Topics: Parallel Languages

We now have a look at some languages that were designed specifically with parallelism in mind

Topics: Parallel Languages

We now have a look at some languages that were designed specifically with parallelism in mind

- Occam (channels)
- Erlang (explicit parallelism)
- Go (explicit parallelism)
- Rust (explicit parallelism)
- SISAL (implicit parallelism)
- Strand (declarative)

Picked pretty much at random: by no means an exhaustive or even comprehensive list, many other languages exist

Occam

Occam was a language that was based on *Communicating Sequential Processes* (CSP) a theoretical model of parallel computation: a *process algebra* (c.f., Lambda Calculus)

Occam

Occam was a language that was based on *Communicating Sequential Processes* (CSP) a theoretical model of parallel computation: a *process algebra* (c.f., Lambda Calculus)

CSP models processes that communicate by passing messages between themselves along *channels*

Occam

Occam was a language that was based on *Communicating Sequential Processes* (CSP) a theoretical model of parallel computation: a *process algebra* (c.f., Lambda Calculus)

CSP models processes that communicate by passing messages between themselves along *channels*

In the algebra there are various rules on combining processes and descriptions on how these combined objects behave

Occam

Occam was a language that was based on *Communicating Sequential Processes* (CSP) a theoretical model of parallel computation: a *process algebra* (c.f., Lambda Calculus)

CSP models processes that communicate by passing messages between themselves along *channels*

In the algebra there are various rules on combining processes and descriptions on how these combined objects behave

Then theoreticians get busy on proving that behaviours of various systems are equivalent (or not)

A note on channels

The channel concept is quite simple and so appears in many languages and systems

A note on channels

The channel concept is quite simple and so appears in many languages and systems

You put data in one end, it comes out the other end

A note on channels

The channel concept is quite simple and so appears in many languages and systems

You put data in one end, it comes out the other end

The simplicity is probably why it appears in so many guises

A note on channels

The channel concept is quite simple and so appears in many languages and systems

You put data in one end, it comes out the other end

The simplicity is probably why it appears in so many guises

They are good for structuring your code

A note on channels

The channel concept is quite simple and so appears in many languages and systems

You put data in one end, it comes out the other end

The simplicity is probably why it appears in so many guises

They are good for structuring your code

But channels are as fast or slow as the underlying mechanism, e.g., network messages in MPI or shared memory in shared memory machines. They can't magic away the cost of communications

Occam

Occam was a realisation of CSP, designed hand-in-hand with the hardware it would run on: the *transputer*

Occam

Occam was a realisation of CSP, designed hand-in-hand with the hardware it would run on: the *transputer*

The transputer (early 1980s) was going to be the future of parallel processing: a new hardware architecture explicitly supporting message passing between cores

Occam

Occam was a realisation of CSP, designed hand-in-hand with the hardware it would run on: the *transputer*

The transputer (early 1980s) was going to be the future of parallel processing: a new hardware architecture explicitly supporting message passing between cores

Unfortunately, the level of technology of the time was not really up to the task: they had problems with clock speeds and heat management

Occam

Occam was a realisation of CSP, designed hand-in-hand with the hardware it would run on: the *transputer*

The transputer (early 1980s) was going to be the future of parallel processing: a new hardware architecture explicitly supporting message passing between cores

Unfortunately, the level of technology of the time was not really up to the task: they had problems with clock speeds and heat management

There was no real advantage to using a transputer over existing, classical processors (like Intel), so it never managed to sell in numbers large enough to be successful

Occam

Occam was a realisation of CSP, designed hand-in-hand with the hardware it would run on: the *transputer*

The transputer (early 1980s) was going to be the future of parallel processing: a new hardware architecture explicitly supporting message passing between cores

Unfortunately, the level of technology of the time was not really up to the task: they had problems with clock speeds and heat management

There was no real advantage to using a transputer over existing, classical processors (like Intel), so it never managed to sell in numbers large enough to be successful

But the transputer was designed primarily to run Occam

Occam

Occam has explicit parallelism of tasks:

```
PAR
  f(x)
  g(y)
```

runs `f` and `g` concurrently

Occam

Occam has explicit parallelism of tasks:

```
PAR
  f(x)
  g(y)
```

runs `f` and `g` concurrently

More unusually, Occam has explicit sequentiality:

Occam

Occam has explicit parallelism of tasks:

```
PAR
  f(x)
  g(y)
```

runs `f` and `g` concurrently

More unusually, Occam has explicit sequentiality:

```
SEQ
  f(x)
  g(y)
```

Occam

Occam has explicit parallelism of tasks:

```
PAR
  f(x)
  g(y)
```

runs `f` and `g` concurrently

More unusually, Occam has explicit sequentiality:

```
SEQ
  f(x)
  g(y)
```

runs `f`, then `g`

Occam

Occam has explicit parallelism of tasks:

```
PAR
  f(x)
  g(y)
```

runs `f` and `g` concurrently

More unusually, Occam has explicit sequentiality:

```
SEQ
  f(x)
  g(y)
```

runs `f`, then `g`

This is because in CSP sequential composition of code is of equal note to parallel composition of code

Occam

Communication between processes is via channels

```
ch ! x
```

writes the value of `x` down the channel named `ch`

```
ch ? y
```

reads a value into `y` from the channel named `ch`

Occam

Communication between processes is via channels

```
ch ! x
```

writes the value of `x` down the channel named `ch`

```
ch ? y
```

reads a value into `y` from the channel named `ch`

Both are blocking: the write will wait for the corresponding read; the read will wait for the corresponding write

Occam

Thus we get communication and synchronisation between threads

```
INT x:
CHAN INT ch:
PAR
  SEQ
    print("hello")
    ch ! 42
  SEQ
    ch ? x
    print(" world")
```

will print `"hello world"`

Occam

There is also non-deterministic choice

```
ALT
  in1 ? x
    SEQ
      x := x+1
      out1 ! x
  in2 ? x
    SEQ
      x := x-1
      out2 ! x
```

will wait until data arrives on channel `in1` or `in2` and will then execute the relevant section of code

Occam

There is also non-deterministic choice

```
ALT
  in1 ? x
    SEQ
      x := x+1
      out1 ! x
  in2 ? x
    SEQ
      x := x-1
      out2 ! x
```

will wait until data arrives on channel `in1` or `in2` and will then execute the relevant section of code

If data arrives on both simultaneously, one branch will be taken non-deterministically

Occam

The only way for tasks to communicate is via channels

Occam

The only way for tasks to communicate is via channels

There is no concept of shared or distributed, so a program should work equally on shared or distributed memory

Occam

The only way for tasks to communicate is via channels

There is no concept of shared or distributed, so a program should work equally on shared or distributed memory

This is a bit like MPI messaging: it provides independence from the hardware

Occam

Plus loads more features: boolean guards (on ALT); timeouts on guards; priority ordered ALTs; functions; procedures; arrays; while loops; etc.

Occam

Plus loads more features: boolean guards (on ALT); timeouts on guards; priority ordered ALTs; functions; procedures; arrays; while loops; etc.

A program is a bunch of processes (threads in modern terms), joined by `PAR`s, that send data along channels to each other

Occam

Plus loads more features: boolean guards (on ALT); timeouts on guards; priority ordered ALTs; functions; procedures; arrays; while loops; etc.

A program is a bunch of processes (threads in modern terms), joined by `PAR`s, that send data along channels to each other

By being closely related to CSP, there were opportunities to do proofs on Occam programs

Occam

Plus loads more features: boolean guards (on ALT); timeouts on guards; priority ordered ALTs; functions; procedures; arrays; while loops; etc.

A program is a bunch of processes (threads in modern terms), joined by `PAR`s, that send data along channels to each other

By being closely related to CSP, there were opportunities to do proofs on Occam programs

Thus Occam can be said to provide both mechanism and analysis for concurrency

Occam

Occam never took off as transputers were not really up to it

Occam

Occam never took off as transputers were not really up to it

Programmers never got the hang of it, either

Occam

Occam never took off as transputers were not really up to it

Programmers never got the hang of it, either

It has, however had a long-lasting influence on the design of other modern languages

Occam

Occam never took off as transputers were not really up to it

Programmers never got the hang of it, either

It has, however had a long-lasting influence on the design of other modern languages

There was an extension: Occam-π. This was a realisation of the π-calculus, which is itself a generalisation of CSP, where channels and processes are first class objects, e.g., pass a channel down a channel

Occam

Occam never took off as transputers were not really up to it

Programmers never got the hang of it, either

It has, however had a long-lasting influence on the design of other modern languages

There was an extension: Occam-π. This was a realisation of the π-calculus, which is itself a generalisation of CSP, where channels and processes are first class objects, e.g., pass a channel down a channel

A good model to revisit in light of the current obsession with mobile processes

Occam

Occam never took off as transputers were not really up to it

Programmers never got the hang of it, either

It has, however had a long-lasting influence on the design of other modern languages

There was an extension: Occam-π. This was a realisation of the π-calculus, which is itself a generalisation of CSP, where channels and processes are first class objects, e.g., pass a channel down a channel

A good model to revisit in light of the current obsession with mobile processes

Big Exercise Implement Occam on top of MPI, or OpenMP

Occam

Exercise Read about the Xc language that is like C with distinct Occam flavour:

```
int main() {
   par {
      foo(0);
      bar(1);
      baz(3);
   }
   return 0;
}
```

Erlang

Erlang is a single assignment functional language, with explicit support for MIMD parallelism

Erlang

Erlang is a single assignment functional language, with explicit support for MIMD parallelism

A program can contain a large number of very lightweight threads: 20 million is possible they claim

Erlang

Erlang is a single assignment functional language, with explicit support for MIMD parallelism

A program can contain a large number of very lightweight threads: 20 million is possible they claim

Thus these threads do not correspond directly to OS threads, but are managed by the Erlang runtime (a VM; c.f. Go)

Erlang

Erlang is a single assignment functional language, with explicit support for MIMD parallelism

A program can contain a large number of very lightweight threads: 20 million is possible they claim

Thus these threads do not correspond directly to OS threads, but are managed by the Erlang runtime (a VM; c.f. Go)

Having *no shared state*, the threads act more like OS processes than normal threads

Erlang

They do not share state because the processes (they call their threads “processes”) may be on distributed memory

Erlang

They do not share state because the processes (they call their threads "processes") may be on distributed memory

Or two processes might be on the same local shared memory, but you cannot assume that

Erlang

They do not share state because the processes (they call their threads “processes”) may be on distributed memory

Or two processes might be on the same local shared memory, but you cannot assume that

Also, this fits in nicely with the functional style: everything is local to the process and everything is referentially transparent

Erlang

They do not share state because the processes (they call their threads "processes") may be on distributed memory

Or two processes might be on the same local shared memory, but you cannot assume that

Also, this fits in nicely with the functional style: everything is local to the process and everything is referentially transparent

An important consideration is that the overheads of creation, destruction and context switching are very small: thus encouraging many small, short-lived, single-use processes

Erlang

An Erlang runtime will typically run one OS-style thread per core; each running an Erlang scheduler

Erlang

An Erlang runtime will typically run one OS-style thread per core; each running an Erlang scheduler

These schedulers will choose and run the Erlang-style processes in a manager/worker fashion

Erlang

An Erlang runtime will typically run one OS-style thread per core; each running an Erlang scheduler

These schedulers will choose and run the Erlang-style processes in a manager/worker fashion

Thus it avoids the overhead of OS thread creation/deletion

Erlang

An Erlang runtime will typically run one OS-style thread per core; each running an Erlang scheduler

These schedulers will choose and run the Erlang-style processes in a manager/worker fashion

Thus it avoids the overhead of OS thread creation/deletion

In one Erlang implementation, a process requires approximately 600 bytes of state

Erlang

An Erlang runtime will typically run one OS-style thread per core; each running an Erlang scheduler

These schedulers will choose and run the Erlang-style processes in a manager/worker fashion

Thus it avoids the overhead of OS thread creation/deletion

In one Erlang implementation, a process requires approximately 600 bytes of state

Thus enabling a large number of processes

Erlang

An Erlang runtime will typically run one OS-style thread per core; each running an Erlang scheduler

These schedulers will choose and run the Erlang-style processes in a manager/worker fashion

Thus it avoids the overhead of OS thread creation/deletion

In one Erlang implementation, a process requires approximately 600 bytes of state

Thus enabling a large number of processes

Exercise Find out the memory overhead of a normal pthread in your favourite operating system

Erlang

Erlang threads communicate via messages like Occam/CSP, but they are *asynchronous*, unlike Occam/CSP

Erlang

Erlang threads communicate via messages like Occam/CSP, but they are *asynchronous*, unlike Occam/CSP

Again, messages works equally over shared and distributed memory

Erlang

Erlang threads communicate via messages like Occam/CSP, but they are *asynchronous*, unlike Occam/CSP

Again, messages works equally over shared and distributed memory

Also, Erlang does not have named channels, but each process has a “mailbox” where it receives all its messages

Erlang

Erlang threads communicate via messages like Occam/CSP, but they are *asynchronous*, unlike Occam/CSP

Again, messages works equally over shared and distributed memory

Also, Erlang does not have named channels, but each process has a “mailbox” where it receives all its messages

Alternative point of view: the process “name” is the name of the (only) channel to a process

Erlang

The messages can be values, tuples of values, or any other datatype, including closures (functions)

Erlang

The messages can be values, tuples of values, or any other datatype, including closures (functions)

And there is pattern matching to fetch messages from the mailbox (a bit like MPI tags, but more general matching, so more like Linda)

Erlang

```
Otherproc ! { hello, 99 }
```

sends a tuple with *atom* (like a Lisp symbol) `hello` and the integer 99 to the process named by `Otherproc` (variables start with capital letters)

Erlang

```
Otherproc ! { hello, 99 }
```

sends a tuple with *atom* (like a Lisp symbol) `hello` and the integer 99 to the process named by `Otherproc` (variables start with capital letters)

```
receive
  { hello, X } -> io:format("x was ~B~n", [X]);
  { bye, X } -> io:format("time to go~n", []);
  _ -> io:format("eh?~n", [])
end.
```

Erlang

```
Otherproc ! { hello, 99 }
```

sends a tuple with *atom* (like a Lisp symbol) `hello` and the integer 99 to the process named by `Otherproc` (variables start with capital letters)

```
receive
  { hello, X } -> io:format("x was ~B~n", [X]);
  { bye, X } -> io:format("time to go~n", []);
  _ -> io:format("eh?~n", [])
end.
```

an underscore matches any message; this is like an ALT in Occam

Erlang

Creation of processes is via `spawn`

```
factrec(0) -> 1;
factrec(N) when N > 0 -> N*factrec(N-1).
fact(N, Ans) -> Ans ! factrec(N).

FactPid = spawn(fact, [5, self()]).

receive
  F -> io:format("factorial is ~B~n", [F])
end.
```

is clumsy code to make a new process running `fact` with arguments 5 and the process identifier (PID) of the current process

Erlang

Creation of processes is via `spawn`

```
factrec(0) -> 1;
factrec(N) when N > 0 -> N*factrec(N-1).
fact(N, Ans) -> Ans ! factrec(N).

FactPid = spawn(fact, [5, self()]).

receive
  F -> io:format("factorial is ~B~n", [F])
end.
```

is clumsy code to make a new process running `fact` with arguments 5 and the process identifier (PID) of the current process

The `receive` causes the current process (`self()`) to wait for a message (from anyone), and stores it in `F`

Erlang

A PID is the way you refer to a process, in particular for sending a message to it

N.B. some liberties taken with Erlang modules here

Erlang

Erlang is quite popular in real systems as it has lots of useful features

Erlang

Erlang is quite popular in real systems as it has lots of useful features

For example, *Process Restart*, where a process is immediately restarted by the runtime if it crashes for any reason

Erlang

Erlang is quite popular in real systems as it has lots of useful features

For example, *Process Restart*, where a process is immediately restarted by the runtime if it crashes for any reason

This allows Erlang to cope with hardware failure and buggy code

Erlang

Erlang is quite popular in real systems as it has lots of useful features

For example, *Process Restart*, where a process is immediately restarted by the runtime if it crashes for any reason

This allows Erlang to cope with hardware failure and buggy code

In fact, Erlang has hot swap of code: code can be changed (fixed or updated) while the main program is running

Erlang

Erlang is quite popular in real systems as it has lots of useful features

For example, *Process Restart*, where a process is immediately restarted by the runtime if it crashes for any reason

This allows Erlang to cope with hardware failure and buggy code

In fact, Erlang has hot swap of code: code can be changed (fixed or updated) while the main program is running

Load balancing of processes is done by the runtime VM

Erlang

Originally designed by Ericsson to support (soft) realtime systems that can’t be taken down for maintenance (like telephone exchanges), it has found use in other areas

Erlang

Originally designed by Ericsson to support (soft) realtime systems that can't be taken down for maintenance (like telephone exchanges), it has found use in other areas

Companies like Yahoo, Facebook, WhatsApp, Bet365, etc. use it for some element of their products

Erlang

Originally designed by Ericsson to support (soft) realtime systems that can't be taken down for maintenance (like telephone exchanges), it has found use in other areas

Companies like Yahoo, Facebook, WhatsApp, Bet365, etc. use it for some element of their products

Somewhat an under-appreciated language

Erlang

Originally designed by Ericsson to support (soft) realtime systems that can't be taken down for maintenance (like telephone exchanges), it has found use in other areas

Companies like Yahoo, Facebook, WhatsApp, Bet365, etc. use it for some element of their products

Somewhat an under-appreciated language

Exercise Have a look at `http://learnyousomeerlang.com/content`

Go

Go we have seen before, so here's just a short discussion (in the context of these other parallel-aware languages)

Go

Go we have seen before, so here's just a short discussion (in the context of these other parallel-aware languages)

It has goroutines, communicating via channels, similar to Occam and Erlang

Go

Go we have seen before, so here's just a short discussion (in the context of these other parallel-aware languages)

It has goroutines, communicating via channels, similar to Occam and Erlang

Channels are type safe ("channel of int") and blocking

Go

Go we have seen before, so here's just a short discussion (in the context of these other parallel-aware languages)

It has goroutines, communicating via channels, similar to Occam and Erlang

Channels are type safe ("channel of int") and blocking

There is a `select` that acts like Occam's `ALT` waiting on multiple channels

Go

Synchronisation and communication are provided by channels

Go

Synchronisation and communication are provided by channels

Libraries provide condition variables, mutexes, atomics and a variety of other low-level functionality

Go

Synchronisation and communication are provided by channels

Libraries provide condition variables, mutexes, atomics and a variety of other low-level functionality

Channels are the recommended ways of passing data between threads; though you can also use shared variables

Go

Synchronisation and communication are provided by channels

Libraries provide condition variables, mutexes, atomics and a variety of other low-level functionality

Channels are the recommended ways of passing data between threads; though you can also use shared variables

Though shared variables are not recommended as Go provides no inherent protection against the usual data races (if you don't remember to use mutexes and the like)

Go

From the Go website (worth repeating!):

> *Share memory by communicating; don't communicate by sharing memory.*

Go

Go has a race detector tool: compiling with `-race` checks memory accesses and spots unsynchronised accesses

Go

Go has a race detector tool: compiling with `-race` checks memory accesses and spots unsynchronised accesses

This

- is run time detection
- slows the execution by an order of magnitude
- only finds races that actually happen in a run

Go

Go is used widely, with some vocal proponents

Go

Go is used widely, with some vocal proponents

It was designed by people with a considerable amount of expertise, but doesn’t bring anything new to the table in terms of tackling parallelism

Go

Go is used widely, with some vocal proponents

It was designed by people with a considerable amount of expertise, but doesn't bring anything new to the table in terms of tackling parallelism

> *. . . in fact, there isn't much to Go other than channels and goroutines!*

Stjepan Glavina

Rust

A language originally designed and developed by the Mozilla team, with the eventual aim of reimplementing the Firefox browser in Rust, but now a general-purpose language in its own right

Rust

A language originally designed and developed by the Mozilla team, with the eventual aim of reimplementing the Firefox browser in Rust, but now a general-purpose language in its own right

A lot of the problems in many applications (including browsers) are to do with bad memory management

Rust

A language originally designed and developed by the Mozilla team, with the eventual aim of reimplementing the Firefox browser in Rust, but now a general-purpose language in its own right

A lot of the problems in many applications (including browsers) are to do with bad memory management

> *Around 70 percent of all the vulnerabilities in Microsoft products addressed through a security update each year are memory safety issues*
> Matt Miller, Microsoft security engineer, Feb 2019

Rust

So Rust is a *memory safe* language, meaning it can not have problems like dangling pointers (null pointers), uninitialised variables, use after free errors, or buffer overflows

Rust

So Rust is a *memory safe* language, meaning it can not have problems like dangling pointers (null pointers), uninitialised variables, use after free errors, or buffer overflows

Or, at least, it makes it very hard for the programmer to produce such bad code

Rust

So Rust is a *memory safe* language, meaning it can not have problems like dangling pointers (null pointers), uninitialised variables, use after free errors, or buffer overflows

Or, at least, it makes it very hard for the programmer to produce such bad code

Unlike many languages, such as C and C++, that make it very easy

Rust

Memory safety is not new: many memory safe (or nearly safe) languages have been devised, with various trade-offs to get this safety

Rust

Memory safety is not new: many memory safe (or nearly safe) languages have been devised, with various trade-offs to get this safety

For example, runtime checks on accesses to buffers; garbage collectors; and so on

Rust

Memory safety is not new: many memory safe (or nearly safe) languages have been devised, with various trade-offs to get this safety

For example, runtime checks on accesses to buffers; garbage collectors; and so on

A lot of these have runtime overhead, i.e., your program is safer, but runs more slowly

Rust

Memory safety is not new: many memory safe (or nearly safe) languages have been devised, with various trade-offs to get this safety

For example, runtime checks on accesses to buffers; garbage collectors; and so on

A lot of these have runtime overhead, i.e., your program is safer, but runs more slowly

And they are not always completely successful, e.g., Java can have null pointers

Rust

Rust takes a different approach and tries to put as much checking as possible into the compiler: your code is safe, and fast

Rust

Rust takes a different approach and tries to put as much checking as possible into the compiler: your code is safe, and fast

But the trade-off is this: it does this by having a concept of the *owner* of a memory location and tracking that ownership in the compiler

Rust

In an assignment `y = x;` the ownership of the memory referred to by `x` is transferred to `y`. It is now illegal/impossible to use the variable `x` in subsequent code

Rust

In an assignment `y = x;` the ownership of the memory referred to by `x` is transferred to `y`. It is now illegal/impossible to use the variable `x` in subsequent code

The **compiler** would flag any later reference to `x` as an error and refuse to compile

Rust

In an assignment `y = x;` the ownership of the memory referred to by `x` is transferred to `y`. It is now illegal/impossible to use the variable `x` in subsequent code

The **compiler** would flag any later reference to `x` as an error and refuse to compile

This helps with memory management, as the compiler can precisely track the lifetime of a value and so its memory can be deallocated automatically when the compiler can prove it is not longer accessible and without the need for a garbage collector

Rust

In an assignment `y = x;` the ownership of the memory referred to by `x` is transferred to `y`. It is now illegal/impossible to use the variable `x` in subsequent code

The **compiler** would flag any later reference to `x` as an error and refuse to compile

This helps with memory management, as the compiler can precisely track the lifetime of a value and so its memory can be deallocated automatically when the compiler can prove it is not longer accessible and without the need for a garbage collector

Thus avoiding the programming errors common to C-like languages and the runtime complexities of GC languages