MPI Performance

Pedro Ojeda, Joachim Hein
HPC2N, Umeå University
LUNARC & Centre of Mathematical Sciences
Lund University

Speedup and efficiency

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

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

And the efficiency by:

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

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

Speedup, efficiency and cost

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

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

And the efficiency by:

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

The cost is:

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

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

An Introduction to Parallel Programming, Peter S. Pacheco

Amdahl's law

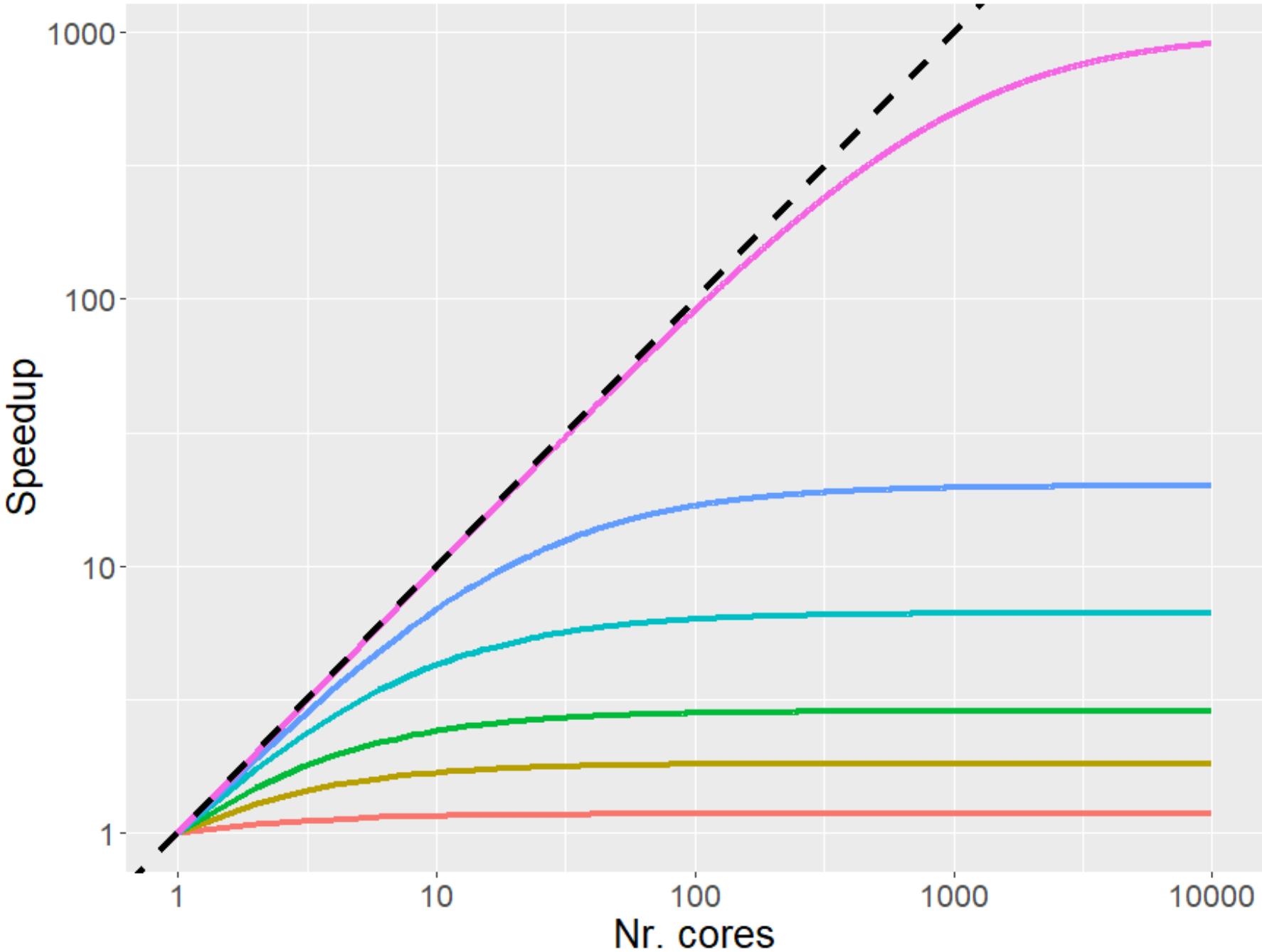
Let us call the proportion of the code that can be parallelized by $\boldsymbol{\tau}$, the parallel time using $\mathbf{p}$ processors can be written as,

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

Then, the speedup will be

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

AFIPS spring joint computer conference, 1976, Gene M. Amdahl



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

This behavior is known as **strong scaling**

Speedup in practice

For a certain code the speedup is computed as follows:

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

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

Gustafson's law

In general, we 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 law[1] for problems with a variable size and gave an alternative based on E. Barsis suggestion:

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

where $\boldsymbol{\tau}$ 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.

[1]Communications of the ACM, J. L. Gustaftson (1988)

Scaled speedup in practice

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

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

Where $\mathbf{T_{serial}}$ is the simulation time with a single core and $\mathbf{T_{parallel}}$ 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 to 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 to 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 $\mathbf{M_{n\times n}b_n} = \mathbf{c_n}$ using a parallel algorithm. The total number of operations is $\mathbf{2n^2 - n}$ and the total time for computation is $\mathbf{(2n^2 - n) * T_{1Op}}$ ($\mathbf{T_{1Op}}$ 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 $\mathbf{(n^2 + n) * T_{comm}}$ ($\mathbf{T_{comm}}$ being the time for transfer a single element). Thus, the ratio of computation and communication is

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

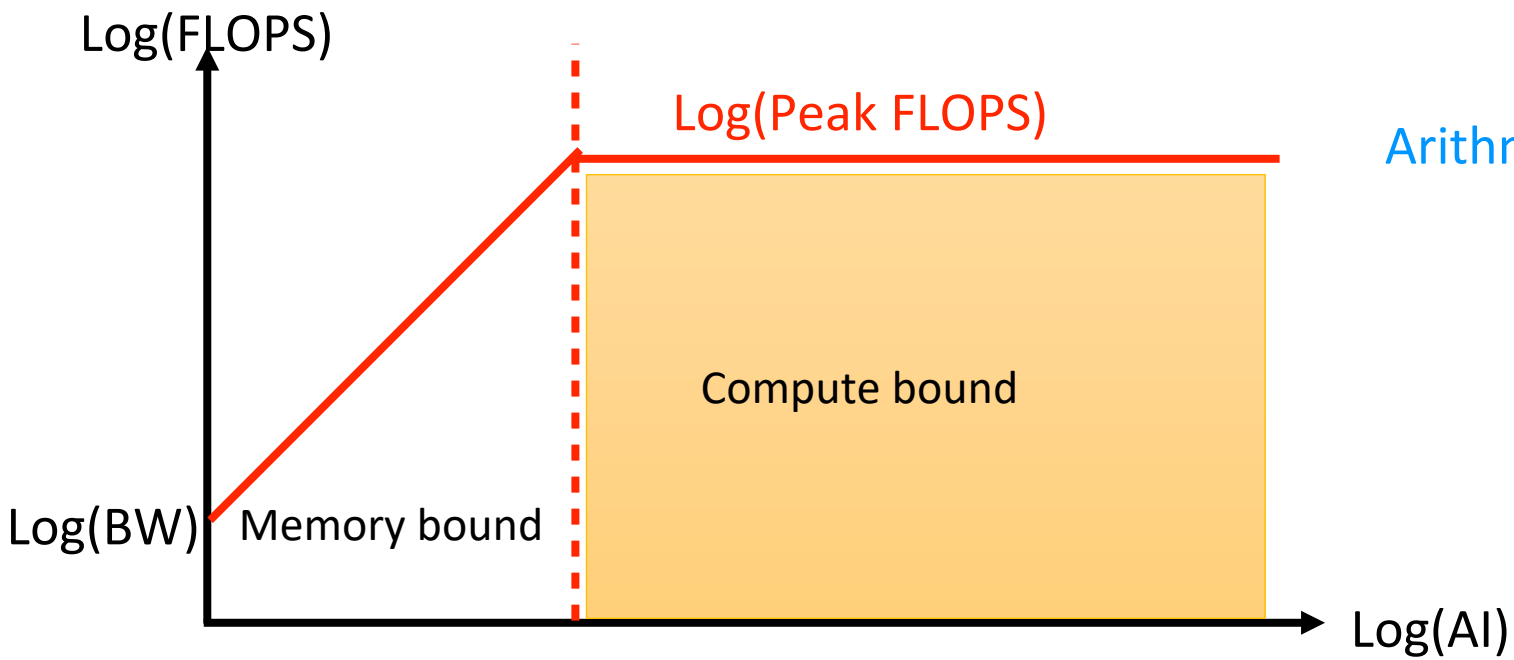
An Introduction to Parallel Programming, Peter S. Pacheco

Measuring code performance

Bandwidth (BW)

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

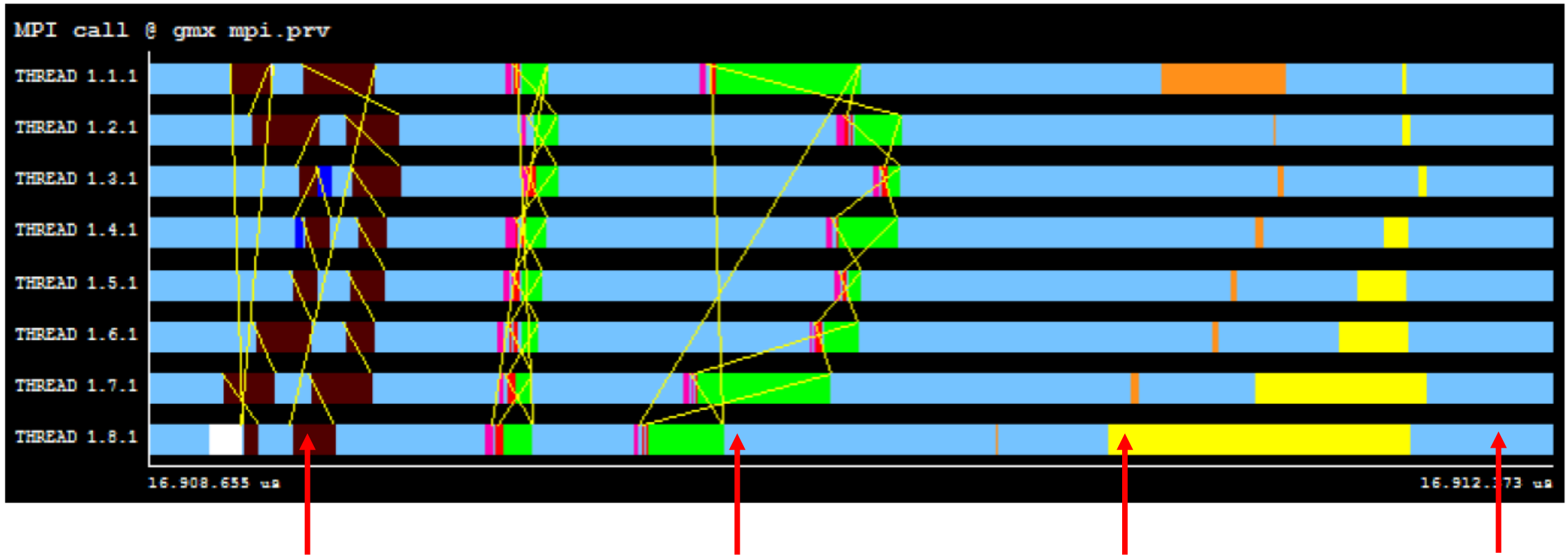
Arithmetic Intensity (AI)



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

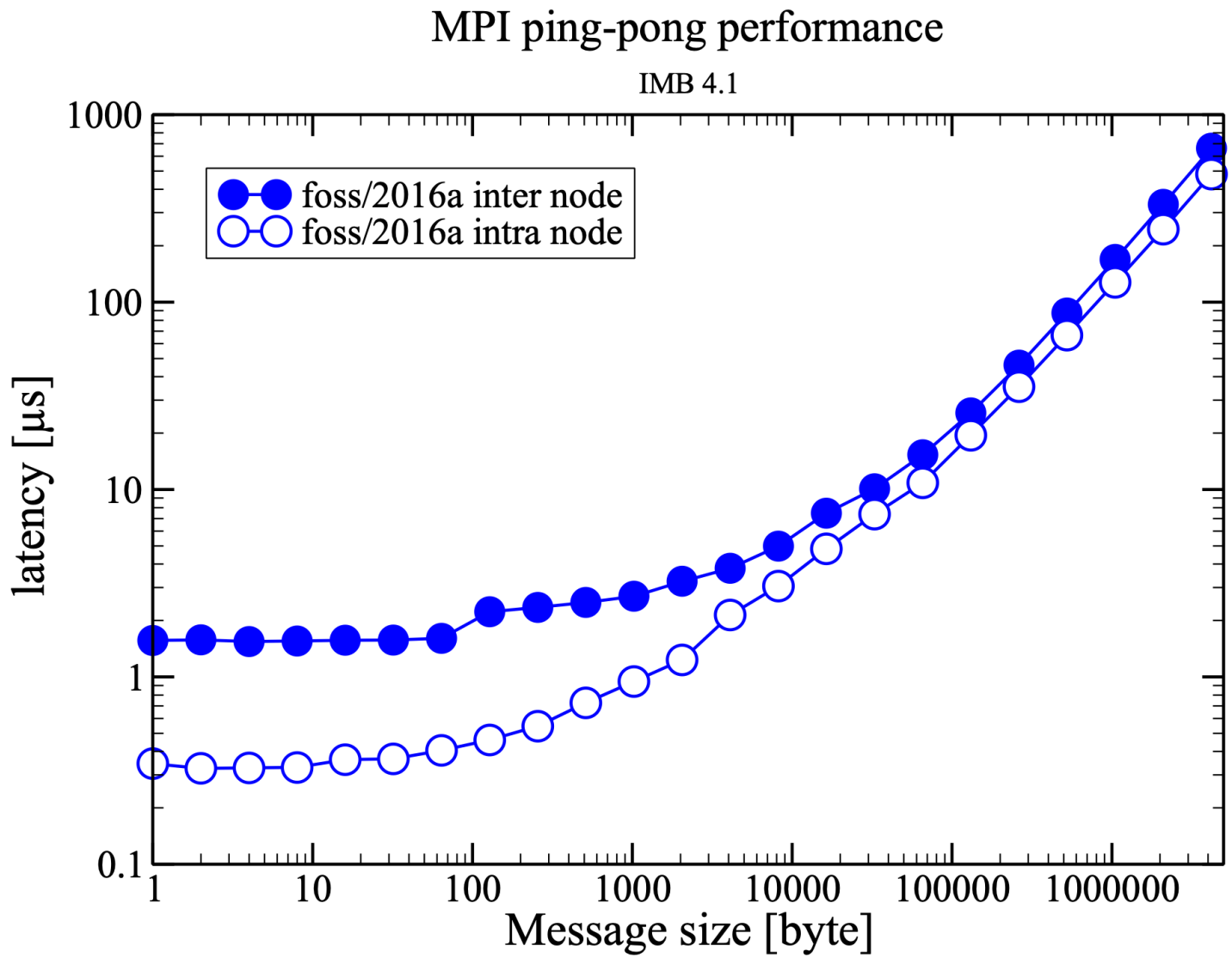
Overhead in MPI programs

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



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 cycles) - an eternity!
- Modern hardware transfers several GB per second
  - min message size for that ≈ 10 kB
- For message smaller than ≈ 1 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

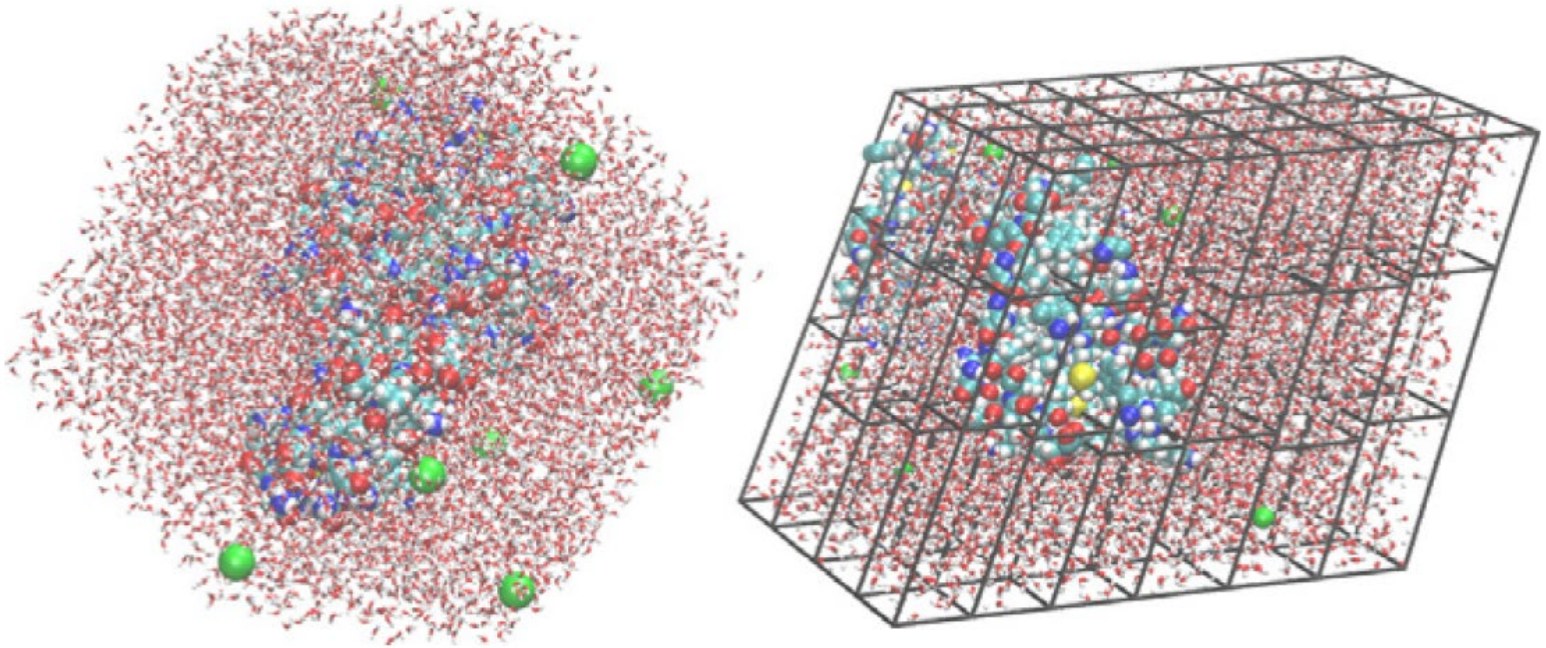


Fig. 2. *Left:* The protein lysozyme (24,119 atoms) in a compact unit cell representation corresponding to a rhombic dodecahedron with close-to-spherical shape. *Right:* Internally, this is represented as a triclinic cell and a load-balanced staggered $6 \times 4 \times 3$ domain decomposition grid on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

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

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)

- Obtain a plot of the Scaled speedup for the same code (change both the number of bins and the number of processors)