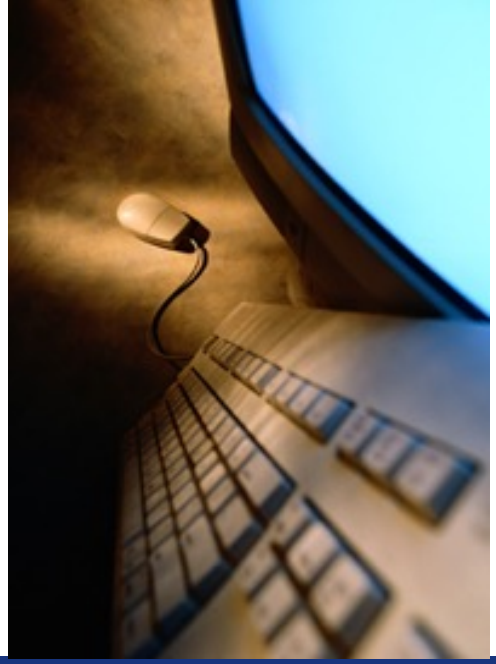
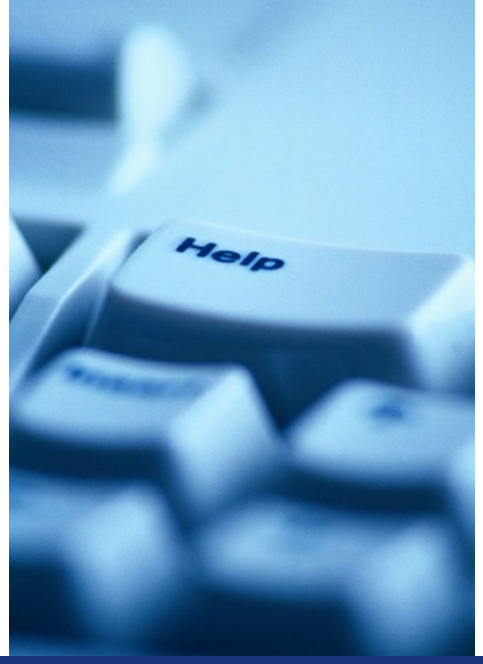
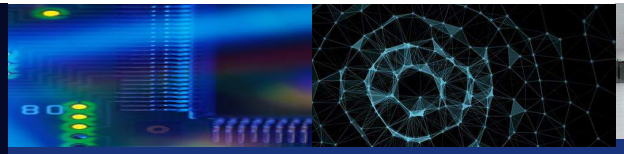



并行与分布式计算

Parallel & Distributed Computing

陈鹏飞
数据科学与计算机学院
2021-09-24

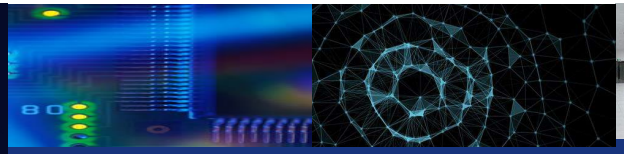


Lecture 3 — Parallel Programming Model

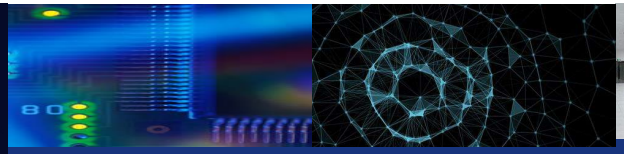
Pengfei Chen

School of Data and Computer Science

Sep 24, 2021



Outline:



Parallel programming models

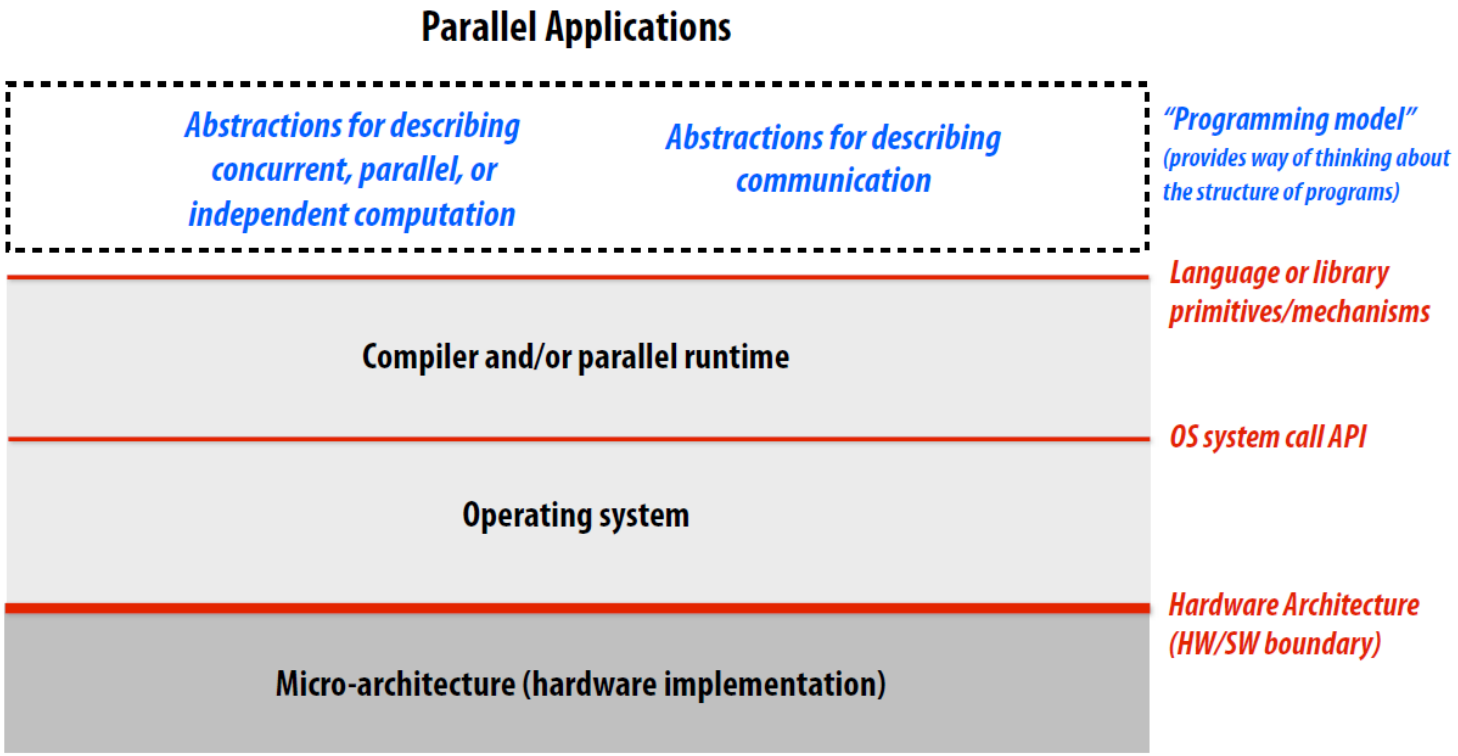
INTRODUCTION

Critical Points

Abstraction VS Implementation

Critical Points

System layers: interface, implementation, interface, ...

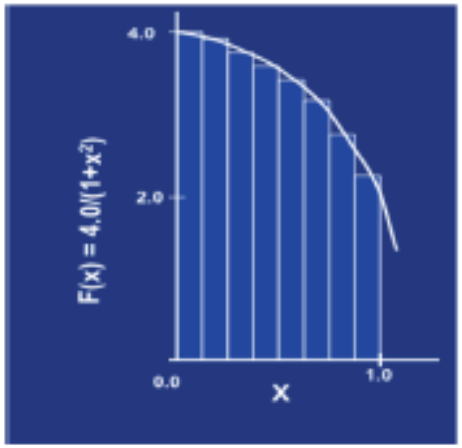


Blue italic text: abstraction/concept
Red italic text: system interface
Black text: system implementation

Examples

- **Computing π**

$$\left.\begin{array}{c} arctan(1) = \pi/4 \\ arctan(0) = 0 \\ \frac{d}{dx} arctan(x) = 1/(1+x^2) \end{array}\right\} \Rightarrow \pi = \int_0^1 \frac{4}{1+x^2} dx$$

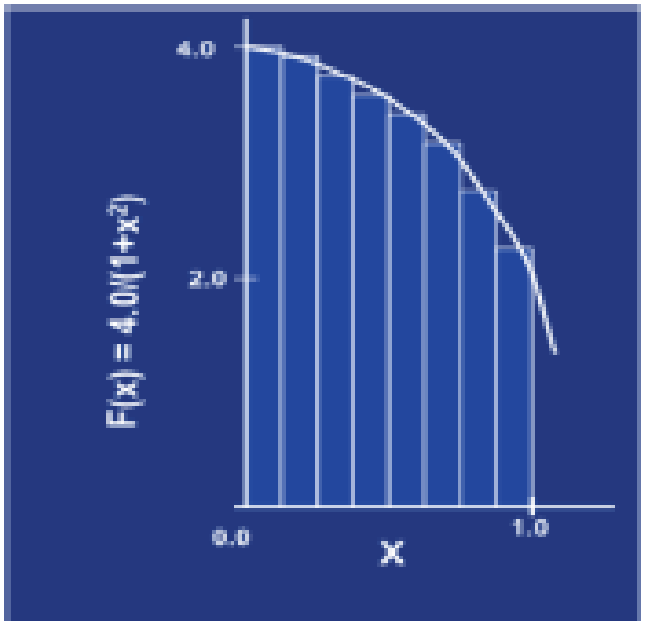


Barbara Chapman, “A Guide to OpenMP,” 2010.

Examples

- **Computing π： *Sequential Code***

```
double compute_pi(int n) {
double sum = 0.0;
for (int i = 0; i < n; i++) {
    double x = (i + 0.5) / n;
    sum += 1.0 / (1.0 + x*x);
}
    double pi = 4.0 * sum / n;
    return pi;
}
```



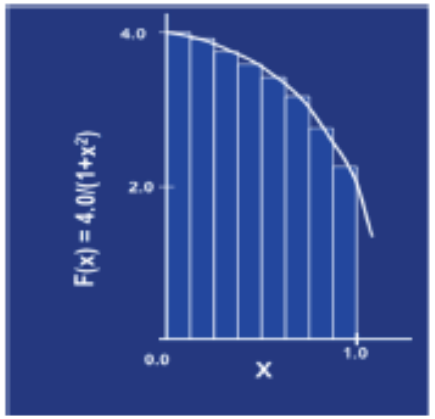
Examples

➢ Computing π: *POSIX Thread Version*

```
// global variables
int thread_count;
int n;
double* local_sum;

// multithreaded version
double compute_pi () {
  thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));
  local_sum = (double*) malloc (thread_count*sizeof(double));
  // parallel part
  for (thread = 0; thread < thread_count; thread++)
    pthread_create(&thread_handles[thread], NULL, thread_sum, (void*)thread);
  for (thread = 0; thread < thread_count; thread++)
    pthread_join(thread_handles[thread], NULL);
  // sequential part
  double sum = 0.0;
  for (thread = 0; thread < thread_count; thread++)
   sum += local_sum[thread];
  double pi = 4.0 * sum / n;
  return pi;
}
```

```
void* thread_sum(void* rank) {
  int my_rank = (int) rank;
  double my_sum = 0.0;
  // domain decomposition
  for (int i = my_rank; i < n; i += thread_count) {
    double x = (i + 0.5) / n;
    my_sum += 1.0 / (1.0 + x * x);
  }
  local_sum[my_rank] = my_sum;
  return NULL;
}
```

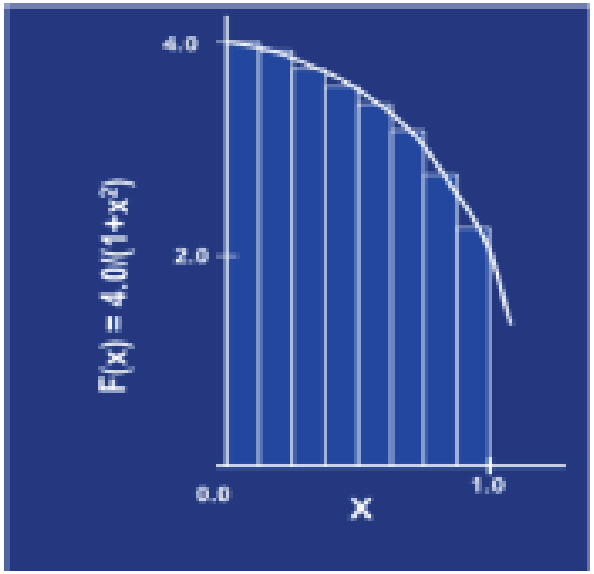


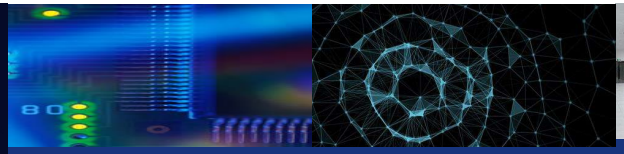
A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Examples

- **Computing π: *OpenMP Version***

```
/*  compile as $> gcc -fopenmp -lm
 *  run as        $> OMP_NUM_THREADS=2 ./a.out
 */
double compute_pi(int n) {
  int i;
  double sum = 0.0;
#pragma omp parallel for reduction(+: sum) schedule(static)
  for (i = 0; i < n; ++i) {
    double x = (i + 0.5) / n;
    sum += 1.0 / (1.0 + x * x);
  }
  double pi = 4.0 * sum / n;
  return pi;
}
```





Examples

➢ Computing π: *MPI Version*

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] ) {
int n, my_rank, numprocs, i;
double mypi, pi, h, sum, x;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
while (1) {
  if (my_rank == 0) {
     printf("Enter the number of intervals: (0 quits) ");
     scanf("%d",&n);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n == 0) break;
else {
sum = 0.0;
for (i = my_rank; i <; i += numprocs) {
      x = (i + 0.5) / n;
      sum += 4.0 / (1.0 + x*x);
}
  mypi = sum / n;
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
  MPI_COMM_WORLD);
  if (my_rank == 0) printf("pi is approximately %.16f\n", pi));
}
}
  MPI_Finalize();
  return 0;
}
```

Examples

➢ Computing π: *OpenCL Version*

```
/* to be executed on deivce */
__kernel void pi(const int niters, const float step_size,
__local float* local_sums, __global float* partial_sums) {
int num_wrk_items = get_local_size(0);
int local_id = get_local_id(0);
int group_id = get_group_id(0);
float x, accum = 0.0f;
int i,istart,iend;
istart = (group_id * num_wrk_items + local_id) * niters;
iend = istart+niters;
for(i= istart; i<iend; i++) {
    x = (i+0.5f)*step_size;
    accum += 4.0f/(1.0f+x*x);
}
local_sums[local_id] = accum;
barrier(CLK_LOCAL_MEM_FENCE);
reduce(local_sums, partial_sums);
}
/* to be executed on host */
pi_res = 0.0f;
for (unsigned int i = 0; i < nwork_groups; i++) { pi_res += h_psum[i]; }
pi_res *= step_size;
```

Examples

- Computing π: *Summary*

Different parallel granularity

Different parallel implementation (implicit VS explicit)

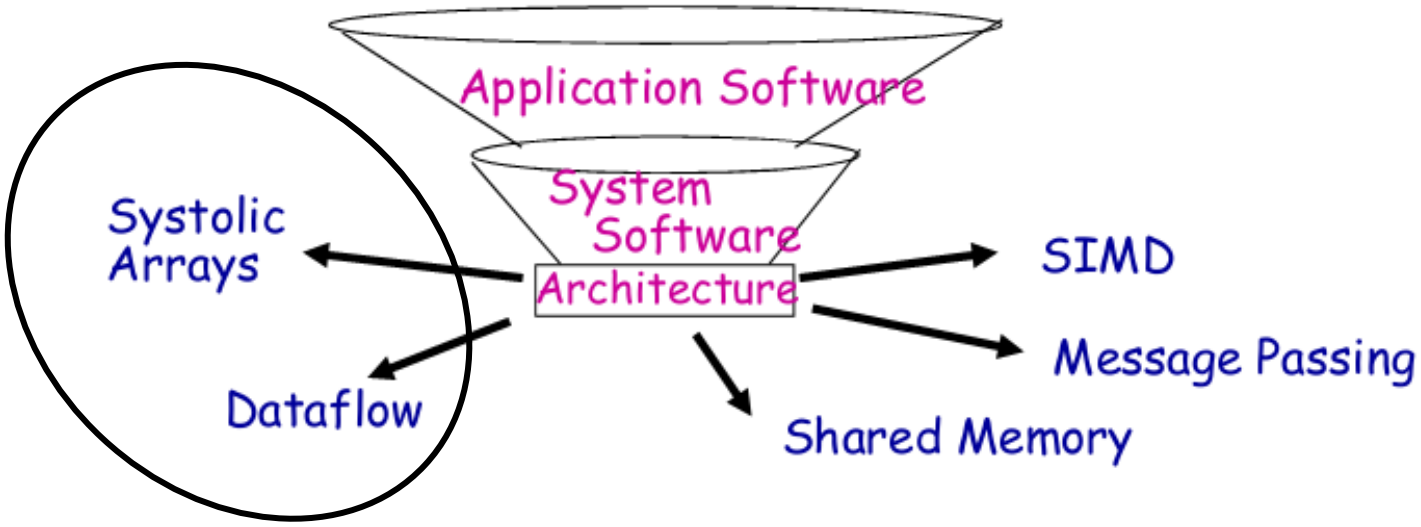
Different code scale

Different hardware architecture

Different what?

History

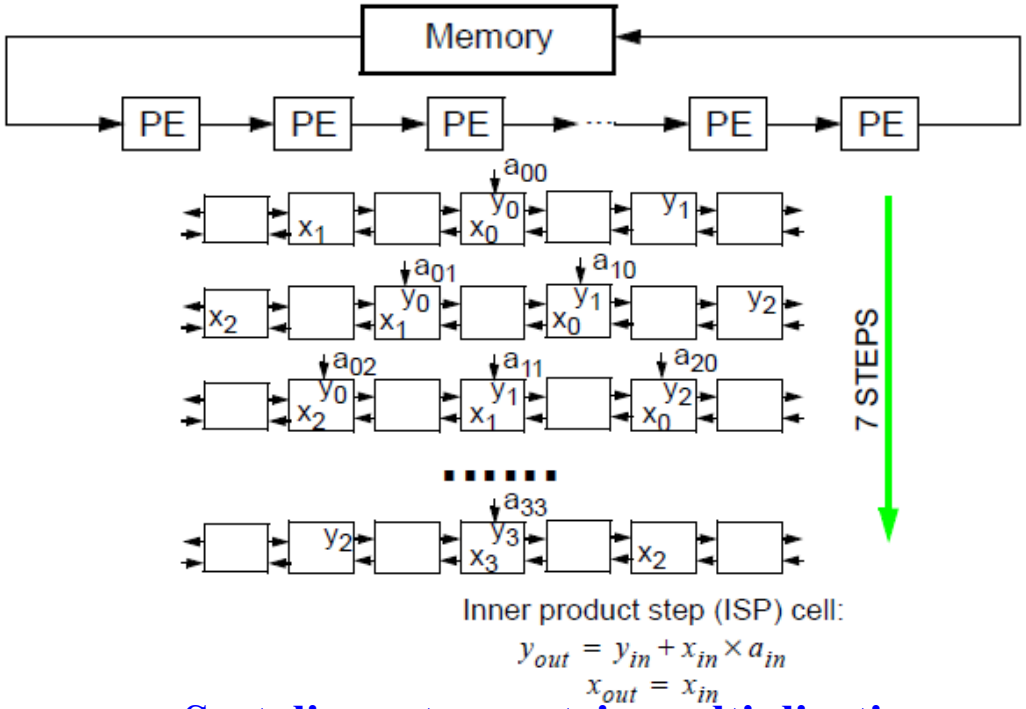
- **Historically (1970s – early 1990s), each parallel machine was unique, along with its programming model and language**
  - Architecture = prog. model + comm. abstraction + machine;
  - parallel architectures tied to programming models;
- **Divergent architectures, with no predictable pattern of growth**



Uncertainty of direction paralyzed parallel software development!

Systolic Array

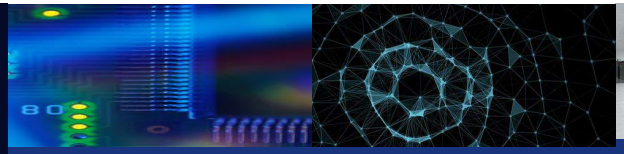
- **Types of special-purpose computers:**
  - Inflexible and highly dedicated structures
  - Structures, enabling some programmability and reconfiguration.
  - Data would move through the system at regular “heartbeats” as determined by local state.



Systolic vector-matrix multiplication

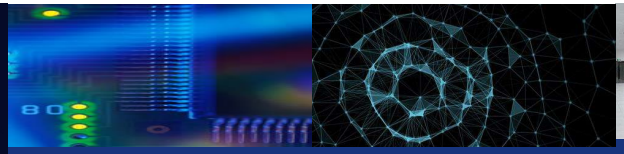
Today

- **Nowadays we separate the programming model from the underlying parallel machine architecture**
  - Dominant: shared address space, message passing, data parallel
  - Others: data flow, systolic arrays
- **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**



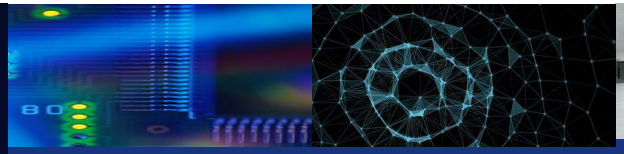
Programming Model

- **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 Model

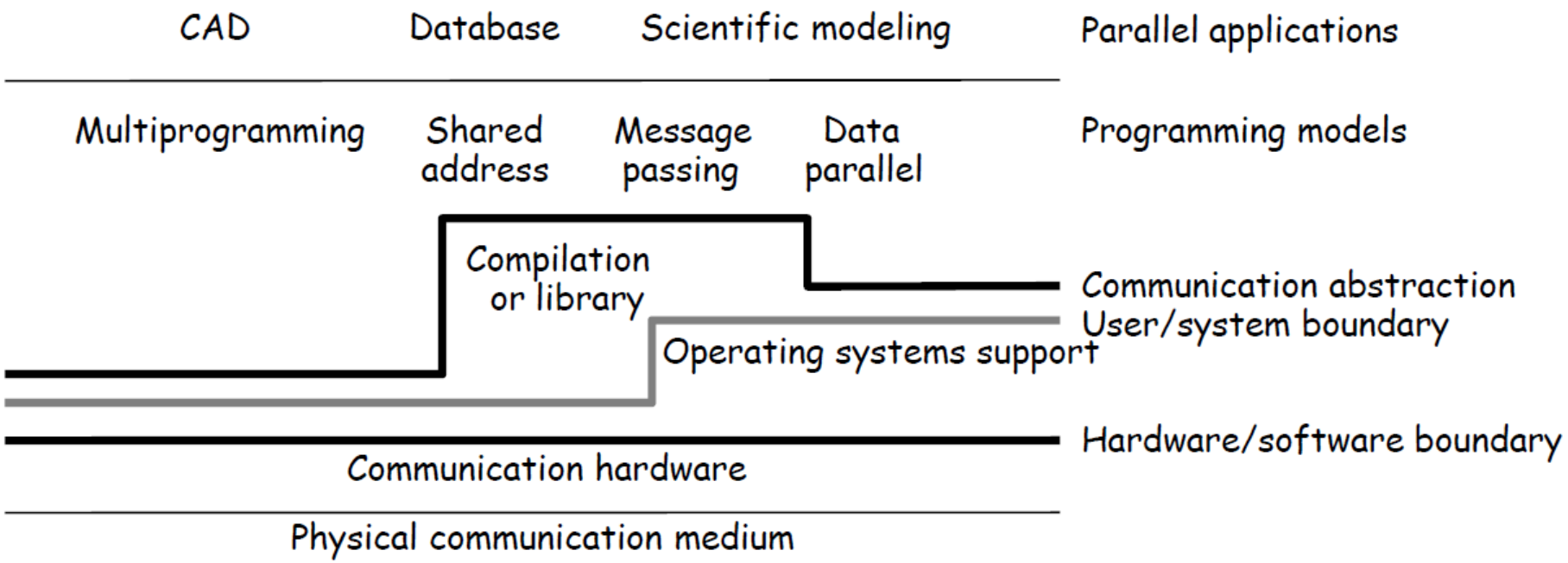
- **von Neumann model**
  - Execute a stream of instructions (machine code)
  - Instructions can specify
    - Arithmetic operations
    - Data addresses
    - Next instruction to execute
  - Complexity
    - Track billions of data locations and millions of instructions
    - Manage with
      - ✓ Modular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



Programming Model

- **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”

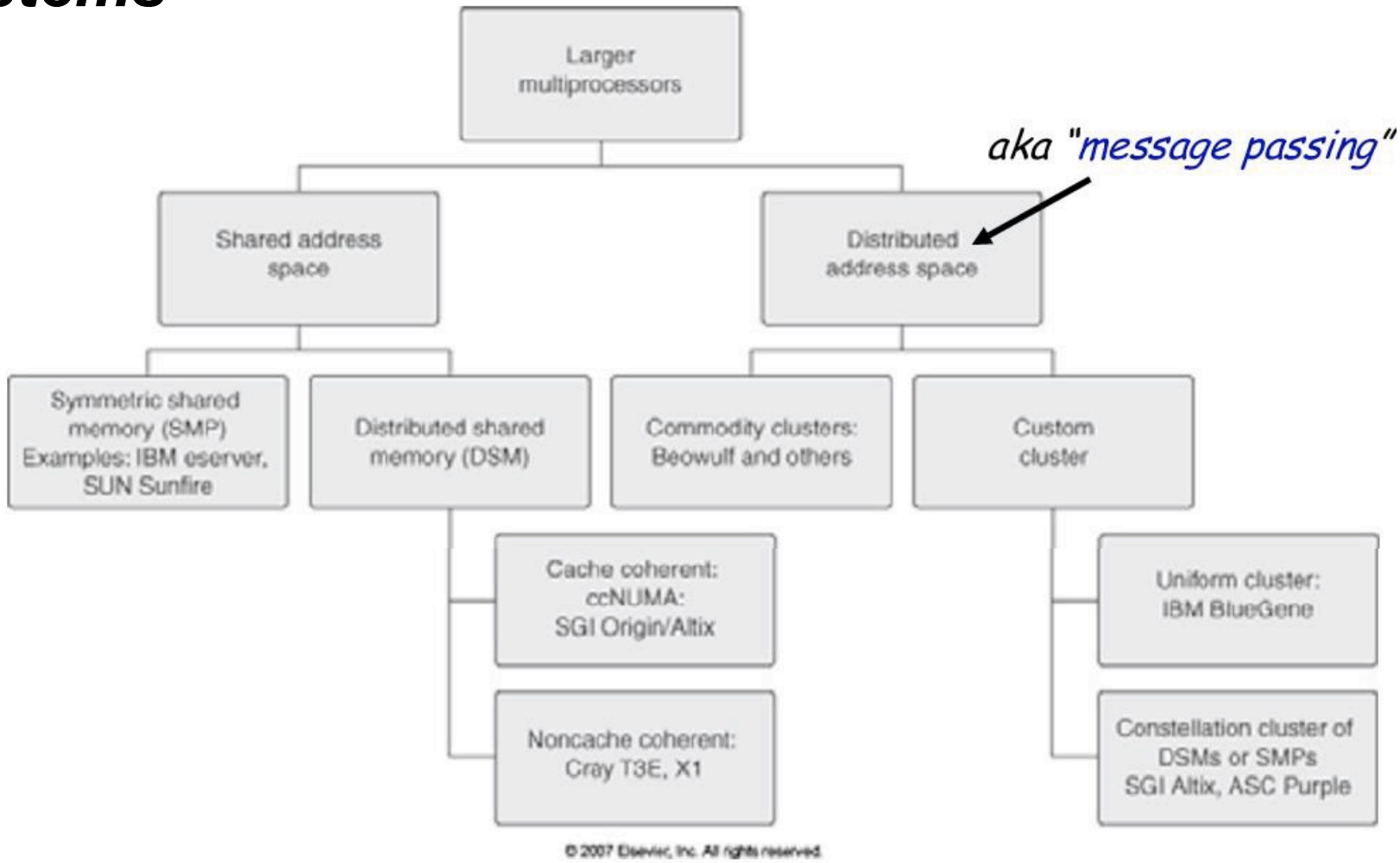
Modern Layered Framework

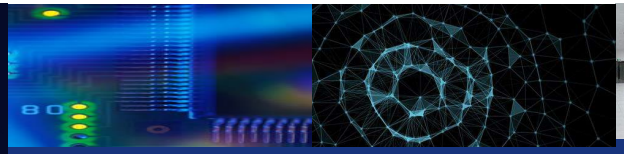


Evolution of Architectural Models

- **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

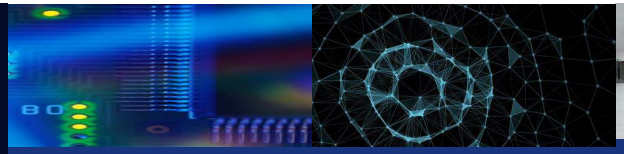
Taxonomy of Common Large-Scale SAS and MP Systems





Aspects of a Parallel Programming Model

- **Control**
  - How is parallelism created?
  - In what order should operations take place?
  - How are different threads of control synchronized?
- **Naming**
  - What data is private vs. shared?
  - How is shared data accessed?
- **Operations**
  - What operations are atomic?
- **Cost**
  - How do we account for the cost of operations?



Parallel programming models

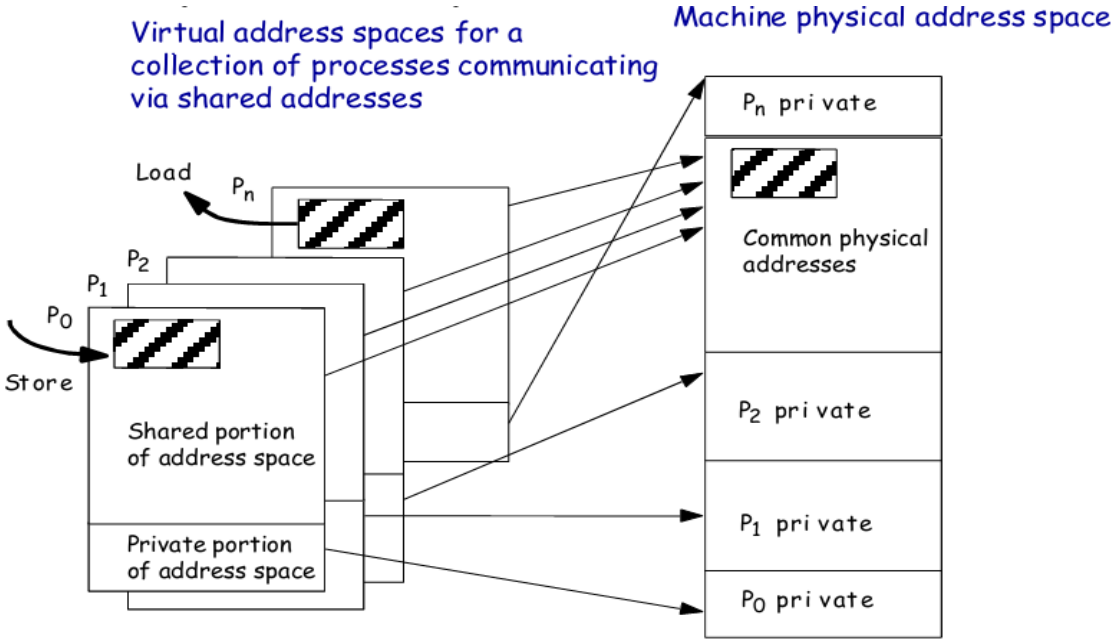
SHARED MEMORY MODEL

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 (abstraction)

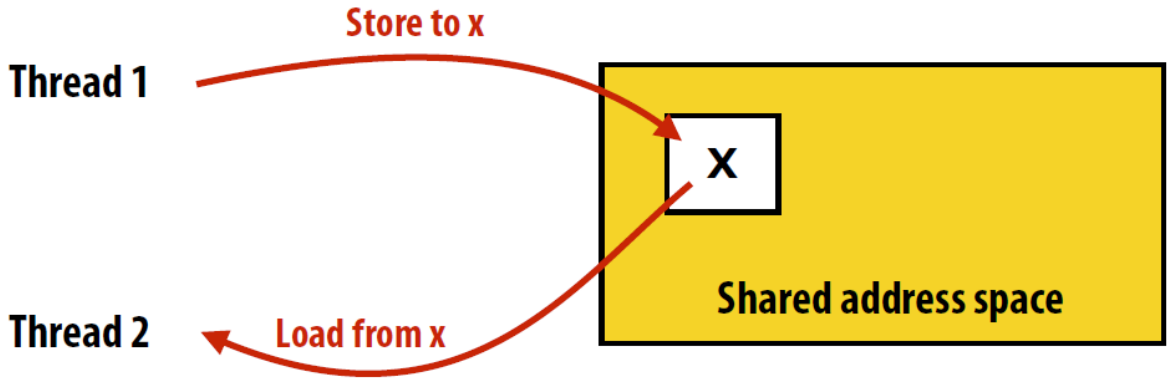
- Threads communicate by reading/writing to shared variables
- Shared variables are like a big bulletin board

Thread 1:

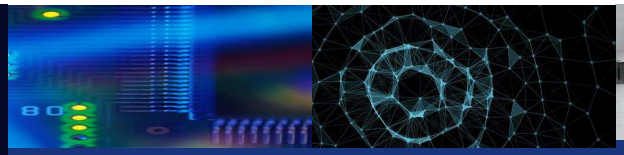
```
int x = 0;
spawn_thread(foo, &x);
x = 1;
```

Thread 2:

```
void foo(int* x) {
  while (x == 0) {}
  print x;
}
```



(Communication operations shown in red)



Shared Memory Model (abstraction)

Synchronization primitives are also shared variables: e.g., locks

Thread 1:

```
int x = 0;
Lock my_lock;

spawn_thread(foo, &x, &my_lock);


mylock.lock();
x++;
mylock.unlock();
```

Thread 2:

```
void foo(int* x, lock* my_lock)
{
  my_lock->lock();
  x++;
  my_lock->unlock();

  print x;
}
```

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

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

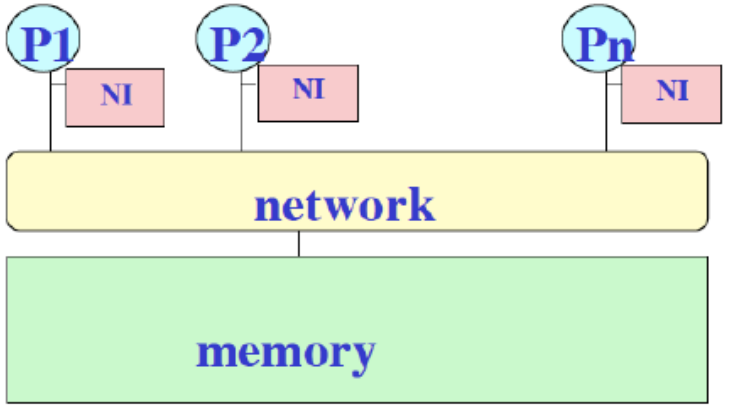
Implementations

- Native compilers and/or hardware translate user program variables into actual memory addresses, which are global
  - On stand-alone SMP machines, this is straightforward

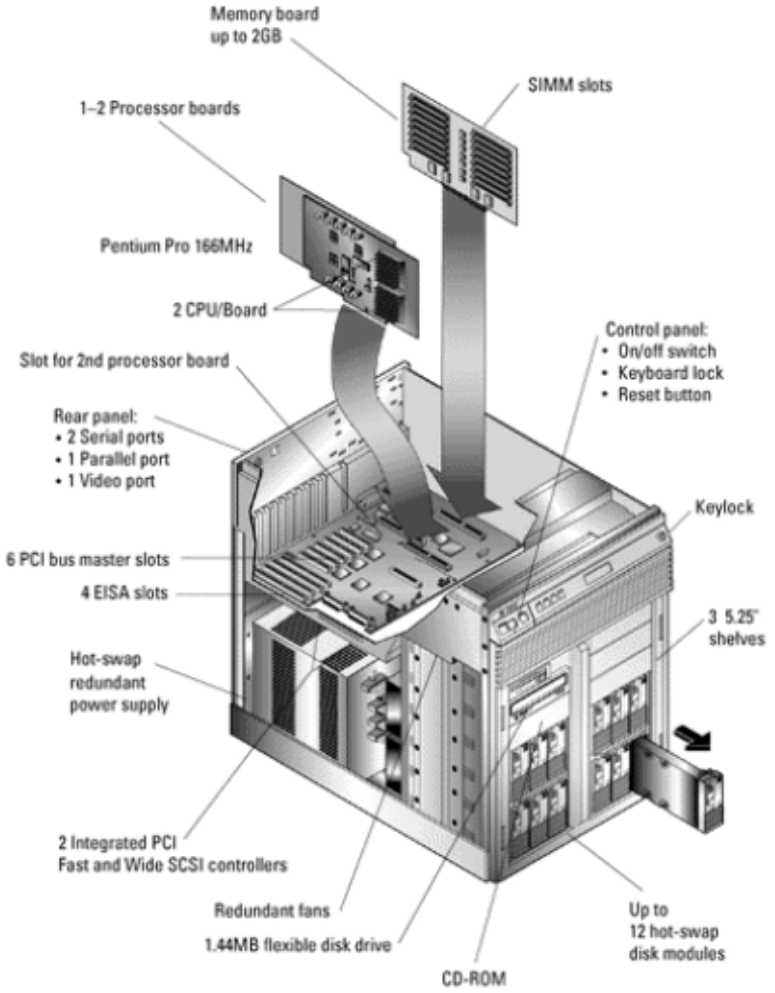
- On distributed shared memory machines, such as the SGI Origin, memory is physically distributed across a network of machines, but made global through specialized hardware and software

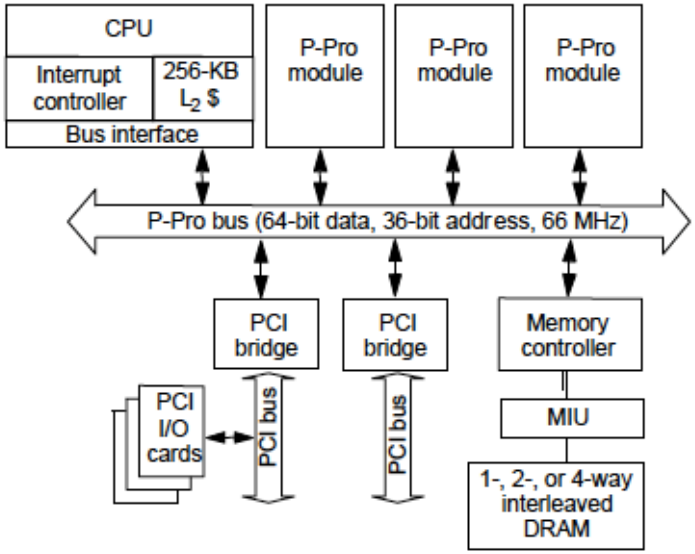
SAS Machine Architecture

- **One representative architecture: SMP**
  - **Used to mean *Symmetric MultiProcessor***
    - All CPUs had equal capabilities in every area, e.g., in terms of I/O as well as memory access
  - **Evolved to mean *Shared Memory Processor***
    - Non-message-passing machines (included crossbar as well as bus based systems)
  - **Now it tends to refer *to bus-based shared memory machines***
    - Small scale: < 32 processors typically



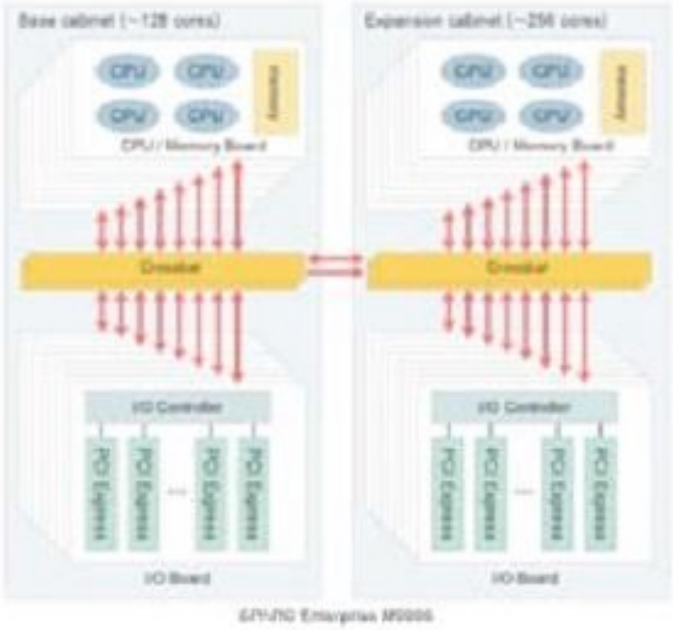
SAS Example: Intel Pentium Pro Quad





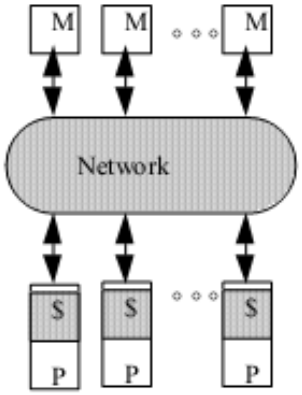
- All coherence and multiprocessing glue in processor module
- Highly integrated, targeted at high volume
- Low latency and high bandwidth

SAS Example: Sun SPARC Enterprise M9000

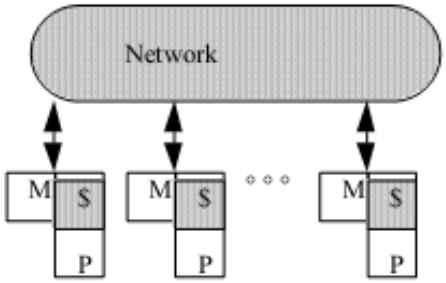


- 64 SPARC64 VII+ quad-core processors (i.e. 256 cores)
- Crossbar bandwidth: 245 GB/sec (snoop bandwidth)
- Memory latency: 437-532 nsec (i.e. 1050-1277 cycles @ 2.4 GHz)
- Higher bandwidth, but also higher latency

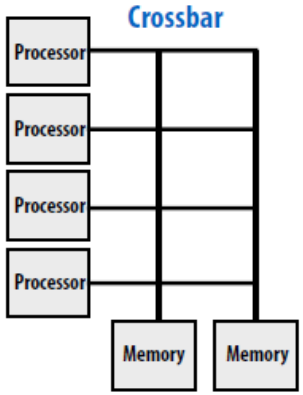
Scaling Up

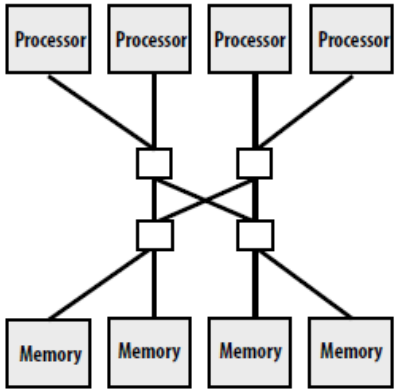


"Dance hall"



Distributed memory



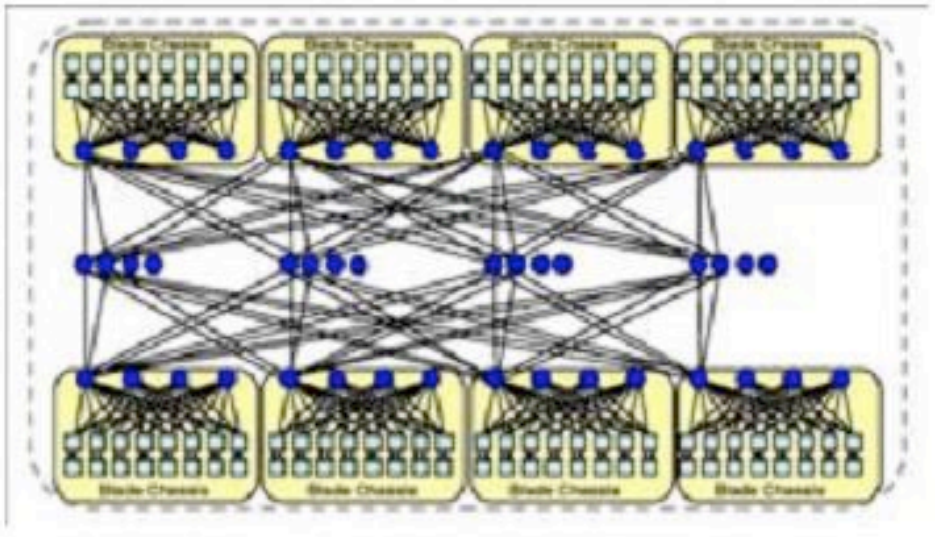


Multi-stage network

- 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)

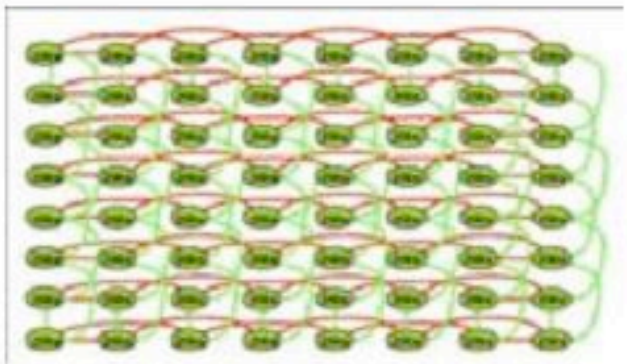
Example: SGI Altix UV 1000

Blacklight at the PSC (4096 cores)



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

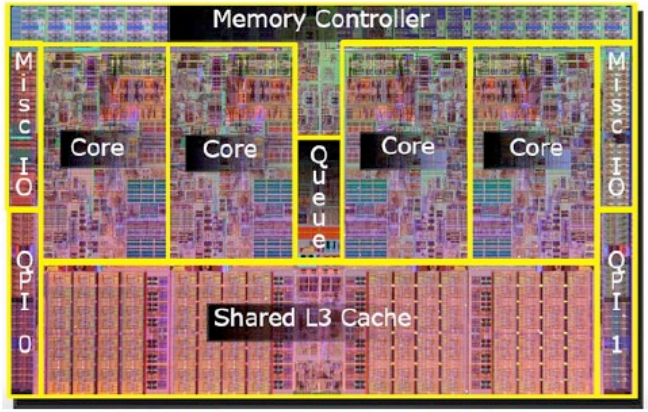
- Scales up to 131,072 cores
- 15GB/sec links
- Hardware cache coherence



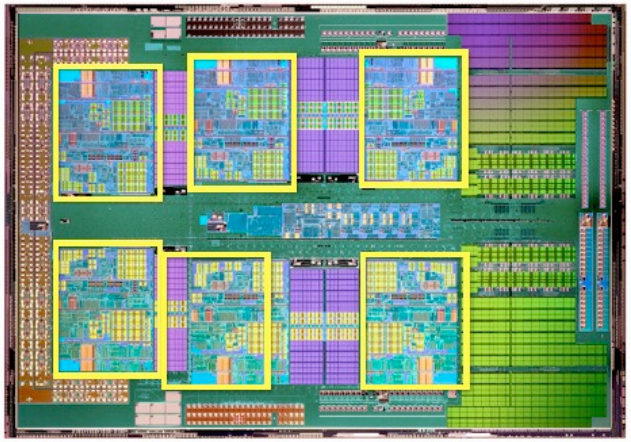
8x8 torus

Shared address space HW achitectures

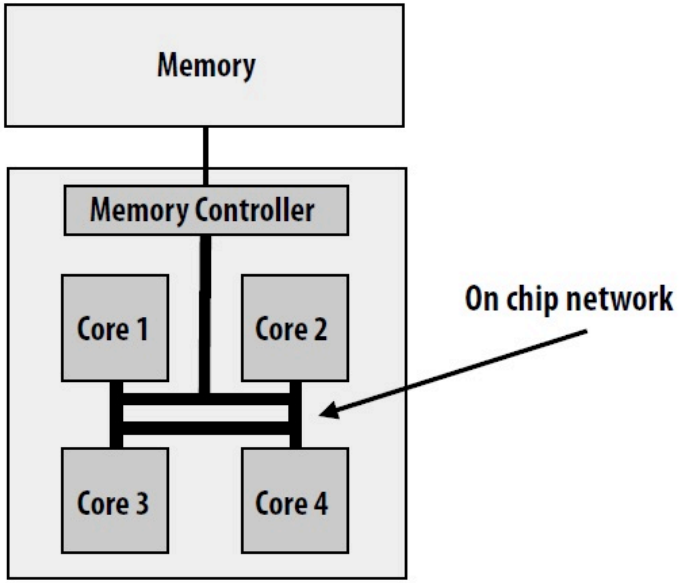
Commodity x86 examples



Intel Core i7 (quad core)
(interconnect is a ring)

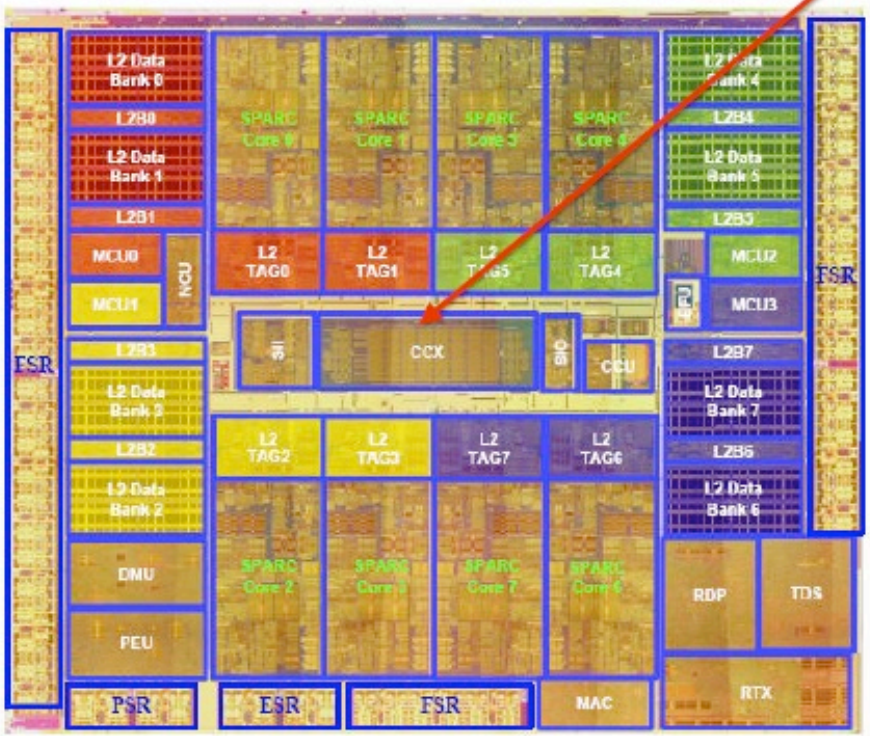


AMD Phenom II (six core)

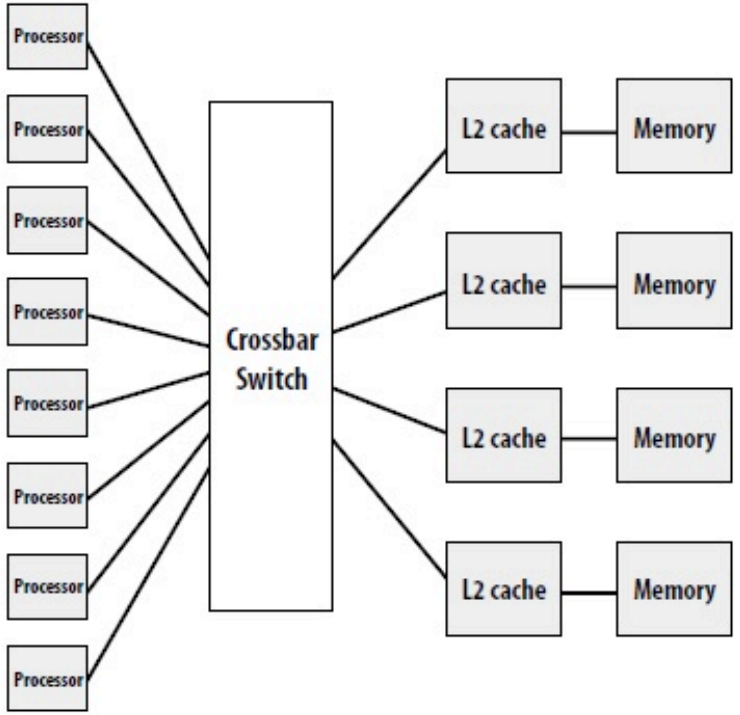


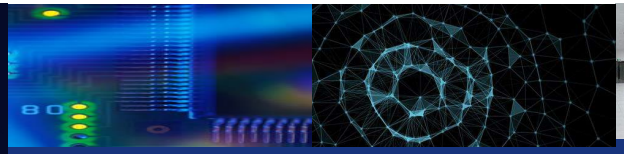
Shared address space HW achitectures

Note area of crossbar: about die area of one core



Eight cores





Parallel programming models

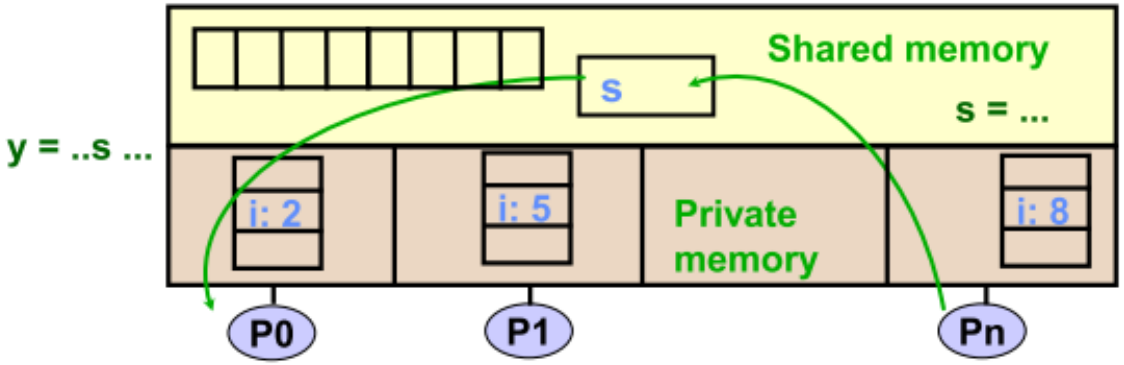
THREAD MODEL

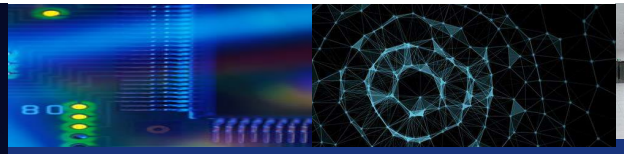
Threads Model

- 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, 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





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)

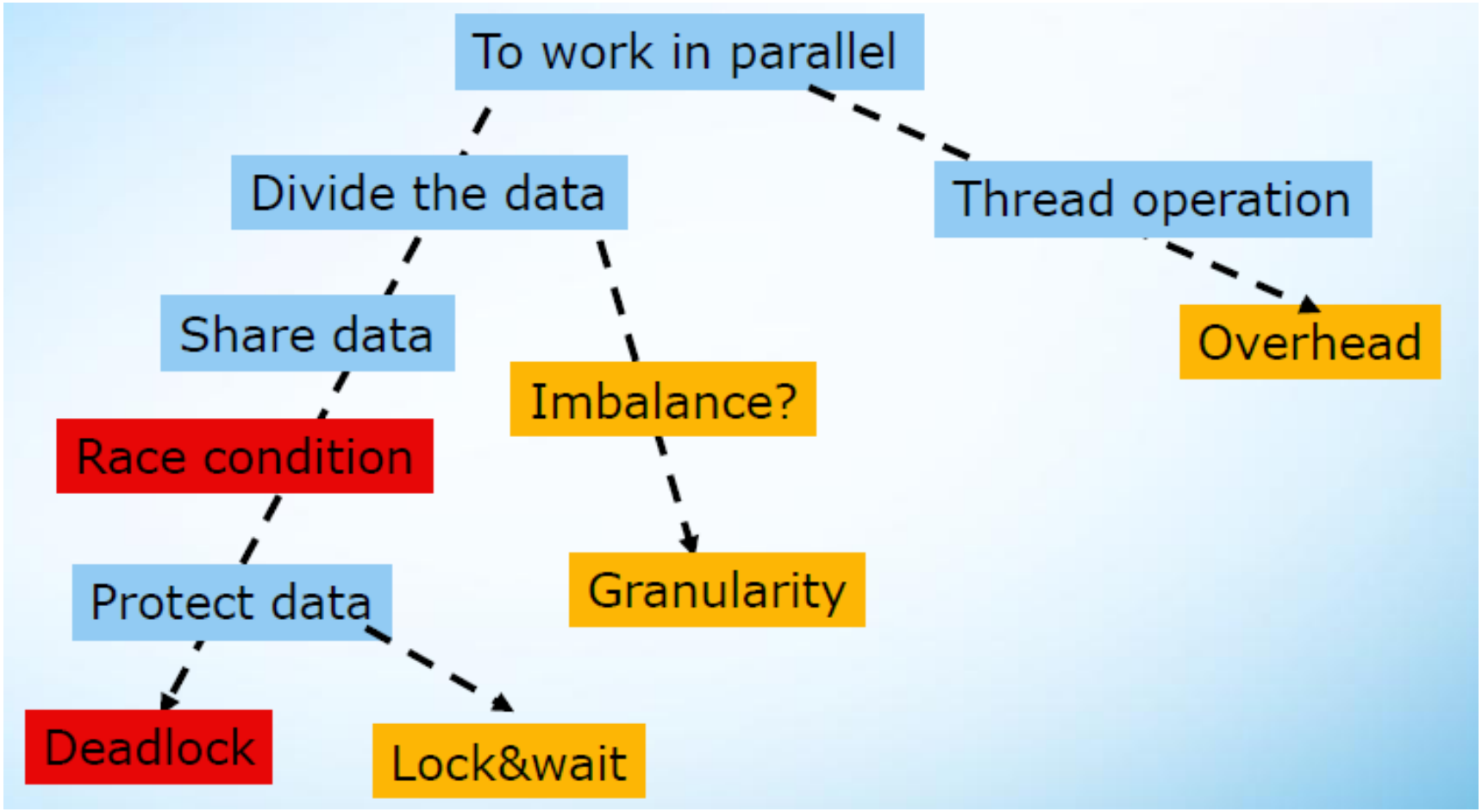
$$Speedup_{Amdahl} = \frac{1}{(1-f) + \frac{f}{n}}$$

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

f is the parallel portion

Implications

Where Are the Problems From?



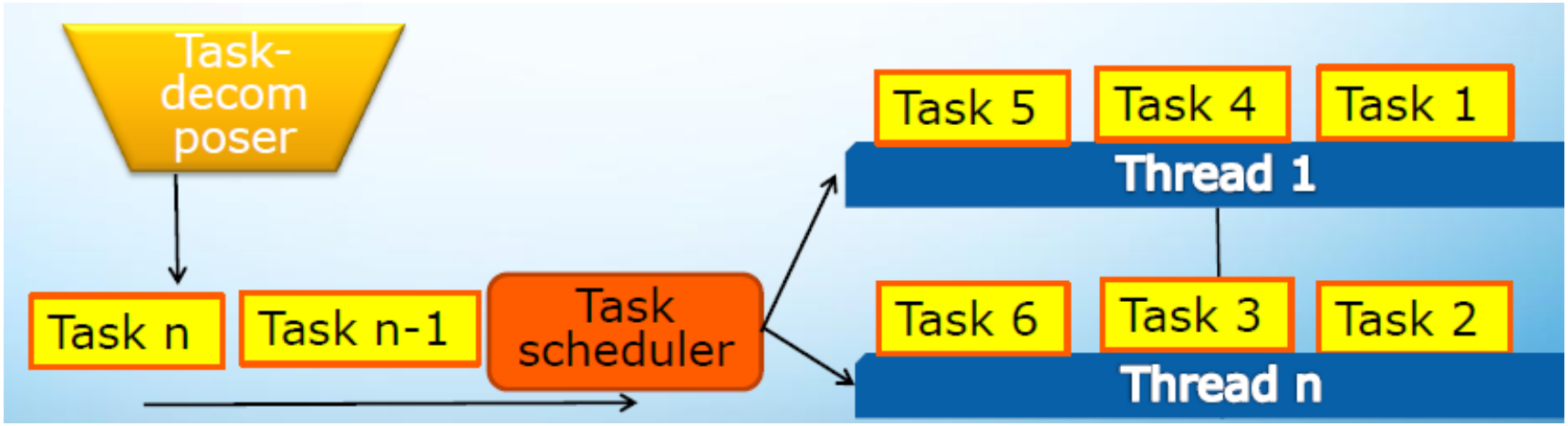
Remove the error **Tune for high speedup**

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（交响乐）
- **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

Race Conditions

A **determinacy race** occurs when two parallel strands access the same memory location and at least one strand performs a write operation. The program result depends on which strand "wins the race" and accesses the memory first.

A **data race** is a special case of a determinacy race. A data race is a race condition that occurs when two parallel strands, holding no locks in common, access the same memory location and at least one strand performs a write operation. The program result depends on which strand "wins the race" and accesses the memory first.

If the parallel accesses are protected by locks, there is no data race. However, a program using locks may not produce deterministic results. A lock can ensure consistency by protecting a data structure from being visible in an intermediate state during an update, but does not guarantee deterministic results.

Deadlock

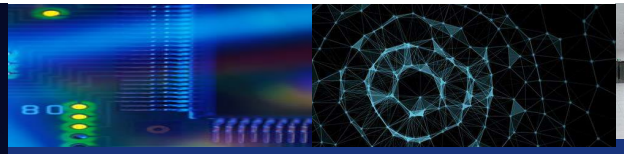
- 2 or more threads wait for each other to release a resource
- A thread waits for a event that never happen, like suspended lock
- Most common cause is locking hierarchies
- Considerations
  - Always lock and un-lock in the same order, and avoid hierarchies if possible
  - Use atomic

```
DWORD WINAPI threadA(LPVOID arg)
{
  EnterCriticalSection(&L1);
    EnterCriticalSection(&L2);
      processA(data1, data2);
    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);
}
```



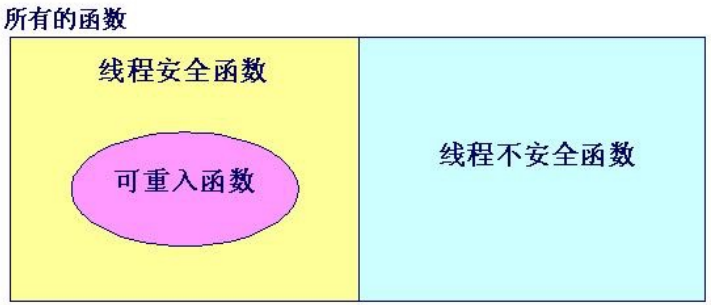
Thread Safe Routine/Library

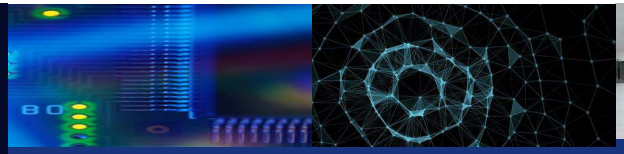
- It functions correctly during simultaneous execution by multiple threads
- Non-thread-safe indicators
  - Access global/static variables or the heap
  - Allocate/reallocate/free resources that have global scope (files)
  - Indirect accesses through handles and pointers
- Considerations
  - Any variables 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

Reentrant function

- 可重入：多个执行流反复执行一个代码，其结果不会发生改变，通常访问的都是各自的私有栈资源；
- 可重入函数：当一个执行流因为异常或者被内核切换而中断正在执行的函数而转为另外一个执行流时，当后者的执行流对同一个函数的操作并不影响前一个执行流恢复后执行函数产生的结果；
- 可重入函数满足的条件：
  - 不使用全局变量或静态变量；
  - 不使用malloc或者new开辟出的空间；
  - 不调用不可重入函数；
  - 不返回静态或全局数据，所有数据都由函数的调用者提供；
  - 使用本地数据，或者通过制作全局数据的本地拷贝来保护全局数据；
  - 不调用标准I/O；





Reentrant function

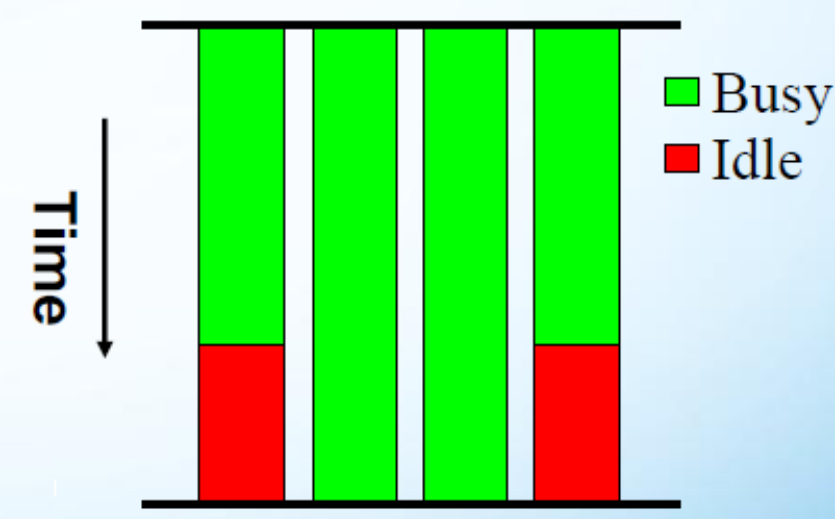
Linux可重复函数：

Demo

accept	fchmod	lseek	sendto	stat
access	fchown	lstat	setgid	symlink
aio_error	fcntl	mkdir	setpgid	sysconf
aio_return	fdatasync	mkfifo	setsid	tcdrain
aio_suspend	fork	open	setsockopt	tcflow
alarm	fpathconf	pathconf	setuid	tcflush
bind	fstat	pause	shutdown	tcgetattr
cfgetispeed	fsync	pipe	sigaction	tcgetpgrp
cfgetospeed	ftruncate	poll	sigaddset	tcsendbreak
cfsetispeed	getegid	posix_trace_event	sigdelset	tcsetattr
cfsetospeed	geteuid	pselect	sigemptyset	tcsetpgrp
chdir	getgid	raise	sigfillset	time
chmod	getgroups	read	sigismember	timer_getoverrun
chown	getpeername	readlink	signal	timer_gettime
clock_gettime	getpgrp	recv	sigpause	timer_settime
close	getpid	recvfrom	sigpending	times
connect	getppid	recvmsg	sigprocmask	umask
creat	getsockname	rename	sigqueue	uname
dup	getsockopt	rmdir	sigset	unlink
dup2	getuid	select	sigsuspend	utime
execle	kill	sem_post	sleep	wait
execve	link	send	socket	waitpid
_Exit & _exit	listen	sendmsg	socketpair	write

Imbalanced Workload

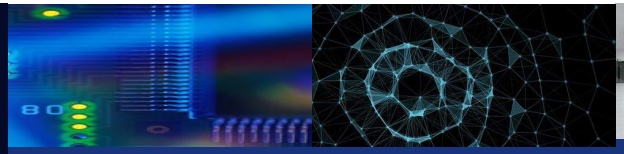
- All threads process the data in same way, but one thread is assigned more work, thus require more time to complete it and impact overall performance



- Considerations
  - Parallelize the inner loop
  - Incline（倾向于）to fine-grained
  - Choose the proper algorithm
  - Divide and conquer, master and worker, work-stealing

Granularity（任务粒度）

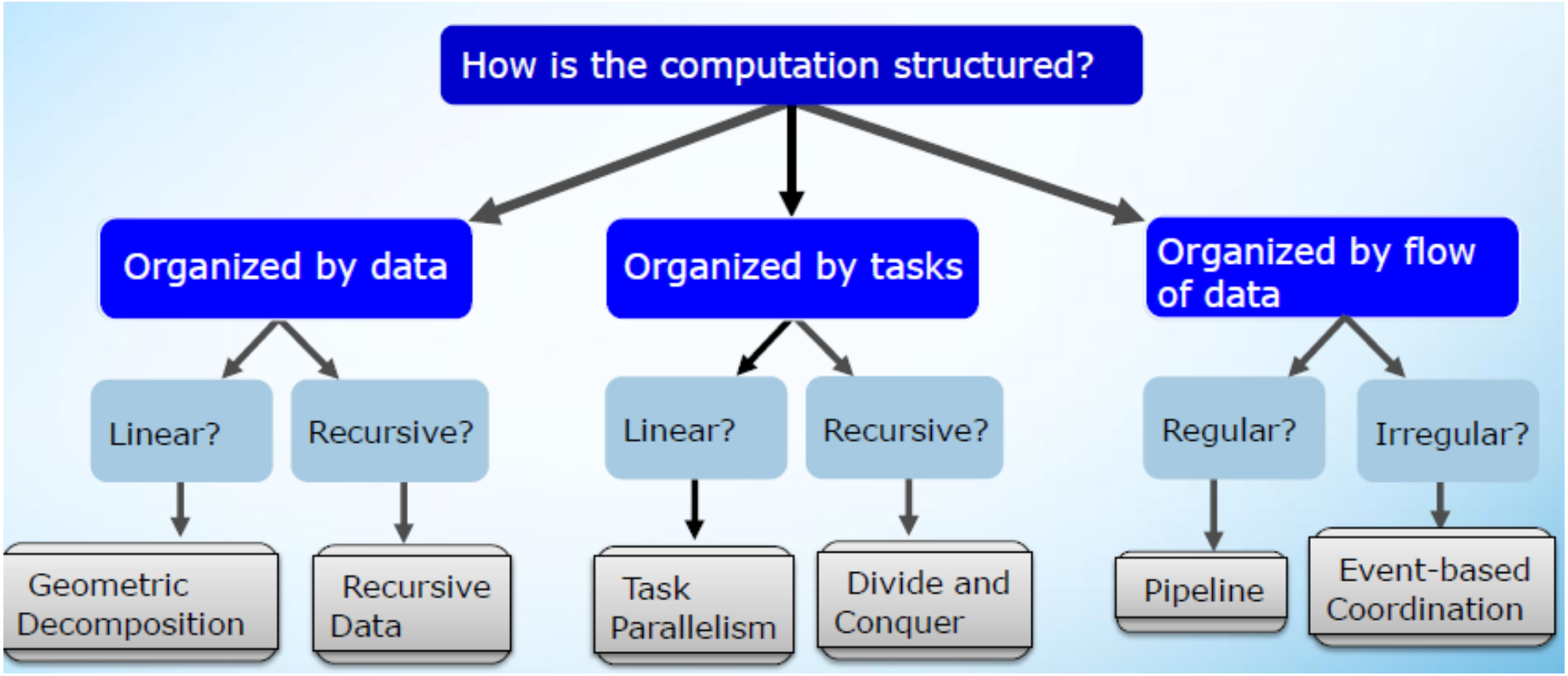
- An extent to which a larger entity is subdivided
- Coarse-grained means fewer and larger components
- Fine-grained means more and smaller components
- **Consideration**
  - Fine-grained will increase the workload for task scheduler
  - Coarse-grained may cause the workload imbalance
  - Benchmark to 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，Spin_lock
  - Reduce lock granularity
  - Don’t be a lock hub
  - Introduce a concurrent container for shared data

Parallel Algorithm



Hadoop Tensorflow Dataflow

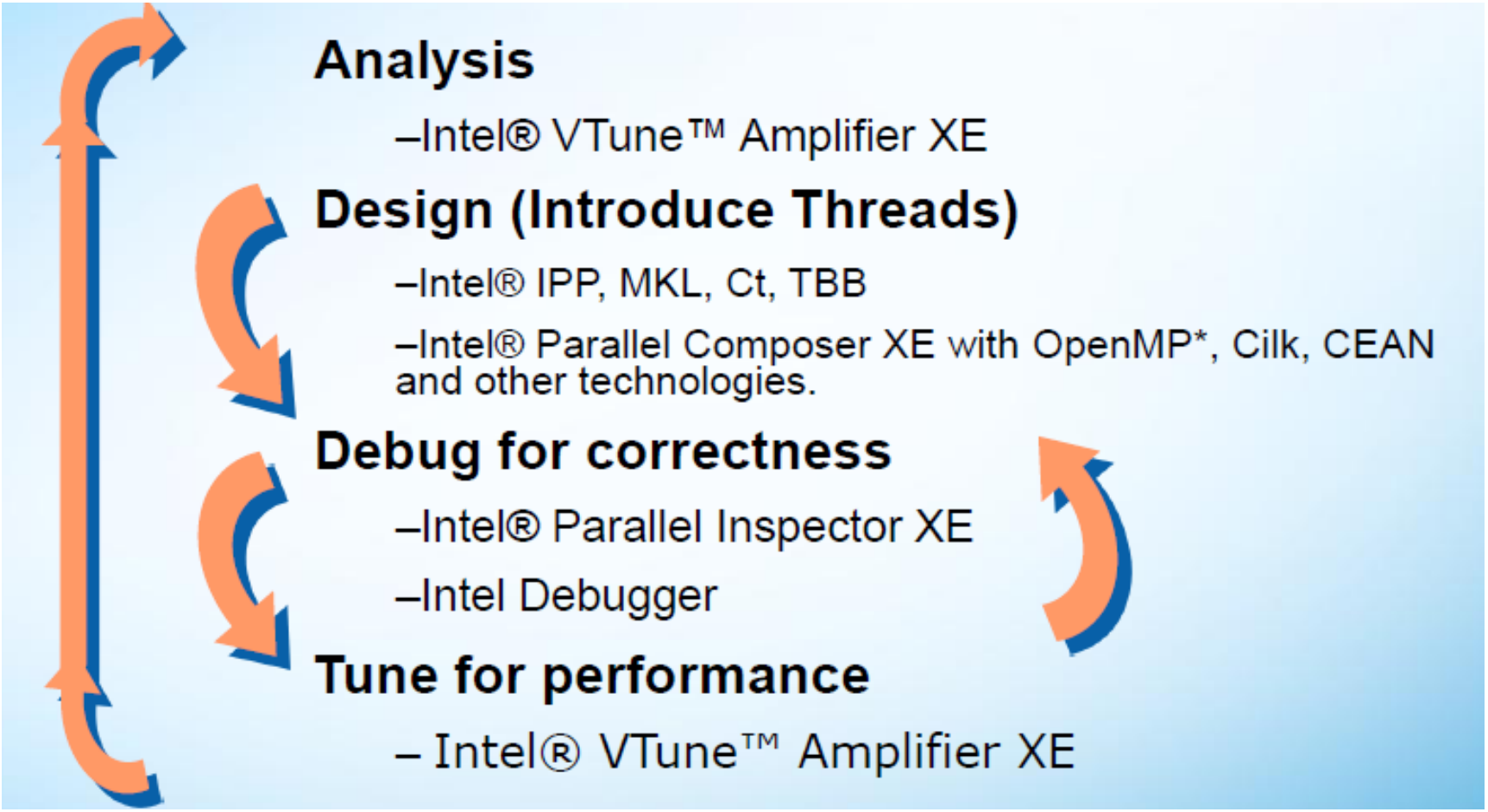
A Generic Development Cycle (1)

- **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 (abstraction)**
  - Use **proved parallel implementations (Implementation)**
  - 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（LLVM，Soot）
- **Tune for performance**
  - Balance the workload
  - Adjust lock & wait
  - Reduce thread operation overhead
  - Set the right granularity
  - Benchmark for scalibility

Intel Generic Development Cycle

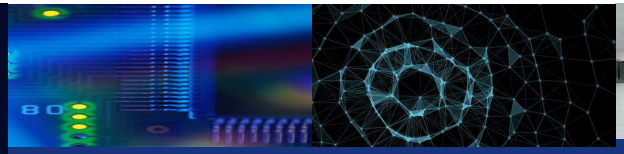


Summary

- **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

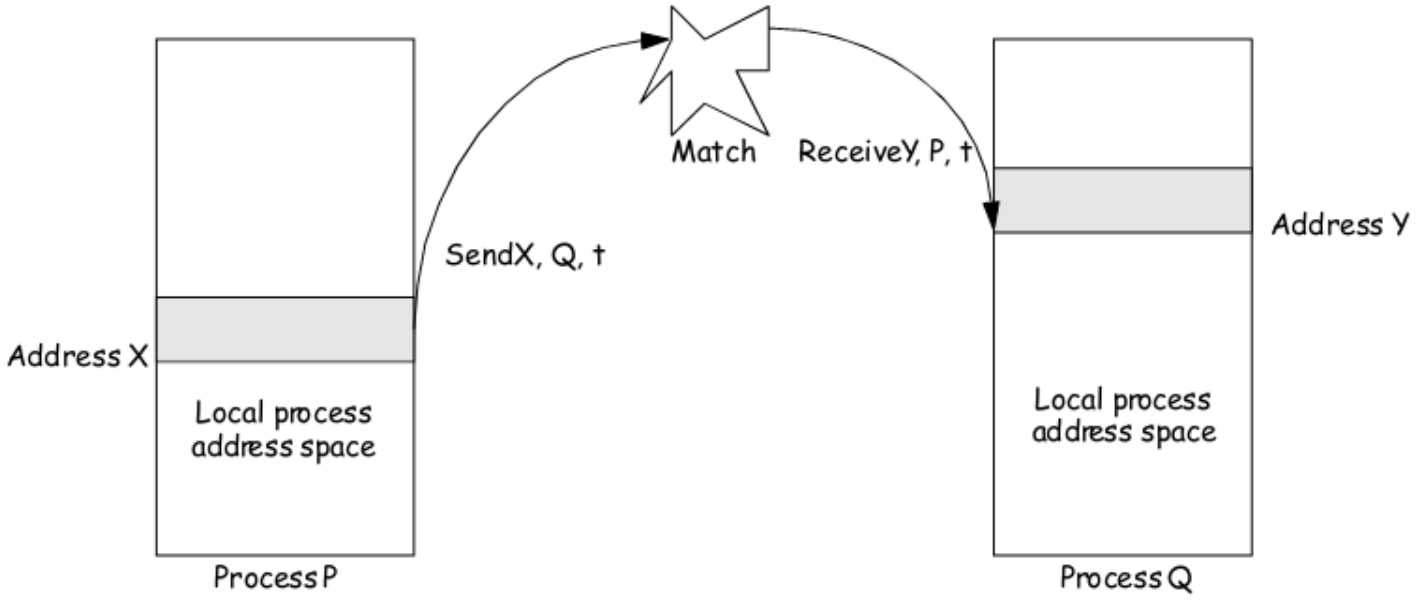
MESSAGE PASSING MODEL



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

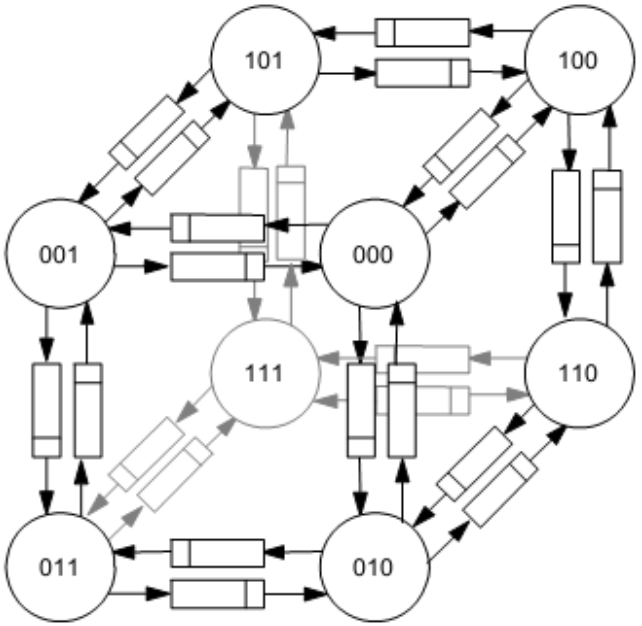
Message Passing Abstraction



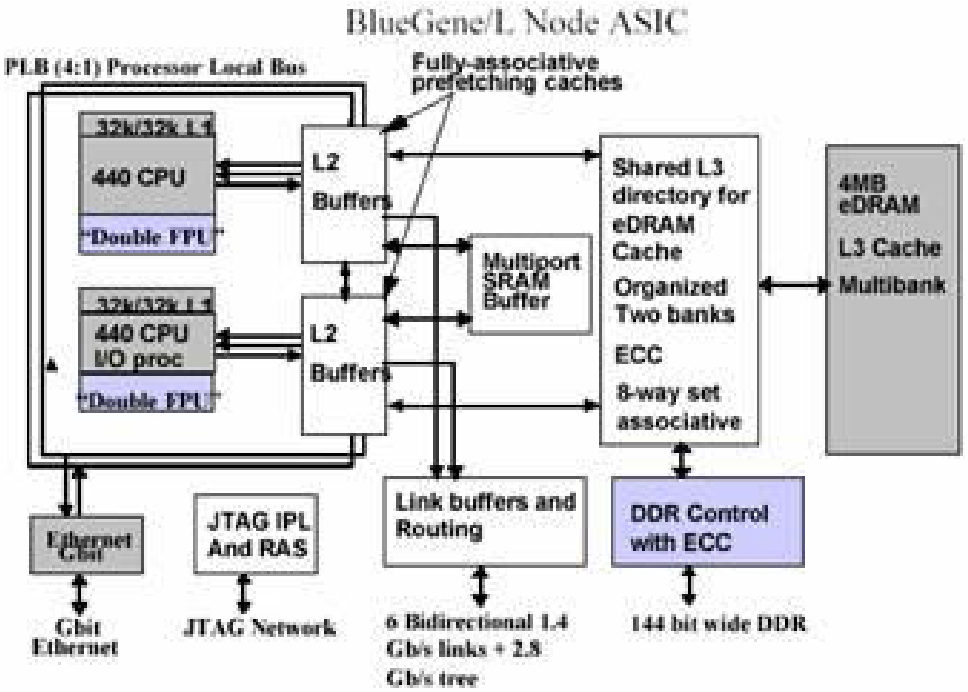
- Send specifies buffer to be transmitted and sending process
- Recv specifies receiving process and application storage to receive into
- Memory to memory copy, but need to name processes
- Optional tag on send and matching rule on receive
- Many overheads: copying, buffer management, protection

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

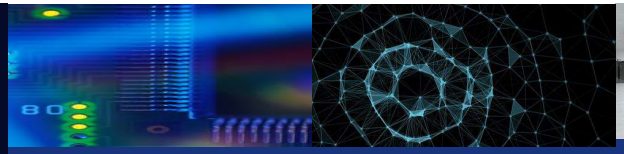


Nodes: 2 PowerPC 440s; everything except DRAM on one chip

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



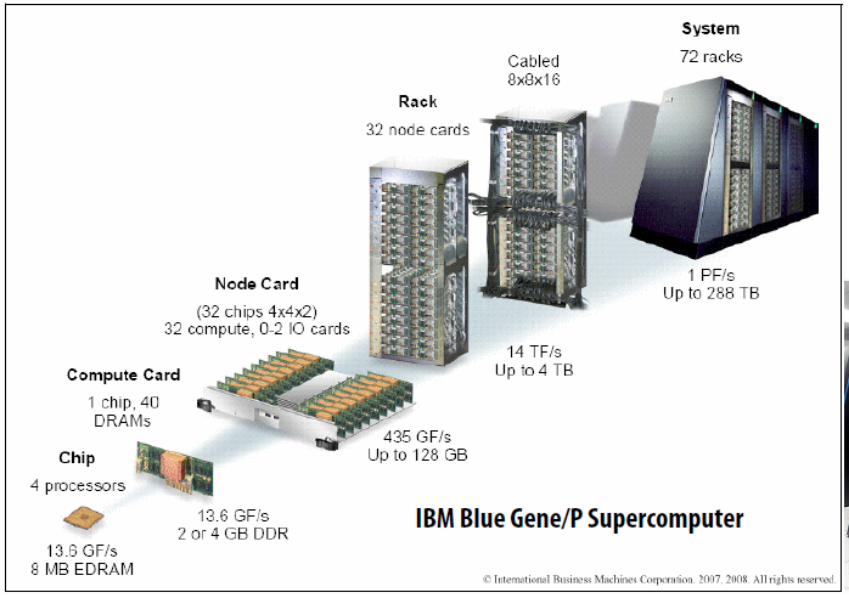
Implementations

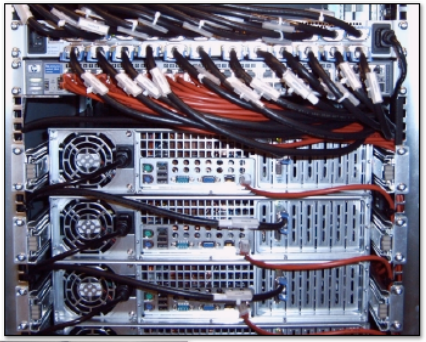
- From a programming perspective
  - Message passing implementations usually comprise a library of subroutines
  - Calls to these subroutines are imbedded in source code
  - The programmer is responsible for determining all parallelism
- Historically, a variety of message passing libraries have been 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

Message passing (implementation)

- Popular software library: MPI (message passing interface)
- Hardware need not implement system-wide loads and stores to execute message passing programs (need only be able to communicate messages)
  - Can connect commodity systems together to form large parallel machine (message passing is a programming model for clusters)





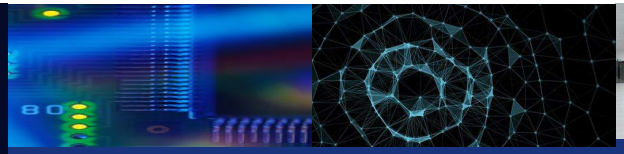
Cluster of workstations
(Infiniband network)

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://wwwunix.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

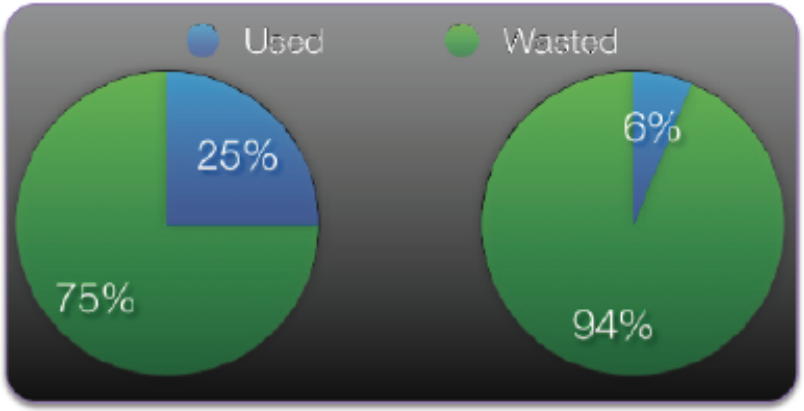


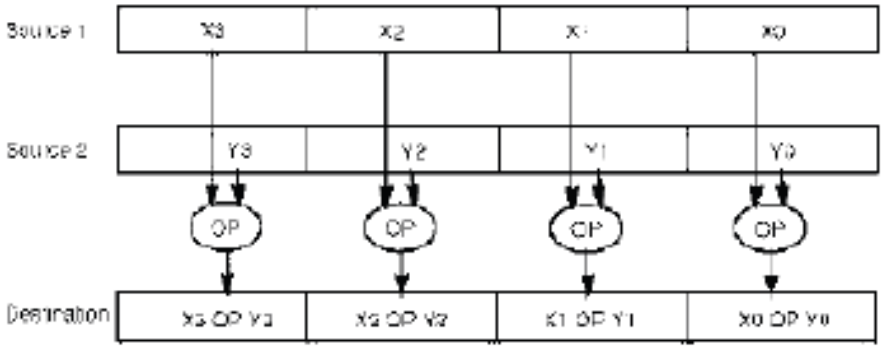
Parallel programming models

GPGPU PROGRAMMING MODEL

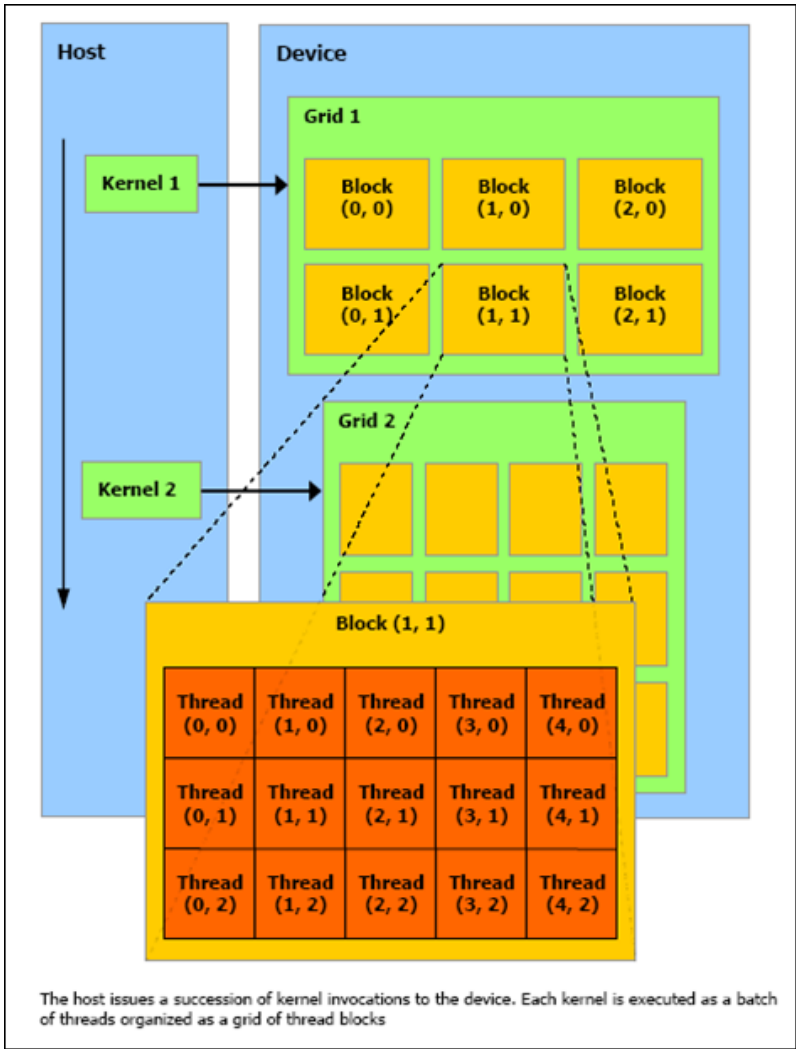
CUDA Goals: SIMD Programming

- 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

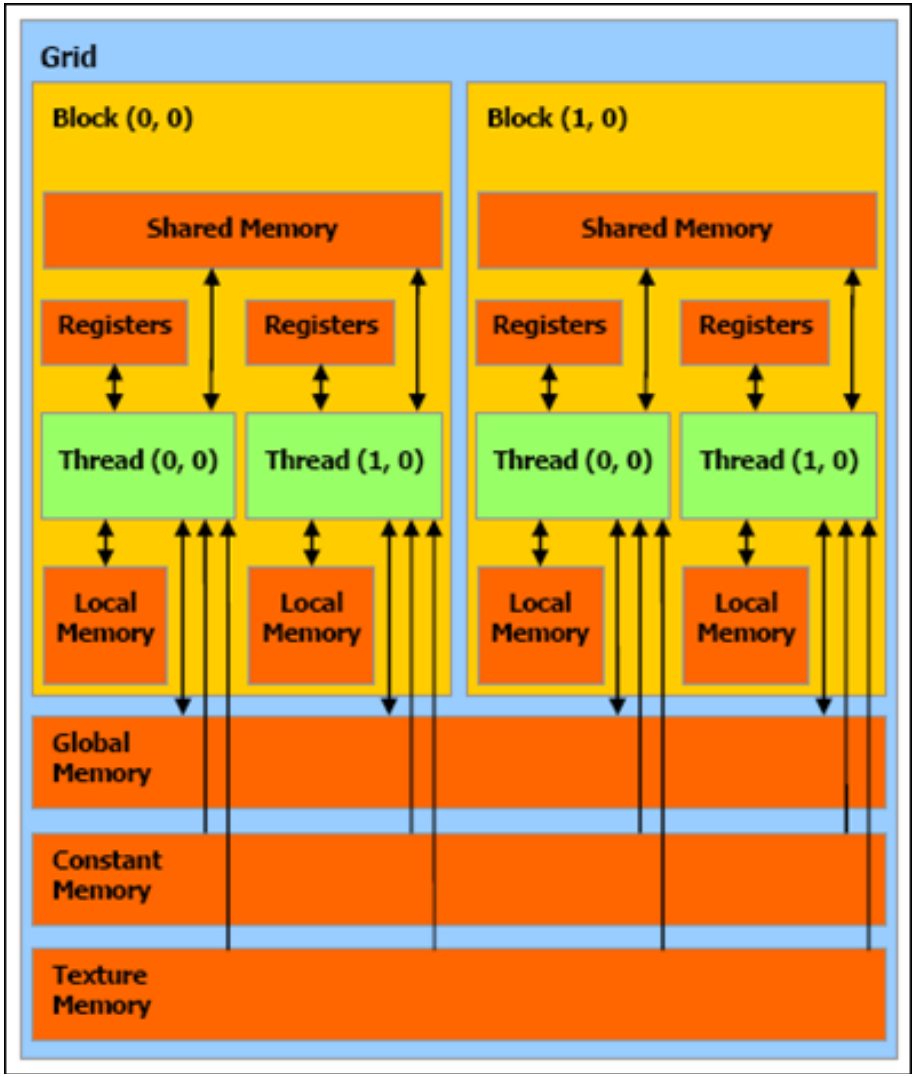




CUDA Programming Model



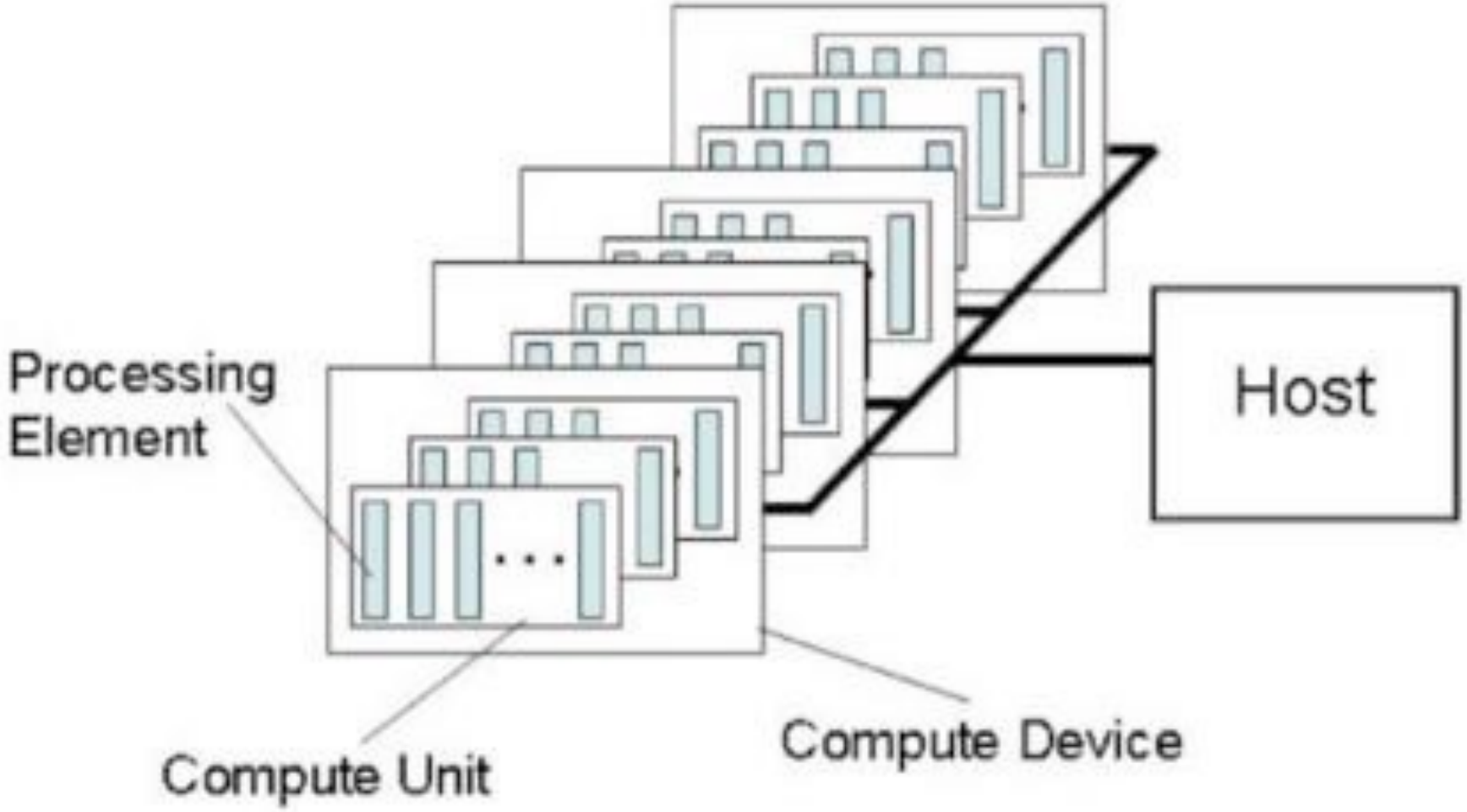
The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks



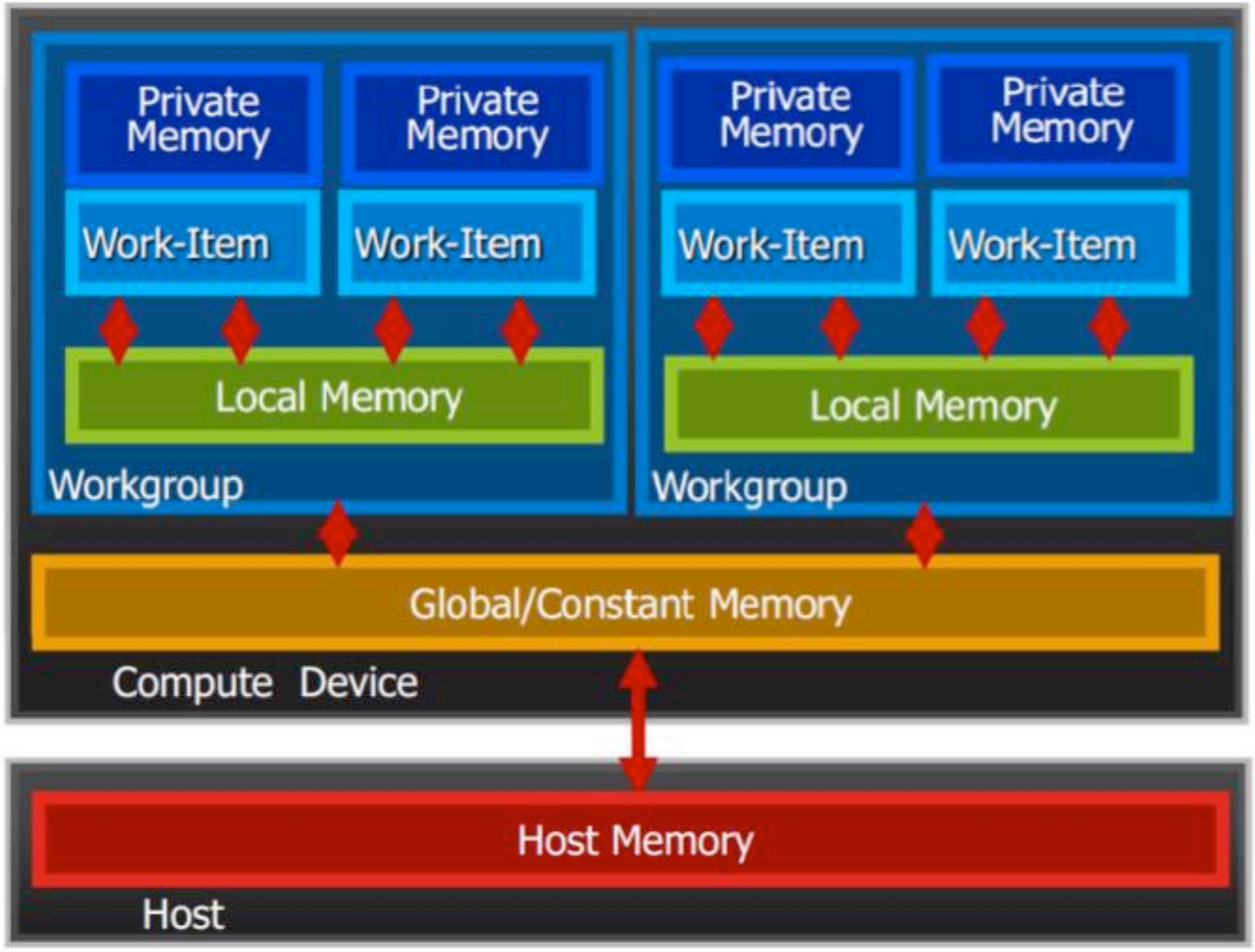
OpenCL Programming Model

- 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

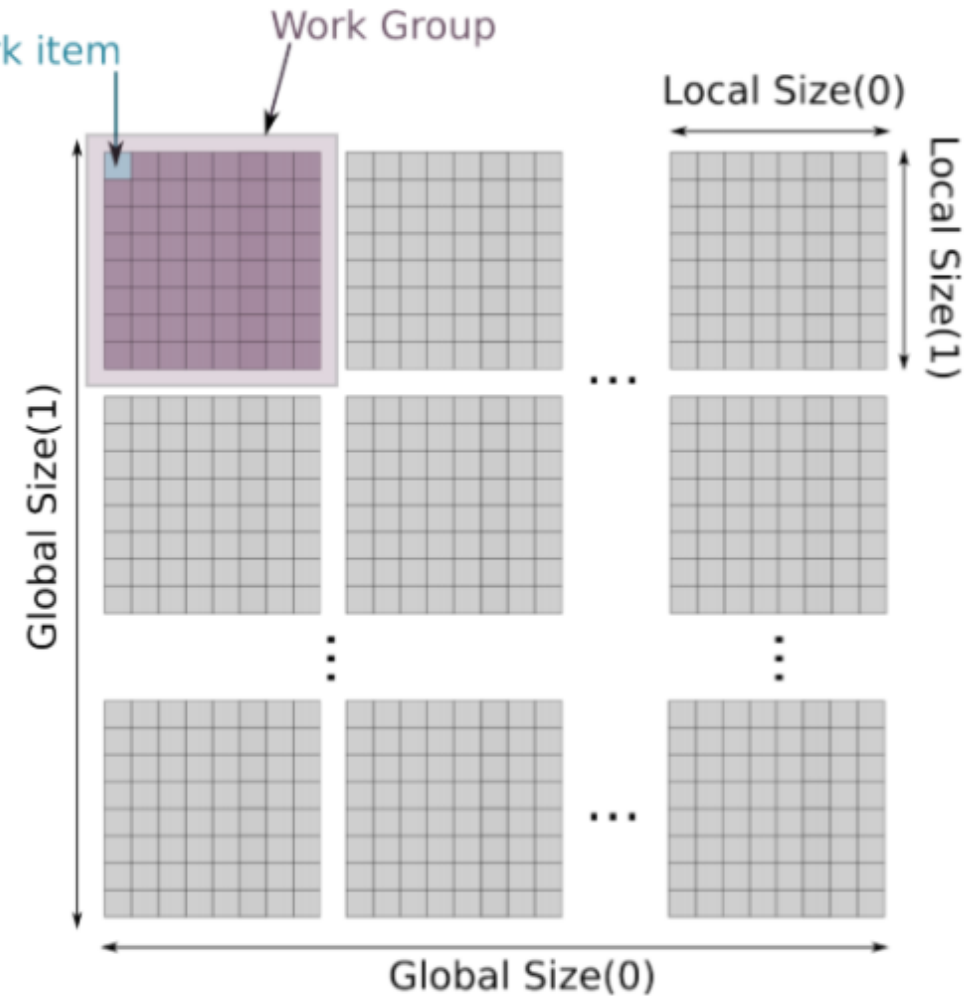
OpenCL Platform Model



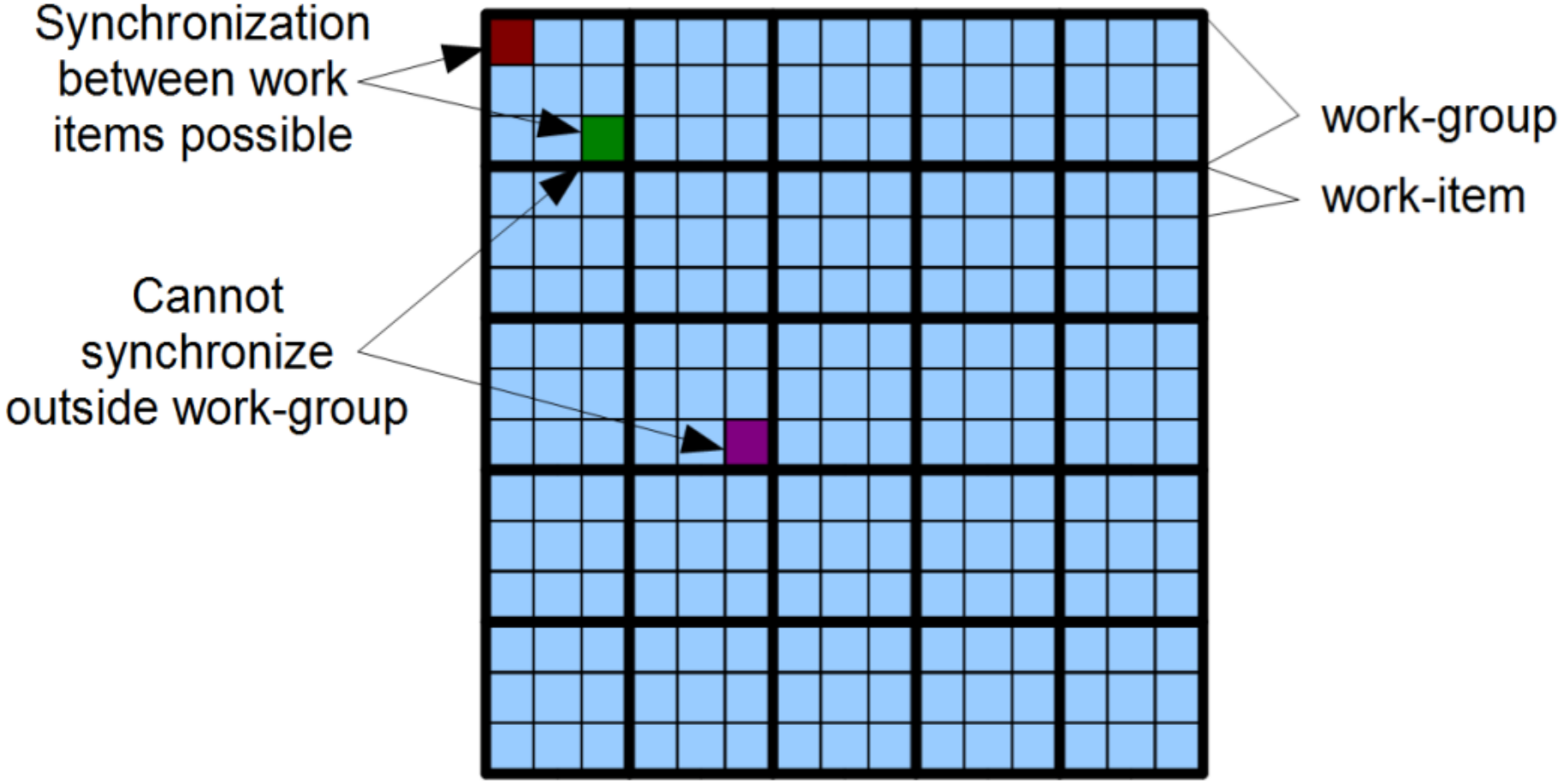
OpenCL Memory Model



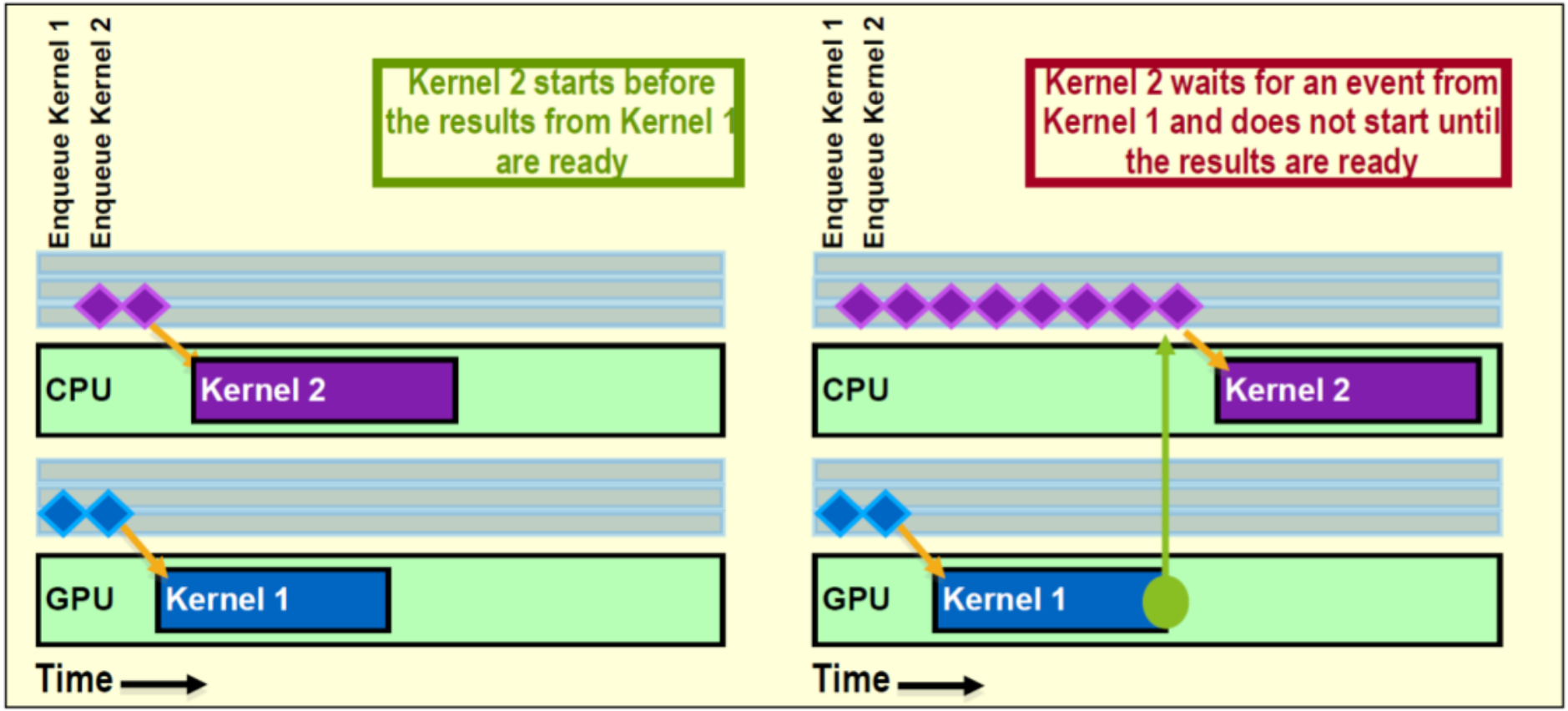
2D Data-Parallel Execution in OpenCL



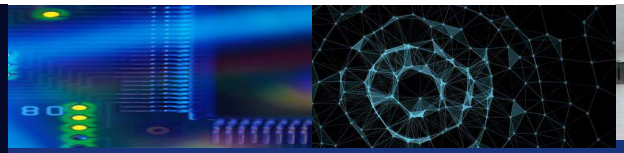
OpenCL Work-group / Work-unit Structure



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

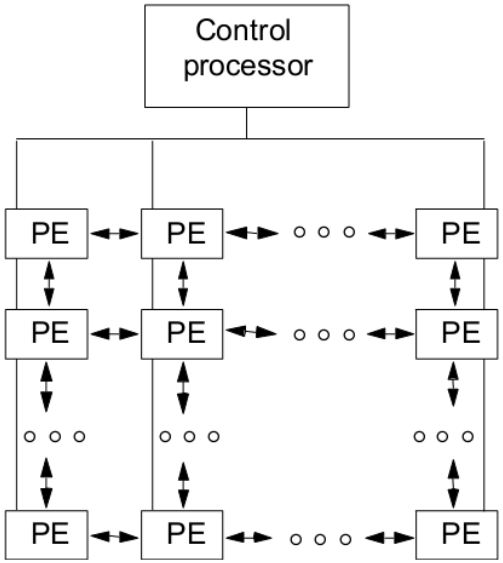
Both approaches are viable CPU options

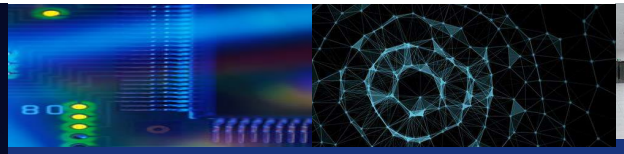
Parallel programming models

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





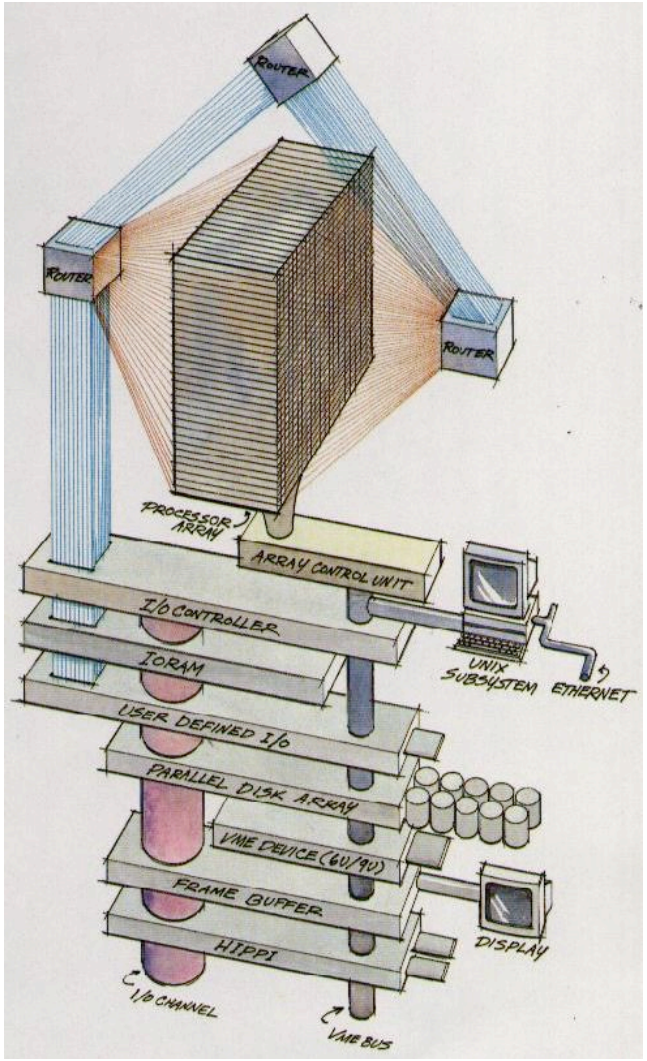
Application of Data Parallelism

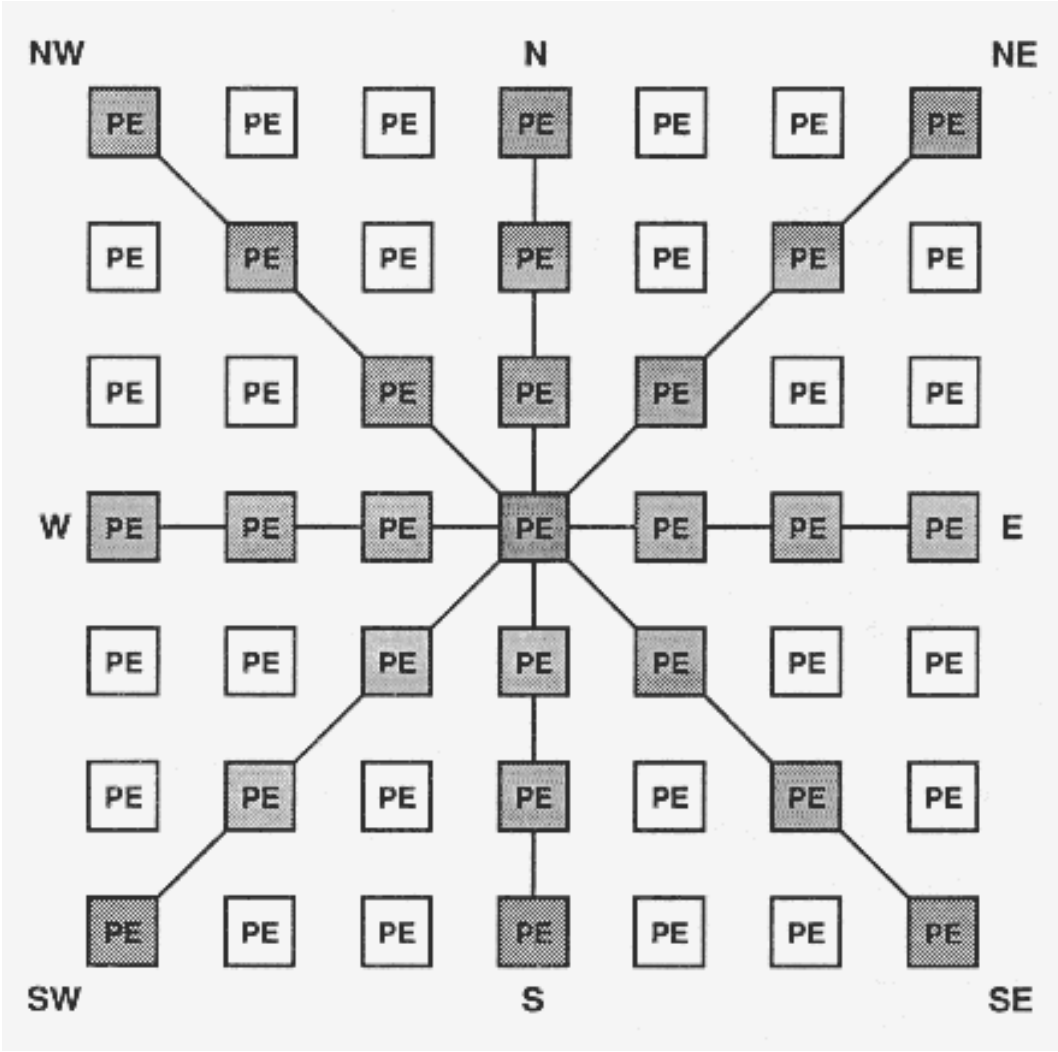
- Example
  - Each PE contains an employee record with his/her salary

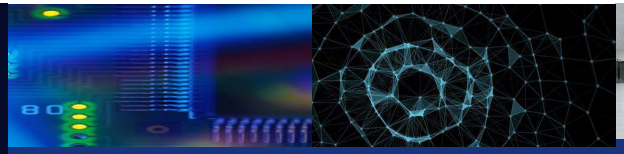
```
If salary > 100K then
    salary = salary *1.05
else
    salary = salary *1.10
```

  - 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



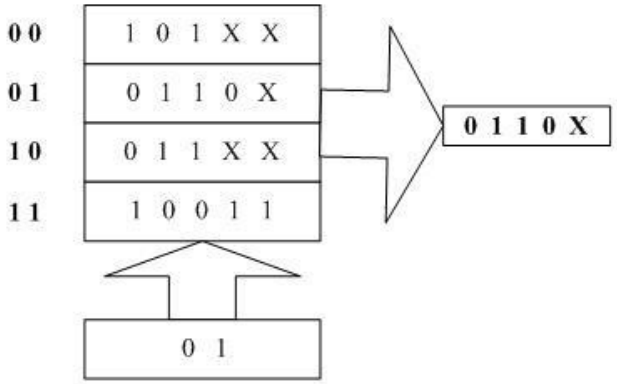




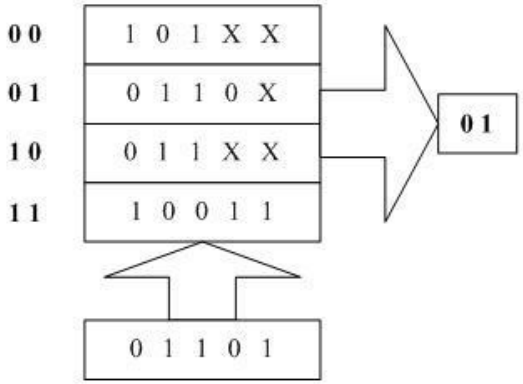
Dataflow Architecture

- 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)

Content addressable memory

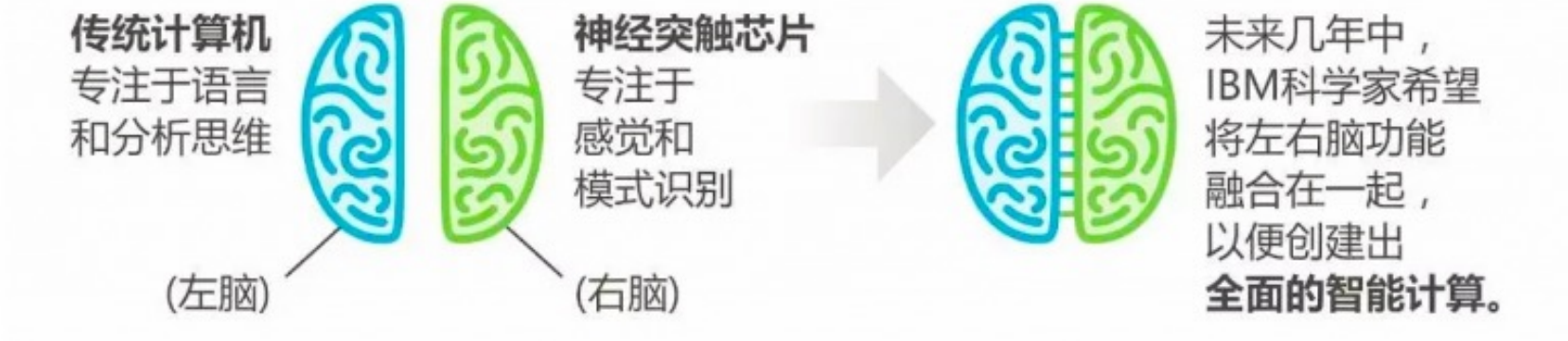


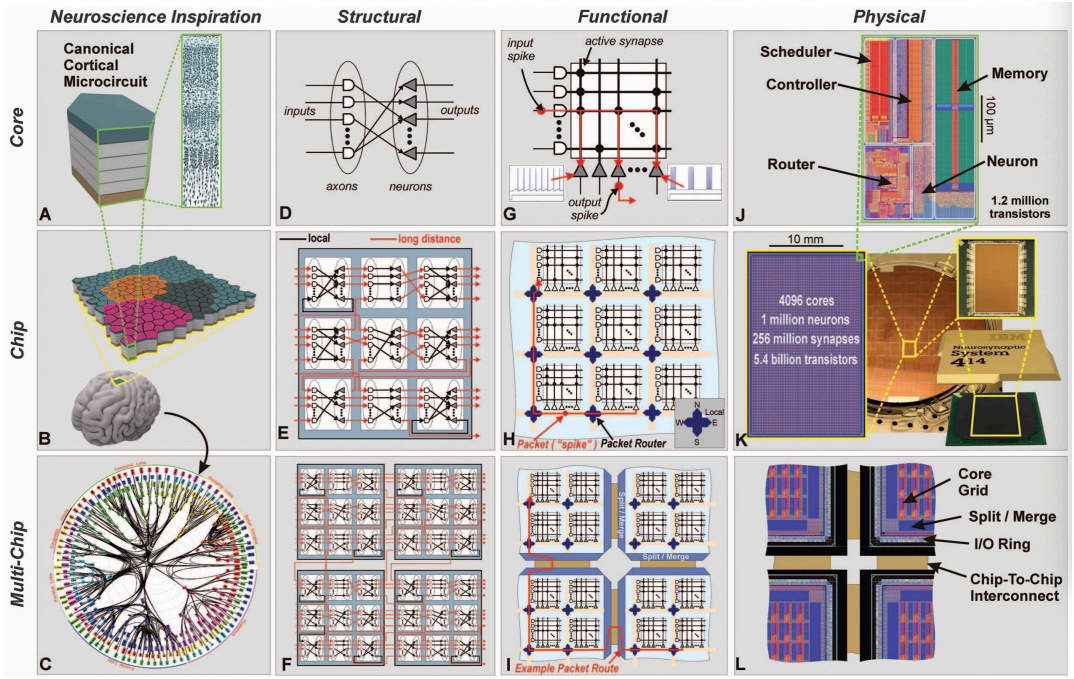
Traditional Memory



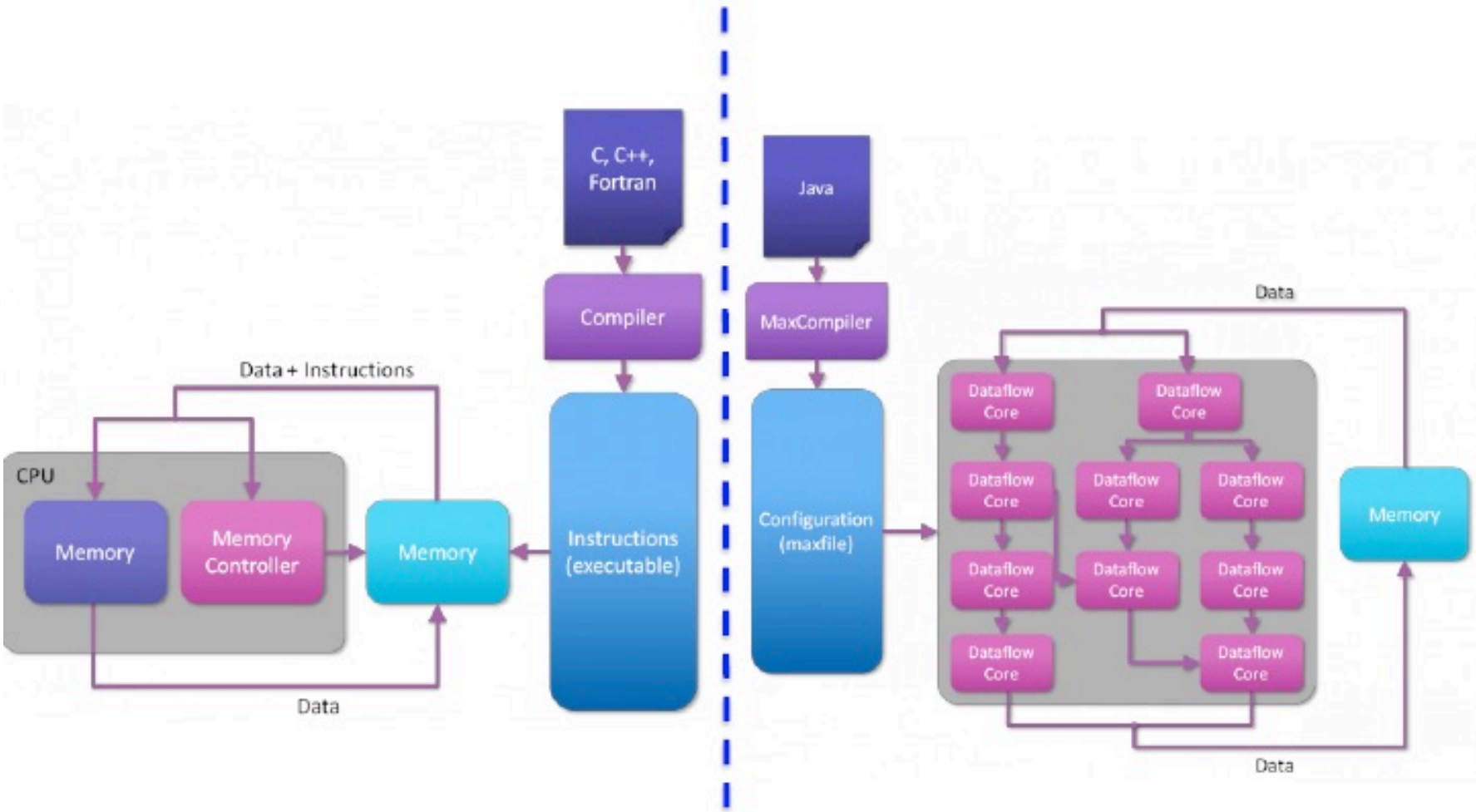
Content Addressable Memory

IBM Truenorth



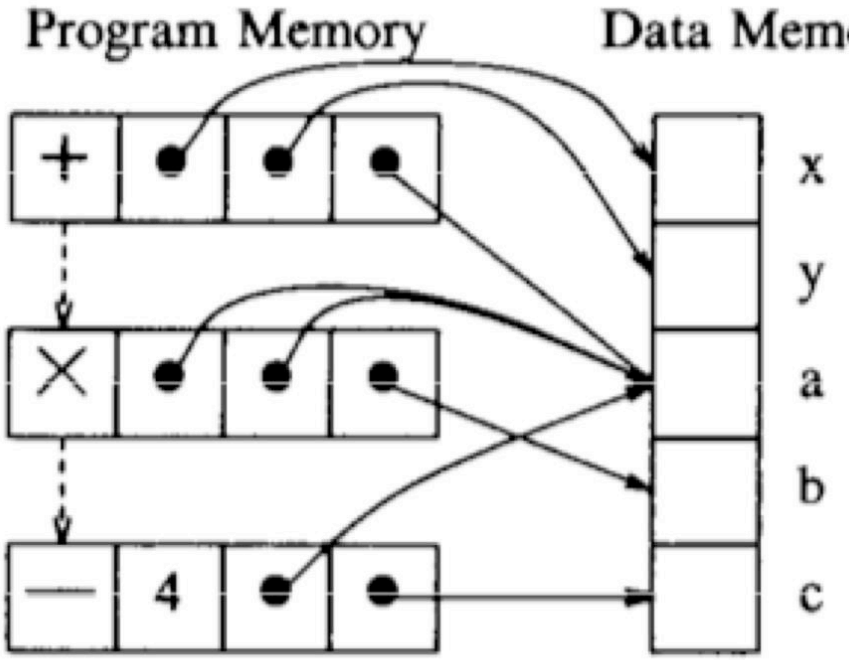


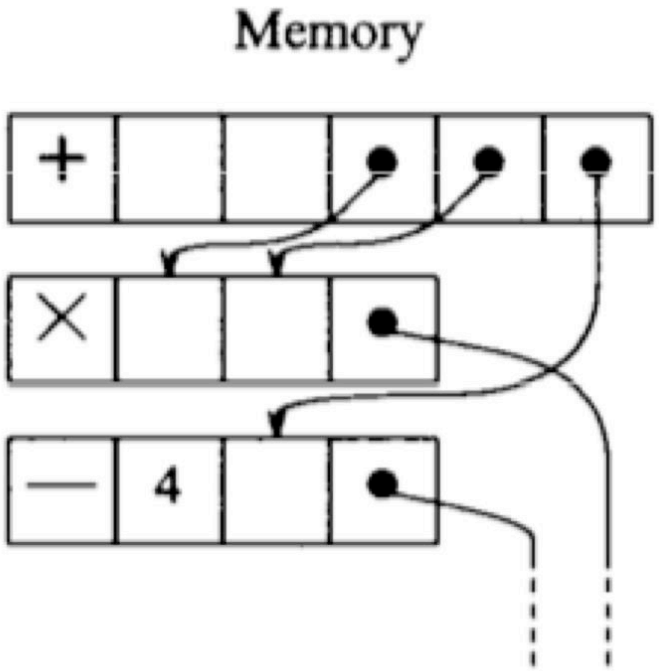
Computing with Control Flow/ Data Flow Cores

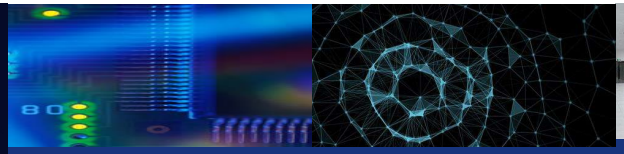


Control Flow vs. Data Flow

a := x + y
b := a × a
c := 4 — a







Thank You !