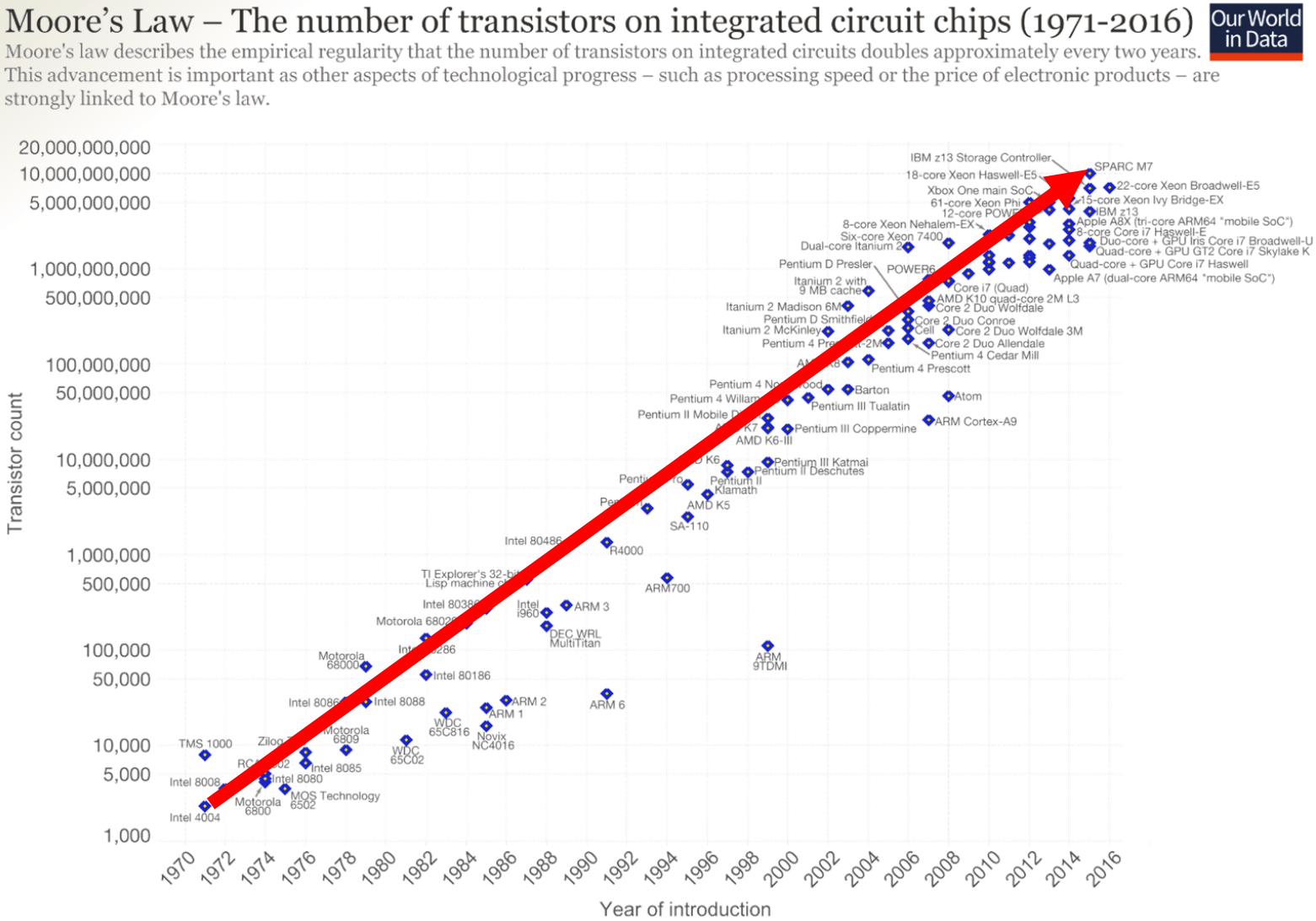
RHD algorithms for the exa-scale era

Some technical notes

Bert Vandenbroucke (bv7@st-andrews.ac.uk)

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
Year of introduction
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
IBM z13 Storage Controller
18-core Xeon Haswell-E5
Xbox One main SoC
61-core Xeon Phi
12-core POWER8
8-core Xeon Nehalem-EX
Six-core Xeon 7400
Dual-core Itanium 2
Pentium D Presler
POWER6
Itanium 2 with 9 MB cache
Itanium 2 Madison 6M
Pentium D Smithfield
Itanium 2 McKinley
Pentium 4 Prescott-2M
Pentium 4 Willamette
Pentium II Mobile Dixon
Pentium 4 Northwood
Pentium III Tualatin
Pentium III Coppermine
AMD K6-III
Pentium III Katmai
Pentium II Deschutes
Pentium II Klamath
AMD K5
SA-110
Intel 80486
R4000
TI Explorer's 32-bit Lisp machine chip
ARM700
Intel 80386
Intel i960
ARM 3
Motorola 68020
DEC WRL MultiTitan
Motorola 68000
Intel 80186
ARM 9TDMI
Intel 8086
Intel 8088
ARM 2
ARM 1
ARM 6
WDC 65C816
Novix NC4016
Motorola 6809
WDC 65C02
TMS 1000
Intel 8085
Intel 8080
Intel 8008
MOS Technology 6502
Motorola 6800
Intel 4004
SPARC M7
22-core Xeon Broadwell-E5
15-core Xeon Ivy Bridge-EX
IBM z13
Apple A8X (tri-core ARM64 "mobile SoC")
8-core Core i7 Haswell-E
Duo-core + GPU Iris Core i7 Broadwell-U
Quad-core + GPU GT2 Core i7 Skylake K
Quad-core + GPU Core i7 Haswell
Apple A7 (dual-core ARM64 "mobile SoC")
Core i7 (Quad)
AMD K10 quad-core 2M L3
Core 2 Duo Wolfdale
Core 2 Duo Conroe
Cell
Core 2 Duo Wolfdale 3M
Core 2 Duo Allendale
Pentium 4 Cedar Mill
Pentium 4 Prescott
Barton
Atom
ARM Cortex-A9
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.

Computational speedup

Assume the computational speed in 1970 is S_{1970}

The computational speed in 2010 is

$$S_{2010} \approx 2^{20} S_{1970} \approx 10^6 S_{2010}$$

The computational load ΔC for a run time Δt in 1970 is

$$\Delta C_{1970} = S_{1970} \Delta t_{1970}$$

So in 2010 a factor 10^6 can be used to reduce Δt and increase ΔC

Example

Price (1969): 10^6 photon packets, 18 zones (cells), 200 mins

typical CPU speed: 740 KHz

Wood et al. (2004): 10^6 photon packets per min, 65^3 cells

CPU speed: 2.4 GHz

Vandenbroucke & Wood (2018): 10^7 photon packets per min, 64^3 cells

CPU speed: 2.5 GHz

However…

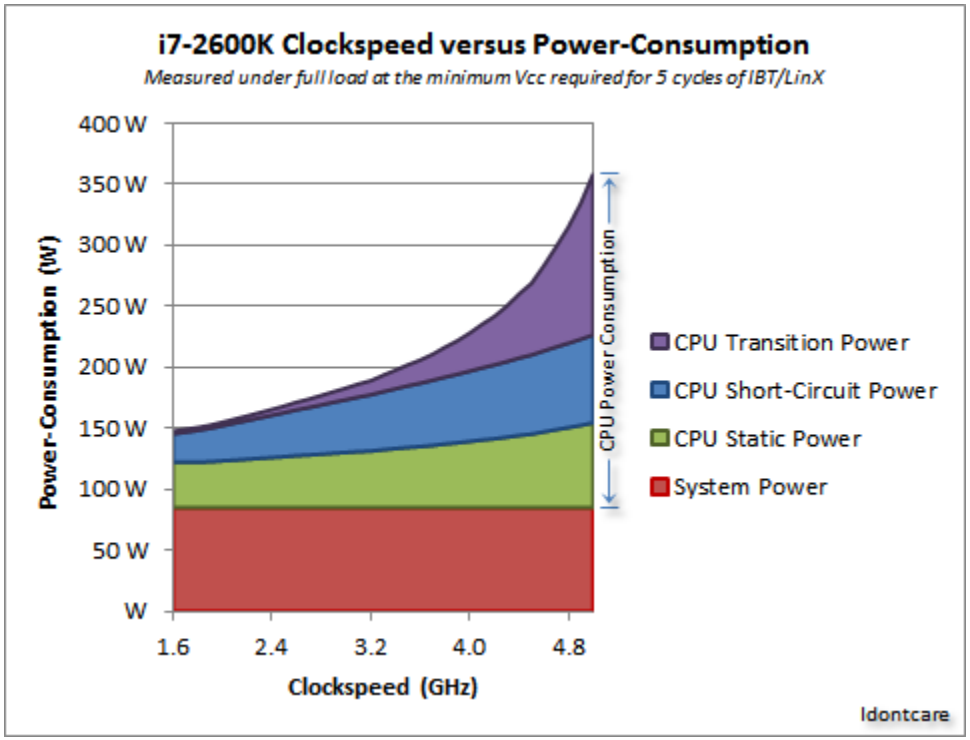
Physics lurks around the corner:

$$P_{CPU} \sim fV^2$$

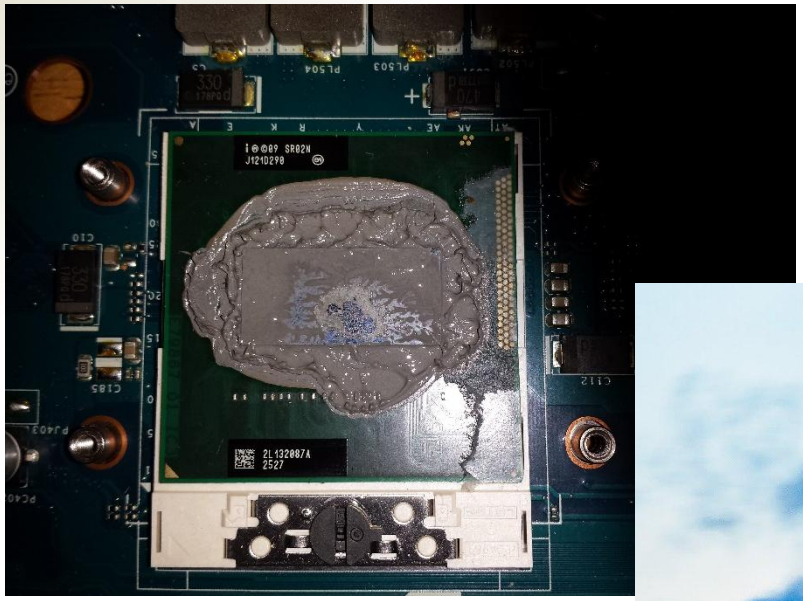
P_{CPU} is the power consumed by the CPU

f is the CPU frequency

V is the CPU voltage



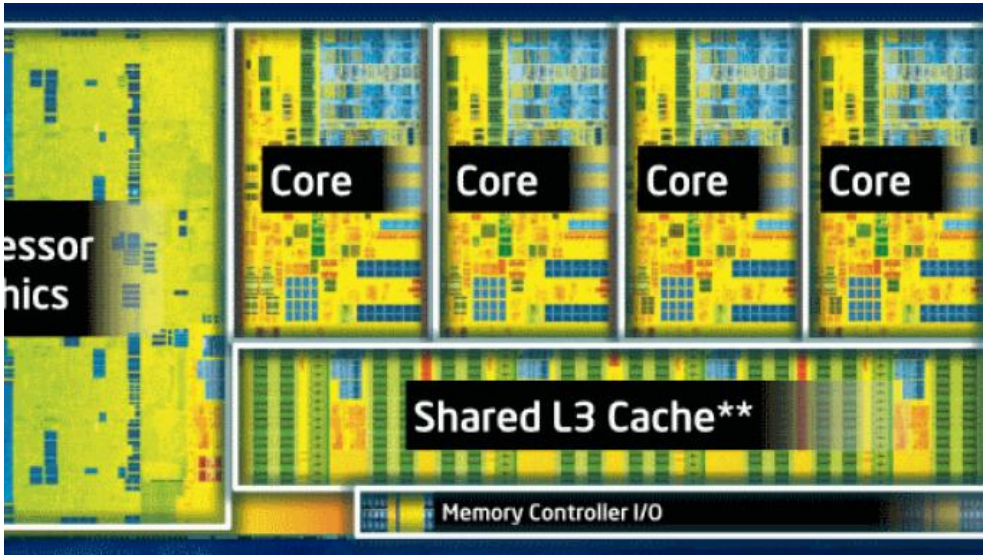
Power means heat



Inefficient heat dissipation makes it very hard to make CPUs with frequencies above $\approx$ 4 GHz

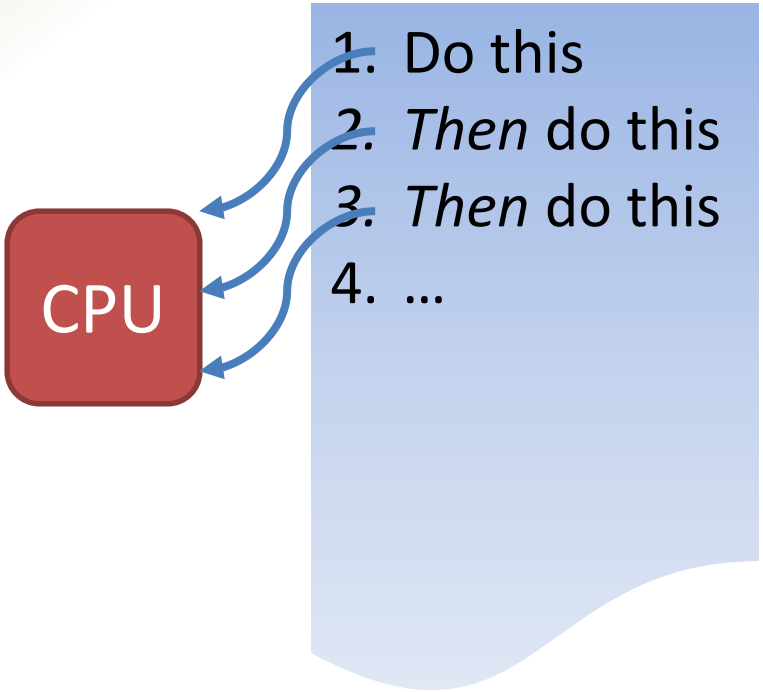
Multicore CPUs

Solution: split CPU into multiple cores that run at lower frequencies
Total computational power of the CPU still goes up, heat production stays under control
BUT cores act as individual CPUs

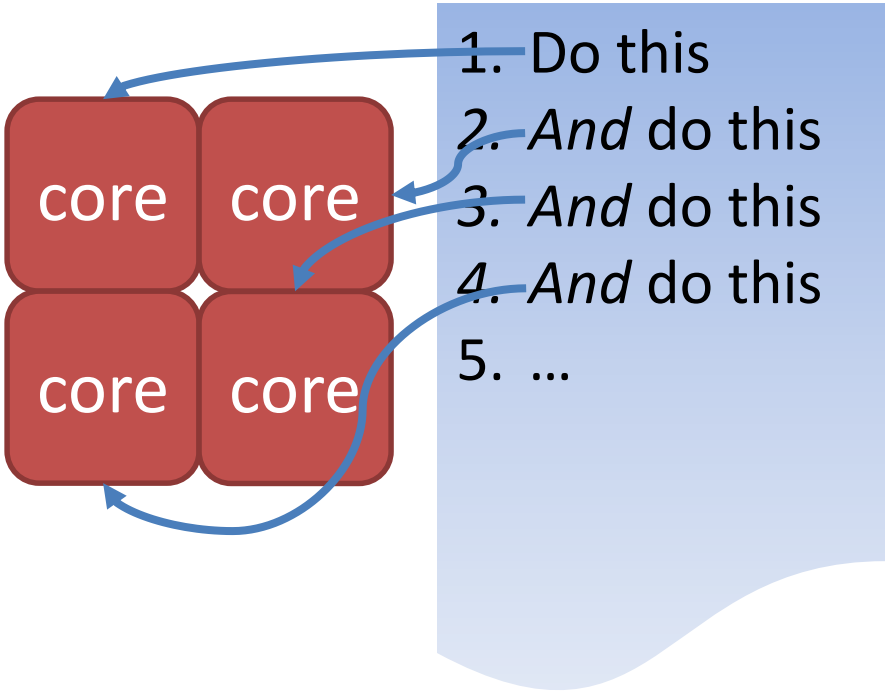


Parallel programming

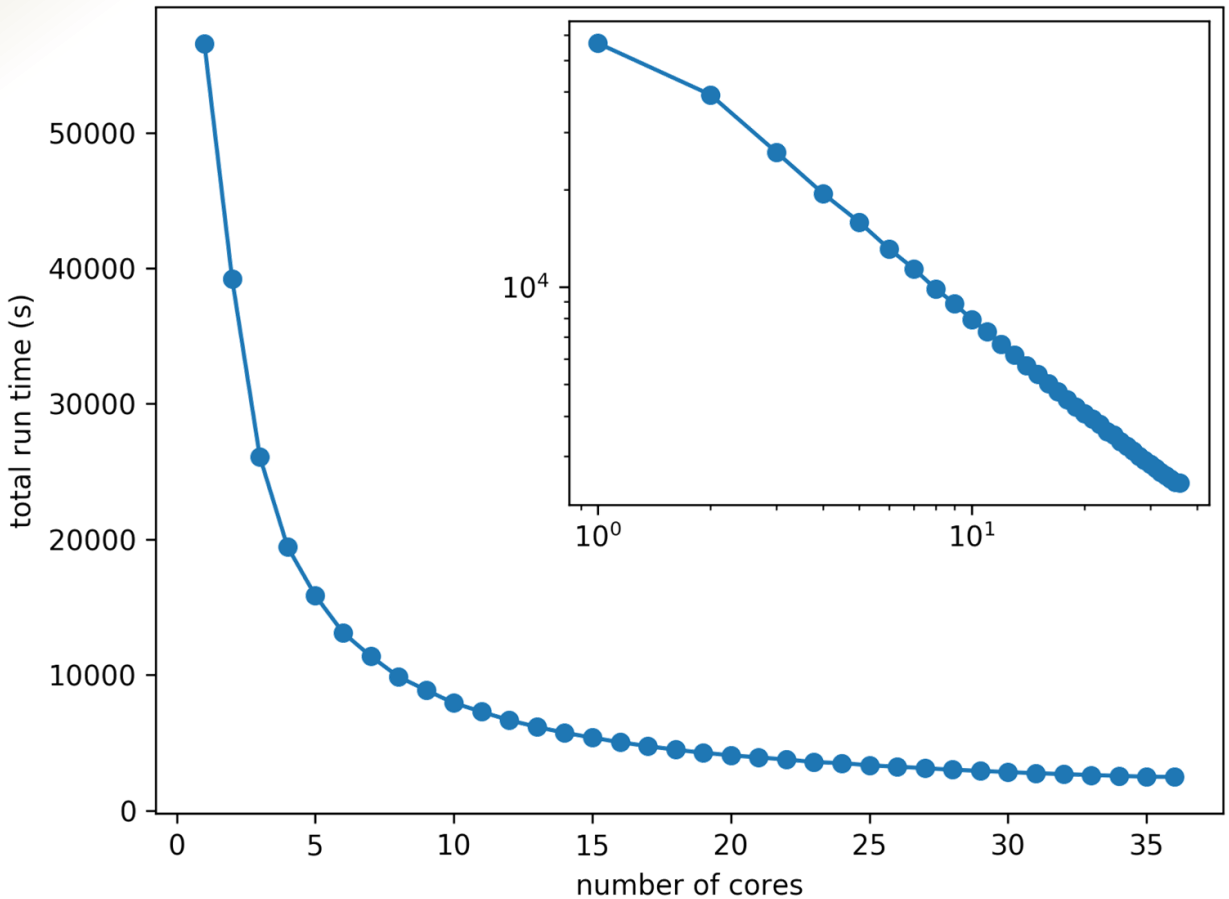
Serial programming => order



Parallel programming => chaos?



Parallelisation impact



Vandenbroucke & Wood (2018)

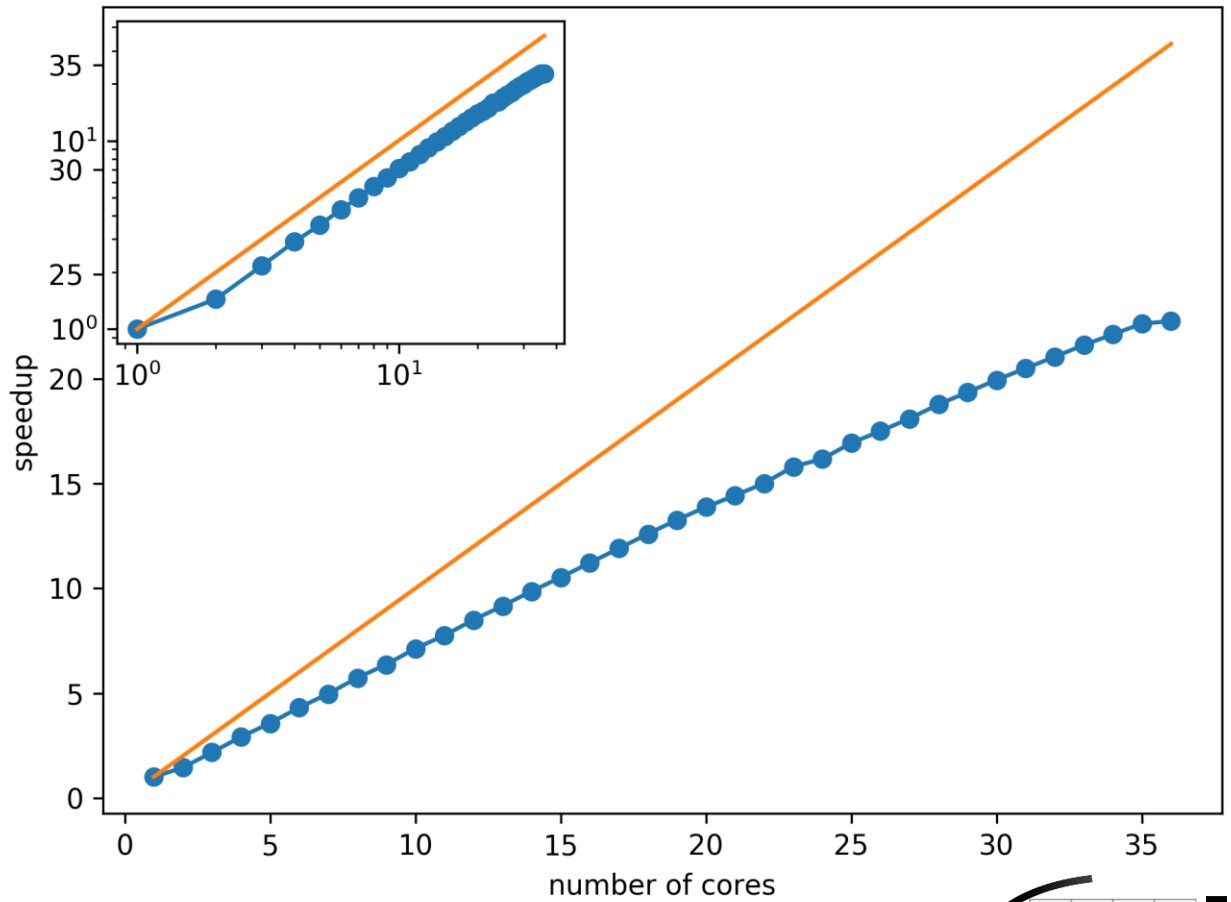
Speedup

$$S(n) = \frac{t_T(1)}{t_T(n)}$$

$t_T(n)$ is the total time using n cores

$S(n)$ is the speedup gained

Ideally, $S(n) = n$

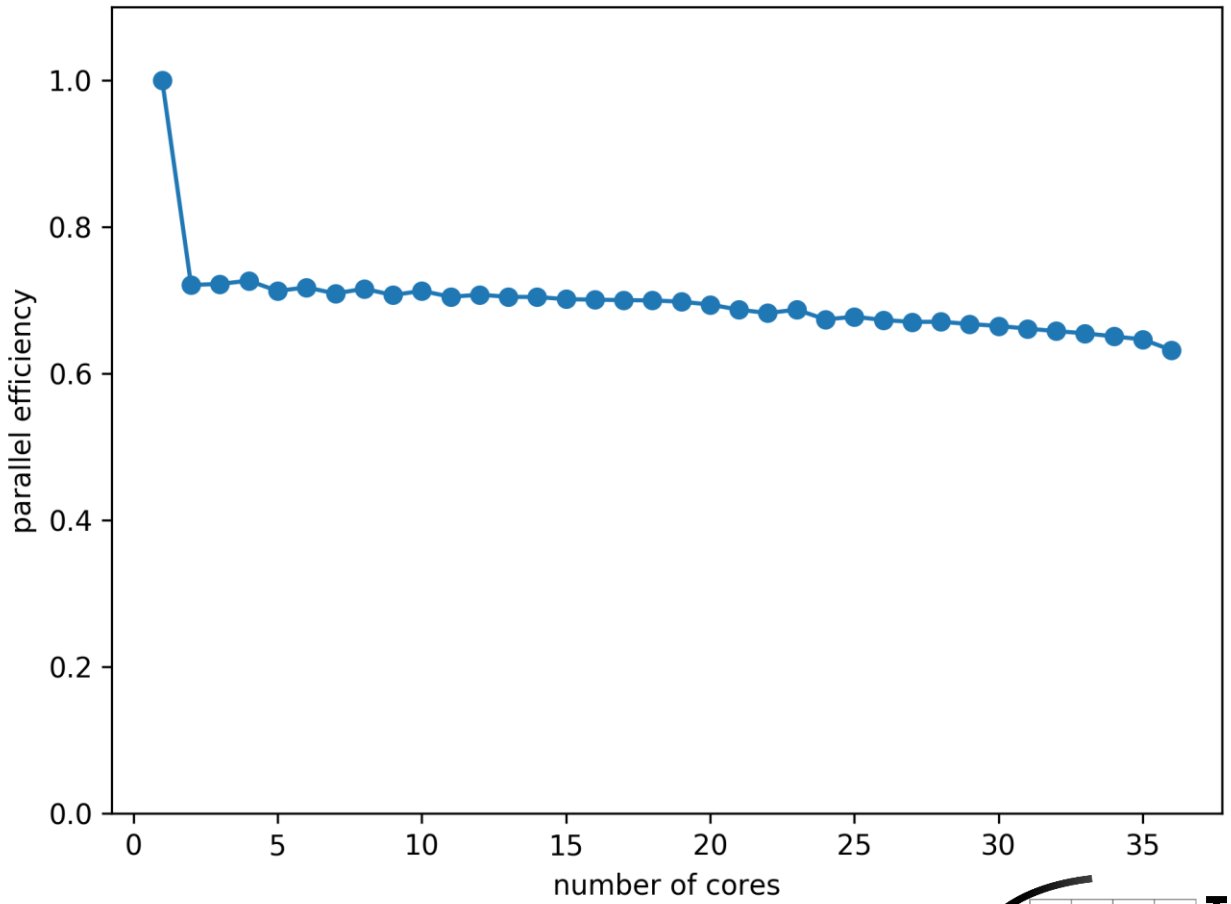


Vandenbroucke & Wood (2018)

Parallel efficiency

$$E(n) = \frac{S(n)}{n}$$

$E(n)$ is the parallel efficiency



Vandenbroucke & Wood (2018)

Amdahl's law

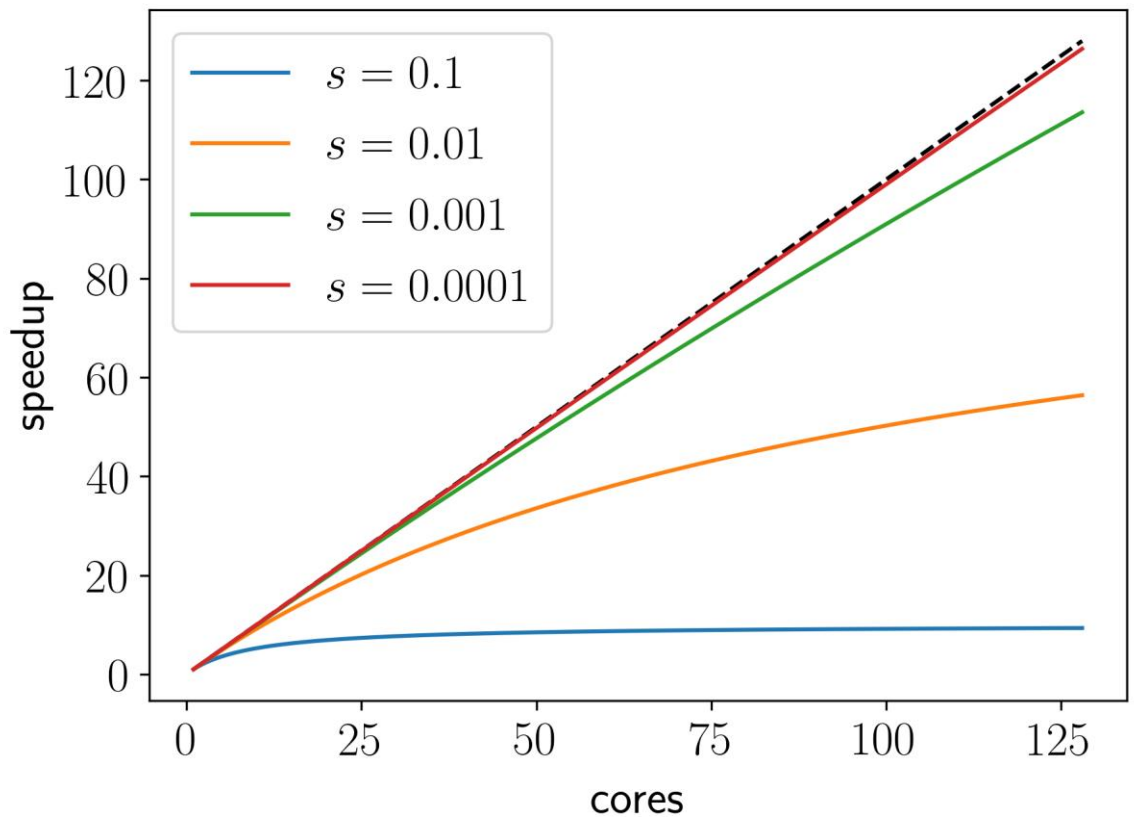
$$t_T(n) = s t_T(1) + \frac{p}{n} t_T(1)$$

s is the serial fraction

p is the parallel fraction

$$S(n) = \frac{1}{s + \frac{p}{n}} < \frac{1}{s}$$

Maximum achievable speedup depends on how parallel your code is!



Gustafson's law

$$t_T(1,n) = st_T(1,1) + pnt_T(1,1)$$

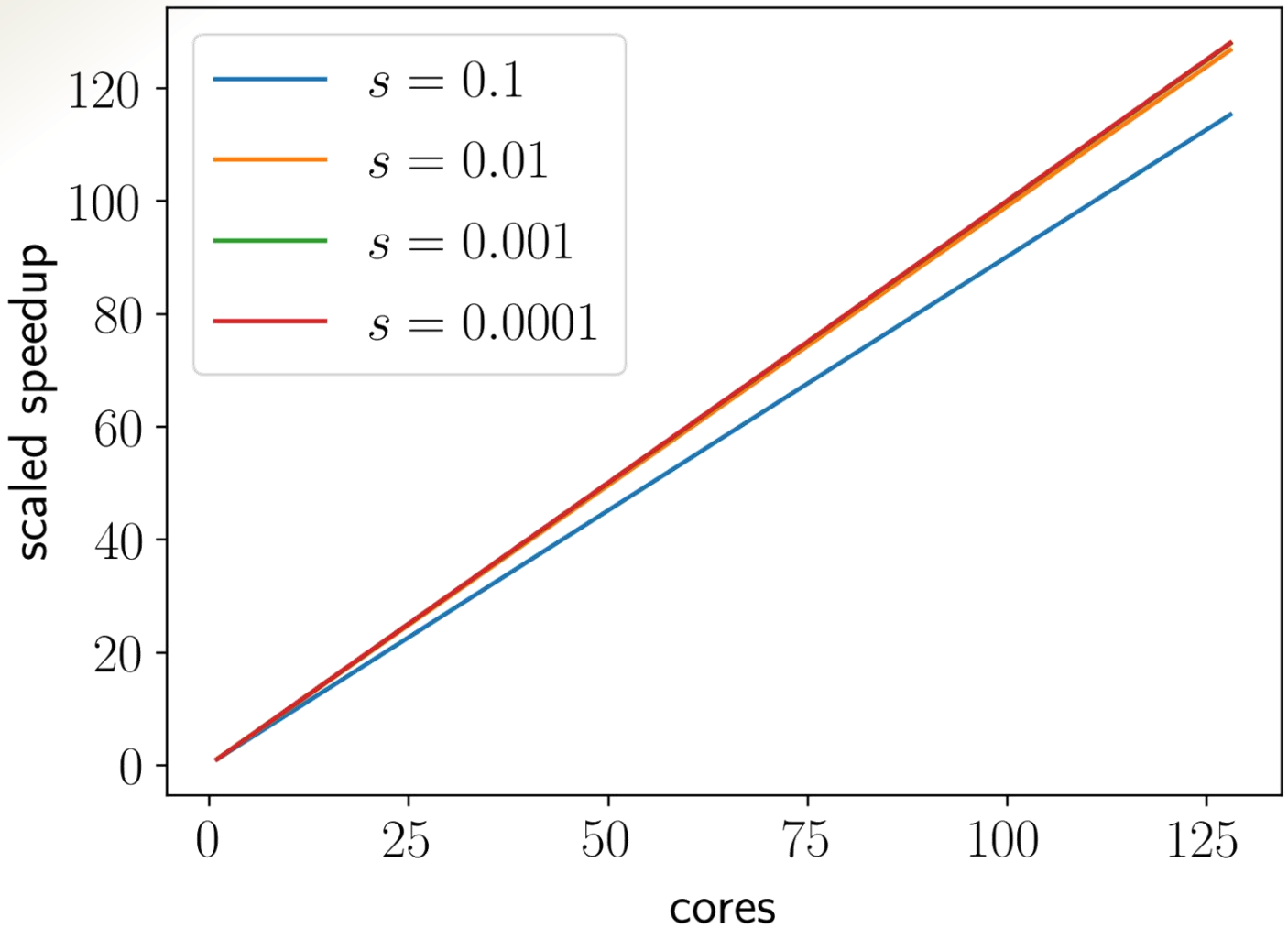
$$t_T(n,n) = st_T(1,1) + pt_T(1,1)$$

$t_T(n,c)$ is the time it takes to execute a simulation with size c using n cores

If instead of simply increasing n, we also increase s to roughly get the same execution time t_T, we can define the *scaled speedup*

$$S_s(n) = \frac{t_T(1,n)}{t_T(n,n)} = \frac{s+pn}{s+p} = s + pn \sim n$$

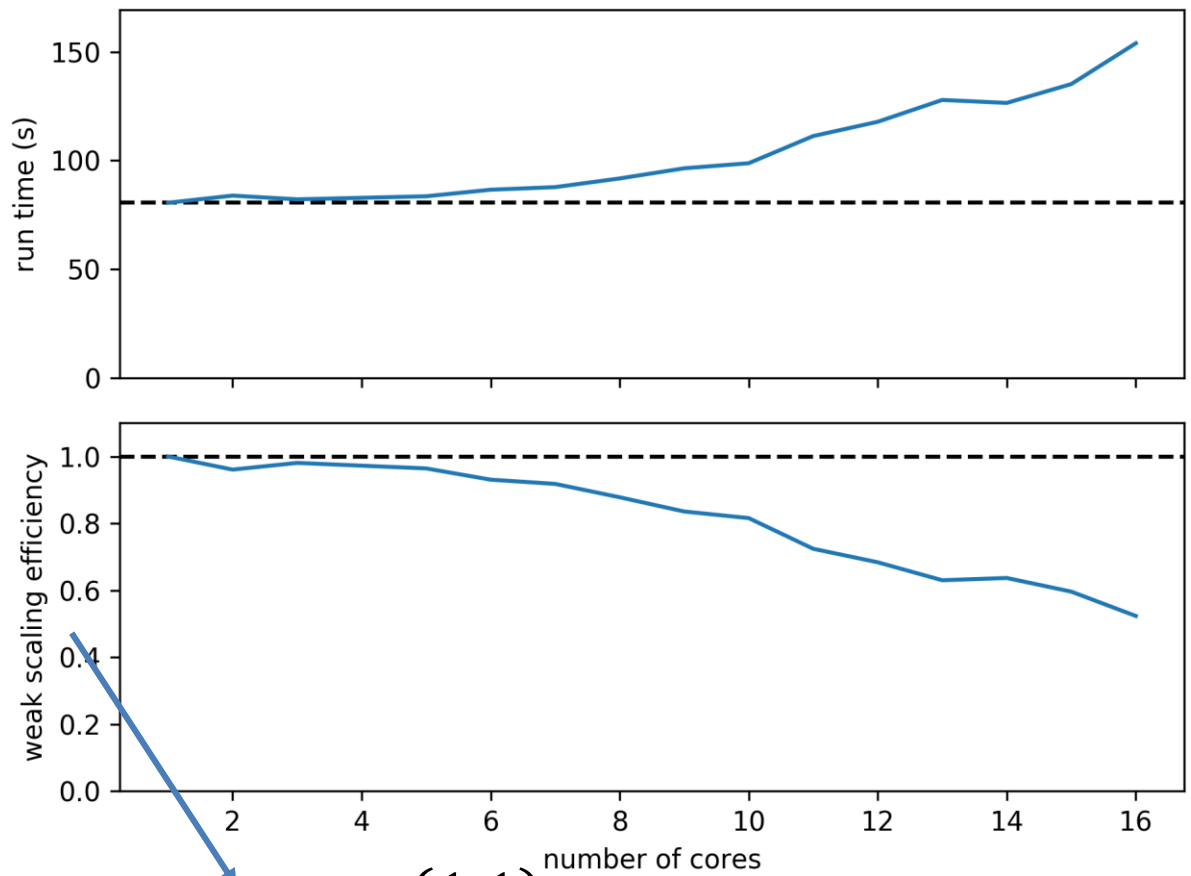
Gustafson's weak scaling



Weak scaling

Increase both the problem size and the number of cores, e.g. MCRT simulation using same grid but $n \times 10^6$ photon packets

Easier to achieve, especially when the workload scales linearly with problem size



$$E_w(n) = \frac{t_T(1,1)}{t_T(n,n)}$$

Strong vs weak scaling

Strong scaling (reducing the execution time for a problem) is incredibly hard

Weak scaling (increasing the problem size for the same execution time) is not

However, codes hardly ever manage to reach theoretical speedups…

How do you use this information?

Computational power has a cost (on Archer: 5p / core hr)

Ideally, we want to minimise the cost of our computation

If $t_T(1)$ is the total expected serial time for the computation, then

$$t_T(n) = \frac{t_T(1)}{S(n)}$$

is the expected run time on n threads

How do you choose resources?

The total cost for the computation is

$$C(n) = t_T(n) \times n \times c = \frac{t_T(0)}{S(n)} \times n \times c$$

where c is the cost per core hour

For perfect scaling, $S(n) = n$, and the cost is constant

For real scaling, $S(n) < n$, and the cost always increases with n

Serial jobs are optimal!

The cost of waiting

The total cost for the computation should be

$$C(n) = \frac{t_T(0)}{S(n)} \times (n \times c + c_w)$$

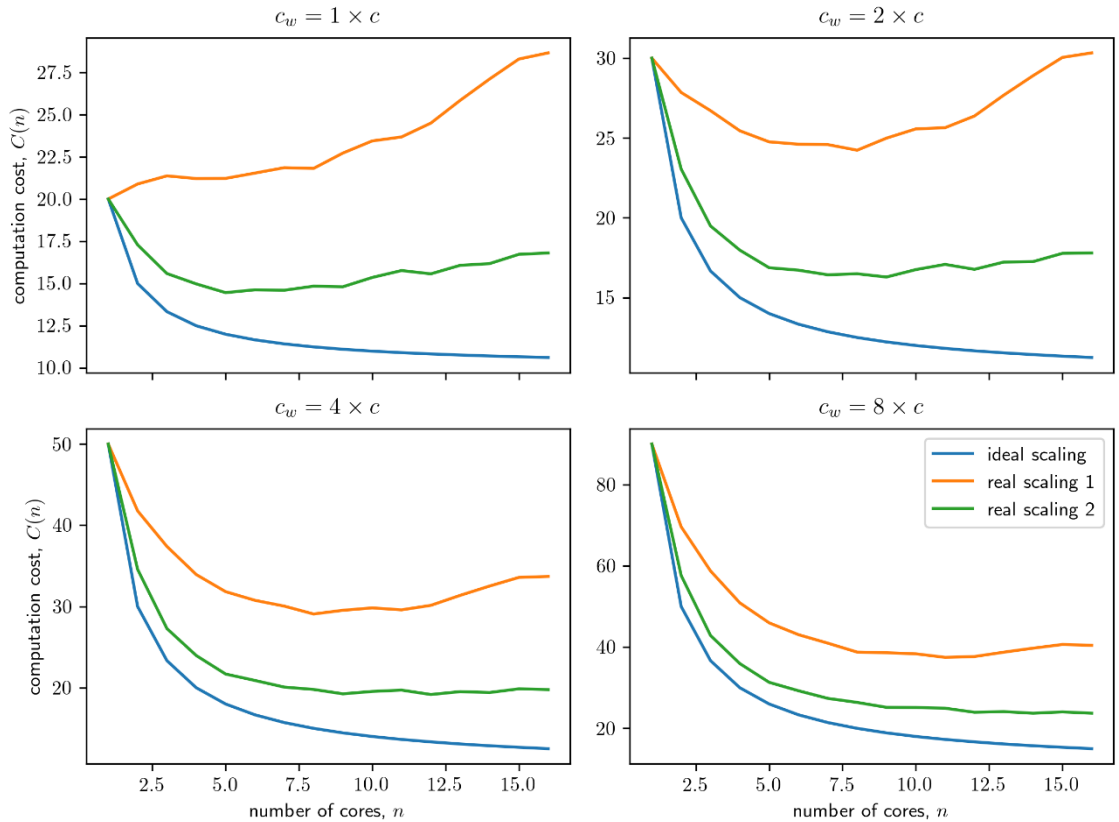
where c_w is the cost for waiting for the computation to finish

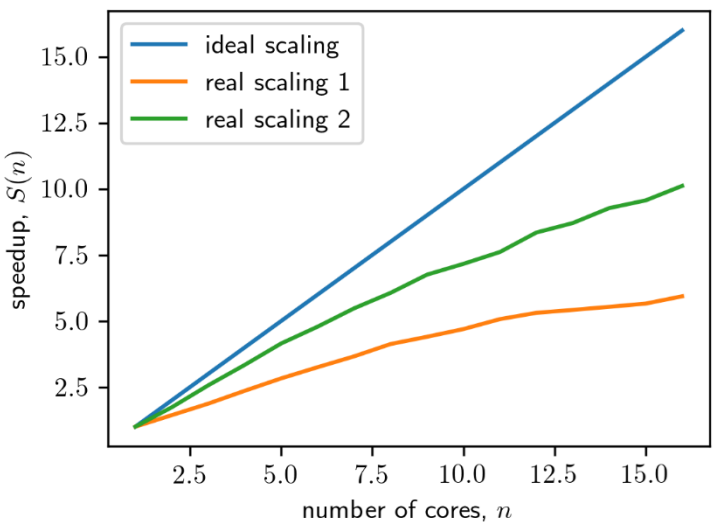
What is this waiting cost?

e.g. typical PhD salary in UK ≈ £7.5 / hr

Example

The optimal number of threads shifts to higher n as the cost of waiting becomes more dominant

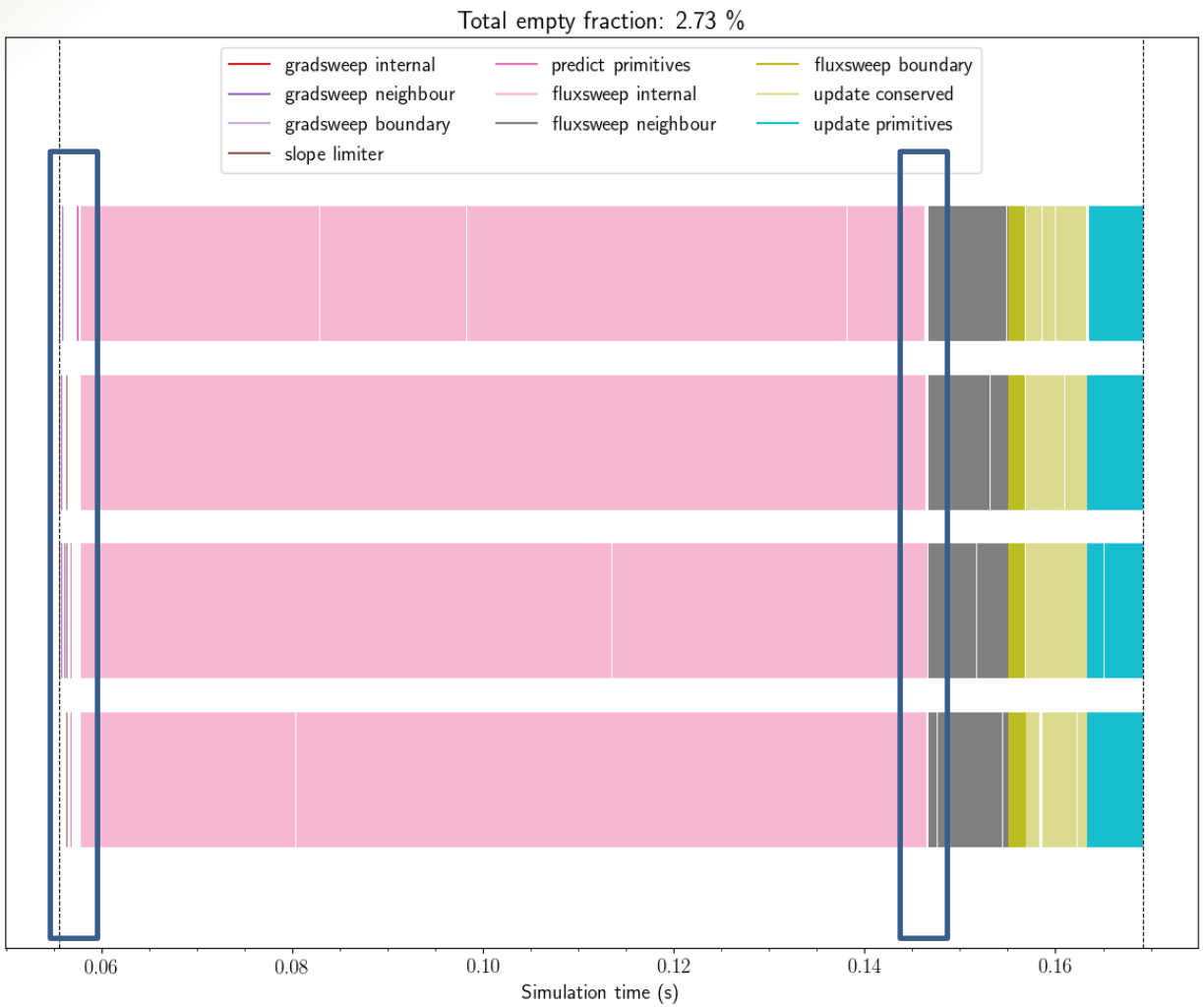




Since $c \ll c_w$ in many cases, this is not a very restrictive decision strategy

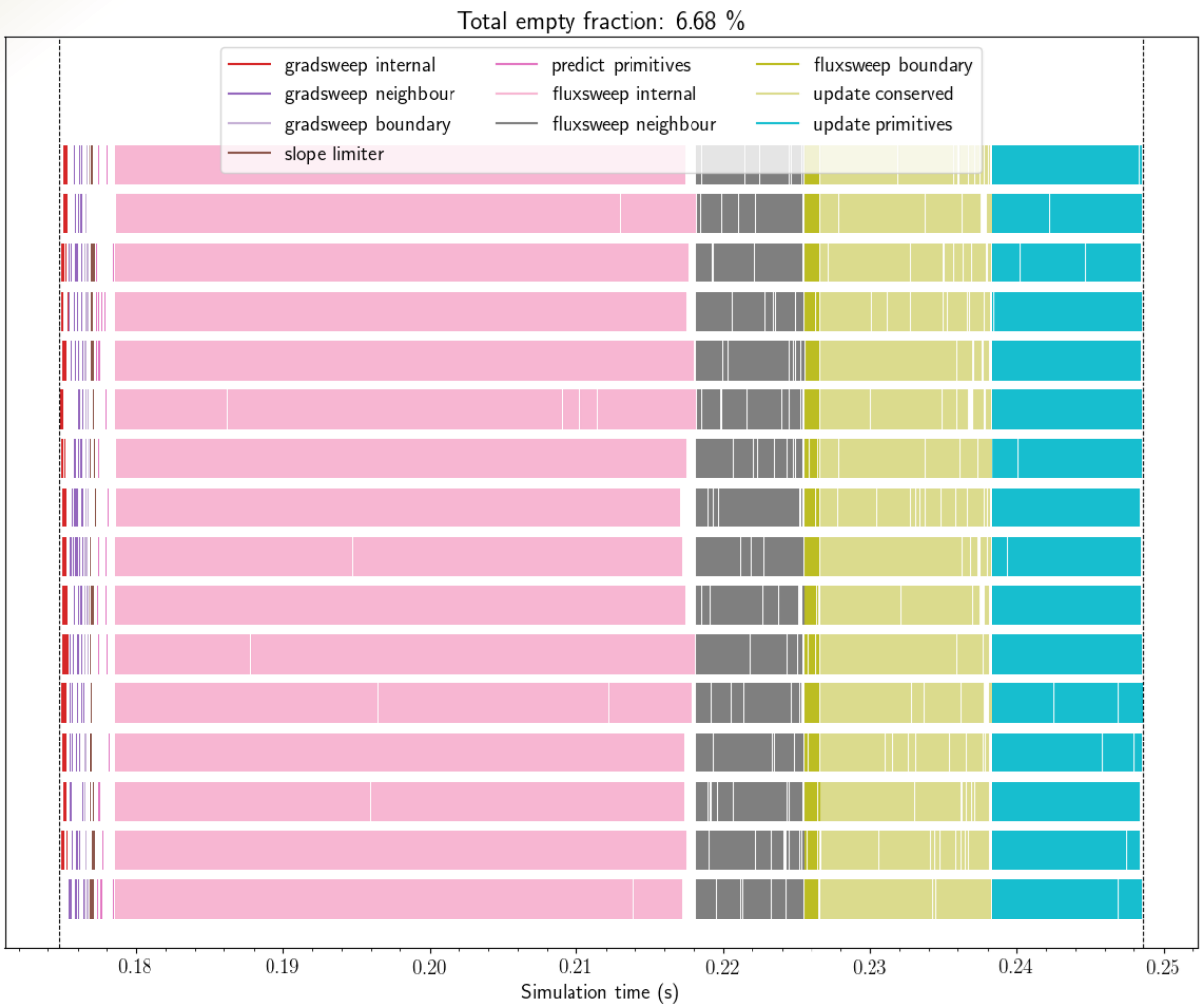
Factors limiting scaling

Serial code: cannot be done in parallel

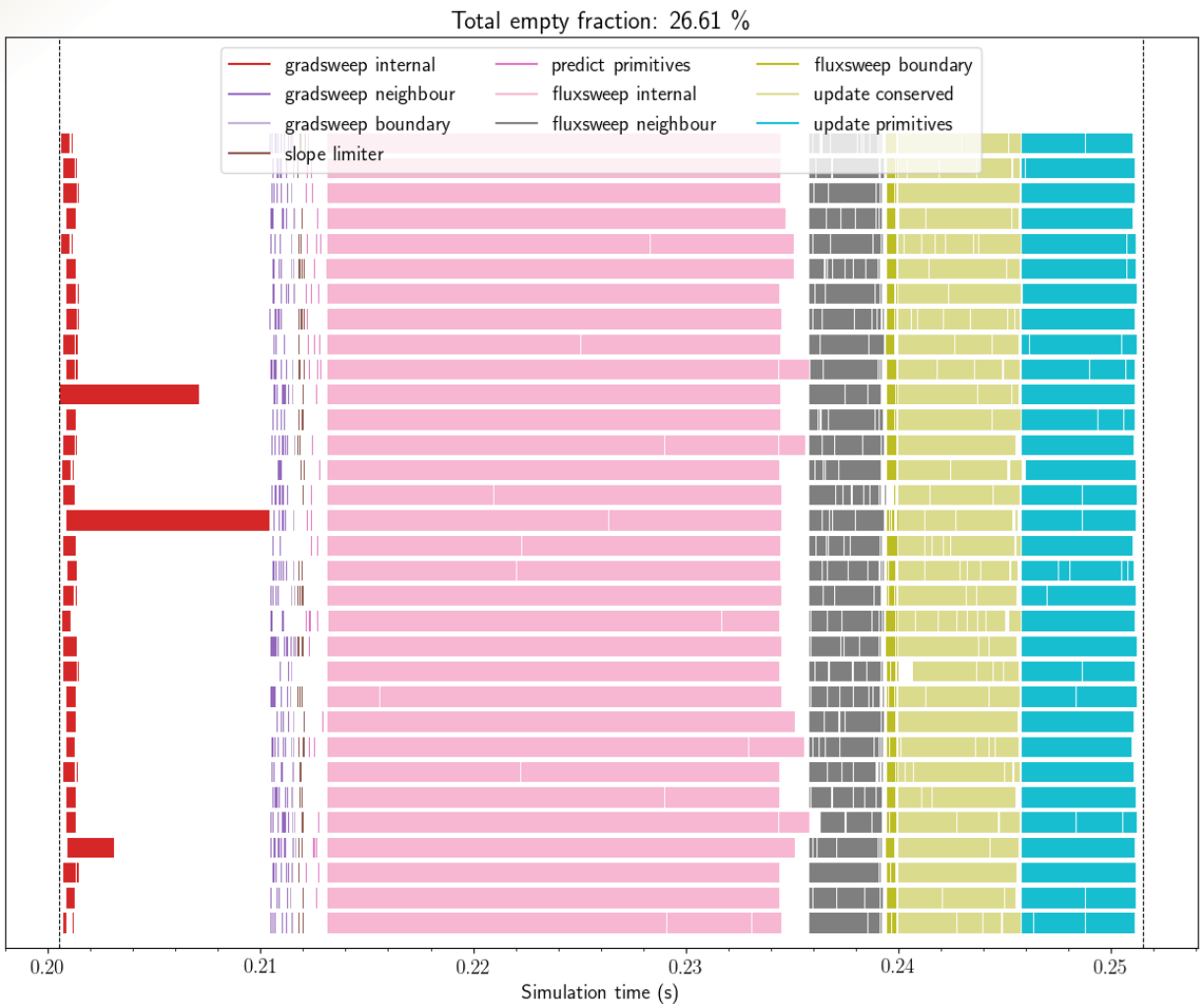


Load imbalances: cores are waiting for other cores to finish

Factors limiting scaling (16 cores)

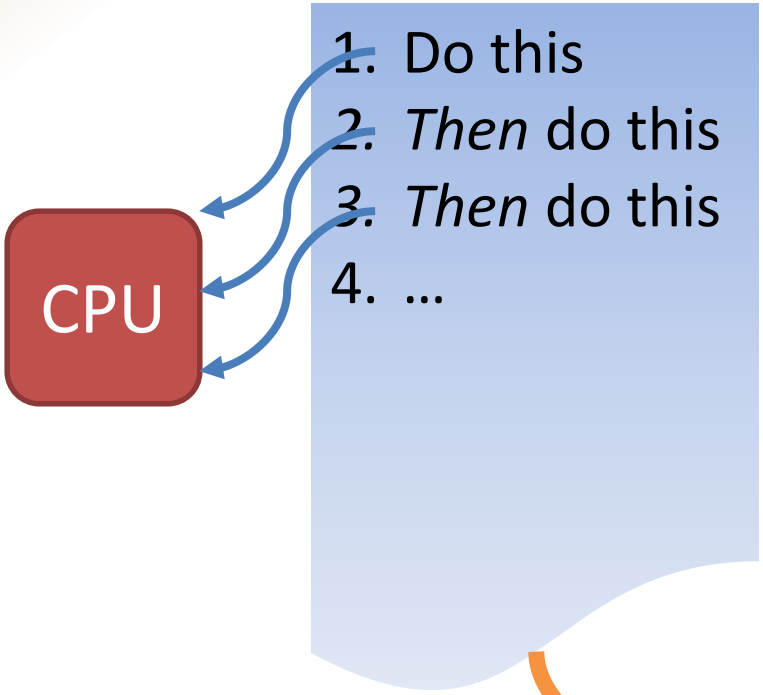


Factors limiting scaling (32 cores)

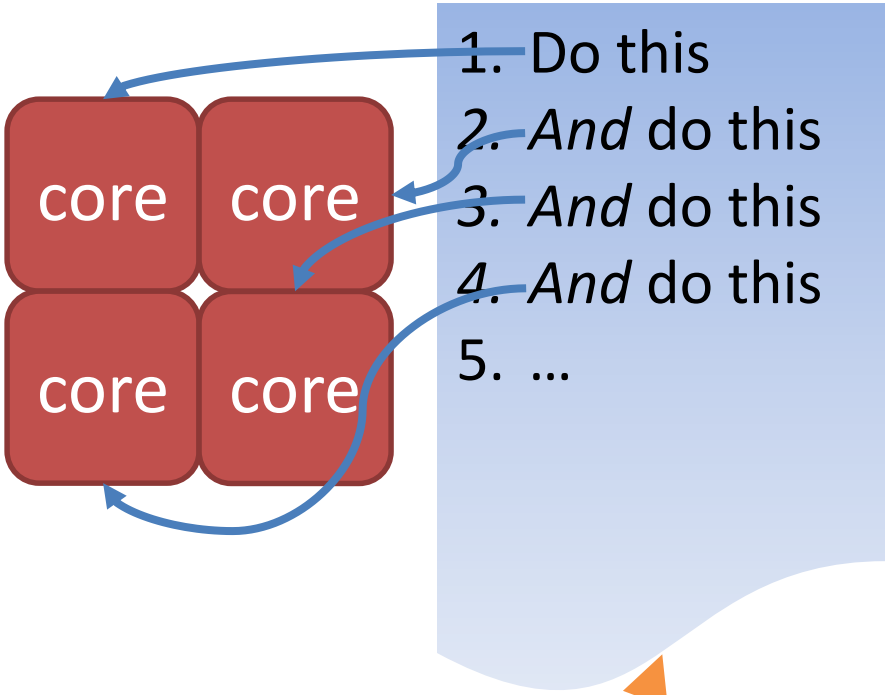


Parallelisation strategies

Serial programming => order



Parallel programming
=> ordered chaos



(someone adds parallelisation)

Parallelised serial code

relatively easy to write

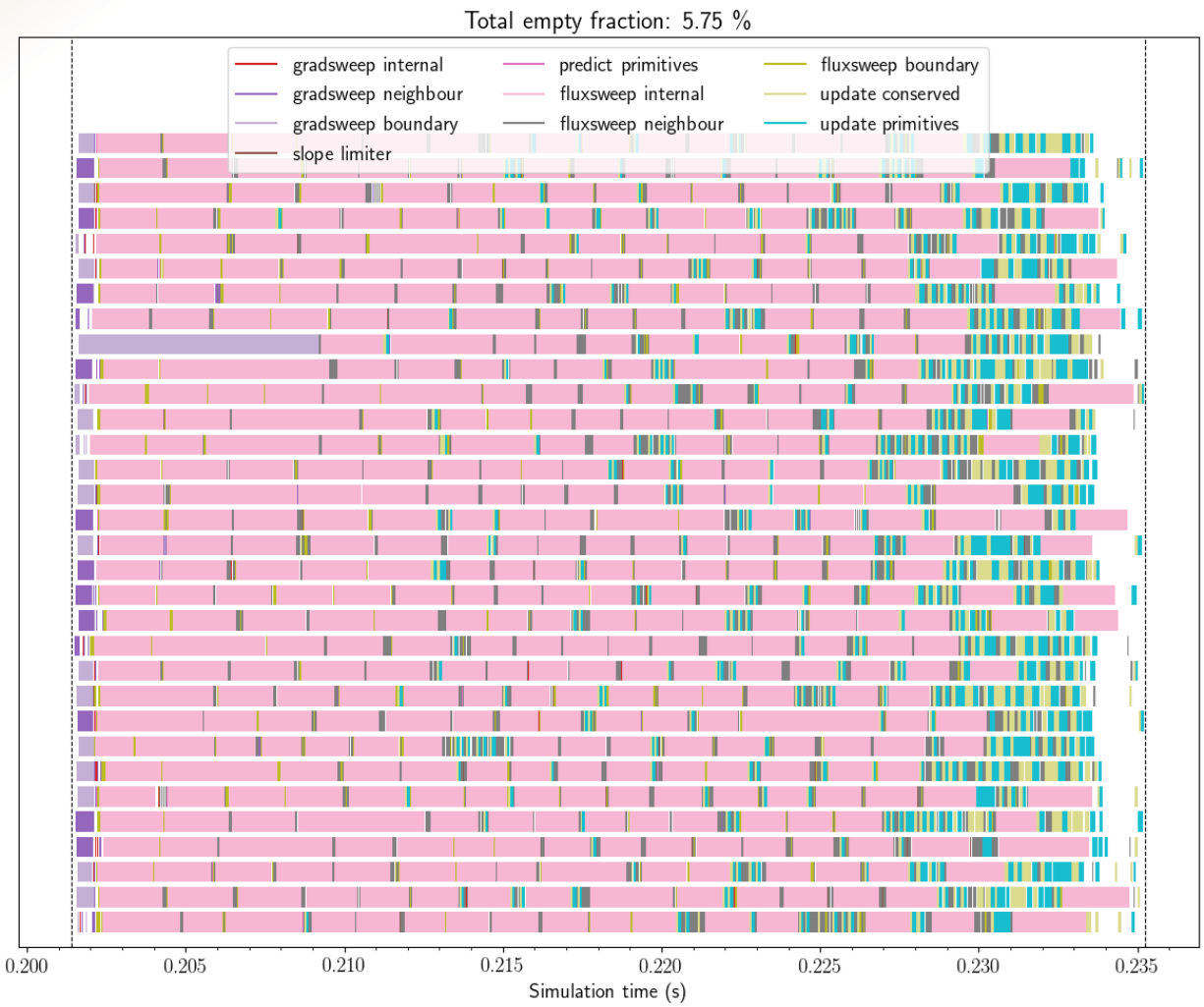
parallelises the sub-steps in a generally serial timeline

will inevitably contain a lot of load imbalance bottlenecks

usually depends on monolithic memory structures,
e.g. a single grid

=> does not strong scale beyond $\sim 10 - 100$ cores

Truly parallel code



Task-based parallel code

requires a complete rewrite of an existing code

constructs a truly parallel timeline in which independent tasks can be scheduled if they do not have unmet dependencies

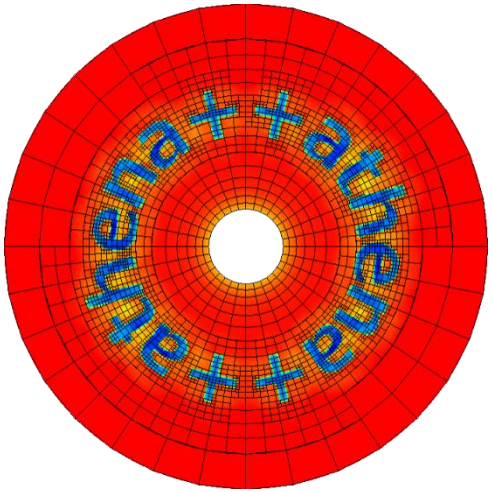
almost completely eliminates load imbalance bottlenecks

crucially depends on small, managed memory structures

=> does strong scale to much larger systems

Examples

(Gonnet et al., 2013; Borrow et al., 2018)



DISPATCH (Nordlund, 2017)

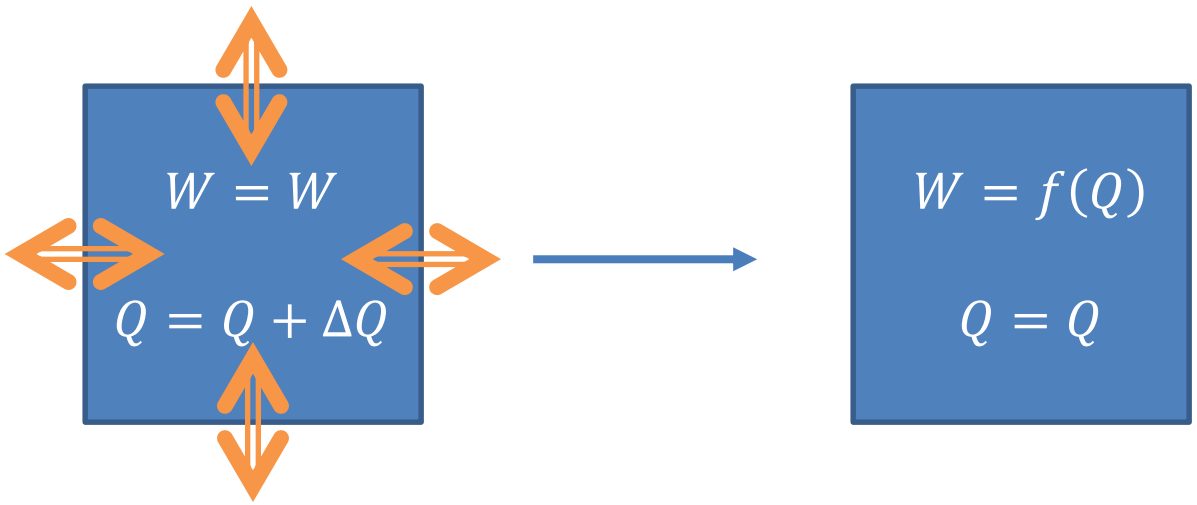
Athena++ (Stone et al., in prep.)

2.0

(Vandenbroucke, in prep.)

How does this work?

Most basic finite volume scheme imaginable

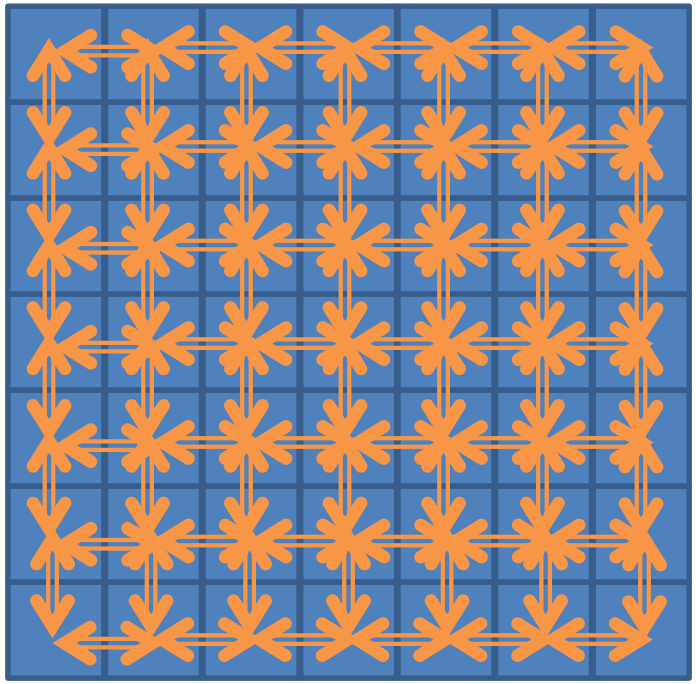


Flux exchange
Update conserved variables

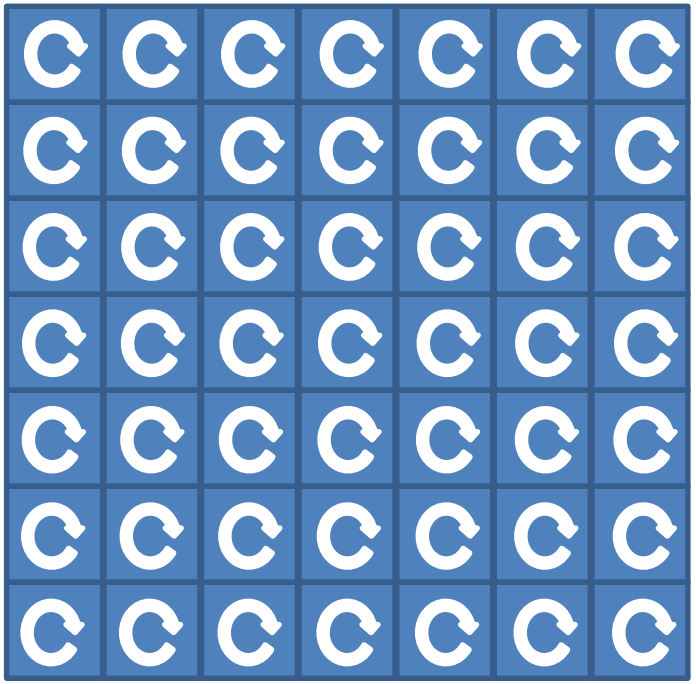
Update primitive variables based on new conserved variables

Serial parallel finite volume scheme

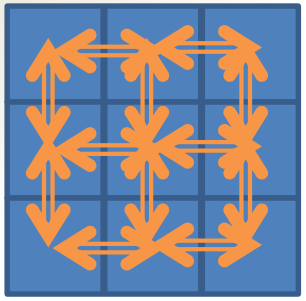
Flux exchange for all cells
Update conserved variables



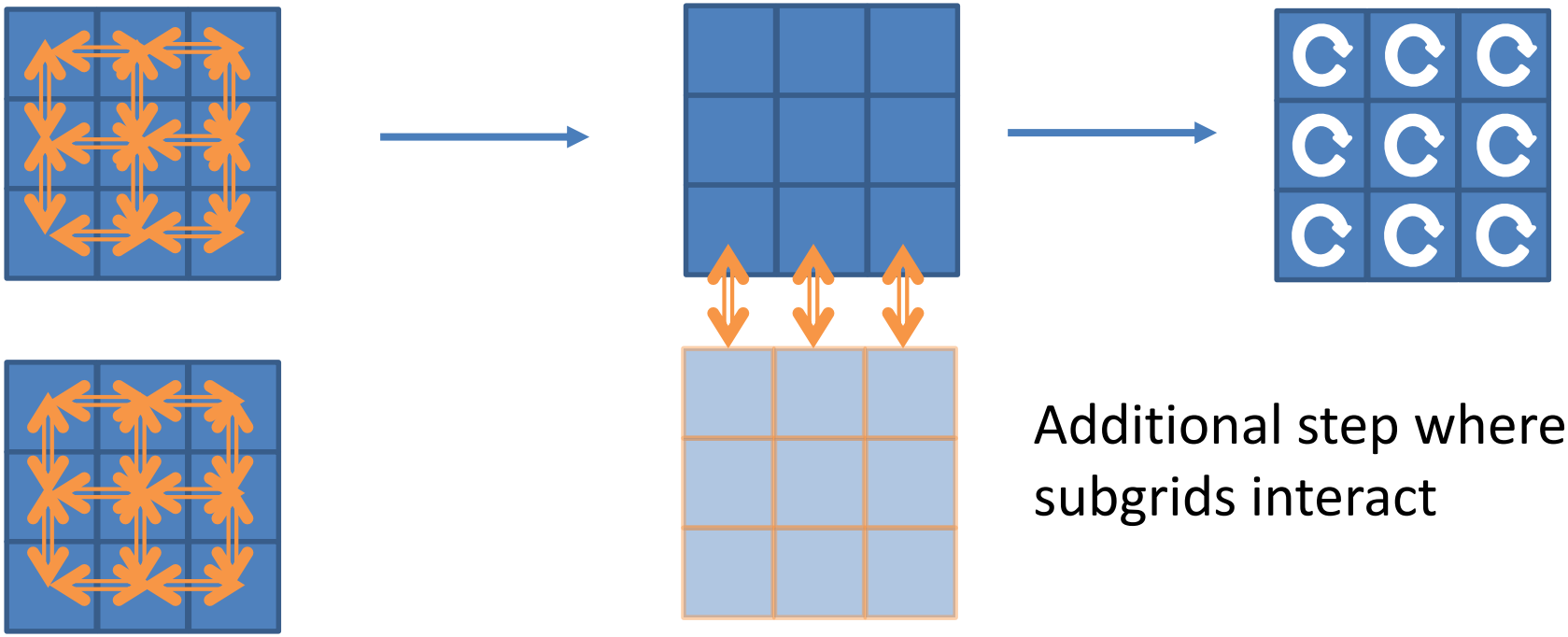
Update primitive variables based on new conserved variables for all cells



Task-based finite volume scheme

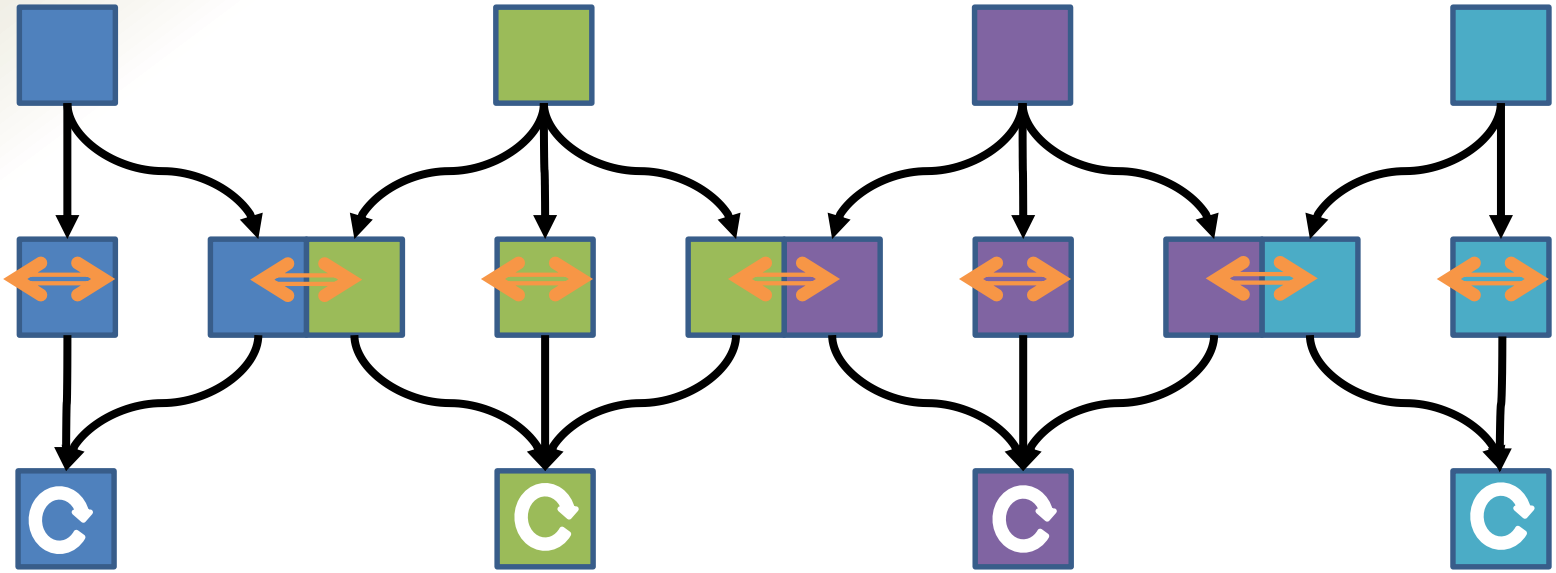


Update small parts of the grid
Each subgrid done in serial, but many in parallel
Serial time line only required for individual subgrids



Additional step where subgrids interact

Dependencies



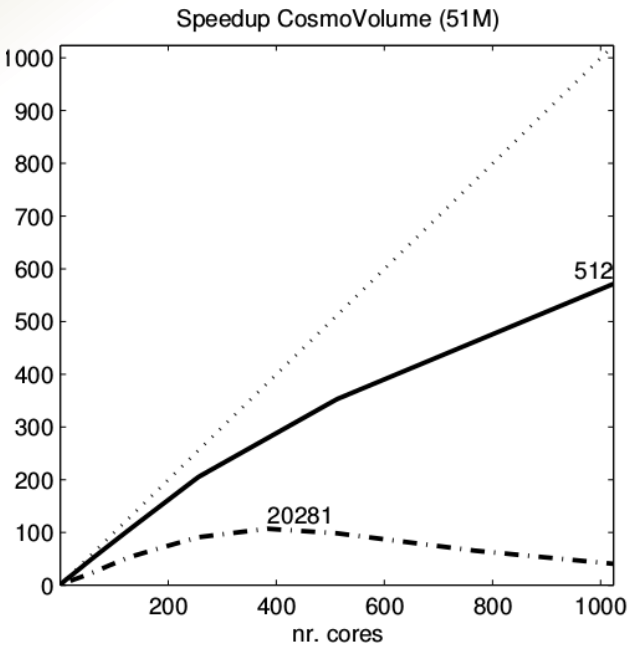
Dependencies encode which tasks cannot be executed:

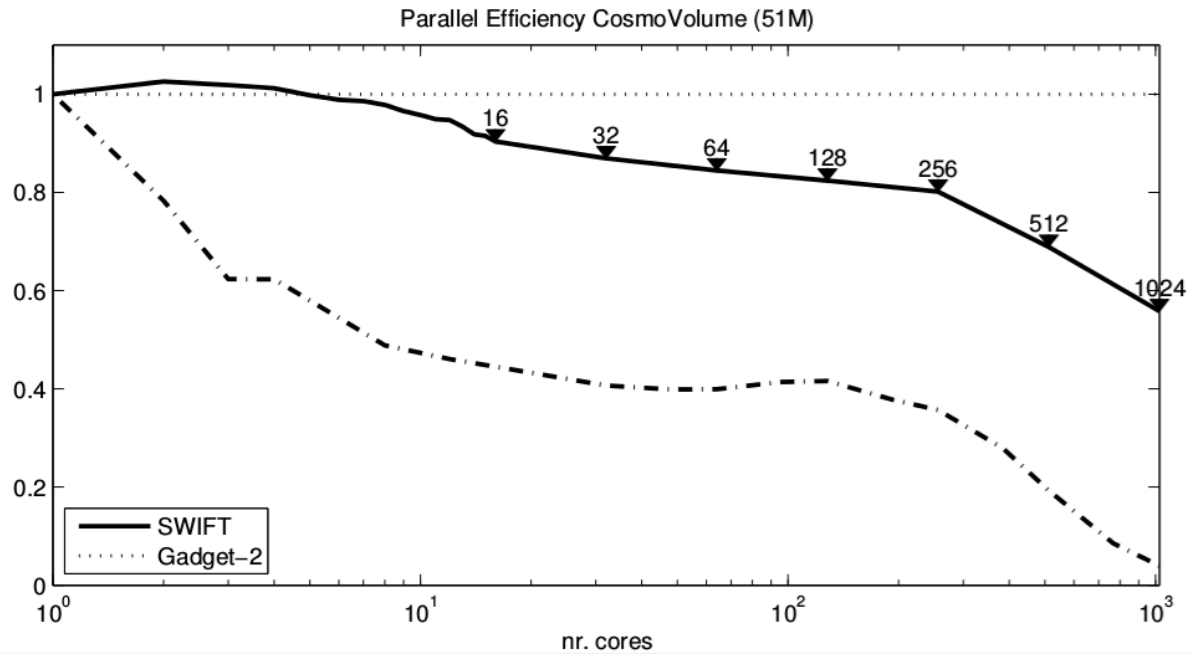
- tasks that require another task to be executed first
- tasks that use resources that are already in use

All other tasks are safe to execute in parallel

Impact

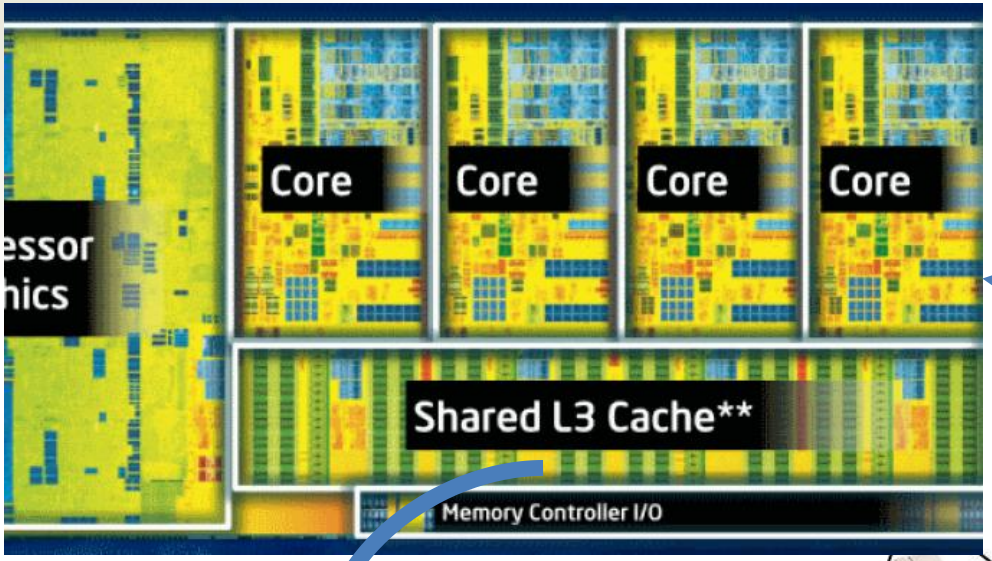
A task-based approach offers a significantly better scaling compared to a serial parallel algorithm for high core numbers





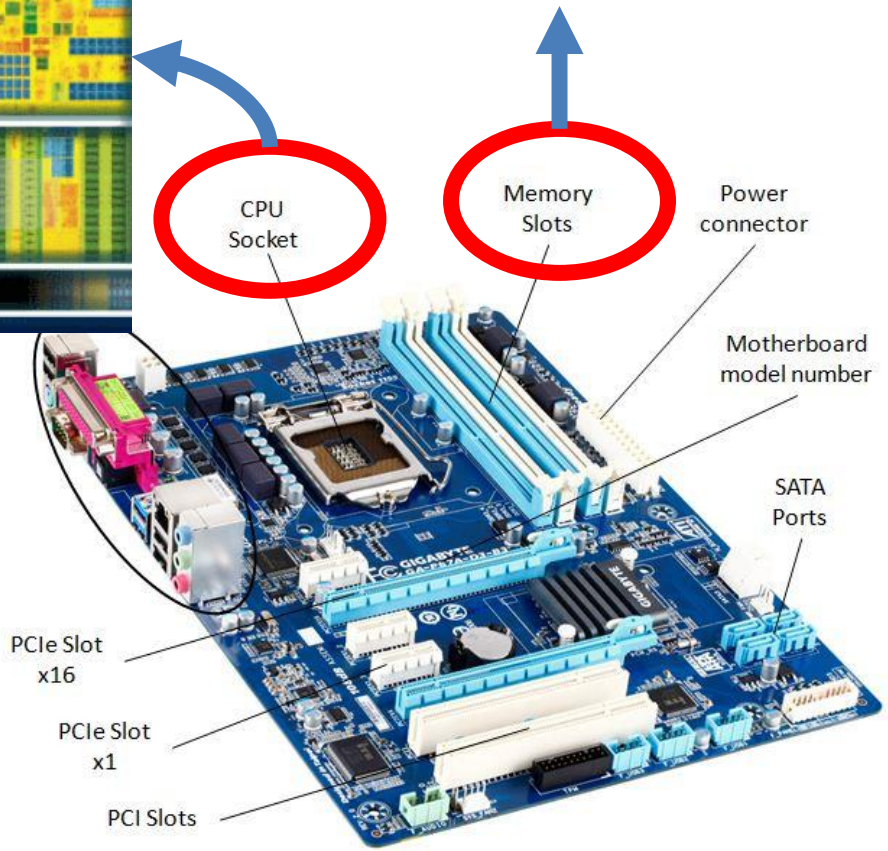
Gonnet, 2013

Memory bandwidth

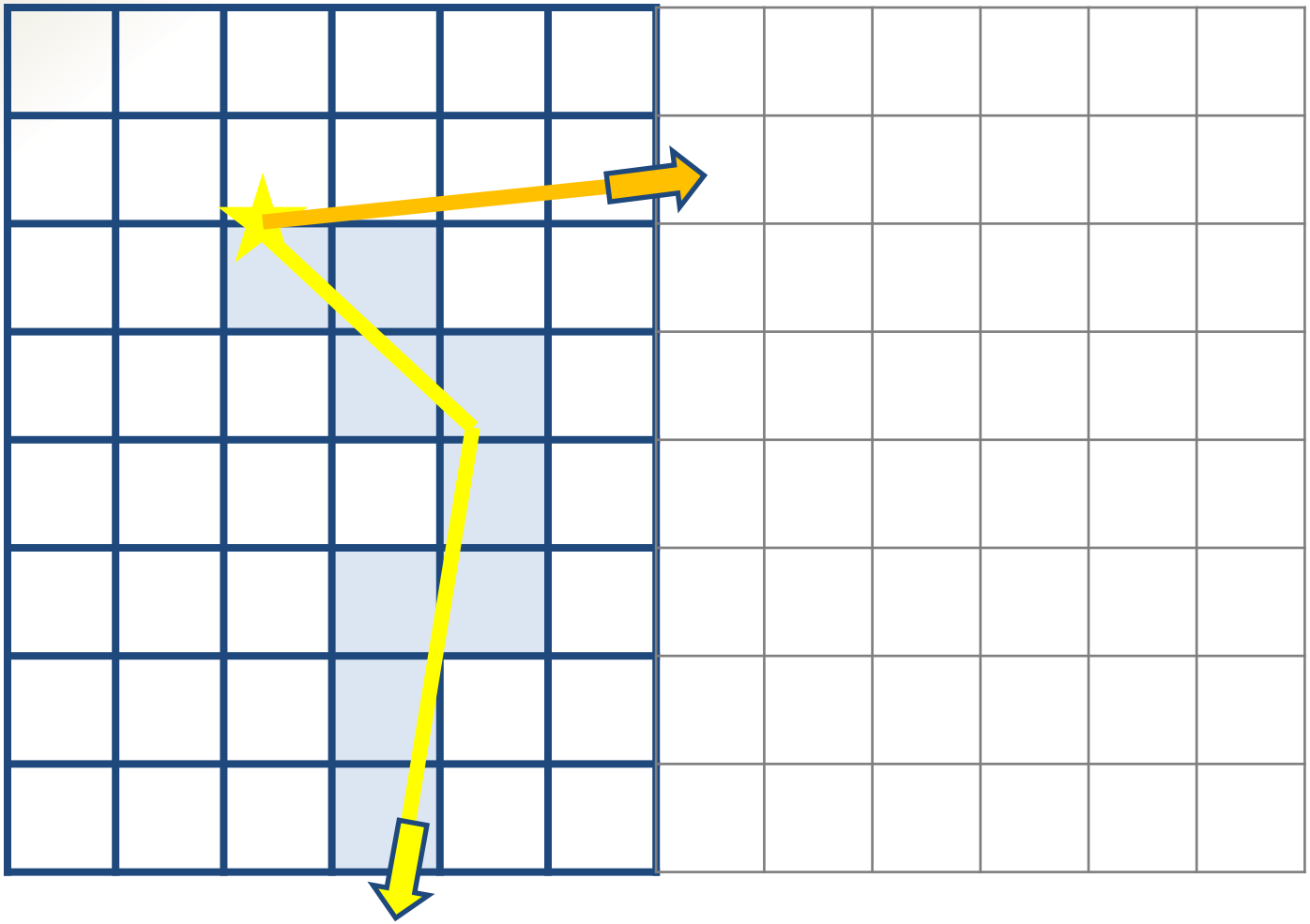


RAM memory
(nowadays: 10 GB – 1 TB)

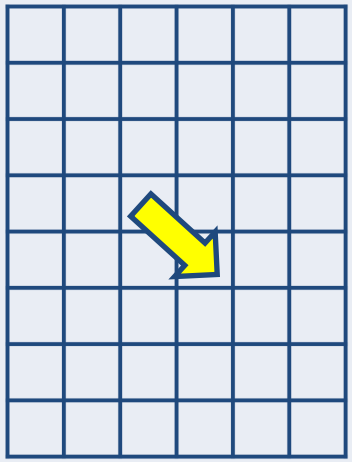
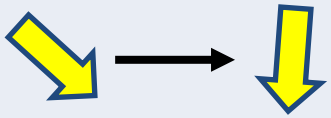
CPU caches
(nowadays: 1 – 100 MB)



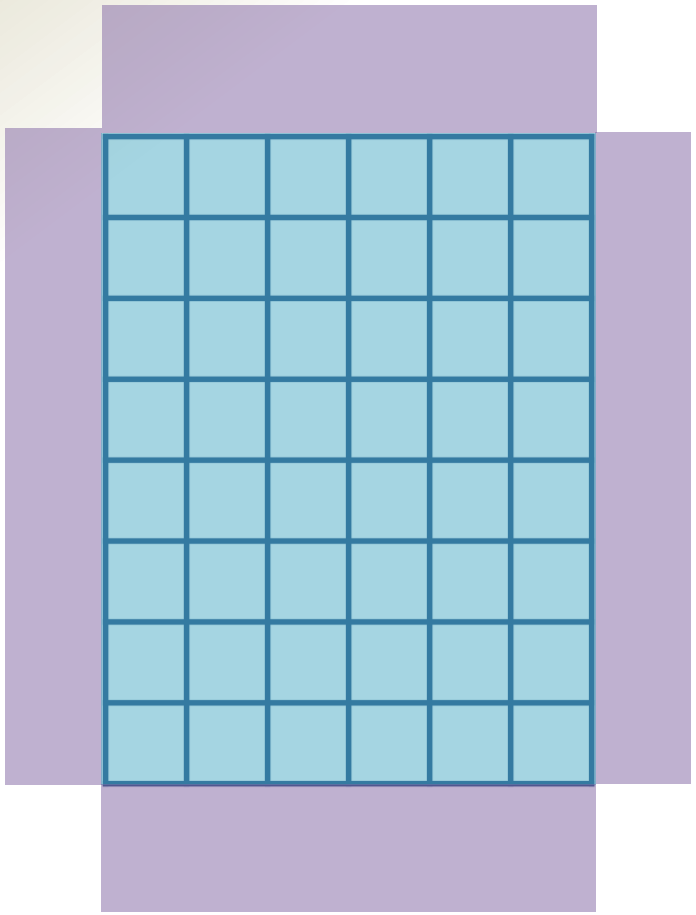
Splitting up the Monte Carlo grid



A task-based MCRT algorithm

	PACKET GENERATION	PACKET PROPAGATION	PACKET SCATTERING
IN	nothing	grid (part), 	
			
OUT		grid (part), 	

Photon packet buffers



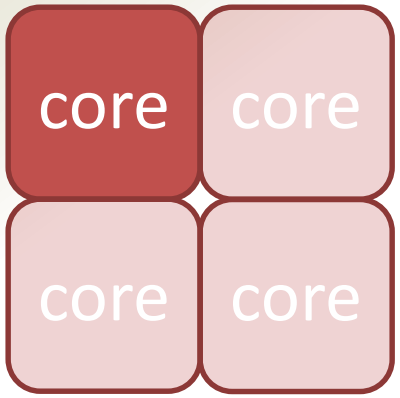
Each grid (part) has output buffers:

- 1 internal buffer
- 6 direct (face) neighbours

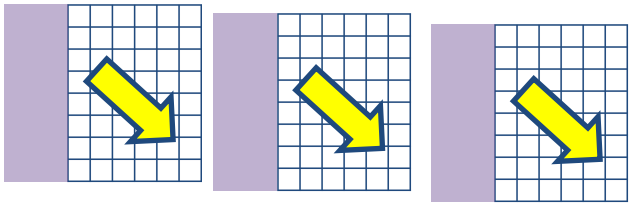
The packet traversal task takes an INPUT buffer and deposits photon packets in the OUTPUT buffers, according to the outgoing direction (absorbed photons are put in the internal OUTPUT buffer)

Full buffers are converted into input buffers for neighbouring subgrids

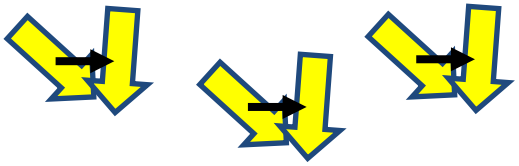
Thread point of view



1.

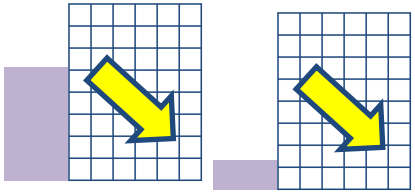


2.



3.

4.



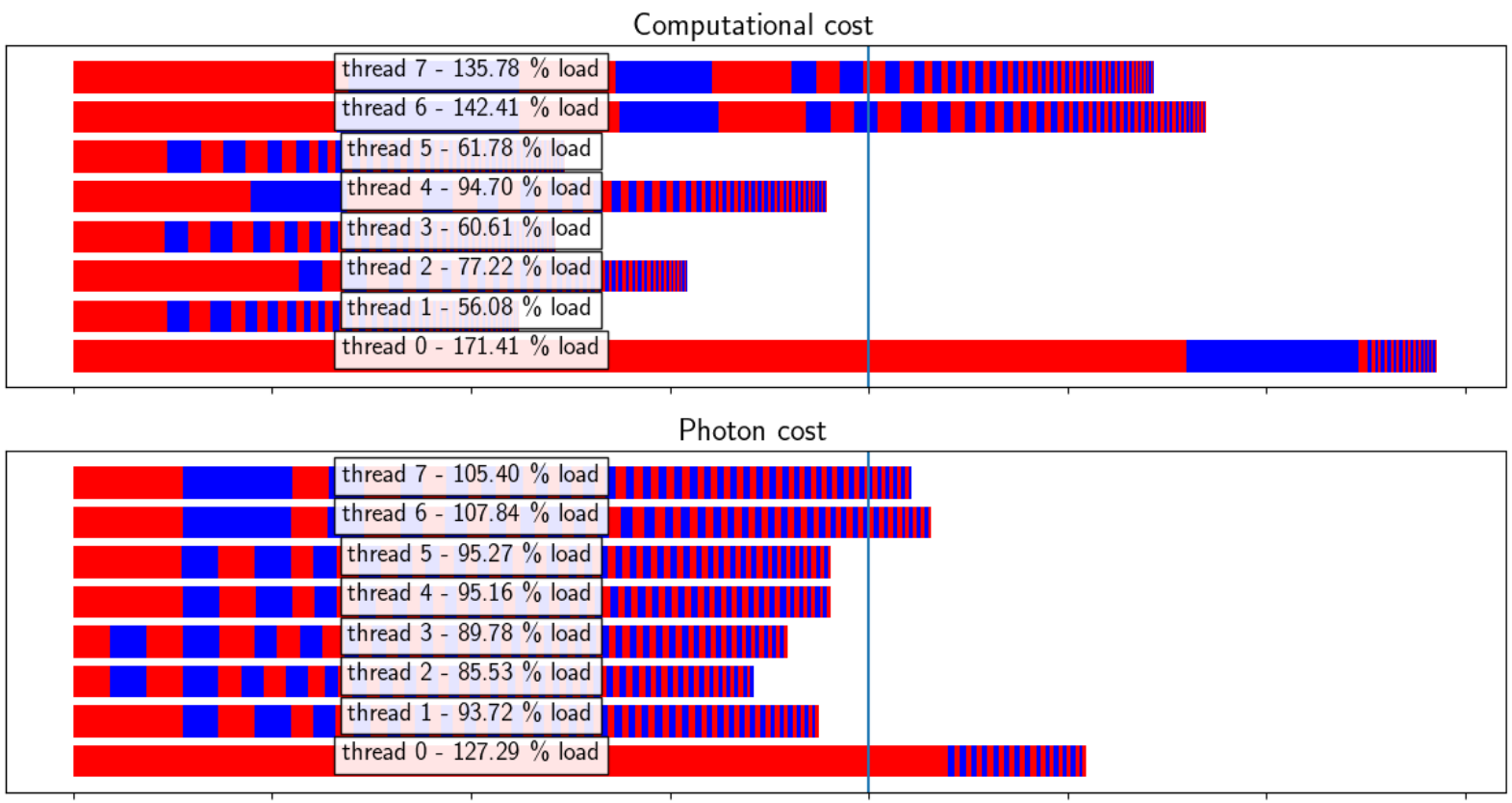
Each thread tries to obtain the first available task with the highest possible priority

This strategy tries to minimize the number of active photon packets

Subgrid load

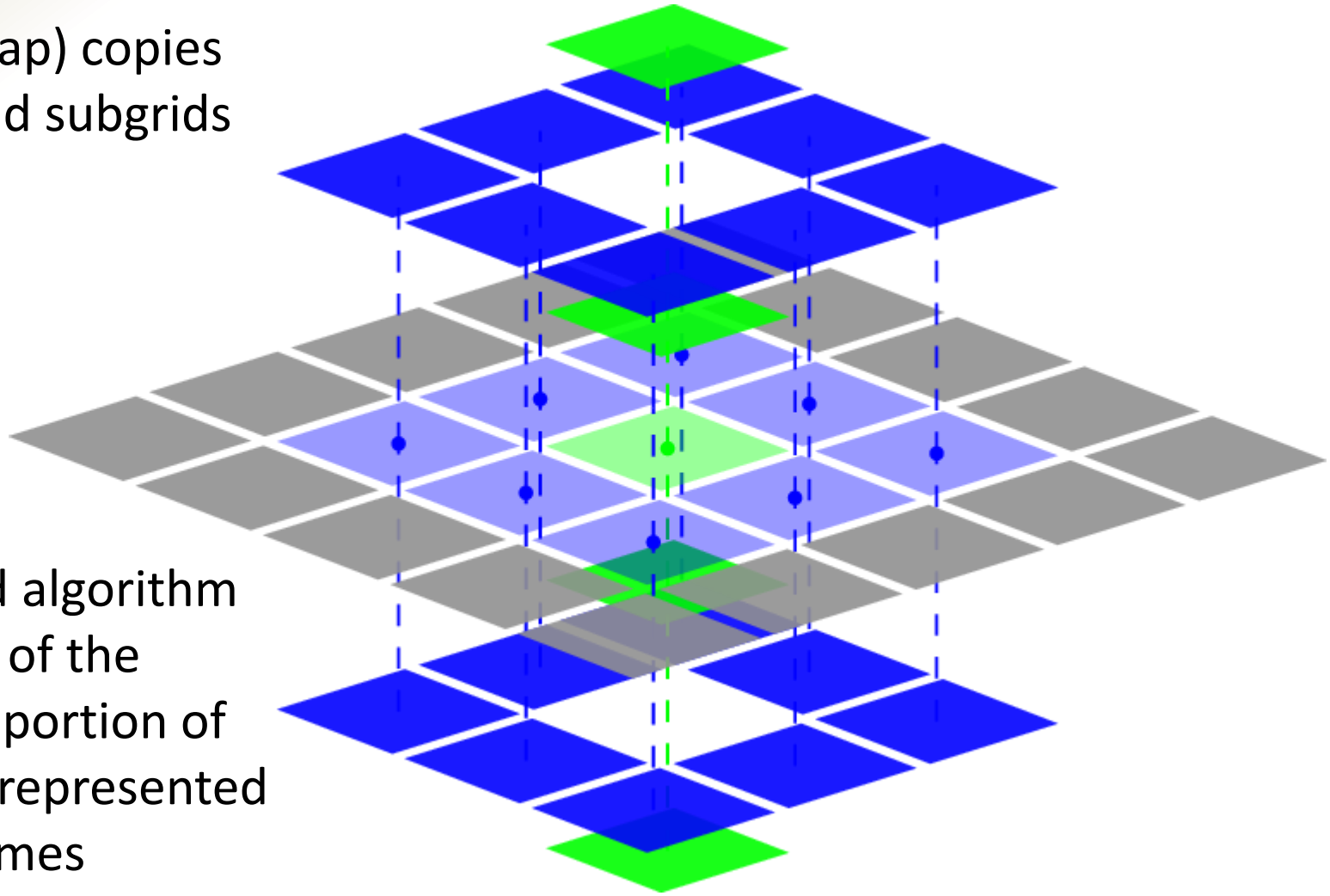
Some subgrids (the ones containing sources) have a significantly higher load than others

This load can be higher than the average load per thread



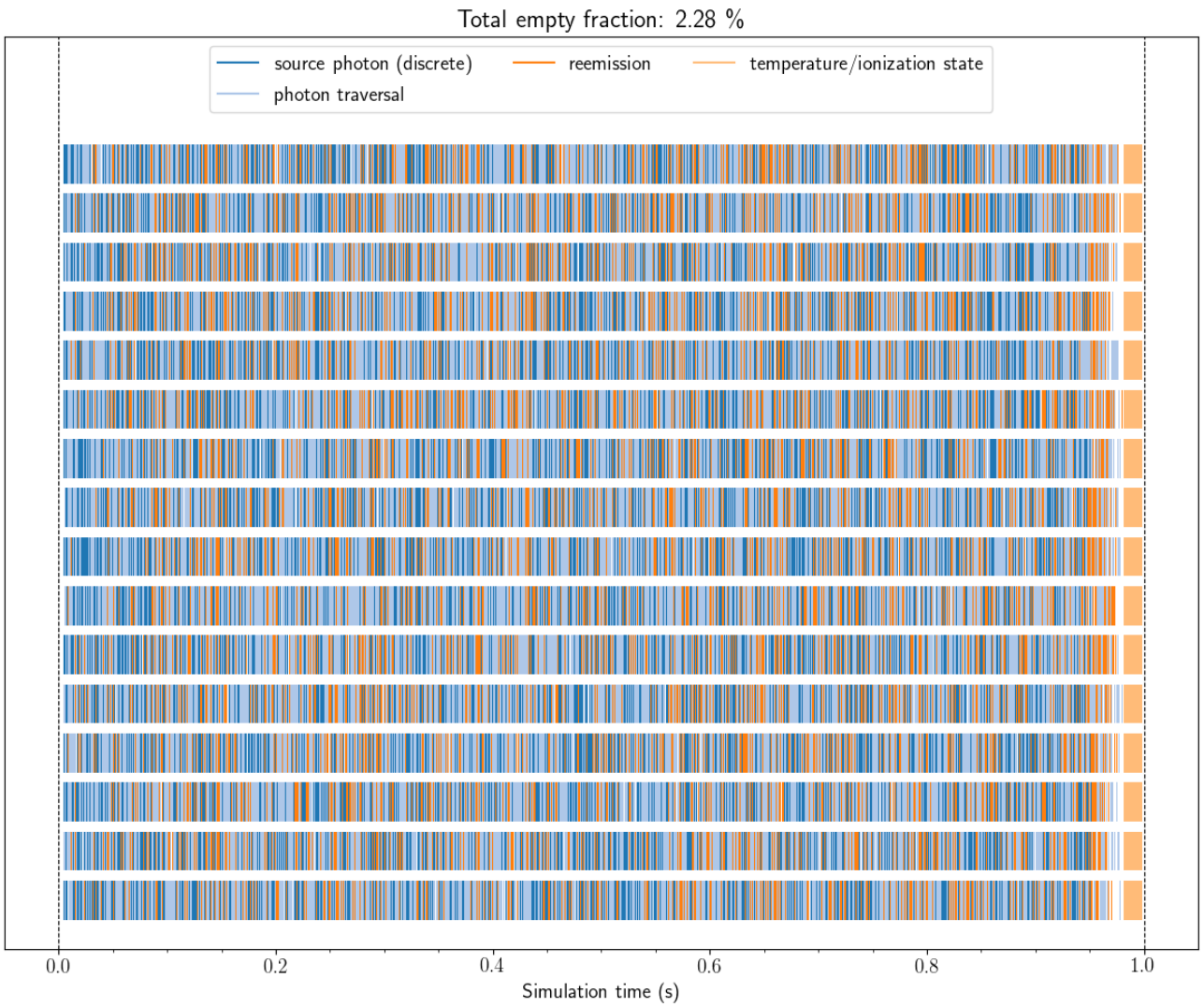
Subgrid copies

Make (cheap) copies of high-load subgrids



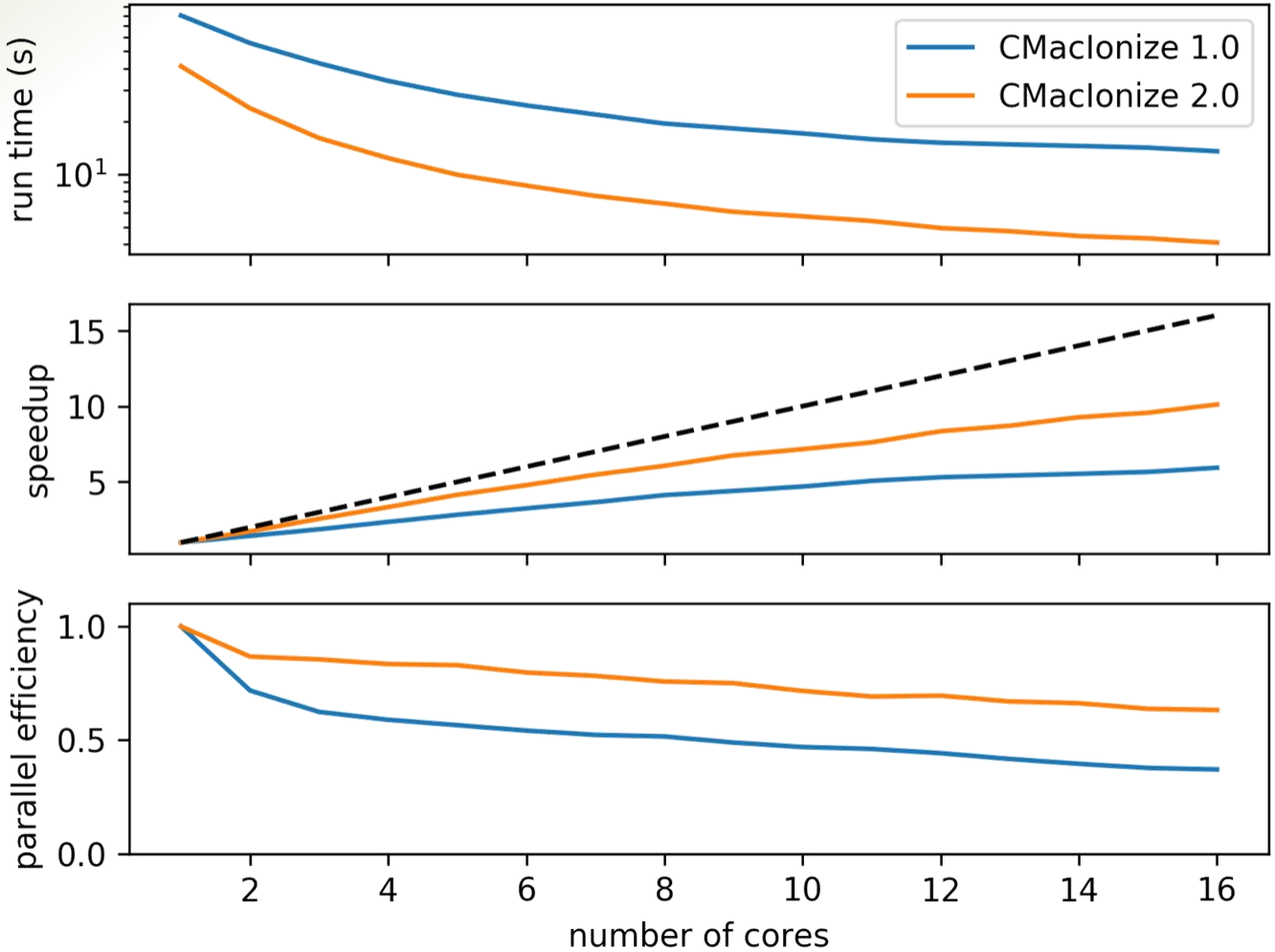
Task-based algorithm is agnostic of the fact that a portion of the grid is represented multiple times

Does this work?



CMacIonize 2.0

Strong scaling



Summary

Moore's law for single CPUs is dead

$\Rightarrow$ simulations no longer get faster for free

Huge speedup available on parallel systems,

BUT requires parallel code

Parallelisation not so much a matter of OpenMP/MPI(/…), but of a good parallelisation strategy => truly parallel algorithms

Summary (2)

Good parallel code requires computer science

$\Rightarrow$ people that know how computers work

$\Rightarrow$ people that care about making code faster

$\Rightarrow$ probably not astronomers

Writing your own code is very good to learn about algorithms and to get experience,
BUT you should probably not use it if you want to run state-of-the-art simulations