High Performance Computing for Science and Engineering I

Petros Koumoutsakos

Lecture 3:

Parallel Computing Concepts

Architectures

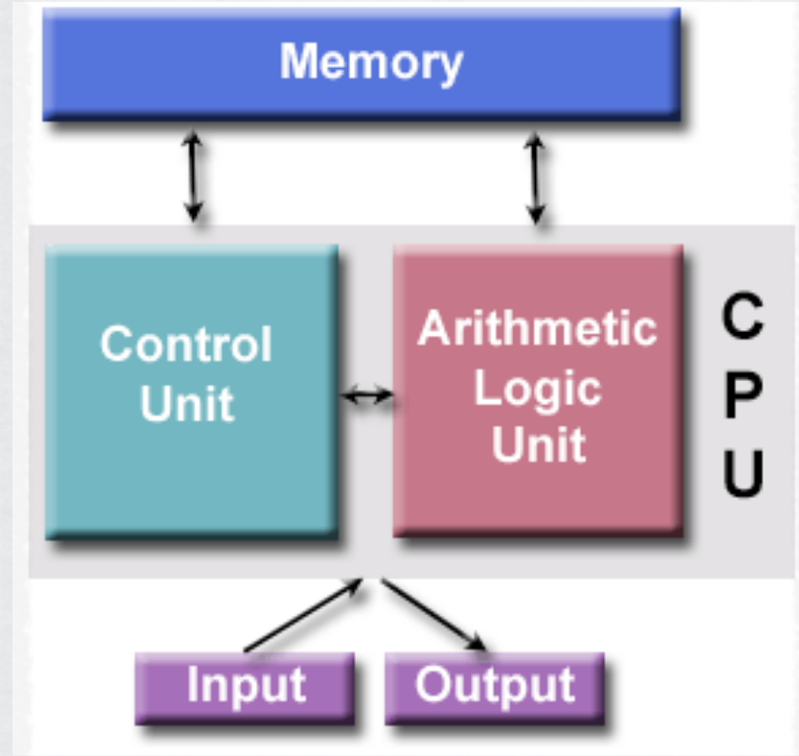
Performance Metrics

PARALLEL COMPUTING CONCEPTS & TERMINOLOGY

von Neumann Architecture

Four main components:

- Memory
- Control Unit
- Arithmetic Logic Unit
- Input/Output



Read/write, Random Access Memory stores both program instructions and data

- Program instructions are coded data which tell the computer to do something
- Data is simply information to be used by the program

Control unit fetches instructions/data from memory, decodes the instructions and then ***sequentially*** coordinates operations to accomplish the programmed task.

Arithmetic Unit performs basic arithmetic operations

Input/Output is the interface to the human operator

***Many** parallel computers follow this basic design, just multiplied in units. The basic, fundamental architecture remains the same.*

Elements of Parallel Computers

- **Hardware**
  - Multiple Levels of Parallelism (ILP, DLP, TLP)
  - Multiple Memories
  - Interconnection Network

- **System Software**
  - Parallel Operating System
  - Programming Constructs to Orchestrate Concurrency

- **Application Software**
  - Parallel Algorithms

- ***GOALs***
  - **Achieve Speedup: Tp = Ts/p**
  - **Solve problems requiring a large amount of memory**

Parallel Computing Platforms

- **LOGICAL Organization**

The user's view of the machine as it is being presented via its system software

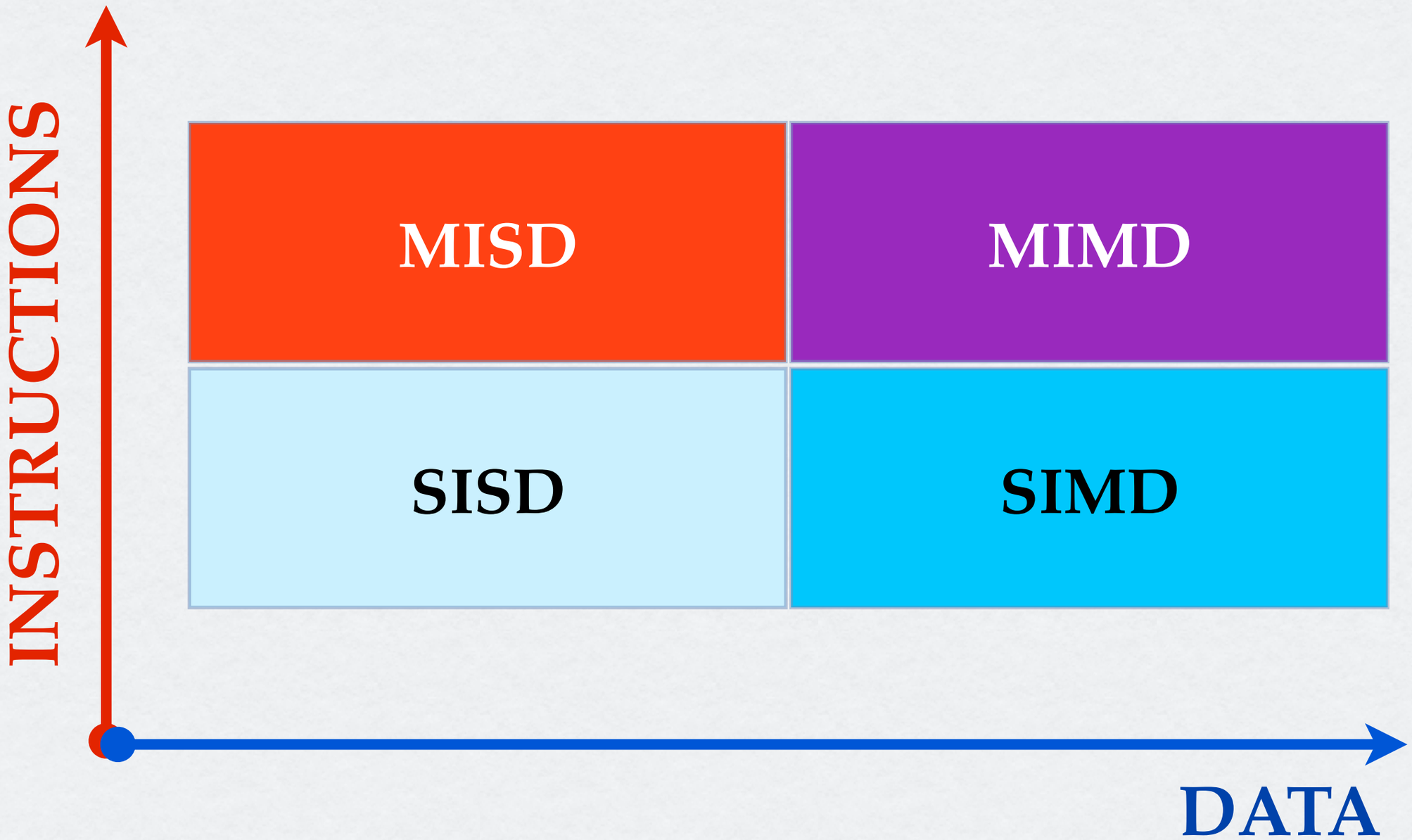
- **PHYSICAL Organization**

The actual hardware architecture

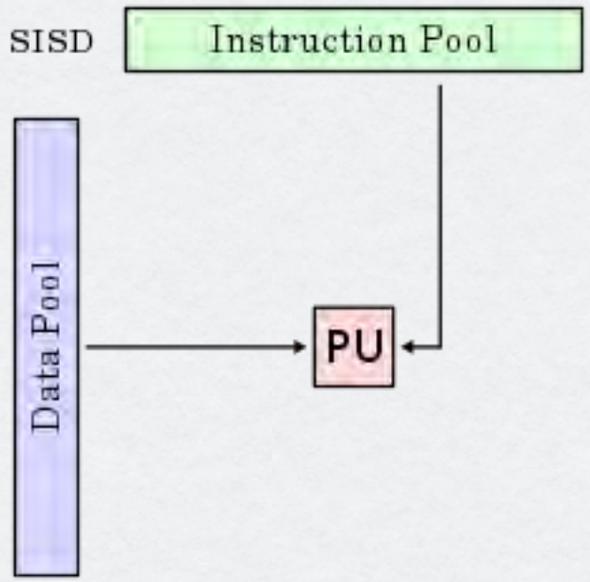
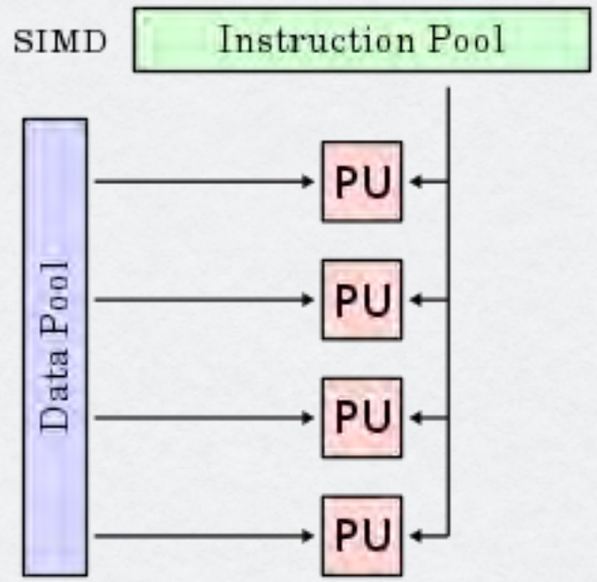
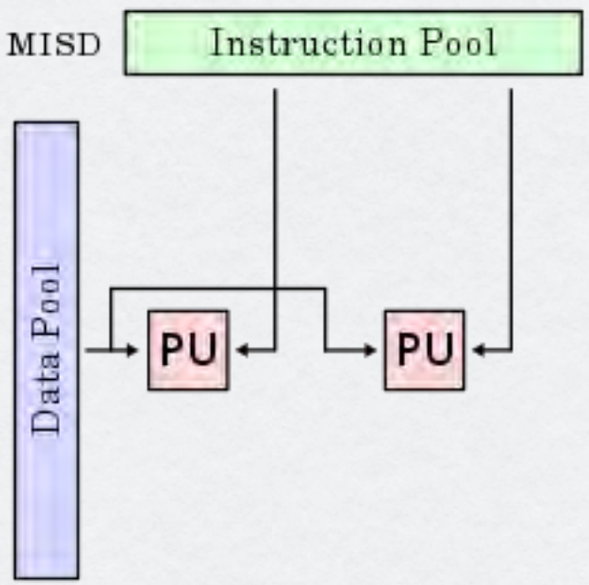
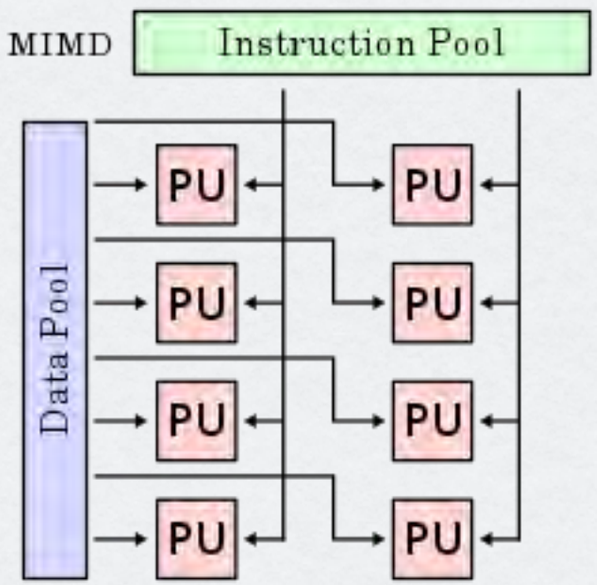
NOTE : Physical Architecture is to a large extent independent of the Logical Architecture

Parallel Computing Platforms

Flynn's Taxonomy (1966)



Flynn's Taxonomy

	Single Data	Multiple Data
Single Instruction		
Multiple Instructions		

Source: Wikipedia

SISD, MIMD, SIMD, SPMD and vector

Data Streams

Flynn's Taxonomy	SINGLE	MULTIPLE
SINGLE	SISD Pentium 4 (an x86 architecture)	SIMD SSE instructions on x86
MULTIPLE	space shuttle flight control computer was MISD (wikipedia)	MIMD Intel Xeon e5345

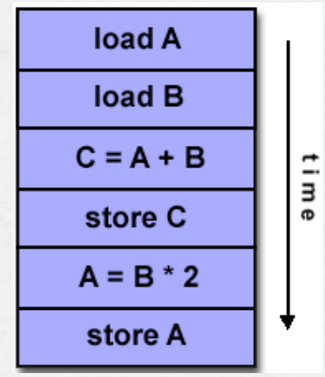
Instruction Streams

SISD (single instruction stream - single data stream): uniprocessor

MISD (multiple instruction stream - single data stream): multiprocessor

SISD

A serial (non-parallel) computer
Single Instruction: Only one instruction stream is being acted on by the CPU during any one clock cycle
Single Data: Only one data stream is being used as input during any one clock cycle
Deterministic execution
This is the oldest and even today, the most common type of computer



UNIVAC1

IBM360

UNIVAC1

CRAY I

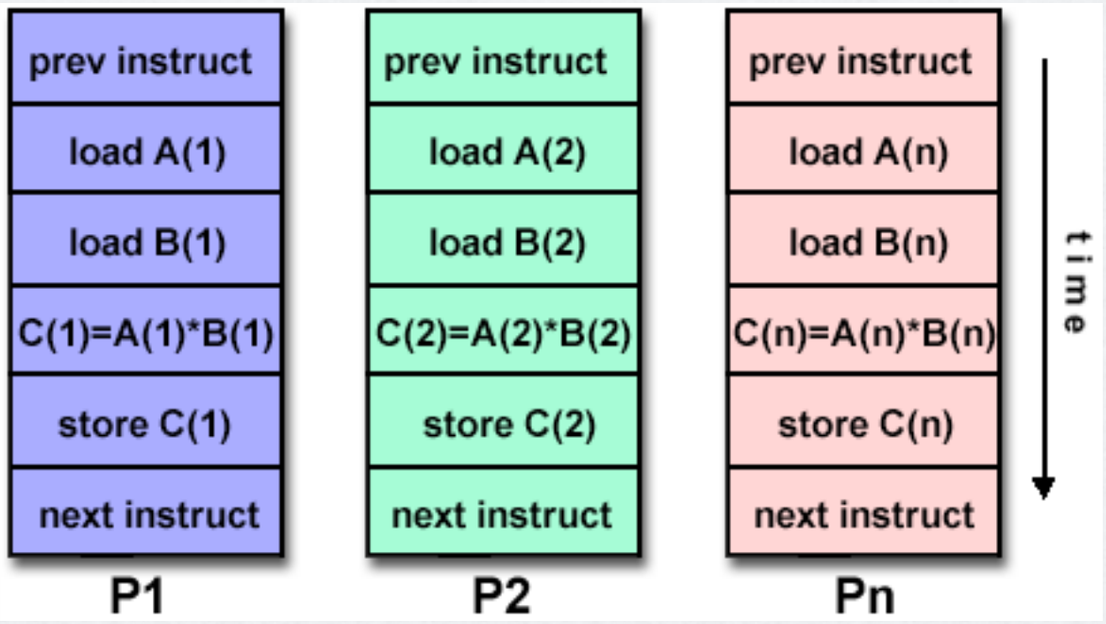
CDC7600

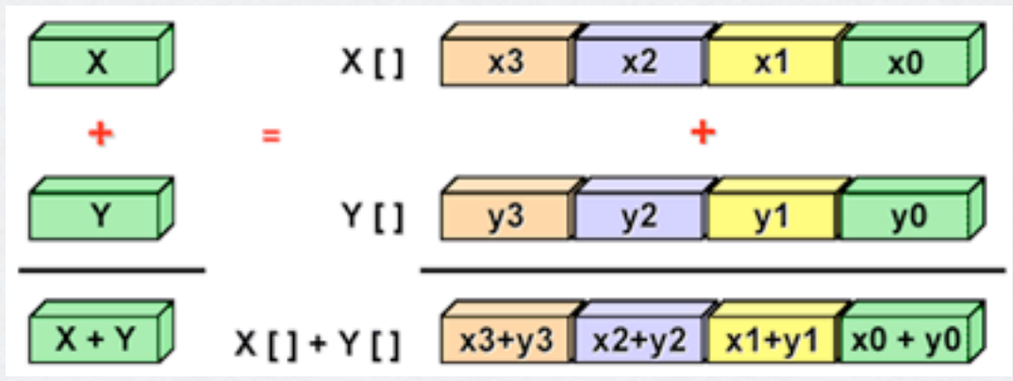
PDP1

Dell Laptop

SIMD

A type of parallel computer
Single Instruction: All processing units execute the same instruction at any given clock cycle
Multiple Data: Each processing unit can operate on a different data element





- **Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.**
- Two varieties: Processor Arrays and Vector Pipelines
- Examples:
  - Processor Arrays: Connection Machine CM-2, MasPar MP-1 & MP-2, ILLIAC IV
  - **Vector Pipelines:** IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10
- **Most modern computers, particularly those with GPUs employ SIMD instructions and execution units**

SIMD Machines

ILLIAC IV

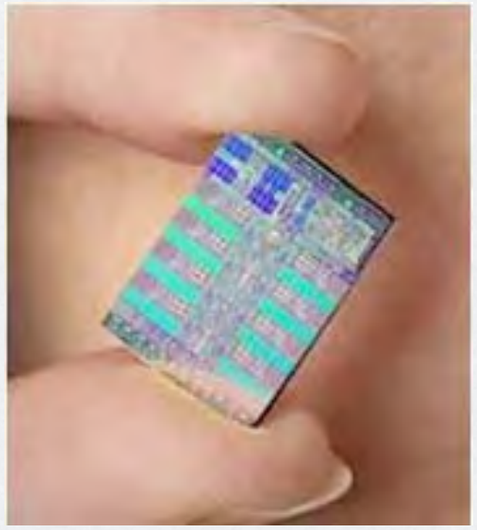
MasPar

Cray X-MP

Cray Y-MP

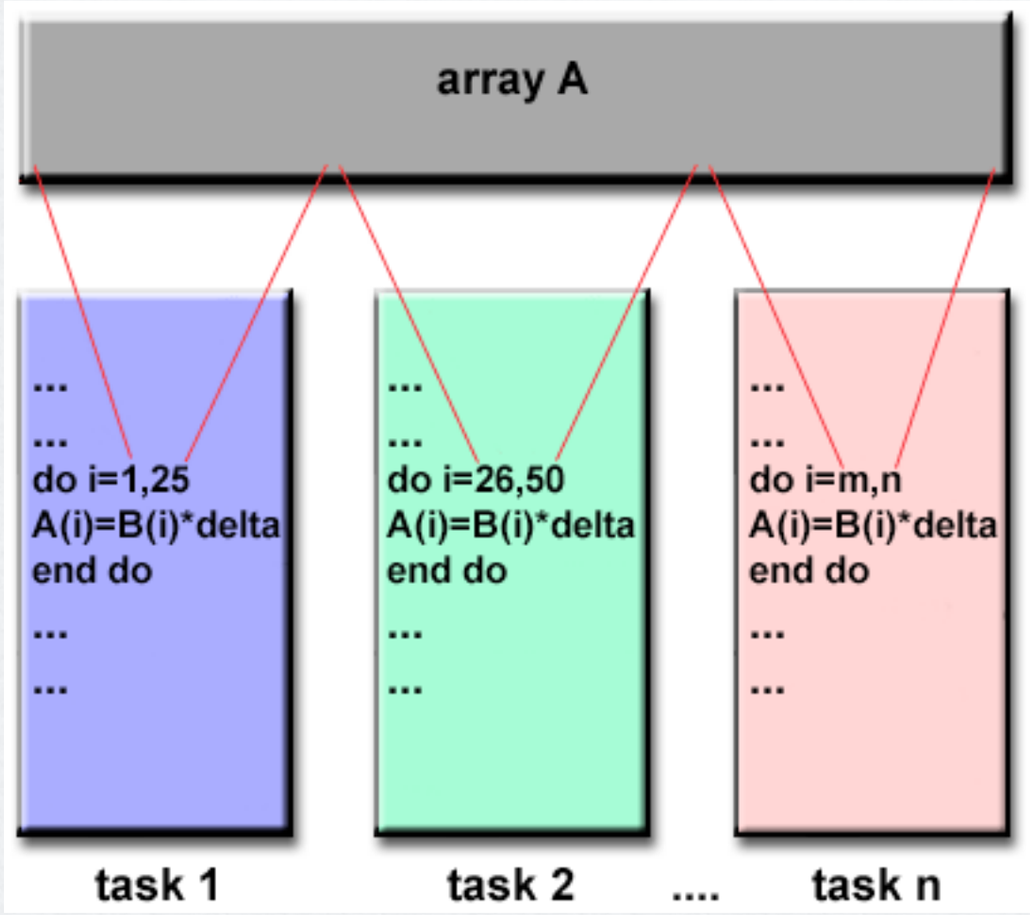
Thinking Machines CM-2

Cell Processor



DATA PARALLEL MODEL -> GPUs

- This model has the following characteristics:
  - Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
  - A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
  - Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".



- On **shared memory** architectures, all tasks may have access to the data structure through global memory.
- On **distributed memory** architectures the data structure is split up and resides as "chunks" in the local memory of each task.

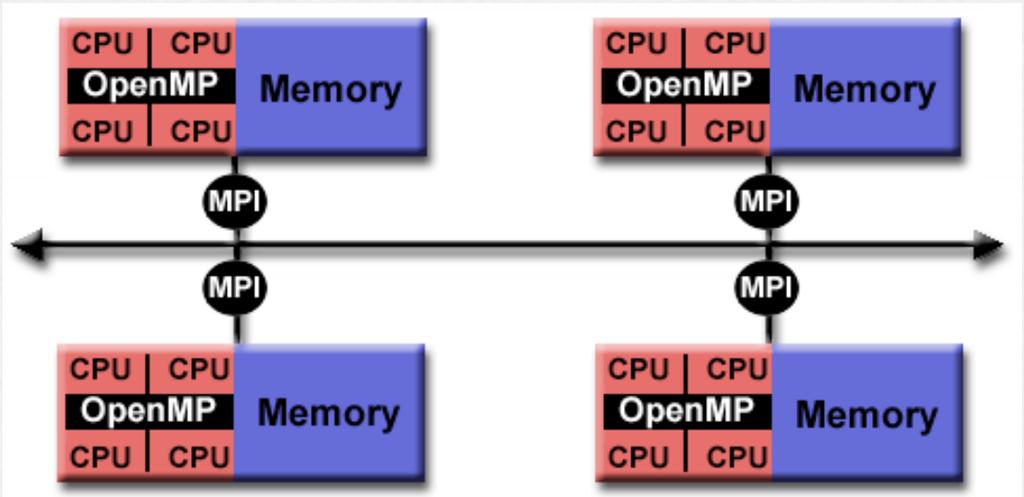
Data Level Parallelism

- Lots of identically structured data
- SIMD has problems with switch statements where each execution unit must perform a different operation on its data, depending on what data it has

HYBRID MODEL (most common today)

- **A hybrid model combines more than one of the previously described programming models.**

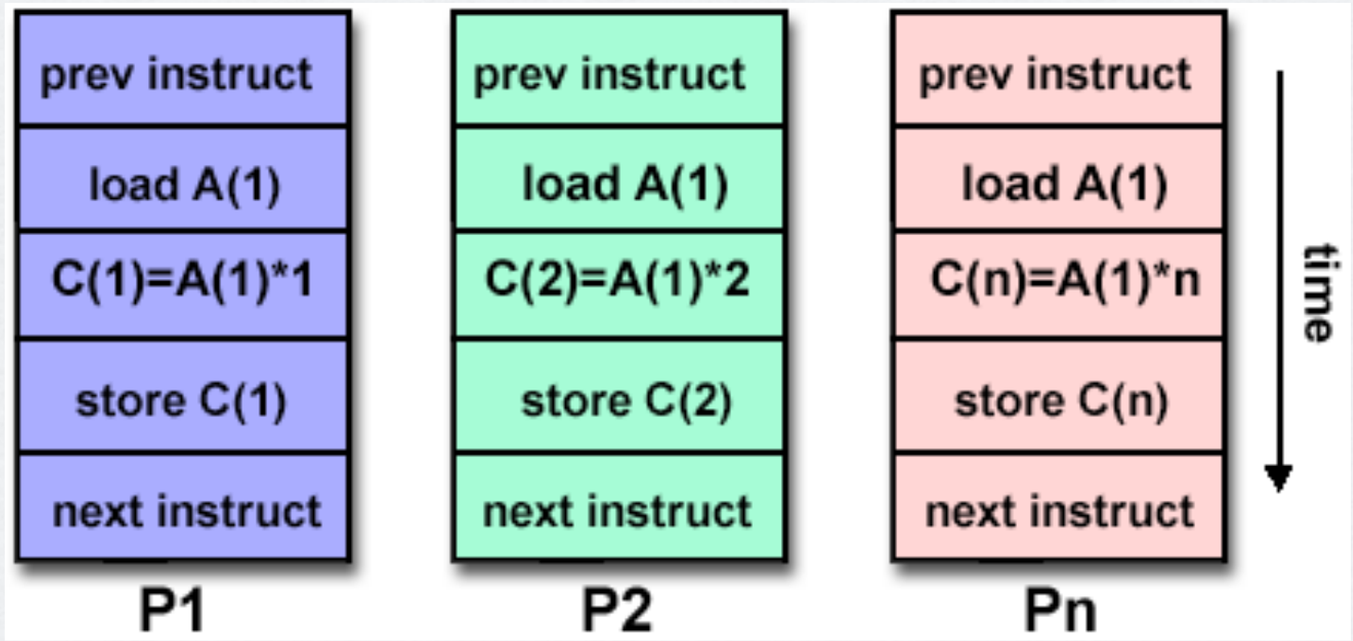
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with the threads model (OpenMP).
  - Threads perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI



- This hybrid model lends itself well to the increasingly common hardware environment of clustered multi/many-core machines.
- Another similar and increasingly popular example of a hybrid model is using MPI with GPU (Graphics Processing Unit) programming.
  - GPUs perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI

MISD

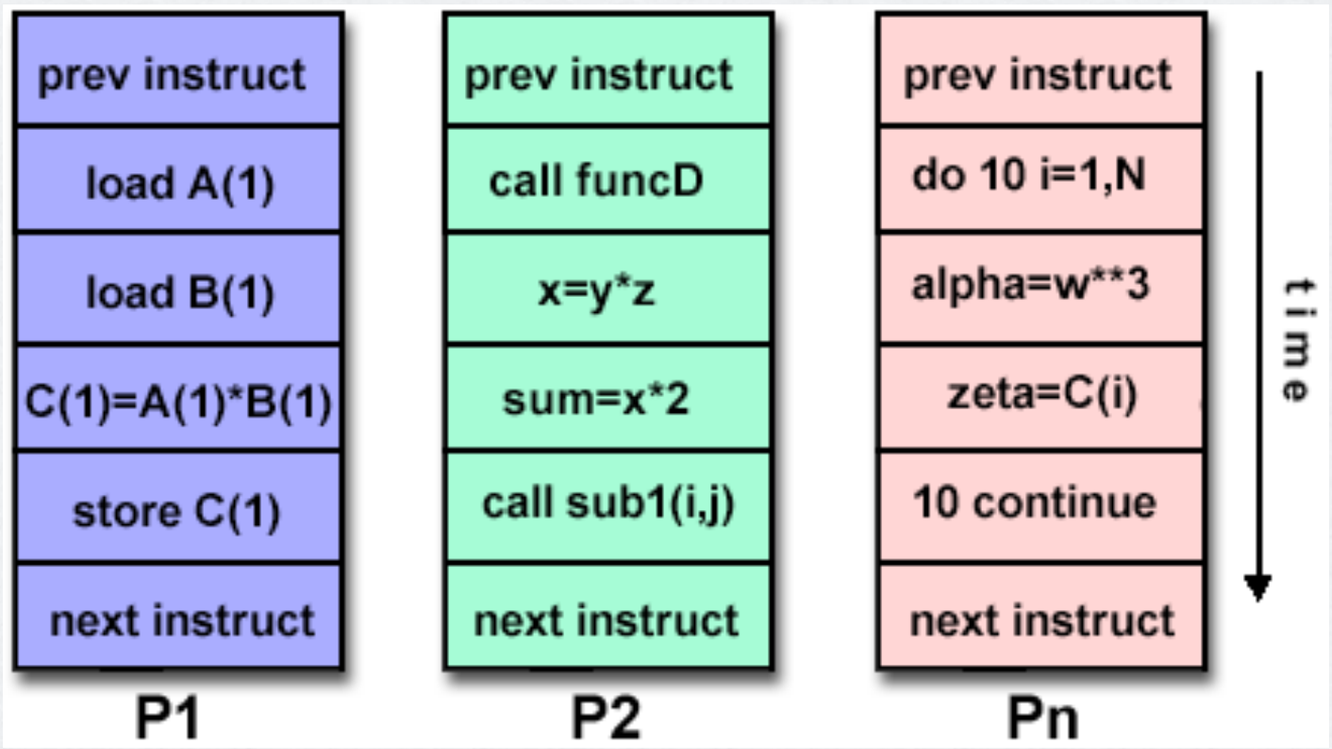
Multiple Instruction: Each processing unit operates on the data independently via separate instruction streams.
Single Data: A single data stream is fed into multiple processing units.



- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).
- Some conceivable uses might be:
  - multiple frequency filters operating on a single signal stream
  - multiple cryptography algorithms attempting to crack a single coded message.
  - stochastic simulations

MIMD

Multiple Instruction: Every processor may be executing a different instruction stream
Multiple Data: Every processor may be working with a different data stream



- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- **Currently, the most common type of parallel computer - most modern supercomputers fall into this category.**
- Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.
- Note: many MIMD architectures also include SIMD execution sub-components

MIMD Machines

IBM POWER5

HP/Compaq Alphaserver

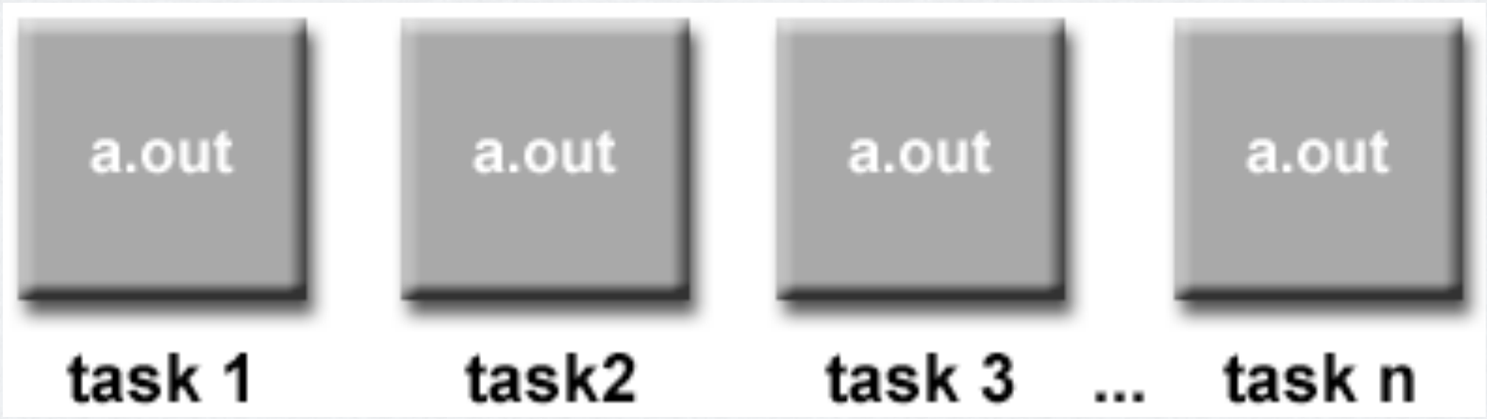
Intel IA32

AMD Opteron

Cray XT3

IBM BG/L

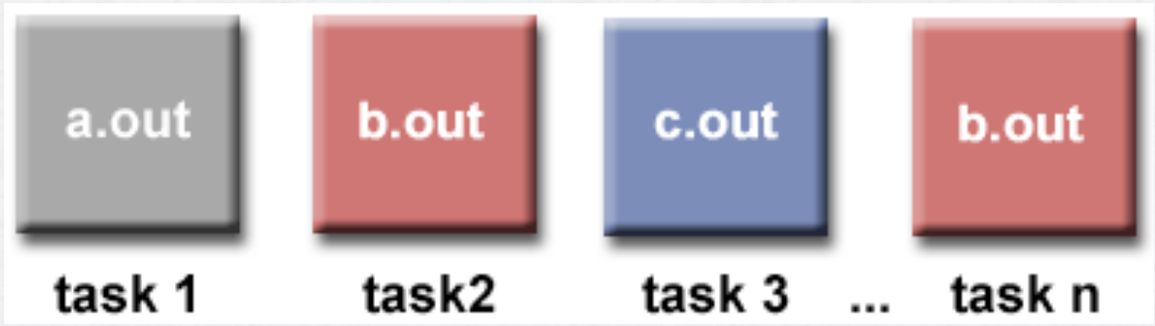
Single Program Multiple Data (SPMD):



- SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.

- SINGLE PROGRAM: All tasks execute their copy of the same program simultaneously. This program can be threads, message passing, data parallel or hybrid.
- MULTIPLE DATA: All tasks may use different data
- SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.
- The SPMD model, using message passing or hybrid programming, is probably the most commonly used parallel programming model for multi-node clusters.

Multiple Program Multiple Data (MPMD):



- Like SPMD, MPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.

- MULTIPLE PROGRAM: Tasks may execute different programs simultaneously. The programs can be threads, message passing, data parallel or hybrid.
- MULTIPLE DATA: All tasks may use different data
- MPMD applications are not as common as SPMD applications, but may be better suited for certain types of problems, particularly those that lend themselves better to functional decomposition than domain decomposition (see Partioning).

SISD, MIMD, SIMD, SPMD and vector

- MIMD: Writing a single program (with **IF**S for different processors) is the Single Program Multiple Data paradigm

- **SIMD** computers operate on **vectors of data**: for example a single SIMD instruction may add 64 numbers by sending 64 data streams to 64 ArithmeticLogicalUnits (ALUs) to form 64 sums within a clock cycle.

➡ all parallel execution units are synchronized and they all respond to a single instruction that emanates from a single program counter (PC)

- Works best when dealing with arrays in `for loops`

Vector Architectures < SIMD

- Vector Architecture: An older interpretation of SIMD
- Rather than having 64 ALUs perform 64 additions simultaneously the vector architectures pipelined the ALU to get good performance at lower cost.

- **BASIC PHILOSOPHY**
  - Collect data from memory
  - Put them in order into a large set of registers
  - Operate on them sequentially on registers
  - Write results back to memory

- **KEY FEATURE:** set of vector registers

Vector Instructions

- Suppose we extend the MIPS instruction set architecture with vector instructions and vector registers.
  - Vector operations, append the letter **‘v’** at the end
  - **`lv` (vector load)** and **`sv` (vector store)**:
    - they load/store an **entire vector** of double precision data.
      - One operand: vector register to be loaded/stored.
      - Other operand: starting address of the vector in memory.

MIPS vs Vectors

MIPS vs Vector for **`Y=a*X+Y`**

`X`, **`Y`** are vectors of 64 double precision FP numbers initially in memory (starting addresses of **`X`** and **`Y`** are **`$S0`** and **`$S1`**)

`a` is a scalar double precision variable

MIPS vs Vectors

MIPS code

```
      l.d   $f0, a($sp)        ;load scalar a
      addiu r4,  $s0,   #512   ;upper bound of load

loop: l.d   $f2, 0($s0)        ; load x(i)
      mul.d $f2, $f2,   $f0    ; a*x(i)
      l.d   $f4, 0($s1)        ; load y(i)
      add.d $f4, $f4,   $f2    ; a*x(i) + y(i)
      s.d   $f4, 0($s1)        ; store into y(i)
      addiu $s0, $s0,   #8     ;increment x index

      addiu $s1, $s1,   #8     ;increment y index

      subu  $t0, r4,    $s0    ; compute bound
      bne   $to, $zero, loop   ; check if done
```

Vector code

```
l.d      $f0, a($sp)        ; load scalar a
lv       $v1, 0($s0)        ; load vector x
mulvs.d  $v2, $v1,   $f0    ; vector-scalar
                            ;multiply
lv       $v3, 0($s1)        ; load vector y
addv.d   $v4, $v2,   $v3    ; add y to product
sv       $v4, 0($s1)        ; store the result
```

MIPS vs Vectors

- The vector code greatly reduces the dynamic instructions bandwidth, executing in only 6 instructions vs. ~600 for MIPS
    - Vector operations work on 64 elements
    - Overhead instructions in MIPS are not there in vector code
- Reduced instructions fetched and executed saves power
- Reduced frequency of pipelined hazards
    - In MIPS every `add.d` must wait for a `mul.d` and every `s.d` must wait for the `add.d`
    - On the vector processor, each vector instruction will only stall for the first element in each vector and then subsequent elements will flow smoothly down the pipeline.
    - Pipeline stalls are required only once per vector operation, rather than once per vector element

NOTE: Pipeline stalls can be reduced in MIPS by loop unrolling.
Large difference in instruction bandwidth cannot be reduced

Vector vs. Scalar

- **Single vector instruction** specifies a great deal of work - it is **equivalent to executing an entire loop**. The instruction fetch & decode bandwidth needed is dramatically reduced

- By using vector instructions, the compiler or programmer indicates that the computation of each result in the vector is independent of other results in the same vector, so hardware does not have to check for data hazards within a vector instruction.

- **Vector compilers better for data-level parallelism.**

Vector vs. Scalar

- Checking for data hazards between two vector instructions once per vector operand, not once for every element within the vectors. Reduced checking can save also power.
- Vector instructions that access memory have a known access pattern. If the **vector's elements are all adjacent, then fetching the vector from a set of heavily interleaved memory banks works very well.** The **cost of latency** to main memory is seen only once for the entire vector, rather than once for each word of the vector.

Vector vs. Scalar

- Control hazards in loops are non-existent as the vector instructions are pre-determined.

- Savings in instruction bandwidth & hazard checking plus the efficient use of memory bandwidth give vector architectures advantages in power & energy vs. scalar architectures.

GPUs (in brief)

- GPUs are accelerators that supplement a CPU, so that they do not need to be able to perform all the tasks of a CPU.
  So a hybrid system can do both tasks necessary. CPU-GPU are heterogeneous multi-processors.

- Programming interfaces to GPUs are high-level Application Programming Interfaces, such as OpenGL

GPUs (in brief)

- Graphics processing: drawing vertices of 3D geometry primitives such as lines and triangles and shading or rendering pixel fragments of geometric primitives. Video games draw 20 to 30 times as many pixels as vertices.
  - Each vertex can be drawn independently and each pixel fragment can be rendered independently.
    To render many millions of pixels per frame the GPU evolved to execute many threads from vertex and pixel shader programs in parallel.

- Great deal of data level parallelism

CPU vs GPU: Key Differences

- The GPUs do not rely on multi-level caches to overcome the long latency to memory as do CPUs. Instead GPUs rely on having enough threads to hide the long latency to memory.
  - Between the time of a memory request and the time the data arrives, the GPUs execute ~10^5 threads that are independent of that request.

- GPUs rely on excessive parallelism to obtain high performance, implementing many parallel processors and many parallel threads

CPU vs GPU: Key Differences

- The GPU main memory is thus oriented towards bandwidth rather than latency.
  There are separate DRAM chips that are wider and have higher bandwidth than DRAM chips for CPUs.

- GPUs have smaller main memories (~1 GB)

- For GPU computing it is important to include the time to transfer the data between the CPU memory and the GPU memory as the GPU is a co-processor.

CPU vs GPU: Key Differences

- Newer GPUs are becoming more like multicore machines.
- In contrast to vector architectures, that rely on

Vector

vector compilers to recognize data level parallelism at compile time and generate vector instructions,

GPU

hardware implementations of Tesla GPUs, discover data-level parallelism among threads at runtime.

Introduction to Multi-processor Network Technologies

➡ Multicore chips requires **networks on chips** to connect the cores together

Network Performance:

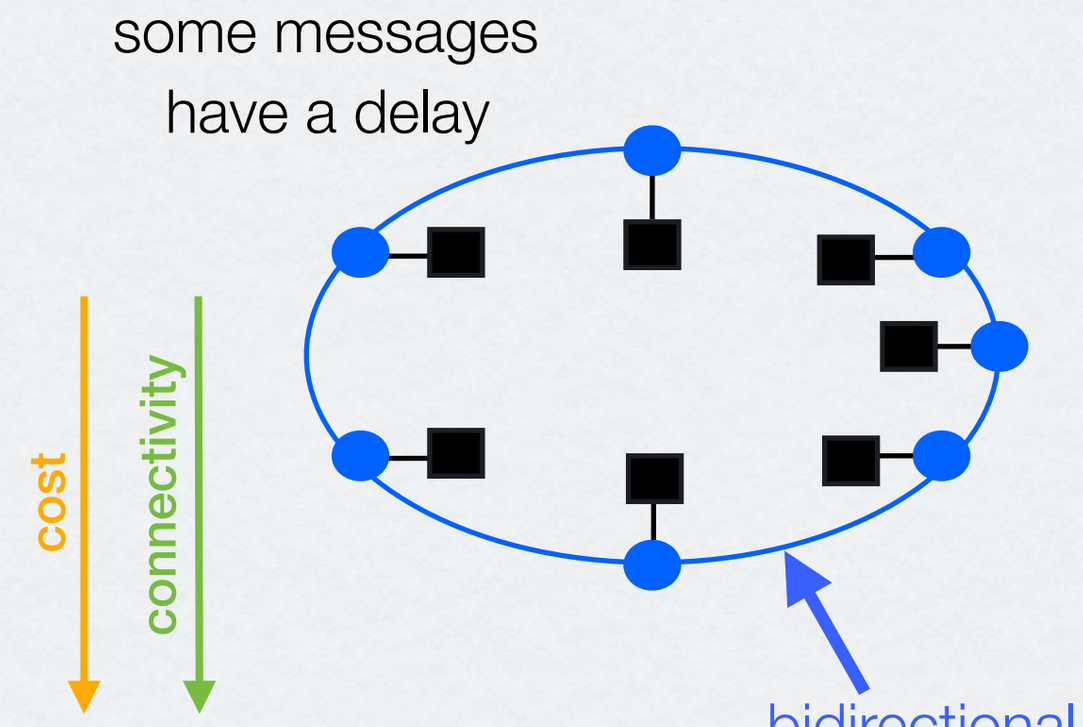
- **latency** of an unloaded network to send/receive message
- **throughput** (max # of messages that can be transmitted in a given time period)
- **delays** caused by contention for a portion of the network
- variability due to **patterns of communications**
- **fault tolerance** (there may be broken components)
- **power efficiency** of different organizations may trump other concerns

Topologies

1. The Ring

➡ it is the first improvement over a bus

Bisection bandwidth=2



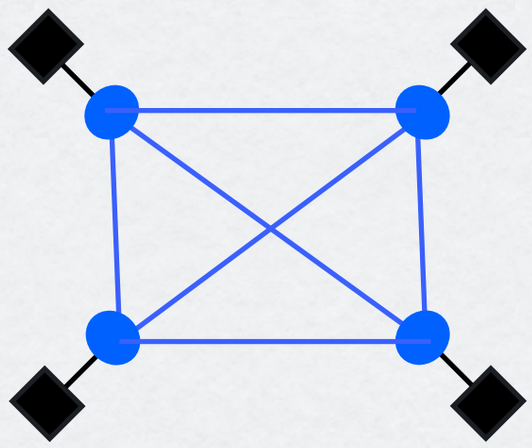
2. Fully Connected Network

- A network that connects processor-memory nodes, by supplying a dedicated communication link between every node.

➡ total network bandwidths: $\frac{P \times (P-1)}{2}$

➡ the bisection bandwidth is: $\left(\frac{P}{2}\right)^2$

Number of links needed to be broken to create two partitions



Network Bandwidth

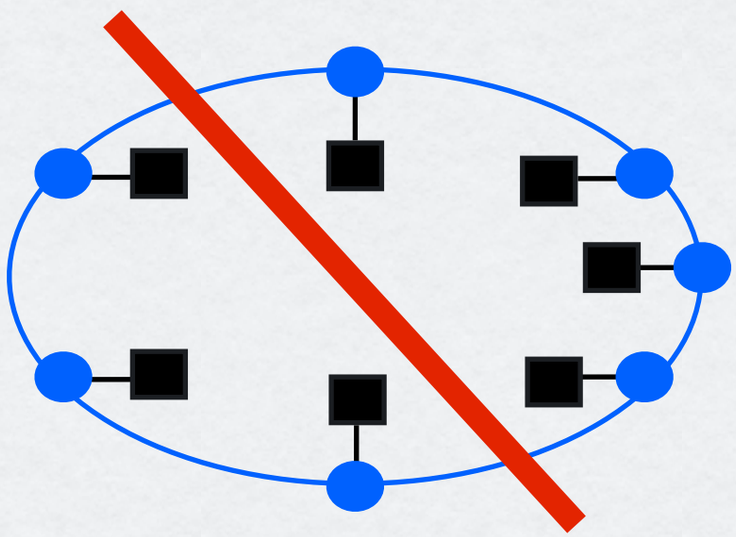
- Bandwidth of each link multiplied by the number of links
- This is the **very best case**
- Examples
  - Ring: bandwidth is P (# of processors)
  - Full connection: $\frac{P \times (P-1)}{2}$
- For a bus thus is 2x bandwidth of bus or 2x bandwidth of a link

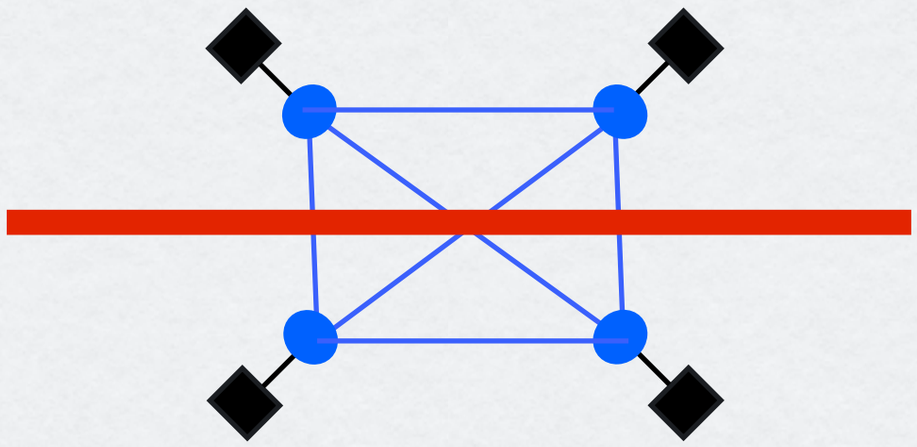
Bisection bandwidth

- **Worst case**

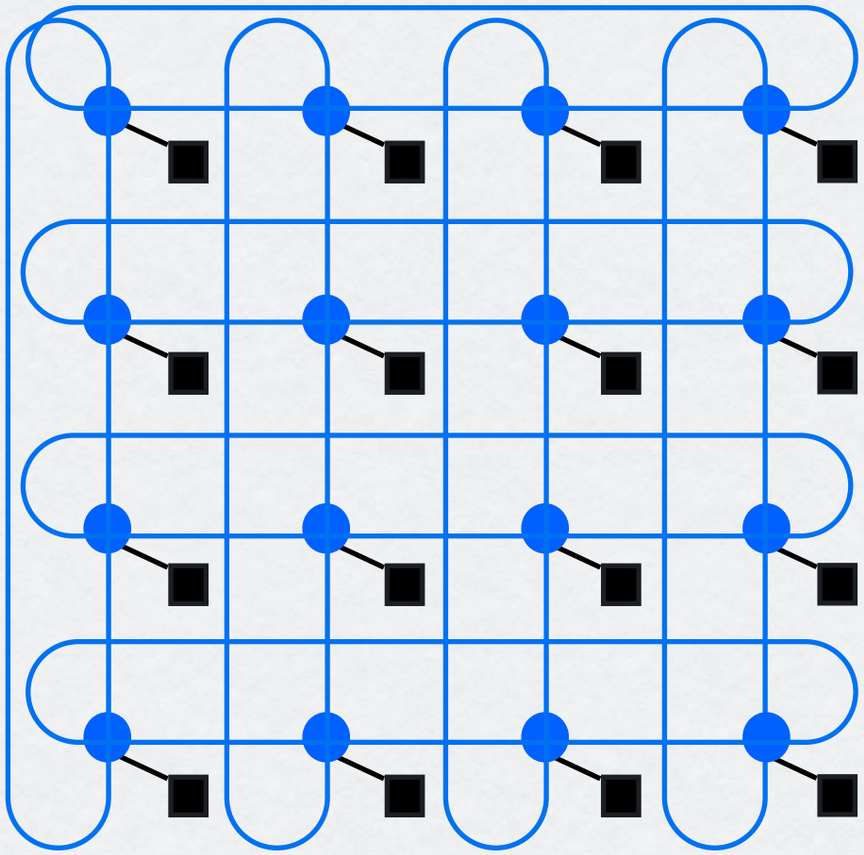
- Divide the machine in two parts, each with half the nodes. Then sum the bandwidth of the links that cross that imaginary dividing line

- for the ring it is 2x link bandwidth
- for the bus it is 1x link bandwidth
- for the full connection $(p/2)^2$



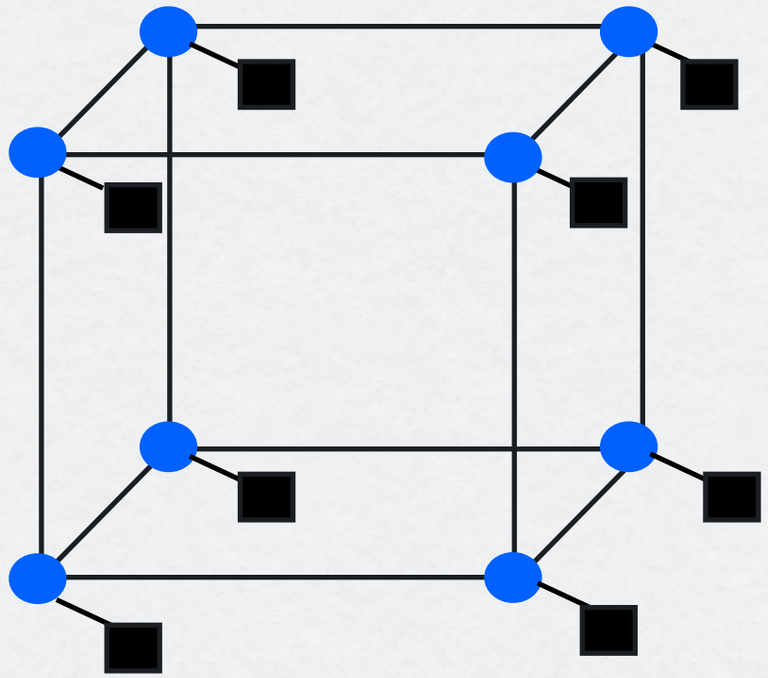


Other Network Topologies



2D grid or 16 node mesh

this is a 2D torus



n-cube

n=3 ~ 8 nodes

Designing Parallel Programs

Parallel Program Design : Understand the Problem I

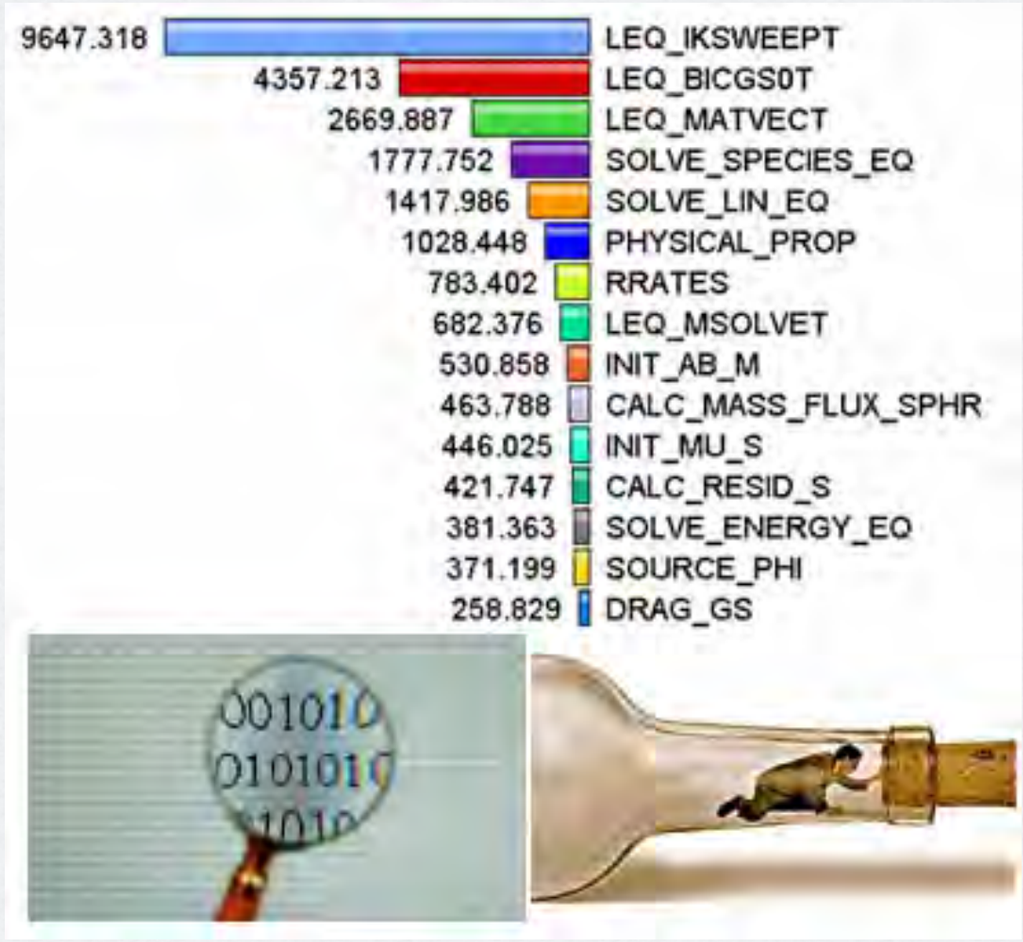
- The first step in developing parallel software is to first understand the problem that you wish to solve in parallel.
- Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.

  - Example of Parallelizable Problem:
  - **Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.**
  - This problem is able to be solved in parallel. Each of the molecular conformations is independently determinable. The calculation of the minimum energy conformation is also a parallelizable problem.

  - Example of a Non-parallelizable Problem:
  - **Calculation of the Fibonacci series (0,1,1,2,3,5,8,13,21,...) by use of the formula: F(n) = F(n-1) + F(n-2)**
  - This is a non-parallelizable problem because the calculation of the Fibonacci sequence as shown would entail dependent calculations rather than independent ones. The calculation of the F(n) value uses those of both F(n-1) and F(n-2). These three terms cannot be calculated independently and therefore, not in parallel.

Parallel Program Design : Understand the Problem II

- Identify the program's ***hotspots***:
  - Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
  - Profilers and performance analysis tools can help here
  - Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.



- Identify ***bottlenecks*** in the program
  - Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
  - May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas

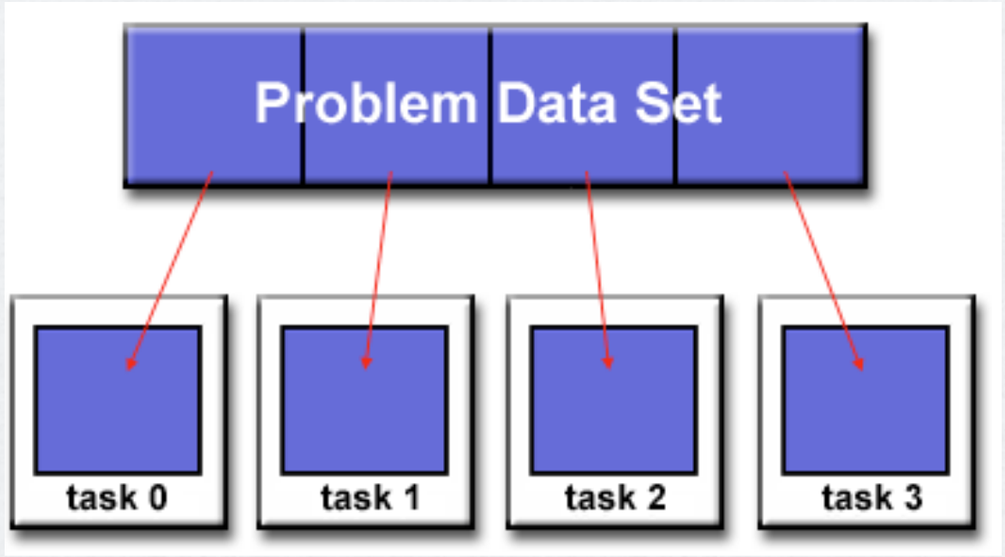
- Identify **inhibitors to parallelism**. One common class of inhibitor is ***data dependence***, as demonstrated by the Fibonacci sequence above.
- Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.

Decomposition/Partitioning

- One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as **decomposition or partitioning.**
- Two basic ways to partition work among parallel tasks: ***domain decomposition*** and ***functional decomposition***.

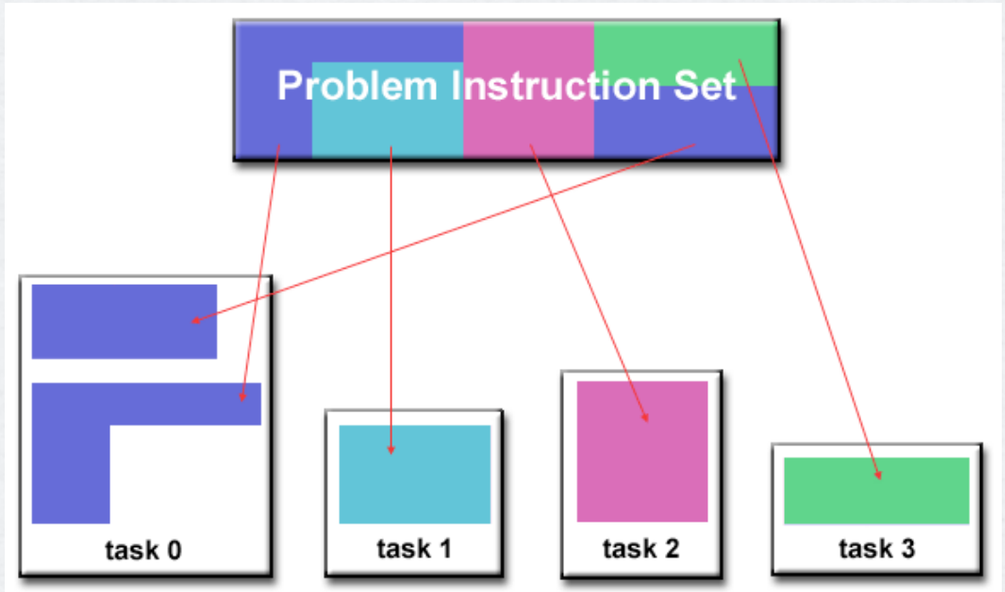
Domain Decomposition:

- In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of of the data.



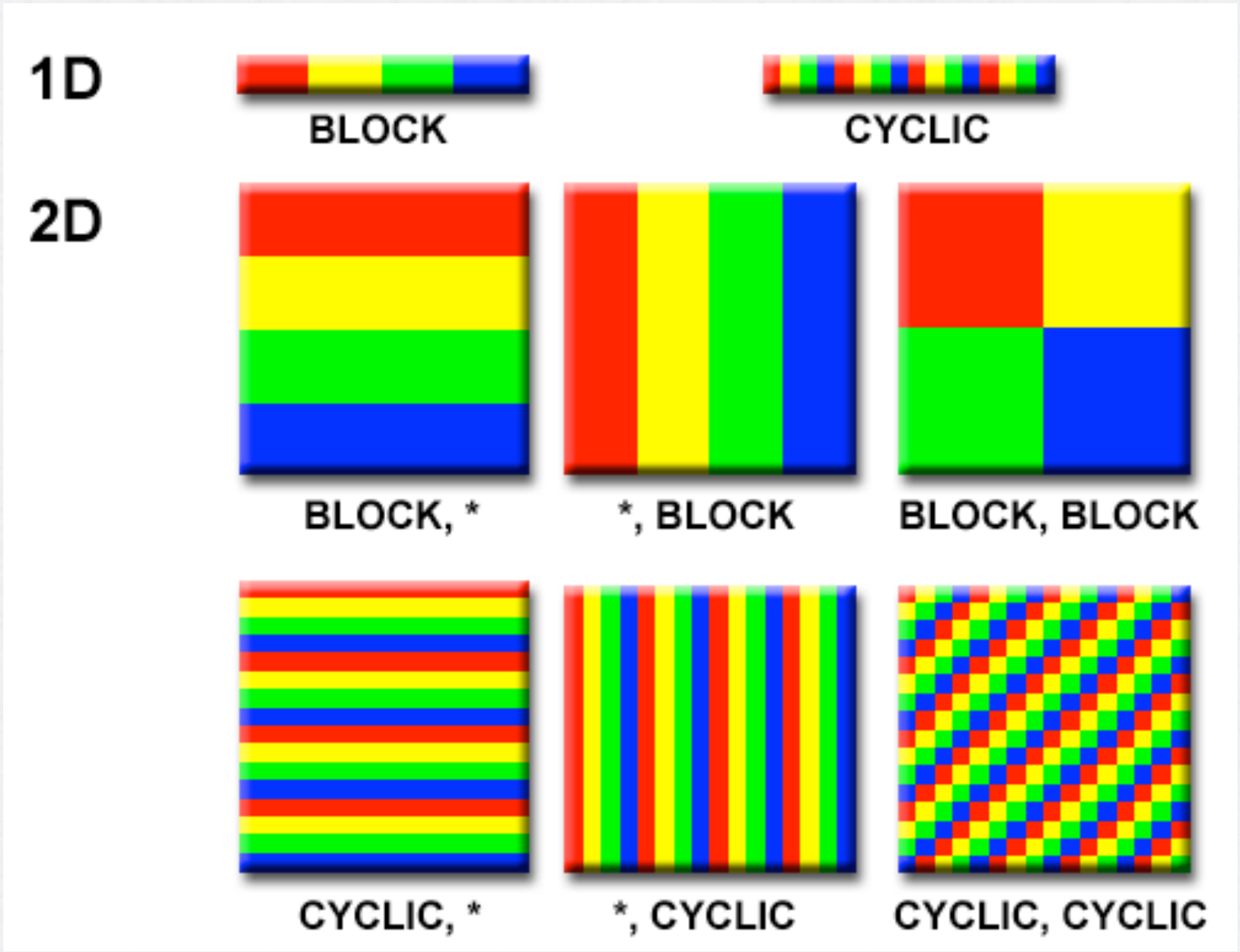
Functional Decomposition:

- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.



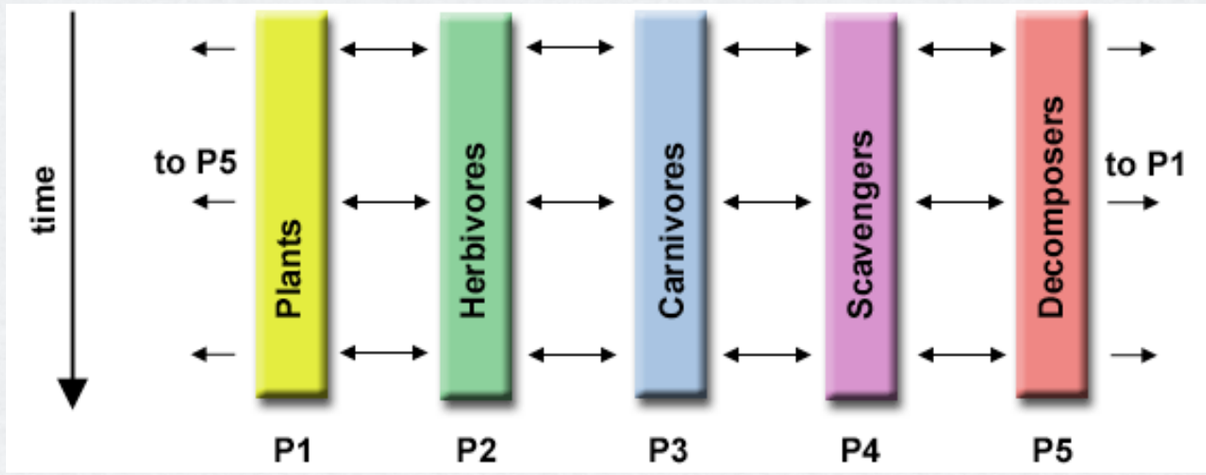
Domain Decomposition

There are different ways to partition data:

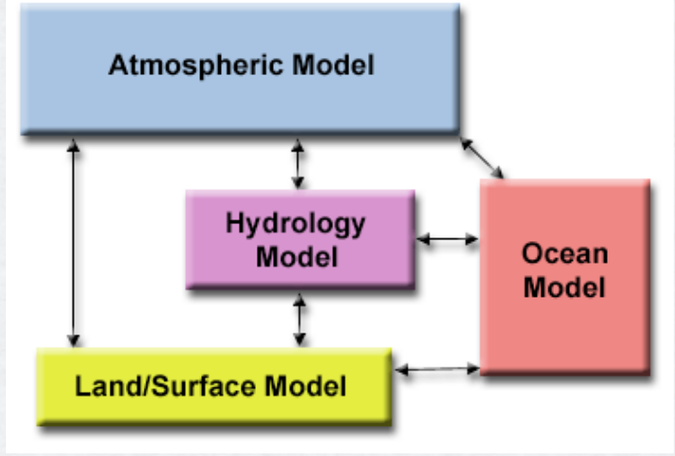


Functional Decomposition

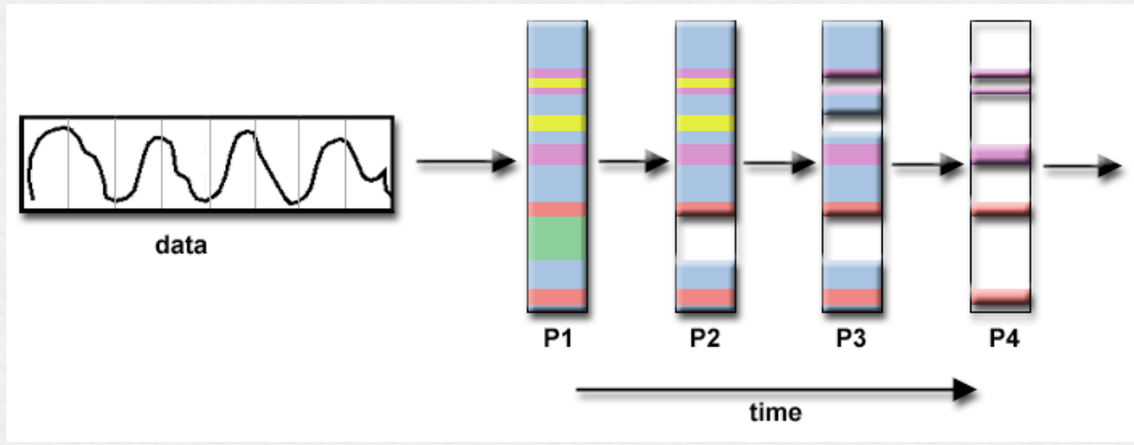
Ecosystem Modeling
Each program calculates the population of a given group, where each group's growth depends on that of its neighbors. As time progresses, each process calculates its current state, then exchanges information with the neighbor populations. All tasks then progress to calculate the state at the next time step.



Climate Modeling
Each model component can be thought of as a separate task. Arrows represent exchanges of data between components during computation: the atmosphere model generates wind velocity data that are used by the ocean model, the ocean model generates sea surface temperature data that are used by the atmosphere model, and so on.



Signal Processing
An audio signal data set is passed through four distinct computational filters. Each filter is a separate process. The first segment of data must pass through the first filter before progressing to the second. When it does, the second segment of data passes through the first filter. By the time the fourth segment of data is in the first filter, all four tasks are busy.



COMMUNICATION

Who Needs Communications?

- The need for communications between tasks depends upon your problem:

- **You DON'T need communications**
  - Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data. For example, imagine an image processing operation where every pixel in a black and white image needs to have its color reversed. The image data can easily be distributed to multiple tasks that then act independently of each other to do their portion of the work.
  - These types of problems are often called ***embarrassingly parallel*** because they are so straight-forward. Very little inter-task communication is required.

- **You DO need communications**
  - Most parallel applications are not quite so simple, and do require tasks to share data with each other. For example, a 3-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data. Changes to neighboring data has a direct effect on that task's data.

COMMUNICATION - FACTORS I

- **Cost of communications**
  - Inter-task communication virtually always implies overhead.
  - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
  - Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
  - Competing communication traffic can saturate the available network bandwidth, aggravating performance

- **Latency vs. Bandwidth**
  - ***latency*** is the time it takes to send a minimal (0 byte) message from point A to point B. Expressed as microseconds.
  - ***bandwidth*** is the amount of data that can be communicated per unit of time. Expressed as Mb/sec or Gb/sec.
  - Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

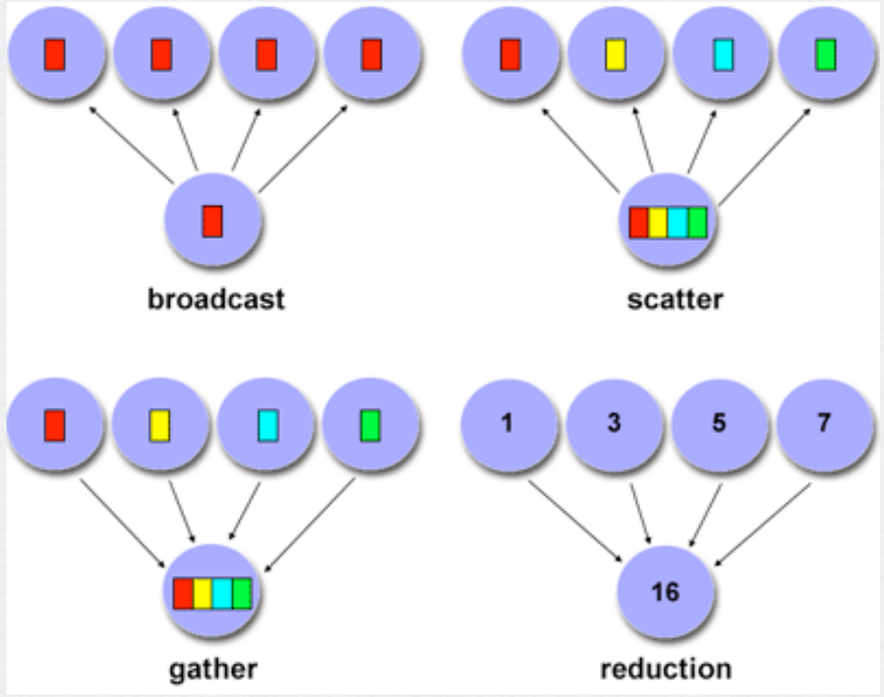
- **Visibility of communications**
  - With the Message Passing Model, communications are explicit and generally quite visible and under the control of the programmer.
  - With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory architectures. The programmer may not be able to know how inter-task communications are accomplished.

- **Synchronous vs. asynchronous communications**
  - Synchronous communications require some type of "handshaking" between tasks that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer.
  - Synchronous communications are often referred to as ***blocking*** as other work must wait until they have completed.
  - Asynchronous communications allow tasks to transfer data independently from one another. E.g. task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 receives the data doesn't matter.
  - Asynchronous communications are often referred to as ***non-blocking*** communications since other work can be done while the communications are taking place.
  - Interleaving computation with communication is the single greatest benefit for using asynchronous communications.

COMMUNICATION - FACTORS II

Scope of communications

- Knowing which tasks must communicate with each other is critical during the design stage of a parallel code. Both of the two scopings described below can be implemented synchronously or asynchronously.
- ***Point-to-point*** - involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
- ***Collective*** - involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective. Some common variations (there are more):



Efficiency of communications

- Very often, the programmer will have a choice with regard to factors that can affect communications performance. Only a few are mentioned here.
- Which implementation for a given model should be used? Using the Message Passing Model as an example, one MPI implementation may be faster on a given hardware platform than another.
- What type of communication operations should be used? As mentioned previously, asynchronous communication operations can improve overall program performance.
- Network media - some platforms may offer more than one network for communications. Which one is best?

this is only a partial list of things to consider!!!

SYNCHRONIZATION

Types of Synchronization:

- **Barrier**
  - Usually implies that all tasks are involved
  - Each task performs its work until it reaches the barrier. It then stops, or "blocks".
  - When the last task reaches the barrier, all tasks are synchronized.
  - What happens from here varies. Often, a serial section of work must be done. In other cases, the tasks are automatically released to continue their work.

- **Lock / semaphore**
  - Can involve any number of tasks
  - Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
  - The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
  - Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.
  - Can be blocking or non-blocking

- **Synchronous communication operations**
  - Involves only those tasks executing a communication operation
  - When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication. For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.
  - Discussed previously in the Communications section.

DATA DEPENDENCIES

A ***dependence*** exists between program statements when the order of statement execution affects the results of the program.
A ***data dependence*** results from multiple use of the same location(s) in storage by different tasks.
Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism.

EXAMPLE : Loop carried data dependence

```
DO 500 J = MYSTART,MYEND
   A(J) = A(J-1) * 2.0
500 CONTINUE
```

The value of A(J-1) must be computed before the value of A(J), therefore A(J) exhibits a data dependency on A(J-1). Parallelism is inhibited.

If Task 2 has A(J) and task 1 has A(J-1), computing the correct value of A(J) necessitates:

- Distributed memory architecture - task 2 must obtain the value of A(J-1) from task 1 after task 1 finishes its computation
- Shared memory architecture - task 2 must read A(J-1) after task 1 updates it

EXAMPLE : Loop Independent data dependence

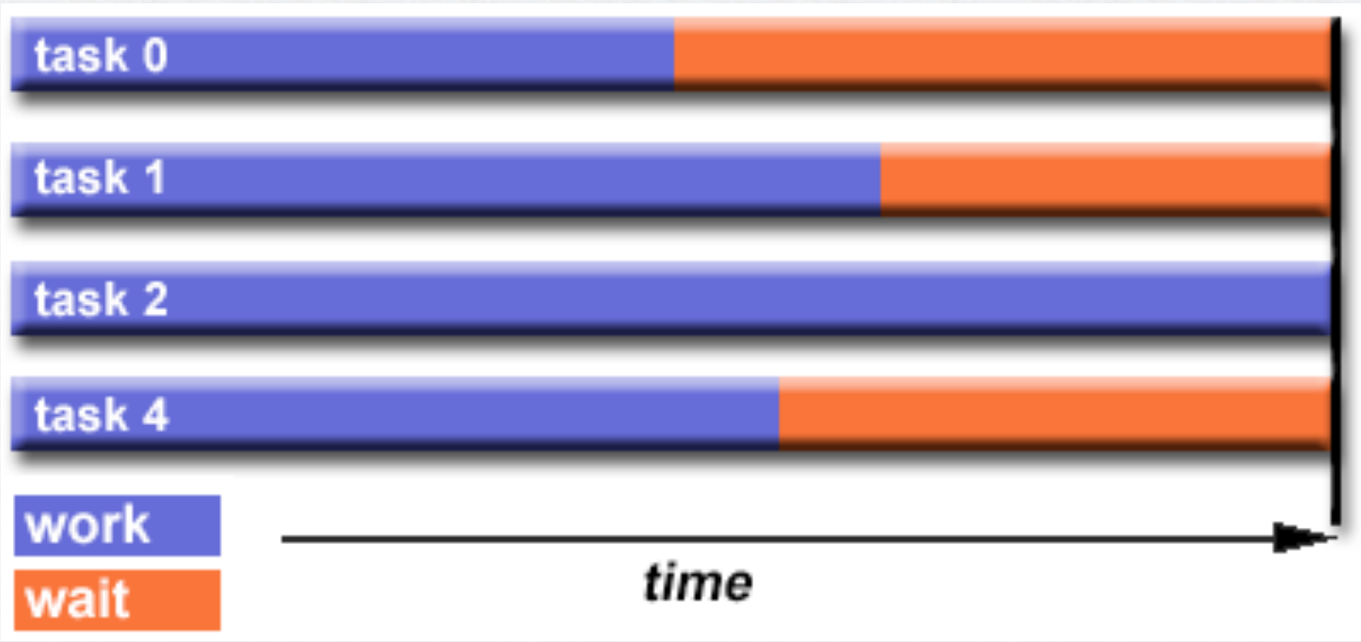
```
task 1          task 2
------          ------

X = 2           X = 4
  .               .
  .               .
Y = X**2        Y = X**3
```

- As with the previous example, parallelism is inhibited. The value of Y is dependent on:
  - Distributed memory architecture - if or when the value of X is communicated between the tasks.
  - Shared memory architecture - which task last stores the value of X.
- Although all data dependencies are important to identify when designing parallel programs, loop carried dependencies are particularly important since loops are possibly the most common target of parallelization efforts.

How to Handle Data Dependencies:

- Distributed memory architectures - communicate required data at synchronization points.
- Shared memory architectures -synchronize read/write operations between tasks.

LOAD BALANCING



- Load balancing refers to the practice of distributing work among tasks so that ***all*** tasks are kept busy ***all*** of the time. It can be considered a minimization of task idle time.

- Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.

(how to achieve) LOAD BALANCING

- **Equally partition the work each task receives**
  - For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
  - For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
  - If a heterogeneous mix of machines with varying performance characteristics are being use be sure to use some type of performance analysis tool to detect any load imbalances.

- **Use dynamic work assignment**
  - Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:
    - Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros".
    - Adaptive grid methods - some tasks may need to refine their mesh while others don't.
    - *N*-body simulations - where some particles may migrate to/from their original task domain to another task's; where the particles owned by some tasks require more work than those owned by other tasks.
  - When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a ***scheduler - task pool*** approach. As each task finishes its work, it queues to get a new piece of work.
  - It may become necessary to design an algorithm which detects and handles load imbalanc as they occur dynamically within the code.

GRANULARITY

Computation / Communication Ratio:

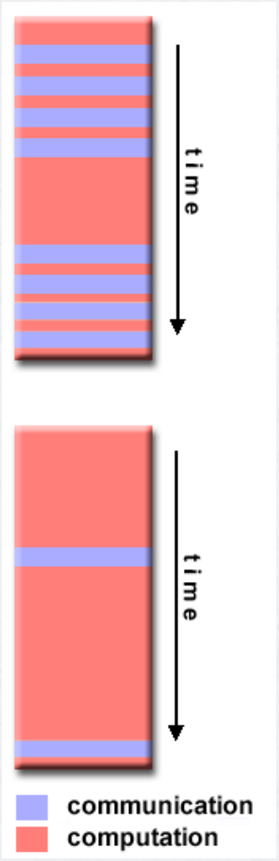
- In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
- Periods of computation are typically separated from periods of communication by synchronization events.

Fine-grain Parallelism:

- Relatively small amounts of computational work are done between communication events
- Low computation to communication ratio
- Facilitates load balancing
- Implies high communication overhead and less opportunity for performance enhancement
- If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.

Coarse-grain Parallelism:

- Relatively large amounts of computational work are done between communication/ synchronization events
- High computation to communication ratio
- Implies more opportunity for performance increase
- Harder to load balance efficiently



Which is Best?

- The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.
- In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
- Fine-grain parallelism can help reduce overheads due to load imbalance.

I/O

The Bad News:

- I/O operations are generally regarded as inhibitors to parallelism
- Parallel I/O systems may be immature or not available for all platforms
- In an environment where all tasks see the same file space, write operations can result in file overwriting
- Read operations can be affected by the file server's ability to handle multiple read requests at the same time
- I/O that must be conducted over the network (NFS, non-local) can cause severe bottlenecks and even crash file servers.

The Good News:

- Parallel file systems are available. For example:
  - GPFS: General Parallel File System for AIX (IBM)
  - Lustre: for Linux clusters (Oracle)
  - PVFS/PVFS2: Parallel Virtual File System for Linux clusters (Clemson/Argonne/Ohio State/others)
  - PanFS: Panasas ActiveScale File System for Linux clusters (Panasas, Inc.)
  - HP SFS: HP StorageWorks Scalable File Share. Lustre based parallel file system (Global File System for Linux) product from HP

- The parallel I/O programming interface specification for MPI has been available since 1996 as part of MPI-2. Vendor and "free" implementations are now commonly available.

I/O Pointers

A few pointers:

- Rule #1: Reduce overall I/O as much as possible
- If you have access to a parallel file system, investigate using it.
- Writing large chunks of data rather than small packets is usually significantly more efficient.
- Confine I/O to specific serial portions of the job, and then use parallel communications to distribute data to parallel tasks. For example, Task 1 could read an input file and then communicate required data to other tasks. Likewise, Task 1 could perform write operation after receiving required data from all other tasks.
- Use local, on-node file space for I/O if possible. For example, each node may have /tmp filespace which can used. This is usually much more efficient than performing I/O over the network to one's home directory.

Performance I

Amdahl's Law


245 DL


Golders Green
13
ASSASSINS CREED
STOP MEN
MOV
LS05 RBF

Why writing fast parallel programs is hard

- Essential to know the hardware to get the best out of software

- Finding enough parallelism (Amdahl's Law)
- Granularity – how big should each parallel task be
- Locality – moving data costs more than arithmetic
- Load balance – don't want 1K processors to wait for one slow one
- Coordination and synchronization – sharing data safely
- Performance modeling/debugging/tuning

➡ All of these things makes parallel programming even harder than sequential programming.

(Parallel) Computing Challenges

- **Multi-core/processor**
  - Reduce communication/synchronization overload
  - **Load imbalance**
  - Cache coherence (different caches with same values)

- **Single Core**
  - Memory hierarchy
  - Instruction synchronization
    - **Arithmetics:** associativity (instruction ordering matters)
    - Advanced instructions: hardware and compilers

- **I/O:** redundant arrays of inexpensive disks (RAID: multiple resources for I/O - goes for fault tolerance as well)

The Difficulty of Parallel Processing Programs

- We must get **efficiency** - else use single processor
- Instruction level parallelism done by the compiler can help (out of order execution,etc)

Challenges

scheduling, load balancing, time for synchronization, overhead for communication between parts

Same for:

(1) 8 reporters writing together on a story
(2) 8 processors working in parallel

NOTE: The more the processors/reporters the harder these problems.

LOAD IMBALANCE

- Load imbalance is the time that some processors in the system are idle due to:
  - insufficient parallelism (during that phase)
  - unequal size tasks
- Examples of the latter
  - adapting to “interesting parts of a domain”
  - tree-structured computations
  - fundamentally unstructured problems
- Algorithm needs to balance load
  - Sometimes can determine work load, divide up evenly, before starting
    - “Static Load Balancing”
  - Sometimes work load changes dynamically, need to rebalance dynamically
    - “Dynamic Load Balancing,” eg work-stealing

Amdahl's Law

- How much can a problem be improved?
- e.g.: We want a speed up of 90x faster with 100 processors

$$\text{Exec time after improvement} = \frac{\text{Exec time affected by improvement}}{\text{Amount of improvement}} + \text{Exec time unaffected}$$

$$\text{Speed-up} = \frac{\text{Exec time before}}{(\text{Exec time before} - \text{Exec time affected}) + \frac{\text{Exec time affected}}{100}}$$

Processors

$$\text{Speed-up} = \frac{1}{(1 - \text{Fraction time affected}) + \frac{\text{Fraction time affected}}{100}}$$

Assumption of perfect load balancing

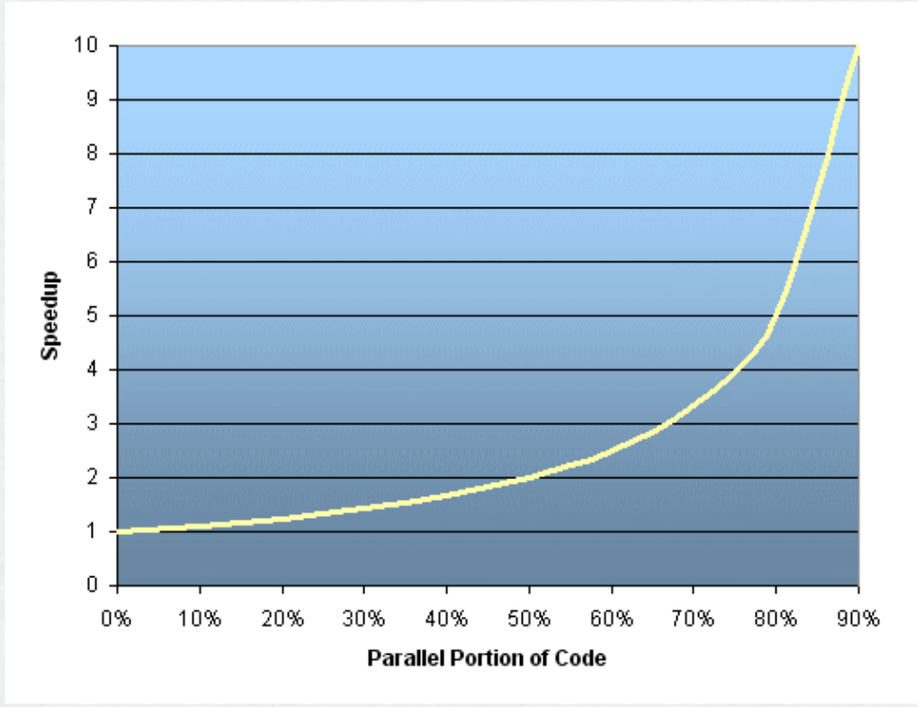
LIMITS AND COSTS

Amdahl's Law:

- **Amdahl's Law** states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

$$\text{speedup} = \frac{1}{\frac{1}{1 - P}}$$

- If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup).
- If all of the code is parallelized, P = 1 and the speedup is infinite (in theory).
- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.



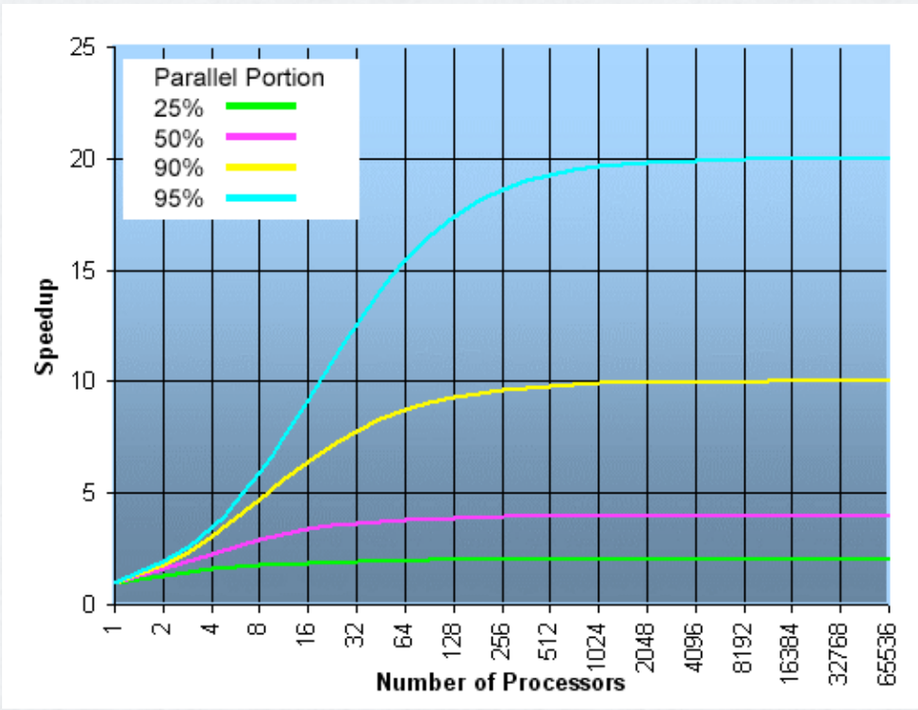
- Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:

$$\text{speedup} = \frac{1}{\frac{P}{N} + S}$$

where P = parallel fraction, N = number of processors and S = 1-P = serial fraction.

- It soon becomes obvious that there are limits to the scalability of parallelism. For example:

	speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02
100000	1.99	9.99	99.90



Amdahl's Law : Example 1

We want a speed up of 90x faster with 100 processors

$$90 = \frac{1}{(1 - \text{Fraction time affected}) + \frac{\text{Fraction time affected}}{100}}$$

$$\rightarrow \text{Fraction time affected} = \frac{89}{89.1} = 0.999$$

So to get a speed-up of 90 from 100 processors the sequential part can only be 0.1%

Amdahl's Law : Example 2 (Bigger Problem)

- Suppose you want to perform simultaneously **two sums**:
  - sum of 10 scalar variables
  - matrix sum of pairs of 2D arrays with dimension 10x10

➡ **What speed up you get with 10 vs 100 processors?**

➡ **Calculate the speed-ups assuming matrices grow to 100x100**

Amdahl's Law : Example 2 (Bigger Problem)

Answer

Assume time **t** for the performance of an addition.
Then there are (for 100 processors) 100 additions that scale and 10 that do not.

$$\text{Time for 1 processor} = 100t + 10t = 110t$$

Amdahl's Law : Example 2 (Bigger Problem)

$$\text{Time for 10 processors} =$$

$$\text{Time after improvement} = \frac{\text{Exec time affected}}{\text{Amount of improvement}} + \text{Exec time unaffected}$$

$$= \frac{100t}{10} + 10t = 20t$$

$$\boxed{\text{Speed-up}_{10} = \frac{110t}{20t} = 5.5 \text{ (out of 10)}}$$

$$\text{Time for 100 processors} = \frac{100t}{100} + 10t = 11t$$

$$\boxed{\text{Speed-up}_{100} = \frac{110t}{11t} = 10 \text{ (out of 100)}}$$

For 110 numbers we get 55% of the potential speedup with 10 processors but only 10% of 100 processors.

Amdahl's Law : Example 2 (Bigger Problem)

What happens when we increase the matrix order?

$$\text{Time for 1 processor} = 10t + 10000t = 10010t$$

10 processors

$$\text{Exec time after improvement} = \frac{10000t}{10} + 10t = 1010t$$

$$\text{Speed-up}_{10} = \frac{10010t}{1010t} = 9.9 \text{ (out of 10)}$$

100 processors

$$\text{Exec time after improvement} = \frac{10000t}{100} + 10t = 110t$$

$$\text{Speed-up}_{100} = \frac{10010t}{110t} = 91 \text{ (out of 100)}$$

For larger problem size we get 99% with 10 processors and 91% with 100 processors

Amdahl's Law : Example 3 - Load Balancing

- In the previous example, in order to achieve the speed-up of 91 **(for the larger problem)** with 100 processors, we assumed that the load was perfectly balanced.

Perfect balance: each of the 100 processors has 1% of the work to do

What if: 1 of the100 processors load is higher than all the other 99 ?
Calculate for increased loads of 2% and 5%

Amdahl's Law : Example 3 - Load Balancing

- If one has 2% of the parallel load then it must do
  2% x 10000 (larger problem) = 200 additions
- The other 99 will share 9800

Since they operate simultaneously:

$$\text{Exec time after improvement} = \max\left(\frac{9800t}{99}, \frac{200t}{1}\right) + 10t = 210t$$

The speed-up drops to: $\text{Speed-up}_{100} = \frac{10010t}{210t} = 48$

$$\text{Speed-up}_{100}^{10000} \,|^{\text{NB}} = 48\% \text{ (instead of 91\%)}$$

- If one processor has 5% of the load, then it must perform
  5% x 10000 = 500 additions

Fallacies and Pitfalls - I

PITFALL: Expecting the improvement of one aspect of a computer to increase *overall* performance by an amount proportional to the size of an improvement

Example: A program runs in 100s on a computer, with multiply operators responsible for 80s of this time. How much do I have to improve the speed of multiplication if I want to run 5 times faster?

$$\text{Exec. time after the improvement} = \frac{\text{Exec. time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected} =$$

$$= \frac{80\text{s}}{n} + (100 - 80) = \frac{80}{n} + 20$$

Asking for the left side to be 20 requires $n = \infty$

Fallacies and Pitfalls - II

FALLACY: Computers at low utilization use little power

Servers at 10% of load can utilize up to 2/3 of peak power

Towards Hardware for energy aware computing

Fallacies and pitfalls -III

PITFALL: Using a subset of performance equation as a performance metric

Example: use as an alternative to the Time metric for performance the MIPS (Million Instructions Per Second) - **is it enough ?**

$$\text{MIPS} = \frac{\#\text{Instruction}}{\text{Execution time}} \times 10^{-6} = \frac{\#\text{Instructions}}{\frac{\#\text{Instructions} \times \text{CPI}}{\text{Clockrate}}} \times 10^{-6} = \frac{\text{Clock rate}}{\text{CPI}} \times 10^{-6}$$

Problem: instruction count may vary, MIPS varies between programs

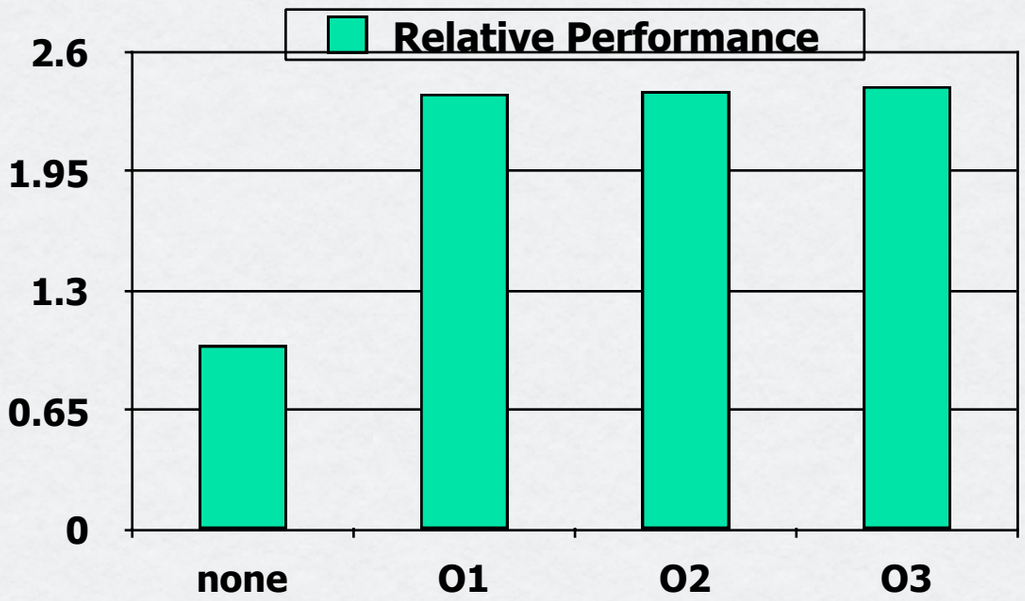
$\Longrightarrow$ no single MIPS rating for computers - hardware dependency

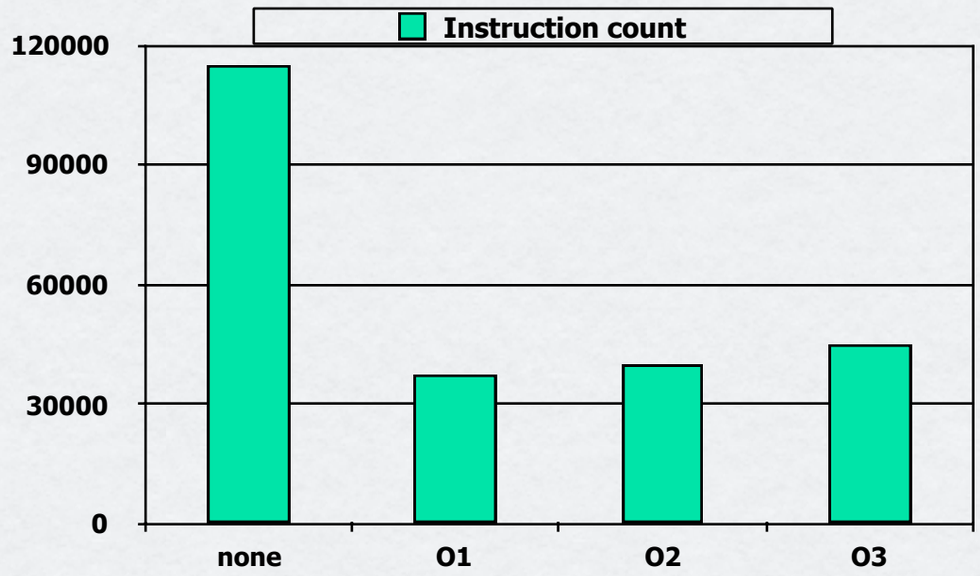
Execution time: the **only** valid/unimpeachable metric of performance

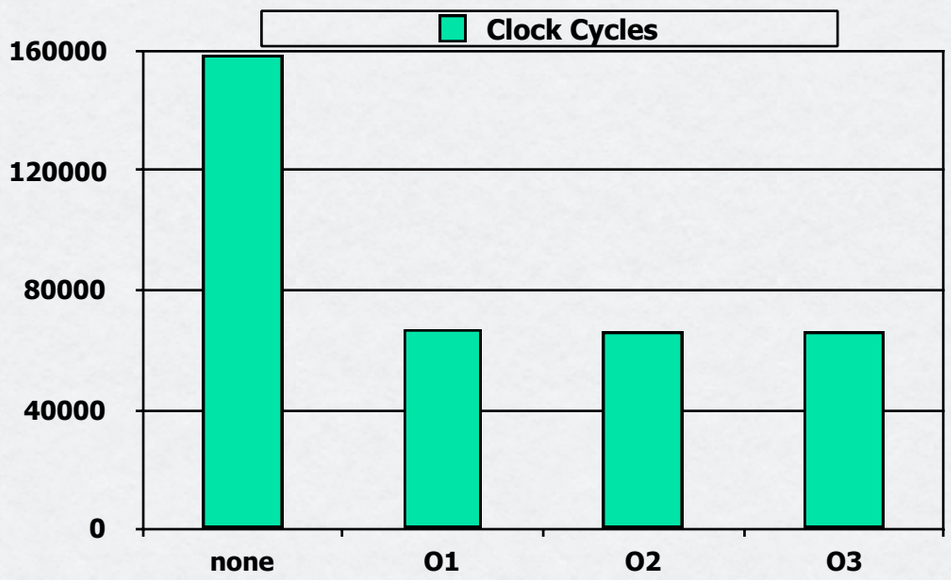
$$\text{Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instructions}} \times \frac{\text{Seconds}}{\text{Clock cycles}}$$

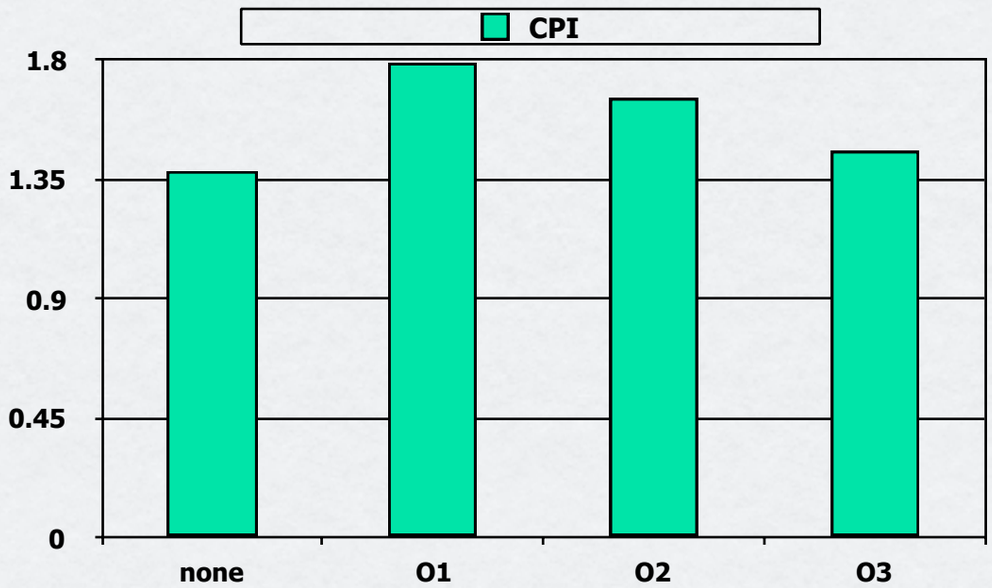
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

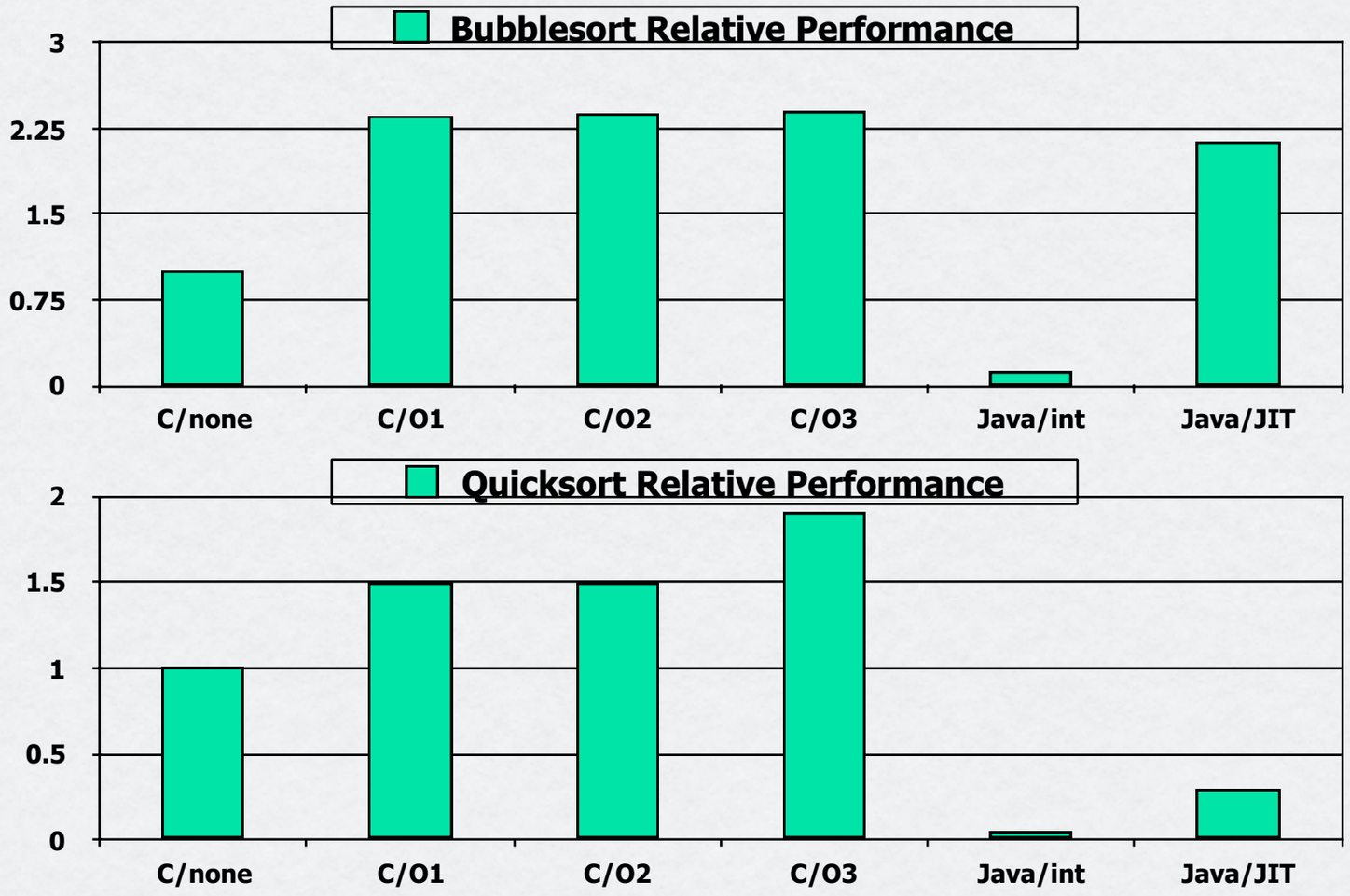


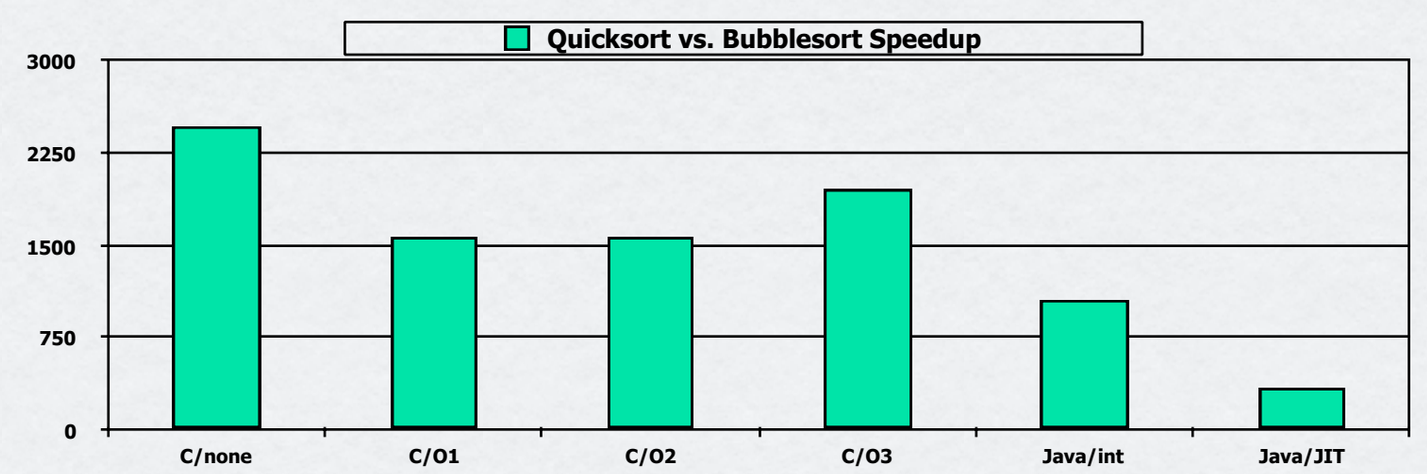






Effect of Language and Algorithm



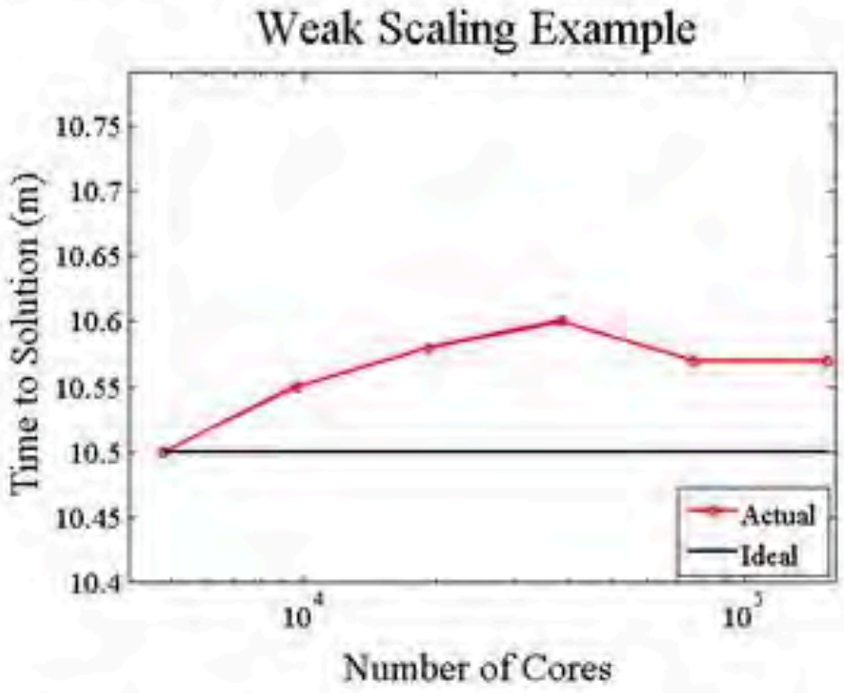


PERFORMANCE: SCALING

- **STRONG SCALING**:
  speed-up on multiprocessors **without** increase on problem size (**Amdahl's law** considers the strong scaling)

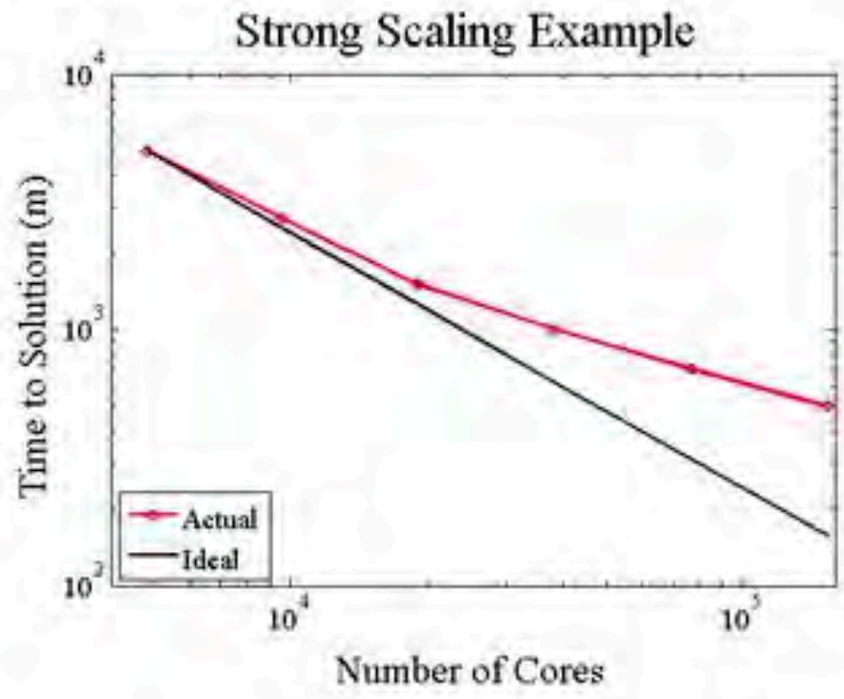
- **WEAK SCALING**:
  speed-up on multiprocessors, while increasing the size of the problem *proportionally* to the increase in the number of processors

Weak Scaling Example



nProc	Time to Solution (m)	Ideal Time to Solution (m)
4800	10.50	10.50
9600	10.55	10.50
19200	10.58	10.50
38400	10.60	10.50
76800	10.57	10.50
153600	10.57	10.50

Strong Scaling Example



nProc	Time to Solution (m)	Ideal Time to Solution (m)
4800	5000.00	5000.00
9600	2725.00	2500.00
19200	1500.00	1250.00
38400	1000.00	625.00
76800	700.00	312.50
153600	500.00	156.25

SCALABILITY

Certain problems demonstrate **increased performance by increasing the problem size.**

For example:

- **2D Grid Calculations 85 seconds 85%**
- **Serial fraction 15 seconds 15%**

We can increase the problem size by doubling the grid dimensions and halving the time step. This results in four times the number of grid points and twice the number of time steps. The timings then look like:

- **2D Grid Calculations 680 seconds 97.84%**
- **Serial fraction 15 seconds 2.16%**

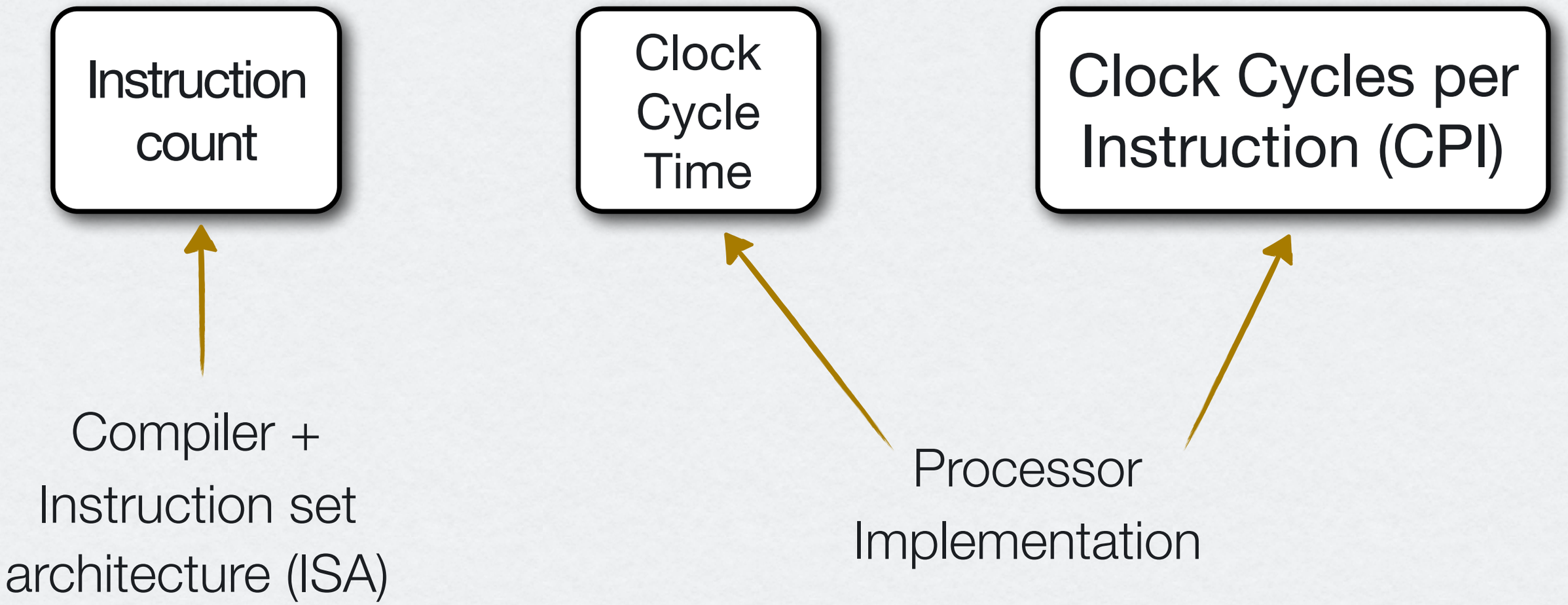
Problems that increase the percentage of parallel time with their size are more ***scalable*** than problems with a fixed percentage of parallel time.

Scalability:

- The ability of a parallel program's performance to scale is a result of a number of interrelated factors. Simply adding more machines is rarely the answer.
- The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease. Most parallel solutions demonstrate this characteristic at some point.
- Hardware factors play a significant role in scalability. Examples:
  - Memory-cpu bus bandwidth on an SMP machine
  - Communications network bandwidth
  - Amount of memory available on any given machine or set of machines
  - Processor clock speed

- Parallel support libraries and subsystems software can limit scalability independent of your application.

Performance

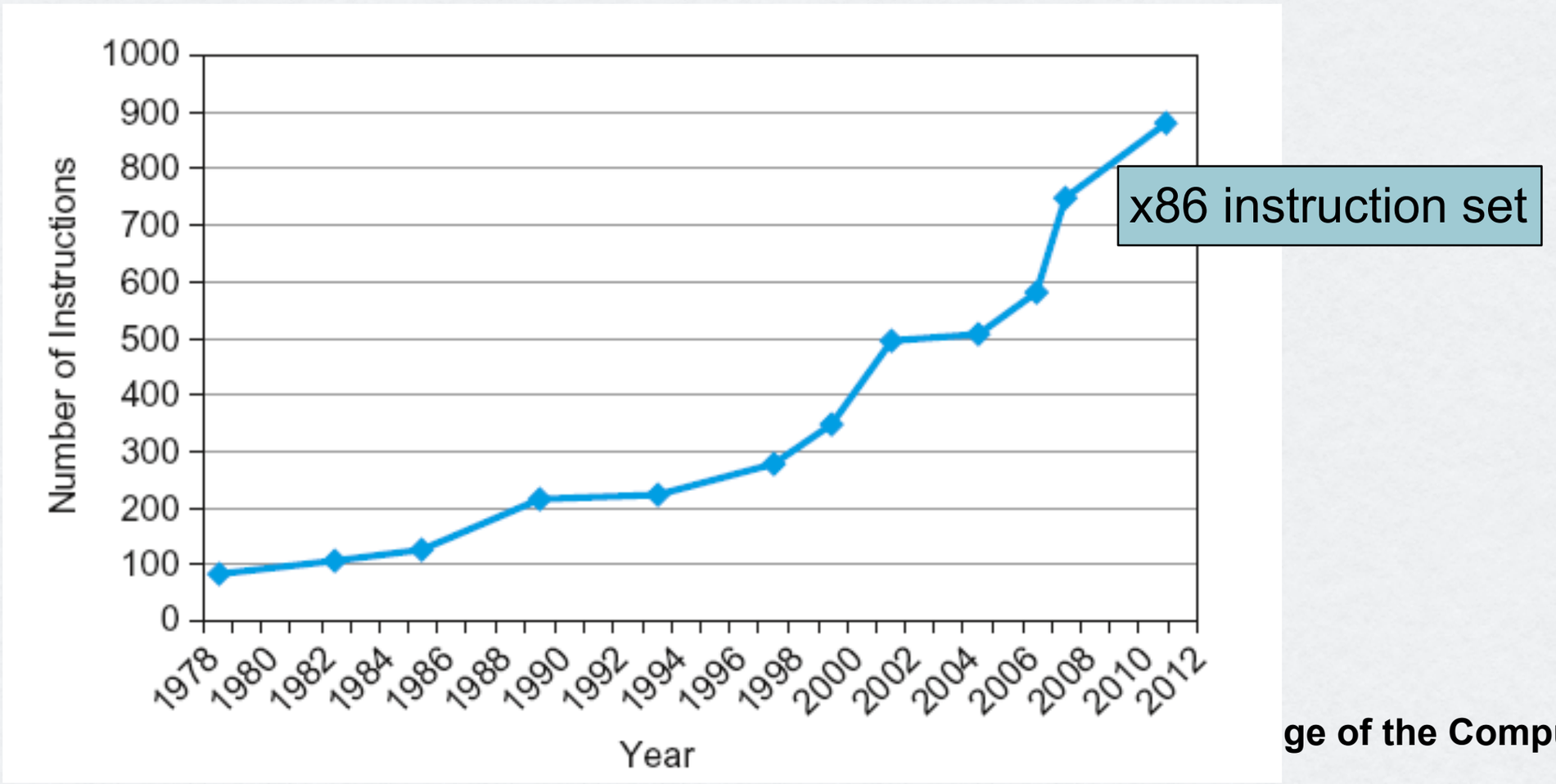


Fallacies

- **Powerful instruction ⇒ higher performance**
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
    - Compilers are good at making fast code from simple instructions
- **Use assembly code for high performance**
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Fallacies

- **Backward compatibility ⇒ instruction set doesn't change**
  - But they do accrete more instructions



Lessons Learnt

- **Instruction count and CPI are not good performance indicators in isolation**
- **Compiler optimizations are sensitive to the algorithm**
- Java/JIT compiled code is significantly faster than JVM interpreted - Comparable to optimized C in some cases
- **Nothing can fix a dumb algorithm!**

Parallelism and Instructions: Synchronization

- To know when a task has finished writing so that it is safe for another to read, the tasks need to synchronize

- **No synchronization:** danger of a data race: the results of the program depend on how program was run
  - Consider the example of 8 reporters writing a common story

- In computers, lock and unlock are synchronization operations.
  Lock and unlock create mutual exclusions

Parallelism and Instructions: Synchronization

- A typical operation for building synchronization operations is the atomic exchange or atomic swap. This call **interchanges a value in a register for a value in memory**

- Example - Simple lock:

0 indicates that lock is free, 1 that it is unavailable

  - A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock.
  - Return is:
    - 1 if some other processor had claimed access
    - 0 otherwise ⇒ value changed to 1

      (prevents anybody else from getting 0)

Memory

Memory

Library Paradigm: think of writing an important review on computer hardware.

You go to the library and collect a few books that you put on your desk to consult.

This is faster than taking one book at a time and returning to bookcases every time you need something.

Locality

Same principle allows us to create an illusion of a large memory that we can access as fast as very small memory

➡ HAVE NEAR YOU ONLY WHAT YOU NEED

Principle of LOCALITY

- Temporal locality:

  if an item is referenced, it will need to be referenced again soon

  if you recently brought a book to your desk chances are you need it again

- Spatial locality:

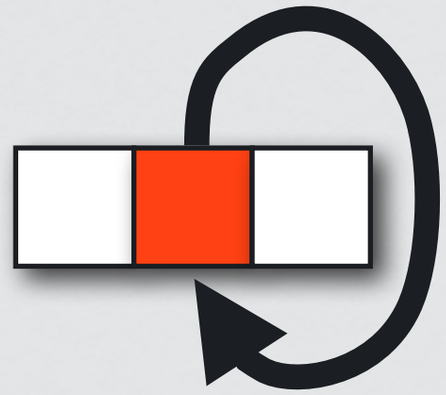
  if an item is referenced, items whose addresses are close by, will tend to be referenced soon

  libraries put books on similar topics nearby to increase spatial locality

Locality examples

- **DATA ACCESS**
  - ➡ sequential access to elements of an array
  - ➡ SPATIAL LOCALITY

- **LOOPS:**
  - ➡ instructions and data accessed repeatedly
  - ➡ TEMPORAL LOCALITY

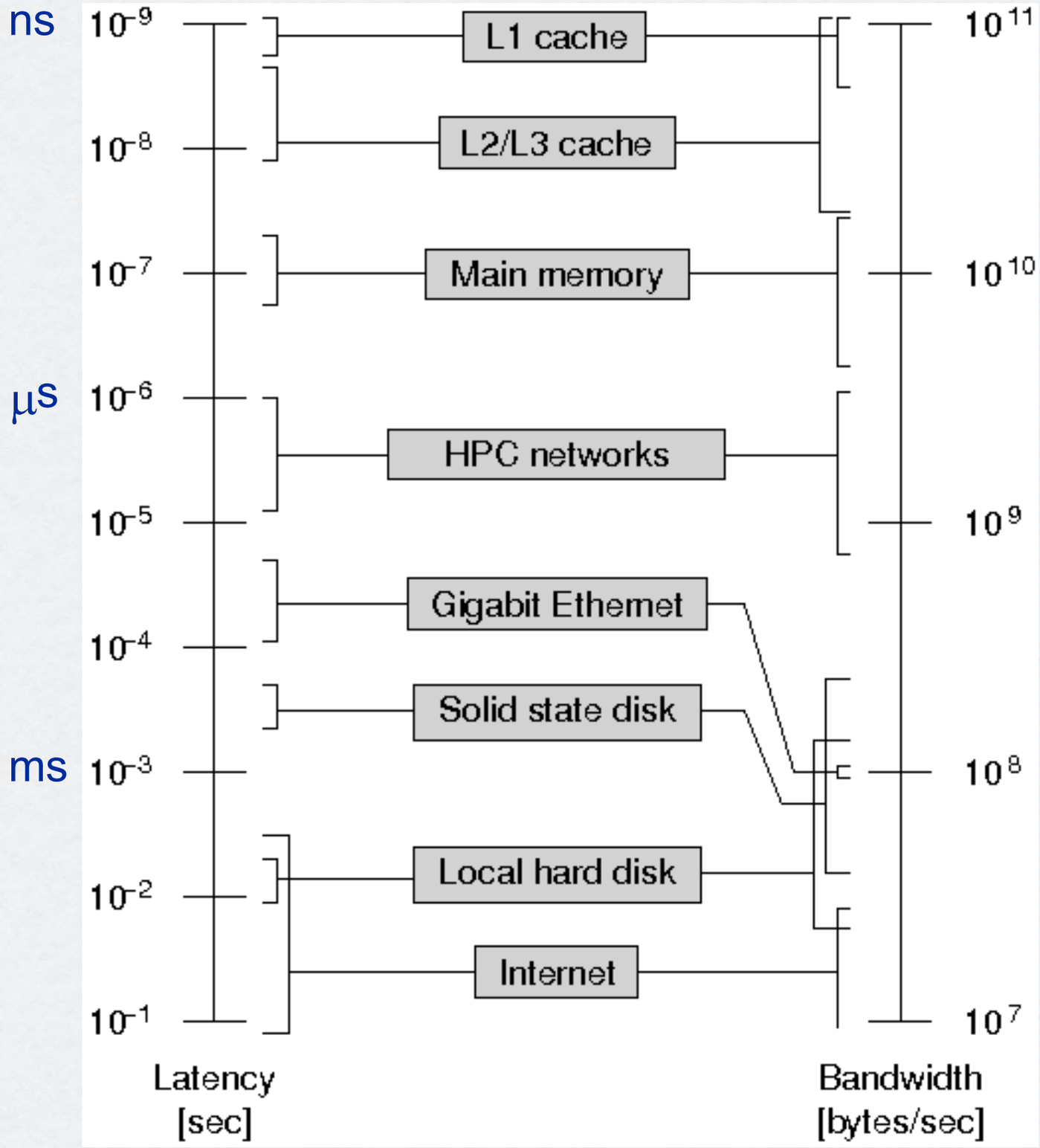


Memory Hierarchy

Multiple levels of memory with different speeds and sizes.

Memory	Access time (ns)	$ per GB (2008)	Library Example
SRAM	0.5 - 2.5	2000 - 5000	*Books on desk*
DRAM	50 - 70	20 - 75	*Books in library*
Magnetic Disk	5×10^6 - 20×10^6	0.2 - 2	*Libraries on campus*

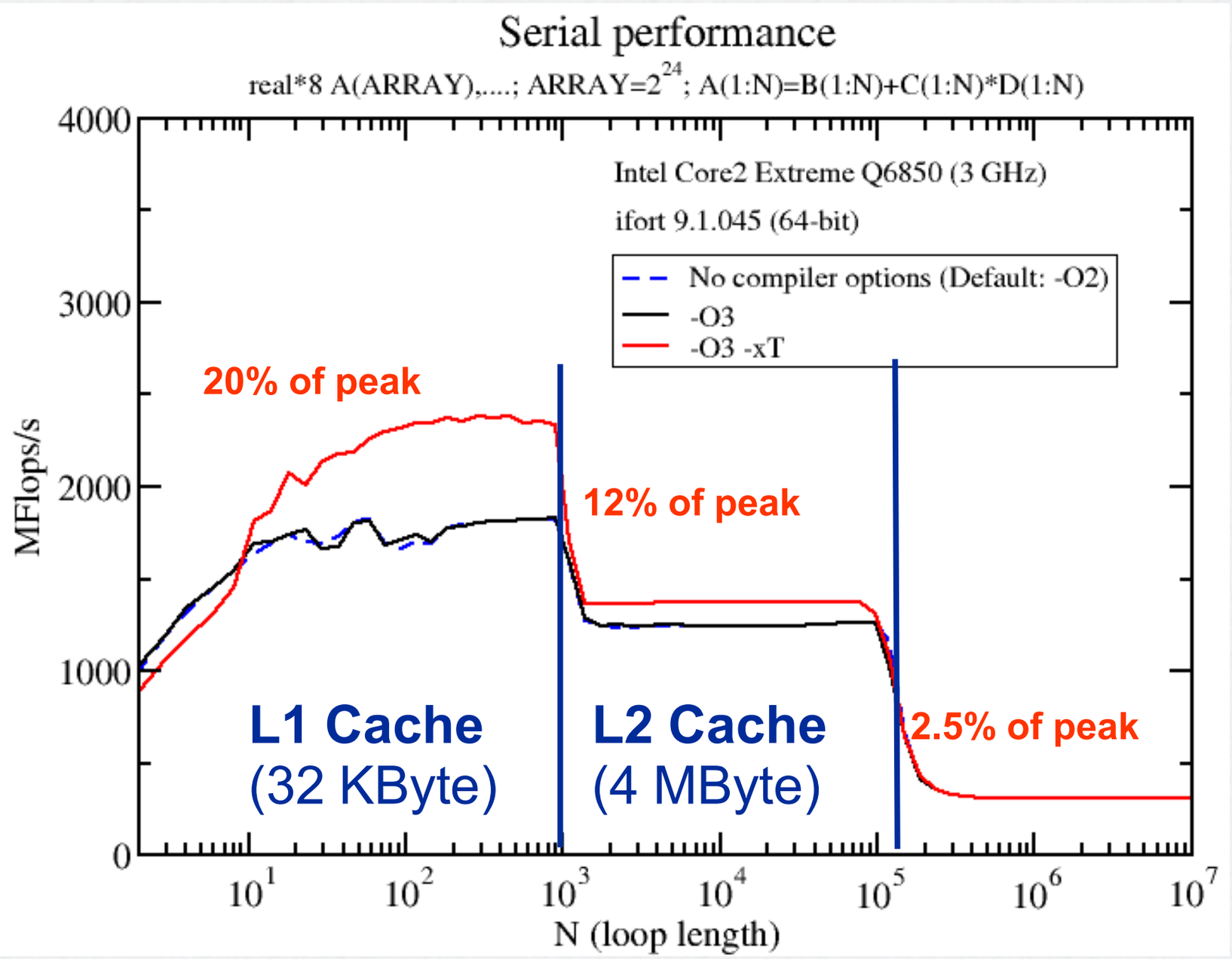
Latency and bandwidth in modern computer environments



Characterization of Memory Hierarchies: Intel Core2 Q6850

Peak Performance for 1 core of Intel Core2 Q6850 (DP):

3 GHz * (2 Flops (DP-Add) + 2 Flops (DP-Mult)) = **12 GFlops/s**



Performance decreases if data set exceeds cache size

`-xT` : Enables vectorization & improves in-cache performance: Packed SSE instructions

Cache: Terminology

- **Block/Line**: minimum unit of information that can be present or not present in a cache *(a book on the desk)*

- **Hit**: data request by the processor, encountered in some block in the *upper (closer to processor)* level of memory hierarchy. If the data is not found, it is a **miss**. Then a *lower (further from processor)* level is accessed to retrieve the data *(you go from your desk to the shelves to find a book)*.

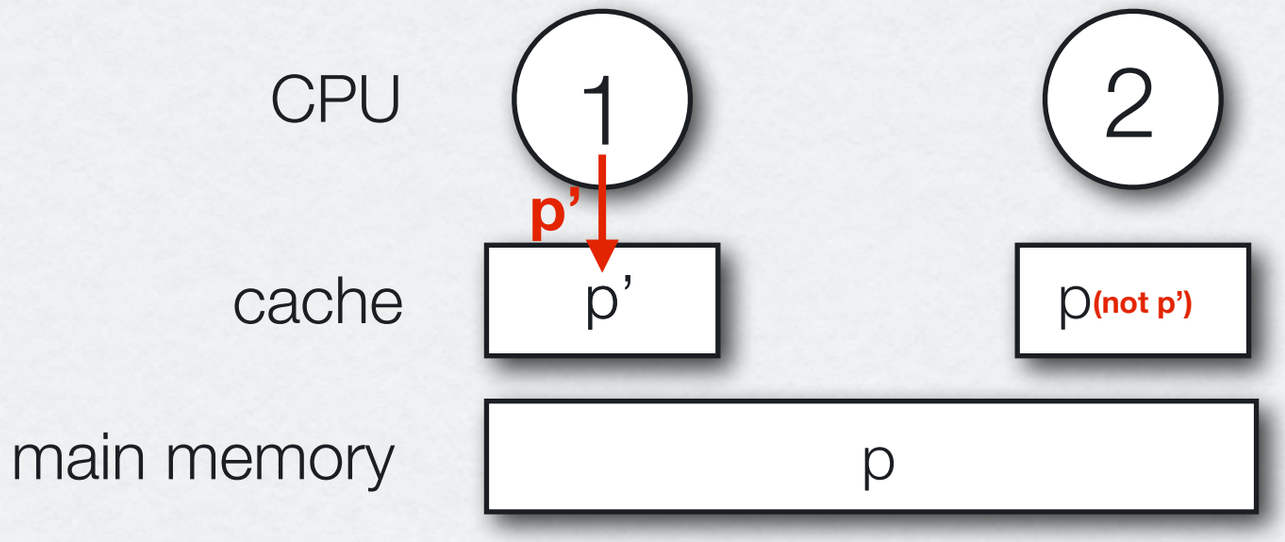
- **Hit rate**: fraction of memory accesses found in the upper level.

- **Hit time**: time to access upper level including time to determine if it is a hit or a miss.

- **Miss penalty**: time to replace a block in the upper level with the corresponding block from the lower level.

Parallelism and Memory Hierarchies

- **Multicore multiprocessor** = multiple processors on a single chip
- Processors (most likely) share a common physical address space
- ***Caching shared data:*** view of memory for each processor through their individual caches so it differs if changes are made.
- CAREFUL: 2 different processors can have 2 different values for the same location **->** cache coherence problem



Cache Coherency

A memory system is coherent if:

1. A read by processor P to location X, that follows a write by P to X, with no writes to X by another processor occurring between the write and read by P, always returns the value written by P.

2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occurs between the 2 accesses. ⇒ need controller

3. Writes to the same location are serialized: that is 2 writes to the same location by any 2 processors are seen in the same order by all processors.

Enforcing Coherence

- **Memory latency** refers to delays in transmitting data between the CPU and Memory. Latency is often measured in memory bus clock cycles. However, the CPU operates faster than the memory, so it must wait while the proper segment of memory is located and read, before the data can be sent back. This also adds to the total Memory latency. (*source: Wikipedia*) ***(it's the time required to go to the library and back.)***

- **Migration**: a data item can be moved to a *local cache* and used there in a transparent fashion. Reduces latency access of remote data and bandwidth of shared memory. ***(bring a book to the desk)***

- **Replication**: when shared data are being read simultaneously, the caches make a copy of the data item in the local cache. Reduces access latency and contention for a read in shared data items.

Enforcing Coherence

Protocols are maintained for cache coherence by **tracking the state of any sharing** of a data block.

- Example -> **Snooping protocols**: every cache with a copy of the data from a block of physical memory, also has a copy of the sharing status of the block, but no centralized state is kept.

- The caches are all accessible via some broadcast medium (bus or network) and all cache controllers monitor (snoop) on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access.

Memory Usage: Remarks

- The difficulty of building a memory system to keep pace with faster processors is underscored by the fact that the raw material for main memory, DRAM, is essentially the **same** for fastest as well as for cheapest and slowest computers.

- Principle of Locality: gives chance to overcome long latency of memory access.

Memory Usage: Remarks

- **Multilevel caches** make it possible to use more cache optimization, more easily:
  - design parameters of a lower-level cache are different from a first level cache **(as lower level cache is larger, then we use larger block sizes).**
  - lower level cache is not constantly used by processor, as a first level cache **(allow lower level cache to prevent future misses)**.

Memory Usage: Remarks

- **Software** for improved memory usage. Compilers to transform programs.
  - recognize program to **enhance its spatial and temporal locality** (loop-oriented programs, using large arrays as the major data structure; e.g. large linear algebra problems) by restructuring the loops (to improve locality) large cache performance

- **Prefetching**: a block of data is brought to cache before it is referenced. Hardware to predict accesses that may not be detected by software.

- Cache-aware instructions to optimize memory transfer.

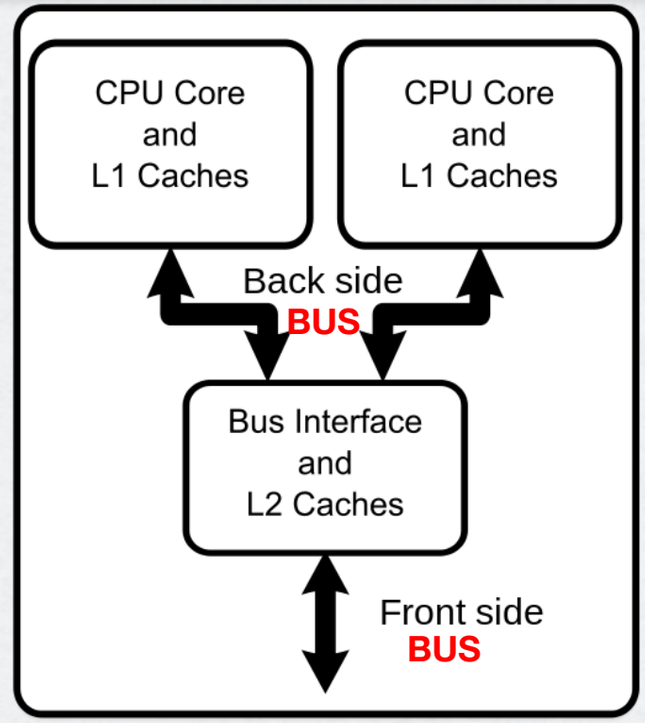
CONNECTING: Processors, Memory and I/O Devices

- **BUS:**
  - A shared communication link, which uses one set of wires to connect multiple subsystems.
  - Used for communication between memory, I/O and processors.

- **versatility**: since we have a single connection scheme, new devices can be added
- **simplicity**: single set of wires is shared in multiple ways
- **communication bottleneck**: limiting the I/O throughput as all information passes a single wire

CONNECTING: Processors, Memory and I/O Devices

- **Processor-Memory Bus**: bus that connects processor and memory, and that is short, high speed and matched to the memory system so as to *maximize memory-processor bandwidth*.



- **I/O Buses**: longer wires and many devices attached and have a wide range of data bandwidth. They do not interface the memory but they go through processor or the backplane bus.

- **Backplane Bus**: a bus designed so that processor, memory and I/O devices can coexist on a single bus.

BUS LIMITS: the length of the bus and the number of devices

Multicore, Multiprocessor & Clusters

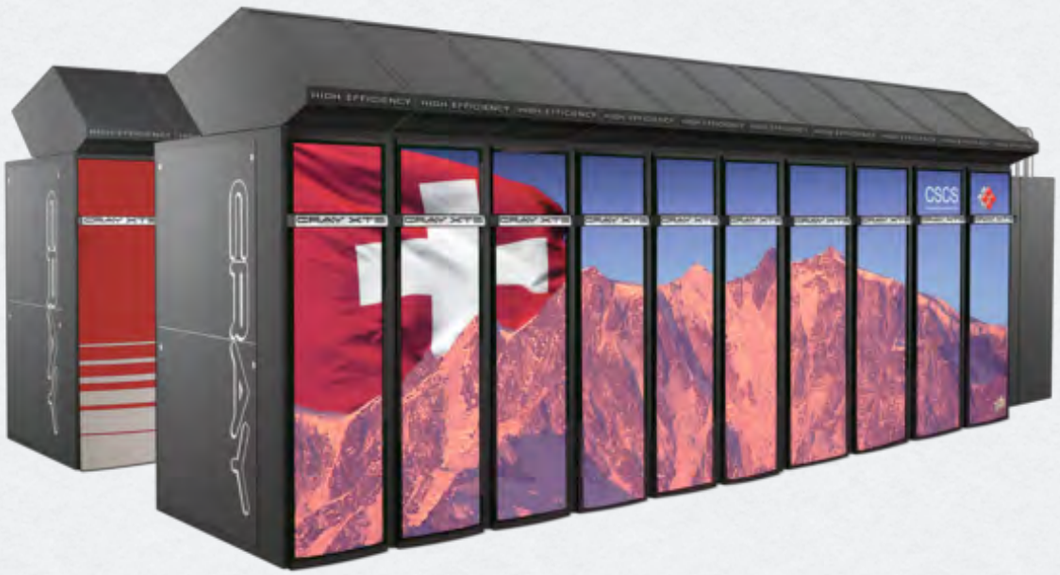
The El-Dorado of Computer Design:

Creating powerful computers simply by connecting many existing smaller ones.

- **Why Smaller computers:**
  - better performance/watt
  - scalability

Cluster

- A set of computers connected over a Local Area Network (LAN) that functions as a single large multiprocessor.
- Performance comes from more processors per chip rather than higher clock



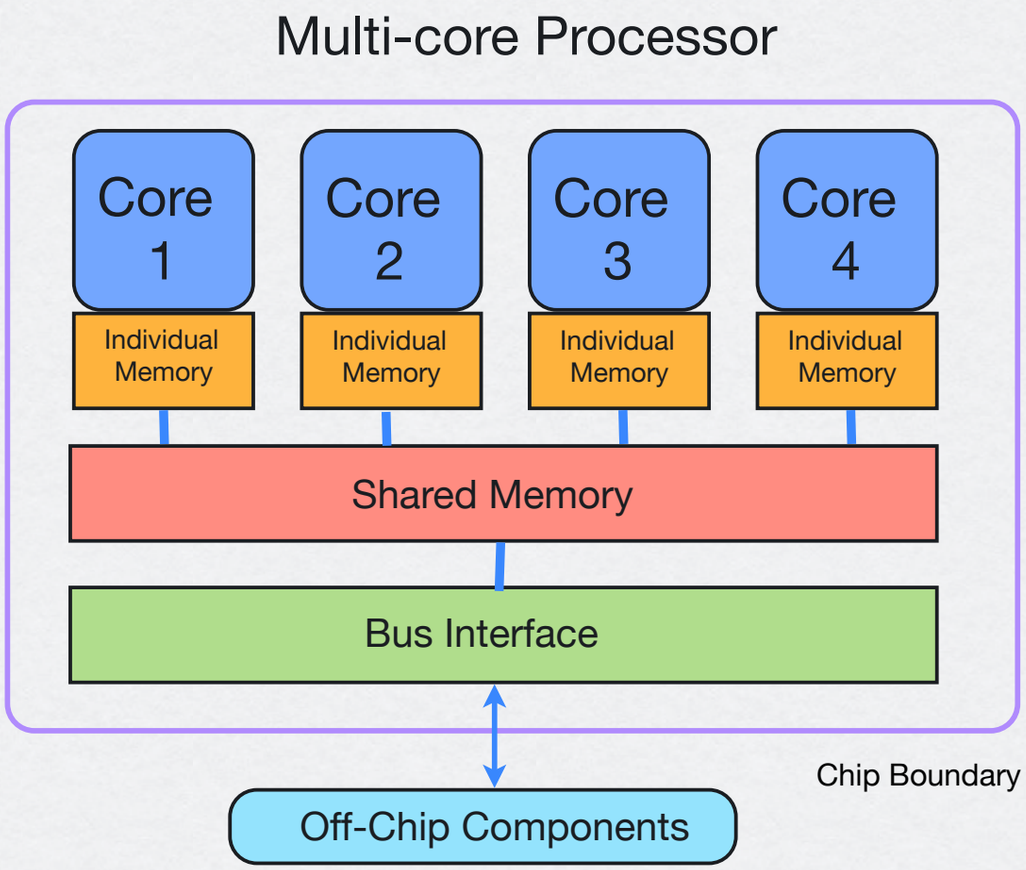
Cray XE6 – Monte Rosa (CSCS)

CERN Server

Multicores, Multiprocessors

- multiple processors per chip
- processors are referred to as cores

⟶ number of cores per chip is expected to double per year



An Intel Core 2 Duo E6750 dual-core processor.

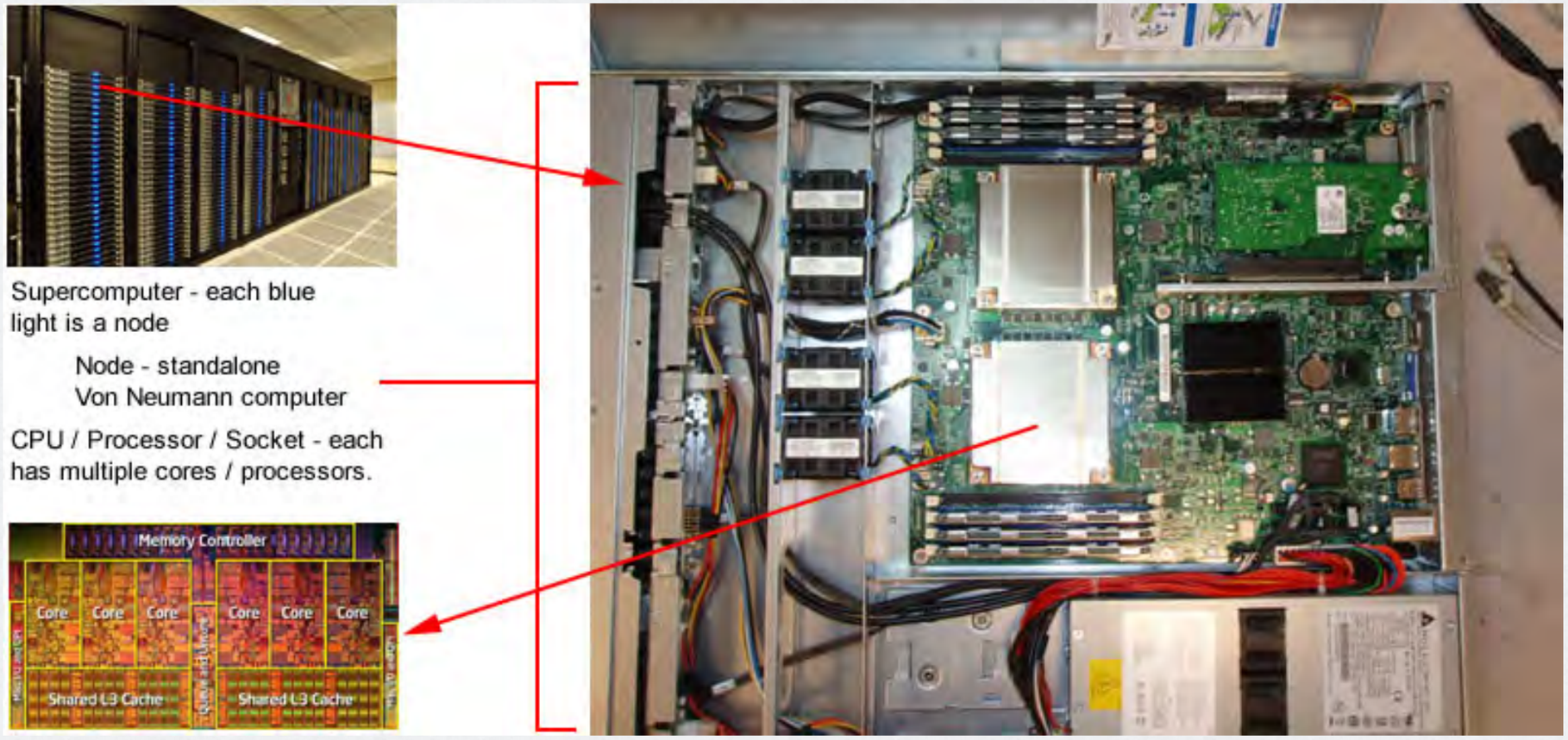
Multicore, Multiprocessor & Clusters

- Job/Process level parallelism:
  - high throughput for *independent* jobs
  - one of the most popular levels of parallelism

- Parallel processing program:
  - a program that runs on multiple processors simultaneously

Terminology

Node
A standalone "computer in a box". Usually comprised of multiple CPUs/processors/cores. Nodes are networked together to comprise a supercomputer.



CPU / Socket / Processor / Core
This varies, depending upon who you talk to. In the past, a **CPU** (Central Processing Unit) was a singular execution component for a computer. Then, multiple CPUs were incorporated into a **node**. Then, individual CPUs were subdivided into multiple **"cores"**, each being a unique execution unit. ***CPUs with multiple cores are sometimes called "sockets"*** - vendor dependent. The result is a node with multiple CPUs, each containing multiple cores. The nomenclature is confused at times.

Task
A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor. ***A parallel program consists of multiple tasks running on multiple processors.***

Pipelining
Breaking a task into steps performed by different processor units, with *inputs streaming through*, much like an assembly line; a type of parallel computing.

Terminology II

Shared Memory

From a strictly **hardware point of view**, describes a computer architecture where all processors have direct (usually bus based) access to ***common physical memory***. In a **programming sense**, it describes a model where parallel tasks all have the ***same "picture" of memory*** and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

Symmetric Multi-Processor (SMP)

Hardware architecture : multiple processors share a single address space and access to all resources; shared memory computing.

Distributed Memory

In hardware, refers to *network based memory access* for physical memory that is not common. As a programming model, ***tasks can only logically "see" local machine memory*** and must use **communications** to access memory on other machines where other tasks are executing.

Communications

Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed.

Synchronization

The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point.
Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.

Terminology III

Granularity

In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

- ***Coarse:*** relatively large amounts of computational work are done between communication events
- ***Fine:*** relatively small amounts of computational work are done between communication events

Observed Speedup

Observed speedup of a code which has been parallelized, defined as:

$$\frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}$$

One of the simplest and most widely used indicators for a parallel program's performance.

Parallel Overhead

The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead includes:

- Task start-up time
- Synchronizations
- Data communications
- Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
- Task termination time

Terminology IV

Massively Parallel
Refers to the hardware that comprises a given parallel system - having many processors. The meaning of "many" keeps increasing, but currently, the largest parallel computers can be comprised of processors numbering in the hundreds of thousands (example: GPUs)

Embarrassingly Parallel
Solving many similar, but independent tasks simultaneously; *little to no need for coordination between the tasks.*

Scalability
Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include:

- Hardware - particularly memory-cpu bandwidths and network communications
- Application algorithm
- Parallel overhead related
- Characteristics of your specific application and coding

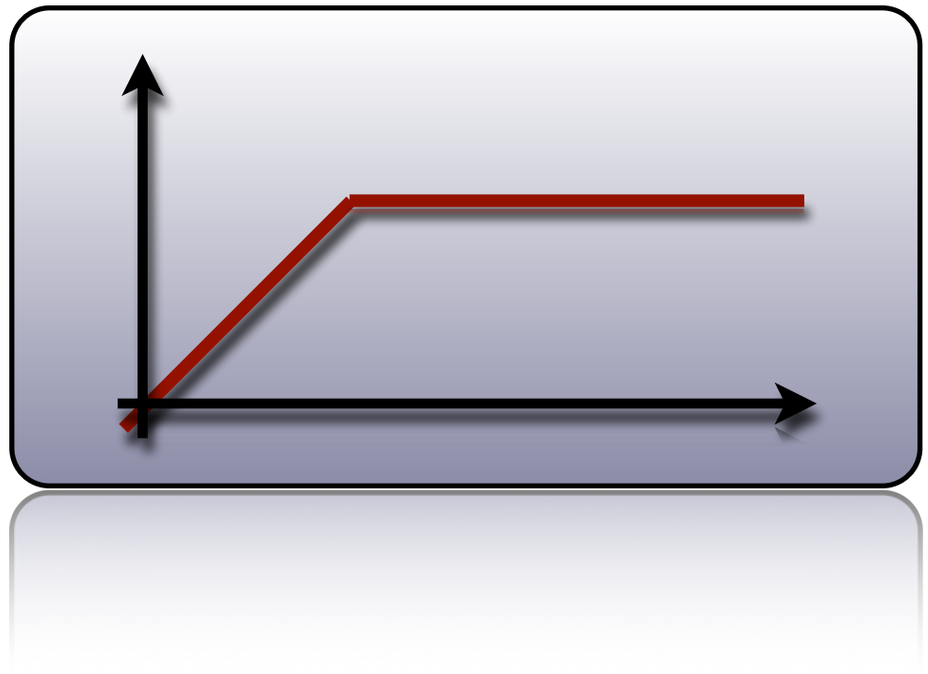
Multicore, Multiprocessor & Clusters

- Job/Process level parallelism:
  - high throughput for *independent* jobs
  - one of the most popular levels of parallelism

- Parallel processing program:
  - a program that runs on multiple processors simultaneously

Performance II
The Roofline model

Outline



- Motivation
- 3C model
- The roofline model
- Roofline-based software optimization
- Study case: grid-based simulations on CPU/GPU
- Conclusions

Information taken from:
[1] Roofline: an insightful visual performance model for multicore architectures,
Williams, S. and Waterman, A. and Patterson, D., Communication to ACM, 2009
[2] Intel 64 and IA-32 Architectures Optimization Reference Manual:
http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html
[3] Software Optimization Guide for AMD Family 15h Processors:
http://support.amd.com/us/Processor_TechDocs/47414.pdf

Motivation (1)

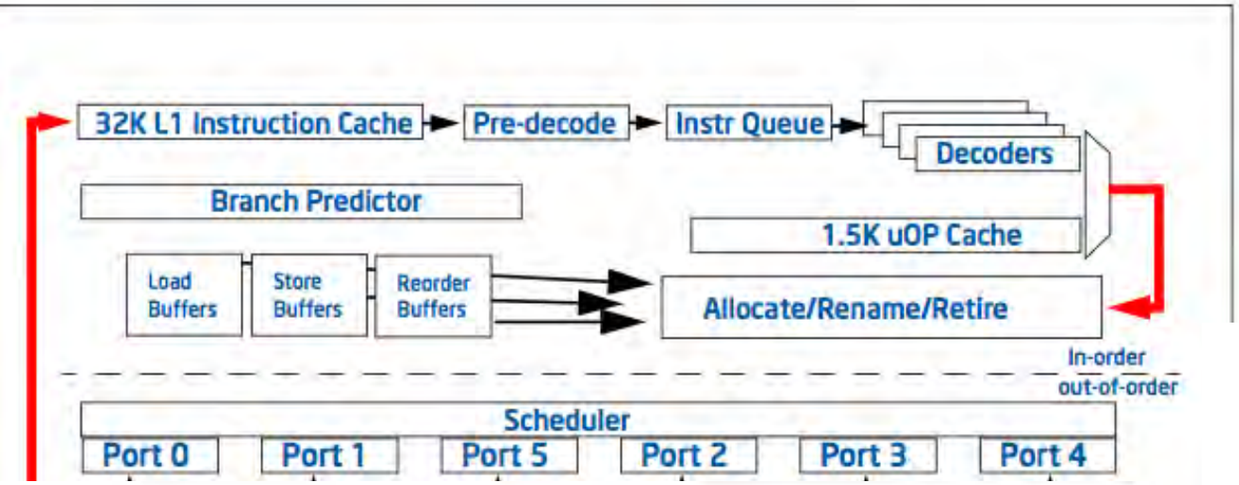
Industry switch to multi/many-core platforms:

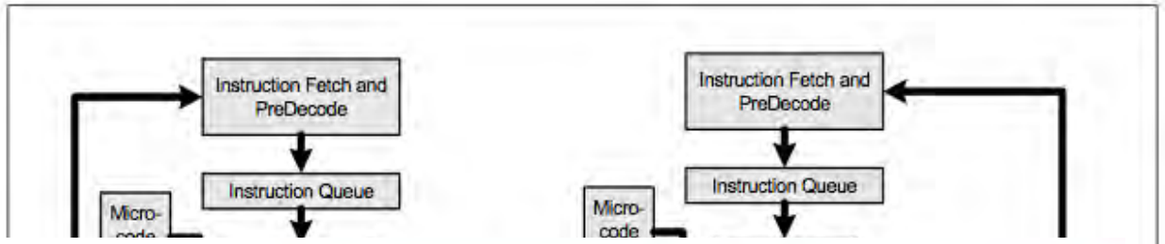
- No conventional wisdom has yet emerged

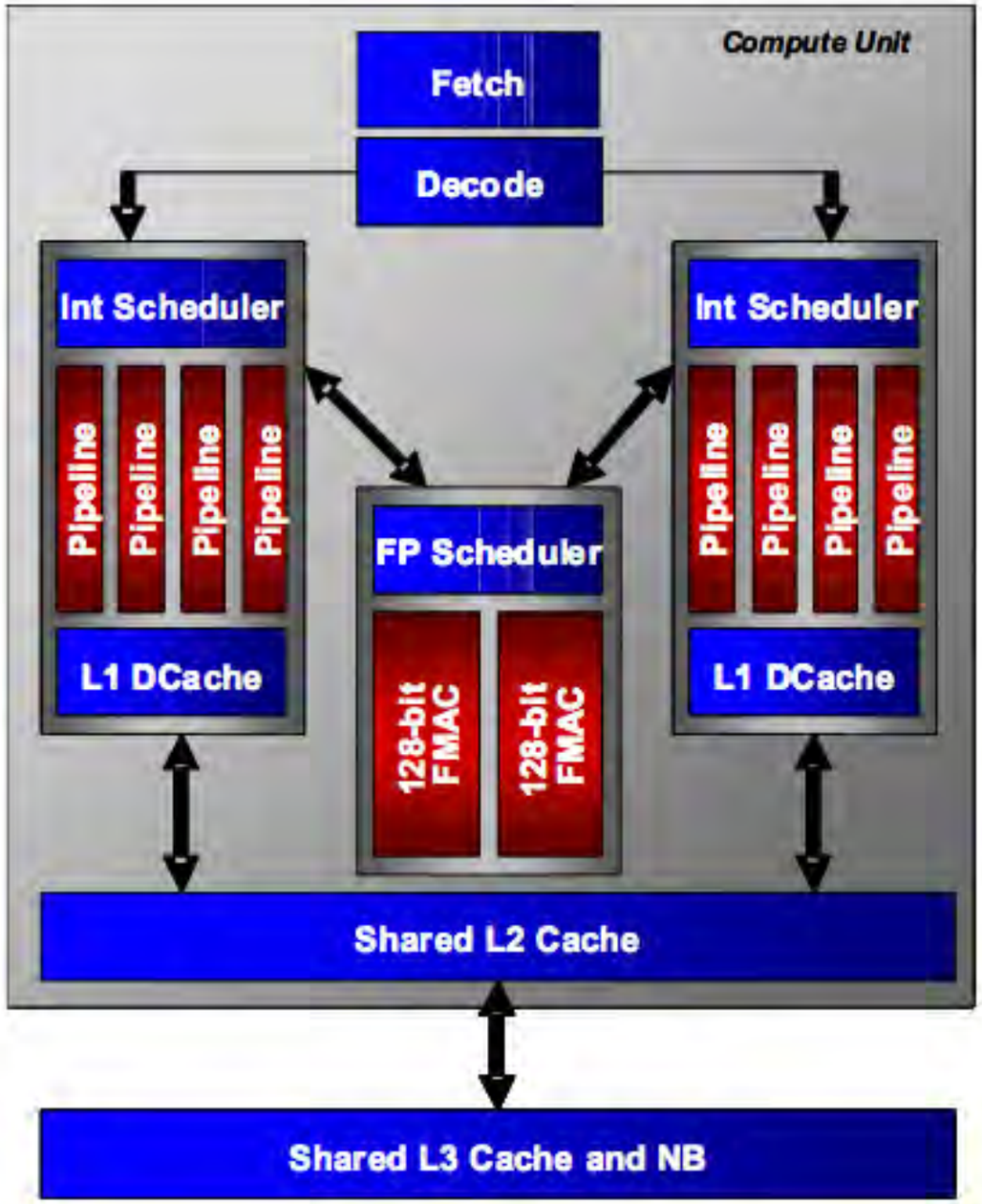
x Micro processors are diverse

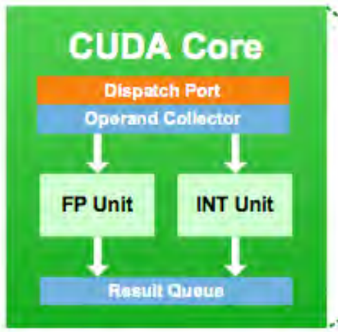
- Differences in:

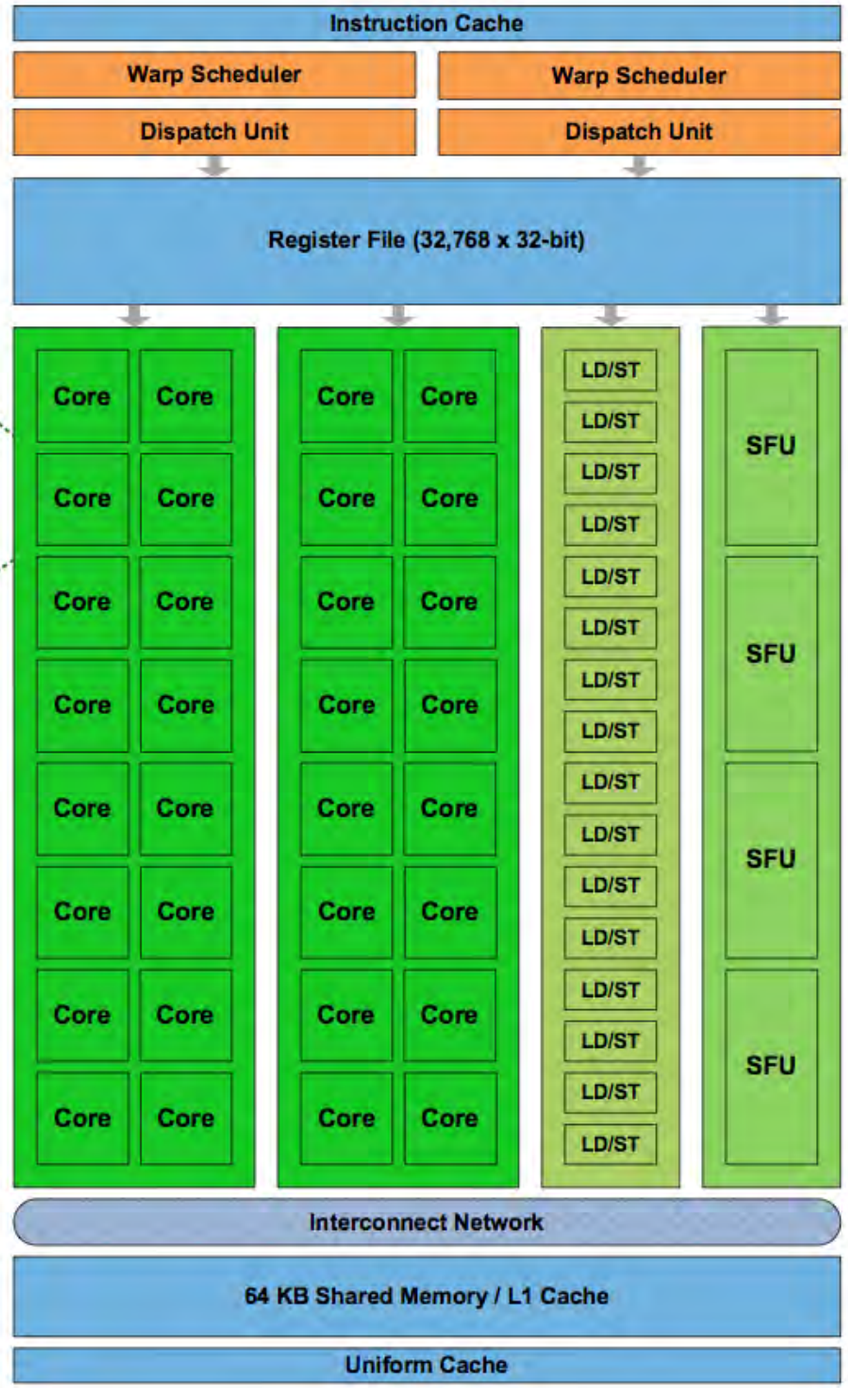
  1. Number of processing elements
  2. Number pipeline stages/execution ports
  3. Instruction issues/scheduling
  4. Memory/cache hierarchies











AMD Family 15h Processor Block Diagram

Motivation (2)

Are these platforms converging into a single one?

- Difficult to say, but it is unlikely (manufacturers perspective):
  - ➡ Moore's law (number of cores will increase)
  - ➡ Vendors want to cover many price-performance points
- Hard times for software developers:
  - ➡ Are there general performance guidelines?
  - ➡ Are there insightful performance models?

Performance model: 3C

"3C" stands for cache misses:

- Compulsory
- Capacity
- Conflict

Used for software optimization [1]:

- Popular for nearly 20 years
- Insightful model
- Not perfect (ambiguity in the cache misses types)

It focuses on:

- Take full advantage of **cache hierarchies**
- Maximize **data reuse**

[1] Hill, M. And Smith, A. Evaluating Associativity In CPU Caches. IEEE Transactions On Computers 38, 1989

The roofline model

Recently proposed by Williams, Waterman and Patterson [1]:

- Crucial in performance predictions
- Helpful for software optimization

"Bound-and-bottleneck" analysis:

- Provides valuable *performance insight*
- Focuses on the *primary performance factors*
- Main *system bottleneck* is highlited and quantified

[1] Roofline: an insightful visual performance model for multicore architectures, Williams and Waterman and Patterson, Communication to ACM, 2009

Transfer-Computation *overlap*

On CPU:

- Superscalar execution (multiple instructions per cycle)
- *In principle*: automatic overlap (balanced instructions)

✓ *In practice*: enforced through software prefetching

On GPU:

- Superscalar execution (VLIW for AMD GPUs)

✓ Achieved by high occupancies (automatic ILP increase)

✓ Achieved by manually improving the ILP [1]
(by avoiding RAW/WAR conflicts, no renaming on GPU)

[1] V.Volkov. Better performance at lower occupancy, GTC 2010.

The roofline model

Main assumptions/issues:

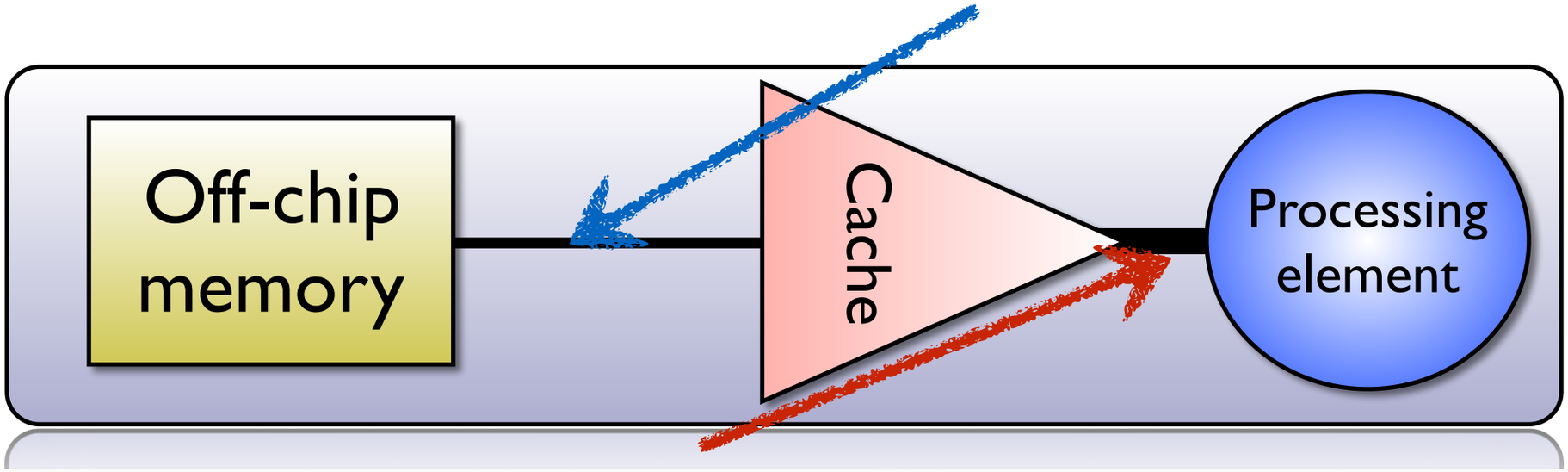
- The memory bandwidth is the constraining resource *(Off-chip system memory)*
- Transfer-computation overlap
- Memory footprint does not fit in the cache

We want a model that relates:

- Computing performance [GFLOP/s]
- Off-chip memory traffic [GB]

➡ New concept: the operational intensity [FLOP/Byte]

Operational intensity



- Not equivalent to arithmetic intensity [1], machine balance [2] (which refer to traffic between the processor and the cache)
- Not forcedly bound to FLOP/Bytes (e.g. Comparison/Byte)

[1] Harris, M. Mapping Computational Concepts To Gpus. In ACM SIGGRAPH Courses, 2005.
[2] Callahan, D., Cocke, J., And Kennedy, K. Estimating Interlock And Improving Balance For Pipelined Machines (1988)

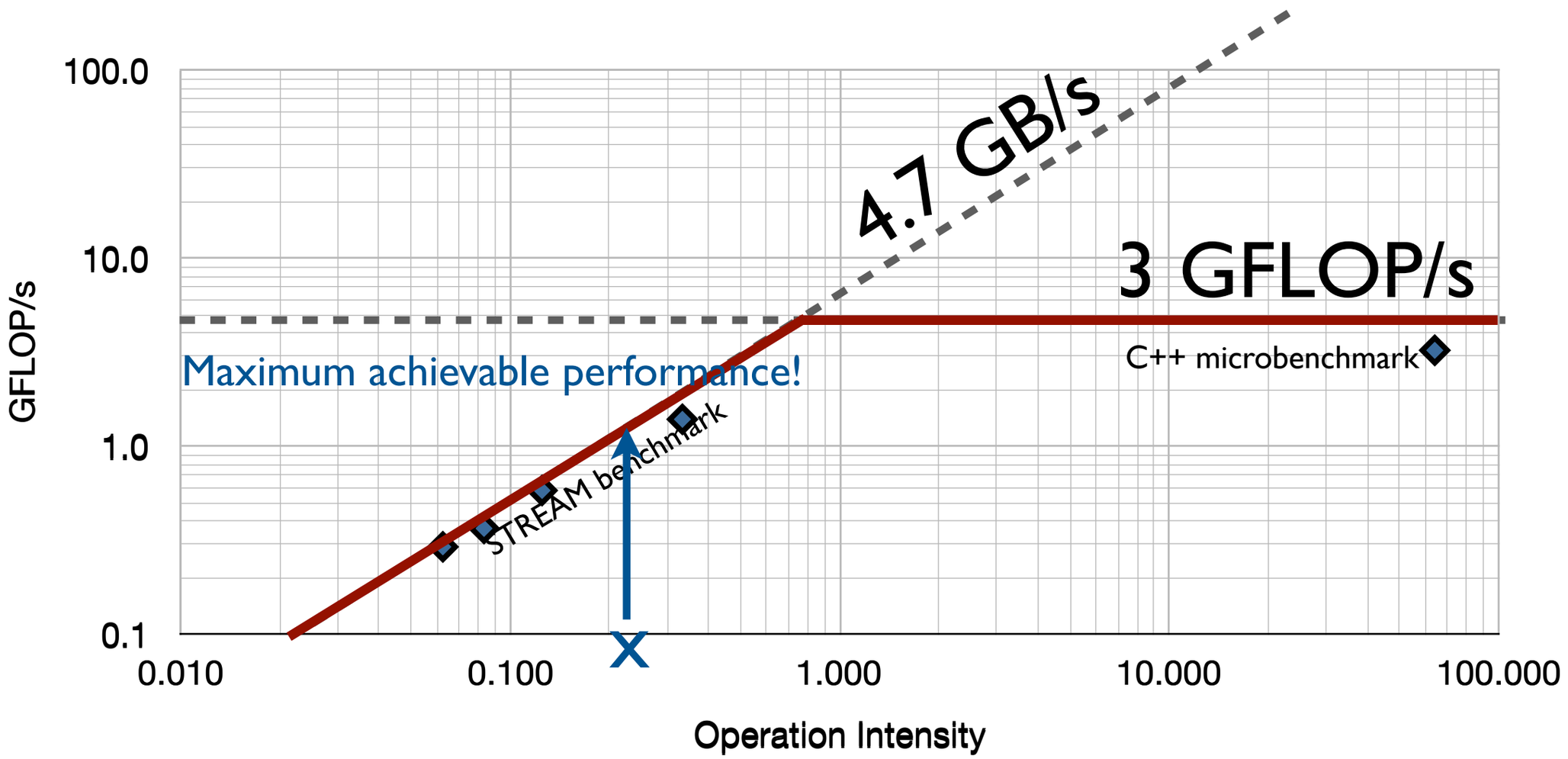
The roofline model

- The roofline is a ***log-log plot***
- It relates:
  - Performance f [FLOP/s] with
  - Operational intensity r [FLOP/Byte]
- Two theoretical regimes for a kernel k:
  1. Performance of k is limited by the DRAM bandwidth:

     → $f(r_k) = r_k \; b_{peak}$
  2. Performance of k is limited by the compute power:

     → $f(r_k) = f_{peak}$

The roofline model



◆ 4x Quad-Core AMD Opteron 8380 @ 2.5GHz - 1 Thread - C++

Nominal performance

How to estimate nom

- From the hardw
- On Linux
  - type `cat /proc/cpuir`
  - And `sudo dmidecode`

```
menahel@Cyrus:~$ sudo dmidecode --type memory
[sudo] password for menahel:
# dmidecode 2.9
SMBIOS 2.6 present.

Handle 0x001D, DMI type 16, 15 bytes
Physical Memory Array
        Location: System Board Or Motherboard
        Use: System Memory
        Error Correction Type: None
        Maximum Capacity: 32 GB
        Error Information Handle: Not Provided
        Number Of Devices: 4

Handle 0x001F, DMI type 17, 28 bytes
Memory Device
        Array Handle: 0x001D
        Error Information Handle: Not Provided
        Total Width: 64 bits
        Data Width: 64 bits
        Size: 4096 MB
        Form Factor: DIMM
        Set: None
        Locator: DIMM3
        Bank Locator: DIMM 3
        Type: <OUT OF SPEC>
        Type Detail: Synchronous
        Speed: 1333 MHz (0.8 ns)
        Manufacturer: 0x04CD
        Serial Number: 0x00000000
```

Examples

Processor Clock/sec

Vector size ((SSE, AVX,..)

instructions per clock, FMAs

No. of cores

➡ **PP**: 2.5 [Ghz] * 4 [SIMD-wi]

➡ **PB**: 1.3 [Ghz] * 64 [bits]

Memory Clock/sec

Channel size

No. channels

bits/Byte

Measured performance

Microbenchmarks:

- STREAM benchmark or similar
- Code: computepower.cpp, `bandwidth.cpp`

Expected discrepancy from nominal quantities:

✓ FLOP/s: 90-100% of nominal performance

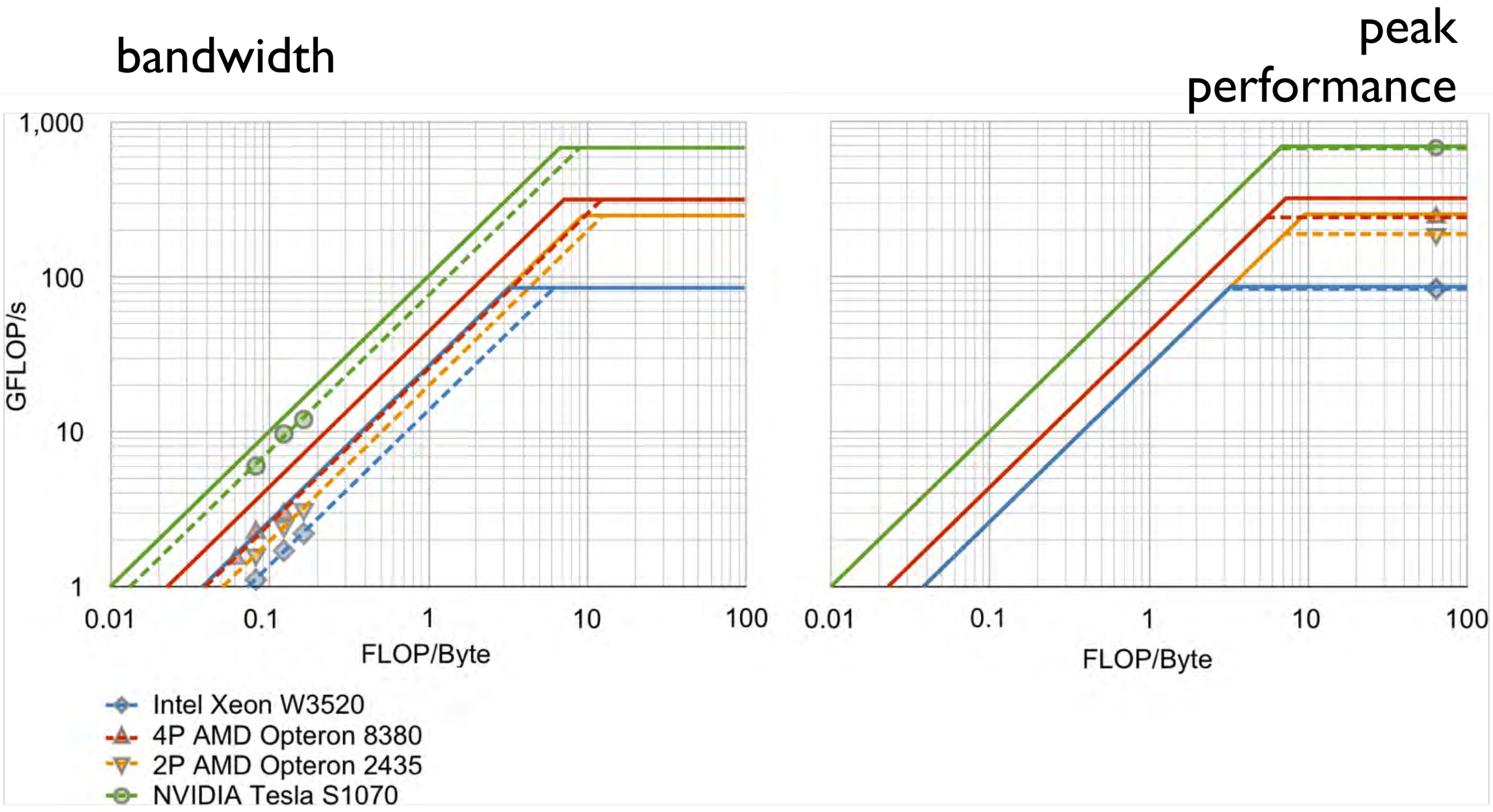
x GByte/s : 50-70% of nominal performance

Discrepancies reveal:

➡ *Programming skills in extracting system performance*

➡ *Best case scenario for more complex kernels*

Discrepancies (measured(dashed lines) vs nominal (solid lines))



- Real scenario: brutus cluster node
- 16 CPU cores (max), 1 Tesla GPU
- Bandwidth discrepancies more pronounced

GPU gain vs. CPU

	Observed Peak [GFLOP/s]	Theoretical Peak [GFLOP/s]	Efficiency [%]
Xeon W3520	83.2	85.4	97
Opteron 8380	237.2	320.0	74
Opteron 2435	184.9	249.6	74
Tesla S1070	684.0	691.2	99

	Observed Peak [GB/s]	Theoretical Peak [GB/s]	Efficiency [%]
Xeon W3520	13.4	25.5	52
Opteron 8380	24.5	42.5	58
Opteron 2435	19.2	25.6	75
Tesla S1070	73.8	100.2	74

GPUs are faster than CPUs

- **GFLOP/s: 2X-8X**
- **GB/s: 1.7X-5.5X**

The roofline model

✓ Run once per platform, not once per kernel

x Estimation of operational intensities (Flops/byte) can be tricky

x What happens if you compute them wrong?

	add $z_i = x_i + y_i$	scale $z_i = \alpha x_i$	triad $z_i = \alpha x_i + y_i$
Intel Xeon W3520	$\frac{1}{12}$ 2 read (x,y) 1 write(z)	$\frac{1}{8}$ 1 read (x) 1 write(z)	$\frac{2}{12}$ 2 read (x,y) 1 write(z)
4P AMD Opteron 8380	$\frac{1}{16}$ 3 read (x,y, z) 1 write(z)	$\frac{1}{12}$ 2 read (x,z) 1 write(z)	$\frac{2}{16}$ 3 read (x,y,z) 1 write(z)
2P AMD Opteron 2435	$\frac{1}{12}$	$\frac{1}{8}$	$\frac{2}{12}$
NVIDIA Tesla S1070	$\frac{1}{12}$	$\frac{1}{8}$	$\frac{2}{12}$

NOTE:
Cache Dependent Numbers

➡ *What is the operational intensity of your algorithm steps?*

Operational

How to determine the DRAM traffic?

1. By hand (difficult)

- Without cache hierarchy: $t^{write} = t^{r}$
- With cache hierarchy:
  - Write-through: $t^{write} = t^{read}$
  - Write-back(*): $t^{read} = 2\ t^{write}$

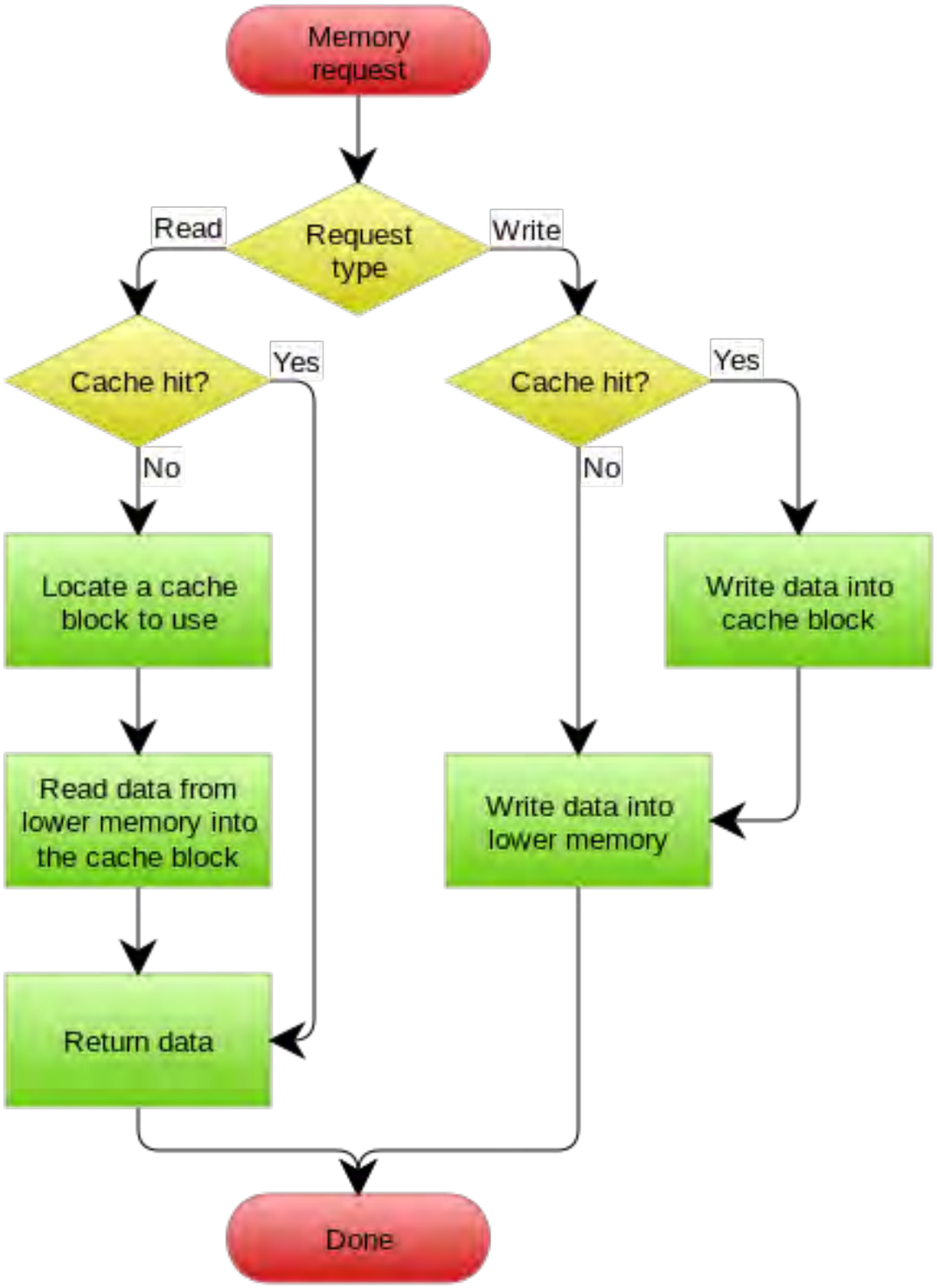
2. With perfomance counters (difficu

(*)For write back : a read cache miss requires 2 memory accesses: one is to write back to the memory the data and one to read another piece of data

HYPERTRANSPORT_LINK0:ADDRESS_EXT_DWORD_SENT:	265,803	529,596
HYPERTRANSPORT_LINK0:PER_PACKET_CRC_SENT:	0	0
HYPERTRANSPORT_LINK0:SUBLINK_MASK:	0	0
HYPERTRANSPORT_LINK0:ALL:	0	0
HYPERTRANSPORT_LINK1:COMMAND_DWORD_SENT:	480	546
HYPERTRANSPORT_LINK1:DATA_DWORD_SENT:	54	63
HYPERTRANSPORT_LINK1:BUFFER_RELEASE_DWORD_SENT:	49	66
HYPERTRANSPORT_LINK1:NOP_DWORD_SENT:	5,987,813	6,952,146
HYPERTRANSPORT_LINK1:ADDRESS_EXT_DWORD_SENT:	427	495
HYPERTRANSPORT_LINK1:PER_PACKET_CRC_SENT:	0	0
HYPERTRANSPORT_LINK1:SUBLINK_MASK:	0	0
HYPERTRANSPORT_LINK1:ALL:	0	0
HYPERTRANSPORT_LINK2:COMMAND_DWORD_SENT:	267,797	533,392
HYPERTRANSPORT_LINK2:DATA_DWORD_SENT:	6,491	11,494
HYPERTRANSPORT_LINK2:BUFFER_RELEASE_DWORD_SENT:	533,832	1,059,054
HYPERTRANSPORT_LINK2:NOP_DWORD_SENT:	13,859,790	15,150,480
HYPERTRANSPORT_LINK2:ADDRESS_EXT_DWORD_SENT:	266,455	528,669
HYPERTRANSPORT_LINK2:PER_PACKET_CRC_SENT:	0	0
HYPERTRANSPORT_LINK2:SUBLINK_MASK:	0	0
HYPERTRANSPORT_LINK2:ALL:	0	0
HYPERTRANSPORT_LINK3:COMMAND_DWORD_SENT:	567,900	490,222
HYPERTRANSPORT_LINK3:DATA_DWORD_SENT:	770,644	870,606
HYPERTRANSPORT_LINK3:BUFFER_RELEASE_DWORD_SENT:	795,446	791,627
HYPERTRANSPORT_LINK3:NOP_DWORD_SENT:	606,005	797,625
HYPERTRANSPORT_LINK3:ADDRESS_EXT_DWORD_SENT:	571,646	454,861
HYPERTRANSPORT_LINK3:PER_PACKET_CRC_SENT:	571,646	454,861
HYPERTRANSPORT_LINK3:SUBLINK_MASK:	571,646	454,861
HYPERTRANSPORT_LINK3:ALL:	1,043,153	1,476,221
READ_REQUEST_TO_L3_CACHE:READ_BLOCK_EXCLUSIVE:	292,322	404,937
READ_REQUEST_TO_L3_CACHE:READ_BLOCK_SHARED:	271,892	233,901
READ_REQUEST_TO_L3_CACHE:READ_BLOCK_MODIFY:	65,176	38,351
READ_REQUEST_TO_L3_CACHE:CORE_0_SELECT:	271,892	233,901
READ_REQUEST_TO_L3_CACHE:CORE_1_SELECT:	70,299	36,586
READ_REQUEST_TO_L3_CACHE:CORE_2_SELECT:	70,299	36,586
READ_REQUEST_TO_L3_CACHE:CORE_3_SELECT:	70,299	36,586
READ_REQUEST_TO_L3_CACHE:ALL:	503,989	598,574
L3_CACHE_MISSES:READ_BLOCK_EXCLUSIVE:	1,954	1,081
L3_CACHE_MISSES:READ_BLOCK_SHARED:	1,954	1,081
L3_CACHE_MISSES:READ_BLOCK_MODIFY:	1,954	1,081
L3_CACHE_MISSES:CORE_0_SELECT:	85,326	26,817
L3_CACHE_MISSES:CORE_1_SELECT:	90,804	26,770
L3_CACHE_MISSES:CORE_2_SELECT:	144,110	84,079
L3_CACHE_MISSES:CORE_3_SELECT:	144,110	84,079
L3_CACHE_MISSES:ALL:	232,850	109,626
L3_FILLS_CAUSED_BY_L2_EVICTIONS:SHARED:	710,687	884,724
L3_FILLS_CAUSED_BY_L2_EVICTIONS:EXCLUSIVE:	710,687	884,724
L3_FILLS_CAUSED_BY_L2_EVICTIONS:OWNED:	0	0
L3_FILLS_CAUSED_BY_L2_EVICTIONS:MODIFIED:	0	0
L3_FILLS_CAUSED_BY_L2_EVICTIONS:CORE_0_SELECT:	0	0
L3_FILLS_CAUSED_BY_L2_EVICTIONS:CORE_1_SELECT:	0	0
L3_FILLS_CAUSED_BY_L2_EVICTIONS:CORE_2_SELECT:	0	0
L3_FILLS_CAUSED_BY_L2_EVICTIONS:CORE_3_SELECT:	0	0
L3_FILLS_CAUSED_BY_L2_EVICTIONS:ALL:	711,311	884,820
L3_EVICTIONS:SHARED:	31,575	30,449
L3_EVICTIONS:EXCLUSIVE:	31,253	30,543
L3_EVICTIONS:OWNED:	31,223	30,047
L3_EVICTIONS:MODIFIED:	31,466	30,830
L3_EVICTIONS:ALL:	31,503	31,262
PAGE_SIZE_MISMATCHES:GUEST_LARGER:	59	73
PAGE_SIZE_MISMATCHES:MTRR_MISMATCH:	59	262,223
PAGE_SIZE_MISMATCHES:HOST_LARGER:	69	65
PAGE_SIZE_MISMATCHES:ALL:	69	262,218
RETIRED_X87_OPS:ADD_SUB_OPS:	67,019,470	25,164,200
RETIRED_X87_OPS:MUL_OPS:	67,019,460	25,164,190
RETIRED_X87_OPS:DIV_OPS:	67,019,480	25,164,220
RETIRED_X87_OPS:ALL:	67,019,470	25,164,200
IBS_OPS_TAGGED:	27	33

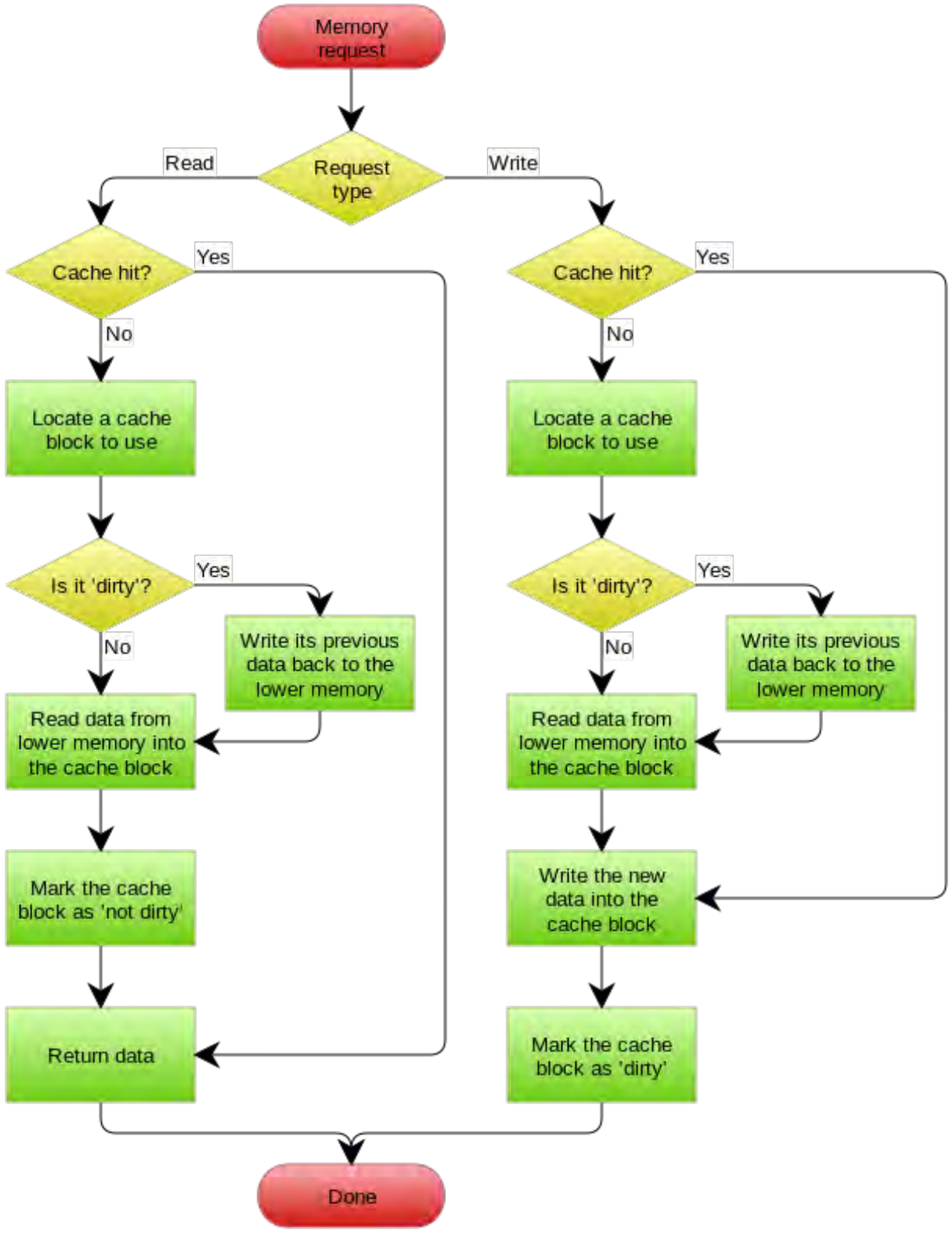
WRITE THROUGH CACHE

update different levels of cache and main memory



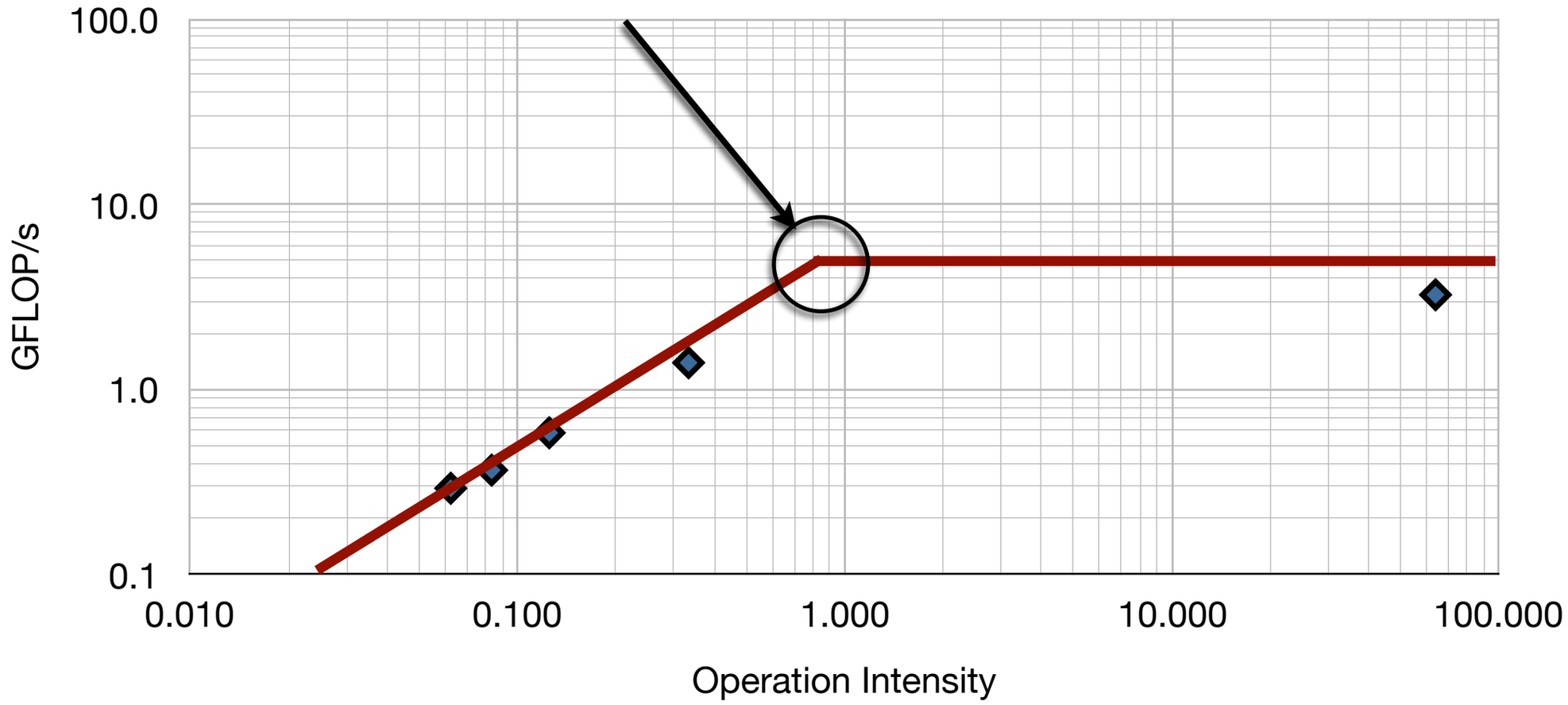
WRITE BACK CACHE

write on main memory only when you need to clean some level of cache



http://www.computerweekly.com/feature/Write-through-write-around-write-back-Cache-explained

The ridge point



- Ridge point **characterize the overall machine performance**

➡ Ridge point "to the left": it is relatively easy to get peak performance

➡ Ridge point "to the right": it is difficult to get peak performance

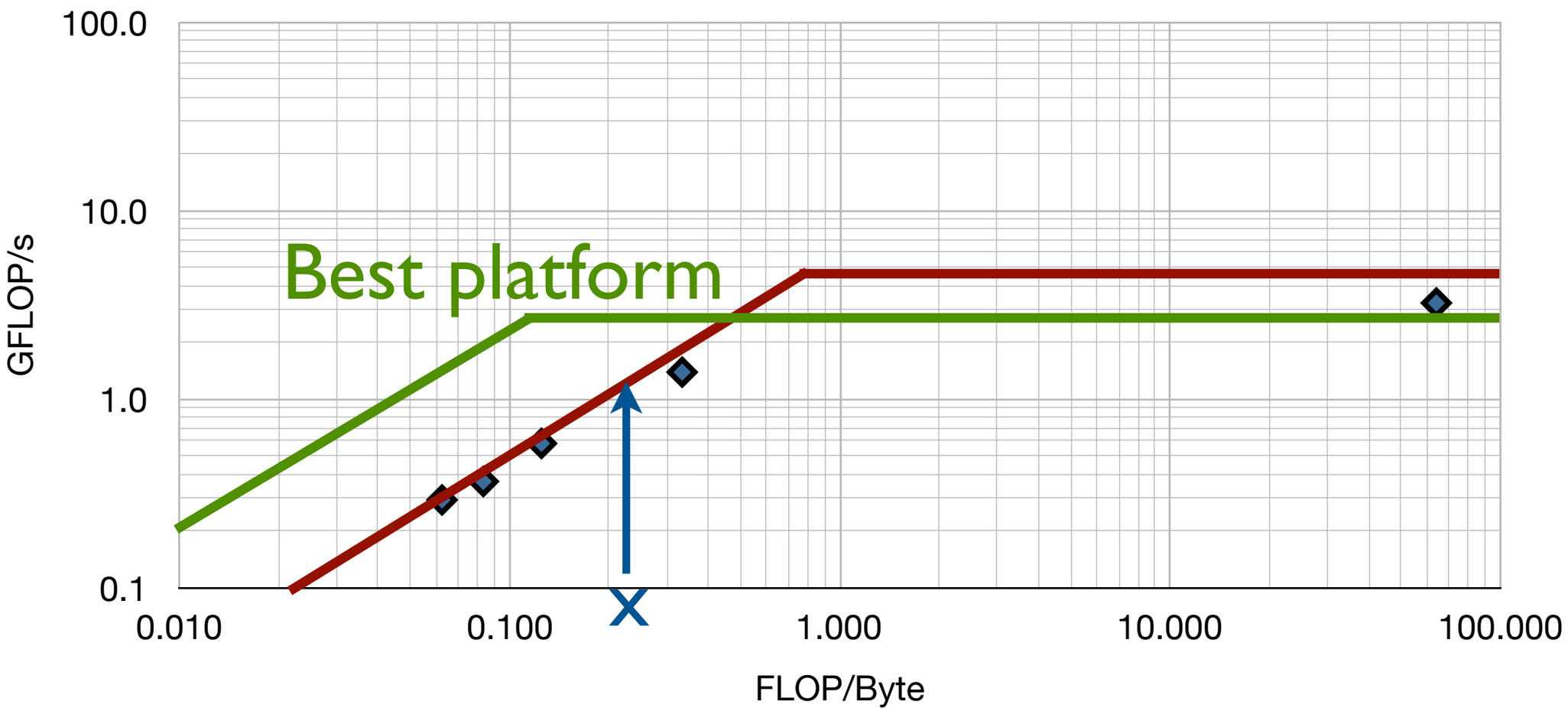
What does it mean "a ridge point to the right" anyway?

Production software

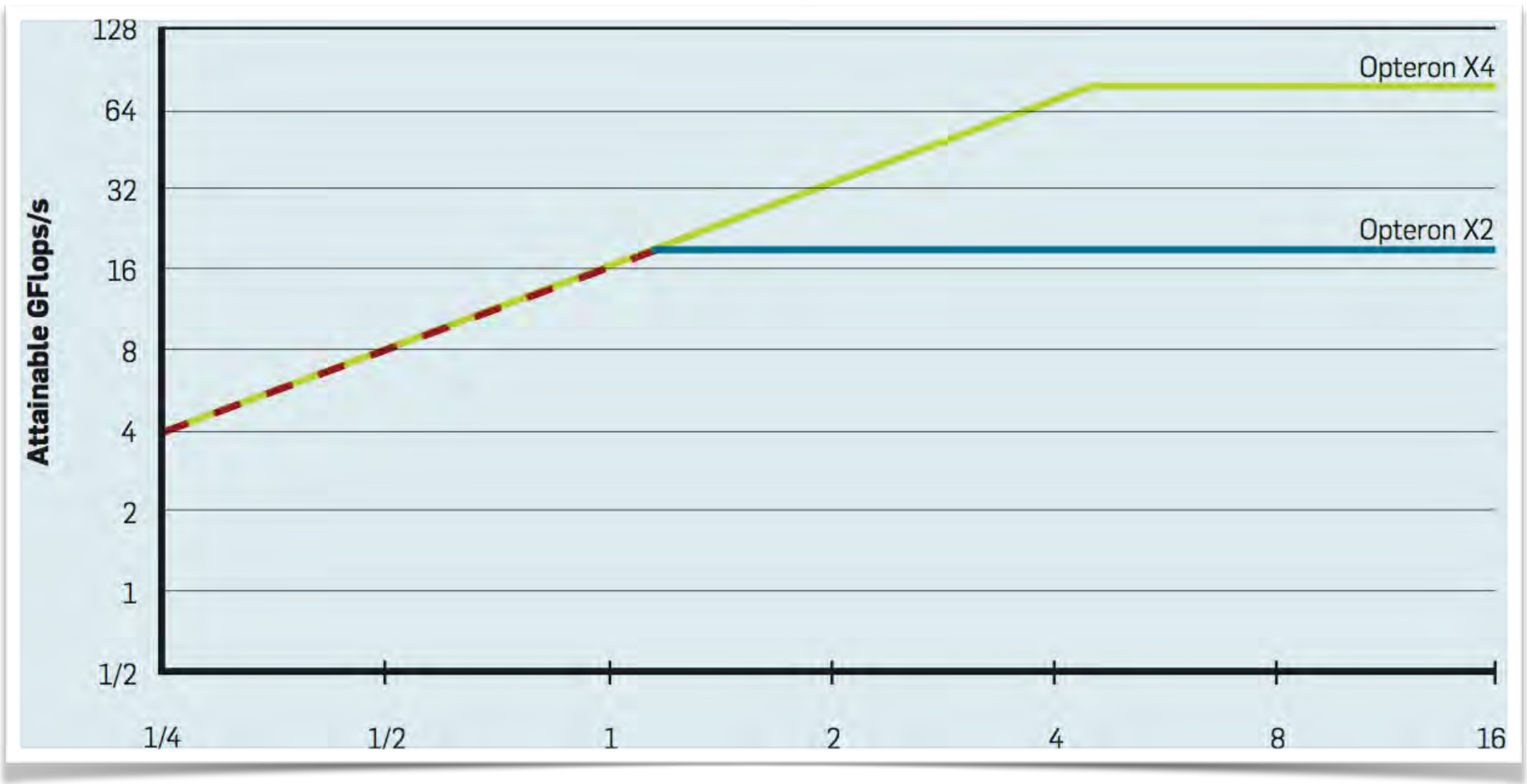
Assumption: production-ready software

- Limited set of algorithms
- Fixed set of kernels
- Fixed operational intensities

} ➡ Best hardware solution?



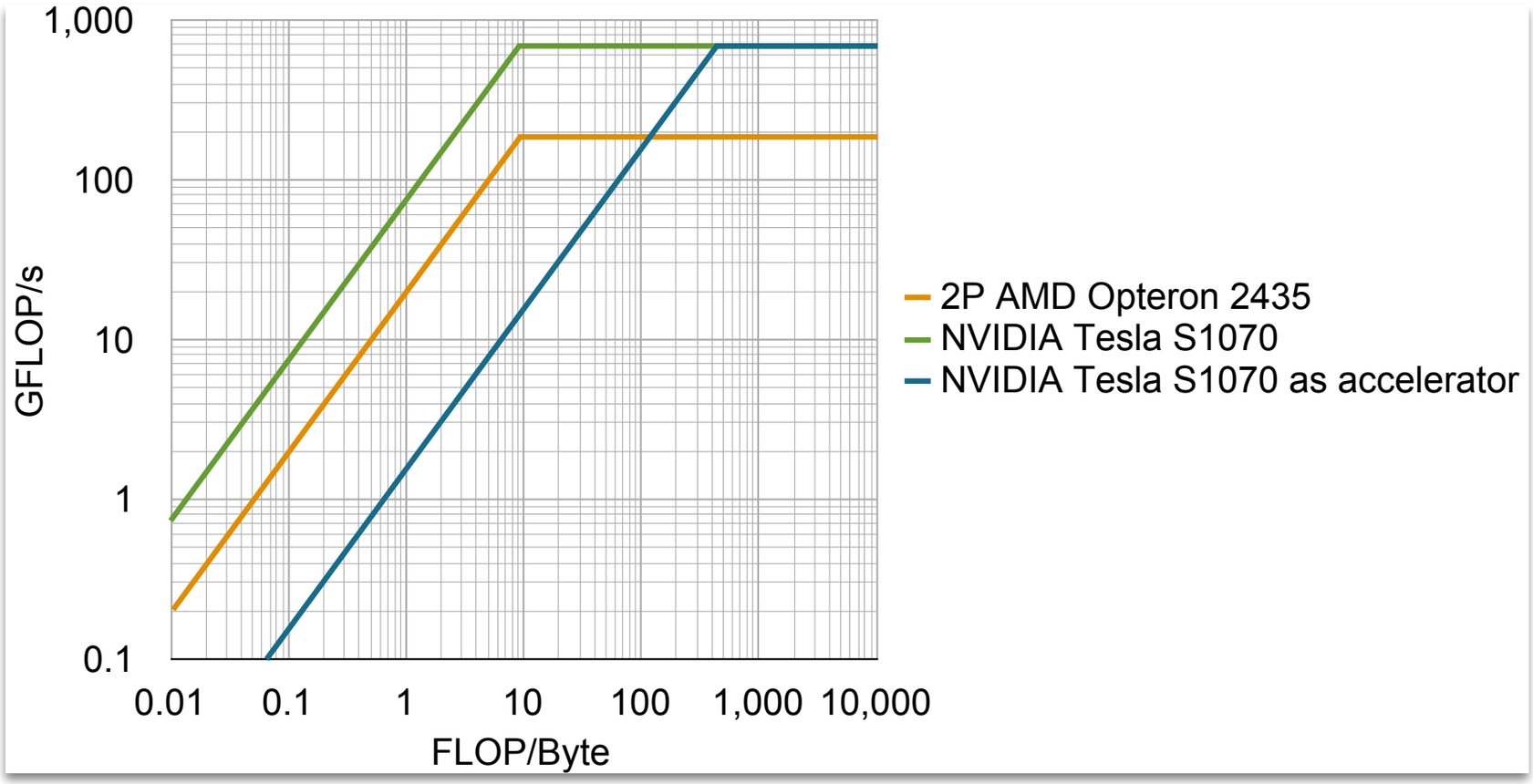
Is Moore worth?



It depends:

- On the ridge point
- On the operational intensity of the considered kernels

GPU as an "accelerator"



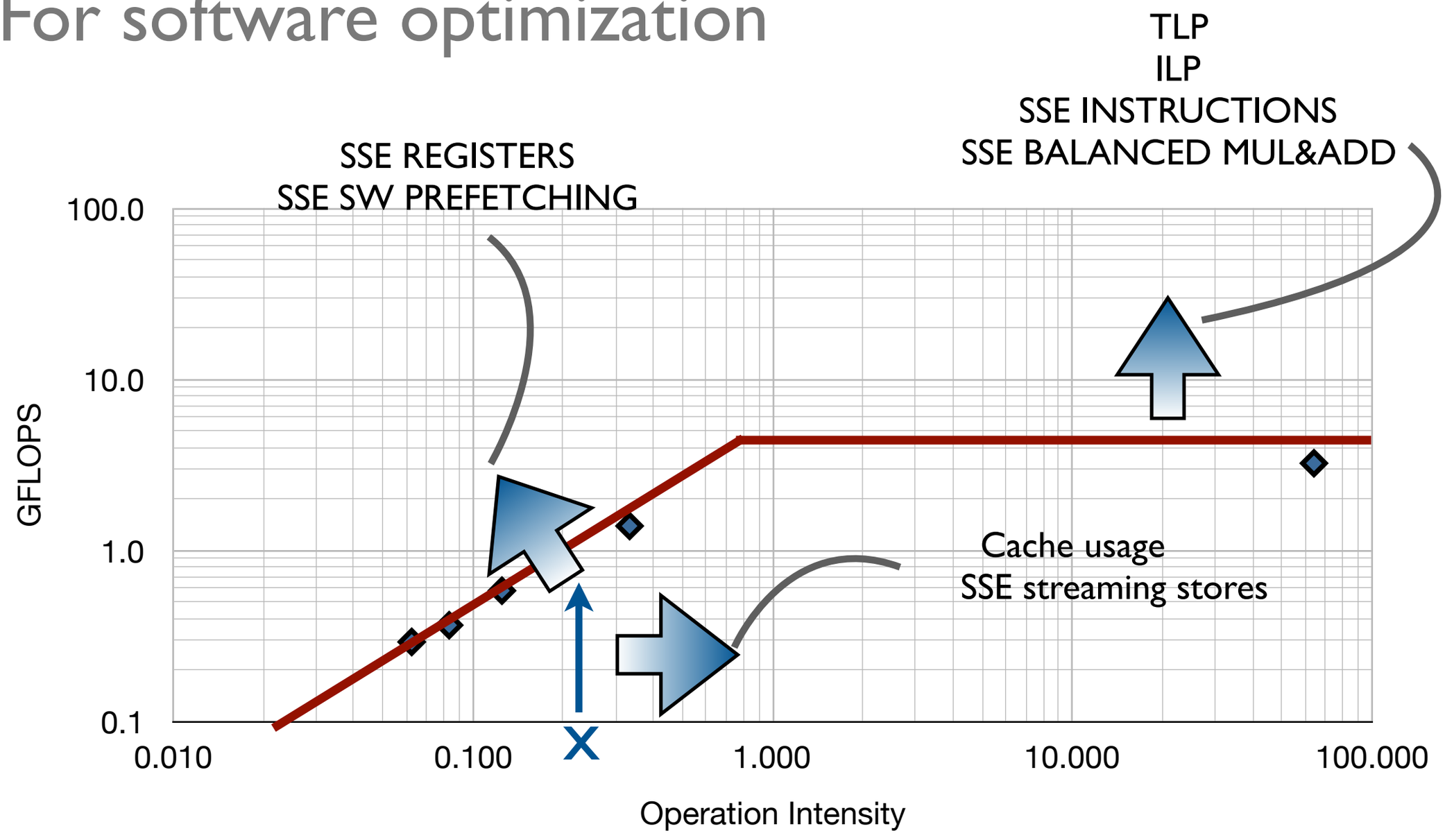
- Ridge point at 400 FLOP/Byte
- Almost no kernels can approach this value

➡ Move all data to the GPU (bad idea in general)

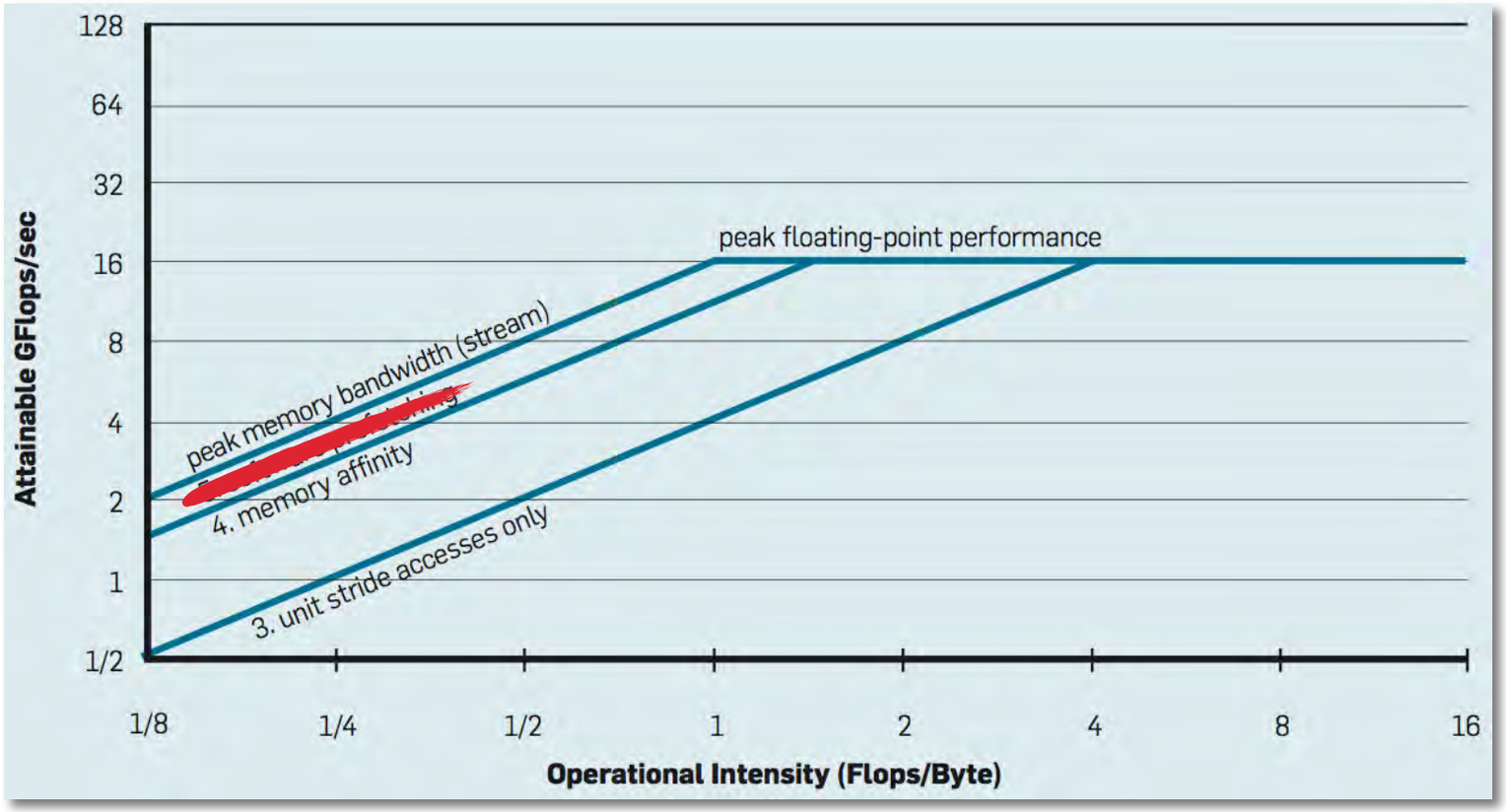
➡ Hide data transfers with GPU computation

The roofline model

For software optimization



Bandwith ceilings



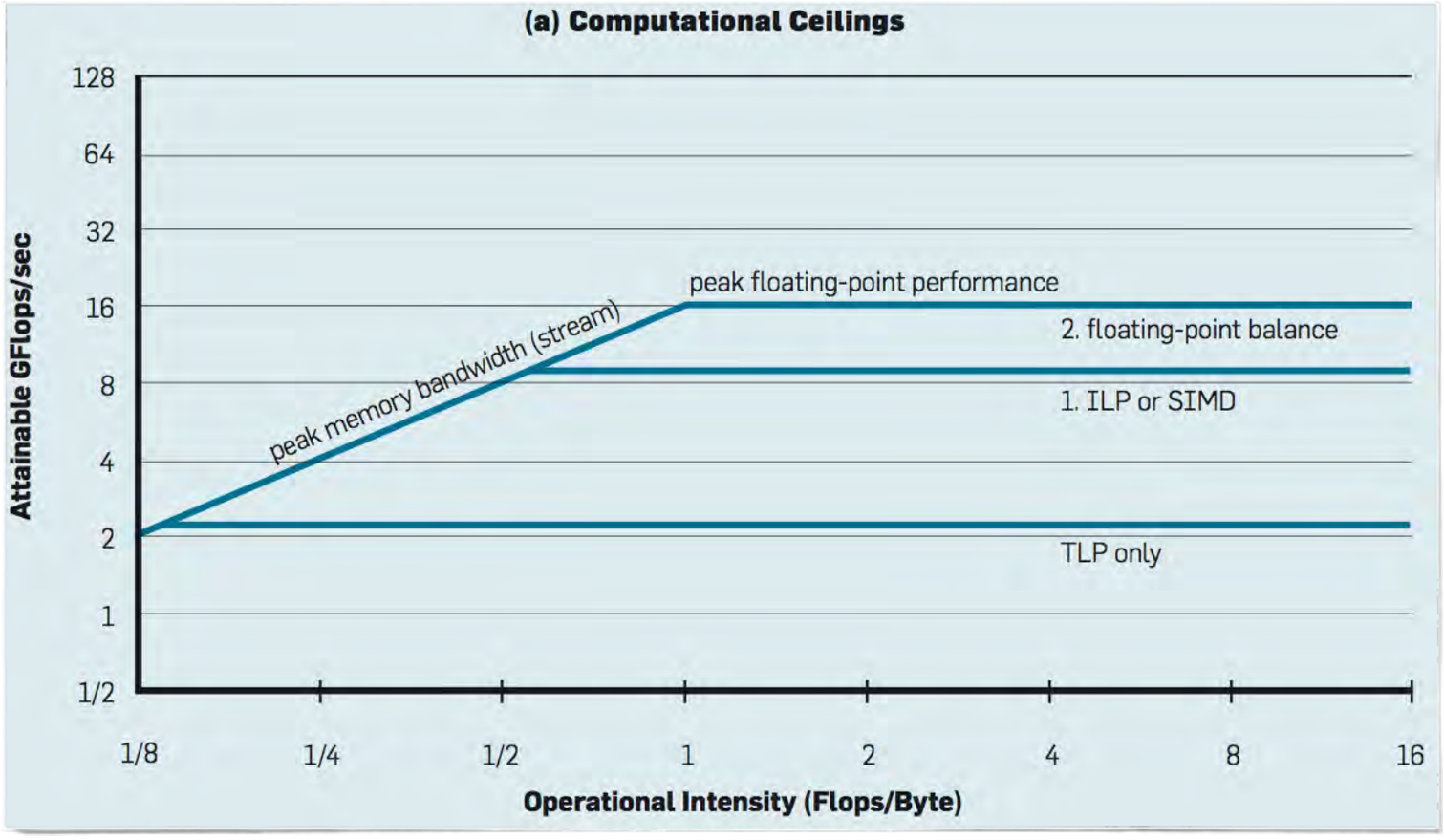
GPU:

- Coalesced access
- Non-temporal writes
- Use of textures

CPU:

- Unit-stride access
- NUMA-awareness
- Non-temporal writes

Performance ceilings



GPU optimization:

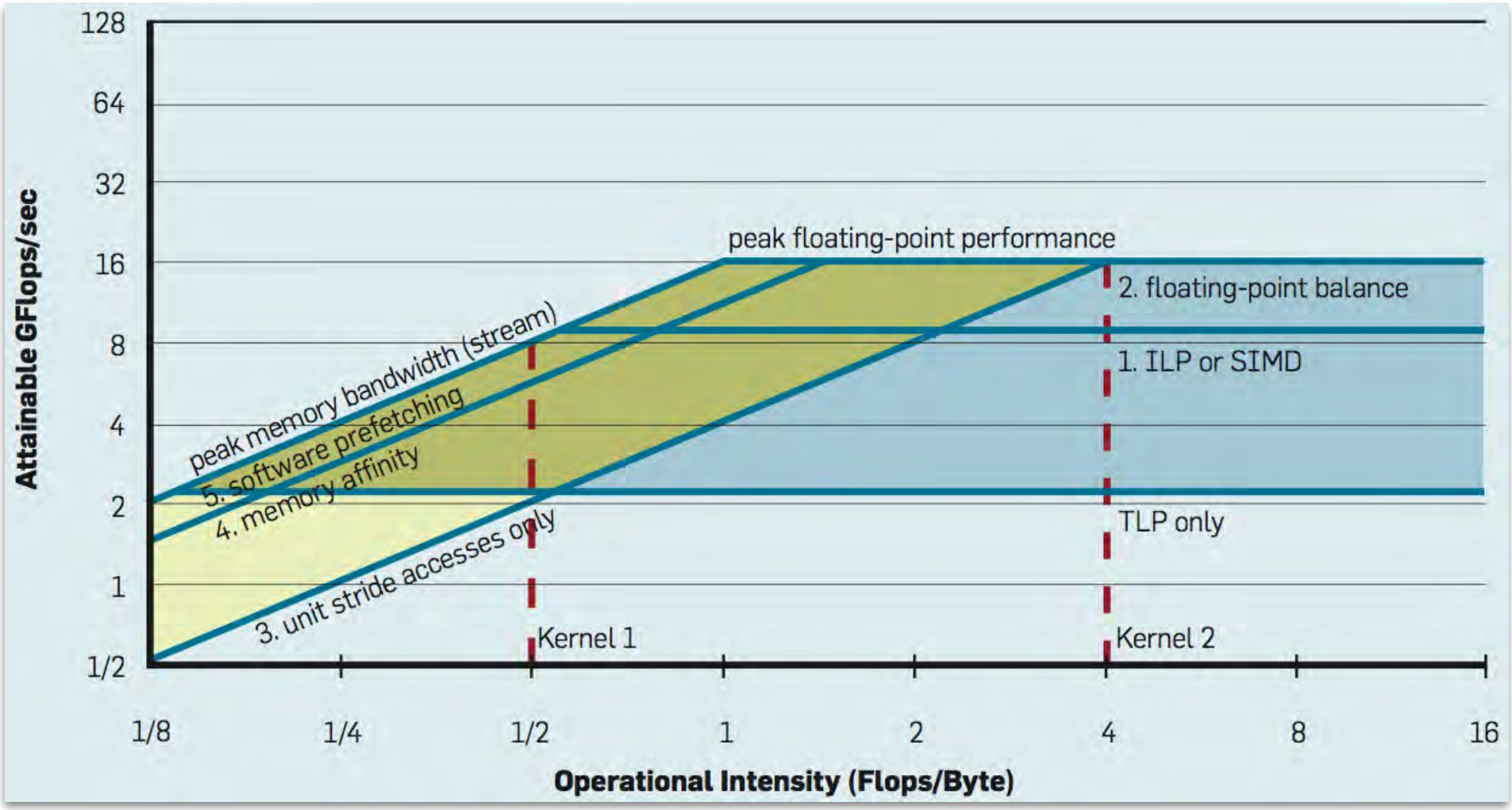
- Increase DLP
- Increase ILP
- Loop unrolling
- Use FMAD

- The roofline is an upper bound to performance
- Optimization steps should break performance ceilings

➡ What software optimization should we perform?

➡ In what order?

In what order?



- Some optimizations are not orthogonal
- Break the next performance ceiling
- Ceilings are ranked bottom-to-top
  - Bottom: little effort
  - Top: high effort OR inherently lacking

The first optimization step

For a kernel at the left of the ridge point:

- First step: ignore ceilings, maximize reuse

➡ Increased operational intensity

How to achieve this?

1. Minimize conflict cache misses (data reordering)
2. Minimize capacity cache misses
    - Computation reordering
    - Non-temporal writes/streaming stores

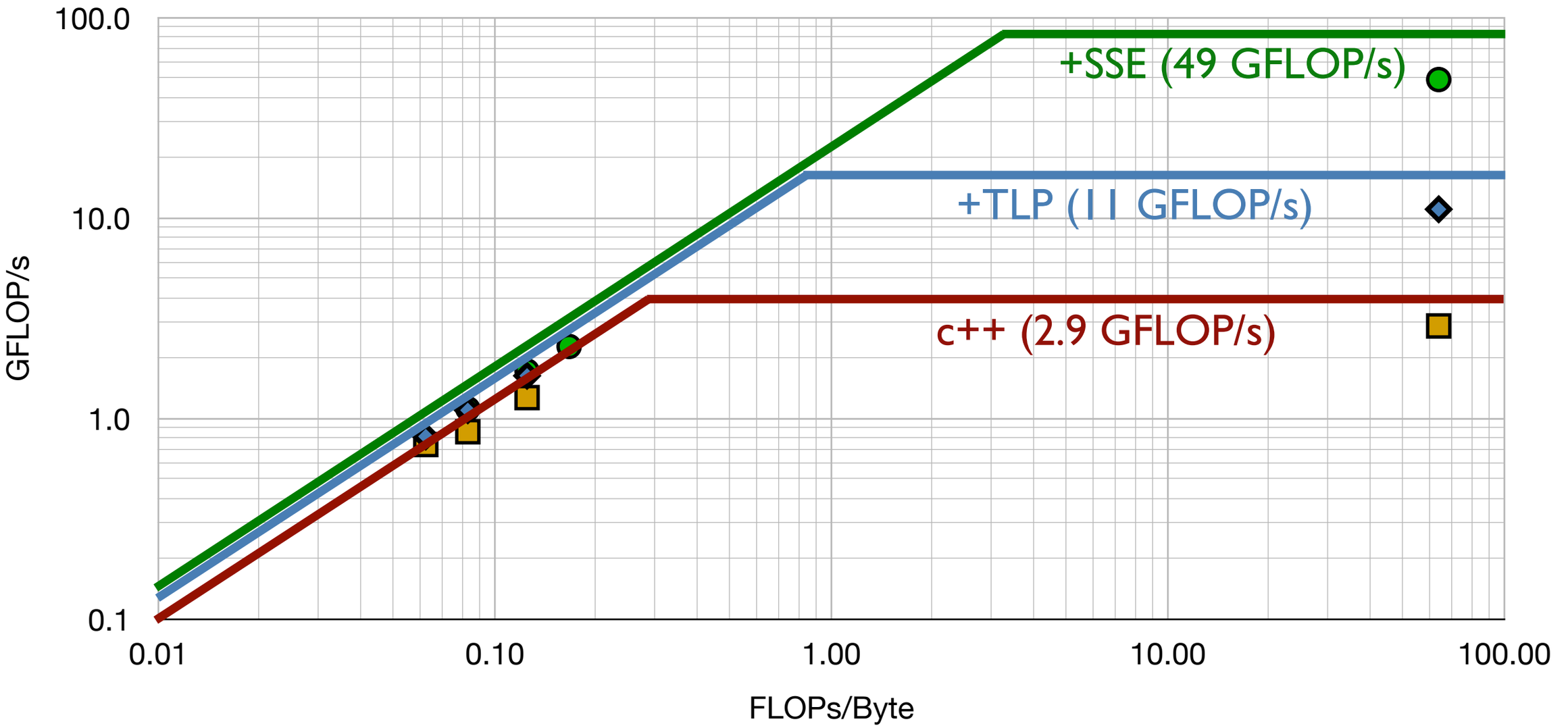
How to achieve reordering?

- Tiling
- Blocking
- Space filling curves

What is the upper bound?

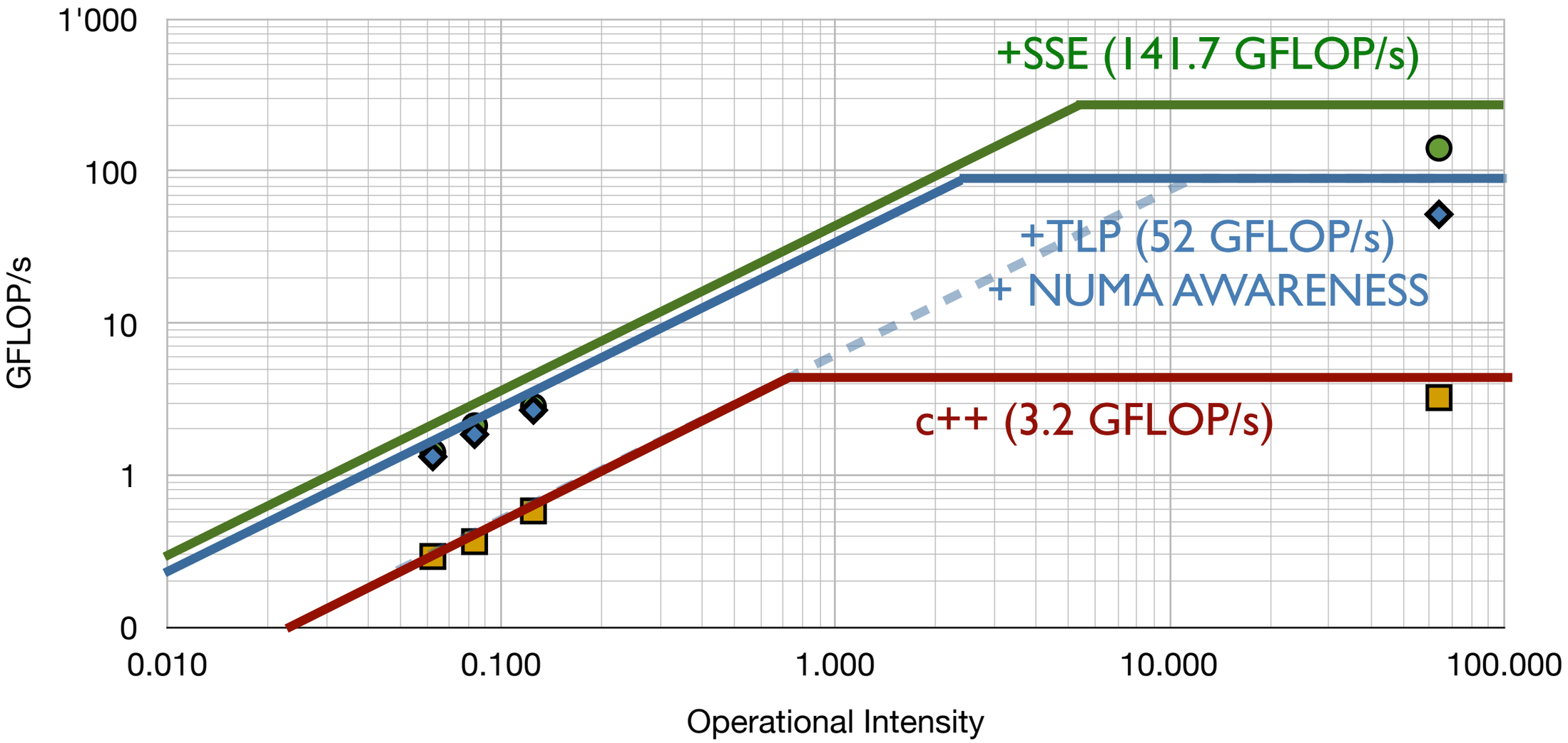
➡ Compulsory cache misses

Performance ceilings



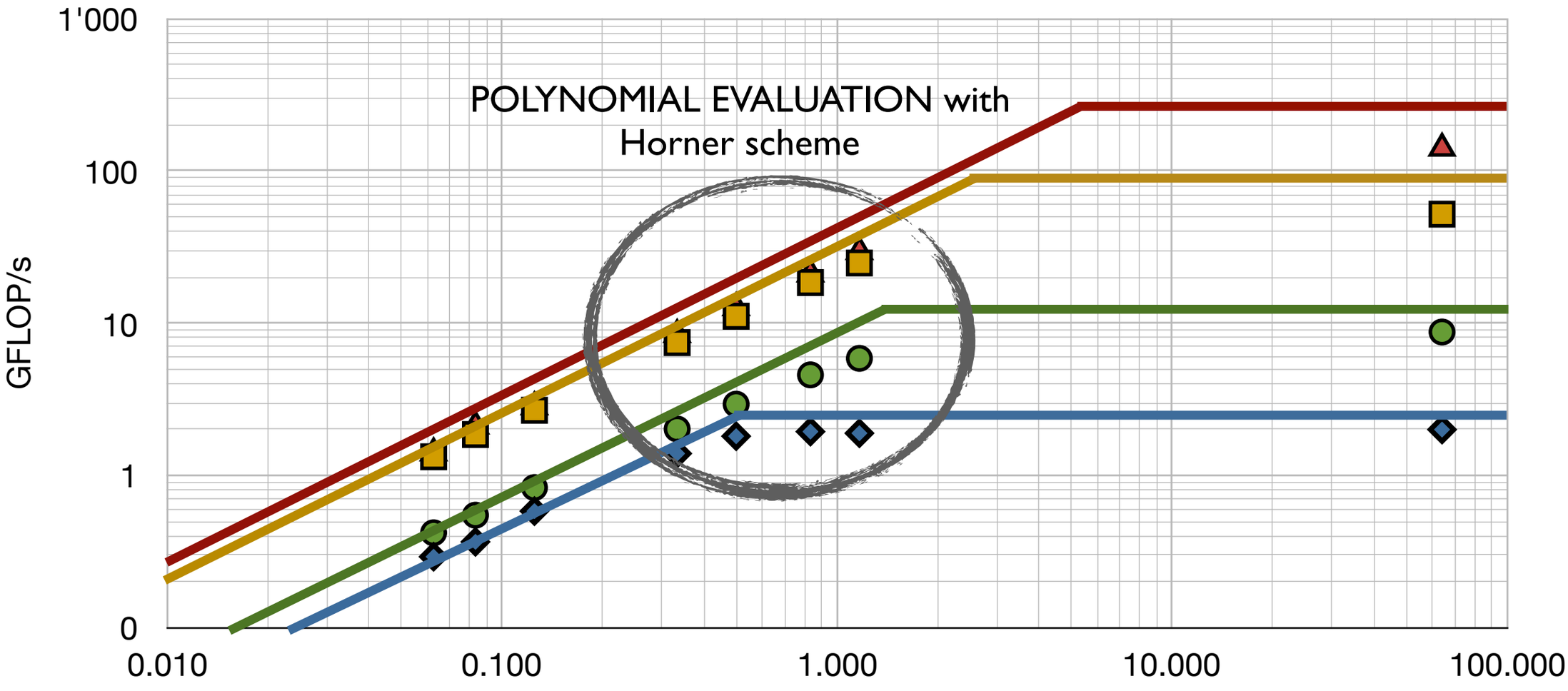
PERF. GAIN:17X

Performance ceilings



PERF. GAIN: 44X

Confirming the roofline



The roofline is there.
(for real)

Quiz (1)

- Does the roofline model account for caches and advanced memory types (eg. caches, textures, non-temporal writes)?

- When the operational intensity is low:
  by doubling the cache size, does the operational intensity increase?

- Does the model take into account long memory latencies?

Quiz (2)

- Does the roofline model have little to do with multi/ many-core computing?

- Does the FLOP/s roofline, ignoring the integer computation, possibly produce a wrong estimation of the performance?

- Is the roofline model limited to easily optimized kernels that never hit the cache?

Quiz (3)

- Is the roofline model limited to the performance analysis of kernels performing floating-point computations?

- Is the roofline model forcedly bound to the off-chip memory traffic?

Study case

Computational settings:

- Grid-based simulation of the diffusion process
- Two-dimensional problem
- Structured grid, uniform grid spacing
- Explicit time discretization

Computer settings:

- Grid data large enough to not fit in the cache
- Floating point arithmetic in single precision

$$u : [0,T] \times \Omega^2 \rightarrow R$$

$$\partial_t u - (\partial_{xx} + \partial_{yy})u = 0$$

$$u(t,x) = u_0(x), \;\; x \in \Gamma$$

$$\begin{aligned} \text{tmp} &:= \Delta\omega^n \\ \omega^{*,n} &:= \omega^n + \frac{\delta t}{2}\,\text{tmp} \\ \text{tmp} &:= \Delta\omega^{*,n} \\ \omega^{n+1} &:= \omega^n + \delta t\,\text{tmp}. \end{aligned}$$

Step 1:

$$\text{tmp}_{i,j} = a_0 \cdot u^n_{i,j} + a_1 \cdot (u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j-1} + u^n_{i,j+1})$$

Step 2:

$$u^{n+1}_{i,j} = u^n_{i,j} + a_2 \cdot (\text{tmp}^n_{i+1,j} + \text{tmp}^n_{i-1,j} + \text{tmp}^n_{i,j-1} + \text{tmp}^n_{i,j+1} - 4\text{tmp}_{i,j})$$

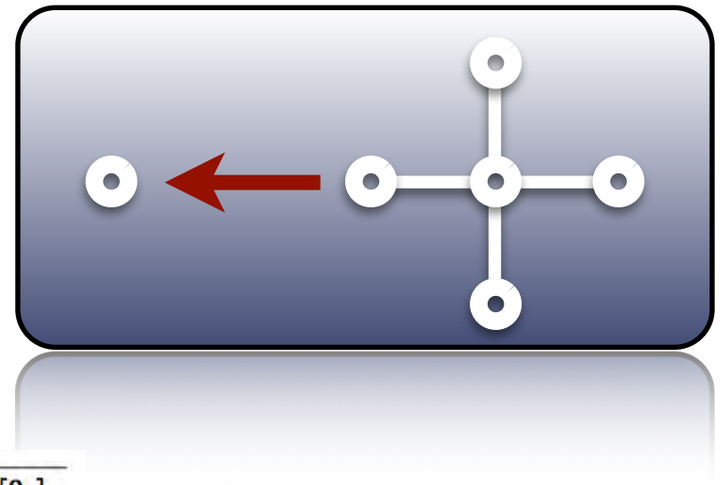
$$a_0 = 1 - 2\frac{dt}{h^2} \qquad a_1 = \frac{dt}{2h^2} \qquad a_2 = \frac{dt}{h^2}$$

Operational intensities

NO WB-WRITE ALLOCATE CACHE MISSES

NAIVE, STEP1: R = 6 FLOPS/(4*4 BYTES) = 0.375

TILED, STEP 1: R = (3/4)*(256/272) = 0.47



	Observed Peak [GB/s]	Theoretical Peak [GB/s]	Efficiency [%]
Xeon W3520	13.4	25.5	52
Opteron 8380	24.5	42.5	58
Opteron 2435	19.2	25.6	75
Tesla S1070	73.8	100.2	74

EXPECTED PERFORMANCE (SIMPLE MEM-LAYOUT):

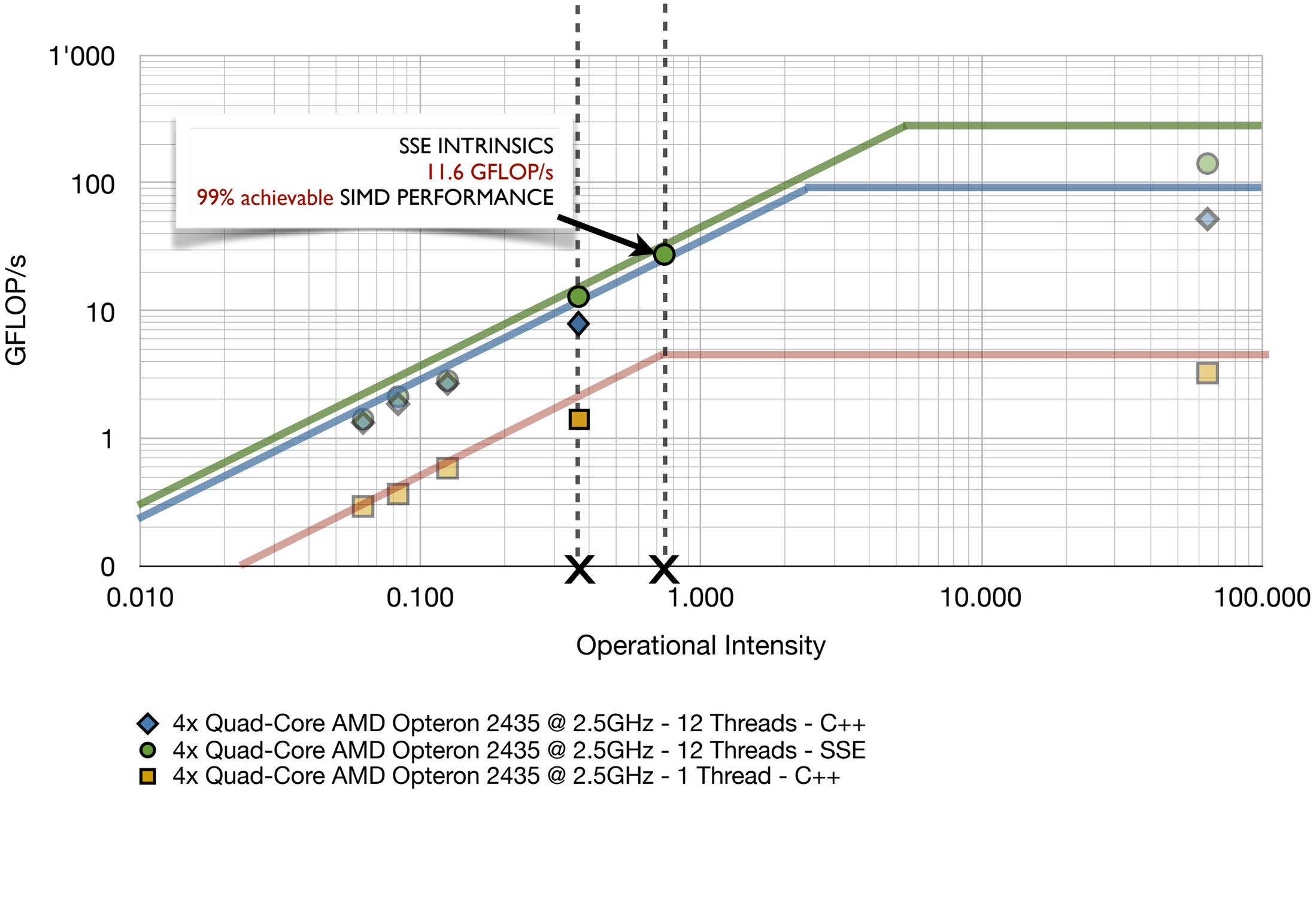
OPTERON 2435: 19.2*0.375 = **7.2** [GFLOP/S]

TESLA S1070: 73.8*0.375 = **27.7** [GFLOP/S]

TILING, EXPECTED PERFORMANCE:

OPTERON 2435: 19.2*0.47 = **9** [GFLOP/S]

CPU (similar) results



Results

EXPECTED PERFORMANCE (SIMPLE MEM-LAYOUT):

OPTERON 2435: 19.2*0.375 = 7.2 GFLOPS

TESLA S1070: 73.8*0.375 = 27.7 GFLOPS

CPU-TILING, EXPECTED PERFORMANCE:

OPTERON 2435: 19.2*0.47 = 9 GFLOPS

OBSERVED PERFORMANCE (SIMPLE MEM-LAYOUT):

OPTERON 2435: **6.6 [GFLOP/S]** (**91%** EFF.)

TESLA S1070: **30 [GFLOP/S]** (~**100%**, CACHING EFFECTS)

CPU-TILING, OBSERVED PERFORMANCE:

OPTERON 2435: **7.8 [GFLOP/S]** (**86%** EFF.), MORE COMPLEX CODE!

COMPILER OPTIMIZATIONS: LESS EFFECTIVE

GPU gain vs. CPU

Back to the 100X story...

SPEEDUP "A LA NVIDIA WHITE PAPER"

OPTERON 2435: 1.2 GFLOPS (86% EFF.)

TESLA S1070: 30.1 GFLOPS (~100% EFF.)

25.8X

BEST PERFORMANCES:

OPTERON 2435: 7.8 GFLOPS (86% EFF.)

TESLA S1070: 30.1 GFLOPS (~100% EFF.)

3.9X

NVIDIA Fermi M2050 versus Magny-Cours
CPUtiled: 18 [GFLOP/s]
GPU: 32 [GFLOP/s]

1.8X

Th

```
void D
{
    co
    co
#pragm
    fo
    {
```

```
void Diffu
{
    const
#pragma om
    for(in
        fo
        {
```

```
}
```

```
}
```

```
Void DiffusionSSEtiledHilbert::_step1(const float * const input, float * const output, const int NX, const int NY, float factor)
{
    const float a0 = (1.f-4.f*factor);

    const int NT = NTX*NTY;

#pragma omp parallel
    {
        float __attribute__((aligned(16))) lab[_LABSIZEX_*_LABSIZEY_];
        assert(((size_t)&lab[0] & 0xf) == 0 );

#pragma omp  for
        for(int t = 0; t<NT; t++)
        {
            float * const mytile = output + t*_TILESIZE_*_TILESIZE_;

            LabLoader_H labloader(lab, input, t, NTX, NTY, level);

            HilbertIndexer2D indexer;

            HilbertIndexer2D::CartesianIndex index = indexer.decode(t, level);
            const int gtx = index.x;
            const int gty = index.y;

            {
                const int offset = _LEFTGHOSTS_ + _LABSIZEX_;
                const int ys = (int)(gty == 0);
                const int ye = _TILESIZE_ - (int)(gty == NTY-1);

                for(int iy=ys; iy<ye; iy++)
                {
                    const float * const lab_stripe = lab + iy*_LABSIZEX_ + offset;
                    float * const output_stripe = mytile + iy*_TILESIZE_;

                    const __m128 myfactor = _mm_set_ps1(factor);
                    const __m128 mya0 = _mm_set_ps1(a0);

#define LEFT    _mm_shuffle_ps(_mm_shuffle_ps(W, C, _MM_SHUFFLE(0,0,3,3)), \
_mm_shuffle_ps(W, C, _MM_SHUFFLE(2,1,3,3)),\
_MM_SHUFFLE(3,2,2,1))

#define RIGHT   _mm_shuffle_ps(_mm_shuffle_ps(C, E, _MM_SHUFFLE(0,0,2,1)),\
_mm_shuffle_ps(E, C, _MM_SHUFFLE(3,3,0,0)),\
```

```
texture<float, 2, cudaReadModeElementType> u_tex;

__global__ void kernel1(float * output, float a0, float a1, int NX, int NY)
{
    const int tx = 1+threadIdx.x + blockIdx.x*blockDim.x;
    const int ty = 1+threadIdx.y + blockIdx.y*blockDim.y;

    if(tx < NX-1 && ty < NY-1)
    {
        output[ty*NX + tx] = a0*tex2D(u_tex, tx, ty) +
        a1*(tex2D(u_tex, tx-1, ty) + tex2D(u_tex, tx+1, ty) +
            tex2D(u_tex, tx, ty-1) + tex2D(u_tex, tx, ty+1));
    }
}
```

```
                C = E;
            }

            //CENTRAL
            for(int ix=4; ix<_TILESIZE_-4; ix+=4)
            {
                //C,W already loaded!!!
                const __m128 S = _mm_load_ps(lab_stripe+ix-_LABSIZEX_);
                const __m128 E = _mm_load_ps(lab_stripe+ix+4);
```

```
actor)
```

```
oat factor)
```

```
*C);
```

Conclusions

- When is the roofline model useless?
  - x When you discuss performance in terms to time-to-solution.
- When is the roofline model crucial?
  - ➡ When you want to optimize your code (*data reuse, ceilings*)
  - ➡ To predict maximum achievable performance (*roofline, ridgepoint*)
  - ➡ To systematically assess your performance (*roofline, op. int.*)
- What do you do if all your kernels have a bad op. int.?
  - ~ Either live with it
  - ✓ Go back to equations, pick better discretizations/algorithms (leading to a higher op. int.)
  - ✓ Wanted: less simulation steps, but more costly (High order schemes)

Programming Models

PROGRAMMING MODELS - OVERVIEW

There are several parallel programming models in common use:

1. **Shared Memory (without threads)**
2. **Shared Memory (with threads)**
3. **Distributed Memory / Message Passing**
4. **Data Parallel**
5. **Hybrid**
6. **Single Program Multiple Data (SPMD)**
7. **Multiple Program Multiple Data (MPMD)**

Parallel programming models exist as an abstraction above hardware and memory architectures.

Although it might not seem apparent, these models are **NOT** specific to a particular type of machine or memory architecture. In fact, any of these models can (theoretically) be implemented on any underlying hardware.

- **Which model to use?** This is often a combination of what is available and personal choice. There is no "best" model, although there certainly are better implementations of some models over others.
- The following sections describe each of the models mentioned above, and also discuss some of their actual implementations.

153

SHARED MEMORY MULTIPROCESSOR **(SMP)**

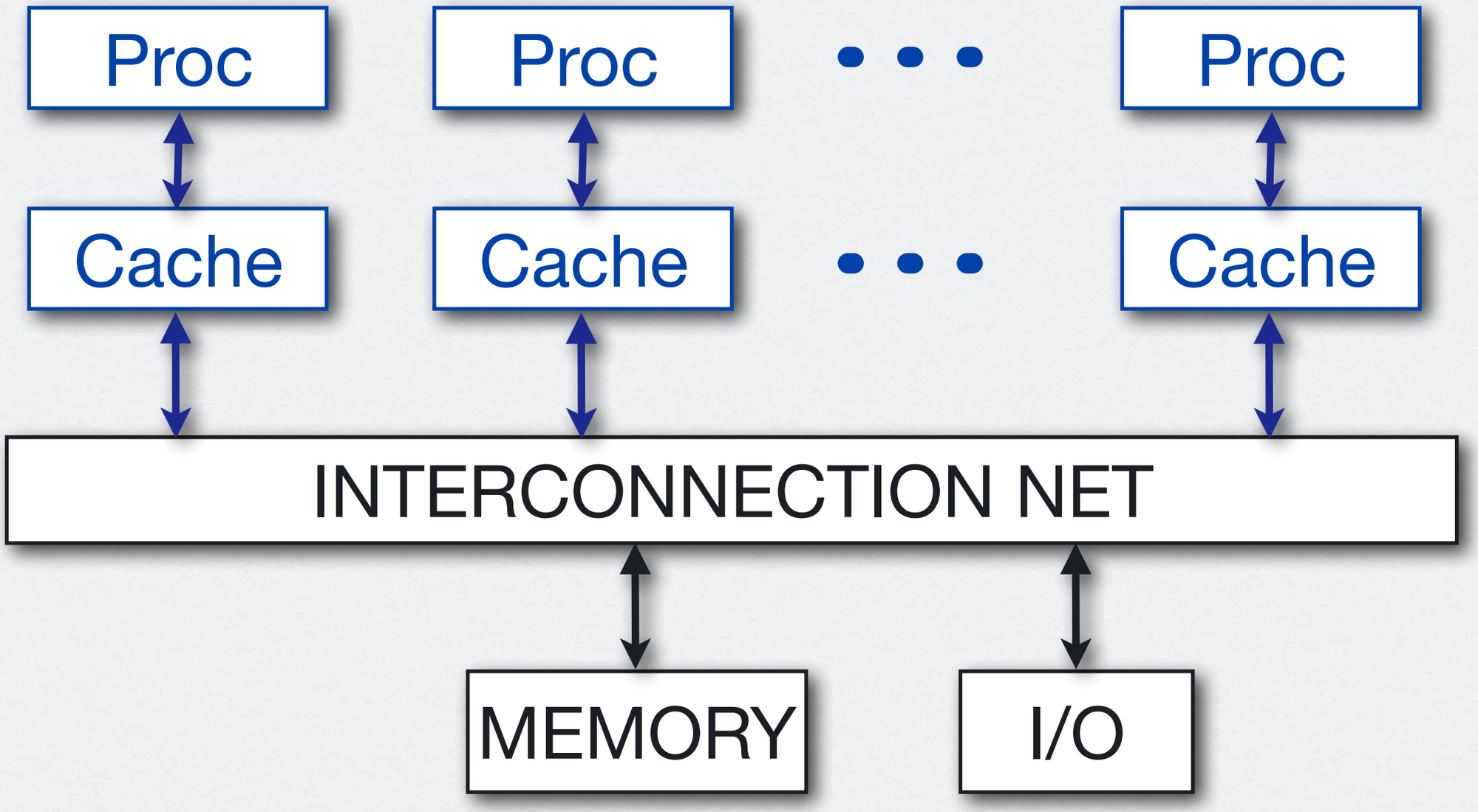
- **Simplest way to parallelizing:**
  - Provide a single physical address space that all processors can share (no worries about where one runs, just execute)
    - All program variables can be available at any time to any processor.
    - Hardware can provide cache coherence to give memory a consistent view.

- Alternative

➡ Separate address space per processor; sharing must be explicit.

SHARED MEMORY MULTIPROCESSOR **(SMP)**

- Parallel processor with a single address space across all processors, implying implicit communication with loads and stores



SHARED MEMORY MULTIPROCESSOR **(SMP)**

- Shared memory parallel computers vary widely, but generally have in common the ability for **all processors to access all memory as global address space.**

- Multiple processors can operate independently but share the **same memory** resources.

- Changes in a memory location effected by one processor are **visible to all** other processors.

- Shared memory machines can be divided into two main classes based upon memory access times: ***UMA*** and ***NUMA***.

SHARED MEMORY MULTIPROCESSOR **(SMP)**

- **UMA (uniform memory access): same time to access memory** no matter which processor requests it and which word it is requested.

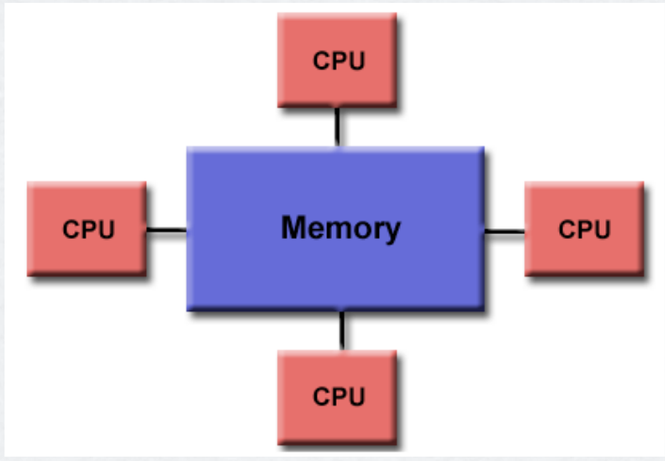
- UMA is not scalable

- **NUMA:** some memory accesses are **much faster than others**, depending on which processor asks for which word.

- Harder to program NUMA, but can be larger and have lower latency to nearby memory.

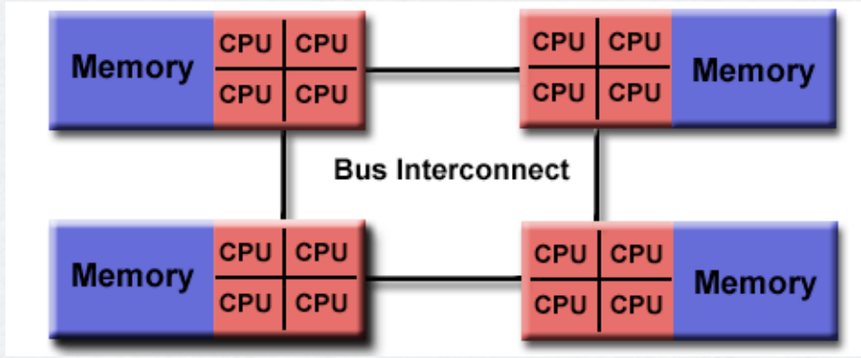
UMA vs NUMA

Uniform Memory Access (UMA):



- Most commonly represented today by Symmetric Multiprocessor (SMP) machines
- Identical processors
- Equal access and access times to memory
- Sometimes called CC-UMA - **Cache Coherent UMA.** *Cache coherent* means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

Non-Uniform Memory Access (NUMA):



- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower
- If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

UMA **Advantages:**

- Global address space provides a user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

UMA **Disadvantages:**

- Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
- Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

SMP: Synchronization

- As processors share memory/data, they also need to cooperate when operating on **shared data**.

- **To avoid conflicts:**
  - **Synchronization**: The process of coordinating the behavior of two or more processes, which may be running on different processors.
  - **Lock:** A synchronization device that allows access to data to only one processor at a time.

SMP: Example

- Sum 100000 number on a shared memory multiprocessor with UMA time. Assume 100 processors

- **Step 1**: SPLIT SET OF NUMBERS INTO SUBSETS OF EQUAL SIZE
  - No different memory space, just a different starting address.

```
Pn = number for each processor, Pn ϵ [0, 99]

sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i=i+1)
   sum[Pn] = sum[Pn] + A[i];/* sum areas assigned */
```

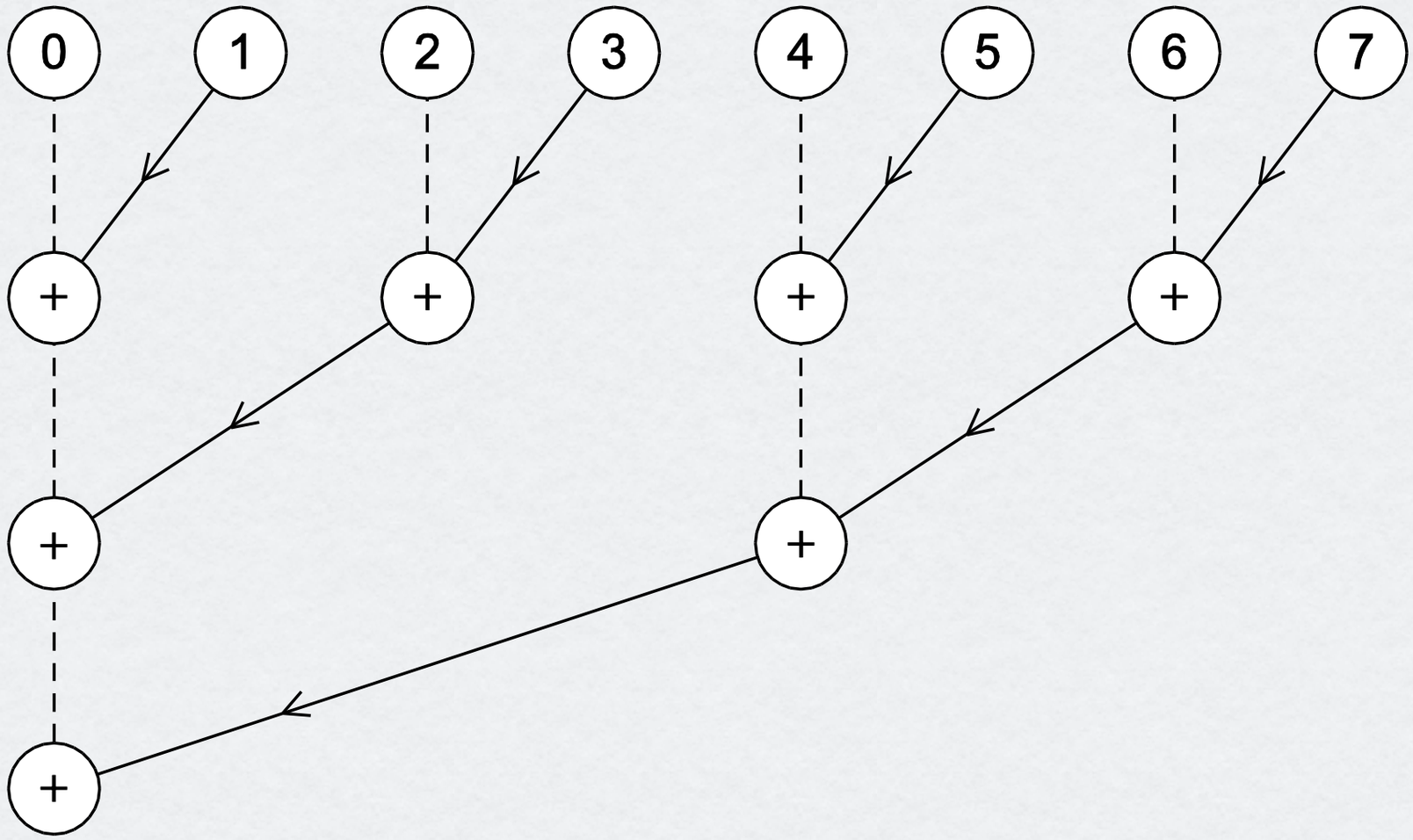
SMP: Example

```
half = 100; /* 100 processors in multiprocessor */
repeat
      synch(); /* wait for partial sum completion */
      if (half%2 != 0 && Pn == 0)
          sum[0] = sum[0] + sum[half-1]; /* Conditional
          sum needed when half is odd; Processor 0 gets
          missing element */
        half = half/2; /* dividing line of who sums */
        if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1); /* exit with final sum in sum[0] */
```

SMP: Example

- **Step 2**: ADD THESE MANY PARTIAL SUMS → **REDUCTION**

- **DIVIDE & CONQUER:** Half of the processors add pairs of partial sums, and then a quarter add pairs of the new partial sums and so on until we have the single, final sum



SMP: Example

- In the above example, the two processors must **synchronize** before the "consumer" processor tries to read the results from the memory location written by the "producer" processor; otherwise the consumer may read the old value of the data.

- Each processor must have its own version of the counter (variable **`i`**), so we must indicate it as "private" (**`half`** is also "private").

SHARED MEMORY - NO THREADS

- Tasks share a **common address space**, which they **read and write to asynchronously.**
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.

- An **advantage** of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. Program development can often be simplified.
- An important **disadvantage** in terms of performance is that it becomes more difficult to understand and manage data locality.
  - Keeping data local to the processor that works on it conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processors use the same data.
  - Unfortunately, controlling data locality is hard to understand and beyond the control of the average user.

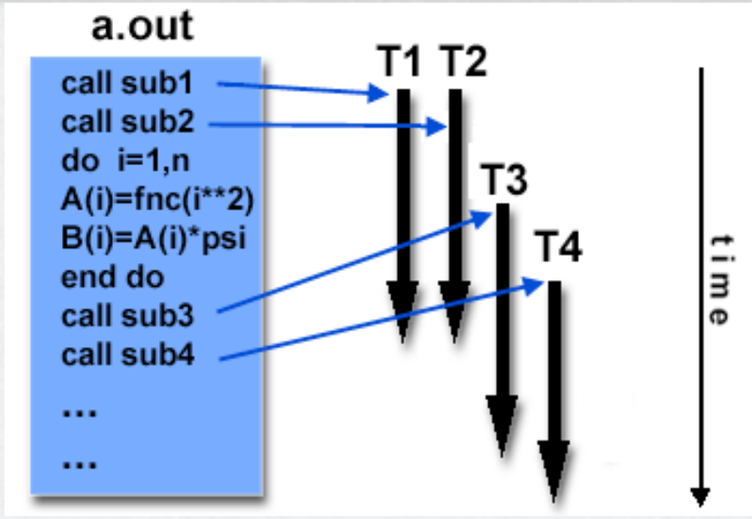
Implementations:

- Native compilers and/or hardware translate user program variables into actual memory addresses, which are global. On stand-alone SMP machines, this is straightforward.
- On distributed shared memory machines, such as the SGI Origin, memory is physically distributed across a network of machines, but made global through specialized hardware and software.

SHARED MEMORY - with threads

★In the threads model of parallel programming, a single process can have multiple, concurrent execution paths.

★**SIMPLE ANALOGY** to describe threads: concept of a single program that includes a number of subroutines:



- The main program **a.out** is scheduled to run by the native operating system. a.out loads and acquires all of the necessary system and user resources to run.
- a.out performs some serial work, and then creates *a number of tasks (threads)* that can be scheduled and run by the operating system concurrently.

  - ➡ Each thread has local data, but also, shares the entire resources of a.out. This saves the overhead associated with replicating a program's resources for each thread. Each thread also benefits from a global memory view because it shares the memory space of a.out.
  - ➡ A thread's work may best be described as a subroutine within the main program. Any thread can execute any subroutine at the same time as other threads.
  - ➡ Threads communicate with each other through global memory (updating address locations). This **requires synchronization constructs** to ensure that more than one thread is not updating the same global address at any time.
  - ➡ Threads can come and go, but a.out remains present to provide the necessary shared resources until the application has completed.

SHARED MEMORY - with threads

Implementations:

- From a programming perspective, threads implementations commonly comprise:
  - A library of subroutines that are called from within parallel source code
  - A set of compiler directives imbedded in either serial or parallel source code
- In both cases, the **programmer is responsible for determining all parallelism**.

- Threaded implementations are not new in computing. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP***.

OpenMP

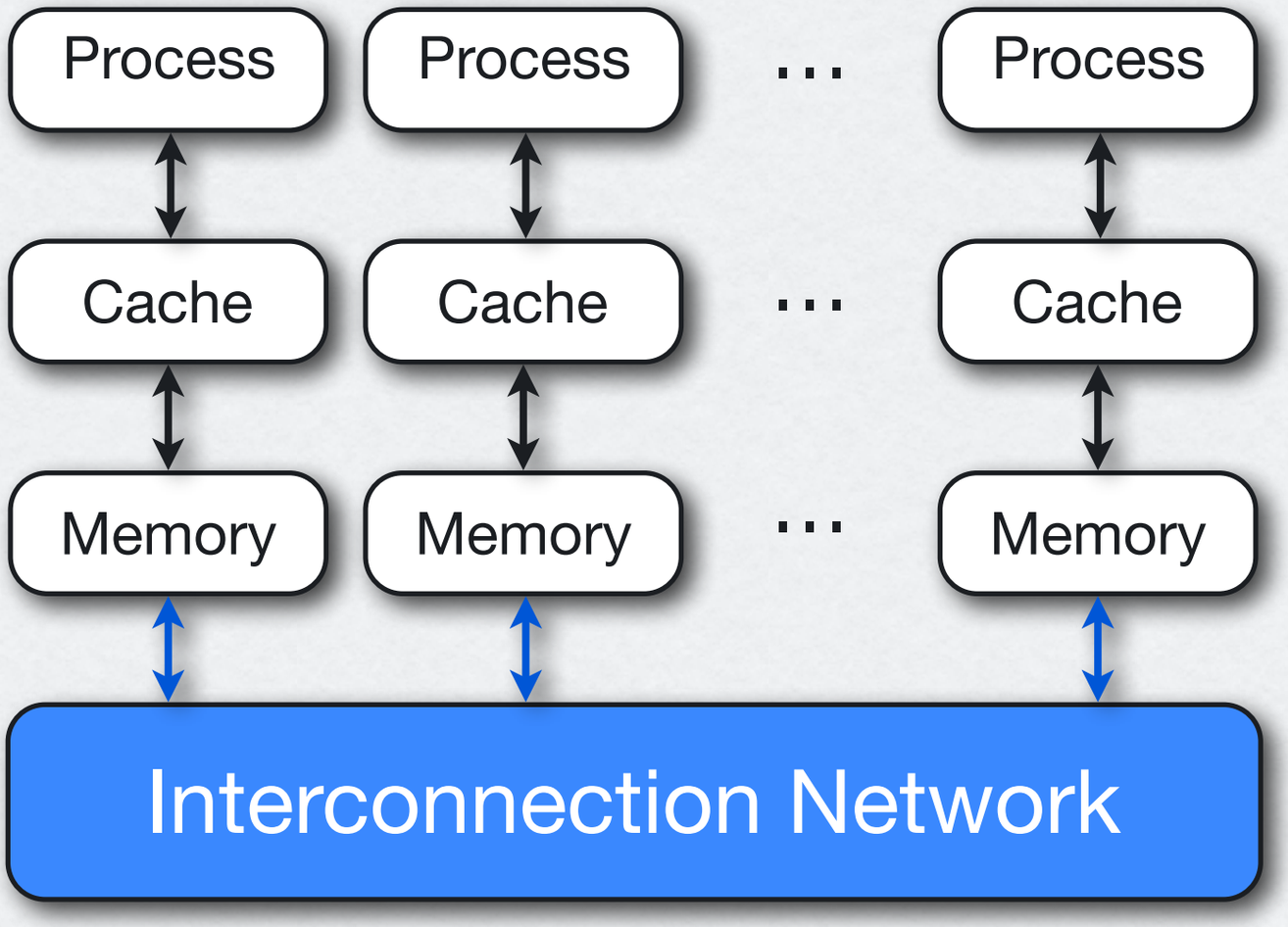
- Compiler directive based; can use serial code
- Jointly defined and endorsed by a group of major computer hardware and software vendors. The OpenMP Fortran API was released October 28, 1997. The C/C++ API was released in late 1998.
- Portable / multi-platform, including Unix, Windows, Mac platforms
- Available in C/C++ and Fortran implementations
- Can be very easy and simple to use - provides for "incremental parallelism"

OpenMP tutorial: computing.llnl.gov/tutorials/openMP

166

CLUSTERS/MESSAGE PASSING MULTIPROCESSORS

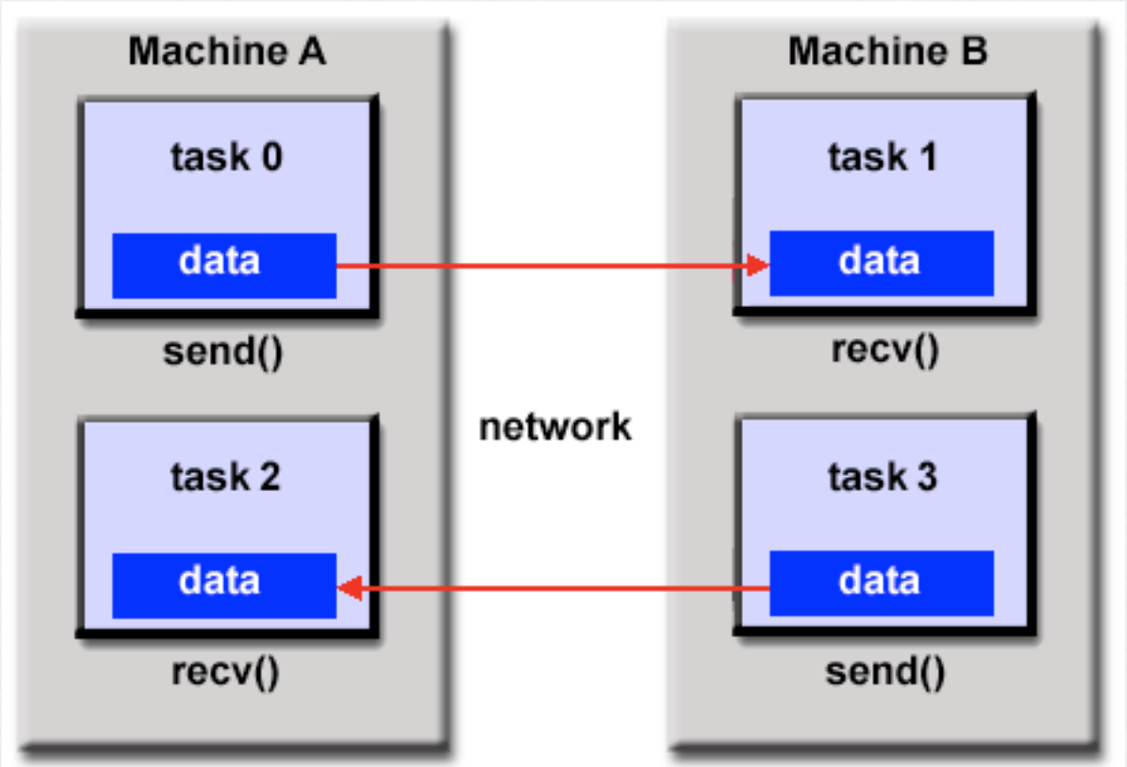
The alternative approach to sharing an address space is for the processors themselves to have their private address space



CLUSTERS/MESSAGE PASSING MULTIPROCESSORS

Message Passing characteristics:

- A set of tasks that use their own local memory during computation**. Multiple tasks** can reside on the same physical machine and/or across an arbitrary number of machines.
- Tasks exchange data through communications by **sending and receiving messages**.
- Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.



CLUSTERS/MESSAGE PASSING MULTIPROCESSORS

IMPLEMENTATION:

- From a programming perspective, message passing implementations usually comprise a library of subroutines. Calls to these subroutines are imbedded in source code. **The programmer is responsible for determining all parallelism.**

- **MPI**
- Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.
- Part 1 of the **Message Passing Interface (MPI)** was released in 1994. Part 2 (MPI-2) was released in 1996. Both MPI specifications are available on the web at http://www-unix.mcs.anl.gov/mpi/.
- MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. MPI implementations exist for virtually all popular parallel computing platforms. Not all implementations include everything in both MPI1 and MPI2.
- More Information:
  - **MPI tutorial:** computing.llnl.gov/tutorials/mpi

CLUSTERS/MESSAGE PASSING MULTIPROCESSORS

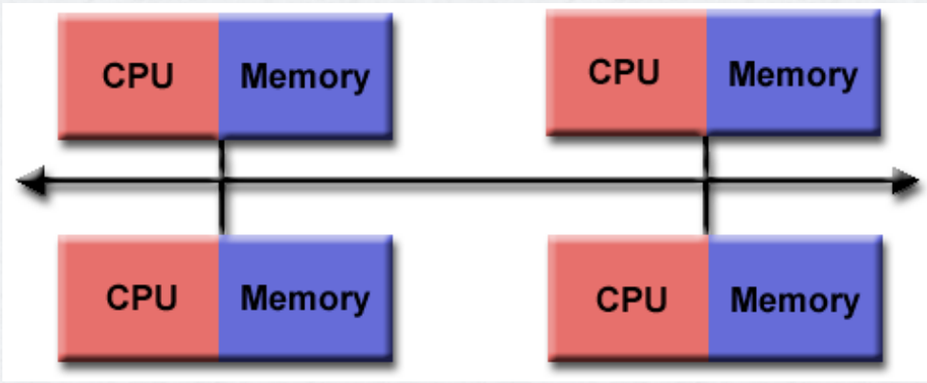
- Highly suitable for job-level parallelism
- High performance message passing networks are too expensive and too specialized
- Clusters: Commodity computers tat are connected to each other over their I/O interconnect via standard network switches and cables. Each runs a distinct copy of the operating system

Drawback with multiprocessors: Processors are connected using the I/O interconnect of each computer, whereas the cores in a multiprocessor are on the memory interconnect of the computer. The memory interconnect has higher bandwidth and lower latency, allowing for much better communication performance

CLUSTERS/MESSAGE PASSING MULTIPROCESSORS

Distributed memory systems also vary widely but share a common characteristic :
they require a communication network to connect inter-processor memory.

- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the **concept of cache coherency does not apply**.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. **Synchronization** between tasks is likewise the **programmer's responsibility.**
- The network "fabric" used for data transfer varies widely, though it can can be as simple as Ethernet.



Advantages:

- Memory is scalable with the number of processors. Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.

Disadvantages:

- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- Non-uniform memory access (NUMA) times

Example of parallel program for message passing

Suppose we want to sum 100,000 numbers in a message passing multiprocessor with 100 processor each with private memory

Step 0: Processor with 100,000 numbers sends the subsets to each of the 100 processor-memory nodes

Step 1: Sum for each subprocessor (local variable):

```
float sum = 0;
for (int i = 0; i < 1000; i = i + 1) // Loop over each array element
sum = sum + arr[i];                  // Sum local arrays
```

Step 3: Reduction: add the 100 partial sums

sequential → One way is to send all numbers to the main processor and add.

Better way: Divide and Conquer:

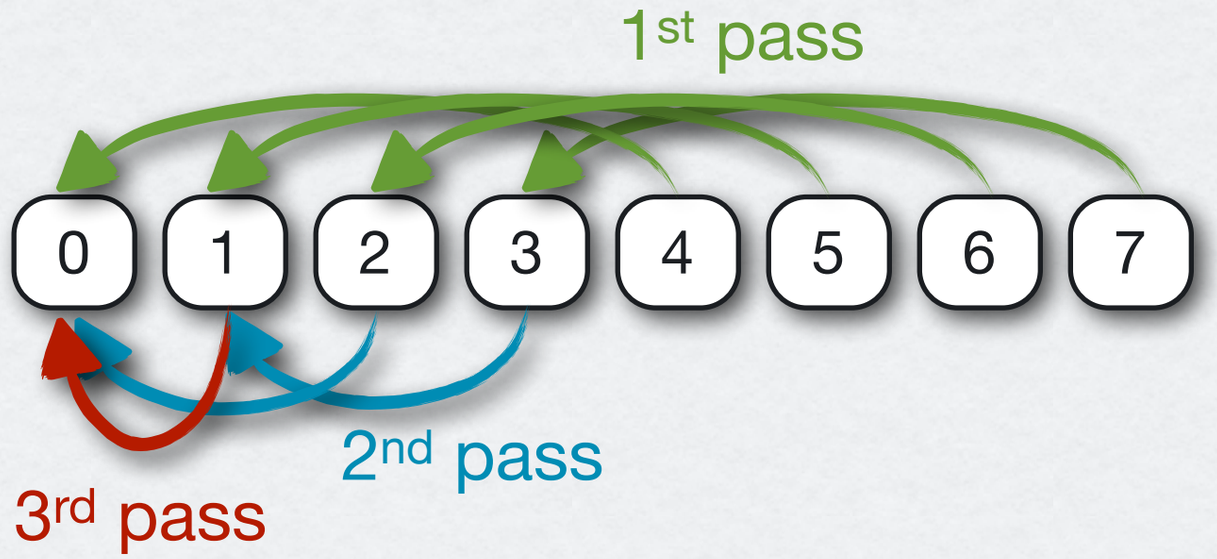
parallel →

- 1/2 of the processors send their sum to the other 1/2 one sum is performed on each of them
- Then 1/4 of the processors send their new sum to the other 1/4
- .....
- Last sum is performed on the main processor

Divide and conquer

Let:

- **Pn** be the number (id) of the processor
- **send(x, y)** – a routine to send value y to the processor x over the network
- **receive()** – a function that accepts a value for this network from this processor




```
int limit = 100;    // 100 processors
int half  = 100;
while (half != 1)   // exit with final sum
{
   half = (half + 1) / 2;          // send vs receive dividing limits

   if (Pn >= half && Pn < limit) // send if processor is in the second half
    send(Pn - half, sum);
   if (Pn < (limit/2))           // receive if processor is in the first half
    sum = sum + receive();

   limit = half;    // upper limit of the senders
}
```

Divide and conquer

- The code divides processors into senders and receivers and each receiving processor gets only one message

- So we can presume that a receiving processor will stall until it receives a message

- `send()` and `receive()` can be used as primitive for synchronization as well as for communication, as the processors are aware of the transmission data

Message Passing: Big picture

- Message passing hardware is easier to build but more difficult to program

- Cache-coherent hardware makes the hardware figure out what data needs to be communicated

- Clusters are easier to expand

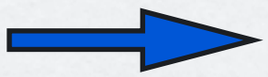
HARDWARE MULTITHREADING

Hardware Multi-threading allows multiple threads to share the functional units of a single processor in a overlapping fashion. To permit this sharing, the processor must duplicate the independent state of each thread:

- e.g. each thread would have a separate copy of the register file and the PC

- memory can be shared through virtual memory mechanisms

- ***thread*** **switches are 10^5 processor cycles faster than a** ***process*** **switch**



threading = light weight process

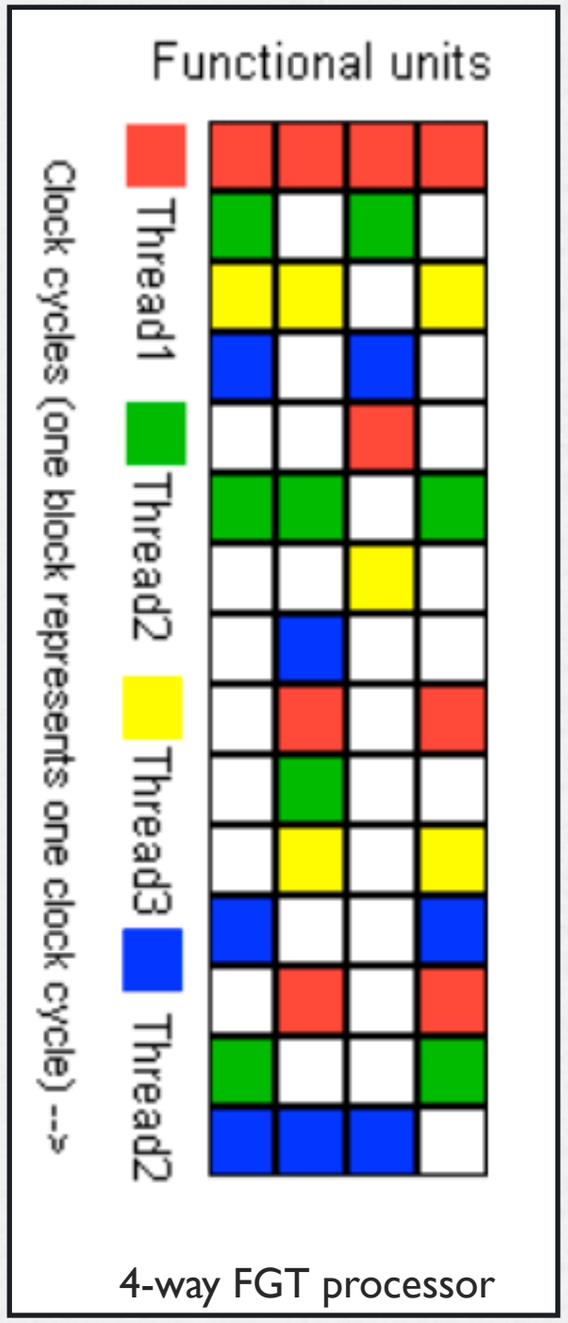
Two approaches to hardware multi-threading:

1) Fine grained multi-threading

2) Coarse-grained multi-threading

1) Fine grained multi-threading (FGT)

- **Switches between threads on each instruction**, resulting in interleaved execution of multiple threads
- Interleaving is done in round-robin fashion *skipping any threads that are stalled at any time*
- Processor must be able to switch threads at *every* cycle



Key advantage

Hardware can hide the throughput losses that arise from both short and large stalls since instructions from other threads can be executed when one thread stalls

Key disadvantage

It slows down execution of individual threads, since a thread that is ready to execute without stalls will be delay by instructions from other threads

Fig. source: http://www.slcentral.com/articles/01/6/multithreading/

2) Coarse grained multi-threading (CGM)

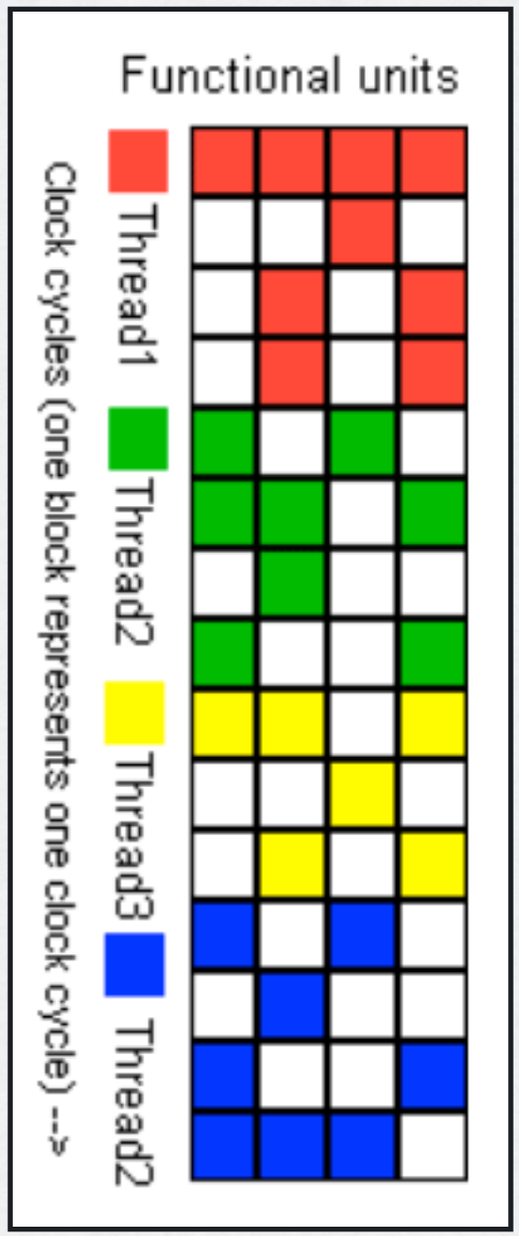
- **Switches threads only on costly stalls,** such as second-level cache misses.

Key advantage

Efficient as other threads will only be issued when a thread encounters a costly stall.

Key disadvantage

- limited in its ability to overcome throughput losses, especially from shorter stalls
- due to the pipe-line short up costs of CGM
- because a processor with CGM issues instruction from a single thread, when a stall occurs, the pipeline must be emptied or frozen
- the new thread must fill the pipeline before instructions will be able to complete



→ CGM is more effective for reducing the penalty of high-cost stalls, where pipeline refill is negligible compared to the stall time

Fig. source: http://www.slcentral.com/articles/01/6/multithreading/

Simultaneous Multithreading (SMT)

- **Variance of hardware multithreading**
- uses resources of a multiple-issue, dynamically scheduled processor to exploit, thread-level parallelism at the same time, it exploits instruction-level parallelism

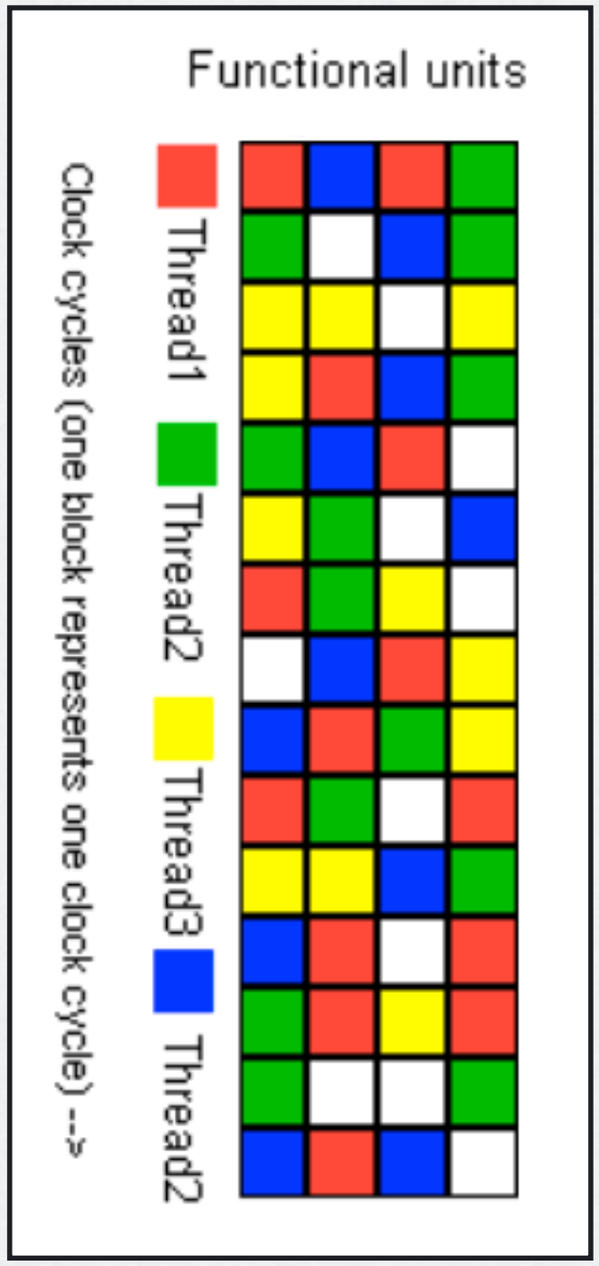


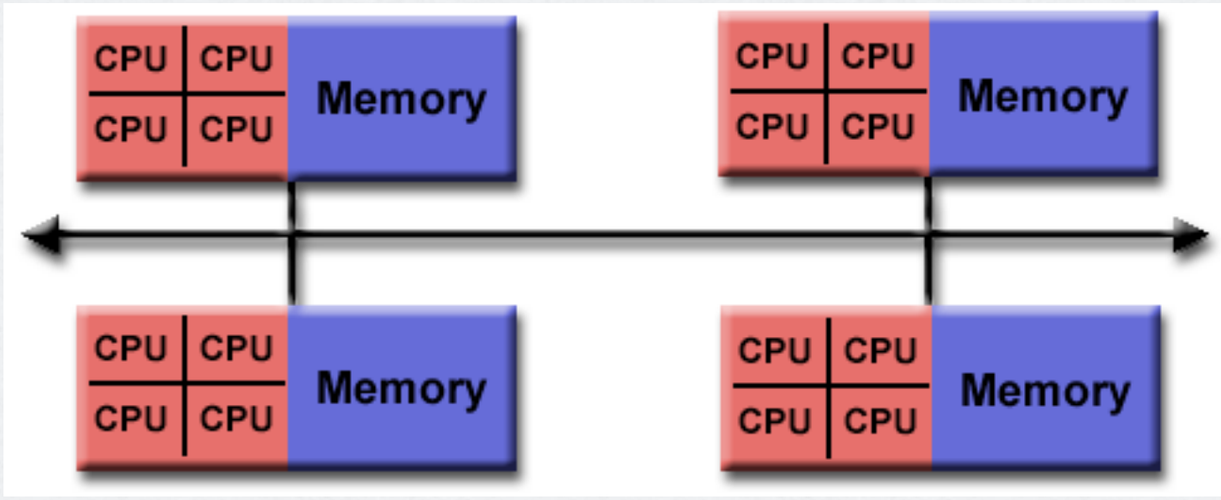
Fig. source: http://www.slcentral.com/articles/01/6/multithreading/

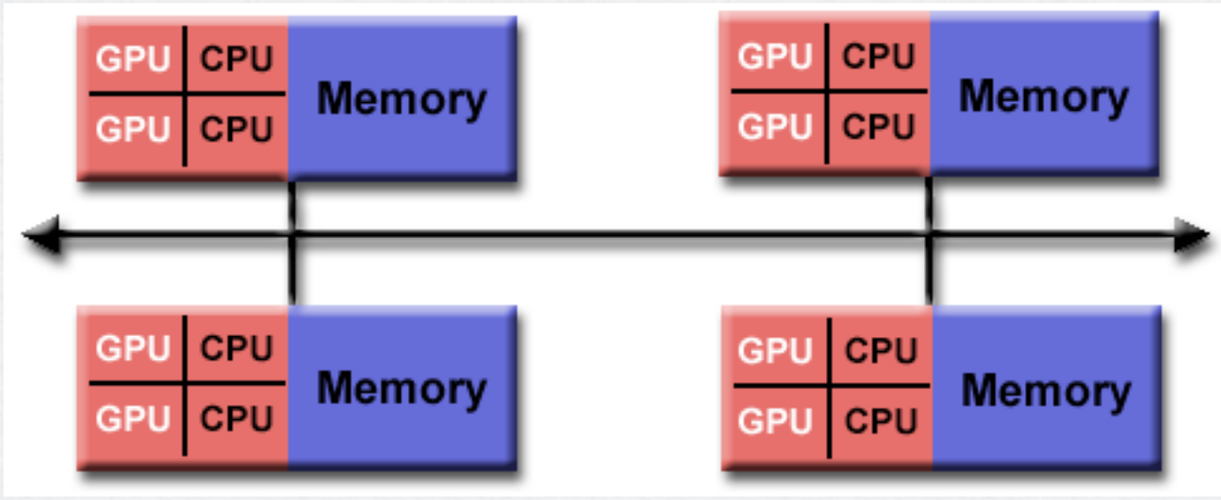
HYBRID COMPUTER ARCHITECTURES

The largest and fastest computers in the world today employ both shared and distributed memory architectures.

- The shared memory component can be a cache coherent SMP machine and/or graphics processing units (GPU).
- The distributed memory component is the networking of multiple SMP/GPU machines, which know only about their own memory - not the memory on another machine. Network communications required to move data between SMP/GPUs

- Current trends seem to indicate that this type of memory architecture ***will continue to prevail and increase*** at the high end of computing for the foreseeable future.





- Advantages and Disadvantages: whatever is common to both shared and distributed memory architectures.