CS-251: Parallel and Distributed Computing

Lecture 01 – An Overview of Parallel Computing

Dr. Zeeshan Rafi

PhD MIS, MPhil IT, BS Software Engineering
Former Software Engineer, Database Administrator
System Analyst, Big Data Analyst
Member Turkish Scientific & Technological Research Council

Department of Computing and Information Technology

Istanbul University, TR

KHAS University, TR

University of Gujrat, PK

GCF University, Pk

An Overview of Parallel Computing

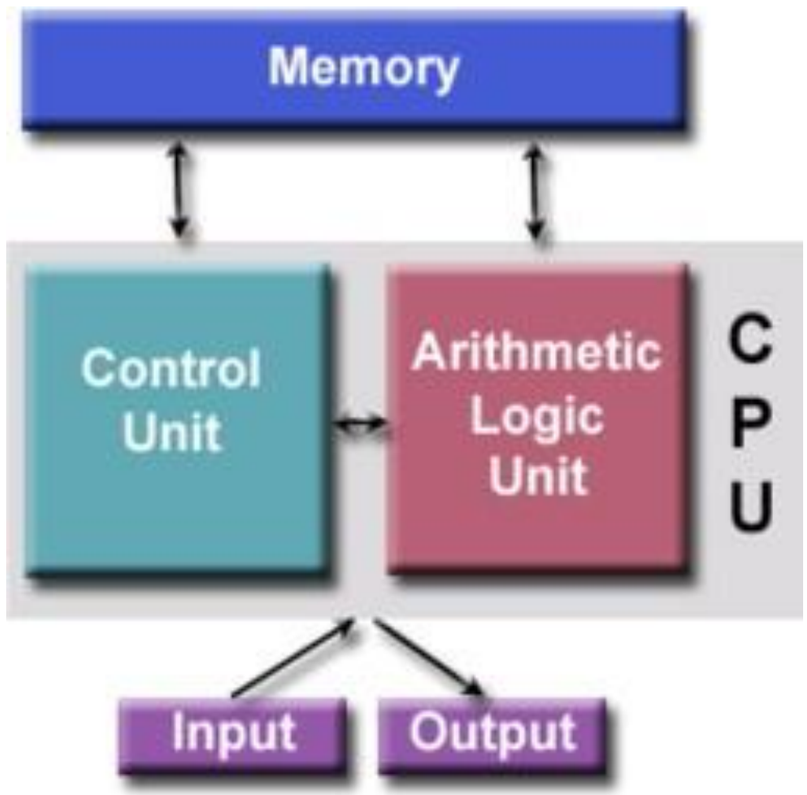
Study Plan

1. Hardware

2. Types of Parallelism

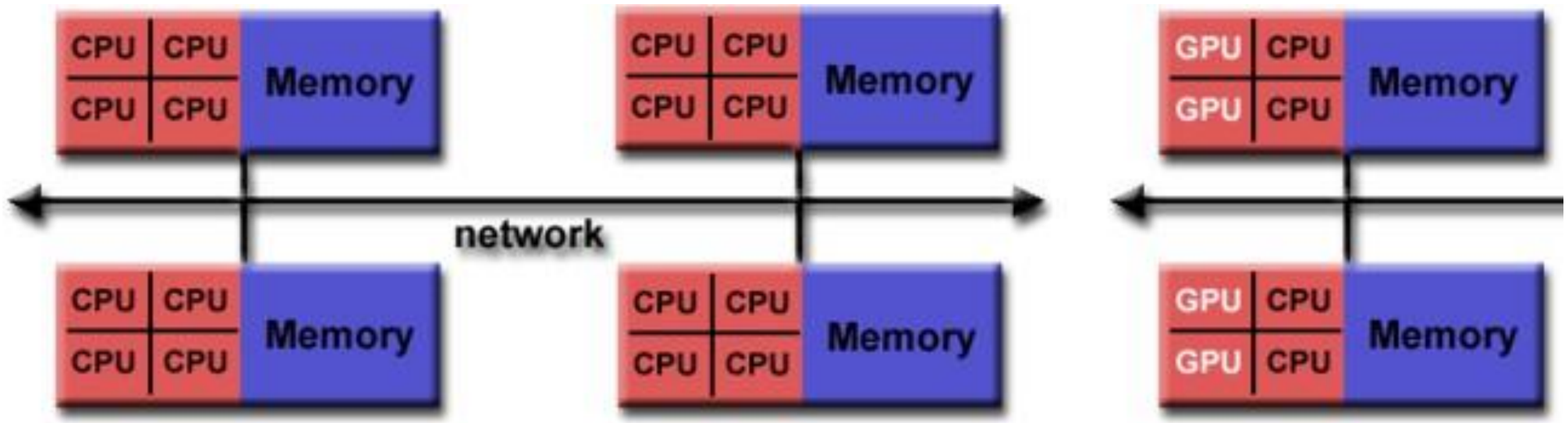
Von Neumann Architecture

- In 1945, the Hungarian mathematician John von Neumann proposed the above organization for hardware computers.
- The Control Unit fetches instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task.
- The Arithmetic Unit performs basic arithmetic operation, while Input/Output is the interface to the human operator.



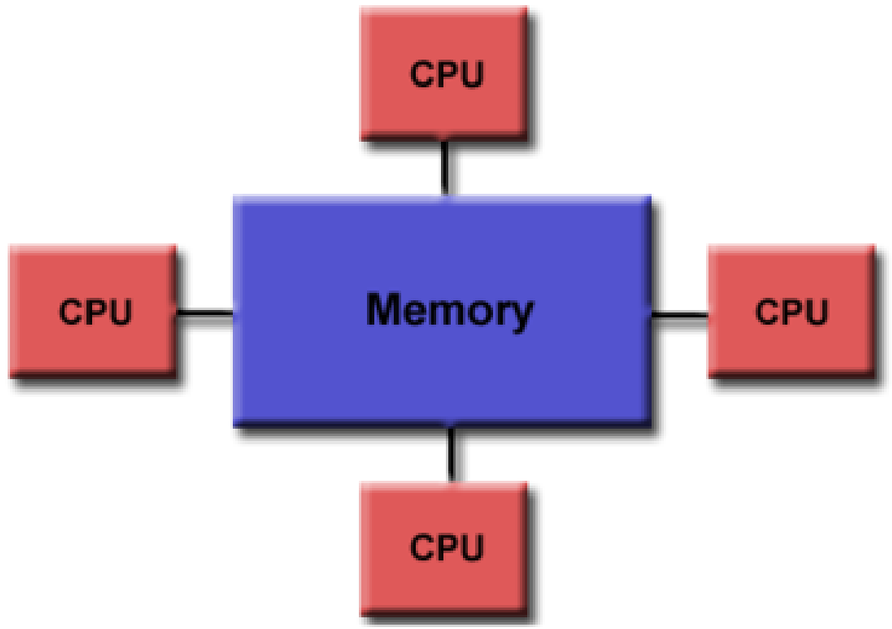
The Pentium Family

Parallel computer hardware



- Most computers today (including tablets, smartphones, etc.) are equipped with several processing units (control+arithmetic units).
- Various characteristics determine the types of computations: shared memory vs distributed memory, single-core processors vs multicore processors, data-centric parallelism vs task-centric parallelism.
- Historically, shared memory machines have been classified as UMA and NUMA, based upon memory access times.

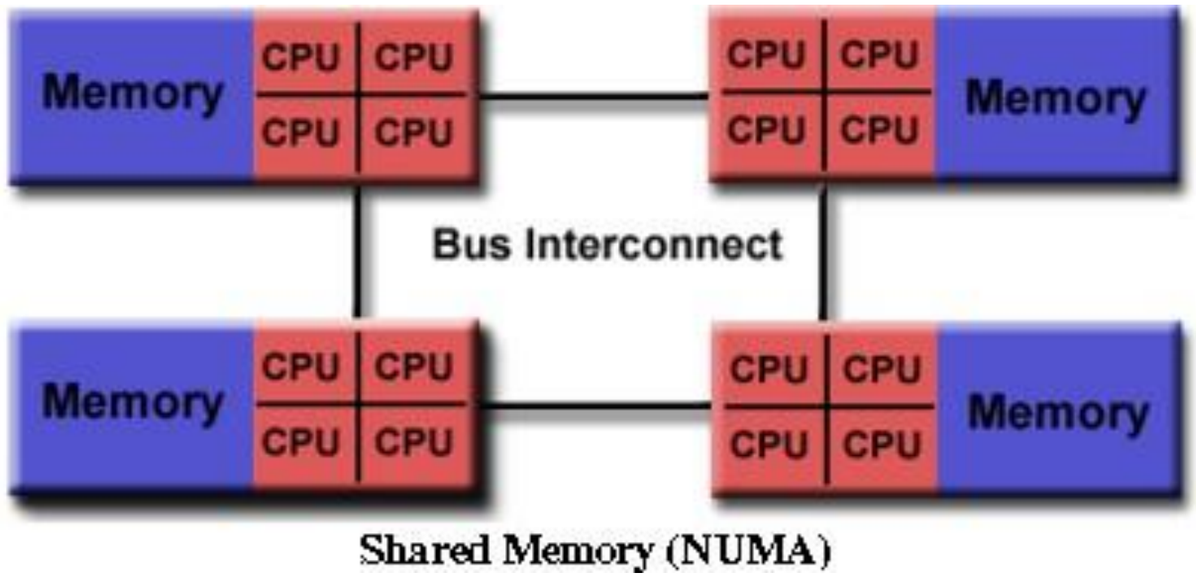
Uniform memory access (UMA)



Shared Memory (UMA)

- Identical processors, equal access and access times to memory.
- In the presence of cache memories, cache coherency is accomplished at the hardware level: if one processor updates a location in shared memory, then all the other processors know about the update.
- UMA architectures were first represented by Symmetric Multiprocessor (SMP) machines.
- Multicore processors follow the same architecture and, in addition, integrate the cores onto a single circuit die.

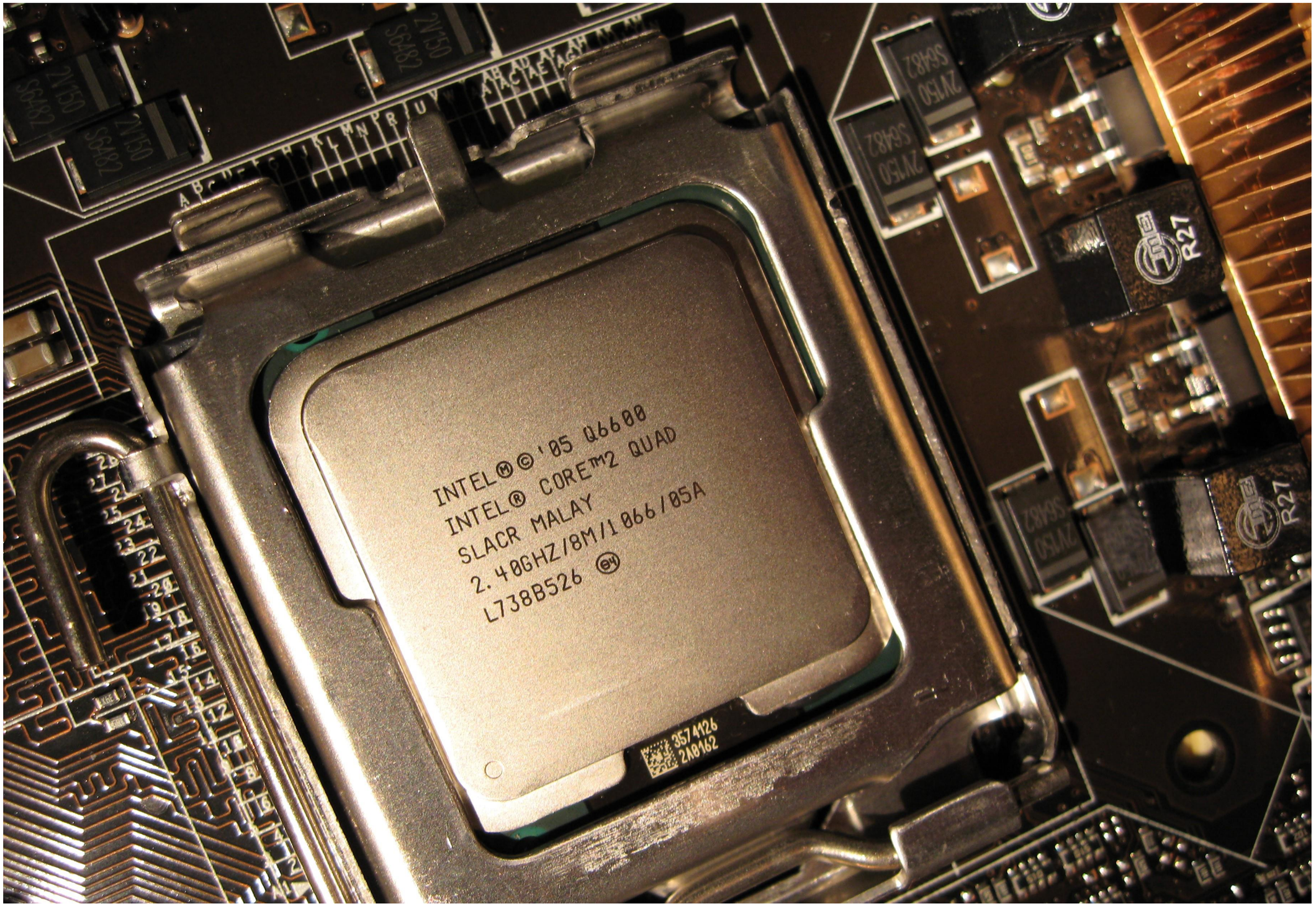
Non-uniform memory access (NUMA)



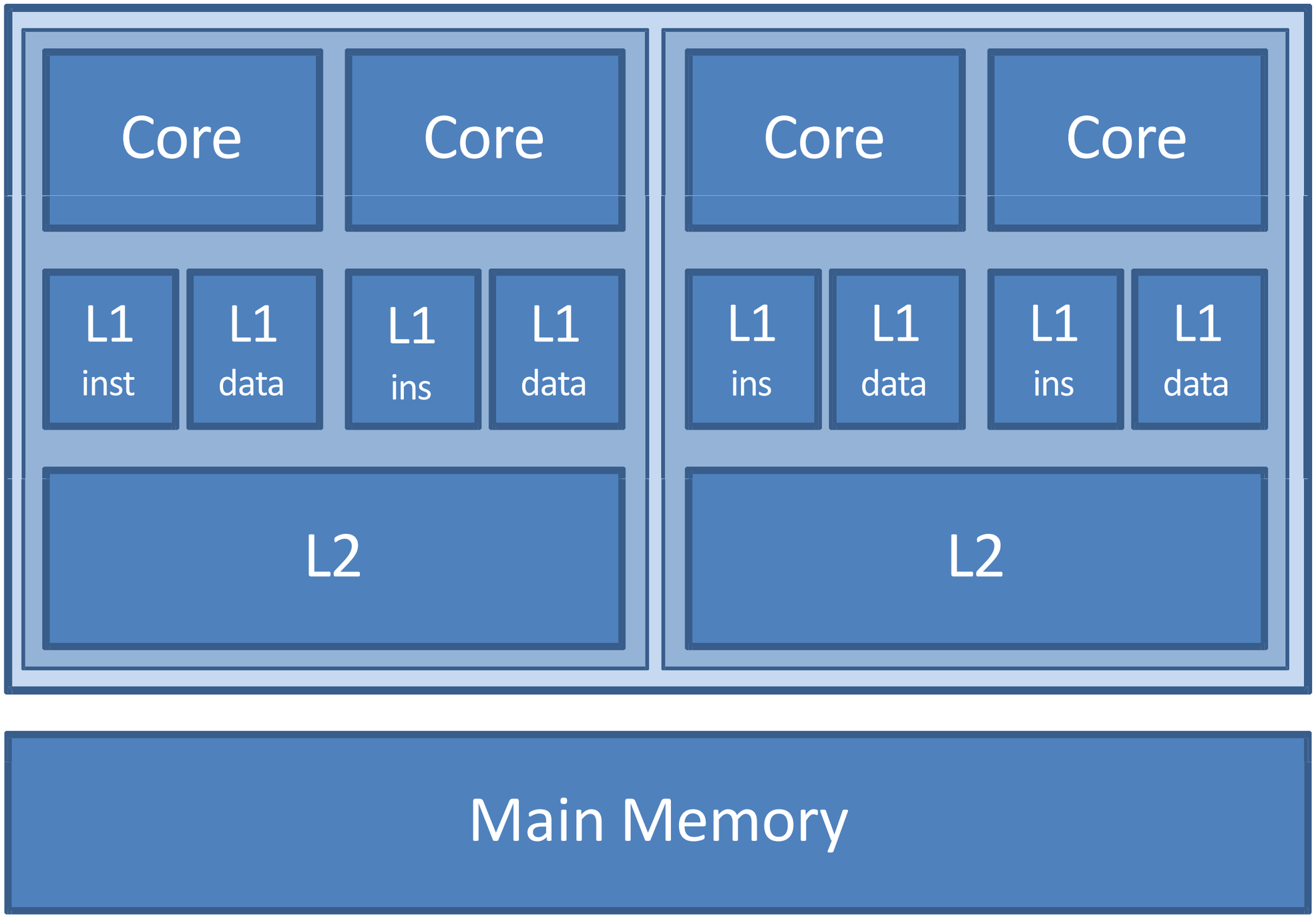
Shared Memory (NUMA)

- Often made by physically linking two or more SMPs (or multicore processors).
- Global address space provides a user-friendly programming perspective to memory, that is, it feels like there is a single large memory where all data reside.
- However, not all processors have equal access time to all memories, since memory access across link is slower.
- In fact, memory contention (that is, traffic jam) often limits the ability to scale of these architectures.

Multicore processors

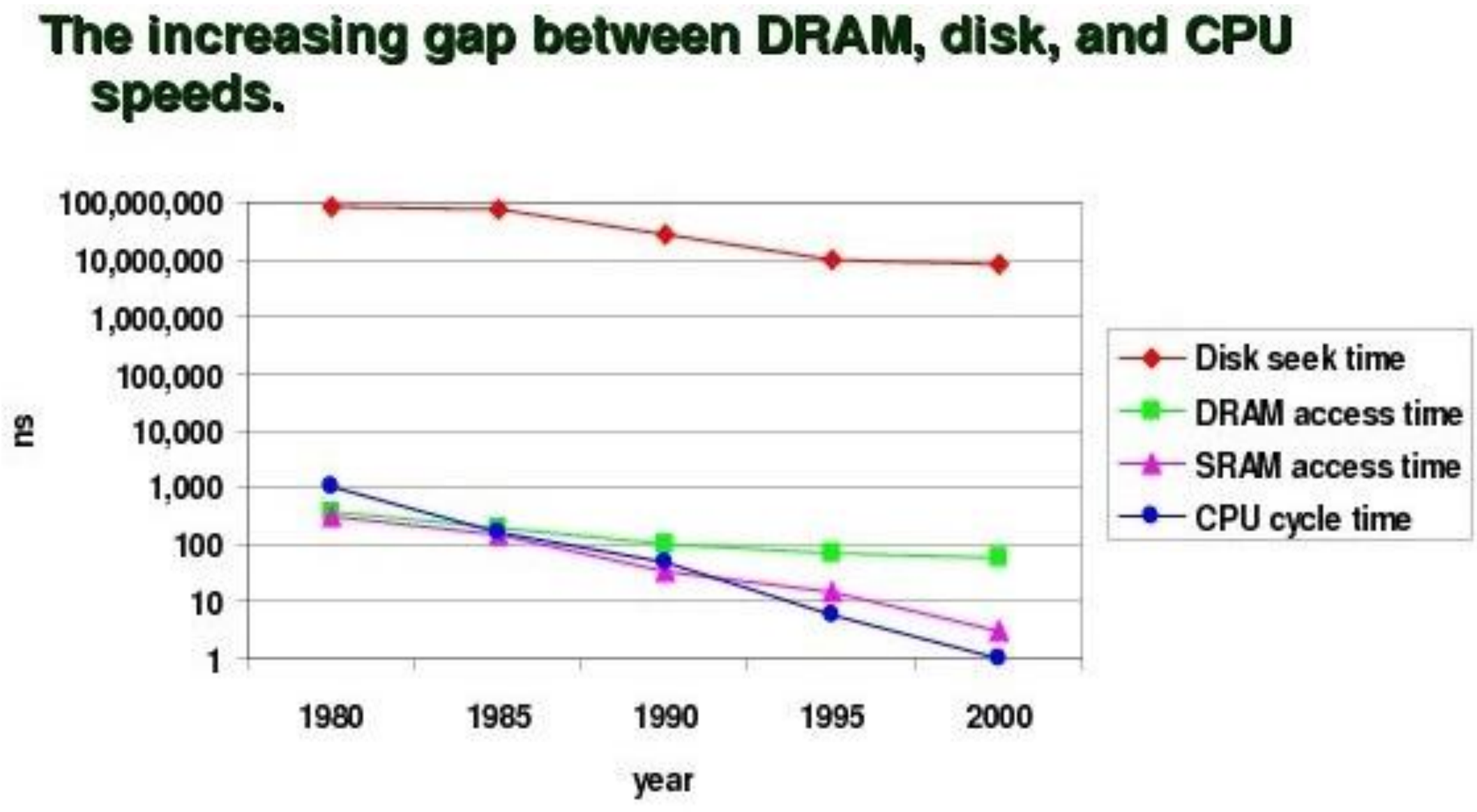


Multicore processors

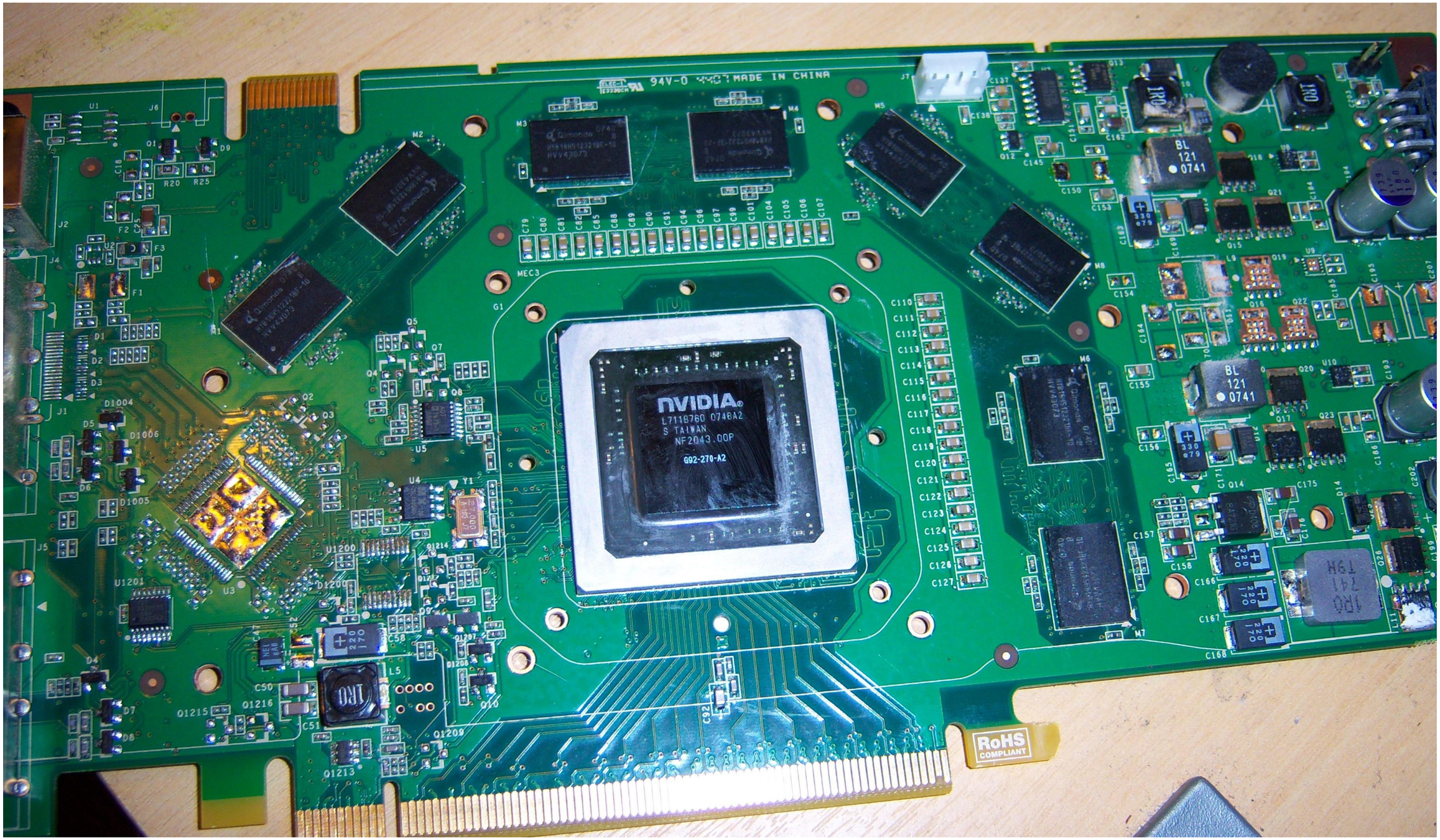


The CPU- Memory GAP

Once uopn a time, every thing was slow in a computer …

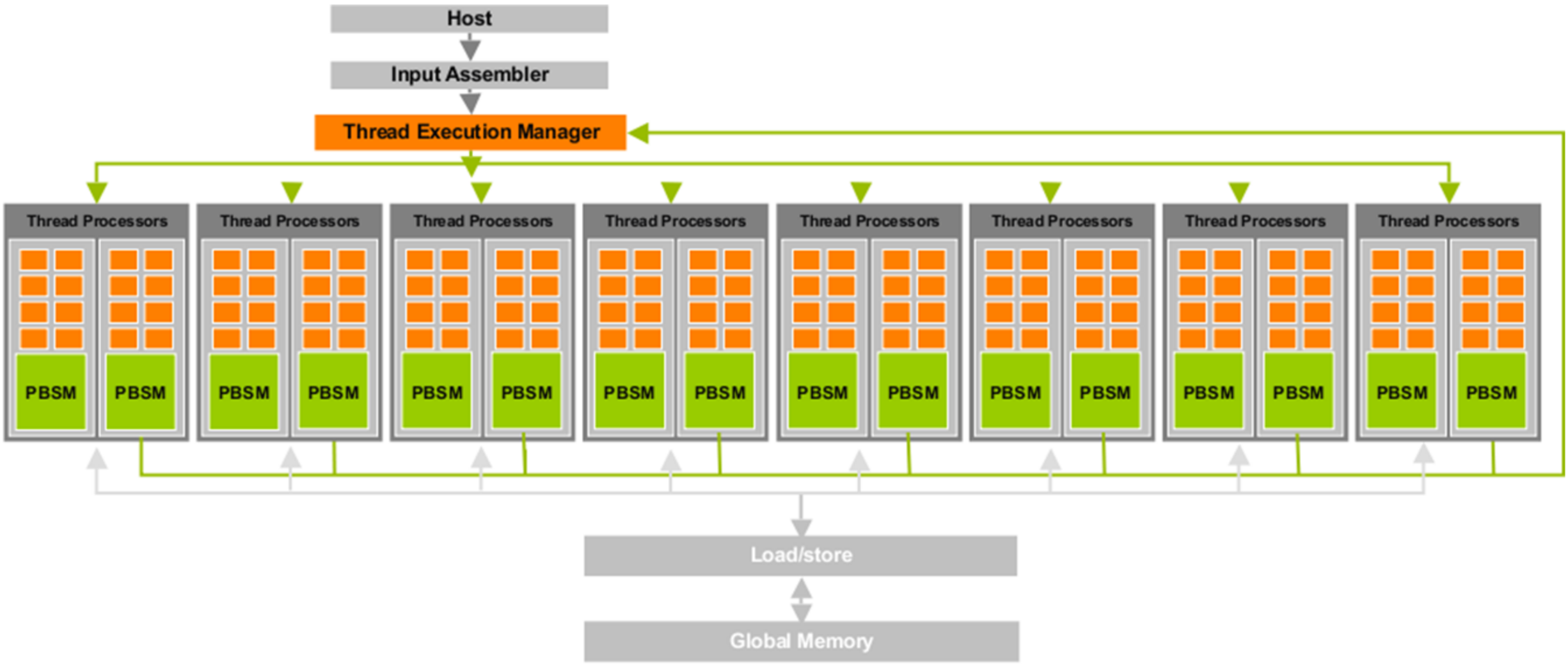


Graphics processing units (GPUs)

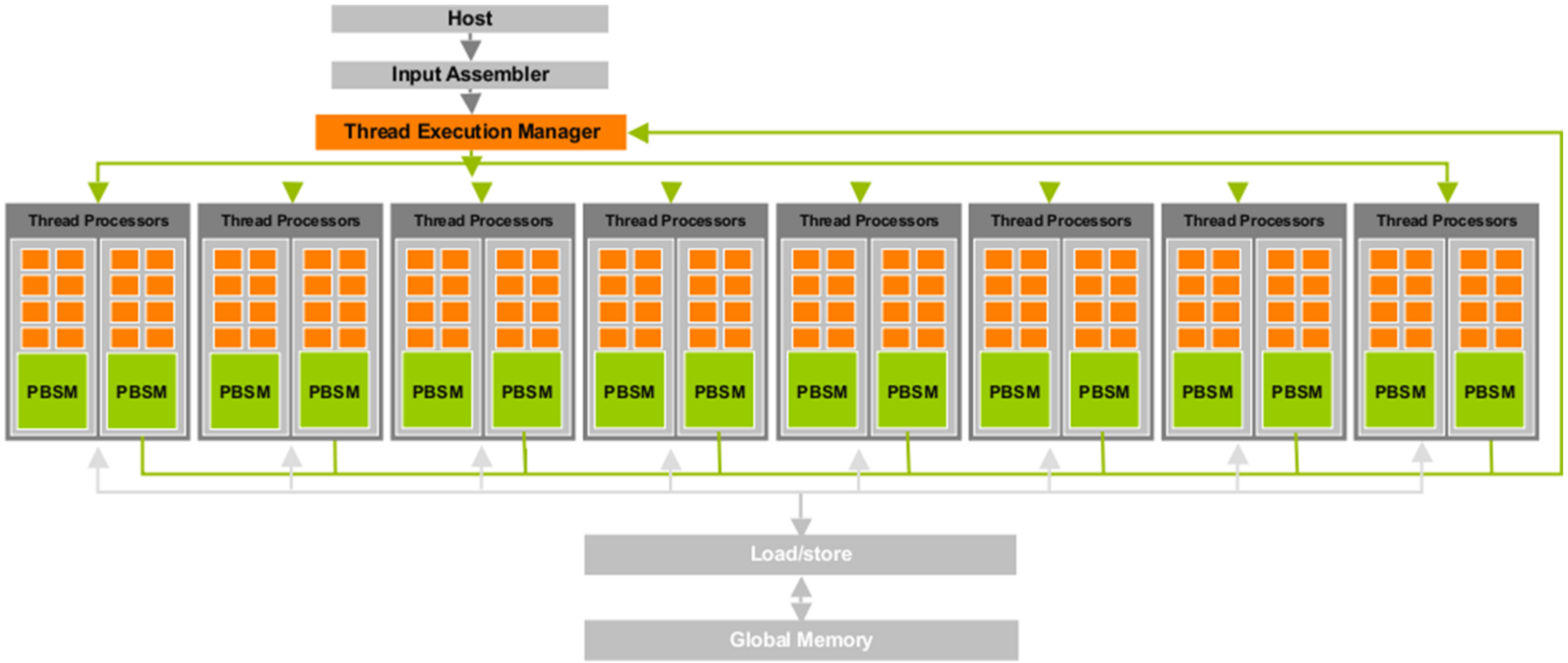


Graphics Processing Units (GPUs)

- A GPU consists of several streaming multiprocessors (SMs) with a large shared memory. In addition, each SM has a local (and private) and small memory. Thus, GPUs cannot be classified as UMA or NUMA.

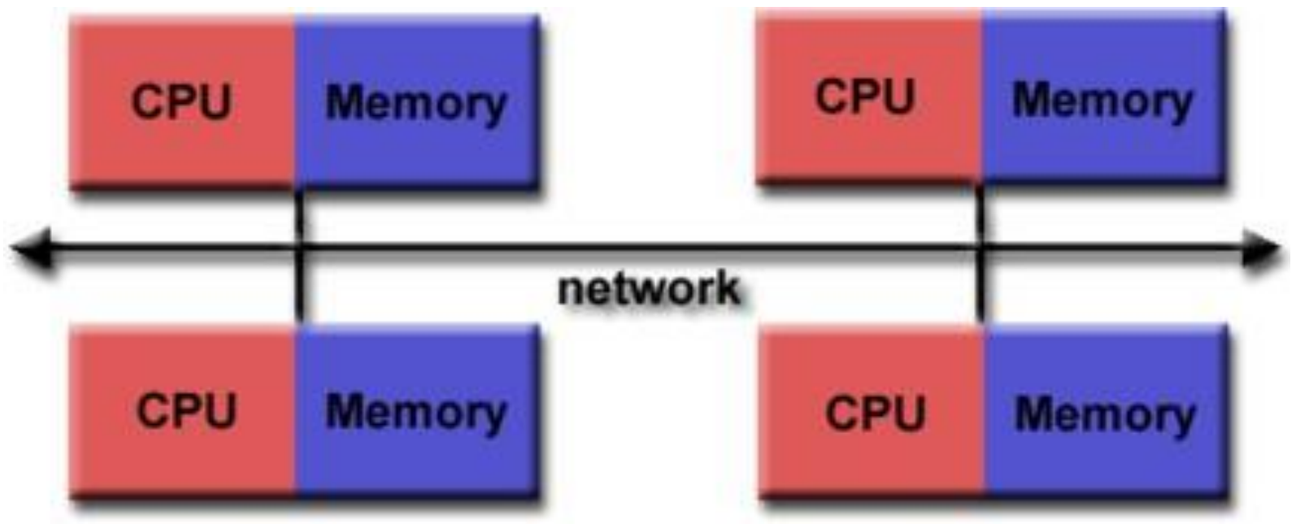


Graphics processing units (GPUs)



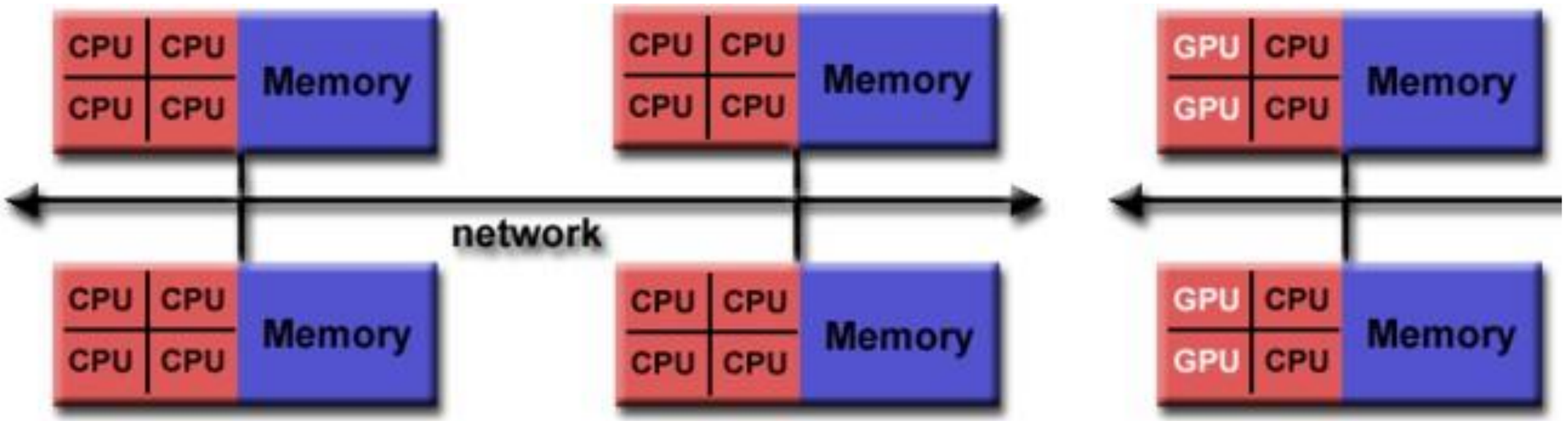
- In a GPU, the small local memories have much smaller access time than the large shared memory.
- Thus, as much as possible, cores access data in the local memories while the shared memory should essentially be used for data exchange between SMs.

Distributed Memory



- Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory and operate independently.
- Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Data exchange between processors is managed by the programmer , not by the hardware.

Hybrid Distributed-Shared Memory

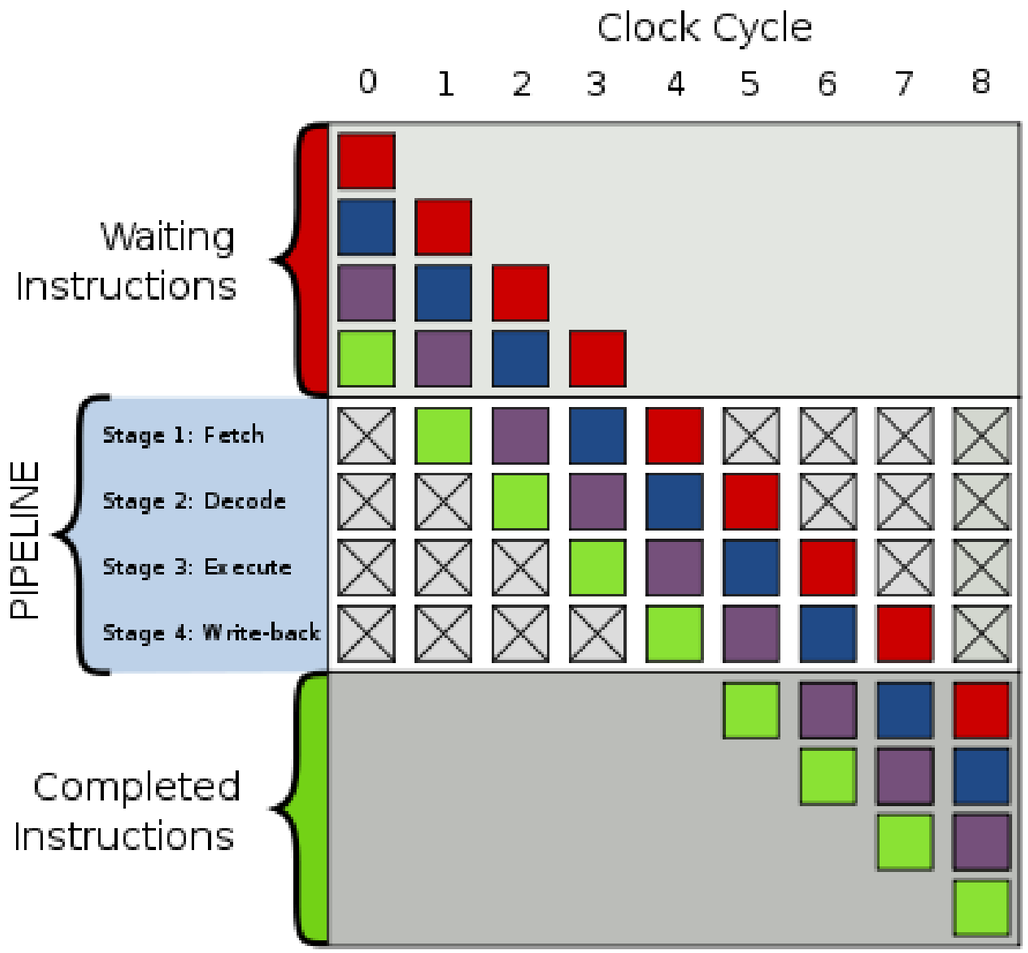


- The largest and fastest computers in the world today employ both shared and distributed memory architectures.
- Current trends seem to indicate that this type of memory architecture will continue to prevail.
- While this model allows for applications to scale, it increases the complexity of writing computer programs.

Types of Parallelism

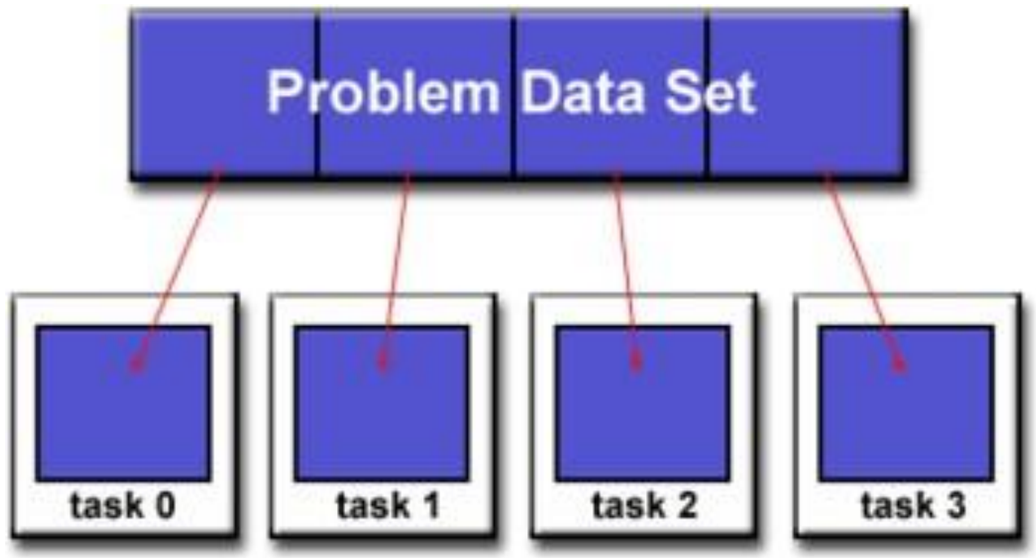
- Pipelining is a common way to organize work with the objective of optimizing throughput.
- It turns out that this is also a way to execute concurrently several tasks (that is, work units) processable by the same pipeline.

Instruction pipeline



- Above is a generic pipeline with four stages: Fetch, Decode, Execute, Write-back.
- The top gray box is the list of instructions waiting to be executed; the bottom gray box is the list of instructions that have been completed; and the middle white box is the pipeline.

Data Parallelism



- The data set is typically organized into a common structure, such as an array.
- A set of tasks work collectively on that structure, however, each task works on a different region.
- Tasks perform the same operation on their region of work, for example, ”multiply every array element by some value”.

Data parallelism (2/2)

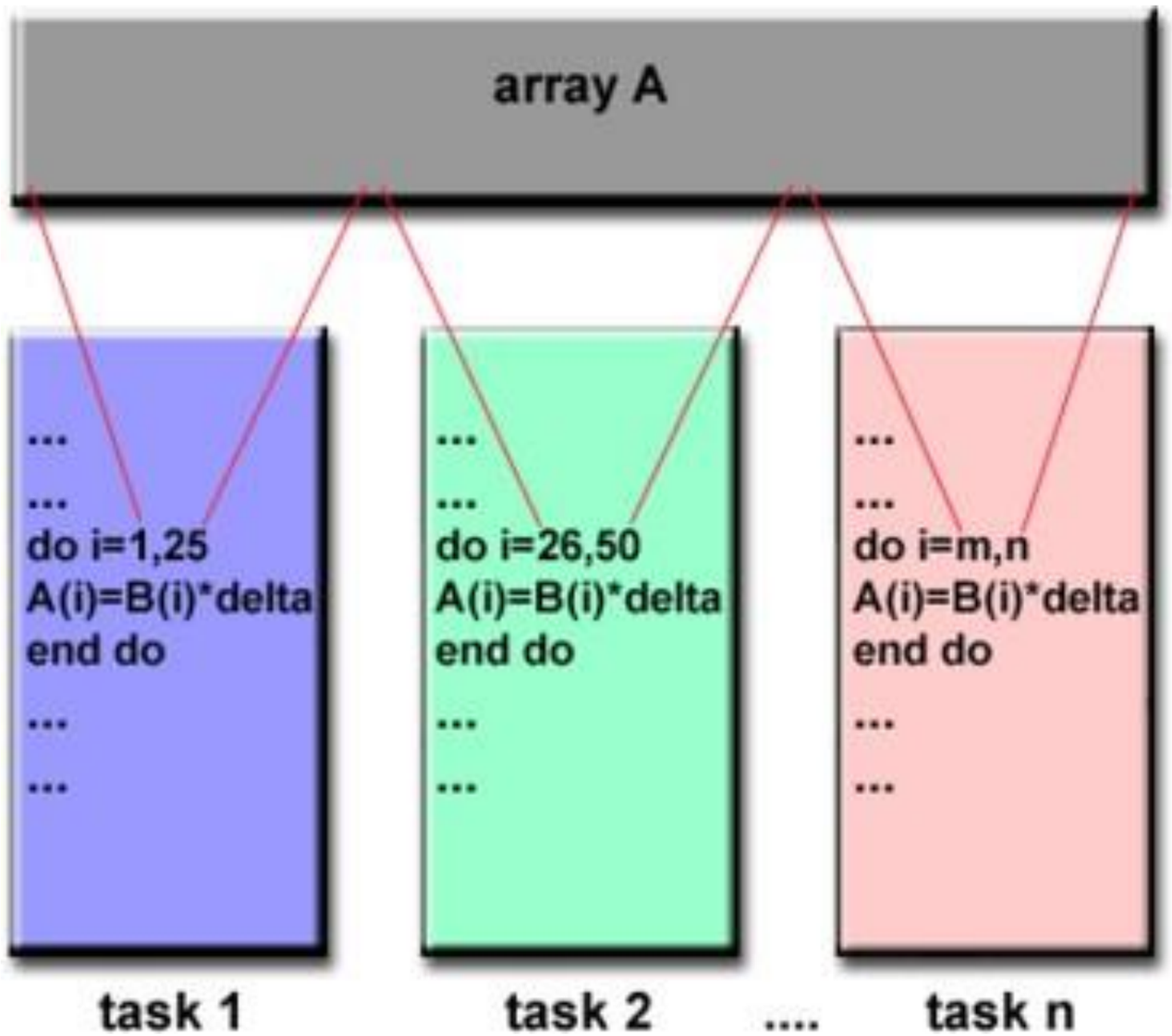


Illustration of a data-centric parallel program.

Task parallelism (1/4)

```
program:
 ...
 if CPU="a" then
     do task "A"
 else if CPU="b" then
     do task "B"
 end if
 ...
 end program
```

- Task parallelism is achieved when each processor executes a different thread (or process) on the same or different data.
- The threads may execute the same or different code.

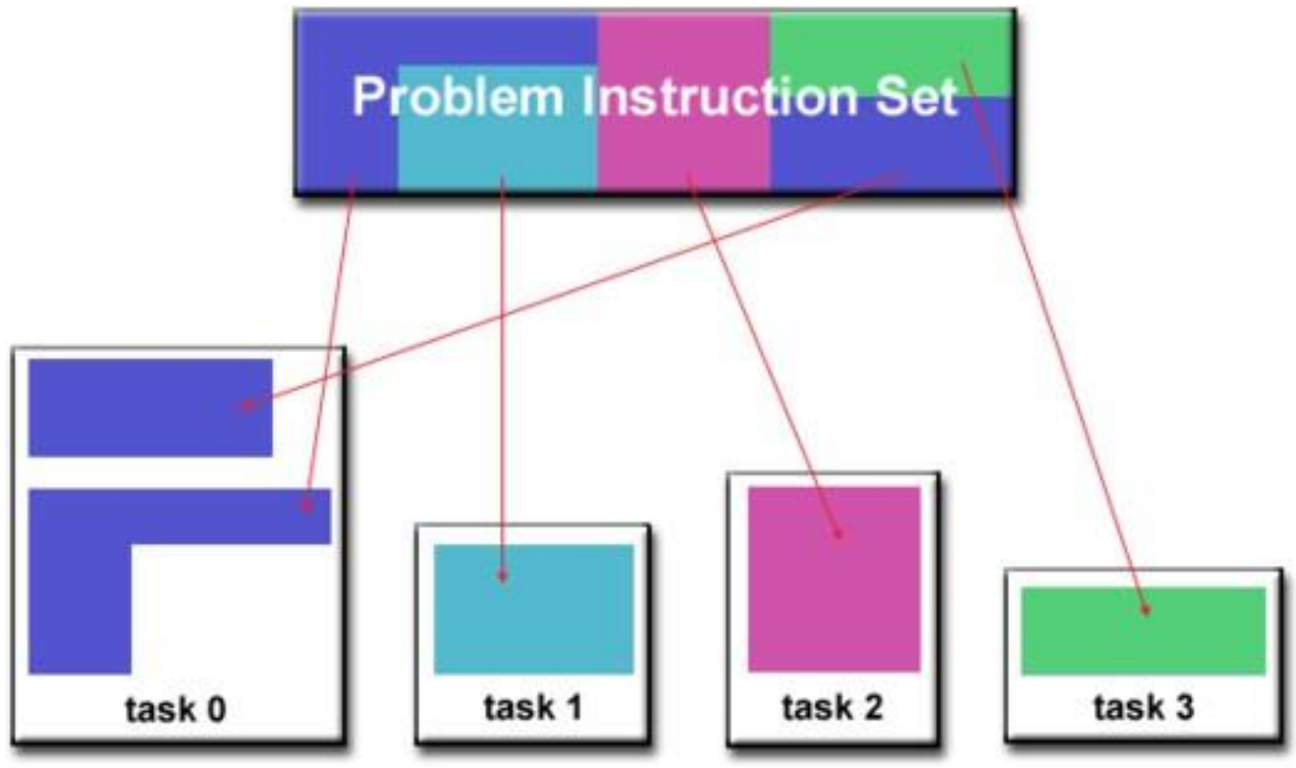
Task parallelism (2/4)

```
Code executed by CPU "a":

 program:
 ...
 do task "A"
 ...
 end program
```

```
Code executed by CPU "b":

 program:
 ...
 do task "B"
 ...
 end program
```

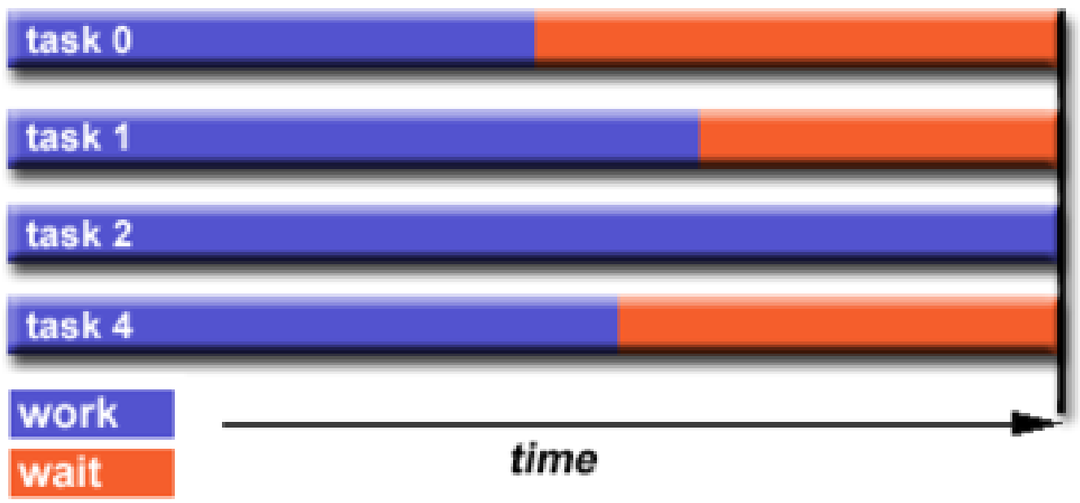
- In the general case, different execution threads communicate with one another as they work.
- Communication usually takes place by passing data from one thread to the next as part of a work-flow.

Task parallelism (3/4)



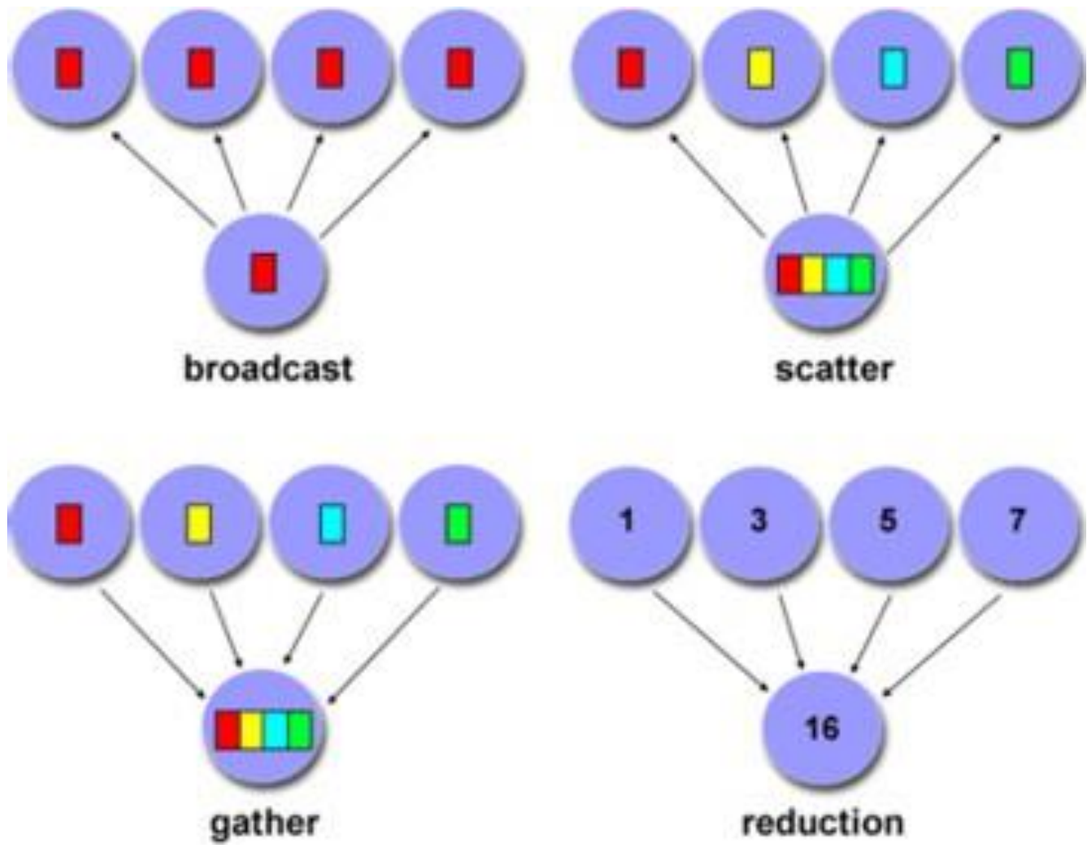
- Task parallelism can be regarded as a more general scheme than data parallelism.
- It applies to situations where the work can be decomposed evenly or where the decomposition of the work is not predictable.

Task parallelism (4/4)



- In some situations, one may feel that work can be decomposed evenly. However, as time progresses, some tasks may finish before others Then,
- some processors may become idle and should be used, if other tasks can be launched. This mapping of tasks onto hardware resources is called scheduling.
- In data-centric parallel applications, scheduling can be done off-line (that is, by the programmer) or by the hardware (like GPUs).
- For task-centric parallel applications, it is desirable that scheduling is done on-line (that is, dynamically) so as to cover cases where tasks consume unpredictable amounts of resources.

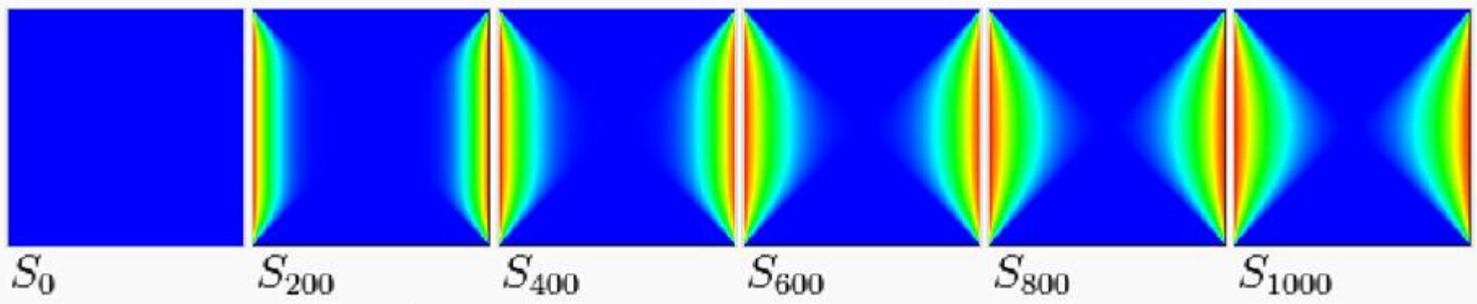
Patterns in task or data distribution



- Exchanging data among processors in a parallel fashion provides fundamental examples of concurrent programs.
- Above, a master processor broadcasts or scatters data or tasks to slave processors.
- The same master processor gathers or reduces data from slave processors.

Stencil computations

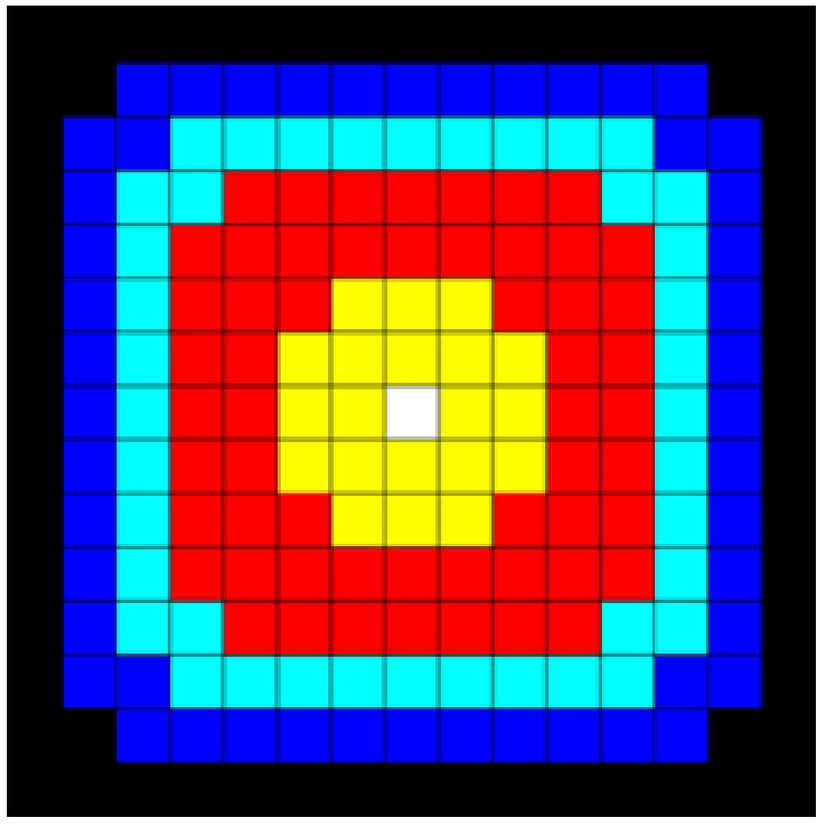
$$I = [0, \ldots, 99]^2$$
$$S = \mathbb{R}$$
$$S_0 : \mathbb{Z}^2 \to \mathbb{R}$$
$$S_0((x,y)) = \begin{cases} 1, & x < 0 \\ 0, & 0 \le x < 100 \\ 1, & x \ge 100 \end{cases}$$
$$s = ((0,-1),(-1,0),(1,0),(0,1))$$
$$T : \mathbb{R}^4 \to \mathbb{R}$$
$$T((x_1,x_2,x_3,x_4)) = 0.25 \cdot (x_1 + x_2 + x_3 + x_4)$$



2D Jacobi Iteration on a 100^2 Array

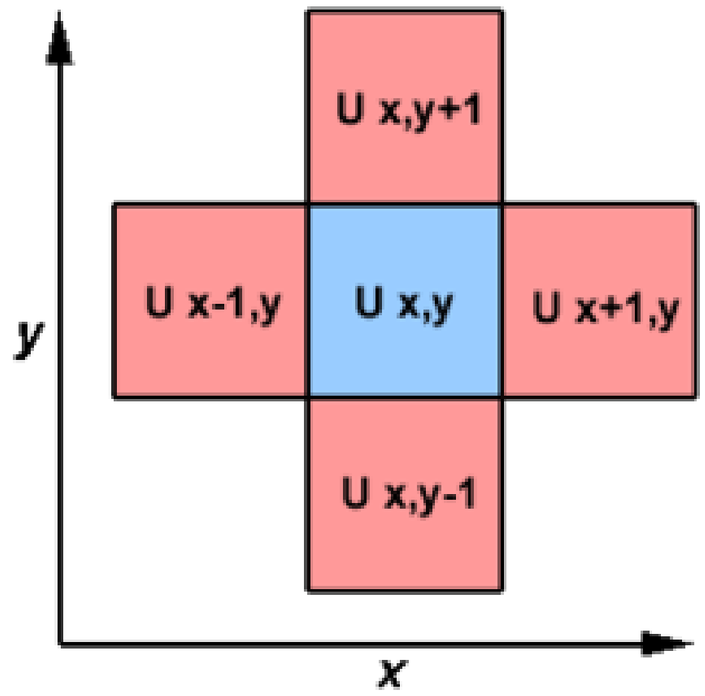
- In scientific computing, stencil computations are very common.
- Typically, a procedure updates array elements according to some fixed pattern, called stencil.
- In the above, a 2D array of 100×100 elements is updated by the stencil T.

Stencil computations (2/3)



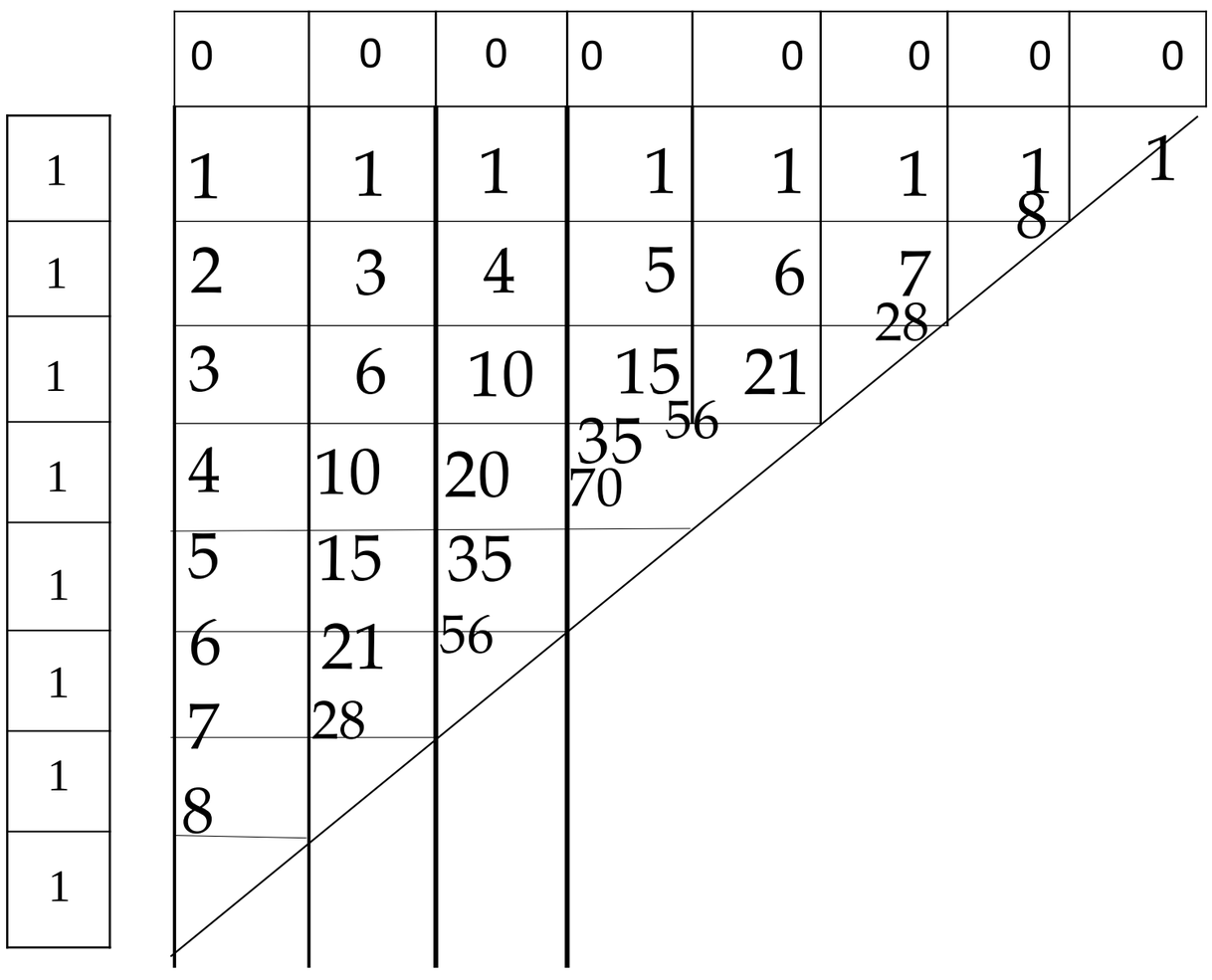
- The above picture illustrates dissipation of heat into a 2D grid.
- A differential equation rules this phenomenon.
- Once this discretized, through the finite element method, this leads a stencil computation.

Stencil computations (3/3)



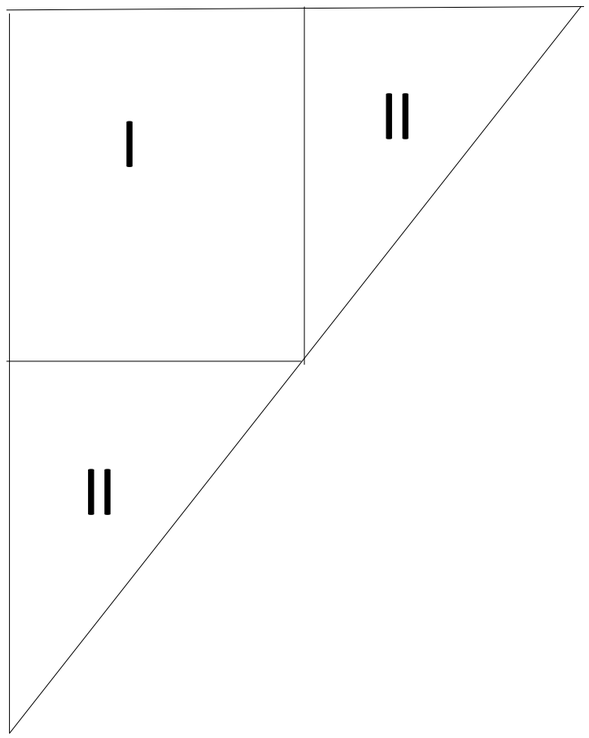
- The above picture illustrates dissipation of heat into a 2D grid.
- A differential equation rules this phenomenon.
- Once this discretized, through the finite element method, this leads a stencil computation.

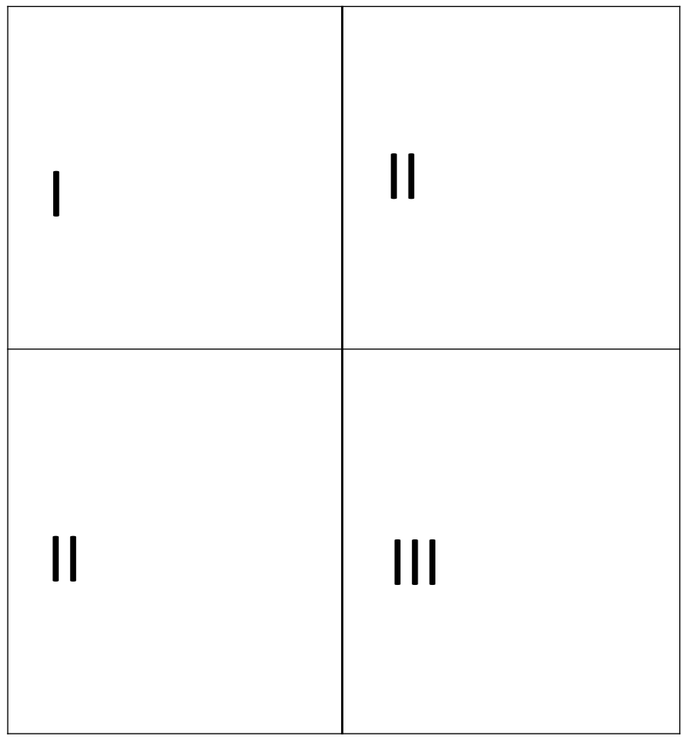
Pascal triangle construction: another stencil computation



Construction of the Pascal Triangle: nearly the simplest stencil computation!

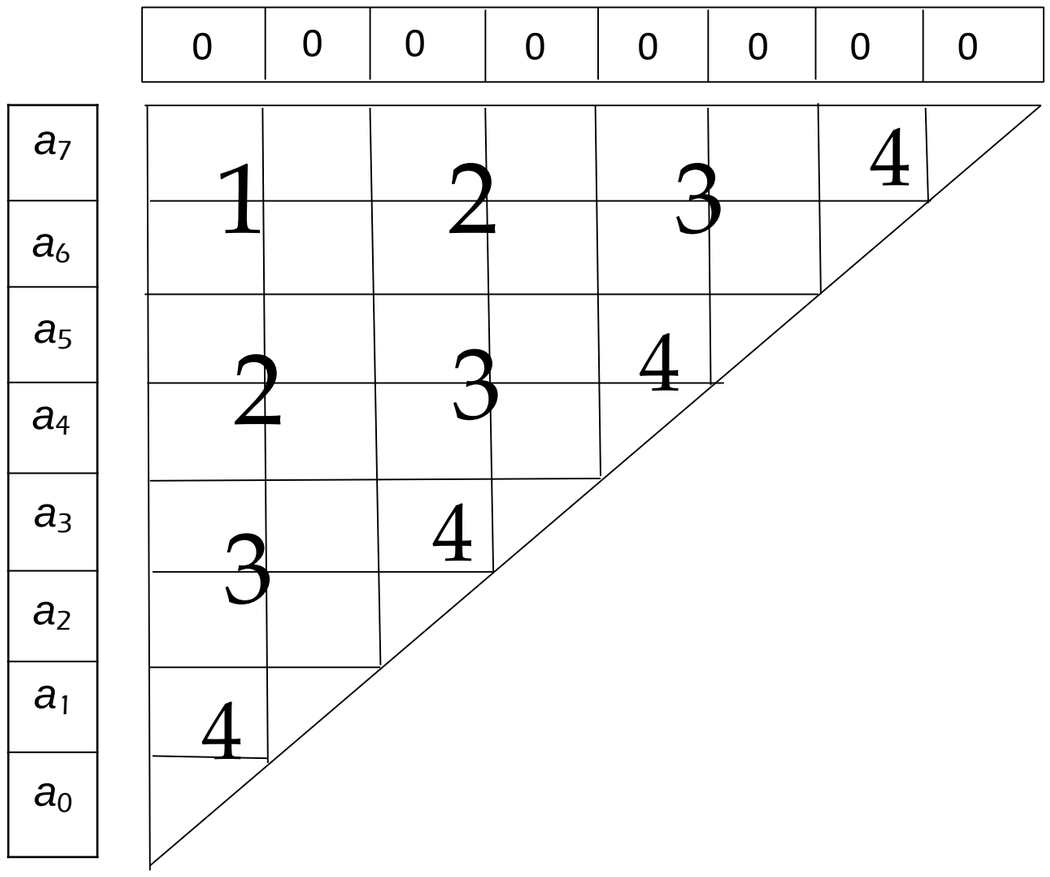
Divide and conquer: principle





- Each triangle region can computed as a square region followed by two (concurrent) triangle regions.
- Each square region can also be computed in a divide and conquer manner.

Blocking strategy: principle



- Let *B* be the order of a block and *n* be the number of elements.
- Each block is processed serially (as a task) and the set of all blocks is computed concurrently.