

第二讲 概述(2)

- 概念：并行计算机、并行计算机模型
- 并行程序开发考虑
 - 并行程序开发环境
 - 问题的并行性
 - 计算任务的划分
 - 计算任务的通行、同步
 - 数据相关性、负载均衡问题
 - 我们能从并行计算技术期待什么？

并行计算机

- 通常，一个计算系统的组成包括两部分——硬件资源和软件资源
- A *parallel computer* is a set of processors that are able to work cooperatively to solve a computational problem

高层软件，例如用户程序

第三层软件，例如编译和运行库

第二层软件，例如**system functions**

第一层软件，例如**OS**内核

硬件资源

并行计算机两方面的含义

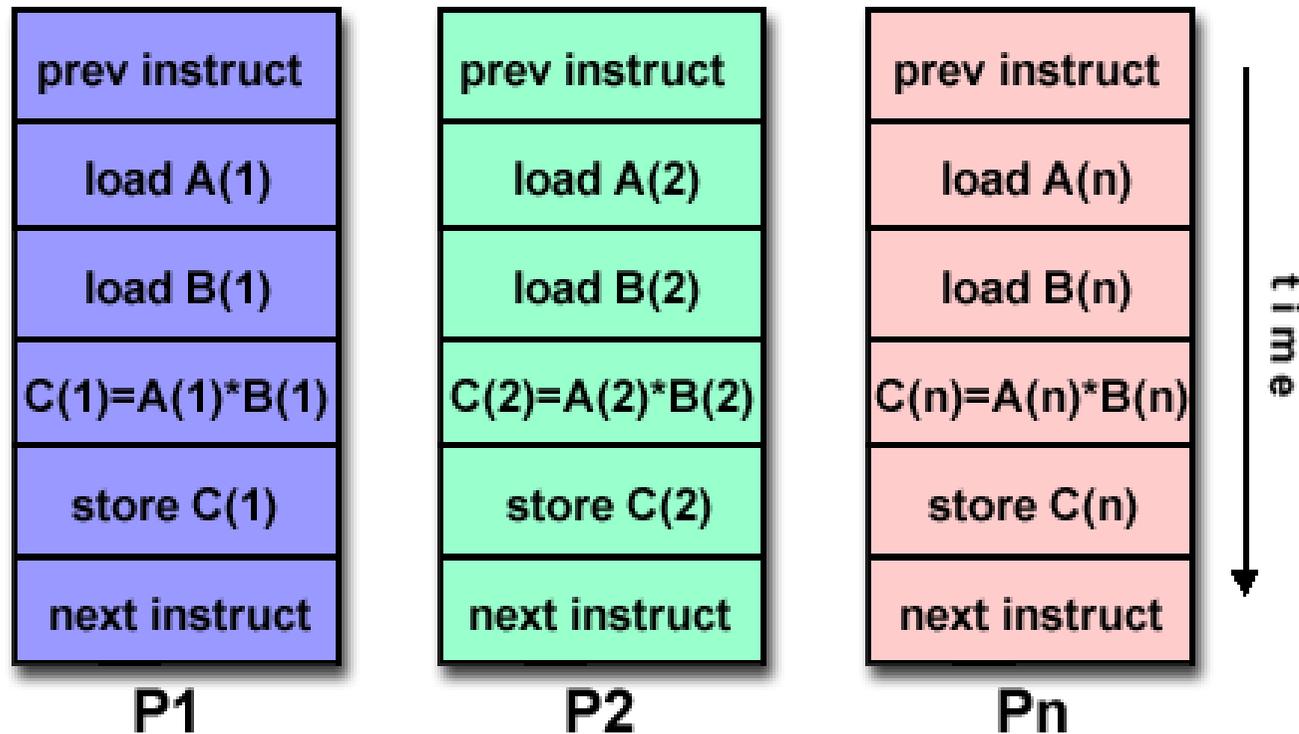
- 从硬件的**拓扑结构**的角度看
 - 共享存储体系结构：UMA、NUMA
 - 分布存储体系结构：CLUSTER (UP)
 - 分布-共享存储体系结构：CLUSTER (SMP/multi-core)
- 从硬件的**行为特征**的角度看
 - MISD
 - SIMD
 - MIMD

并行计算机的分类：Flynn分类法(行为特征)

<p>S I S D</p> <p>Single Instruction, Single Data</p> <p>串行计算机(von Neumann计算机)</p>	<p>S I M D</p> <p>Single Instruction, Multiple Data</p> <p>适用性很有限(如MPEG类计算、字符串匹配计算)</p>
<p>M I S D</p> <p>Multiple Instruction, Single Data</p> <p>为分类的的完美而设置，意义不大</p>	<p>M I M D</p> <p>Multiple Instruction, Multiple Data</p> <p>常见的并行计算机都可归入此类 MPP/Cluster/SMP/当前基于Cache的Multi-core (Intel、AMD)</p>

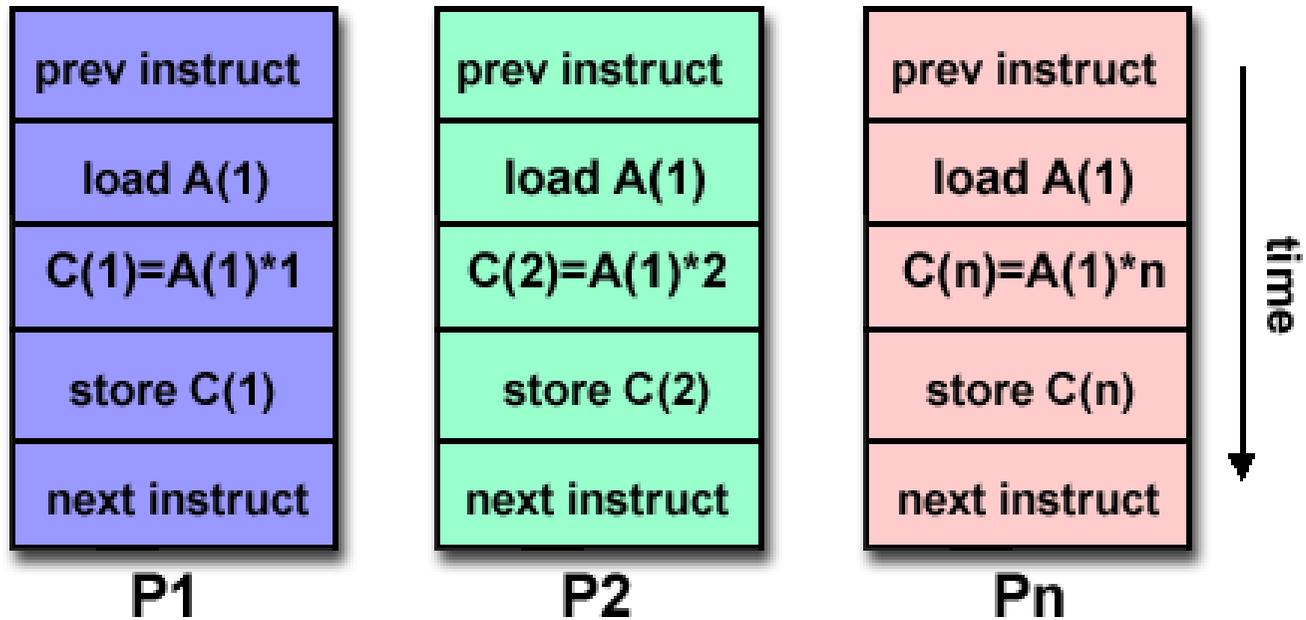
SIMD

- 处理器阵列机、向量机：CELL、GPU
- 适用于非常规则的计算，例如：视频、音频处理的MPEG算法；密集矩阵的运算



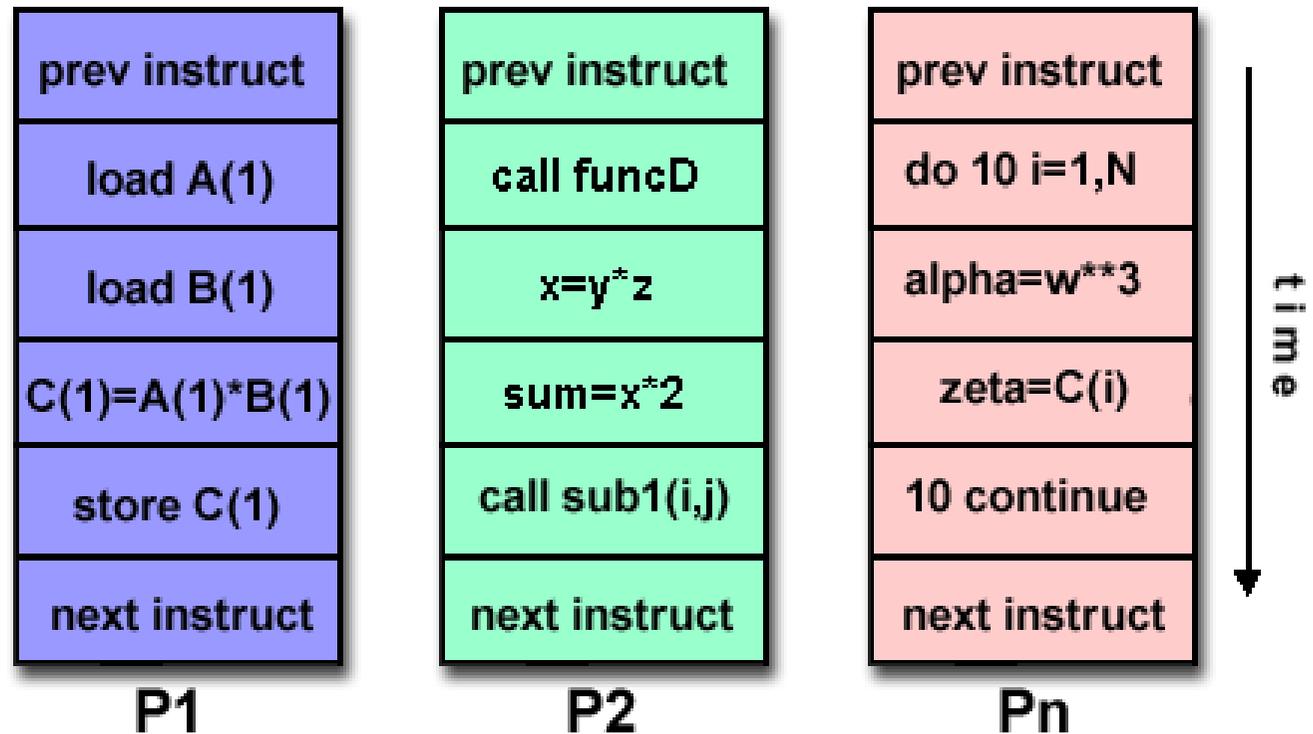
MISD

- 很少见



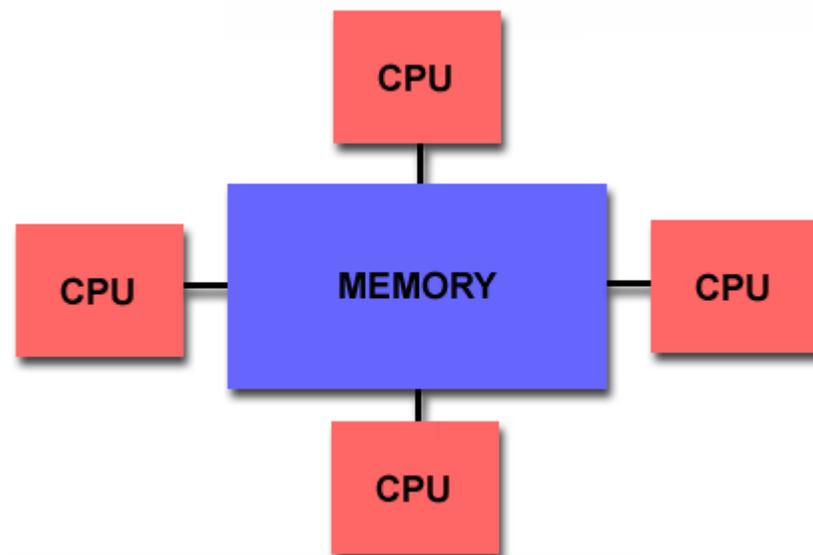
MIMD

- 最常见的并行计算机

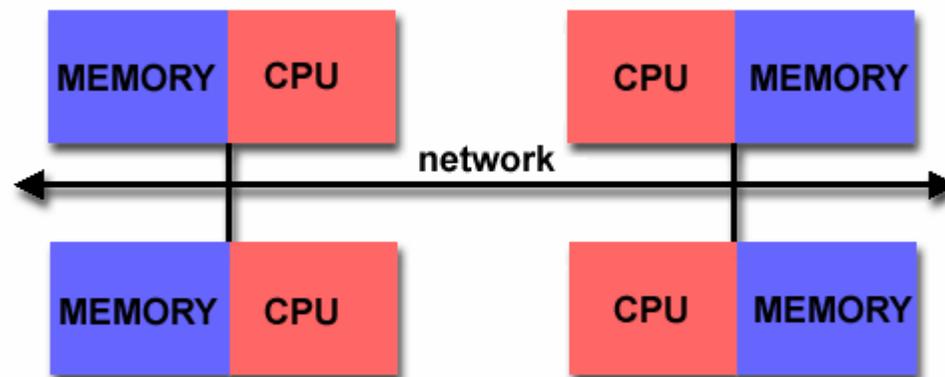


常见的并行机存储体系结构

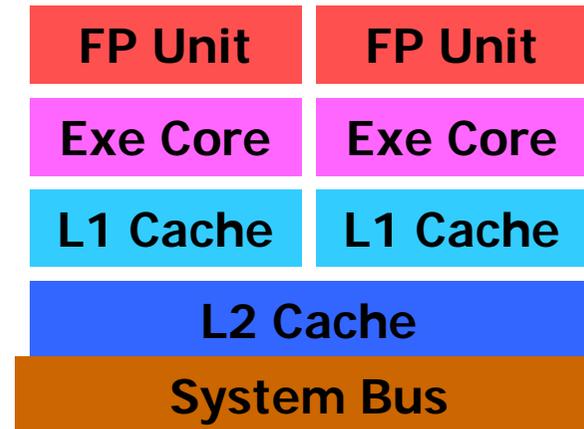
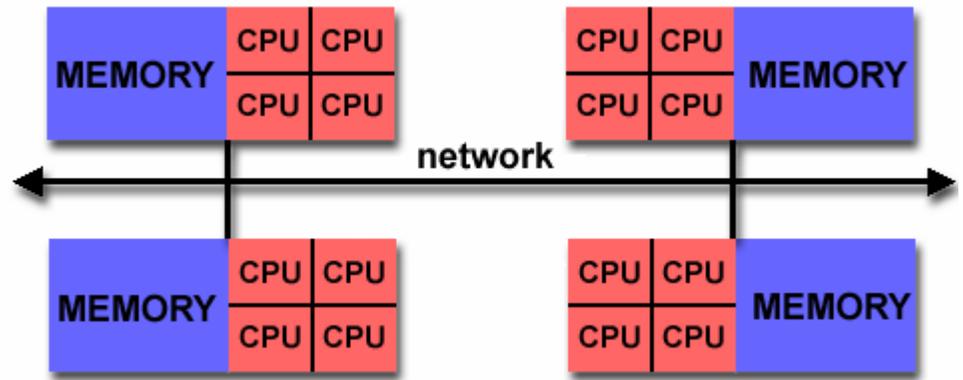
- 共享存储体系结构
 - UMA(当前主要是SMP)
 - NUMA(通常多个SMP互联)



- 分布存储体系结构



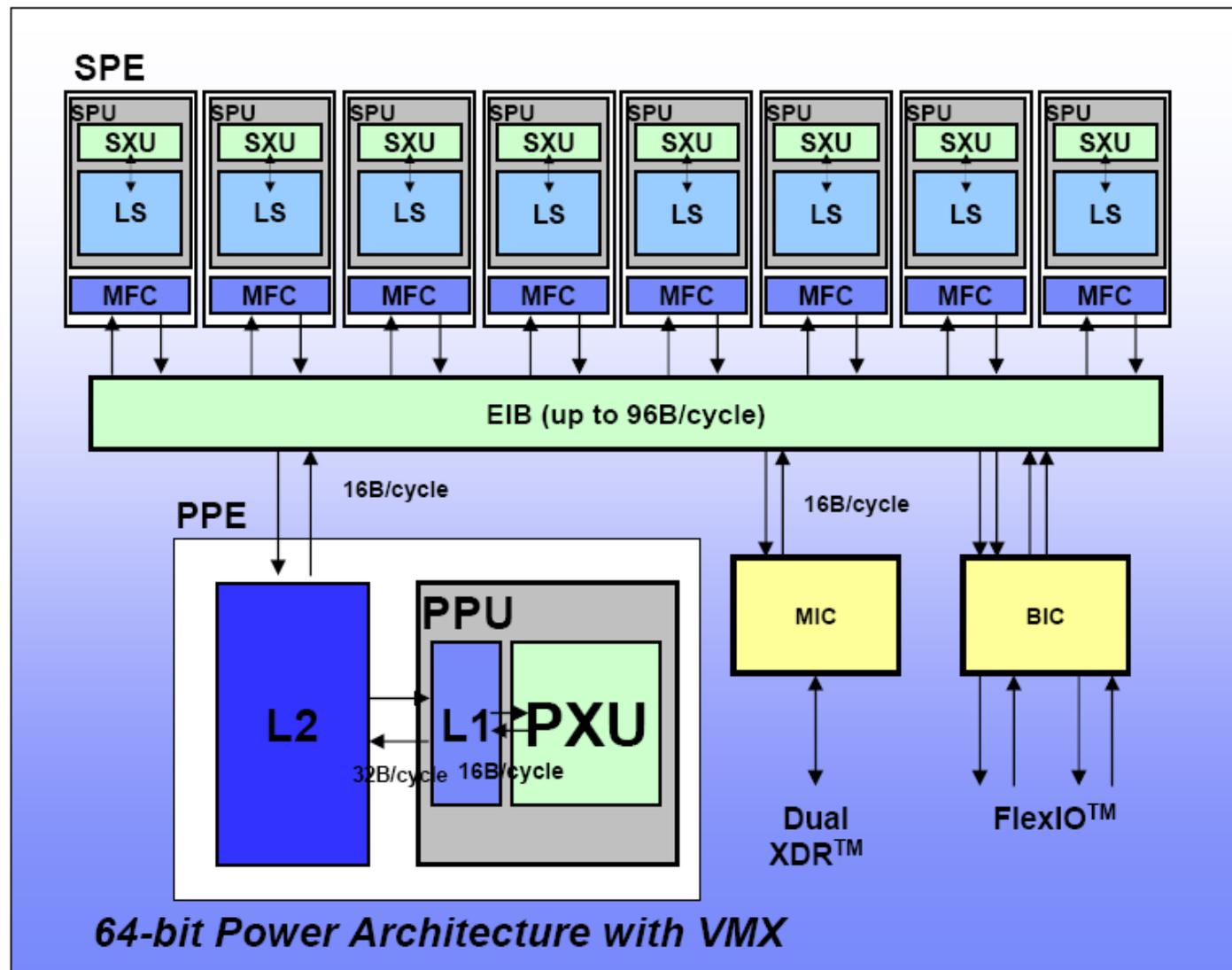
- 分布-共享的混合存储体系结构
 - 当前、以及今后一段时间里的并行机主流结构
- 北大科学工程中心的两台超级计算机均采用该体系结构
 - IBM RS/6000 SP3: 4节点, 每个节点16颗CPU
 - HP Cluster: 128节点, 每个节点2颗CPU
- 基于Cache的multi-core
 - 一个CPU中有多个执行核(execution core)
 - 每个执行核有自己独立的一级cache



Intel Core微架构

基于DMA的multi-core

Cell BE Block Diagram



硬件的行为取决于软件

- A **parallel machine model** is an abstract parallel computer from the programmer's viewpoint, analogous to the von Neumann model for sequential computing

程序员看到的计算机：
编程模型(编程语言、函数包的API)

高层软件，例如用户程序

编程模型：编程语言、函数包API

第三层软件，例如编译和运行库

第二层软件，例如system functions

第一层软件，例如OS内核

可执行程序：SIMD/MIMD/MISD

硬件资源：共享/分布/分布-共享存储体系结构

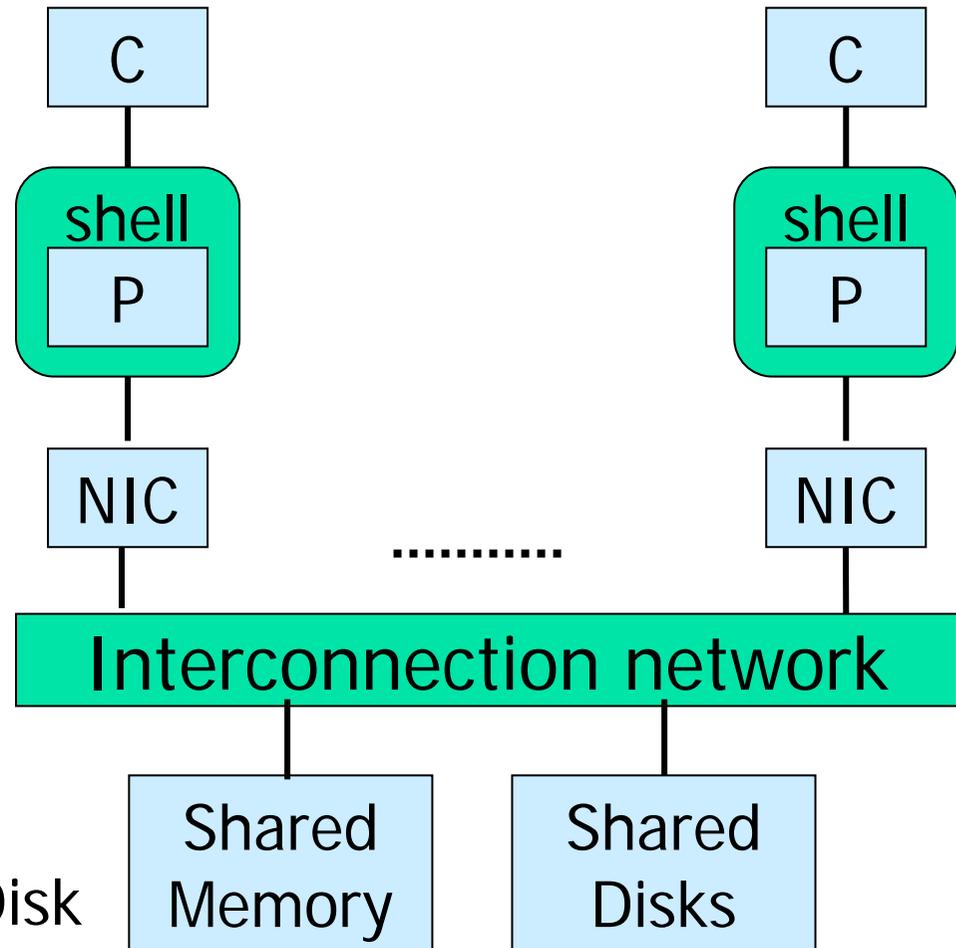
常见的并行编程模型

- 站在程序开发的角度，把并行计算机划分成两类
 - 多处理器multi-processor：程序运行时，只有一个进程，其中可以包括多个并发运行的线程
 - 进程：OS的资源分配单位。一个进程有一个存储空间，进程的所有操作都是在这个存储空间上进行的
 - 线程：处理器资源的调度单位
 - 多计算机multi-computer：程序运行时，有多个进程
 - 每个进程存储程序的一部分数据、执行一部分计算任务
 - 进程之间有协作：同步、交换数据
- Multi-processor
 - 并发threads
 - pthreads：POSIX并发线程模型
 - OpenMP
 - CELL BE线程模型
 - PRAM (Parallel Random Access Machine：理论模型)

- Multi-computer
 - Message passing
 - Data parallel
 - BSP (Bulk Synchronous Parallelism: 理论模型; 同时定义了一个函数包, 可以用于并行程序开发)
- 并行程序的代码结构
 - SPMD: Single Program Multiple Data。编写一个程序, 求解不同的子任务。不同的子任务, 执行的控制流不同
 - 控制转移、分支
 - pthread、OpenMP、Data parallel、Message passing(通常用SPMD并行程序)
 - MPMD: Multiple Program Multiple Data。编写多个程序, 分别求解不同的子任务
 - CELL BE线程模型、Message passing(支持MPMD并行程序)
- 理论模型: 分析算法的复杂性, 指导并行算法设计和程序性能优化

程序员眼中的Multi-processor

- 有一个全局的存储空间
- 每个处理器都可以直接访问任何存储地址
- 用“锁”/“信号”的办法解决对共享变量的竞争



C Cache M Memory

D Disk

Shared
Memory

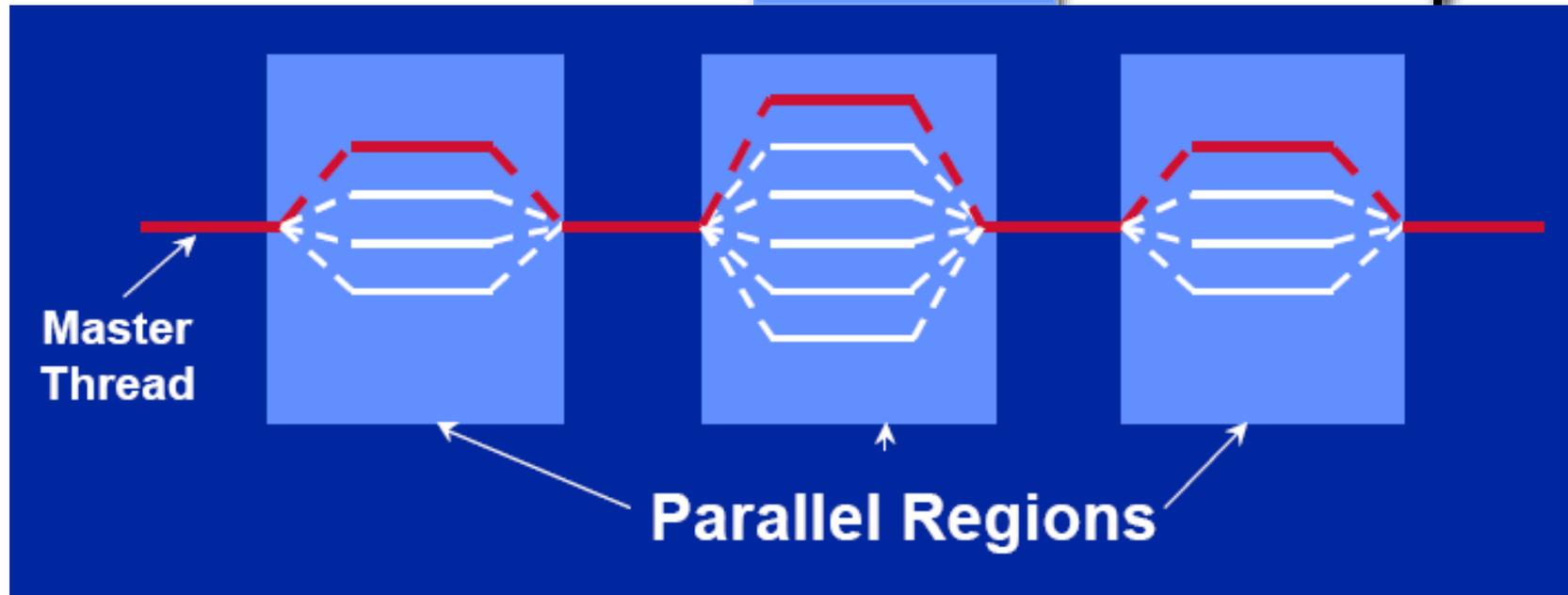
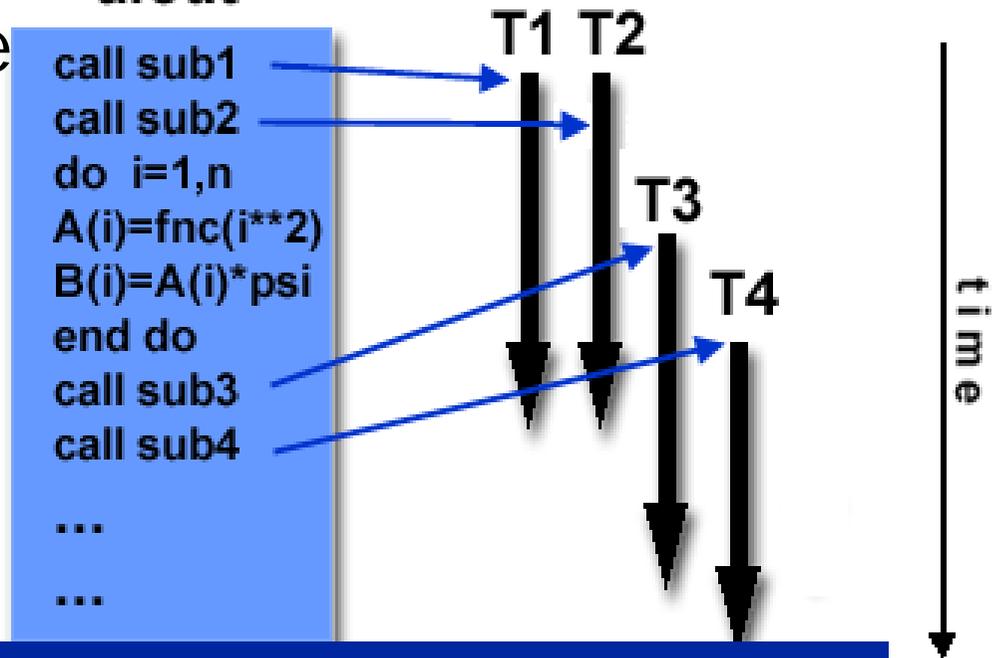
Shared
Disks

P Processor NIC Network interface circuitry

shell A custom-designed circuitry that interfaces a commodity microprocessor to C, M, D, NIC

Threads model a.out

- a single process can have multiple, concurrent execution paths
- Master thread (the main program) performs: serial work; fork/join slave threads



- 每个线程分别由OS调度到不同的CPU上运行
- 从存储空间的角度，看线程之间的关系(与C程序对比)
 - Master thread: main function
 - Slave threads: subroutines called in the main function
- Master thread与slave threads共享存储空间
 - main function and the called subroutines share global variables
- Master thread与slave threads都可以有自己的私有数据
 - Both the main function and called subroutines can declare local variables

从程序运行行为的角度，看线程之间的关系(与C程序类似)

■ main function

- call一个subroutine后，停止自己的工作，直到subroutine 执行完，再继续自己的工作
- subroutine 在运行过程中独占global memory space的访问权
- subroutine在运行结束后，可以通过返回参数将结果传递给main function的local memory space

■ Master thread

- fork 一个slave thread后，继续做自己的工作，将与slave thread并发执行
- slave thread只能将自己的结果写到global memory space
- 每个thread访问global memory space时，需要考虑与其它thread的同步(“锁”、“信号量”)

例子：计算小于N的全部素数

- 判断K是素数的算法：K不能被[2, x]之间的素数整数
 - $x^2 < K < (x+1)^2$
 - K从3开始

- 并行程序

```
primes[5000]; //存储找到的素数
totalPrimes; //找到的素数总数
pthreads=0; //fork的线程总数
K = 3;
main(){
    k=3;
    primes[0] = 2;
    primes[1] = 3;
    totalPrimes = 1;
    创建一组线程;
    searchPrimes();
    等待所创建的线程结束, 此时pthreads=0;
}
```

- 每个线程执行

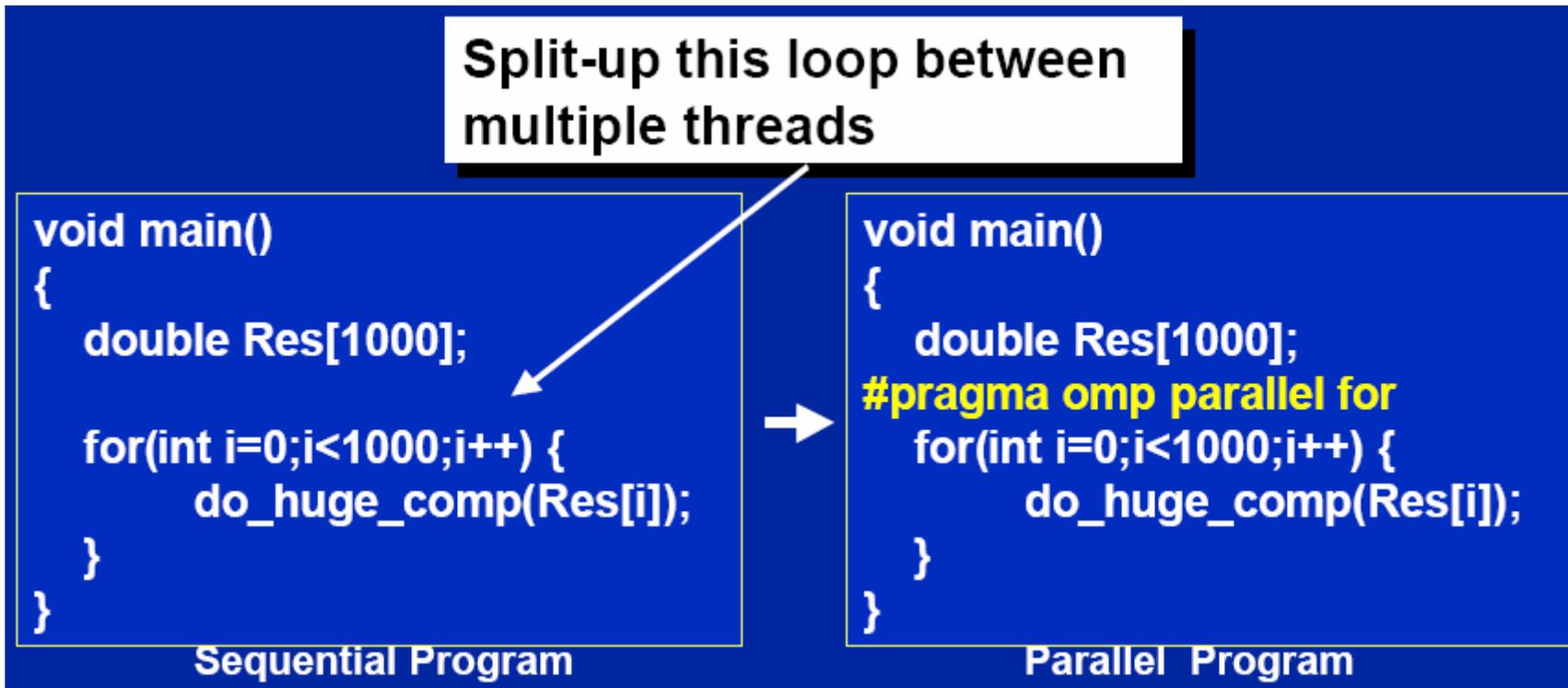
```
pthread++;  
searchPrimes();  
pthread--;
```
- `searchPrimes()`: 从剩下的数据中取一个最小的数K, 判断它是否为素数, 直到小于等于N的全部数都判断完为止。算法的实现并不复杂, 但需要使用
 - “锁”`lock1`: 确保每个k只被取一次
 - “锁”`lock2`: 确保每次只有一个线程在对pthread执行自增、自减运算
 - “锁”`lock3`: 确保每次只有一个线程在对primes执行write操作
 - “信号量”`signal`: 确保素数在primes中从小到大顺序存储
 - 找到一个素数prime后, 查看期待的下一个素数是否是prime, 即: `primes[totalPrimes+1]==prime??`
 - 如果不是, 等到`primes[totalPrimes+1]`被改变的信号出现每个数字k在判断完成后, 无论k是否是素数, 都将期待的下一个素数的值置为K+1

Pthreads

- 定义了一个函数包，用来实现“锁”、“信号量”机制
- 支持用C开发并行程序

OpenMP

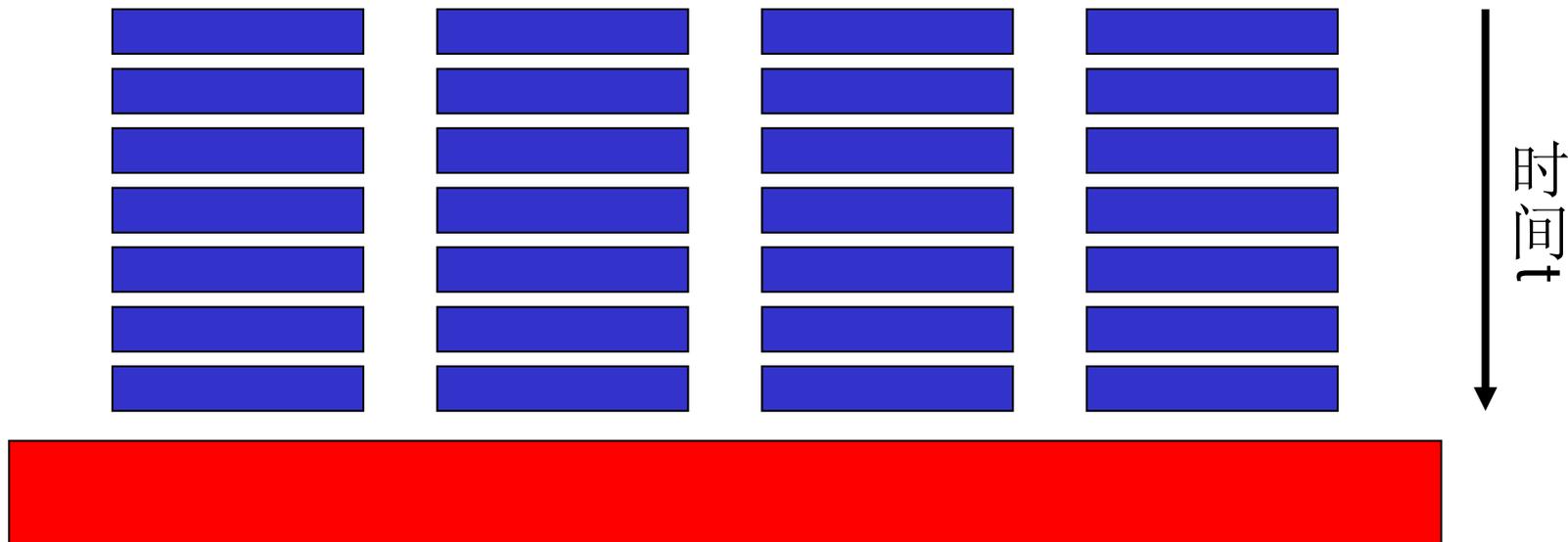
- 基于编译制导语句
- 为C/C++和Fortran分别提供了不同的接口
- 由并行编译器根据指导语句提供的信息，生成可执行的并行程序
- 有兴趣的同学可以继续读课程网站提供的资料



PRAM

- 在串程序设计中，采用 *von Neumann computer* 刻画影响程序运行性能的关键因素。
 - 串程序的复杂性度量：程序中循环的迭代空间与问题规模 N 的关系，例如 $O(N)$ 、 $O(N \log N)$ 、 $O(N^2)$ 、.....。
 - 每个循环体运行一次的时间取决于
 - 问题的特征：循环体需要的运算量
 - CPU 的速度
 - MEMORY 的数据存取速度
- 对于 multi-processor
 - 如何在算法设计和程序开发中解决存储一致性问题？
 - 如何衡量并行程序的性能？

- PRAM试图对multi-processor进行抽象，刻画其中对性能其本质作用的特征
 - N个处理器
 - 一个全局的内存空间，由N个处理器共享
 - 每个处理器都可以通过系统总线直接访问共享内存中的任何元素
 - 对存储空间中的任何元素，每个处理器的访问开销是相同的



PRAM上的并程序

- 一个并程序由N个process(进程?我觉得线程更合适)组成, 第I个进程位于第I个处理器上
 - 每个进程是一组指令的线性序列
 - 在每个时间步(basic time step/cycle)上, 每个进程分别执行一条指令
 - 数据传输指令
 - 数学/逻辑运算指令
 - 控制转移指令
 - I/O指令
 - NULL指令等
- 进程之间通过共享变量进行交互

- 共享内存的访问冲突问题——一致性原则(consistency rule)
 - EREW(exclusive read exclusive write): 每个CYCLE上, 一个内存单元最多只能被一个进程访问
 - CREW(concurrent read exclusive write): 每个CYCLE上, 一个内存单元可以被多个进程读, 但最多只能被一个进程写
 - CRCW(concurrent read concurrent write): 每个CYCLE上, 多个进程可以读、写同一个内存单元的数据, 而当发生写冲突时, 则采用预先确定的访问策略解决冲突问题
 - common: 所有进程写的数据相同
 - priority: 由最优先的那个进程写数据
 - arbitrary: 允许任意处理器自由写

PRAM对Pthreads、OpenMP程序开发的指导

- 进程之间按照CYCLE进行进度同步
 - “锁”、“信号”机制的实现：插入NULL指令
- 影响性能的因素
 - 单个进程中的指令数量：并行程序需要的CYCLE数
 - CYCLE需要的实际时间
 - 程序中NULL指令的数量：并行程序的EFFICIENCY
- 提高并行程序性能的策略
 - 负载均衡：均衡每个进程中的非空指令数
 - 优化子任务的划分：减少子任务之间的数据相关
 - 调度子任务内部的指令：减少为实现同步需要的NULL指令

PRAM并不仅仅用来刻画multi-processor系统

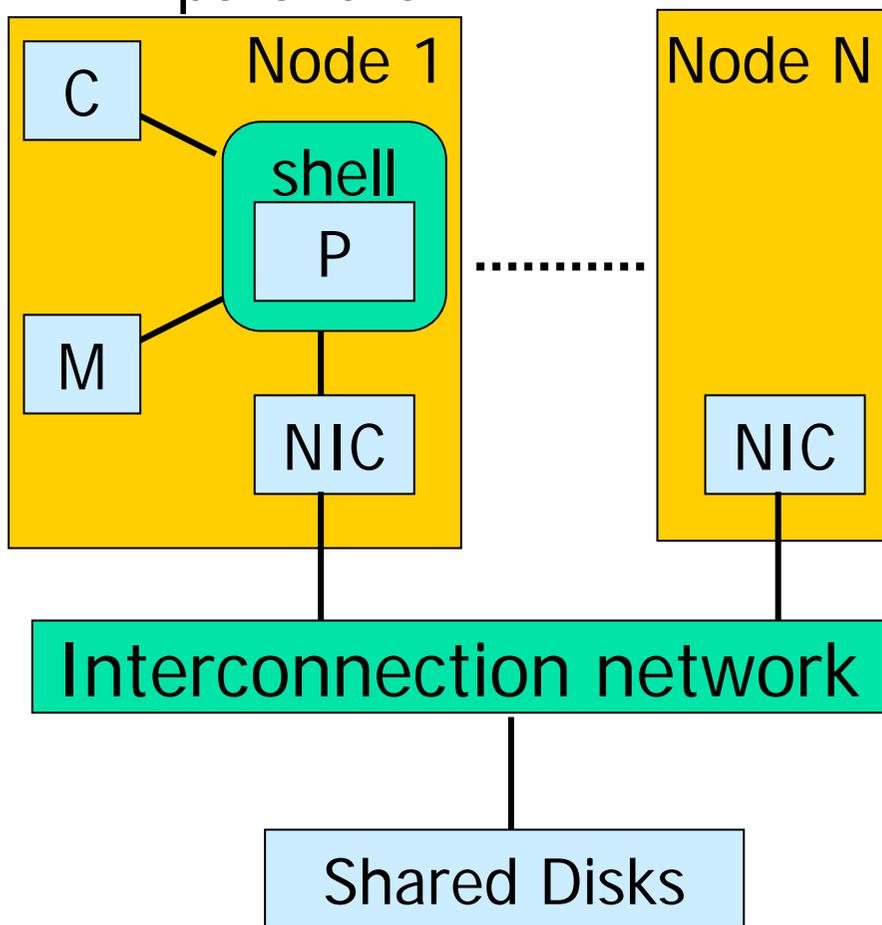
- INTEL的IA64处理器
 - 按照bundle来组织应用程序中的指令，每个bundle最多可以包括3条指令，其中可以包括NULL指令
 - 按照bundle调度执行应用程序， bundle中的指令并行执行
- IA64与PRAM
 - Bundle VS CYCLE
 - 为IA64开发编译运行支持系统是重点：IA64上仍然采用Fortran、C/C++、Java等串行语言
- 站在编译器的角度看， IA64系统可以理解成一个有3颗处理器的PRAM
- 站在普通程序员的角度看， IA64系统仍然是一台Von Neumann计算机

- 当前国际上正在兴起“多核”处理器技术(mult-core):
 - 在一个处理器芯片上聚合多个运算器单元
 - 策略一：编译技术(通常建议使用OpenMP指导语句)
 - 用Fortran、C/C++、Java等语言编写程序
 - 由编译器和运行支持系统负责把并发的指令调度到不同的运算器单元上执行
 - 策略二：线程(POSIX线程、CELL BE线程：**本课程重点之一**)
 - 策略一和策略二互相补充？
- **小结（？）**：C/C++ V.S. X86 and IA64
 - 计算机模型与计算机的体系结构并不完全一致
 - 计算机模型是程序开发环境展现给编程人员的
 - 采用同一计算机模型开发的应用程序，在不同体系结

程序员眼中的Multi-computer

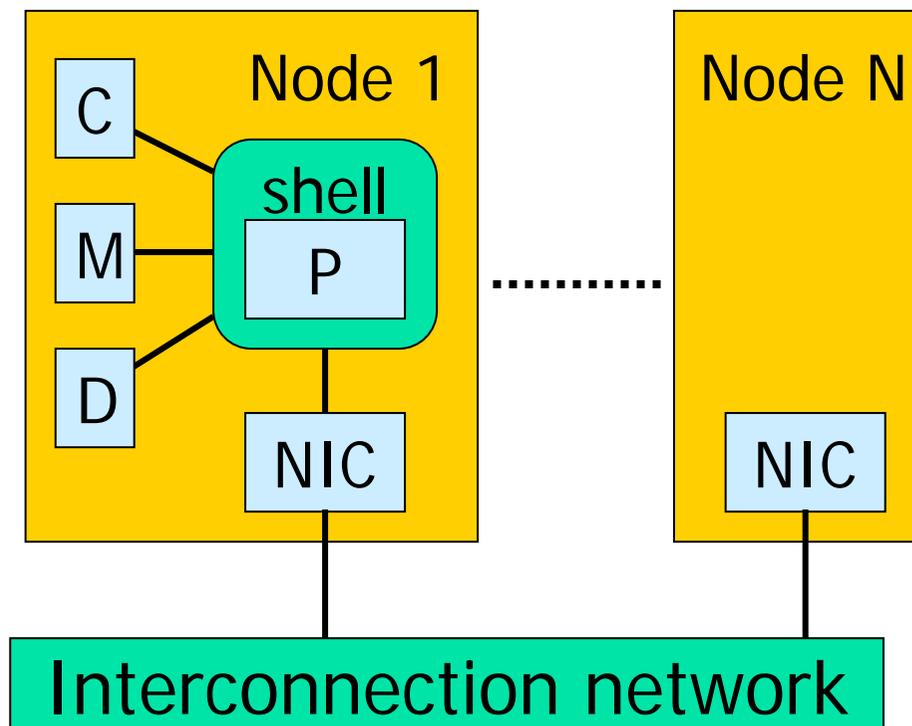
- Shared disks

- HPF(data parallel)
- BSP: Bulk Synchronous parallelism



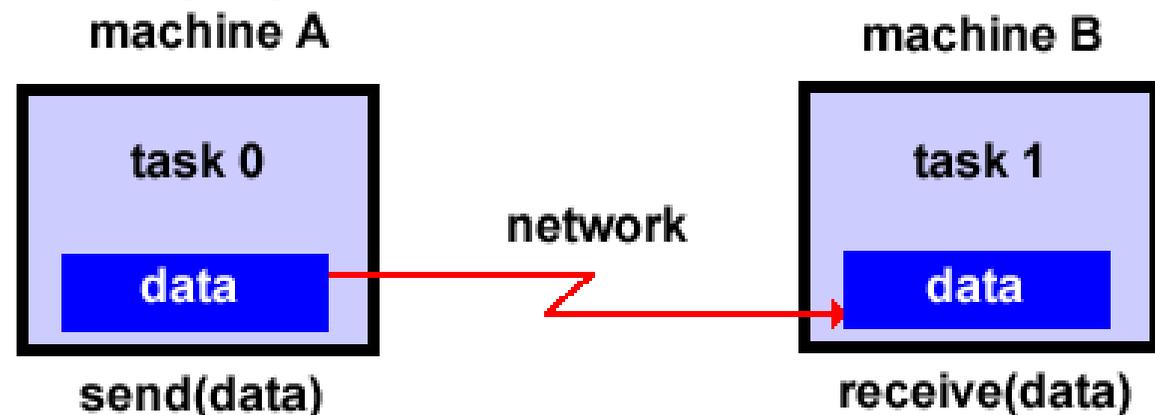
- Shared nothing

- MPI(message passing)



Message passing

- A set of tasks that use their own local memory during computation
 - Each task is implemented as a process
 - Multiple processes may reside on one physical machine
- Tasks exchange data through communications by sending and receiving messages
- Data transfer usually requires cooperative operations to be performed by each process.
- These processes are coded with some serial language and a library of routines for exchanging data between processes

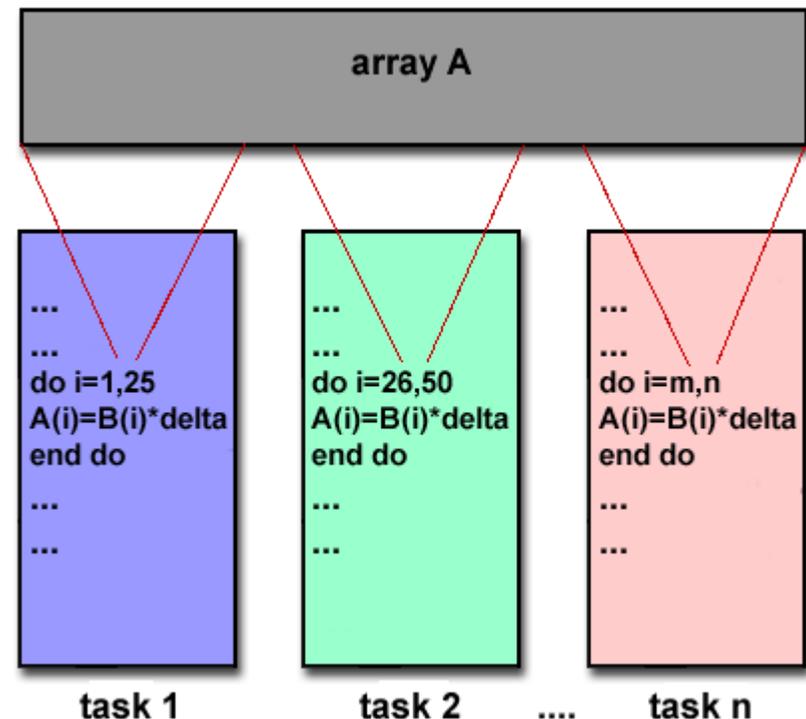


Data parallel

- The parallel computer is NUMA (Non-Uniform-Memory-Access)
- A set of tasks work collectively on the same data structure
 - each task works on a different partition of the same data structure
 - Tasks perform the same operation on their partition of work
- 有兴趣的同学可以看课程网站chp7, DBPP

```

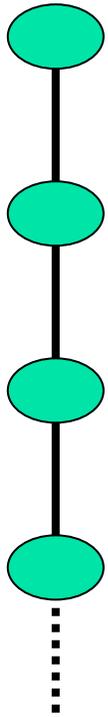
REAL A(100), B(100)
HPF$ PROCESSORS procs(4)
HPF$ DISTRIBUTE BLOCK ONTO procs::A
DO k=1, num_iter
    FORALL (i=1:100)
        A(i)=B(i)*delta
    END FORALL
END DO
    
```



■ 基于编译制导语句

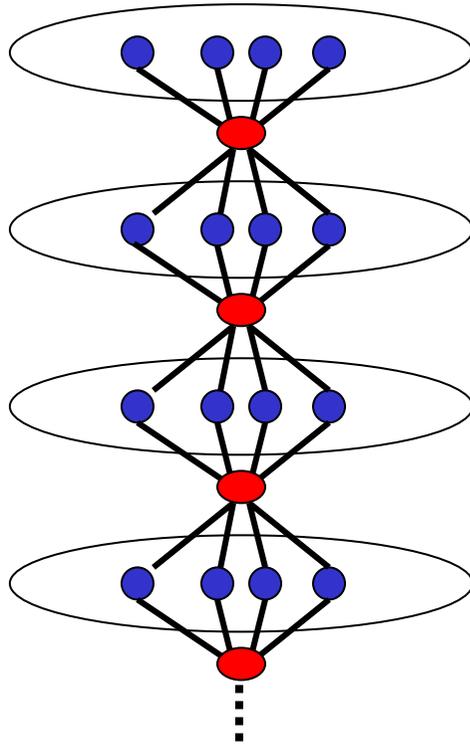
- 在处理器之间如何分布数据
- 哪些计算是可以并行执行的

HPF程序



全局数据
NUMA

HPF程序的执行



全局数据
NUMA

- 处理器实际执行的操作
 - 可以是一个操作
 - 可以是一组操作
 - 可以是NULL
 - 每个非NULL的操作都是对标量的运算
 - 每个非NULL的操作可以直接访问全局存储空间中的任何一个元素
- 处理器之间的同步

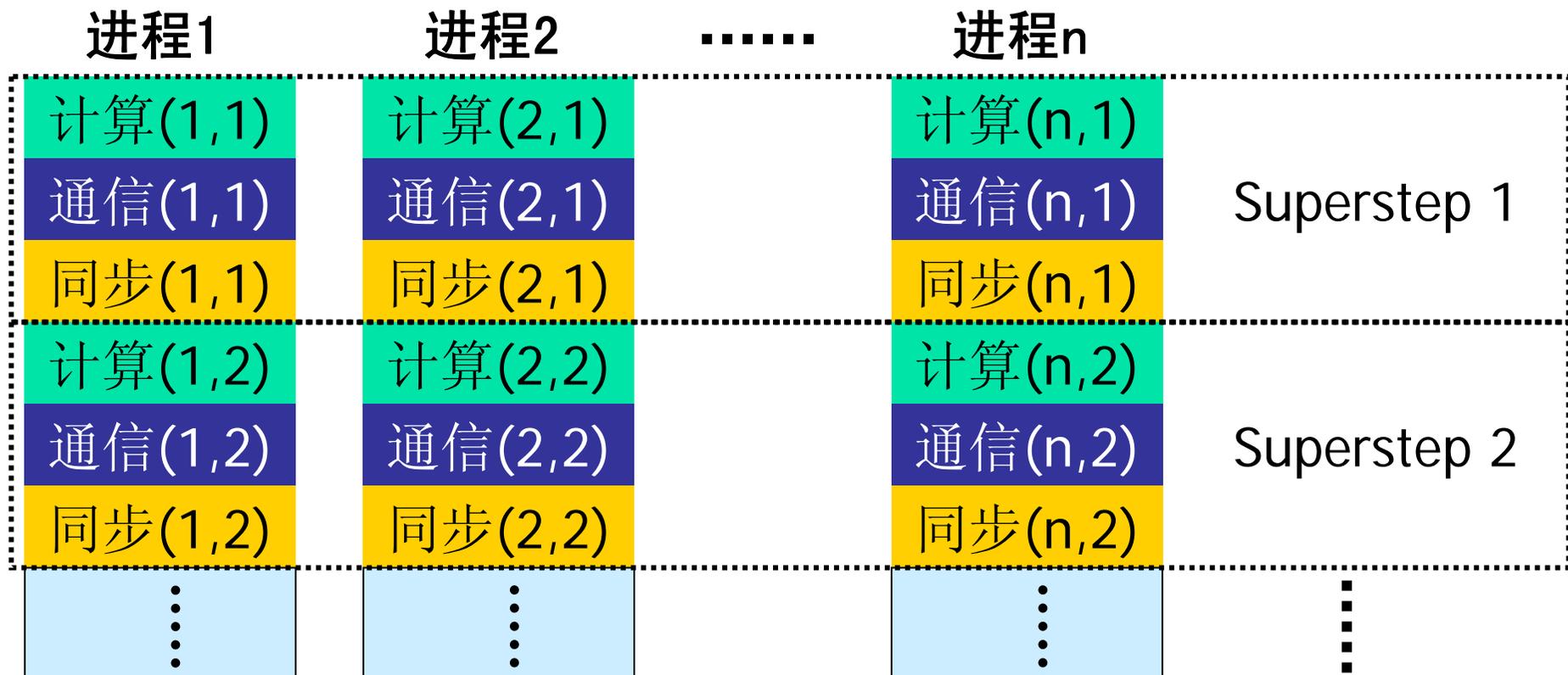
● HPF编程模型接口提供的数据处理操作，可以是对标量(单个的数据元素)的运算，也可以是对向量(数据元素的集合)的运算

BSP

- 在multi-computer上：无论是MPI、还是HPF，最终
 - 在每个处理器上所执行的程序是由计算语句和通信语句组成的线性序列
 - 整个并行程序由不同处理器上的程序通过它们中的消息“收发-接收”对联系成一个整体
- 在设计算法和开发并行程序时，需要考虑
 - 有什么机制来实现消息的传递
 - 有什么机制来实现消息收发过程中相关的进程之间的进度同步
- 如何衡量这些消息收发对对并行程序性能的影响？
 - 传递消息包的大小
 - 消息“收发-接收”对所在两个进程执行进度的匹配

■ BSP试图对multi-computer进行抽象

- 体系结构：N个Von Neumann Computer通过网络连接起来
- 每个处理器分别有一个自己独占的局部存储空间
- 每个处理器只能对自己局部空间中的数据元素进行读写运算
- 处理器访问自己局部存储空间数据元素的速度远高于处理器之间交换数据的速度



BSP上的并行程序

- 一个并行程序由N个process组成，第I个进程位于第I个处理器上
 - 每个进程都被组织成M段代码的一个线性序列
 - 全部进程的第I段代码构成并行程序的第I个superstep
 - 整个并行程序被严格划分成M个superstep
- 进程之间按照superstep进行同步
 - 在superstep内通过消息传递来进行数据交换
- 程序启动时，数据被划分到各个进程，分别存储在自己的本地存储空间中

- 每段代码都包括三部分，依次如下
 - **Computation operations:** 执行对局部存储空间中数据元素的运算，最多需要 w 个cycle
 - **Communication operation:** 执行与其它节点之间的消息传递，最多需要 gh 个cycle
 - 执行当前Computation operations结果的异地数据更新
 - 取得下一个Computation operations需要读的异地数据
 - **Barrier synchronization:** 强制同步，所有进程都到达完成了当前的Computation operations和Communication operation后，才可以执行下面的代码

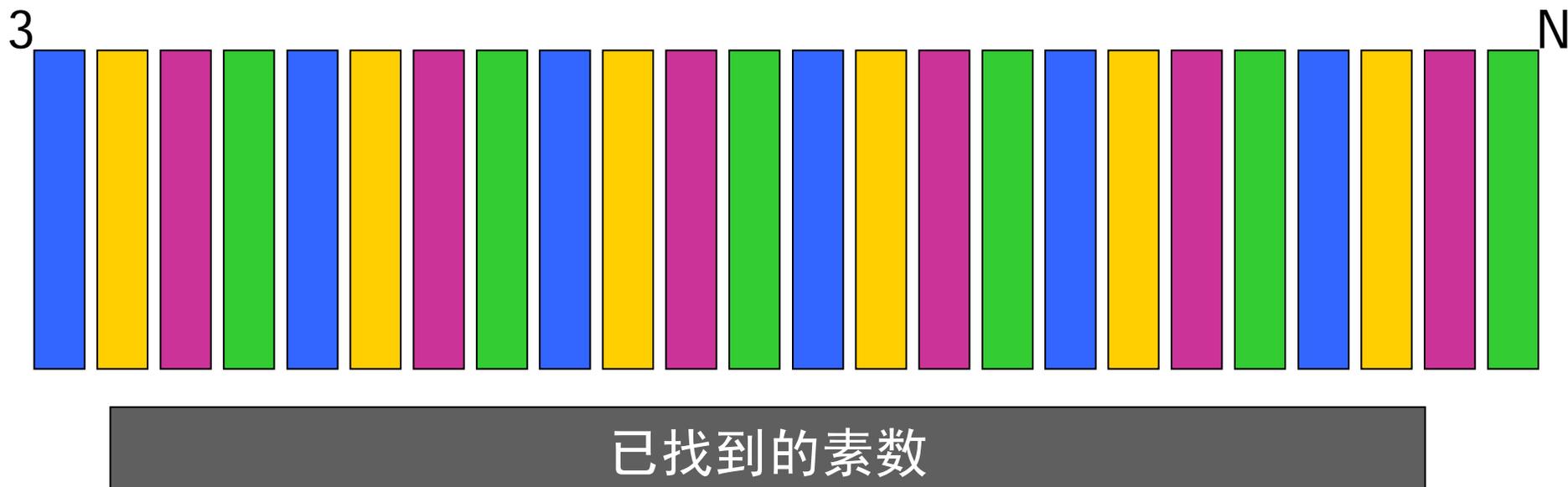
BSP上并行程序的性能分析

- 影响并行程序性能的因素
 - SUPERSTEP的数量
 - 每个SUPERSTEP上，各个进程COMPUTATION OPERATIONS的均衡性：衡量并行程序的EFFICIENCY
 - 每个SUPERSTEP上，交换消息的大小：数据局部性
- 并行程序的执行时间： Σ 第I个superstep的执行时间
 - 第I个SUPERSTEP的执行时间： $\text{MAX} \{ \text{进程J在第I个SUPERSTEP上的COMPUTATION OPERATIONS时间} + \text{COMMUNICATION OPERATION时间} \}$

- 提高并行程序性能的策略
 - 负载均衡：均衡每个每个时间步上各个进程中的
COMPUTATION OPERATIONS
 - 优化子任务的划分：减少子任务之间的数据相关，避免COMMUNICATION OPERATION中大量的数据交换

在multi-computer上求解小于N的全部素数

- 将 $[3, N)$ 划分成m个片段
- 每个处理器一次处理一个片段：第1个片段给第一个处理器、第2个片段给第2个处理、如此循环
 - 每次计算完一个片段后，将找到的结果都通知给其它的处理器



■ 在处理器1上求解

■ 在处理器2上求解

■ 在处理器3上求解

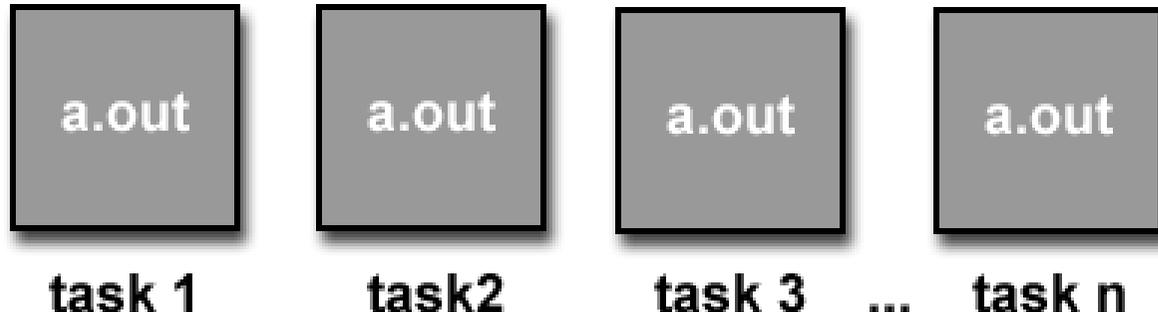
■ 在处理器4上求解

Multi-computer上并行程序代码结构的特征

■ Single Program Multiple Data (SPMD)

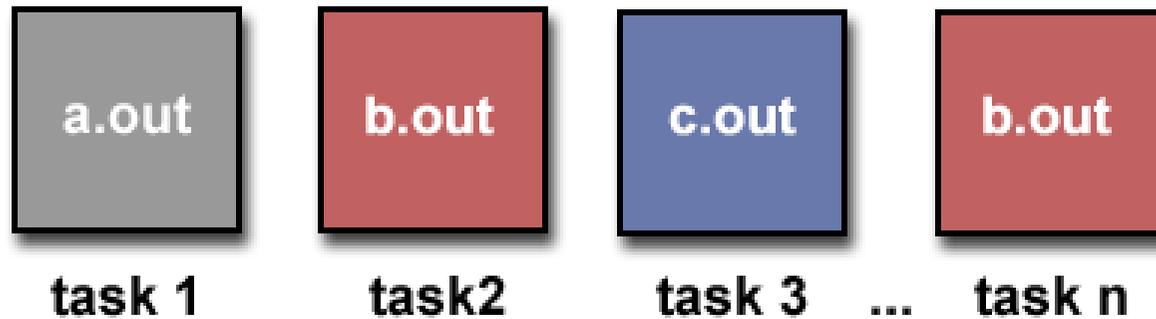
- 在每个处理器上，分别是不同子任务的数据
- 所有处理器上运行同一个可执行程序，它根据处理器号、处理器上的数据确定自己的运行逻辑

```
if (rank == 0) {  
    dest = 1; source = 1;  
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);  
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,  
    &Stat);  
} else if (rank == 1) {  
    dest = 0; source = 0;  
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,  
    &Stat);  
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);  
}  
MPI_Finalize();
```



■ Multiple Program Multiple Data (MPMD)

- 并行程序包括多个可执行程序
- 在每个处理器上，分别是不同子任务的数据；并根据所安排的子任务，安排相应的可执行程序



- 采用Message Passing、BSP模型编写并行程序
 - 既可采用SPMD的表示、也可以用MPMD的表示
 - 通常采用SPMD的表示
- 采用Data Parallel模型编写并行程序
 - 并行程序的表示由编译器确定
 - 既可采用SPMD的表示、也可以用MPMD的表示
 - 通常采用SPMD的表示

开发并行程序的三种途径

- 串行程序自动并行化
 - SUIF
- 编译指导
 - HPF、OpenMP
- 手工开发
 - Pthreads、MPI、CELL BE的C/C++扩展

串行程序自动并行化

- 20世纪70年代后，编译的理论和技術都已经成熟
- 人们习惯于用Fortran、C这样的高级串行语言开发应用程序
- 已经有了很多非常成熟的应用软件
 - 重新开发的工作量非常大
- 实践表明这条途径有局限
 - 编译技术只能从代码本身的数据相关性进行分析
 - 主要用于循环
 - 代码复杂后，寻找可并行的计算很困难。特别有条件分支的情况
 - 编译技术不能对运算量进行估算，即使找到了可并行代码，难以判断是否值得并行。并行本身带来的开销不可忽视
 - 编译器中一旦没有考虑到某种数据相关的出现形式，就可能产生错误的结果

串行程序：求小于N的全部素数

```
void main(){
    int number;    // 小于N的素数个数;
    int primes[n]; // 从primes[0] – primes[number-1] 中存放生成的素数;
    int i, j, k;
    primes[0] = 2;
    for(i=3, j=1; i<n; i++){ // 从整数3开始检查 i 是否为素数
        for(k=0; p[k]*p[k]<i; k++) // 依次检查 i 是否可以被前面的素数整除
            if( i % p[k] == 0) break;
        if(p[k]*p[k]>i){ // 如果 i 不能被前面的素数整除,
            // 则将它作为新素数存入数组
                p[j] = i;
                j++;
        }
    }
}
```

- 外层循环相关，内层循环并行做不值得(多数情况下只要计算前面的几个素数就可以得出不是i不是素数的结论)

并行编译指导语句

- 对串行语言扩充，增加说明性的指导语句、并行算子
- 程序员以指导语句的形式，告诉编译器
 - 哪些计算是可以并行运行的
 - 怎样提高并行计算的效率
- 优势：
 - 在从程序员开来，并行程序与串行程序的差别不算大
 - 程序员为开发并行程序所增加的工作量比较小
- 局限
 - 一些并行的形式表达起来不方便、或者不能表达全部的并行性信息，比如稀疏矩阵的计算
 - 问题复杂后，并行计算的效果不能达到预期的目标

以并行方式求解小于N的全部素数

- 对每个整数K，需要将K与[2, x]之间的素数依次整除。这是整个问题的主要计算开销所在(**hotspots**)
 - $x^2 < K < (x+1)^2$
- 在开发并行程序时，重点是将hotspots划分(**partitioning**)到不同的处理器上并行执行
- 分析所给的两种并行方案
 - **Multi-processor**: 对每个整数K，分别由一个处理器判断是否是素数
 - **Multi-computer**: 将[3, N)划分成m个片段，分别由一个处理器搜索其中的素数
- **Multi-processor**上，“锁”和“信号量”的管理是并行计算带来的额外开销(**parallel overhead**)，是决定程序性能的关键因素。对它们的操作是整个程序的“瓶颈”(**bottleneck**)

- Multi-computer上，每个处理器搜索完一个片段的素数后，要把结果广播给其他的处理器，这也带来了并行计算的额外开销(**parallel overhead**)
 - 负载不均衡(**load-balancing**): 每个数值K需要的除法运算次数相差很大
 - 通信开销(**communication**): 通信启动的次数、每次交换的数据量
- 提高并行的粒度(**granularity**), 可以部分地解决“瓶颈”问题
 - 将 $[3, N)$ 划分成m个片段，分别由一个处理器搜索其中的素数

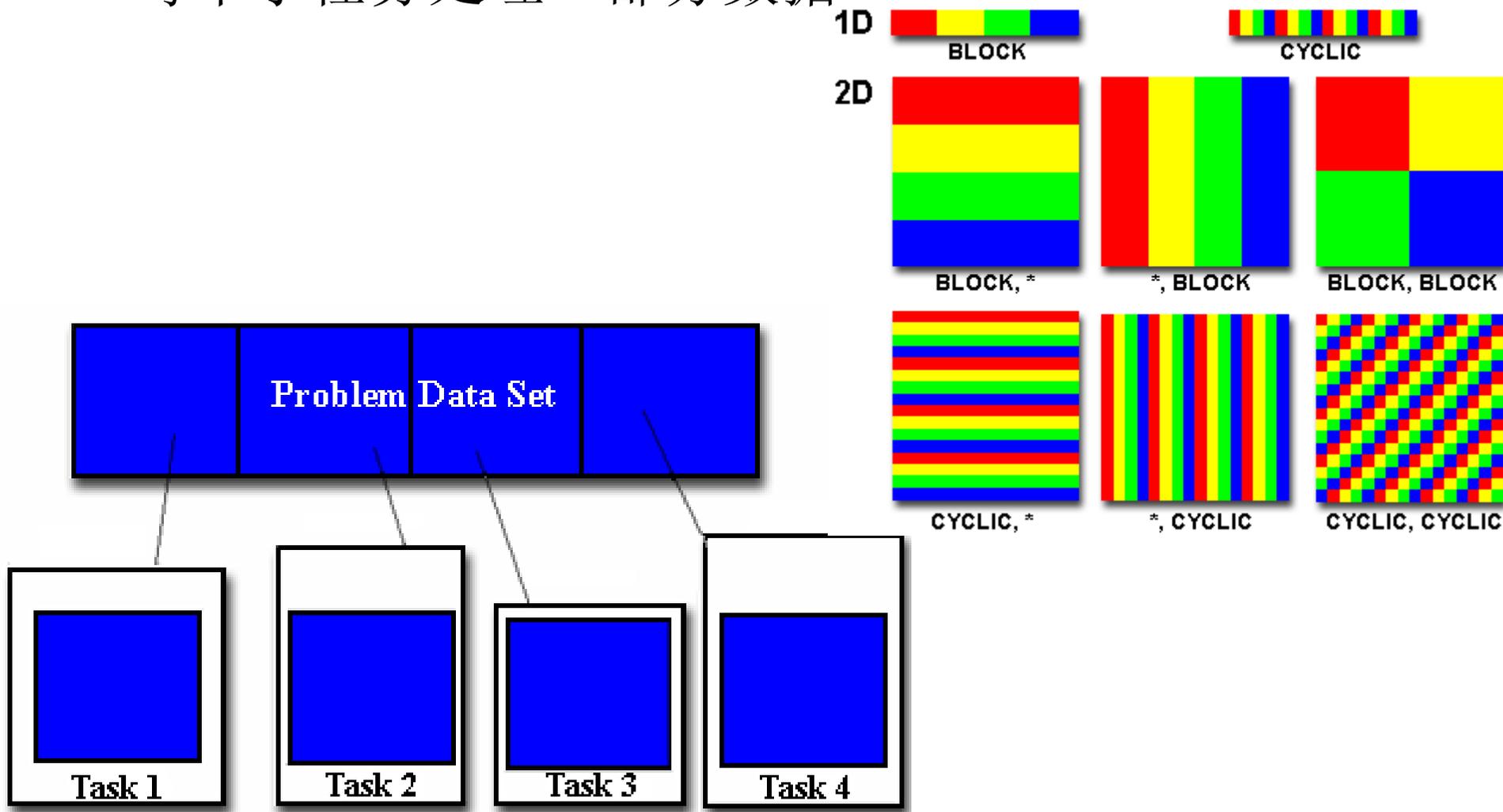
- 但并行的粒度要适中
 - 粒度太小(**fine-grain**), 对解决“瓶颈”的作用不大
 - 粒度太大(**coarse-grain**), 将带来两方面的问题
- 问题1: 已完成了 $[3, x]$ 中素数的寻找, 现在判断 K 是否是素数, 但 $x^2 < K$, 需要对检查 $[x+1, y]$ 之间的奇数能否整除 K ($y^2 < K < (y+1)^2$), 增加了除法运算的次数
- 问题2: 处理器上分得的片段数会相差1, 每个片段越大, 负载越不均衡
- 这两个问题的解决只能靠人, 根据具体的问题特征来解决

并行程序开发涉及的问题

- 问题的并行性分析：并行程序开发不是简单地将串行程序的循环分配到不同的处理器上执行
 - Identify hotspots(通过编译技术自动识别有难度)
 - Identify bottlenecks (通过编译技术自动识别有难度)
 - 有时，需要设法打破**数学算法**引起的相关性(通过编译技术自动解决，一般效果不理想)
 - 在判断一个整数K是否是素数时，需要用到比K小的素数
 - 用数组存储找到的素数时，从小到大依次存储
 - 特别是对一些计算开销大的循环，表面看起来是相关的，无法并行。仔细分析之后，是可以并行计算的
 - 计算小于N的素数
 - $\sum a_i$

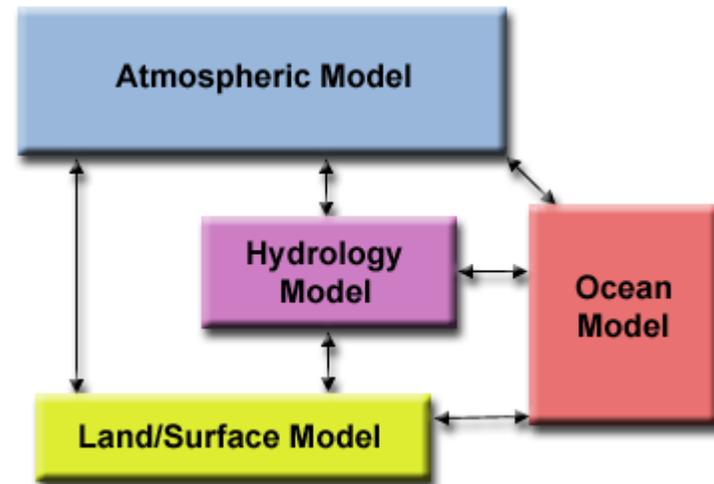
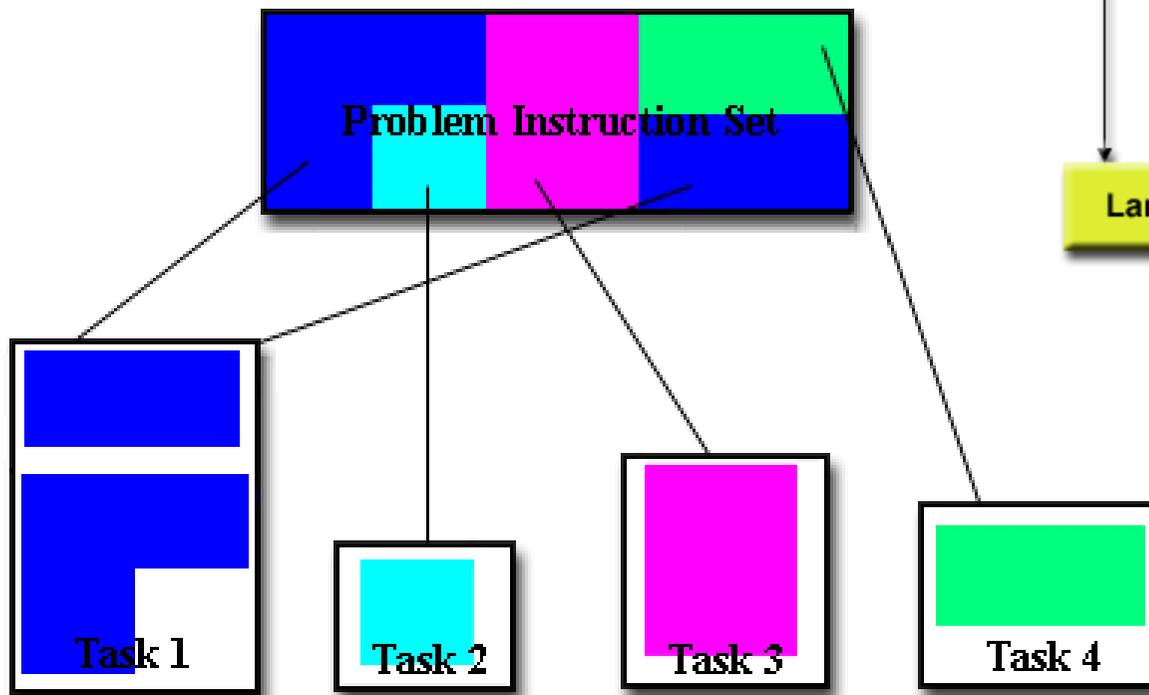
■ 划分子任务的策略

- 数据划分，也称问题域划分(domain decomposition)
- 对问题涉及的数据进行划分
- 每个子任务处理一部分数据



■ 划分子任务的策略

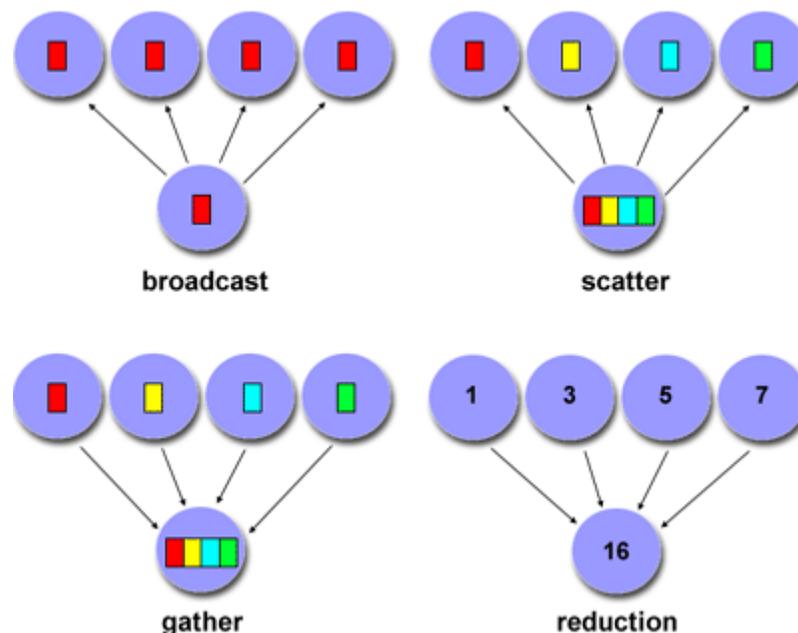
- 任务划分，也成功能分解(functional decomposition)
- 对要完成的功能进行分解
- 每个任务完成一个子功能



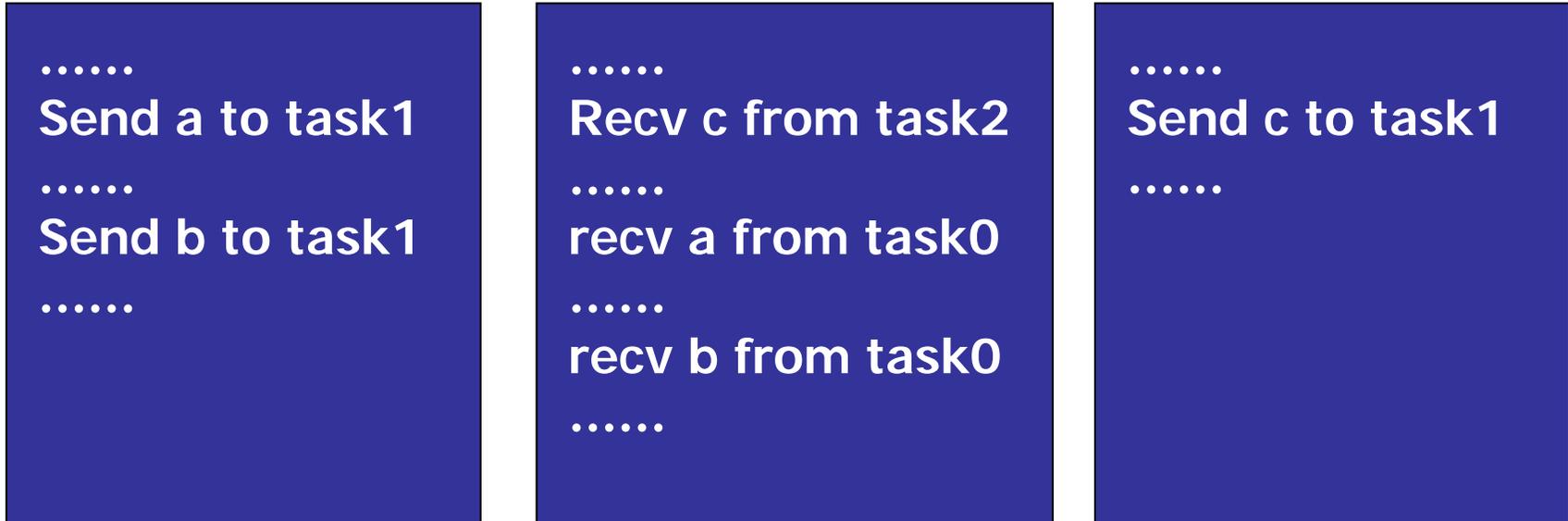
- 通信问题：子任务之间需要互相交换数据
- 通信开销
 - 启动通信语句/原语
 - 交换的数据量
 - 发送方、接收方的同步：一方到达通信点后，要等待另一方也到达对应的通信点
- 通信的方式
 - 同步通信(***blocking***):
 - require some type of "handshaking" between tasks that are sharing data
 - other work must wait until the communications have completed
 - 异步通信(***non-blocking***)
 - allow tasks to transfer data independently from one another
 - other work can be done while the communications are taking place

■ 数据交换的范围

- ***Point-to-point*** : involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
- ***Collective***: involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective. Some common variations (there are more):

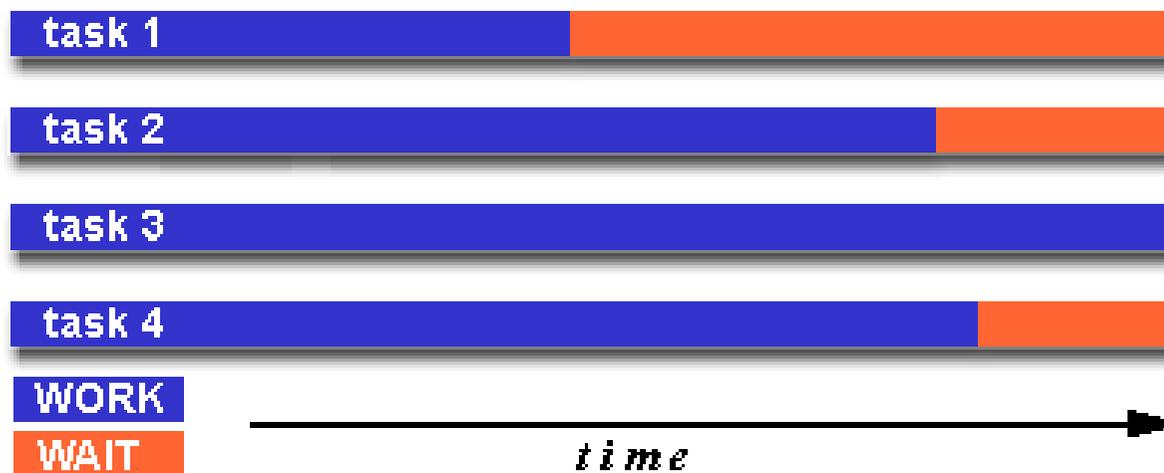


- 同步问题：子任务的运行进度要协调，使得计算结果唯一、正确
- 数据交换的同步

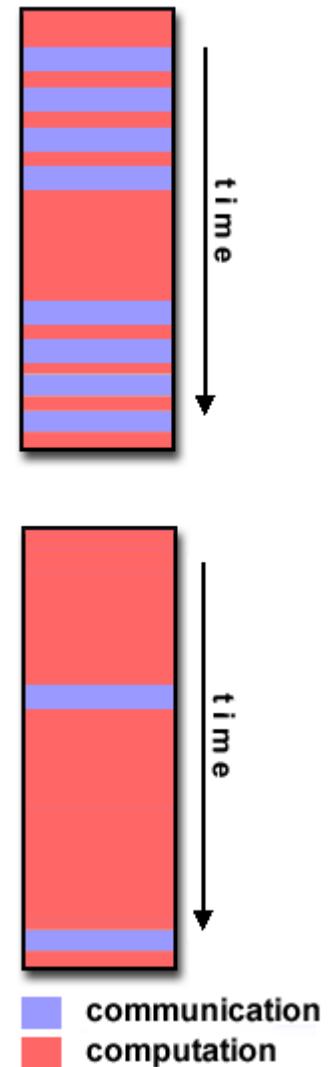


- 数据相关的同步
 - 将小于N的素数按照从小到大的顺序存储在数组primes中

- 负载均衡问题：每个处理器都一直在做工作，没有等待其它处理器的中间结果，也没有因无工作可做而空闲
 - 各并行的子任务需要的开销相差不大：每个处理器负责的子任务数量相同
 - 并行子任务需要的开销相差很大，例如计算素数问题：维护一个子任务“池”，每个处理器到“池”中领取子任务



- 粒度问题：每个处理器完成了多少计算后，才进行一次通信
- 细的粒度
 - 子任务之间偶合程度高
 - 有利于平衡各个处理器上的工作量
 - 通信频繁、并行计算的效率有限
- 粗的粒度
 - 子任务偶合程度低
 - 不容易平衡各个处理器上的工作量
 - 计算划分合理的话，能取得不错的并行效率
- 并行任务的粒度也可能受到问题的算法限制
 - 计算小于N的素数，划分的片段太大，将带来两方面的问题
- 此外，还包括：IO问题、移植性问题、伸缩性问题



练习

- 在PRAM并行计算机上，执行数组A的求和运算，假设A有n 个元素，PRAM上有m颗处理器。在一个CYCLYE上，每个处理器可以完成一次加法运算，包括：从存储空间X和Y取出加法运算的两个操作数，求出两个操作数的和，并写到存储空间Z。假设每个存储单元存储A的一个元素、或者它们的和。计算最少需要多少个CYCLE才能完成数组A的求和运算、共使用了多少个存储单元。简述计算的依据

- 在BSP并行计算机上，执行数组A的求和运算，假设A有n个元素， BSP上有m颗处理器。在每个处理器上
 - 执行一次加法运算需要 T_1 时间，包括：从本地存储空间X和Y取出加法运算的两个操作数，求出两个操作数的和，并写到本地存储空间Z
 - 从其他处理器获得一个数据、存储到本地存储空间需要 T_2 时间
 - 与其它全部处理器同步一次至少需要 T_3 时间，即：在该处理器执行同步操作之前，其他所有处理器都在等待与它同步

计算最少需要多少个CYCLE才能完成数组A的求和运算、共使用了多少个存储单元。简述计算的依据