
DALL·E

Concurrent Computation Server

DATA RACE

```
range R := 0..720

Cell[v,rd,wr]:= rd!v . Cell[v,rd,wr] + wr?x:R . Cell[x,rd,wr]
Cells := Cell[0,rdA,wrA] | Cell[0,rdB,wrB]

Serve := mult? . rdA?x . rdB?y . Iter[0,x,y]
       + add? . rdA?x . rdB?y . println!(x+y) . Serve

Iter[z,x,y] := when (x>0) i . Iter[z+y,x-1,y]
             + when (x==0) println!z . Serve

   Use := wrA!6 . wrB!5 . mult! . 0
   Use := wrA!7 . wrB!3 . add! . 0
⟹ Use := wrA!6 . wrB!5 . mult! . wrA!7 . wrB!3 . add! . 0

(Cells | Serve | Use) \ {rdA, wrA, rdB, wrB, mult, add}
```

Status Quo

We have the tools required to model the execution of concurrent programs.

And we can already see interesting phenomena, like DATA RACEs.

But modelling programs in CCS is a lot of work!

If only there was an easier way to do this…

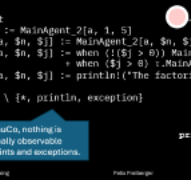
Introducing pseuCo

```
mainAgent {
    int n, j;
    n = 1;

    for (j = 5; j > 0; j--) {
        n = n * j;
    }

    println("The factorial of 5 is " + n + ".");
}
```

pseuCo Compiler

Local variables turn into processes parameters.

```
// Main agent
MainAgent[a] := MainAgent_2[a, 1, 5]
MainAgent_1[a, $n, $j] := MainAgent_2[a, $n, $j - 1]
MainAgent_2[a, $n, $j] := when (!($j > 0)) MainAgent_3[a, $n, $j]
                        + when ($j > 0) t.MainAgent_1[a, $n * $j, $j]
MainAgent_3[a, $n, $j] := println!("The factorial of 5 is " ^ $n ^ ".").0

MainAgent[1] \ {*, println, exception}
```



In pseuCo, nothing is externally observable besides prints and exceptions.

Concurrent Programming

```
void countDutch() {
    println("Een");
    println("Twee");
    println("Drie");
}

void countEspañol() {
    println("  Uno");
    println("  Dos");
    println("  Tres");
}

mainAgent {
    agent a1 = start(countDutch());   // start one agent
    agent a2 = start(countEspañol()); // start another one
    println ("Both count!"); // let everyone know
}
```

Two procedures

```
Een
Twee
Both count!
  Uno
  Dos
Drie
  Tres
```

Agents execute procedures independently and concurrently.

pseuCo So Far

So far, pseuCo has capabilities for…

- programming language basics (math, conditions, loops, …) and
- starting agents.

We can write concurrent programs where agents work in isolation and translate them to CCS.

What's missing?

PseuCo doesn't allow any form of interaction or coordination between agents, besides starting them (and passing simple arguments to them on start).

Message Passing

- processes can "simply" share **global variables**
- communication is implicit through access to global memory

```
Process1 := [...] sync!x [...]
Process2 := [...] sync?x [...]
```

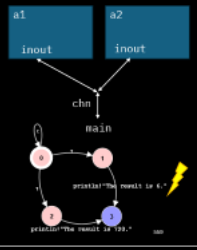
```
Cell[value] := read!value.Cell[value]
             + write?value:R.Cell[value]
```

- processes can send and receive **messages** over **channels**
- communication is explicit and can be synchronous or asynchronous

Shared Memory

Cost Saving Measures

```
void factorial(intchan inout) {
    int j, n;
    int z = <? inout;
    n = 1;
    for (j = z; j > 0; j--) {
        n = n * j;
    }
    inout <! n; // send result
}

mainAgent {
    intchan chn;
    agent a1 = start(factorial(chn));
    agent a2 = start(factorial(chn));
    chn <! 3;
    int fin = <? chn;
    println("The result is " + fin + ".");
}
```



Loads of Factorials

```
intchan10 c1, c2;

void factorial() {
    while (true) {
        int j, n;
        int z = <? c1; // get an input
        n = 1;
        for (j = z; j > 0; j--) {
            n = n * j;
        }
        c2 <! n; // send result
    };
}

mainAgent {
    start(factorial());
    start(factorial());
    for (int i = 3; i <= 10; i++) {
        c1 <! i;
    };
    for (int i = 2; i <= 10; i++) {
        println("Result: " + <? c2 + ".");
    };
}
```

For convenience, global channel declarations are allowed, too.

But nothing else, and you may not reassign them!

PseuCo.com will not check this. PseuCo Book will.

Choice in pseuCo

To enable attempting multiple message passing actions concurrently, pseuCo has the `select-case-statement`:

This statement can do three things:

- receive a value on c, store it in x, then continue with the first code block,
- send 42 on c, then continue with the second code block, or
- simply jump into the third code block.

```
select {
    case x = <? c: {
        // ...
    }
    case c <! 42: {
        // ...
    }
    default: {
        // ...
    }
}
```

If several options are enabled, the choice is made **nondeterministically.** (But only between enabled options, so no deadlock can occur simply because the one impossible option was chosen. Like 4!)

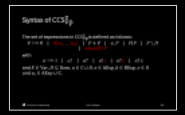

DALL·E

Under the Hood: Synchronous Channels

```
intchan c;

void printer() {
    println(<? c);
}

mainAgent {
    start(printer());
    c <! 42;
}
```

```
Channel_cons[i] := channel_create!i . Channel_cons[i - 1]

GlobalVariablesInitialisations
  := channel_create?$0 . env_global_set_c!$0 . 1

// Procedures
Proc_printer[a] := receive($0)?$1 . println!$1 . 1

// Main agent
MainAgent[a] := [...] receive($1)!42 . MainAgent_1[a, 42])
MainAgent_1[a, t0] := 0
```

termination (see exercise EB.17)



part of channel name, ensures both sides hold the same channel

π-calculus

Under the Hood: Asynchronous Channels

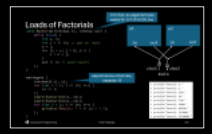
```
// System processes
ChannelManager[next_i] := channel_new!next_i . ChannelManager[next_i + 1]
Channel3[i, c, v0, v1, v2] :=  when (c == 0) put(i)?v0 . Channel3[i, c + 1, v0, v1, v2]
                             + (when (c == 1) put(i)?v1 . Channel3[i, c + 1, v0, v1, v2]
                             + (when (c == 2) put(i)?v2 . Channel3[i, c + 1, v0, v1, v2]
                             + when (c > 0) receive(i)!v0 . Channel3[i, c - 1, v1, v2, 0]))
Channel3_cons := channel3_prepare? . channel_new?next_i . channel3_create!next_i
               . (Channel3_cons | Channel3[next_i, 0, 0, 0, 0])

GlobalVariablesInitialisations := channel3_prepare! . channel3_create?$0 . [...]

// Procedures
Proc_printer[a] := [...] receive($0)?$1 . println!$1 . 1

// Main agent
MainAgent[a] := [...] put($1)!42 . MainAgent_1[a, 42] [...]
MainAgent_1[a, t0] := 0
```

```
intchan3 c;

void printer() {
    println(<? c);
}

mainAgent {
    start(printer());
    c <! 42;
}
```

Under the Hood: Channel IDs

`intchan` as argument type works for `intchan10`, too

```
void factorial(intchan in, intchan out) {
```



Receive on `in`: `receive($in)?$0`

Send on `out`:

```
  when ($out >= 0) put($out)!42 . [...]
+ when ($out <  0) receive($out)!42 . [...]
```

Negative channel ids are used for synchronous channels, and always operate on the `receive` action.
Positive channel ids are used for asynchronous channels, which receive values on the `receive` action, but send using `put`.

Run, pseuCo, Run!

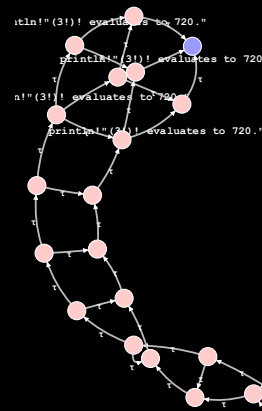
Message Passing in pseuCo in Action

Checking Primes

```
bool checkPrime(int n) {
    for (int i = 2; i < n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}

void primer(intchan in, intchan nout, boolchan rout) {
    while(true) {
        int n = <? in;
        bool result = checkPrime(n);
        nout <! n;
        rout <! result;
    }
}
```

```
mainAgent {
    intchan10 nums;
    intchan2 checkedNums;
    boolchan2 results;
    start(primer(nums, checkedNums, results));
    start(primer(nums, checkedNums, results));
    for (int i = 2; i <= 100; i++) {
        nums <! i;
    }
    while (true) {
        println("The number " + <? checkedNums + " is " + (<? results ? "" : "not ") + "prime.");
    };
}
```

Executing pseuCo



pseuCo → CCS → LTS

- gives the full semantics of any pseuCo program
- implemented in pseuCo.com
- details insight into how pseuCo's features are implemented
- helpful addition: ≅ minimization
- main semantics – do understand it!
- …but sometimes too low-level

pseuCo → Java

- allows you to execute any pseuCo program
- compiler and pseuCo IDE for download on pseuCo.com
- conceptually simple translation – we'll discuss it in unit G
- technically fiddly, optional

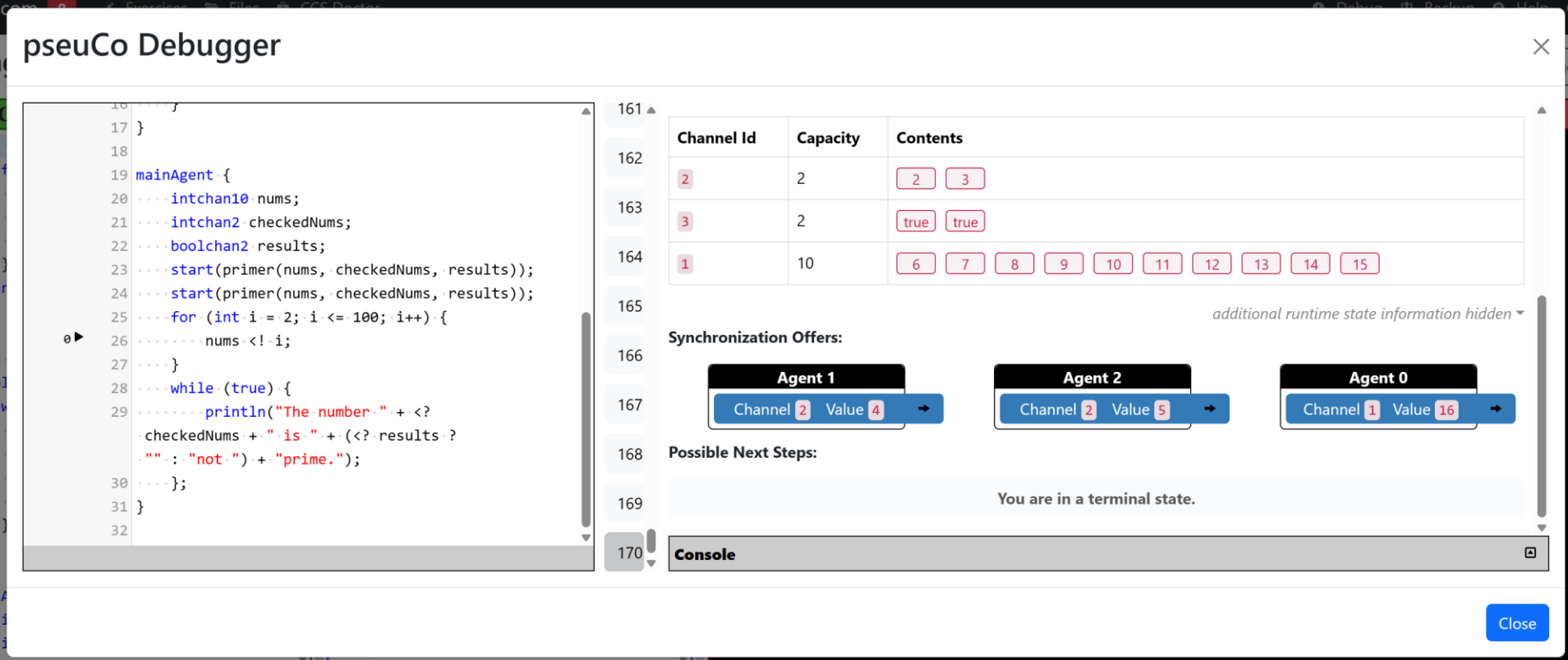
pseuCo → Petri nets → pseuCo Debugger

Checking Primes

Deadlock!

```
bool checkPrime(int n) {
    for (int i = 2; i < n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}

void primer(intchan in, intchan nout, boolchan rout) {
    while(true) {
        int n = <? in;
        bool result = checkPrime(n);
        nout <! n;
        rout <! result;
    }
}
```

```
mainAgent {
    intchan10 nums;
    intchan2 checkedNums;
    boolchan2 results;
    start(primer(nums, checkedNums, results));
    start(primer(nums, checkedNums, results));
    for (int i = 2; i <= 100; i++) {
        nums <! i;
    }
    while (true) {
        println("The number " + <? checkedNums + " is “
         + (<? results ? "" : "not ") + "prime.");
    };
}
```



In pseuCo Debugger, we found the issue:

- The channel `nums` is full.
- The `mainAgent` still wants to write a bunch of numbers to it.
- The primer agents have some results, but can't store more results.

Checking Primes

And now?
Is it correct?

```
bool checkPrime(int n) {
    for (int i = 2; i < n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}

void primer(intchan in, intchan nout, boolchan rout) {
    while(true) {
        int n = <? in;
        bool result = checkPrime(n);
        nout <! n;
        rout <! result;
    }
}
```

Message confusion!

```
mainAgent {
    intchan10 nums;
    intchan2 checkedNums;
    boolchan2 results;
    start(primer(nums, checkedNums, results));
    start(primer(nums, checkedNums, results));
    for (int i = 2; i <= 10; i++) {
        nums <! i;
    }
    while (true) {
        println("The number " + <? checkedNums + " is “
         + (<? results ? "" : "not ") + "prime.");
    };
}
```

Checking Primes

And now?
Is it correct?

```
bool checkPrime(int n) {
    for (int i = 2; i < n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}

void primer(intchan in, intchan nout, boolchan rout) {
    while(true) {
        int n = <? in;
        bool result = checkPrime(n);
        nout <! n;
        rout <! result;
    }
}
```

```
mainAgent {
    intchan10 nums;
    intchan1 checkedNums;
    boolchan1 results;
    start(primer(nums, checkedNums, results));
    start(primer(nums, checkedNums, results));
    for (int i = 2; i <= 10; i++) {
        nums <! i;
    }
    while (true) {
        println("The number " + <? checkedNums + " is “
         + (<? results ? "" : "not ") + "prime.");
    };
}
```

Message confusion (still)!

Checking Primes

And now?
Is it correct?

```
bool checkPrime(int n) {
    for (int i = 2; i < n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}

void primer(intchan in, intchan nout, boolchan rout) {
    while(true) {
        int n = <? in;
        bool result = checkPrime(n);
        nout <! n;
        rout <! result;
    }
}
```

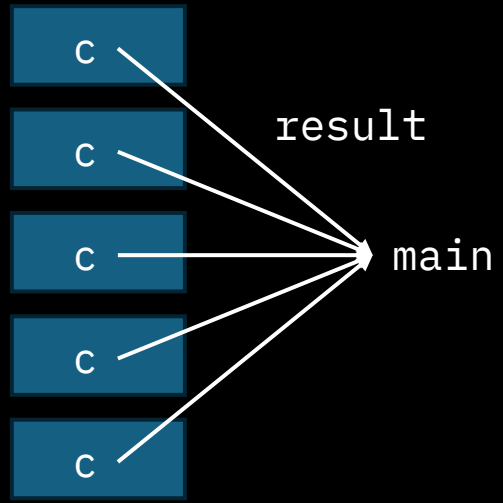
```
mainAgent {
    intchan10 nums;
    intchan checkedNums;
    boolchan results;
    start(primer(nums, checkedNums, results));
    start(primer(nums, checkedNums, results));
    for (int i = 2; i <= 10; i++) {
        nums <! i;
    }
    while (true) {
        println("The number " + <? checkedNums + " is “
         + (<? results ? "" : "not ") + "prime.");
    };
}
```

Results are unsorted!
This easily happens when splitting work. Whether that’s OK depends on the context.

Yo! I heard you like factorials.

Sorted Factorials

```
void factorial(int z, intchan res) {
    int n = 1;
    for (int j = z; j > 0; j--) {
        n = n * j;
    }
    res <! n;
}

mainAgent {
    intchan[5] result;
    for (int j = 0; j < 5; j++) {
        start(factorial(j + 1, result[j]));
    }
    for (int j = 0; j < 5; j++) {
        println("The factorial of " + (j + 1) + " is " + <? result[j] + ".");
    }
}
```

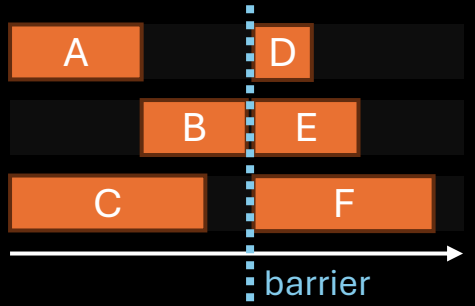


Message Passing Building Blocks

Producer & Consumer, Pipelining, and Other Patterns


DALL·E

Barriers



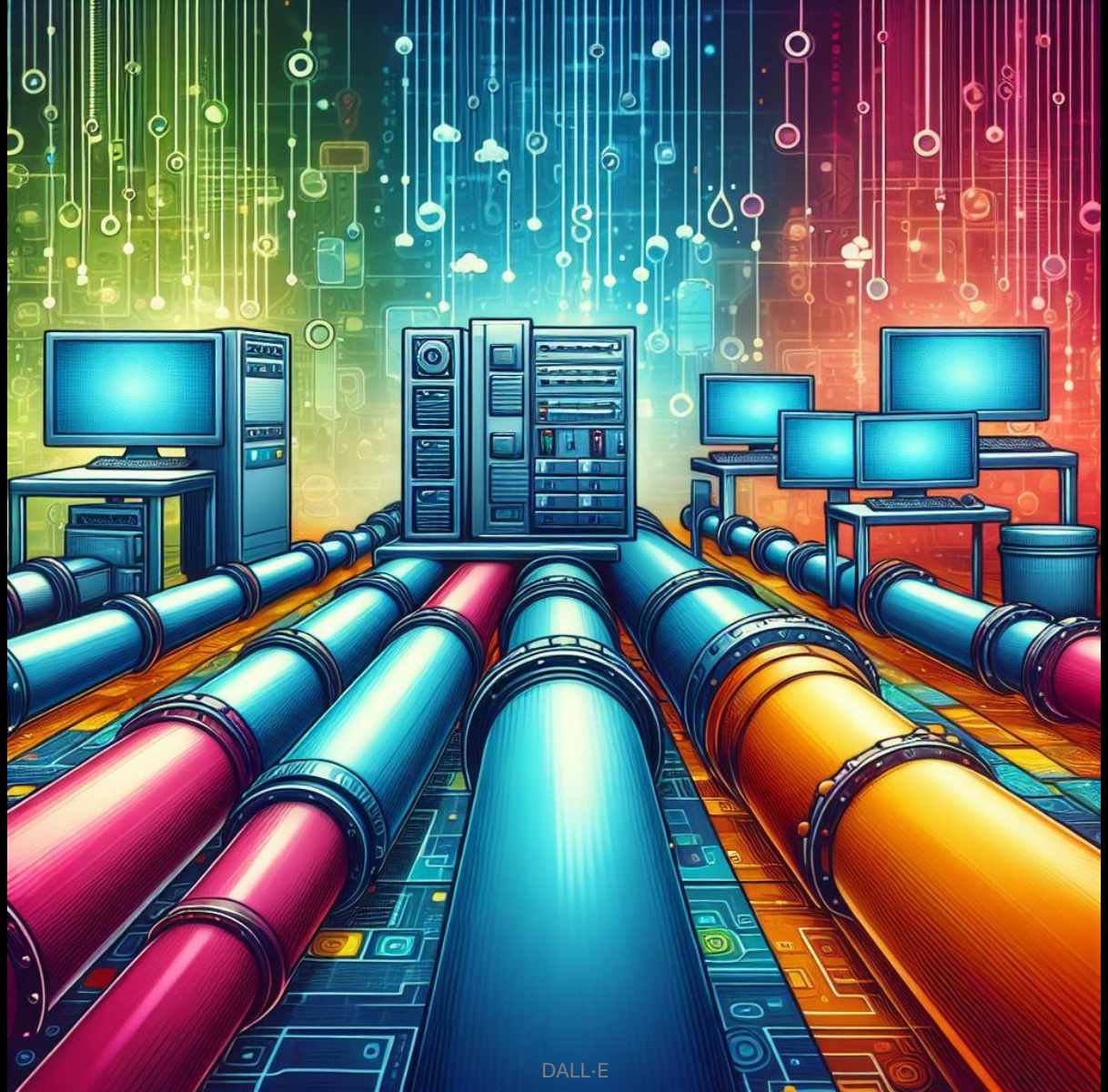
A typical problem when writing concurrent programs:
We want a group of agents to wait for each other before all may progress together.

```
void barrier(int num, boolchan ready, boolchan go) {
    while(true) {
        for (int j = 0; j < num; j++) {
            <? ready;
        }
        println("Everyone jump!");
        for (int j = 0; j < num; j++) {
            go <! true;
        }
    }
}
```

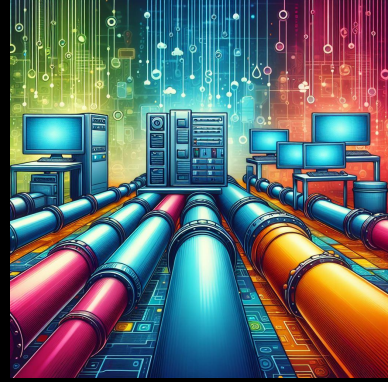
```
boolchan ready;
boolchan go;
```

To synchronize:

```
ready <! true;
<? go;
```


DALL·E

Back to Our Prime Example

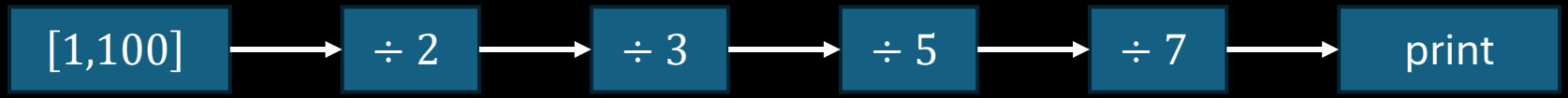


Our previous approach to computing primes was parallel, but extremely inefficient. How can we do this better?

- Math Trick: To check whether n is prime, it is sufficient to look for potential divisors d in the range $2 \leq d \leq \sqrt{n}$.
- Another Math Trick: It is sufficient to check divisors that are prime.

But how do we structure our program to do this easily?
Let's assume we want all primes up to 100.

"pipelining"



generates all numbers

check divisibility by their respective prime, pass on non-divisible numbers, drop the rest

prints survivors

Piping Primes

```
void primer(int v, intchan in, intchan out) {
    while(true) {
        int p = <? in;
        if (p % v == 0) continue;
        out <! p;
    }
}
```

```
mainAgent {
    intchan[5] pipe;
    start(primer(2, pipe[0], pipe[1]));
    start(primer(3, pipe[1], pipe[2]));
    start(primer(5, pipe[2], pipe[3]));
    start(primer(7, pipe[3], pipe[4]));

    for(int i = 2; i <= 100; i++) {
        pipe[0] <! i;
    }

    while(true) {
        println(<? pipe[4] + " is a prime number.");
    }
}
```

I hid a bug! Can you find it?

Deadlock: Pipeline full!

Piping Primes

```
void primer(int v, intchan in, intchan out) {
    while(true) {
        int p = <? in;
        if (p % v == 0) continue;
        out <! p;
    }
}
```

```
void generator(intchan out) {
    for(int i = 2; i <= 100; i++) {
        out <! i;
    }
}
```

```
mainAgent {
    intchan[5] pipe;
    start(primer(2, pipe[0], pipe[1]));
    start(primer(3, pipe[1], pipe[2]));
    start(primer(5, pipe[2], pipe[3]));
    start(primer(7, pipe[3], pipe[4]));

    start(generator(pipe[0]));

    while(true) {
        println(<? pipe[4] + " is a prime number.");
    }
}
```

Does it work now?

Console

```
11 is a prime number.
13 is a prime number.
17 is a prime number.
```

Piping Primes

Bonus Question:
Is swapping the primes a good idea?

No, it's slower.

```
void primer(int v, intchan in, intchan out) {
    while(true) {
        int p = <? in;
        if (p % v == 0 && p != v) continue;
        out <! p;
    }
}
```

```
void generator(intchan out) {
    for(int i = 2; i <= 100; i++) {
        out <! i;
    }
}
```

```
mainAgent {
    intchan[5] pipe;
    start(primer(2, pipe[0], pipe[1]));
    start(primer(3, pipe[1], pipe[2]));
    start(primer(5, pipe[2], pipe[3]));
    start(primer(7, pipe[3], pipe[4]));

    start(generator(pipe[0]));

    while(true) {
        println(<? pipe[4] + " is a prime number.");
    }
}
```

Does it work now?

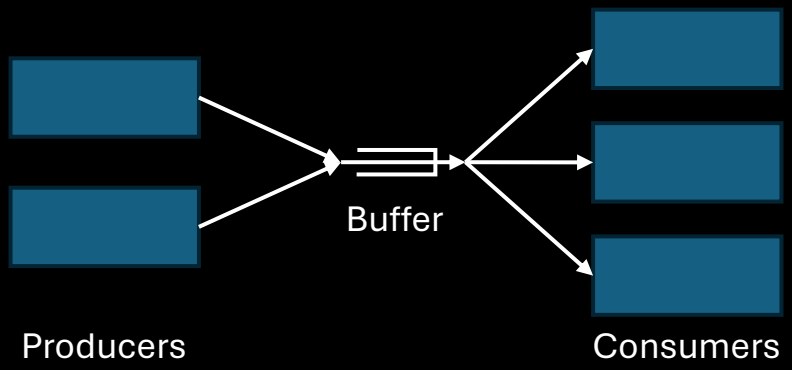
Console

```
2 is a prime number.
3 is a prime number.
5 is a prime number.
7 is a prime number.
11 is a prime number.
13 is a prime number.
17 is a prime number.
```

Producer & Consumer

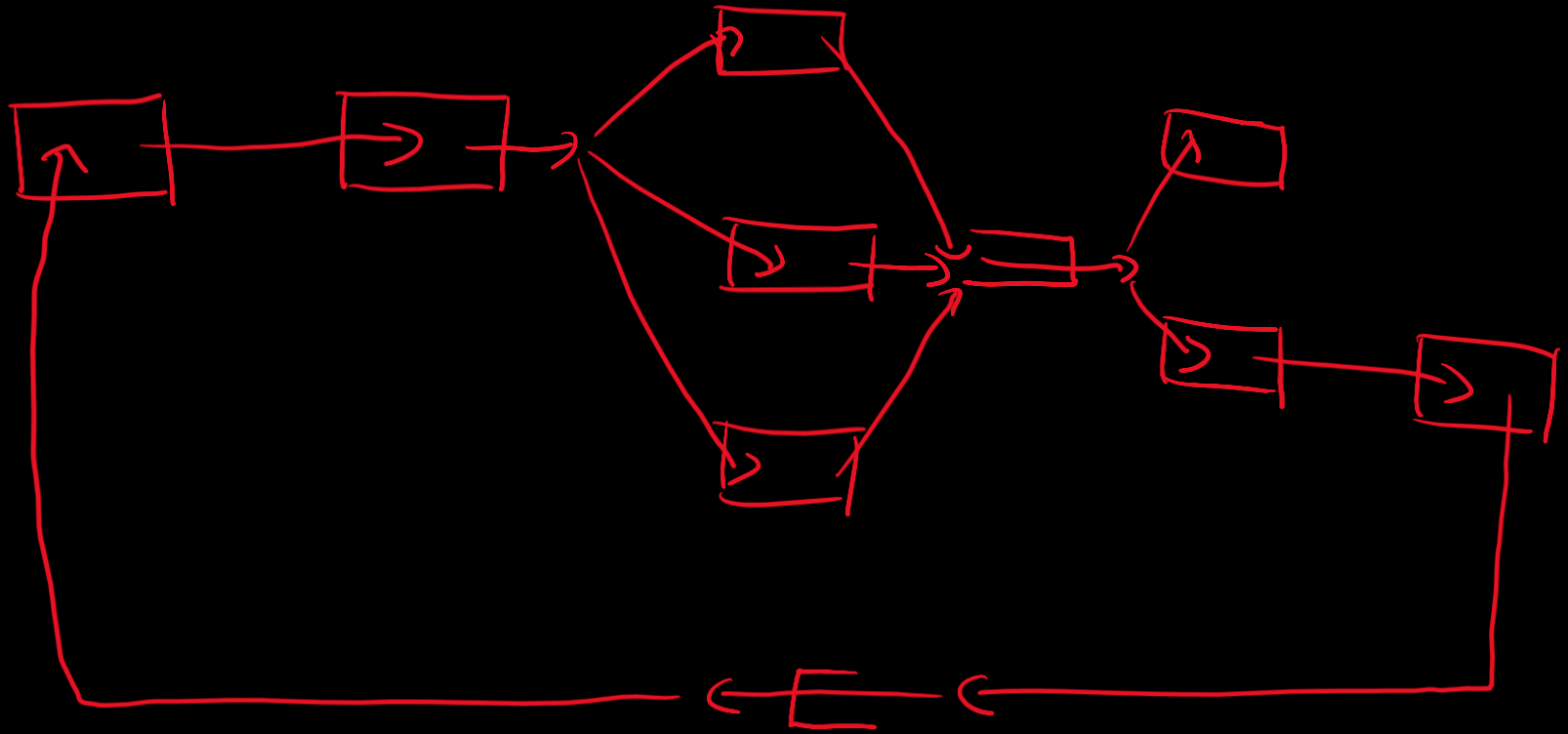
Many problems can be split into two parts, where the first one consists of producing intermediate results and the second one processes them:

- finding primes by 1 generating numbers and 2 checking them
- computing 2!, 3!, 5!, 7!, … by 1 finding primes and 2 factorizing them
- making an exam by 1 generating ideas and 2 filtering out bad ones

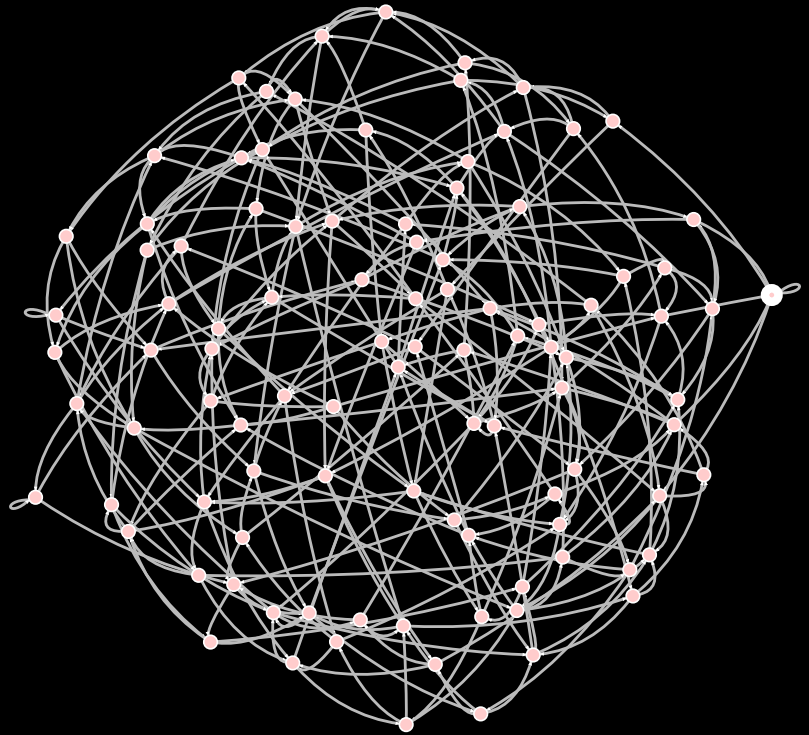


And in the Real World?

The real world is complicated.
These patterns are just examples,
and they can mix and match.



Search in Large Graphs



Search in Large Graphs

```
mainAgent {
    intchan1000 queue;
    queue <! 0;



    while(true) {
        int current = <? queue;
        if (isTarget(current)) {
            println("Found it!");
            break;
        } else {
            println("I am working on node " +  current);

            int[3] next = post(current);
            for (int i = 0; i < 3; i++) {
                int succ = next[i];
                queue <! succ;
            }
        }
    }
}
```

May not terminate. Why?
Cycles!

Search in Large Graphs

```
mainAgent {
    intchan1000 queue;
    queue <! 0;

    bool[100] seen;

    while(true) {
        int current = <? queue;
        if (isTarget(current)) {
            println("Found it!");
            break;
        } else if (!seen[current]) {
            println("I am working on node " +  current);
            seen[current] = true;
            int[3] next = post(current);
            for (int i = 0; i < 3; i++) {
                int succ = next[i];
                queue <! succ;
            }
        }
    }
}
```

How to parallelize this?

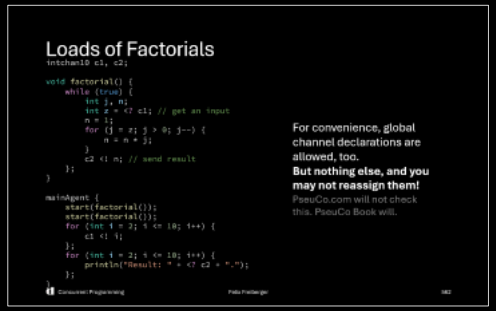
```
intchan1000[4] queue;
```

Concurrent Search in Large Graphs

```
void searcher(int id, boolchan foundIt) {
    bool[100] seen;

    while(true) {
        int current = <? queue[id];
        if (isTarget(current)) {
            foundIt <! true;
            break;
        } else if (!seen[current]) {
            println("Agent " + id + " is working on node " +  current);
            seen[current] = true;
            int[3] next = post(current);
            for (int i = 0; i < 3; i++) {
                int succ = next[i];
                queue[succ % 4] <! succ;
            }
        }
    }
}
```

```
mainAgent {
    boolchan foundIt;
    for (int i = 0; i < 4; i++)
        start(searcher(i, foundIt));
    queue[0] <! 0;
    <? foundIt;
    println("The target state was found!");
}
```

pseuCo-MP Terms of Service



For a pseuCo program to be a pseuCo-MP program, you may not:

- declare a global variable unless it is a channel,
- write to any global variable,
- use any of the following constructs (that we'll introduce later): `lock`, `join()`, `monitor`, `condition`,
- start a procedure as an agent that receives an argument that is a struct, an array, or an agent, or
- start a method of a struct as an agent.

PseuCo.com does not check this!
PseuCo Book will, in the Message Passing section.

```
void sneakyWrite(int[1] sneaky) {
    sneaky[0] = 42;
}

mainAgent {
    int[1] sneaky = { 1 };
    start(sneakyWrite(sneaky));
    println(sneaky[0]);
}
```