Computer Architecture (x86-64)

Devin A. Matthews
UT Austin

MolSSI Software Summer School 2017

Computing isn't like your Gran^H^H^H^H Advisor Remembers it…

- Computers have gotten faster (more GHz), but they've also changed and developed on a fundamental level
  - Example: a Cray-YMP (1988) could deliver near-peak speed when streaming from memory
  - A multi-core Broadwell Xeon can only deliver about 1% of peak performance from memory

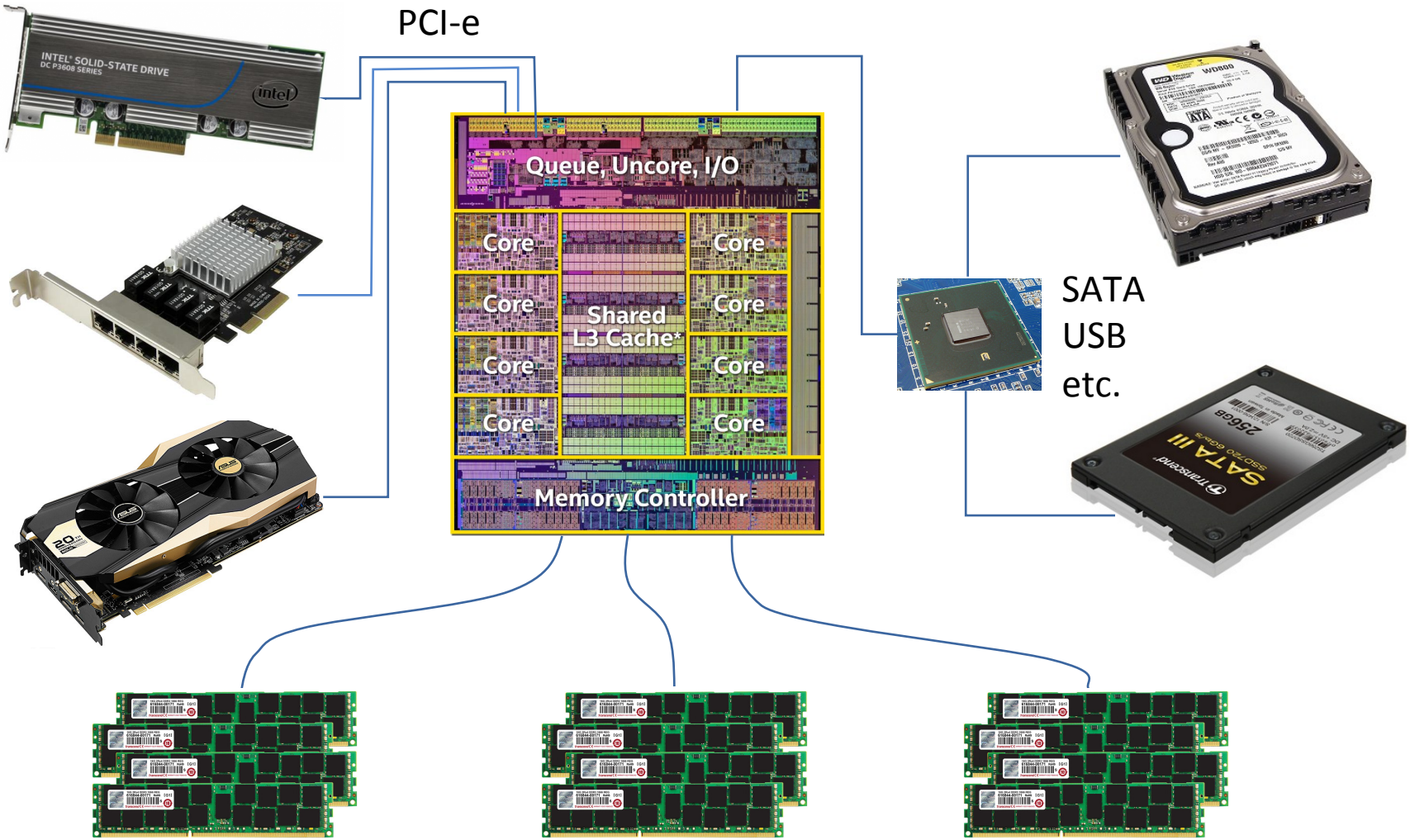
- New features make old optimization strategies obsolete:
  - Out-of-order execution
  - Superscalar execution
  - Vectorization (aka SIMD, not to be confused with vector machines!)
  - Multiple levels of cache
  - Multi-core
  - Many-core
  - Multi-socket
  - Distributed computing
  - And so on…

- Compilers (and languages) have gotten better too, but they aren't magic. To get the best performance, you have to be aware of what the computer is actually doing.

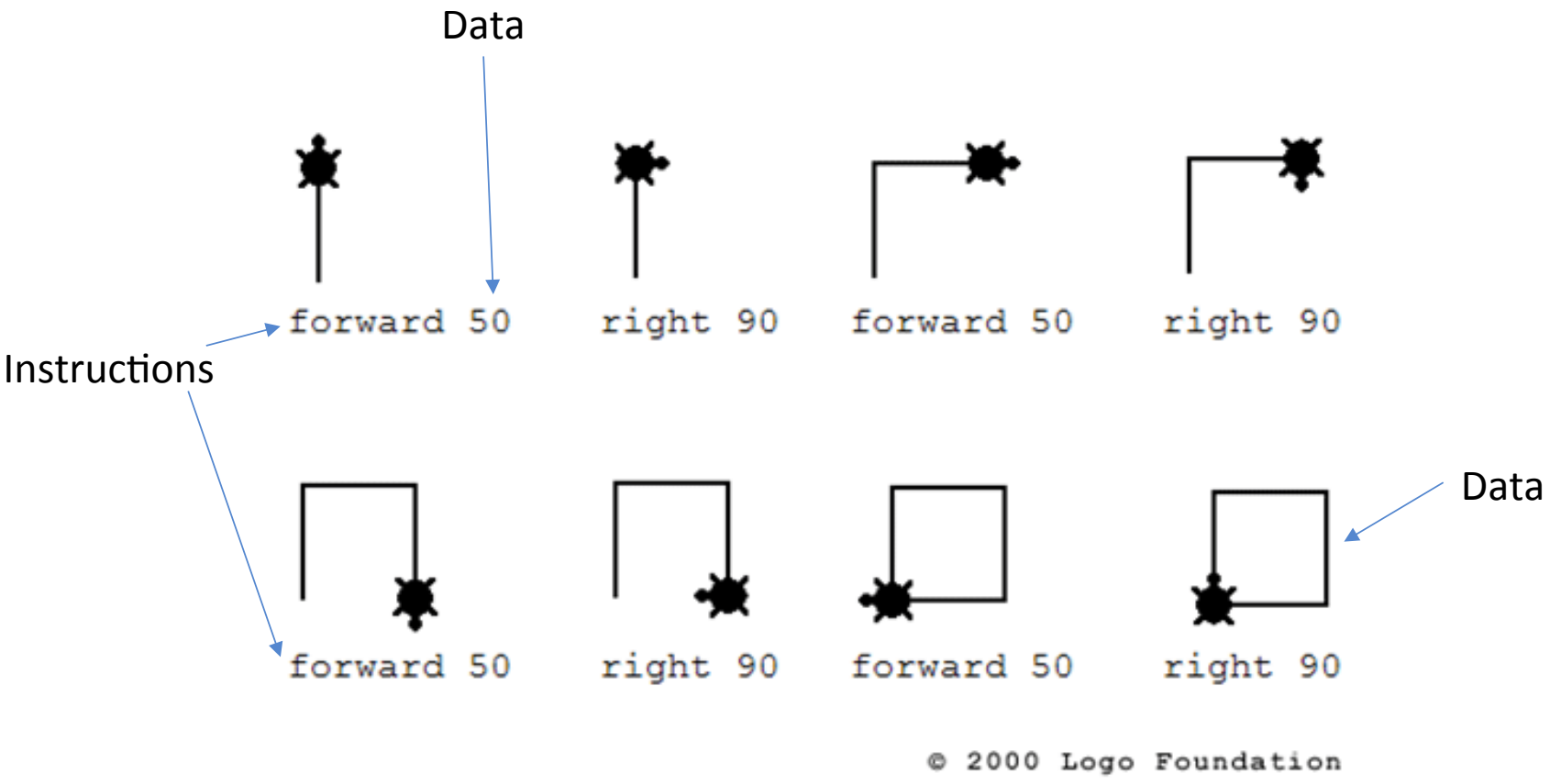
Why is fast code important?

- Writing fast code takes time and effort, but when done right it saves a lot more time than you put in.
- Computer time costs $$ and grant-writing time. Why not use it effectively?
- Fast code allows new and bigger simulations. Sometimes this leads to being able to do fundamentally new science.
- Execution time savings are multiplied by number of users and number of times run. So write code that can be reused and shared. Corollary: *don't* write it if someone else already has.

So what's in a modern computer?

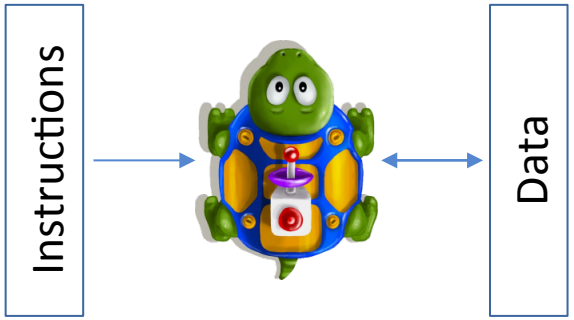


A simple model:

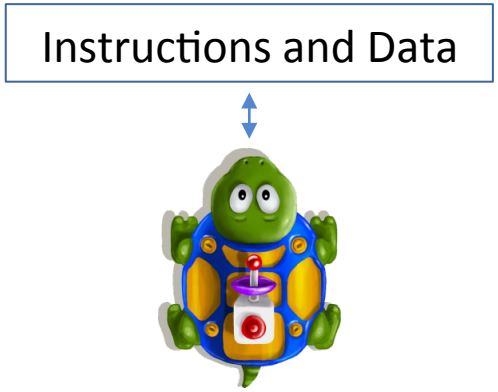


Instructions vs. data

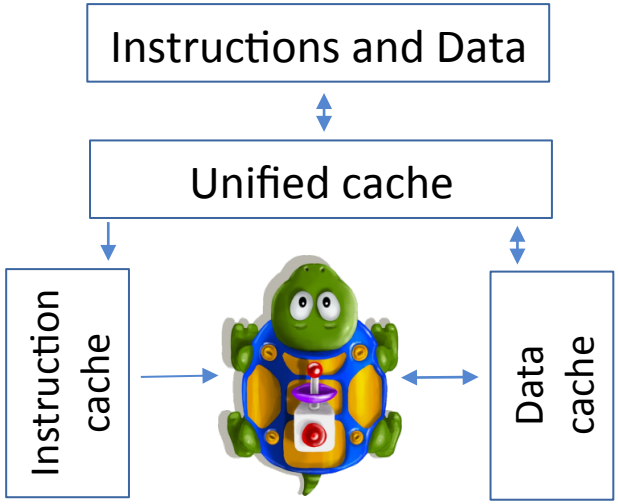
Harvard Architecture:



Von Neumann Architecture:



Most Modern Architectures:



Instruction examples

- Math ADD SUB MUL DIV SHL/SHR/SAR

  VADDPD VFMA231SD VSQRTSD VRCP28PS

- Logic TEST CMP SETxx CMOVxx AND/OR/XOR/…

- Memory MOV LEA PREFETCH VMOVUPS VBROADCASTSS

- Flow control CALL/RET JMP Jxx REP/REPxx

(xx = AE, NZ, GE, etc.)

Instruction latency and throughput

- **Latency** is how long it will take in cycles for the results of the instruction to be ready. *Lower is better.*

- **(Reciprocal) Throughput** is how many cycles on average do you have to wait between two instructions of the same type. This can be lower than the latency because of **pipelining** and **superscalar execution**. This is also called **CPI** (the inverse of **IPC**). *Lower is better.*

VMULPD

Performance

Architecture	Latency	Throughput
Skylake	4	0.5
Broadwell	3	0.5
Haswell	5	0.5
Ivy Bridge	5	1

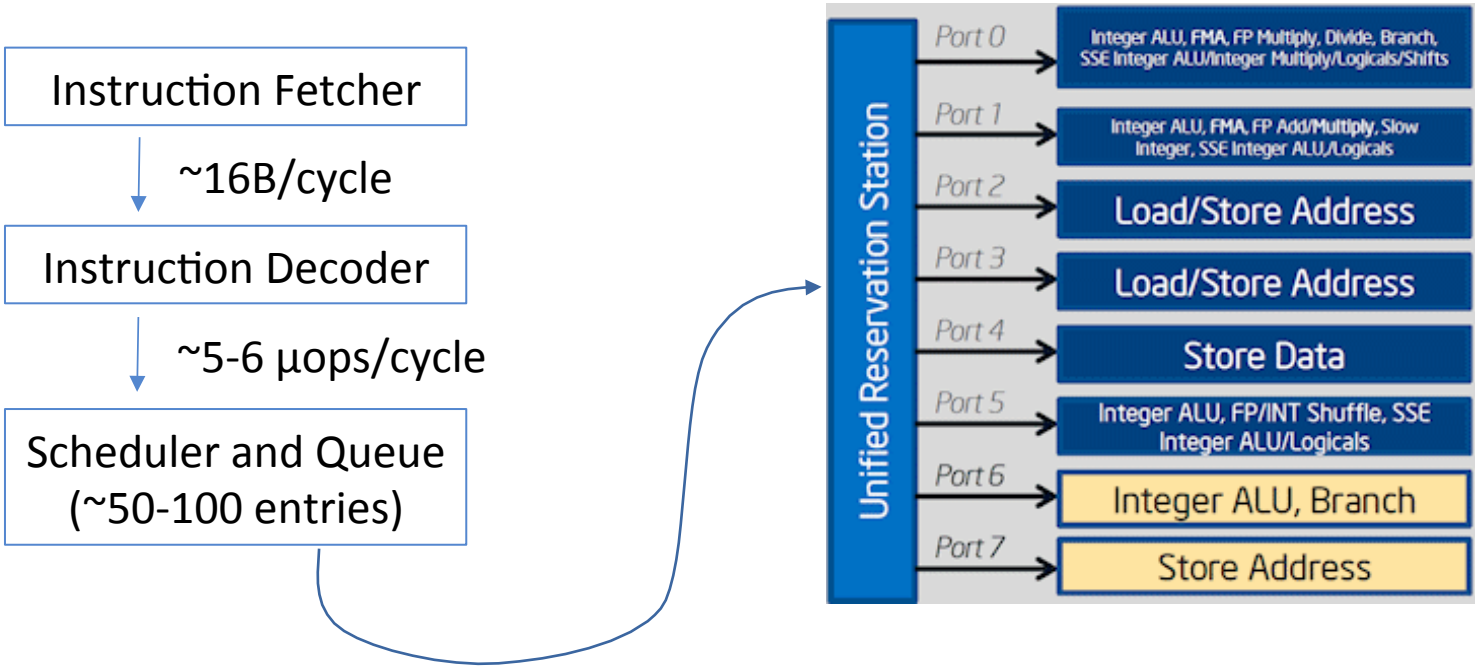
VDIVPD

Performance

Architecture	Latency	Throughput
Knights Landing	38	~10
Skylake	14	8
Broadwell	16-23	16
Haswell	25-35	27
Ivy Bridge	27-35	28

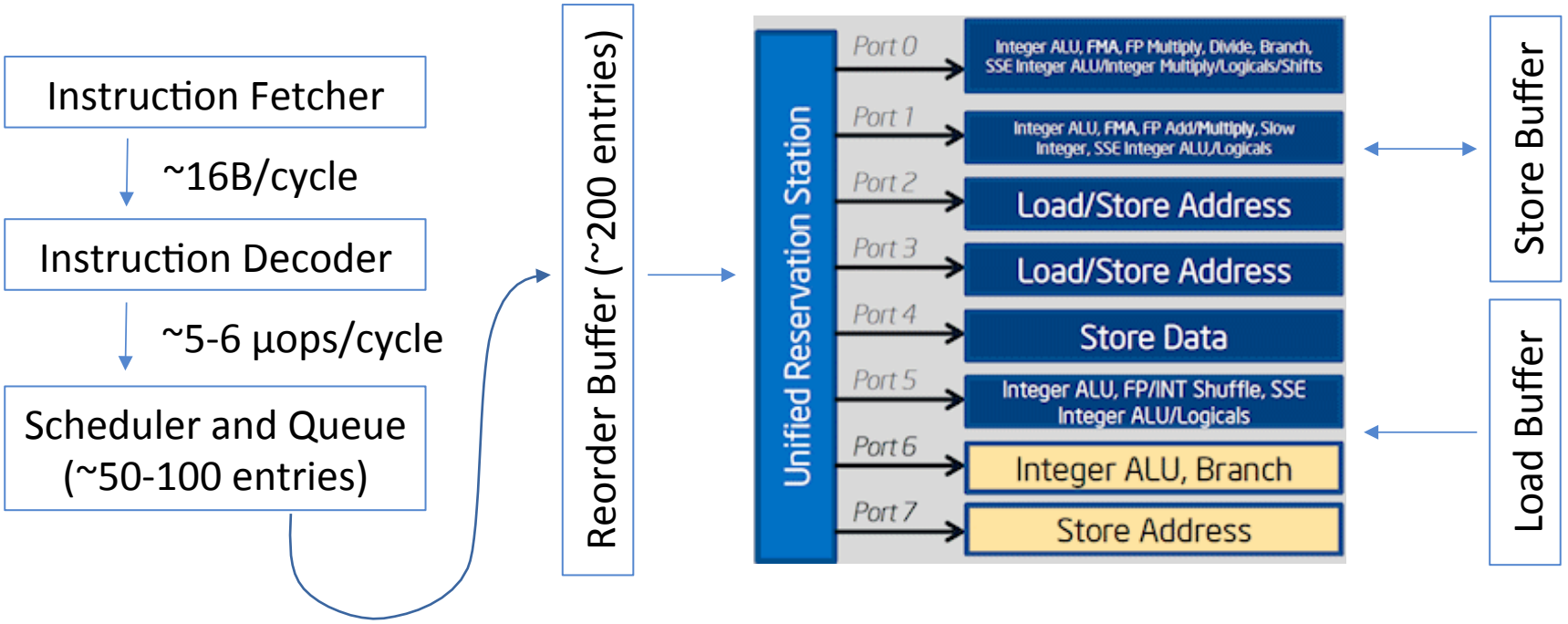
Superscalar execution

- Modern processor break instructions down in smaller pieces called **µops**.
- The rate at which instructions can be completed is limited by 1) instruction **fetch and decode**, and 2) availability of **execution ports** for the resulting µops.



Out-of-order execution

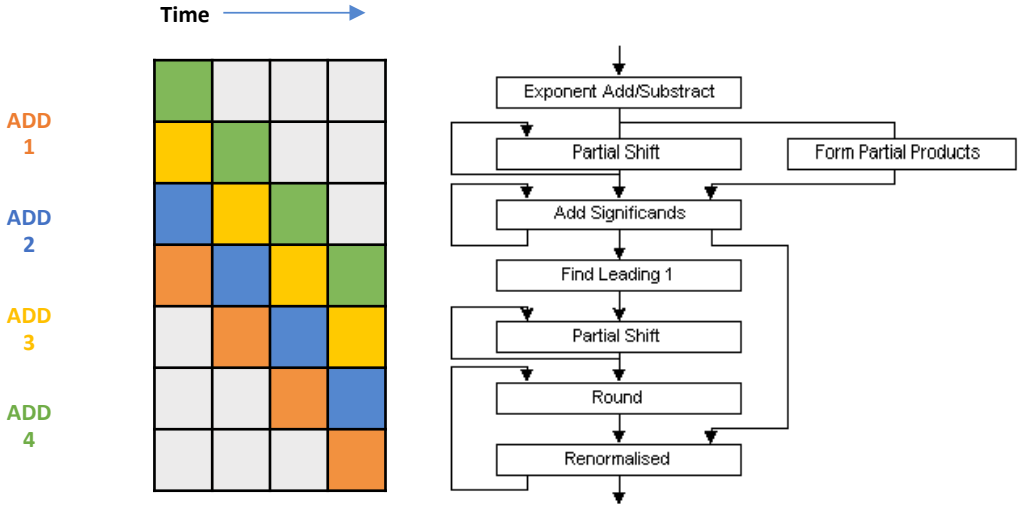
- If the processor needs to wait for data to arrive, it can execute other instructions in the mean time. This is only possible when the instructions do not have a **data dependency**, i.e. the output of one is an input of another.



Pipelining

- In addition to executing multiple instructions at a time in *different* execution units, the *same* execution unit can execute multiple instructions through **pipelining**. This is especially important for +, -, and * operations of floating point data (but not /!).

- Note that pipelining also exists in the interaction of instruction fetching, decoding, and µop scheduling: these all run at the same time on different instructions.



Branches

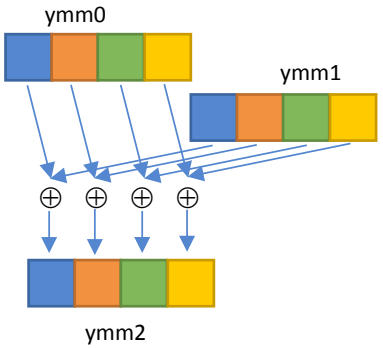
- Most kinds of flow control get converted into **branch instructions** (or **jumps**). The processor has to decide which branch is more likely so that it can continue to prefetch instructions and execute out-of-order (this is called **speculative execution**). If the processor guesses wrong, you pay a **branch misprediction penalty** (~15-20 cycles), plus a possible pipeline flush.

- Processors have gotten pretty good at predicting branches, including pattern recognition for nested loops, but reducing the number of branches can still help a lot.

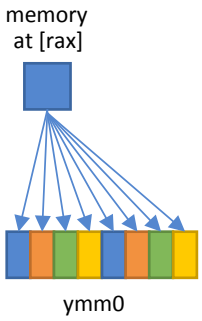
Single Instruction Multiple Data (SIMD)

- Basically all processors for the past 10 years can operate on **vectors** of floating point and (more recently) integer numbers:
  - **MMX, 3dNow!:** legacy SIMD extensions (90's)
  - **SSE2-SSE4.2:** 2x double or 4x float (+integers) (2000's)
  - **AVX:** 4x double or 8x float (2010's)
  - **AVX2:** same as AVX but adds integer support and FMA
  - **AVX512:** 8x double or 16x float (2016)
    - Currently on KNL, just released for Xeon.

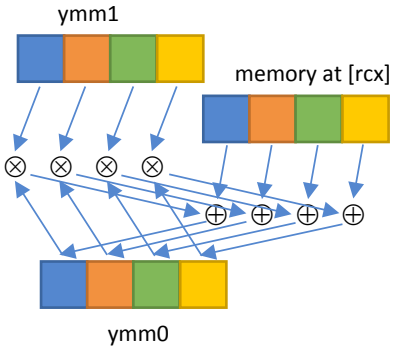
vaddpd ymm2, ymm1, ymm0



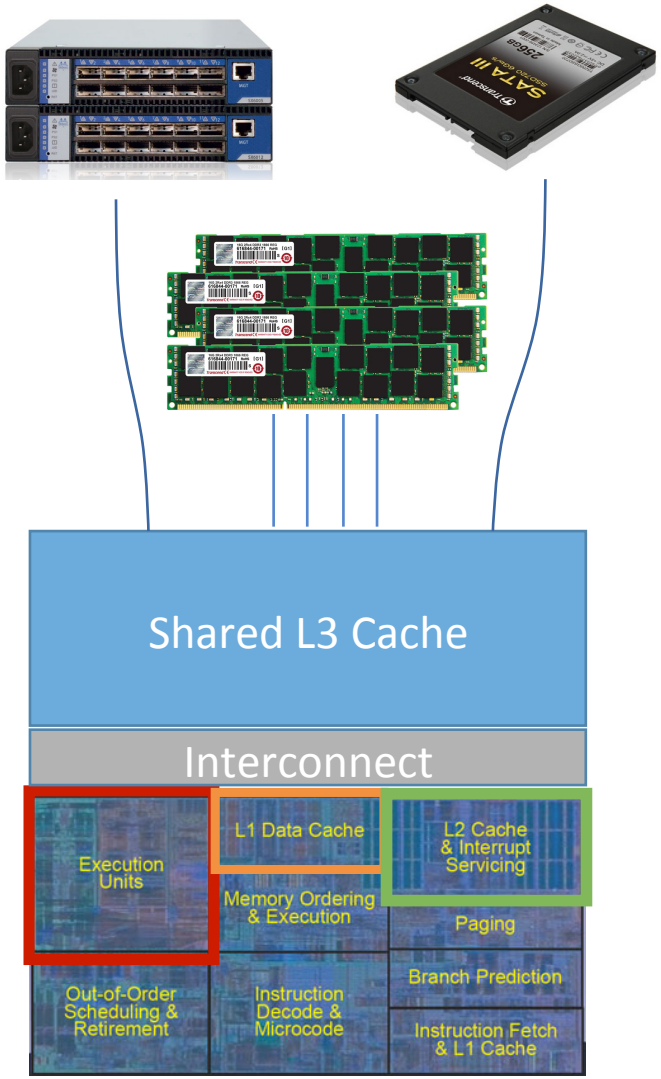
vbroadcastss ymm0, [rax]



vfmadd213pd ymm0, ymm1, [rcx]



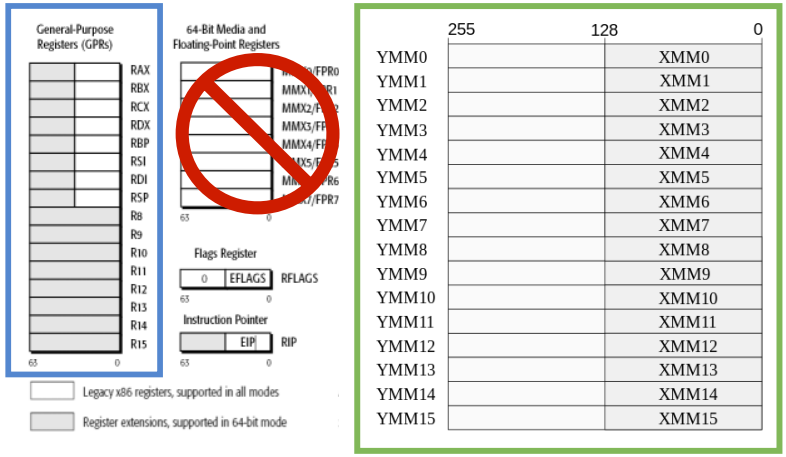
The memory hierarchy



Memory Level	Size	Latency	Bandwidth	Technologies
Registers	~5K, 624B visible	n/a	n/a	SRAM/DFF
L1 cache	32K	~5 cycles	64B read 32B write per cycle	SRAM
L2 cache	256K-1M	~15 cycles	64B per cycle	SRAM
L3 cache	~1M-50M	~20-30 cycles	32B per cycle	SRAM
Memory	GBs	~100 cycles	~100GB/s = <10B/cycle/core	DRAM
Network (remote DMA)	PBs	1000s of cycles	1-10GB/s	Ethernet, Infiniband, Custom
Disk (SSD)	TBs	~100,000 cycles	~1GB/s	Flash memory, PCIe or SATA

Registers

- x86-64 processors have 16 64-bit **integer registers** and 16 128-bit **vector registers** (xmm0-15). The x87 FPU is not used in modern code.
- AVX extends the vector registers to 256 bits (ymm0-15).
- AVX512 increases this to 32 512-bit vector registers (zmm0-31, not shown).
- Most inputs and outputs of instructions must be registers. Some instructions allow one memory operand.
- The "name" of each register is not important to the compiler. **Register renaming** allows the processor to use different physical registers for instructions that use the same register name.
- Since the number of physical registers (the **register file**) is much larger than the number of names (~10x), this avoids many false data dependencies.

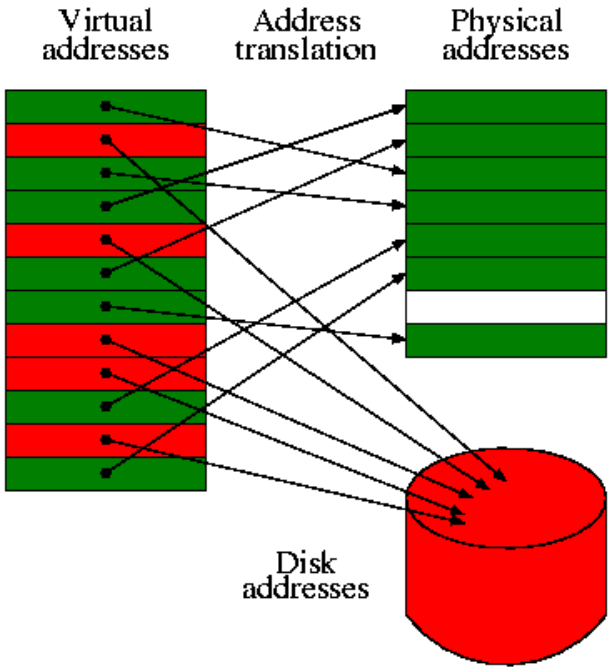


Cache

- The level 1, 2, and 3 caches are all accessed in units of (generally) 64B, called a **cache line**.
- If the entire cache line is not used, bandwidth can be wasted.
- When the caches become full, cache lines are **evicted** based on some policy such as least-recently used (LRU).
- If a piece of data is used again, it can be accessed using **non-temporal** loads and stores, to keep from evicting other data.
- Cache lines be proactively moved to one of the caches by a **prefetch** instruction. The processor also does **hardware prefetching** based on observed usage patterns.
- A given entry in the cache can only store cache lines from certain memory locations. This is referred to as **cache associativity**. One ramification of this is that repeated accesses to data at intervals of large powers of 2 in bytes ($2^n \geq 4KB$) reduces the effective cache size drastically.

Virtual memory

- The memory of a program is divided into **pages**, generally of size 4KB.
- The pages are mapped by the hardware and operating system onto the physical memory.
- Since the location of data doesn't match the underlying hardware, this is called **virtual memory**.
- The mapping from virtual to physical memory is cached in the **TLB**, but only for a limited number of pages.
- Accessing a page that does not have its mapping cached causes a **TLB miss**. This generally costs O(100) cycles.
- If the page has not been assigned a physical address (e.g. if it hasn't been accessed before), then a **page fault** occurs, costing an additional O(1000) cycles.
- Additionally, virtual memory pages may be **memory mapped** to files on disk, where the OS transparently handles I/O and caching.



Single-core summary

- The processor core can do lots of things at once:
  - Execute **out-of-order**.
  - Execute multiple instructions (**superscalar**).
  - **Pipeline** sequences of instructions.
  - **Predict branches** and speculatively execute.
  - Calculate multiple results simultaneously with **SIMD**.

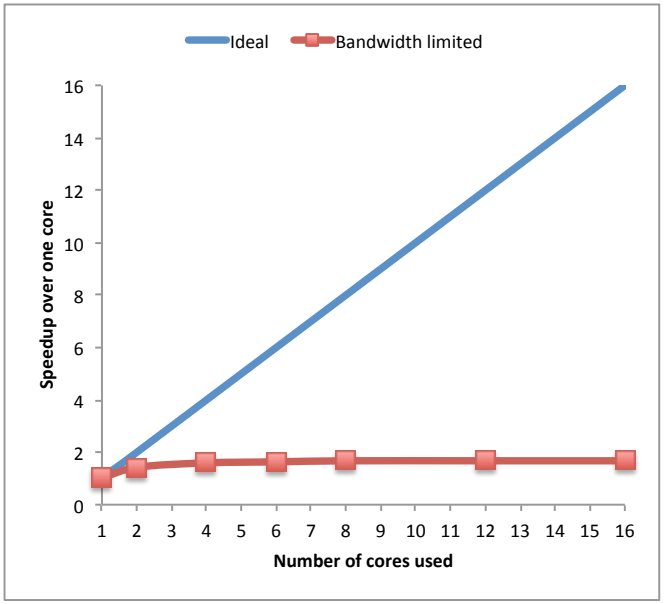
- The bottleneck is getting data from memory (or beyond)
  - (Almost) all data has to be brought into **registers** before being used.
  - **Caches** hold a small amount of data closer to the processor.
  - **Cache lines** and **cache associativity** make some access patterns more efficient.
  - **Virtual memory** paging can cause additional delays if too many pages are accessed.

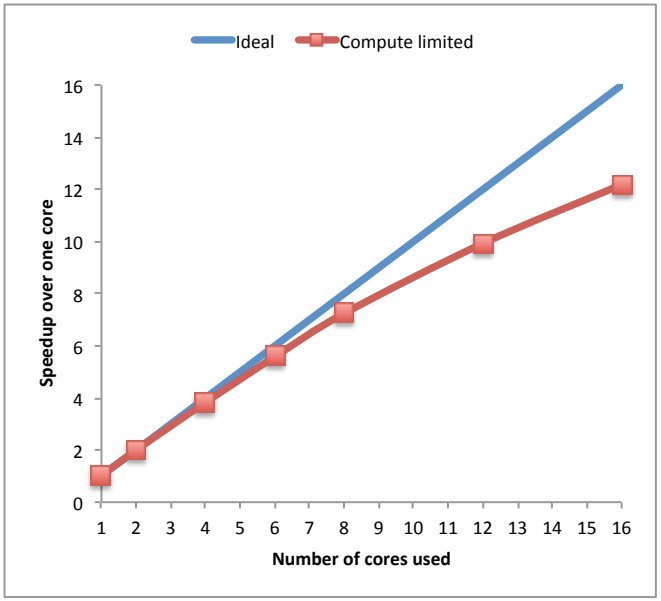
Multi-core

- Modern processor contain multiple processor **cores**.
- Individual cores may also allow multiple **threads** to run at the same time, sharing some resources such as the execution units. This is called **Simultaneous Multi-Threading** (**SMT**, aka **Hyper-Threading**).
- This can hide additional latency and increase responsiveness.
- It is not usually very helpful in HPC. In fact, using all hyper-threads can hurt performance in some cases.

Shared vs. private resources

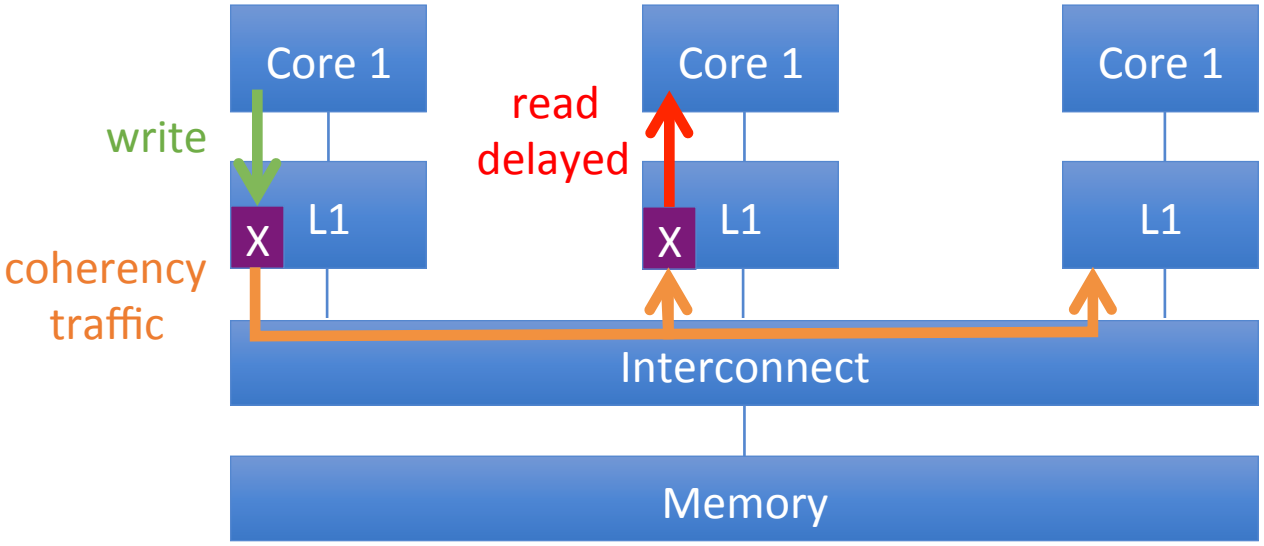
- Some resources in the processor are **private** to a core (such as the execution units and L1 and L2 caches), and some are **shared** (the L3 cache, memory, everything else).
- (Note that for hyper-threads, only a very few resources are private, such as the register file.)
- Which resources are the most heavily used affects how well an algorithm **scales** with the number of cores:





Data sharing and cache coherence

- When processors have the same data in their private caches, what happens when they both write to?
- Modern systems use a **cache coherence** protocol to synchronize writes to cached data so that all processors see writes to some value in the same order (this is called **sequential consistency**). **Note that only writes to the same value are consistent.



False sharing

- **False sharing** occurs when two or more cores both read and write to different data in the same cache line.
- The cores will repeatedly invalidate the others' cache lines, causing significant overhead.
- **Thread-private** data should be kept on it's own cache line.

Q: What's wrong with this code?

```
double sum[NUM_THREADS] = {};

#pragma omp parallel for
for (int i = 0;i < N;i++)
{
    int tid = omp_get_thread_num();
    sum[tid] += array[i];
}
```

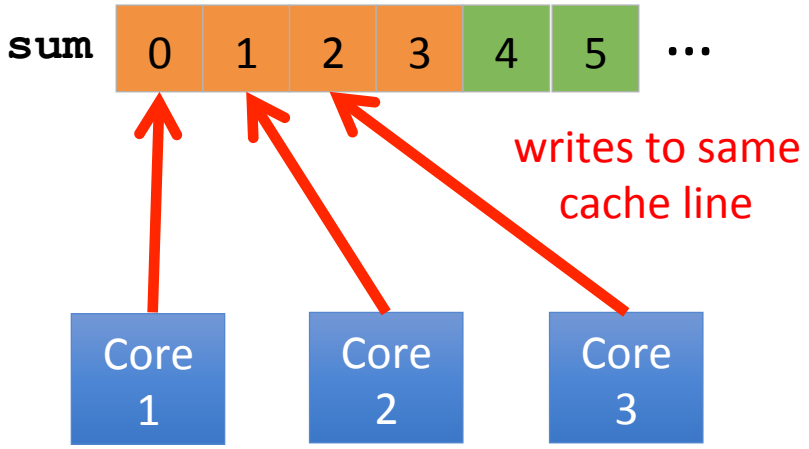
False sharing

- **False sharing** occurs when two or more cores both read and write to different data in the same cache line.
- The cores will repeatedly invalidate the others' cache lines, causing significant overhead.
- **Thread-private** data should be kept on it's own cache line.

Q: What's wrong with this code?

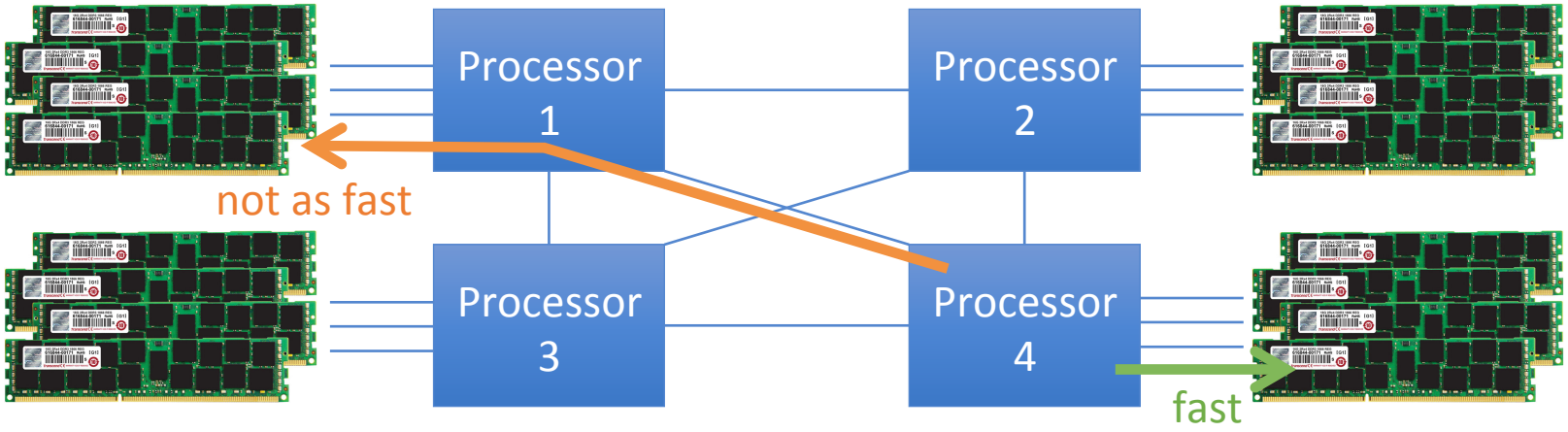
```
double sum[NUM_THREADS] = {};

#pragma omp parallel for
for (int i = 0;i < N;i++)
{
    int tid = omp_get_thread_num();
    sum[tid] += array[i];
}
```

A: False sharing in **sum**



Multi-socket and NUMA

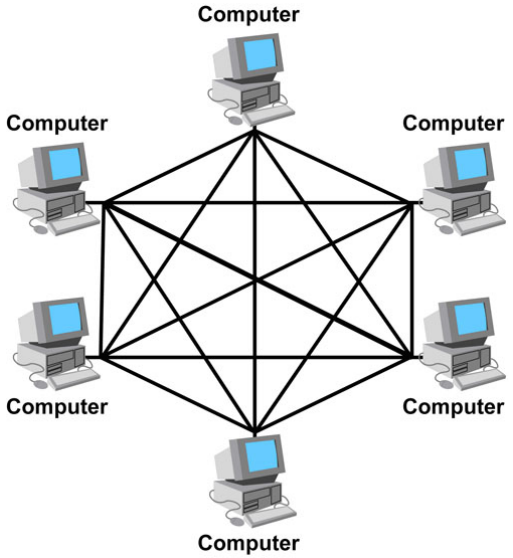
- Multiple processor chips can be connected together in a **multi-socket** or **Multi-Processor** (MP) arrangement.
- Each processor generally "owns" its own set of physical memory.
- Data from other processors is shared through an interconnect such as **QPI** or **HyperTransport**.
- Since some data takes longer to access than others, this is called a **Non-Uniform Memory Architecture** (NUMA).
- Where the data gets stored is usually determined by the **first-touch** principle.



Distributed computing

- For even more parallelism, individual machine (nodes) can be connected together with a **network**.

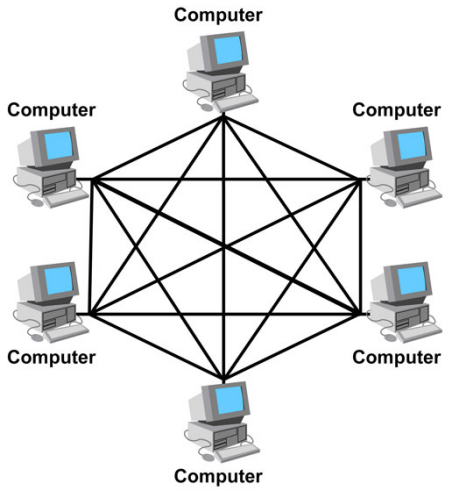
=



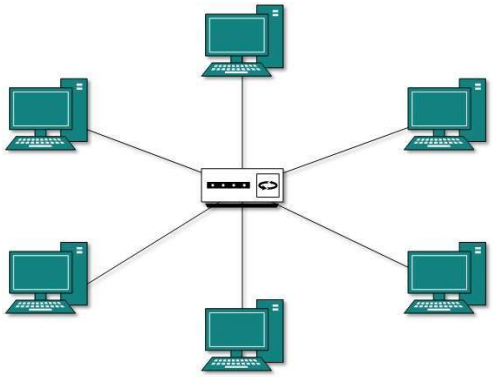
- The most common way to parallelize across nodes is with all of them running the program (**single-program multiple data**, SPMD), but they can also run other programs such as client-server (**multiple-program multiple data**, MPMD).

Network topologies

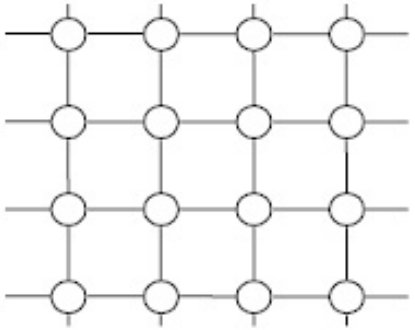
Fully connected



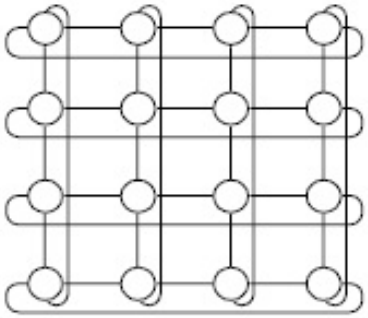
Star



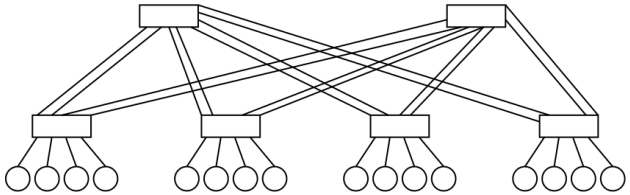
Mesh



Torus

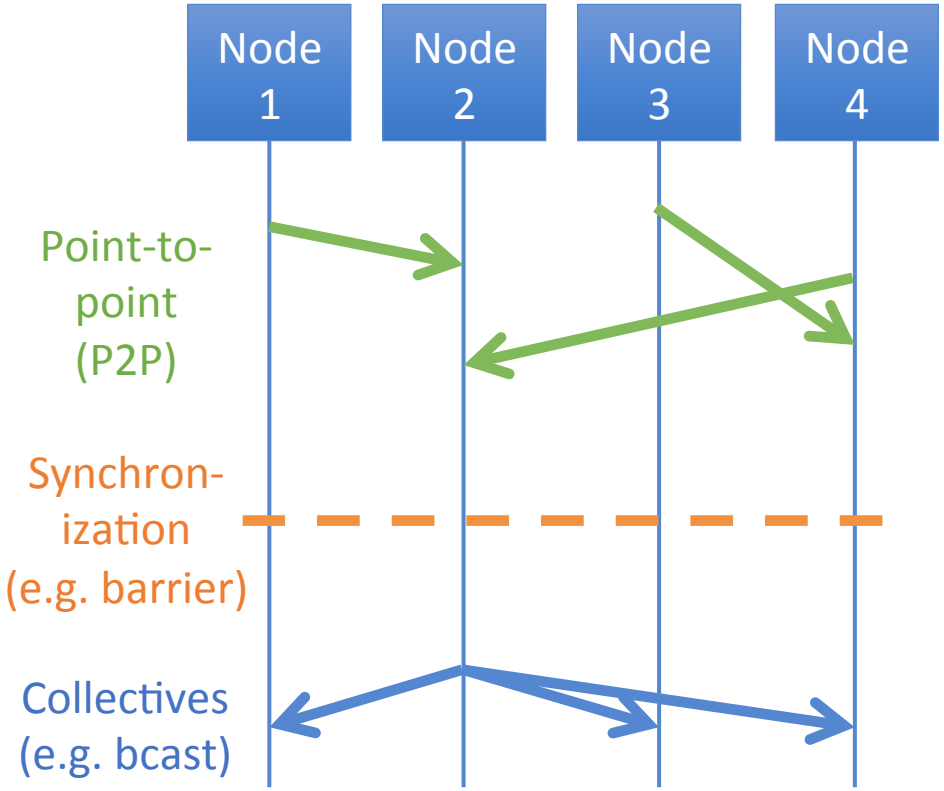


Fat tree

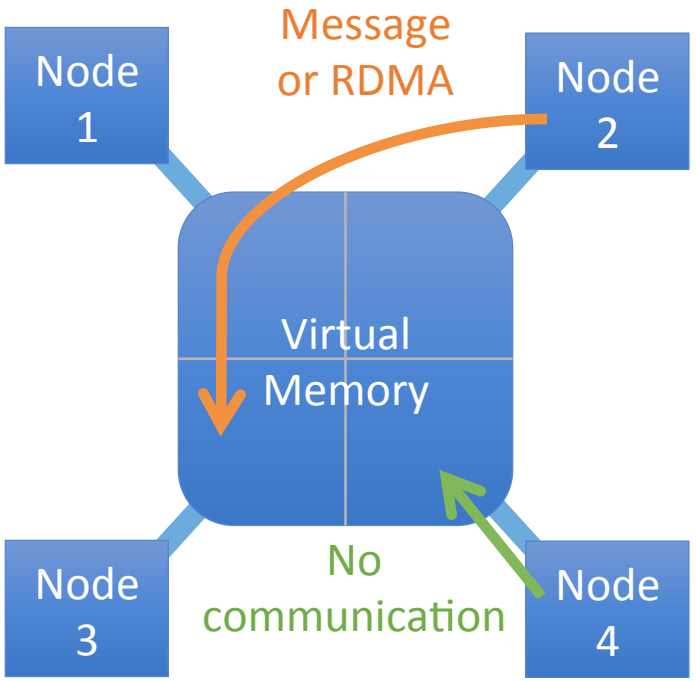


Distributed programming models

Message Passing

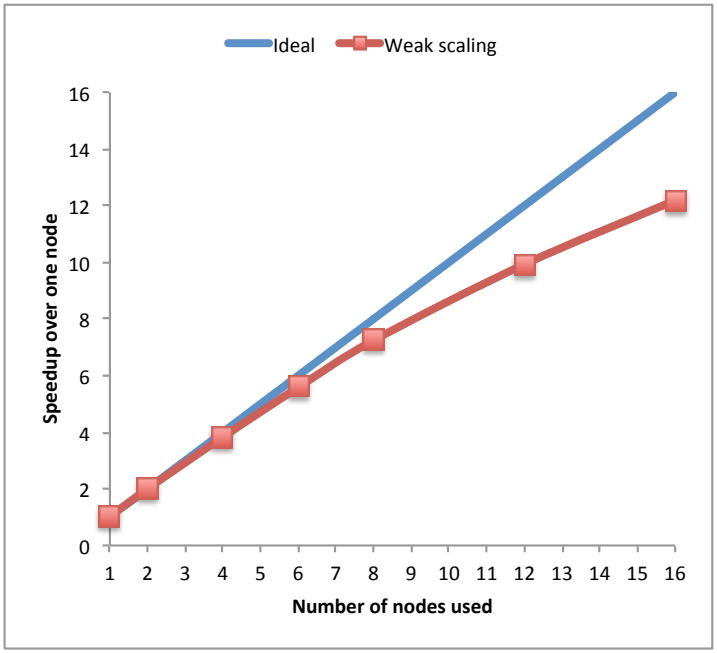


Partitioned Global Address Space (PGAS)

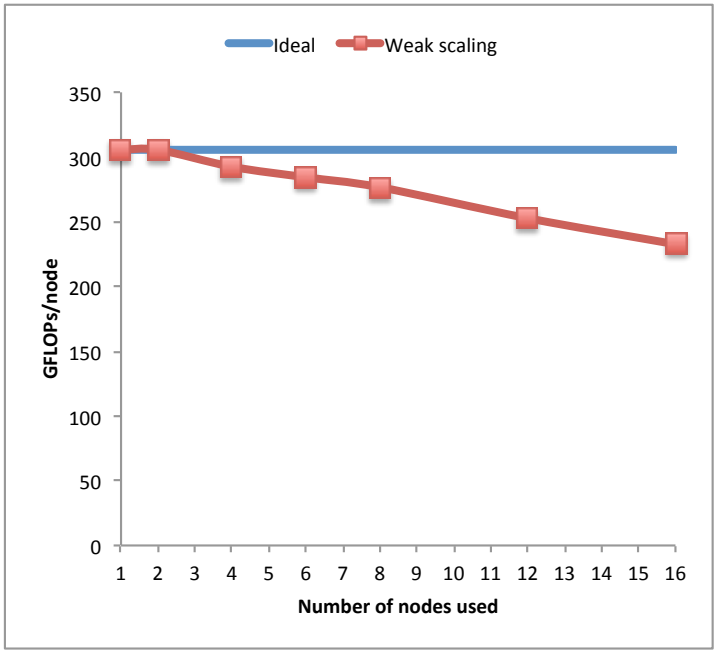


Types of scaling

- **Strong scaling** measures how much faster multiple processors can do the same work. The quantity usually measures is **speedup** ($T_{p=1}/T_{p=n}$).



- **Weak scaling** measures how efficient the computation is when the amount of work increases with the number of processors.



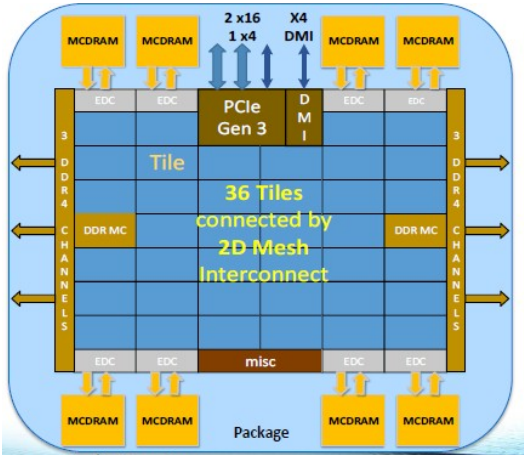
Multi-core and distributed summary

- HPC can exploit multiple levels of parallelism outside the processor core:
  - Multiple **cores**, including multiple threads per core (**hyper-threading**).
  - Multiple processors (**multi-socket**).
  - Multiple nodes (**distributed computing**).

- High-performance programs have to be careful about how they access data:
  - **Cache coherency**.
  - **False sharing**.
  - **NUMA**.
  - **Local** vs. **remote** memory.

Other architectures: Xeon Phi

- Xeon Phi (past: Knight's Corner, present: Knight's Landing, future: Knight's Mill) is a **many-core** architecture with 64-72 cores per processor.
- Knight's Landing is an x86-64 processor, but:
  - The cores are fairly slow for non-floating point operations.
  - There is no shared L3 cache.
  - There are two types of main memory: **high-bandwith memory** (16GB at ~400GB/s) and regular DDR (up to 384 GB at ~80GB/s). The HBM can be configured as either cache or as a separate NUMA domain.



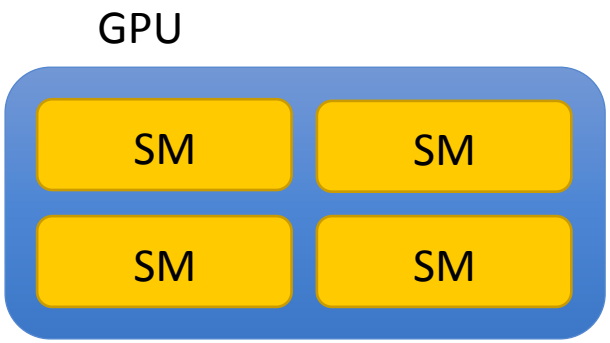
Other architectures: GPU

CPU:

- Fast, smart cores
- Large caches
- Low memory bandwidth (except KNL)
- Each core executes vector instructions
- Best scheduled one thread per core

GPU:

- Slow, dumb cores
- Small caches
- High memory bandwidth (100s of GB/s)
- Threads act in teams (warps) to execute on multiple cores.
- Many more threads then cores to hide latency (1000s of threads)



Questions?