PHYSM0032 Advanced Computational Physics
Lecture 1

Dr. Simon Hanna

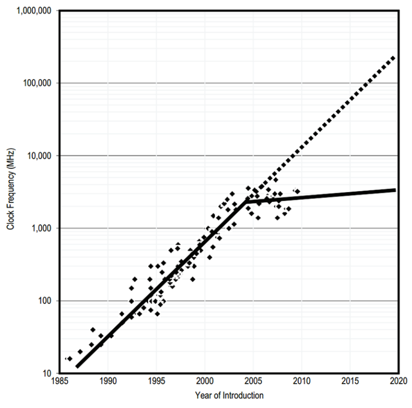
September 28, 2021

1 Introduction

1.1 Aims

- To introduce concepts of high performance computing for physicists, including parallel computing and GPU arrays.
- Course aimed at students who might go forward to research in computational physics and want to obtain maximum benefit from current generation of supercomputers.

1.2 Motivation



- Many physics problems are very slow to solve on a computer e.g. a molecular dynamics simulation of 10^6 atoms with a timestep of 10^{-15}s, requires 10^{21} "operations" to simulate 1 s of real time. On a modern processor, allowing 10^{-9}s per operation, this would take 32,000 years.
- In the past, up until 2005, processor speed (clock frequency) could be relied upon to increase yearly. This is no longer the case (see graph).
- Although transistors have continued to grow smaller (65 nm in 2005, now down to 14, 10, 7 or even 5 nm (Apple M1)) transistor operating speed has stopped improving due to the gate capacitance, which depends on layer thickness of insulator which has bottomed out at 0.9 nm. So, a single compute core can only get faster by:
  1. clever design techniques, including instruction level parallelisation;
  2. increasing the architecture from say 32 to 64 bits (or more), so more data is processed per clock cycle;
  3. increasing the operating voltage (which increases the power consumption drastically).
- This has led to the prevalence of multi-core designs since 2007; to get the most of them, we need to learn **parallel programming**.

1.3 General Description

- Course will consist of lectures and on-line tasks, supported by regular drop-in sessions (from week 2 onwards).
- Assessment will be by a mini-project.
- Topics include (in no particular order):
  1. Basic microprocessor architecture (as much as is needed to understand how to optimise programs)
  2. Overview of different types of language (Python, Cython, C/C++, Fortran etc)
  3. Parallel processing architecture: shared memory versus distributed systems
  4. Parallel programming paradigms: shared memory (OpenMP) versus distributed memory (MPI)
  5. Data structures: variables and objects, arrays, memory handling
  6. Parallel algorithms for linear algebra and n-body simulations
- The mini-project can be completed using either Python/-Cython, C/C++ or Fortran.

1.4 Learning Outcomes

After taking this unit, students should:

- have a thorough grasp of parallel computing architectures for applications in physics research, as well as parallel algorithms for linear algebra and techniques for performing physics simulations across multiple processors;
- be aware of the scalability of these techniques and the need to tune the size of the simulation to optimise its efficiency on a given computer system;
- be able to construct a working parallel program to solve a given physical problem.

1.5 Recommended Book

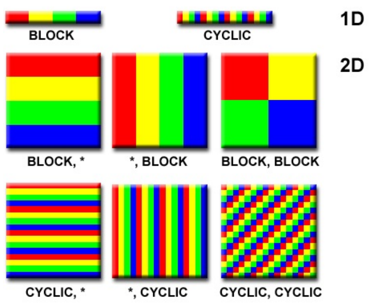
- "Introduction to High Performance Scientific Computing" by Victor Eijkhout (available from Amazon and free download from author).

2 Parallel Concepts

2.1 Problem space partitioning

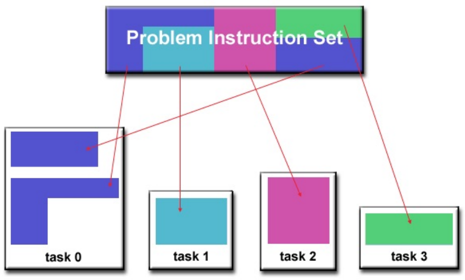
- First need to break problem into discrete "chunks" of work that can be distributed to multiple tasks: known as decomposition or partitioning.
- Two basic ways to partition between parallel tasks: domain decomposition and functional decomposition.

2.1.1 Domain Decomposition:

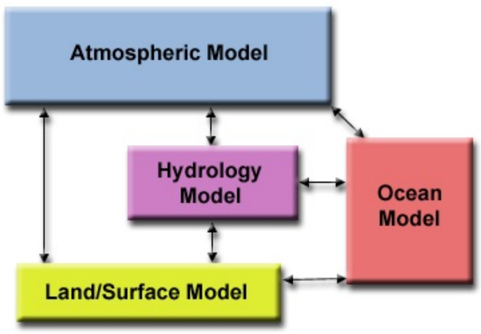


- Dataset associated with a problem is decomposed.
- Each parallel task works on a section of the data.
- There are different ways to partition data.
- Data can be partitioned in a physical domain e.g. spatial decomposition.

2.1.2 Functional Decomposition:



- Focus is on computation to be performed rather than the data.
- Problem is decomposed according to work that must be done.
- Each task performs a portion of the overall work.
- Method lends itself well to problems that can be split into different tasks.



- E.g.: Climate Modeling:
  - Each component of model could be separate task.
  - Arrows indicate data exchange between components: atmosphere model generates wind velocities needed by ocean model; ocean model generates sea surface temperatures used by atmosphere model, etc.
- Combining both types of decomposition is common and natural.

2.2 Basic parallel execution

Typically most physics problems will fit into this type of scheme:

- Spread operations over many processors
- If n operations take time t on 1 processor...
- Does this become t/p on p processors ($p \le n$)?
- e.g. consider a simple loop in C, Python or Fortran:

```python
for i in range(n):
    a[i] = b[i]+c[i]
```

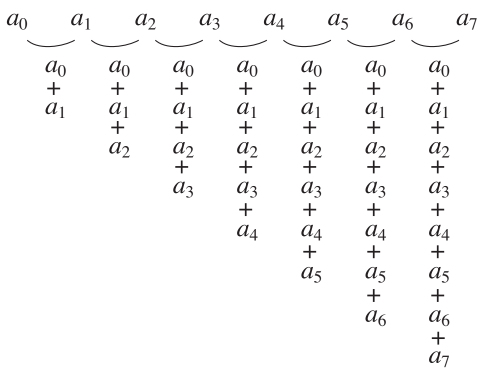
```c
for (i=0; i<n; i++){
    a[i] = b[i]+c[i];
    }
```

```fortran
DO I = 1, N
   A(I) = B(I)+C(I)
ENDDO
```

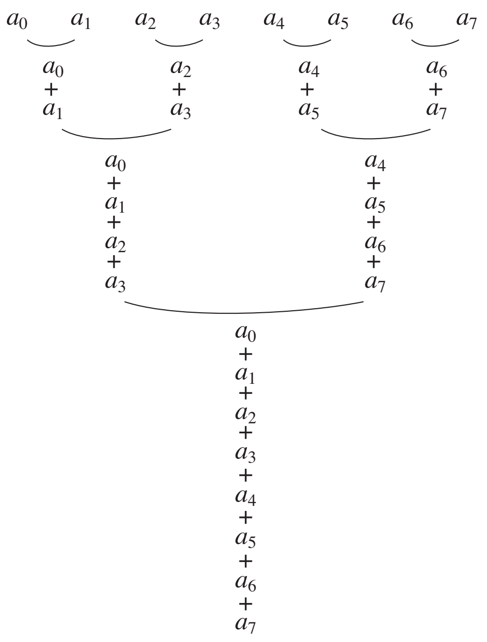
In the simplest case, each iteration of the loops could be given to a separate processor.

- Optimally, p processors give $T_p = T_1/p$
- Speedup $S_p = T_1/T_p$, is p at best
- Generally less than optimal: overhead, sequential parts, dependencies etc
- Efficiency $E_p = S_p/p$
- Perfect speedup in "embarrassingly parallel applications"
- Superlinear speedup not possible in theory, sometimes happens in practice (e.g. for memory limited problems).
- Scalability: efficiency generally decreases with increasing p.

Not all algorithms have ideal speed-up. e.g. consider summing of n numbers. In serial computation, this would be achieved in $n-1$ steps:



However, in parallel mode, n numbers can be summed using $n/2$ processors, in total time $\propto \log_2 n$:



i.e. with 8 numbers we expect a compute time of 7 units for the serial case but 3 units for the parallel case, a speed up of $7/3$. Clearly the benefits increase with increasing n.

2.3 Scaling

- Increasing the number of processors for a given problem makes sense up to a point. ($p > n/2$ in the addition example above has no use).
- **Strong scaling:** problem constant, number of processors increasing
- **Weak scaling:** More realistic—scaling up problem and processors simultaneously, for instance to keep data per processor constant.

2.4 Amdahl's law

- Some parts of a code are not parallelizable – they ultimately become a bottleneck.
- According to Amdahl's law, if $x\%$ of a program is sequential, the maximum speedup available will be $100/x$, no matter the number of processors, p.
- Formally, define fractions F_p and F_s of parallel and sequential code, such that $F_p + F_s = 1$. Then:

$$T_p = T_1 \left(F_s + \frac{F_p}{p} \right)$$

so T_p approaches $T_1 F_s$ as p increases.

2.5 Parallel computer architectures

- Flynn's (1966) taxonomy classifies parallel computers into four basic types:

SISD: Single instruction, single data:
- Most desktop machines until about 10 years ago

SIMD: Single instruction, multiple data:
- Old style Cray computers, and other vector machines and array processors
- Parts of modern GPUs
- Register level vectorisation (AVX et al.)

MISD: Multiple instruction, single data:
- Special purpose machines; rare

MIMD: Multiple instruction, multiple data
- Nearly all of today's parallel machines including modern desktops and laptops.

2.6 Memory models on supercomputers

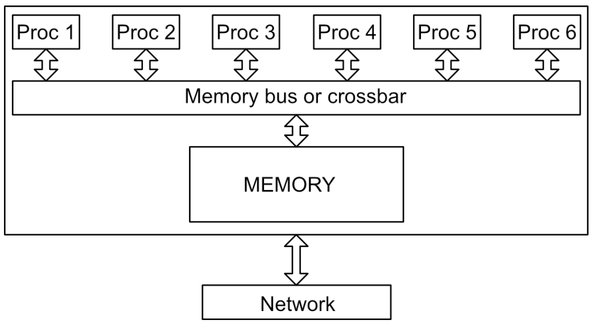
Shared memory:

- all processors share the same address space
- OpenMP (see later): directives-based programming

Distributed memory:

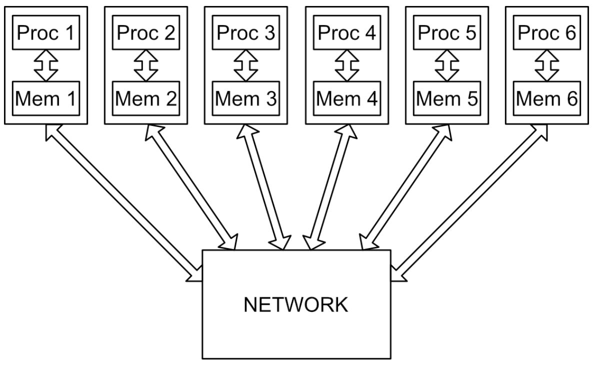
- every processor has its own address space
- MPI (see later): Message Passing Interface

2.6.1 Shared memory



- Single address space
- All processors have access to a pool of shared memory. (e.g., Single Cluster node (2-way, 4-way, ...). BlueCrystal Phase 4 has 28 cores per cluster node).
- Shared memory (or SMP: Symmetric Multi-Processor) is easy to program (OpenMP–see later) but hard to build
  - bus-based systems can become saturated
  - large, fast (high bandwidth, low latency) crossbars are expensive
  - cache-coherency is hard to maintain at scale

2.6.2 Distributed memory



- Each processor has its own local memory.
- Must do message passing to exchange data between processors. (examples: Linux Clusters, Cray XT3, BlueCrystal)
- Distributed memory is easy to build (bunch of PCs, ethernet) but hard to program (MPI–see later)
  - You have to spell it all out
  - interconnects have higher latency, so data is not immediately there
  - makes parallel algorithm development and programming harder

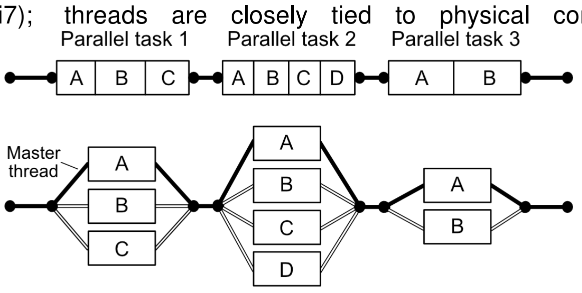
2.6.3 Blue Crystal 4

- A distributed memory system, consisting of a large number (500) of cluster nodes, each with a large number of cores (28).
- Each cluster node could be treated as a shared memory system.
- In practice each core and associated share of the memory is often treated as a "node" and MPI used throughout.

2.7 Parallel paradigms

2.7.1 Multi-threading

- Suited to shared-memory systems.
- Multiple "program threads" execute different parts of the problem.
- Typically one program thread executes on each CPU core e.g.: quad-core—use 4 threads (8 with hyperthreading e.g. Intel i7); threads are closely tied to physical cores



- Symmetrical multi-threading – e.g. OpenMP – program splits into multiple threads for the parallel sections (see figure)
- Transactional multi-threading – e.g. pthreads in C, thread library in C++ – a master thread creates worker threads and distributes work to them as required.

2.7.2 Message passing

- Suited to distributed memory systems.
- Separate program tasks, running on different CPU cores, often on different nodes of a cluster, all run copies of the same code, exchanging data as required via a high speed "network" (see later section).