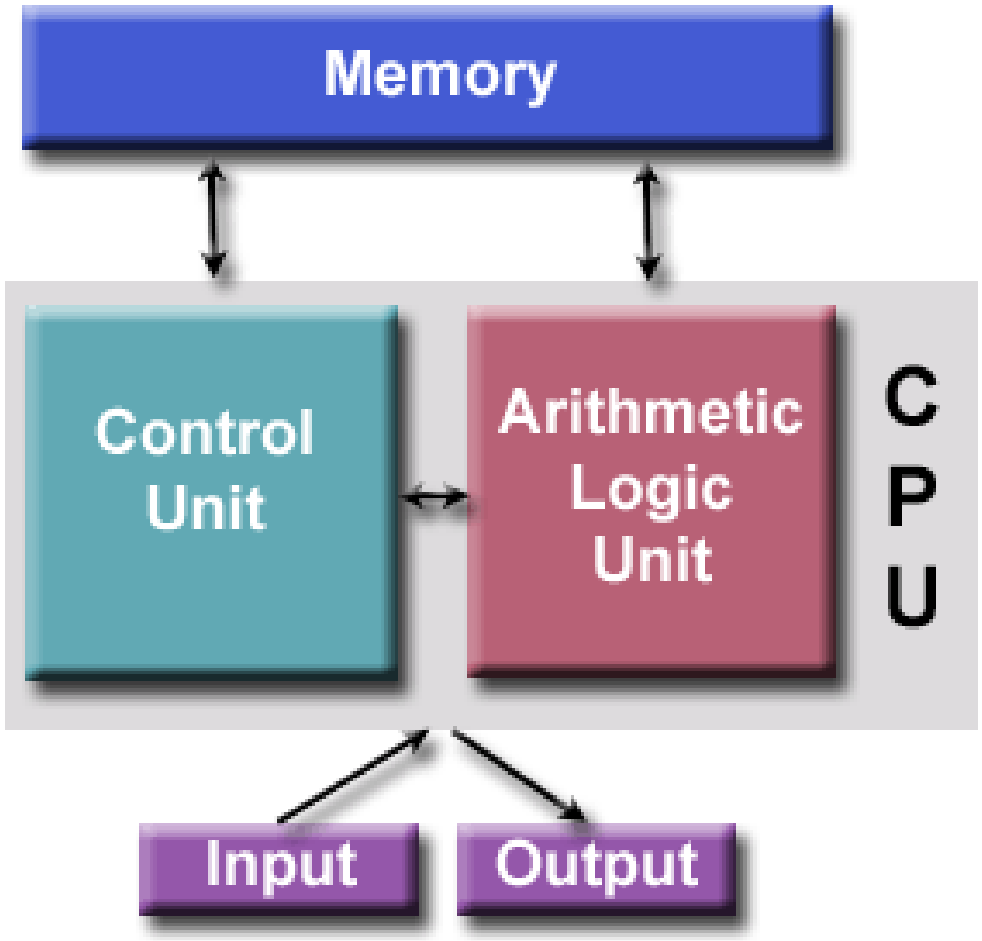
今天的内容

- 计算系统概述
- 并行程序模型

从最简单的说起



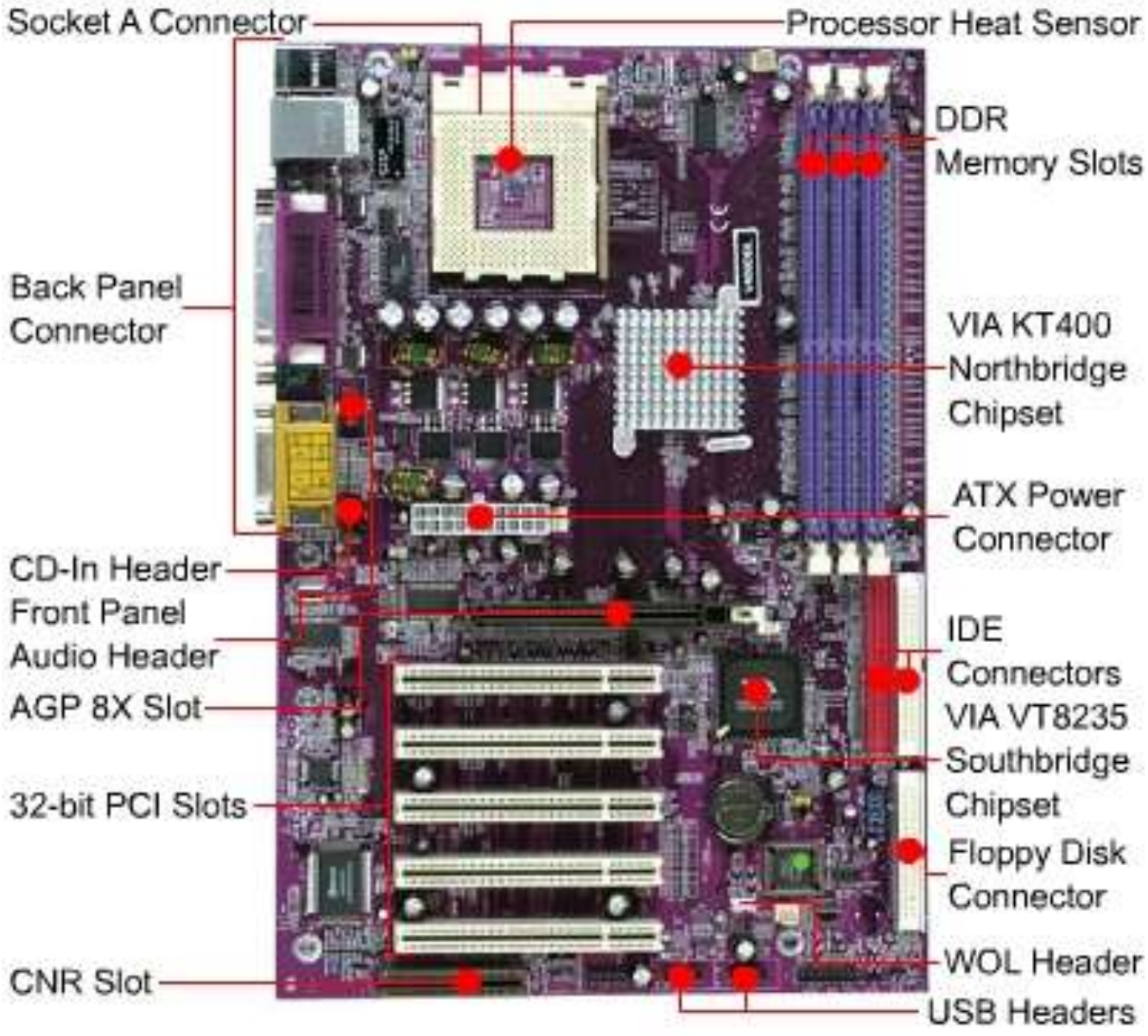
Read/write, random access **memory** is used to **store both program instructions and data**

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

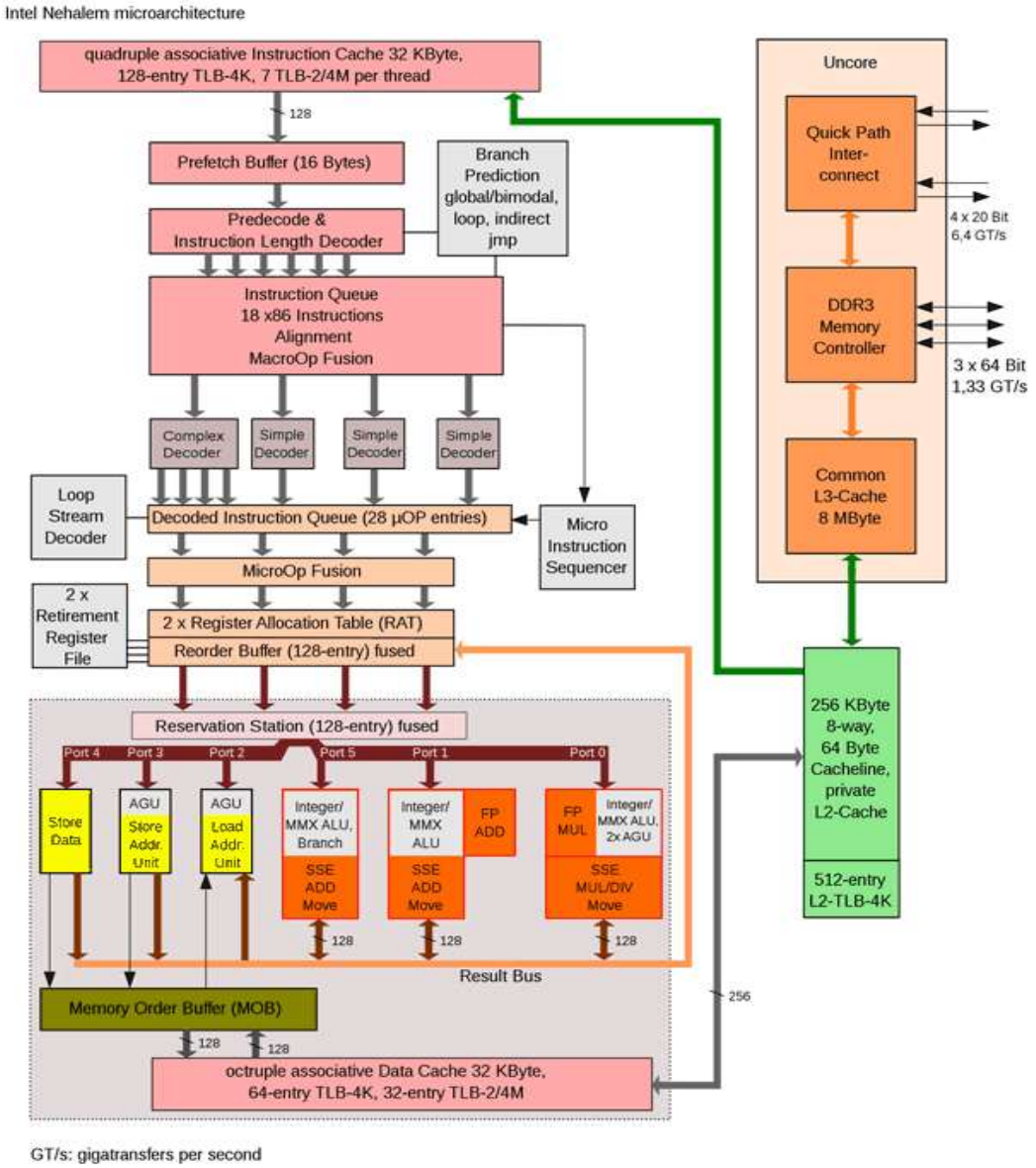
Aritmetic Unit performs basic arithmetic **operations**

Input/Output is the **interface** to the **human operator**

你对自己的计算机了解多少？

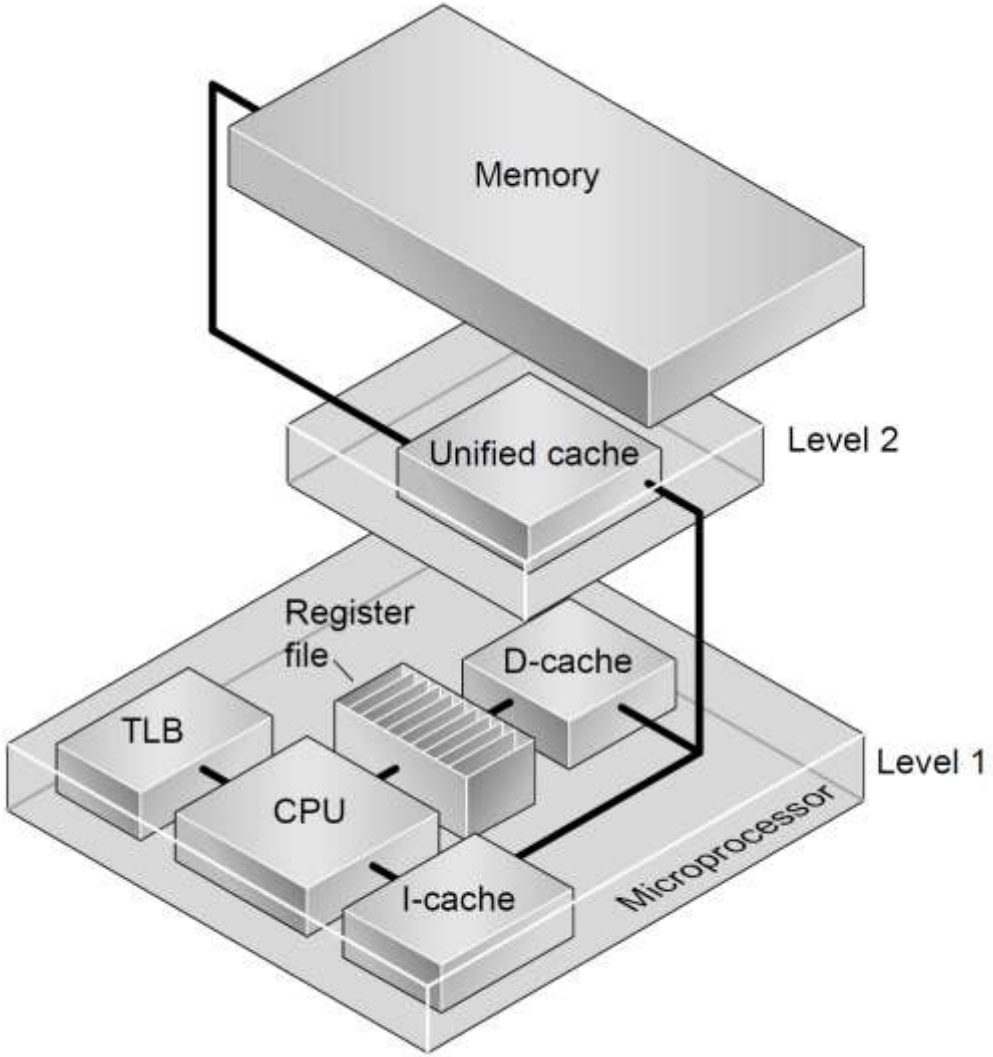


对CPU了解多少？



- Registers
- Cache
- Execution Units (向量)

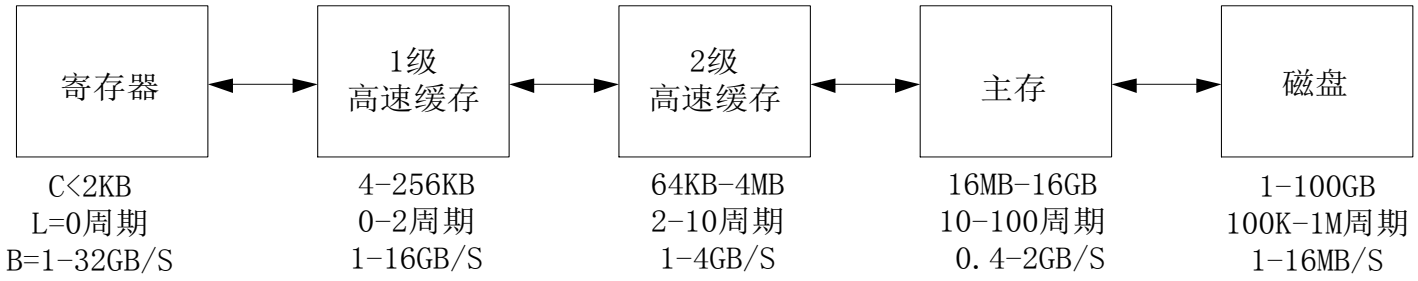
Memory Hierarchy

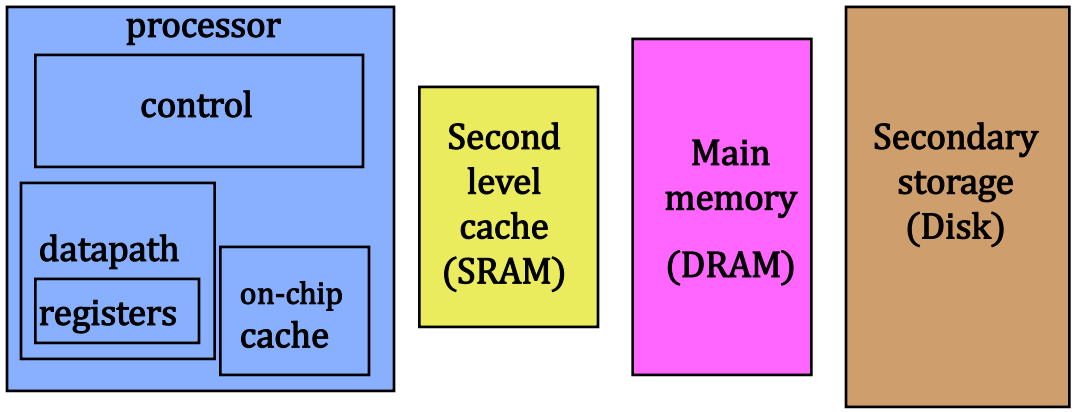


- For a programmer, we can not control the caches. But we should aware of cache.

存储器层次

- 存储器的层次结构





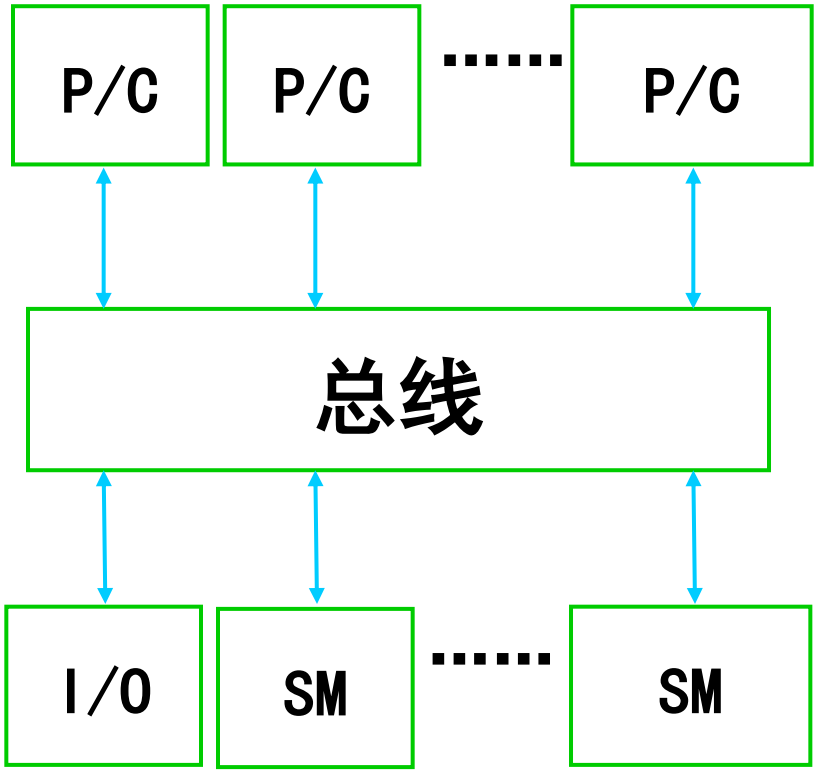
Speed	1ns	10ns	100ns	10ms
Size	KB	MB	GB	TB

典型并行计算系统

并行计算机的分类

- 典型的大规模并行处理系统
  - 对称多处理机（SMP）
  - 分布存储并行机
    - 集群系统（Cluster）

传统的对称多处理机（SMP）



会有什么问题？

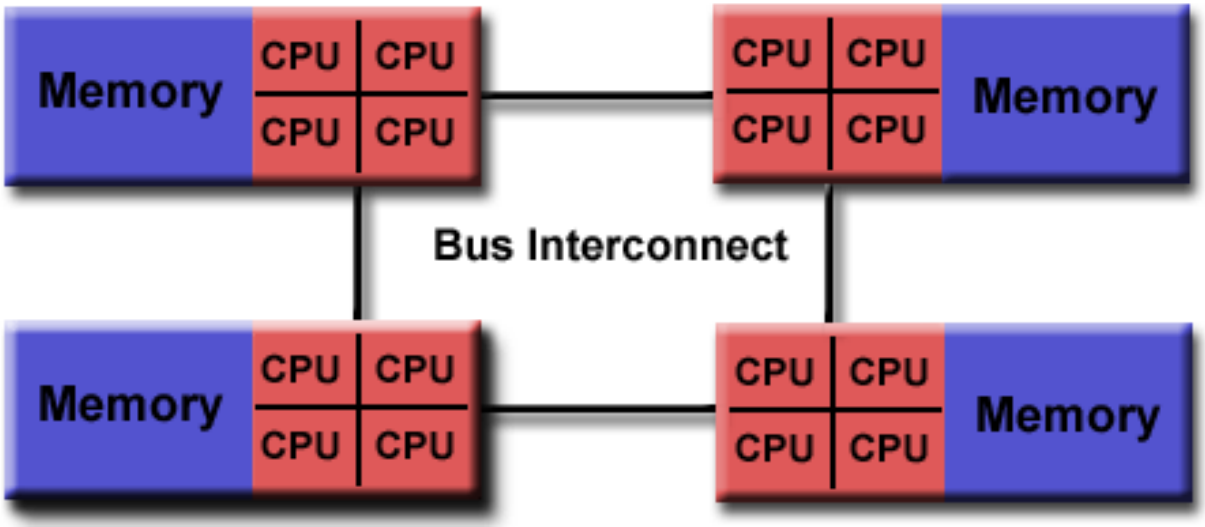
总线可能出现瓶颈，扩展性不足

P/C：处理器和高速缓存
SM：共享存储器

典型系统有：SGI Power Challenge

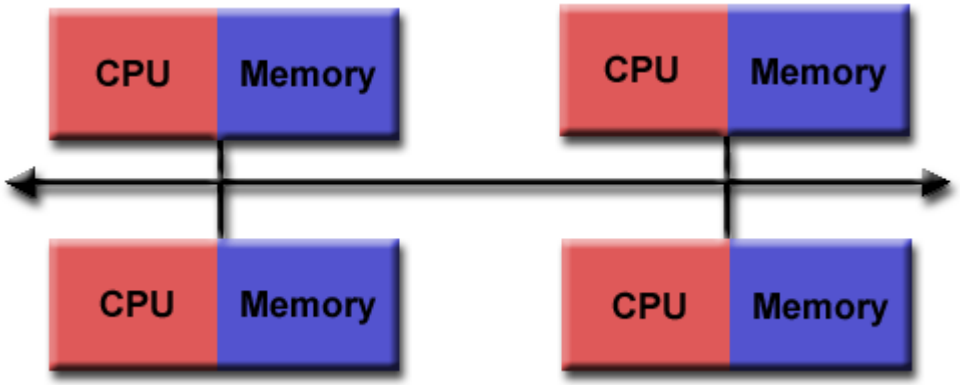
NUMA

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



这对我们程序员有什么要求？

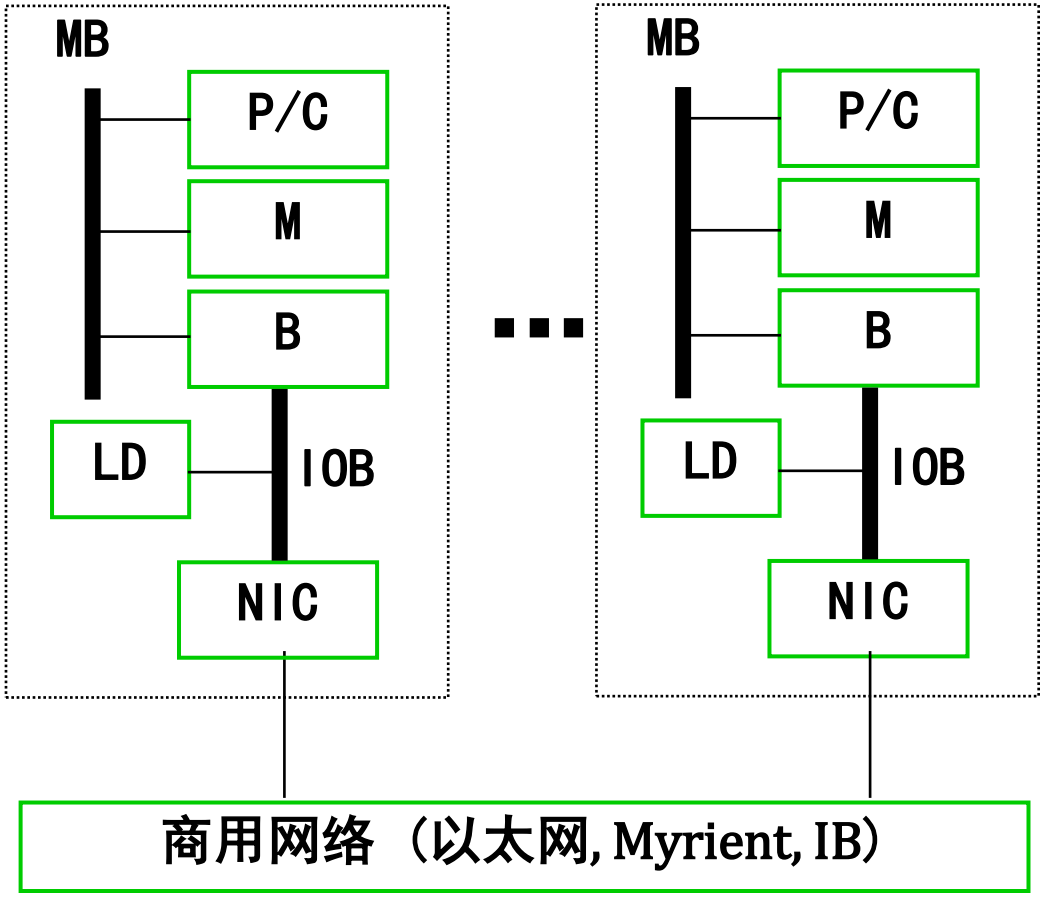
分布式存储并行机



Advantages:

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

集群系统

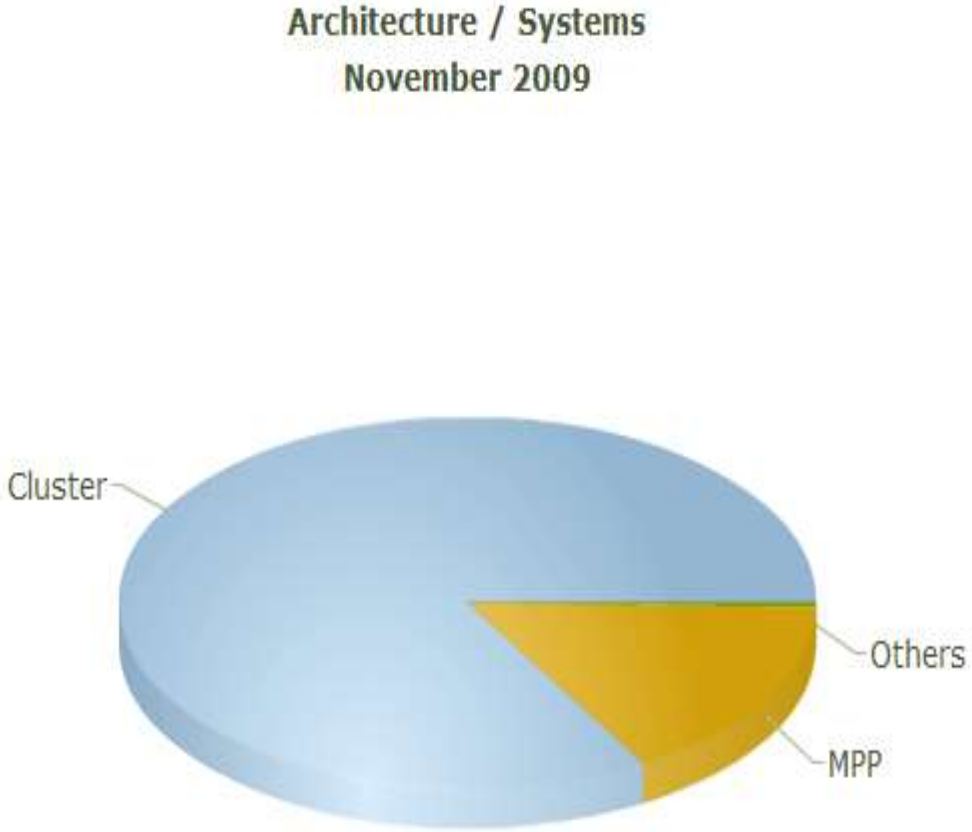


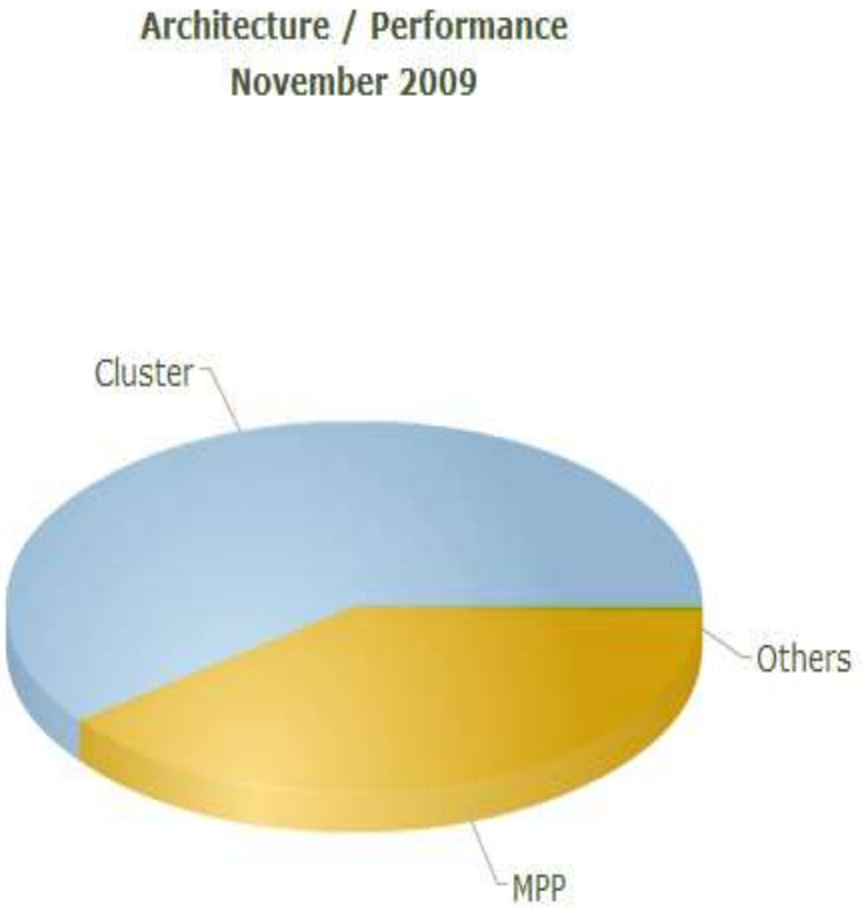
B：存储总线与I/O总线的接口；LD：本地磁盘
IOB：I/O总线

完全采用
商用硬件

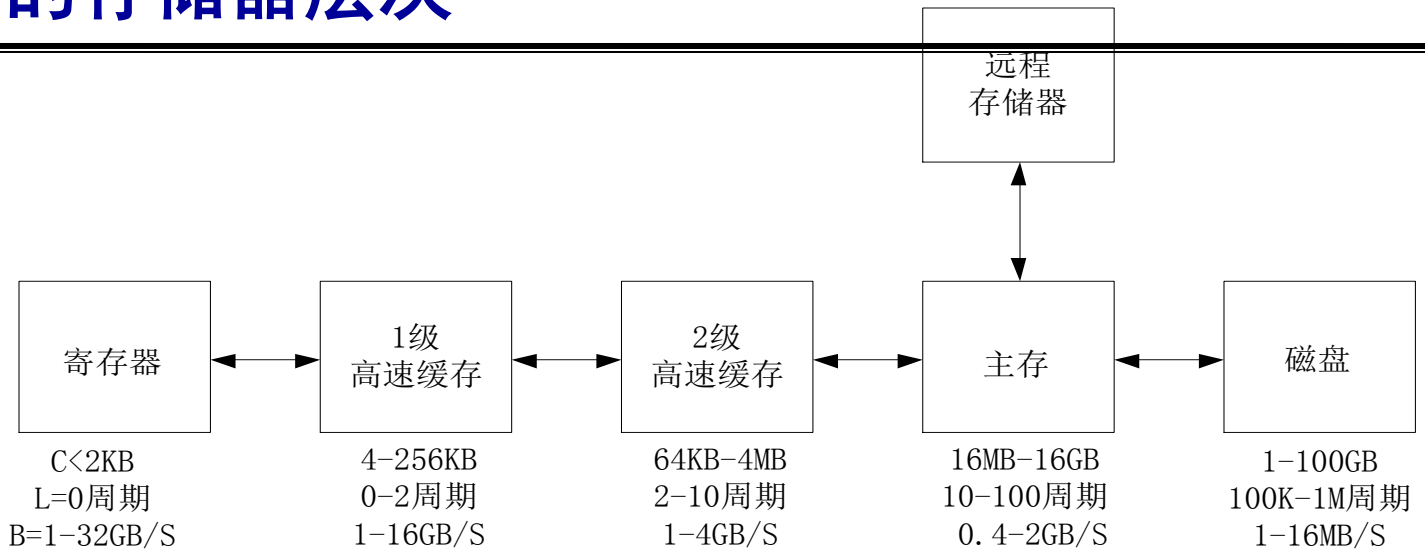
- 每个结点都是一个完整计算机
- 各结点通过低成本的商用网络互连
- 每个结点上驻留有完整的操作系统。

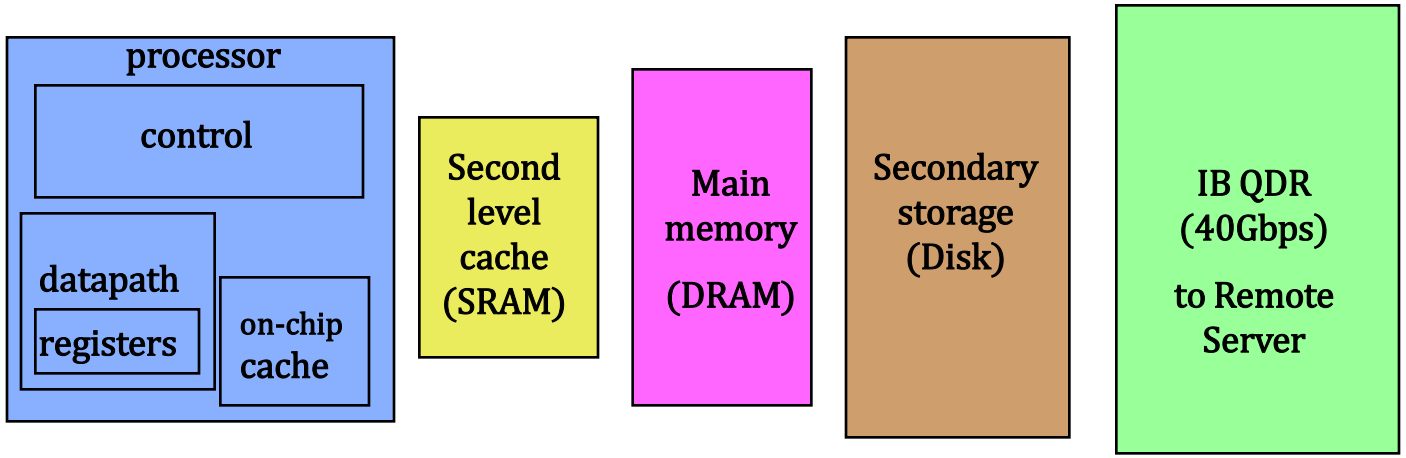
TOP 500 Lists





新的存储器层次





Speed	1ns	10ns	100ns	10ms	~1-2us
Size	KB	MB	GB	TB	~4GB/s

并行程序模型

提纲

1 为什么需要并行程序模型？

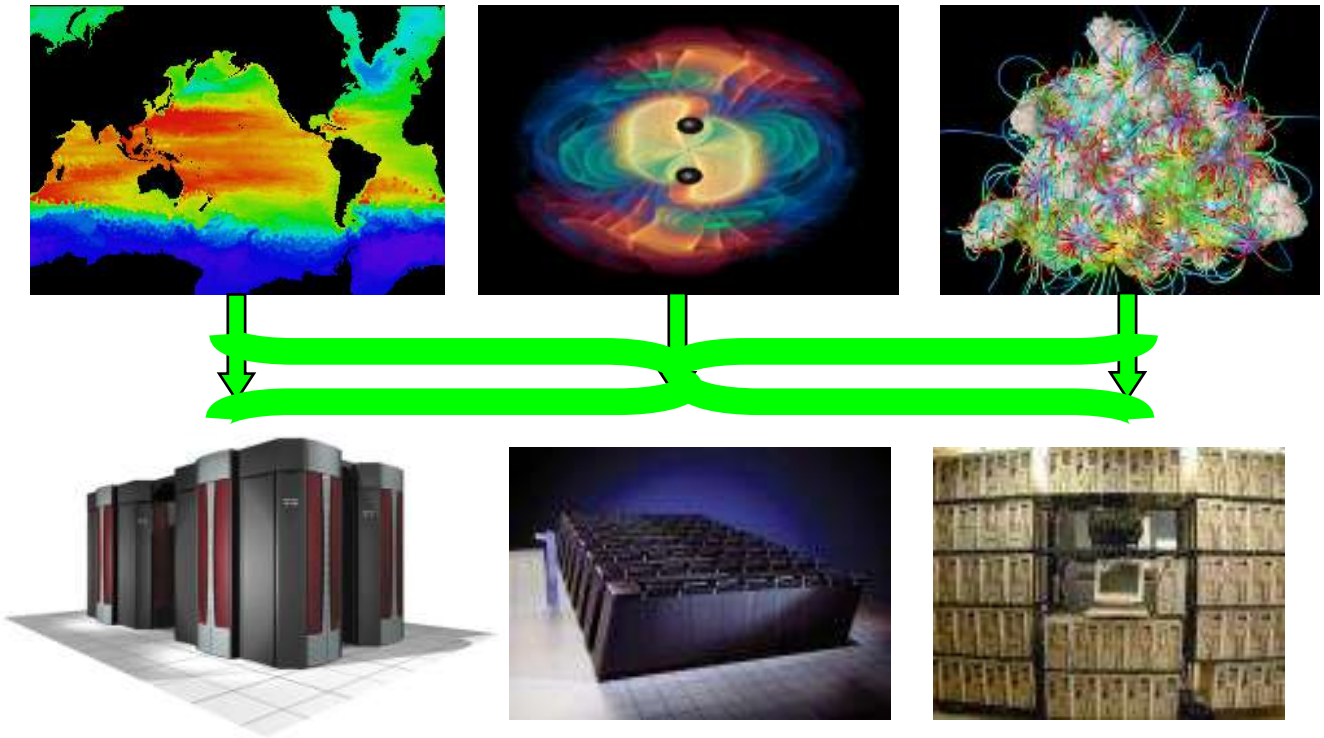
2 什么是并行程序模型

3 主要的并行程序模型

- 共享存储模型(Shared memory)
- 消息传递模型(Message passing)
- 数据并行模型(Data parallel)

4 小结

为什么需要并行程序模型？



Parallel Programming environments in the 90's

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL

CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda

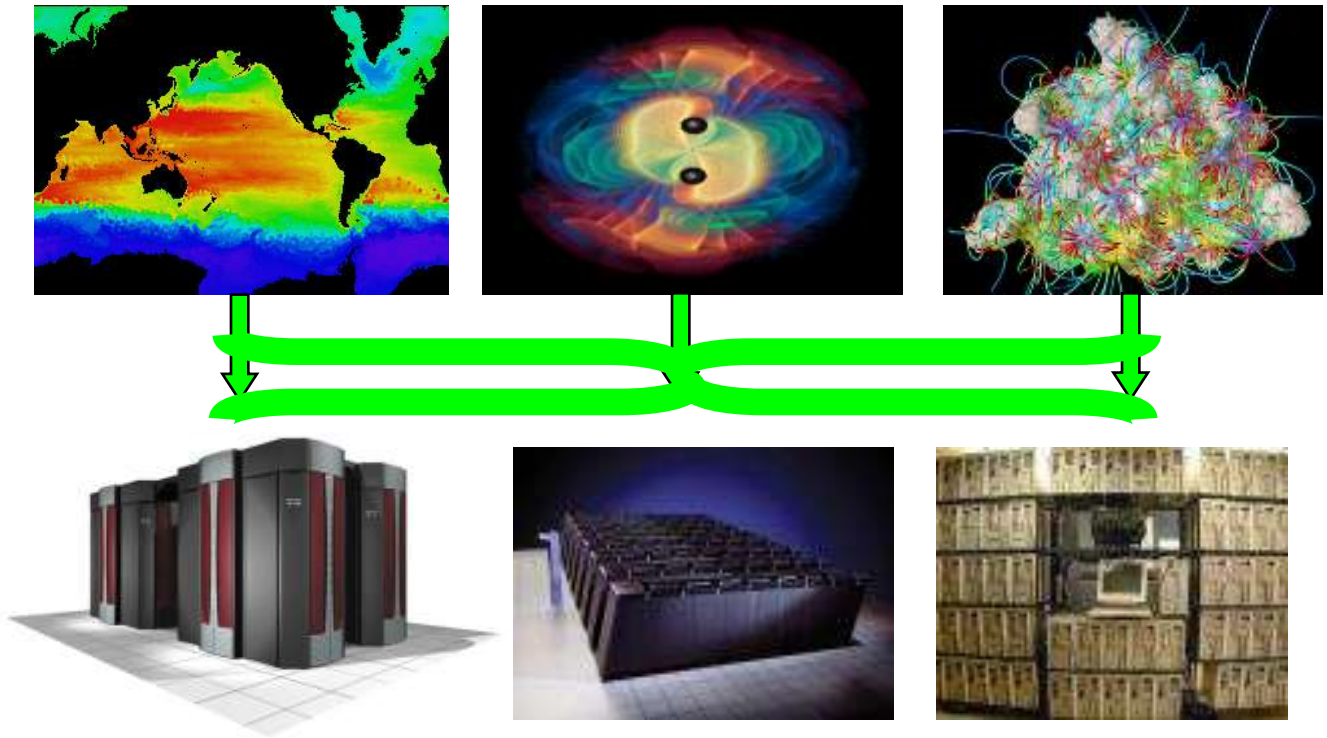
GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold

Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm

Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed
smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

为什么需要并行程序模型？



复杂的任务“简单化”

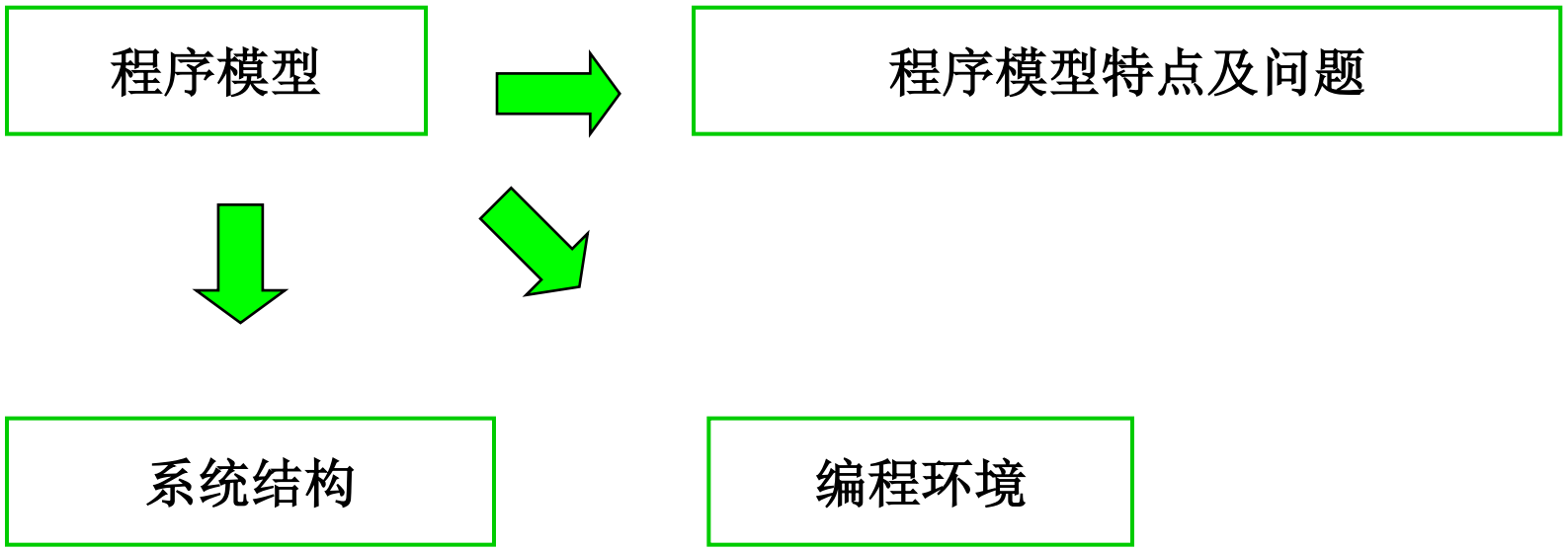
在底层计算系统与上层应用间建立虚拟层

- 屏蔽底层细节，建模主要特征，降低软件开发和移植难度
- 实现标准化，促进并行软件移植

什么是并行程序模型？并行程序抽象的运行行为

- 虚拟层的抽象描述
  - 控制方式
    - 怎样建立并行？/ 怎样控制操作次序？
  - 数据管理
    - 私有数据和共享数据？/ 如何实施共享数据的访问和通信？
  - 同步
    - 如何协调并行？ / 原子化操作？
- 定义
  - 一种程序抽象的集合，它给程序员提供计算机硬件与软件系统透明的简图，程序员利用这些模型就可以为多处理机、多计算机和工作站机群等设计并行程序。
  - 陈国良《并行计算－结构、算法、编程》(修订版) P310
- 具体化形成并行编程标准和编程环境
  - **MPI**
  - **OpenMP/Pthread**
  - **…**

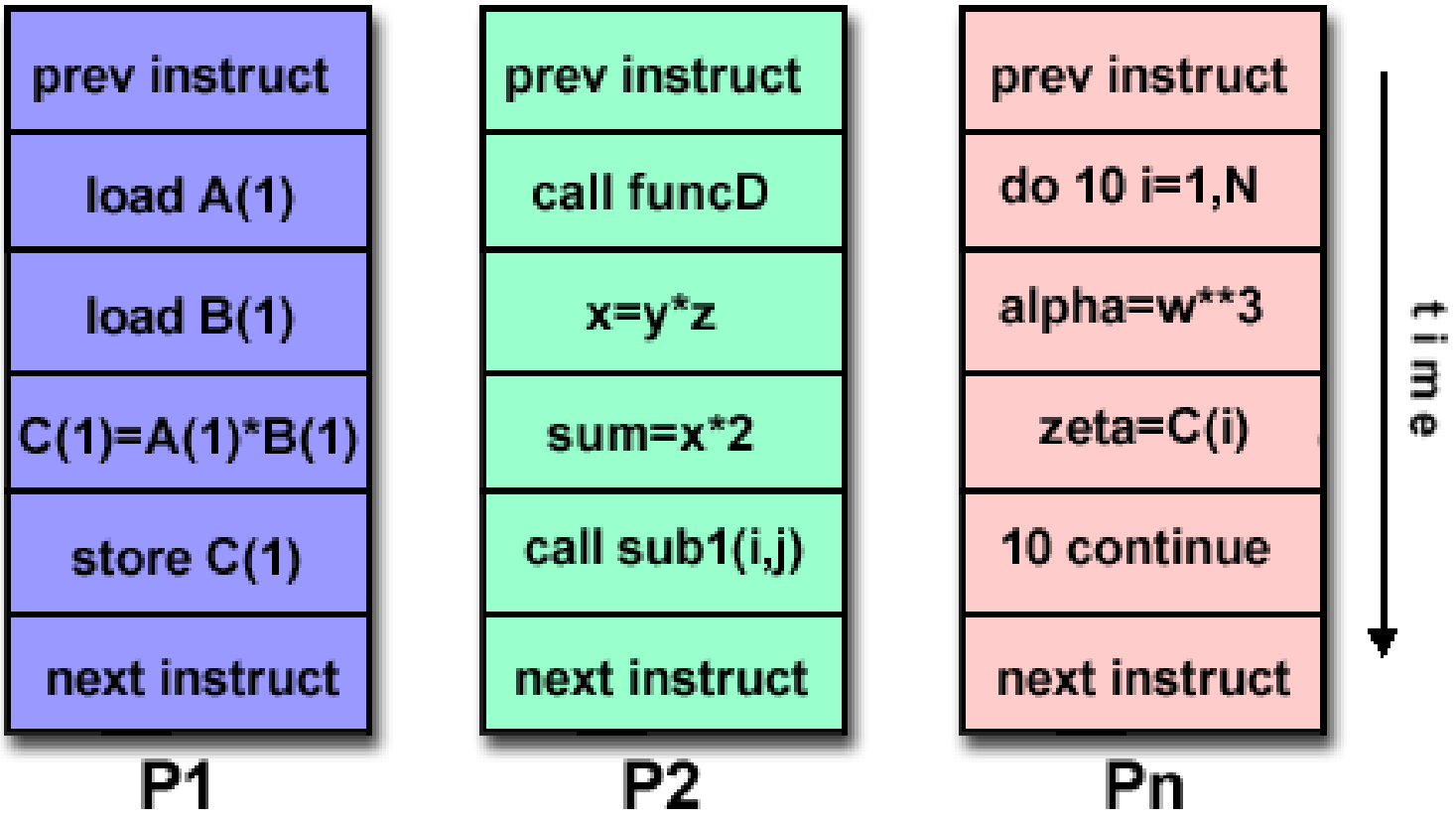
本堂课的视角



普遍联系—>具体化和形象化

- 主流的并行程序模型
  - 共享存储模型
  - 消息传递模型
  - 数据并行模型

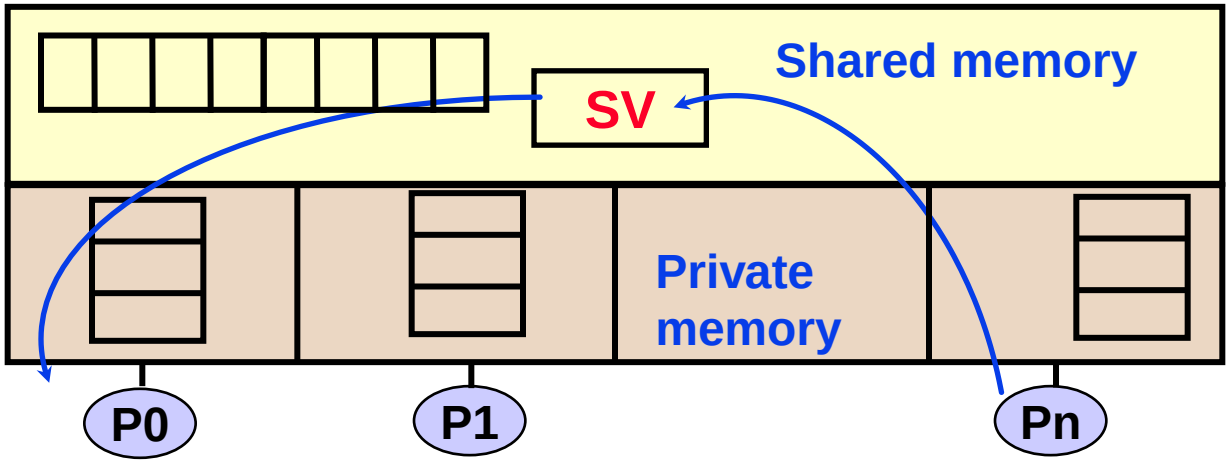
泛化的并行程序的执行模型(MIMD)



主流并行程序模型—共享存储(Shared Memory)

并行程序由一组进程/线程组成

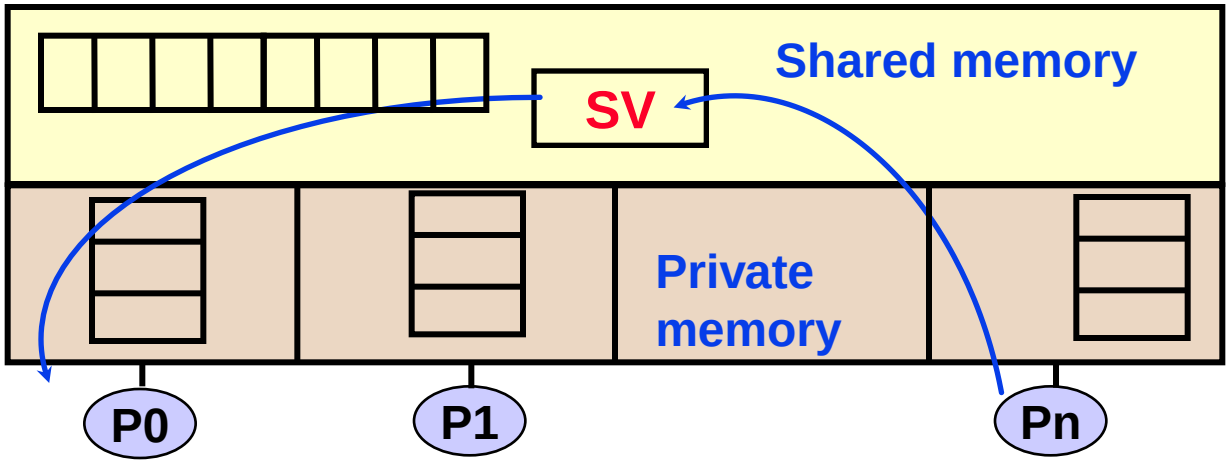
- 如何分配和管理数据？
- 如何通讯？
- 如何控制执行顺序？



主流并行程序模型—共享存储(Shared Memory)

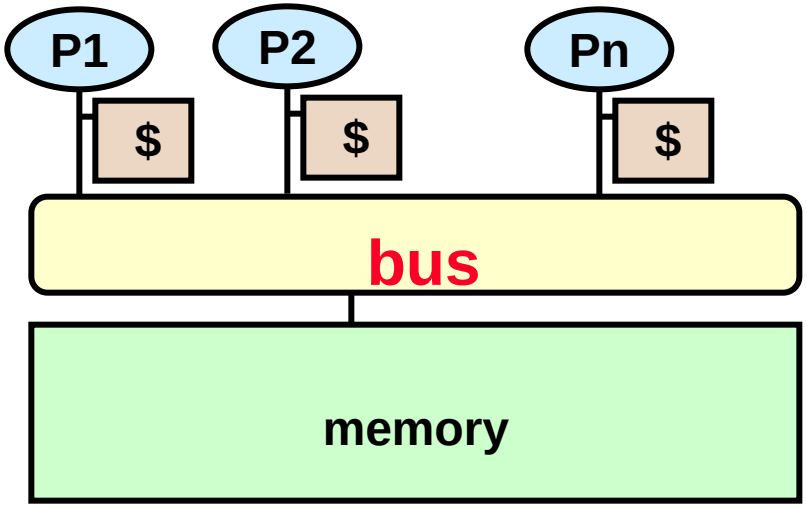
并行程序由一组进程/线程组成

- 数据驻留在单一地址空间，不需显式分配数据
- 每个进程/线程可以有局部变量和共享变量
- 通讯通过读写共享变量隐式完成
- 各个进程/线程异步执行
- 显式的同步来控制正确的执行顺序
- 线程可以动态产生和消亡



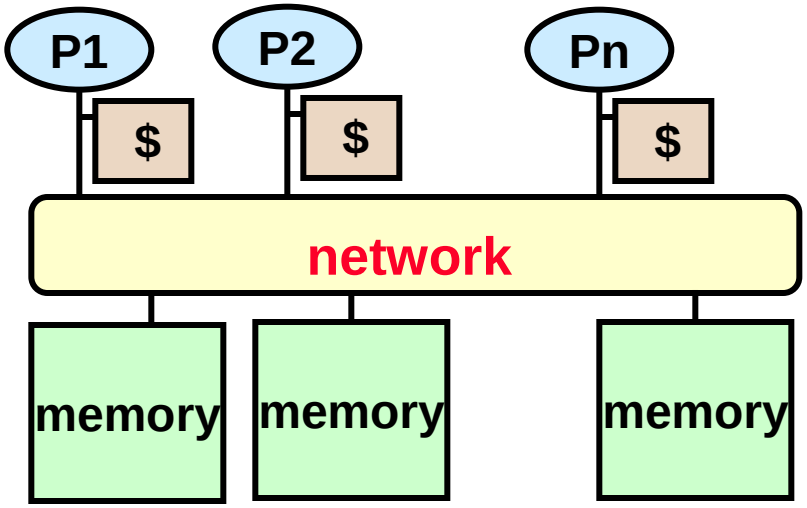
与共享存储模型相适应的并行系统(1)

- 所有处理器与大的共享内存相连
  - 对称多处理器系统 (SMPs)
  - **Sun, HP, Intel, IBM SMPs**
- 不足：难于扩展到大量处理器
  - 典型情况下，**<32** 处理器
  - 总线成为瓶颈
- 均匀存储访问 **(UMA)**：系统对称，处理器之间无主从之分，可等同地访问共享存储器和其他资源



与共享存储模型相适应的并行系统(2)

- 分布式共享存储系统(Distributed Shared Memory)
  - 存储器逻辑上共享但物理上分布
  - 处理器和内存通过网络互连
  - 任何处理器都可以访问任何的内存地址



SGI Origin(SGI® NUMAflex™)

- 可以建立更多处理器的系统(512处理器系统已面世)
- Cache一致性维护：需要有效地处理多处理器中相同数据的cache一致性

一个共享存储模型的例子

向量函数加和问题

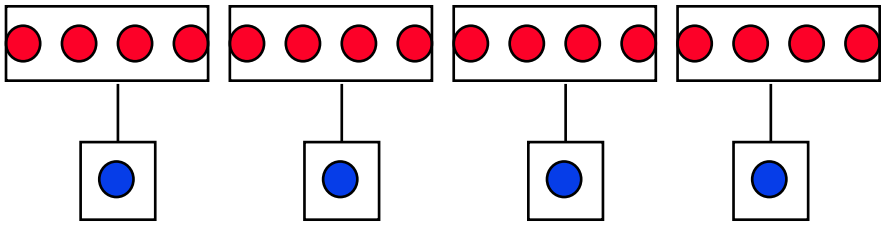
$$\sum_{i=0}^{n-1} f(A[i])$$

- 任务分解
- 数据分配
  - 共享数据
  - 私有数据

基于共享存储模型的例子

向量函数加和问题 $\sum_{i=0}^{n-1} f(A[i])$

- 任务分解：在p个进程上对n个元素平均分配



- 共享数据与私有数据
  - 共享数据：n、原有的n个元素、全局和
- 私有数据
  - 独立的函数赋值

基于共享存储模型的例子

```
static int s = 0;
```

Thread 1

```
for i = 0, n/2-1
    s = s + f(A[i])
```

Thread 2

```
for i = n/2, n-1
    s = s + f(A[i])
```

基于共享存储模型的例子

```
static int s = 0;
```

Thread 1

```
for i = 0, n/2-1
    s = s + f(A[i])
```

Thread 2

```
for i = n/2, n-1
    s = s + f(A[i])
```

```
static int s = 0;
```

Thread 1

```
….
 compute f([A[i]) and put in reg0
 reg1 = s
 reg1 = reg1 + reg0
 s = reg1
…
```

Thread 2

```
…
 compute f([A[i]) and put in reg0
  reg1 = s
  reg1 = reg1 + reg0
  s = reg1
 …
```

基于共享存储模型的例子

```
static int s = 0;
```

Thread 1		Thread 2	
….		…	
compute f([A[i]) and put in reg0	7	compute f([A[i]) and put in reg0	9
reg1 = s	27	reg1 = s	27
reg1 = reg1 + reg0	34	reg1 = reg1 + reg0	36
s = reg1		s = reg1	
…		…	

- 假设线程1上s=27, f(A[i])=7，而线程2上f(A[i])= 9

基于共享存储模型的例子

```
static int s = 0;
```

Thread 1		Thread 2	
….		…	
compute f([A[i]) and put in reg0	7	compute f([A[i]) and put in reg0	9
reg1 = s	27	reg1 = s	27
reg1 = reg1 + reg0	34	reg1 = reg1 + reg0	36
s = reg1	34	s = reg1	36
…		…	

- 假设线程1上s=27, f(A[i])=7，而线程2上f(A[i])= 9
- 正确结果 s=43
  - 但是结果却可能是 43，34，或 36

基于共享存储模型的例子

```
static int s = 0;
```

```
Thread 1

  for i = 0, n/2-1
      s = s + f(A[i])
```

```
Thread 2

  for i = n/2, n-1
      s = s + f(A[i])
```

- 错误来源
  - 共享变量s的并发访问，引发竞争
- 竞争在共享存储编程中非常常见，是确保正确性的关键
  - 两个处理器(或两个线程) 访问一个变量，且其中至少有一个写者
  - 两个访问同时发生

基于共享存储模型的例子

```
static int s = 0;
static lock lk;
```

Thread 1

```
local_s1= 0
for i = 0, n/2-1
    local_s1 = local_s1 + f(A[i])
lock(lk);
s = s + local_s1
unlock(lk);
```

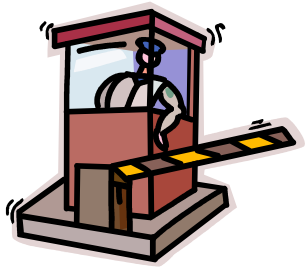
Thread 2

```
local_s2 = 0
for i = n/2, n-1
    local_s2= local_s2 + f(A[i])
lock(lk);
s = s +local_s2
unlock(lk);
```

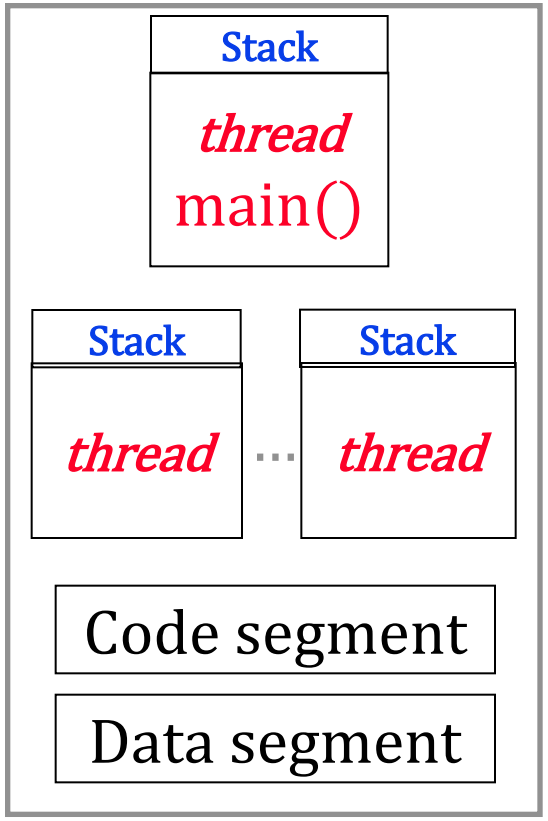
- 一种解决的办法—引入局部和
  - 主要的计算在局部变量上完成
  - 共享变量的访问次数减少，提高计算速度，降低了发生竞争的概率
  - 仍然存在竞争风险，需要采用同步策略解决竞争

共享存储模型下解决竞争的办法

- 互斥(**Mutual Exclusion**)
  - 临界区：访问共享变量的代码段
  - 互斥：任一时刻强制要求单线程访问临界区的代码
  - 例子: 保险箱，服务员确保保险箱使用的互斥
- 同步对象用于确保互斥
  - Lock, semaphore, critical section, event, condition variable, atomic
  - One thread "holds" sync. object; other threads must wait
  - When done, holding thread releases object; some waiting thread given object



Processe vs. Thread



- 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
  - Each thread gets its own stack
- All threads within a process share code & data segments

好处到底是什么？

Why Use Threads

- Benefits
  - Efficient data sharing
    - Sharing data through memory more efficient than message-passing
  - Increased performance
    - Easy method to take advantage of multi-core
  - Better resource utilization
    - Reduce latency (even on single-core processor systems)
- Risks
  - Increases complexity of application
  - Difficult to debug (data races, deadlocks, etc.)

共享存储并行编程环境

- Posix threads(Portable Operating System Interface)
  - 需要编写并行程序，通过函数库调用
  - 面向C语言

功能	含义
pthread_create	线程生成
pthread_exit	线程退出
pthread_join	线程合并
pthread_self	返回线程ID
pthread_mutex_init	生成新互斥变量
pthread_mutex_destroy	释放互斥变量
pthread_mutex_lock	锁住互斥变量
pthread_mutex_trylock	试探锁住互斥变量
pthread_mutex_unlock	解锁互斥变量

共享存储并行编程标准－OpenMP

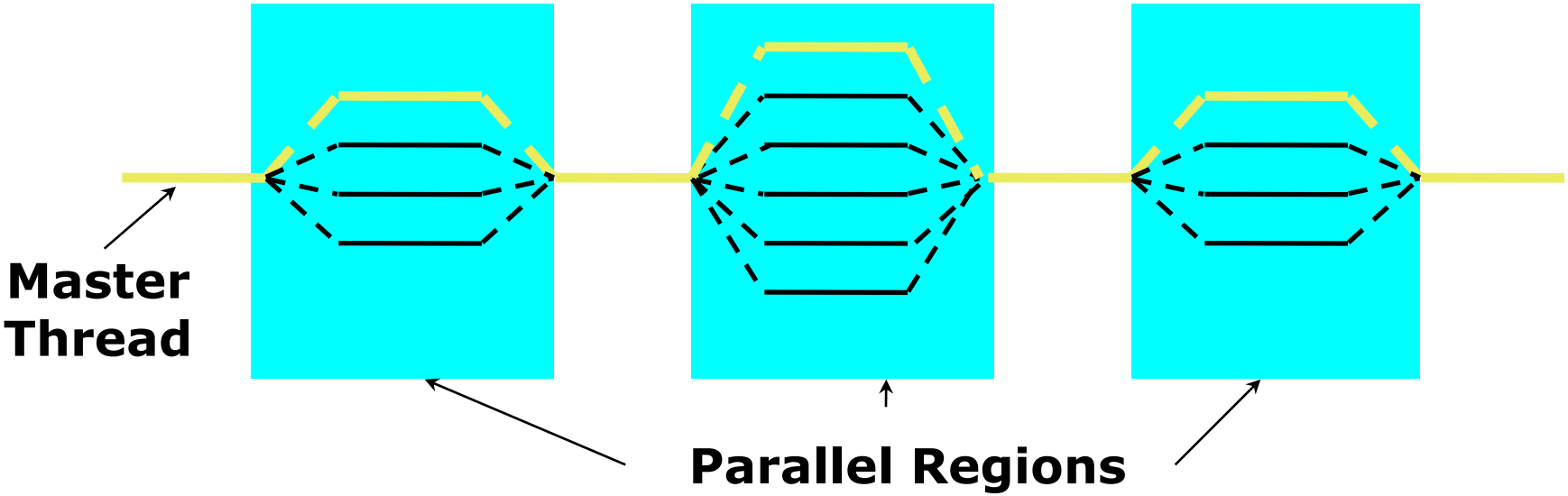
- OpenMP是一种面向共享存储的并行程序设计标准
  - OpenMP形成于1996年，Fortran/C/C++的OpenMP API最新的标准都是3.0
  - Microsoft，Intel，IBM，HP和SUN等公司都是其支持者，已经成为一个事实上的工业标准

- OpenMP的实现不是一种编程语言，而主要采用指导语句(Directives)的方式，编程需要编译器支持，其实现被IBM，DEC, SGI，Intel和PGI等公司的产品支持

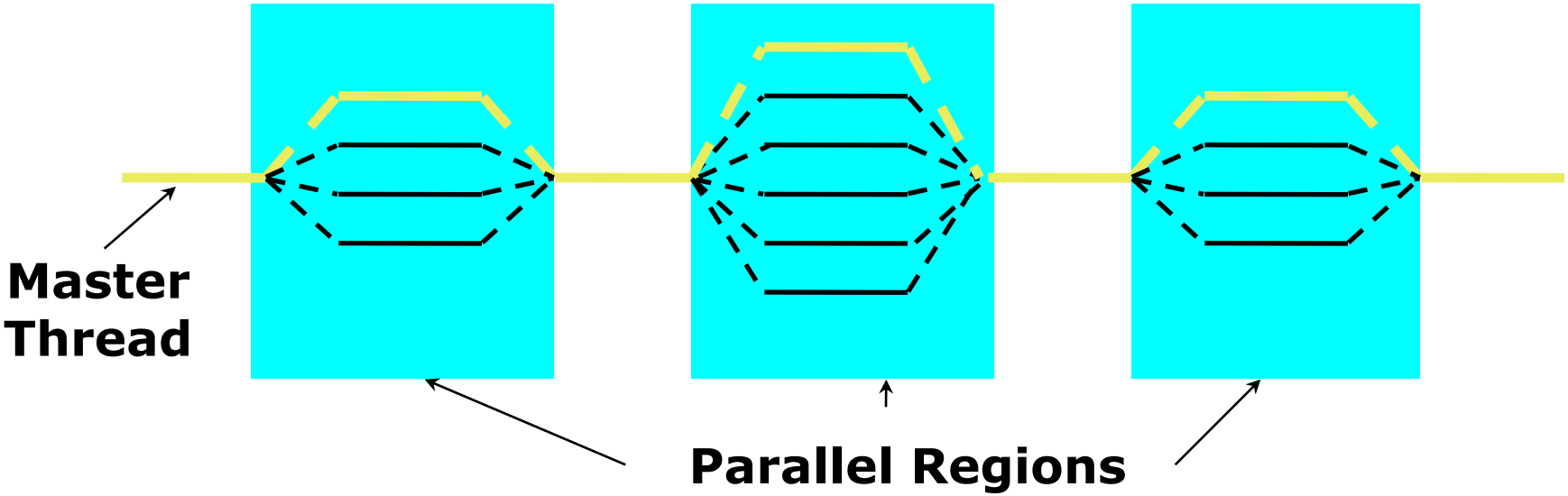
OpenMP的主要特点

- 面向共享存储体系结构，特别是SMP系统
- 基于fork-join的多线程执行模型，但同样可以开发SPMD（Single Program Multi-Data)类型的程序
- 可以进行增量式并行开发(Incremental development)，支持条件编译(Conditional Compilation)和条件并行。
- 允许嵌套的并行性（nested Parallelism)
  - 并不是在所有的编译器实现中支持
- OpenMP支持多种并行方式
  - 数据与控制并行
- 支持多种同步结构
  - MASTER，CRITICAL，BARRIER，ATOMIC

OpenMP的程序结构



OpenMP的程序结构



OpenMP需要支持哪些功能？

1 定义并行区

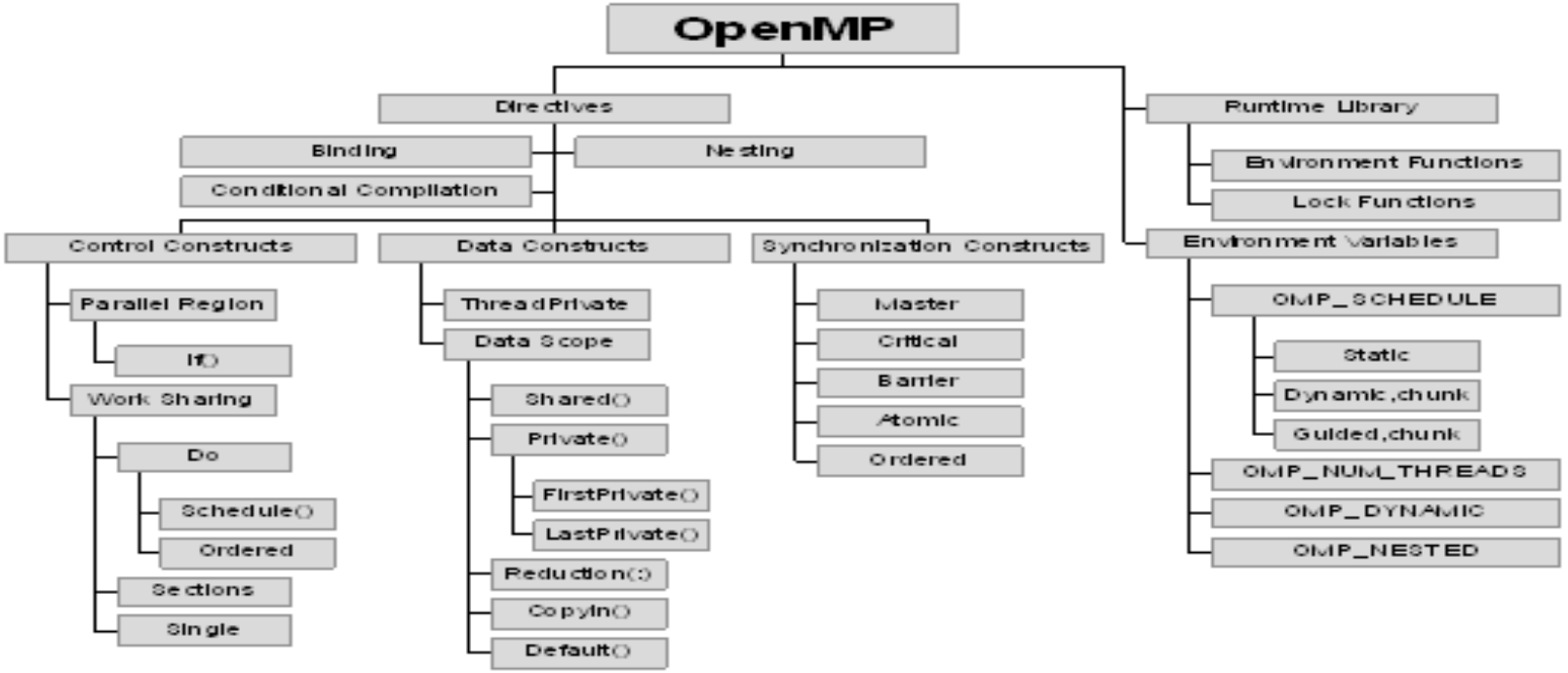
2 设置并行度

3 并行结构

4 并行区数据管理

5 同步机制

1.A.3 OpenMP Constructs



1. Overview 16

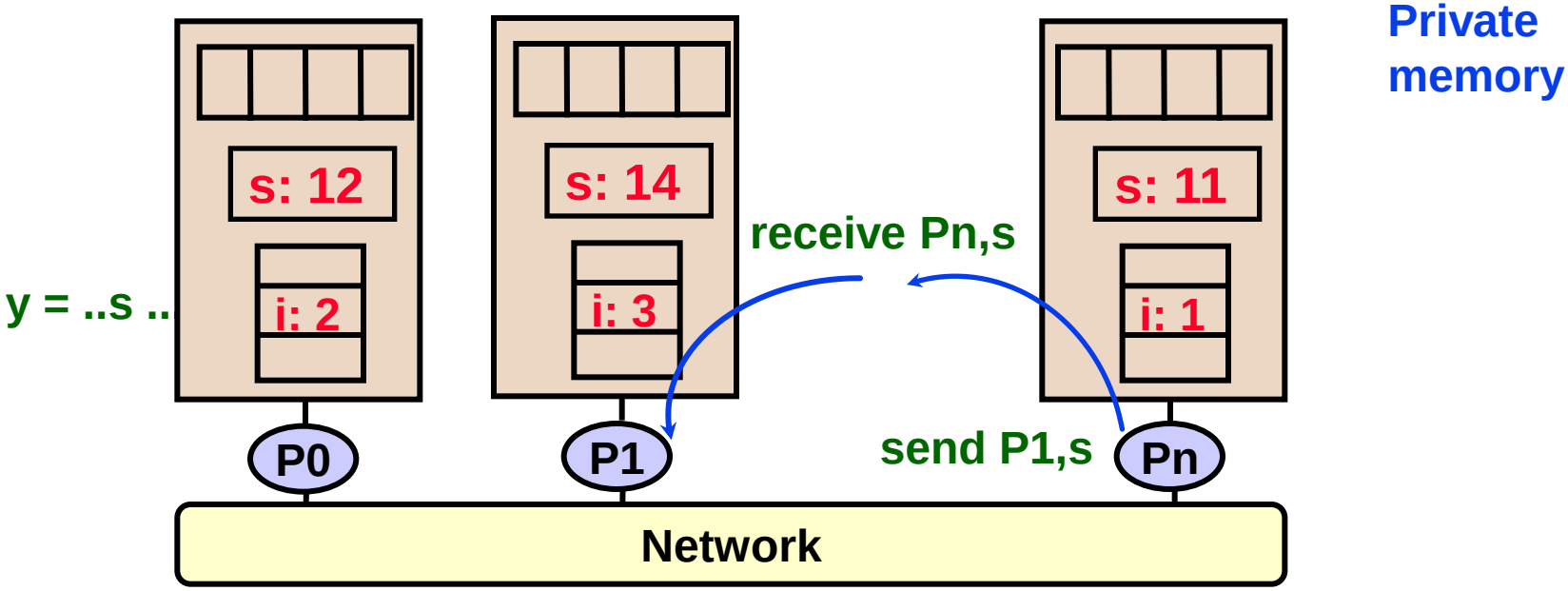
- **Parallel and work sharing directives**
- **data environment directives**
- **synchronization directives**

- 主流的并行程序模型
  - 共享存储模型
  - 消息传递模型
  - 数据并行模型

主流并行程序模型—消息传递(Message Passing)

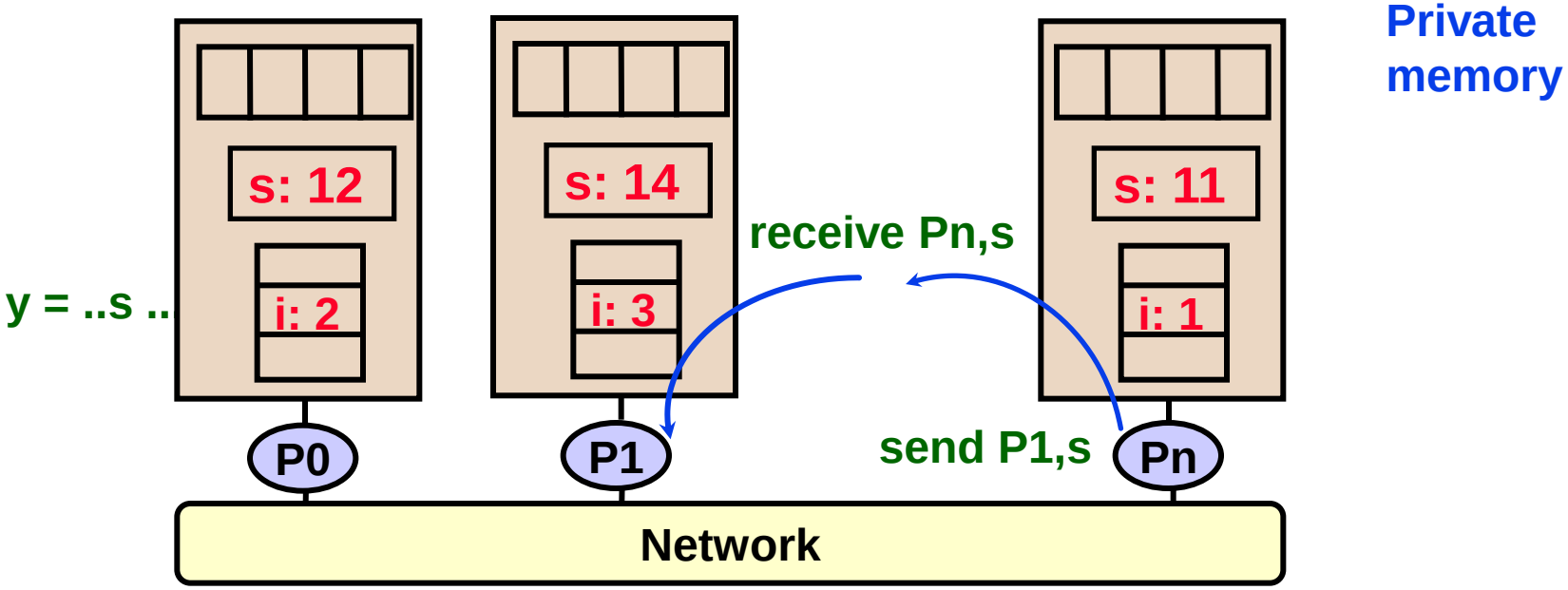
并行程序由一组命名进程组成

- 如何分配和管理数据？
- 如何通讯？
- 如何控制执行顺序？



主流并行程序模型—消息传递(Message Passing)

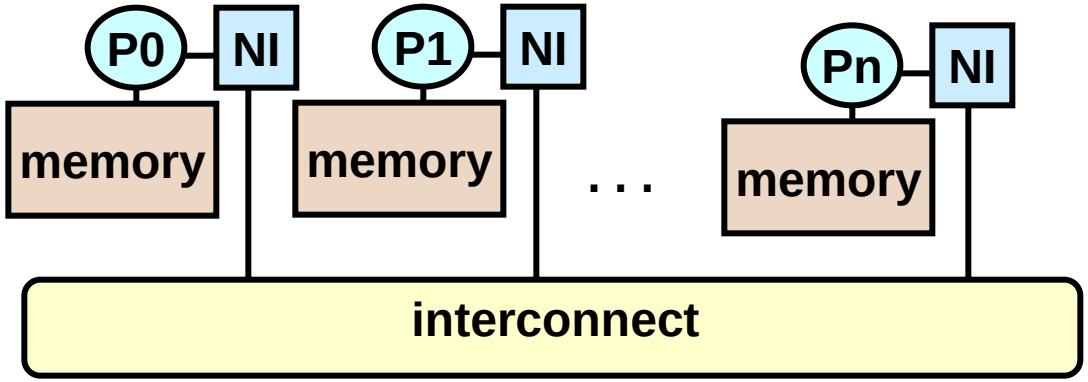
- 并行程序由一组命名进程组成
  - 分开的地址空间：没有共享数据，数据和负载显式分配
  - 进程间通信采用显式的send/receive组
  - 显式同步以确保执行顺序
  - 多进程，异步并行
  - 进程数多在程序启动时确定

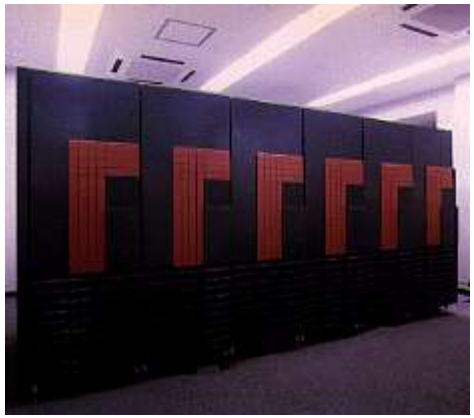


与消息传递模型相适应的并行系统

分布存储系统

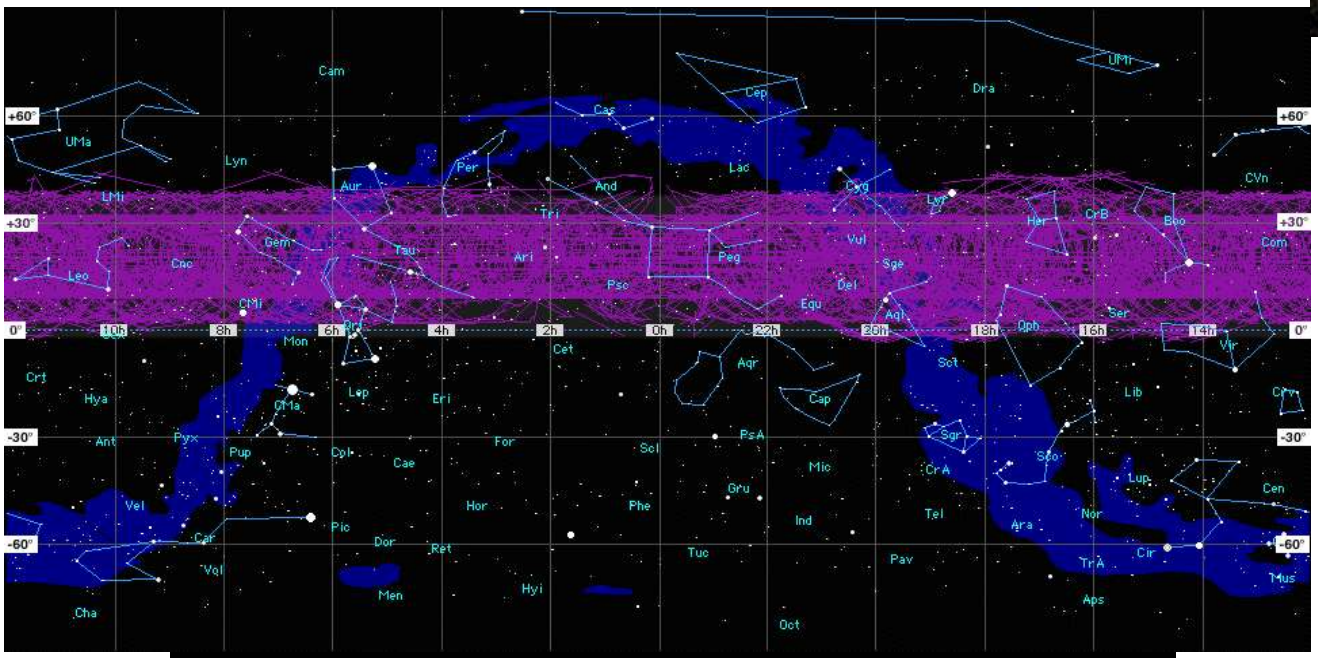
- 每个处理器拥有自己的内存和cache，不能访问其他处理器的数据
- 每个节点通过网络接口完成通信和同步
- 典型系统：
  - **Cray T3E, IBM SP2**
  - **机群系统 (Berkeley NOW, Beowulf)**

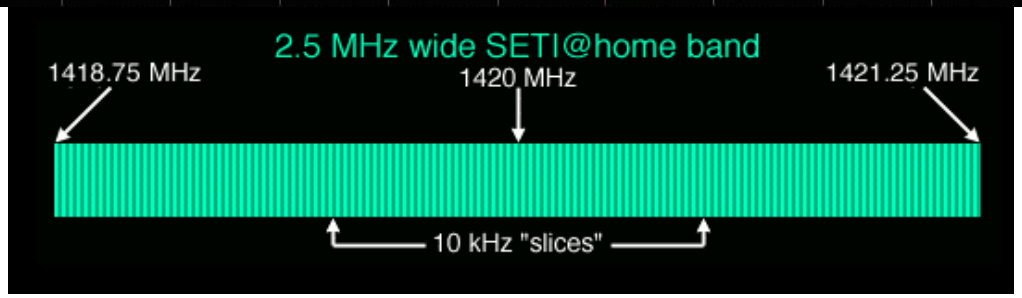




Internet/Grid Computing

- SETI@Home: Running on 500,000 PCs
  - ~1000 CPU Years per Day
  - 485,821 CPU Years so far
- Sophisticated Data & Signal Processing Analysis
- Distributes Datasets from Arecibo Radio Telescope





Next Step-
Allen Telescope Array

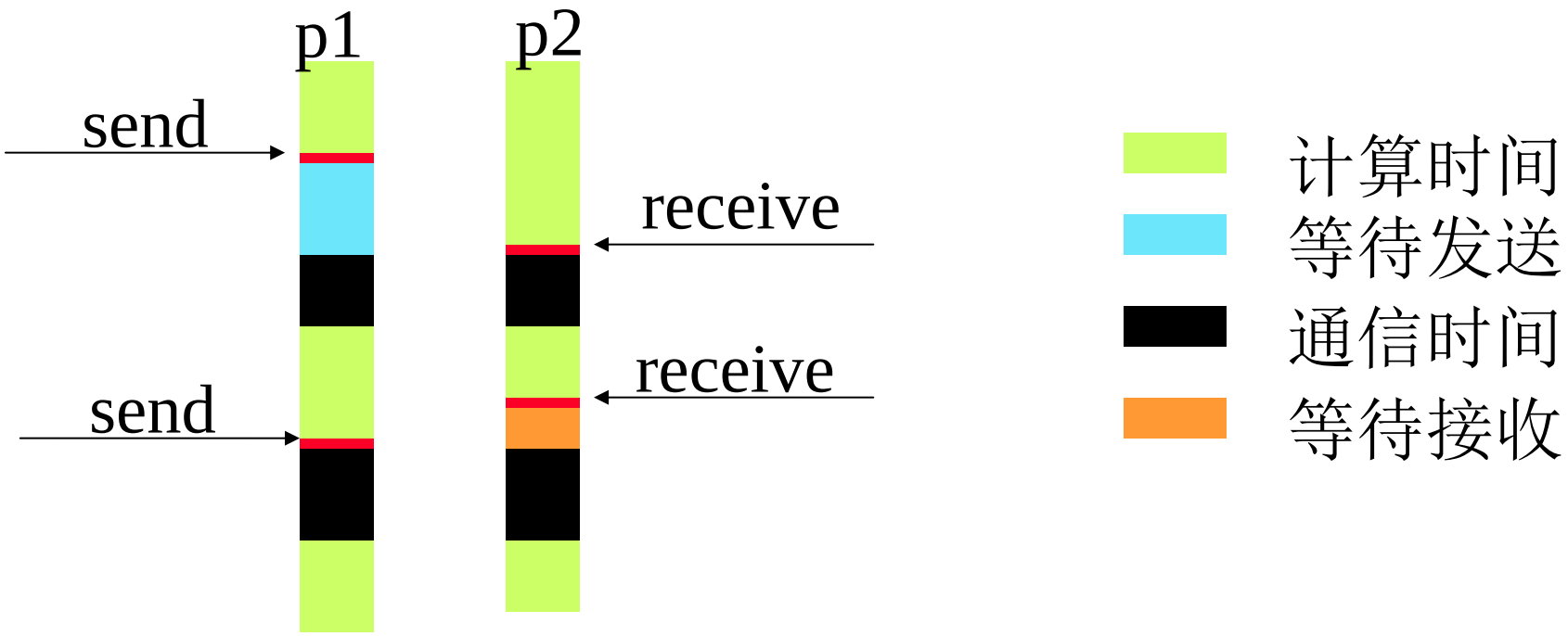
消息传递模型的特点与发展现状

- 消息传递模型的特点
  - 普遍适用

- 消息传递模型的现状
  - 消息传递模型在并行程序设计中广泛采用
  - 消息传递程序相对简单的硬件和软件的要求，通常会增加消息传递程序本身的开发的复杂性

阻塞的通信操作

- 消息传递程序的基本操作是发送和接收消息。
- 两种基本消息通信形式：
  1. 阻塞式通信：电话系统
  - 非阻塞式通信：邮局



基于消息通信模型的例子–s = A[1]+A[2]

求解策略 – 对吗?

```
Processor 1
    xlocal = A[1]
    send xlocal, proc2
    receive xremote, proc2
    s = xlocal + xremote
```

```
Processor 2
    xlocal = A[2]
    send xlocal, proc1
    receive xremote, proc1
    s = xlocal + xremote
```

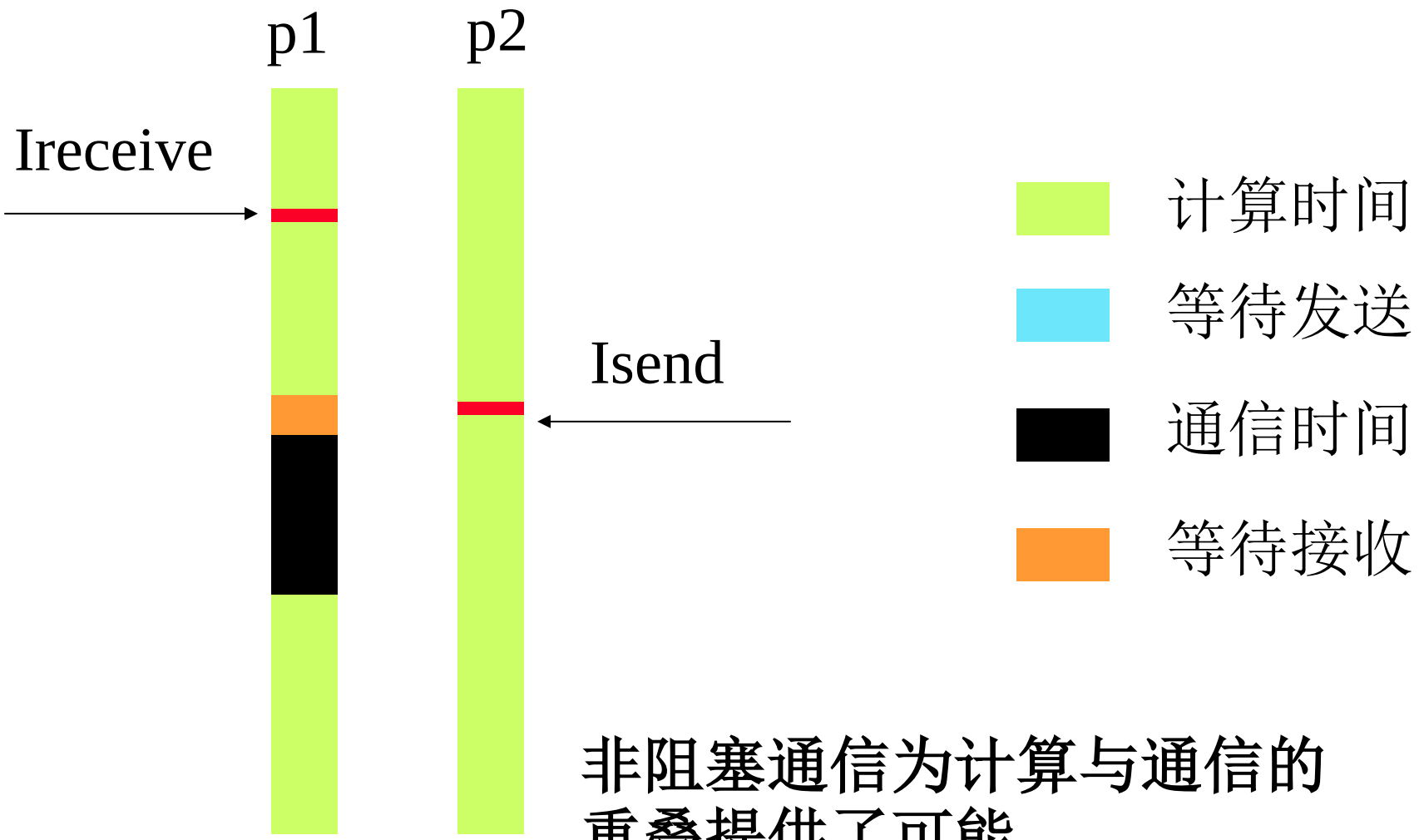
- 阻塞通信情况下的正确求解策略

```
Processor 1
    xlocal = A[1]
    send xlocal, proc2
    receive xremote, proc2
    s = xlocal + xremote
```

```
Processor 2
    xloadl = A[2]
    receive xremote, proc1
    send xlocal, proc1
    s = xlocal + xremote
```

• 两种基本消息通信形式：

2. 非阻塞式通信：邮局



非阻塞通信为计算与通信的重叠提供了可能

破坏了严格的同步语义

消息通信并行编程标准–Message Passing Interface

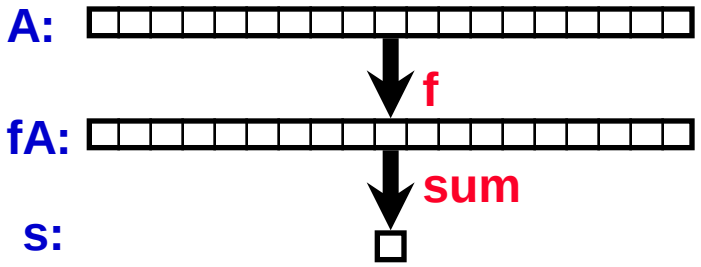
- 目前应用最为广泛的消息传递程序设计标准，也是实际意义上的并行计算编程标准
- 语言绑定：**MPI的实现是一个库**而不是一种程序语言，因此对**MPI**的使用必须和特定的程序语言结合起来进行

功能	含义
mpi_init	启动MPI
mpi_finalize	结束MPI计算
mpi_comm_size	确定进程数
mpi_comm_rank	返回进程ID
mpi_send	发送一条消息
mpi_recv	接收一条消息

主流并行程序模型—数据并行(Data Parallel)

- 单线程模式
  - 并行操作于聚合数据结构上，一般是数组
  - 隐式相互作用，不需要显式同步
  - 隐式数据分配

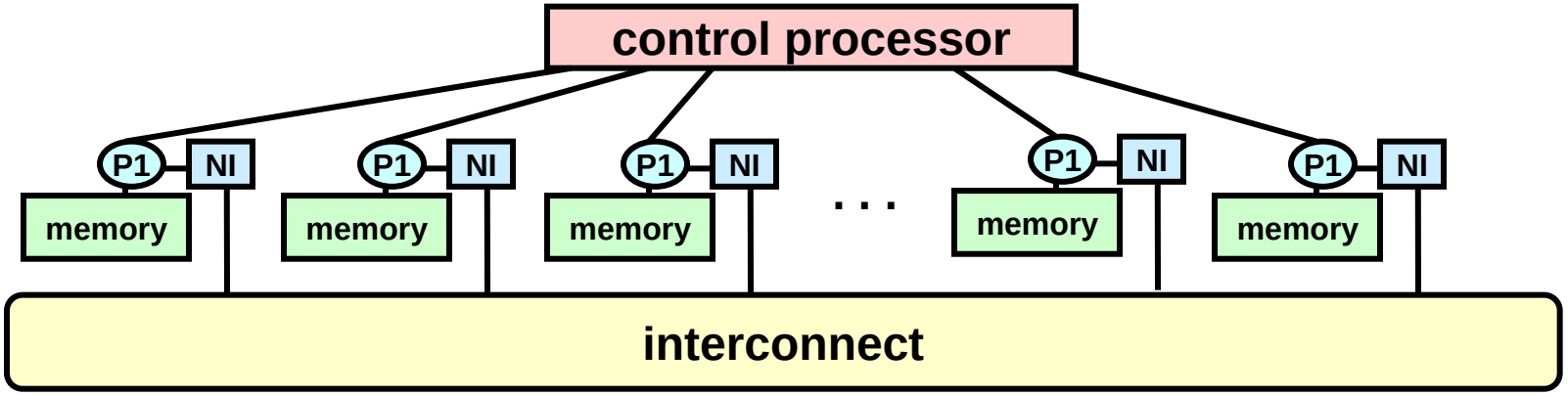
```
A = array of all data
fA = f(A)
s = sum(fA)
```



- 缺点
  - 不是所有应用模式都适合这种模型
  - 在粗粒度的并行机上难于映射

与数据并行模型相适应的并行系统(1)

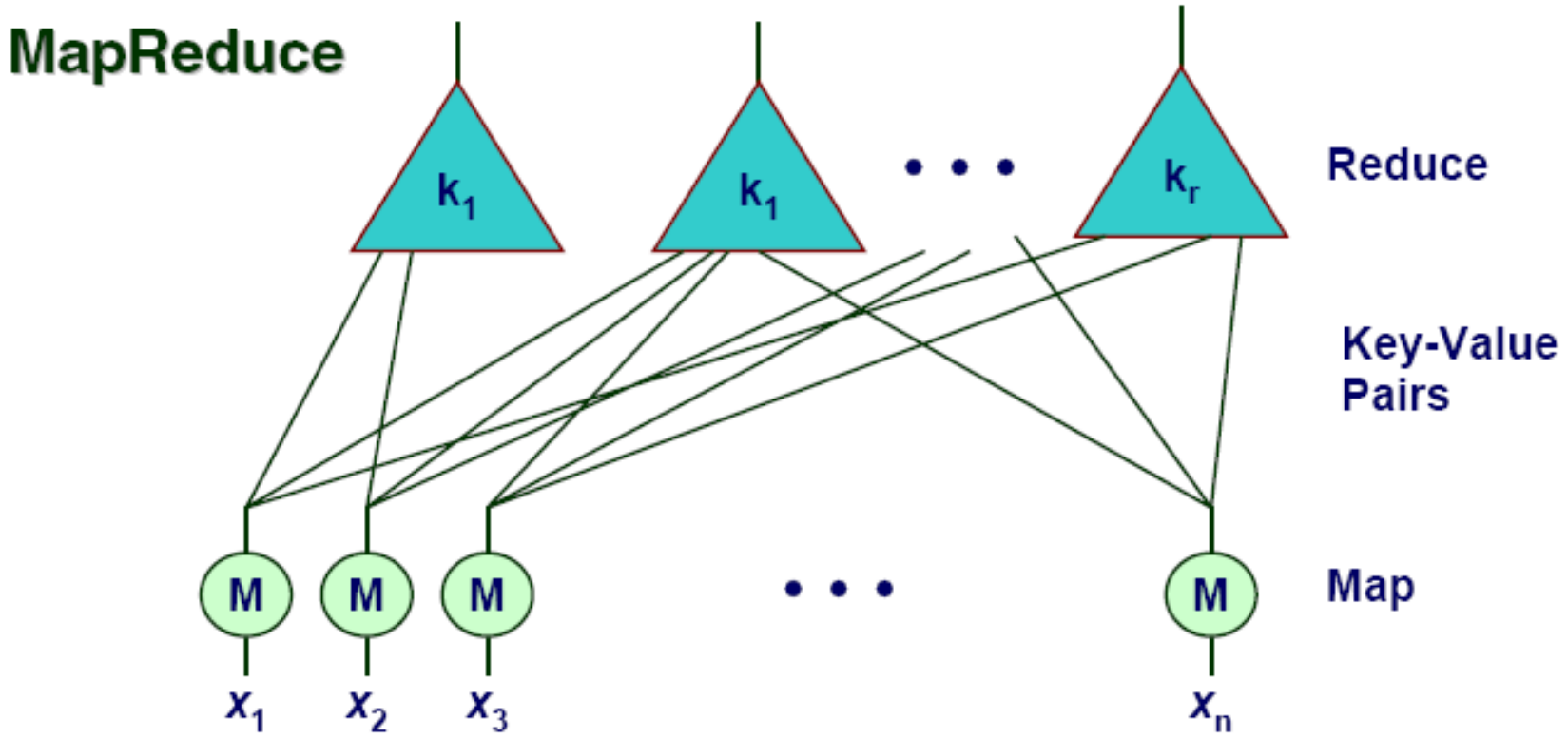
- 单指令多数据系统(SIMD)
  - 大量的“小”处理器
  - 单个控制处理器负责分发指令
  - 每个处理器执行相同的指令
  - 参与计算的处理器可调整
- 多为面向科学计算的专用系统，不流行
- 编程模型通过编译器实现，如：HPF



与数据并行模型相适应的并行系统(2)

- 并行向量处理机
  - 曾经非常重要，后来逐渐被取代
  - 近年来重新出现
    - 地球模拟器 **Earth Simulator (NEC SX6)**
    - **SIMD extensions to microprocessors**
      - **SSE(Streaming SIMD Extension), SSE2, SSE3 (Intel: Pentium/IA64) …**
      - **Altivec (IBM/Motorola/Apple: PowerPC)**
      - **VIS (Sun: Sparc)**

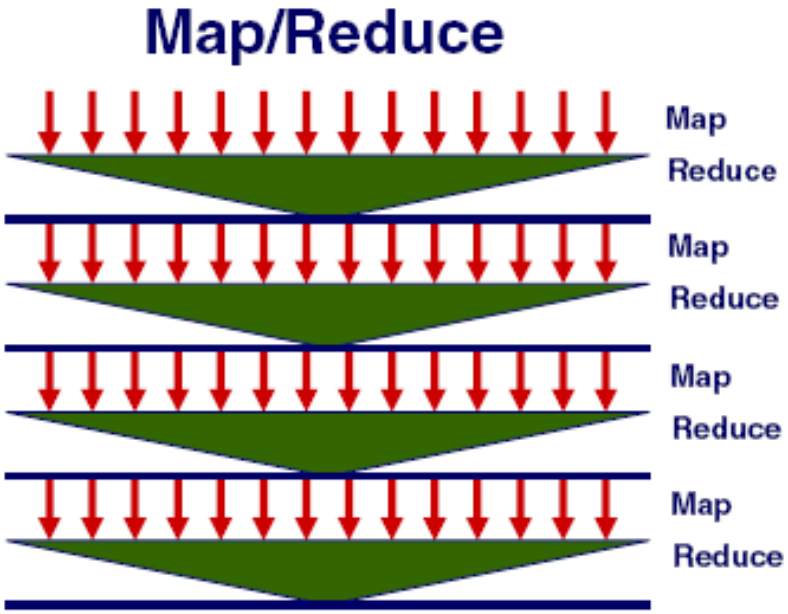
MapReduce Programming Model



- Map computation across many objects
  - E.g., 10^{10} Internet web pages
- Aggregate results in many different ways
- System deals with issues of resource allocation & reliability

Dean & Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters", OSDI 2004

Map/Reduce Operation



Characteristics

- Computation broken into many, short-lived tasks
  - Mapping, reducing
- Use disk storage to hold intermediate results

Strengths

- Great flexibility in placement, scheduling, and load balancing
- Handle failures by recomputation
- Can access large data sets

Weaknesses

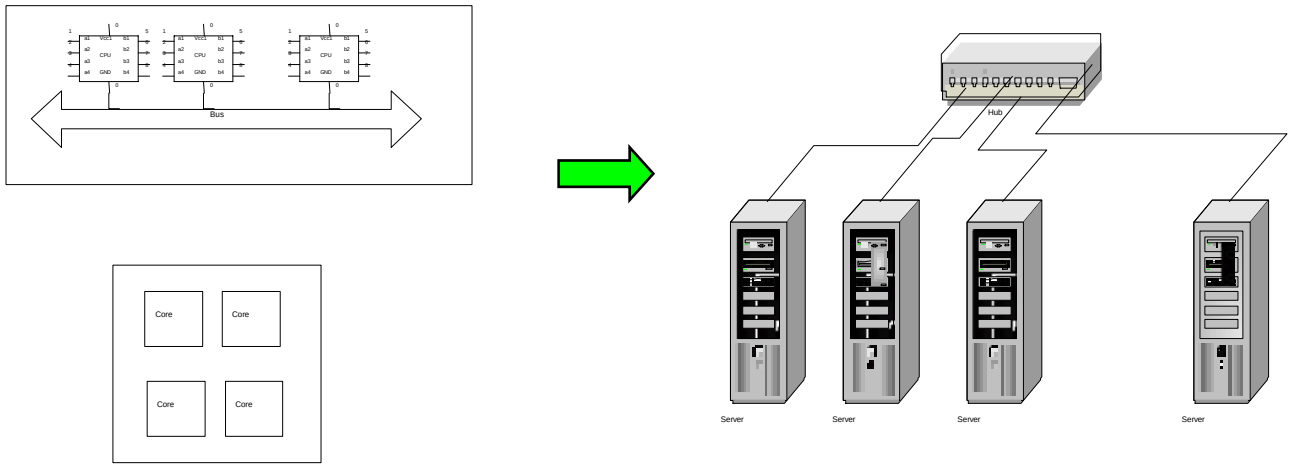
- Higher overhead
- Lower raw performance

小结(1)

- 并行程序模型的目标
  - 屏蔽并行计算系统细节，并进行合理抽象
  - 支持开发通用的并行代码
- 共享存储模型与消息传递模型是如今常用的并行程序模型
  - 共享存储模型：隐式通信＋显式同步—>竞争
  - 消息传递模型：显式的通信和同步，阻塞与非阻塞通信，复杂度高
- 消息传递模型是更加通用的模型，MPI目前具有最好可移植性

小结(2) – 混合的程序模型

- 并行程序模型的研究仍是一个开放的研究领域
  - 仍然存在多种并行程序模型，融合的难度大
  - 新的体系结构仍在推出，需要与之相适应的程序模型
    - **CLUSTER of SMPs, CLUMP**，多核
    - 许多现代的高性能计算机是**CLUMP**：**IBM SPs, Blue Gene**
    - 可以采用消息传递模型，简单但忽略了不同的存储层次
    - **SMP**节点采用共享存储模型，而节点间采用消息传递



特别说明

- 下周内容
  - 介绍FIT集群计算环境的使用方法
- 希望和大家讨论的问题
  - 为什么选这门课？
  - 大家希望讲什么内容？

参考资料

- LLNL Tutorials: **Introduction to Parallel Computing**

- **希望能开始自学OpenMP部分**