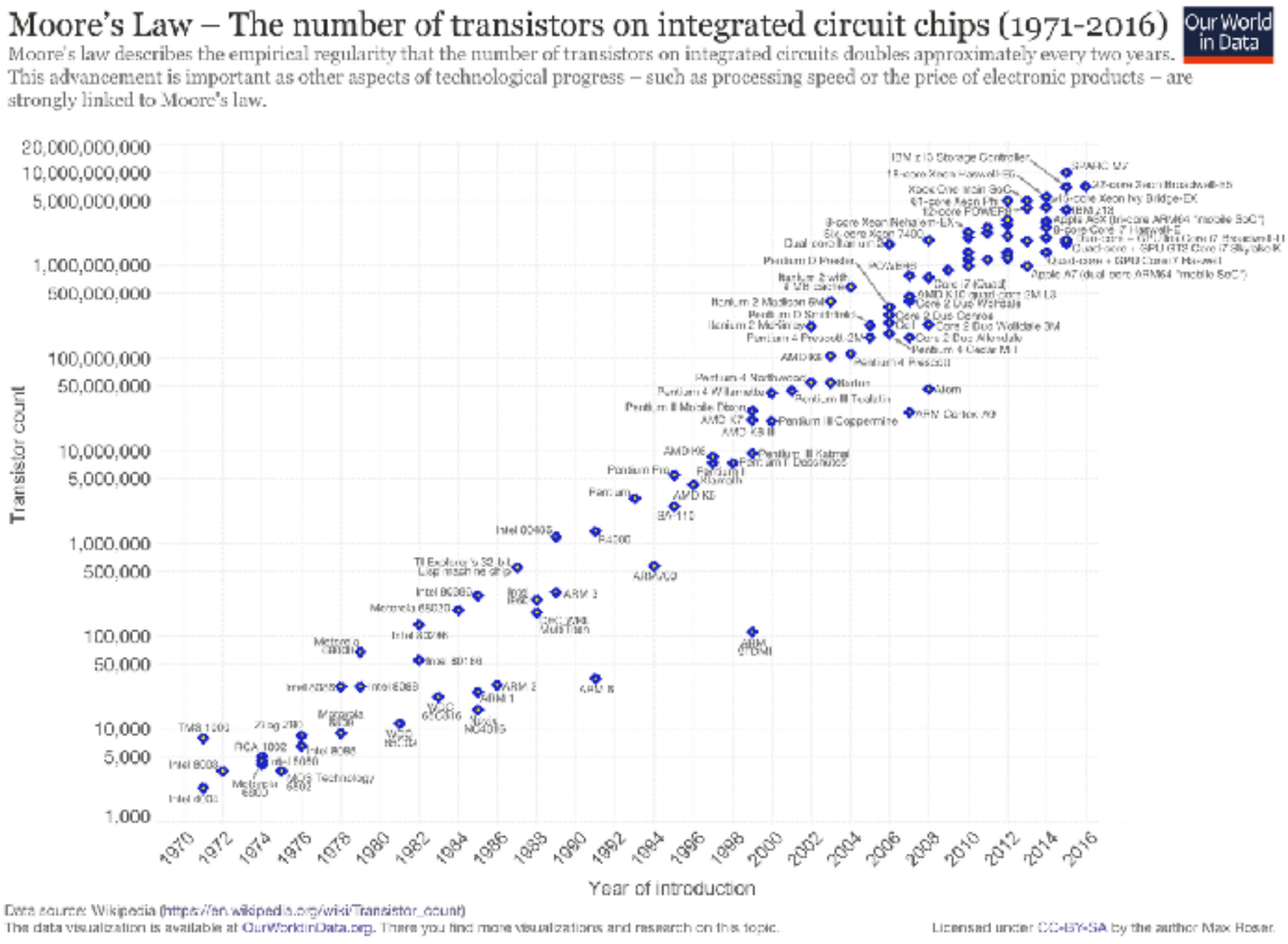
Introduction to High Performance Computing

J. R. C. C. C. Correia
Programmer's Club

Moore’s Law

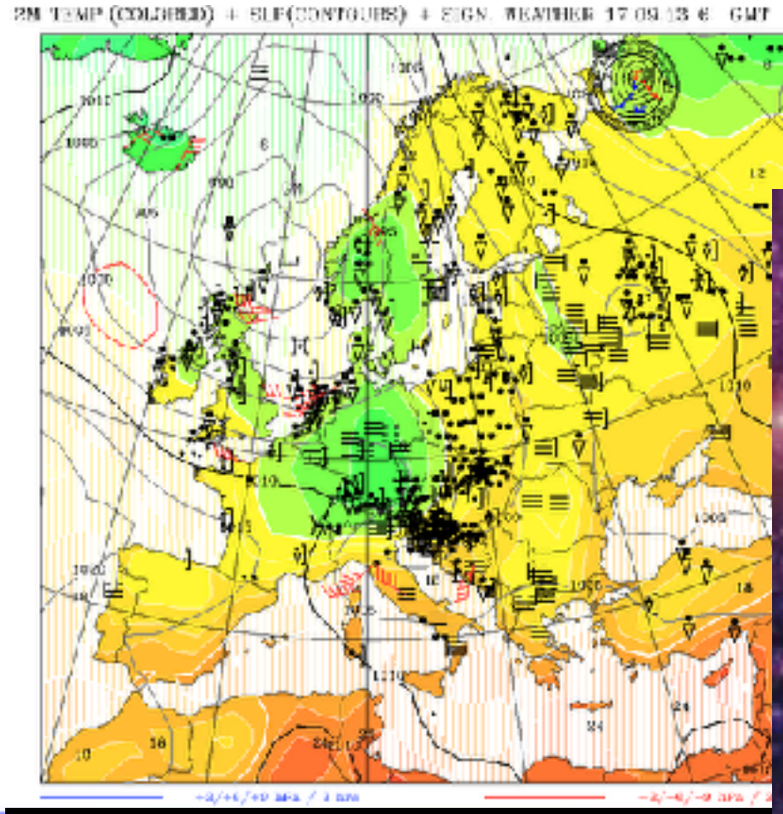

Moore’s Law – The number of transistors on integrated circuit chips (1971-2016)
Our World in Data
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.
Transistor count
20,000,000,000
10,000,000,000
5,000,000,000
1,000,000,000
500,000,000
100,000,000
50,000,000
10,000,000
5,000,000
1,000,000
500,000
100,000
50,000
10,000
5,000
1,000
1970
1972
1974
1976
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002
2004
2006
2008
2010
2012
2014
2016
Year of introduction
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.
Licensed under CC-BY-SA by the author Max Roser.

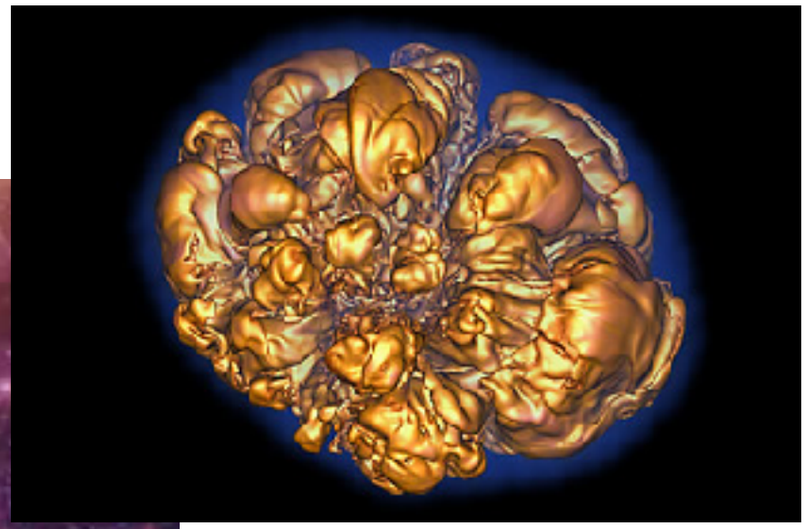
Glue logic

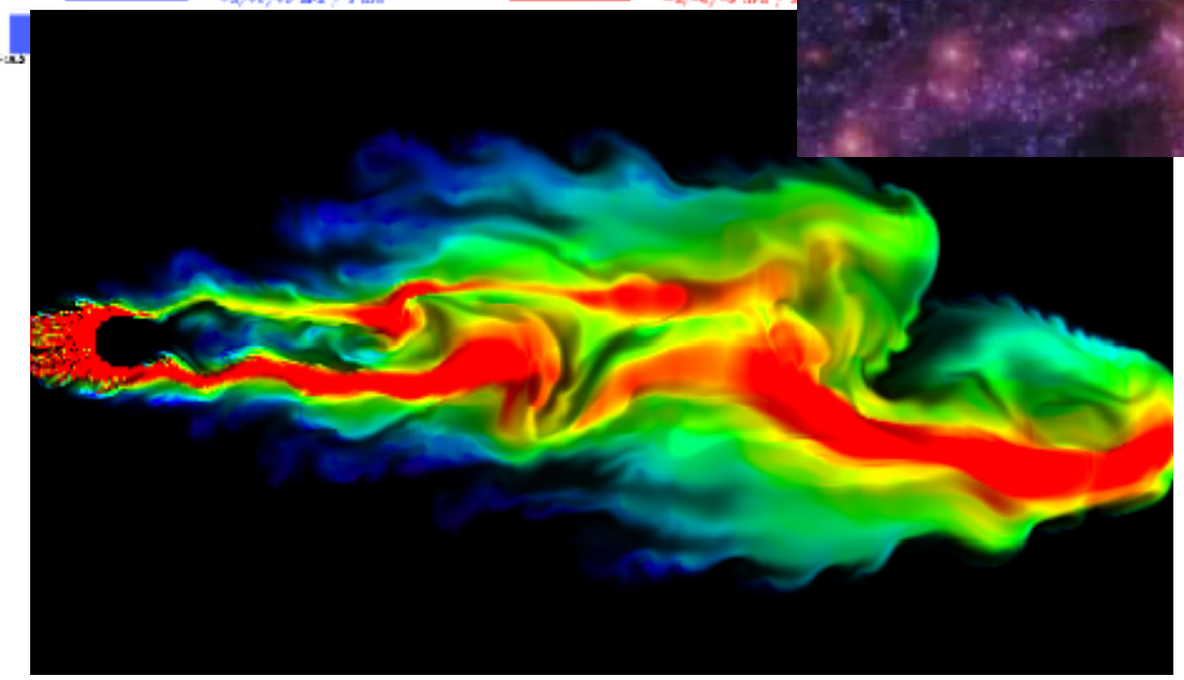
- Extra transistors:
  - more cache (small but fast-access memory);
  - more complex designs;
  - higher clock speeds;
- Eventually however, we hit the physical limitations of a uniprocessor design (size, power consumption).
- Solution: glue logic! With P processors, if a problem is split into P tasks that can be executed at the same time, ideally, this will P times faster.

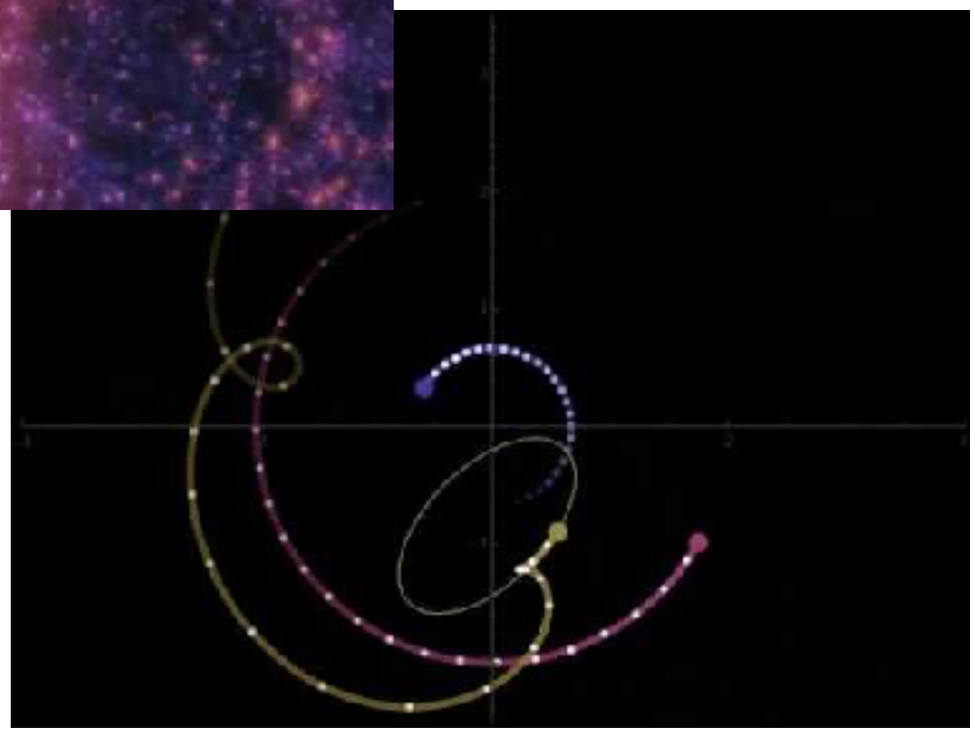
For consumer processors:

Scientific Computing







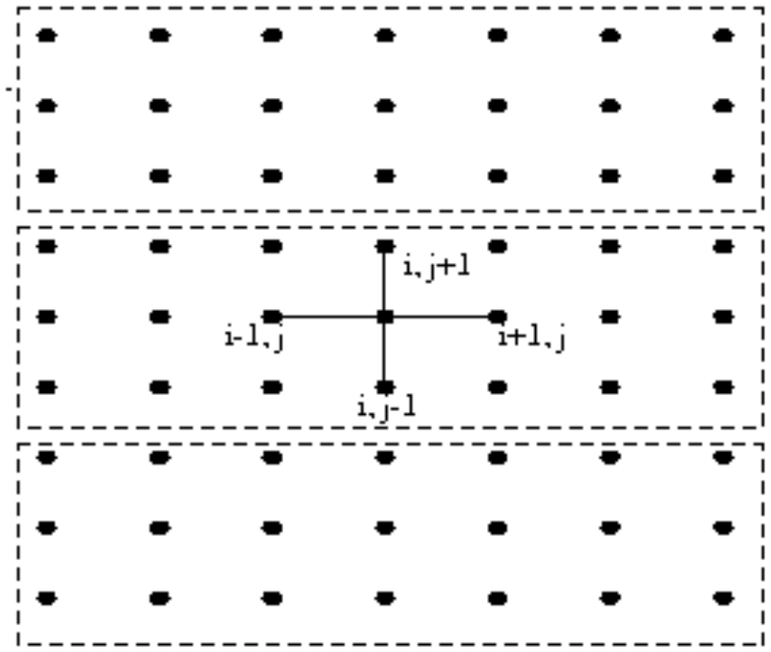


Parallel Computing

- ➡ The art of taking advantage of these architectures in order to solve computationally difficult problems

- Not everything can be parallelised. Even in the same problem, some tasks can be made parallel, others cannot. An example: an iterative solver using a 5-point stencil

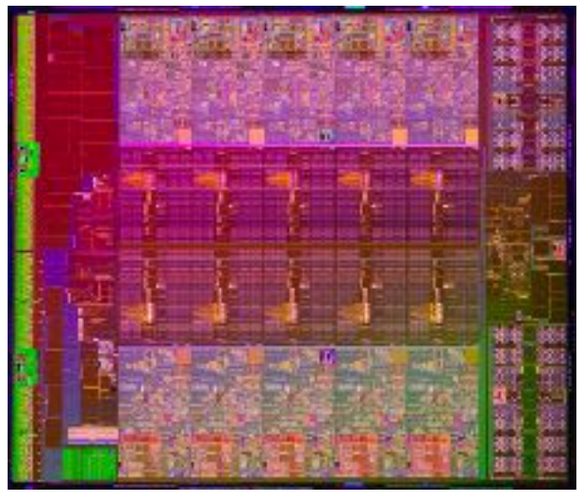
$$(\nabla^2\phi)_{i,j} = \phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j}$$



Architectures vs. Programming paradigms

- Modern supercomputers combine two types of architecture. The <u>first</u> one is,
  - <u>Shared memory</u>

Ex: Processors (see 10-core Xeon) or graphics cards

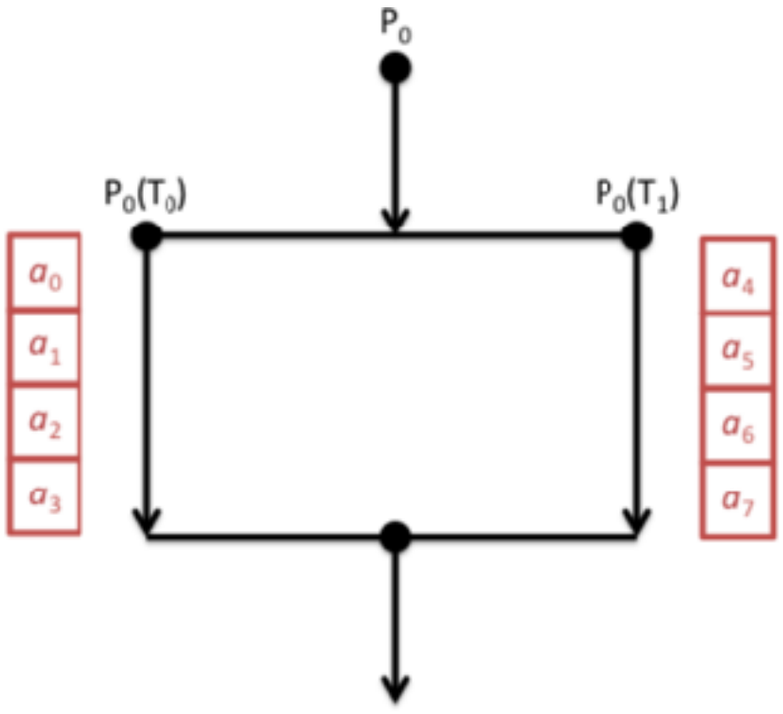


- Maps well to threaded programming:

[ARCHER Introduction to HPC]

Threaded programming

- A process (instance of program) walks into a bar and spawns two threads…



[ARCHER Introduction to HPC]

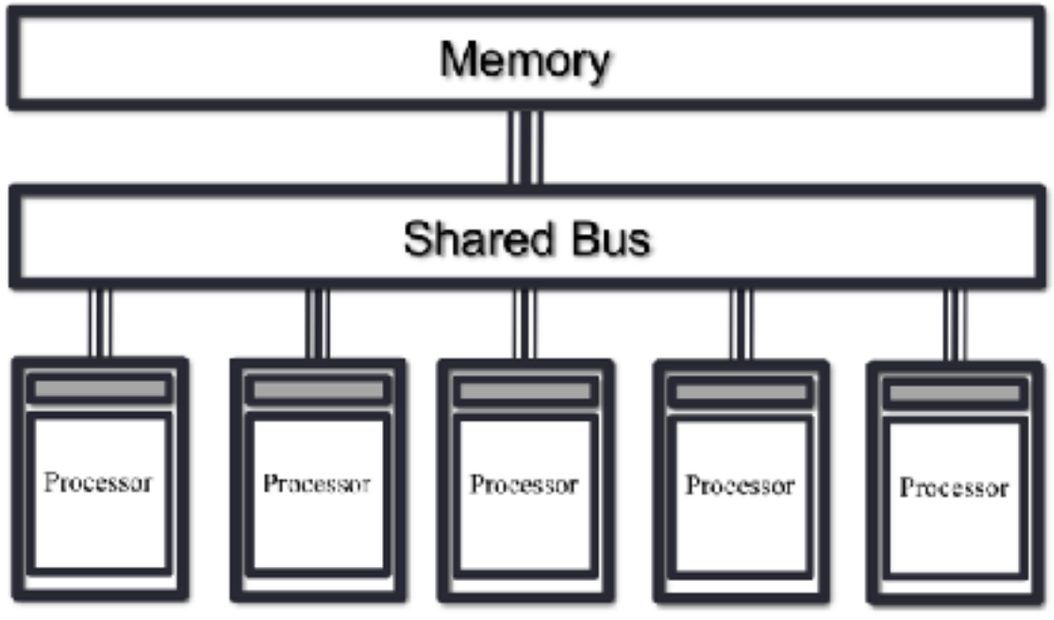
… and assigns half of an array of size eight to each thread.

- Each thread operates on half the data (potentially 2x as fast).

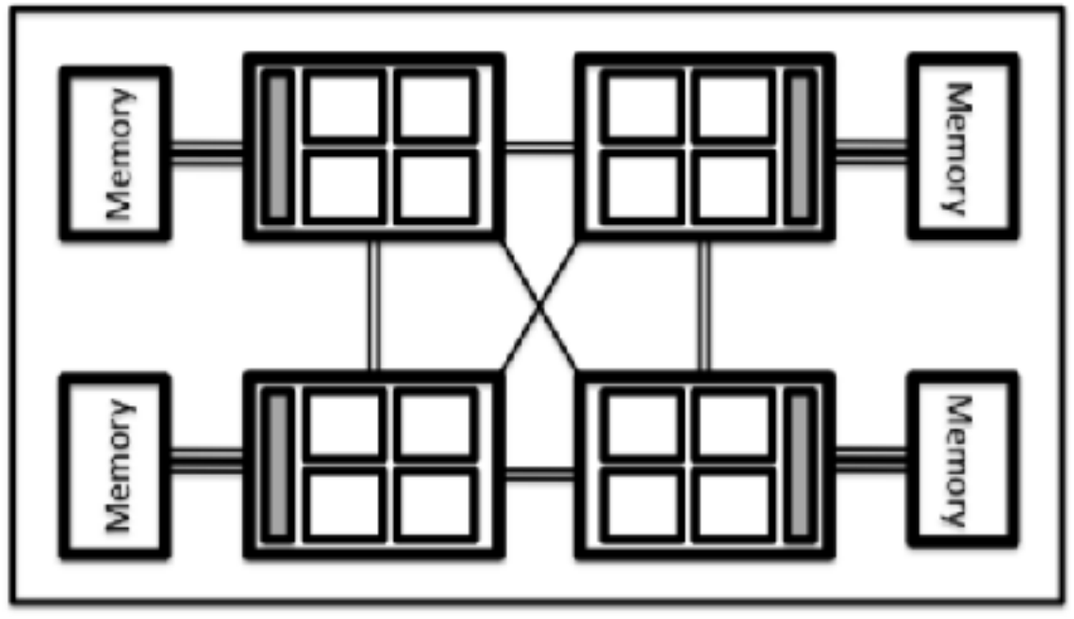
- WARNING: even though one could naively think of a thread as a sub-process, the difference is that threads can access memory from parent process

- Ex: OpenMP, OpenACC (directive based), OpenCL, CUDA (kernel execution based)

Symmetric Multi-Processing



Non-Uniform Memory Access



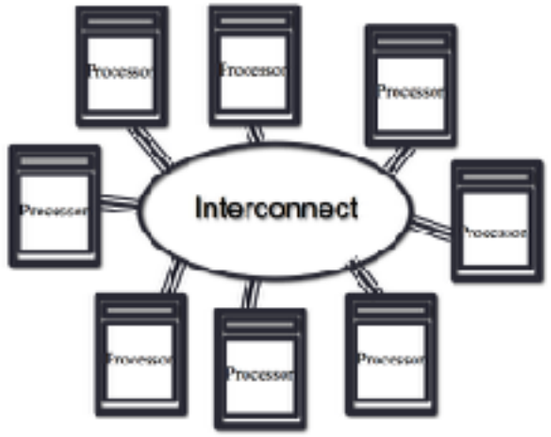
[ARCHER Introduction to HPC]

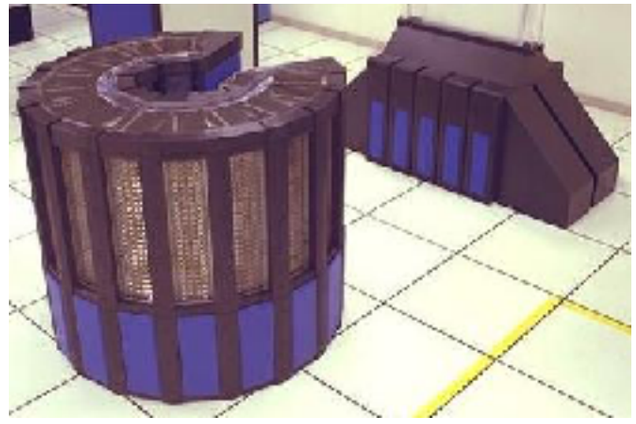
- However! It's difficult to build shared-memory systems with large core counts
  - Not cost-effective to manufacture
  - Power hungry
  - Scaling the Operating System becomes a herculean task

Architectures vs. Programming paradigms

- Modern supercomputers combine two types of architecture. The second one is,
  - Distributed memory

Ex: ARCHER,
COSMOS, PizDaint





- Maps well to message-passing (between processes):

[ARCHER Introduction to HPC]

Message-Passing programming

Program

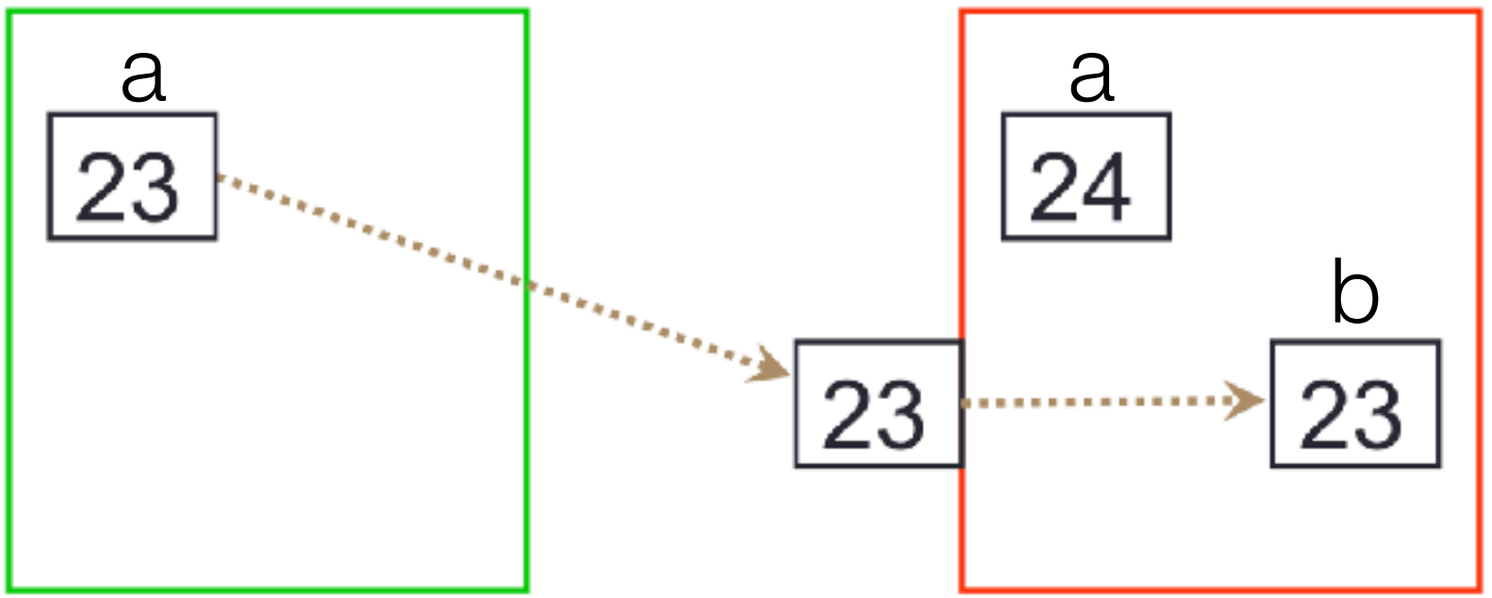
Process 1

```
a=23
Send(2,a)
```

Process 2

```
Recv(1,b)
a=b+1
```

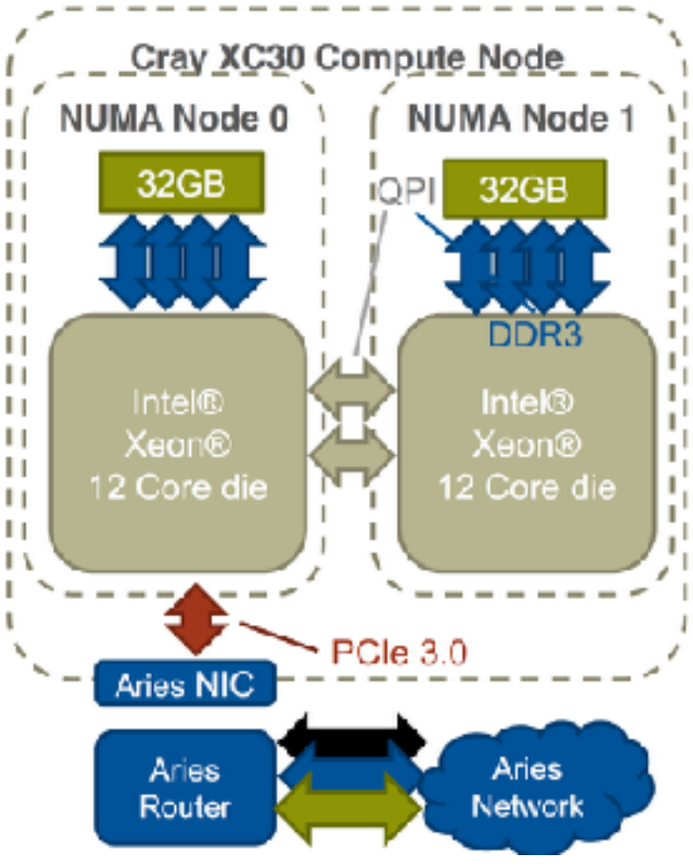
Data



Ex: Message-Passing Interface

Hybrid Architectures

- As mentioned most machines in High Performance Computing employ a mixture of both architectures. An example would be ARCHER, Piz Daint, or even Supernova



- Should one use hybrid paradigms as well? Advantages and disadvantages (however this is beyond the scope of this talk).

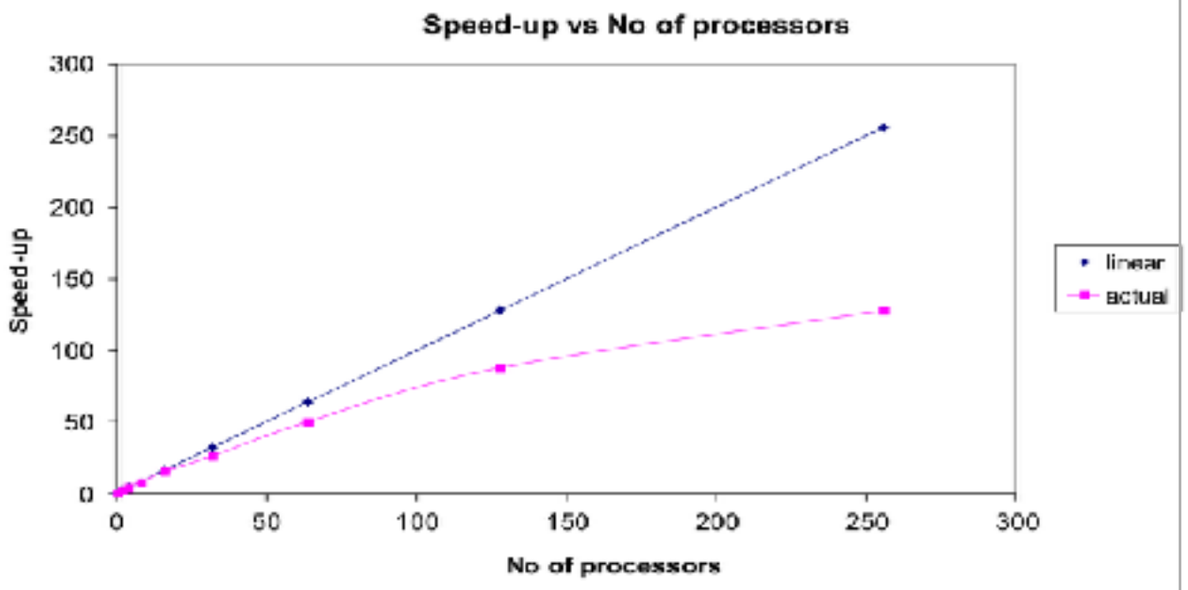
Performance metrics and scaling

- The first universal performance metric is (non-surprisingly) execution time T, and its derivative metrics:
  - Speed-up $$S(N,P) = \frac{T(N,1)}{T(N,P)}$$
  - Parallel efficiency $$E(N,P) = \frac{T(N,1)}{PT(N,P)}$$
  - Serial efficiency $$E_s(N,P) = \frac{T_{best}(N)}{T(N,1)}$$

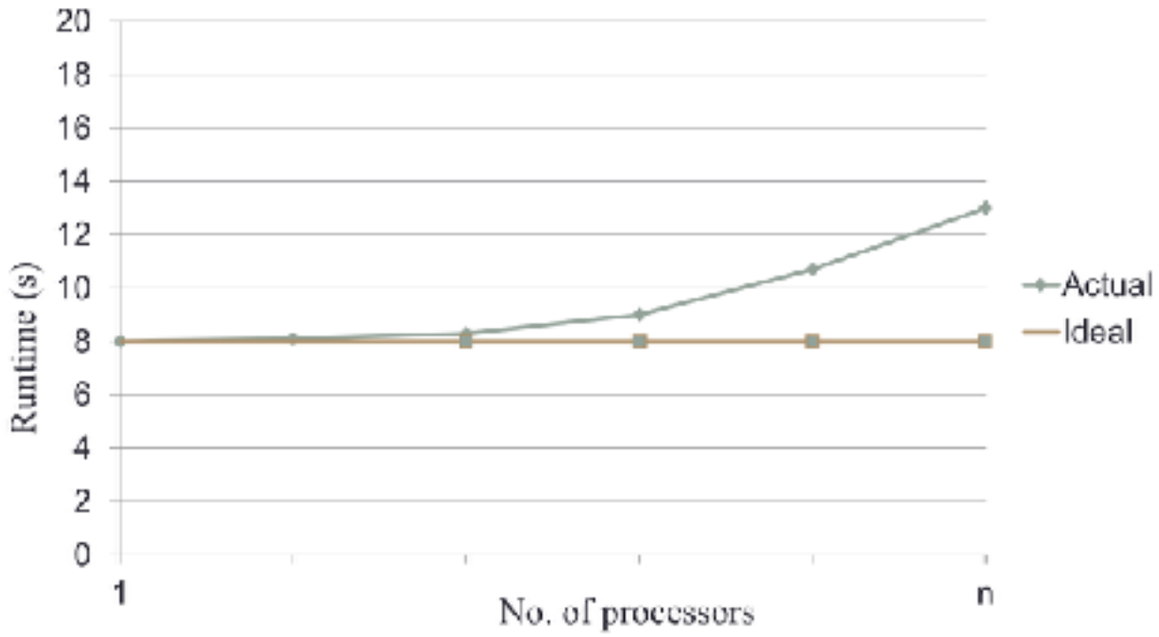
Where P is the number of processors and N the problem size

Performance metrics and scaling

- Strong scaling
  - fixed total problem size, number of processors increases



- Weak scaling
  - problem size increases at the same rate of number of processors (fixed amount of work per processor)



Amdahl's Law - no Silver Bullet

"The performance improvement to be gained by parallelisation is limited by the proportion of the code which is serial" - Gene Amdahl, 1967

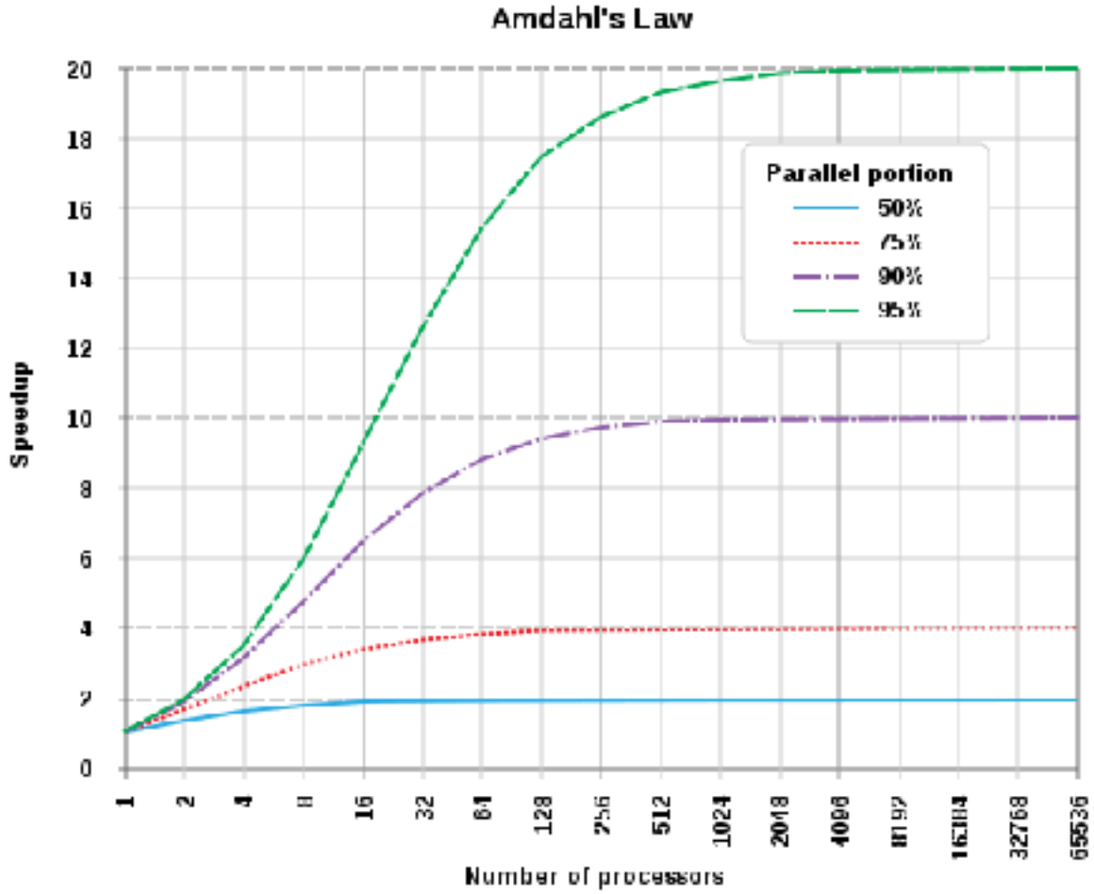
- Limited by serial fraction - α,
- Runtime:

$$T(N,P) = \alpha T(N,1) + \frac{(1-\alpha)T(N,1)}{P}$$

- Speed-up:

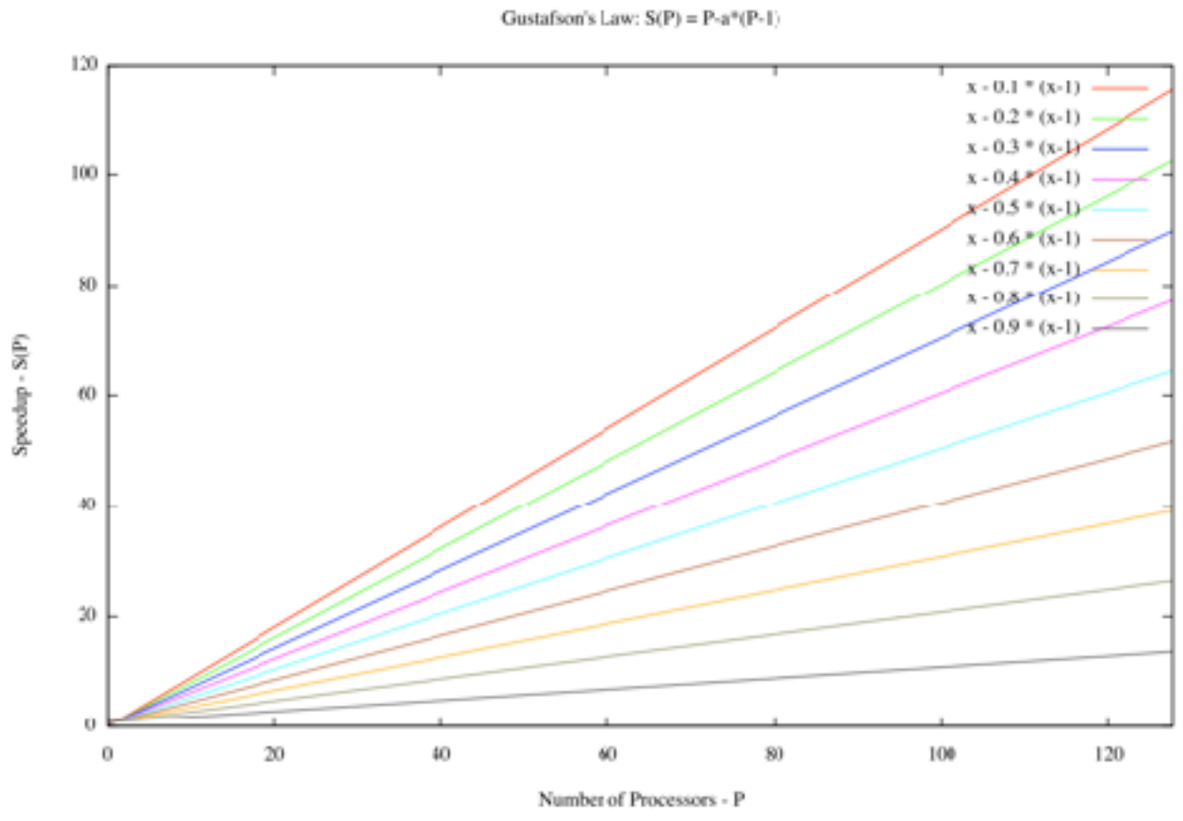
$$S(N,P) = \frac{P}{\alpha P + (1-\alpha)}$$

- Fixed problem size (strong scaling)



Gustafson's Law - no Silver Bullet

"…speedup should be measured by scaling the problem to the number of processors, not fixing problem size." - John Gustafson, 1998



- Runtime:

$$T(N,P) = T_{serial}(N,P) + T_{parallel}(N,P)$$
$$= \alpha T(1,1) + \frac{(1-\alpha)NT(1,1)}{P}$$

- Speed-up:

$$S(N,P) = \frac{T(N,1)}{T(N,P)}$$

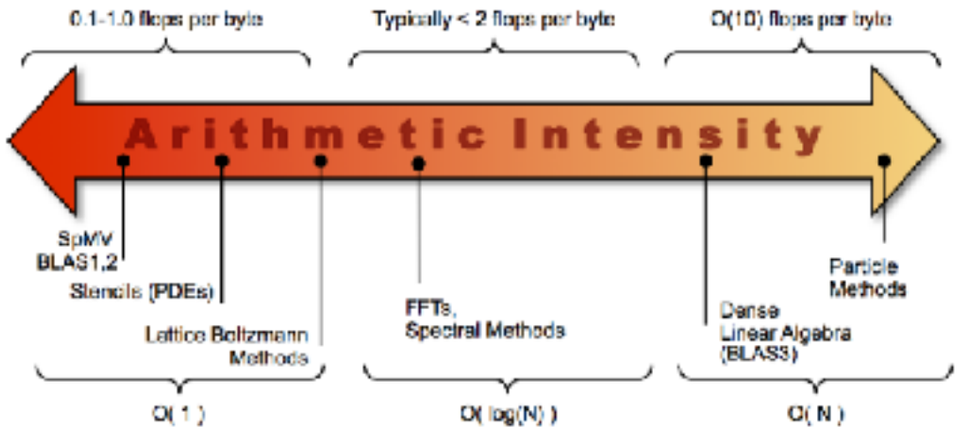
- Scale problem size with CPU's (N=P),

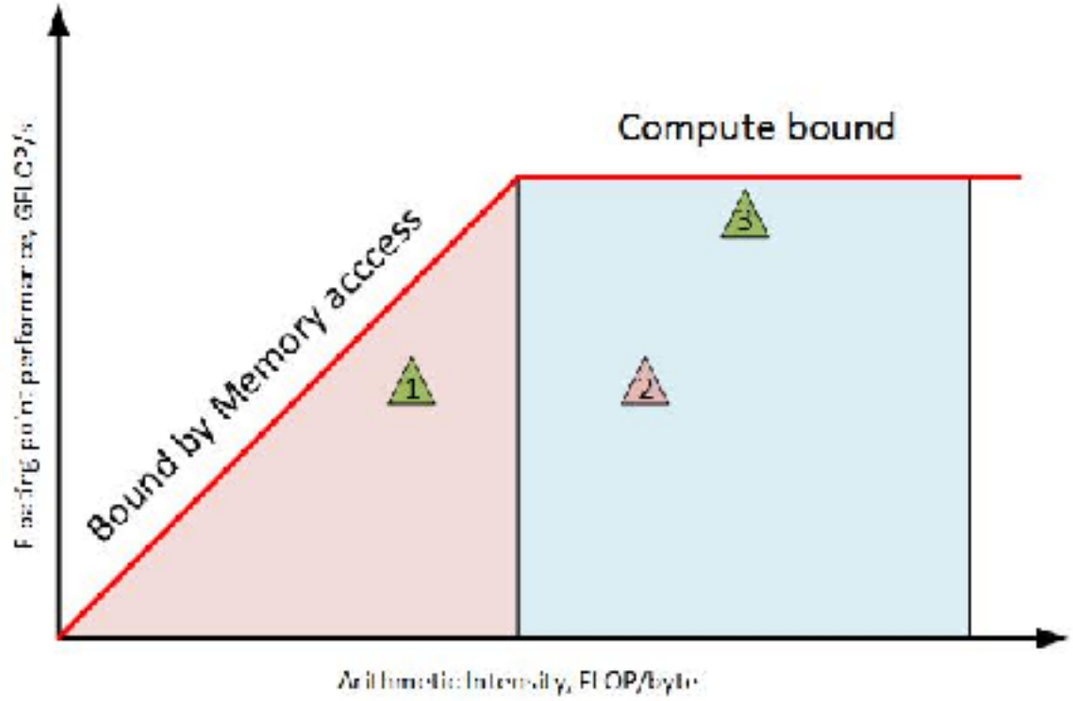
$$S(P,P) = \alpha + (1-\alpha)P$$

Identifying bottlenecks - Roofline model

- Take two quantities:
  - π - Peak theoretical performance (FLOPs/s)
  - I - Arithmetic intensity (FLOPs/byte)
- Trace two curves: $P = \pi$ and $P = \beta \times I$

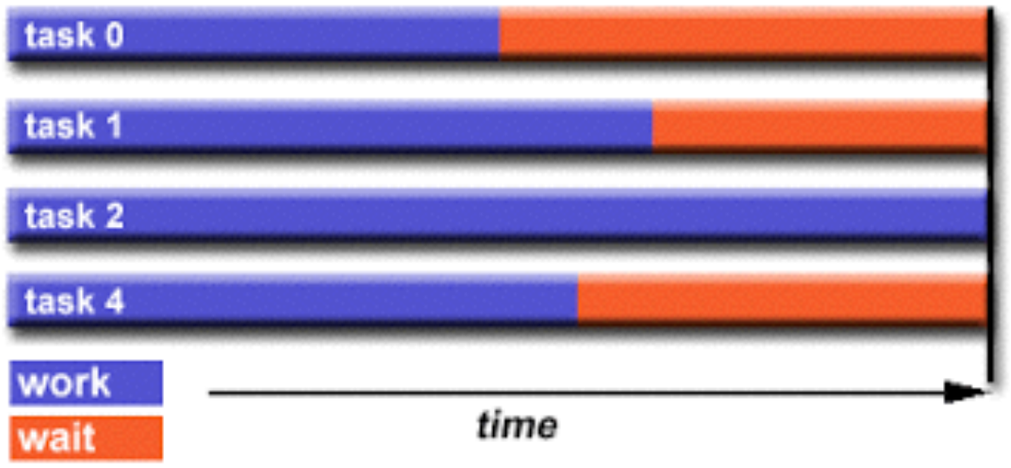
  (β is a bandwidth)
  - If $I \leq \pi/\beta \Rightarrow$ Memory-bound
  - If $I \geq \pi/\beta \Rightarrow$ Compute-bound



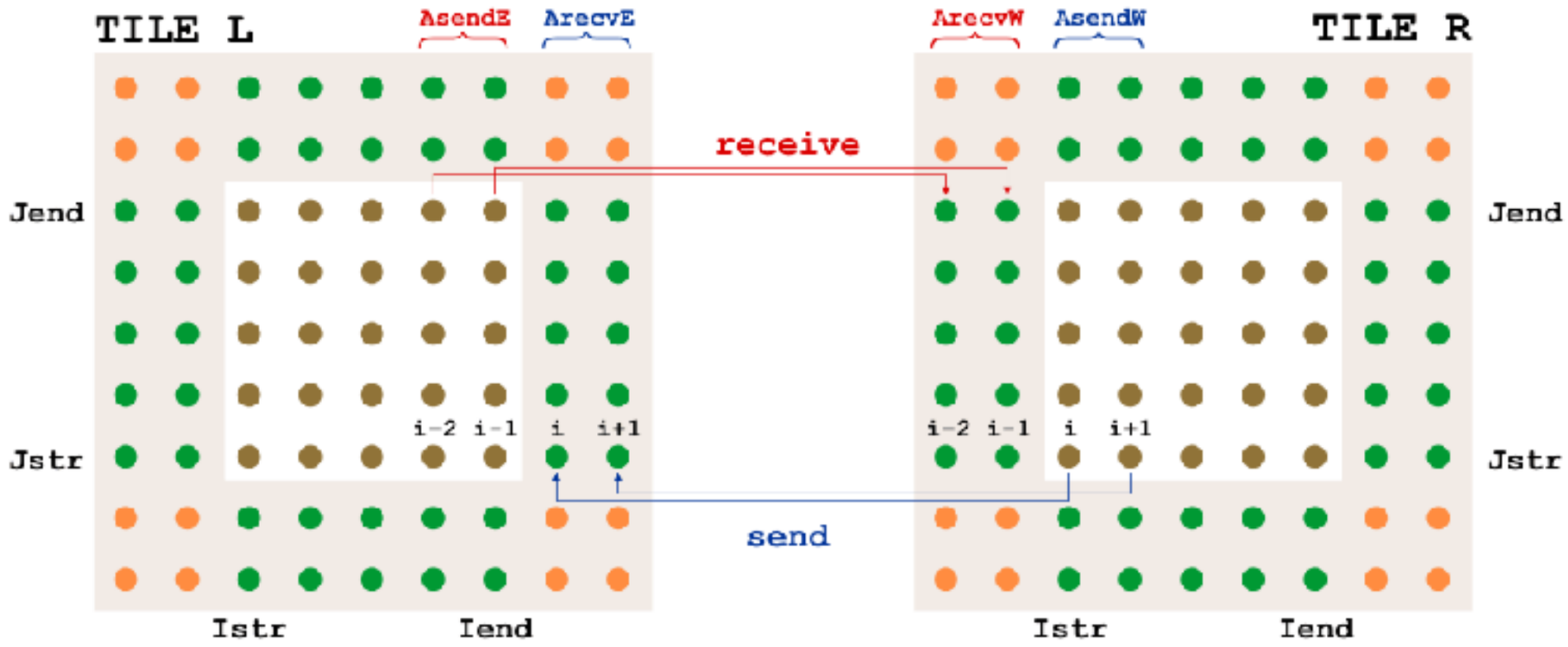


[Code project, 2017]

Identifying bottlenecks - Load Imbalance



Load Imbalance
vs
Communication



Resources and libraries

- Resources:
  - ARCHER Training (check past courses/youtube channel for material!)
- A few good books:

- Libraries:
  - Chombo, GAMER
  - Boost.compute
  - Latfield2
  - Tensorflow
  - cuFFT, cuDDN, cuRAND, cuBLAS…
  - Arrayfire

Roses are red

Violets are blue

You’re missing a ‘}’ on line 32

Practical Exercise

- Time to get our hands dirty :)

- Login to your favourite cluster (in this example I'm gonna go with supernova) and download this image sharpening program:

```
ssh username@sol.astro.up.pt
```

```
ssh username@supernova

wget http://www.archer.ac.uk/training/course- material/2017/07/intro-epcc/
exercises/sharpen.tar.gz


tar -xvzf sharpen.tar.gz
```

- You can find several versions of the same code, we only need C-MPI and C-SER

- First we need to make sure we can compile both of them!

Practical Exercise

```
cd sharpen/C-SER

make

cd ../C-MPI

module load mpi

make
```

- If the second make (on C-MPI fails) then edit the file Makefile, fourth line to CC= mpicc

```
#!/bin/bash
#
#SBATCH --job-name=MPI-test
#
# Specify number of parallel tasks
#SBATCH -ntasks=1
#SBATCH --output=array_%A.out

mpirun ./sharpen
```

- If everything went along peachy, time to create a submission script for C-SER and C-MPI. Example of a submission script (for slurm) in orange <-

Practical Exercise

With Slurm:

sbatch submit.sh - submit job
squeue - list jobs
scancel - self-explanatory

when it finishes running, check array_JOB_ID_ARRAY_ID.out ;)

NOTE:

- for ntasks==1,use C-SER
- for ntasks larger than one use C-MPI

# Cores	Overall run time	Calculation time	IO time	Total CPU time
1				
2				
4				
7				
10				

Ideal Speedup	Overall speedup	Calculation Speedup	IO Speedup

Overall run time - Calculation time = IO time
Total CPU time = Overall run time * Cores
Ideal Speed-up = Cores

Practical Exercise