《高性能计算与云计算》

复习资料

2012 计算机全英创新班

黄炜杰(201230590051)

1. 解释并比较以下基本概念

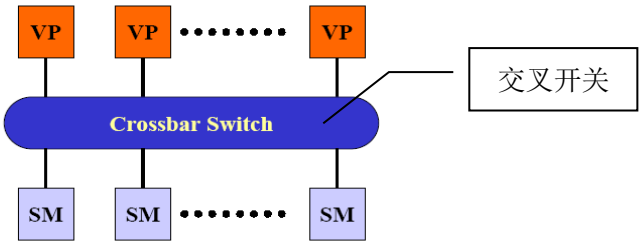
PVP, SMP, MPP, DSM, Cluster, Constellation

UMA, NUMA, CC_NUMA, CORMA, NORMA

HPC, HPCC, Distributed computing, Cloud computing

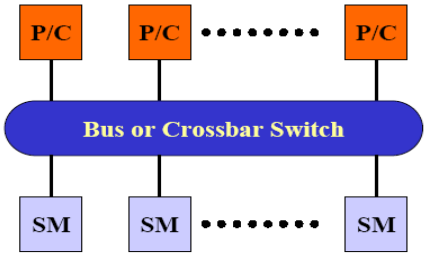
PVP：并行向量处理机。系统中包含为数不多的高性能特制的向量处理器，使用专门设计的高带宽交叉开关网络将向量处理器连向共享存储模块。通常不使用高速缓存，而使用大量的向量寄存器和指令缓冲器。

VP : Vector Processor　　SM : Shared Memory

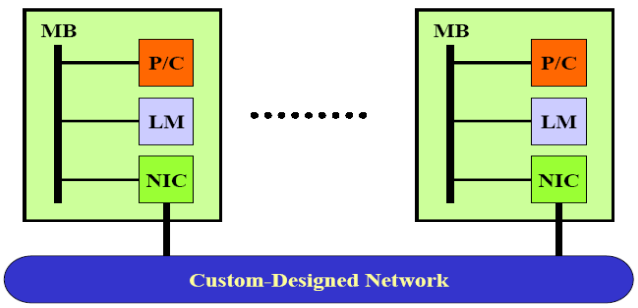


SMP：对称多处理机。节点包含两个或两个以上完全相同的处理器，在处理上没有主/从之分。每个处理器对节点计算资源享有同等访问权。SMP系统使用商品微处理器（具有片上或外置高速缓存），它们经由高速总线或交叉开关连向共享存储器。

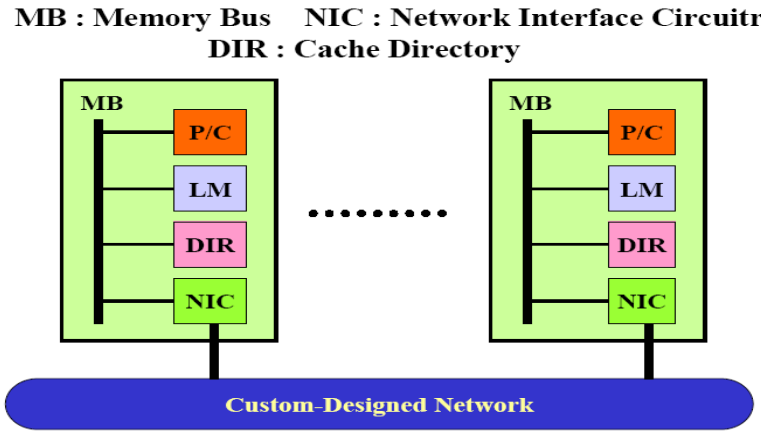
P/C : Microprocessor and Cache



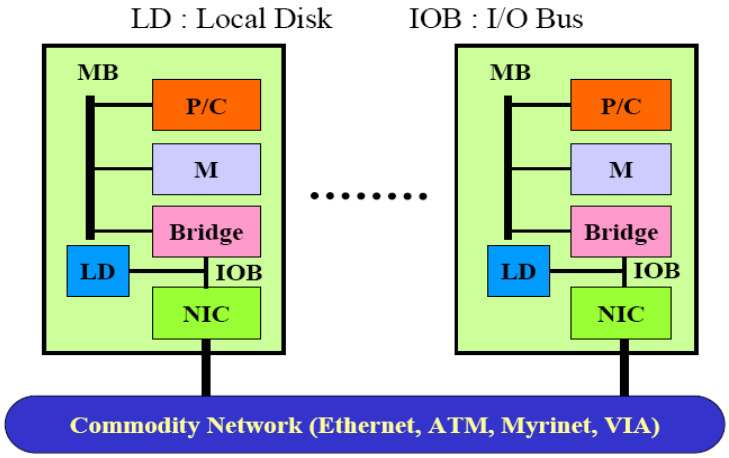
MPP：大规模并行处理机。节点传统上是由单一CPU、少量的内存、部分I/O、节点间的互联以及每个节点的操作系统的一个实例组成。节点间的互联(以及驻留于各节点的操作系统实例)不需要硬件一致性，因为每个节点拥有其自己的操作系统以及自己唯一的物理内存地址空间。因而，一致性是在软件中通过“消息传送”(message passing)实现的。



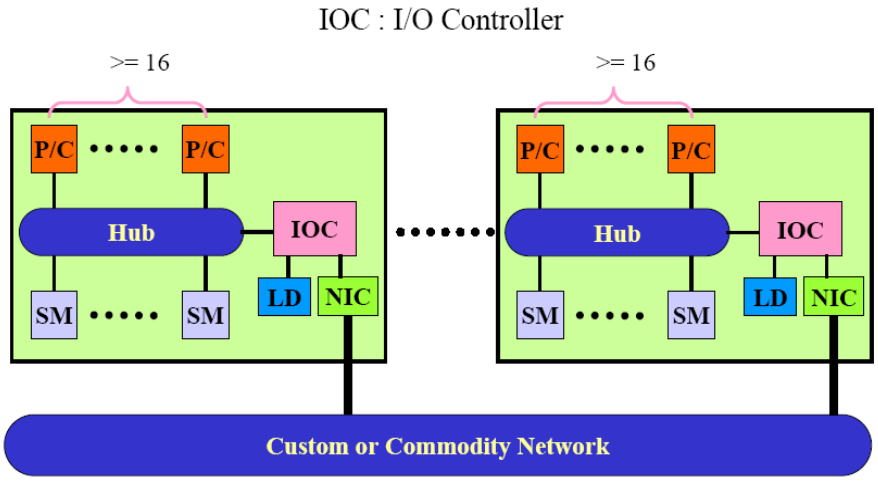
DSM：分布共享存储多处理机。在物理上有分布在各节点的局部存储器，从而形成一个共享的存储器。对用户而言，系统硬件和软件提供了一个单地址的编程空间。



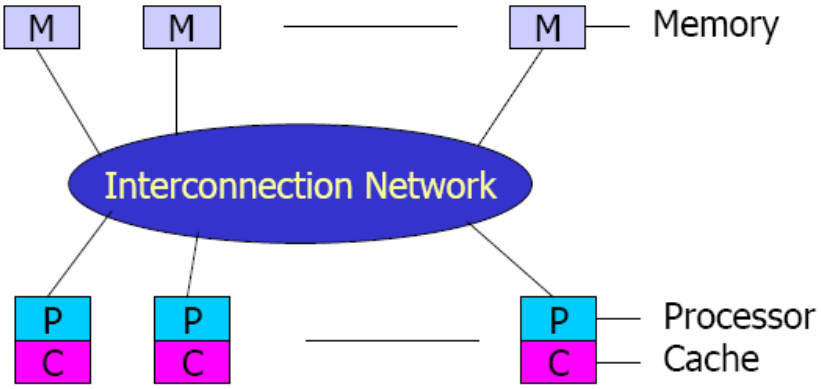
Cluster：集群。系统中的每个节点拥有小于16个处理器。Cluster是一种并行或分布式处理系统，由一系列通过网络互连的互相协同工作的单机组成，形成单一、整合的计算资源。



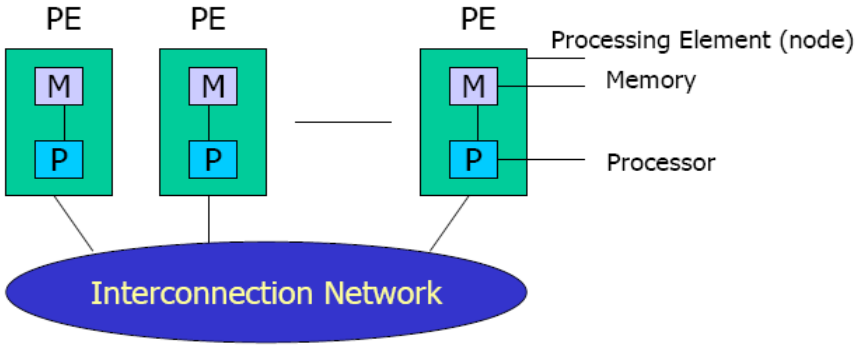
Constellation：系统中的每个节点拥有大于或等于16个处理器。



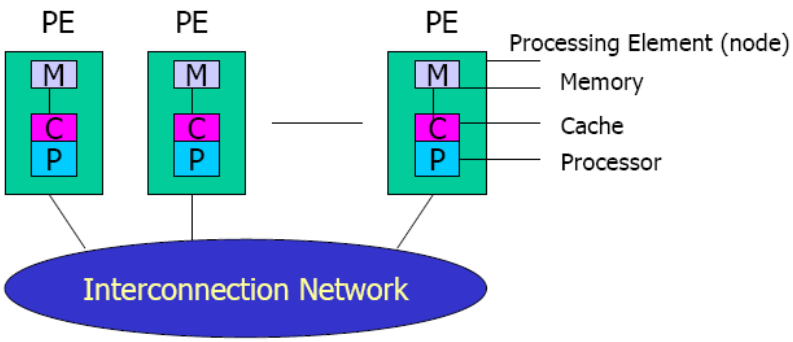
UMA：Uniform Memory Access. 均匀存储访问模型。特点：1.物理存储器被所有处理器均匀共享，所有处理器访问任何存储单元花费相同的时间；2.每台处理器可带私有高速缓存；3.外围设备也可以一定形式共享。



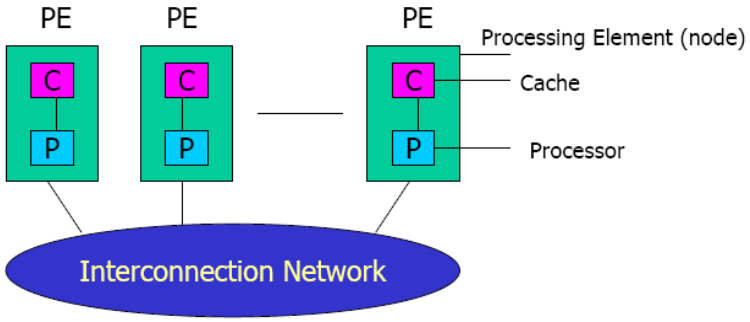
NUMA：Nonuniform Memory Access. 非均匀存储访问模型。特点：1.被共享的存储器在物理上是分布在所有的处理器中的，组成全局地址空间；2.处理器访问存储器的时间是不同的，访问本地存储器或群内共享存储器较快，访问外地存储器或全局存储器较慢；3.每台处理器可带私有高速缓存，外设可以某种形式共享。



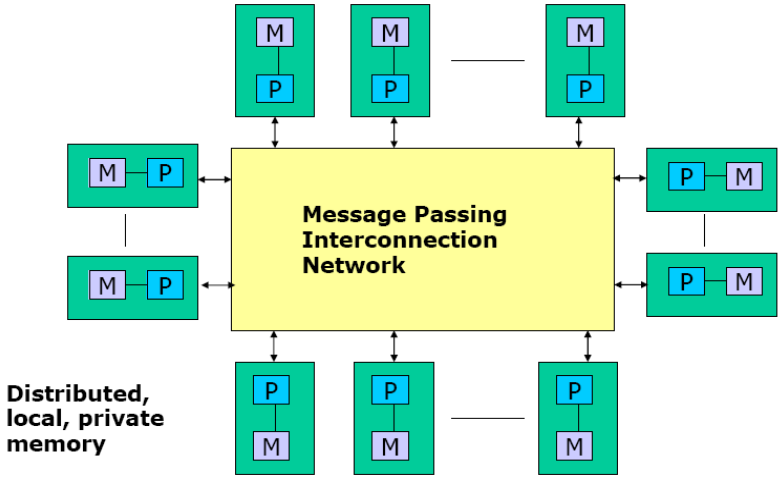
CC_NUMA：Coherent-Cache Nonuniform Memory Access. 高速缓存一致性非均匀存储访问模型。 实际上是将一些SMP机器作为一个单节点而彼此连接起来所形成的一个较大的系统。特点：1. 使用基于目录的高速缓存一致性协议；2. 保留SMP结构易于编程的优点，改善了常规SMP的可扩放性问题；3.实际上是一个分布共享存储的DSM多处理机系统；4.最显著的优点是程序员无需明确地在节点上分配数据，在运行时高速缓存一致性硬件会自动地将数据迁移到要用到的它的地方。



COMA： Cache-Only Memory Access. 全高速缓存存储访问。是NUMA的一种特例。特点：1.各处理器节点中没有存储层次结构，全部高速缓存组成了全局地址空间；2.利用分布的高速缓存目录进行远程高速缓存的访问；3.高速缓存容量一般大于2级高速缓存容量；4.使用COMA时，数据开始可以任意分配，因为在运行时它最终会被迁移到要用到的它的地方。



NORMA： No-Remote Memory Access. 非远程存储访问模型。分布存储的多计算机系统，所有的存储器都是私有的，仅能由其处理器访问。绝大多数NORMA不支持远程存储器访问。



HPC: High Performance Computing 高性能计算，即并行计算。在并行计算机或分布式计算机等高性能计算系统上所做的超级计算。

HPCC： High Performance Computing and Communication 高性能计算与通信。指分布式高性能计算、高速网络和 Internet 的使用。

Distributed computing： 分布式计算。在局域网环境下进行的计算。比起性能来说，它更注重附加功能。一个计算任务由多台计算机共同完成，由传统的人和软件之间的交互变成软件和软件之间的数据交互。

Cloud computing： 云计算是一种新兴的共享基础架构的方法，通过互联网将资源以“按需服务”的形式提供给用户，利用互联网连接的数据中心和服务器进行高效计算和信息存取的系统,使计算能力可以向电能一样提供给客户（高度可扩展）

2. 试比较 SMP、MPP 和 Cluster 并行机结构的不同点，并以典型系统举例说明。

属性	SMP	MPP	COW
结构类型	MIMD	MIMD	MIMD
处理器类型	商用	商用	商用
互连网络	总线、交叉开关	定制网络	商用网络（以太 ATM）
通信机制	共享变量	消息传递	消息传递
地址空间	单地址空间	多地址空间	多地址空间
系统存储器	集中共享	分布非共享	分布非共享
访存模型	UMA	NORMA	NORMA
代表机器	IBM R50，SGI Power Challenge，曙光 1 号	Intel Paragon，IBMSP2，曙光 1000/2000	Berkeley NOW，Alpha Farm

3. 列出常用静态和动态网络的主要参数（节点度、直径、对剖带宽和链路数）以及复杂度、网络性能、扩展性和容错性等。常用的标准互联网络有哪些？

网络名称	网络规模	节点度	网络直径	对剖宽度	对称	链路数
线性阵列	N 个节点	2	$N-1$	1	非	$N-1$
环形	N 个节点	2	$\lfloor N/2 \rfloor$（双向）	2	是	N
2-D 网孔	$(\sqrt{N}\times\sqrt{N})$ 个节点	4	$2(\sqrt{N}-1)$	$\sqrt{N}$	非	$2(N-\sqrt{N})$
Illiac 网孔	$(\sqrt{N}\times\sqrt{N})$ 个节点	4	$\sqrt{N}-1$	$2\sqrt{N}$	非	$2N$
2-D 环绕	$(\sqrt{N}\times\sqrt{N})$ 个节点	4	$2\lfloor\sqrt{N}/2\rfloor$	$2\sqrt{N}$	是	$2N$
二叉树	N 个节点	3	$2(\lceil\log N\rceil-1)$	1	非	$N-1$
星形	N 个节点	$N-1$	2	$\lfloor N/2 \rfloor$	非	$N-1$
超立方	$N=2^n$ 节点	n	n	$N/2$	是	$nN/2$
立方环	$N=k\cdot 2^k$ 节点	3	$2k-1+\lfloor k/2 \rfloor$	$N/(2k)$	是	$3N/2$

表 2.1 静态互连网络特性一览表

网络特性	总线系统	多级互连网络	交叉开关
硬件复杂度	$O(n+w)$	$O((n\log_k n)w)$	$O(n^2w)$
每个处理器带宽	$O(wf/n)\sim O(wf)$	$O(wf)$	$O(wf)$
报道的聚集带宽	SunFire 服务器中的 Gigaplane 总线：2.67GB/s	IBM SP2 中的 512 节点的 HPS：10.24GB/s	Digital 的千兆开关：3.4GB/s

表 2.2 动态互连网络的复杂度和带宽性能一览表

网络特性	最小时延	每个处理器带宽	开关复杂度	连线复杂度	连接特性
总线	恒定（轻负载）	O(w/n)~O(w)	O(n)	O(w)	一次只能一对一
多级互联	O((nlog$_k$n))	O(w) ~O(nw)	O((nlog$_k$n))	O(nwlog$_k$n)	是阻塞型网络
交叉开关	恒定	O(w) ~O(nw)	O(n^2)	O(n^{2}w)	全置换

常用的**标准互联网**络有：

- HiPPI
- Scalable Coherent Interface (SCI)
- Myrinet
- **Ethernet**
- **Infiniband**

4. 比较并行计算模型 PRAM、BSP 和 logP。评述它们的差别、相对优点以及在模型化真实并行计算机和应用时的局限性。

模型 / 特性	PRAM	BSP	logP
体系结构	SIMD-SM	MIMD-DM	MIMD-DM
计算模型	Synchronous	Asynchronous	Asynchronous
同步机制	Automatic	Barrier	Implicated
参数	l	p,g,l	l,o,g,p
粒度	Fine/Moderate	Moderate /Coarse	Moderate /Coarse
通信	Shared Variable	Message Passing	Message Passing
地址空间	Global	Single/Multiple	Single/Multiple

- **BSP提供了更方便的程序设计环境，LogP更好地利用了机器资源**
- **BSP似乎更简单、方便和符合结构化编程**

PRAM 优缺点

- 优点：
  - 简单：PRAM模型特别适合于并行算法的表达、分析和比较，使用简单
  - 易于扩展：根据需要，可以在PRAM模型中加入一些诸如同步和通信等需要考虑的内容
- 缺点：
  - 模型中使用了一个全局共享存储器，不适合于分布存储结构的并行机
  - PRAM模型是同步的，不能反映现实中很多系统的异步性
  - PRAM模型假设了每个处理器可在单位时间访问共享存储器的任一单元，因此要求处理机间通信无延迟、无限带宽和无开销，这是不现实的

BSP 优缺点

- 优点：
  - BSP模型试图为软件和硬件之间架起一座类似于冯诺伊曼机的桥梁，因此，BSP模型也常叫做桥模型
  - 将处理器和路由器分开，强调了计算任务和通信任务的分开，而路由器仅仅完成点到点的消息传递，不提供组合、复制和广播等功能，这样做既掩盖具体的互连网络拓扑，又简化了通信协议
- 缺点：
  - 需要显式同步，编程效率不高
  - BSP模型中的全局障碍同步假定是用特殊的硬件支持的，这在很多并行机中可能没有相应的硬件

logP 优缺点

- 异步（ Asynchronous ）模型，没有同步障
- 捕捉了并行计算机的通讯瓶颈
- 通信由一组参数描述，但并不涉及具体的网络结构，隐藏了并行机的网络拓扑、路由、协议
- 可以应用到共享存储、消息传递、数据并行的编程模型中

LogP模型的缺点

- 难以进行算法描述、设计和分析
- 对网络中的通信模式描述的不够深入。如重发消息可能占满带宽、中间路由器缓存饱和等未加描述
- LogP模型主要适用于消息传递算法设计，对于共享存储模式，则简单地认为远地读操作相当于两次消息传递，未考虑流水线预取技术、Cache引起的数据不一致性以及Cache命中率对计算的影响

5. 比较在 PRAM 模型和 BSP 模型上 ,计算两个 N 阶向量内积的算法及其复杂度。

PRAM 模型

- Step 1
  - 每个处理器：$2N/p$ 个加法和乘法
- Step 2
  - 通过树归约方法计算p个局部和：$\log p$
- 总的执行时间： $2N/p + \log p$

BSP 模型

用8个处理器进行点积计算

- Superstep 1
  - 计算：本地和 $w = 2N/p$
  - 通信： h=1 (处理器 0,2,4,6 -> 1,3,5,7)
- Superstep 2
  - 计算： 一个加法 $w = 1$
  - 通信： h=1 (处理器1,5 -> 3,7)
- Superstep 3
  - 计算：一个加法 $w = 1$
  - 通信： h=1 (处理器 3 -> 7)
- Superstep 4
  - 计算：一个加法 $w = 1$
- 总的执行时间 = $2N/8 + 3g+3l+3$

总的执行时间= $2N/p + \log p(g+l+1)$

6. 什么是加速比（speed up）、并行效率（efficiency）和可扩展性（scalability）？如何描述在不同约束下的加速比？

加速比（Speedup）:对于一个给定的应用，并行算法（或并行程序）相对于串行算法（或串行程序）的性能提高程度.

并行效率（Parallel Efficiency ）：处理器的利用率

Efficiency = (Sequential execution time) / (Number of processors * Parallel execution time)

可扩展性（ Scalability ）：当系统和问题规模增大时，可维持相同性能的能力 ，即指应用、算法和结构能否充分利用不断增长的处理器的能力

固定负载(Amdahl 定律)：

计算负载是固定不变,增加处理器数来提高计算速度。加速比为：

(Ws 串行分量，Wp 可并行分量)

$$S=\frac{W_s+W_p}{W_s+W_p/p};\text{不带通信开销的计算公式}$$

$$s=\frac{W_x+W_p}{W_s+\frac{W_p}{p}+W_o}\text{带通信开销的计算公式}$$

固定时间(Gustafson 定律)：

时间固定不变，为了提高精度加大计算量，相应的增多处理器数目。加速比为：

$$S'=\frac{Ws+pW_p}{Ws+p*W_p/p}=\frac{Ws+pWp}{Ws+Wp}\text{(不考虑额外开销)}$$

$$S'=\frac{Ws+pWp}{Ws+Wp+Wo}\text{（考虑额外开销）}$$

存储受限（Sun－Ni 定律）：

只要存储空间许可，尽量增大问题的规模以产生更好的或更精确的解，加速比为：

（f:串行分量比例(Ws/W)　G(p):存储容量增加到 P 倍）

$$S''=\frac{fW+(1-f)G(p)W}{fW+(1-f)G(p)W/p}=\frac{f+(1-f)G(p)}{f+(1-f)G(p)/p}\text{(不考虑额外开销)}$$

$$S''=\frac{fW+(1-f)G(p)W}{fW+(1-f)G(p)W/p+Wo}=\frac{f+(1-f)G(p)}{f+(1-f)G(p)/p+Wo/W}\text{(考虑额外开销)}$$

7. 如何进行并行计算机性能评测？什么是基准测试程序？

并行计算机性能评测：通过 CPU 和存储器的某些基本性能指标、并行和通信开销分析、并行机的可用性与好用性以及机器成本、价格与性价比进行机器级性能测评；通过加速比、效率、扩展性进行算法级性能测评；通过 Benchmark 进行程序级性能测评。

基准测试程序:用于测试和预测计算机系统的性能，揭示了不同结构机器的长处和短处，为用户决定购买和使用哪种机器最适合他们的应用要求提供决策。

8. 什么是可扩放性测量标准？等效率函数的涵义是什么？

可扩放性测量标准：增加系统规模（处理器数）会增大额外开销和降低处理器利用率，所以对于一个特定的并行系统（算法或程序），它们能否有效利用不断增加的处理器的能力应是受限的，而度量这种能力就是可扩放性这一指标。

等效率函数的涵义：如果问题规模 W 保持不变，处理器数 p 增加，开销 To 增大，效率 E 下降。为了维持一定的效率（介于 0 与 1 之间），当处理数 p 增大时，需要相应地增大问题规模 W 的值。由此定义函数 fE(p)为问题规模 W 随处理器数 p 变化的函数，为等效率函数。

9. 什么是分治策略的基本思想？举例说明如何应用平衡树方法、倍增技术和流水线技术。

分治策略的基本思想：将一个大而复杂的问题分解成若干特性相同的子问题分而治之。

平衡树方法：可应用于求 n 个数的最大值：叶节点存放待处理的数据，内节点执行相应子问题计算，根节点给出问题的解。

倍增技术：可以应用于求森林根，对于 n 个节点的树执行 log n 次指针跳跃即可找到树的根。

流水线技术：可应用于执行一维脉动阵列上的 DFT 计算。

10. 什么是均匀划分、方根划分、对数划分和功能划分？如何用划分方法解决 PSRS 排序、归并排序和（m,n）选择问题？

均匀划分：将 n 个元素分割成 p 段，每段含有 n/p 个元素且分配给一台处理器。

方根划分：取每第 i√n(i=1,2,…)个元素作为划分元素，而将序列划分成若干段，然后分段处理之。

对数划分：取每第 i log n(i=1,2,….)个元素作为划分元素，而将序列划分成若干段，然后分段处理之。

功能划分：将长为 n 的序列划分成等长的一些组，每组中的元素应大于或等于 m（最后一组除外），然后各组可并行处理。

采用均匀划分方法，解决 PSRS 排序：均匀划分待排序序列成 n 份，对每份作局部排序，再从每份中抽取 n 个样本，对 n^2 个正则样本进行排序，选择主元然后对每部分进行主元划分，把每部分按段号进行全局交换，最后进行归并排序。

采用方根划分方法，解决归并排序：先对序列进行方根划分，然后进行段间、段内比较，最后进行段组归并。

采用功能划分方法解决(m,n)选择问题：先对序列进行功能划分，然后对每个子序列进行局部排序，将排序的各组进行两两比较，形成 MIN 序列，最后重复局部排序和两两比较直至出现 m 个最小者。

11. 并行算法设计的一般过程 PCAM 是指什么？各个步骤中的主要判据是什么？

PCAM 是 Partitioning（划分）、Communication（通信）、Agglomeration（组合）和 Mappin（映射）首字母的拼写，它们代表了使用此法设计并行算法的四个阶段。

划分判据：

划分是否具有灵活性？
划分是否避免了冗余计算和存储？
划分任务尺寸是否大致相当？
任务数与问题尺寸是否成比例？
任务数与问题尺寸是否成比例？
功能分解是一种更深层次的分解，是否合理？

通信判据：

所有任务是否执行大致相当的通信?
是否尽可能的局部通信？
通信操作是否能并行执行?
同步任务的计算能否并行执行？

组合判据：

增加粒度是否减少了通信成本？
重复计算是否已权衡了其得益？
是否保持了灵活性和可扩展性？
组合的任务数是否与问题尺寸成比例?
是否保持了类似的计算和通信？
有没有减少并行执行的机会？

映射判据：

采用集中式负载平衡方案，是否存在通信瓶颈？
采用动态负载平衡方案，调度策略的成本如何？

12. ~~什么是域分解和功能分解？如何将全局通信转换为局部通信？什么是表面-容积效应和重复计算？映射的策略是什么？~~

域分解： 也叫数据划分。所要划分的对象是些数据，这些数据可以是算法（或程序的输入数据，计算的输出数据，或者算法所产生的中间结果。

功能分解： 划分的对象是计算，将计算划分为不同的任务，其出发点不同于域分解。

将全局通信转换为局部通信： 采用分治法。

表面-容积效应: 一个任务的通信需求比例与它所操作的子域的表面积，而计算需求却比例于子域的容积。

重复计算： 它也称为冗于计算。有时候可以采用不必要的多余的计算的方法来减少通信要求和/或执行时间。

映射的策略：

(1) 使得任务可以被不同的处理器并发地执行，增强并发性（concurrency）
(2) 将通信频繁的任务放到同一个处理器上，增强局部性（locality）

13. 掌握算法 6.2(并行快排序),6.5(点对最短路径算法), 7.1 (PSRS 排序算法), 7.8 (求最大值算法), 7.9 (求前缀和算法), 7.10 (求元素表序算法), 7.11 (求森林根算法), 9.5 (Cannon 算法), 9.6 (DNS 算法) 的基本原理和伪代码描述。

并行快排序算法

6.2

算法6.2 PRAM-CRCW

```
Begin
  (1) for each processor i do
    (1.1) root=i          //Pi将处理器号i并发写入SM变量root，root的值是不确定的
    (1.2) fi=root         //Pi并发读入root到LM变量fi中
    (1.3) LCi=RCi=n+1     //LCi和RCi初始化，使得不指向任何处理器
  end for
  (2) repeat for each processor i<>root do   //Ai是LM变量, Afi是SM变量;
    if (Ai<Afi)∨(Ai=Afi∧i<fi) then           (Ai= Afi and i<fi)为了排序稳定
      (2.1) LCfi=i        //Pi将i并发写入SM变量LCfi, 竞争为fi的左孩子
      (2.2) if i=LCfi then exit else fi=LCfi endif
    else
      (2.3) RCfi=i        //Pi将i并发写入SM变量RCfi, 竞争为fi的右孩子
      (2.4) if i=RCfi then exit else fi=RCfi endif
    endif
  end repeat
End
```

时间分析：运行时间 T(*n*)=O(log*n*), 处理器数 p(*n*)=*n*, 并行算法的成本 c(*n*)=O(*n*log*n*)

计算原理

6.5

- 假定图中无负权有向回路，记$d^{(k)}_{ij}$为v_i到v_j至多有k-1个中间结点的最短路径长，$D^k=(d^{(k)}_{ij})_{n\times n}$，则
  1. $d^{(1)}_{ij}=w_{ij}$　当 i<>j (如果v_i到v_j之间无边存在记为∞)
     $d^{(1)}_{ij}=0$　当 i=j
  2. 利用组合最优原理：$d^{(k)}_{ij}=\min_{1\le l\le n}\{d^{(k/2)}_{il}+d^{(k/2)}_{lj}\}$　式(A)

  考虑矩阵乘法D*D，有：$d^{(2)}_{ij}=\sum_{1\le l\le n}(d^{(1)}_{il}* d^{(1)}_{lj})$　式(B)

  发现式(A)与式(B)的相似之处，视：“+” → “×”，“min” → “∑”，则式(A)可看作：

  $d^{(k)}_{ij}=\sum_{1\le l\le n}\{d^{(k/2)}_{il}\times d^{(k/2)}_{lj}\}$

  3. 应用矩阵乘法：$D^1\rightarrow D^2\rightarrow D^4\rightarrow\ldots\rightarrow D^{2^{\log n}}$ (= D^n)

输入： 长度为 n 的无序序列，p 台处理器，每台处理器有 n/p 个元素

输出： 长度为 n 的有序序列

Begin

(1) 均匀划分：n 个元素均匀地划分成 p 段，每台处理器有 n/p 个元素。

(2) 局部排序：各处理器利用串行排序算法，排序 n/p 个数。

(3) 正则采样：每台处理器各从自己的有序段中选取 p 个样本元素。

(4) 样本排序：用一台处理器将所有 p^2 个样本元素用串行排序算法排序之。

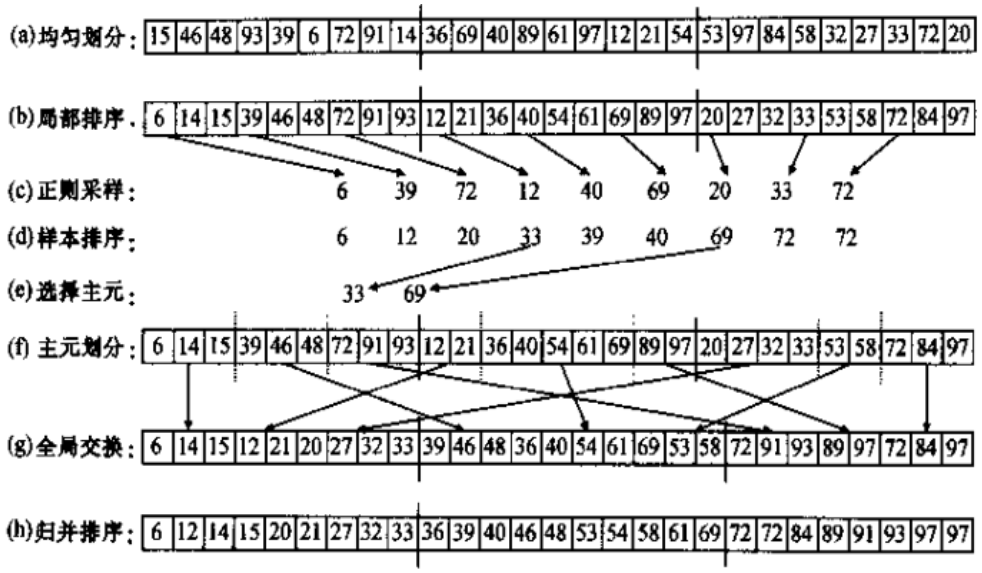
(5) 选择主元：用一台处理器选取 $p-1$ 个主元，并将其播送给其余处理器。

(6) 主元划分：各处理器按主元将各自的有序段划分成 p 段。

(7) 全局交换：各处理器将其辖段按段号交换到相应的处理器。

(8) 归并排序：处理器使用归并排序将所接收的诸段施行排序。

End

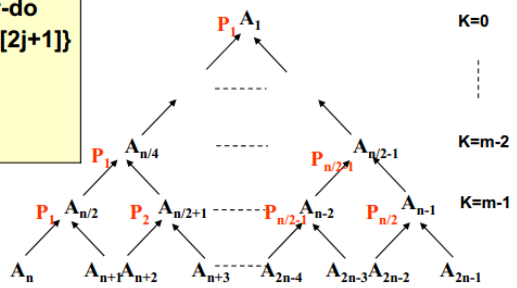


7.1

求最大值

- 算法7.8：SIMD-TC(SM)上求最大值算法

```
Begin
  for k=m-1 to 0 do
    for j=2^k to 2^(k+1)-1 par-do
      A[j]=max{A[2j], A[2j+1]}
    end for
  end for
end
```



时间分析：

$t(n)=m \times O(1)=O(\log n)$　　$p(n)=n/2$

$c(n)=O(n\log n)$ 非成本最优

7.8

前缀和

- 问题定义

  n个元素$\{x_1,x_2,\ldots,x_n\}$，前缀和是n个部分和：

  $S_i=x_1*x_2*\ldots*x_i$，$1\leqslant i\leqslant n$ 这里*可以是＋或×

- 串行算法：$S_i=S_{i-1}*x_i$ 计算时间为 O(n)

- 并行算法：算法7.9 非递归算法

  令A[i]=x_i, i=1~n, B[h,j]和C[h,j]为辅助数组(h=0~logn, j=1~$n/2^h$)

  数组B记录由叶到根正向遍历树中各结点的信息—求和

  数组C记录由根到叶反向遍历树中各结点的信息—播送前缀和

7.9

求元素表序算法

```
(1) 并行做：初始化p[k]和distance[k]                    //O(1)
(2) 执行⌈logn⌉次                                       //O(logn)
    (2.1) 对k并行地做                                   //O(1)
        如果k的后继不等于k的后继之后继，则
        (i)  distance[k]= distance[k]+ distance[p[k]]
        (ii) p[k]=p[p[k]]
    (2.2) 对k并行地做
        rank[k]=distance[k]                             //O(1)
```

7.10

运行时间：t(n)=O(logn)　p(n)=n　（算法7.10）

算法 7.11 SIMD-CREW 上求森林根的算法

输入： 森林 F，弧由$(i,P(i))$指定，$1\leqslant i\leqslant n$

输出： 对每个节点 i，输出包含 i 的树的根 $S(i)$

```
Begin
  for 1≤i≤n par－do
    (1) S(i)=P(i)
    (2) while S(i)≠S(S(i)) do
                S(i)=S(S(i))
        end while
  end for
End
```

7.11

算法:

① 对准:

所有块$A_{i,j}(0\leqslant i, j\leqslant\sqrt{p}-1)$向左循环移动i步;

所有块$B_{i,j}(0\leqslant i, j\leqslant\sqrt{p}-1)$向上循环移动j步;

② 所有处理器$P_{i,j}$做执行$A_{i,j}$和$B_{i,j}$的乘-加运算;

③ 移位:

A的每个块向左循环移动一步;

B的每个块向上循环移动一步;

④ 转②执行$\sqrt{p}-1$次;

9.5

```
//输入: An×n, Bn×n;    输出: Cn×n

Begin
  (1) for k=0 to p^1/2-1 do
        for all Pi,j par-do
          (i) if i>k then
                Ai,j ← Ai,(j+1)mod √p
              endif
          (ii) if j>k then
                Bi,j ← B(i+1)mod √p, j
              endif
        endfor
      endfor
  (2) for all Pi,j par-do Ci,j=0 endfor
  (3) for k=0 to p^1/2-1 do
        for all Pi,j par-do
          (i)  Ci,j=Ci,j+Ai,jBi,j
          (ii) Ai,j ← Ai,(j+1)mod √p
          (iii)Bi,j ← B(i+1)mod √p , j
        endfor
      endfor
End
```

算法9.5

算法: 初始时$a_{i,j}$和$b_{i,j}$存储于寄存器A[0,i,j]和B[0,i,j];

①数据复制:A,B同时在k维复制(一到一播送)

A在j维复制(一到多播送)

B在i维复制(一到多播送)

②相乘运算:所有处理器的A、B寄存器两两相乘

③求和运算:沿k方向进行单点积累求和

9.6

```
//令r(m)表示r的第m位取反;
//{p, rm=d}表示r(0≤r≤p-1)的集合, 这里r的二
//进制第m位为d;
//输入: An×n, Bn×n;    输出: Cn×n

Begin  //以n=2, p=8=2^3举例, q=1, r=(r2r1r0)2
   (1)for m=3q-1 to 2q do  //按k维复制A,B, m=2
        for all r in {p, rm=0} par-do //r2=0的r
          (1.1) Ar(m) ← Ar    //A(100)←A(000)等
          (1.2) Br(m) ← Br    //B(100)←B(000)等
        endfor
      endfor
   (2)for m=q-1 to 0 do      //按j维复制A, m=0
        for all r in {p, rm= r2q+m} par-do //r0=r2的r
          Ar(m) ← Ar  //A(001)←A(000),A(100)←A(101)
        endfor        //A(011)←A(010),A(110)←A(111)
      endfor
   (3) for m=2q-1 to q do  //按i维复制B,m=1
        for all r in {p, rm= rq+m} par-do//r1=r2的r
          Br(m)← Br  //B(010)←B(000),B(100)←B(110)
        endfor       //B(011)←B(001),B(101)←B(111)
      endfor
   (4) for r=0 to p-1 par-do     //相乘, all Pr
        Cr=Ar×Br
      endfor
   (5) for m=2q to 3q-1 do     //求和,m=2
        for r=0 to p-1 par-do
          Cr=Cr+Cr(m)
        endfor
      endfor
End
```

14. 比较并行矩阵乘法 Cannon 和 DNS 的时间复杂度和加速比。

Cannon

- 算法有 $p^{1/2}$ 次循环
- 在每次循环，有 $(n/p^{1/2}) \times (n/p^{1/2})$ 的矩阵乘法： $\Theta(n^3/p^{3/2})$
- 计算复杂度： $\Theta(n^3/p)$
- 在每个循环，每个处理器发送和接收两个大小为 $(n/p^{1/2}) \times (n/p^{1/2})$ 的数据块
  - 每个处理器的通信复杂度： $\Theta(n^2/p^{1/2})$
- 串行算法： $\Theta(n^3)$
- 并行开销： $\Theta(p^{1/2} n^2)$

DNS 运行时间 O(log n)

15. 进程通信的同步方式和聚集方式有哪些？列举主要的通信模式。

同步方式：

原子同步：不可分的操作

控制同步（路障,临界区）：进程的所有操作均必须等待到达某一控制状态

数据同步(锁,条件临界区,监控程序,事件):使程序执行必须等待到某一数据状态

聚集方式：

归约（reduction）

扫描（scan）

主要的通信模式：

点对点（一对一）、广播（一对多）、散播（一对多）、收集（多对一）

全交换（多对多）、移位（置换， 多对多）、归约（多对一）、扫描（多对多）

16. 共享存储编程模型和分布存储编程模型的特征有哪些？举例说明。

共享存储编程模型：

- 多线程：SPMD, MPMD
- 异步
- 单一地址空间
- 显式同步
- 隐式数据分布
- 隐式通信（共享变量的读/写）

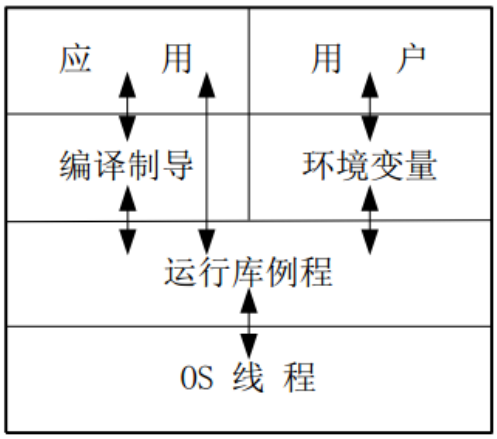
分布存储编程模型：

多地址空间；
消息传递通信；
编程、移植困难；
可伸缩性好
单地址空间

17. 什么是 OpenMP 的编程模型、体系结构、控制结构和数据域子句？

编程模型：基于线程的并行编程模型（Parallel Programming Model），基于编译制导（Compiler Directive Based），支持嵌套并行化（Nested Parallelism Support），支持动态线程（Dynamic Threads），使用Fork-Join并行执行模型。

体系结构：



控制结构：

并行域结构：parallel

共享任务结构：sections，for，single

组合的并行共享工作结构：parallel for，parallel section

同步结构：master，critical，barrier，atomic，flush，ordered

数据域子句：

显式地定义变量的作用范围：private 子句、shared 子句、default 子句、firstprivate 子句、lastprivate 子句、copyin 子句、reduction 子句

18. 什么是 MPI 的消息、数据类型、通信域？如何应用 MPI 的扩展数据类型？

MPI消息：一个消息(message) 指在进程间进行的一次数据交换，在MPI 中，一个消息由通信器、源地址、目的地址、消息标签和数据构成。

MPI 数据类型：预定义类型（Predefined Data Type）和派生数据类型（Derived Data Type）

MPI通信域：提包括进程组（Process Group ）和通信上下文（Communication Context）等内容，用于描述通信进程间的通信关系。

应用扩展数据类型：MPI除了可以发送或接收连续的数据之外还可以处理不连续的数据其基本方法可采取允许用户自定义新的数据类型又称派生数据类型的方法，即可扩展的数据类型。

19. 什么是 MPI 的阻塞通信和非阻塞通信？点到点通信模式有哪些？MPI 群集通信模式？

阻塞通信：当一个通信过程的请求的完成依赖与某个“事件”时，我们说它是阻塞式的。对于发送过程，数据必须被成功地发送或安全地拷贝到系统缓存以便于保存它的应用缓存可以被重用。对于接收操作，数据必须被安全地存储到接收缓存中以便它可以被利用。

非阻塞通信：当一个通信过程的完成不需要等待任何通信事件的完成就可以返回时，它就属于非阻塞通信，例如将数据从用户的内存拷贝到系统缓存或消息的到达。对于一个非阻塞发送，在它完成之后修改或重用应用缓存是很危险的，这就需要编程者保证此时应用缓存是空闲的可以被重用。

点到点通信模式：

标准通信模式、缓冲通信模式、同步通信模式、就绪通信模式。

群集通信模式：

广播、收集、散播、全局收集、全局交换

20. 熟悉掌握用 OpenMP 和 MPI 编写计算 pai 的程序。

MPI-π

$$\pi \approx \sum_{i=1}^{n} \frac{4}{\left(\frac{i-0.5}{n}\right)^2}$$

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.14159265358979323846264643;
   double mypi, pi, h, sum, x, a;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   while (!done)  {
     if (myid == 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;

     h   = 1.0 / (double) n;
     sum = 0.0;
     for (i = myid + 1; i <= n; i += numprocs) {
       x = h * ((double)i - 0.5);
       sum += 4.0 / (1.0 + x*x);
     }
     mypi = h * sum;
     MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);

     if (myid == 0)
       printf("pi is approximately %.16f, Error is
              %.16f\n", pi, fabs(pi - PI25DT));

   }
   MPI_Finalize();
   return 0;
}
```

OpenMp(并行归约)-π

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{     int i;
      double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
      omp_set_num_threads(NUM_THREADS)
      #pragma omp parallel for reduction(+:sum) private(x)
      for (i=1;i<= num_steps; i++){
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
      }
      pi = step * sum;
}
```

21. 了解计算内积、排序、串匹配、Cannon 和 DNS 等算法的 MPI 和 OpenMP 实现。

MPI-点积

```
/* distribute two vectors over all processes such that
   processor 0 has elements 0...99
   processor 1 has elements 100...199
   processor 2 has elements 200...299
   etc.
*/

double dotprod(double a[100], double b[100])
{
   double gresult = lresult = 0.0;
   integer i;
   /* compute local dot product */
   for (i = 0; i < 100; i++) lresult += a[i]*b[i];
   MPI_Allreduce(lresult, gresult, 1, MPI_DOUBLE,
       MPI_SUM, MPI_COMM_WORLD);
   return(gresult);
}
```

MPI-Cannon

```
//输入: An×n, Bn×n;    输出: Cn×n

Begin
   (1) for k=0 to p1/2-1 do
        for all Pi,j par-do
          (i) if i>k then
               Ai,j ← Ai,(j+1)mod √p
                endif
          (ii) if j>k then
               Bi,j ← B(i+1)mod √p , j
              endif
        endfor
      endfor
   (2) for all Pi,j par-do Ci,j=0 endfor
```

算法9.5

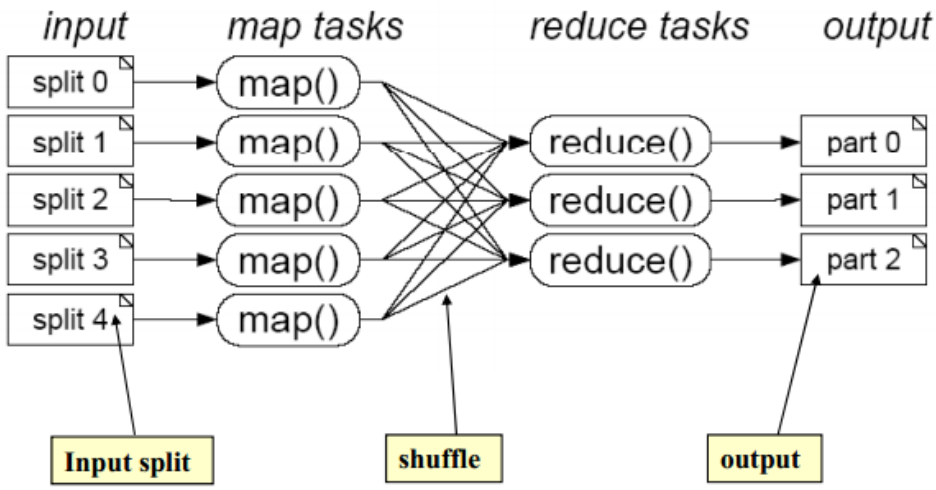
```
(3) for k=0 to p^1/2-1 do
      for all P_i,j par-do
        (i) C_i,j=C_i,j+A_i,jB_i,j
        (ii) A_i,j ← A_i,(j+1)mod √p
        (iii)B_i,j ← B_(i+1)mod√p, j
        endfor
      endfor
End
```

OpenMP-点积

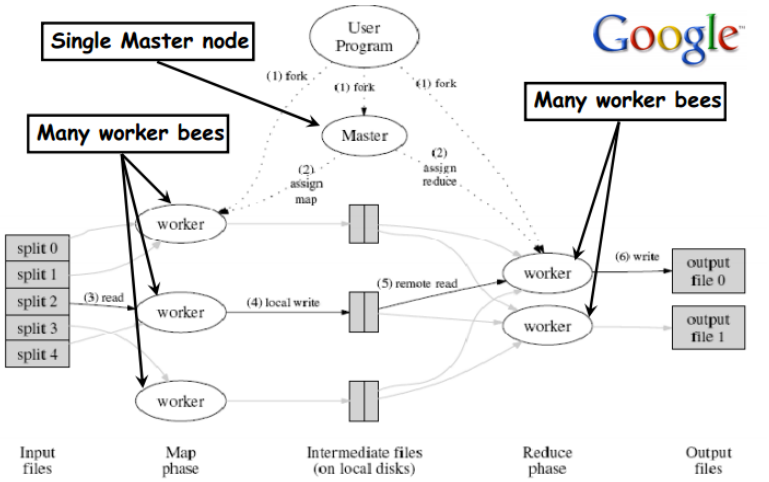
```
#include <omp.h>

main () {
        int   i, n, chunk;
        float a[100], b[100], result;

        /* Some initializations */
        n = 100;
        chunk = 10;
        result = 0.0;
        for (i=0; i < n; i++) {
                a[i] = i * 1.0;
                b[i] = i * 2.0;
        }

        #pragma omp parallel for default(shared) private(i) schedule(static,chunk) \
                                reduction(+:result)
        for (i=0; i < n; i++)
                result = result + (a[i] * b[i]);
        printf("Final result= %f\n",result);
}
```

22. 了解 Map/Reduce 的体系结构和工作原理。

体系结构：



工作原理：



一切都是从最上方的 user program 开始的，user program 链接了 MapReduce 库，实现了最基本的 Map 函数和 Reduce 函数。图中执行的顺序都用数字标记了。

1.MapReduce 库先把 user program 的输入文件划分为 M 份（M 为用户定义），每一份通常有 16MB 到 64MB，如图左方所示分成了 split0~4；然后使用 fork 将用户进程拷贝到集群内其它机器上。

2.user program 的副本中有一个称为 master，其余称为 worker，master 是负责调度的，为空闲 worker 分配作业（Map 作业或者 Reduce 作业），worker 的数量也是可以由用户指定的。

3.被分配了 Map 作业的 worker，开始读取对应分片的输入数据，Map 作业数量是由 M 决定的，和 split 一一对应；Map 作业从输入数据中抽取出键值对，每一个键值对都作为参数传递给 map 函数，map 函数产生的中间键值对被缓存在内存中。

4.缓存的中间键值对会被定期写入本地磁盘，而且被分为 R 个区，R 的大小是由用户定义的，将来每个区会对应一个 Reduce 作业；这些中间键值对的位置会被通报给 master，master 负责将信息转发给 Reduce worker。

5.master 通知分配了 Reduce 作业的 worker 它负责的分区在什么位置（肯定不止一个地方，每个 Map 作业产生的中间键值对都可能映射到所有 R 个不同分区），当 Reduce worker 把所有它负责的中间键值对都读过来后，先对它们进行排序，使得相同键的键值对聚集在一起。因为不同的键可能会映射到同一个分区也就是同一个 Reduce 作业（谁让分区少呢），所以排序是必须的。

6.reduce worker 遍历排序后的中间键值对，对于每个唯一的键，都将键与关联的值传递给 reduce 函数，reduce 函数产生的输出会添加到这个分区的输出文件中。

7.当所有的 Map 和 Reduce 作业都完成了，master 唤醒正版的 user program，MapReduce 函数调用返回 user program 的代码。

所有执行完毕后，MapReduce 输出放在了 R 个分区的输出文件中（分别对应一个 Reduce 作业）。用户通常并不需要合并这 R 个文件，而是将其作为输入交给另一个 MapReduce 程序处理。整个过程中，输入数据是来自底层分布式文件系统（GFS）的，中间数据是放在本地文件系统的，最终输出数据是写入底层分布式文件系统（GFS）的。而且我们要注意 Map/Reduce 作业和 map/reduce 函数的区别：Map 作业处理一个输入数据的分片，可能需要调用多次 map 函数来处理每个输入键值对；Reduce 作业处理一个分区的中间键值对，期间要对每个不同的键调用一次 reduce 函数，Reduce 作业最终也对应一个输出文件。

23. 熟悉掌握 Word Count 算法的 Map/Reduce 实现，了解最短路径、排序、模式匹配等算法的 Map/Reduce 实现。

Word Count（Java）

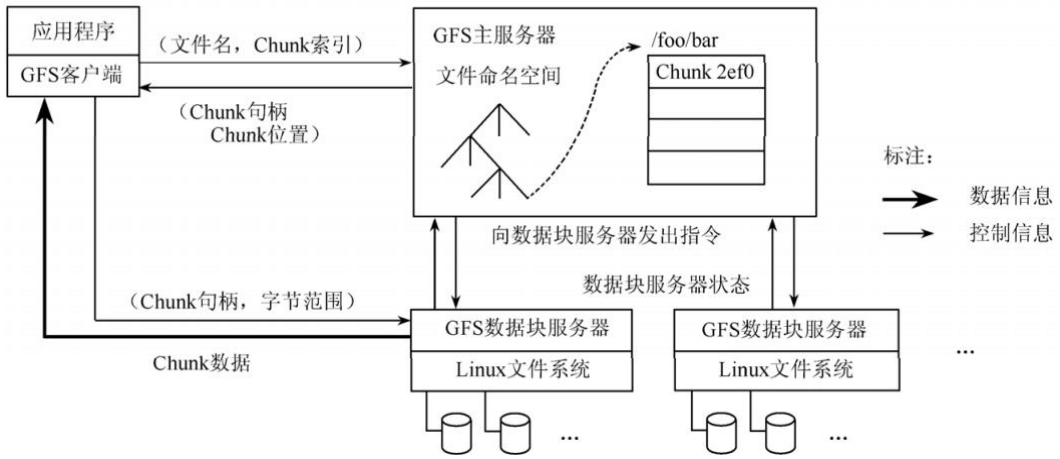
```
1.  package org.myorg;
2.
3.  import java.io.IOException;
4.  import java.util.*;
5.
6.  import org.apache.hadoop.fs.Path;
7.  import org.apache.hadoop.conf.*;
8.  import org.apache.hadoop.io.*;
9.  import org.apache.hadoop.mapreduce.*;
10. import org.apache.hadoop.mapreduce.lib.input.*;
11. import org.apache.hadoop.mapreduce.lib.output.*;
12. import org.apache.hadoop.util.*;
13.
14. public class WordCount extends Configured implements Tool {
15.
16.    public static class Map
17.        extends Mapper<LongWritable, Text, Text, IntWritable> {
18.      private final static IntWritable one = new IntWritable(1);
19.      private Text word = new Text();
20.
21.      public void map(LongWritable key, Text value, Context context)
22.          throws IOException, InterruptedException {
23.        String line = value.toString();
24.        StringTokenizer tokenizer = new StringTokenizer(line);
25.        while (tokenizer.hasMoreTokens()) {
26.          word.set(tokenizer.nextToken());
27.          context.write(word, one);
28.        }
29.      }
30.    }
31.
32.    public static class Reduce
33.        extends Reducer<Text, IntWritable, Text, IntWritable> {
34.      public void reduce(Text key, Iterable<IntWritable> values,
35.          Context context) throws IOException, InterruptedException {
36.
37.        int sum = 0;
38.        for (IntWritable val : values) {
39.          sum += val.get();
40.        }
41.        context.write(key, new IntWritable(sum));
42.      }
```

```
44.
45.     public int run(String [] args) throws Exception {
46.       Job job = new Job(getConf());
47.       job.setJarByClass(WordCount.class);
48.       job.setJobName("wordcount");
49.
50.       job.setOutputKeyClass(Text.class);
51.       job.setOutputValueClass(IntWritable.class);
52.
53.       job.setMapperClass(Map.class);
54.       job.setCombinerClass(Reduce.class);
55.       job.setReducerClass(Reduce.class);
56.
57.       job.setInputFormatClass(TextInputFormat.class);
58.       job.setOutputFormatClass(TextOutputFormat.class);
59.
60.       FileInputFormat.setInputPaths(job, new Path(args[0]));
61.       FileOutputFormat.setOutputPath(job, new Path(args[1]));
62.
63.       boolean success = job.waitForCompletion(true);
64.       return success ? 0 : 1;
65.     }
66.
67.     public static void main(String[] args) throws Exception {
68.       int ret = ToolRunner.run(new WordCount(), args);
69.       System.exit(ret);
70.     }
71. }
72.
```

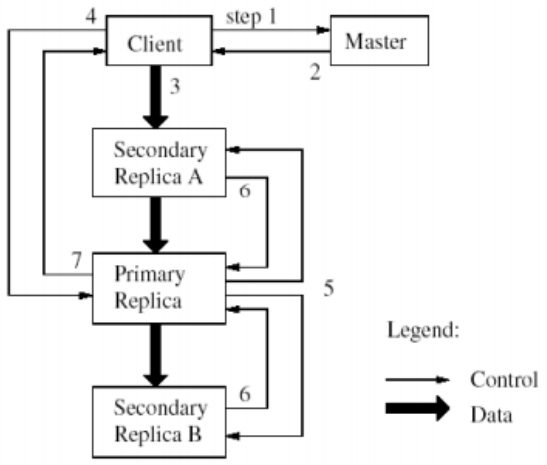
24. 了解分布式文件系统 GFS 的系统架构，数据读取和修改的流程。

GFS 体系结构：

一个 GFS 集群由一个 master 和大量 chunkserver 构成，并被许多客户（Client）访问。



数据读取和修改流程：



1.client 使用固定的块大小将应用程序指定的文件名和字节偏移转换成文件的一个块索引（chunk index）。

2.给 master 发送一个包含文件名和块索引的请求。

3.master 回应对应的 chunk handle 和副本的位置（多个副本）。

4.client 以文件名和块索引为键缓存这些信息。（handle 和副本的位置）。

5.Client 向其中一个副本发送一个请求，很可能是最近的一个副本。请求指定了 chunk handle（chunkserver 以 chunk handle 标识 chunk）和块内的一个字节区间。

6.除非缓存的信息不再有效（cache for a limited time）或文件被重新打开，否则以后对同一个块的读操作不再需要 client 和 master 间的交互。

通常 Client 可以在一个请求中询问多个 chunk 的地址，而 master 也可以很快回应这些请求。

~~25. 什么是虚拟化？服务器虚拟化的类型和目的？虚拟化的关键组件是什么？~~

虚拟化： 在虚拟化技术中，可同时运行多个操作系统，而且每一个操作系统中都有多

服务器虚拟化的类型：

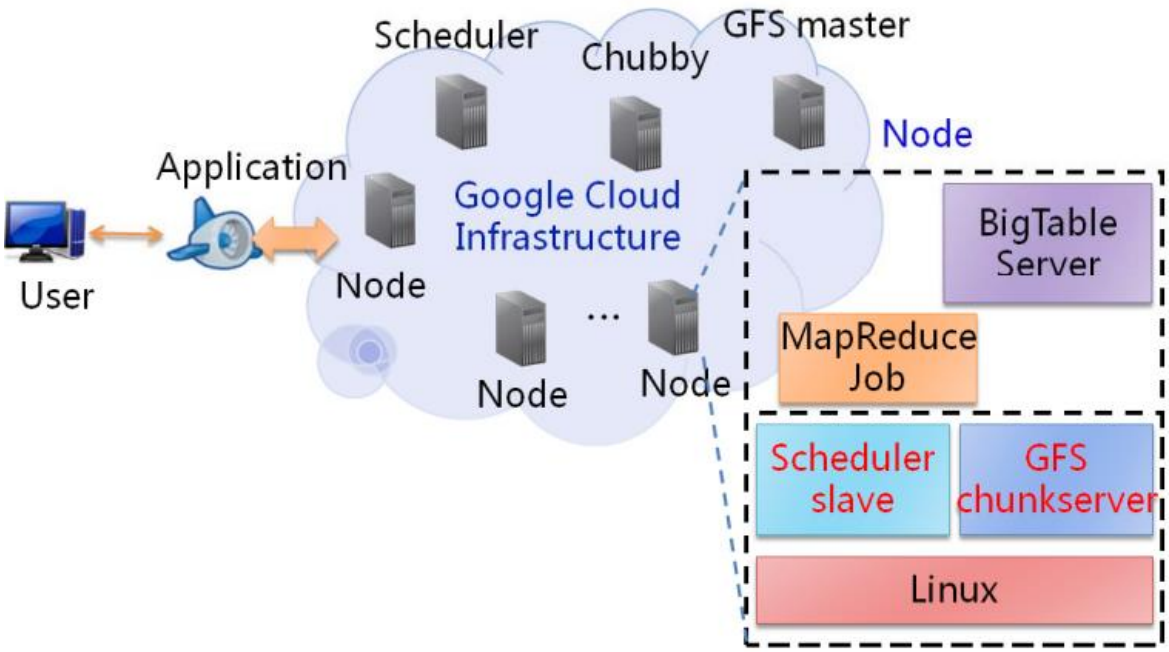
- **拆分（Partition）**
  - **某台计算机性能较高，而工作负荷小，资源没有得到充分利用。可以将这台计算机拆分为逻辑上的多台计算机，同时供多个用户使用。这样可以使此服务器的硬件资源得到充分的利用。**
  - **目的：提高资源利用率**
- **整合（Consolidation）**
  - **如有大量性能一般的计算机，可应用虚拟整合技术，将大量性能一般的计算机整合为一台计算机，以满足客户对整体性能的要求**
  - **目的：通过整合，获得高性能，满足特定数据计算要求。**
- **迁移 （Migration）**
  - **将一台逻辑服务器中的闲置的一部分资源动态的加入到另一台逻辑服务器中，提高另一方的性能。**
  - **目的：实现资源共享，实现跨系统平台应用等。**

服务器虚拟化目的： 程序在运行，每一个操作系统都运行在一个虚拟的 CPU 或者是虚拟主机上将服务器物理资源抽象成逻辑资源，让一台服务器变成几台甚至上百台相互隔离的虚拟服务器，不再受限于物理上的界限，而是让 CPU、内存、磁盘、I/O 等硬件变成可以动态管理的“资源池”，从而提高资源的利用率，简化系 资源池 ，从而提高资源的利用率，简化系统管理，实现服务器整合，让 IT 对业务的变化更具适应力。

虚拟化关键组件： VMM（Virtual Machine Manager），又称为 Hypervisor ，负责为虚拟机统一分配 CPU、内存和外设，调度虚拟资源。

26. 了解典型云计算平台（Google，Hadoop，Amazon，Microsoft）的基本模块及关键技术。

Google的云平台架构

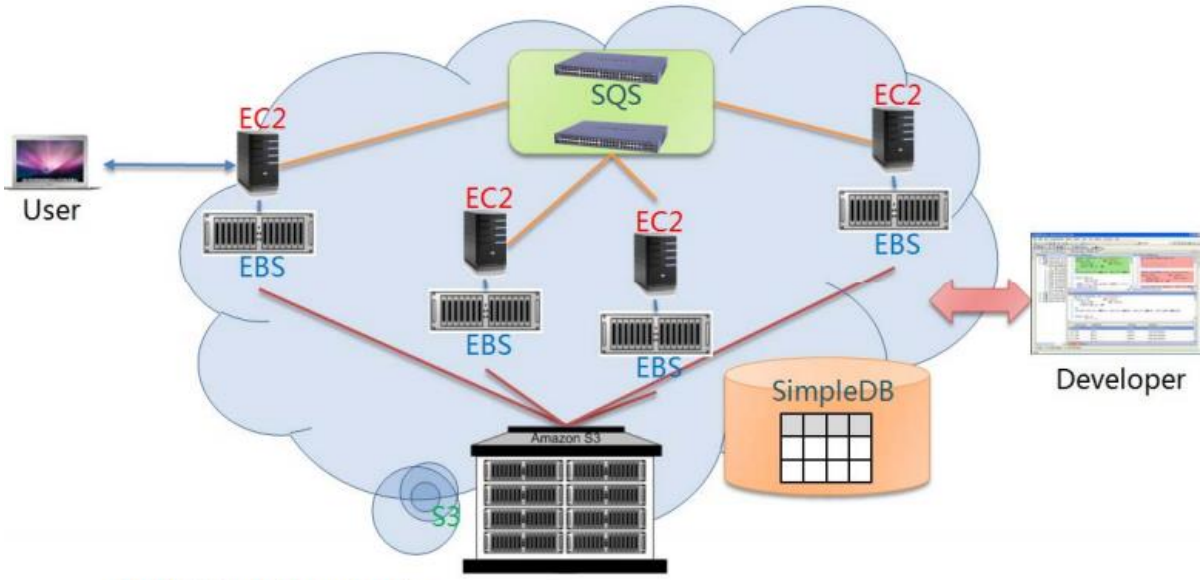


开源云计算系统

开源云计算系统	对应的商用云计算系统
Hadoop HDFS	Google GFS
Hadoop MapReduce	Google MapReduce
Hadoop HBase	Google Bigtable
Hadoop ZooKeeper	Google Chubby
OpenStack, Eucalyptus、Enomaly ECP、Nimbus	Amazon EC2
OpenStack, Eucalyptus	Amazon S3
Sector and Sphere	无直接对应系统
abiquo	无直接对应系统
MongoDB	无直接对应系统

Microsoft 云平台

Amazon Elastic Computing Cloud



- SQS: Simple Queue Service
- EC2: Running Instance of Virtual Machines
- EBS: Elastic Block Service, Providing the Block Interface, Storing Virtual Machine Images
- S3: Simple Storage Service,
- SimpleDB: Simplified Database

亚马逊关键技术：弹性计算云 EC2