Parallel Programming Principle and Practice

Lecture 3 — Parallel Programming Models

Outline

- Introduction
- Shared Memory Model
- Thread Model
- Message Passing Model
- GPGPU Programming Model
- Data Intensive Computing Model

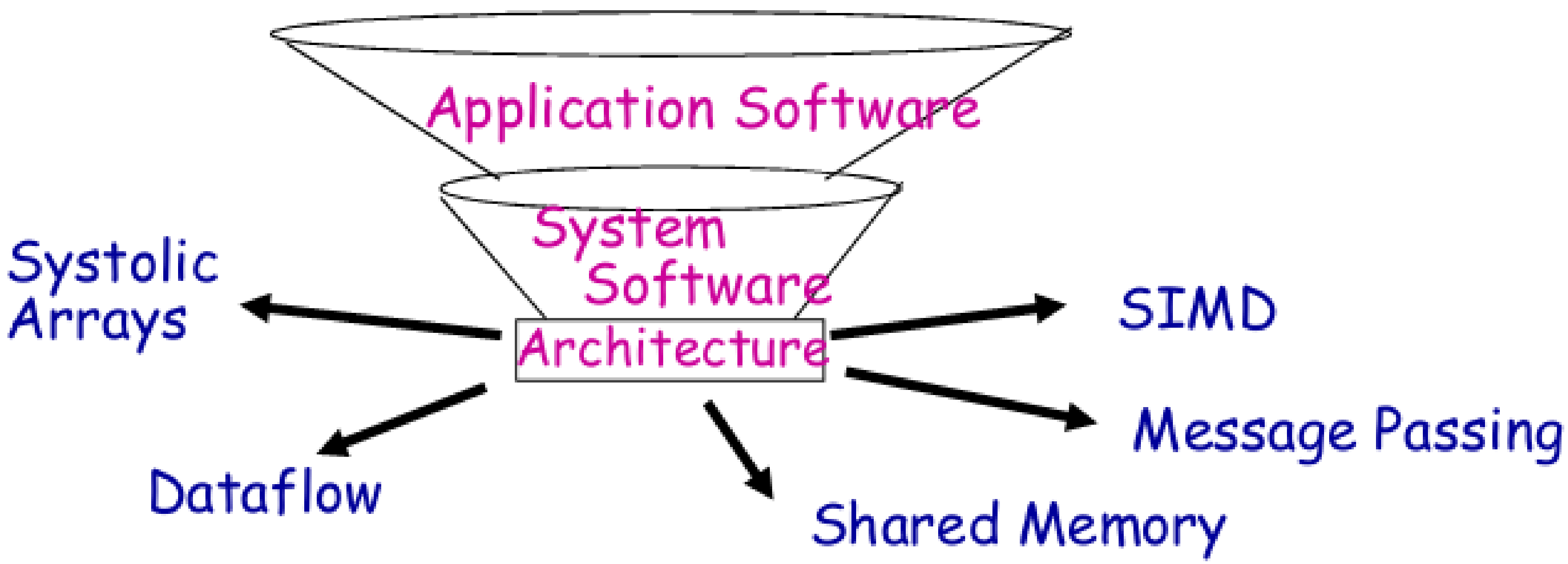
Parallel programming models

INTRODUCTION

History

Historically, parallel architectures tied to programming models

- Divergent architectures, with no predictable pattern of growth



Today

- Extension of "computer architecture" to support communication and cooperation
  - OLD: Instruction Set Architecture
  - NEW: Communication Architecture
- Defines
  - Critical abstractions, boundaries, and primitives (interfaces)
  - Organizational structures that implement interfaces (hw or sw)
- Compilers, libraries and OS are important bridges

Programming Model

- Description
  - The mental model the programmer has about the detailed execution of their application
- Purpose
  - Improve programmer productivity
- Evaluation
  - Expressibility
  - Simplicity
  - Performance

Programming Model

- What programmer uses in coding applications
- Specifies communication and synchronization
- Examples
  - Multiprogramming: no communication or synch. at program level
  - Shared address space: like bulletin board
  - Message passing: like letters or phone calls, explicit point to point
  - Data parallel: more strict, global actions on data
    - Implemented with shared address space or message passing

Programming Models

- von Neumann model
  - Execute a stream of instructions (machine code)
  - Instructions can specify
    - Arithmetic operations
    - Data addresses
    - Next instruction to execute
  - Complexity
    - Track billions of data locations and millions of instructions
    - Manage with
      - ✓ Modular design
      - ✓ High-level programming languages (isomorphic)

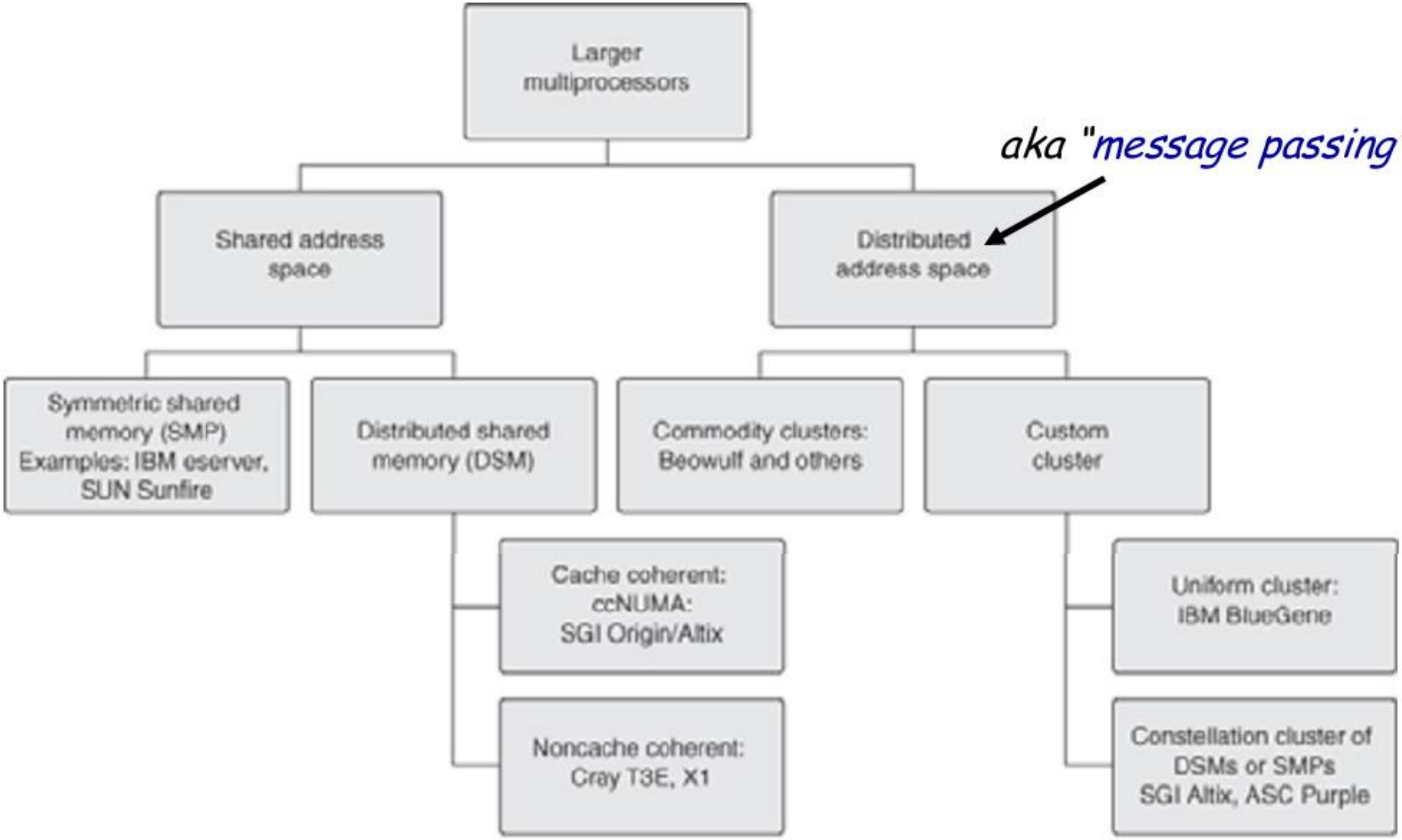
Programming Models

- Parallel Programming Models
  - Message passing
    - Independent tasks encapsulating local data
    - Tasks interact by exchanging messages
  - Shared memory
    - Tasks share a common address space
    - Tasks interact by reading and writing this space asynchronously
  - Data parallelization
    - Tasks execute a sequence of independent operations
    - Data usually evenly partitioned across tasks
    - Also referred to as “embarrassingly parallel”

Evolution of Architectural Models

- Historically, machines tailored to programming models
  - Programming model, communication abstraction, and machine organization lumped together as the "architecture"
- Evolution helps understand convergence
  - Identify core concepts
- Most common models
  - Shared memory model, threads model, distributed memory model, GPGPU programming model, data intensive computing model
- Other models
  - Dataflow, Systolic arrays
- Examine programming model, motivation, intended applications, and contributions to convergence

Taxonomy of Common Large-Scale SAS and MP Systems



Parallel programming models

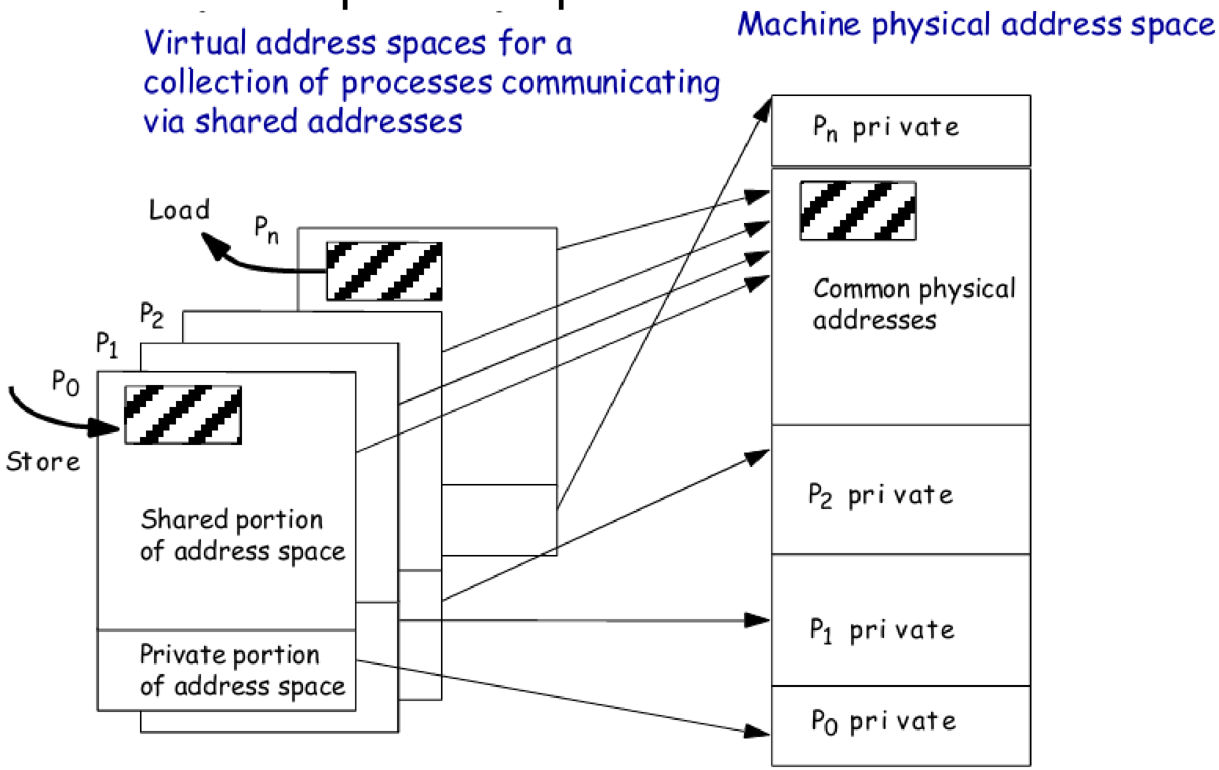
SHARED MEMORY MODEL

Shared Memory Model

- Any processor can **directly** reference any memory location
  - Communication occurs implicitly as result of loads and stores
- Convenient
  - Location transparency
  - Similar programming model to time-sharing on uniprocessors
    - Except processes run on different processors
    - Good throughput on multiprogrammed workloads
- Popularly known as *shared memory* machines or model
  - Ambiguous: memory may be physically distributed among processors

Shared Memory Model

- Process: virtual address space plus one or more threads of control
- Portions of address spaces of processes are shared



- Writes to shared address visible to other threads, processes
- Natural extension of uniprocessor model: conventional memory operations for comm.; special atomic operations for synchronization

Shared Memory Model

- In this programming model, tasks share a common address space, which they read and write 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

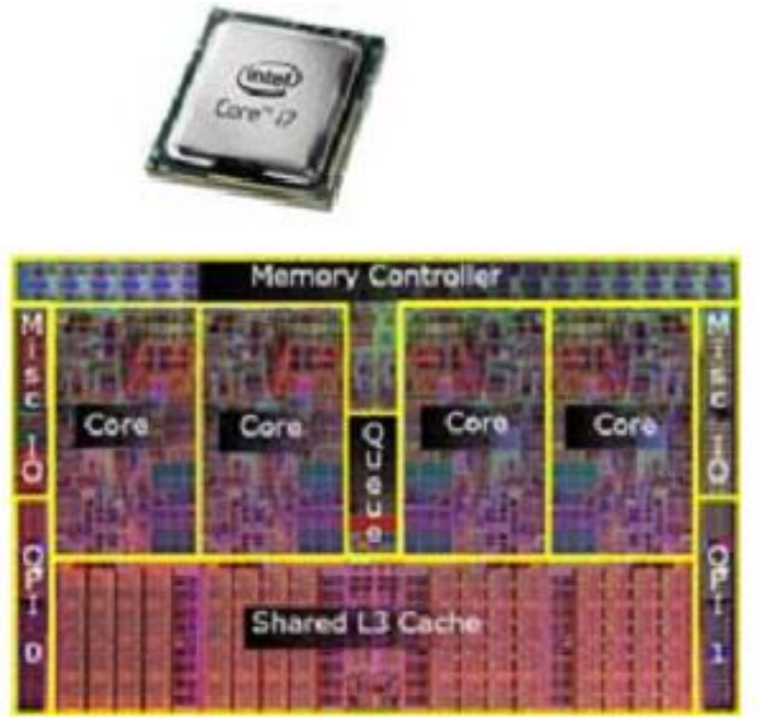
Shared Memory Model

- 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

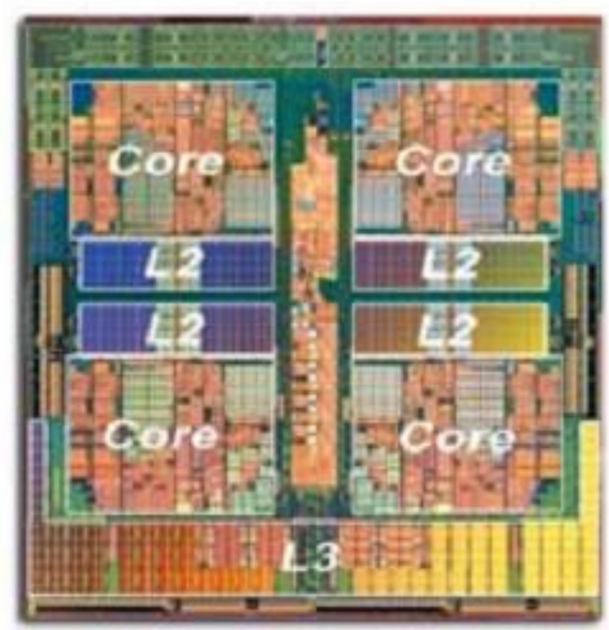
Implementations

- 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

Recent x86 Examples



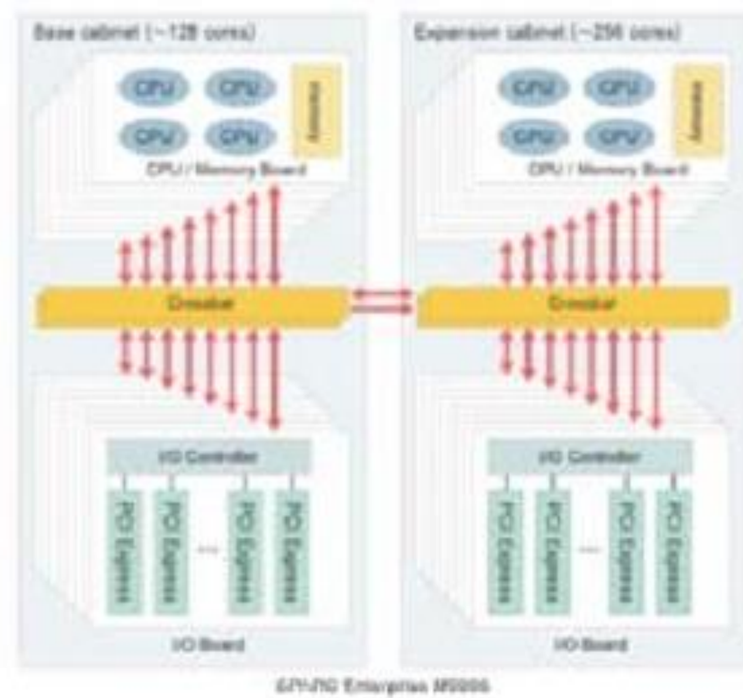
Intel's Quad Core i7



AMD's Quad-Core Phenom II

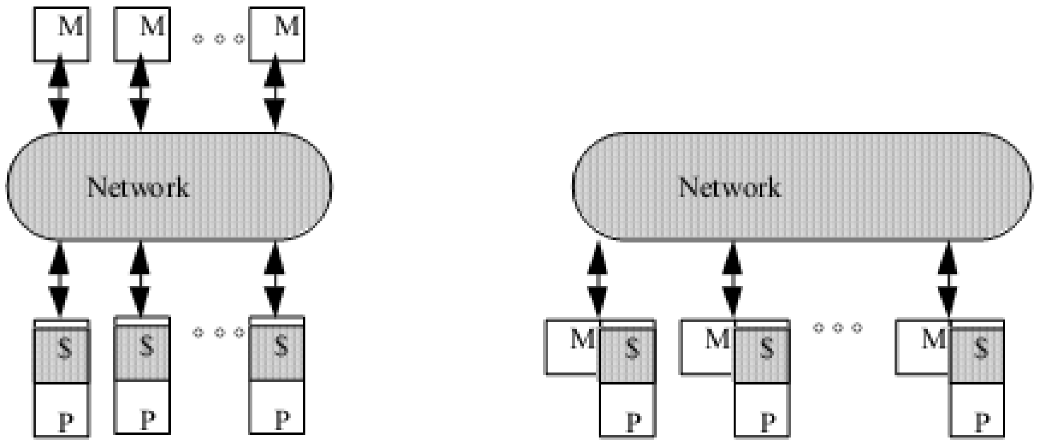
- Highly integrated, commodity systems
- On-chip: low-latency, high-bandwidth communication via shared cache

Example: Sun SPARC Enterprise M9000



- 64 SPARC64 VII+ quad-core processors (i.e. 256 cores)
- Crossbar bandwidth: 245 GB/sec (snoop bandwidth)
- Memory latency: 437-532 nsec (i.e. 1050-1277 cycles @ 2.4 GHz)
- Higher bandwidth, but also higher latency

Scaling Up

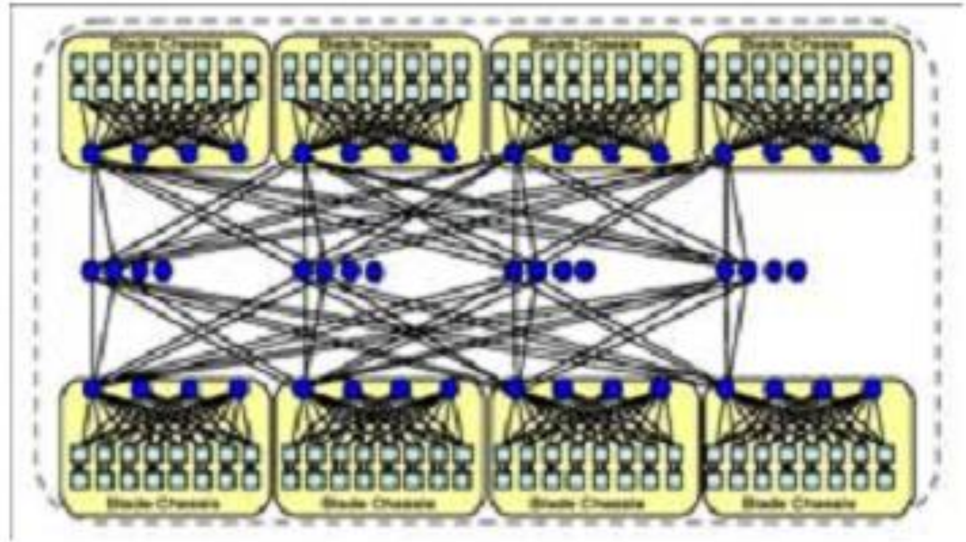


"Dance hall" Distributed memory

- Problem is interconnect: cost (crossbar) or bandwidth (bus)
- Dance-hall: bandwidth is not scalable, but lower cost than crossbar
  - Latencies to memory uniform, but uniformly large
- Distributed memory or non-uniform memory access (NUMA)
  - Construct shared address space out of simple message transactions across a general-purpose network (e.g. read-request, read-response)

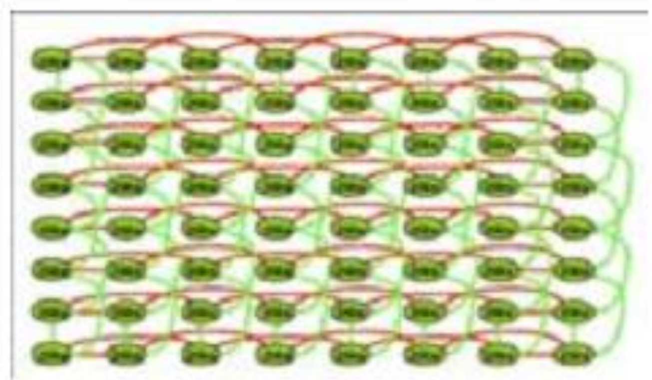
Example: SGI Altix UV 1000

Blacklight at the PSC (4096 cores)



256 socket (2048 core) fat-tree
(this size is doubled in Blacklight via a torus)

- Scales up to 131,072 cores
- 15GB/sec links
- Hardware cache coherence



8x8 torus

Parallel programming models

THREAD MODEL

Threads Model

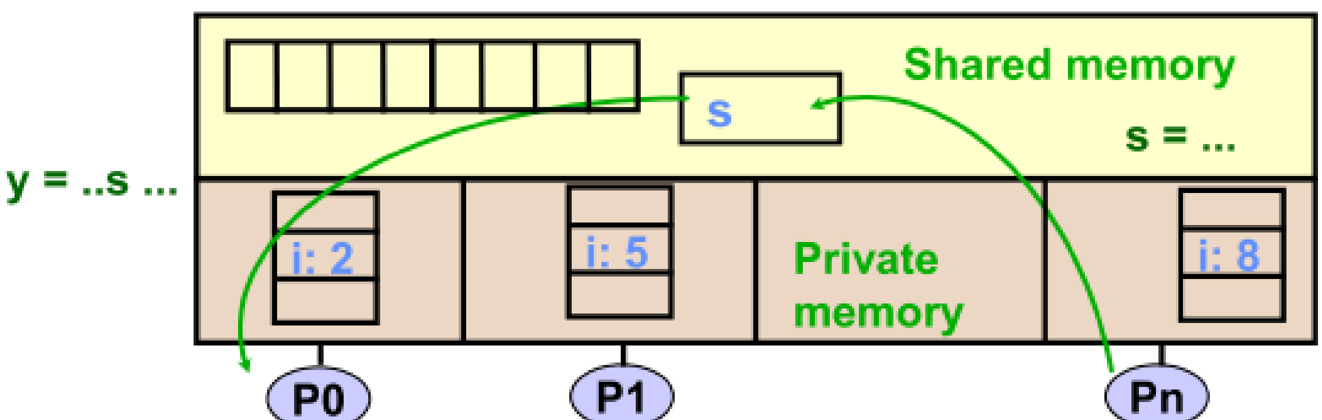
- This programming model **is a type of shared memory programming**

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

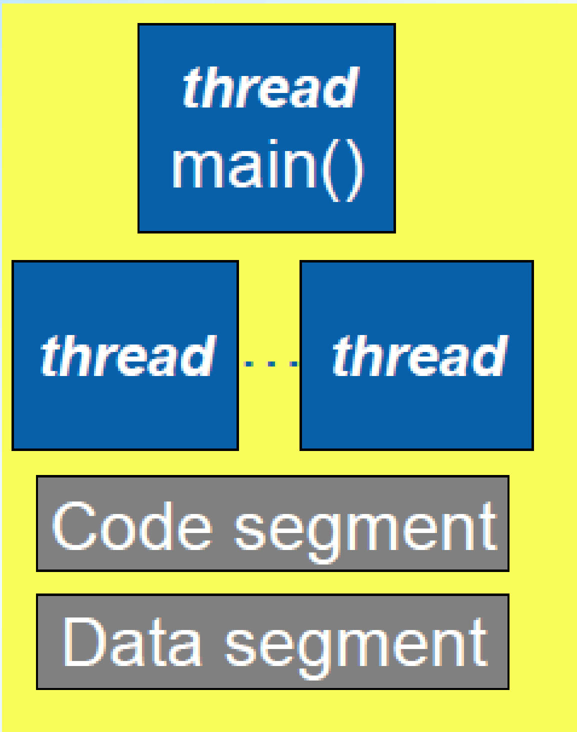
- Perhaps *the most simple analogy* that can be used to describe threads is the concept of a single program that includes a number of subroutines

Threads Model

- Program is a collection of threads of control
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap
  - Threads communicate implicitly by writing and reading shared variables
  - Threads coordinate by synchronizing on shared variables



Processes and Threads



- Modern operating systems load programs as processes
  - Resource holder
  - Execution
- A process starts executing at its entry point as a thread
- Threads can create other threads within the process
- All threads within a process share code & data segments

Amdahl's Law

- Describes the upper bound of parallel speedup (scaling)
- Helps think about the effects of overhead

Gene M. Amdahl, "*Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities*", 1967

Amdahl's law (Amdahl's speedup model)

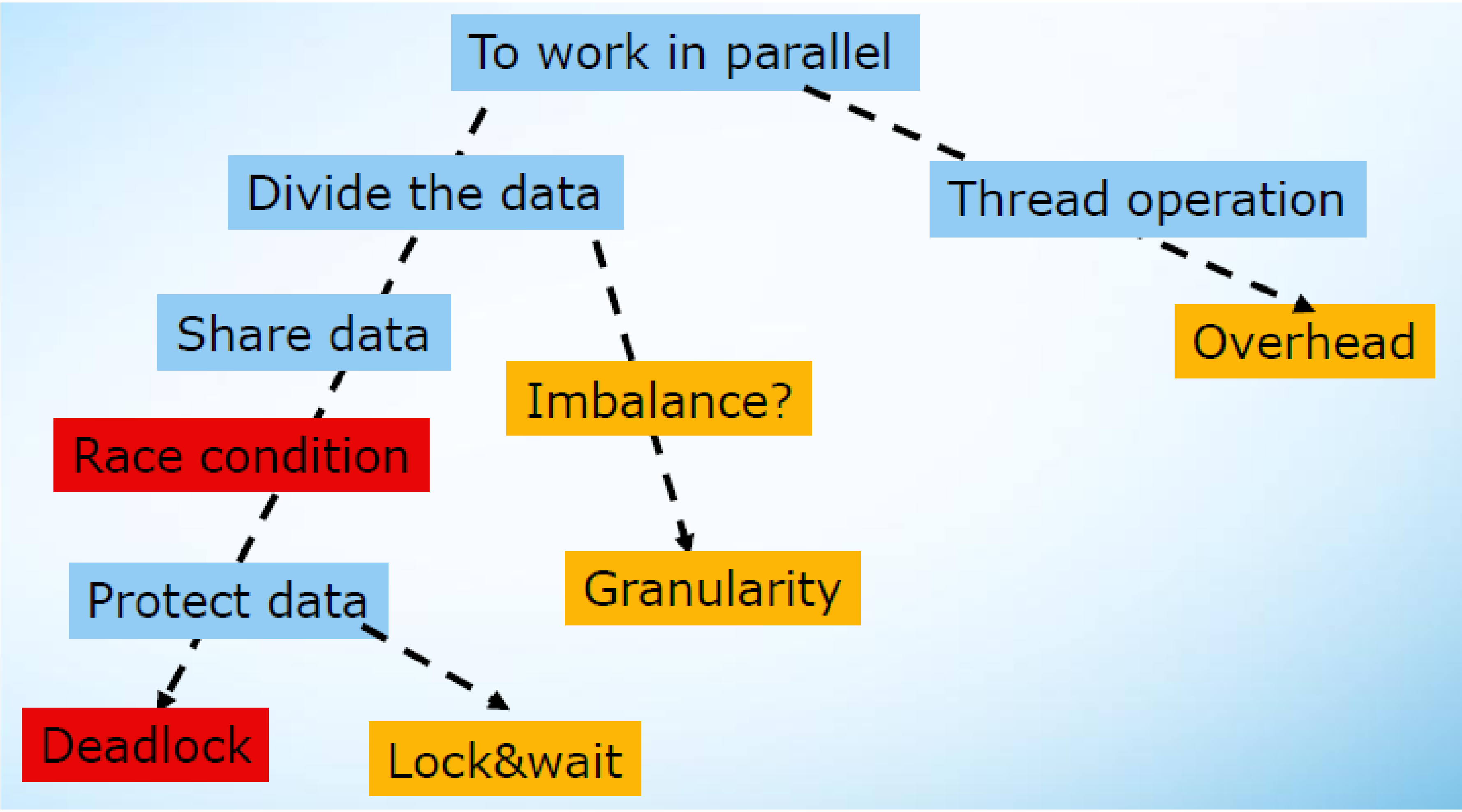
$$Speedup_{Amdahl} = \frac{1}{(1-f)+\frac{f}{n}}$$

$$\lim_{n\to\infty} Speedup_{Amdahl} = \frac{1}{1-f}$$

f is the parallel portion

Implications

Where Are the Problems From?



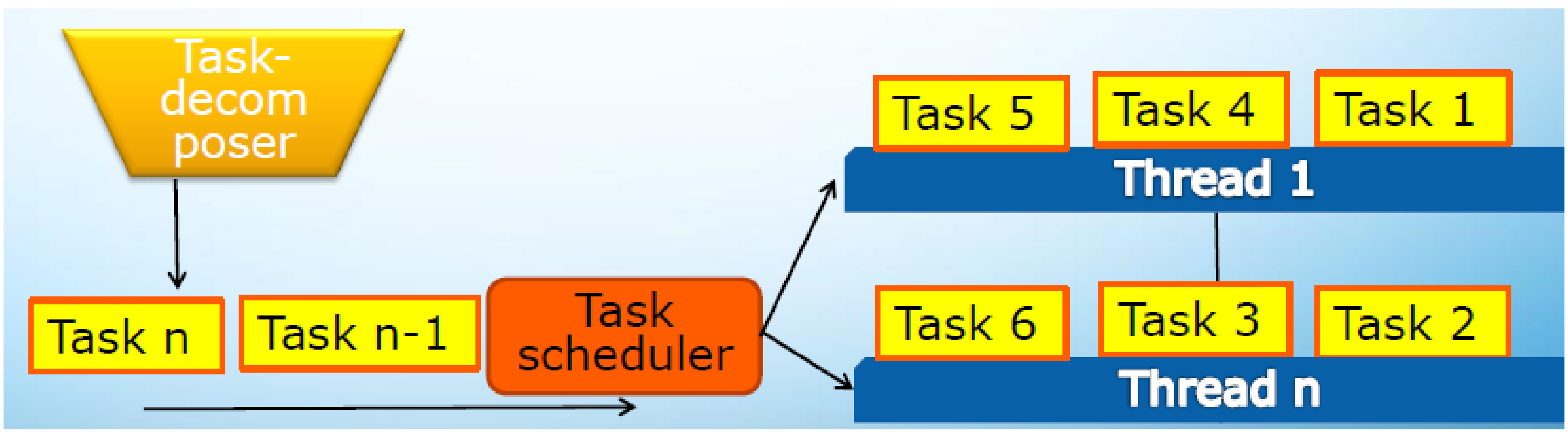
Remove the error **Tune for high speedup**

Decomposition

- Data decomposition
  - Break the entire dataset into smaller, discrete portions, then process them in parallel
  - Folks eat up a cake
- Task decomposition
  - Divide the whole task based on natural set of independent sub-tasks
  - Folks play a symphony
- Considerations
  - Cause less or no share data
  - Avoid the dependency among sub-tasks, otherwise become pipeline

Task and Thread

- A task consists the data and its process, and task scheduler will attach it to a thread to be executed
- Task operation is much cheaper than threading operation
- Ease to balance workload among threads by stealing
- Suit for list, tree, map data structure



Task and Thread

- Considerations
  - Many more tasks than threads
    - More flexible to schedule the task
    - Easy to balance workload
  - Amount of computation within a task must be large enough to offset overhead of managing task and thread
  - Static scheduling
    - Tasks are collections of separate, independent function calls or are loop iterations
  - Dynamic scheduling
    - Task execution length is variable and is unpredictable
    - May need an additional thread to manage a shared structure to hold all tasks

Race Conditions

- Threads "*race*" against each other for resources
  - Execution order is assumed but cannot be guaranteed
- Storage conflict is most common
  - Concurrent access of same memory location by multiple threads, at least one thread is writing
- Determinacy race and data race
- May not be apparent at all times
- Considerations
  - Control shared access with critical regions
    - Mutual exclusion and synchronization, critical session, atomic
  - Scope variables to be local to threads
    - Have a local copy for shared data
    - Allocate variables on thread stack

Deadlock

- 2 or more threads wait for each other to release a resource
- A thread waits for a event that never happen, like suspended lock
- Most common cause is locking hierarchies
- Considerations
  - Always lock and un-lock in the same order, and avoid hierarchies if possible
  - Use atomic

```
DWORD WINAPI threadA(LPVOID arg)
{
  EnterCriticalSection(&L1);
    EnterCriticalSectio
      processA(data1, da
    LeaveCriticalSection(&L2);
  LeaveCriticalSection(&L1);
  return(0);
}
```

ThreadB: L2, then L1

ThreadA: L1, then L2

```
DWORD WINAPI threadB(LPVOID arg)
{
  EnterCriticalSection(&L2);
    EnterCriticalSection(&L1);
      processB(data2, data1) ;
    LeaveCriticalSection(&L1);
  LeaveCriticalSection(&L2);
  return(0);
}
```

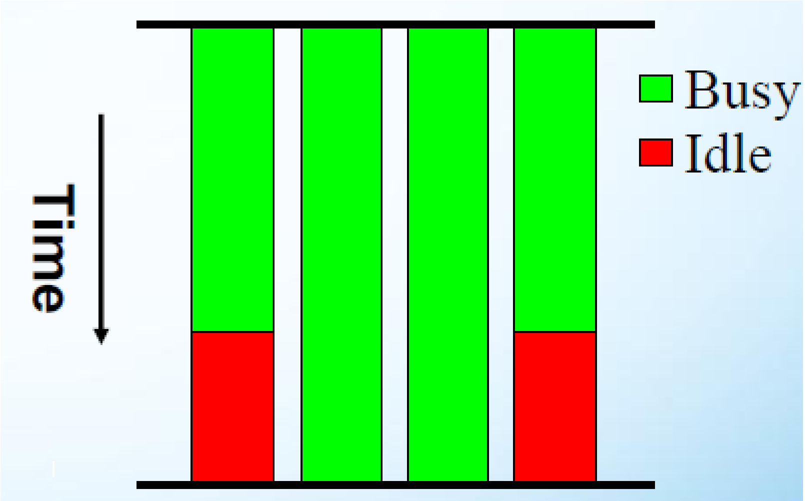
Thread Safe Routine/Library

- It functions correctly during simultaneous execution by multiple threads
- Non-thread-safe indicators
  - Access global/static variables or the heap
  - Allocate/reallocate/free resources that have global scope (files)
  - Indirect accesses through handles and pointers
- Considerations
  - Any variables changed must be local to each thread
  - Routines can use mutual exclusion to avoid conflicts with other threads

It is better to make a routine reentrant than to add synchronization
Avoids potential overhead

Imbalanced Workload

- All threads process the data in same way, but one thread is assigned more work, thus require more time to complete it and impact overall performance



- Considerations
  - Parallelize the inner loop
  - Incline to fine-grained
  - Choice the proper algorithm
  - Divide and conquer, master and worker, work-stealing

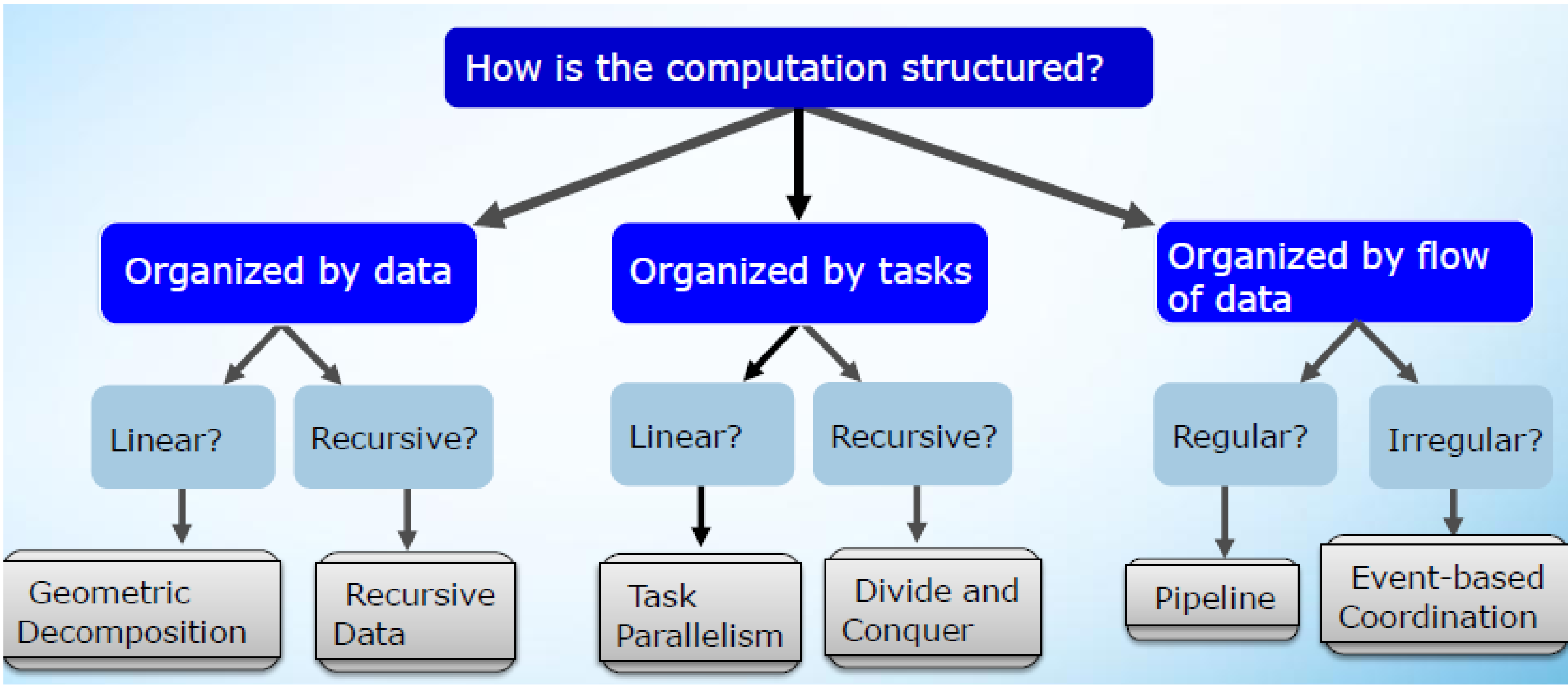
Granularity

- An extent to which a larger entity is subdivided
- Coarse-grained means fewer and larger components
- Fine-grained means more and smaller components
- Consideration
  - Fine-grained will increase the workload for task scheduler
  - Coarse-grained may cause the workload imbalance
  - Benchmark to set the proper granularity

Lock & Wait

- Protect shared data and ensure tasks executed in right order
- Improper usage causes a side-effect
- Considerations
  - Choose appropriate synchronization primitives
    - tbb::atomic, InterlockedIncrement, EnterCriticalSection…
  - Use non-blocking locks
    - TryEnterCriticalSection, pthread_mutex_try_lock
  - Reduce lock granularity
  - Don’t be a lock hub
  - Introduce a concurrent container for shared data

Parallel Algorithm



A Generic Development Cycle (1)

- Analysis
  - Find the hotspot and understand its logic
- Design
  - Identify the concurrent tasks and their dependencies
  - Decompose the whole dataset with minimal overhead of sharing or data movement between tasks
  - Introduce the proper parallel algorithm
  - Use proved parallel implementations
  - Memory management
    - Avoid heap contention among threads
    - Use thread-local storage to reduce synchronization
    - Detecting memory saturation in threaded applications
    - Avoid and identifying false sharing among threads

A Generic Development Cycle (2)

- Debug for correctness
  - Detect race conditions, deadlock, & memory issues
- Tune for performance
  - Balance the workload
  - Adjust lock & wait
  - Reduce thread operation overhead
  - Set the right granularity
  - Benchmark for scalibility

Intel Generic Development Cycle

Analysis

–Intel® VTune™ Amplifier XE

Design (Introduce Threads)

–Intel® IPP, MKL, Ct, TBB

–Intel® Parallel Composer XE with OpenMP*, Cilk, CEAN and other technologies.

Debug for correctness

–Intel® Parallel Inspector XE

–Intel Debugger

Tune for performance

– Intel® VTune™ Amplifier XE

Summary

- Threading applications require multiple iterations of designing, debugging, and performance tuning steps
- Use tools to improve productivity
- Unleash the power of dual-core and multi-core processors

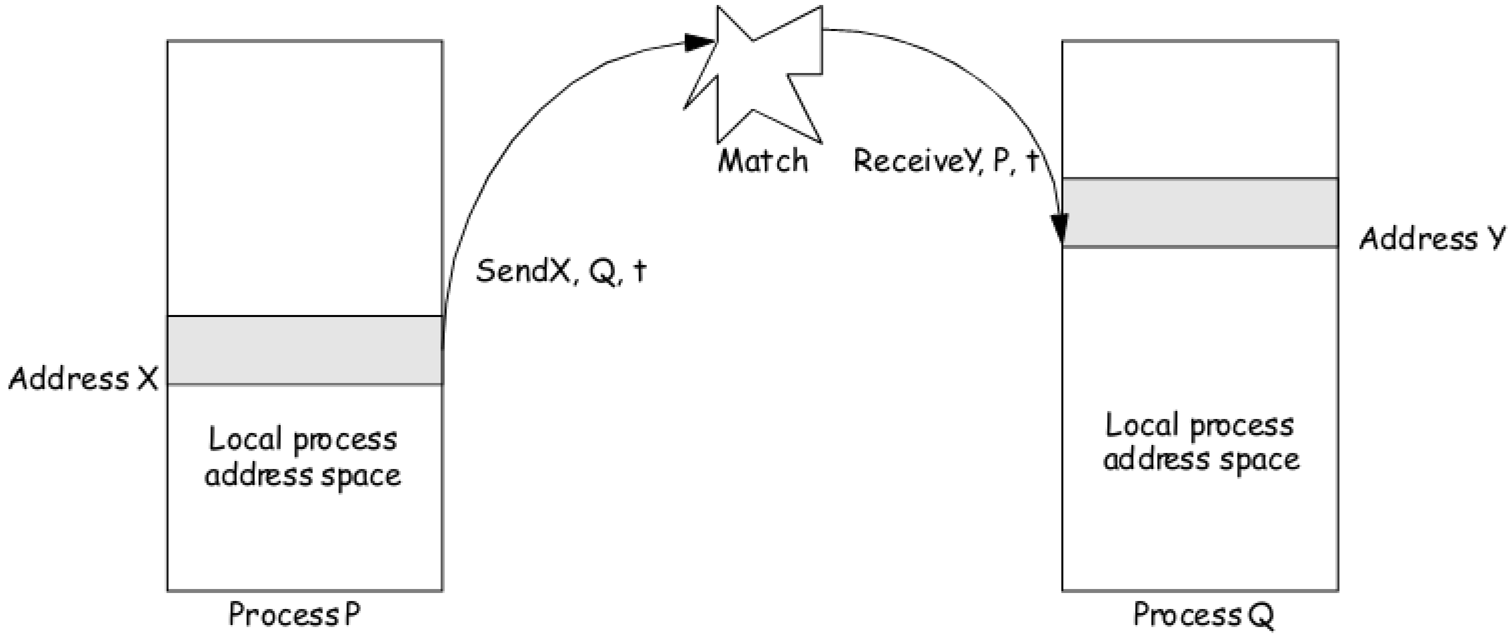
Parallel programming models

MESSAGE PASSING MODEL

Message Passing Architectures

- Complete computer as building block, including I/O
  - Communication via explicit I/O operations
- Programming model
  - directly access only private address space (local memory)
  - communicate via explicit messages (send/receive)
- High-level block diagram similar to distributed-mem SAS
  - But communication integrated at IO level, need not put into memory system
  - Easier to build than scalable SAS
- Programming model further from basic hardware ops
  - Library or OS intervention

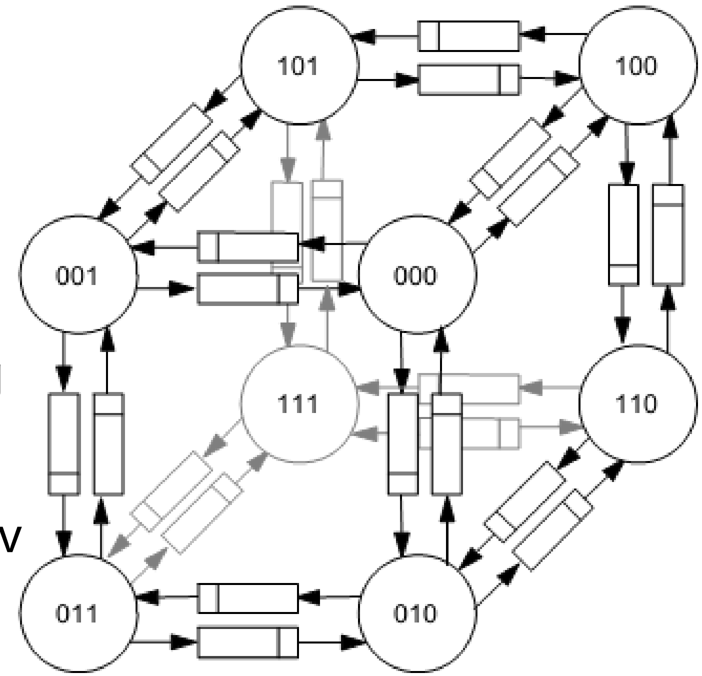
Message Passing Abstraction



- Send specifies buffer to be transmitted and sending process
- Recv specifies receiving process and application storage to receive into
- Memory to memory copy, but need to name processes
- Optional tag on send and matching rule on receive

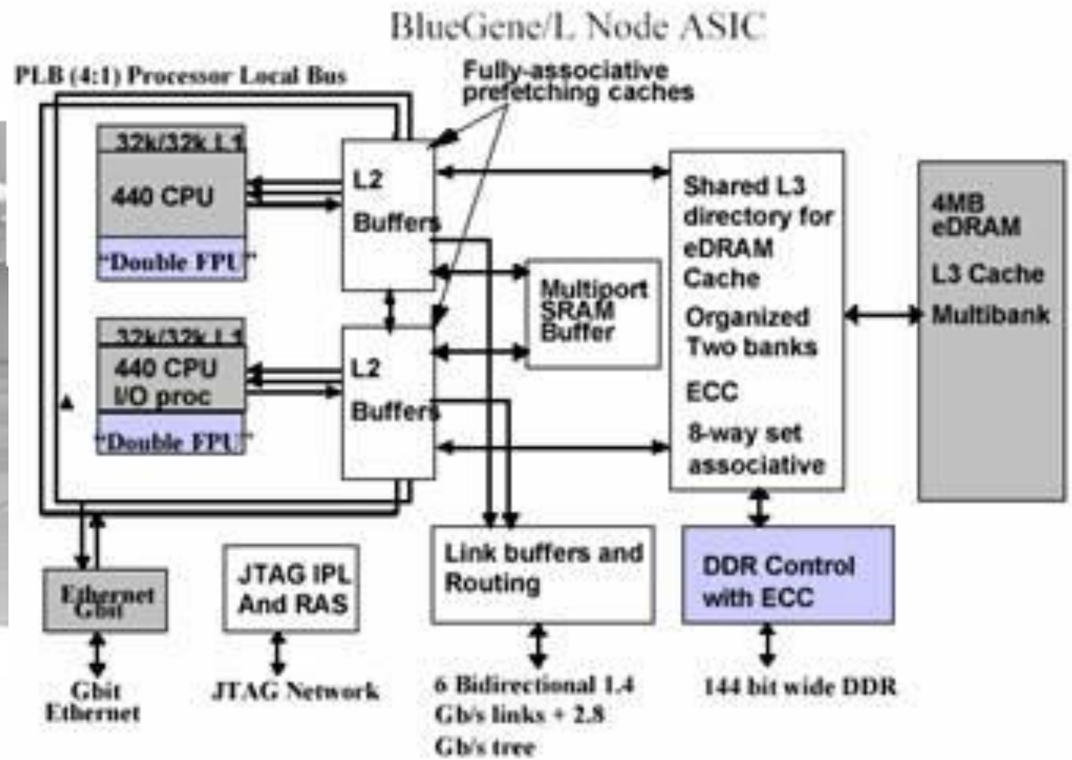
- Many overheads: copying, buffer management, protection

Evolution of Message Passing



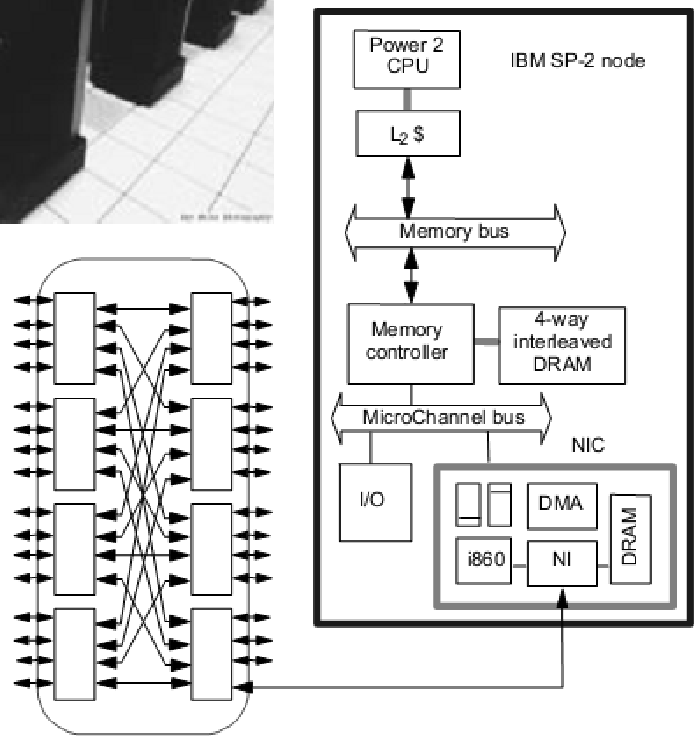
- **Early machines**: FIFO on each link
  - Hardware close to programming model
    - synchronous ops
  - Replaced by DMA, enabling non-blocking ops
    - Buffered by system at destination until recv
- **Diminishing role of topology**
  - Store & forward routing: topology important
  - Introduction of pipelined routing made it less so important
  - Cost is in node network interface
  - Simplifies programming

Example: IBM Blue Gene/L



Nodes: 2 PowerPC 440s; everything except DRAM on one chip

Example: IBM SP-2



- Made out of essentially complete RS6000 workstations
- Network interface integrated in I/O bus (bw limited by I/O bus)

Toward Architectural Convergence

- Evolution and role of software have blurred boundary
  - Send/recv supported on SAS machines via buffers
  - Can construct global address space on MP using hashing
  - Page-based (or fine-grained) shared virtual memory

- Programming models distinct, but organizations converging
  - Nodes connected by general network and communication assists
  - Implementations also converging, at least in high-end machines

Implementations

- 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
- 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

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/

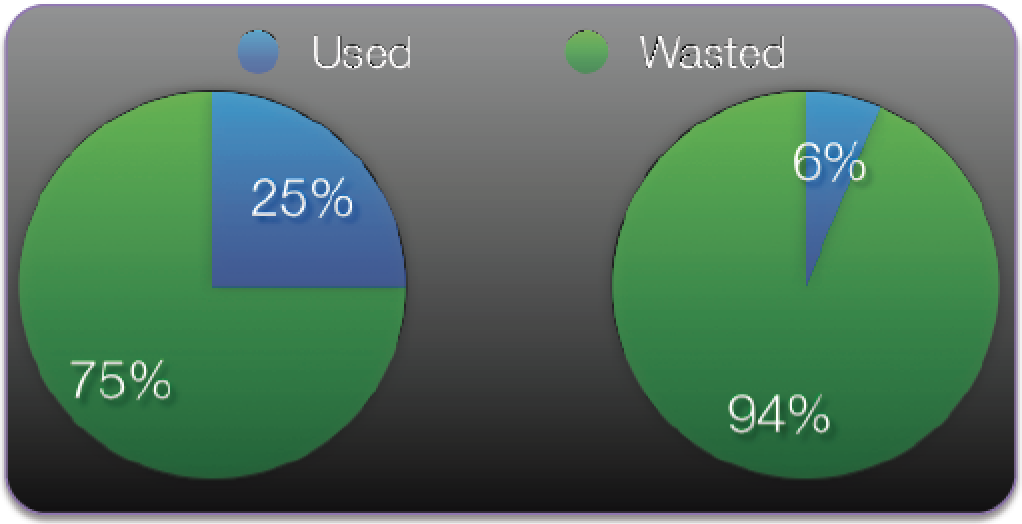
- MPI is now the *de facto* industry standard for message passing, replacing virtually all other message passing implementations used for production work

- MPI implementations exist virtually for all popular parallel computing platforms. Not all implementations include everything in both MPI-1 and MPI-2

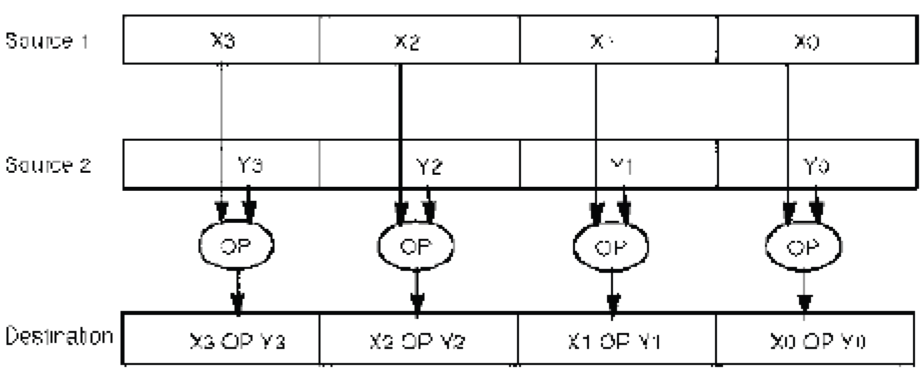
Parallel programming models

GPGPU PROGRAMMING MODEL

CUDA Goals: SIMD Programming

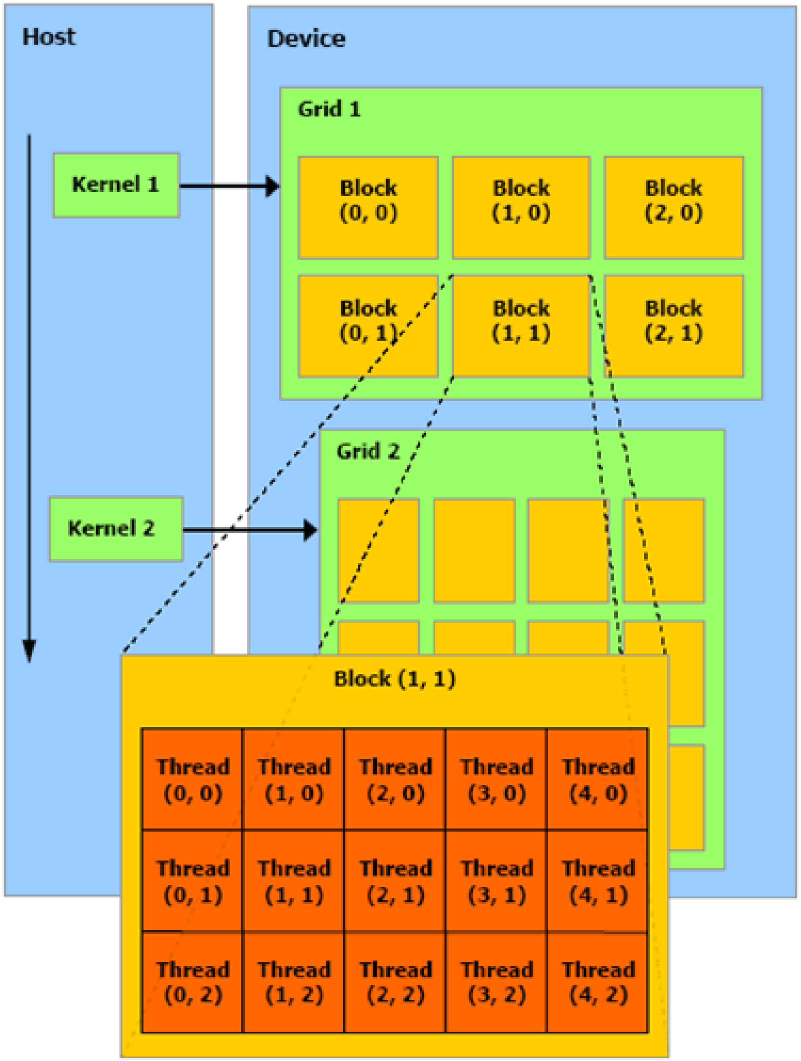


4 way SIMD (SSE) 16 way SIMD (LRB)

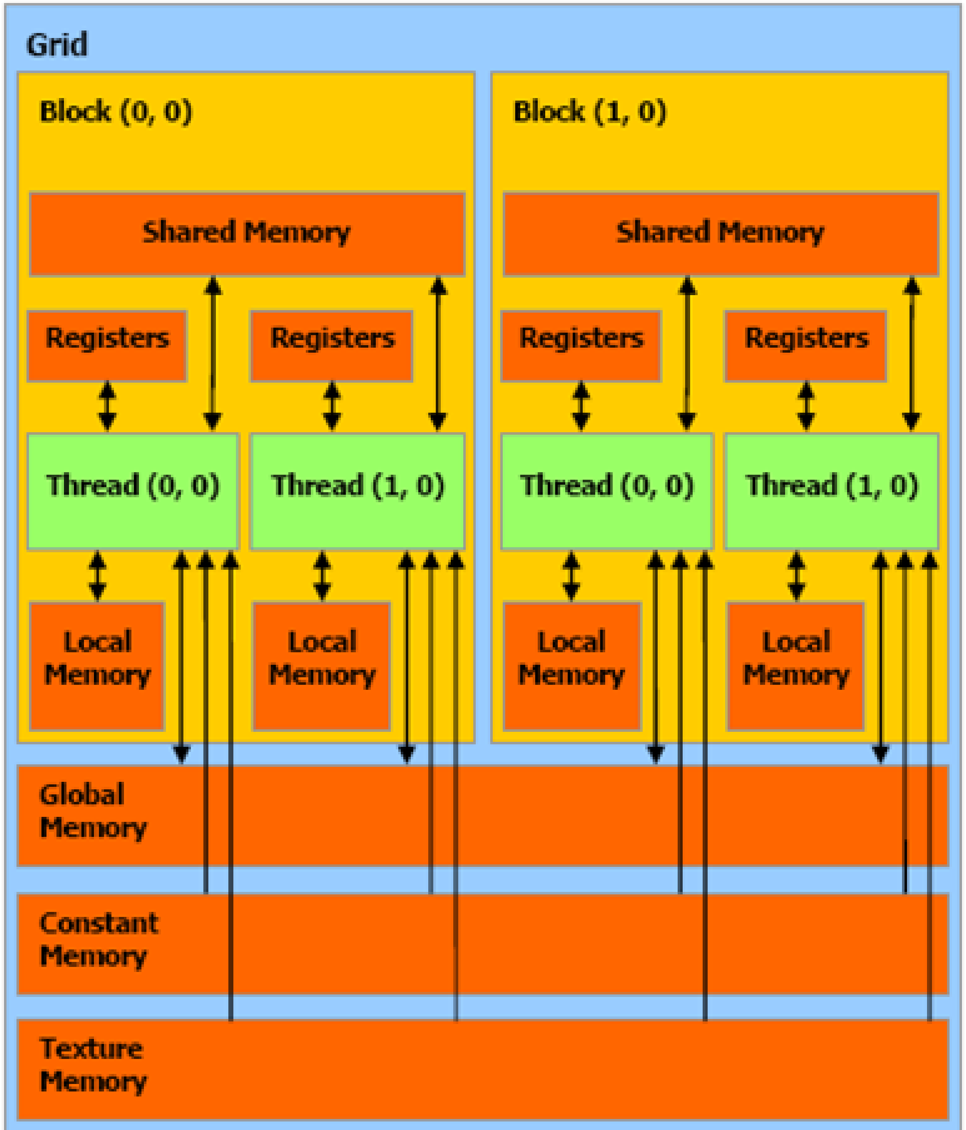


- Hardware architects love SIMD, since it permits a very space and energy-efficient implementation
- However, standard SIMD instructions on CPUs are inflexible, and difficult to use, difficult for a compiler to target
- CUDA thread abstraction will provide programmability at the cost of additional hardware

CUDA Programming Model



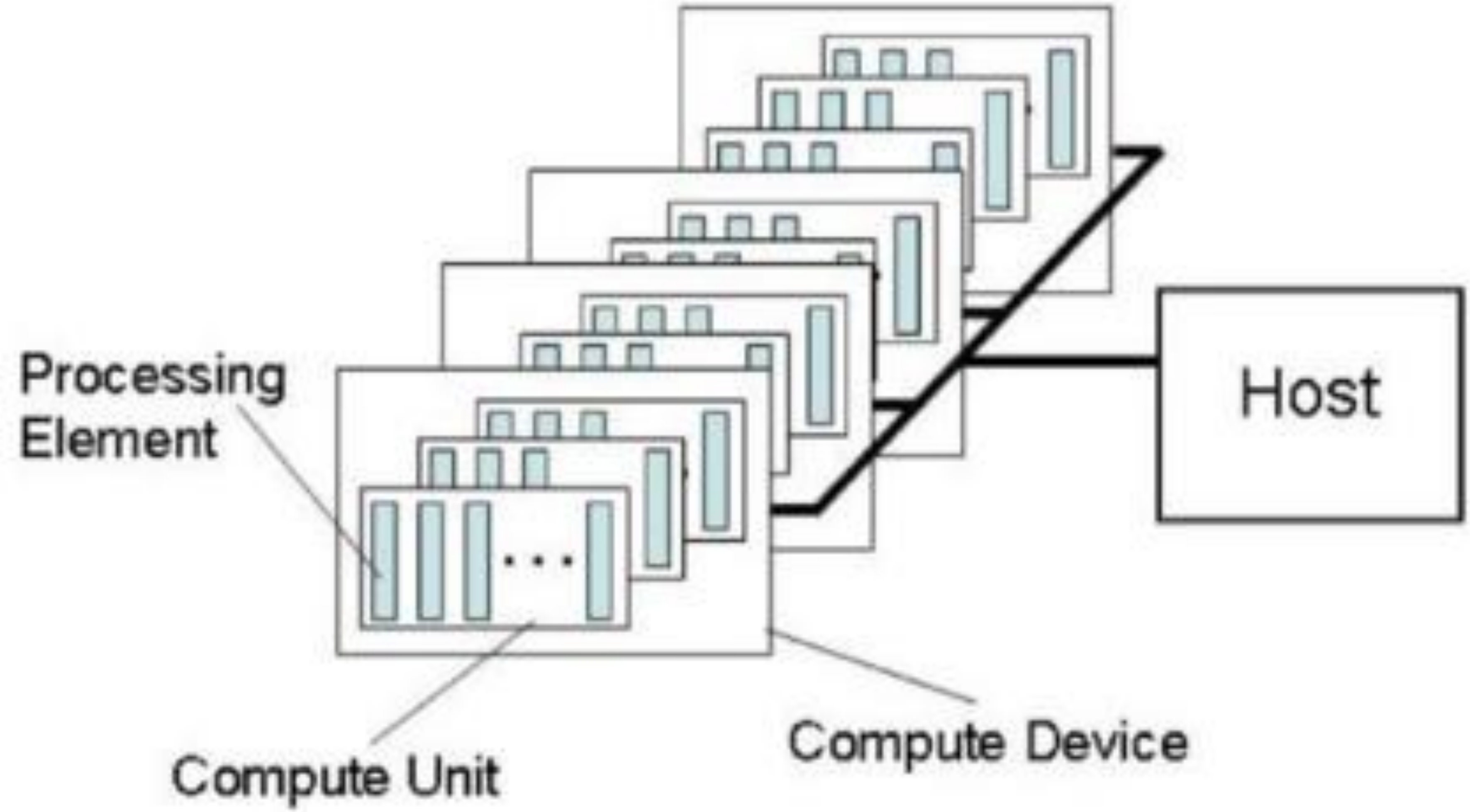
The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks



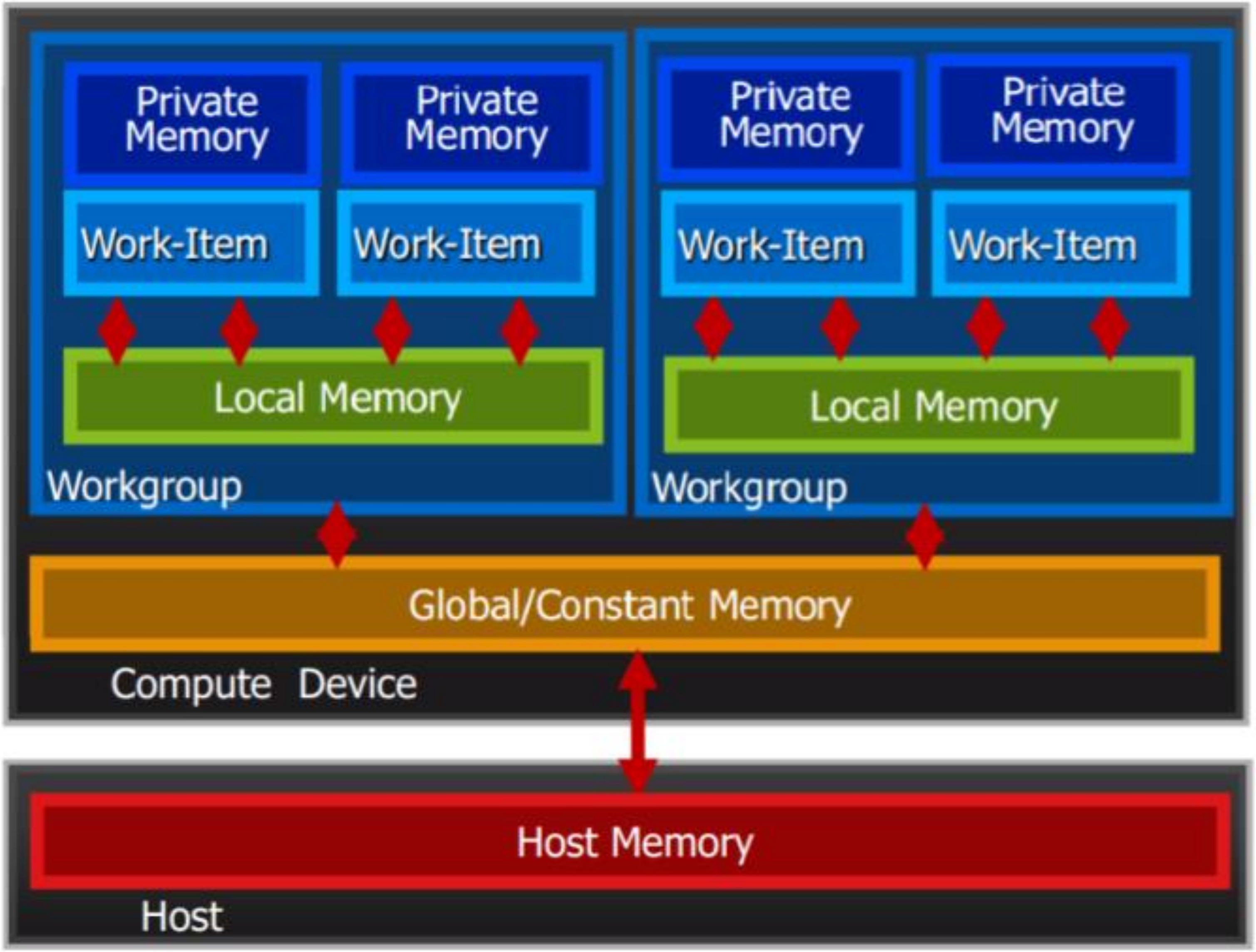
OpenCL Programming Model

- OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, DSPs, FPGAs and other processors or hardware accelerators
- Data Parallel - SPMD
  - Work-items in a work-group run the same program
  - Update data structures in parallel using the work-item ID to select data and guide execution
- Task Parallel
  - One work-item per work group … for coarse grained task-level parallelism
  - Native function interface: trap-door to run arbitrary code from an OpenCL command-queue

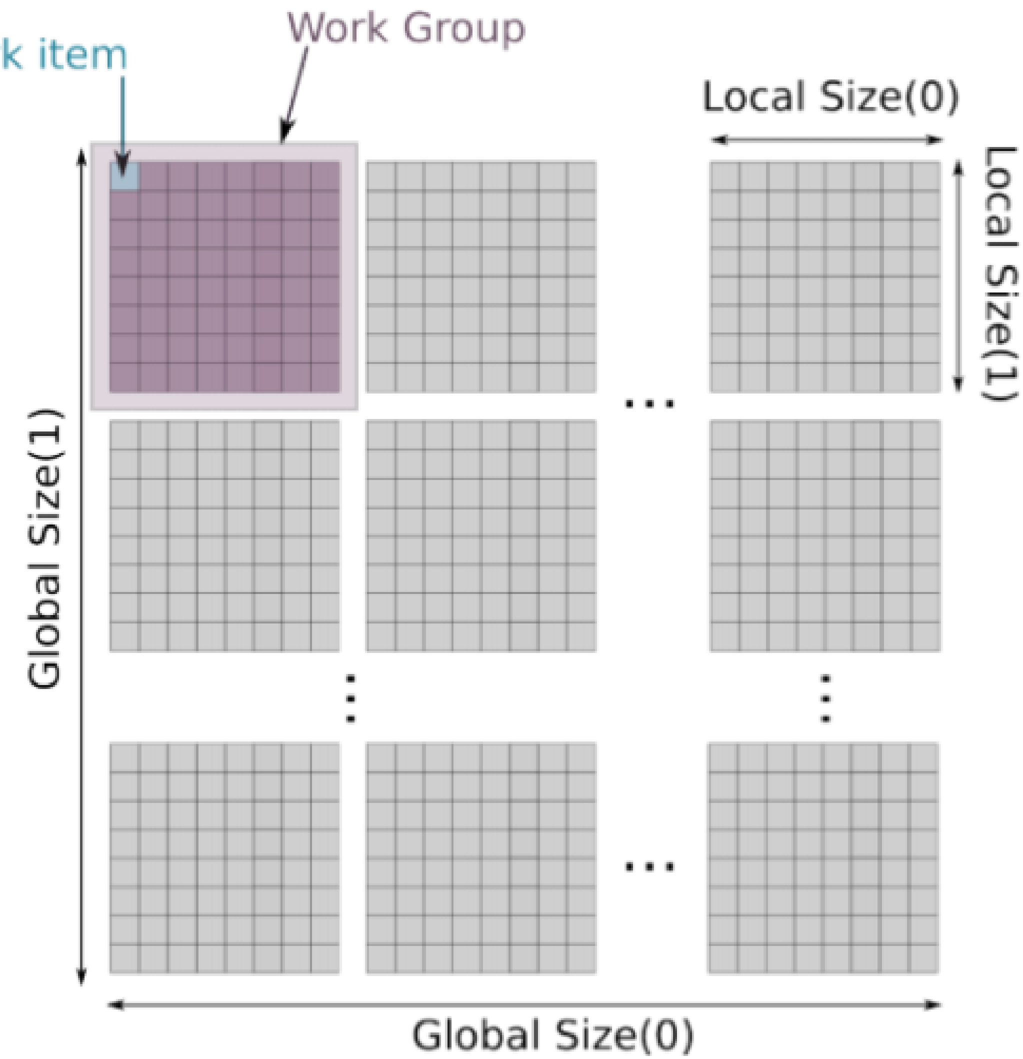
OpenCL Platform Model



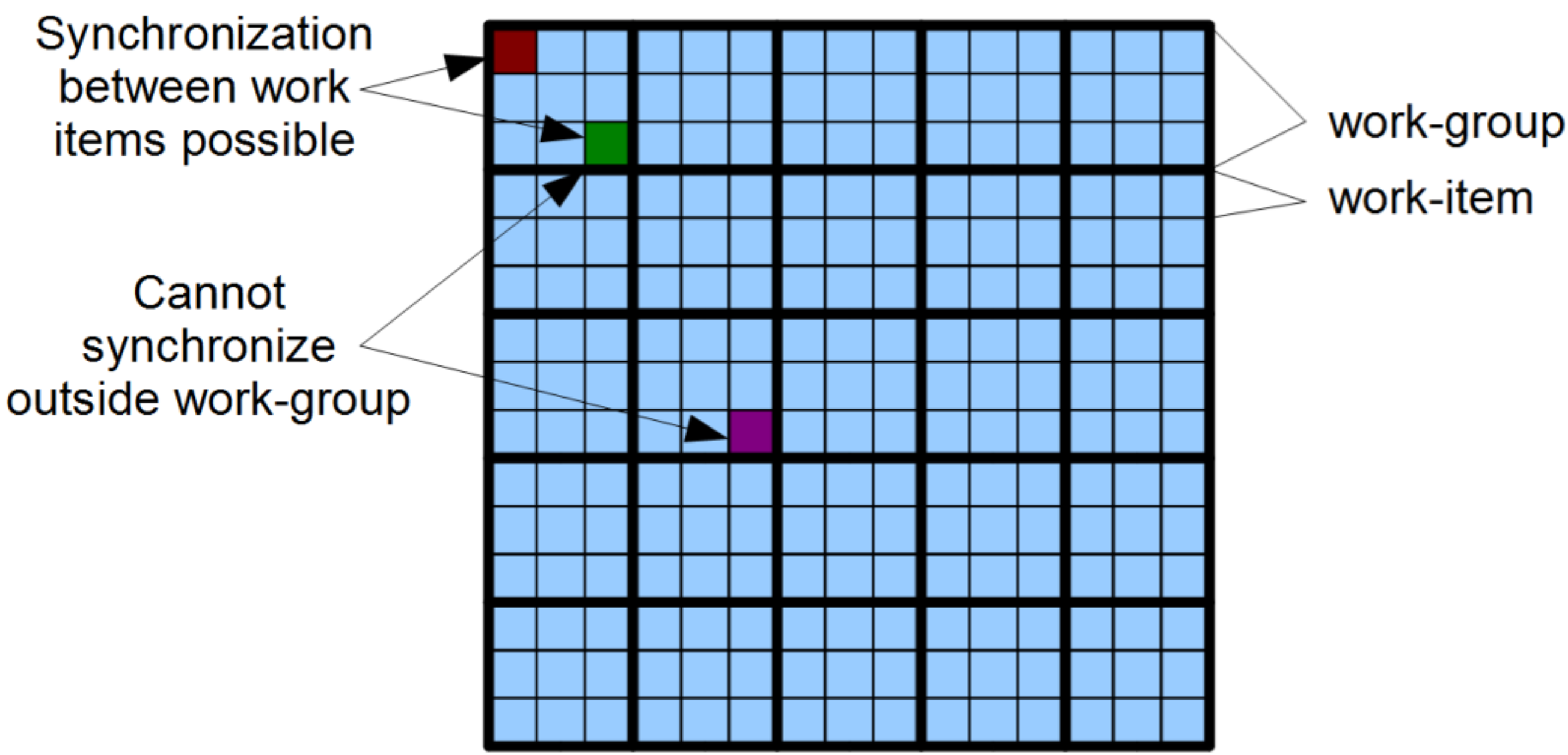
OpenCL Memory Model



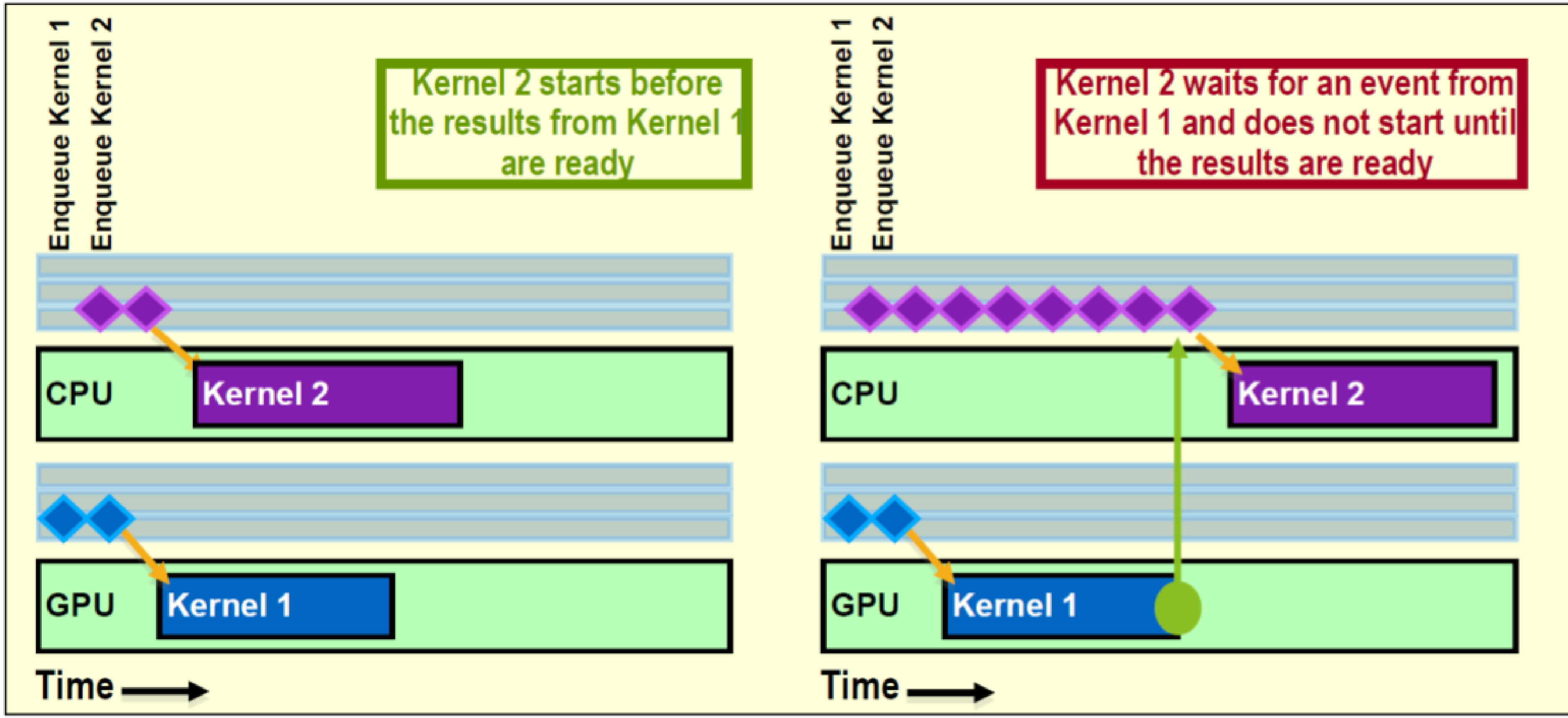
2D Data-Parallel Execution in OpenCL



OpenCL Work-group / Work-unit Structure



Concurrency Control with OpenCL Event-Queueing



*Functions executed on an OpenCL device are called kernels

OpenCL's Two Styles of Data Parallelism

- Explicit SIMD data parallelism
  - The kernel defines one stream of instructions
  - Parallelism from using wide vector types
  - Size vector types to match native HW width
  - Combine with task parallelism to exploit multiple cores
- Implicit SIMD data parallelism (i.e. shader-style)
  - Write the kernel as a "scalar program"
  - Use vector data types sized naturally to the algorithm
  - Kernel automatically mapped to SIMD-compute-resources and cores by the compiler/runtime/hardware

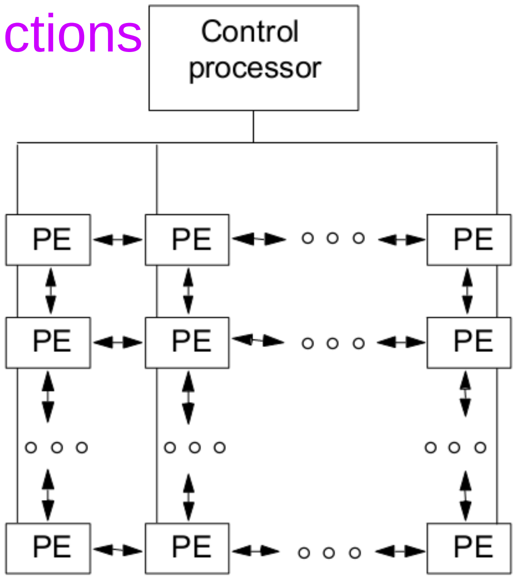
Both approaches are viable CPU options

Parallel programming models

Data Parallel Systems

Data Parallel Systems

- **Programming model**
  - Operations performed in parallel on each element of data structure
  - Logically single thread of control, performs sequential or parallel steps
  - Conceptually, a processor associated with each data element
- **Architectural model**
  - Array of many simple, cheap processors with little memory each
    - Processors don't sequence through instructions
  - Attached to a control processor that issues instructions
  - Specialized and general communication, cheap global synchronization
- **Original motivation**
  - Matches simple differential equation solvers
  - Centralize high cost of instruction fetch & sequencing



Application of Data Parallelism

- Example
  - Each PE contains an employee record with his/her salary

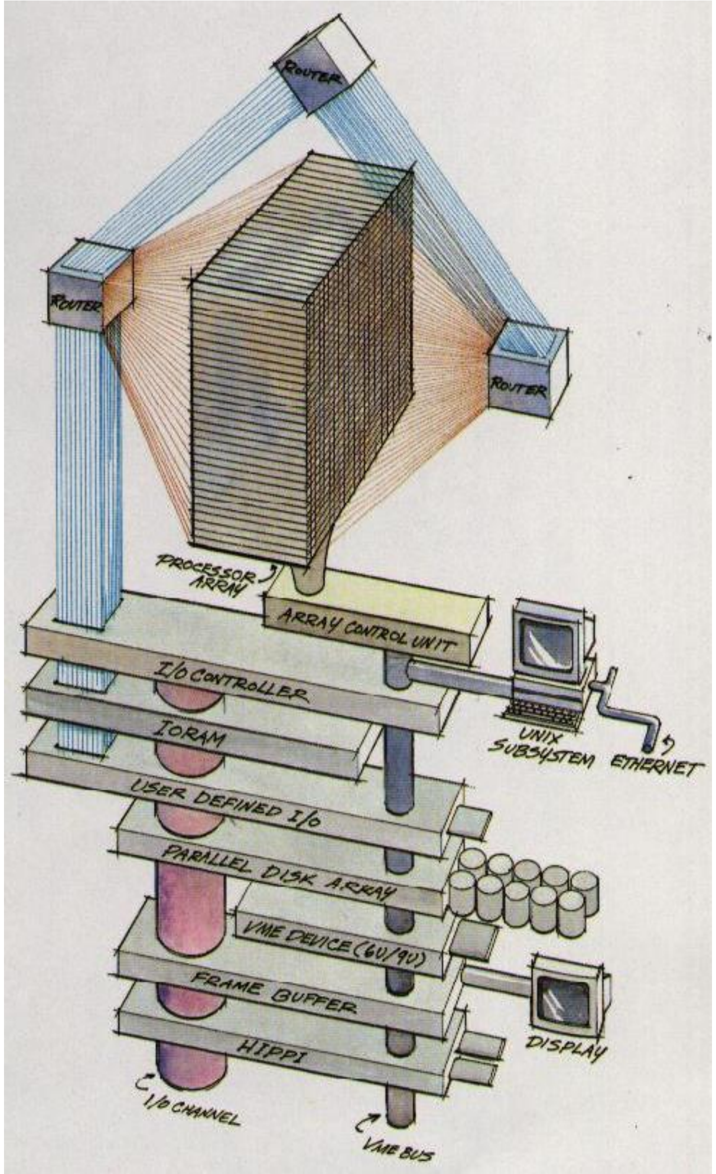
```
If salary > 100K then
      salary = salary *1.05
else
      salary = salary *1.10
```

  - Logically, the whole operation is a single step
  - Some processors enabled for arithmetic operation, others disabled

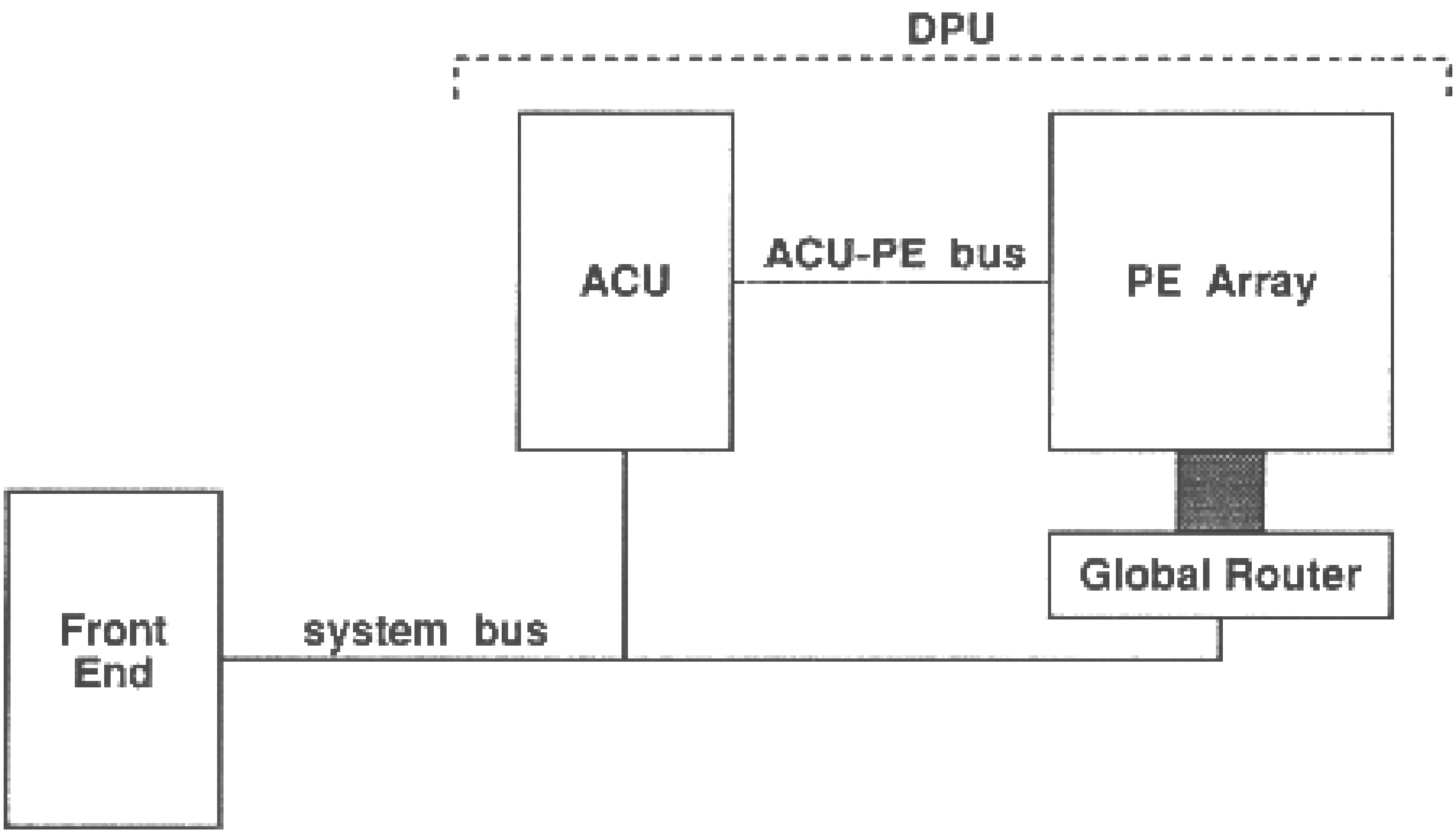
- Other examples
  - Finite differences, linear algebra, ...
  - Document searching, graphics, image processing, ...

- Example machines
  - Thinking Machines CM-1, CM-2 (and CM-5)
  - Maspar MP-1 and MP-2

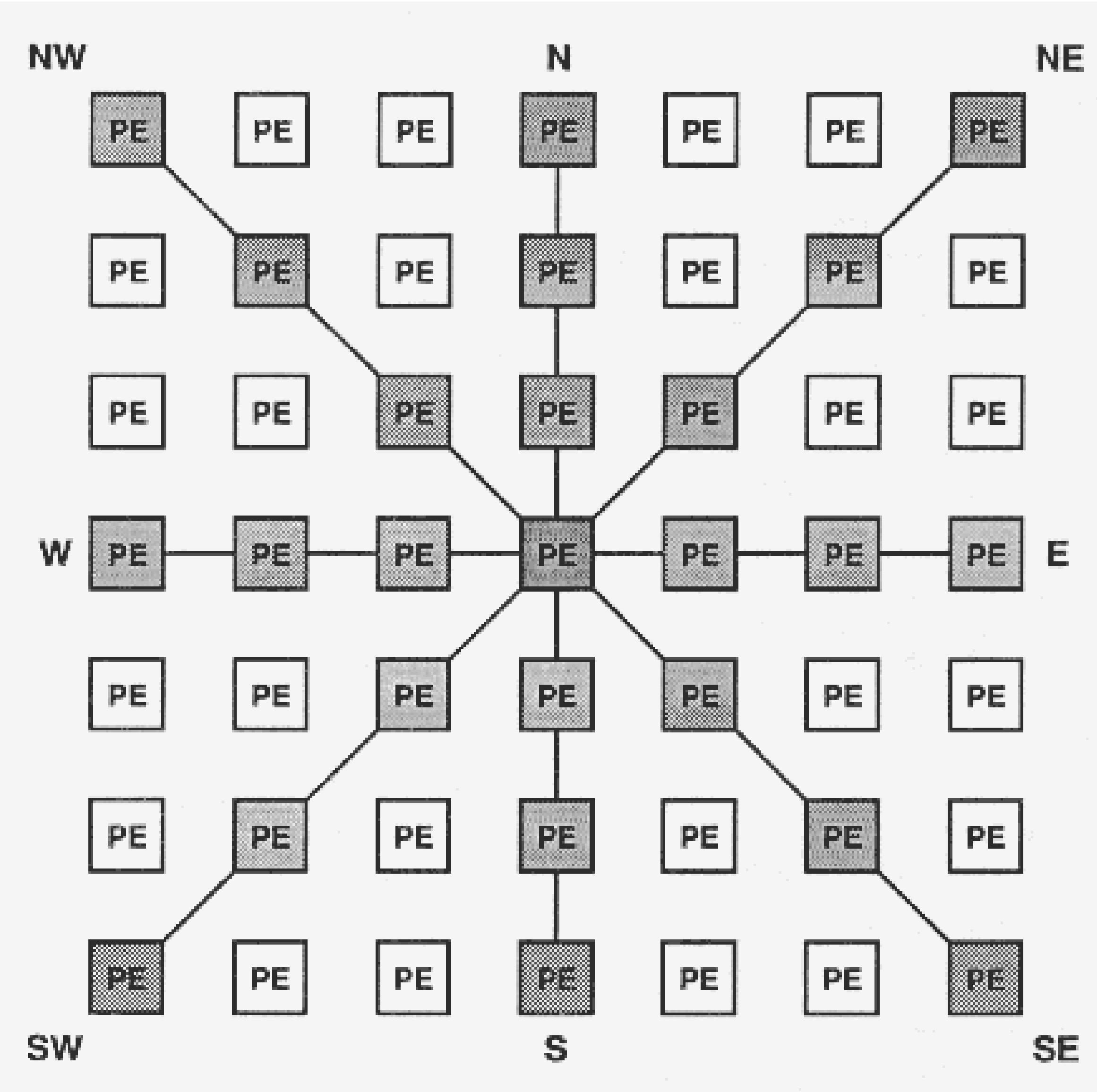
Maspar MP Architecture

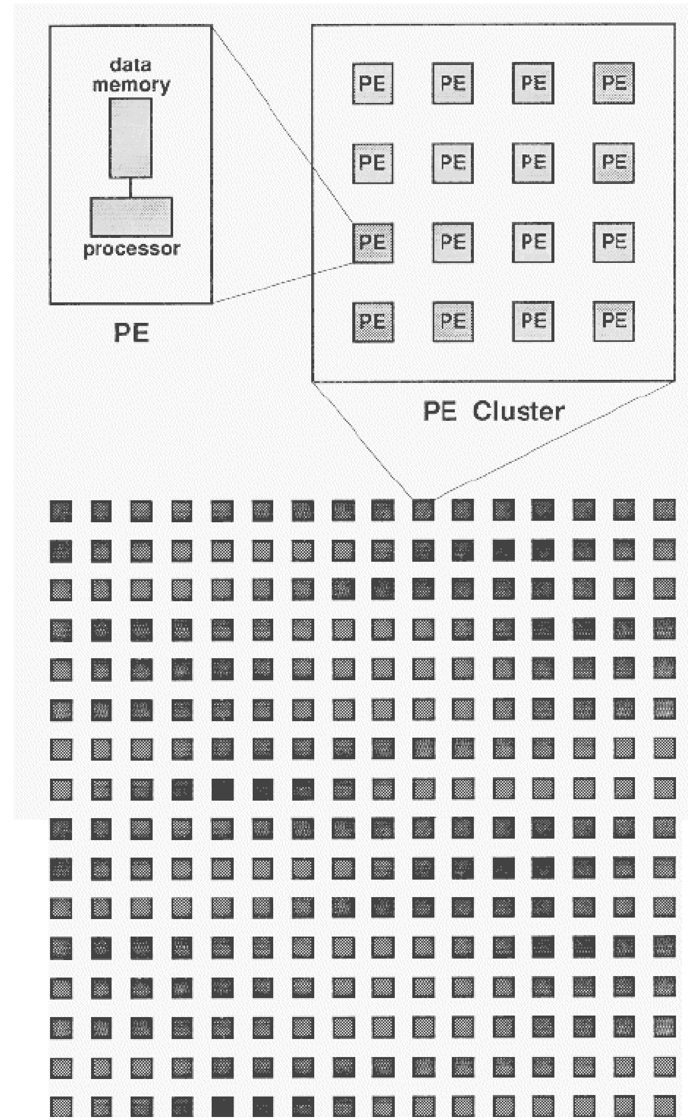


Maspar MP Architecture



Maspar MP Architecture





Dataflow Architecture

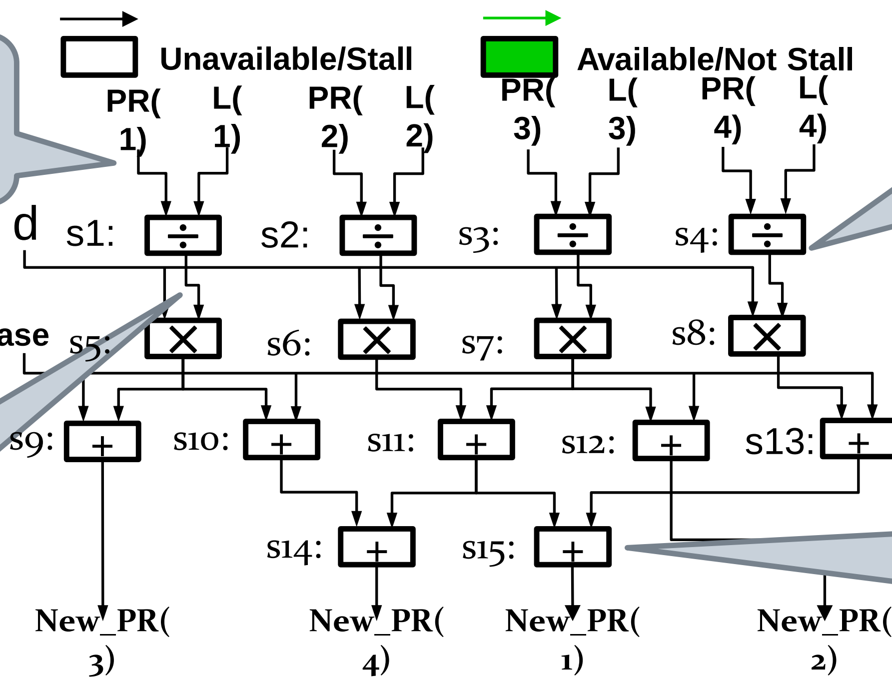
- Non-von Neumann models of computation, architecture, and languages
- Programs are not attached to a program counter
- Executability and execution of instructions is solely determined based on the availability of input arguments to the instructions
- Order of instruction execution is unpredictable: i. e. behavior is indeterministic
- Static and Dynamic dataflow machines
  - Static dataflow machines: use conventional memory addresses as data dependency tags
  - Dynamic dataflow machines: use content-addressable memory (CAM)

Dataflow Execution Model

A node is active if all input tokens are available

Substantial parallelism: the schedule and execution only depends on the availability of source operands

Node s1 would be active once two tokens of *PR*(1) and *L*(1) are available



The retirement of node s4 would not be interrupted by node s5 since they are independent

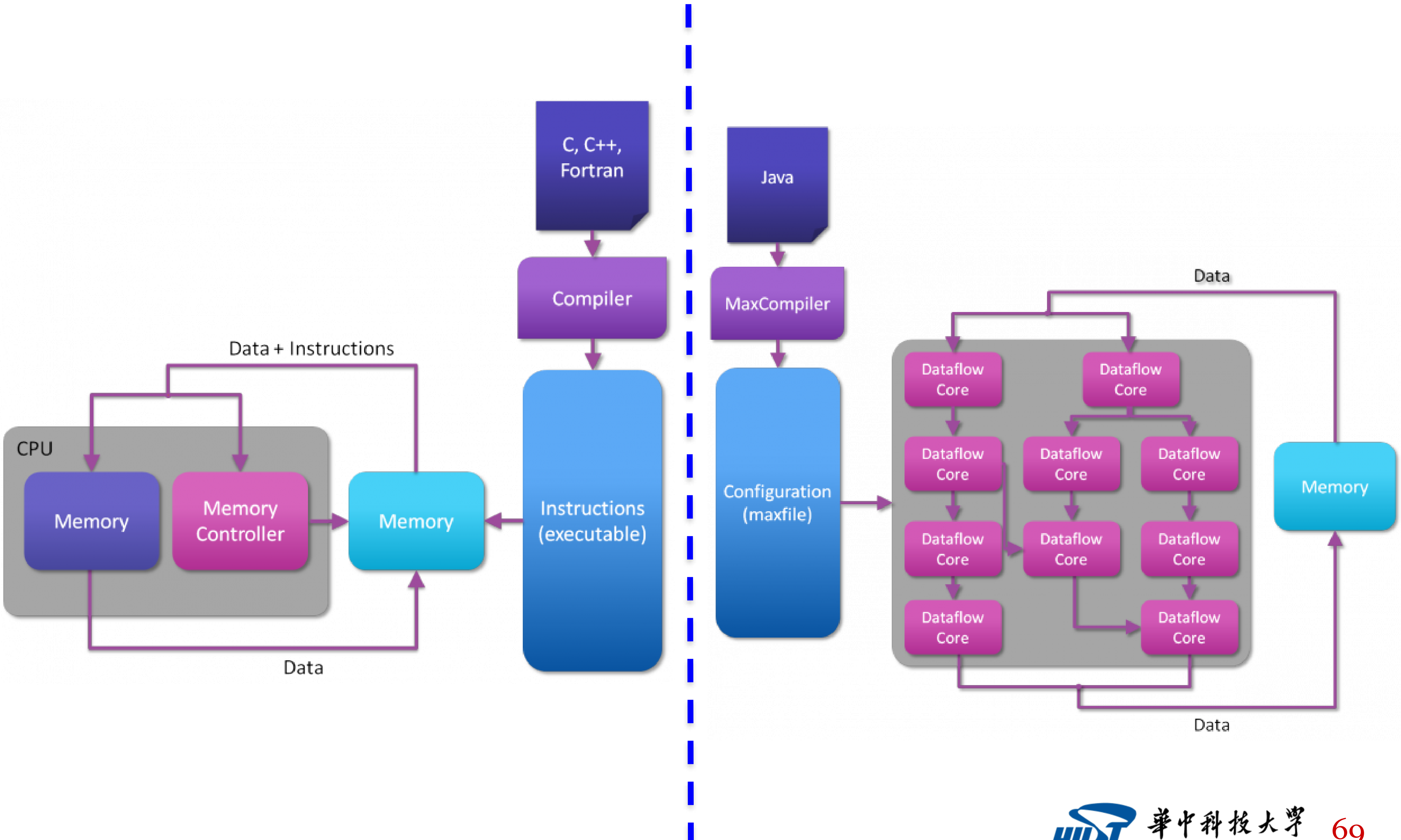
Once active, it would push the new token $d \times (PR(1)/L(1))$ to node s5

The result of node s15 is consistent even if the execution order of node s11 and s12 changes

Once active, the node would produce new tokens for output arcs

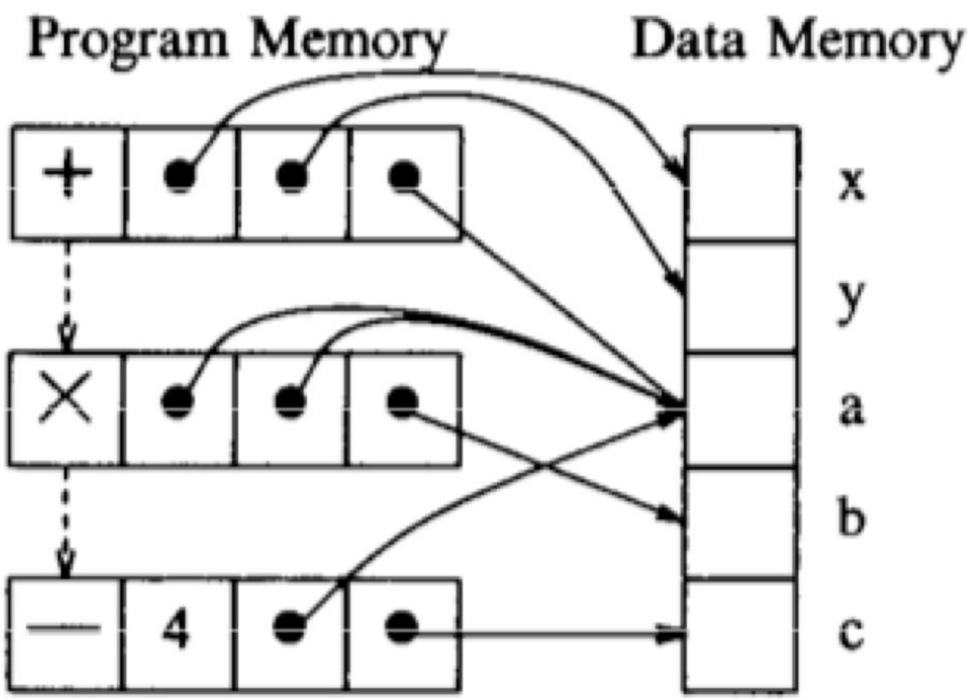
Strong determinacy: computing results are not subject to the execution order of independent nodes

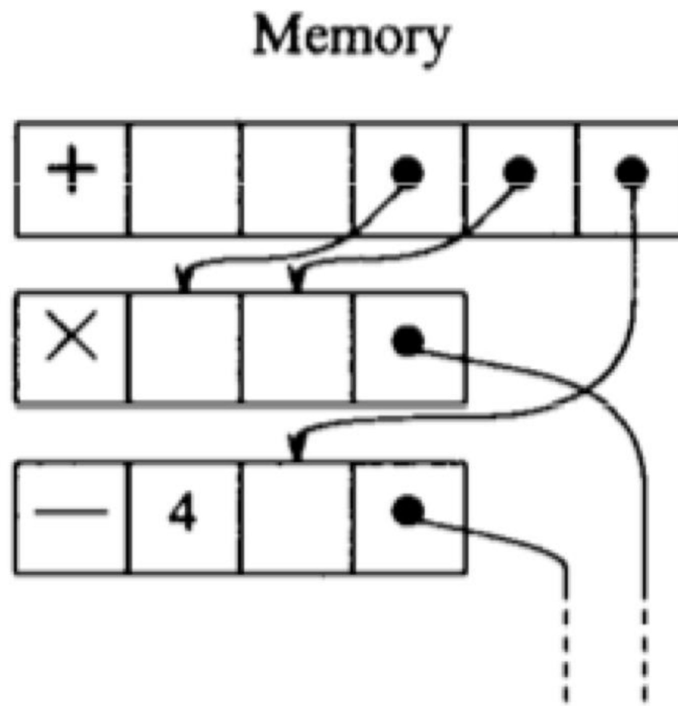
Computing with Control Flow/ Data Flow Cores



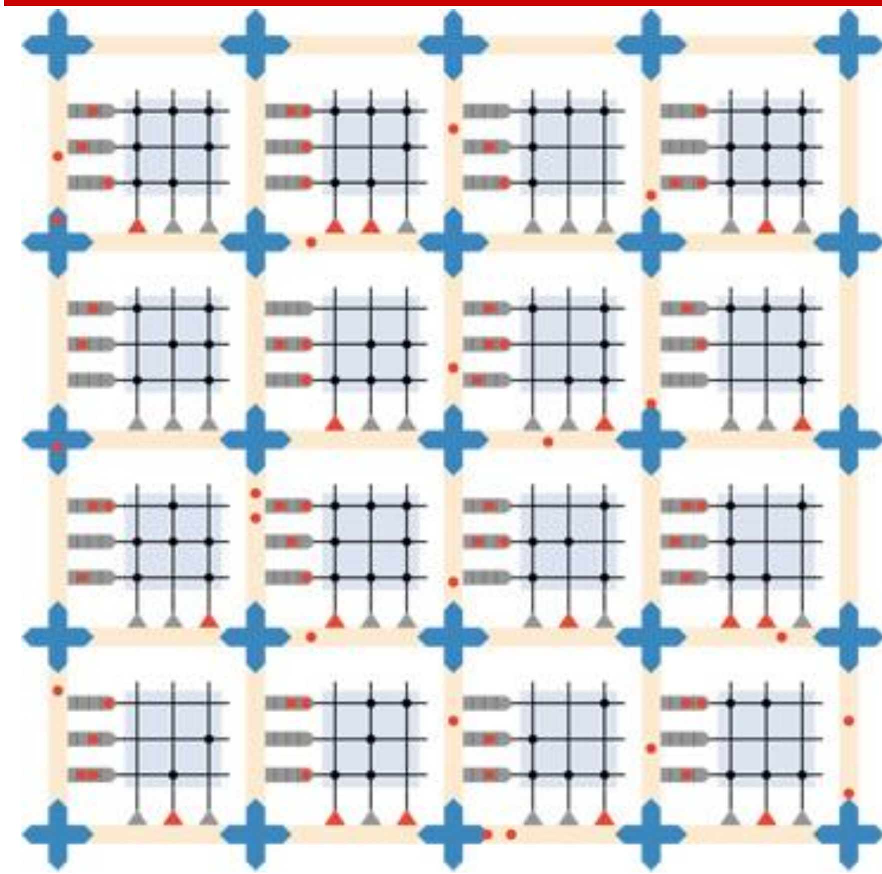
Control Flow vs. Data Flow

$$\begin{aligned} a &:= x + y \\ b &:= a \times a \\ c &:= 4 - a \end{aligned}$$

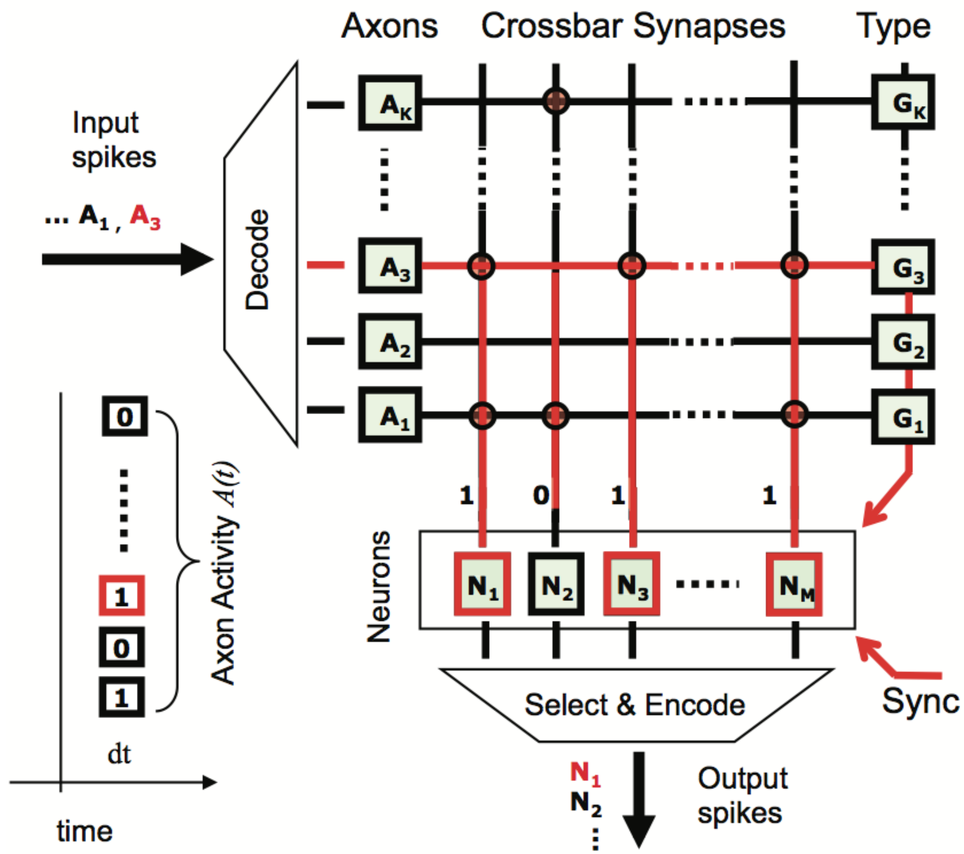




IBM's Brain-Inspired Architecture



5.4 billion transistors, and an on-chip network of 4,096 neurosynaptic cores only consumes 70mW during real-time operation



Evolution and Convergence

- Rigid control structure (SIMD in Flynn taxonomy)
  - SISD = uniprocessor , MIMD= multiprocessor
- Popular when cost savings of centralized sequencer high
  - 60s when CPU was a cabinet; replaced by vectors in mid-70s
  - Revived in mid-80s when 32-bit data path slices just fit on chip
  - No longer true with modern microprocessors
- Other reasons for demise
  - Simple, regular applications have good locality, can do well anyway
  - Loss of applicability due to hardwiring data parallelism
    - MIMD machines as effective for data parallelism and more general
- Programming model converges with SPMD (single program multiple data)
  - Contributes need for fast global synchronization
  - Structured global address space, implemented with either SAS or MP

References

- The content expressed in this chapter comes from
  - Livermore Computing Center’s training materials, (https://computing.llnl.gov/tutorials/parallel_comp/)
  - Carnegie Mellon University’s public course, Parallel Computer Architecture and Programming, (CS 418) (http://www.cs.cmu.edu/afs/cs/academic/class/15418-s11/public/lectures/)
  - Carnegie Mellon University’s public course, Computer Architecture, (CS 740) (http://www.cs.cmu.edu/afs/cs/academic/class/15740-s11/public/lectures/)