Functional Programming and Parallel Computing

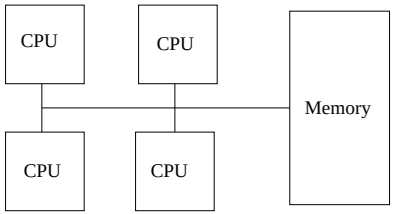
Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

bjorn.lisper@mdh.se
http://www.idt.mdh.se/~blr/

Parallel Processing

Multicore processors are becoming commonplace

They typically consist of several processors, and a shared memory:



The different cores execute different pieces of code in parallel

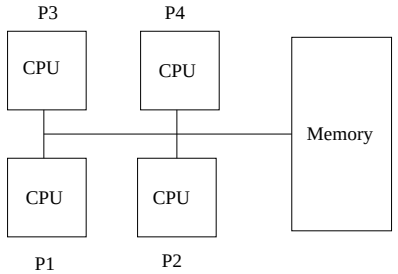
If several cores do useful work at the same time, the processing becomes faster

Imperative Programming and Parallel Processing

Imperative programming is inherently sequential

Statements are executed one after another

However, imperative programs can be made parallel through parallel processes, or threads:



Side Effects and Parallel Processing

Imperative programs have side effects

Variables are assigned values

These variables may be accessed by several processes

This introduces the risk of *race conditions*

Since processes run asynchronously on different processors, we cannot make any assumptions about their relative speed

In one situation one process may run faster, in another situation another one runs faster

A Race Condition Example

Two processes P1 and P2, which both can write a shared variable tmp:

```
P1:              P2:
 .               tmp = 4711
 .               z = tmp + 3
 .                .
tmp = 17          .
y = 2*tmp         .
```

The intention of the programmer was that both P1 and P2 should set tmp and then use its newly set value right away. This would yield the following, intended, final contents of y and z:

$y = 34$, $z = 4714$

But the programmer missed that the processes have a race condition for tmp, since they may run at different speed. Here is one possible situation:

```
P1:              P2:
tmp = 17          .
 .               tmp = 4711
y = 2*tmp         .
 .               z = tmp + 3
```

Final state: $y = 9422$, $z = 4714$

Wrong value for y!

Here is another possible situation:

```
P1:              P2:
 .               tmp = 4711
tmp = 17          .
 .               z = tmp + 3
y = 2*tmp         .
```

Final state: $y = 34$, $z = 20$

Wrong value for z!

Parallel Programming is Difficult

To avoid these race conditions, the programmer must use different synchronization mechanisms to make sure processes do not interfere with each other

But this is difficult! It is very easy to miss race conditions

Debugging of parallel programs is also difficult! Race conditions may occur only under certain conditions, that appear very seldom. It can be very hard to reproduce bugs

This is a very bad situation, since multicore processors are becoming commonplace. We are heading for a software crisis!

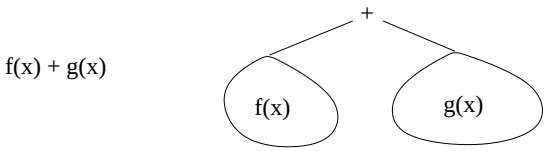
The heart of the problem is the side effects – they allow different processes to thrash each others' data

Pure Functional Programs and Parallel Processing

Pure functional programs have no side-effects

The evaluation order does not matter

Thus, different parts of the same expression can always be evaluated in parallel:



Sometimes called *expression parallelism*

Parallelism in Collection-Oriented Primitives

Data structures like lists, arrays, sequences, are sometimes called *collections*

Functions like `map` and `fold` are called *collection-oriented*

Collection-oriented functions often have a lot of inherent parallelism

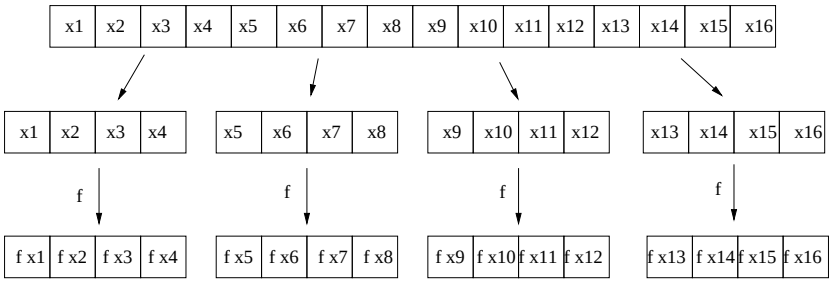
If one can express computations with these primitives, then parallelization often becomes easy

This parallelism is often called *data parallelism*

In imperative programs these computations are often implemented by loops. Loops are sequential. A good parallelizing compiler might retrieve some of it, but there is a risk that parallelism is lost

Map on Arrays

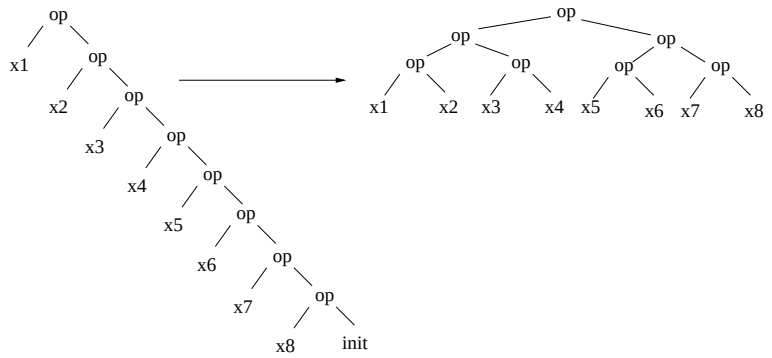
Map is very parallel:



With sufficiently many processors, `map` can be done in $O(1)$ time

Parallel Fold

Fold can be parallelized *if the binary function is an associative operator*:

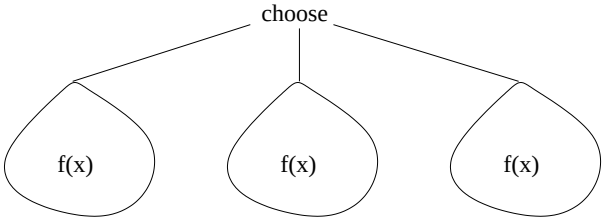


If `op` is associative, then the expression tree can be balanced

With sufficiently many processors, parallel `fold` can be done in $O(\log n)$ time ($n =$ no. of elements)

Fault Tolerance Through Replicated Evaluation

If an expression has no side effects, then it can be evaluated several times without changing the result of the program



This can be used to increase the *fault tolerance*: if one processor fails, we can use the result from another one computing the same expression

Parallelism in F#

F# has some support for parallel and concurrent processing:

The `System.Threading` library gives threads

A data type `Async<'a>` for asynchronous (concurrent) workflows (a kind of computation expressions)

The `System.Threading.Tasks` library yields task parallelism

`Array.Parallel` module provides data parallel operations on arrays

More on this in Section 13 in the book

Example: Expression Parallelism using Parallel Tasks

Evaluating `f x` and `g x` in parallel when computing `(f x) + (g x)`:

```
Open System.Threading.Tasks

let h x =
  let result1 = Task.Factory.StartNew(fun () -> f x)
  let result2 = Task.Factory.StartNew(fun () -> g x)
  result1 + result2
```

Example: Data Parallel Search for Primes

Create a big array, then map a test for primality over it to be done in parallel

Assume a predicate `isPrime : int -> bool` that tests whether its argument is a prime

```
let bigarray = [|1 .. 500000|]

Array.Parallel.map isPrime bigarray
```


Summing Up

Freedom from side effects simplifies parallel processing a lot

Collection-oriented operations are also very helpful for this

It's the design and thinking that is important – not necessarily that a functional language is used

The same principles can be applied also when using conventional programming languages