

MPI Performance

Pedro Ojeda, Joachim Hein

HPC2N, Umeå University

LUNARC & Centre of Mathematical Sciences

Lund University

Speedup and efficiency

Let us call T_{serial} the time a program takes by using a single core and T_{parallel} the time it takes by using p processors, the speedup is defined by:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

And the efficiency by:

$$E = \frac{S}{p} = \frac{T_{\text{serial}}}{p * T_{\text{parallel}}}$$

Can we achieve $T_{\text{parallel}} = T_{\text{serial}}/p$?

Speedup, efficiency and cost

Let us call T_{serial} the time a program takes by using a single core and T_{parallel} the time it takes by using p processors, the speedup is defined by:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

And the efficiency by:

$$E = \frac{S}{p} = \frac{T_{\text{serial}}}{p * T_{\text{parallel}}}$$

The cost is:

$$C = p * T_{\text{parallel}}$$

In general, $T_{\text{parallel}} = \frac{T_{\text{serial}}}{p} + T_{\text{overhead}}$

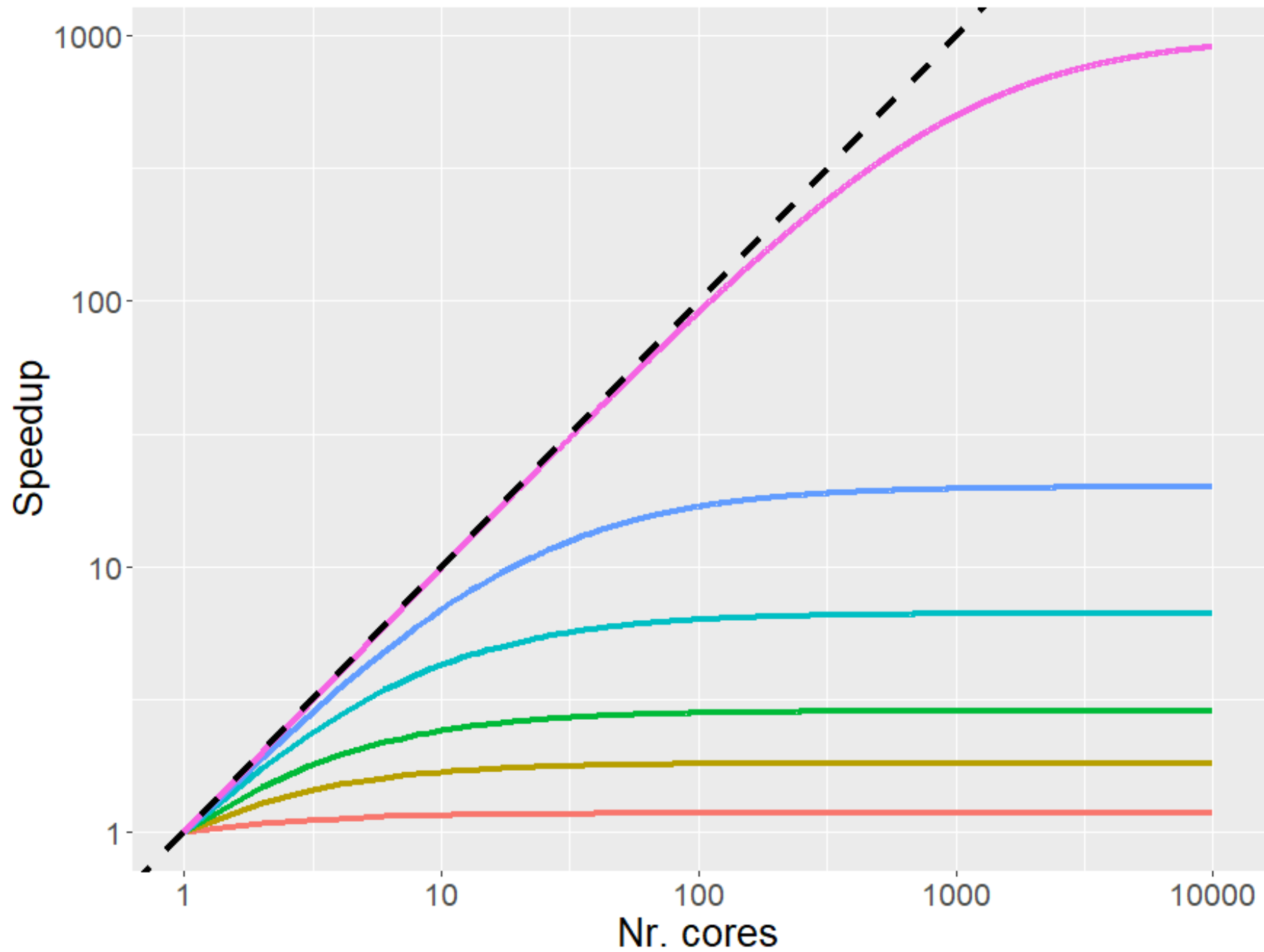
Amdahl's law

Let us call the proportion of the code that can be parallelized by τ , the parallel time using p processors can be written as,

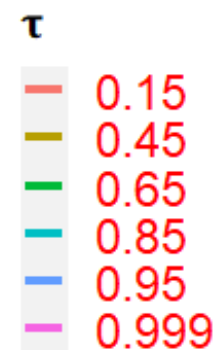
$$T_{\text{parallel}} = \tau * \frac{T_{\text{serial}}}{p} + (1 - \tau) * T_{\text{serial}}$$

Then, the speedup will be

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{1}{1 - \tau * \left(1 - \frac{1}{p}\right)}$$



$$\lim_{p \rightarrow \infty} S = \frac{1}{1 - \tau}$$



This behavior is known as **strong scaling**

Speedup in practice

For a certain code the speedup is computed as follows:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

Where T_{serial} is the simulation time with a single core and T_{parallel} is the time obtained by using 1,2,...p processors.

Gustafson's law

In general, the goal of parallelizing a code is not only to use a large number of cores but also to simulate larger problems in size.

J. L. Gustafson reevaluated the Amdahl's law¹ for problems with a variable size and gave an alternative based on E. Barsis suggestion:

$$\text{Scaled speedup} = 1 + \tau * (p - 1)$$

where τ and p have the same meaning as before. Scaled speedup has a linear dependence with the number of processors. This is known as the **weak scaling** behavior.

¹Communications of the ACM, J. L. Gustafson (1988)

Scaled speedup in practice

For a certain code the Scaled speedup is computed as follows:

$$\text{Scaled speedup} = \frac{p * T_{\text{serial}}}{T_{\text{parallel}}}$$

Where T_{serial} is the simulation time with a single core and T_{parallel} is the time obtained by using 1,2,...p processors while changing the size of the simulated problem.

Timing in MPI

The function `MPI_Wtime()` allows you to set a reference point in time, a time interval can be obtained by taking the difference between two calls of this function.

In C:

```
double MPI_Wtime()
```

In Fortran 90:

```
double precision MPI_WTIME()
```

The time resolution of the hardware can be obtained with `MPI_Wtick()`

Timing in MPI

The function `MPI_Wtime()` allows you to set a reference point in time, a time interval can be obtained by taking the difference between two calls of this function.

In Python:

`MPI.Wtime()`

Returns a float.

The time resolution of the hardware can be obtained with `MPI.Wtick()`

Timing in MPI

Example in Fortran 90:

```
double precision begin_t, end_t

begin_t = MPI_WTIME()

... code to be monitored

end_t = MPI_WTIME()

print *, 'time ', end_t - begin_t, 'sec.'
```

Example in C:

```
double begin_t, end_t;

begin_t = MPI_Wtime();

... code to be monitored

end_t = MPI_Wtime();

printf("time %1.2f sec.", end_t - begin_t);
```

Timing in MPI

Example in Python:

```
from mpi4py import MPI

# get MPI communicator
my_world = MPI.COMM_WORLD

begin_t = my_world.Wtime()

# ... code to be monitored

end_t = my_world.Wtime()

print(f"time {end_t - begin_t:.2f} sec.")
```

Choice of algorithms

Consider the problem of computing the matrix vector multiplication $\mathbf{M}_{n \times n} \mathbf{b}_n = \mathbf{c}_n$ using a parallel algorithm. The total number of operations is $2n^2 - n$ and the total time for computation is $(2n^2 - n) * T_{10p}$ (T_{10p} being the time for a single operation).

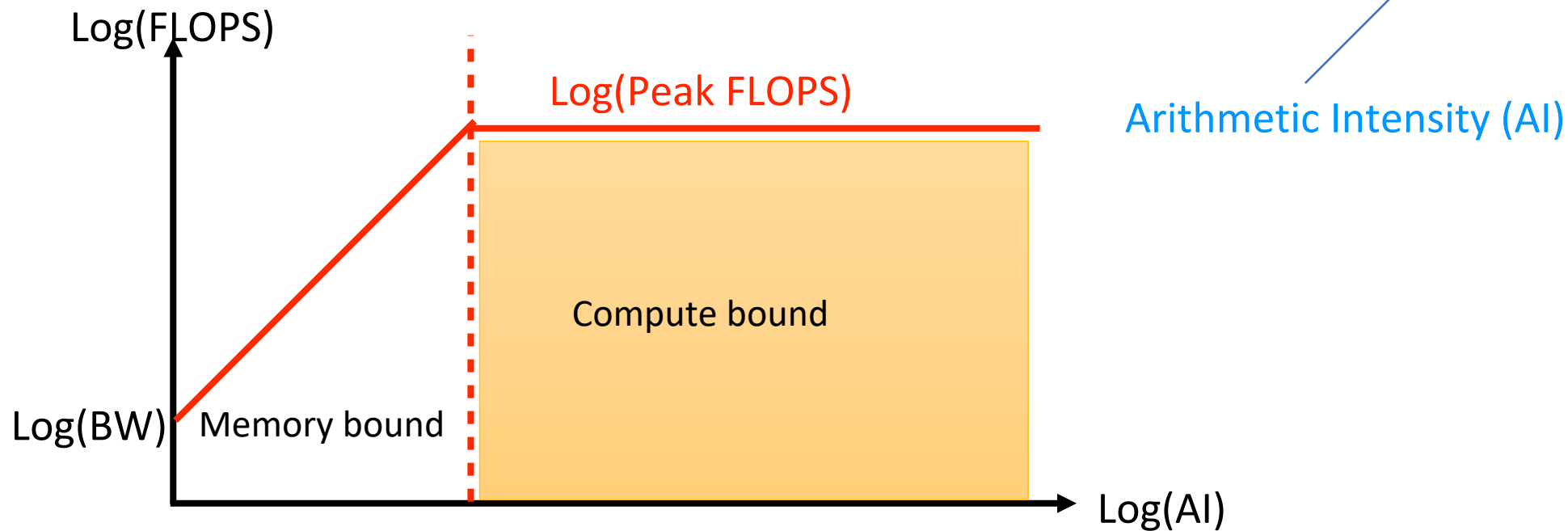
If each row is transferred to the processors (for each element of \mathbf{c}) and the result is transferred back to the master, the total time spent in data movement is $(n^2 + n) * T_{comm}$ (T_{comm} being the time for transfer a single element). Thus, the ratio of computation and communication is

$$r = \left(\frac{n^2 + n}{2n^2 - n} \right) * \left(\frac{T_{comm}}{T_{10p}} \right) \quad \text{and} \quad \lim_{n \rightarrow \infty} r \propto \frac{1}{2}$$

Measuring code performance

Bandwidth (BW)

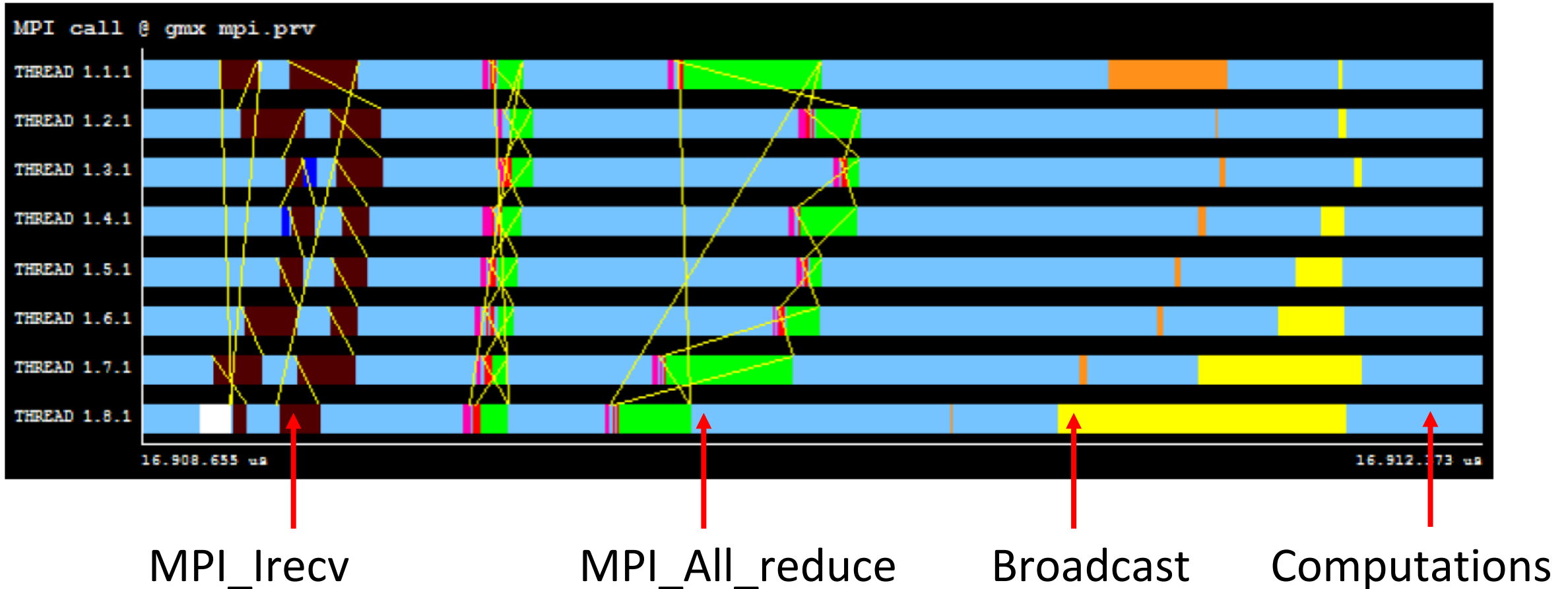
$$\text{Floating Point Operations per second (FLOPS)} = \frac{Nr.FLOP}{1sec} = \frac{Nr.FLOP}{Byte} \times \frac{Byte}{sec}$$



More details: <https://www.telesens.co/2018/07/26/understanding-roofline-charts/>

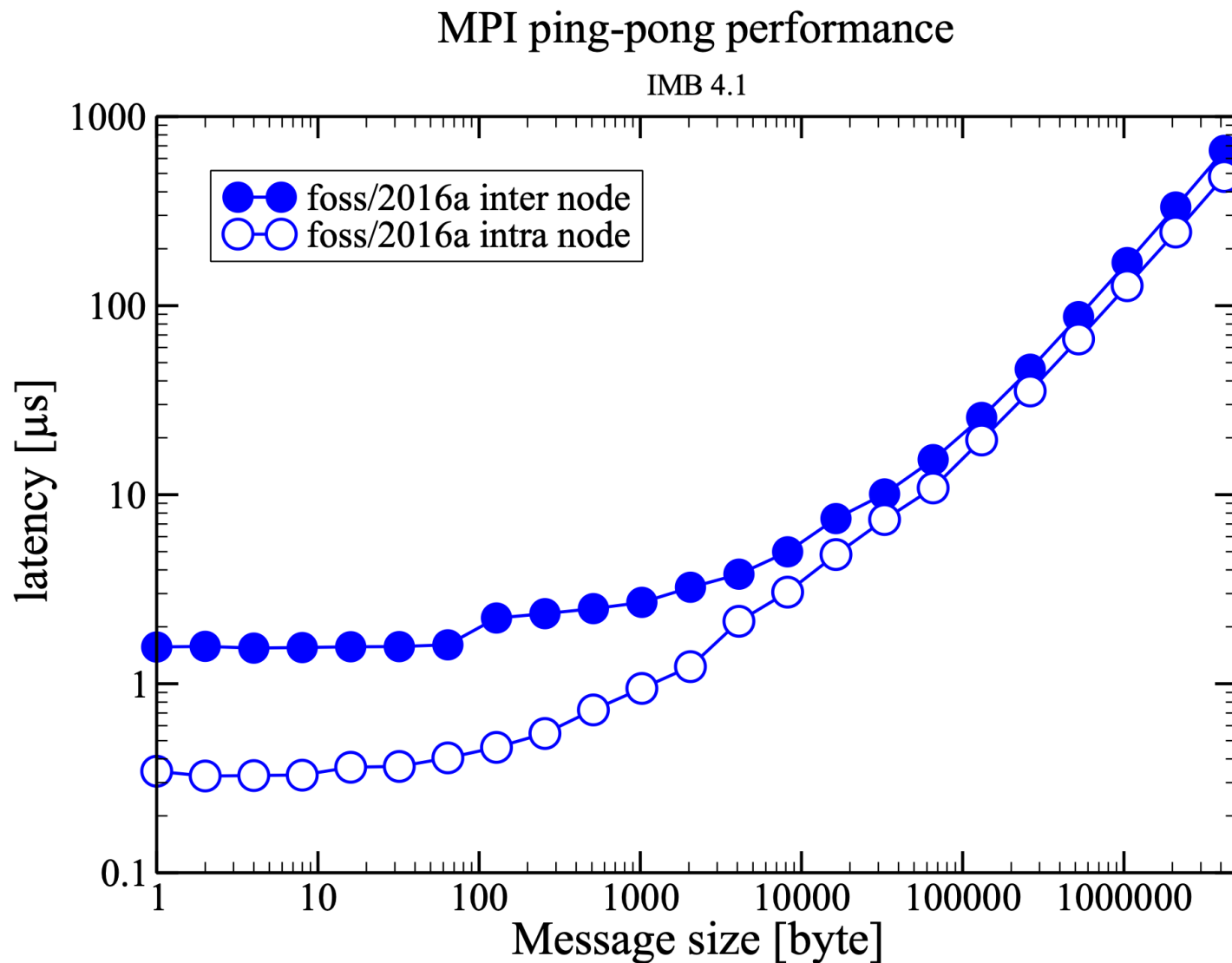
Overhead in MPI programs

$$T_o = p * T_{\text{parallel}} - T_{\text{serial}}$$



MPI profile from a GROMACS simulation obtained with Extrae/Paraver showing the MPI calls metric in *Profiling and Tracing Tools for Performance Analysis of Large Scale Applications*, J. Eriksson, et. al. (<https://prace-ri.eu/wp-content/uploads/WP237.pdf>)

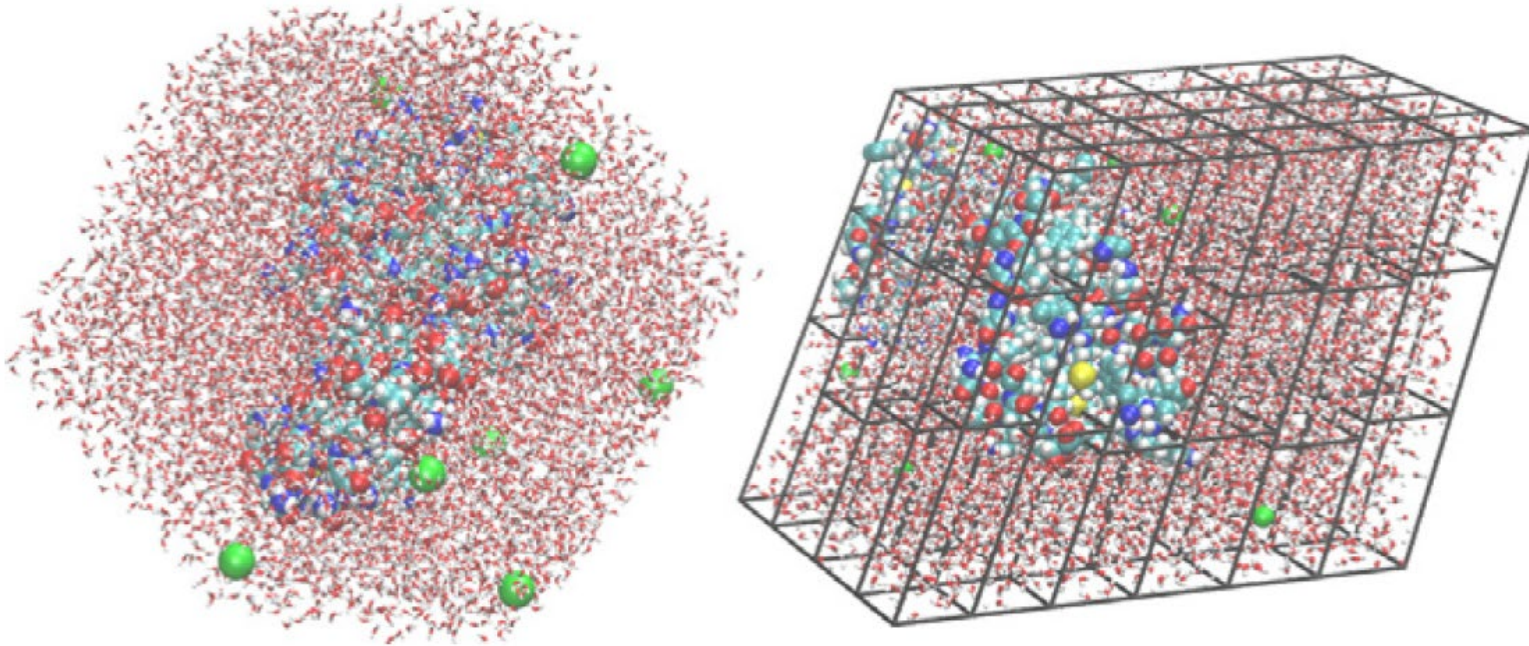
Typical MPI Point-to-Point performance: Aurora



MPI Point-to-Point performance

- Typical latency (minimum time to send anything):
 - between 1 and 5 μsec via network
 - around 1 μsec inside node (shared memory/cache)
 - these are $O(10000 \text{ cycles})$ - an eternity!
- Modern hardware transfers several GB per second
 - min message size for that $\approx 10 \text{ kB}$
- For message smaller than $\approx 1 \text{ kB}$
 - time (almost) independent of message size
- Aim to **aggregate your messages**
 - Always try to send more than a few numbers
 - Still don't bloat

Load imbalance



Source: *GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers*, M. J. Abraham, *SoftwareX*, 1, 19, (2015)

Fig. 2. *Left:* The protein lysozyme (24,119 atoms) in a compact unit cell representation corresponding to a rhombic dodecahedron with close-to-spherical shape. *Right:* Internally, this is represented as a triclinic cell and a load-balanced staggered $6 \times 4 \times 3$ domain decomposition grid on 72 MPI ranks. The PME lattice sum is calculated on a uniform grid of 6×4 MPI ranks (not shown).

Profiling tools

- Extrae/Paraver (<https://tools.bsc.es/paraver>)
- Scalasca (www.scalasca.org)
- Intel tools
- PRACE white paper (<https://prace-ri.eu/wp-content/uploads/WP237.pdf>)

Summary

- Some performance metrics were studied in this section: Speedup, Efficiency, Cost, Overhead, and Scaled speedup. Depending on the parallelization goals, one of them could be more appropriate than the others.
- Strong and weak scaling concepts were introduced.
- The choice of the parallel algorithm is of major importance.
- Regarding data transfer, always aim to send fewer but larger messages

Exercise

- Use the provided code for the 2D integration to get a plot of the Speedup (keep the number of bins constant and change the number of processors)