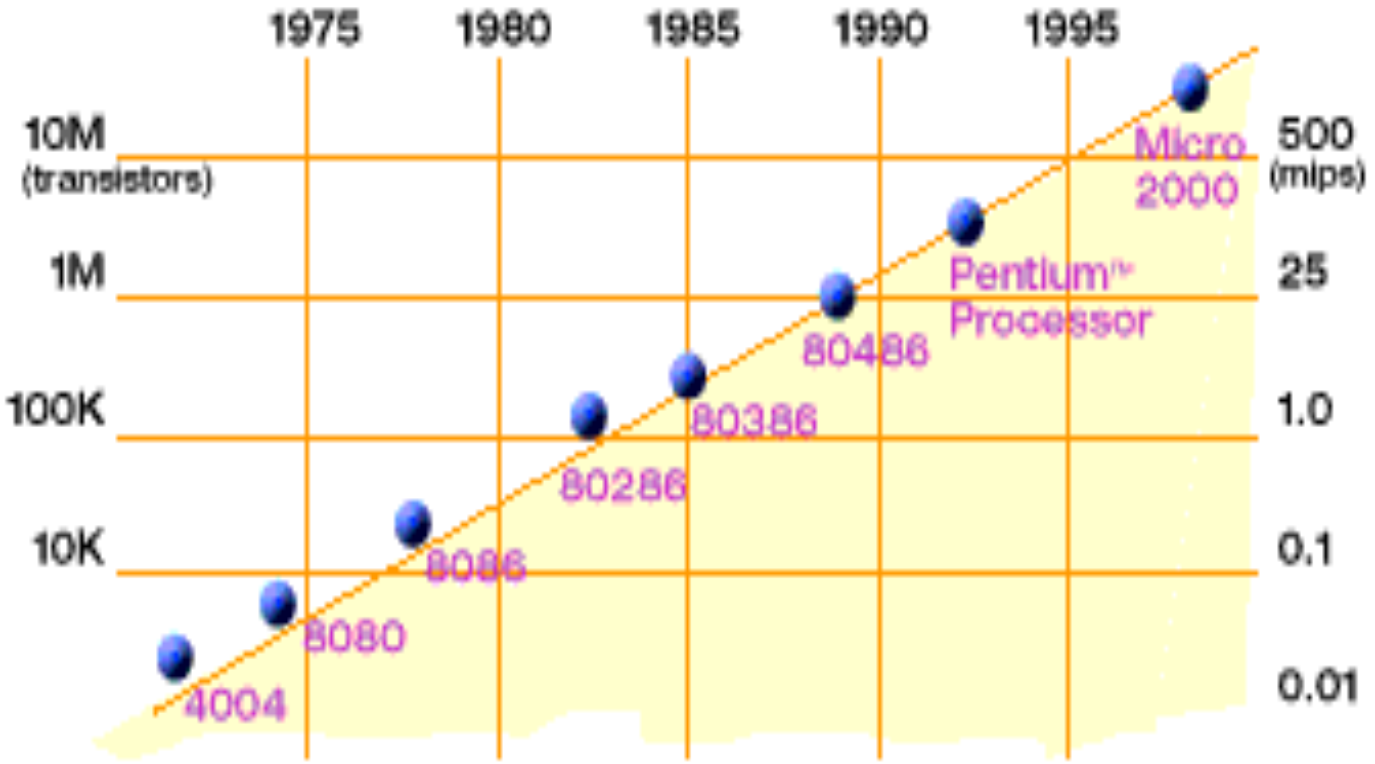
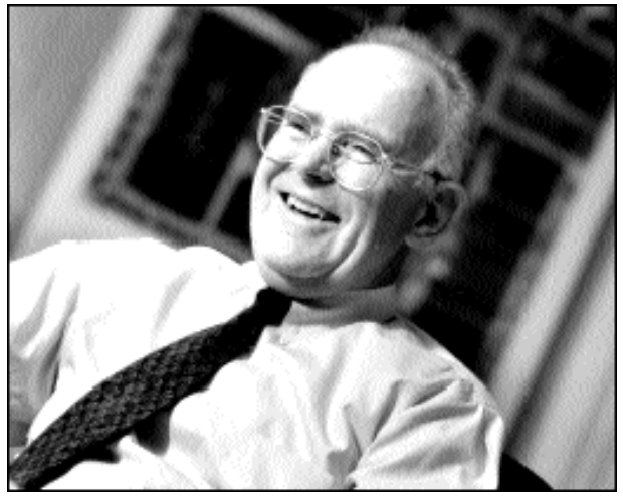
Parallelism, Parallel Systems, and vector computing

Tim demonstrates his new invention: Kayak snorkeling. Palawan Philippines, 2019

Moore's Law



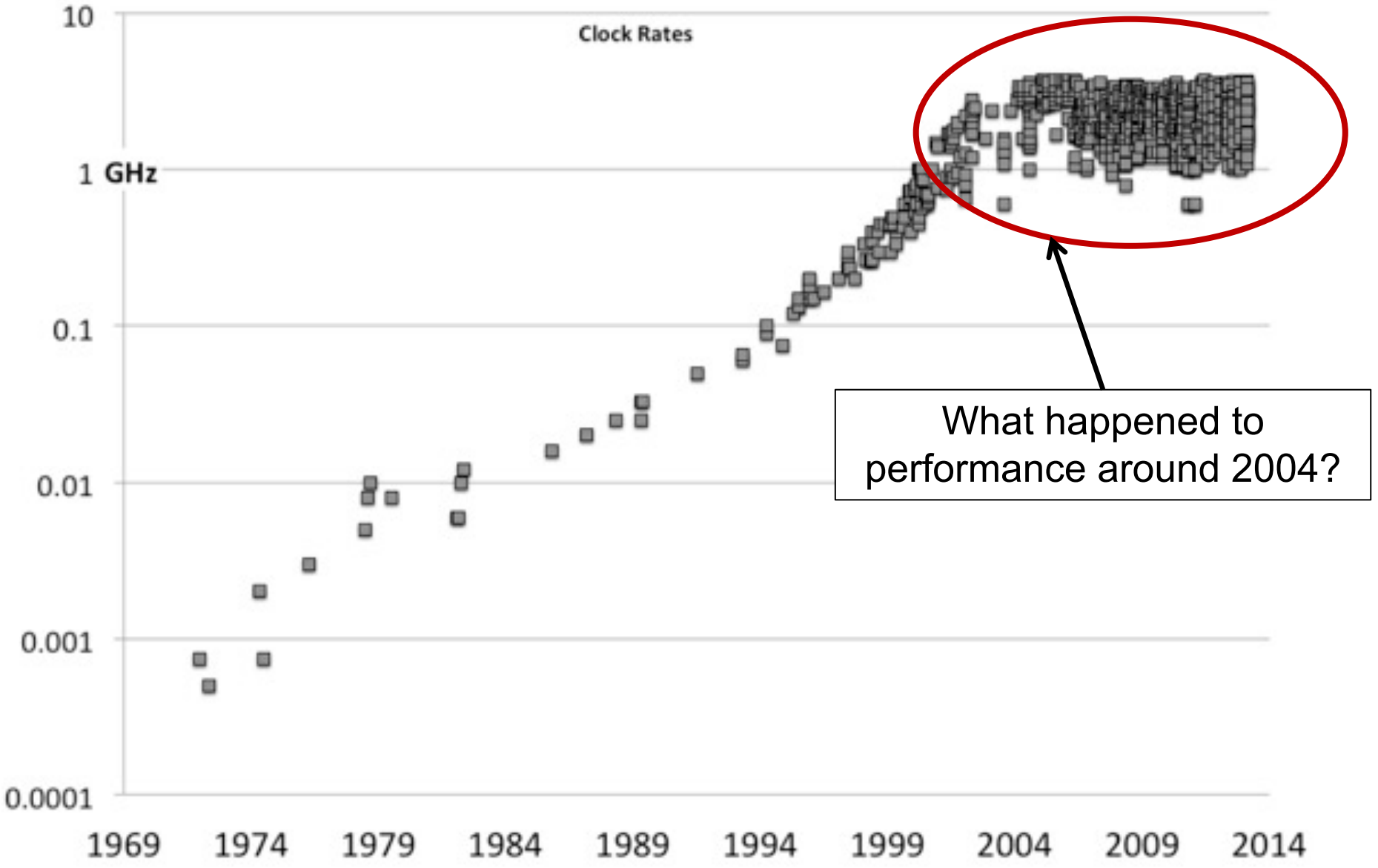


- In 1965, Intel co-founder Gordon Moore predicted (from just 3 data points!) that semiconductor density would double every 18 months.
  - ***He was right!*** Over the last 50 years, transistor densities have increased as he predicted.

"Cramming more components onto integrated circuits", G.E. Moore, Electronics, 38(8), April 1965

Slide source: UCB CS 194 Fall'2010

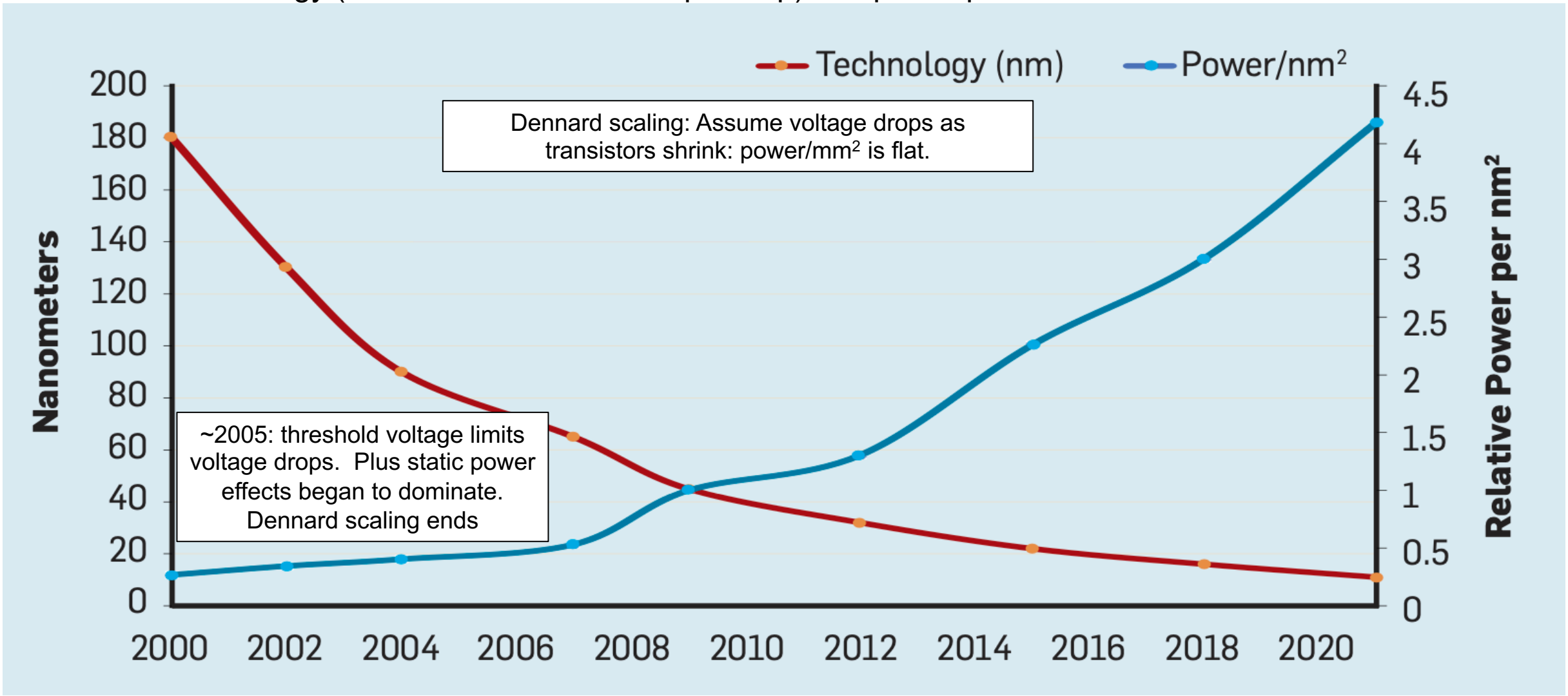
CPU Frequency (GHz) over time (years)



Source: James Reinders (from the book “structured parallel programming”)

Dennard Scaling

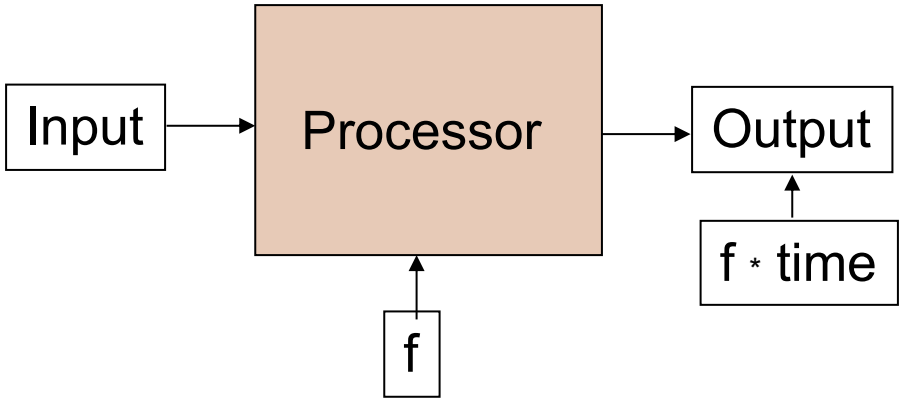
- Process technology (translates to Transistors per chip) and power per mm^2



Process technology nodes defined by the smallest feature on a chip (i.e. gate length in nm). After 22 nm, it's become a marketing term that doesn't map to a specific feature's length.

Image from "A New Golden Age for Computer Architecture", John Hennessy and David Patterson, CACM, 2019

Consider power in a chip …



Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f

C = capacitance … it measures the ability of a circuit to store energy:

$C = q/V \rightarrow q = CV$

Work is pushing something (charge or q) across a "distance" … in electrostatic terms pushing q from 0 to V:

$V * q = W.$

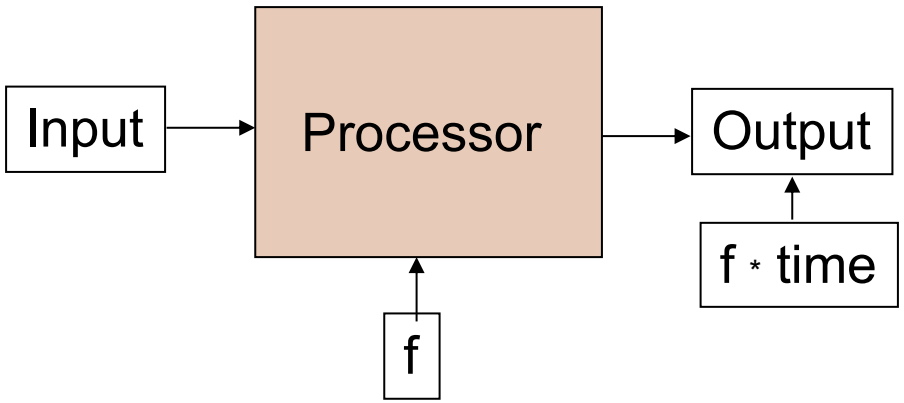
But for a circuit $q = CV$ so

$W = CV^2$

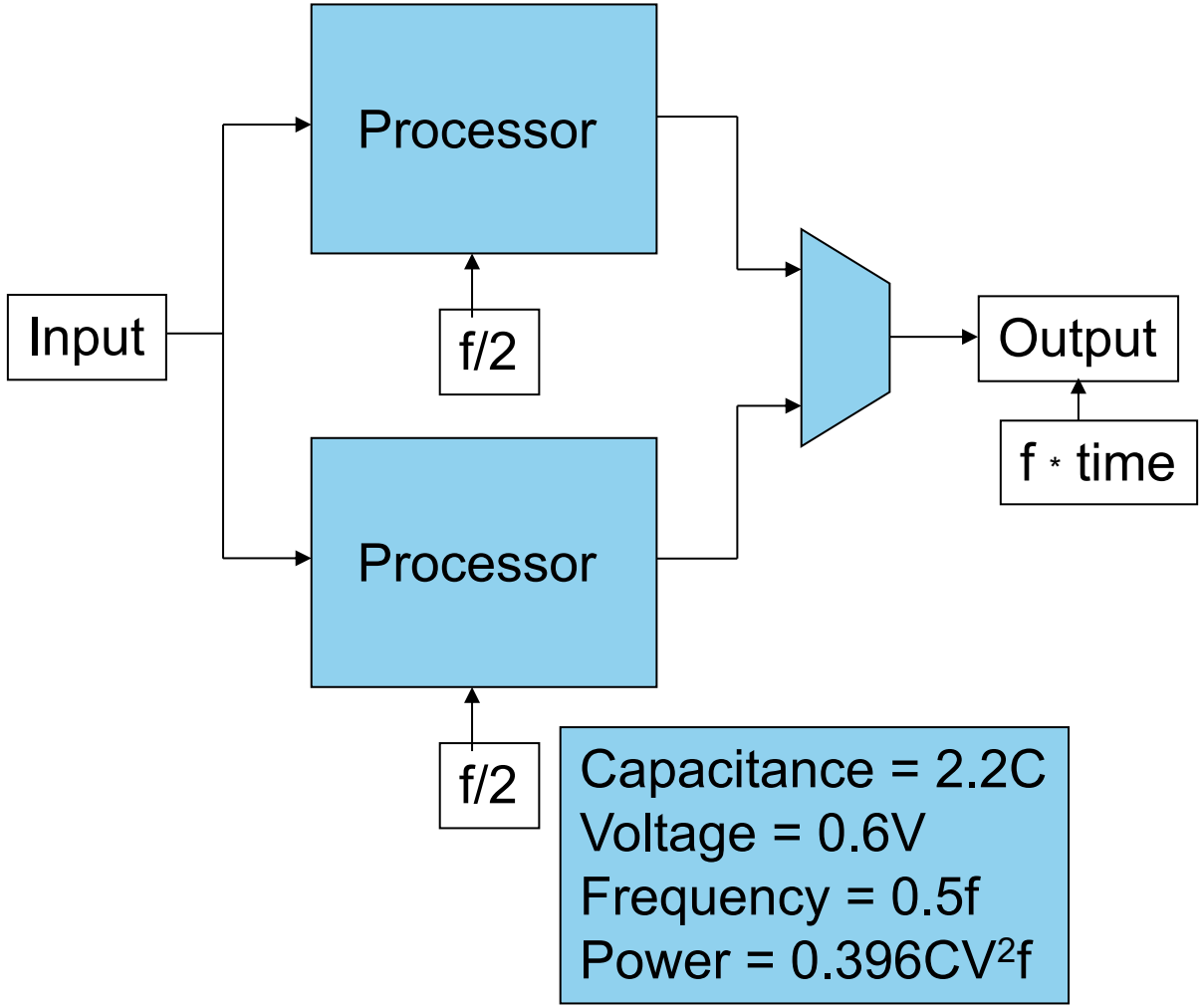
power is work over time … or how many times per second we oscillate the circuit

Power = W* F → Power = CV^2f

... Reduce power by adding cores



Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f

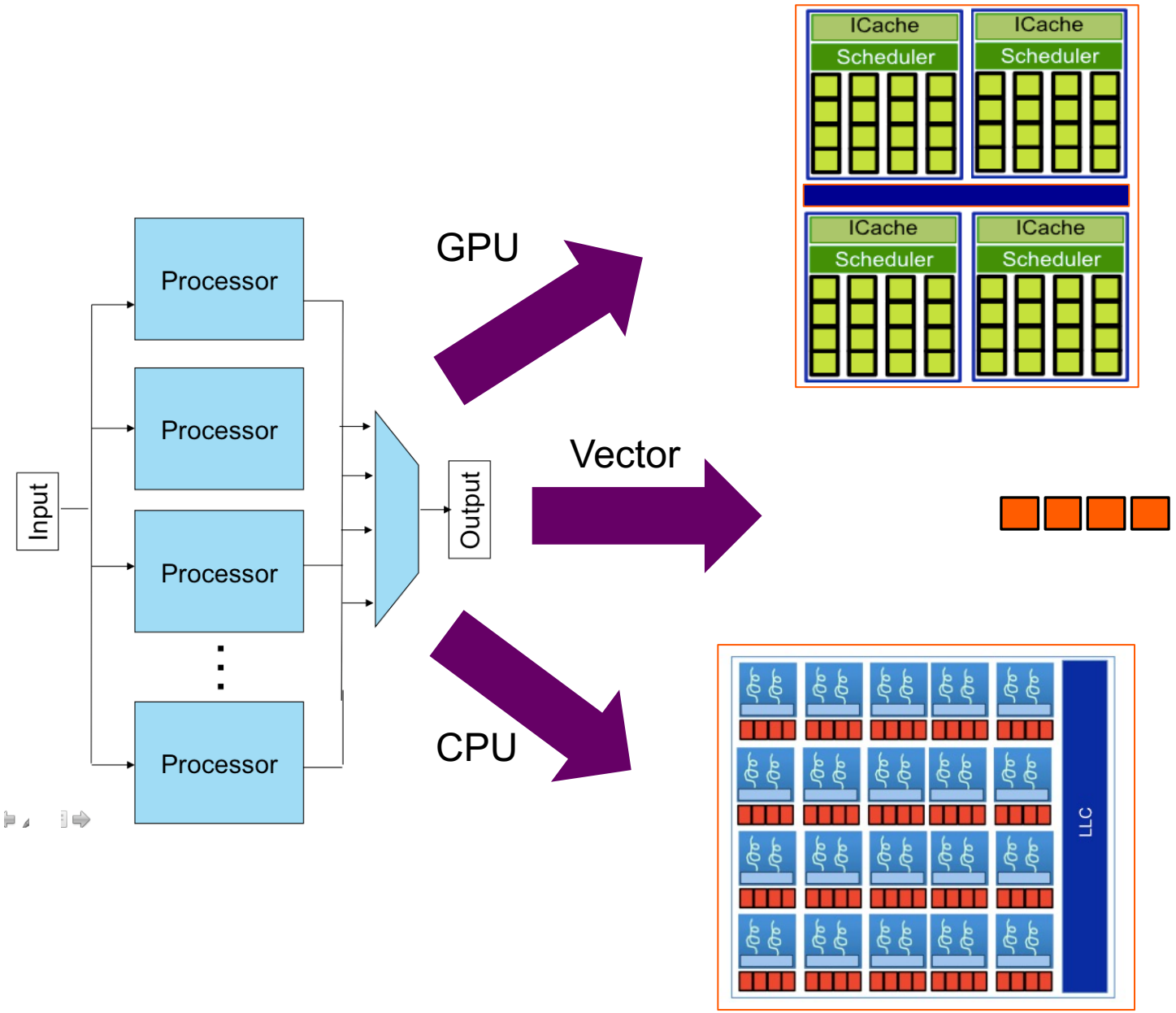


Capacitance = 2.2C
Voltage = 0.6V
Frequency = 0.5f
Power = $0.396CV^2f$

Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, vol.14, no.1, pp.12-31, Jan 1995

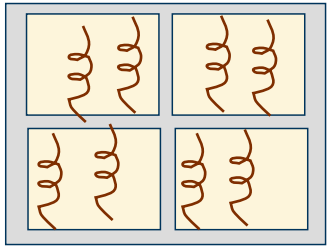
Source: Vishwani Agrawal

Manycore processors: three hardware options



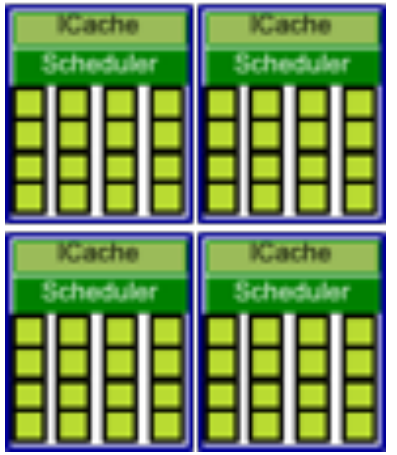
For hardware … parallelism is the path to performance

All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.

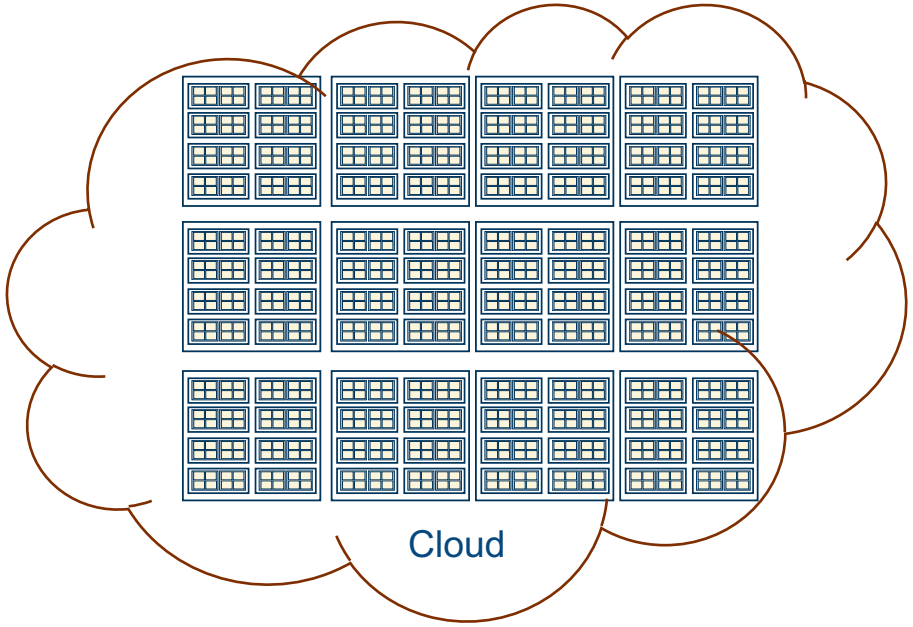


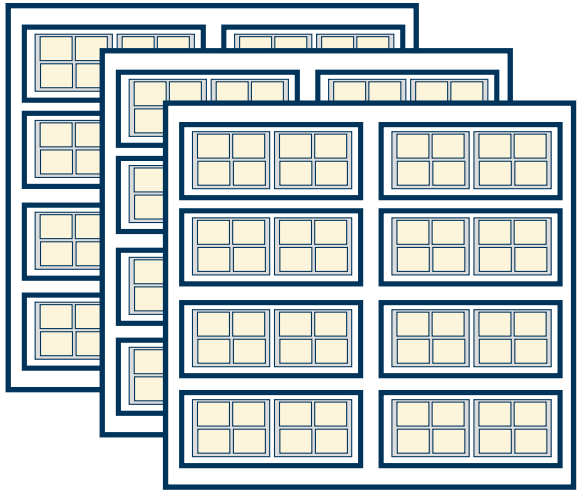
CPU

SIMD/Vector

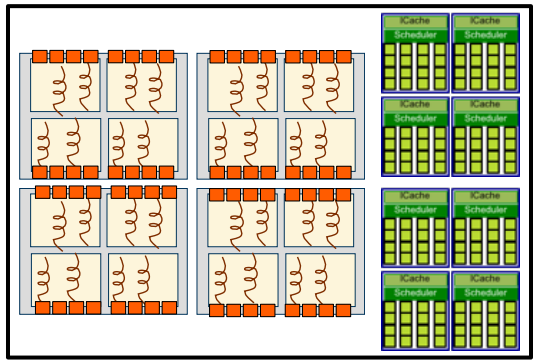


GPU





Cluster



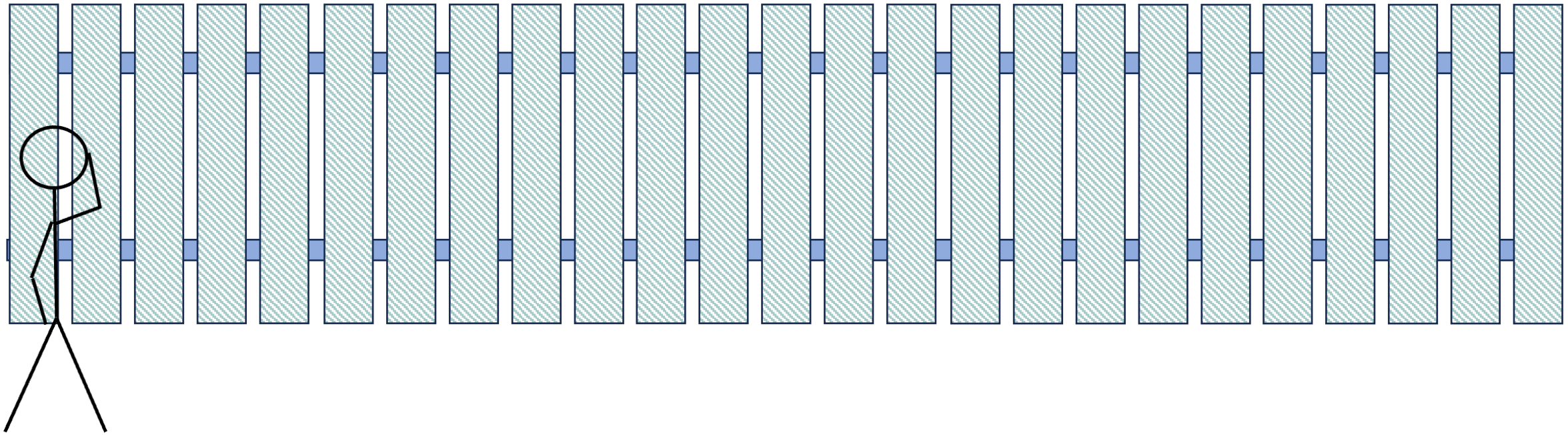
Heterogeneous node

Before we continue …

Let’s discuss the idea of *parallelism and the terminology we use when working with it.*

Painting a fence

- You must paint a fence. It has 25 planks.

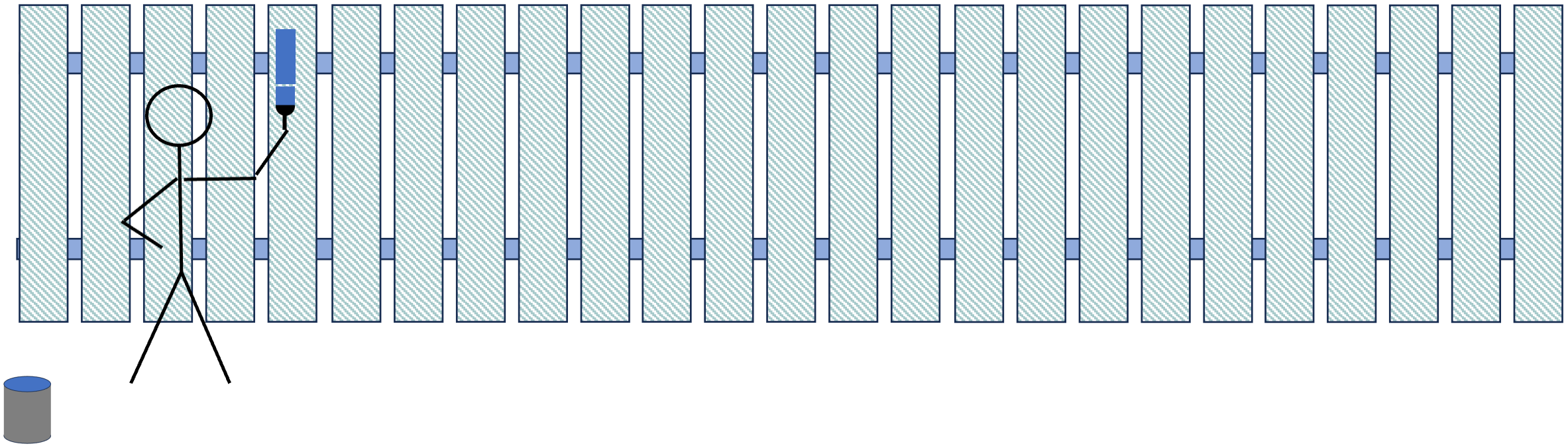


Painting a fence: Doing it as a serial process

- You must paint a fence. It has 25 planks. You are going to do this one plank at a time (**a serial process**)
- How long should it take?

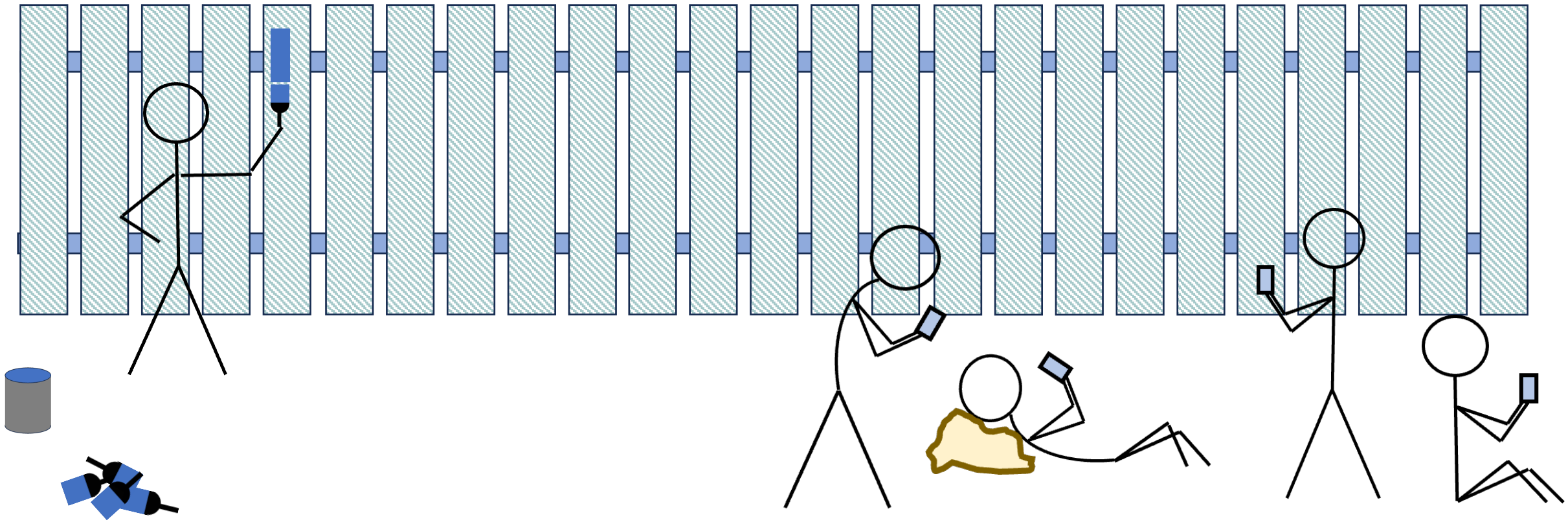
Ideal case

It takes T_p minutes to paint a plank. Therefore it takes $25*T_p$ minutes to paint the entire fence.



Painting a fence: Getting help to finish the job in less time

- You must paint a fence. It has 25 planks.
- You have five brushes, one can of paint, and four friends. How long should the job take with the five of you working **in parallel**?

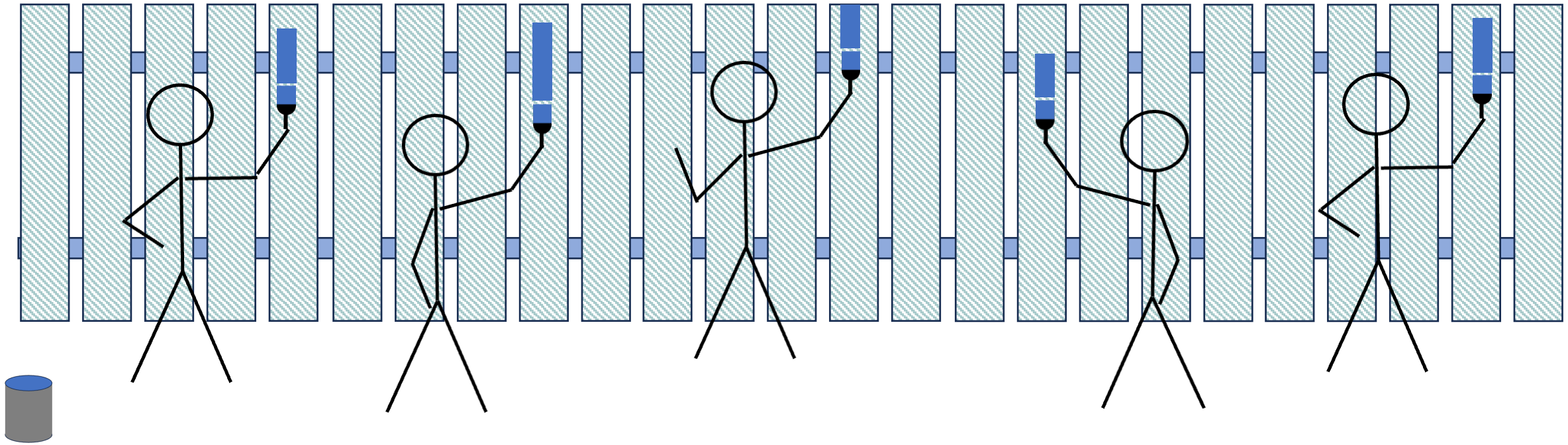


Painting a fence: Getting help to finish the job in less time

- You must paint a fence. It has 25 planks.
- You have five brushes, one can of paint, and four friends. How long should the job take?

Ideal case

It takes $25*T_p$ minutes to paint the fence on your own ("**in serial**"). With N = 5 people each taking an equal chunk of the fence, then each person paints 25/N planks. If they do it all at the same time (that is, "**in parallel**"), the fence will be done in $(25*T_p)/N = 5*Tp$ minutes



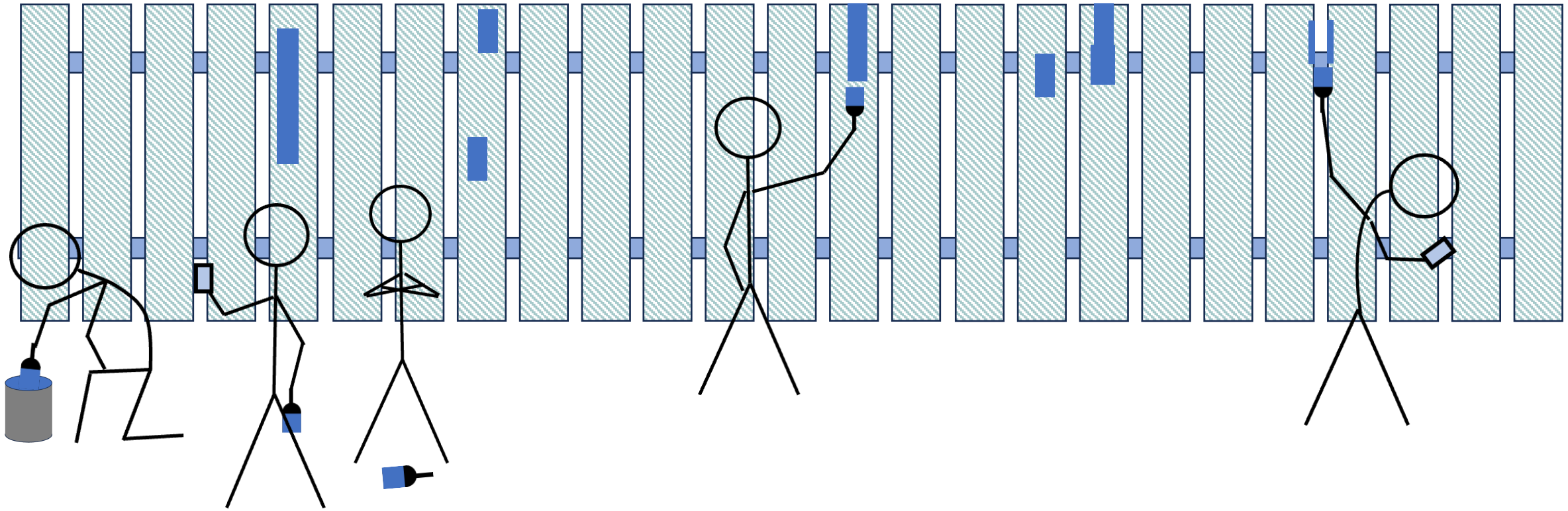
Speedup: How many times faster does your job complete when N people work in parallel? Ideally, Speedup = N.

Painting a fence: Sometimes, Reality Sucks

- You must paint a fence. It has 25 planks.
- You have five brushes, one can of paint, and four friends. How long should the job take?

Reality

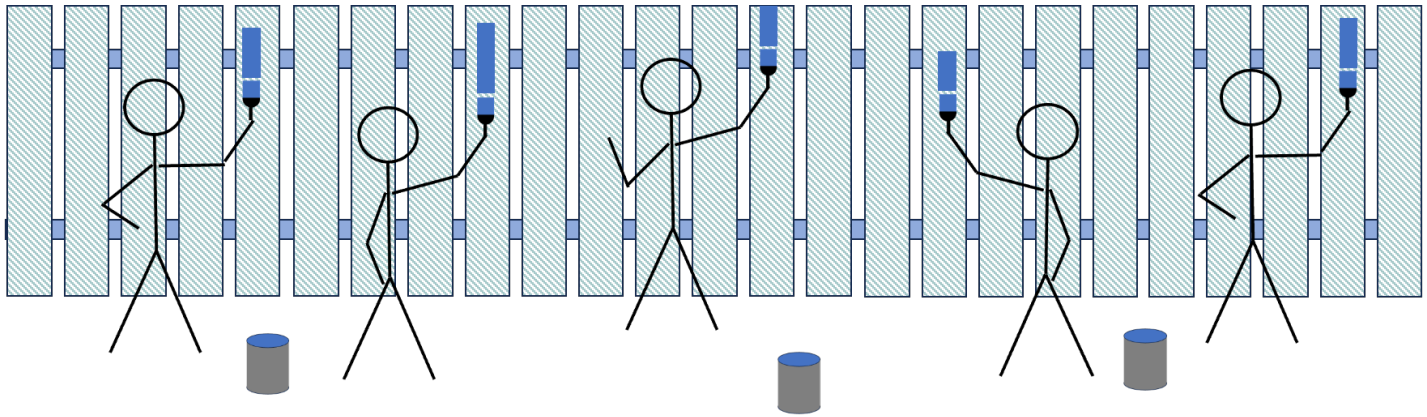
Not everyone works equally hard. More importantly with only one can of paint (a **shared resource**) people waste time waiting for their turn to dip their brush in the paint. These and other issues mean that you almost never achieve the ideal case.



Summary: the concept of parallelism

- This sequence of definitions explains the term “parallelism”
  - **Agent**: A person, process, thread, or other “unit of execution” that can work on a task.
  - **Problem Decomposition**: Break the problem into a collection of distinct tasks that are mostly (if not completely) independent.
  - **Serial**: When a single agent carries out a problem’s tasks one after the other.
  - **Parallel**: When the tasks execute and make forward progress at the same time.
  - **Parallelism**: the features of problem and its solution that support parallel execution.

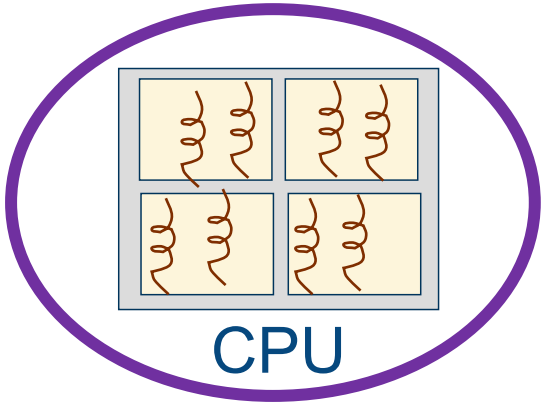
- Assume you have multiple agents to carry out a set of tasks. You are not done until the last agent is done. It never goes as well as you hope.
  - Making the set of tasks (the work) for each agent **balanced** so they all finish at the same time is hard (**load balancing**).
  - Agents share resources (such as paint) and often waste time waiting for their turn for the shared resource (**contention**).
  - Coordinating the work of the multiple agents is extra work you wouldn’t have if you did the job in serial. This is called **parallel overhead**.
  - Recasting the problem into a collection of distinct and largely independent subproblems (tasks) can be difficult
  - There is almost always a small fraction of the work that cannot be done in parallel (**serial fraction**).
  - The serial fraction limits the number of agents that can productively help you complete the job

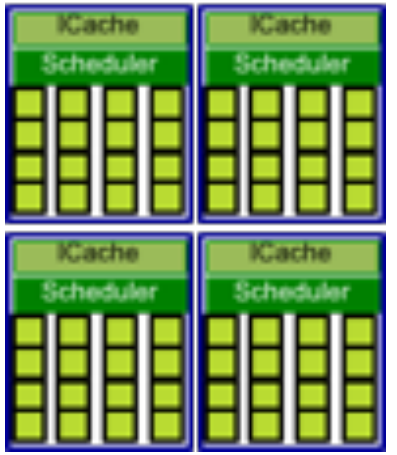


Lets consider parallelism across the major classes of parallel system

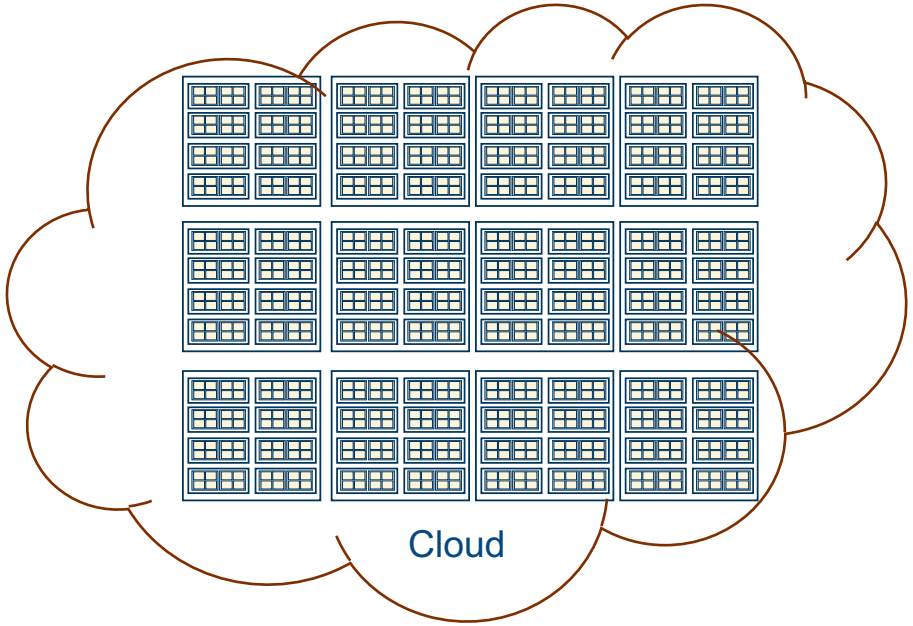
For hardware … parallelism is the path to performance

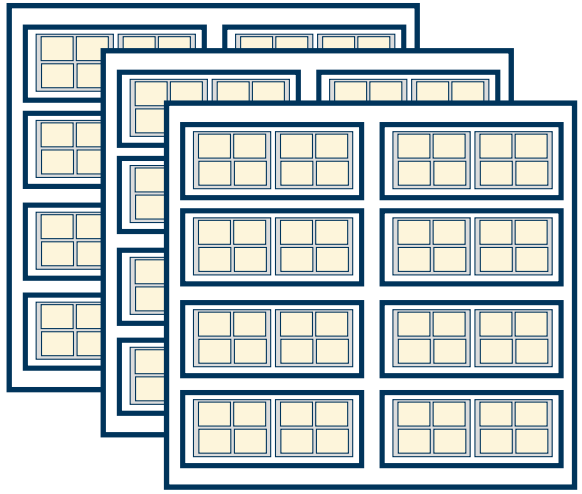
All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.



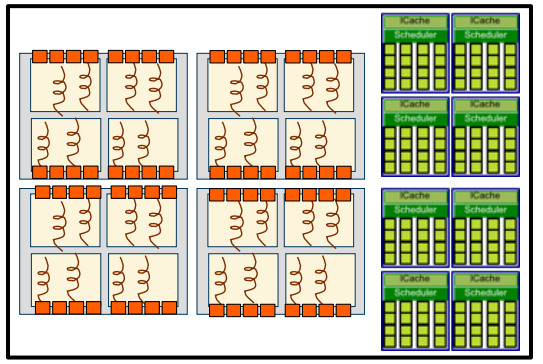


GPU





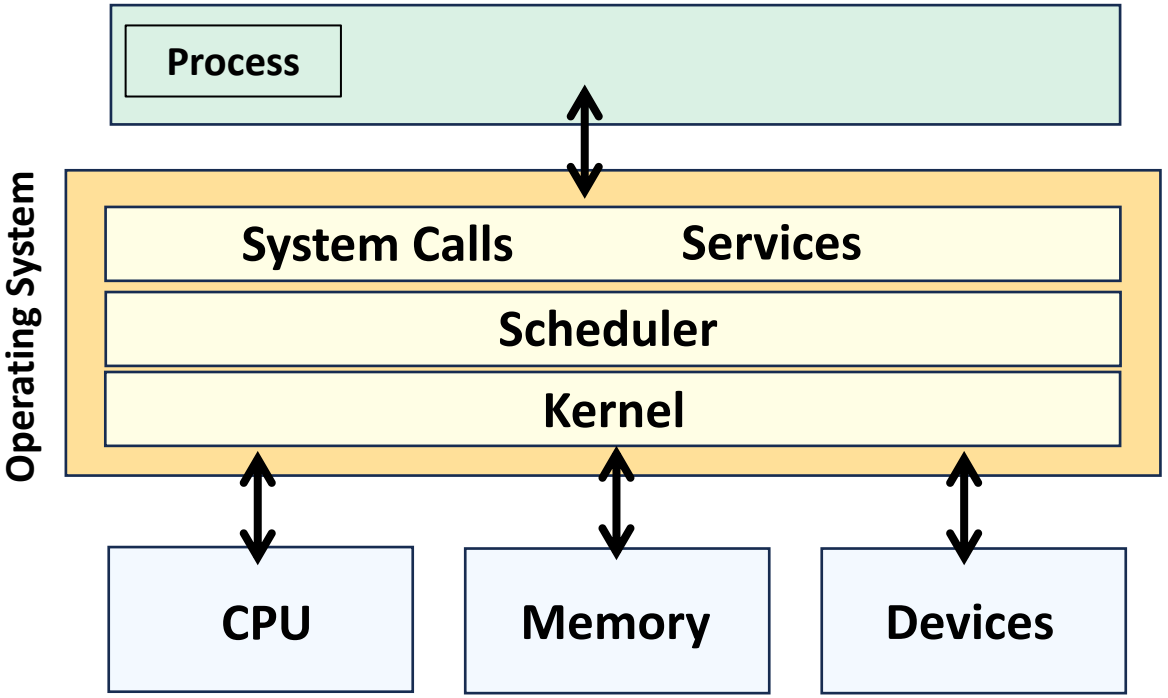
Cluster



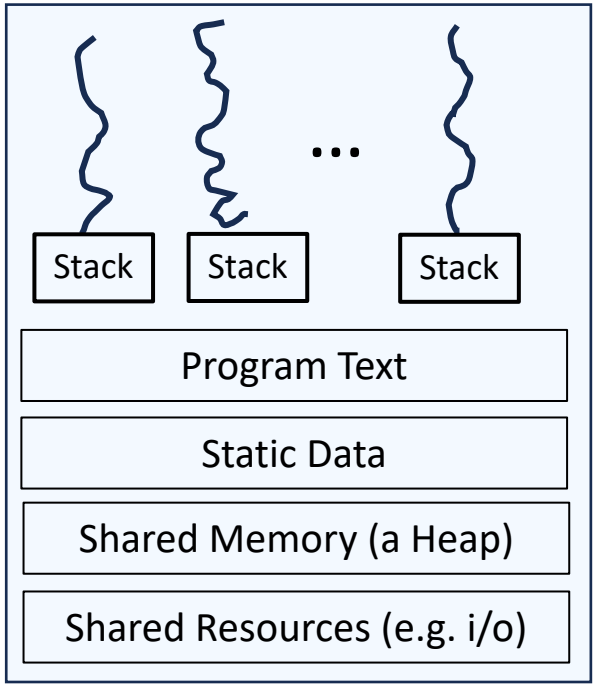
Heterogeneous node

The operating system running on a CPU

- The Operating System (OS) is software the manages the computer hardware.
- It consists of a low-level kernel and a collection of services running on top of the kernel to support the needs of system users.
- The kernel provides security guarantees and isolation between running programs (processes).



- A process is an instance of an executing program.



One or more threads running the same program text

Each thread has private memory (a stack)

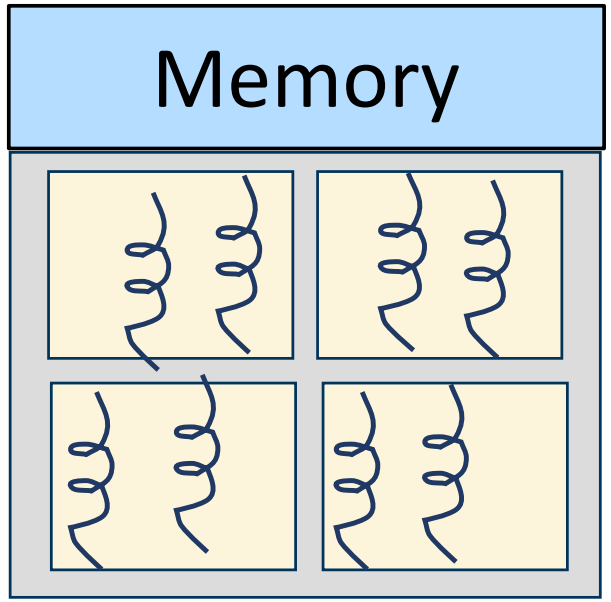
Static Memory fixed at compile time

Shared Memory and shared resources are available to all threads

- Modern operating systems support multi-tasking. This means that multiple processes are active at one time with a scheduler (part of the OS) quickly switching (a context switch) between processes.
- For the user, this creates the illusion that all of the processes are running at the same time (more on this later).

CPU parallelism: Multicore CPUs

- A modern CPU optimized for performance will have multiple cores sharing a single memory hierarchy.
- The memory appears as a single address space.
- An instance of a program is a process.
- The process has a range of available memory addresses, system resources, and one or more threads.
- Parallelism is managed by the programmer as multiple threads mapped to the various cores.
- When every core is treated the same by the operating system (OS) and has an equal cost function to any location in memory, we call this a **symmetric multiprocessor** or **SMP**.

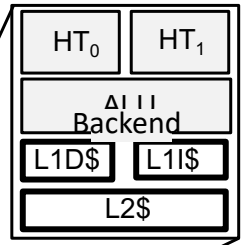


A four core CPU running a process with 8 threads mapped as 2 threads per core

A Harsh dose of reality: System memories are non-uniform

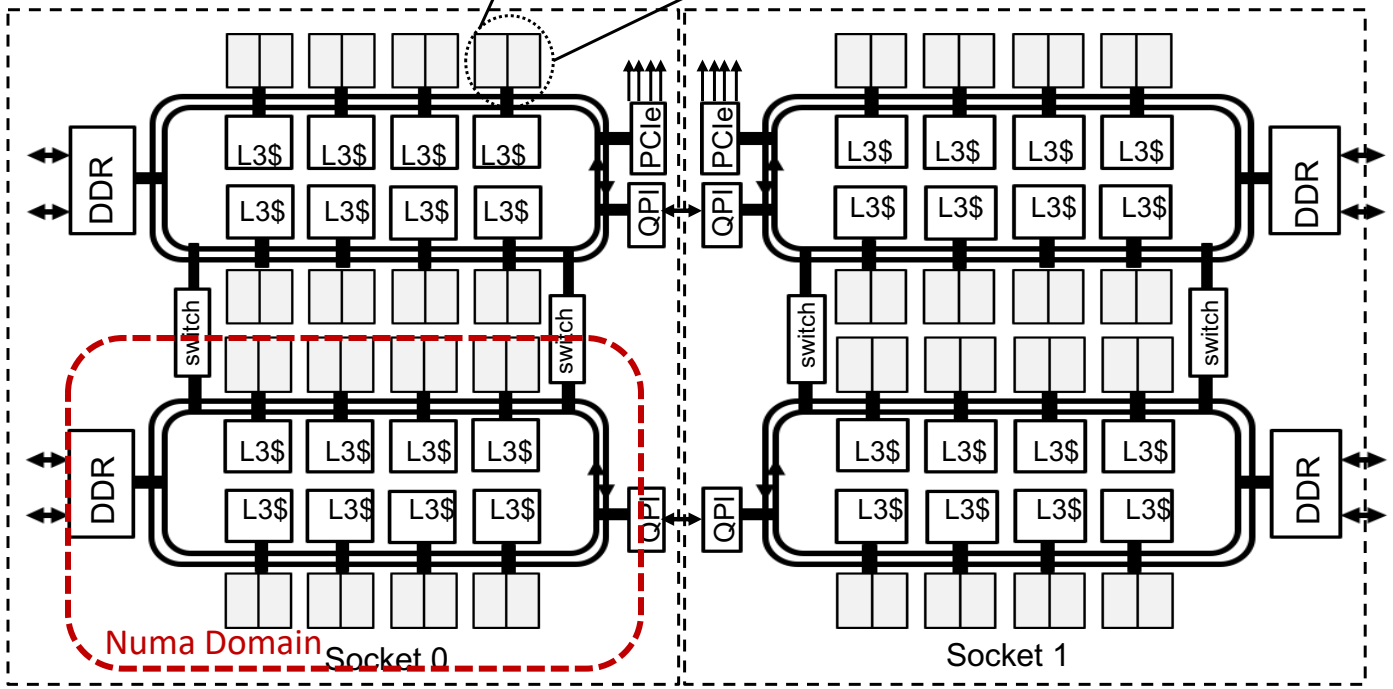
4 blocks of 8 core units connected by an on-chip-network with a DDR memory controller.

Each block is a NUMA domain … memory access from a core to its "own" DDR is less expensive.



2 Hardware threads (HT) per core
Intel® AVX2 (256 bit Vector unit)
L1$ instruction and data: 32 KB
Unified L2$ 256 KB

40 MB shared L3$



DDR: Double Data Rate memory controller

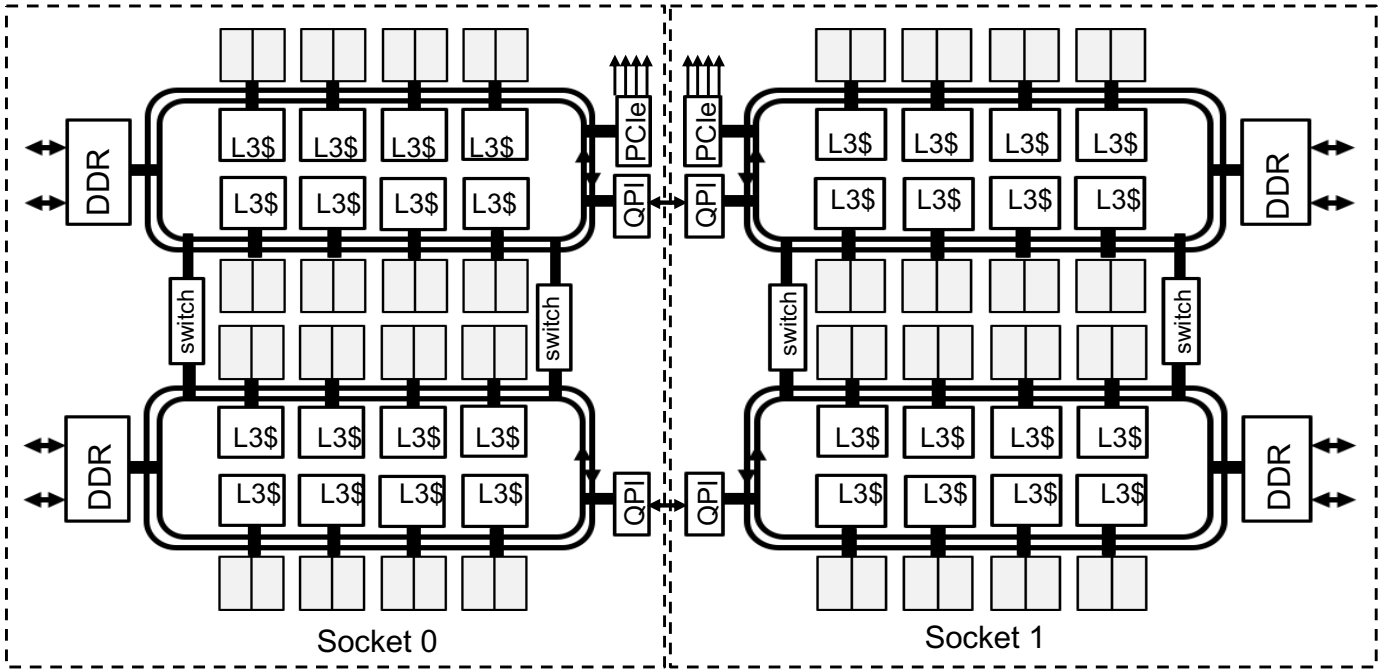
PCIe is the connection from the CPU to other devices in a node.

QPI: Quick Path Interconnect. A coherent interconnect between CPUs. Makes it easy to build multi-CPU nodes.

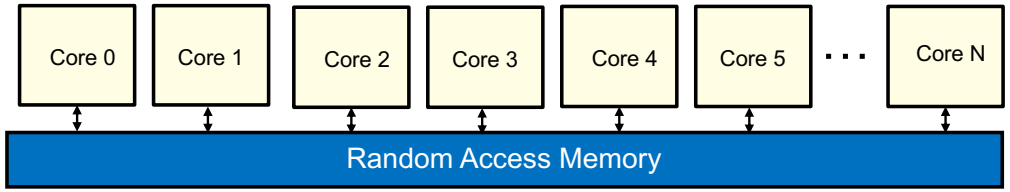
A single dual processor node for the Cori system at NERSC: 2 Intel® Xeon™ E5-2698 v3 CPUs at 2.3 GHz,
2 16 GB DIMMs per **DDR memory controller**, 16 cores per CPU. 2 CPUs connected by a high-speed interconnect (QPI)

All systems today are nonuniform memory architectures (NUMA)

Given all this complexity in real hardware ….



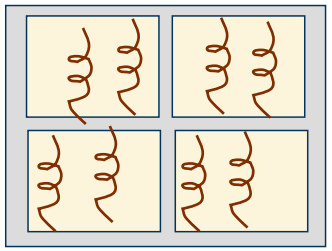
It’s amazing our SMP model works at all ….



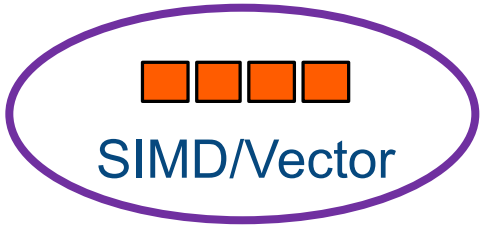
Dual Socket node with Intel® Xeon™ E5-2698v3 CPUs

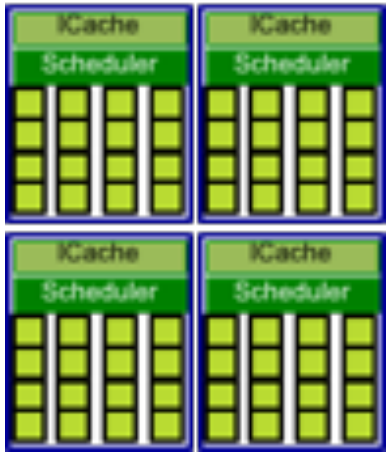
For hardware … parallelism is the path to performance

All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.

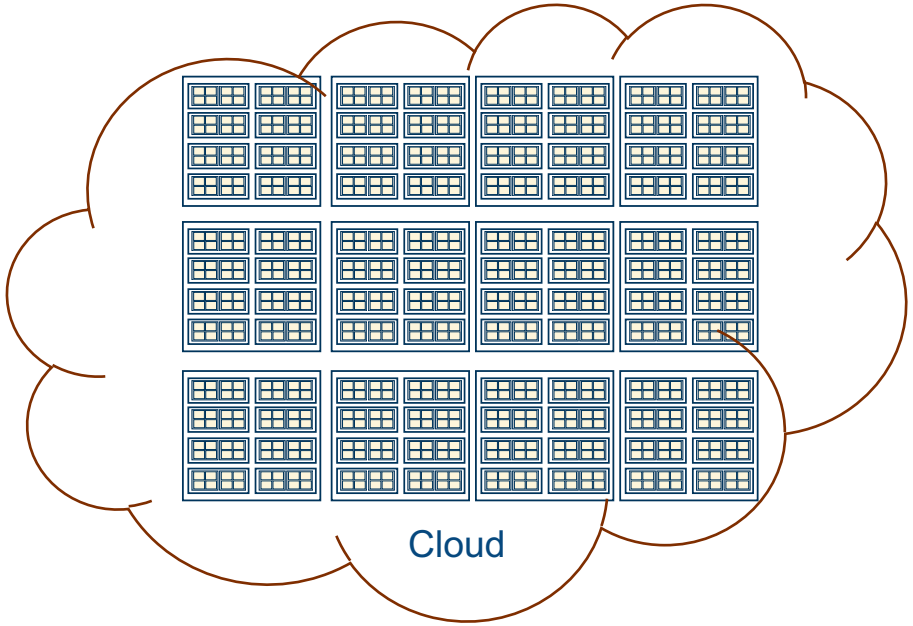


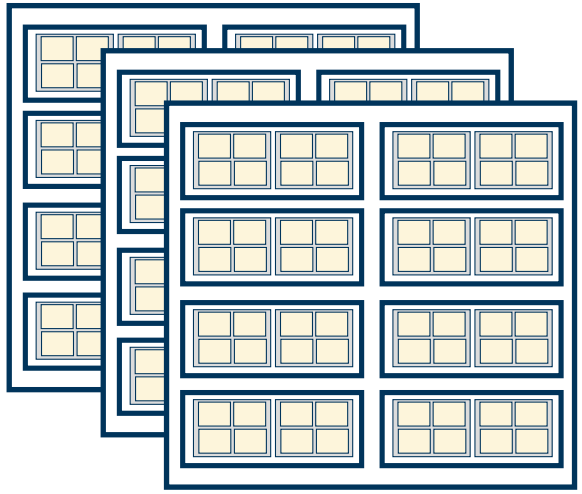
CPU



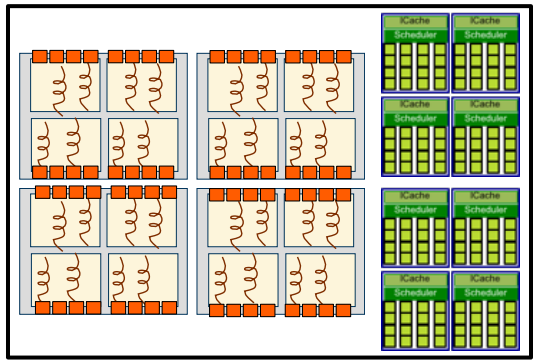


GPU





Cluster

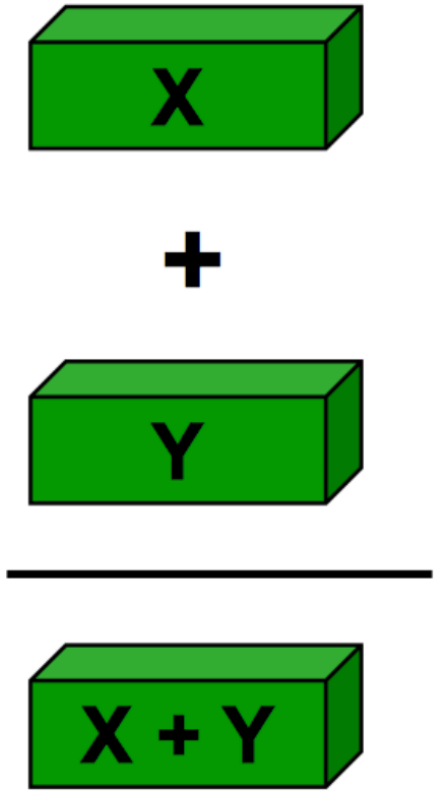


Heterogeneous node

Vector (SIMD) Computing

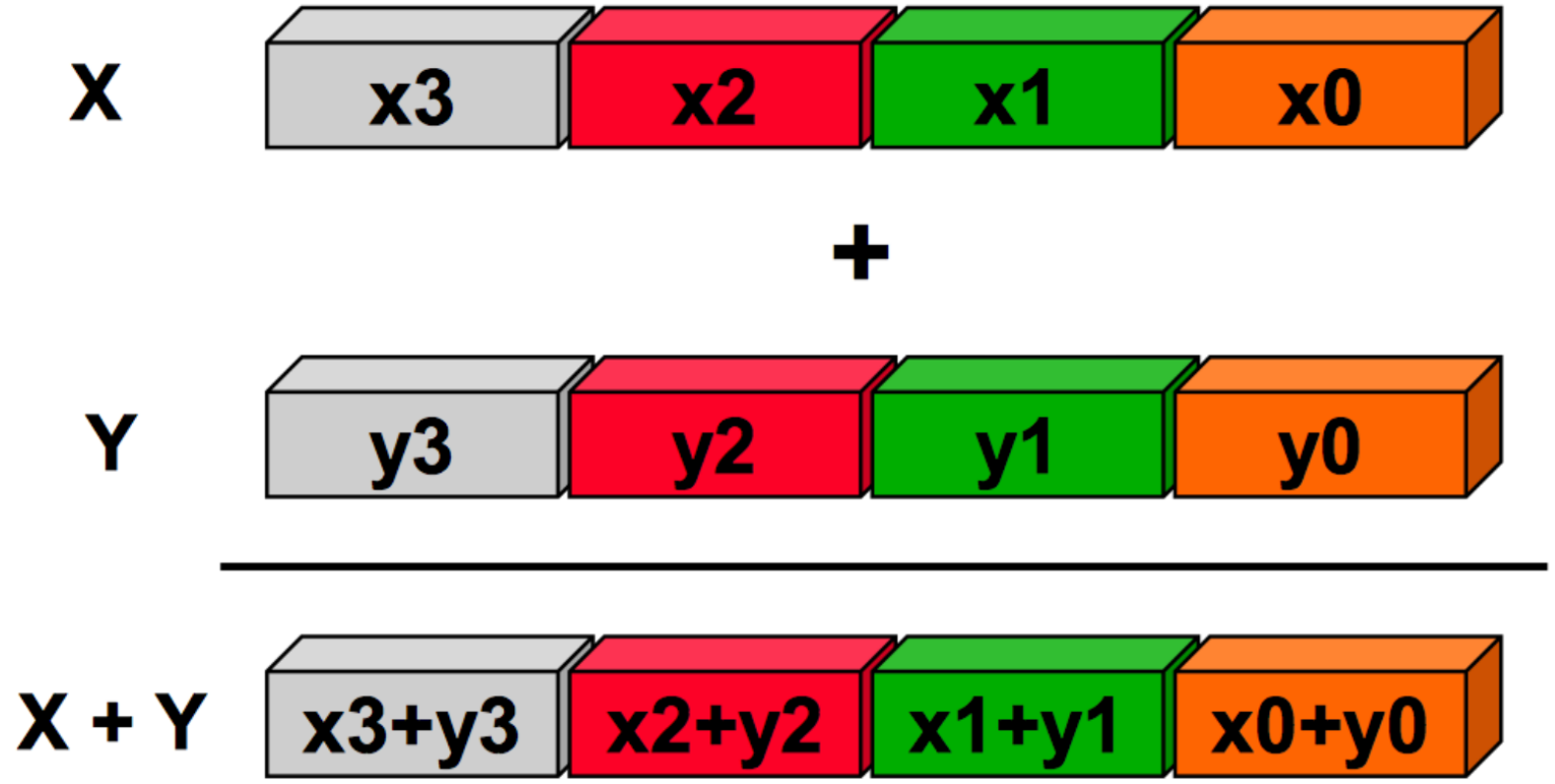
Scalar computing

- One op produces one value



Vector computing .. Pack multiple values into vector registers

- One op produces multiple values



Hardware architects love vector computing, since they permit space- and energy- efficient parallel implementations.

SIMD: Single Instruction Multiple Data … another name for vector coputing

Source: Alex Klimovitski & Dean Macri, Intel Corporation

Packing numbers into fixed width vector registers

Example: 128 bit SSE (from Intel) and Neon (from ARM)

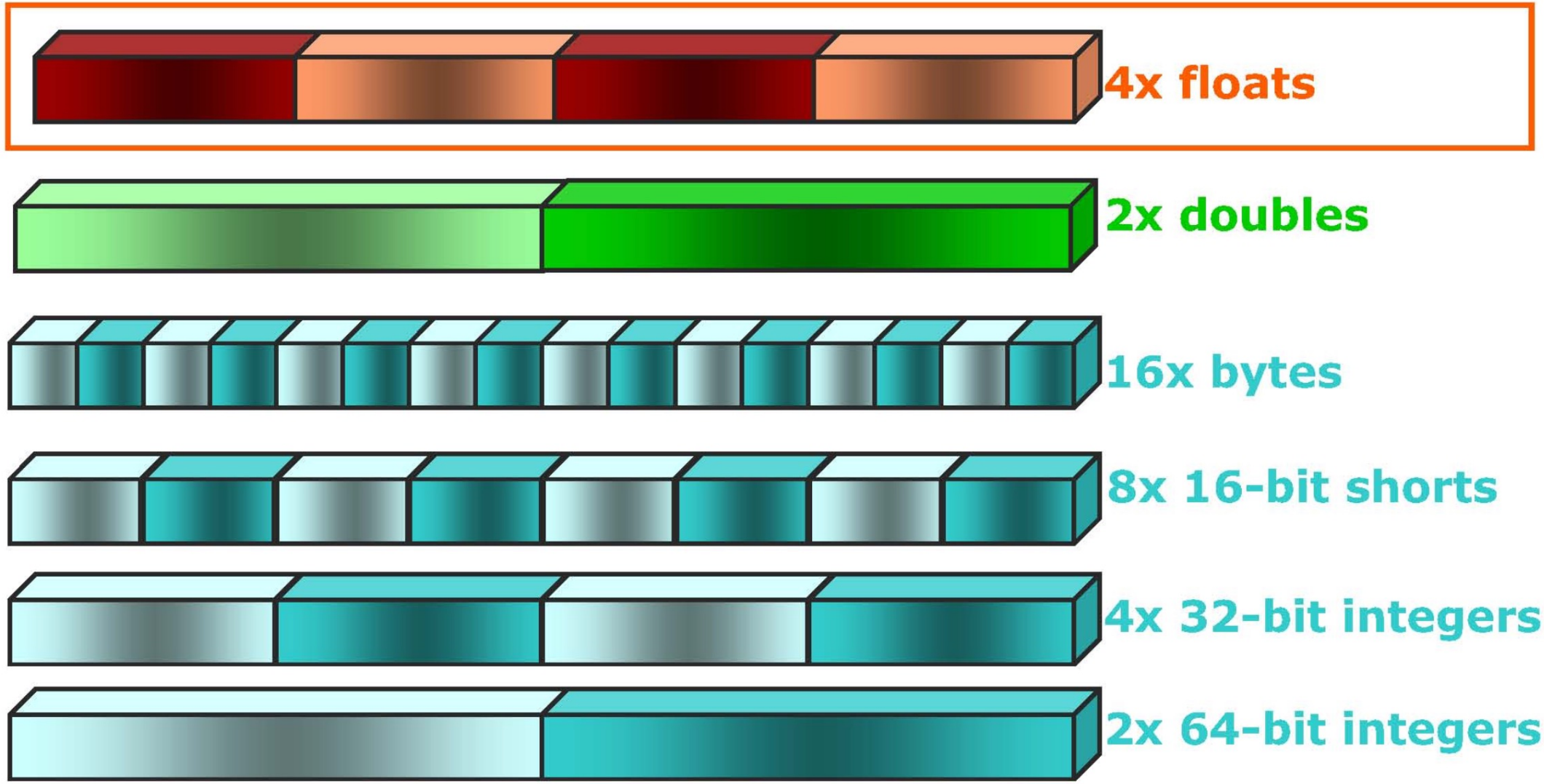


Image Source: Intel

The importance of using the vector units on your CPU

Writing code to fully utilize vector units can be difficult … most applications make poor use of the vector units.

In one corporate sponsored study* of leading applications in engineering and scientific computing, of the instructions retired, on average only 4% were vector instructions.

How much of the available performance are you wasting if you ignore vectorization?

256 bit SIMD (SSE)

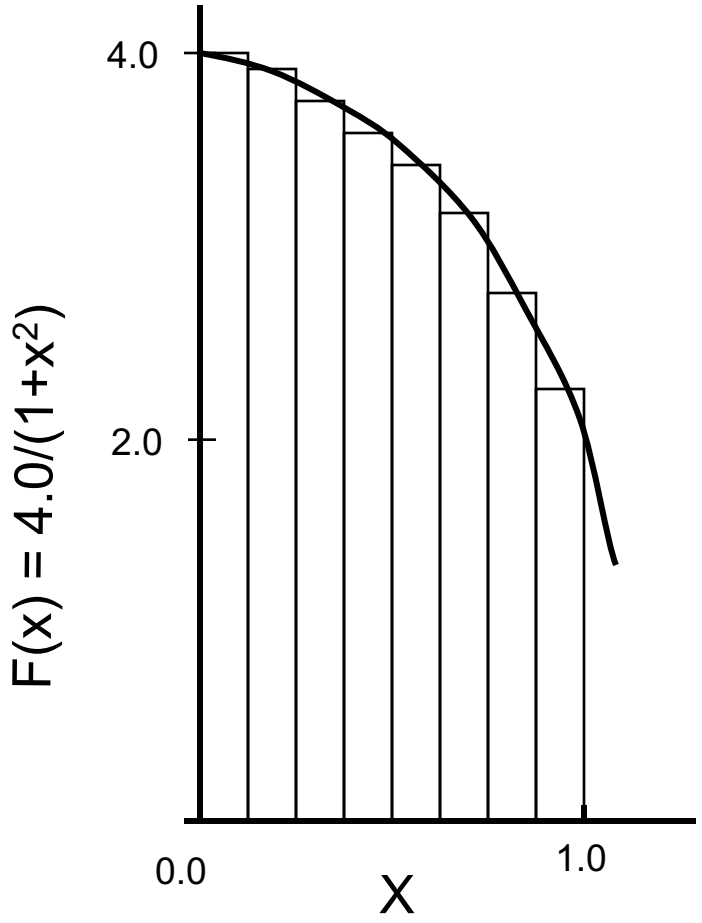
1024 bit SIMD (Xeon™ PHI)

It depends on the width of the vector registers

* I do not have permission to share any details on this study … but the result speaks for itself

Let’s consider a specific example to understand how vectorization works in software

Example Problem: Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial PI program

Literals as double (no-vec), 0.012 secs
Literals as Float (no-vec), 0.0042 secs

```
static long num_steps = 100000;
float step;
int main ()
{          int i;      float x, pi, sum = 0.0;

           step = 1.0/(float) num_steps;

           for (i=0;i< num_steps; i++){
                      x = (i+0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
           }
           pi = step * sum;
}
```

By default, literals (such as 0.5) are double precision. So, all arithmetic involving such literals is done in double precision which reduces how many values can be packed into vector registers).

To avoid this problem, you must indicate the literal is a float by adding and 'f' at the end of the number (e.g. 0.5f).

Normally, I'd use double types throughout to minimize roundoff errors especially on the accumulation into sum. But to maximize impact of vectorization for these exercise, we'll use float types.

Explicit vectorization of our Pi Program: Step 1 ... Unroll the loop

- We need one interation to fit in the vector unit

- What is the width of your vector unit?
- We'll use SSE which is 128 bits wide.
- A float in C is 32 bits wide … 4 floats fits in 128 bits

So, unroll the loop by four

```
float pi_unroll(int  num_steps)
{
   float step, x0, x1, x2, x3, pi, sum = 0.0;

   step = 1.0f/(float) num_steps;

   for (int i=1;i<= num_steps; i=i+4){      //unroll by 4, assume num_steps%4 = 0
      x0 = (i-0.5f)*step;
      x1 = (i+0.5f)*step;
      x2 = (i+1.5f)*step;
      x3 = (i+2.5f)*step;
      sum += 4.0f*(1.0f/(1.0f+x0*x0) + 1.0f/(1.0f+x1*x1)  + 1.0f/(1.0f+x2*x2)  + 1.0f/(1.0f+x3*x3));
   }

     pi = step * sum;
     return pi;
}
```

Explicit SSE* vectorization of our Pi Program: Step 2 ... Add SSE intrinsics

```
#include <immintrin.h>  ← Function prototypes for C functions that map to SSE assembly code
float pi_sse(int  num_steps)
 {
   float scalar_one =1.0, scalar_zero = 0.0,  ival, scalar_four =4.0, step, pi, vsum[4];
   step = 1.0/(float) num_steps;

   __m128 ramp   = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
   __m128 one    = _mm_load1_ps(&scalar_one);
   __m128 four   = _mm_load1_ps(&scalar_four);
   __m128 vstep  = _mm_load1_ps(&step);
   __m128 sum    = _mm_load1_ps(&scalar_zero);
   __m128 xvec;   __m128 denom;  __m128 eye;

  for (int i=0;i< num_steps; i=i+4){  ← Unroll the loop by 4 and assume (num_steps % 4=0)
     ival      = (float)i;
     eye       = _mm_load1_ps(&ival);
     xvec      = _mm_mul_ps(_mm_add_ps(eye,ramp),vstep);
     denom     = _mm_add_ps(_mm_mul_ps(xvec,xvec),one);
     sum       = _mm_add_ps(_mm_div_ps(four,denom),sum);
   }
   _mm_store_ps(&vsum[0],sum);   ←Copy a packed single value in a vector(sum) register into a float array (vsum)
   pi = step * (vsum[0]+vsum[1]+vsum[2]+vsum[3]);
  return pi;
}
```

Load needed constants into vector registers

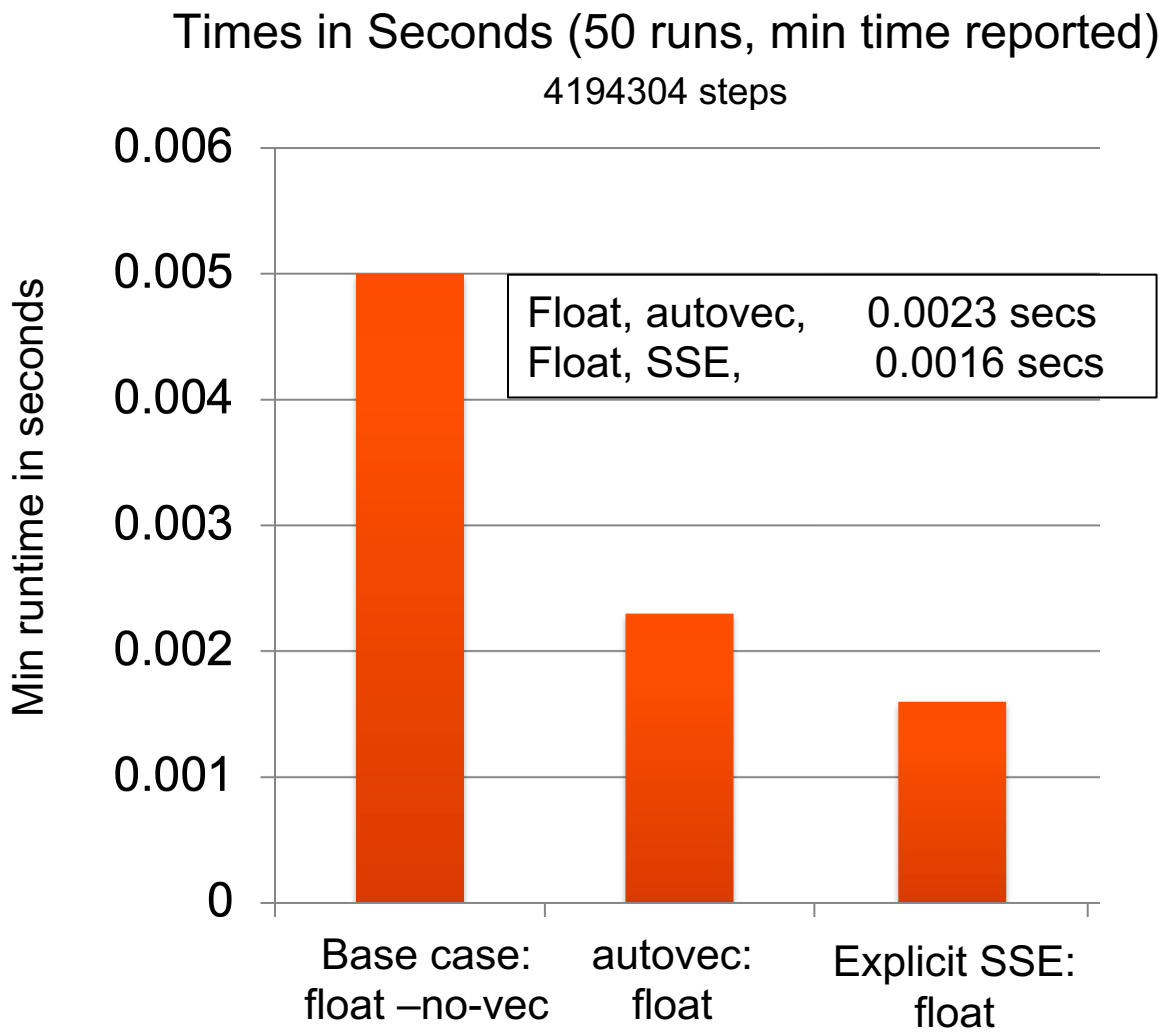
Vector ops on four floats at a time ... i.e., packed single (_ps) values.

The vast majority of programmers never write explicitly vectorized code.

It is important to understand explicit vectorization so you appreciate what the compiler does to vectorize code for you

*SSE is the 128-bit wide vector instruction set for x86

PI program Results:

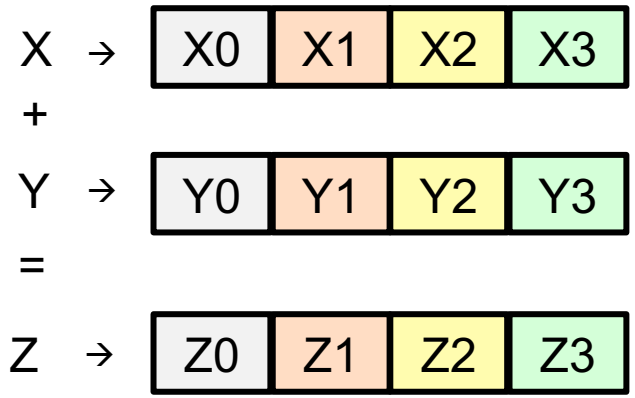


- Intel Core i7, 2.2 Ghz, 8 GM 1600 MHz DDR3, Apple MacBook Air OS X 10.10.5.
- Intel(R) C Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 15.0.3.187 Build 20150408

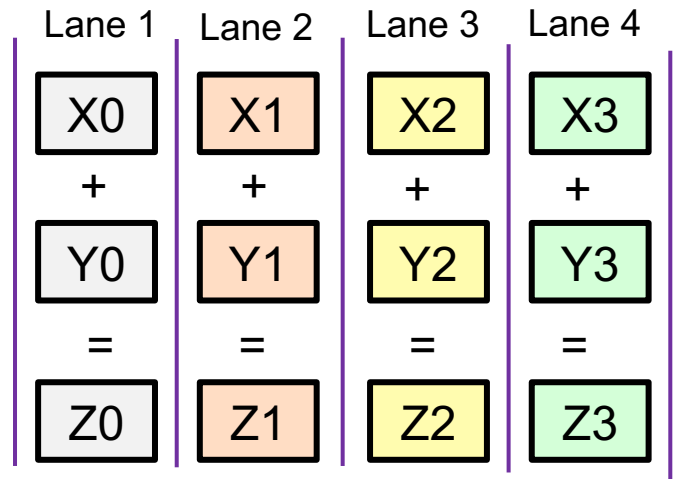
SIMD Lanes

It can help you reason about more complex algorithms or performance issues if you think about execution of instruction on a vector unit in terms of a number of SIMD lanes.

Vector operations acting on vectors



Vector operations as SIMD lanres



How do people actually use vectorization in practice?

Compiler vectorization

- Explicit Vectorization: Vector instructions built around vector registers.
  - It is challenging and to programmers unfamiliar with assembly code, it's a foreign style of programming
  - The code is not portable … for example, moving to Arm CPUs broke all my x86 vector code

Most programmers settle for the vectorization the compiler can generate automatically.

- Compiler Vectorization: Organizes code into blocks, packs variables into vector registers, and uses vector instructions for the operations.
  - Typically loop oriented … unrolling loops to create blocks of instructions (though sometimes can create blocks from non-loop code using SLP ... Superword level parallelization)
  - Vector registers are packed with single data types so compiler must deduce which type to use
  - Dependency analysis for values packed into vector registers must assume worst case scenarios … therefore missing many opportunities for vector parallelism.

```
// Basic block SLP example
void foo ()
{
  unsigned int *pin = &in[0];
  unsigned int *pout = &out[0];

  *pout++ = *pin++;
  *pout++ = *pin++;
  *pout++ = *pin++;
  *pout++ = *pin++;
}
```

Enabling Compiler Vectorization

- Compiler vectorization is enabled with the flag –ftree-vectorize

  % gcc –ftree-vectorize pi.c

- Compiler optimization levels O2 and O3 imply vectorization
  - O2 optimization will optimize code, but it will be conservative in the optimizations to assure the code is correct and does not greatly expand the size of the executable.
  - O3 optimization is aggressive and will take chances with order of instructions and will potentiall greatly increase the size of the executable.

  % gcc –O3 pi.c

- Other compiler flags to consider (for gnu compilers)

  -Ofast relax arithmetic rules and freely reorder operations to maximize performance

  -finfo-opt-vec generate information on loops that were or were-not vectorized (doesn't work on Arm)

Vectorization with O2 was added to gcc with gcc version 12

Helping the compiler vectorize

- When in doubt, the compiler will do the safe thing and not vectorize your code

- Knowing how explicit vectorization works, it's clear the compiler needs:
  - Countable innermost loops

```
// Countable loop: .. But maybe aliasing?
while (--n > 0 L)  *q++ = *p++;
```

```
// Uncountable loop:
while (*p != NULL)  *q++ = *p++;
```

  - Avoid aliasing problems

```
// Tell compiler to ignore potential aliasing
#pragma GCC ivdep
while (--n > 0 L)  *q++ = *p++;
```

  - Types must be consistent

```
float x, step, sum;
for (i=0;i< num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
}
```

The literals 0.5, 4.0 and 1.0 are double by default. So, vectorization over double types even though varialbes are float

Optimizing vector code

- Occupancy
  - You need enough vectorizable work to keep all the SIMD lanes fully occupied

```
short a[3], b[3], c[3];
for(i=0; i<3;i++) a[i]=b[i]/c[i];
```

- Converged execution flow
  - Conditional logic prevents vectorization or causes subset of SIMD lanes to pause/no-op

```
for(i=0; i<N;i++) if(c[i]!=0.0) a[i]=b[i]/c[i];
```

- Memory coalescence
  - Stride one memory access across lanes is best

```
for(i=0; i<N;i++) a[i]=b[i]+c[i];
```

  - Fixed strides may work (depending the quality of the vectorizer)

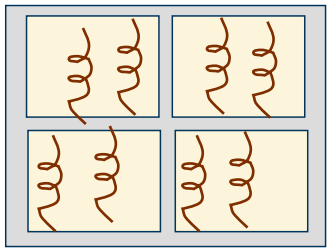
```
for(i=0; i<N;i++) a[i]=b[i+2]+c[i+3];
```

  - Irregular meory access patterns break vectorization.

```
for(i=0; i<N;i++) a[i]+=b[c[i]];
```

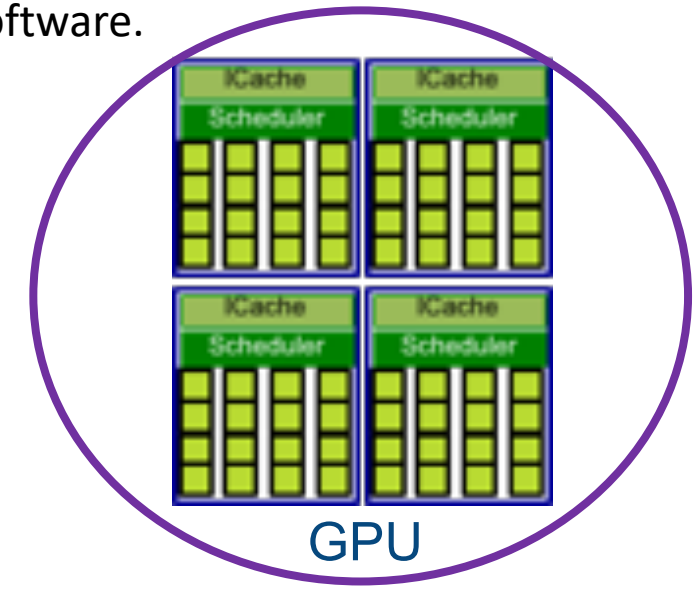
For hardware … parallelism is the path to performance

All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.

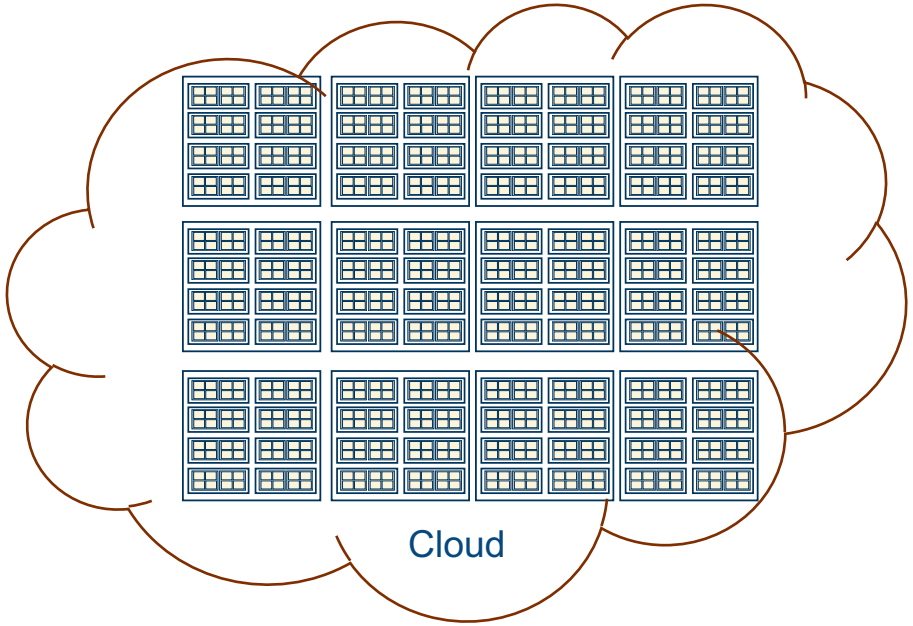


CPU

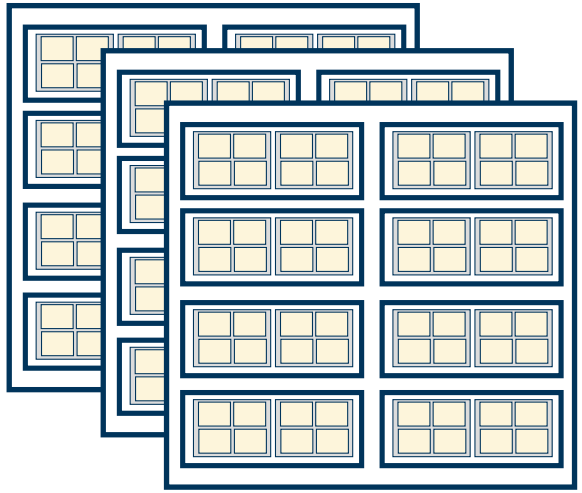
SIMD/Vector



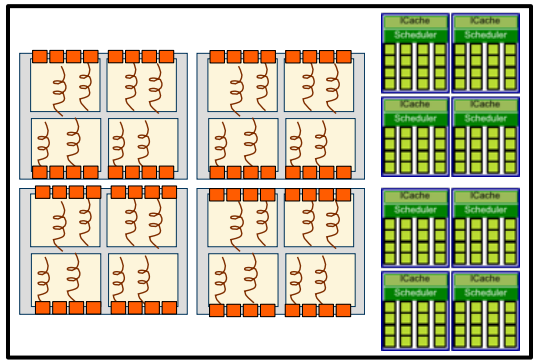
GPU



Cloud



Cluster

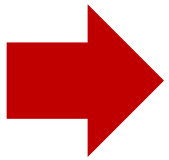


Heterogeneous node

The "BIG idea" Behind GPU programming

Traditional Loop based vector addition (vadd)

```
int main() {
   int N = . . . ;
   float *a, *b, *c;

   a* =(float *) malloc(N * sizeof(float));

   // ... allocate other arrays (b and c)
   // and fill with data

   for (int i=0;i<N; i++)
      c[i] = a[i] + b[i];

}
```



Data Parallel vadd with CUDA

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

Assume a GPU with unified shared memory … allocate on host, visible on device too

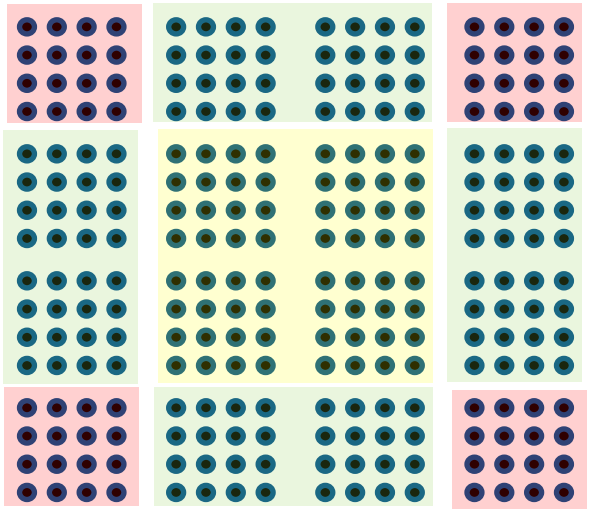
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

1. Turn kernel code into a scalar work-item

```
// Compute sum of order-N matrices: C = A + B
void __global__
matAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j<N) c[i][j] == a[i][j] + b[i][j];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // define threadBlocks and the Grid
   dim3 dimBlock(4,4);
   dim3 dimGrid(4,4);

  // Launch kernel on Grid
    matAdd <<< digGrid,dimBlock>>> (a, b, c, N);
}
```

This is CUDA code

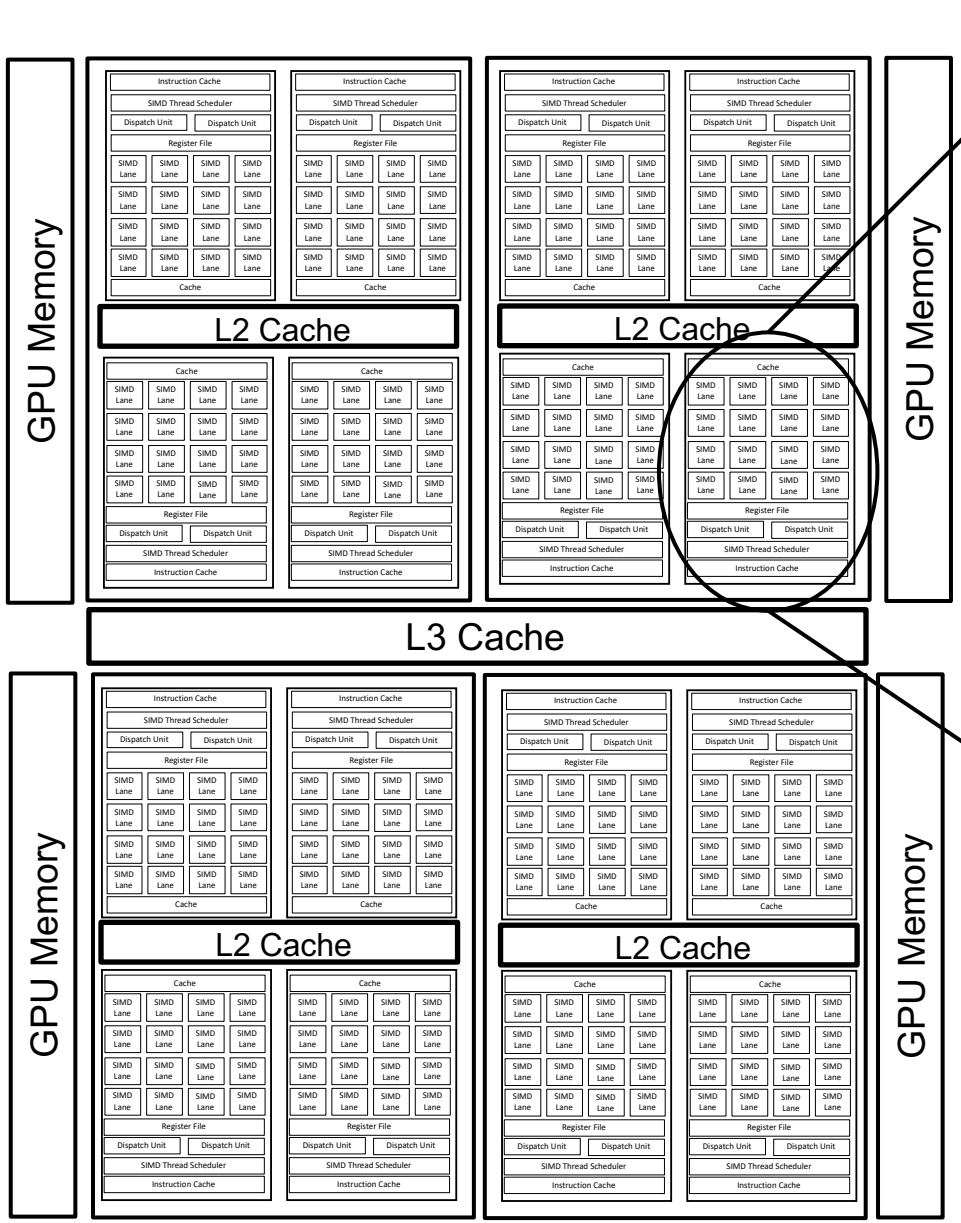
2. Map work-items onto an N dim index space.

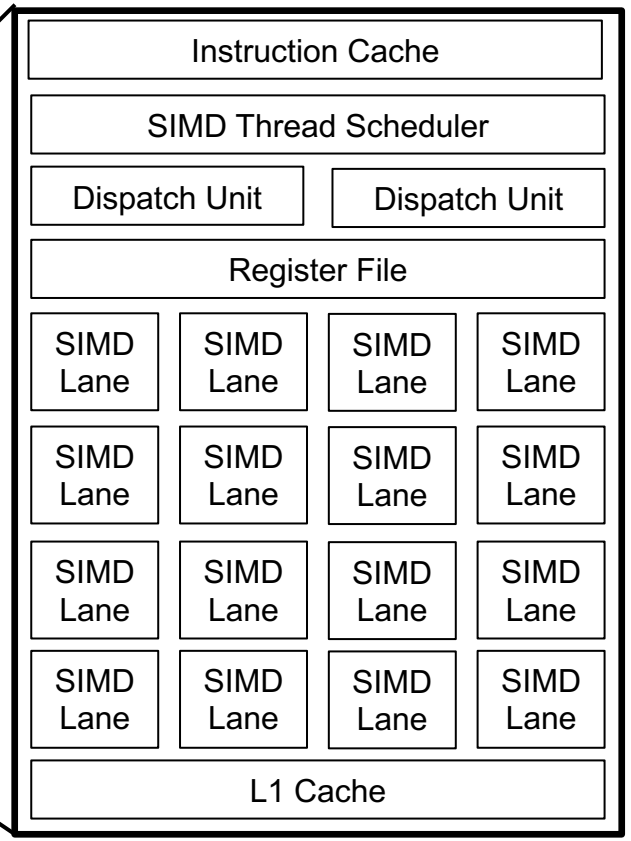


3. Map data structures onto the same index space

4. Run on hardware designed around the same SIMT execution model

A Generic GPU (following Hennessey and Patterson)



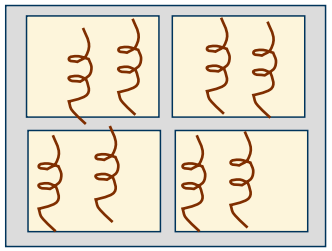


A multithreaded SIMD processor

A Quantitative Approach to Computer Architecture by Hennessey and Patterson … the classic text book we all use to learn computer architecture

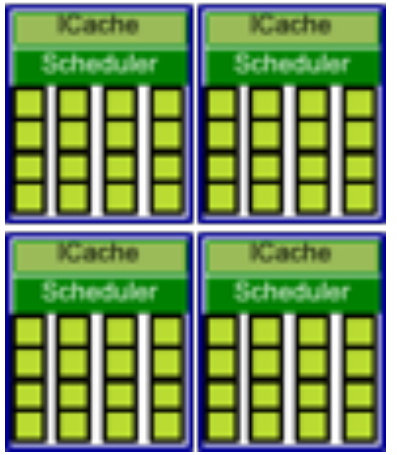
For hardware … parallelism is the path to performance

All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.

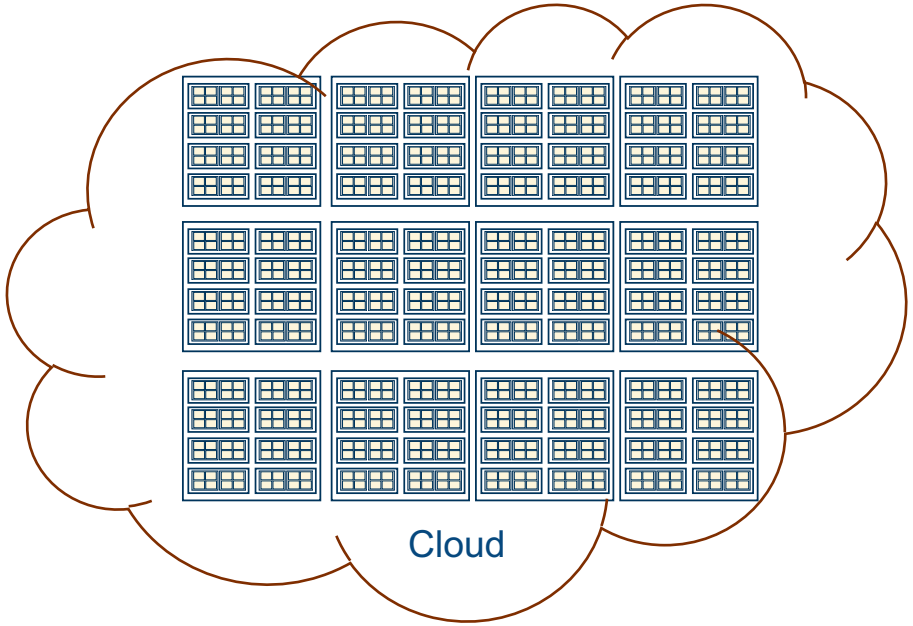


CPU

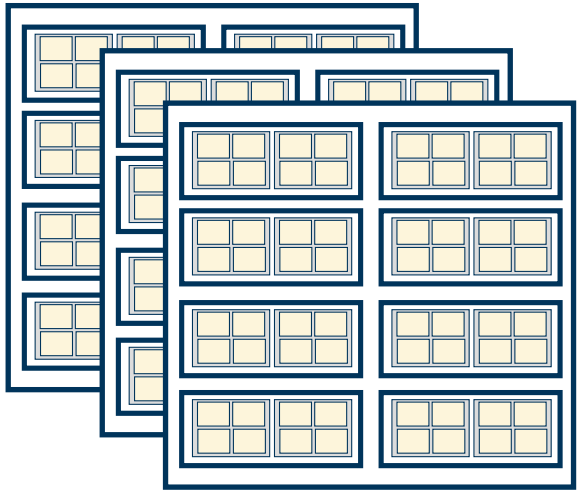
SIMD/Vector



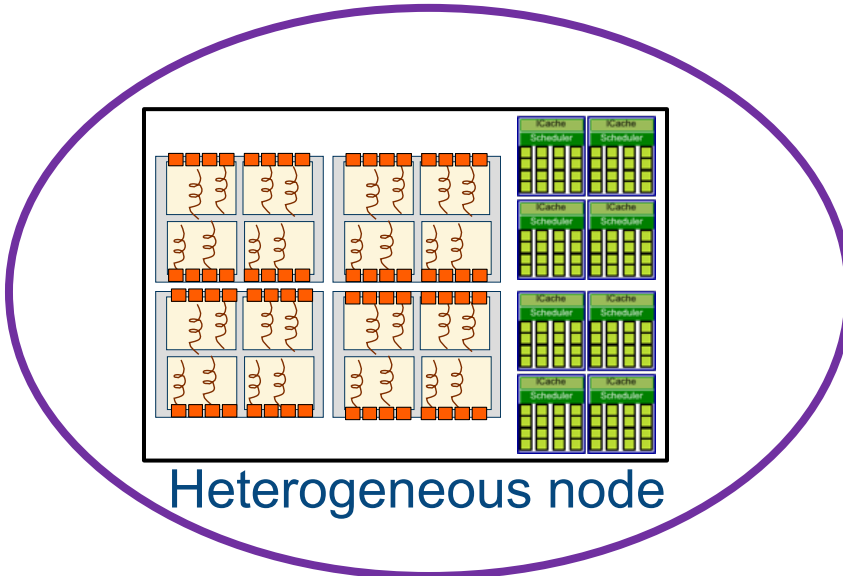
GPU



Cloud

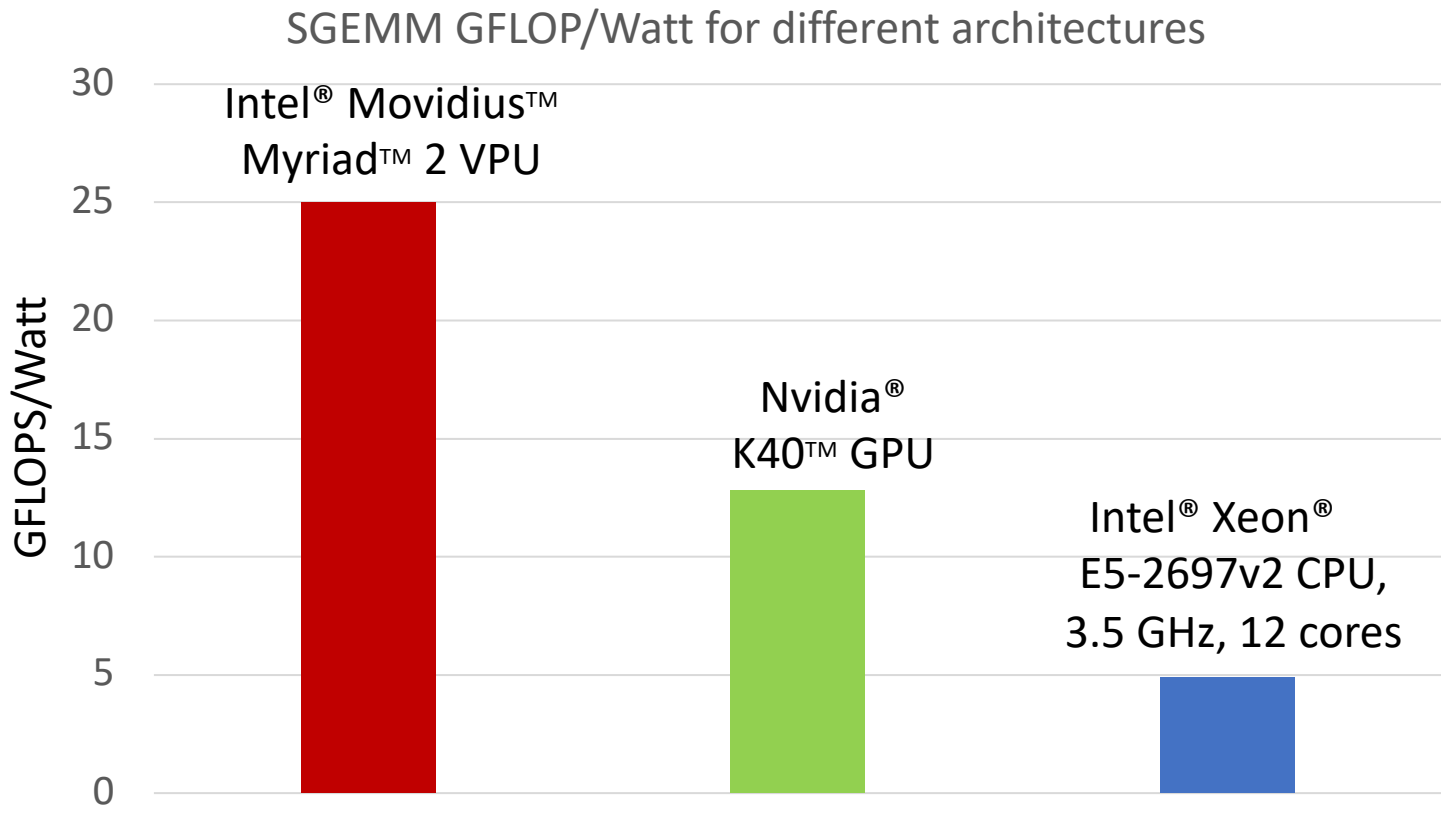


Cluster



If you care about power, the world is heterogeneous?

Specialized processors doing operations suited to their architecture are more efficient than general purpose processors.

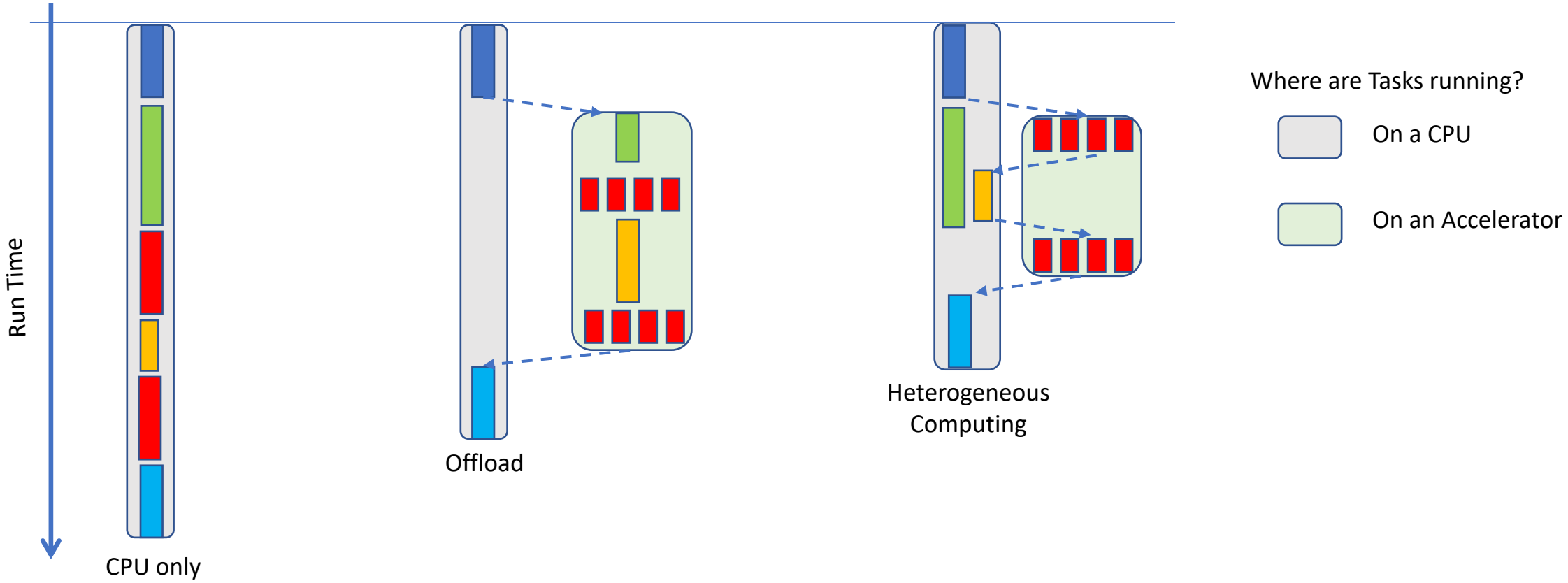


Hence, future systems will be increasingly heterogeneous … GPUs, CPUs, FPGAs, and a wide range of accelerators

Source: Suyash Bakshi and Lennart Johnsson, "A Highly Efficient SGEMM Implementation using DMA on the Intel/Movidius Myriad-2. IEEE International Symposium on Computer Architecture and High Performance Computing, 2020

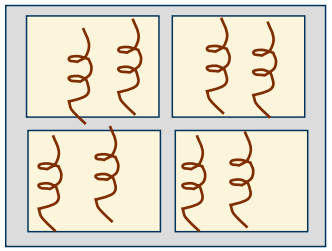
Offload vs. Heterogeneous computing

- **Offload**: The CPU moves work to an accelerator and waits for the answer.
- **Heterogeneous Computing**: Run sub-problems in parallel on the hardware best suited to them.



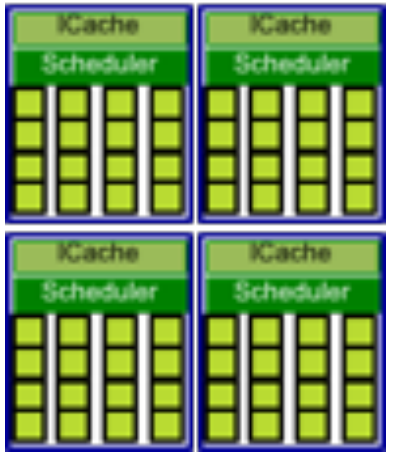
For hardware … parallelism is the path to performance

All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.

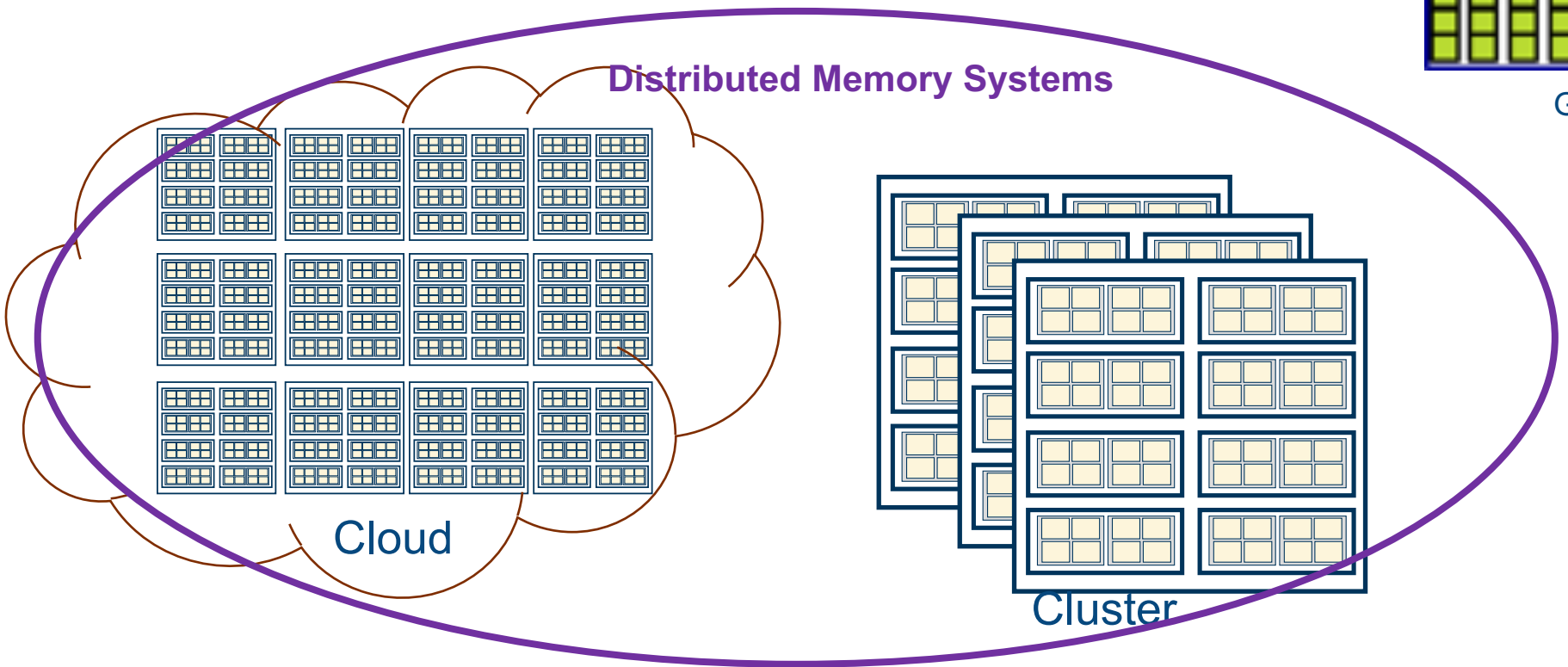


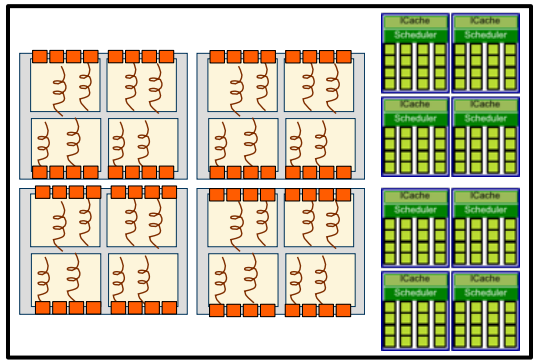
CPU

SIMD/Vector



GPU



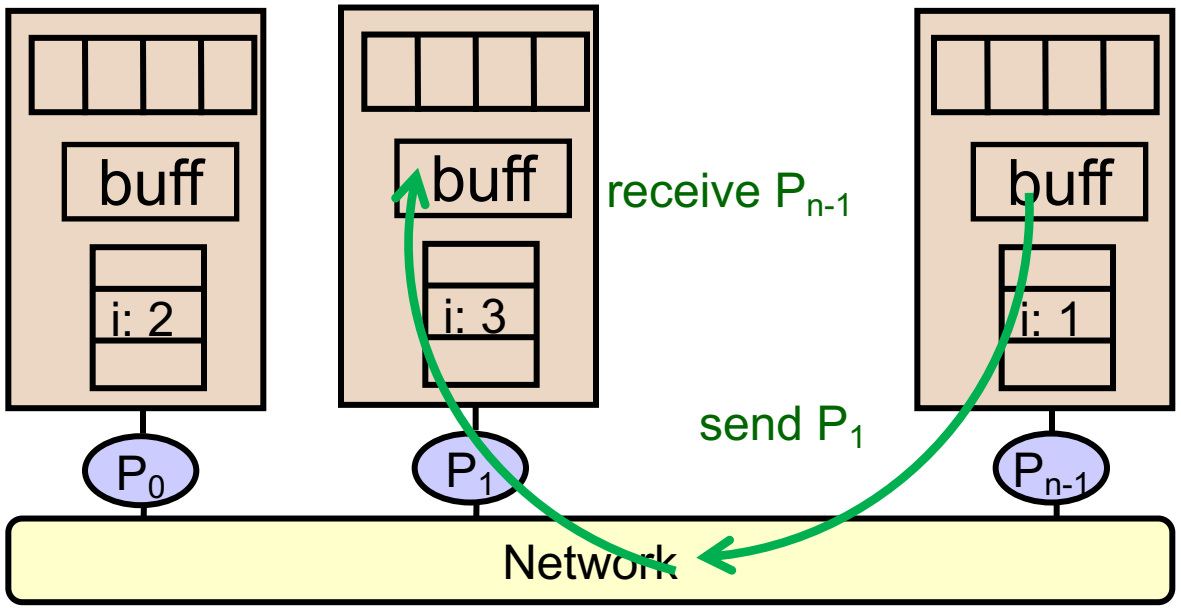


Heterogeneous node

Programming Model for distributed memory systems

- Programs execute as a collection of processes.
  - Number of processes usually fixed at program startup time
  - Local address space per node -- **NO physically shared memory**.
  - **Logically** shared data is partitioned over local processes.

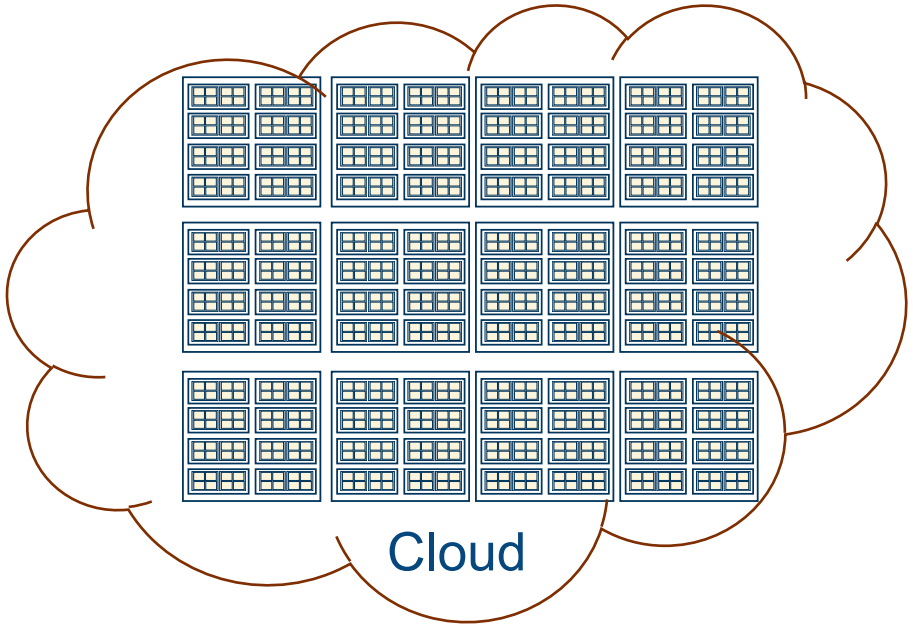
- Processes communicate by messages … explicit send/receive pairs
  - Synchronization is implicit by communication events.
  - MPI (Message Passing Interface) is the most commonly used API



A collection of n MPI processes (P_0 to P_{n-1}) running on n nodes

Throughput oriented computing

- A common pattern occurs when you need to run a large collection of problems that are truly independent
  - Map/Reduce: Run many independent programs (map) and combine results once when they are all done (reduce). Hadoop and Spark are optimized for this type of job
  - Parameter studies: Optimize a set of parameters by sampling a set of parameters and collecting overall statistics when done. Example: Finding a set of small molecules (ligands) that dock with a protein. It's geometry dependent so many ligand orientations must be tested.
  - Data Parallel jobs where each data point is a large compute intensive operation without any dependencies to manage.



- These problems are ideally suited to running in the cloud.

Clusters vs the Cloud for parallel programmers

Beyond embarrassingly parallel, throughput-oriented computing, the cloud is different from cluster computing (unless you pay to define a dedicated cluster in the cloud)

Platform*	HPC Cluster	Cloud
Execution Agent	Processes	Microservices
Memory	Distributed memory, local memory owned by individual processes	Distributed object store (in memory) backed by a persistent storage system
Typical Execution Pattern	SPMD	Event driven tasks, FaaS, and Actors

Parallel Systems are great, but they are not much use without parallel software

Parallel Software

- Vectorization is defined around a narrow range of data parallel operations. That narrow scope and the limited complexity in how data is laid out allows automatic vectorization to work.

- Automatic parallelization beyond vectors, however, has never adequately worked. Why:
  - Decomposing a problem into tasks that can run in parallel effectively is not directly apparent in the code for cases beyond vectors extracted from basic loops
  - Decomposing data to manage memory movement across multiple processing elements requires reasoning across the full range of functions in a program.
  - Typcally, the algorithm that is best for a serial program is different from the algorithm that is berst for parallel execution.

- Hence, software for parallel systems requires parallel programmers using programming models that explicitly support parallelism.

Consider the state of programming models from the early days of parallel computing.

Parallel programming environments in the 90's

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL
CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda
GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold
Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective
Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm
Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick
Threads
Sage++
SCANDAL
SAM
pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed
smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-
NUS
VPE
Win32
threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

Consider the state of programming models from the early days of parallel computing.

Parallel programming environments in the 90's

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
Fork
Fortran-M
FX
GA
GAMMA
Glenda
GLU
GUARD
HAsL.
Haskell
HPC++
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm
Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick
Threads
Sage++
SCANDAL
SAM
pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed
smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-
NUS
VPE
Win32
threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

This diversity made the life of a parallel application developer much more difficult.

Different communities and different system-vendors had different "favorite" preprogramming models. Developers had to waste vast ammounts of time porting applications to different systems.

Furthermore, engineering is a "zero-sum" game … that is time spent supporting multiple programming models means less time to make a small number of common models just work.

Sanity ruled by the end of the end of the 90's

- In HPC, 2 programming environments dominate … covering the major classes of hardware.
  - **MPI**: distributed memory systems … though it works nicely on shared memory computers.
  - **OpenMP**: Shared memory systems …
- Even if you don't plan to spend much time programming with these systems … a well rounded HPC programmer should know what they are and how they work.

Sanity ruled by the end of the end of the 90's

- In HPC, 3 programming environments dominate … covering the major classes of hardware.
  - **MPI**: distributed memory systems … though it works nicely on shared memory computers.
  - **OpenMP**: Shared memory systems … more recently, GPGPU too.

Then GPUs Came and things got a bit more messy

  - **CUDA**, **OpenCL**, **Sycl**, **OpenACC**, **OpenMP** … : GPU programming (use CUDA if you don't mind locking yourself to a single vendor … it is a really nice programming model)

- Even if you don't plan to spend much time programming with these systems … a well rounded HPC programmer should know what they are and how they work.

Sanity ruled by the end of the end of the 90's

- In HPC, 3 programming environments dominate … covering the major classes of hardware.
  - **MPI**: distributed memory systems … though it works nicely on shared memory computers.
  - **OpenMP**: Shared memory systems … more recently, GPGPU too.
  - **CUDA**, **OpenCL**, **Sycl**, **OpenACC**, **OpenMP** … : GPU programming (use CUDA if you don't mind locking yourself to a single vendor … it is a really nice programming model)

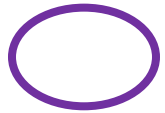
What's different today is C++. OpenMP and MPI work with C++, but they were not created with the structure, style, and capabilities of C++ in mind.

C++ contains threads as first class language elements, standard parallel containers and algorithms, a formal memory model, and a number of high-level programming models specifically designed around the language (e.g,, Kokos, Sycl and TBB)

Fortunately, the C++ language design community is working on support for parallelism and painstakingly working to pull the best ideas into the core language.

Sanity ruled by the end of the end of the 90's

- In HPC, 3 programming environments dominate … covering the major classes of hardware.
  - **MPI**: distributed memory systems … though it works nicely on shared memory computers.
  - **OpenMP**: Shared memory systems … more recently, GPGPU too.
  - **CUDA**, **OpenCL**, **Sycl**, **OpenACC**, **OpenMP** … : GPU programming (use CUDA if you don't mind locking yourself to a single vendor … it is a really nice programming model)



Covered in ESC

What's different today is C++. OpenMP and MPI work with C++, but they were not created with the structure, programming philosophy, and capabilities of C++ in mind.

C++ contains threads as first class language elements, standard parallel containers and algorithms, a formal memory model, and a number of high-level programming models specifically designed around the language (e.g,, Kokos, Sycl and TBB)

Fortunately, the C++ language design community is working on this problem and painstakingly working to pull the best ideas into the core language. Much work remains to be done, but the progress so far is very exiling.

Summary

- High performance computing depends on parallel computing.

- Scientific programmers need to be comfortable with the full range of parallel systems:
  - CPUs with multiple cores sharing an address space … optimized for latency sensitive problems.
  - Vector (or SIME) units integrated with the CPU.
  - GPUs for compute intensive, high throughput problems.
  - Distributed memory systems with heterogeneous nodes built from the above for extreme scale

- It's a lot to learn, but fortunately, these systems can be fully addressed with a modest number of parallel programming models … we covered vectorization in this lecture and we'll cover the others over the next two weeks.

… But for now, let's forget parallel computing and more on to computer arithmetic.

It's boring, but if you care about high quality answers to your scientific problems, you need to understand that **Floating Point numbers are not real**