中国科学技术大学 计算机科学与技术系
University of Science and Technology of China
DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY

算法设计与分析
Design and Analysis of Algorithms

主讲人　徐 云

Fall 2012, USTC

第5章(补充) 并行算法

5.1 概述

- 什么是并行计算
- 为什么要并行计算
- 并行计算的实现方案
- 并行计算的研究领域
- Top500 && China Top100
- 并行机体系结构

什么是并行计算

- A parallel computer is a “collection of processing elements that communicate and cooperate to solve large problem fast”. – David E. Culler
- Or all processors cooperate to solve a single problem
- Daily life examples:
  - House construction
  - Car manufacturing
  - Grocery store operation

为什么要并行计算（1）

- Interest in parallelism since the very ancient era of computers(e.g. ILLIAC IV of 1967 had 64 processors)
- Parallel Processing is an effective answer for the tremendous future computing requirements.
  - i.e. Muti-core CPU and Processor
- applications impulses:
  - Data-intensive applications: videoconferencing, virtual reality, large database and data mining, speech recognition, biology, image and signal processing, etc
  - Computing-intensive applications: numerical simulation(e.g. forecasting, manufacturing, chemistry, aerodynamics)
  - Network-intensive applications

为什么要并行计算（2）

- Grand challenges:
  - Science today: experimentation, theory, simulation(or computation)
  - Simulation relies heavily on parallel processing
  - America HPCC project, ASCI project
- In one words: Parallel processing promises increase of
  - Performance(e.g. large, fast, cost)
  - Preciseness
  - Reliability
  - and, large set of computational problems are inherently parallel in nature. But their existing applications are designed for uniprocessor systems. Their parallelization is required.

并行处理实现方案

- Multi-core CPU and Processor (lowest cost)
- Cluster of workstations (lower cost)
- Multiprocessor workstations ($60,000)
  - DEC Firefly, Apollo DN 10000, SUN SPARCstation 20
- Shared memory multiprocessors ($200,000-400,000)
  -Sequent Symmetry, Encore Multimax, SGI Challenge, SUN SPARCserver 2000
- Distributed memory multicomputers ($200,000-400,000)
  - Intel iPSC/860, NCUBE/2, Meiko
- Massively parallel processors ($5,000,000)
  - Intel Paragon, TMC CM-5, CRAY T3D, IBM SP-2

并行处理的研究领域

- Design of parallel computers: How to the number of processors, communication throughput, data sharing, etc.
- Design of parallel algorithms: Parallel algorithms may be quite different from their sequential counterparts.
- Design of parallel software:
  - Operating systems
  - Compiles
  - Libraries
  - Tools: debuggers, performance analyzers
- Applications of parallel computing

Top500 http://www.top500.org

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT	186368	2566.00	4701.00	4040.00
2	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
3	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 Dawning	120640	1271.00	2984.30	2580.00
4	GSIC Center, Tokyo Institute of Technology Japan	TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows / 2010 NEC/HP	73278	1192.00	2287.63	1398.61
5	DOE/SC/LBNL/NERSC United States	Hopper - Cray XE6 12-core 2.1 GHz / 2010 Cray Inc.	153408	1054.00	1288.63	2910.00

China Top100 '10.11

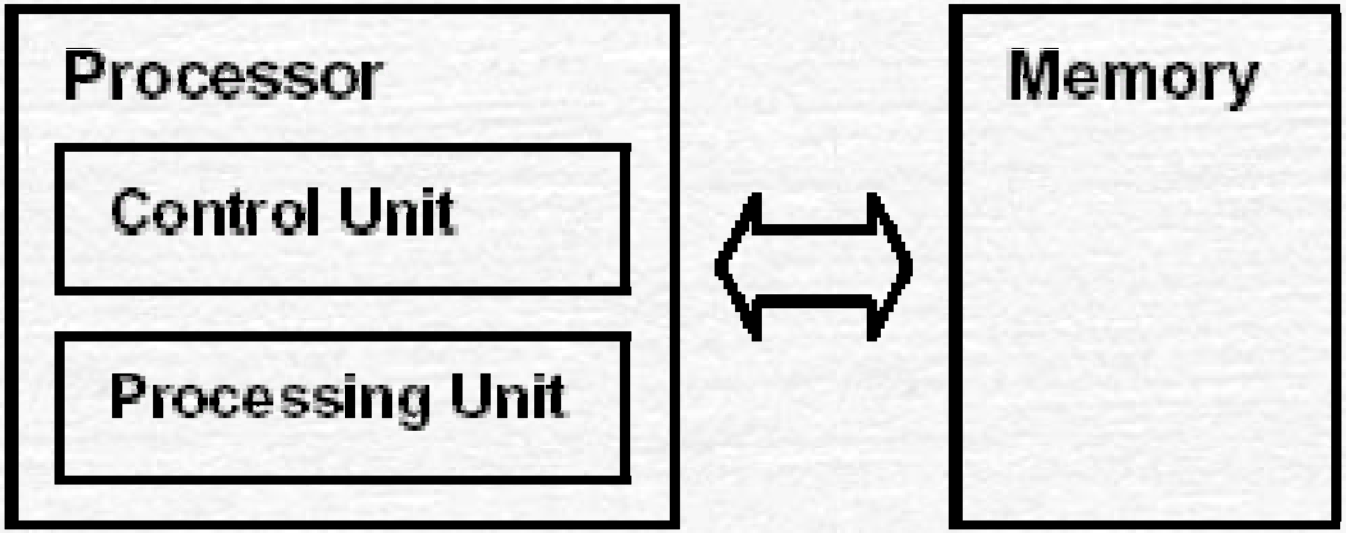
2010年中国高性能计算机性能TOP100排行榜

张云泉 孙家昶 袁国兴 张林波
中国软件行业协会数学软件分会
国家863高性能计算机评测中心
中国计算机学会高性能计算专业委员会
(http://www.samss.org.cn)
(2010年11月1日)

并行机体系结构（1）

单指令流单数据流

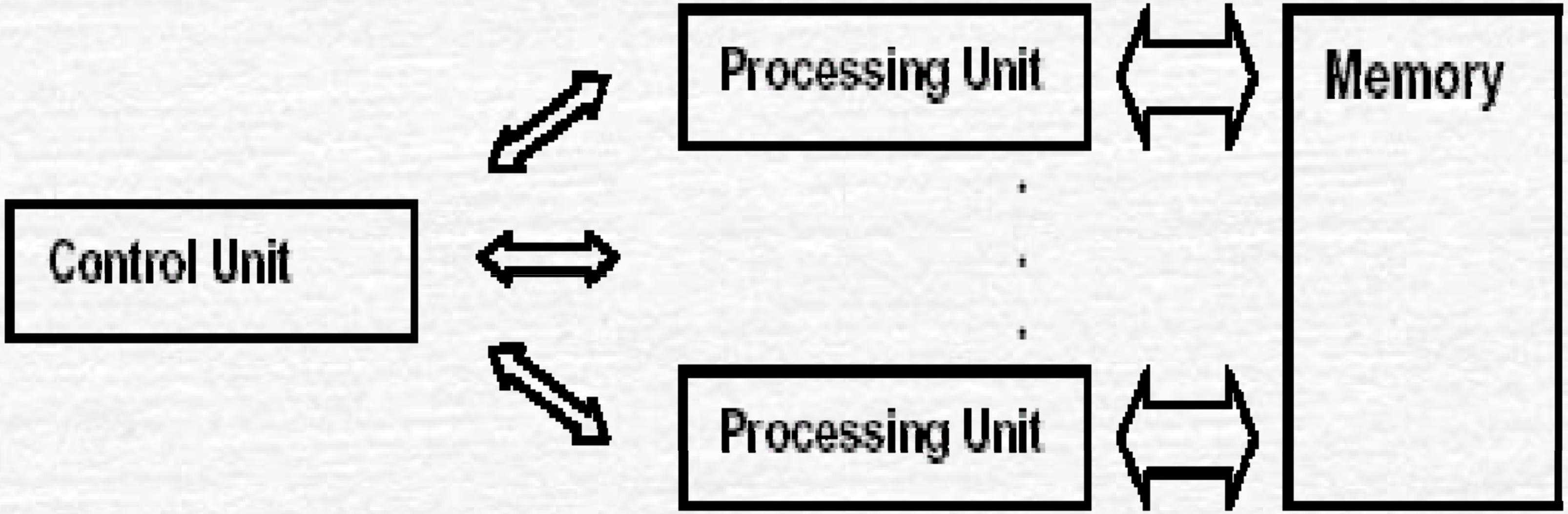
(SISD computer -Von Neumann's model)



Single control unit and single processing unit

并行机体系结构（2）

单指令流多数据流（SIMD computer）

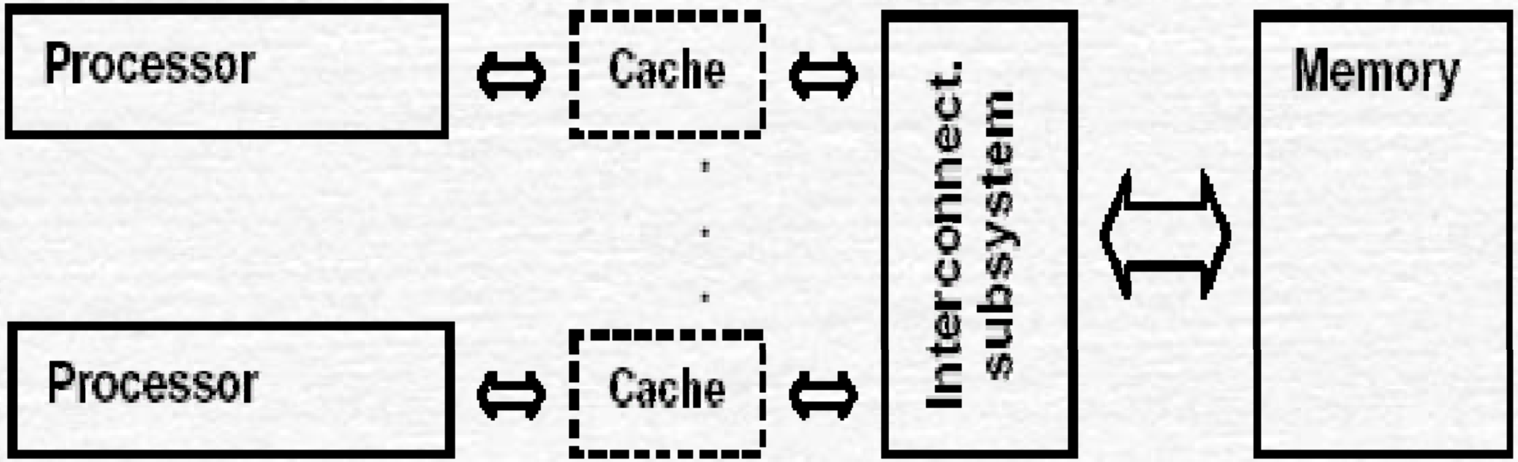


Single control unit and multiple processing unit.

并行机体系结构（3）

对称多处理机SMP

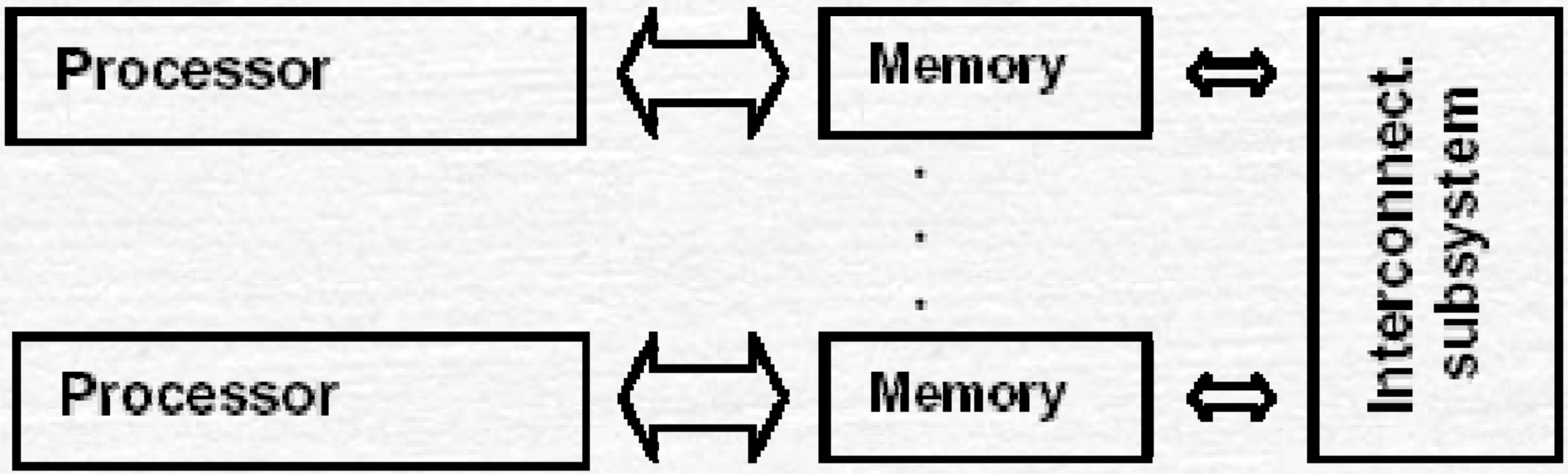
（Symmetric multiprocessor - MIMD-SM）



并行机体系结构（4）

分布式存储并行机MMP

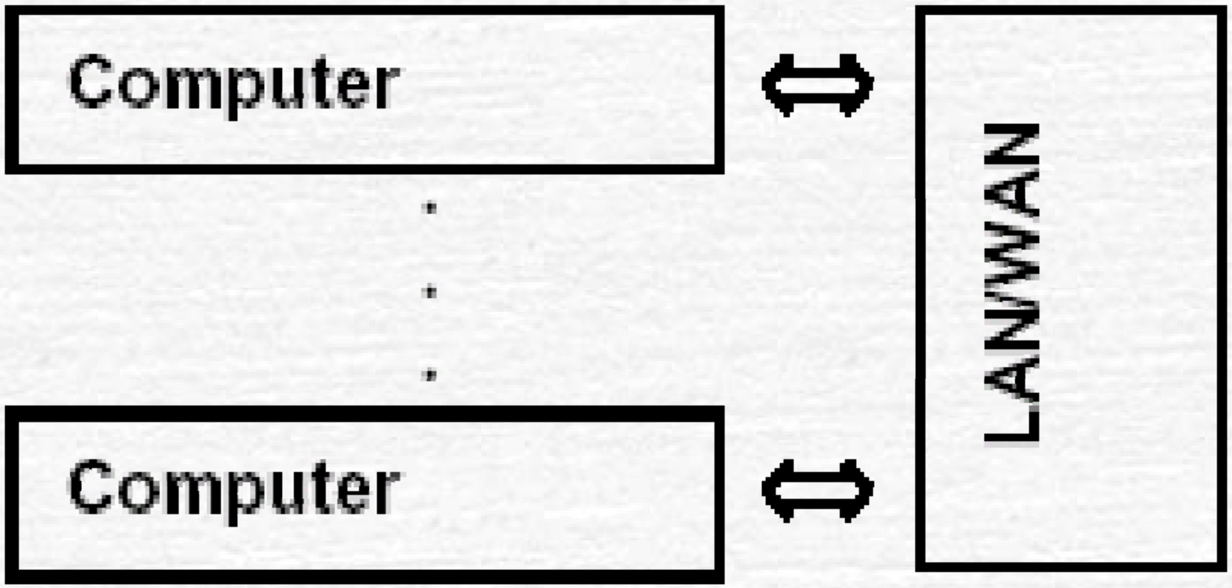
（Massively parallel processor – MIMD-DM/DSM）



并行机体系结构（5）

工作站机群

（Cluster of workstations – MIMD-DM ）



发展为SMP/DSM机群

第5章(补充) 并行算法

5.2 并行算法基础知识

- 并行算法的定义和分类
- 并行算法的表达
- 并行算法的复杂性度量
- 并行计算模型

定义和分类

- 并行算法的定义
  - 一些可同时执行的诸进程的集合，这些进程互相作用和协调动作从而达到给定问题的求解。
- 并行算法的分类
  - 数值计算和非数值计算
  - 同步算法和异步算法
  - 分布算法
  - 确定算法和随机算法

并行算法的表达

- **描述语言**

  可以使用类Algol、类Pascal等；

  在描述语言中引入并行语句。

- **并行语句示例**

  Par-do语句

```
for i=1 to n par-do
    ……
end for
```

  for all语句

```
for all Pi, where 0≤i≤k
  ……
end for
```

复杂性度量

- 并行算法的复杂性度量指标
  - 运行时间$t(n)$:包含计算时间和通讯时间，分别用计算时间步和选路时间步作单位。n为问题实例的输入规模。
  - 处理器数$p(n)$
  - 并行算法成本$c(n)$: $c(n)=t(n)p(n)$
  - 成本最优性：若$c(n)$等于在最坏情形下串行算法所需要的时间，则并行算法是成本最优的。
  - 加速比$S_p(n)$： $S_p(n)=t_s(n)/t_p(n)$，其中$t_s(n)$为求解问题的最快的串行算法在最坏情形下所需的运行时间，$t_p(n)$为求解同一问题的并行算法在最坏情形下的运行时间。

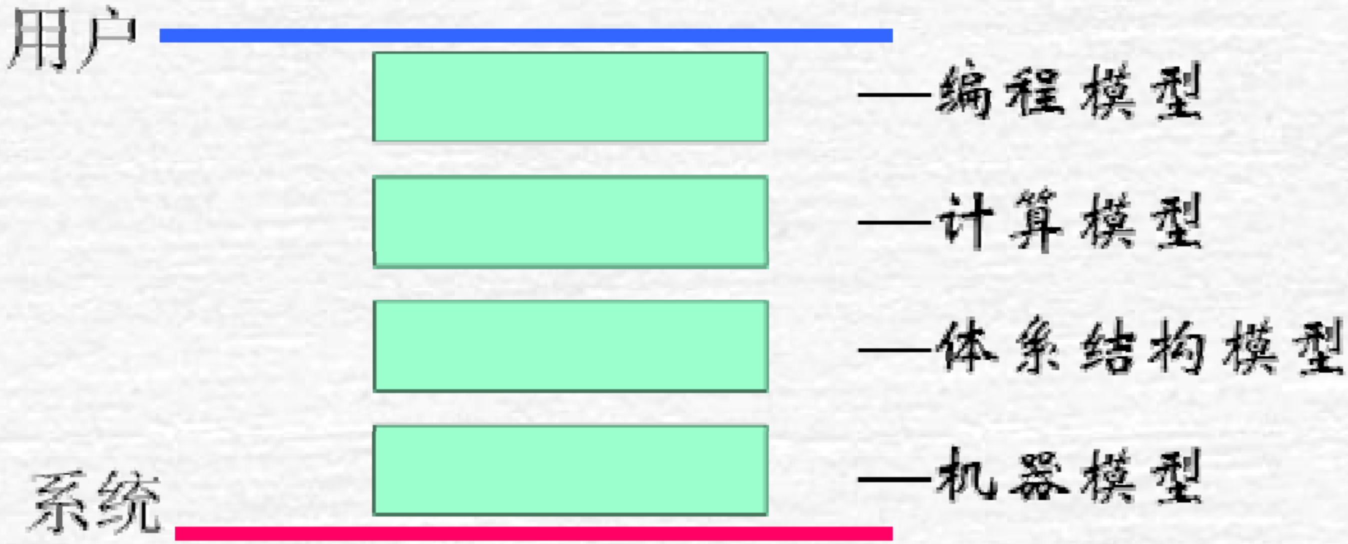
注：1)加速比$S_p(n)$反映算法的并行性对运行时间的改进程度。

2)若$S_p(n)=p(n)$，则达到线性加速；若$S_p(n)>p(n)$，则为超线性加速(一般出现在某些特殊的应用中，如并行搜索等)。

  - 总运算量$W(n)$: 并行算法求解问题所完成的总的操作步数。

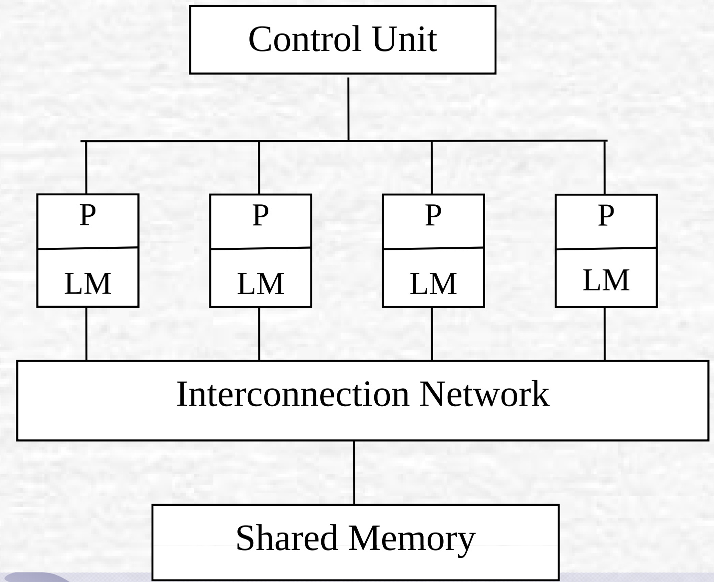
并行计算模型

- 并行计算模型的作用
  - 将并行计算机的基本算法特征抽象出来，作为并行算法分析、设计、性能评测的基础
  - 位于的层次



并行计算模型：PRAM模型（1）

- 基本概念
  - 由Fortune和Wyllie1978年提出，又称SIMD-SM模型。有一个集中的共享存储器和一个指令控制器，通过SM的R/W交换数据，隐式同步计算。
- 结构图



并行计算模型: PRAM模型（2）

- 优点
  - 适合并行算法表示和复杂性分析，易于使用，隐藏了并行机的通讯、同步等细节。
- 缺点
  - 不适合MIMD并行机，忽略了SM的竞争、通讯延迟等因素

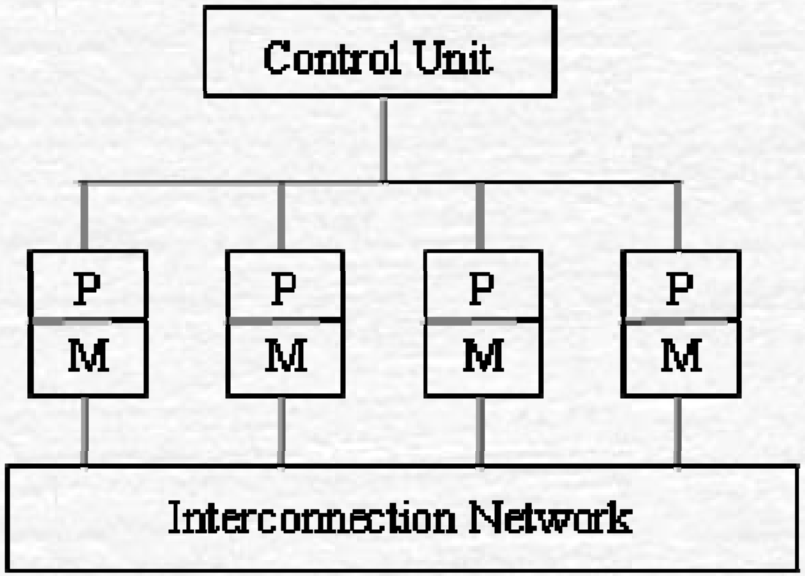
并行计算模型：SIMD-IN模型

- 基本概念

  -又称SIMD-DM模型，分布式存储，处理器通过互连网络相连，用传递数据方式实现通讯，算法时间复杂性考虑计算和选路(时间)，结构图如下：

- 常见模型

  SIMD-LC 一维线性连接
  SIMD-MC 网孔连接
  SIMD-TC 树形连接
  SIMD-MT 树网连接
  SIMD-HC 超立方连接
  SIMD-CCC 立方环连接
  SIMD-SE 洗牌交换连接



并行计算模型：异步APRAM模型（1）

- 基本概念
  - 又称分相（Phase）PRAM或MIMD-SM。每个处理器有其局部存储器、局部时钟、局部程序；无全局时钟，各处理器异步执行；处理器通过SM进行通讯；处理器间依赖关系，需在并行程序中显式地加入同步路障。
- 指令类型

(1)全局读　　(2)全局写

(3)局部操作　　(4)同步

并行计算模型：异步APRAM模型（2）

- 计算过程

  由同步障分开的全局相组成

	处理器1	处理器2		处理器p
	read x_1	read x_2	…	read x_n
phase1	read x_2	*		*
	*	write to B		*
	write to A	write to C		write to D
同步障				
	read B	read A		read C
phase2	*	*		*
	write to B	write to D		
同步障				
	*	write to C		write to B
	read D			read A
				write to B
同步障				

并行计算模型：异步APRAM模型（3）

- **计算时间**

设局部操作为单位时间；全局读/写平均时间为d，d随着处理器数目的增加而增加；同步路障时间为B=B(p)非降函数。

满足关系 $2 \le d \le B \le p$；　$B = B(p) = O(d \log p)$ 或 $O(d \log p / \log d)$

令 t_{ph} 为全局相内各处理器执行时间最长者，则APRAM上的计算时间为

$$T = \sum t_{ph} + B \times \text{同步障次数}$$

- **优缺点**

易编程和分析算法的复杂度，但与现实相差较远，其上并行算法非常有限，也不适合MIMD-DM模型。

并行计算模型：BSP模型（1）

- 基本概念
  - 由Valiant(1990)提出的，“块”同步模型，是一种异步MIMD-DM模型，支持消息传递系统，块内异步并行，块间显式同步。
- 模型参数
  - *p*：处理器数(带有存储器)
  - *l*：同步障时间(Barrier synchronization time)
  - *g*：带宽因子(time steps/packet)=1/bandwidth

并行计算模型：BSP模型（2）

- **计算过程**

  由若干超级步组成，

  每个超级步计算模式为左图

- **优缺点**

  强调了计算和通讯的分离，提供了一个编程环境，易于程序复杂性分析。但需要显式同步机制，限制至多h条消息的传递等。

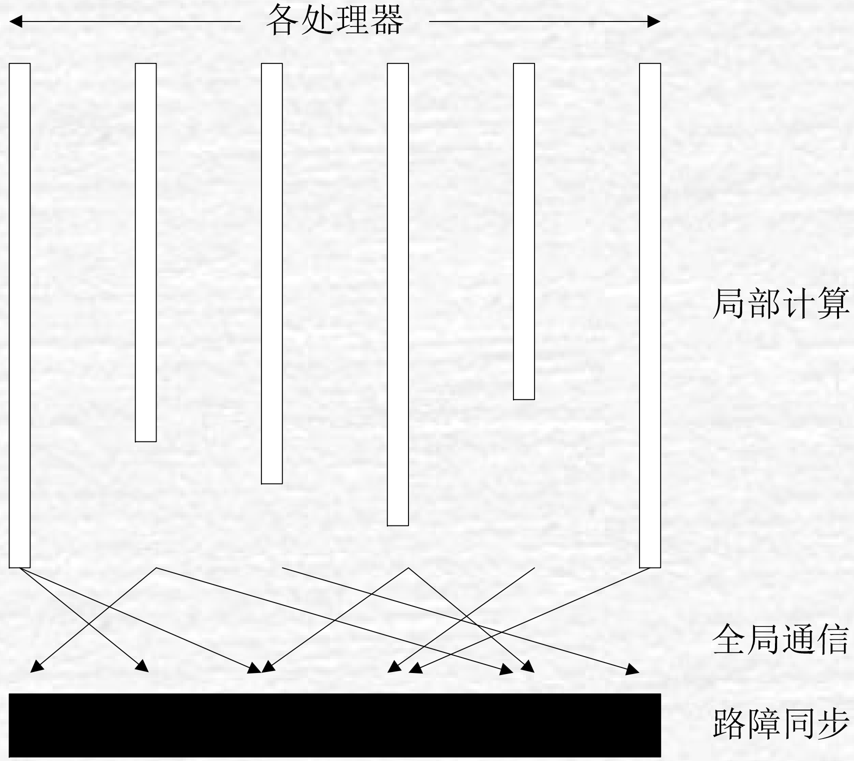


图4.3

并行计算模型：logP模型（1）

- 基本概念
  - 由Culler(1993)年提出的，是一种分布存储的、点到点通讯的多处理机模型，其中通讯由一组参数描述，实行隐式同步。
- 模型参数
  - L： network latency
  - o： communication overhead
  - g： gap=1/bandwidth
  - P： #processors

注：L和g反映了通讯网络的容量

并行计算模型: logP模型 (2)

- 优缺点

  捕捉了MPC的通讯瓶颈，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；但难以进行算法描述、设计和分析。

- BSP vs. LogP

  BSP→LogP：BSP块同步→BSP子集同步→BSP进程对同步 = LogP

  BSP可以常数因子模拟LogP，LogP可以对数因子模拟BSP

  BSP = LogP+Barriers – Overhead

  BSP提供了更方便的程设环境，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

第5章(补充) 并行算法

5.3 并行程序设计

- 并行程序设计现状
- 并行语言的构造方法
- 并行性问题
- 交互/通信问题
- 并行编程标准
- 基本并行化方法
- 两个实例

并行程序设计现状

- 技术先行，缺乏理论指导
- 程序的语法/语义复杂，需要用户自己处理
  - 任务/数据的划分/分配
  - 数据交换
  - 同步和互斥
  - 性能平衡
- 并行语言缺乏代可扩展和异构可扩展，程序移植困难，重写代码难度太大
- 环境和工具缺乏较长的生长期，缺乏代可扩展和异构可扩展
- 并行算法的设计及并行程序的编制已成为目前制约并行计算应用的主要障碍

并行语言的构造方法（1）

串行代码段

```
for ( i= 0; i<N; i++ ) A[i]=b[i]*b[i+1];
for (i= 0; i<N; i++) c[i]=A[i]+A[i+1];
```

(a) 使用库例程构造并行程序

```
id=my_process_id();
p=number_of_processes();
for ( i= id; i<N; i=i+p) A[i]=b[i]*b[i+1];
barrier();
for (i= id; i<N; i=i+p) c[i]=A[i]+A[i+1];
```

例子: **MPI,PVM, Pthreads**

(b) 扩展串行语言

my_process_id,number_of_processes(), and barrier()

```
A(0:N-1)=b(0:N-1)*b(1:N)
c=A(0:N-1)+A(1:N)
```

例子: **Fortran 90**

(c) 加编译注释构造并行程序的方法

```
#pragma parallel
#pragma shared(A,b,c)
#pragma local(i)
{
# pragma pfor iterate(i=0;N;1)
for (i=0;i<N;i++) A[i]=b[i]*b[i+1];
# pragma synchronize
# pragma pfor iterate (i=0; N; 1)
for (i=0;i<N;i++)c[i]=A[i]+A[i+1];
}
```

例子：**SGI power C, OpenMP**

并行语言的构造方法（2）

三种并行语言构造方法比较

方法	实例	优点	缺点
库例程	MPI, PVM	易于实现，不需要新编译器	无编译器检查，分析和优化
扩展	Fortran90	允许编译器检查、分析和优化	实现困难,需要新编译器
编译器注释	SGI powerC, HPF	介于库例程和扩展方法之间，在串行平台上不起作用.	

并行性问题（1）

- 进程的同构性
  - SIMD: 所有进程在同一时间执行相同的指令
  - MIMD:各个进程在同一时间可以执行不同的指令
    - ✓SPMD: 各个进程是同构的，多个进程对不同的数据执行相同的代码(一般是数据并行的同义语)
    - ✓常对应并行循环，数据并行结构，单代码
    - ✓MPMD:各个进程是异构的，多个进程执行不同的代码（一般是任务并行，或功能并行，或控制并行的同义语）
    - ✓常对应并行块，多代码

要为有1000个处理器的计算机编写一个完全异构的并行程序是很困难的

并行性问题（2）

SPMD程序的构造方法

用单代码方法说明SPMD

要说明以下SPMD程序:

```
parfor  (i=0; i<=N, i++) foo(i)
```

用户需写一个以下程序:

```
pid=my_process_id();
numproc=number_of _processes();
parfor  (i=pid; i<=N, i=i+numproc) foo(i)
```

此程序经编译后生成可执行程序A, 用shell脚本将它加载到N个处理结点上:

```
run A –numnodes N
```

用数据并行程序的构造方法

要说明以下SPMD程序:

```
parfor  (i=0; i<=N, i++) {
        C[i]=A[i]+B[i];
        }
```

用户可用一条数据赋值语句:

```
C=A+B
```

或

```
forall (i=1,N) C[i]=A[i]+B[i]
```

并行性问题（3）

MPMD程序的构造方法

用多代码方法说明MPMD

对不提供并行块或并行循环的语言要说明以下MPMD程序:

parbegin S1 S2 S3 parend

用户需写3个程序, 分别编译生成3个可执行程序S1 S2 S3, 用shell脚本将它们加载到3个处理结点上:

run S1 on node1

run S2 on node1

run S3 on node1

S1, S2和S3是顺序语言程序加上进行交互的库调用.

用SPMD形式构造

要说明以下MPMD程序:

parbegin S1 S2 S3 parend

可以用以下SPMD程序:

```
parfor  (i=0; i<3, i++)  {
        if (i=0) S1
        if (i=1) S2
        if (i=2) S3
        }
```

因此，对于可扩展并行机来说，只要支持SPMD就足够了

并行性问题（4）

- 静态和动态并行性

程序的结构：由它的组成部分构成程序的方法

静态并行性：程序的结构以及进程的个数在运行之前(如编译时，连接时或加载时)就可确定，就认为该程序具有静态并行性.

静态并行性的例子：

parbegin P, Q, R parend

其中P,Q,R是静态的

动态并行性：否则就认为该程序具有动态并行性．即意味着进程要在运行时创建和终止

动态并行性的例子：

```
while (C>0) begin
        fork (foo(C));
        C:=boo(C);
end
```

并行性问题（5）

开发动态并行性的一般方法: Fork/Join

```
Process A:
begin
      Z:=1
      fork(B);
      T:=foo(3);
end
```

```
Process B:
begin
      fork(C);
      X:=foo(Z);
      join(C);
      output(X+Y);
end
```

```
Process C:
begin
      Y:=foo(Z);
end
```

Fork: 派生一个子进程

Join: 强制父进程等待子进程

并行性问题（6）

- 并行度(Degree of Parallelism, DOP):同时执行的分进程数.

- 并行粒度(Granularity): 两次并行或交互操作之间所执行的计算负载.
  - ✓指令级并行
  - ✓块级并行
  - ✓进程级并行
  - ✓任务级并行

- 并行度与并行粒度大小常互为倒数: 增大粒度会减小并行度.
- 增加并行度会增加系统(同步)开销

交互/通信问题(1)

- 交互：进程间的相互影响
- 交互的类型
  - 通信：两个或多个进程间传送数的操作
  - 通信方式：
    - ✓共享变量
    - ✓父进程传给子进程(参数传递方式)
    - ✓消息传递

交互/通信问题（2）

- 同步：导致进程间相互等待或继续执行的操作

  同步方式:

  ✓原子同步

  ✓控制同步(路障,临界区)

  ✓数据同步(锁,条件临界区,监控程序,事件)

例子：

原子同步

```
parfor (i:=1; i<n; i++) {
    atomic{x:=x+1; y:=y-1}
}
```

路障同步

```
parfor(i:=1; i<n; i++){
    Pi
    barrier
    Qi
}
```

临界区

```
parfor(i:=1; i<n; i++){
    critical{x:=x+1; y:=y-1}
}
```

数据同步(信号量同步)

```
parfor(i:=1; i<n; i++){
    lock(S);
    x:=x+1;
    y:=y-1;
    unlock(S)
}
```

交互/通信问题（3）

- **聚集(aggregation)**：用一串超步将各分进程计算所得的部分结果合并为一个完整的结果，每个超步包含一个短的计算和一个简单的通信或/和同步.

聚集方式：

- ✓ 归约
- ✓ 扫描

例子：计算两个向量的内积

```
parfor(i:=1; i<n; i++){
    X[i]:=A[i]*B[i]
    inner_product:=aggregate_sum(X[i]);
    }
```

并行编程标准（1）

- 数据编程语言标准
  - Fortran90, HPF(1992), Fortran95/2001: 显式数据分布描述,并行DO循环
- 线程库标准(Thread Library)
  - Win32 API
  - POSIX threads线程模型
- 编译制导(Compiler Directives)
  - OpenMP : portable shared memory parallelism
- 消息传递库标准(Message Passing Libraries)
  - MPI : Message Passing Interface
  - PVM : Parallel Virtual Machine

数据并行编程

共享变量编程

消息传递编程

并行编程标准（2）

- 所有并行编程标准可以分为三类：
  - 数据并行
    - ✓ HPF, Fortran90
    - ✓ 用于SMP, DSM
  - 共享编程
    - ✓ OpenMP
    - ✓ 用于SMP, DSM
  - 消息传递
    - ✓ MPI, PVM
    - ✓ 用于所有并行计算机
- 三者可混合使用：
  - 如，对以SMP为节点的Cluster来说，可以在节点间进行消息传递，在节点内进行共享变量编程

实例：计算Pi的串行程序

- C语言写的串行程序

```
/* Seriel Code */
static long num_steps = 100000;
double step;
void main ()
{     int i;
      double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
      for (i=1;i<= num_steps; i++){
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
      }
      pi = step * sum;
}
```

实例1: 计算Pi的OpenMP程序

- 使用private子句和critical部分并行化的程序

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

实例2: 计算Pi的MPI程序 (1)

```
#include <stdio.h>
#include <mpi.h>
#include <math.h>
long    n,              /*number of slices       */
        i;              /* slice counter         */
double sum,             /* running sum           */
        pi,             /* approximate value of pi */
        mypi,
        x,              /* independent var.      */
        h;              /* base of slice         */
int group_size,my_rank;

main(argc,argv)
int argc;
char* argv[];
```

```
{       int group_size,my_rank;
        MPI_Status status;
        MPI_Init(&argc,&argv);
        MPI_Comm_rank( MPI_COMM_WORLD, &my_rank);
        MPI_Comm_size( MPI_COMM_WORLD, &group_size);

        n=2000;
        /* Broadcast n to all other nodes */
        MPI_Bcast(&n,1,MPI_LONG,0,MPI_COMM_WORLD);
        h = 1.0/(double) n;
        sum = 0.0;
        for (i = my_rank+1; i <= n; i += group_size) {
                x = h*(i-0.5);
                sum = sum +4.0/(1.0+x*x);
                }
        mypi = h*sum;
        /*Global sum */
        MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_W
        ORLD);
        if(my_rank==0) {        /* Node 0 handles output */
                printf("pi is approximately : %.16lf\n",pi);
                }
        MPI_Finalize();
}
```

第5章(补充) 并行算法

5.4 一般设计方法和过程

- 一般设计方法
  - 串行算法的直接并行化
  - 从问题描述开始设计并行算法
  - 借用已有的算法求解新问题
- PCAM设计过程

串行算法的直接并行化

- 方法描述
  - 发掘和利用现有串行算法中的并行性，直接将串行算法改造为并行算法。
- 评注
  - 由串行算法直接并行化的方法是并行算法设计的最常用方法之一；
  - 不是所有的串行算法都可以直接并行化的；
  - 一个好的串行算法并不能并行化为一个好的并行算法；
  - 许多数值串行算法可以并行化为有效的数值并行算法。

从问题描述开始设计并行算法

- 方法描述
  - 从问题本身描述出发，不考虑相应的串行算法，设计一个全新的并行算法。
- 评注
  - 挖掘问题的固有特性与并行的关系；
  - 设计全新的并行算法是一个挑战性和创造性的工作；
  - 利用串的周期性的PRAM-CRCW算法是一个很好的范例；

借用已有的算法求借新问题

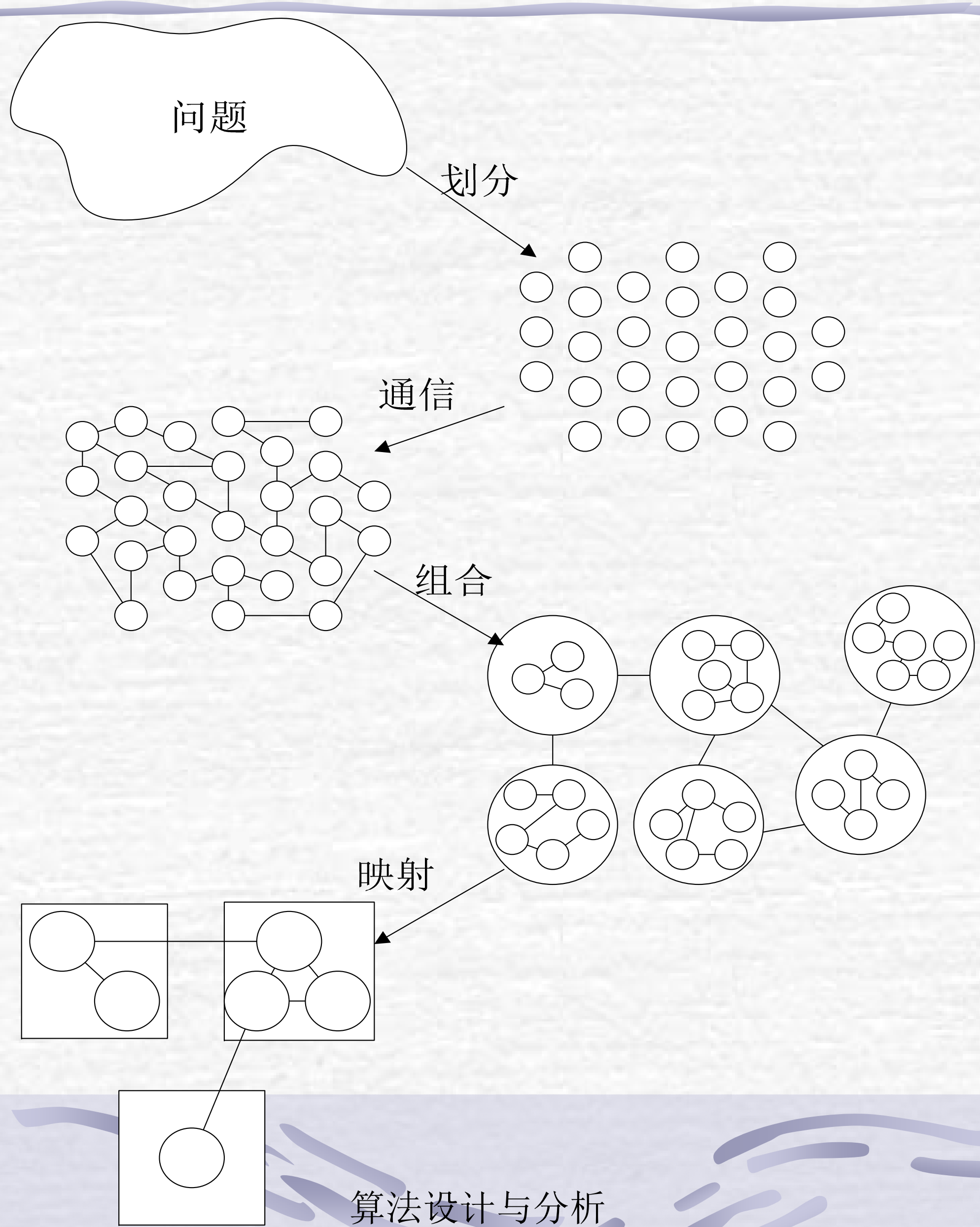
- 方法描述
  - 找出求解问题和某个已解决问题之间的联系；改造或利用已知算法应用到求解问题上。
- 评注
  - 这是一项创造性的工作；
  - 使用矩阵乘法算法求解所有点对间最短路径是一个很好的范例。

PCAM设计过程（1）

- PCAM设计方法学
  - 先找出问题的并发性，满足算法的可扩放性；
  - 其次优化算法的通讯成本和全局执行时间；
  - 最后，经过调整和测试，以达到满意的设计选择。
- PCAM分为四步：
  - 数据和任务划分(Partitioning)；
  - 通讯设计(Communication)；
  - 任务组合(Agglomeration)；
  - 处理器映射(Mapping)；

注：前两阶段考虑与机器无关的特性：并发性和可扩放性；后两阶段考虑与机器相关的特性：局部性等其他与性能相关的问题上。

PCAM设计过程（2）



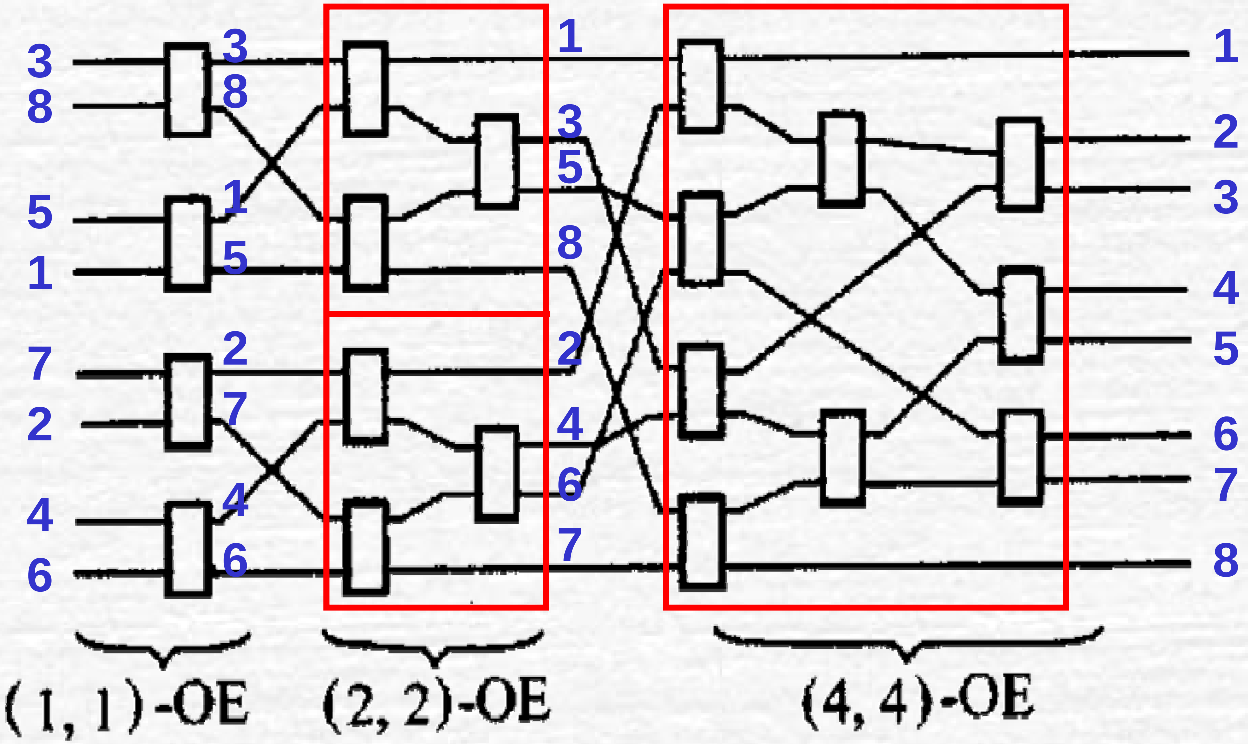
第5章(补充) 并行算法

5.5 算法设计示例

- Batcher排序网络
- 求最大值
- 求前缀和
- PSRS排序算法
- Systolic矩阵乘法

Batcher排序网络

- 基于奇偶归并网络
- 示例： B(8)



求最大值 (1)

- 算法： SIMD-EREW上求最大值算法： O(logn)

```
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
```

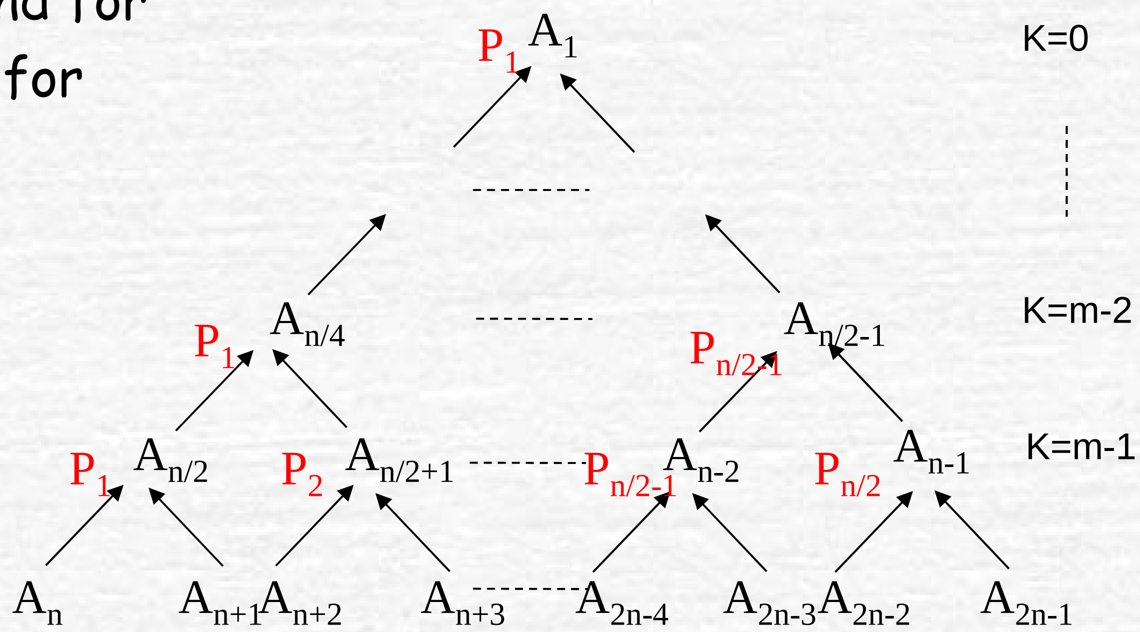
时间分析

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

$p(n)=n/2$

$c(n)=O(n\log n)$ 非成本最优

- 图示



求最大值（2）

- 算法： SIMD-CRCW上求最大值算法：O(1)

```
//输入A[1..p]，p个不同元素
//B[1..p][1..p],M[1..p]为中间处理用的布尔数组，如果M[i]=1，则A[i]为最大值
begin
    (1)for 1≤i, j≤p par-do   //工作量O(p²)；时间O(1),因为允许同时读
            if A[i]≥A[j] then B[i, j]=1 else B[i, j]=0
            end if
        end for
    (2)for 1≤i≤p par-do      //工作量O(p²)；时间O(1),因为允许同时写
            M[i]=B[i,1]∧ B[i,2]∧… ∧B[i,p]
        end for
end
```

- $T(n)=O(1)$
- $W(n)=O(p2)$
- 可以用p^2个处理器实现
- 速度虽快，但不是WT最优

求前缀和（1）

- 问题定义

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

  $S_i=x_1*x_2*\ldots*x_i$，$1\le i\le n$　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：

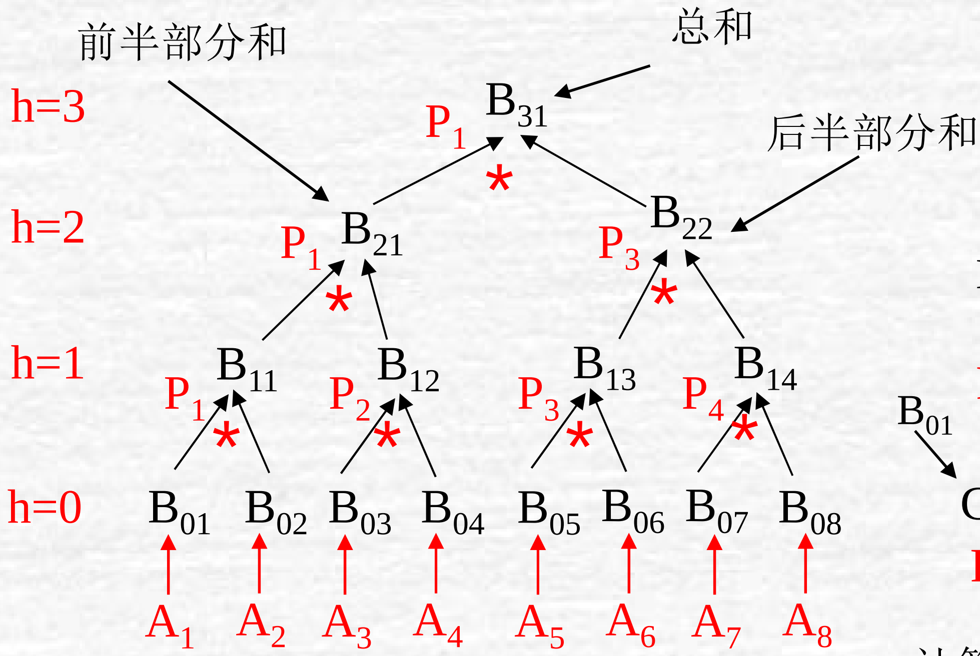
  令$A[i]=x_i$, $i=1\sim n$, $B[h,j]$和$C[h,j]$为辅助数组($h=0\sim\log n$, $j=1\sim n/2^h$)

  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)

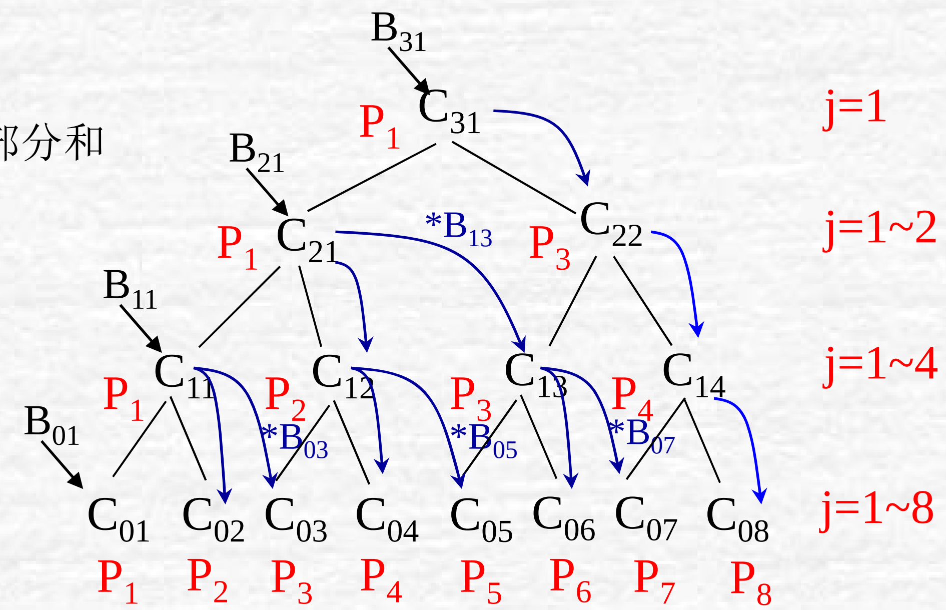
求前缀和（2）

- **例**：$n=8, p=8, C_{01} \sim C_{08}$为前缀和



计算：$B[h,j]=B[h-1,2j-1]*B[h-1,2j]$

正向遍历（求和）



计算：

$C[h,1]=B[h,1]$	$j=1$
$C[h,j]=C[h+1, j/2]$	$j=\text{even}$
$C[h,j]=C[h+1,(j-1)/2]*B[h, j]$	$j=\text{odd}>1$

反向遍历（播送前缀和）

求前缀和（3）

算法: SIMD-SM上非递归算法

```
begin
   (1)for j=1 to n par-do    //初始化
          B[0,j]=A[j]
      end if
   (2)for h=1 to logn do   //正向遍历
          for j=1 to n/2^h par-do
              B[h,j]=B[h-1,2j-1]*B[h-1,2j]
          end for
      end for
   (3)for h=logn to 0 do   //反向遍历
          for j=1 to n/2^h par-do
            (i)  if j=even then   //该结点为其父结点的右儿子
                     C[h,j]=C[h+1,j/2]
                 end if
            (ii) if j=1 then       //该结点为最左结点
                     C[h,1]=B[h,1]
                 end if
            (iii) if j=odd>1 then   //该结点为其父结点的左儿子
                     C[h,j]=C[h+1,(j-1)/2]*B[h,j]
                  end if
          end for
      end for
end
```

时间分析:

(1) O(1) (2) O(logn) (3) O(logn)

===> t(n)=O(logn)，p(n)=n，c(n)=O(nlogn)

PSRS排序算法（1）

- **划分方法**

  n个元素A[1..n]分成p组，每组A[(i-1)n/p+1..in/p]，i=1~p

- **算法：** MIMD-SM模型上的PSRS排序

begin

(1)均匀划分：将n个元素A[1..n]均匀划分成p段，每个p_i处理
A[(i-1)n/p+1..in/p]

(2)局部排序：p_i调用串行排序算法对A[(i-1)n/p+1..in/p]排序

(3)选取样本：p_i从其有序子序列A[(i-1)n/p+1..in/p]中选取p个样本元素

(4)样本排序：用一台处理器对p^2个样本元素进行串行排序

(5)选择主元：用一台处理器从排好序的样本序列中选取p-1个主元，并播送给其他p_i

(6)主元划分：p_i按主元将有序段A[(i-1)n/p+1..in/p]划分成p段

(7)全局交换：各处理器将其有序段按段号交换到对应的处理器中

(8)归并排序：各处理器对接收到的元素进行归并排序

end.

PSRS排序算法（2）

- 排序过程：N=27，p=3，PSRS排序如下：

(a) 均匀划分：	15	46	48	93	39	6	72	91	14	36	69	40	89	61	97	12	21	54	53	97	84	58	32	27	33	72	20
(b) 局部排序：	6	14	15	39	46	48	72	91	93	12	21	36	40	54	61	69	89	97	20	27	32	33	53	58	72	84	97

(c) 正则采样： 6 39 72 12 40 69 20 33 72

(d) 采样排序： 6 12 20 33 39 40 69 72 72

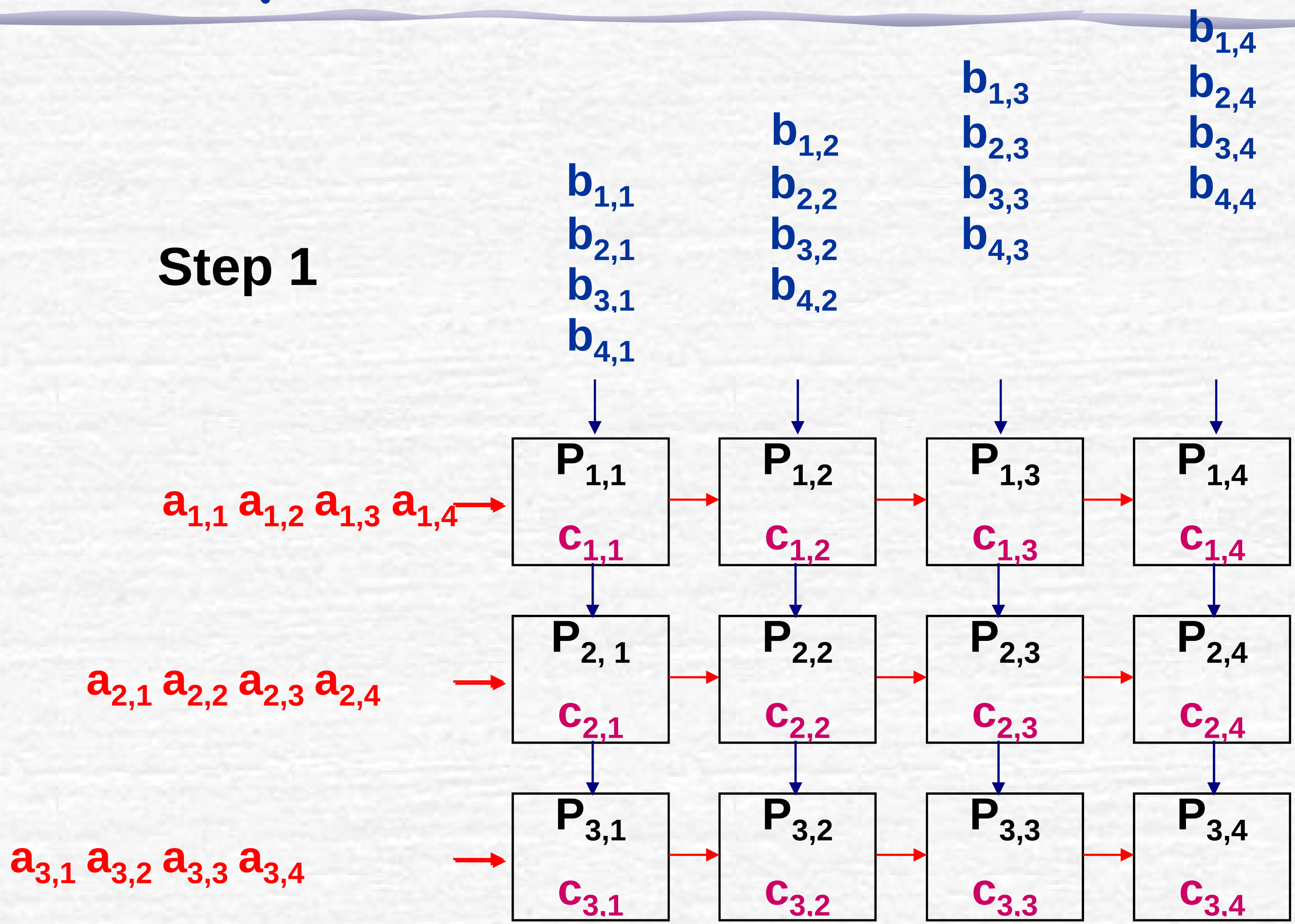
(e) 选择主元： 33 69

(f) 主元划分：	6	14	15	39	46	48	72	91	93	12	21	36	40	54	61	69	89	97	20	27	32	33	53	58	72	84	97
(g) 全局交换：	6	14	15	12	21	20	27	32	33	39	46	48	36	40	54	61	69	53	58	72	91	93	89	97	72	84	97
(h) 归并排序：	6	12	14	15	20	21	27	32	33	36	39	40	46	48	53	54	58	61	69	72	72	84	89	91	93	97	97

图6. 1

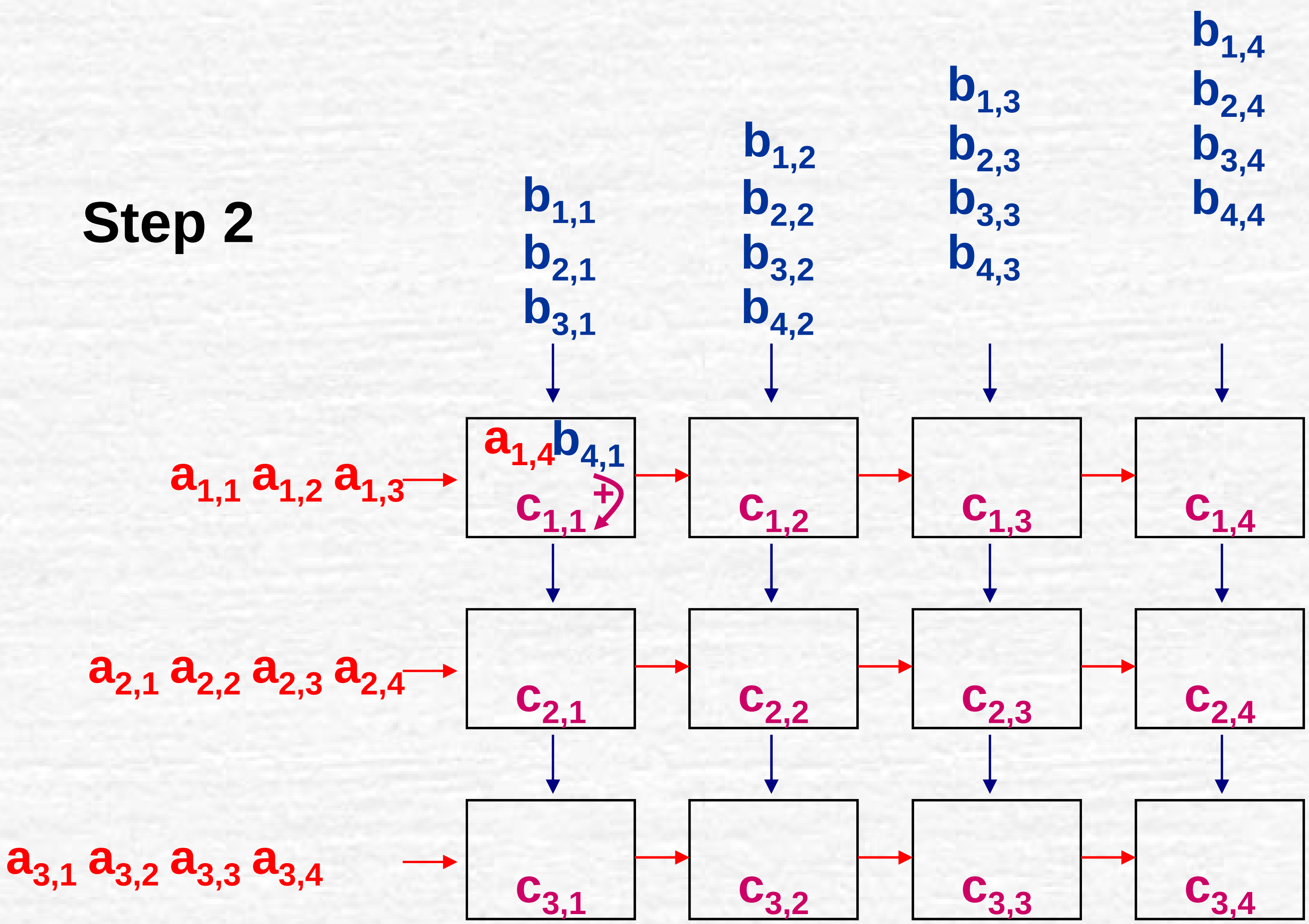
Systolic矩阵乘法（1）

Step 1



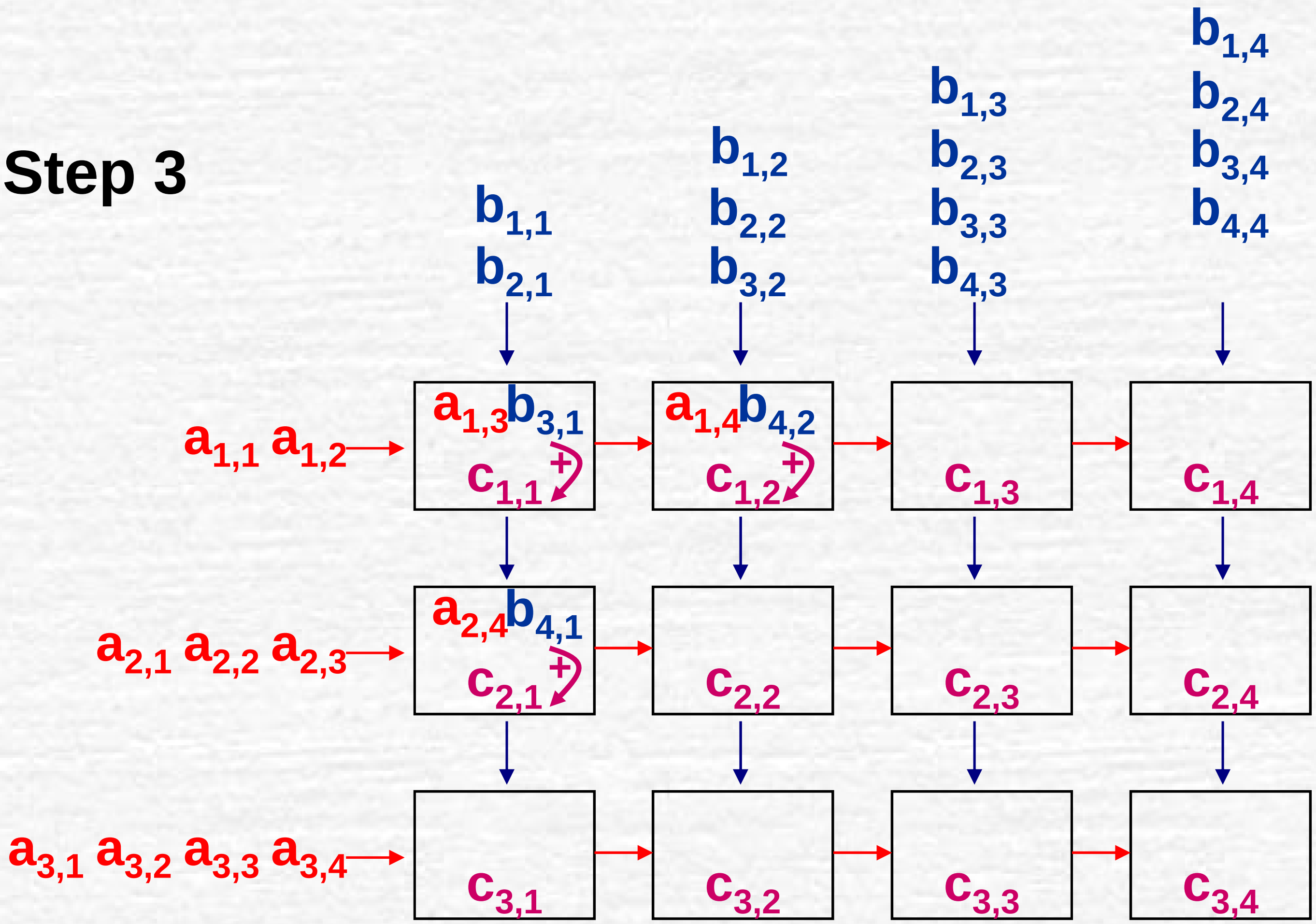
Systolic矩阵乘法（2）

Step 2



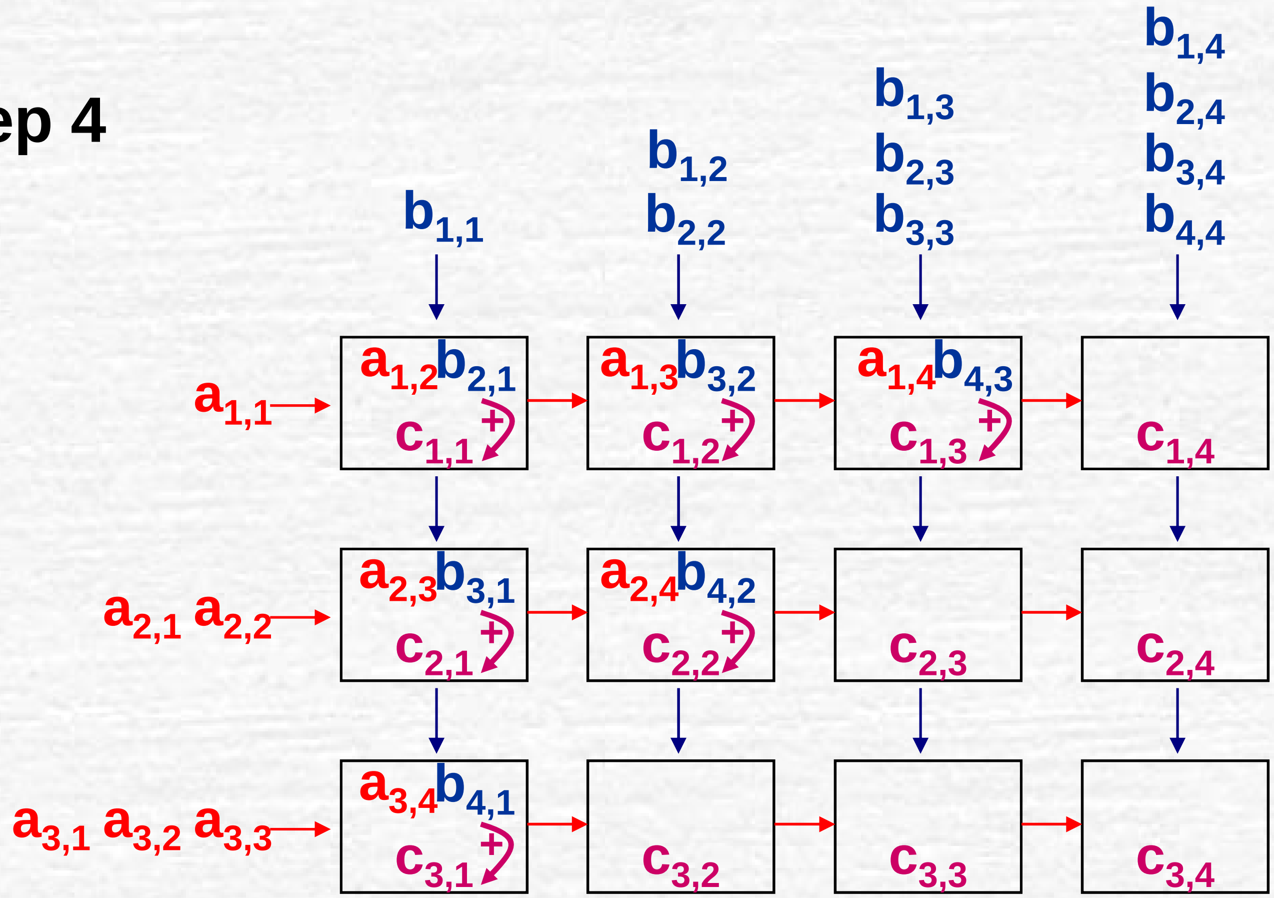
Systolic矩阵乘法（3）

Step 3



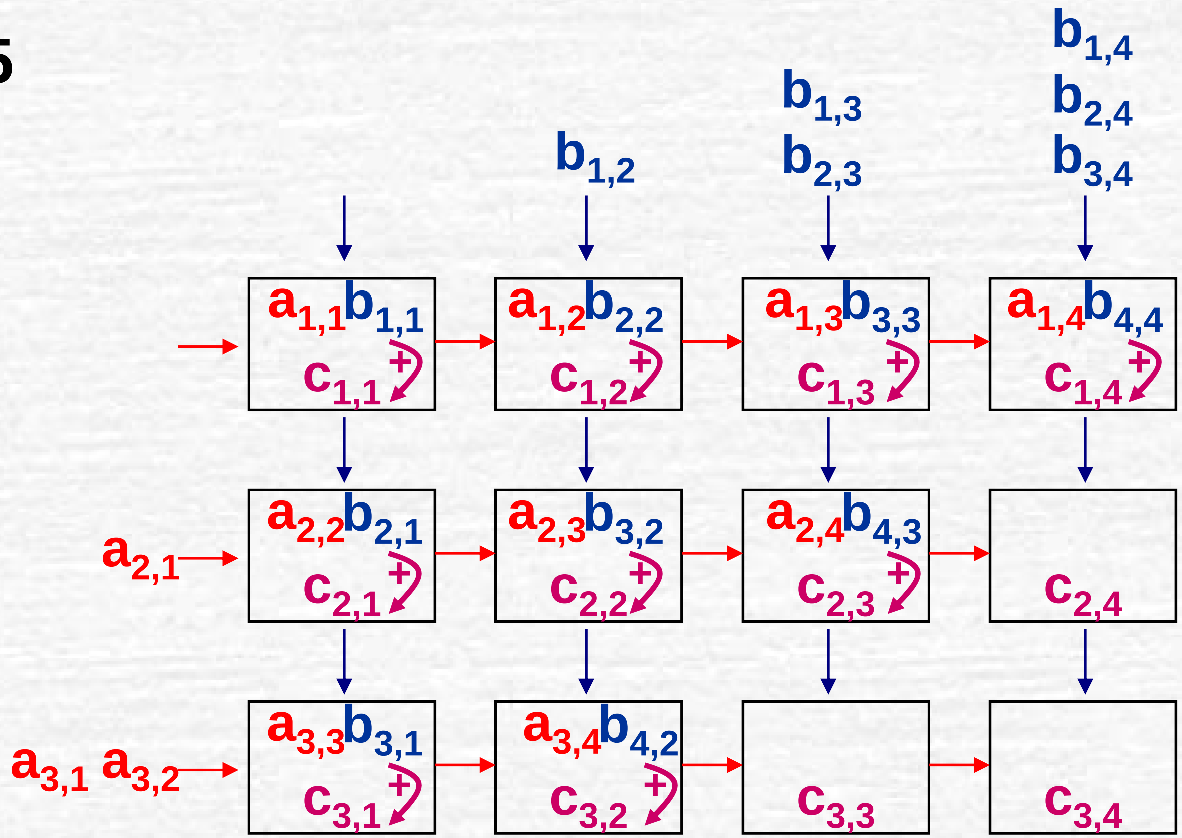
Systolic矩阵乘法（4）

Step 4



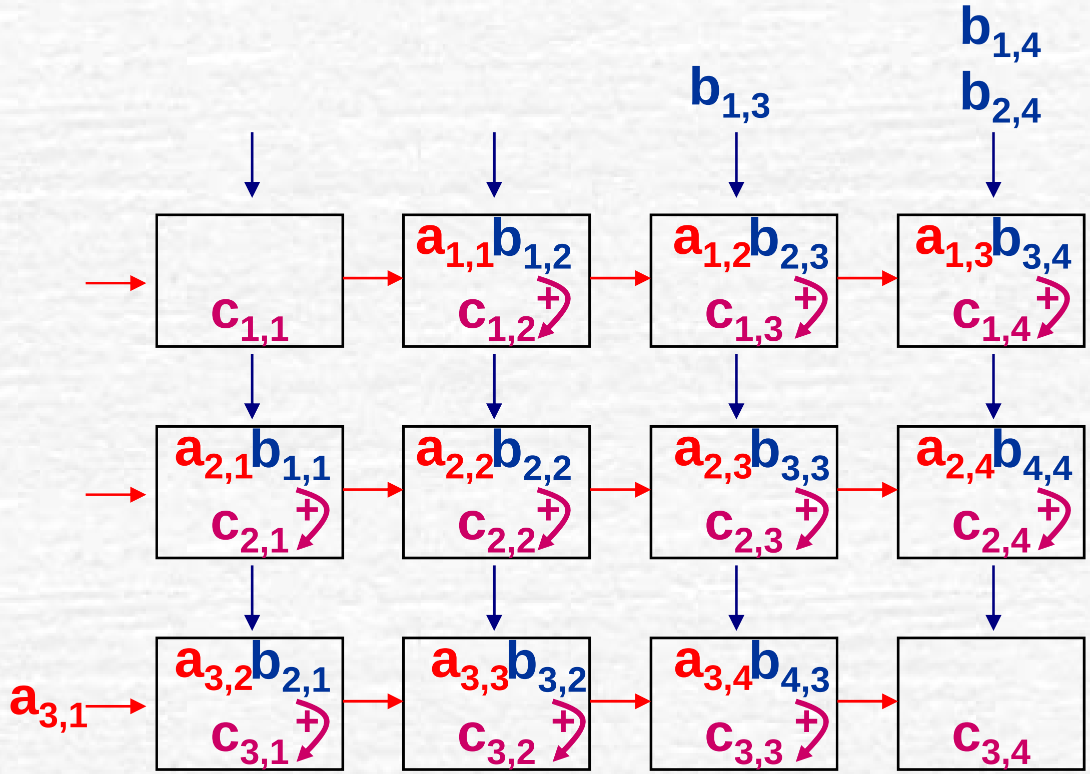
Systolic矩阵乘法（5）

Step 5



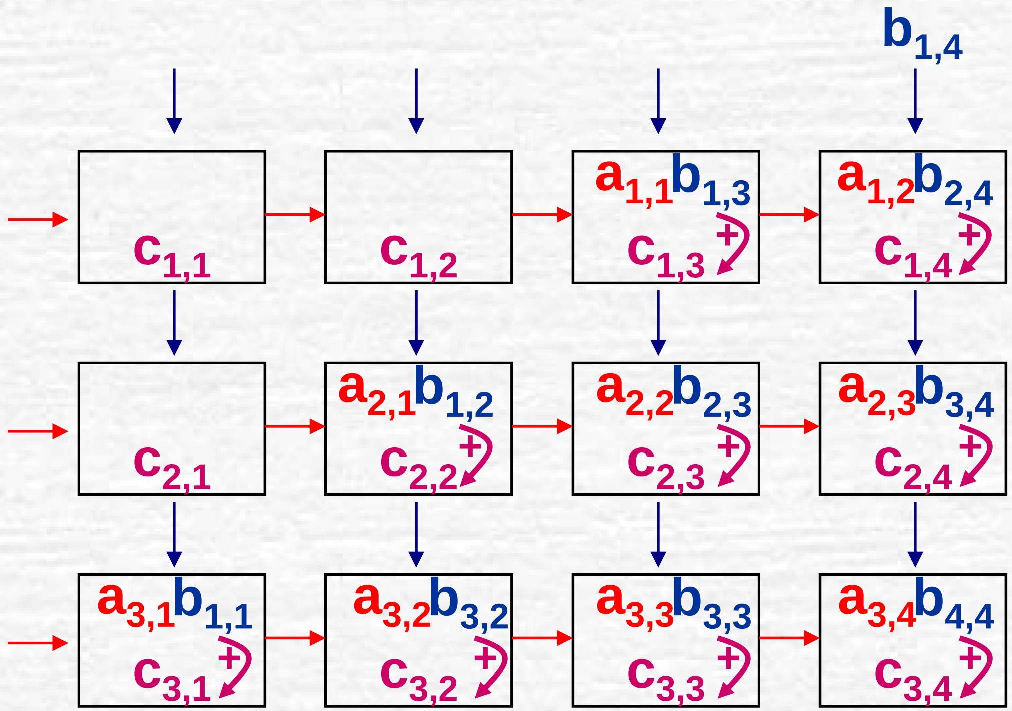
Systolic矩阵乘法（6）

Step 6



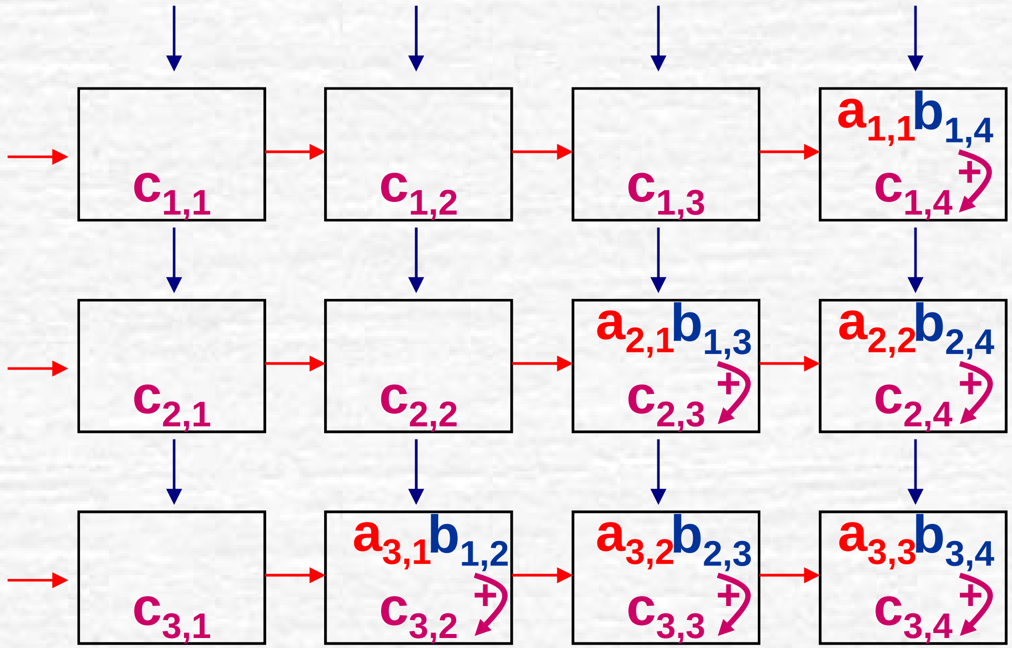
Systolic矩阵乘法（7）

Step 7



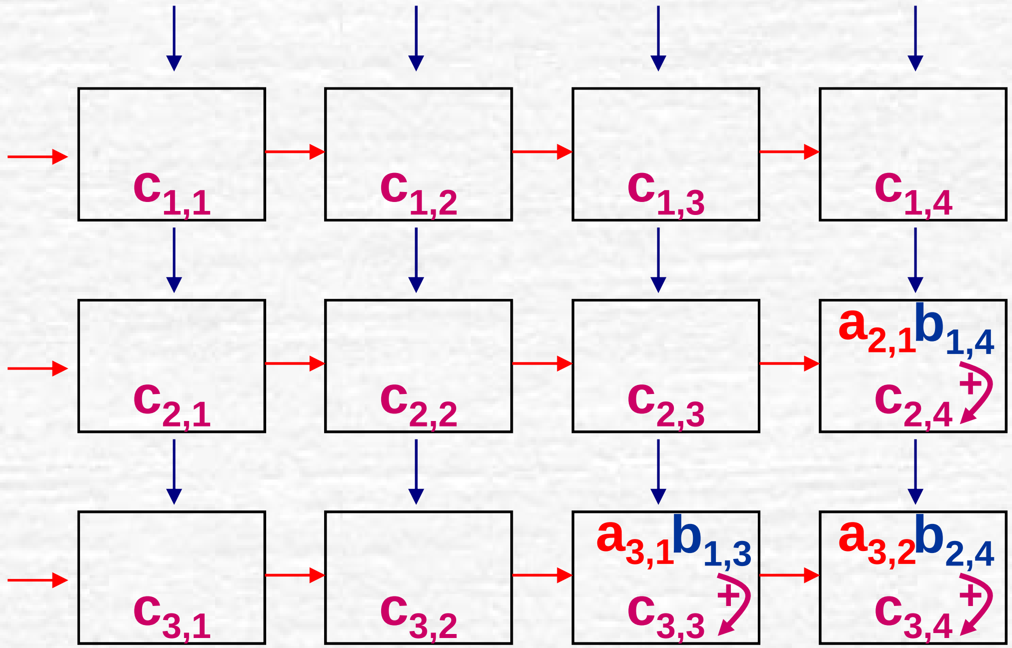
Systolic矩阵乘法（8）

Step 8



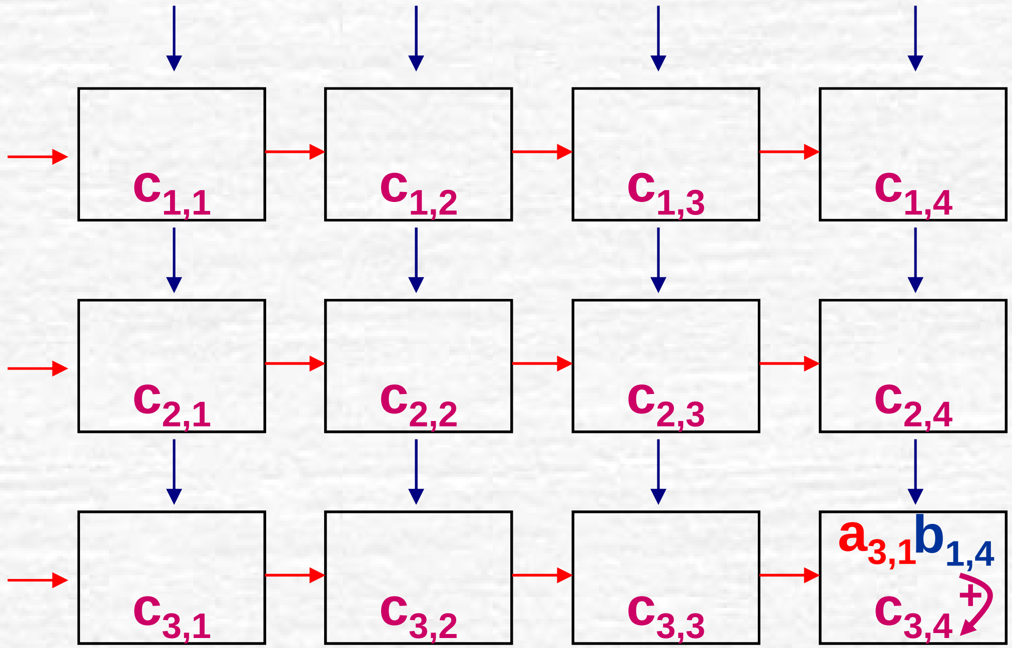
Systolic矩阵乘法（9）

Step 9



Systolic矩阵乘法（10）

Step 10



Systolic矩阵乘法 (11)

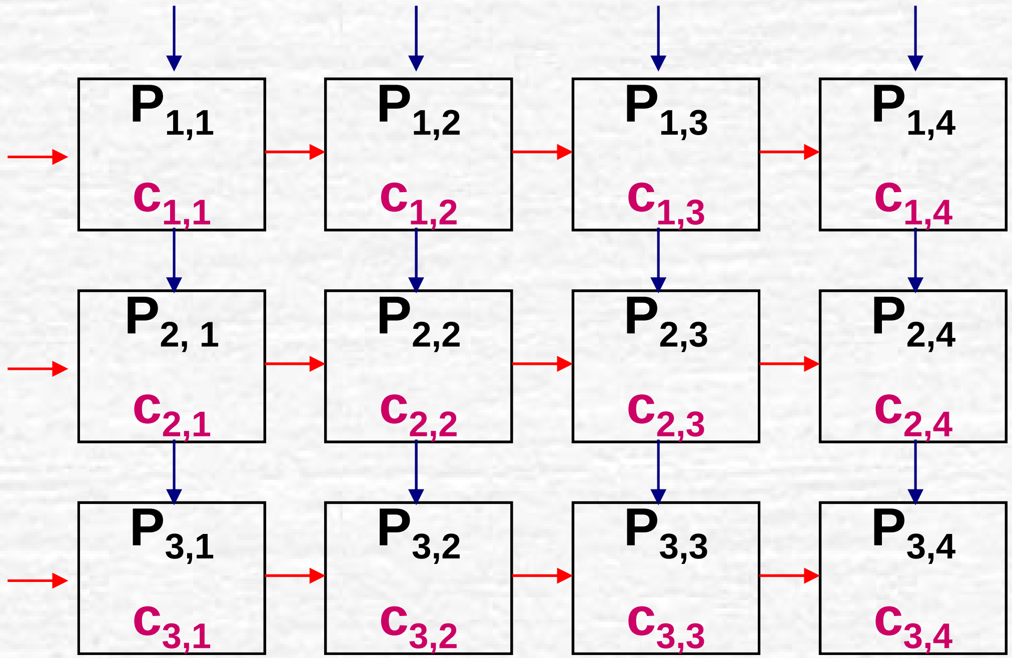
$$c_{1,1} = a_{1,1}\, b_{1,1} + a_{1,2}\, b_{2,1} + a_{1,3}\, b_{3,1} + a_{1,4}\, b_{4,1}$$

$$c_{1,2} = a_{1,1}\, b_{1,2} + a_{1,2}\, b_{2,2} + a_{1,3}\, b_{3,2} + a_{1,4}\, b_{4,2}$$

Over

…………

$$c_{3,4} = a_{3,1}\, b_{1,4} + a_{3,2}\, b_{2,4} + a_{3,3}\, b_{3,4} + a_{3,4}\, b_{4,4}$$



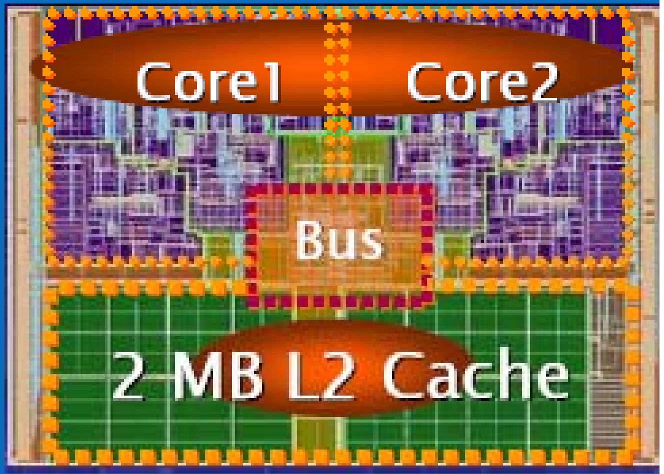
第5章(补充) 并行算法

5.6 Intel Multi-core Archi. & Prog.

- Intel Core Duo Processor Architecture
- Intel Compilers
- Vtune Performance Analyzer
- Intel Math Kernel Library (MKL)
- Multi-core Programming: Basic Concepts
- Programming with Windows Threads
- Programming with OpenMP
- Programming with POSIX Threads
- Development Cycle and An Example
- Reference Resources

Key Features

Intel® Smart Cache



- Shared between the two cores
- Advanced Transfer Cache architecture
- Reduced bus traffic
- Both cores have full access to the entire cache
- Dynamic Cache sizing

Enables Greater System Responsiveness

Summary

- Intel® Core™ Duo processor is at the core of Intel desktop and Mobile Platforms (2GHz, 5333MB/s bandwidth, 31W)
- Dual core enables true Multi-threaded execution
- Intel® Smart Cache enables greater system responsiveness
- Intel® SpeedStep® technology provides very low system down times
- Intel® Dynamic Power Management improves performance and battery life
- Intel® Digital Media Boost delivers a rich multimedia experience

Intel® Core™ Duo: More Performance for Less Power

5.6 Intel Multi-core Archi. & Prog.

- Intel Core Duo Processor Architecture
- Intel Compilers
- Vtune Performance Analyzer
- Intel Math Kernel Library (MKL)
- Multi-core Programming: Basic Concepts
- Programming with Windows Threads
- Programming with OpenMP
- Programming with POSIX Threads
- Development Cycle and An Example
- Reference Resources

General Optimizations

Windows*	Linux*	Mac*	
/Od	-O0	-O0	Disables optimizations
/Zi	-g	-g	Creates symbols
/O1	-O1	-O1	Optimize for Binary Size: Server Code
/O2	-O2	-O2	Optimizes for speed (default)
/O3	-O3	-O3	Optimize for Data Cache: Loopy Floating Point Code

OpenMP* Threading Technology

Pragma based approach to parallelism

Usage:

OpenMP switches: **-openmp** : **/Qopenmp**

OpenMP reports: -**openmp-report** : **/Qopenmp-report**

```
#pragma omp parallel for
 for (i=0;i<MAX;i++)
    A[i]= c*A[i] + B[i];
```

Parallel Diagnostics

Source Instrumentation for Intel Thread Checker

- Allows thread checker to diagnose threading correctness bugs
- To use tcheck/Qtcheck you must have Intel Thread Checker installed
- See thread checker documentation
- http://www.intel.com/support/performancetools/sb/CS-009681.htm

Windows*	Linux*	Mac*
/Qtcheck	-tcheck	No support

Compiler Based Vectorization

Processor Specific

Description	Use	Windows*	Linux*	Mac*
Generate instructions and optimize for Intel® Pentium® 4 compatible processors including MMX, SSE and SSE2.	W	/QxW	-xW	Does not apply
Generate instructions and optimize for Intel® processors with SSE3 capability including Core Duo. These processors support SSE3 as well as MMX,SSE and SSE2.	P	/QxP /QaxP	-xP, -axP	Vector-ization occurs by default

5.6 Intel Multi-core Archi. & Prog.

- Intel Core Duo Processor Architecture
- Intel Compilers
- Vtune Performance Analyzer
- Intel Math Kernel Library (MKL)
- Multi-core Programming: Basic Concepts
- Programming with Windows Threads
- Programming with OpenMP
- Programming with POSIX Threads
- Development Cycle and An Example
- Reference Resources

Agenda

What is the VTune™ Performance Analyzer?

Performance tuning concepts

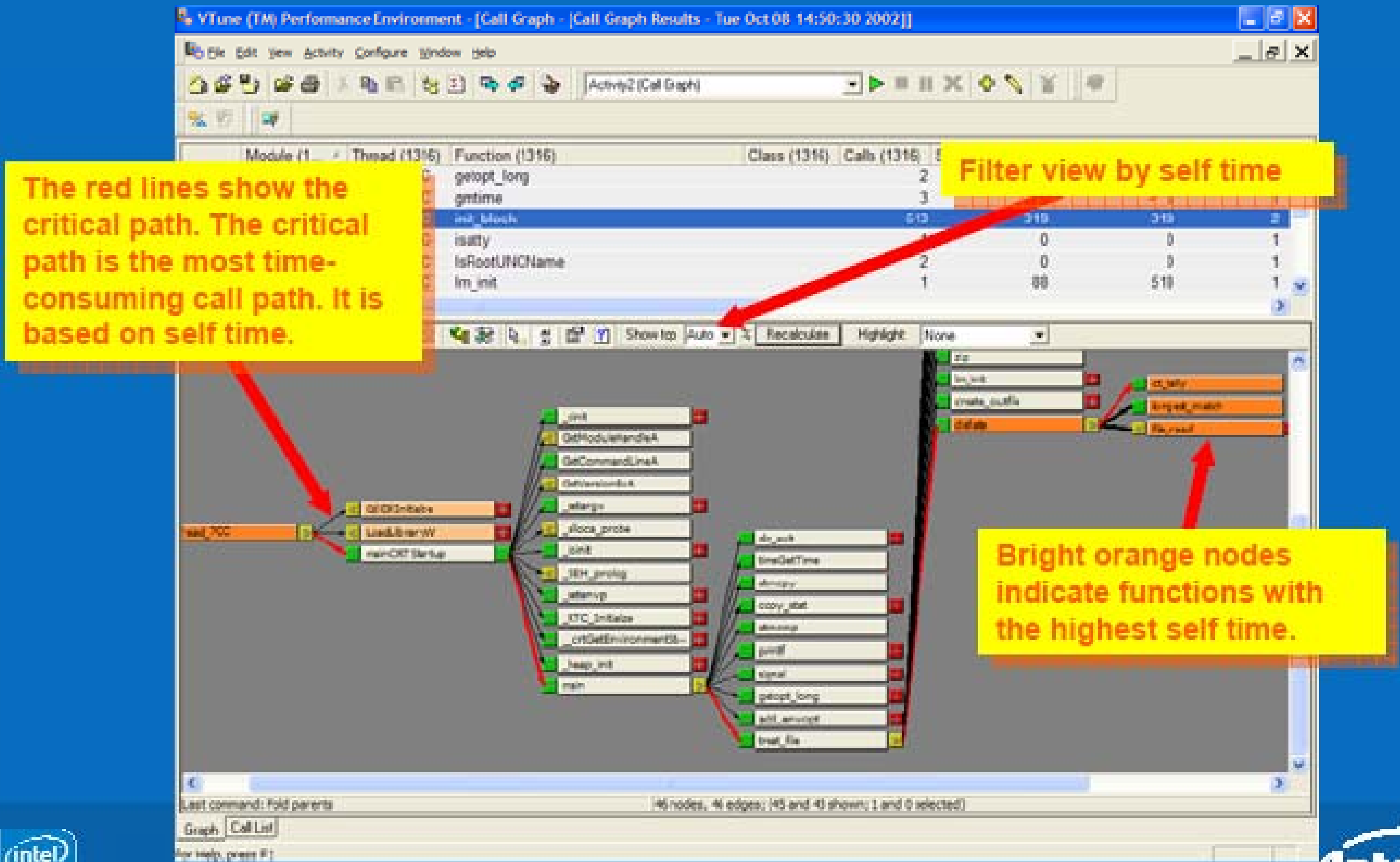
Using the sampling collector

How sampling works

Sampling Over Time

Call Graph

Call Graph View



5.6 Intel Multi-core Archi. & Prog.

- Intel Core Duo Processor Architecture
- Intel Compilers
- Vtune Performance Analyzer
- <u>Intel Math Kernel Library (MKL)</u>
- Multi-core Programming: Basic Concepts
- Programming with Windows Threads
- Programming with OpenMP
- Programming with POSIX Threads
- Development Cycle and An Example
- Reference Resources

Intel® Math Kernel Library Purpose

Performance, Performance, Performance!

Intel's engineering, scientific, and financial math library

Addresses:

- Solvers (BLAS, LAPACK)
- Eigenvector/eigenvalue solvers (BLAS, LAPACK)
- Some quantum chemistry needs (dgemm)
- PDEs, signal processing, seismic, solid-state physics (FFTs)
- General scientific, financial [vector transcendental functions (VML) and vector random number generators (VSL)]

Tune for Intel® processors – current and future

Matrix Multiplication DGEMV/DGEMM

dgemv

```
for( i = 0; i < n; i++ )
    cblas_dgemv( CBLAS_RowMajor, CBLAS_NoTrans, m, n,
                 alpha, a, lda, &b[0][i], ldb, beta, &c[0][i], ldc );
```

dgemm

```
Cblas_dgemm( CblasColMajor, CblasNoTrans, CblasNoTrans, m,
n, kk, alpha, b, ldb, a, lda, beta, c, ldc );
```

5.6 Intel Multi-core Archi. & Prog.

- Intel Core Duo Processor Architecture
- Intel Compilers
- Vtune Performance Analyzer
- Intel Math Kernel Library (MKL)
- Multi-core Programming: Basic Concepts
- Programming with Windows Threads
- Programming with OpenMP
- Programming with POSIX Threads
- Development Cycle and An Example
- Reference Resources

Basic concepts:

- Processes and thresds
- Parallelism and concurrency

Design concepts:

- Threading for functionality or performance?
- Threading for throughput or turnaround?
- Decomposing the work

Correctness concepts:

- Race conditions and Synchronization
- Deadlock

Performance concepts:

- Speedup and Efficiency
- Granularity and load balance

5.6 Intel Multi-core Archi. & Prog.

- Intel Core Duo Processor Architecture
- Intel Compilers
- Vtune Performance Analyzer
- Intel Math Kernel Library (MKL)
- Multi-core Programming: Basic Concepts
- Programming with Windows Threads
- Programming with OpenMP
- Programming with POSIX Threads
- Development Cycle and An Example
- Reference Resources

Basic concepts and problems:

Explore Win32 Threading API functions

- Create threads
- Wait for threads to terminate
- Synchronize shared access between threads

Example: Critical Section

```
#define NUMTHREADS 4
CRITICAL_SECTION g_cs; // why does this have to be global?
int g_sum = 0;

DWORD WINAPI threadFunc(LPVOID arg )
{
  int mySum = bigComputation();
  EnterCriticalSection(&g_cs);
   g_sum += mySum;              // threads access one at a time
  LeaveCriticalSection(&g_cs);
  return 0;
}
main() {
  HANDLE hThread[NUMTHREADS];
  InitializeCriticalSection(&g_cs);
  for (int i = 0; i < NUMTHREADS; i++)
    hThread[i] =
        CreateThread(NULL,0,threadFunc,NULL,0,NULL);
  WaitForMultipleObjects(NUMTHREADS, hThread, TRUE, INFINITE);
  DeleteCriticalSection(&g_cs);
}
```

5.6 Intel Multi-core Archi. & Prog.

- Intel Core Duo Processor Architecture
- Intel Compilers
- Vtune Performance Analyzer
- Intel Math Kernel Library (MKL)
- Multi-core Programming: Basic Concepts
- Programming with Windows Threads
- Programming with OpenMP
- Programming with POSIX Threads
- Development Cycle and An Example
- Reference Resources

What Is OpenMP*?

C$OMP FLUSH

#pragma omp critical

C$OMP THREADPRIVATE(/ABC/)

CALL OMP_SET_NUM_THREADS(10)

C$OMP parallel do shared(a, b, c)

call omp_test_lock(jlok)

call OMP_INIT

C$OMP MASTER

C$OMP SINGLE PRIV

C

ynamic"

C$OMP PARALLEL D

C$OMP ORDERED

C$OMP PARALLEL

IONS

http://www.openmp.org

Current spec is OpenMP 2.5

250 Pages

(combined C/C++ and Fortran)

#pragma omp parallel for private(A, B)

!$OMP BARRIER

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

omp_set_lock(lck)

OpenMP* Critical Construct

```
#pragma omp critical [(lock_name)]
```

Defines a critical region on a structured block

Threads wait their turn – at a time, only one calls `consum()` thereby protecting R1 and R2 from race conditions.

Naming the critical constructs is optional, but may increase performance.

```
float R1, R2;
#pragma omp parallel
{ float A, B;
#pragma omp for
  for(int i=0; i<niters; i++){
    B = big_job(i);
#pragma omp critical (R1_lock)
    consum (B, &R1);
    A = bigger_job(i);
#pragma omp critical (R2_lock)
    consum (A, &R2);
  }
}
```

5.6 Intel Multi-core Archi. & Prog.

- Intel Core Duo Processor Architecture
- Intel Compilers
- Vtune Performance Analyzer
- Intel Math Kernel Library (MKL)
- Multi-core Programming: Basic Concepts
- Programming with Windows Threads
- Programming with OpenMP
- Programming with POSIX Threads
- Development Cycle and An Example
- Reference Resources

What is Pthreads?

POSIX.1c standard

C language interface

Threads exist within same process

All threads are peers

- No explicit parent-child model
- Exception: "main thread" holds process information

Example: Multiple Threads

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) {
      printf("Hello Thread\n");
}

main() {
  pthread_t tid[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], NULL, hello, NULL);

  for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
```

5.6 Intel Multi-core Archi. & Prog.

- Intel Core Duo Processor Architecture
- Intel Compilers
- Vtune Performance Analyzer
- Intel Math Kernel Library (MKL)
- Multi-core Programming: Basic Concepts
- Programming with Windows Threads
- Programming with OpenMP
- Programming with POSIX Threads
- Development Cycle and An Example
- Reference Resources

Threads – Benefits & Risks

Benefits

- Increased performance and better resource utilization
  - Even on single processor systems - for hiding latency and increasing throughput
- IPC through shared memory is more efficient

Risks

- Increases complexity of the application
- Difficult to debug (data races, deadlocks, etc.)

Development Cycle

Analysis

–VTune™ Performance Analyzer

Design (Introduce Threads)

–Intel® Performance libraries: IPP and MKL

–OpenMP* (Intel® Compiler)

–Explicit threading (Win32*, Pthreads*)

Debug for correctness

–Intel® Thread Checker

–Intel Debugger

Tune for performance

–Intel® Thread Profiler

–VTune™ Performance Analyzer

Software

Prime Number Generation

i	factor
3	2
5	2
7	2 3
9	2 3
11	2 3
13	2 3 4
15	2 3
17	2 3 4
19	2 3 4

```
bool TestForPrime(int val)
{   // let's start checking from 3
    int limit, factor = 3;
    limit = (long)(sqrtf((float)val)+0.5f);
    while( (factor <= limit) && (val % factor) )
            factor ++;
```

C:\WINDOWS\system32\cmd.exe

```
C:\classfiles\PrimeSingle\Release>PrimeSingle.exe 1 20
100%

   8 primes found between      1 and     20 in    0.00 secs

C:\classfiles\PrimeSingle\Release>_
```

```
    int range = end - start + 1;
    for( int i = start; i <= end; i += 2 )
    {
        if( TestForPrime(i) )
            globalPrimes[gPrimesFound++] = i;
        ShowProgress(i, range);
    }
}
```

5.6 Intel Multi-core Archi. & Prog.

- Intel Core Duo Processor Architecture
- Intel Compilers
- Vtune Performance Analyzer
- Intel Math Kernel Library (MKL)
- Multi-core Programming: Basic Concepts
- Programming with Windows Threads
- Programming with OpenMP
- Programming with POSIX Threads
- Development Cycle and An Example
- Reference Resources

Reference

Web-based and classroom training

- www.intel.com/software/college

White papers and technical notes

- www.intel.com/ids
- www.intel.com/software/products

Product support resources

- www.intel.com/software/products/support

End of SCh5