

Performance considerations and introduction to parallel computing

CTBP Tech Talk Series

10/29/25

Paul Whitford

Northeastern University

Disclaimer

I'm *only* a good user, not an HPC programmer.

This will not be a comprehensive description of the newest hardware and methods. Instead, let's understand the principles one should be familiar with, in order to navigate the dynamic technical landscape and maximize scientific progress.

What you need to do

1. Learn the terminology – No need for a CS degree, but don't be completely ignorant.
2. Get familiar with the hardware – CPUs, I/O speeds, memory, GPUs, etc.
3. Know your code – e.g. script vs. program
4. Know where and how the code is parallel
 - Understand the algorithms sufficiently well
 - Understand the strength/limits of the applied modes of parallelism
5. Always monitor performance and compare against theoretical peak performance.
6. Always monitor performance.
7. Always monitor performance.

Some useful terms

- “thread” – a sequence of programmed instructions that can be executed independently
 - Threads exist at the software level.
 - If you have ever programmed and you were not familiar with the concept of a thread, then you were probably/maybe only using one.
 - More threads does not necessarily mean more CPU power is available for your calculation.
- “core” – a physical processor that can independently perform calculations
 - Modern CPUs have many cores, where each can be used independently by different tasks/threads/programs. Before the 2000s a “CPU” and “core” were one in the same.
- “node” – a single server within a cluster. Has its own board, CPUs, memory, GPUs, etc.

More useful terms

- “bandwidth” – maximum rate that data can be transferred between two points
 - Can be between computers
 - May be between cores on a node, between cores and hard drive, or between the CPU and GPU, etc.
 - Bigger=better
- “latency” – time lag between initiating a message between cores/nodes.
 - Smaller=better
- “FLOPs” – Floating-point Operations Per Second
 - How many arithmetic operations (+ - x /) can be performed per second
 - On a modern CPU (single core) – typically around 10 Giga-FLOPS (10^{10})
 - On a high-end GPU (e.g. NVIDIA H100) – around 50 Tera-FLOPS (5×10^{13})
 - Keep these numbers in mind when running code. It is always recommended that you try to estimate the number of operations in your code, to make sure you are getting speed that is reasonable. If not careful, you may get orders-of-magnitude lower performance than is possible with your hardware.

What is “parallel computing”?

- Any time that the computer is working on two operations at the same time
- Can
 - be on the same node
 - be on different nodes
 - use full/partial nodes
 - share memory, or not
 - use GPUs
 - use any combination of the above

Example of a cluster

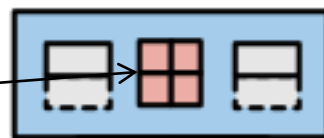
- BlueGene/P supercomputer (ca. 2010)
 - PowerPC 450 processors: 850 MHz
 - High bandwidth and low latency connections
 - Many slower cores can work together at the same time, making the calculation fast.



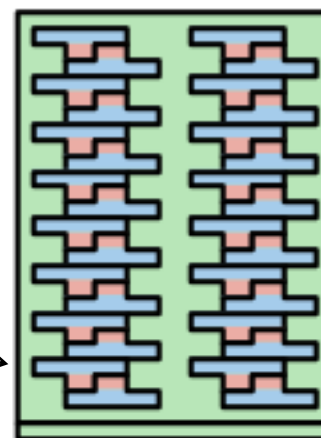
Blue Gene/P chip



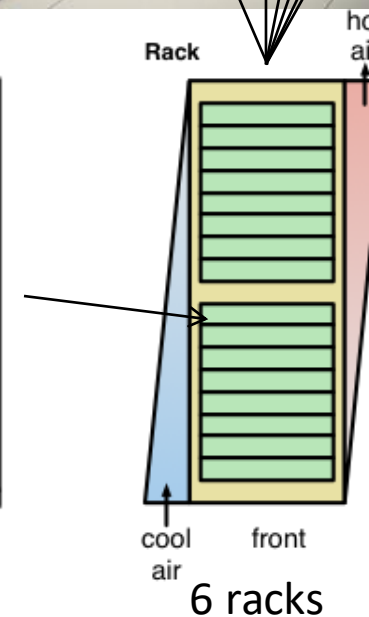
24,576 cores



6,144 cards

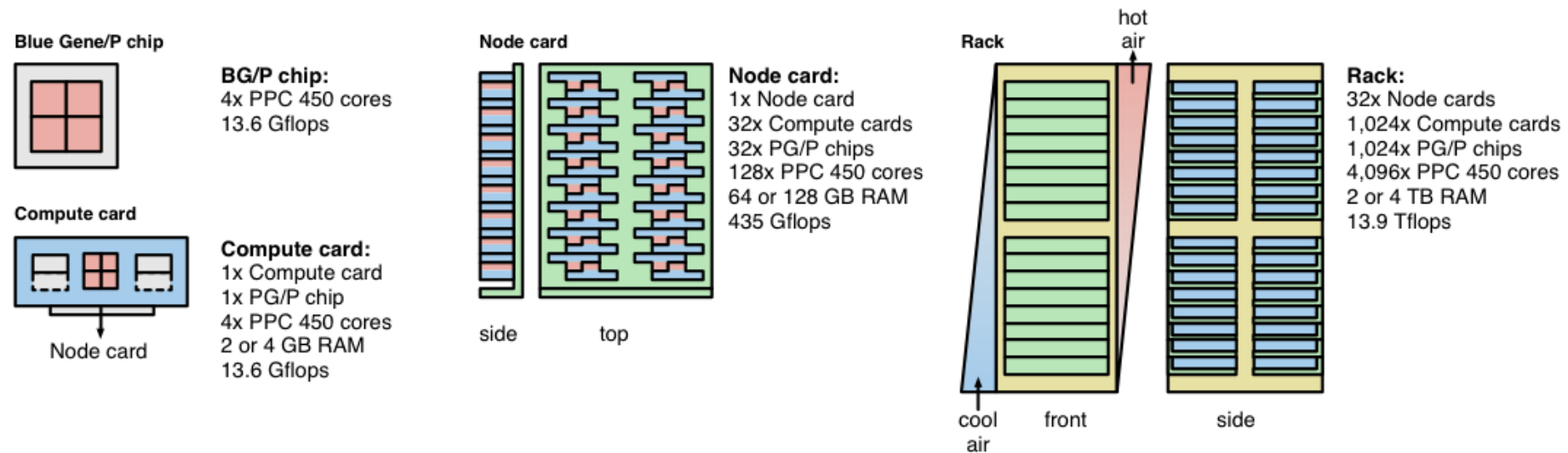


192 cards



2. Understanding the hardware

Physical attributes of a cluster



Each chip has multiple cores (here, 4)

Each node can have many chips (here, 32)

Modern clusters typically have 2-4 CPUs/chips per node, and each will have ~10s of cores

A node can also have GPUs, which work in parallel with the CPUs. Single nodes can have multiple GPUs.

Know your code

- Are you using scripting (e.g. Python) or using compiled code (e.g. C++)?
- Scripting involves an interpreter (e.g. Python, Perl, awk, bash, tcl)
 - When you run a script, the code is read and run, line by line
 - Warning: This requires computational overhead (i.e. slower performance)
 - If the code is ambiguous (e.g. define a variable without a type), the interpreter often guesses what you mean
 - Warning: can lead to erratic results – The calculation can be completely wrong without any indicator.
 - Recasting is often automatic (e.g. a variable is used as a string, and then as an integer)
 - Very very slow
- Compiled code involves building a binary executable from a text file
 - The compiler can optimize the binary. This means it must analyze the full flow of data before you ever try to run the program.
 - Makefiles often automatically select a level of optimization (e.g. -O2), but it is good to double check.
 - Syntax in compiled code is more strict/explicit. Compilers don't guess what you mean.
 - **Generally, everything that is fast is compiled.**

If Python is slow, why is it the “standard” language these days?

- Python has libraries available, but the libraries are not written in Python.
 - NumPy – mostly written in C, but often accessed through a Python interface (API)
 - OpenMM – written in C++, accessed using Python
- You CAN compile Python code, also
 - Using Numba, you can build the code “on the fly”. That is, the code is converted to machine code immediately before execution, rather than being interpreted
 - <https://numba.readthedocs.io/>
- Why does this matter?
 - If you are unaware, you *may* be doing things effectively, or you may be wasting a lot of time.
 - e.g. you access Python libraries and mix their use with native Python instructions. The Python steps may become a bottleneck.
- Use I/O speed and FLOP estimates to be sure things are not too slow
 - Don’t trust the performance is optimal, even when using NumPy.

Understand your mode of parallelism

Types of parallelization

- SIMD (vector-based acceleration)
- CPU-based openMP/threads
- CPU-based MPI
- GPU-based “massively” parallel calculations

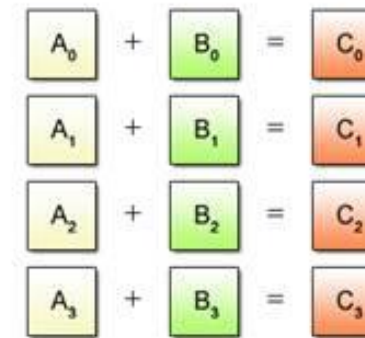
We will only discuss parallelism in the context of compiled code.

SIMD – single instruction, multiple data

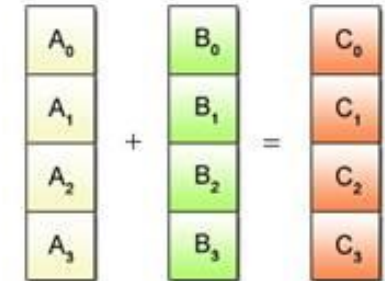
Possible

- Vector-based calculations
- Idea:
 - The CPU register may be able to hold more data than a single variable
 - Load multiple pieces of data into a single processible unit
 - CPU will process identical instruction for all pieces at once
- Sometimes (auto)enabled at the compiler level
 - e.g. SSE, or AVX acceleration
- Manual coding can be very challenging
 - e.g. non-bonded (vdW) routines in GROMACS
- Before each operation, data must be organized for SIMD processing
- Can improve performance on modern cores
- More doesn't always mean faster
 - E.g. AVX256 can be faster than AVX512, even though 512 processes twice the number of vector elements per cycle
 - Slow down can be due to different clock speed with larger registers.

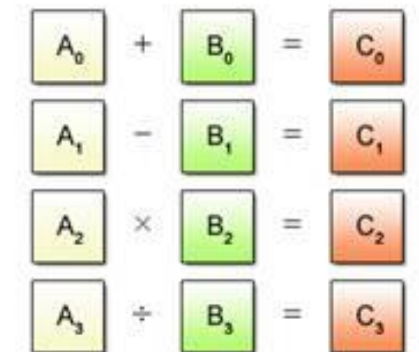
(a) Scalar Operation



(b) SIMD Operation



Not possible



Thread-based parallelization (e.g. OpenMP)

- All threads are on a single node
- Each thread is run on one compute core (threads may share a core)
- All threads share/access the same memory
- Usually good for parallelizing simple portions of code (e.g. “for” loops, initialization, I/O)
- Often only efficient for a low thread count (~10) – depends on the calculation
 - Threads can compete for the same data, or wait for one another to finish.
- Simple to implement (available by default with most/all modern compilers)
- Idea: all threads execute the same instructions on different elements of memory
 - e.g. in a for loop over “i”, core 1 will runs i=0-100, cores 2 will handle i=101-200, etc
- Introduction at https://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPslides_tamu_sc.pdf

Threads are everywhere

- Almost all languages have some capacity for thread-based parallelism
 - Python
 - Fortran
 - C++
- Threads come in different flavors
 - POSIX threads (pthreads)
 - OpenMP

MPI – Message Passing Interface

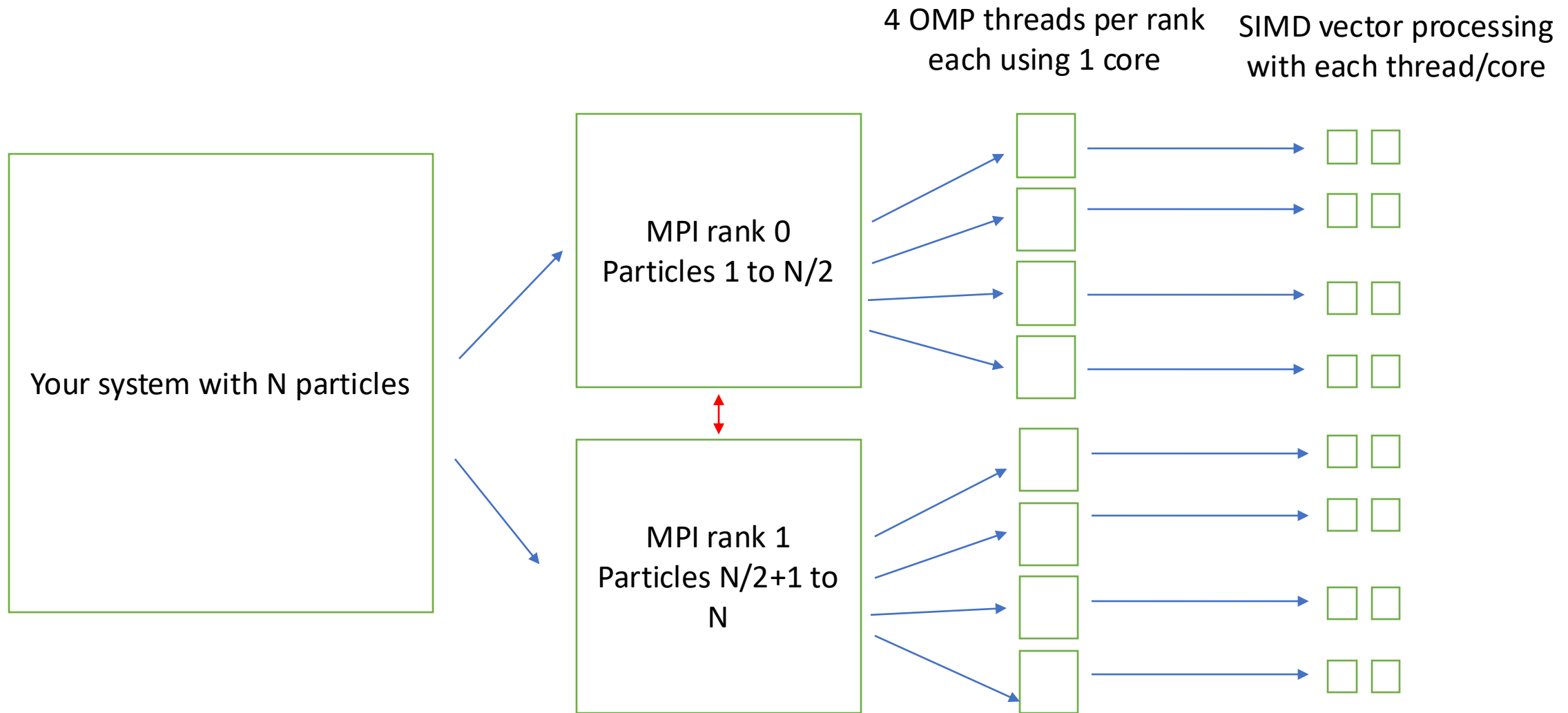
- Each MPI “rank” has separate memory
- Each rank controls its own instructions
- Ranks may be on the same, or different, nodes
- Instructions on each node may be independent of one another
 - e.g. rank 1 could work on 6-12 interactions, rank 2 works on PME, etc
- Usually good for parallelizing complex portions of code, or large-memory calculations (assuming memory doesn't need to be shared)
- Performance strongly depends on the connection between ranks/nodes
 - Latency is often more important than bandwidth
 - The faster the cores/nodes can communicate, the better calculations will scale
- *NOT* simple to implement
 - Lots of room for inefficient calculations
 - Possible that ranks will be sitting idle

Multi-level parallelization

- May integrate MPI and openMP/threads in the same calculation
- Idea:
 - Break a large calculation into chunks with MPI ranks (e.g. molecular system into sets of atoms)
 - Each chunk can have many cores working on it via threads
- Each rank has its own memory
- Within each rank, the threads share memory
- Can allow for very highly-scalable calculations
- Still typically limited to a low thread counts per rank (~10)
- May also integrate SIMD operations
- Nightmare to write the code...

4. Understanding parallelism

Example of multi-level parallelization (as implemented in Gromacs)

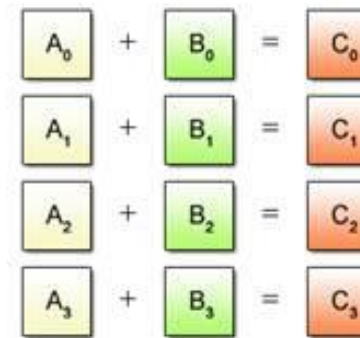


GPUs – “extreme SIMD”

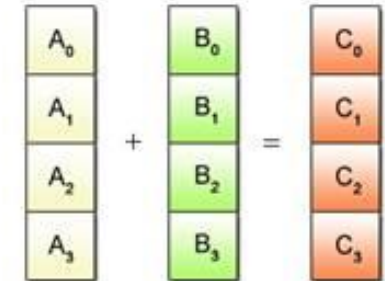
- GPUs are massively parallel
 - A single AMD MI350 has 16,384 cores!
- Unlike CPUs, the cores can not simultaneously work on independent operations/instructions
- Data can only be processed in a SIMD-like form
- Challenge:
 - One may need to organize massive amounts of data for each cycle of the GPU to process
 - Organizing/streaming the data can rapidly become rate limiting
 - Since each cycle of the GPU can only process a single instruction, conditional statements become extremely slow (must execute all possible true and false variations of each calculation)
- If at all possible, leave GPU programming for the pros
- Can provide incredible speedup, or slowdown...

Possible

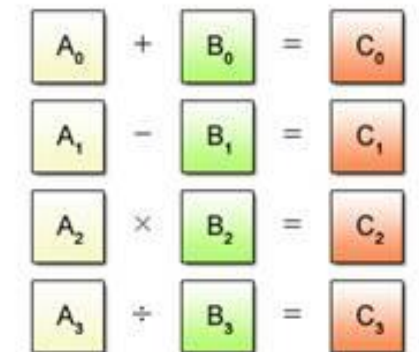
(a) Scalar Operation



(b) SIMD Operation



Not possible

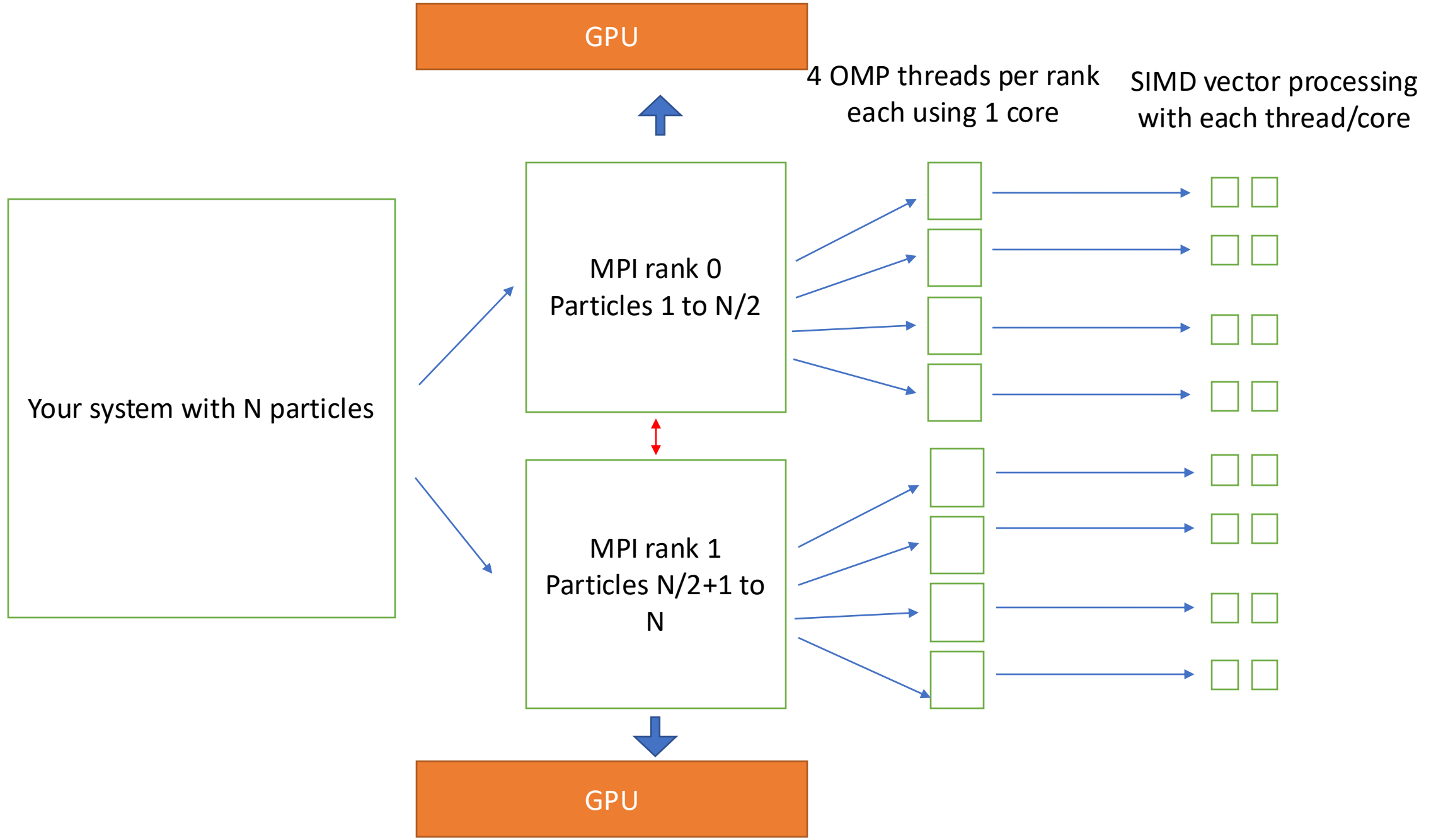


GPU Challenges

- Data is not always easily streamlined for transfer to the GPU
- Communication between the GPU and CPU can be rate limiting
- Ideally one loads a small amount of data on the GPU and does a lot with it
 - Best for repetitive matrix operations that have no conditionals

4. Understanding parallelism

Example of multi-level parallelization with GPUs (as implemented in Gromacs)



GPU computation with OpenMM

- The nonbonded routine takes most of the time in MD
- OpenMM uses a “tiling” strategy
- GPUs can effectively apply all-all force calculations between two tiles
- Just need to sort which tiles are interacting and then a small amount of data is given to the GPU with minimal organization and high compute demand.
- See figure from Eastman and Pande, 2009.

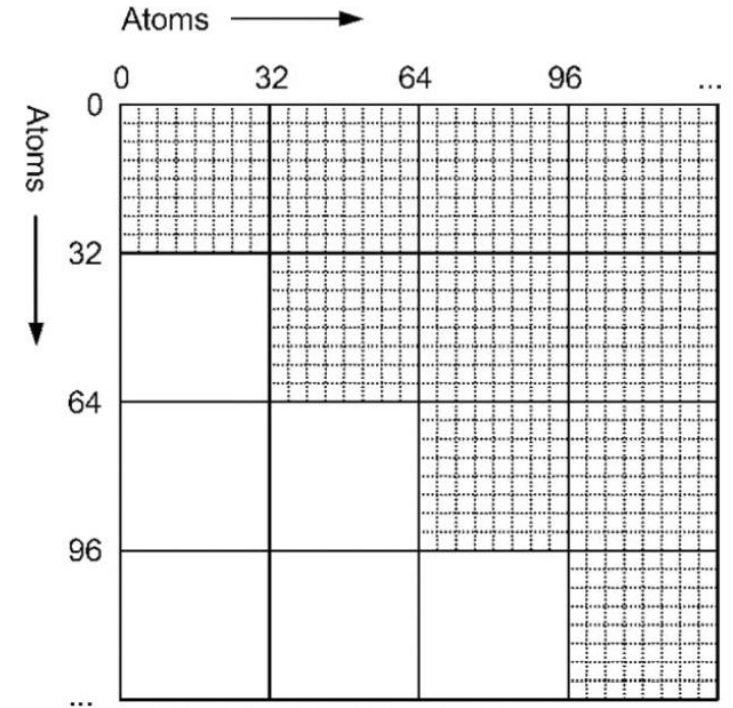
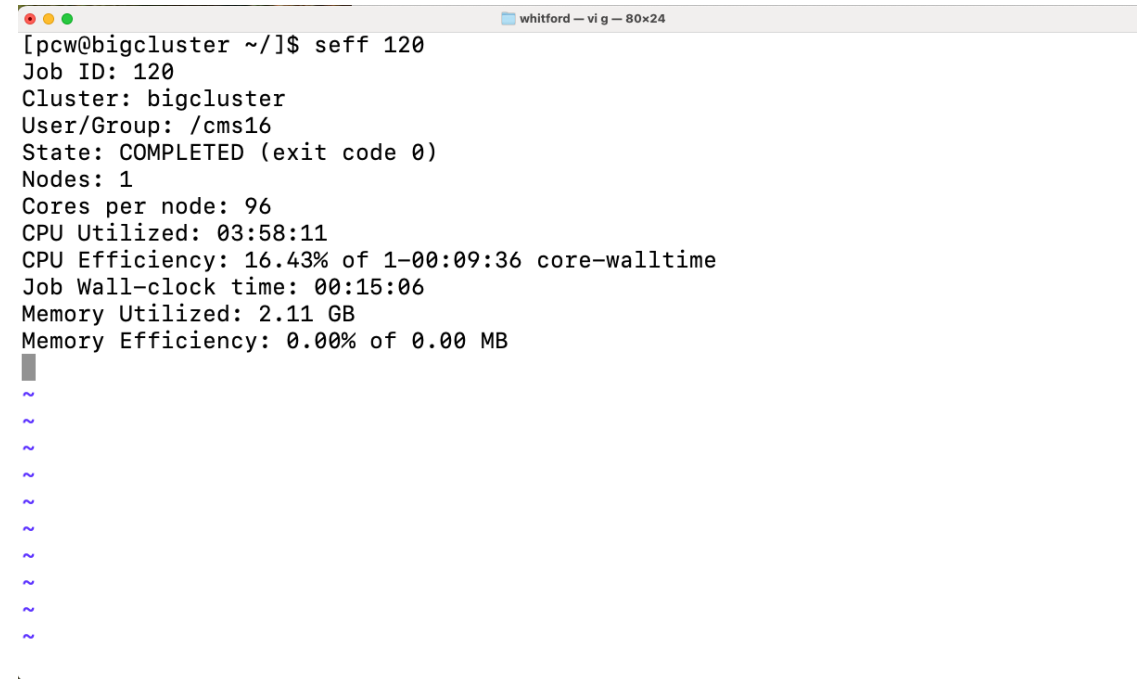


Figure 1. Atoms are divided into blocks of 32, which divides the full set of N^2 interactions into $(N/32)^2$ tiles, each containing 32^2 interactions. Tiles below the diagonal do not need to be calculated, since they can be determined from symmetry.

Common performance issue

- Underutilization of a requested resource is common (but not acceptable).
- At a minimum, when running on a cluster, you should check if the requested hardware was used by your job.
- SLURM has the “seff” command for this.
 - Reports memory and CPU usage
 - Example shows a job where only 8/96 cores were used, but doesn’t specify what was going on with the 8 GPUs
- seff only tells you if the hardware was being actively used. It doesn’t tell you whether the calculation was actually running quickly.



```
[pcw@bigcluster ~/]$ seff 120
Job ID: 120
Cluster: bigcluster
User/Group: /cms16
State: COMPLETED (exit code 0)
Nodes: 1
Cores per node: 96
CPU Utilized: 03:58:11
CPU Efficiency: 16.43% of 1-00:09:36 core-walltime
Job Wall-clock time: 00:15:06
Memory Utilized: 2.11 GB
Memory Efficiency: 0.00% of 0.00 MB
~
~
~
~
~
~
~
~
~
~
```

Common factors that can impact performance in GPU-based calculations

- Bandwidth between CPU and GPU
- Calculations are too small to use the full GPU
- CPU code may not be fully GPU-supported, leading to substantial CPU-GPU communication
 - OpenMM-Plumed has had this issue for us.

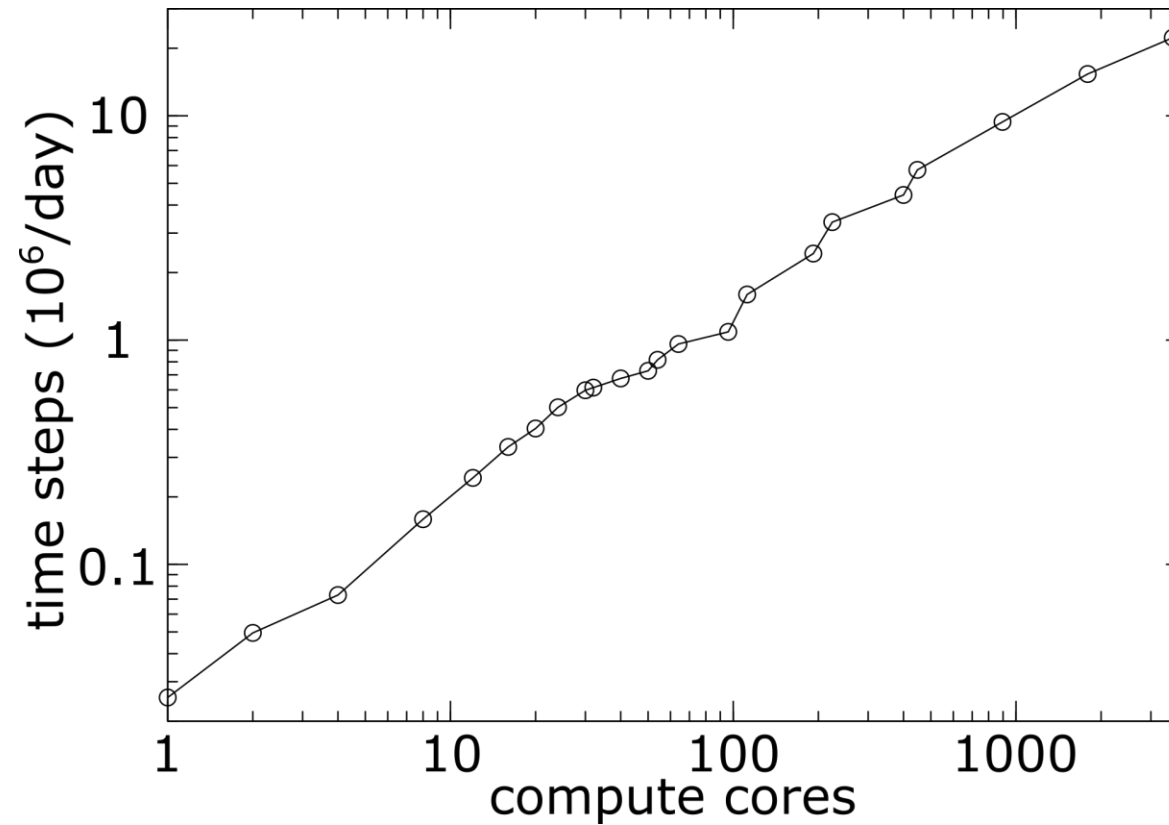
How to figure out what works?

- Read about the algorithms and then test it...
- Test for strong scaling
 - Fixed-size calculation (e.g. N step simulation)
 - Perform on 1 core, 2 cores,...
 - “strong scaling” if time to solution scales with $1/\#$ cores
- Test for weak scaling
 - Variable-size calculations
 - Perform:
 - Small calculation on 1 core, larger on 2, larger on 4, etc.
 - Weak scaling if time to solution remains constant.

Strong scaling strategy

- Design a simulation that is representative of the production calculation
 - Typically the same system for fewer timesteps
- Simulate the system using 1, 2, 4... 2^n cores
- Monitor the time to complete all non-initialization steps
 - e.g. any one-time calculations (e.g. loading the input deck) should not be included
- Plot $1/\text{time}$ versus number of cores
- Compare to linear extrapolation from 1 core
- Can quantify scalability as $(\text{time on 1 core}) / (n * (\text{time on } n \text{ cores}))$
 - If perfectly scalable, then each core will have $1/n$ of the load and take $1/n$ of the time
 - If not perfectly scalable, each core will take longer than $1/n$ of the single-core time.
- Test all combinations of parallelization (MPI, openMP, SIMD)
 - The balance between ranks and threads can have tremendous impacts on performance
 - More cores doesn't always mean better performance

Example of strong scaling



Gromacs 5.1.4 with modified non-bonded kernels

Example of performance rollover

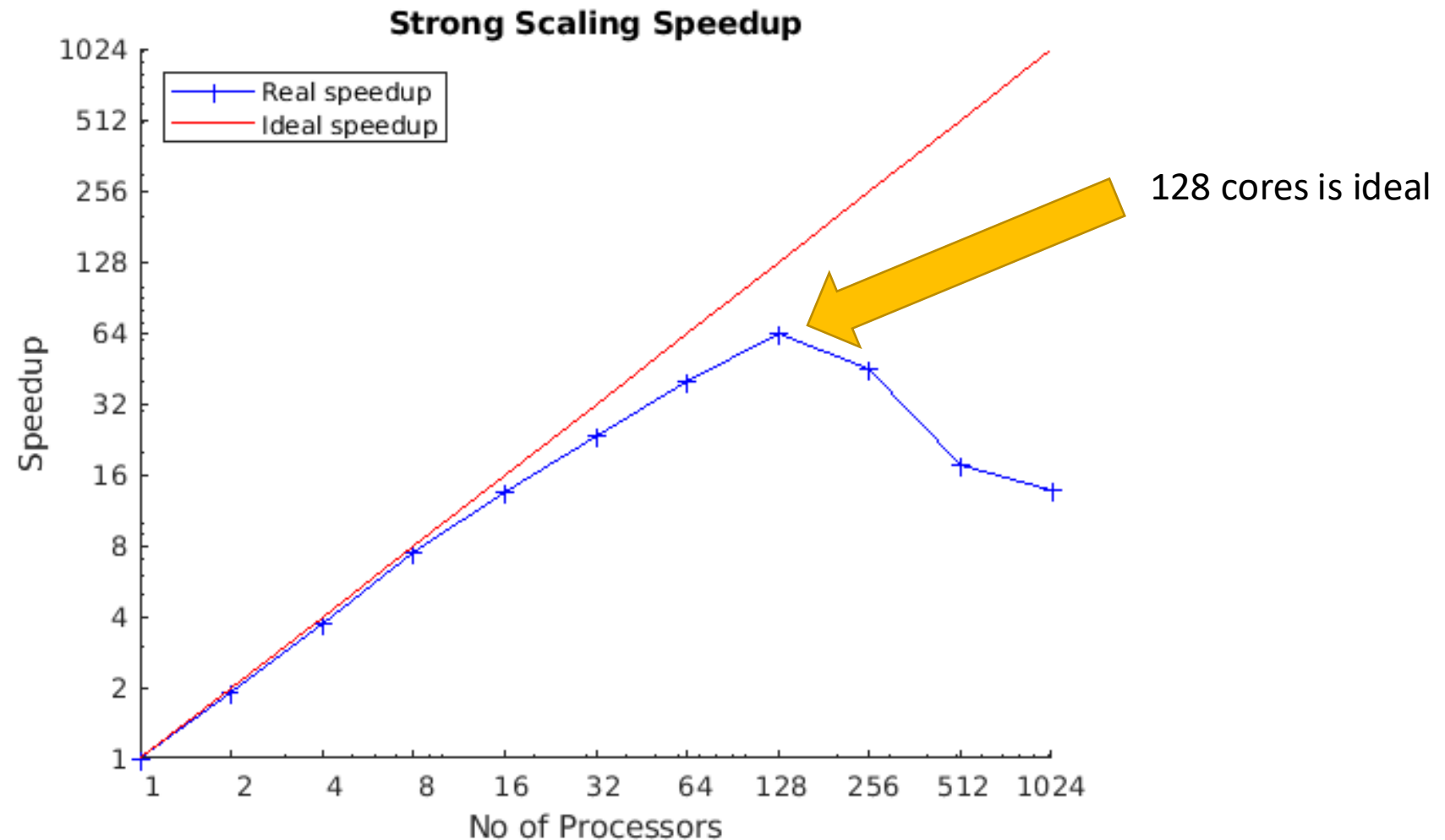


Image from https://hpc-wiki.info/hpc/Scaling_tutorial

Performance testing GPUs

- Very important to compare single-core performance with single-core+GPU performance
 - Make sure your “single-core” calculation is actually using a single core
 - Python libraries are often “greedy” by default (i.e. use all cores it can see)
 - If you don’t see a major speedup (100x, or more), perhaps the code is not very GPU-friendly
- Can test for multiple-GPU performance
 - No guarantee performance will be better
 - Often performance degrades
- GPU performance can depend heavily on the type of calculation, even with the same software (e.g. if PME or 6-12 interactions are offloaded to GPU)
- Different GPUs require different languages (e.g. CUDA, OpenCL)
 - Same package may use completely different routines for each type of GPU
 - Differences in performance may not be due to hardware
- **Not unusual for a GPU to give performance that is 1000x that of a single CPU core.... But, also not unusual for the speedup to only be 2x, or even be slower.**

Example of GPU-based performance stats

All calculations with identical CPUs and GPUs

Comparable performance on GPU, even though systems ranged in scale by 10x
 CPU performance correlated with # atoms

Same performance on GPU or CPU

~100x speedup. Nice.

1600x speedup!

TABLE 3 Performance benchmarks when using SMOG2-generated models in OpenSMOG

System	Resolution	Num. of particles	GPU performance ($\times 10^6$ steps/day)	CPU performance ($\times 10^6$ steps/day)
Standard SMOG models				
CI2	C_α	64	635 (540)	507
CI2	All-atom	521	506 (457)	62.0
EF-G ²⁸	C_α	686	530 (463)	69.5
EF-G	All-atom	5,301	403 (331)	5.7
Spike protein	All-atom	15,207	255 (251)	2.14
Ribosome ²⁹	All-atom	149,587	50 (67)	0.182
Non-standard (custom) contacts				
CI2	C_α	64	630 (540)	615
CI2	All-atom	521	550 (455)	61.5
EF-G	C_α	686	545 (464)	74
EF-G	All-atom	5,301	403 (332)	5.4
Ribosome	All-atom	149,587	50 (66)	0.175
Custom non-bonded interactions				
CI2 w/ions	All-atom	4,670	202 (191)	1.6
CI2 w/ions	All-atom	42,020	104 (116)	0.77
Ribosome w/ions	All-atom	186,929	5.2 (7.3)	0.00326