Parallel programming is ~~not~~ easy

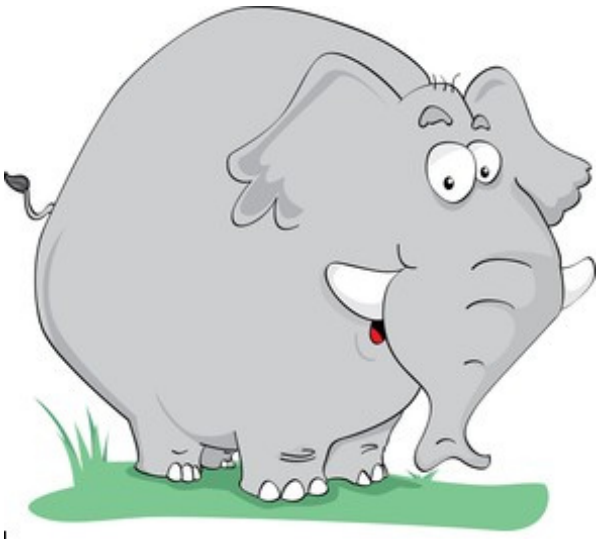
Lukas van de Wiel

April 4, 2024

Problem description

A single thread computation...

...does not fit in memory



...takes too long

Problem description

A single thread computation...

...does not fit in memory

...takes too long

The dilemma

Two ways to solve this:

spend time spend money

What does money get you?

CPU speed is limited, but memory is cheap (€5.30 / GB):

What does time get you?

Software optimization and parallellisation is all about time management. We face another choice between two options:

Optimizing the code

Work on something else while the program runs

Is the software suitable for parallellisation?

Can our algorithm be parallellised at all? Does the implementation have:

1. loops with independent operations?
2. independent processes?

Good news!

1. Iterative algorithm

Bad news!

Is the hardware suitable for parallellisation?

When did we first get parallel hardware?

- first compute cluster: 1969 (ARPANET project)
- first multicore processors: 2001 (IBM Power 4)

What is the state of parallel hardware today?

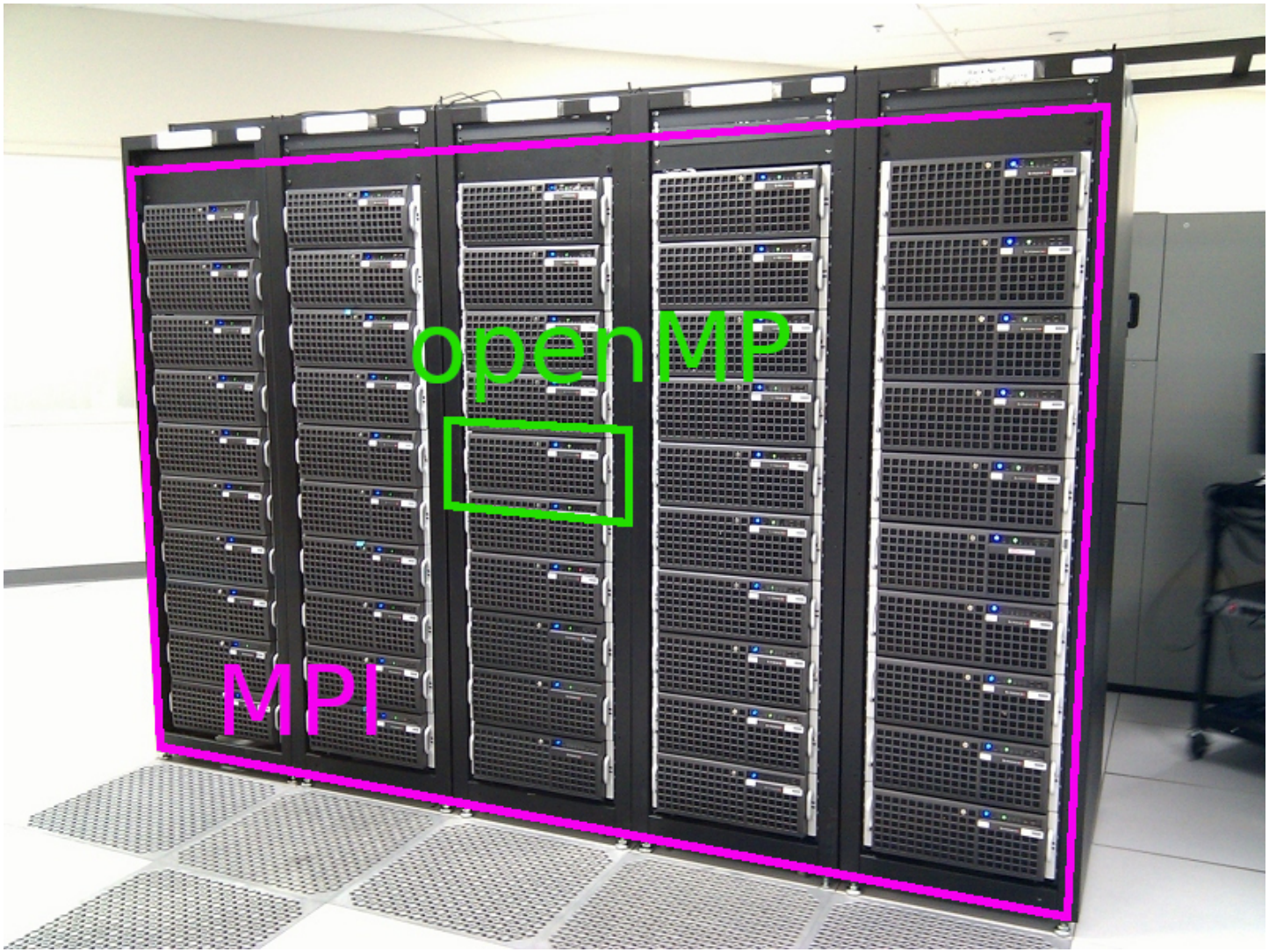
- highest CPU core count processor: 128 (AMD EPYC 9754)
- highest GPU core count processor: 18.176 (NVIDIA RTX 6000 Ada)
- biggest cluster: $\sim 2 \times 10^5$ CPU cores (Summit)

Even phones?

- Yes: 10 cores (Samsung Galaxy S24; Exynos 2400)

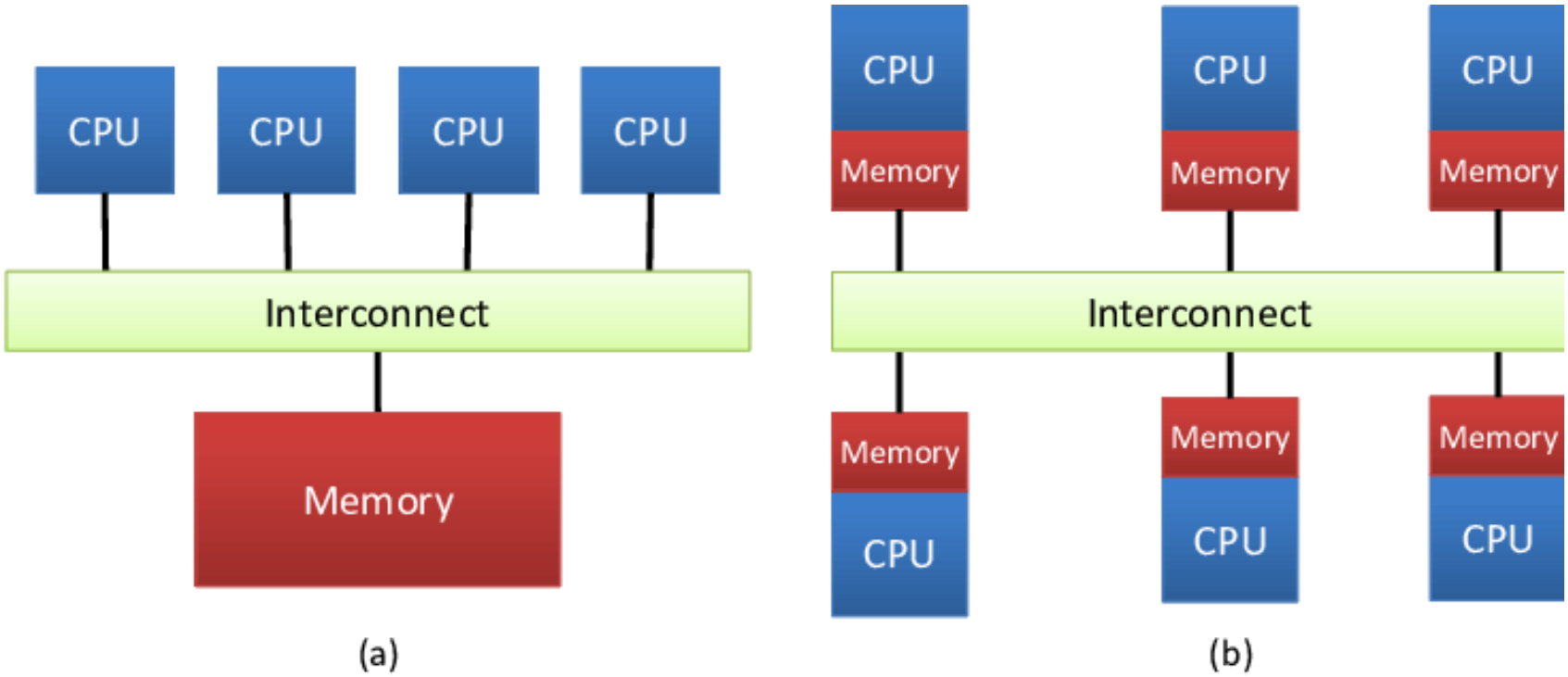
Single thread programming in 2024: Big Sad! 

How much do we need?



10g cluster from Jefferson Lab

openMP vs MPI



openMP (a) vs MPI (b). Source: Luciano Ost; *Adaptation Strategies in Multiprocessors System on Chip*

But how?

Let us look at some examples!

Without openMP. Serial. Slow:

```
integer                 :: i
double precision        :: a(800)

do i=1,800
  call doIndependentWork(a(i))
enddo
```

With openMP. Parallel. Fast:

```
use omp_lib

integer                 :: i
double precision        :: a(800)

!$omp parallel do

do i=1,800
  call doIndependentWork(a(i))
enddo

!$omp end parallel do
```

Python

Python suffers from GIL (Global Interpreter Lock): Only one thread can use the interpreter at any give time.

Python has no explicit openMP.

Parallel openMP-like functionality is used in C functions that are called through scipy/numpy/etc:

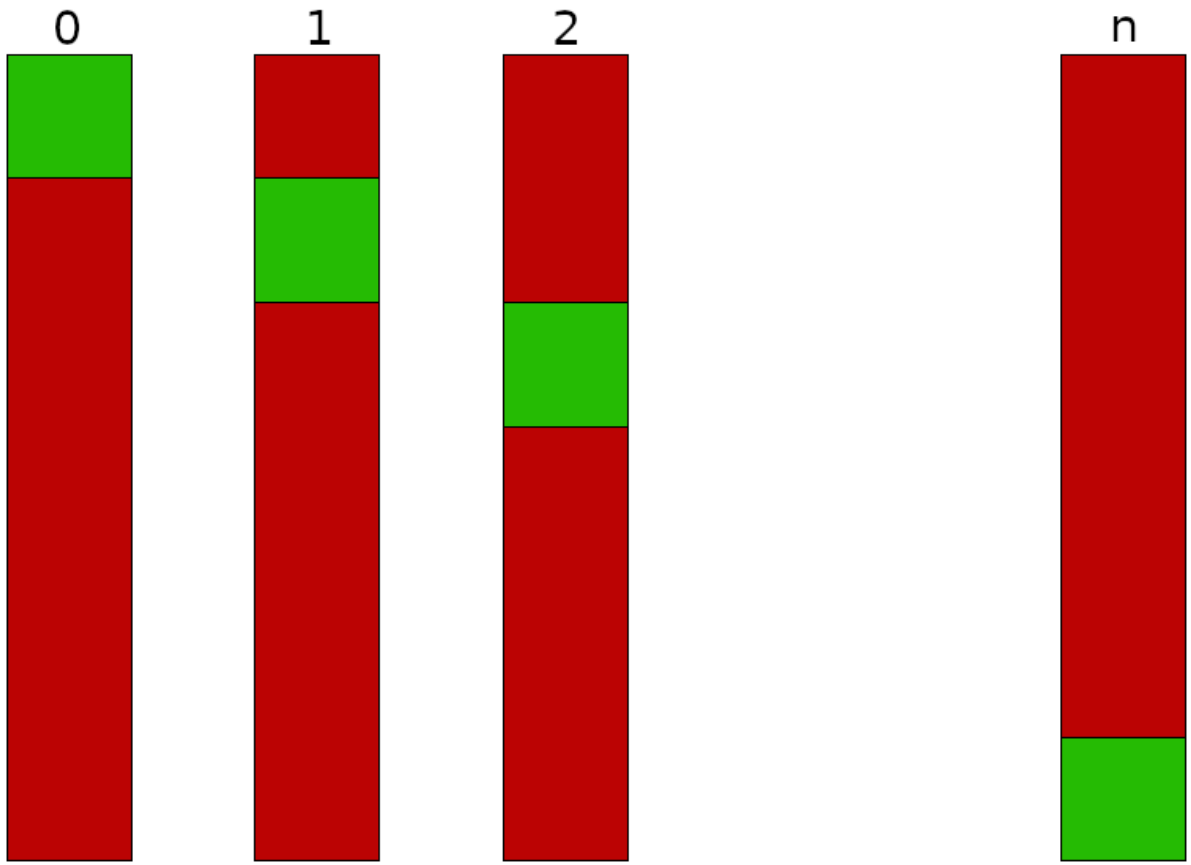
```
import numpy

size = 10000

m  = numpy.random.random((size,size))
mi = numpy.linalg.inv(m)
```

MPI, requires a bit more effort:

```
use mpi

integer                 :: i, iE, nProcs, myProcID, f
double precision        :: a(800)

call mpi_init(iE)
call mpi_comm_rank(MPI_COMM_WORLD, myProcID, iE)

f = myProcID * 100 + 1

do i = f, f+99
  call doIndependentWork(a(i))
enddo

call mpi_finalize(iE)
```

MPI, a more realistic example:

But wait, we cheated! We ignored the 'distributed memory' aspect of openMPI. Each thread had all the input data (a) and only its own data computed, but no single thread has all the output:

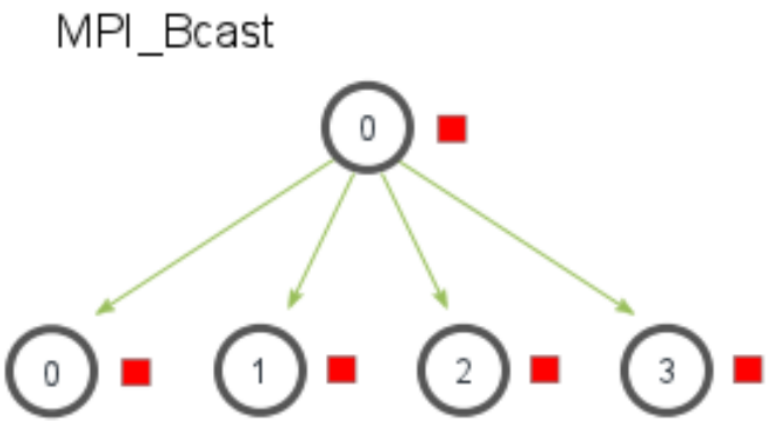


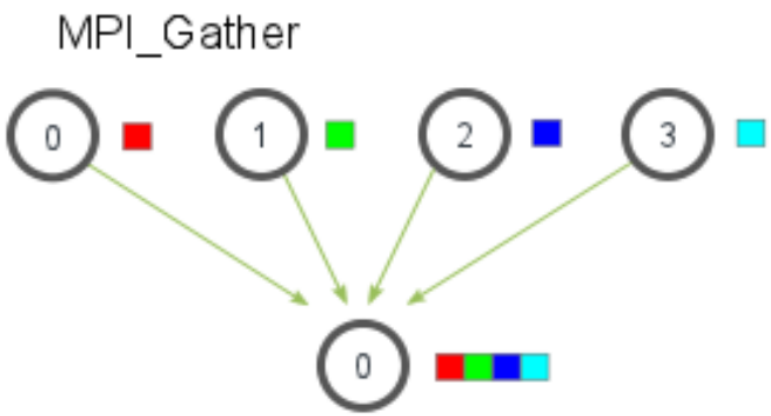
MPI, a more realistic example:

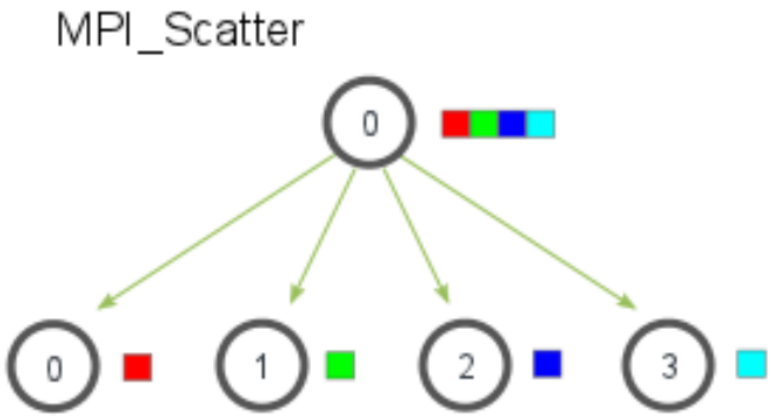
Real life situations are typically of the workflow:

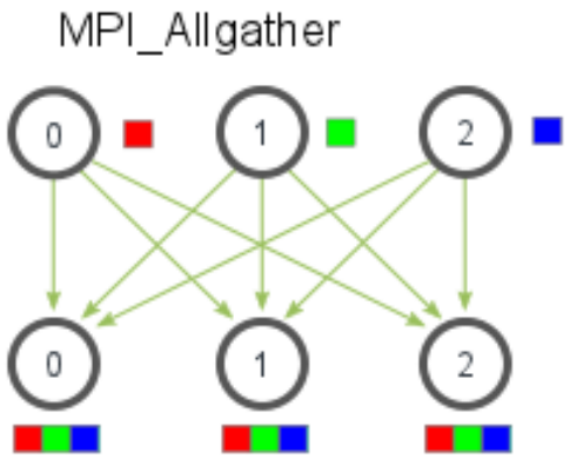
1. Thread 0 reads input data from file.
2. Input data is distributed over all the threads.
3. All threads do their computations in parallel.
4. Results are copied back from each thread to thread 0.
5. Thread 0 writes output data to file.

So we need to copy data









source: `https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/`

Python: the gentle version

```
from mpi4py import MPI
import numpy
globIn  = numpy.zeros(80)
globOut = numpy.zeros(80)
loc     = numpy.zeros(10)

rank = MPI.COMM_WORLD.Get_rank()

if rank == 0:
    globIn = numpy.loadtxt("inputData.txt")

MPI.COMM_WORLD.Scatter(globIn, loc, root=0)
loc = 10 * loc
MPI.COMM_WORLD.Gather(loc, globOut, root=0)

if rank == 0:
    numpy.savetxt("outData.txt", globOut)
```

MPI, MPI_Scatter

```
MPI_Scatter(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator,
    int error)
```

MPI, MPI_Gather

```
MPI_Gather(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator,
    int error)
```

MPI, Python

```
import mpi4py
```

has a nice little extra!

send and receive datatype is no longer bound to MPI_Datatypes.

Any object can be scattered and gathered!

MPI, Python, one little danger

Remember the implicit parallellisation of the matrix inversion?

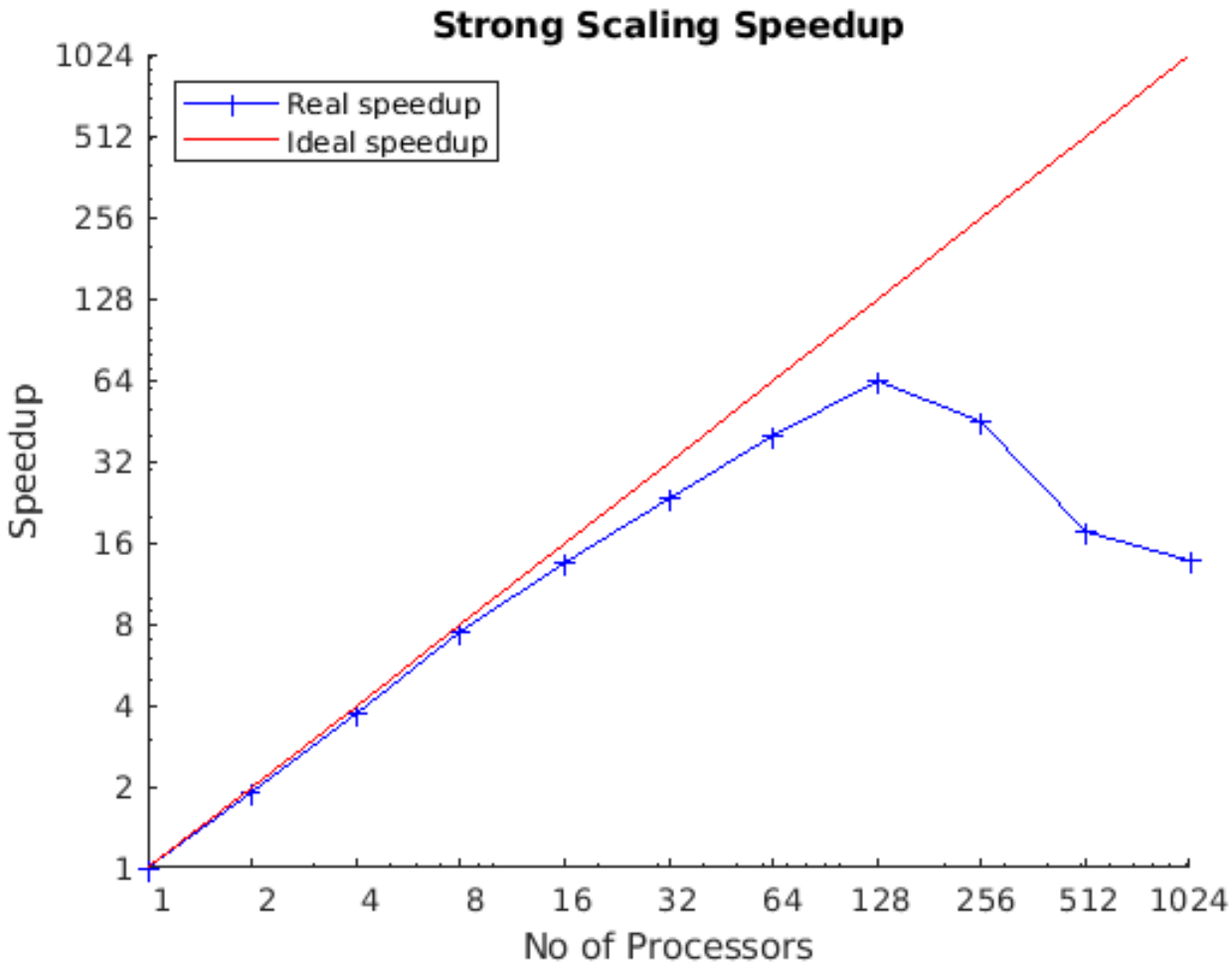
```
> mpirun -np 48 python3 myCoolCode.py
```

Thread 0 thinks: I have 48 processors for my openMP processes!
Thread 1 thinks: I have 48 processors for my openMP processes!
Thread 2 thinks: I have 48 processors for my openMP processes!
Thread 3 thinks: I have 48 processors for my openMP processes!
Thread 4 thinks: I have 48 processors for my openMP processes!
...

Scaling test

No scaling test? Not finished!

Scaling test



source: `https://hpc-wiki.info`

Conclusion

We have learned three things here:

1. Only go parallel when you have to. If you can get away with spending a small amount of money, or if you have other work to do while the software runs, that is the wiser option.
2. Take the easiest route of parallellisation. If OpenMP is enough, add a few tags and be done. If not, go full MPI.
3. Always test your scaling and find the optimal number of cores to run on.

The end

Thread 0 says: Thank you for your time

Thread 1 says: Thank you for your attention

Thread 2 says: Find all material at:
`github.com/LukasvdWiel/parProgIsEasy`