda v. e \equiv \text{lambda } v: e$
Ruby

- $v \equiv v$
- $f(e) \equiv f(e)$
- $\lambda v. e \equiv \text{lambda } \{ \mid v \mid \text{return e} \}$
JavaScript

- $v \equiv \text{v}
- f(e) \equiv f(e)
- \lambda v.e \equiv \text{function (v) { return e ; }}
Java

- $v \equiv v$

- $f(e) \equiv f.call(e)$

- $\lambda v.e \equiv \text{new Value} () \{ \text{public Value call (Value v) \{ return e \; \}} \} \}$
\( \lambda \text{-fortified} \)

- Lisp
- SML
- Haskell
- Scala
- Java
- C#
- C++
- Python
- Ruby
- Smalltalk
- JavaScript
- PHP(!)
Today’s menu

- Parallel iteration
- Futures
- Speculation
- Channels
- Transactions
- Autoparallelization
Parallel iteration
for (int i = 0; i < length; i++) {
    ...
    ...
}
for (int i = 0; i < length; i++) {
    ...
}
b = new T[a.length] ;
for (int i = 0; i < a.length; i++) {
    b[i] = f(a[i]) ;
}
\[ b = \text{map} \ f \ a \]
b = map (x => x + 1) a
(pmap f a)
\[ \sum_{i=1}^{8} a_i \]
\[ a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8 \]
c = 0;
for (int i = 0; i < a.length; i++) {
    c += a[i];
}

Thursday, November 18, 2010
\((((((a_1 + a_2) + a_3) + a_4) + a_5) + a_6) + a_7) + a_8\)
\((a_1 + a_2) + (a_3 + a_4)) + ((a_5 + a_6) + (a_7 + a_8))\)
reduce \( (\oplus) \vec{a} = a_1 \oplus \cdots \oplus a_n \)
\[ \sum = \text{reduce} \ (+) \]
reduce
assumes
associativity
Futures
a = f(x) ;
b = g(y) ;
return a + b;
pidf = spawn(f,&a,x) ;
pidg = spawn(g,&b,y) ;
join(pidf) ;
join(pidg) ;
return a + b ;
let a = f(x)  
    b = g(y)  
in a + b
let a = future f(x)
    b = future g(y)
in a + b
let a = future f(x)
b = future g(y)
in h(a,b)
Speculation
immutability

= 

no dependence
laziness = things to do
\( f(x, g(y)) \)
\( f(a,b) = \)
\[
\text{let } z = \text{BB}(a) \\
\text{in } z + b
\]
Explicit speculation
par n f(x,n)
where n = g(y)
\[ f(a,b) = \par b \]
\[
(\text{let } z = \text{BB}(a) \\
\text{in } z + b)
\]
Channels
Hard to reason about shared state.
Message-passing is reasonable.
\Pi-calculus

\[ P, Q ::= P | Q \]
\[ \quad | \text{let } v = c.\text{receive()} \text{ in } P \]
\[ \quad | c.\text{send(exp)}; P \]
\[ \quad | \text{let } c = \text{new Channel()} \text{ in } P \]
\[ \quad | !P \]
\[ \quad | P + Q \]
\[ \quad | \text{halt} \]
Erlang example

ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished~n", []).

ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).
Erlang example

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.
Erlang example

```erlang
start() ->
    Pong_PID = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_PID]).
```
Is message-passing efficient?
If data is immutable, message-passing can be implemented as shared memory.
Transactions
Locks are bad.
lock (mylock) ;
// do things
unlock (mylock) ;
atomic {
    // do things
}

Thursday, November 18, 2010
atomic {
    x += 1
}

atomic {
    x += 1
}
atomic {
    x += 1
}

atomic {
    x += 1
}
**imperative**: mutable by default

**functional**: immutable by default
Automatic parallelization
f() ;

g() ;
Dependence analysis
Dependence analysis
Dependence analysis
Dependence analysis

What resources are written?

What resources are read?

Which calling contexts are live on stack?
Context-sensitive dependence graphs
Context-sensitive dependence graphs

Resources \rightarrow v \text{ bound in } k \rightarrow f \text{ called in } k' \leftarrow \text{ Calls}
f() \parallel g()
Questions?

matt.might.net
@mattmight