Parallelism

Supercomputer Architectures and Overview of Parallelisation

Parallelism

Supercomputer Architectures

- CPUs (central processing units) have been the workhorse of computing since the 1960s, taking over from mechanical punch cards

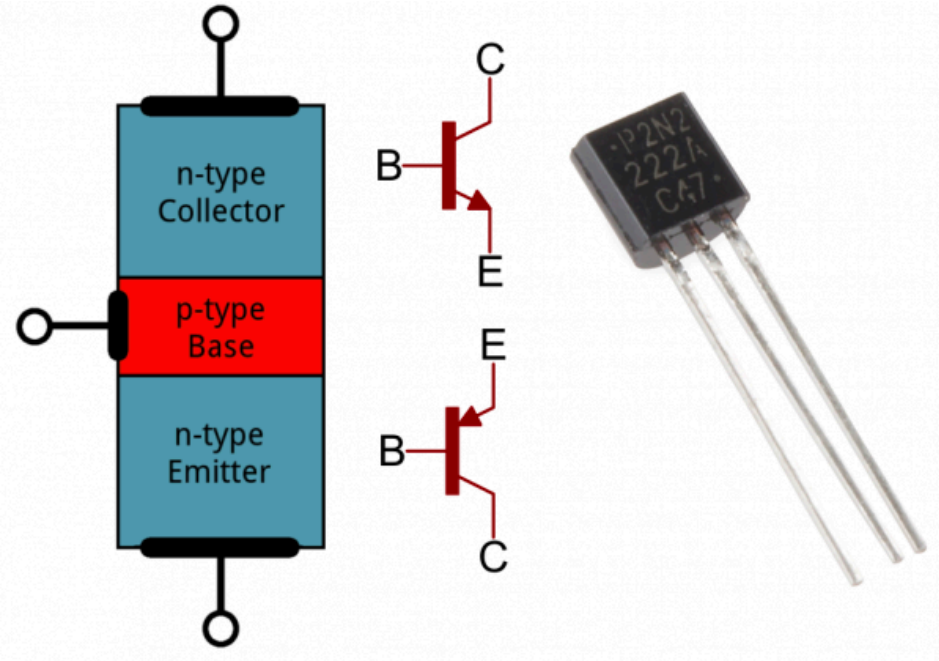
- From the 1960s, it was possible to speed up a CPU by adding more *transistors* to a chip, due to a reduction in the transistor size over time

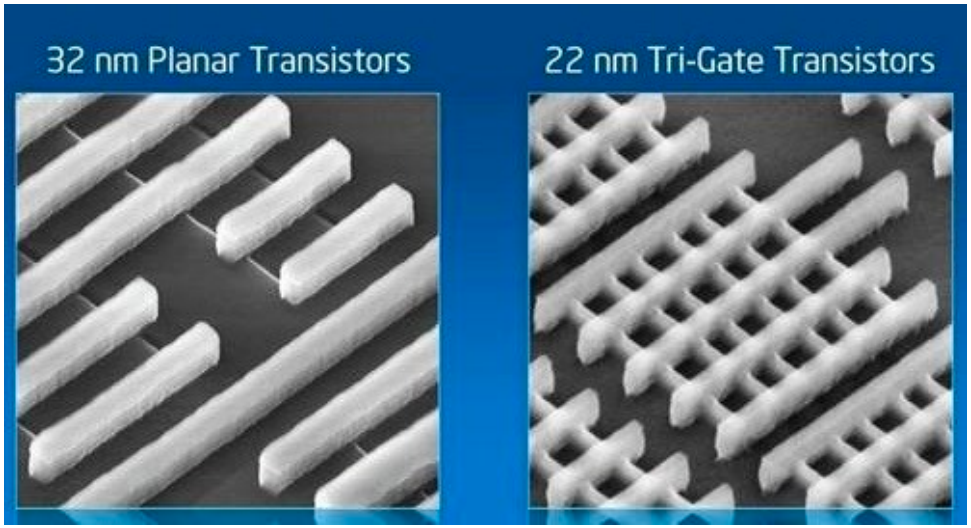
  - A transistor is a semiconductor device that is used heavily in electronics

  - They can amplify or switch electric signals via *logic gates* - used to switch between binary 0s and 1s

Parallelism

Supercomputer Architectures





Parallelism

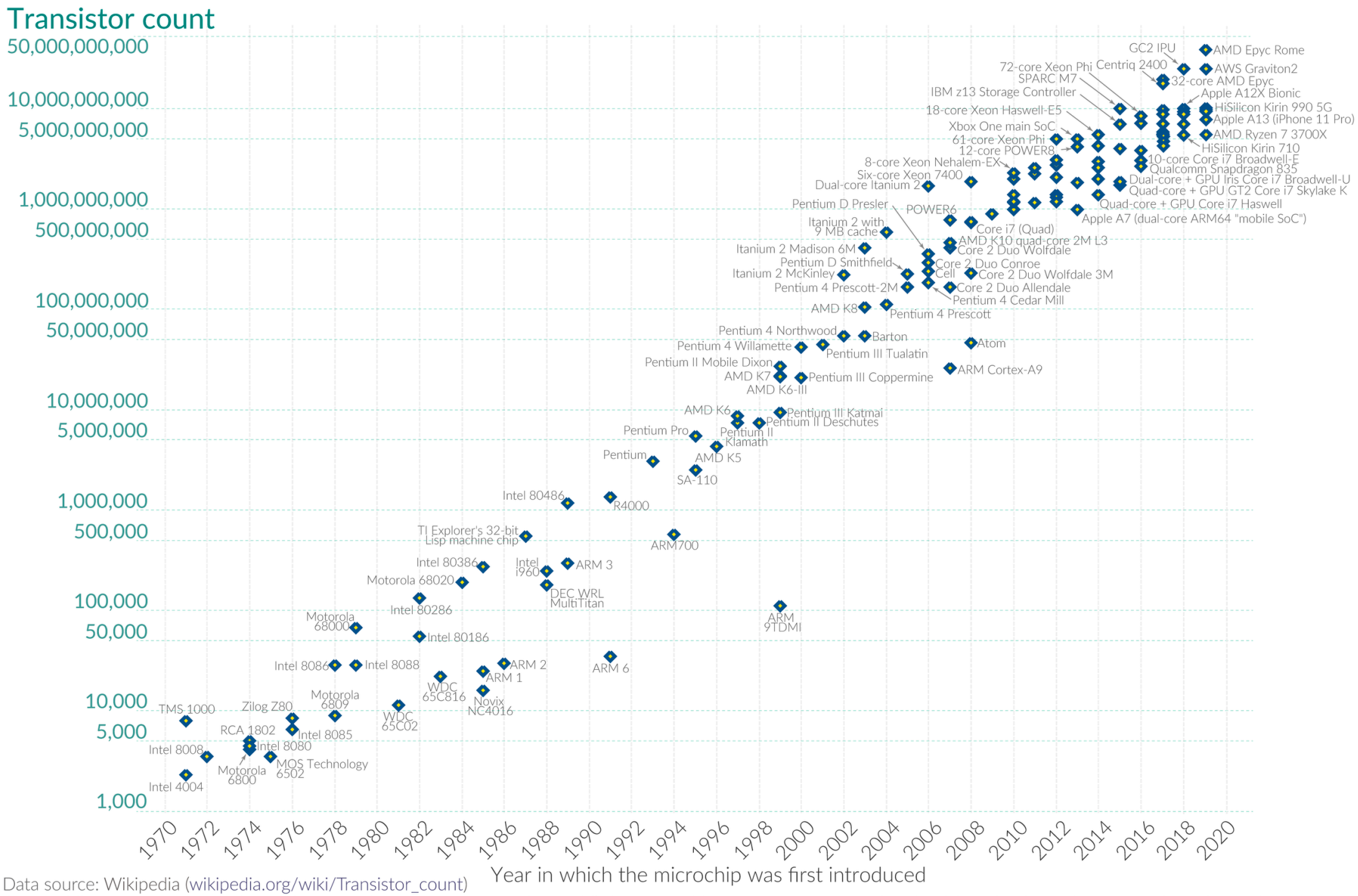
Supercomputer Architectures

- This is encapsulated in *Moore's Law* - 'the number of transistors in an integrated circuit doubles about every two years' i.e. exponential growth
  - The key reason for this growth in number was due to a *reduction in size* of the transistor
  - Currently, the most modern chips can fit tens of billions of transistors
- In general, this doubling would come at *no increase in cost*

Moore's Law: The number of transistors on microchips doubles every two years

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 for other aspects of technological progress in computing – such as processing speed or the price of computers.

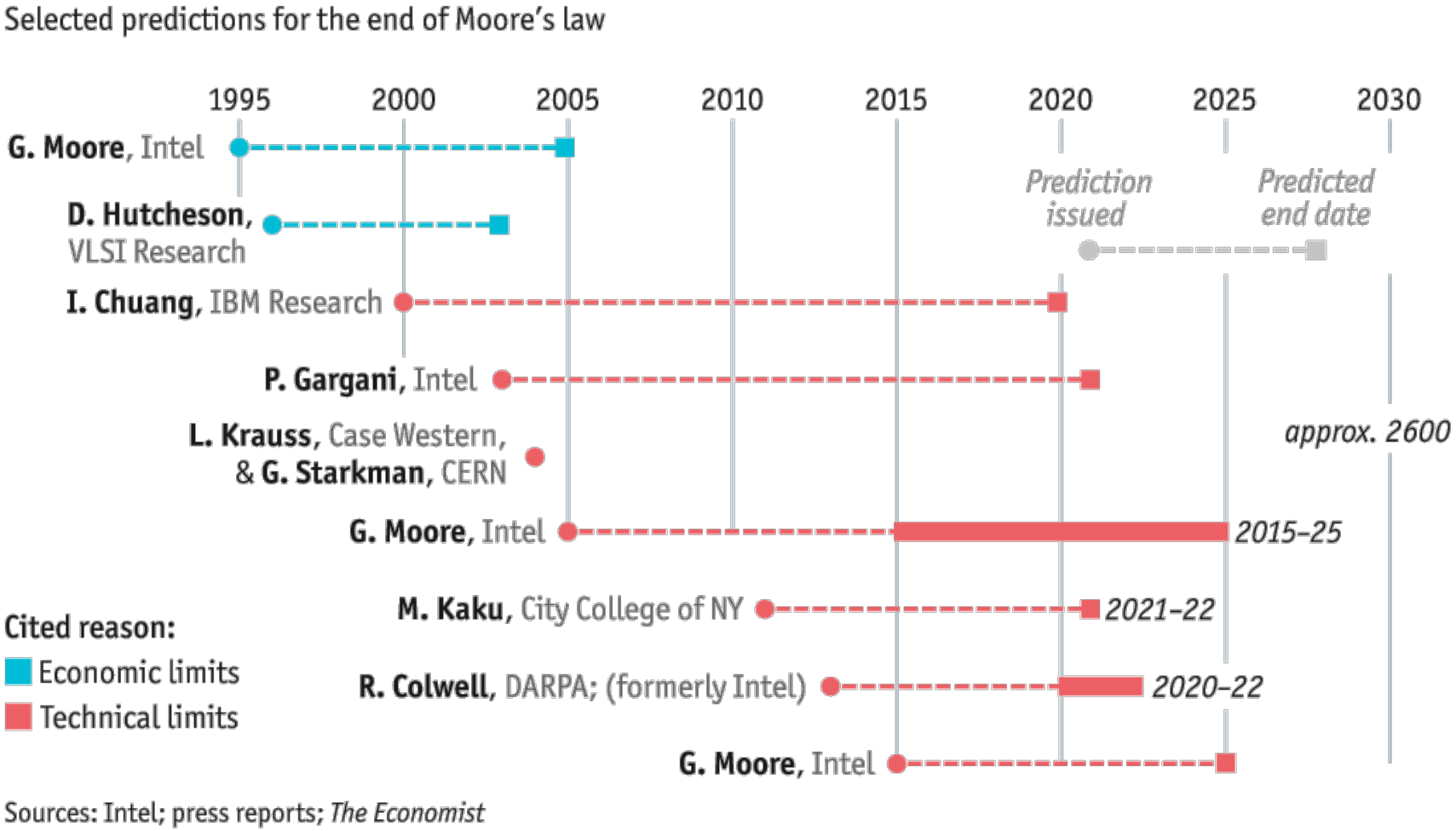


Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

Parallelism

Supercomputer Architectures

- There have been many predictions over when Moore's law will end…



Parallelism

Supercomputer Architectures

- Moore's Law can be combined with *Dennard's (MOSFET) scaling:*
  - *F*or a transistor, $P = CV^2f$,
    - C - capacitance, scales with linear size of transistor
    - V - voltage, scales with linear size (electric field constant)
    - Current and transition time also scaled down with linear size
    - f - frequency scales with 1/delay
    - P - power consumption decreases, scaling with area
  - As the transistors get smaller, the *power density* stays constant - if power density doubles, power consumption (twice the transistor number) stays the same

Parallelism

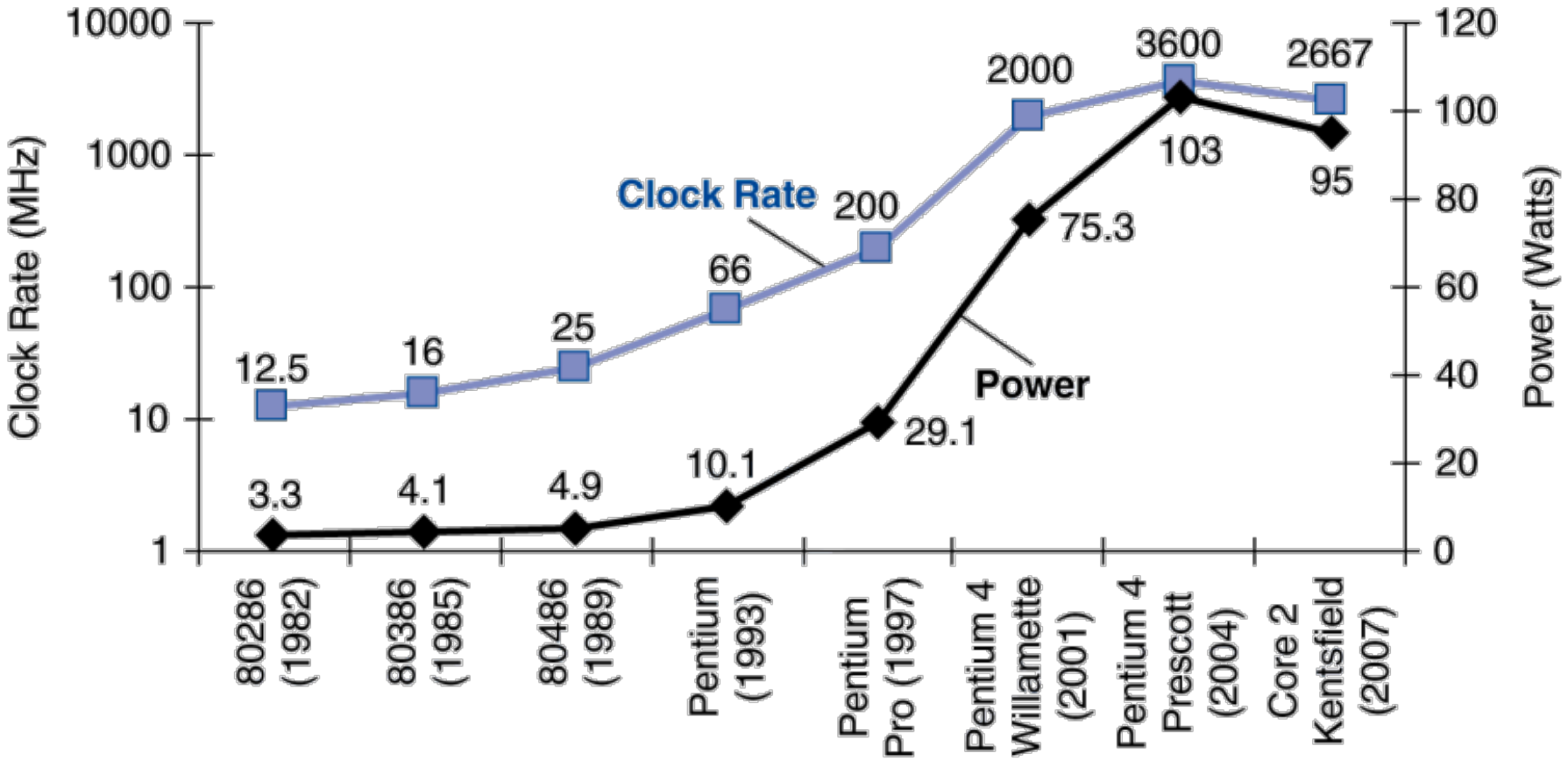
Supercomputer Architectures

- Together, this meant that performance per joule of energy grew even faster than the transistor number, doubling every ~18 months
- Smaller transistors enables a higher frequency of operations for a fixed power
- Over this time, *clock speed* of CPUs increased dramatically
  - Clock speed is the number cycles per unit time
- This meant that, in practice, you could speed up your program simply by using more up-to-date CPUs
- This is no longer the case…

Parallelism

Supercomputer Architectures

- Although Moore's law seems to be holding for now, for the last 20 years or so, the performance of CPUs has stalled



Parallelism

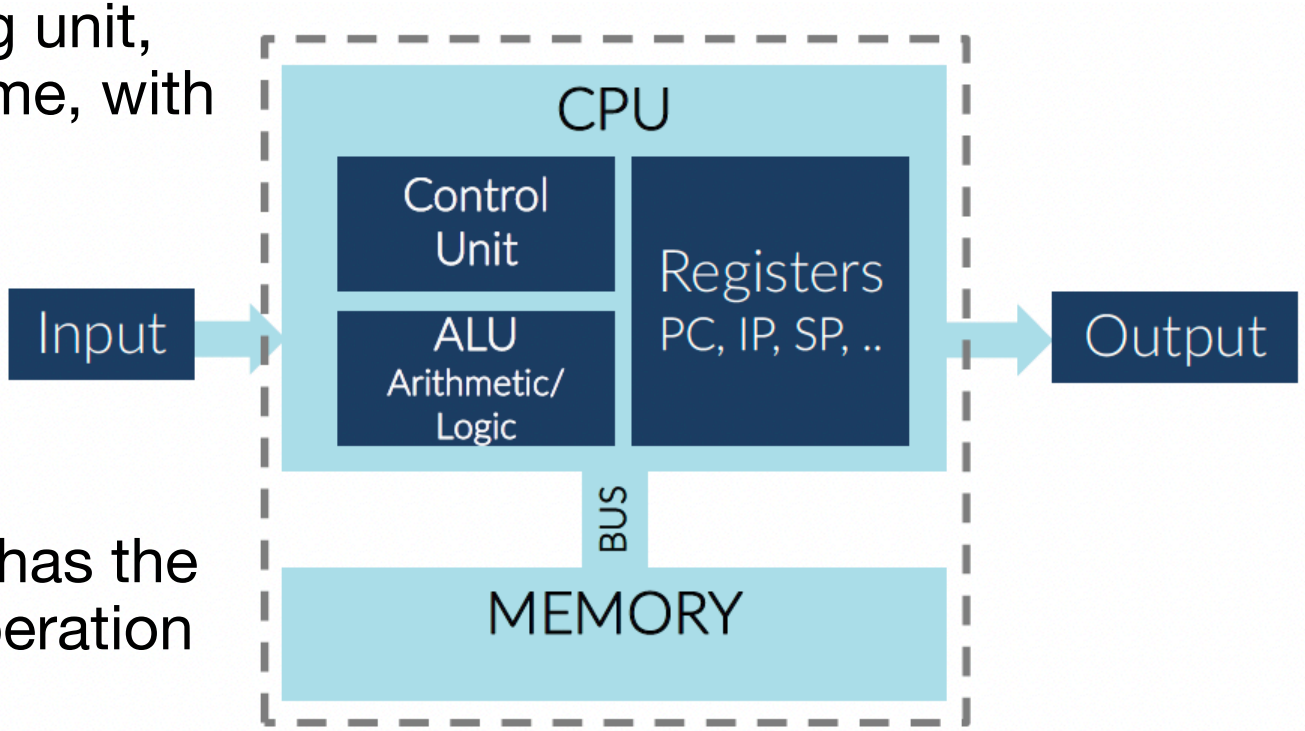
Supercomputer Architectures

- This is because physical limits have come into play - Dennard's scaling is broken
- Since the mid 2000s, as transistors have become smaller, the power density has *increased*, limiting the clock rate
- This increase is caused by *quantum effects*, such as:
  - Current leakage - also causes the chip to heat up
  - Threshold voltage
  - Physical limits and atomic scales
- One intrinsic limit on CPUs is the clock rate
- However, CPUs contain other elements that can also limit performance…

Parallelism

Supercomputer Architectures

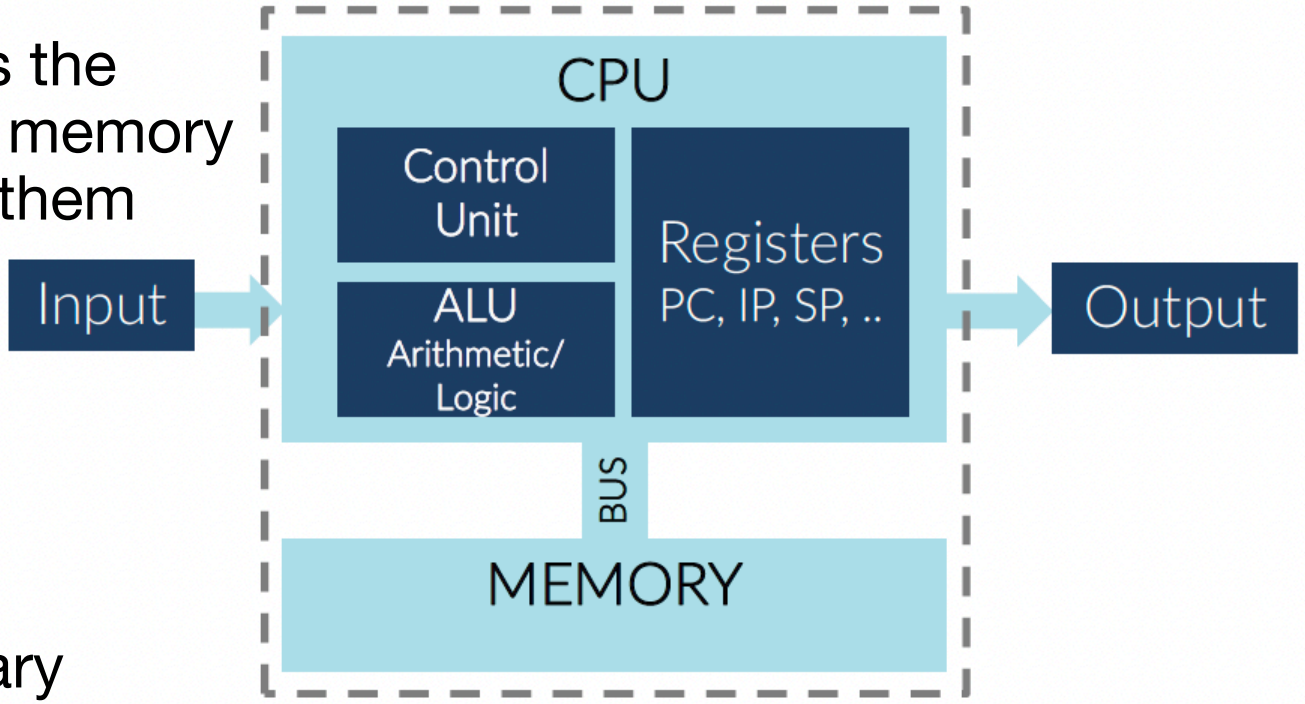
- To understand how a CPU works, it is useful to understand the *architecture*
- We first look at a *single core* (von Neumann architecture)
- This is one processing unit, one instruction at a time, with `flat' memory, i.e.
  - Access to any memory location has the same cost
  - Access to memory has the same cost as an operation execution



Parallelism

Supercomputer Architectures

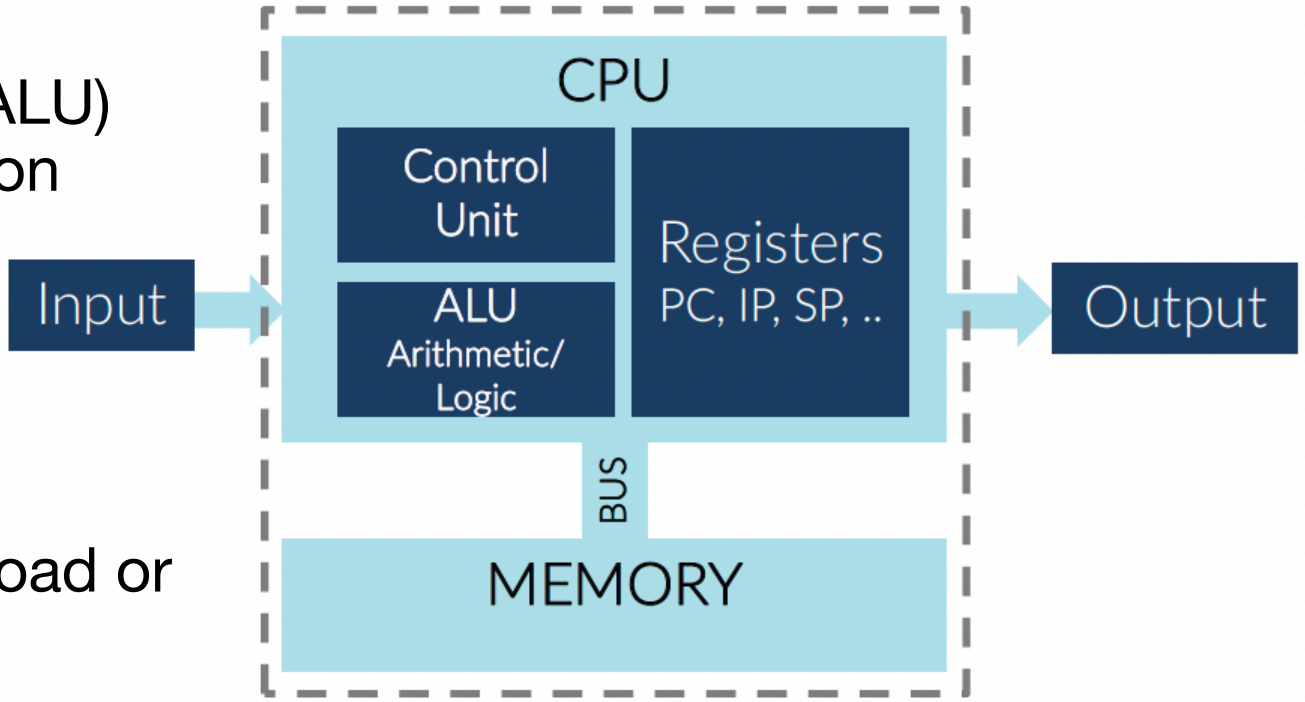
- Registers - high speed storage areas in the CPU where all data must be stored before it can be processed
- Arithmetic and Logic Unit - enables arithmetic and logic operations
- Control Unit - controls the operation of the ALU, memory and I/O devices, tells them how to respond to program instructions from memory unit
- Bus - transmits data
- Memory - RAM, primary



Parallelism

Supercomputer Architectures

- First, control unit fetches the next instruction from memory to the instruction register (IR) (instructions are stored as data, with memory addresses)
- The control unit decodes the instruction on the IR
- The processing unit (ALU) executes the instruction
- The control unit stores the result to memory
- Input and output eg. load or display results



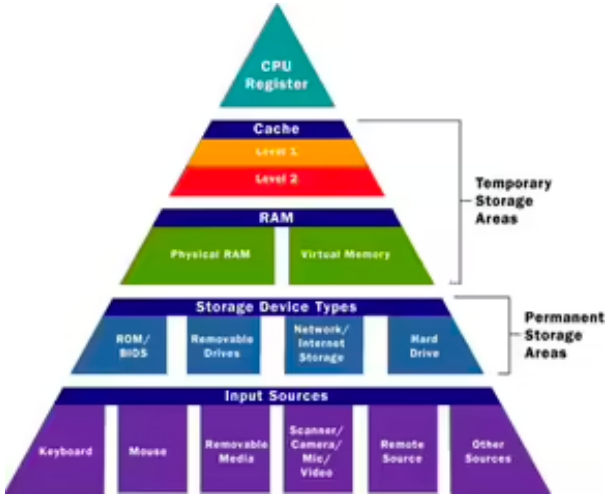
Parallelism

Supercomputer Architectures

- In the 1990s, the CPU speed began to overtake the speed of the memory transfer from RAM
  - The time between initiating a request for something stored in memory and it being received by a processor is called the *memory latency* - higher latency = longer wait
- This is the so-called *memory wall,* creating another limit on the overall performance
- To get around this, it was necessary to speed up the memory
- This was achieved via a mechanism called the *cache*
  - This term comes from wineries - we will see why!

Parallelism

Supercomputer Architectures



- For memory to be faster, it has to be *closer* to the CPU - it is (mostly) part of the CPU itself, rather than being accessed via bus
- The cache is therefore significantly smaller than the RAM
- Overall, cache memory can operate 10-100x faster than RAM
- There are different levels of cache memory:
  - L1 (Level 1) - fastest memory (lowest latency), contains the data the CPU is most likely to need. Usually < 1MB (although no standard size), ~100x faster than RAM. Assigned for each core.
  - L2 - slower, but larger in size, such as few MB, ~25x faster than RAM. Assigned for each core.
  - L3 - slower again, larger again ~ a few tens of MB. Can be used by entire CPU, i.e. shared by many cores (see later)

Parallelism

Supercomputer Architectures

- The CPU decides what to store in the cache using *prefetching*
- For example, if we define an array `a[100]` and access the first element, it is very likely we will also access the other elements - the CPU will prefetch the array to the cache
- More generally, `local' data are likely to be in the cache, by which we mean either:
  - Temporal locality - if an address is referenced, it is likely to be referenced again soon
  - Spatial locality - if an address is referenced, nearby addresses are likely to be referenced again soon
- If we are successful in fetching data from the cache, this is a *cache hit*
- If we are unsuccessful, this is a *cache miss* - these can give significant performance penalties

Parallelism

Supercomputer Architectures

Recent example of optimising using cache blocking (from Wu Hyun Sohn, one of James' ex-PhD students!) - this is an optimisation of a code for CMB bispectrum estimation of primordial non-Gaussianity:

for each set of modes (p_1, p_2, p_3) **do**
  for each pixel n **do**
    $\beta^{\rm cub}(i, p_1, p_2, p_3)$ += $m(p_1, n) \cdot m(p_2, n) \cdot m(p_3, n)$
    $\beta^{\rm lin}(i, p_1, p_2, p_3)$ += $C(p_1, p_2, n) \cdot m(p_3, n)$
  end for
end for

- The original algorithm fetches 4 large arrays from RAM, as each of them are too large to be stored in the cache

Parallelism

Supercomputer Architectures

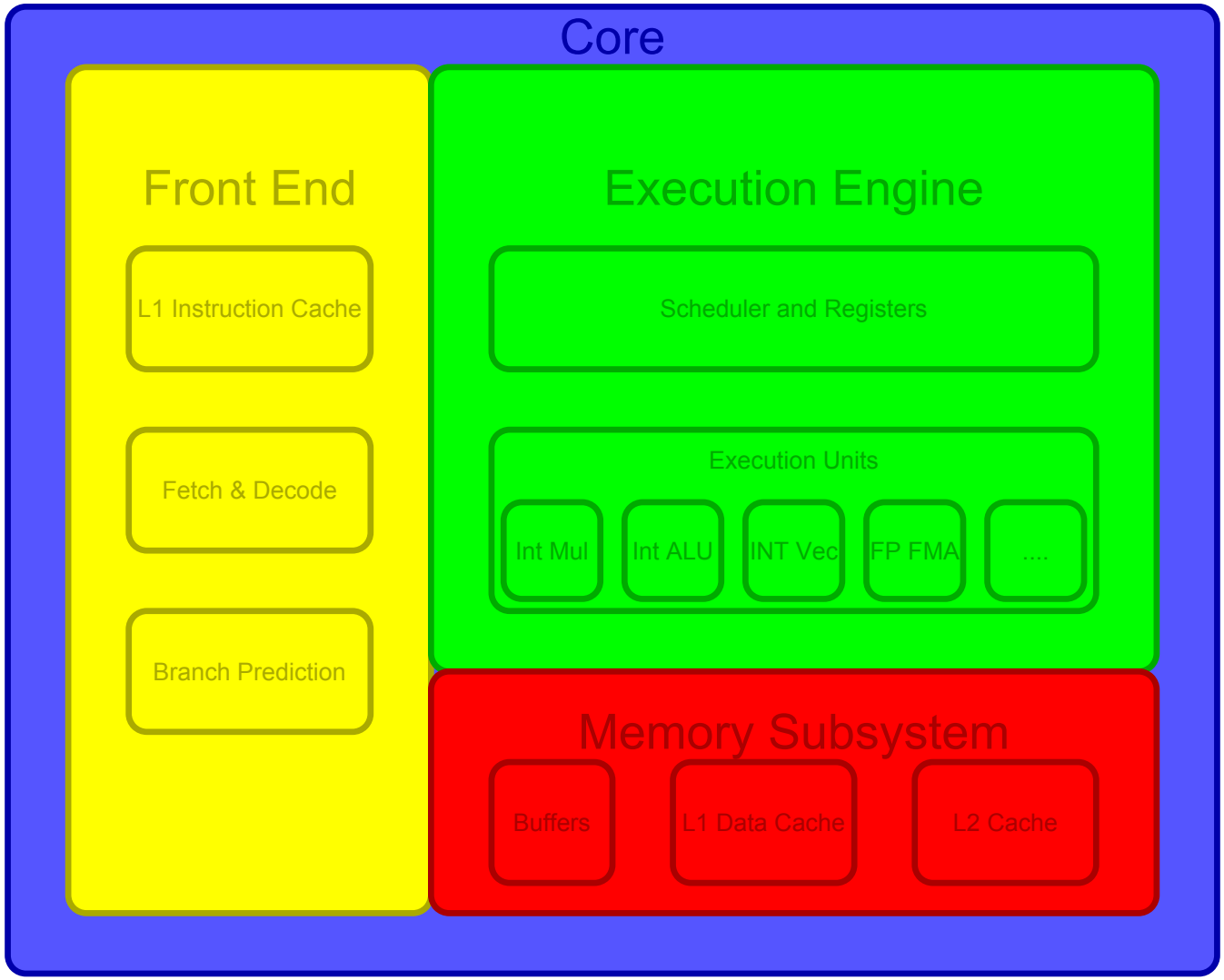
- We can divide these large arrays into blocks that fit inside the cache memory:

for each block b **do**
  for each set of modes (p_1, p_2, p_3) **do**
    for each pixel n' in block **do**
      $\beta^{\text{cub}}(i, p_1, p_2, p_3) \mathrel{+}= m(p_1, n') \cdot m(p_2, n') \cdot m(p_3, n')$
      $\beta^{\text{lin}}(i, p_1, p_2, p_3) \mathrel{+}= C(p_1, p_2, n') \cdot m(p_3, n')$
    end for
  end for
end for

- Here, the new pixel number is calculated as $n' = B \cdot b + n$, where B is the size of each block and $o \leq n < B$
- Data locality is greatly improved, as well as temporal locality

Parallelism

Supercomputer Architectures



Parallelism

Supercomputer Architectures

- Since the introduction of caches in the 1970s, there have been continuous improvements made to CPUs to enhance performance
- These include:
  - *Superscalar capacity* - the ability to execute more than one instruction per cycle
  - Hardware that facilitates *(multiple) pipelines* - detaching the fetching, decoding, execution and writeback stages of an instruction
  - *Vector registers* - large special registers in the CPU that can be subdivided into independent chunks, over which the same operation can be performed (see James' next lecture)
- Development of CPUs has been very effective, however…

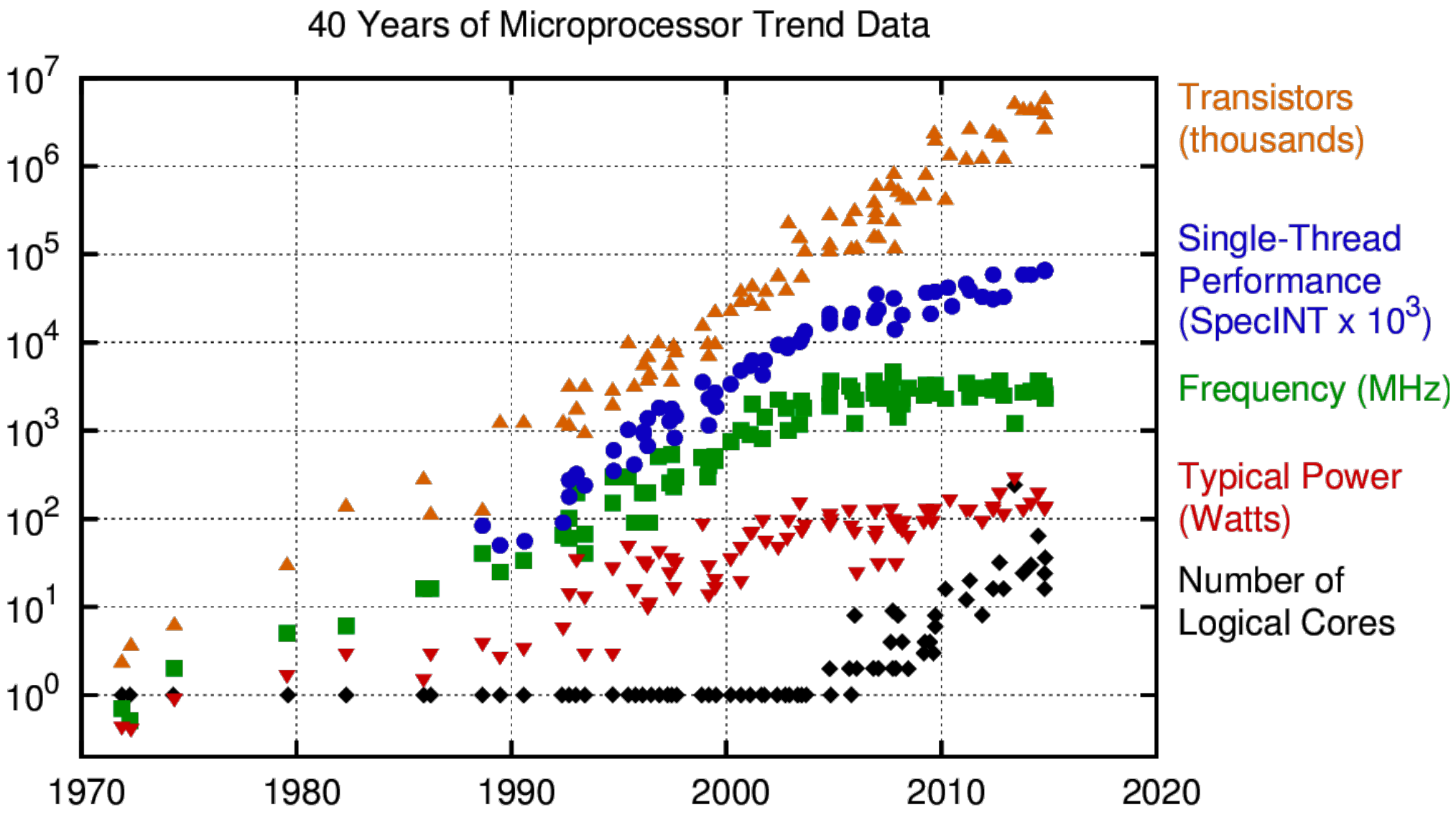
Parallelism

Overview of Parallelisation

- As we have seen, speeding up the CPU ad infinitum is not possible, due to breaking of Dennard's scaling
- This means that we must use *parallelism*, usually over *multiple cores* in order to increase performance
- In short, this means that we split our program (or, more commonly, parts of a program) into smaller chunks, each of which can be run independently
  - Note that this is most effective for *large problems* - for small problems that fit on one core, splitting across multiple processors may not speed up your program (it could even slow it down, due to communication costs)
  - If your problem is inherently not parallelisable, e.g. each operation must be performed in sequence, throwing more cores at it will not give any performance increase - you will be limited by the clock speed
- Why can't we just use one very big core? Both physical (limits on eg. heat dissipation) and business (cost, size) reasons

Parallelism

Overview of Parallelisation



Parallelism

Overview of Parallelisation

Example: parallelisable vs non-parallelisable programs

- Non-parallelisable - calculation of Fibonacci series

```
for (k=0, k < n, k++) {
   F(k+2) = F(k+1) + F(k);}
```

- Parallelisable - calculation of a sum

```
my_sum = 0;
my_first_i = ..;
my_last_i  = ..;
for (my_i = my_first_i; my_i < my_last_i, my_i++) {
   my_x = compute_next_value(..);
   my_sum += my_x; }
```

Parallelism

Overview of Parallelisation

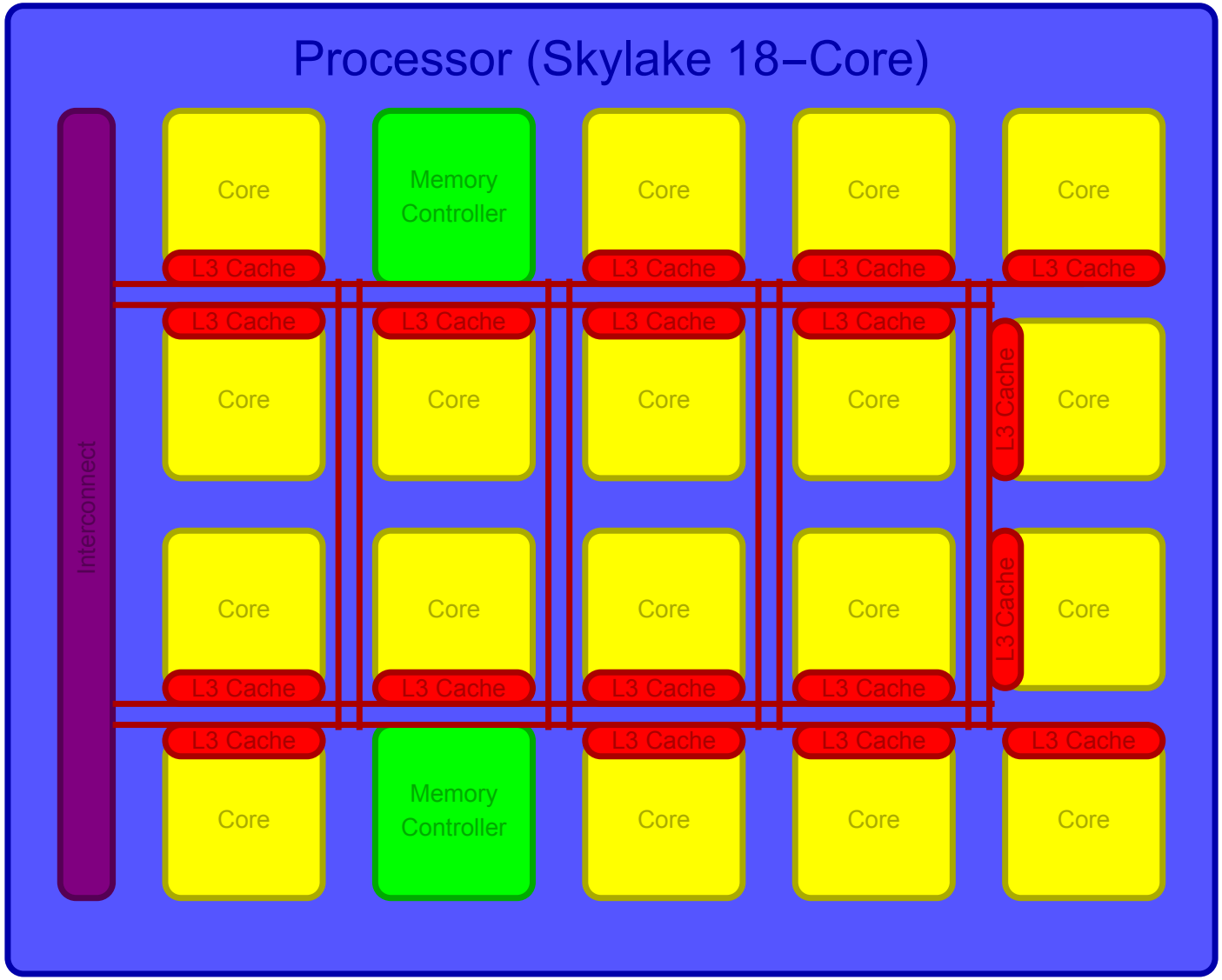
- The potential performance gains from parallelism are given by *Amdahl's law:*

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

  - S_{latency} - theoretical speedup of a whole program
  - s - speedup from parallelism (or number of processors)
  - p - fraction of the program that can be parallelised

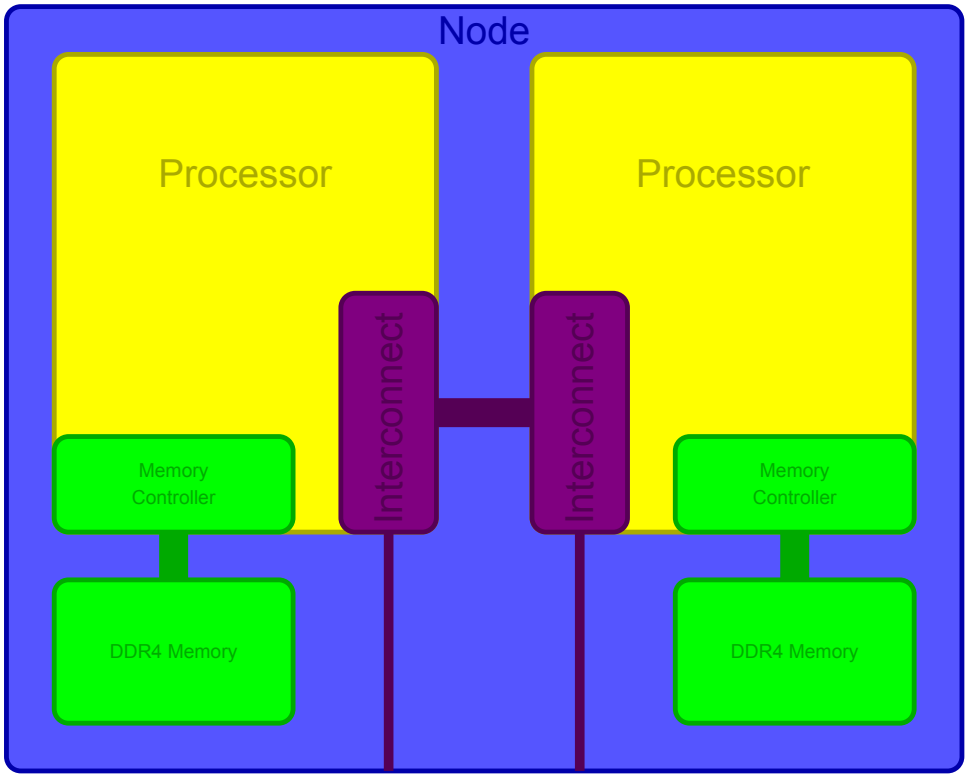
Parallelism

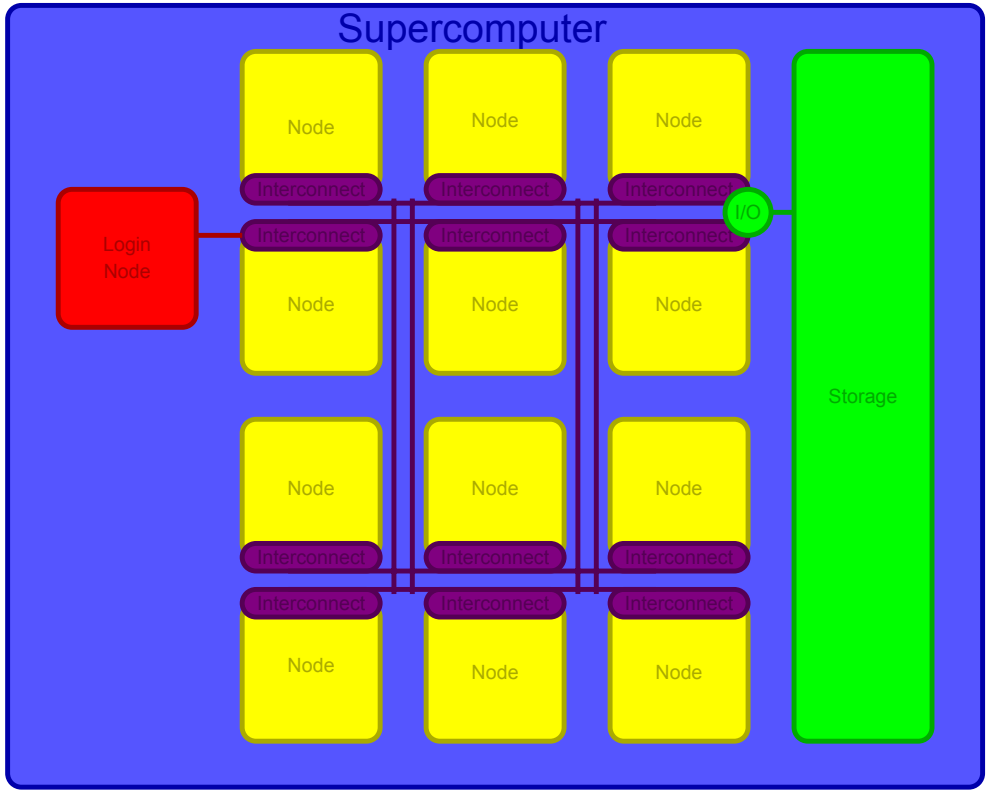
Overview of Parallelisation



Parallelism

Overview of Parallelisation

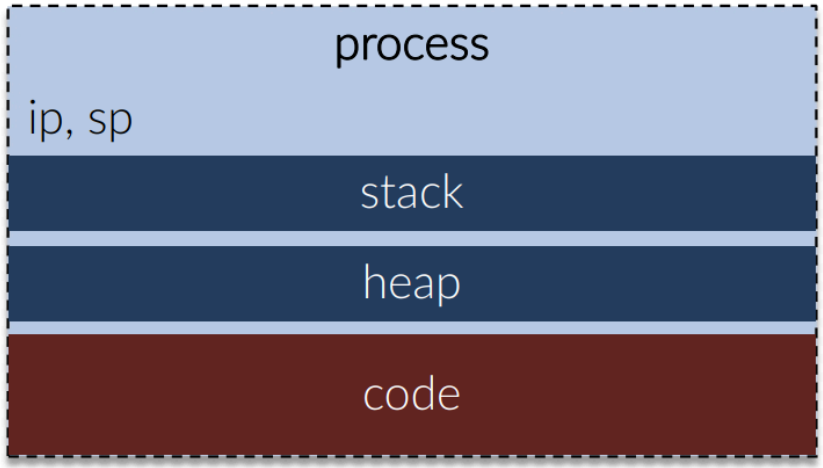


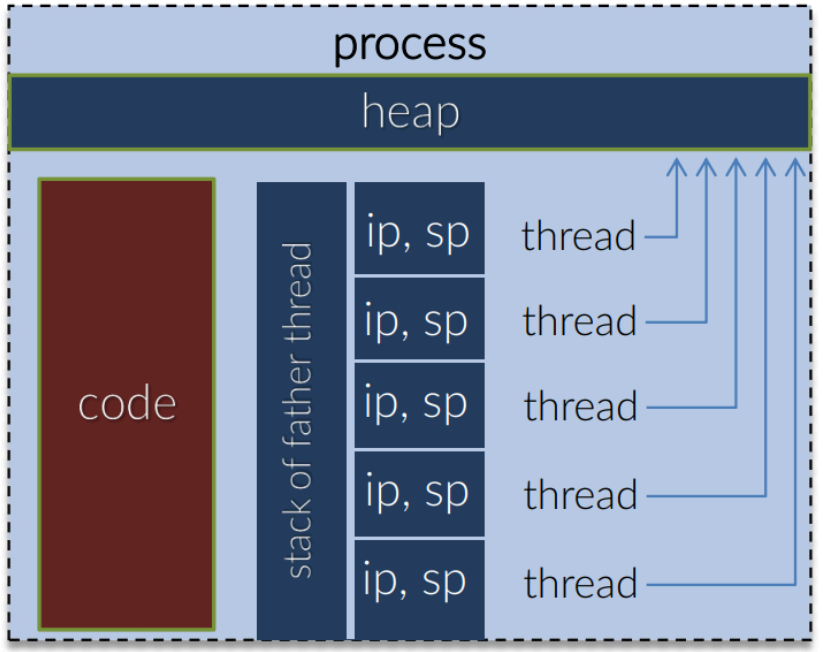


Parallelism

Overview of Parallelisation

- How do we actually implement parallelism?
- Parallelism makes use of two important concepts:
  - *Process* - this is an independent sequence of instructions and all of the resources needed for their execution
  - *Thread* - an independent instance of code execution *within* a process
    - Each thread shares the same code, memory address space and resources to its process
    - Each thread has its own *stack*, but the *heap* is shared between threads which operate in *shared memory*

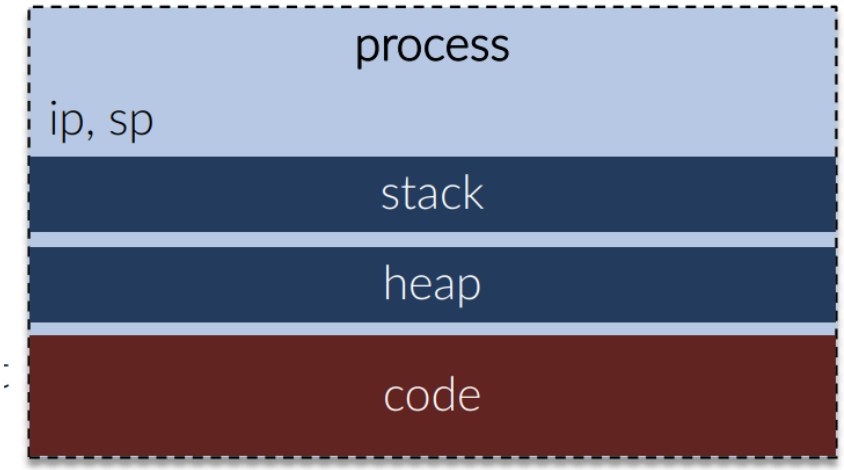


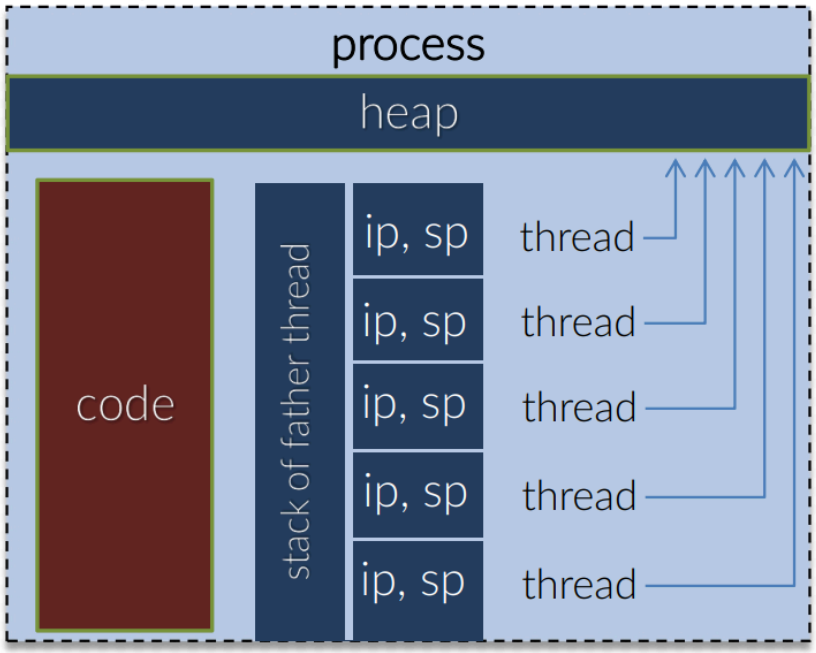


Parallelism

Overview of Parallelisation

- `Stack' and `heap' are both types of memory that are stored in RAM
- What is the stack?
  - `Local' memory that contains local variables of each function
  - Very limited scope - can only be accessed by the current function or its callees
  - Allocated to you by the operating system
  - Used primarily for *static allocation* (ie. at compile time)
  - A lot faster and more optimised
  - (Cf stackoverflow)

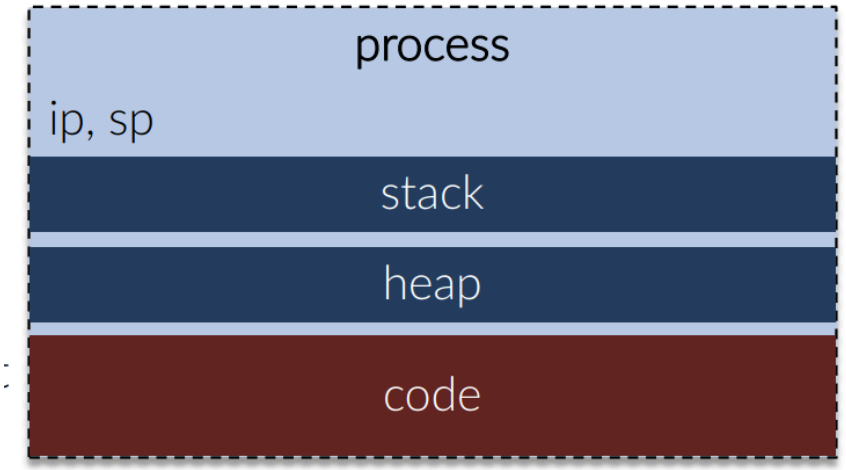


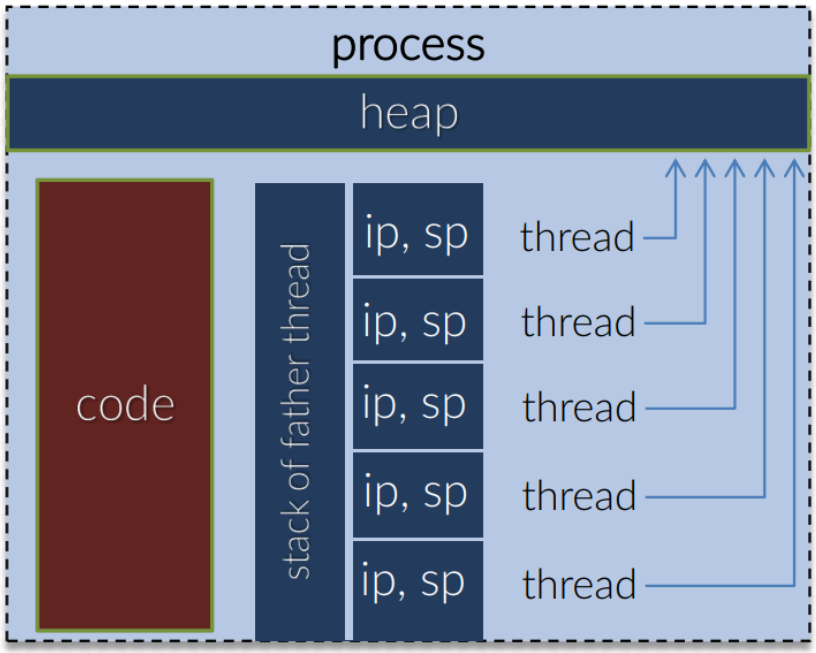


Parallelism

Overview of Parallelisation

- What is the heap?
  - Memory that stores all of your data and global variables
  - Accessible from all functions in all code units
  - Used primarily for *dynamic allocation* (ie. at runtime)
- So, to recap - *processes* have access to both stack and heap memory, *threads* have access to the heap, each with their own stack
- In general, creating threads inside a process is much less costly than creating more processes





Parallelism

Overview of Parallelisation

- In order to parallelise a program, we need to assign different chunks to different threads, different processes, or a combination
- The main API (application programming interface) for using threads for parallelism is *OpenMP* - OPEN specifications for MultiProcessing
  - Allows you to (quite) easily reformat your code using `pragma` statements
- For using processes, the most common method is MPI - Message Passing Interface
  - Usually requires more involved code restructuring with explicit communication between processors
- We usually assign one thread or one process per core (unless we have hyperthreading - multiple threads per core)
- We will cover how to implement OpenMP and MPI in future lectures…

Parallelism

Overview of Parallelisation

- It is not always clear which of the two methods is the most appropriate for a given problem
- One useful `rule' - OpenMP is most useful for running over multiple cores within *one CPU*
- This is because OpenMP does not include a mechanism for sharing data between multiple CPUs
- MPI facilitates the sharing of data between CPUs, so this is usually more appropriate in this case (although MPI is also often used to run over multiple cores within one CPU)
- It is also necessary to keep in mind how much memory your program will require
- Usually, a hybrid approach is the most effective

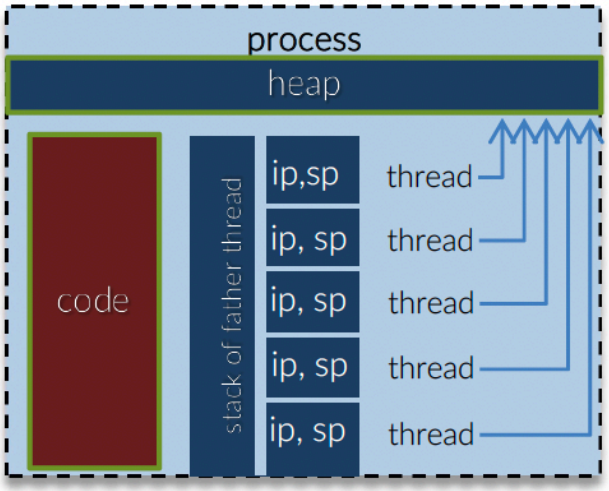
Parallelism

Overview of Parallelisation

Shared-Memory

(e.g. OpenMP)

A unique process that spawns a number of threads. There is a unique memory space that is accessible by all the threads



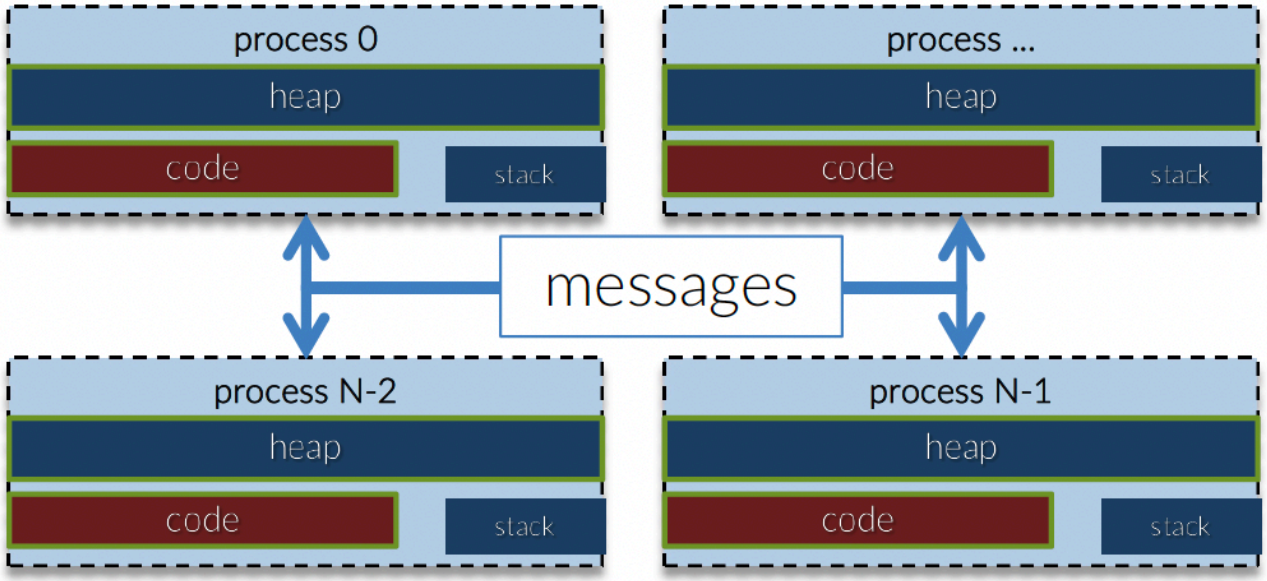
Distributed-Memory

(e.g. MPI)

N processes are created, each with its own copy of the code and its own memory space.

A process ***can not*** access the memory space of another process.

The processes communicate through *messages*.



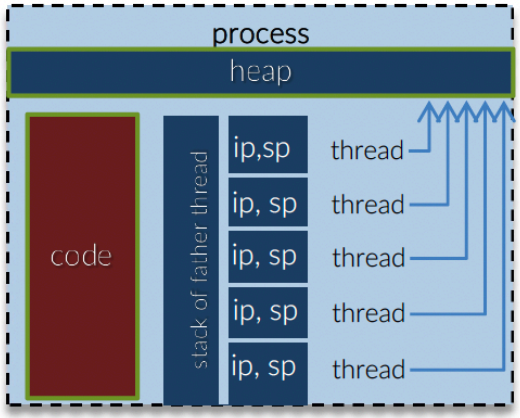
Parallelism

Overview of Parallelisation

Shared-Memory

(e.g. OpenMP)

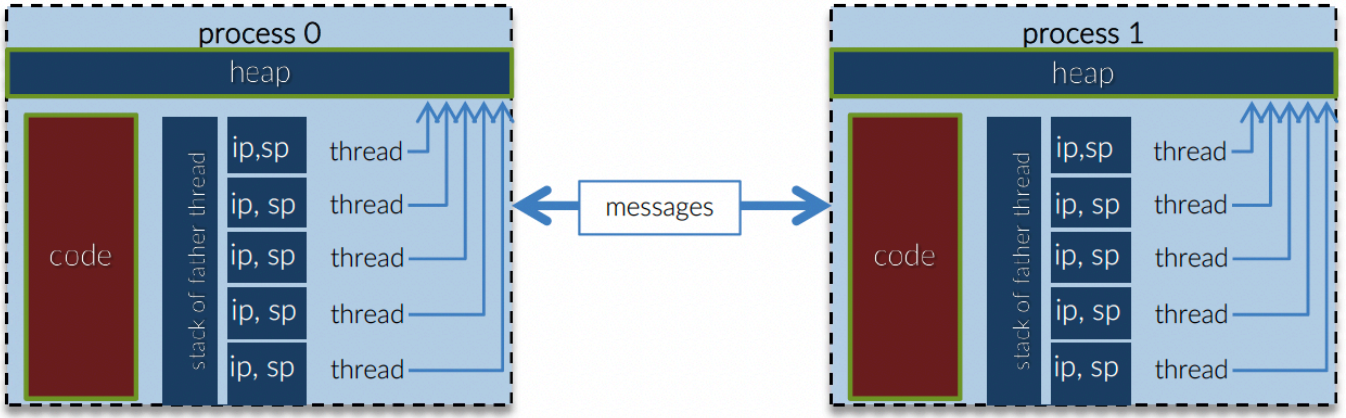
A unique process that spawns a number of threads. There is a unique memory space that is accessible by all the threads



Distributed-Memory

(e.g. MPI) + Shared-Memory

N processes are created, each with its own copy of the code and its own memory space. Each process may spawn a number of threads as in shared-memory.
A process *can not* access the memory space of another process (nor any of its threads can). The processes communicate through *messages*.



Parallelism

Overview of Parallelisation

- The next generation of supercomputing is *zettascale computing*
- Zettascale projects aim for 10^{21} operations per second (flops)
- See e.g. Cambridge Open Zettascale Lab https://www.zettascale.hpc.cam.ac.uk/
- Current state of the art supercomputing happens at *exascale* (10^{18} flops)
- Another important consideration is *energy usage* - one exascale goal is to reach exaflops at 20MW of electric power
  - Moving memory is the most expensive operation
- Some HPC systems eg. LUMI in Finland heat local towns using the extra heat from the system