Parallel programming

Johan Montelius

KTH

VT23

Parallelism

Parallelism vs Concurrency

Concurrency:

- multiple threads of control
- structure of architecture
- particularly suited for interactive applications

Parallelism:

- main goal - increase performance
- make use of parallel hardware

A concurrent program could be parallelized.

types of parallel hardware

- Multiple Instructions Multiple Data (MIMD) : what is provided by a multi-core CPU but also by a distributed system
- Single Instruction Multiple Data (SIMD): typically used by graphics cards, the same operations should be performed but on many objects or pixels
- Pipeline : processing units that work in series, for example the stages of execution of an instruction in a CPU

We will try to utilize a MIMD architecture.

types of parallel programming

Several models of parallel computations:

- loop parallelism: identify a loop where each iteration is independent
- map-reduce: for each element in a set - perform an operation and collect the result
- task parallelism: independent tasks are generated and executed in parallel
- stream parallelism: a stream of events should be processes by several combinators

A concurrent program could be executed in parallel but the focus is then concurrency not parallelism.

language support

What language support do we have:

- parallel operators: extraction of parallelism by compiler, loop parallelism, map-reduce etc
- concurrency: concurrent processes that the can be execute in parallel
  - synchronization by shared memory/objects - Java, C++, ...
  - synchronization by message passing - Erlang/Elixir, Go, MPI, ...

how do we exploit parallelism

```
def fib(0) do 1 end
def fib(1) do 1 end
def fib(n) do
   f1 = fib(n-1)
   f2 = fib(n-2)
   f1 + f2
end
```

```
def fib(0) do 1 end
def fib(1) do 1 end
def fib(n) do
   f1 = spawn(fn() -> fib(n-1) end)
   f2 = spawn(fn() -> fib(n-2) end)
   f1 + f2
end
```

Ehhh, not the best thing to do - does not work, does it?

how do we exploit parallelism

```
def fib(0) do 1 end
def fib(1) do 1 end
def fib(n) do
   r1 = make_ref()
   r2 = make_ref()
   parallel(fn() -> fib(n-1) end, r1)
   parallel(fn() -> fib(n-2) end, r2)
   f1 = collect(r1)
   f2 = collect(r2)
   f1 + f2
end
```

```
def parallel(fun, ref) ->
   self = self()
   spawn(fn() ->
     res = fun.()
     send(self, {:ok, ref, res})
     end)
end

def collect(ref) do
   receive  do
      {:ok, ^ref, res} ->
          res
   end
end
```

All right, let's roll!

benchmark

fib(30), parallel vs sequential, 2 x AMD Opteron 12 cores

- sequential: 64 ms
- parallel: .... 1800 ms

so much for parallelism

granularity

We need to control the granularity of tasks:

```
def fix(0, _) do 0 end
def fix(1, _) do 1 end
def fix(n, m) when n > m do
   r1 = make_ref()
   r2 = make_ref()
   parallel(fn() -> fix(n-1, m) end, r1)
   parallel(fn() -> fix(n-2, m) end, r2)
   f1 = collect(r)
   f2 = collect(r2)
   f1 + f2
end
def fix(n, _) do fib(n) end
```

All right, let's roll!

finding the granularity

```
> Fib.bench(40,20)
fib(40)  sequential: 7000 ms
fix(40,38) : 2900 ms
fix(40,36) : 1100 ms
fix(40,34) : 610 ms
fix(40,32) : 530 ms
fix(40,30) : 490 ms
fix(40,28) : 490 ms
fix(40,26) : 480 ms
fix(40,24) : 480 ms
fix(40,22) : 480 ms
fix(40,20) : 490 ms
ok
```

When does it pay off, what is the overhead?

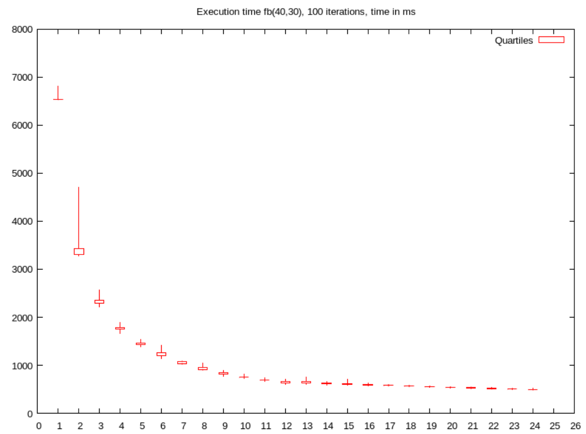
the null test

```
:
fix(n,_) do 1 end
```

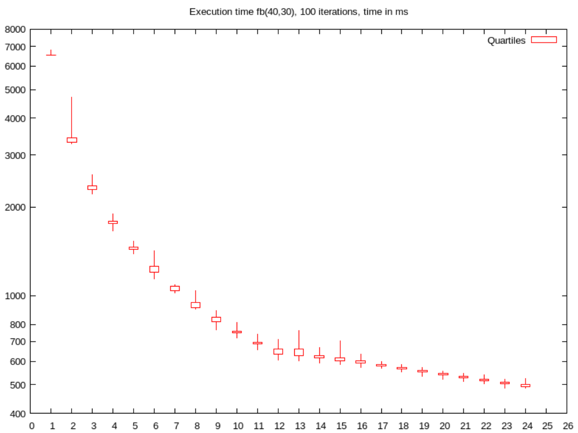
```
> Fib.bench(40,20)
fib(40)  sequential: 7000 ms
fix(40,38) : 0 ms
fix(40,36) : 0 ms
fix(40,34) : 0 ms
fix(40,32) : 0 ms
fix(40,30) : 1 ms
fix(40,28) : 1 ms
fix(40,26) : 3 ms
fix(40,24) : 6 ms
fix(40,22) : 17 ms
fix(40,20) : 32 ms
ok
```

How many processes are created in fix(40,20)?

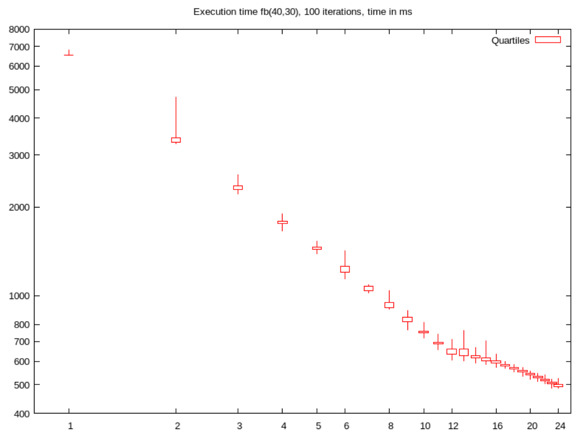
how do we scale



log scale



log-log scale



as good as it gets

- speed-up 1 - 2 cores : 1.9
- speed-up 2 - 4 cores : 1.9
- speed-up 4 - 8 cores : 1.9
- speed-up 6 - 12 cores : 1.9
- speed-up 12 - 24 cores : 1.3

Calculating Fibonacci in parallel is an example of an “embarrassingly easy parallel program”.

Image processing

Assume that we want to transform an image to a gray scale, and then reduce the color depth of the image.

question

How do we parallelize this?

alternatives

- Parallelize the gray transformer and/or the depth transformer, or
- let the gray transformer feed the color-depth transformer, line by line.

A pipe-line: reader - transform - transform - writer

the pipeline

```
def stream() do
  writer = PPM.writer("reduced.ppm", self())
  reducer = Stream.start(gray_reduce(), writer)
  grayer = Stream.start(rgb_to_gray(), reducer)
  PPM.reader("hockey.ppm", grayer)
  receive do
   :done
  end
end
```


a transformer

a transformer

benchmark

```
> Test.batch()
reading in  118 ms
gray in      66 ms
reduce in    66 ms
writing in   96 ms

total in    349 ms
```

This is using only one scheduler.

```
> Test.stream()

 total in 260 ms
```


benchmark

```
> Test.stream()
reading turning gray, reducing and writing in 260 ms
```

```
> :erlang.system_flag(:schedulers_online, 2)
1
```

```
> Test.stream()
reading turning gray, reducing and writing in 161 ms
```


a sequence of task

Assume we have a sequence of independent task (for example images that should processes) how do we parallelize the execution?

- pipe-line, each task passes a sequence of processes
- task parallel, each task is executed in a separate process

Pros and cons?

stream parallelism

Assume we have a flow of events (a twitter feed) and collect statistics of the most frequent word during a minute, these words are then forwarded to a counter etc.

Create a network of processes, each process receives events, processes them and forwards them to other processes.

Apache Storm.

Summary

- parallelism vs concurrency
- concurrency as a tool for parallelism
- embarrassing easy parallelism is often easy
- pipe-line parallelism
- task parallelism
- stream parallelism