Vrije Universiteit Brussel

Parallel Computing applied to RL

Denis Steckelmacher

AI-lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel

Outline

Why parallel?

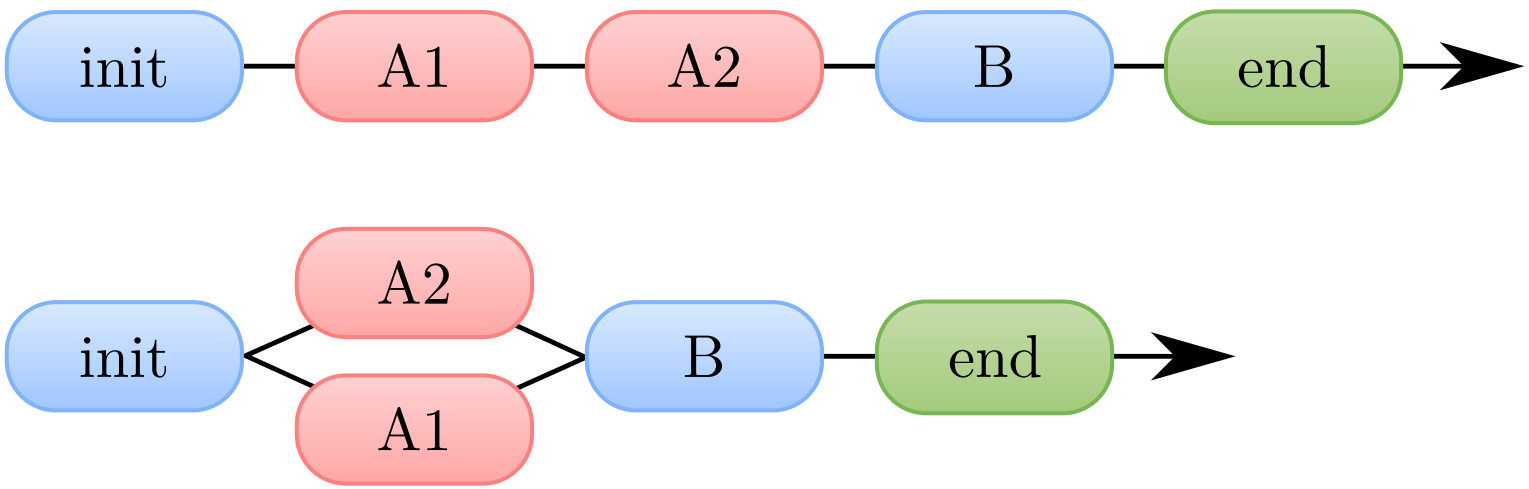
- More speed
- Better efficiency

But require some effort!

Amdahl's law

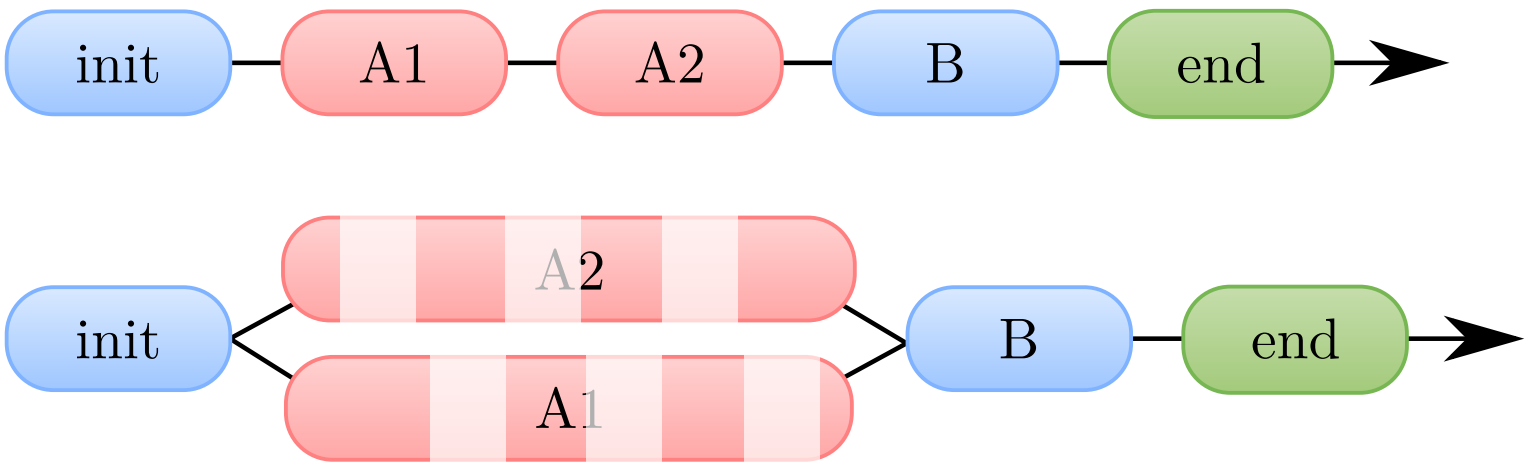
- Non-parallel portions of programs slow things down

$$T = \frac{1}{1 - p + \frac{p}{s}} \tag{1}$$



Communication overhead

- Exchanging information takes time



Threads

- Separate contexts
- Shared memory
- Shared filesystem

```
def thread_func(numbers):
    for i in range(len(numbers)):
        numbers[i] = math.log(numbers[i])

t1 = threading.Thread(target=thread_func, args=(n1,))
t2 = threading.Thread(target=thread_func, args=(n2,))

t1.start(); t2.start();
```

Processes

- Separate contexts
- Separate memory
- Shared filesystem

```
def tf(numbers):
    for i in range(len(numbers)):
        # Will not work!
        numbers[i] = math.log(numbers[i])

p1 = multiprocessing.Process(target=tf, args=(n1,))
p2 = multiprocessing.Process(target=tf, args=(n2,))

p1.start(); p2.start();
```

Distributed computing

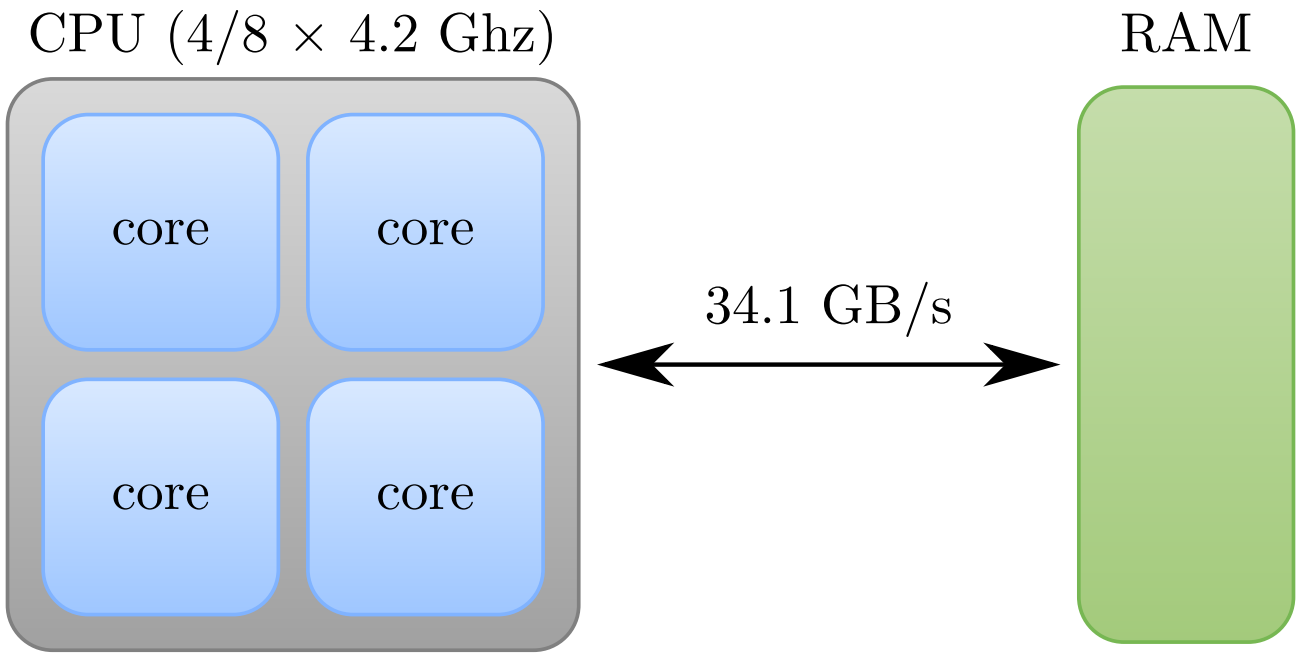
- Separate contexts
- Separate memory
- Separate filesystems

Implemented by launching processes that communicate over the network.

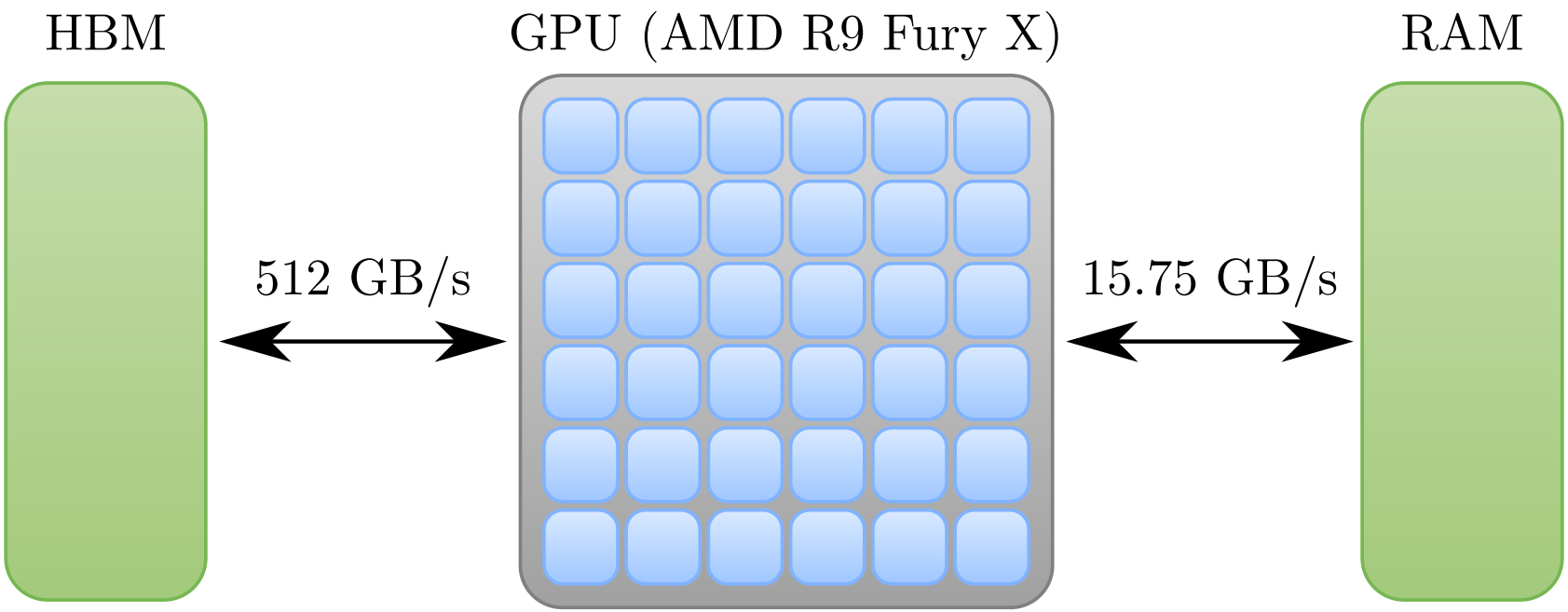
Introduction

What causes programs to be slow? Will parallelization help?

A modern multi-core CPU



A modern GPU



Performance tips

- Avoid slow operations (sin, exp, log)
- Ensure continuous memory accesses
- Use optimized native-code libraries where possible (Numpy, Matlab, ...)

GNU Parallel

- UNIX tool that spawns processes
- No communication between processes
- Distributed computing
- Perfect parallelization efficiency

```
ls *.tar | parallel -j 4 xz {}
```

Parameter search

```
parallel \
    -j 4 \          # Launch 4 jobs at a time
    --header : \    # Allow the ":::" syntax
    ./my_experiment -o {size}_{lr} -n {size} -lr {lr} \
    ::: size 150 200 300 \
    ::: lr 1e-3 1e-4 1e-5
```

Distributed computing

```
parallel \
    --wd . \        # Set current working directory
    -S host1,host2,host3 \
    --header : \
    ./my_experiment -o {size}_{lr} -n {size} -lr {lr} \
    ::: size 150 200 300 \
    ::: lr 1e-3 1e-4 1e-5
```

Jobs run on different computers, be careful of the filesystem!

OpenMP

- Easy parallelization of C/C++/Fortran code
- Uses threads
- Will soon support GPUs
- No support for distributed computing

```
#pragma omp parallel for
for (int i=0; i<N; ++i) {
    rs[i] = std::log(rs[i]);
}
```

Critical sections

- Some operations can be executed by only one thread at once
- Incrementing counters, accessing global variables, ...

```
int nulls = 0;

#pragma omp parallel for
for (int i=0; i<N; ++i) {
    rs[i] = std::log(rs[i]);

    if (rs[i] < 1e10) {
        #pragma omp critical
        nulls += 1;
    }
}
```

Reductions

```
float sum = 0.0f;

#pragma omp parallel for reduction(+:sum)
for (int i=0; i<N; ++i) {
    sum += std::log(values[i]);
}
```

MPI

- Multi-language communication library
- Allows to exchange data between processes
- Used on (nearly) all supercomputers

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

print("Hello, I'm node", rank)
```

Running jobs

- The `mpiexec` command runs your executable several times
- You are not responsible for launching processes
- `rank` and `size` allow you to discover what MPI has done

```
mpiexec -np 4 python3 myprogram.py
```

Distributed computing

Host file:

```
user@host1 slots=4
user@host2 slots=8
user@host3 slots=4
```

Command:

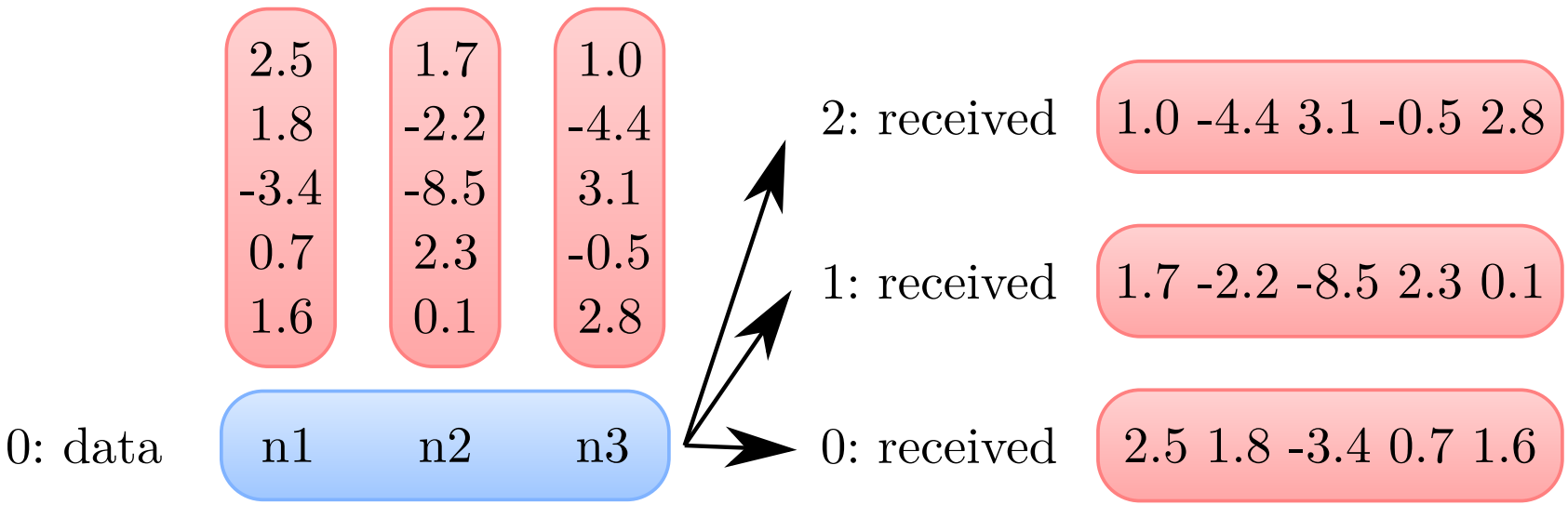
```
mpiexec -np 4 --hostfile hosts.txt python3 myprogram.py
```

Scatter

```
if rank == 0:
    data = [n1, n2, n3]
else:
    data = None

received = comm.scatter(data, root=0)
```

Scatter


2.5
1.8
-3.4
0.7
1.6
1.7
-2.2
-8.5
2.3
0.1
1.0
-4.4
3.1
-0.5
2.8
0: data
n1
n2
n3
2: received
1.0 -4.4 3.1 -0.5 2.8
1: received
1.7 -2.2 -8.5 2.3 0.1
0: received
2.5 1.8 -3.4 0.7 1.6

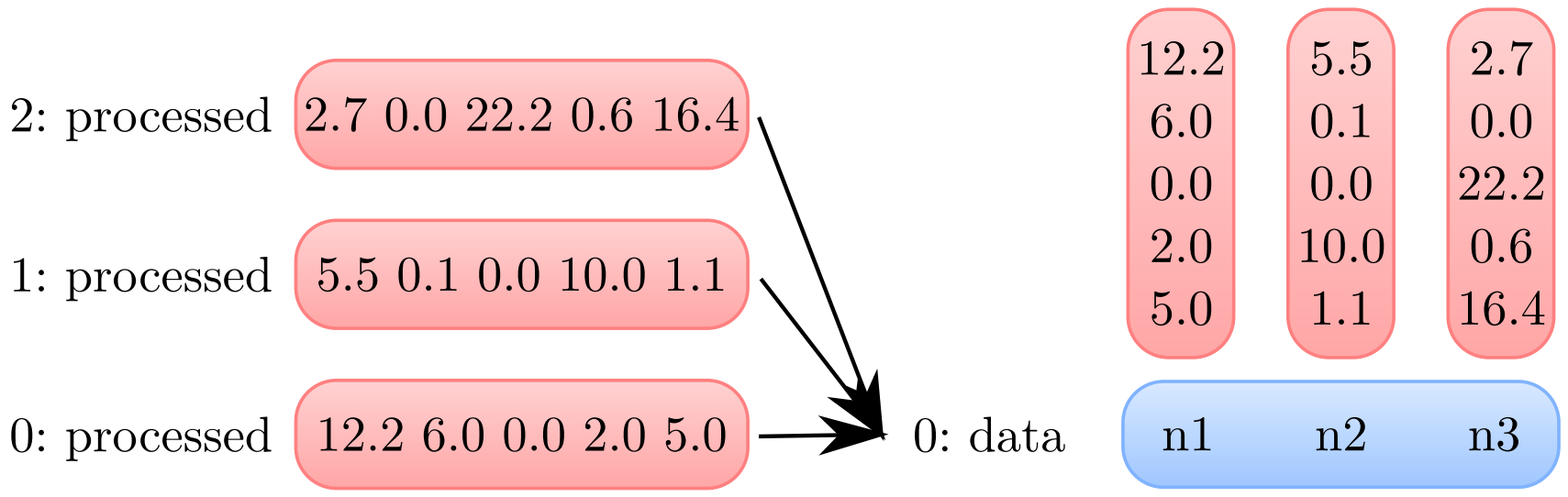
Gather

```
processed = [math.exp(i) for i in received]

data = comm.gather(processed, root=0)
n1, n2, n3 = data
```

`data` is now valid on node 0, equal to `None` on the other ones.

Scatter


2: processed
2.7 0.0 22.2 0.6 16.4
1: processed
5.5 0.1 0.0 10.0 1.1
0: processed
12.2 6.0 0.0 2.0 5.0
0: data
12.2
6.0
0.0
2.0
5.0
5.5
0.1
0.0
10.0
1.1
2.7
0.0
22.2
0.6
16.4
n1
n2
n3

Conclusion

- Many ways to parallelize
- MPI does it all but requires much work
- MPI is slower than threads
  - Use threads or OpenMP at each node, MPI for inter-node communication
- Most high-level tools already parallelized

Small example: RL

An RL agent using Q-Learning spends time doing:

microseconds	Matrix multiplications in neural networks (Q function)
milliseconds	Gradient descent steps
milliseconds	Time-steps, full of $a = \text{argmax}_a Q(s_t, a)$
seconds/minutes	Episodes
minutes/hours	Runs for different parameters

Small example: RL

What can be parallelized:

Matrices	GPU for very large matrices only
Gradient descent	Threads, then average
Time-steps	Impossible to parallelize
Episodes	MPI if learning still possible (we can talk about that)
Runs	GNU Parallel, up to one computer per run configuration