PRACE AUTUMN SCHOOL 2021

FUNDAMENTALS OF BIOMOLECULAR SIMULATIONS AND VIRTUAL DRUG DEVELOPMENT

September 20—24 2021

Organized by

The school is organised in partnership with AstraZeneca and NostrumBiodiscovery

Parallelisation Paradigms

Introduction to methodologies for parallelisation, application of MPI and OpenMP

Dr. Valentin Pavlov

September 20, 2021

Contents

- Introduction – the case for parallel; levels of parallelism;
- Discussion about each level;
- The million dollar question – does it scale?

Introduction

The case for parallel

- In many research fields performing a virtual experiment *in silico* is much cheaper and faster than performing a physical one *in vivo* or *in vitro*;
- Such experiments usually involve numerical solutions of some kind of differential equations, be it of a system of particles, a fluid, or some properties (e.g. heat exchange);
- Specifically, molecular dynamics (MD) simulations work by solving the equations of motion of a system of particles, in which the interactions are given by the chemical bonds and constraints between atoms, van Der Waals forces, Coulomb forces, etc.;
- Given the forces that act on each particle and its mass, using Newton's 2^{nd} Law, the acceleration is calculated and then integrated to get the new position of each particle;

The case for parallel

- New positions lead to new forces and the whole process is repeated in a next time step;
- A typical time step in a MD simulation is around 2 fs;
- The required simulation time depends on the experiment, but is usually in hundreds and thousands of ns range;
- A 100 ns experiment requires 50,000,000 steps of 2 fs each;
- The number of particles in the simulation also varies widely by experiment, but can easily be in the order of 100M or even billions of atoms;

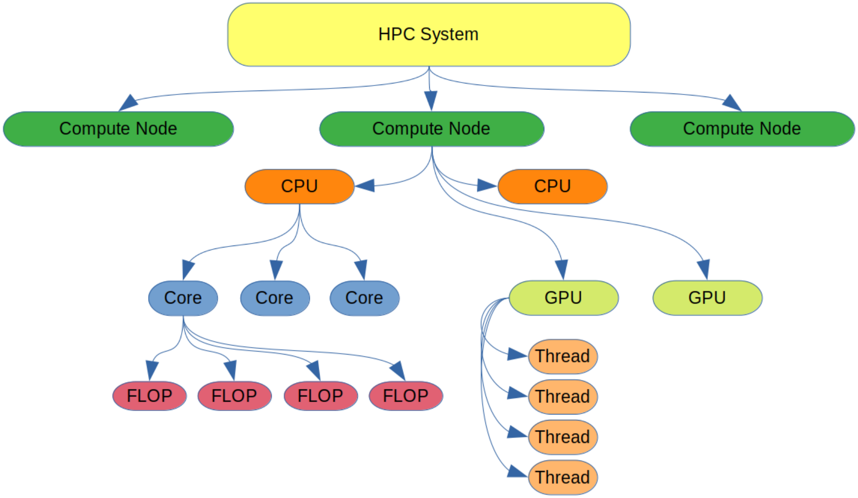
The case for parallel

- A simulation of 100M atoms over 50M time steps would take several months on a single contemporary computing core, obviously unacceptable;
- The solution is to *parallelise*: split the task among multiple computing cores, which work together towards achieving the common goal;
- Different algorithms require different ways to split the work;
- Different hardware architectures support different options;
- Thus there are different parallelisation paradigms to choose from;

Levels of parallelism

- Modern HPC systems support:
  - Instruction-level parallelism (superscalar pipelined cores, hyper-threading);
  - Superword-level parallelism (vectorization, SIMD);
  - Core-level parallelism (multiple cores in a single CPU socket);
  - Socket-level parallelism (multiple multi-core CPUs inside a compute node);
  - Node-level parallelism (multiple independent nodes with multiple multi-core CPUs in each of them);
- In addition some systems support accelerators (e.g. GPU cards):
  - A single GPU card can execute hundreds of threads simultaneously (SIMT);
  - A single compute node can have multiple GPU cards installed;
  - An HPC system can have multiple compute nodes with multiple GPU cards each;

Levels of parallelism



Levels of parallelism

- A further level of complexity comes with the memory hierarchy – not all storage is created equal;
  - HDD/SSD offers big persistent storage, but are slow;
  - DDR RAM offers relatively fast access, but is not persistent;
  - Multi-level cache – L3, L2, L1 – each of them faster, but smaller than the previous;
- A good parallel implementation must take into account all of the above and utilize all possible levels of parallelism while observing data locality (keeping the data as close as possible to the process that uses it);
- This is quite a complex problem, and not only for the programmers, but also for the end-users – while a package may support all of the above, it is usually not trivial to make a run as optimal as possible;

Parallelisation details

Instruction-level parallelism

- Happens inside a single core;
- Allows a single core to execute several instructions per cycle;
- Achieved by having instruction pipelines with several independent processing stages;
- Known commercially as Hyper-threading;
- Usually bad for HPC – hyper-threads share the same L1 cache, which leads to cache misses if utilized;
- Know your machine – how many *actual* cores per node it has;
- Rule of the thumb – **inside a node, do not parallelise more than the number of cores; do not count on hyper-threading**;

Superword-level parallelism

- Allows for the simultaneous execution of several identical operations on different inputs (e.g. 4 simultaneous additions);
- Known as *vectorization* or *SIMD* – Single Instruction Multiple Data;
- All modern CPUs cores support such instructions, usually some form of SSE, AVX, etc.
- Latest AVX-512 supports 16 simultaneous FLOP;
- From user perspective vectorization is *usually* hidden – there is nothing to do, it all happens automatically;
- Know your machine and your software – what SIMD instructions are available and if the package is compiled with the proper support for them;

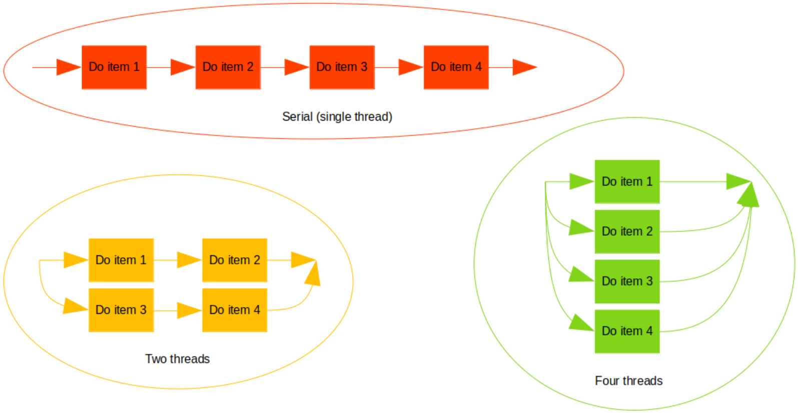
Core-level parallelism and Socket-level parallelism

- A compute node in an HPC system usually contains several CPUs (sockets), and each CPU contains several compute cores;
- Each core can idenepdently run a separate logical *thread* of execution;
- A single computing process can split itself into logical threads, each carrying out a portion of the computation;
- This is known as *multi-threading*;
- The logical threads have access to the same memory, hence it is a.k.a. *shared memory* model;
- Shared memory is good, since it means no data needs to be transferred;
- But still, it requires *synchronization* between threads so the result is deterministic;

Multi-threading

- There are many multi-threading parallelisation paradigms, the two most notable being:
  - fork-join model;
  - task-based parallelism;
- The main difference between them is that in the fork-join model each thread is supposed to execute the same sequence of steps (task), while in the task-based parallelism each thread can execute a completely different task;
- For example:
  - Mount the 4 wheels of the car at the same time – fork-join model;
  - Mount the back bumber while at the same time loading washer liquid for the wipers – task-based parallelism;

Fork-join paradigm



OpenMP, pthreads, TBB, etc.

- OpenMP is a set of libraries, tools and compiler support;
- It is the *de-facto* standard for multi-threading in HPC applications;
- pthreads is the linux native thread support;
- TBB (Thread Building Blocks) is a library mostly used for task-based parallelism (but not only);
- Many other alternatives;

Core/Socket configuration and Thread affinity

- The most beneficial configuration is for one core in a node to run one thread;
- However, there is a *nuance* – it is not the same if we have 1 socket with 12 cores or 2 sockets with 6 cores each;
- The difference is that cores inside a socket share its cache (the fastest memory);
- If cores inside a socket run threads that work with data chunks that are close to one another, we could benefit from cache hits;
- This helps with *data locality*, which is **extremely** important and means that the data is as close (fast to reach) as possible to the core that needs it;

Core/Socket configuration and Thread affinity

- Every multi-threading implementation has some way to control the distribution of threads between cores – so called *thread affinity*;
- Compare (for 2 sockets with 2 cores each):
- “close” thread affinity:
  - thread 0 goes to core 0, which is on socket 0;
  - thread 1 goes to core 1, which is on socket 0;
  - thread 2 goes to core 2, which is on socket 1;
  - thread 3 goes to core 3, which is on socket 1;
- “spread” thread affinity:
  - thread 0 goes to core 0, which is on socket 0;
  - thread 1 goes to core 2, which is on socket 1;
  - thread 2 goes to core 1, which is on socket 0;
  - thread 3 goes to core 3, which is on socket 1;

Some OpenMP environment variables

- OpenMP runtime configuration is controlled through OS environment variables;
- There are a lot of them, but the most important ones are:
    - `OMP_NUM_THREADS` – controls how many threads each process will see;
    - `OMP_PLACES` – “sockets”, “cores”, “master”. Most probably “cores” is what you want;
    - `OMP_PROC_BIND` – controls thread affinity when bound to cores. “close” or “spread”;

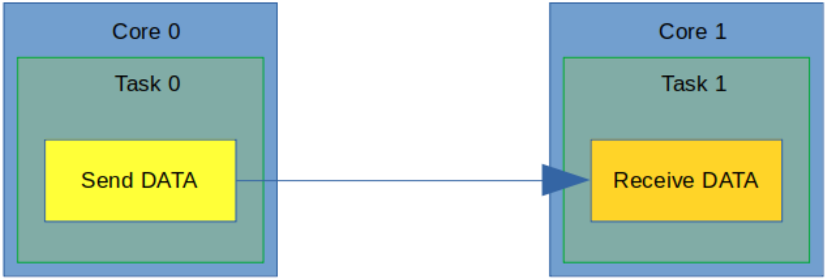
Important things for end-users re: multi-threading

- Know your hardware:
    - Know the total number of cores in a node;
    - Know the core/socket configuration;
- Know your software:
    - Does it support multi-threading of some form (usually: yes);
    - What exactly (OpenMP, pthreads, TBB, etc.);
    - How to enable it and how to control the number of threads;
    - How to control thread affinity and which setting is beneficial (trial and error unless specified in the package docs);

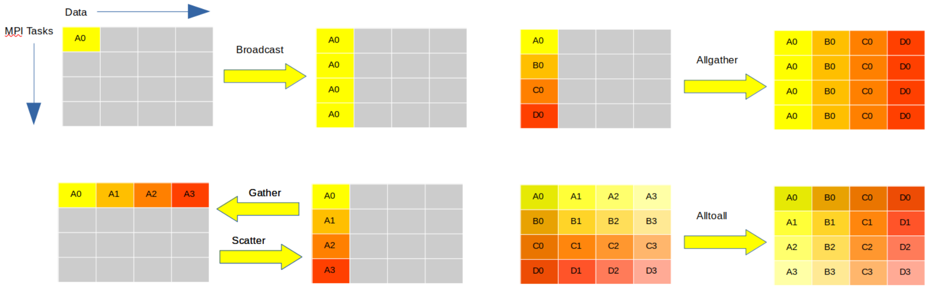
Node-level parallelism

- An HPC system is composed of a multitude of compute nodes, each of which is multi-socket, multi-core, each core SIMD-capable and superscalar;
- Usually the compute nodes have direct access only to their own RAM: *distributed-memory* model;
- Except in some special cases (*embarassingly parallel*), this requires *communication* between the compute nodes in order to transfer the data to the process that needs it, and this adds overhead;
- This communication can be *point-to-point* (one process sends data to one [other] process), or *collective* (one or more processes send data to one or more processes);
- There is a problem with collective communications: in the worst case, it grows as the *square* of the number of the participants;

Point-to-point (P2P) MPI



Collective MPI



Important things for end-users re: MPI

- MPI is available on all kinds of hardware, including laptops;
- All compute cores in a HPC system / cluster participate in a pool; MPI decides which copy of your process on which node/core to send;
- Each copy is identified by its *MPI rank*. The rank is usually written in the log files and may help trace faults during execution;
- Similar to thread affinity, the MPI system may have a way for you to specify process affinity – which rank goes to which core on which node;

Important things for end-users re: MPI

- A program is run (usually) by e.g. `mpirun -np 4 /bin/date`
- (But not on HPC system, at least not directly, where it happens through a job scheduler (e.g. slurm);
- The cores of a node can also be used to run MPI tasks (processes) instead of OpenMP threads of a single process (for example when a program does not support OpenMP);
- So, if we have 4 nodes with 16 cores each, we can:
    - `OMP_NUM_THREADS=16 mpirun -np 4 someprogram`
    - `OMP_NUM_THREADS=1 mpirun -np 64 someprogram`
- Usually, the first of these is more beneficial, since the 16 threads on each node share the memory of a single process and no data transfers are needed;

Important things for end-users re: MPI

- Know your hardware and OS:
  - Know the number of nodes available and the core/socket configuration of each node;
  - Know how to run MPI jobs, particularly the batch files of the scheduler on a HPC system;
  - Know how to query the scheduler for the status of your jobs;
- Know your software:
  - Does it support multi-threading of some form – if yes, it is usually more beneficial to make the number of ranks = number of nodes and number of threads = number of cores on each node; otherwise number of ranks = total number of cores;
  - Limitations of their software and its *scalability* – to not waste resources;

GPUs

- A compute node may have one or more accelerators, usually GPU card(s), which are exceptionally fast for certain computations and are utilized in many implementations;
- They utilize SIMT – Single Instruction Multiple Threads – similar to SIMD but with hundreds ot identical threads working on different data;
- Apart from the synchronization overhead, there is also overhead from the need to transfer the data from the CPU memory to the GPU memory and back;
- If multiple GPUs on different nodes are used, these memory transfers pile up on top of the transfers needed between the nodes;
- Some systems support Direct GPU-GPU transfer, which *might* be faster;

Strong and Weak Scaling

Serial and parallel regions

- An algorithm is a sequence of steps, much like a cooking recepie;
- Some of the steps are independent of one another and so can be executed in parallel;
- That is, step A is independent of Step B, if Step B cannot influence Step A's input (there is no data dependency);
- The portions of the algorithm that can be parallelized are called *parallel regions*, the rest are *serial regions*;
- The more time an algorithm spends in parallel regions, the better it is suited for HPC – it is more *scalable*;

Speedup

$$S = \frac{T_1}{T_N}$$

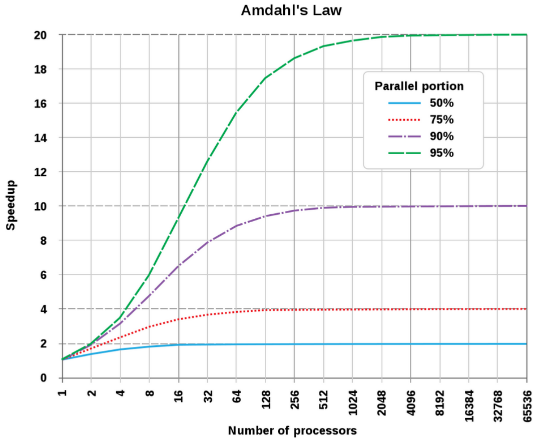
- T_1 is the time for running the algorithm on 1 processor; T_N is the time for running it on N processors;
- Ideally $S = N$, independent of N (linear scaling), but this is rarely achieved in practice;
- Although for small N sometimes $S >$N is achieved due e.g. to cache (superscalability);

Amdahl's Law

$$S = \frac{1}{(1-p) + \frac{p}{N}}$$

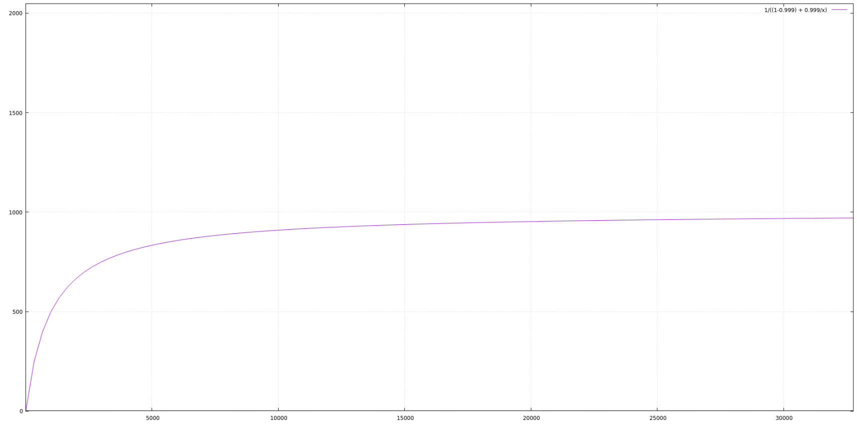
- p is the portion of the parallel regions of the algorithm (e.g. 70%);
- This is valid for *fixed workload*, while the number of processes N vary;
- The dependence of S from N under fixed workload is called *strong scaling*;

Amdahl's Law



Daniels220, CC BY-SA 3.0, `https://commons.wikimedia.org/wiki/File:AmdahlsLaw.svg`, via Wikimedia Commons

Amdahl's Law


1/((1-0.999) + 0.999/x)
2000
1500
1000
500
0
5000
10000
15000
20000
25000
30000

Amdahl's Law – observation

- When $N \to \infty$, $S \to \frac{1}{1-p}$
- Even if 99.9% of the algorithm is parallel, the speedup can not go over 1,000, regardless of the number of processors used;
- This looks very hopeless and discouraging;
- Note that the communication time counts towards the serial regions;
- That's why it is extremely important for the programmers to try and hide the communication time and overlap it with computation – otherwise there is no hope for strong scalability;

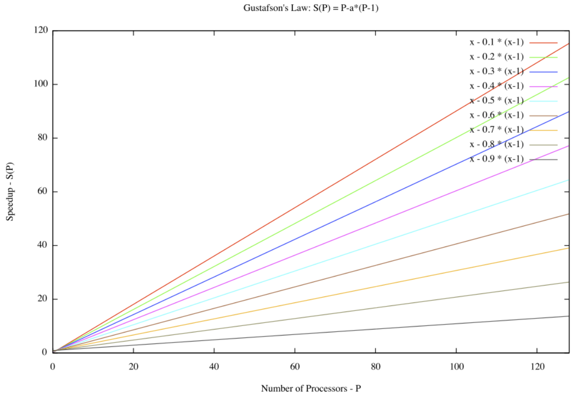
Gustafson's Law

- Amdahl's Law is valid when there is a fixed workload – e.g. it shows what will happen if you keep the problem the same, but throw more resources on it;
- There is another way to utilize more resources – use them to solve bigger problems;

$$S = (1 - p) + p \times N$$

- This is valid for *fixed time*, but the workload and number of processors N vary;
- The dependence of S from N under fixed time is called *weak scaling*;
- Shows how the solution time varies for a *fixed problem size per processor*;

Gustafson's Law



Does it scale?

- It's generally easier to achieve weak scaling;
- But if the software uses naked collective MPI (e.g. alltoall), weak scaling will degrage as well;
- Also, not all algorithms are created equal – e.g. an $O(N^3)$ algorithm will only gain 26% speedup upon doubling the resources;
- At some point it becomes worthless to throw resources at a given problem, and it is up to *you* to be able to judge that limit;
- For this, you need to perform scalability tests – e.g. run your simulation, *but with limited number of time steps*, and run it with different configuration (threads/mpi tasks), on increasing number of cores/nodes until you find out that it doesn't make sense to increase them anymore;
- Then, request that much resources for your final simulation;

Questions and Answers

Thank you for your attention!
Questions?