# Introduction to High Performance Computing

Alejandro Cárdenas-Avendaño

# Serial Performance

# Serial Performance

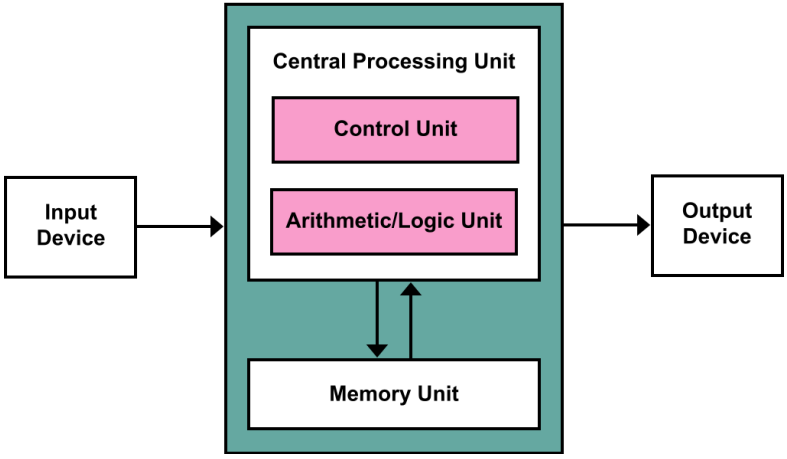
We have studied some simple tools that can tell us the **hot spots** of our (serial) application (where it spends most of its time or memory).

To be able to **improve the performance** of these hot spots, we can:

1. Try and optimize the serial program.
2. Try and parallelize the program, to use more cores.

Usually you should try both. The following classes will be on the second option.

# Von Neumann Architecture






- Central Processing Unit (CPU)
  - control unit (instruction decoder)
  - arithmetic and logic units (ALU)
- Memory System
- Input and output devices

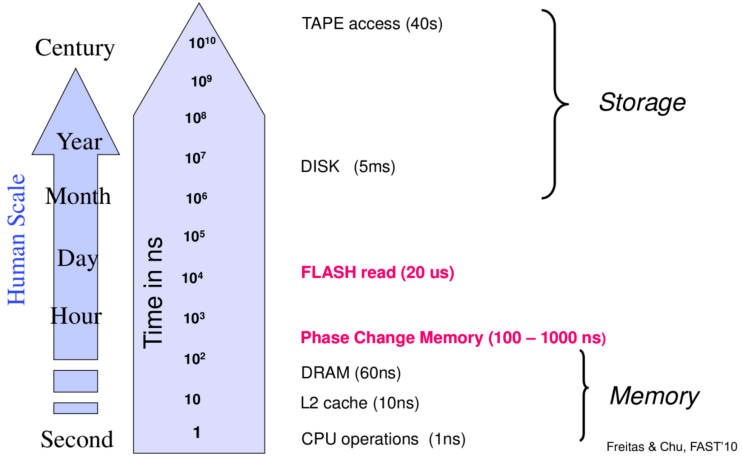
# Performance factors

Performance estimates based on instructions+memory picture cannot be always trusted. We'll discuss the following **architectural details** that make performance deviate from what you might expect:

- Memory hierarchy
- Registers
- Cache
- File IO
- NUMA  (Non Uniform Memory Access)
- Vectorization Multicore CPUs
- Networked Clusters

# Memory/storage hierarchy

| *Storage type* | *Latency* | *Bandwidth* |
|---|---|---|
| CPU registers | . | . |
| CPU cache | 5-10 ns | 50 GB/s |
| RAM | 60 ns | 10 GB/s |
| Flash(SSD) | 0.1 ms | 250 MB/s |
| Hard drives | 5 ms | 100 MB/s |
| Tape | 1 min | 80 MB/s |




**Latency:** Wait time before getting the first data.

**Bandwidth:** Access speed once data starts flowing.

# CPU Registers

- At the top of the hierarchy are the registers: very fast pieces of memory, close to, or part of, the CPU.
- Typically, when an instruction uses registers, access is as fast as execution.
- Now, although that means it makes little sense to assign an access speed to registers, for the purpose of comparison with the other storage types, let's say that, for a CPU with a clock speed of 4 GHz, the access time ("*latency*") is 1/4GHz = **0.25 ns**.
- Number of registers depends on the CPU architecture. Typically numbers of registers are 16, 32, **48**.
- Any data on the other storage types must first be loaded in register.
- It's the **compiler's job** to use the small number of registers optimally, and to insert the appropriate loads and stores.

# Cache memory

- This relatively small bit of memory is not tied to the instruction set, but is ‘baked into’ the CPU.
- Cache size is at most **several MB**.
- Cache is an intermediate memory for data that is currently used by the CPU.
- Once data is in cache, access times range from **a couple of ns to dozens of ns**. (There are usually several levels of cache, adding to the memory hierarchy.)
- Data will be automatically copied in and out of the larger data pool (RAM, or main memory).
- You do not control the copying in and out directly.

  (Except for prefetching CPU instructions, best left to the compiler.)

# How does cache work (1/2)?

We have the following situation:

- Large, slow main memory storing our data.
- Small, high-speed cache as working memory (~MB).
- For small temporary data, the fast cache can be used.

What about larger data structures that need to be in main memory?

- When a part of such data is needed in an instruction, it has to be copied from main memory to cache.
- If every memory access required a copy from RAM to cache, we'd be better off not using cache.
- One-by-one copy of 64 bytes: $t = 64 \times (\text{bandwidth} + \text{latency})$

  Single copy of 64 bytes: $t = 64 \times \text{bandwidth} + 1 \times \text{latency})$ ← **faster**
- A **cache line** is a number (often 64) of contiguous bytes in memory.
- Cache is used by copying a cacheline at time when data is needed.

# How does cache work (2/2)?

- Cache copies a cacheline at time when data is needed that is not yet in cache (a cache miss).
- The memory address of the first element of a cacheline is a multiple of the number of bytes in a cacheline (they are **aligned**).
- When cache is full, it releases the oldest cacheline (typically).
- Because most instructions do not operate on 64 contiguous bytes, we should make sure that our subsequent operations accessing memory contiguously.
- i.e, in an array *double* a[100], requesting element a[0] also loads a[1] through a[7]. If the elements are accessed in order, we just got 8 transfers from main memory for the price of one.
- This assumes the array was cache-aligned. If not, when a[0] is requested, the cache system will fetch a set of bytes that includes a[0], but starts at a cache line boundary.
- If we can reuse data in cache, while it is still there, we get free data!

# Effects on performance

Performance will suffer if:

- Data is not close to the CPU when needed.
- Data that is used more than once by the CPU should not be transferred more than once (or as little as possible).
- Access of data involves separate transaction (incurring latency multiple times)

Let's look specifically the effects of two aspects:

- cache behavior
- file IO.

# Von Neumann Architecture - example

```
const int n=1e6; // reserve memory and store value
int a[n]; // reserve memory
int b[n]; // reserve some more memory
int c; // reserve even more memory
for (int i=0;i<n;i++) a[i] = 123; // store values
for (int j=0;j<n;j++) b[j] = 929; // store values
c = a[9]+b[4];   // read those values
                 // then add and store
```

- *The interpretation in the comments is **how we imagine** the computer is executing this code.*
- This program is, in some form, stored as instructions in memory.
- The Control Unit reads the instructions and tells the other units what to do.
- **But this picture does not reflect what the computer actually does!**

# Von Neumann Architecture - example

```
const int n=1e6; // reserve memory and store value
int a[n]; // reserve memory
int b[n]; // reserve some more memory
int c; // reserve even more memory
for (int i=0;i<n;i++) a[i] = 123; // store values
for (int j=0;j<n;j++) b[j] = 929; // store values
c = a[9]+b[4];   // read those values
                 // then add and store
```

- ***const int n=1e6;***
  - *An int is small and would fit in cache. However, this statement is unlikely to reserve memory. The compiler may substitutes the value, or use a register.*
- ***int a[n]; int b[n];***
  - *If we assume the cache size is 1MB, this memory must live in main memory.*
- ***int c;***
  - *Small enough to be in cache, but likely in register.*

# Von Neumann Architecture - example

```
const int n=1e6; // reserve memory and store value
int a[n]; // reserve memory
int b[n]; // reserve some more memory
int c; // reserve even more memory
for (int i=0;i<n;i++) a[i] = 123; // store values
for (int j=0;j<n;j++) b[j] = 929; // store values
c = a[9]+b[4];   // read those values
                 // then add and store
```

- ***for (int i=0;i<n;i++) a[i] = 123;***
  - *At the first iteration (i=0), a[0]..a[15] are loaded (if sizeof(int)=4).*
  - *At the second iteration, the memory system notices that a[1] is already in cache.*
  - *At i=16, next cacheline is loaded.*
  - *When cache is depleted, the space occupied by the oldest cachelines get repurposed.*
- ***c = a[9]+b[4]***
  - *Since a[9] and b[4] are likely out-of-cache by now, a[0..15] and b[0..15] are loaded.*

# Cache-aware coding example

From cacheV1.cpp

```
// Function 1
int fun1(long int arr[n][n])
{
    int sum = 0;
    for (int i=0; i<n; i++)
      for (int j=0; j<n; j++)
          sum += arr[i][j];
    return sum;
}
```

**Time: 2118**

```
// Function 2
int fun2(long int arr[n][n])
{
    int sum = 0;
    for (int j=0; j<n; j++)
      for (int i=0; i<n; i++)
          sum += arr[i][j];
    return sum;
}
```

**Time: 3860**

# Vectorization

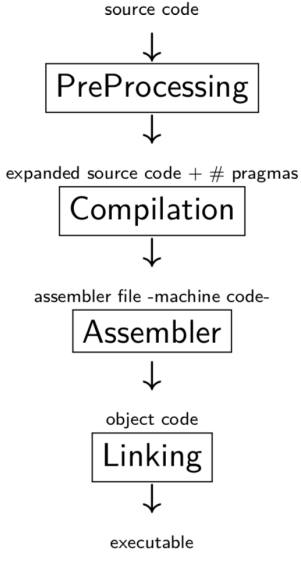
- Early computers generally had one logic unit that sequentially executed one instruction on one operand pair at a time.
- Computer programs and programming languages were accordingly designed to execute sequentially.
- Modern computers can do many things at once.
- Can be seen as a special case of automatic parallelization
- A set of instructions in a program are converted from a scalar implementation, which processes a single pair of operands at a time, to a vector implementation, which processes one operation on multiple pairs of operands at once.
- **SIMD/SPMD**: Single **Instruction/Program** Multiple Data
- Modern compilers will attempt to vectorize your code whenever is possible to do so
- For such, it is crucial that the sequential parts are transformed into equivalent parallel ones
- i.e. producing code that will utilize a ***vector processor*** generating the same results.

# Automatic Vectorization Techniques

- Compilers will modify and reshuffle your code, in order to optimize it.
- Automatic vectorization, like any loop optimization or other compile-time optimization, must exactly preserve program behavior.
- Data Dependency

  All dependencies must be respected during execution to prevent incorrect results.
- Data Precision

  Integer precision must be kept during vector instruction execution.



# Automatic Vectorization Techniques I

- To vectorize a program, the compiler's optimizer must first understand the dependencies
- between statements and re-align them, if necessary.
- Once the dependencies are mapped, the optimizer must properly arrange the implementing instructions changing appropriate candidates to vector instructions, which operate on multiple data items.
  - Build the **Dependency Graph**

    identifying which statements depend on which other statements. This involves examining each statement and identifying every data item that the statement accesses, mapping array access modifiers to functions and checking every access' dependency to all others in all statements.
  - The dependency graph contains all local dependencies with distance not greater than the vector size.
  - So, if the vector register is 128 bits, and the array type is 32 bits, the vector size is 128/32 = 4.
  - All other non-cyclic dependencies should not invalidate vectorization, since there won't be any concurrent access in the same vector instruction.

# Automatic Vectorization Techniques II

- **Clustering**

  Using the graph, the optimizer can then cluster the **strongly connected components** (SCC) and separate vectorizable statements from the rest.

- **Idioms**

  Some non-obvious dependencies can be further optimized based on specific idioms.

- a[i] = a[i] + a[i+1];, these self-data-dependencies can be vectorized because the value of the RHS values are fetched and then stored on the LHS value, so there is no way the data will change within the assignment.
- Self-dependence by scalars can be vectorized by variable elimination.

# Loop Vectorization I

- Loop vectorization converts procedural loops that iterate over multiple pairs of data items and assigns a separate processing unit to each pair.
- In many cases, programs spend most of their execution times within such loops.
- Vectorizing loops can lead to significant performance gains without programmer intervention, especially on large data sets.

```
// simple for-loop -- easy to vectorize: c = a + b
for (i=0; i<N; i++) c[i] = a[i] + b[i];
```

```
// what about this loop?
for (i=1; i<N; i++){
a[i] = b[i] + 1;
c[i] = a[i] + 2; }
```

- Vectorization can sometimes instead slow execution because of pipeline synchronization, data movement timing and other issues.

# Loop Vectorization II

- Another caveat to consider is that Loop-Vectorization is not always a legal and a profitable transformation.
- Compiler needs:
  - Compute the dependences
  - The compiler figures out dependences by
    - Solving a system of (**integer**) equations (with **constraints**)
    - Demonstrating that there is no solution to the system of equations
  - Remove cycles in the dependence graph
  - Determine data alignment
  - Vectorization is profitable

# Loop Vectorization III

General Framework for loop-vectorization:

- **Prelude:** Where the loop-independent variables are prepared to be used inside the loop. This normally involves moving them to vector registers with specific patterns that will be used in vector instructions. This is also the place to insert the run-time dependence check. If the check decides vectorization is not possible, branch to Cleanup.
- **Loop(s):** All vectorized (or not) loops, separated by SCCs clusters in order of appearance in the original code.
- **Postlude:** Return all loop-independent variables, inductions and reductions.
- **Cleanup:** Implement plain (non-vectorized) loops for iterations at the end of a loop that are not a multiple of the vector size or for when run-time checks prohibit vector processing.

# Loop-Vectorization Techniques

- Loop-vectorization transforms a series of instructions so that the same operation is performed at the same time on several of the elements of the vectors
- When vectorizing a loop with several statements the compiler needs to strip-mine the loop and then apply loop distribution
- **Stripmining** is a simple transformation employed by the compiler to help with data locality.

```
// for-loop
for (i=1; i<N; i++){ ...
}
```

```
// what about this loop?
for (i=1; i<N; i++){
a[i] = b[i] + 1;
c[i] = a[i] + 2; }
```

```
// StripMine
/* N is a multiple of strip_size */
for (k=1; k<N; k+=strip_size){
    for (i=k; i<k+strip_size -1; i++){ ...
     }
}
```

```
// stripmining ...
for (k=1; k<N; k+=strip_size){
/* strip_size could be size of vector register */
    for (i=k; i < k+strip_size; i++){
         a[i] = b[i] + 1;
         c[i] = a[i] + 2; }
}
```

# Loop-Vectorization Techniques

- Compiler Directives
- Loop Distribution or loop fission

  Reordering Statements
- Node Splitting
- Scalar expansion
- Loop Peeling
- Loop Fusion
- Loop Unrolling
- Loop Interchanging

**Conditionals:**

- Loops with conditional statemetns,

  need to add a #pragma instruction:

  ***#pragma vector always***
- Since the compiler does not know if

  vectorization will be profitable
- The conditional may prevent from an

  exception

```
// loop w/conditional
#pragma vector always
for (int i = 0; i < N; i++){
    if (c[i] < (float) 0.0)
        a[i] = a[i] * b[i] + d[i];
}
```

# Loop-Vectorization: Compiler Directives

- The compiler will try and actually vectorize many loops, but many more can be vectorized if the appropriate directives are used...

| Compiler Hints (for Intel ICC) | Semantics |
|---|---|
| `#pragma ivdep` | Ignore assume data dependences |
| `#pragma vector always` | override efficiency heuristics |
| `#pragma novector` | disable vectorization |
| `__restrict__` | assert exclusive access through pointer |
| `__attribute__ ((aligned(int-val)))` | request memory alignment |
| `memalign(int-val,size);` | malloc aligned memory |
| `__assume_aligned(exp, int-val)` | assert alignment property |

# General Compiler Optimization Flags I

| Flag | Description |
|---|---|
| `-O0` | Turn off optimization, eg. used when debugging. |
| `-O` or `O1` | Optimized compile. |
| `-O2` | More extensive optimization. This is recommended flag for most codes. |
| `-O3` | More aggressive than `-O2`, with longer compile times. Recommended for codes that loops involving intensive floating point calculations. |
| `-Ofast` | `-O3` plus some extras. The GNU documentation notes that this option results in a disregard of "strict standards compliance." |
| `-flto` | (GNU only) Link-time optimization, a step that examines function calls between files when the program is linked. This flag must be used to compile and when linking. Compile times are very long with this flag, however depending on the application there may be appreciable performance improvements when combined with the `-O`$x$ flags. This flag and any optimization flags must be passed to the linker, and gcc/g++/gfortran should be called for linking instead of calling ld directly. |

# General Compiler Optimization Flags II

**Single Instruction Multiple Data  SIMD Instructions...**

| Flag | Description |
|---|---|
| `-mtune=processor` | This flag does additional tuning for specific processor types, however it does not generate extra SIMD instructions so there are no architecture compatibility issues. The tuning will involve optimizations for processor cache sizes, preferred ordering of instructions, and so on. |
| `-mcpu` | see details at `http://gcc.gnu.org/onlinedocs/gcc/x86-Options.html` |
| `-march=native` | Take advantage of the hardware details (CPU) where the code is being compiled. Do not use if the code is going to be run in a different machine. |
| `-xHost` | Equivalent flag to `-march=native` for Intel compilers. |

# General Compiler Optimization Flags III

| option | optimization level | execution time | code size | memory usage | compile time |
|---|---|---|---|---|---|
| `-O0` | optimization for compilation time (default) | + | + | - | - |
| `-O1/-O` | optimization for code size and execution time | - | - | + | + |
| `-O2` | optimization more for code size and execution time | – | | + | ++ |
| `-O3` | optimization more for code size and execution time | — | | + | +++ |
| `-Os` | optimization for code size | | – | | ++ |
| `-Ofast` | O3 with fast none accurate math calculations | — | | + | +++ |

+ increase | ++ increase more | +++ increase even more || - reduce | – reduce more | — reduce even more

# General Compiler Optimization Flags IV

**And there are many more flags one could investigate...**

-mcmodel, -no-prec-div, fexcess-precision=style, -ffast-math, -fno-rounding-math, -fno-signaling-nans, -fcx-limited-range, -fno-math-errno, -funsafe-math-optimizations, -fassociative-math, -freciprocal-math, -ffinite-math-only, -fno-signed-zeros, -fno-trapping-math, -frounding-math, -fsingle-precision-constant, -fcx-fortran-rules, ...

References:

- https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc/
- https://wiki.gentoo.org/wiki/GCC_optimization

# Summary

- **Memory performance**
  - Data Locality
  - Data Reuse
  - Spatial Locality
- **I/O Performance**
  - Binary data
  - Large files
  - Spatial Locality
- **Optimization**
  - Many things are done automatically by the compiler (eg. **vectorization**)
  - Several flags are available to further exploit and assist optimization (eg. **-Ox**, **-march=native**, …)

*Overall, getting the best performance in your code involves, understanding very well how your code works, but also being familiar with the hardware and how the compiler does certain things...*

# Intro to Parallel Programming

# Concurrency

Following our initial considerations the next step is parallel computing in **n** cores:

- All these cores need something to do.
- We need to find parts of the program that can done independently, and therefore on different cores concurrently.
- We would like there to be many such parts.
- Ideally, the order of execution should not matter either.
- However, data dependencies limit concurrency.

# Supercomputer architectures

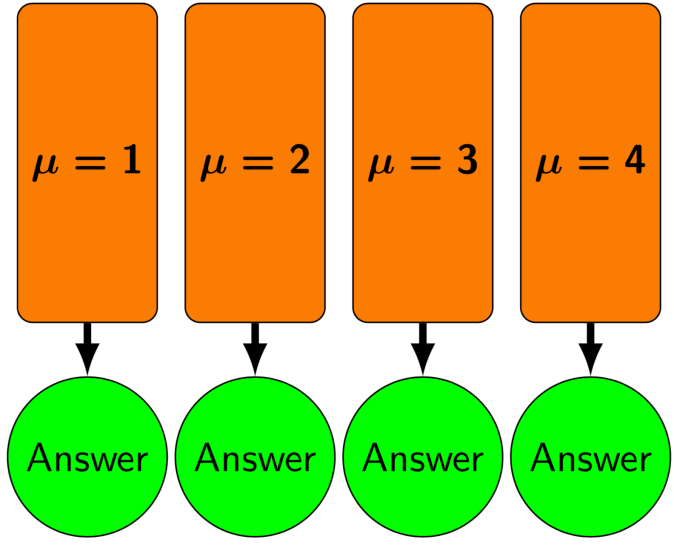
Supercomputer architectures come in a number of different types:

- **Clusters**, or distributed-memory machines, are in essence a bunch of desktops linked together by a network (”interconnect”). Easy and cheap.
- **Multi-core machines**, or shared-memory machines, are a collection of processors that can see and use the same memory. Limited number of cores, and much more expensive when the machine is large.
- **Accelerator machines**, are machines which contain an ”off-host” accelerator, such as a GPGPU or Xeon Phi, that is used for computation. Quite fast, but complicated to program.
- **Vector machines** were the early supercomputers. Very expensive, especially at scale. These days most chips have some low-level vectorization, but you rarely need to worry about it.

Most supercomputers are a hybrid combo of these different architectures.

# Parameter study: best case scenario

- Suppose the aim is to get results from a model as a parameter varies.
- We can run the serial program on each processor at the same time.
- Thus we get 'more' done.



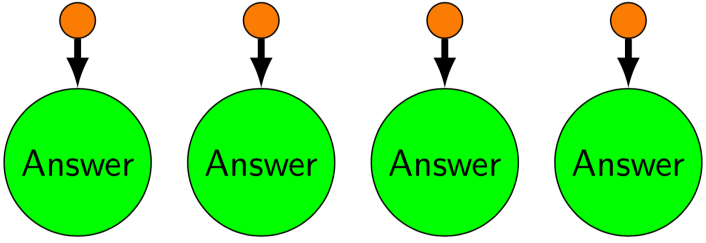
# Throughput

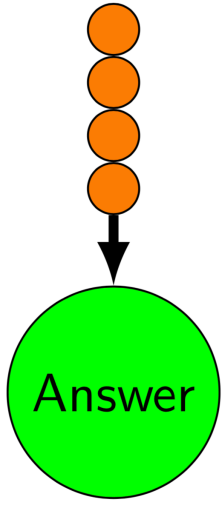
- *How many tasks can you do per unit time?*

$$\text{throughput} = H = \frac{N}{T}$$

**N** is the number of tasks, **T** is the total time.

- Maximizing **H** means that you can do as much as possible.
- Independent tasks: using **P** processors increases **H** by a factor of **P**




$$T = NT_1$$

$$H = 1/T_1$$

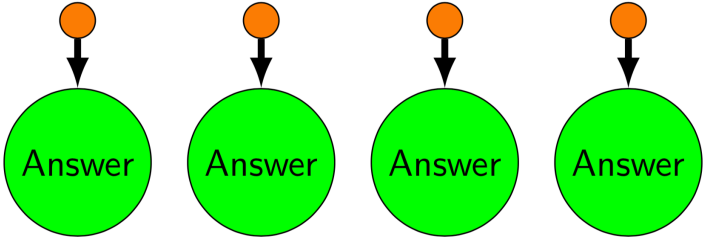
# Throughput

- *How many tasks can you do per unit time?*

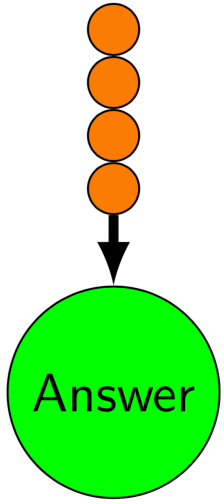
$$\text{throughput} = H = \frac{N}{T}$$

**N** is the number of tasks, **T** is the total time.

- Maximizing **H** means that you can do as much as possible.
- Independent tasks: using **P** processors increases **H** by a factor of **P**




$T = NT1/P \qquad H = P/T1$



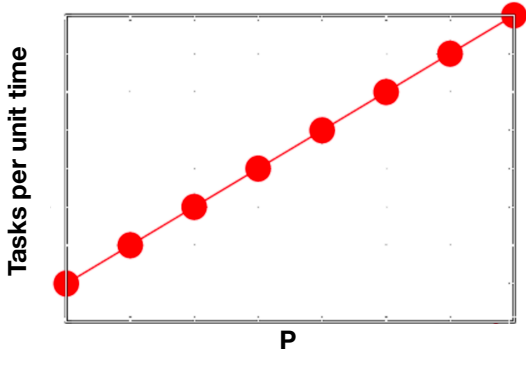

$$T = NT_1$$
$$H = 1/T_1$$

# Scaling — Throughput

- How a given problem's throughput scales as processor number increases is called "strong scaling".
- In the previous case, linear scaling:

$$H \propto P$$

- This is perfect scaling. These are called "embarrassingly parallel" jobs.

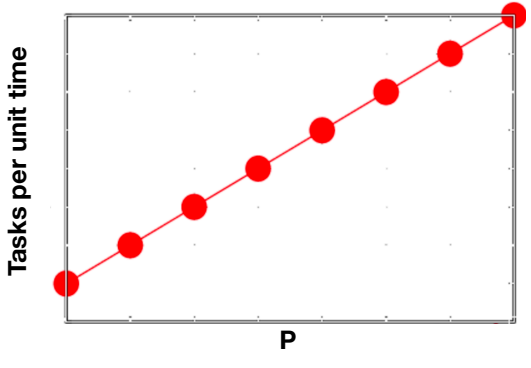


# Scaling — Speedup

- Speedup: how much faster the problem is solved as processor number increases.
- This is measured by the serial time divided by the parallel time

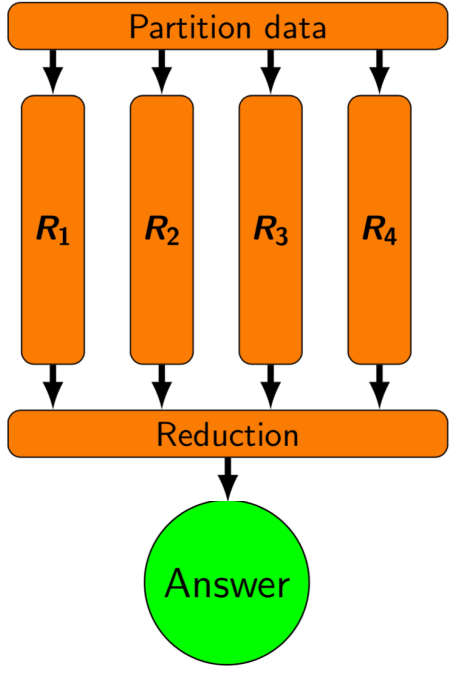
$$S = \frac{T_{\text{serial}}}{T(P)}$$

- For embarrassingly parallel applications, **S** $\propto$ **P**: linear speed up.



# Non-ideal cases

- Say we want to integrate some tabulated experimental data.
- Integration can be split up, so different regions are summed by each processor.
- Non-ideal:
  - We first need to get data to each processor.
  - At the end we need to bring together all the sums: '**reduction**'.



# Non-ideal cases

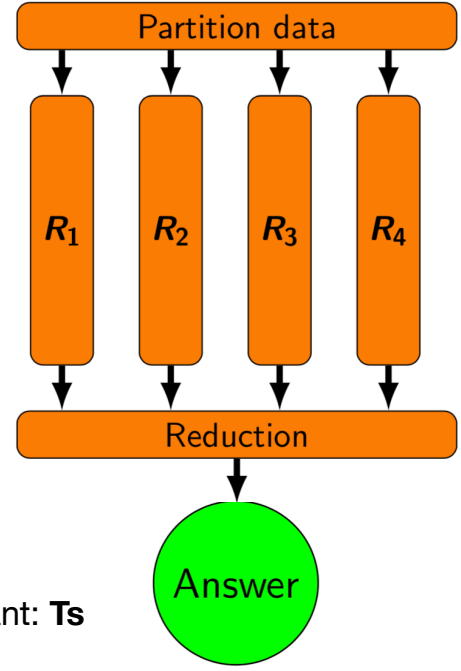
Parallel Overhead ->

Parallel Region ->

Perfectly parallel

(for large N)

Serial Portion ->




- Suppose non-parallel part is constant: **Ts**

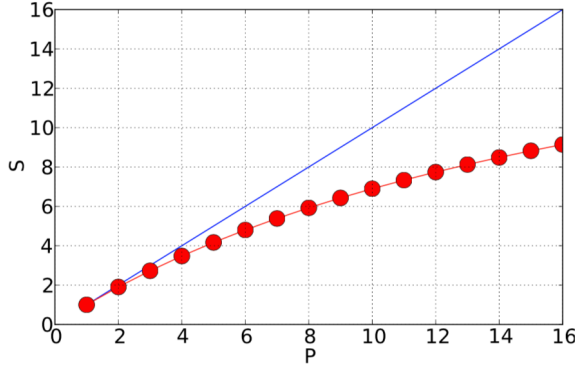
# Amdahl's law

- **Speed-up** (without parallel overhead):

$$S = \frac{T_{\text{serial}}}{T(P)} = \frac{NT_1 + T_s}{\frac{NT_1}{P} + T_s}$$

or, calling **f = Ts /(Ts + NT1 )** the serial fraction,

$$S = \frac{1}{f + (1-f)/P} \xrightarrow{P\to\infty} \frac{1}{f}$$




- The serial part dominates asymptotically. The speed-up is limited, no matter what size of **P**. **f = 5%** above.

# Amdahl's law, example

An example of Amdahl's law:

- Suppose your code consists of a portion which is serial, and a portion that can be parallelized. Suppose further that, when run on a single processor,
  - the serial portion takes one hour to run.
  - the parallel porition takes nineteen hours to run.
- Even if you throw an infinite number of processors at the parallel part of the problem, the code will never run faster than 1 hour, since that is the amount of time the serial part needs to complete.

The goal is to structure your program to minimize the serial portions of the code.

# Scaling efficiency

Speed-up compared to ideal factor **P**:

$$\text{Efficiency} = \frac{S}{P}$$

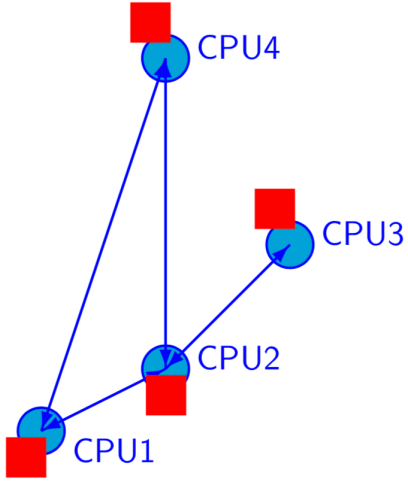
This will invariably fall off for larger **P**, except for embarrassingly parallel problems.

$$\text{Efficiency} \sim \frac{1}{fP} \overset{P \to \infty}{\longrightarrow} 0$$

- You cannot get 100% efficiency in any non-trivial problem.
- All you can aim for here is to make the efficiency as high as possible.

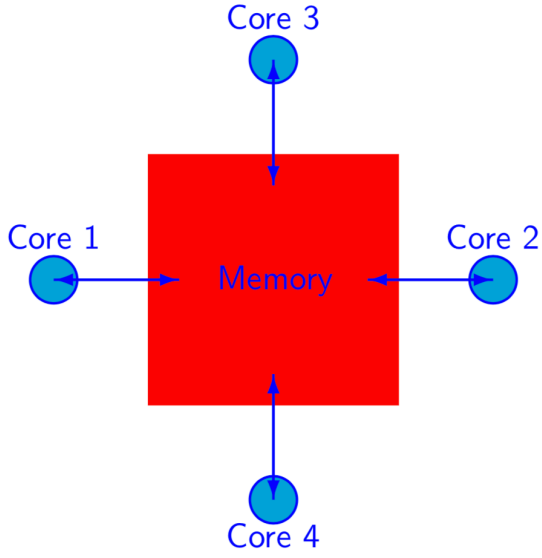
# Distributed Memory: Clusters

- Each Processor is independent! Programs run on separate processors, communicating with each other when necessary.
- Each processor has its own memory! Whenever it needs data from another processor, that processor needs to send it.
- All communication must be **hand-coded**: harder to program.
- **MPI** programming is used in this scenario.



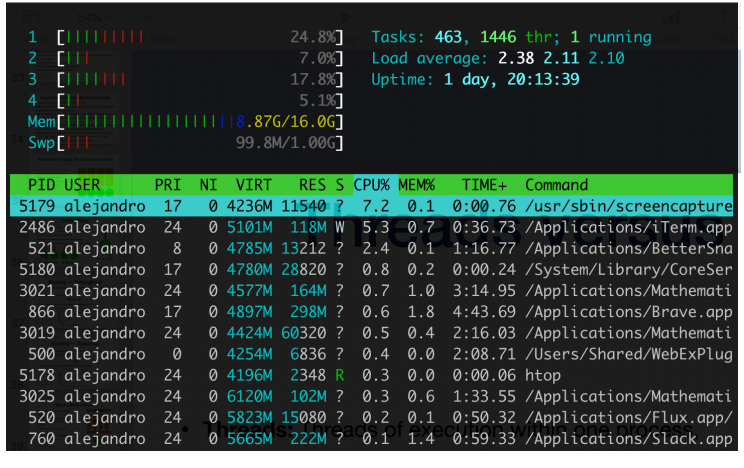
# Distributed Memory: Clusters

- Different processors acting on one large bank of memory. All processors ‘see’ the same data.
- All coordination/communication is done through memory.
- Each core is assigned a thread of execution of a single program that acts on the data.
- Your desktop uses this architecture, if it’s multi-core.
- Can also use hyper-threading: assigning more than one thread to a given core.
- **OpenMP** is used in this scenario.



# Threads versus Processes

- **Threads:** Threads of execution within one process, with access to the same memory etc.
- **Processes:** Independent tasks with their own memory and resources



# Share memory communication cost

| | Latency | Bandwidth |
|---|---|---|
| Gigabit Ethernet | $10\mu$s (10,000 ns) | 1 Gb/s (60 ns/double) |
| Infiniband | $2\mu$s (2,000 ns) | 2-10 Gb/s (10 ns/double) |
| NUMA (shared memory) | $0.1\mu$s (100 ns) | 10-20 Gb/s (4 ns/double) |

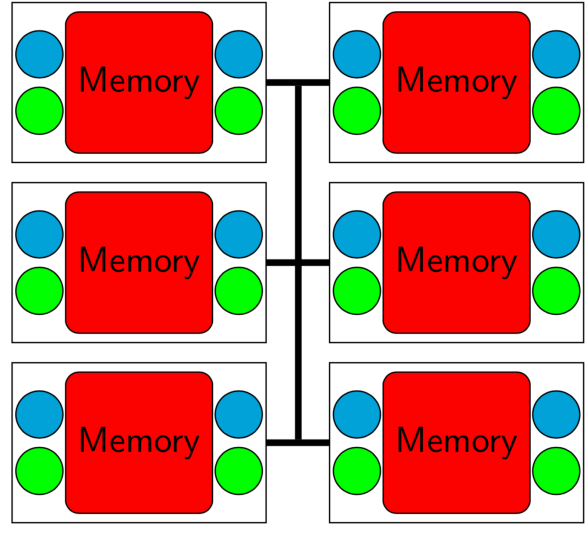
**Latency:** Wait time before getting the first data.

**Bandwidth:** Access speed once data starts flowing.

- Processor speed: (~GFlop) ~ a few ns or less.
- Communication is always the slowest part of your calculation!

# Hybrid architectures: accelerators

- Multicore nodes linked together with an (high-speed) interconnect.
- Nodes also contain one or more accelerators, GPGPUs (General Purpose Graphics Processing Units) or Xeon Phis.
- These are specialized, super-threaded (500-2000+) processors.
- Specialized programming languages, CUDA and OpenCL, are used to program these devices.



# Choosing your programming approach

The programming approach you use depends on the type of problem you have, and the type of machine that you will be using:

- Embarrassingly parallel applications: scripting, **GNU Parallel**.
- Shared memory machine: **OpenMP**, p-threads.
- Distributed memory machine: **MPI**, PGAS (UPC, Coarray Fortran).
- Graphics computing: CUDA, OpenACC, OpenCL.
- Hybrid combinations.

We focus on OpenMP, MPI programming in this course

# Data or computation bound?

The programming approach you should use also depends upon the type of problem that is being solved:

- Computation bound, requires task parallelism
  - Need to focus on parallel processes/threads.
  - These processes may have very different computations to do.
  - Bring the data to the computation.
- Data bound, requires data parallelism
  - The focus here is the operations on a large dataset.
  - The dataset is often an array, partitioned and tasks act on separate partitions.
  - Bring the computation to the data.

# Granularity

The degree to which parallelizing your algorithm makes sense affects the approach used:

- Fine-grained (loop) parallelism
  - Smaller individual tasks.
  - The data is transferred among processors frequently.
  - Shared Memory Model, OpenMP.
  - Scale Limitations
- Coarse-grained (task) parallelism
  - Divide and conquer.
  - Data communicated infrequently, after large amounts of computation.
  - Distributed memory, MPI.

Too fine-grained → overhead issues.

Too coarse-grained → load imbalance issues.

The balance depends upon the architecture, access patterns and the computation.

# Summary

- You need to learn parallel programming to truly use the hardware that you have at your disposal.
- The serial only portions of your code will truly reduce the effectiveness of the parallelism of your algorithm. Minimize them.
- There are many different hardware types available: distributed-memory cluster, shared-memory, hybrid.
- The programming approach you need to use depends on the **nature of your problem**.

# References


- SciNet Education (https://support.scinet.utoronto.ca/education/help/help_about.php)
- International HPC Summer School 2018 Lectures
- ARCHER UK National Supercomputing Service Training Courses (http://www.archer.ac.uk/training/)
- Muna, D & Price-Whelan, A. SciCoder Workshop. scicoder.org
- Peter Norvig, “What to demand from a Scientific Computing Language”, Mathematical Sciences Research Institute, 2010.