第二篇 并行算法的设计

第四章 并行算法的设计基础

第五章 并行算法的一般设计方法

第六章 并行算法的基本设计技术

第七章 并行算法的一般设计过程

分布式算法

- 所谓分布式计算就是在两个或多个软件互相共享信息，这些软件既可以在同一台计算机上运行，也可以在通过网络连接起来的多台计算机上运行。
- 研究如何把一个需要非常巨大的计算能力才能解决的问题分成许多小的部分，然后把这些部分分配给许多计算机进行处理，最后把这些计算结果综合起来得到最终的结果。
- 1.解决较为复杂的数学问题，例如：GIMPS（寻找最大的梅森素数）。
  2.研究寻找最为安全的密码系统，例如：RC-72（密码破解）。
  3.生物病理研究，例如：Folding@home（研究蛋白质折叠，误解，聚合及由此引起的相关疾病）。
  4.各种各样疾病的药物研究，例如：United Devices（寻找对抗癌症的有效的药物）。
  5.信号处理，例如：SETI@Home（在家寻找地外文明）。

第四章 并行算法的设计基础

4.1 并行算法的基础知识

4.2 并行计算模型

随机化算法

- 随机化算法是这样一种算法，在算法中使用了随机函数，且随机函数的返回值直接或间接地影响了算法的执行流程或执行结果。
- 随机化算法的基本原理是：当某个决策中有多个选择，但又无法确定哪个是好的选择，或确定好的选择需要付出较大的代价时，如果大多数选择是好的，那么随机地选一个往往是一种有效的策略。

并行算法的定义和分类

- 并行算法的定义
  - 算法:是指完成一个任务所需要的具体步骤和方法
  - *systematic procedure that produces—in a finite number of steps—the answer to a question or the solution of a problem.*
  - In computer science, a parallel algorithm, as opposed to a traditional serial algorithm, is one which can be executed a piece at a time on many different processing devices, and then put back together again at the end to get the correct result.
- 并行算法的分类
  - 数值计算和非数值计算
  - 同步算法(SIMD算法)和异步算法(MIMD算法)
  - 分布式算法
  - 确定算法和随机算法

- PARTITION_R(A,lo,hi)
  - r←RANDOM(hi-lo+1)+lo
  - 交换A[lo]和A[r]
  - return PARTITION(A,lo,hi)
- QUICKSORT_R(A,lo,hi)
  - if lo < hi
  - p←PARTITION_R(A,lo,hi)
  - QUICKSORT_R(A,lo,p)
  - QUICKSORT_R(A,p+1,hi)

4.1.2 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
```

算法复杂性度量

- 基本操作
- 问题规模
- 时间复杂度
- 空间复杂度

算法A ⇨ 基本操作执行次数 ⇨ f(n)

⇨ 算法复杂度为O(g(n))

有效算法与NP问题：

$O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(n^3)$

$O(2^n)<O(n!)<O(n^n)$

算法复杂性的渐近表示

$N=\{0,1,2,...\}$ 非负整数集

$N^+=\{1,2,3,...\}$ 自然数集

R 实数集

R^+ 正实数集

$R^*=R^+\cup\{0\}$

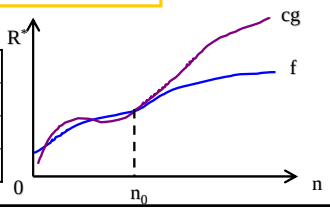
$g, f: N^+ \to R^*$

f和g之间有如下关系：

$(\exists c\in R^+)(\exists n_0\in N^+)(\forall n\ge n_0)(f(n)\le cg(n))$

称g是f的一个上界，记为$f=O(g)$

f	g	c	n_0
$2n^2$	$3n^2$	>=2/3	1
n^2+2n	n^2	>=3	>=1




```
例：Bubble排序算法。
   In: n个被排序的数在数组A中。
   Out: 排序的结果为数组B
begin
   for i=1 to n do B(i)<-A(i) endfor
   t=1
   while(t==1)do
      t=0
       for h=1 to n-1 do
         if B(h)>B(h+1) then
            swap(B(h),B(h+1))
            t=1
         endif
      endfor
   endwhile
end
```

$g, f: N^+ \to R^*$

f和g之间有如下关系：

$(\exists c\in R^+)(\exists n_0\in N^+)(\forall n\ge n_0)(f(n)\ge cg(n))$

称g是f的一个下界，记为$f=\Omega(g)$

$g, f: N^+ \to R^*$

f和g之间有如下关系：

$(\exists c_1, c_2\in R^+)(\exists n_0\in N^+)(\forall n\ge n_0)(c_1 g(n)\le f(n)\le c_2 g(n))$

称g是f的一个紧致界，记为$f=\Theta(g)$

算法名称	A_1	A_2	A_3	A_4	A_5
时间复杂度	n	nlogn	n^2	n^3	2^n
问题规模(延长时间)	1s	1000	140	31	10	9
	1m	$6*10^4$	4893	244	39	15
	1h	$3.6*10^6$	$2*10^5$	1987	153	21
问题规模(提高速度)	单位时间	S_1	S_2	S_3	S_4	S_5
	10倍速	$10S_1$	近$10S_2$	$3.16S_3$	$2.15S_4$	S_5+3.3
	1万倍速	1万S_1	$9000S_2$	$100S_3$	$21.54S_4$	S_5+13.3

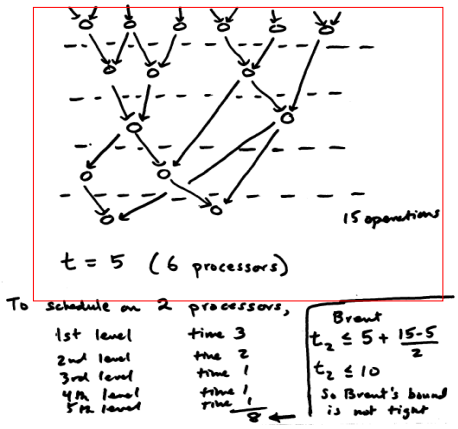
复杂度的选取

- **最坏情况下的复杂度(Worst-CASE Complexity)**

给定规模为n的问题，各种输入会导致不同的复杂度（如排序）。

- **期望复杂度(Expected Complexity)**

对输入的性态进行平均，平均性态下的输入的复杂度。



Brent定理

算法{抽象描述： 与处理器数和任务分配无关
具体描述（实现）： 使用p台处理器

令W(n)是某并行算法A在运行时间T(n)内所执行的运算量，则A使用*p*台处理器可在t(n)=O(W(n)/p+T(n))时间内执行完毕。

例：SIMD-SM上的求和算法。

In: $n=2^k$，被求和的数在数组A中。

Out: S

```
begin
   (1) for i=1 to n par-do B(i)<-A(i) endfor
   (2) for h=1 to logn do
          for i=1 to n/(2^h) par-do
          B(i)<-B(2i-1)+B(2i)
          endfor
       endfor
   (3) S<-B(1)
end
```

$$W(n) = n + \sum_{j=1}^{\log n} \frac{n}{2^{j-1}} + 1 = O(n) \qquad T(n) = O(\log n)$$

证明：将T(n)分成若干个时间步(并行步)，对应第i个时间步的计算量是$W_i(n)$，在p个处理机时可用不超过$\left\lceil \frac{W_i(n)}{p} \right\rceil$并行步模拟，那么该算法在p个处理机上的执行时间

$$\sum_{i=1}^{T(n)} \left\lceil \frac{W_i(n)}{p} \right\rceil \le \sum_{i=1}^{T(n)} \left(\left\lfloor \frac{W_i(n)}{p} \right\rfloor + 1 \right) \le \frac{1}{p} \sum_{i=1}^{T(n)} W_i(n) + T(n) \le \frac{1}{p} W(n) + T(n)$$

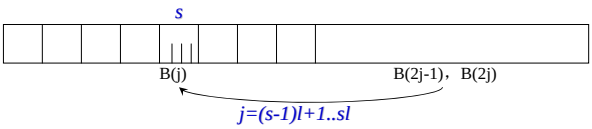
抽象算法看作T(n)个时间步（也就是并行步，这个总数是正整数，可数），令W_i＝抽象算法第i个并行步内的操作数目（运算量）,对应的具体算法中所用时间 $\left\lceil \frac{W_i}{p} \right\rceil \le \frac{W_i + p - 1}{p}$

总的时间:

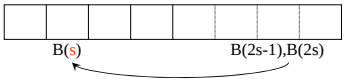
$$\le \sum_{i=1}^{t} \left\lceil \frac{W_i}{p} \right\rceil \le \sum_{i=1}^{t} \frac{W_i + p - 1}{p} = \sum_{i=1}^{t} \frac{p}{p} + \sum_{i=1}^{t} \frac{W_i - 1}{p} = t + \frac{\sum W_i - \sum 1}{p} = T(n) + \frac{W(n) - T(n)}{p}$$

注意t是并行步的编号，这些编号的个数之和等于T(n)

- **具体算法：$p=2^q$, $n=2^k$, 段长l,段号s，对应处理机Ps，所有下标都从1开始。**
- **计算过程中的段长变化分别为：$2^{k-q-1}, 2^{k-q-2}, \ldots, 2^0$. 即$2^{k-q-h}$, 其中h=1,2,…,k-q**



- **当段长为1时，段数减半求和，直到段数为1为止。**



```
(1) for j=1 to l=n/p do
      B(l(s-1)+j):=A(l(s-1)+j)
       endfor
(2) for h=1 to log n do
   (2.1) if (k-q-h>=0) then
         for j=2^(k-h-q)(s-1)+1 to 2^(k-h-q)s do
         B(j):=B(2j-1)+B(2j)
         endfor
      endif
   (2.2) if(s<=2^(k-h)) then
         B(s):=B(2s-1)+B(2s)
      endif
    endfor
(3) if(s=1)then S:=B(1) endif
end
```

并行算法的通信

- 通信
  - 共享存储多处理器：global read(X,Y)和global write(X,Y)
  - 分布存储多计算机：send(X,i)和receive(Y,j)
- 通信语句示例
  - 算法4.2 分布存储多计算机上矩阵向量乘算法

输入：处理器数p, A划分为B=A[1..n,(i-1)r+1..ir],
x划分为w=w[(i-1)r+1;ir]
输出：P_1保存乘积AX

Begin
(1) Compute z=Bw
(2) if i=1 then y_i=0 else receive(y,left) endif
(3) y=y+z
(4) send(y,right)
(5) if i=1 then receive(y,left)
End

(1) O(l)=O(n/p)
(2) 第h次迭代时所用时间O(n/(2^hp))
找最长执行时间的p，每次迭代中：段长大于1时，计算量为段长数；段长等于1且段数大于1时，计算量为1。所以总时间：

$$\sum_{j=1}^{\log n}\left\lceil \frac{n}{2^{j-1}p} \right\rceil$$

(3) O(1)

$$\sum_{j=1}^{\log n}\left\lceil \frac{n}{2^{j-1}p} \right\rceil \le \log n + \left\lfloor \sum_{j=1}^{\log n} \frac{n}{2^{j-1}p} \right\rfloor = \log n + \left\lfloor \frac{n}{p} 2\cdot(1-\frac{1}{n}) \right\rfloor$$

$$t(n) = O(\frac{n}{p} + \log n)$$

$W(n)=n$, $p=p$, $T(n)=\log n$,所以满足Brent定理。

第四章 并行算法的设计基础

4.1 并行算法的基础知识
4.2 并行计算模型

4.1.4 并行算法的同步

- 同步概念
  - 同步是在时间上强使各执行进程在某一点必须互相等待;
  - 可用软件、硬件和固件的办法来实现。
- 同步语句示例
  - 算法4.1 共享存储多处理器上求和算法

输入：A=(a_0,…,a_{n-1}),处理器数p
输出：S=Σ a_i

Begin
(1)S=0
(2)for all Pi where 0≤i≤p-1 do
(2.1) L=0
(2.2) for j=i to n step p do
L=L+a_j
end for
end for
(2.3) lock(S)
S=S+L
(2.4) unlock(S)
end for
End

4.2 并行计算模型

4.2.1 PRAM模型
4.2.2 异步APRAM模型
4.2.3 BSP模型
4.2.4 logP模型

4.2.1 PRAM模型

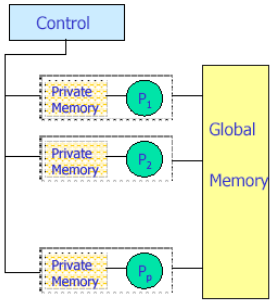
- 基本概念
  - 由Fortune和Wyllie1978年提出，又称SIMD-SM模型。
  - **PRAM（Parallel Random Access Machine）模型是单指令流多数据流（SIMD）并行机中的一种具有共享存储的模型。它假设有一个无限大容量的共享存储器，并且有多个功能相同的处理器，在任意时刻处理器可以访问共享存储单元。在PRAM中有一个同步时钟，所有的操作都是同步进行的。**
  - **PRAM is an abstract machine for designing the algorithms applicable to parallel computers. It eliminates the focus on miscellaneous issues such as synchronization and communication**

PRAM访存类型

- **PRAM-EREW互斥读互斥写**
  - Exclusive Read (ER) – all processors can simultaneously read from distinct memory locations
  - Exclusive Write (EW) – all processors can simultaneously write to distinct memory locations
- **PRAM-CREW并发读互斥写**
  - Concurrent Read (CR) – All processors can simultaneously read from any memory location
- **PRAM-CRCW并发读并发写**
  - Concurrent Write (CW) – All processors can write to any memory location

Parallel Random Access Machine (PRAM)

- Collection of numbered processors
- Accessing shared memory cells
- Each processor could have local memory (registers)
- Each processor can access any shared memory cell in unit time
- Input stored in shared memory cells, output also needs to be stored in shared memory
- PRAM instructions execute in 3-phase cycles
  - Read (if any) from a shared memory cell
  - Local computation (if any)
  - Write (if any) to a shared memory cell
- Processors execute these 3-phase PRAM instructions synchronously



Concurrent Write (CW)

- What value gets written finally?
  - Priority CW – processors have priority based on which write value is decided
  - Common CW – multiple processors can simultaneously write only if values are same
  - Arbitrary/Random CW – any one of the values are randomly chosen

PRAM特点

- **Unbounded** number of local memory cells
- Each memory cell can hold an integer of **unbounded** size
- All operations take **unit time**
- Unbounded collection of RAM processors – P0, P1, …,
- Each processor has unbounded registers
- Unbounded collection of shared memory cells
- Processors have a read, compute, write phase that happen synchronously
  - e.g. for all i, do A[i] = A[i-1]+1;
  - Read A[i-1], compute add 1, write A[i]
- Some subset of the processors can remain idle
- Think of it as SIMD parallelism

Strength of PRAM models

- Model A is computationally stronger(>=) than model B iff any algorithm written for B will run unchanged on A

Priority >= Arbitrary >= Common >= CREW >= EREW

Most powerful → Least powerful

Least realistic ← Most realistic

$$T_{EREW} = O(T_{CREW} \cdot \log p) = O(T_{CRCW} \cdot \log p)$$

Theorem. **A p-processor CRCW algorithm can be no more than O($\log p$) time faster than the best p-processor EREW algorithm for the same problem.**

Proof.

The proof is a simulation argument. We simulate each step of the CRCW algorithm with an O(log p)-time EREW computation.

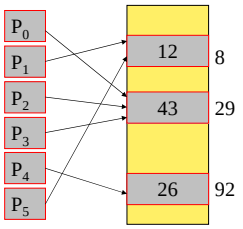
Because the processing power of both machines is the same, we need only focus on memory accessing.

Let's present the proof for simulating concurrent writes here. Implementation of concurrent reading is left a exercise.

End of the proof

- **Since the array A is sorted by first coordinate, only one of the processors writing to any given location actually succeeds, and thus the write is exclusive.**
- **This process thus implements each step of concurrent writing in the common CRCW model in O(log p) time**

- **The p processors in the EREW PRAM simulate a concurrent write of the CRCW algorithm using an auxiliary array A of length p.**



1.When CRCW processor P_i, for $i=0,1,\ldots,p-1$, desires to write a datum x_i to location l_i, each corresponding EREW processor P_i instead writes the ordered pair (l_i,x_i) to location A[i].

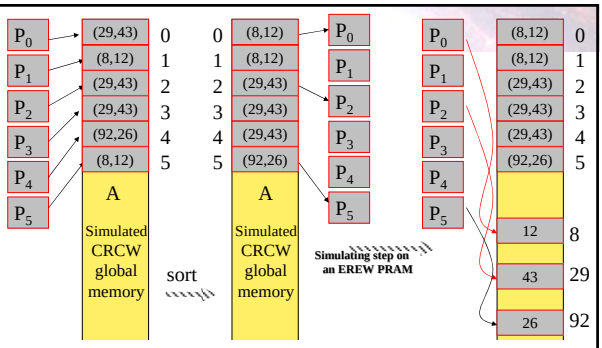
2. This writes are exclusive, since each processor writes to a distinct memory location.

3. Then, the array A is sorted by the first coordinate of the ordered pairs in O(log p) time, which causes all data written to the same location to be brought together in the output

The issue arises, therefore, of which model is preferable – CRCW or EREW

- **Advocates of the CRCW models point out that they are easier to program than EREW model and that their algorithms run faster**
- **Critics contend that hardware to implement concurrent memory operations is slower than hardware to exclusive memory operations, and thus the faster running time of CRCW algorithm is fictitious.**
  - **In reality, they say, one cannot find the maximum of n values in O(1) time**
- **Others say that PRAM is the wrong model entirely. Processors must be interconnected by a communication network, and the communication network should be part of the model**

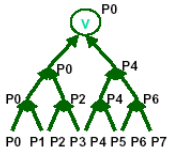
It is quite clear that the issue of the "right" parallel model is not going to be easily settled in favour of any one model. The important think to realize, however, is that these models are just that: models!



4. Each EREW processor P_i now inspects A[i]=(l_j,x_j) and A[i-1]=(l_k,x_k), where j and k are values in the range $0\le j,k\le p-1$. If $l_j \neq l_k$ *or* $i=0$ then P_i writes the datum x_j to location l_j in the global memory. Otherwise, the processor does nothing.

A Basic PRAM Algorithm

- Let there be "n" processors and "2n" inputs
- PRAM model: EREW
- Construct a tournament where values are compared



Processor k is active in step j
    if (k % 2ʲ) == 0
At each step:
    Compare two inputs,
    Take max of inputs,
    Write result into shared memory

Details:
    Need to know who is the "parent" and whether you are left or right child
    Write to appropriate input field

Example CREW-PRAM

- **Assume initially table A contains [0,0,0,0,0,1] and we have the parallel program**

for each $1 \le i \le 5$ do in parallel
$A[i] := A[i] + A[i+1]$

then the consecutive values of the tables A (in parallel step 0, 1, 2, 3, 4, 5) correspond to the Pascal triangle, the nonzero elements in the n-th row are

$\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \cdots \binom{n}{n}$
for $n = 0, 1, 2, 3, 4, 5, 6.$

PRAM-CRCW上快排序算法

输入：序列(A_1,…,A_n)和n个处理器
输出：供排序用的一棵二叉排序树

Begin

for each processor i do
  root=i
  f_i=root
  LC_i=RC_i=n+1
  end for

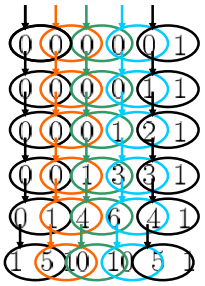
repeat for each processor i<>root do
  if (A_i<A_{fi})∨(A_i=A_{fi}∧i<f_i) then
    LC_{fi}=i
    if i=LC_{fi} then exit else f_i=LC_{fi} endif
    else
      RC_{fi}=i
      if i=RC_{fi} then exit else f_i=RC_{fi} endif
      endif
    end repeat
  End

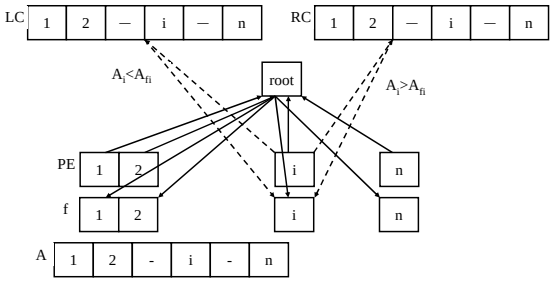
Pascal triangle

$\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \cdots \binom{n}{n}$
for $n = 0, 1, 2, 3, 4, 5, 6.$

PRAM CREW

for each $1 \le i \le 5$ do in parallel
$A[i] := A[i] + A[i+1]$





Finding Maximum: CRCW Algorithm

Given n elements A[0, n-1], find the maximum.
With n^2 processors, each processor (i,j) compare A[i] and A[j], for 0≤ i, j ≤n-1.

```
FAST-MAX(A):
1.  n←length[A]
2.  for i ←0 to n-1, in parallel
3.       do m[i] ←true
4.  for i ←0 to n-1 and j ←0 to n-1, in parallel
5.       do if A[i] < A[j]
6.            then m[i] ←false
7.  for i ←0 to n-1, in parallel
8.       do if m[i] =true
9.            then max ← A[i]
10. return max
```

	A[j]	5	6	9	2	9	m
A[i]	5	F	T	T	F	T	F
	6	F	F	T	F	T	F
	9	F	F	F	F	F	T
	2	T	T	T	F	T	F
	9	F	F	F	F	F	T

max=9

The running time is $O(1)$.
Note: there may be multiple maximum values, so their processors Will write to max concurrently. Its *work* = $n^2 \times O(1) = O(n^2)$.

PRAM模型

- 优点
  - 结构简单，便于进行理论分析（适合并行算法表示和复杂性分析）；易于使用，隐藏了并行机的通通信、同步等细节。
- 缺点
  - 不现实，容量无限大的存储器是不存在；不适合MIMD并行机；忽略了SM的竞争、通信带宽等因素的影响。

How practical is PRAM

- **Unbounded** number of local memory cells
  - Not true, memory is the bottleneck of many applications
- Each memory cell can hold an integer of **unbounded** size
  - We don't care much about this
- All operations take **unit time**
  - Very unrealistic for memory operations
  - As we traverse up the memory hierarchy, access time changes by a factor

Some variants of PRAM

- LPRAM
  - Charge a cost of *L* units to access global memory
  - Any algorithm that runs properly in a *p* processor PRAM can run in this model with a loss of factor *L*
- BPRAM
  - Charge *L* units to access first message
  - *B* units for each subsequent message

PRAM – unaccounted costs

- Non-local memory access
- Latency
- Bandwidth (greater problem in PRAM)
- Memory access contention
- Synchronization
  - What were the synchronization issues in our example problem

参考文献

- **钟诚, 陈国良, PRAM和LARPBS模型上的近似串匹配并行算法, Journal of Software, Feb. 2004,15(2):159-169**

Some variants of PRAM

- Bounded shared memory PRAM, PRAM(m)
  - Global memory segmented into modules
  - Any given step, only *m* memory accesses can be serviced
- Bounded number of processor PRAM
  - Any problem that can be solved for a *p* processor PRAM in *t* steps can be solved in a *p'* processor PRAM in *t'=O(tp/p')*steps

异步APRAM模型

- 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

APRAM模型的指令类型

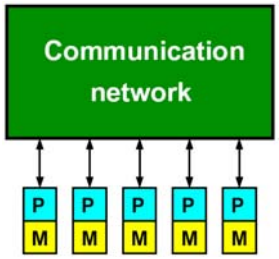
- APRAM中的指令有四种类型:
  - 全局读，将全局存储单元中的内容读到处理器的局部存储单元中;
  - 局部操作，对局部存储器中的数据执行局部操作，操作的结果存放到局部存储器中;
  - 全局写，将局部存储器单元中的内容写入全局存储单元中;
  - 同步，同步是计算中的一个逻辑点，在该点各处理器均需要等待其他的处理器也到达该点后才能继续执行它们的局部程序。

- 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_3	…	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				

BSP模型

- 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。



L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.

APRAM模型

- 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。

BSP模型

- A BSP computer consists of a collection of processors, each with its own memory. It is a distributed-memory computer.
- Access to own memory is fast, to remote memory slower.
- Uniform-time access to all remote memories.
- No need to open the black box of the communication network. Algorithm designers should not worry about network details, only about global performance.
- Algorithms designed for a BSP computer are portable: they can be run efficiently on many different parallel computers.

The BSP model emphasized the separation of communication from computation by incorporating the bulk-synchrony with a distributed memory model over message passing.

- A set of processor-memory pairs.
- A communication network that delivers messages in a point-to-point manner.
- A mechanism for the efficient barrier synchronization for all or a subset of the processes.
- There are no special combining, replicating, or broadcasting facilities.

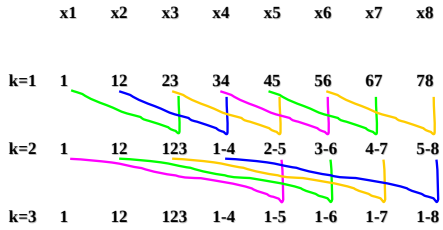
BSP algorithm

- A BSP algorithm consists of a sequence of supersteps.
- A computation superstep consists of many small steps, such as the floating-point operations (flops) addition, subtraction, multiplication, division.
- A communication superstep consists of many basic communication operations, each transferring a data word such as a real or integer from one processor to another.
- In our theoretical algorithms we distinguish between the two types of supersteps. This helps in the design and analysis of parallel algorithms.
- In our practical programs, we drop the distinction and mix computation and communication freely in each superstep.

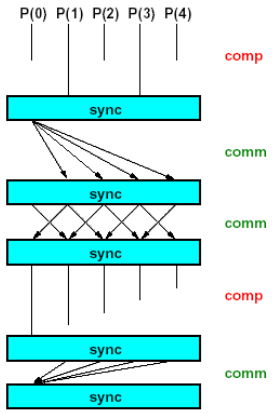
The communication network or *router* is described by only two parameters:
The *message latency L* : the time needed by a short message to travel across the network to its destination processor.
The *bandwidth factor g* : the parameter corresponds to a capacity constraint on the network. More precisely, it is defined as the ratio of local operations performed by all processors in one time unit to the total number of messages delivered by the router in the same time unit.

A BSP program is a sequence of *supersteps*. During each superstep, the processors (processor/memory modules) perform arbitrary local computations. At the end of each superstep, the processors synchronize and communicate by sending messages over the network (router). The router realizes supersteps in which each processor sends and receives at most *h* messages (*h*-relation). This pattern of independent computations followed by synchronization and communication steps is called *bulk-synchronous*.

n个元素{x1,x2,...,xn}，前缀和是n个部分和：
Si=x1*x2*...*xi, 1≤i≤n 这里*可以是＋或×



lg p. supersteps such that during the k^{th} superstep, the processes in the range $2^{k-1} \leqslant i \leqslant p$ each combine their local partial sums with process $i-2^{k-1}$.

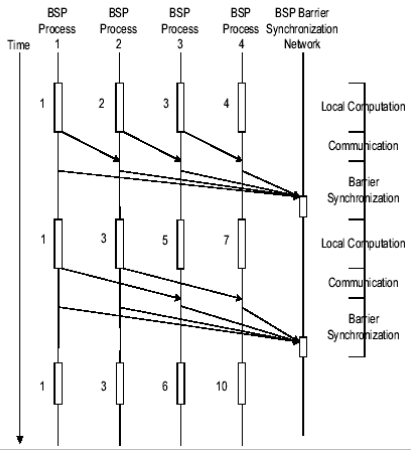



```
int bsp_allsums1( int x ){
  int k, left, right;
  bsp_pushregister( &left, sizeof(int));
  bsp_sync();
  right = x;
  for(k=1; k<bsp_nprocs(); k*=2){
    if( bsp_pid()+k < bsp_nprocs())
      bsp_put( bsp_pid()+k, &right, &left, 0,
       sizeof( int ));
    bsp_sync();
    if( bsp_pid() >= k ) right = left + right;
  }
  bsp_popregister( &left );
  return( right );
}
```

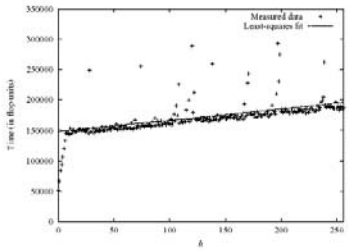
Make region global visible

Barrier

Push to remote memory



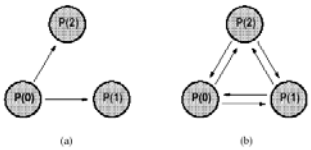
Time of an h-relation on an 8-processor IBM SP2



r = 212 Mflop/s, $p = 8$, $g = 187$ flop (0.88μs),
$l = 148212$ flop (698 μs)

Communication superstep: h-relation

2-relations:



- An h-relation is a communication superstep in which every processor sends and receives at most h data words: $h = \max\{h_s, h_r\}$.
- h_s is the maximum number of data words sent by a processor.
- h_r is the maximum number of data words received by a processor.

Cost of computation superstep

- $T = w + l$, where w is the maximum number of flops of a processor in the superstep.
- Processors with less than w flops have to wait. This waiting time is called idle time.
- To measure T, a wall clock is needed, giving the elapsed time. A CPU timer will not work, since it does not measure idle time.
- Same l as in communication superstep, for simplicity.

Cost of communication superstep

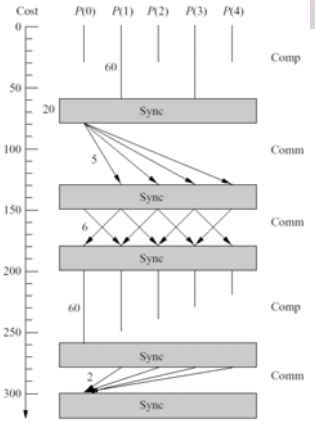
- $T(h) = hg + l$, where g is the time per data word and l the global synchronization time.
- Motivation hg: h determines communication time, since entry/exit of *processor* is the bottleneck .
- Motivation l: contains fixed overhead such as start-up costs of sending data, costs of checking whether all data have arrived at their destination, and costs of the synchronization mechanism itself.

Cost of algorithm

- The cost of a BSP algorithm is an expression of the form $a + bg + cl$:
  - This cost is obtained by adding the costs of all the supersteps.
  - Note that $g = g(p)$ and $l = l(p)$ are in general a function of the number of processors p.
  - The parameters a; b; c depend in general on p and on a problem size n.

Parallel algorithm supersteps

- Cost of BSP algorithm for p = 5, g = 2.5, l = 20 is 320 flops.
- First computation superstep costs 60 + 20 = 80 flops.
- First communication superstep costs 4*5*2.5 + 20 = 70 flops.



logP模型

- 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

Summary

- An abstract BSP machine is just a BSP(p, r, g, l) computer. This is all we need to know about the machine for developing algorithms. The parameters are:
  - p number of processors
  - r computing rate (in flop/s)
  - g communication cost per data word (in flop time units)
  - l global synchronisation cost (in flop time units)
- The BSP model consists of
  - a distributed-memory architecture with a black box communication network providing uniform-time access to remote memories;
  - an algorithmic framework formed by a sequence of supersteps;
  - a cost model giving cost expressions of the form $a + bg + cl$.

Lecture 1.2 Bulk Synchronous Parallel Model

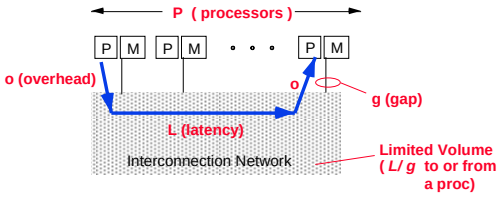
Deriving the LogP Model

° Processing
– powerful microprocessor, large DRAM, cache => P
° Communication
+ significant latency (100's of cycles) => L
+ limited bandwidth (1 – 5% of memory bw) => g
+ significant overhead (10's – 100's of cycles) => o
- on both ends
– no consensus on topology
=> should not exploit structure
+ limited capacity
– no consensus on programming model
=> should not enforce one

4.2 并行计算模型

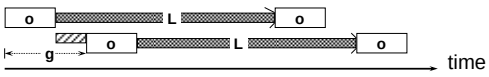
4.2.1 PRAM模型
4.2.2 异步APRAM模型
4.2.3 BSP模型
4.2.4 logP模型

LogP



- Latency in sending a (small) message between modules
- overhead felt by the processor on sending or receiving msg
- gap between successive sends or receives (1/BW)
- Processors

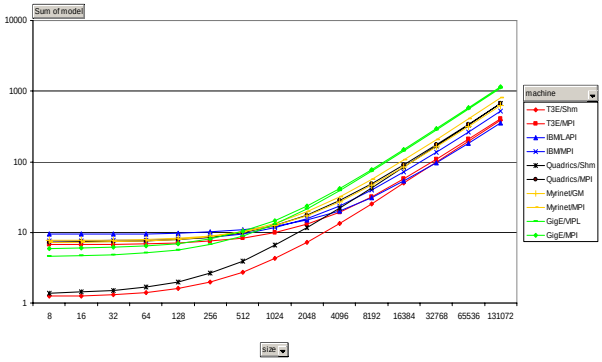
Using the Model



- Send n messages from proc to proc in time *2o + L + g(n-1)*
  - each processor does *o n* cycles of overhead
  - has *(g-o)(n-1) + L* available compute cycles
- Send *n* messages from *one to many* in same time

- Send *n* messages from *many to one* in same time
  - all but *L/g* processors block so fewer available cycles

Model Time Varying Message Size & Machines



Latency and Bandwidth Model

- Time to send message of length n is roughly

  Time = latency + n*cost_per_word
  = latency + n/bandwidth

- Topology is assumed irrelevant.
- Often called "α–β model" and written

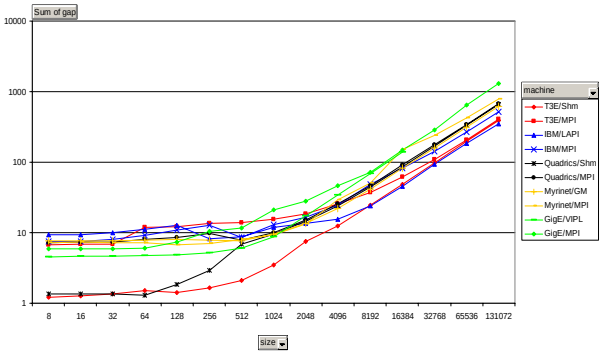
  Time = α + n*β

- Usually α >> β >> time per flop.
  - One long message is cheaper than many short ones.

    α + n*β << n*(α + 1*β)

  - Can do hundreds or thousands of flops for cost of one message.
- Lesson: Need large computation-to-communication ratio to be efficient.

Measured Message Time



Alpha-Beta Parameters on Current Machines

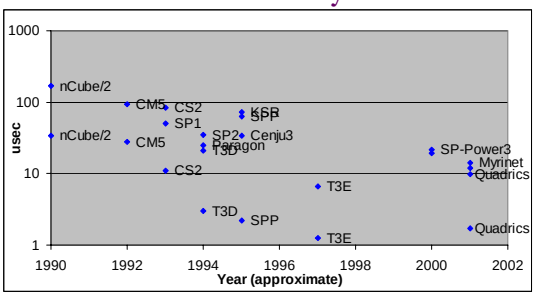
- These numbers were obtained empirically

machine	α	β
T3E/Shm	1.2	0.003
T3E/MPI	6.7	0.003
IBM/LAPI	9.4	0.003
IBM/MPI	7.6	0.004
Quadrics/Get	3.267	0.00498
Quadrics/Shm	1.3	0.005
Quadrics/MPI	7.3	0.005
Myrinet/GM	7.7	0.005
Myrinet/MPI	7.2	0.006
Dolphin/MPI	7.767	0.00529
Giganet/VIPL	3.0	0.010
GigE/VIPL	4.6	0.008
GigE/MPI	5.854	0.00872

α is latency in usecs
β is BW in usecs per Byte

How well does the model Time = α + n*β predict actual performance?

End to End Latency Over Time

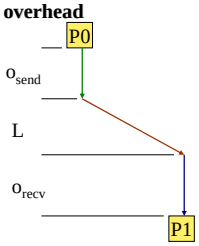


- Latency has not improved significantly, unlike Moore's Law
  - T3E (shmem) was lowest point – in 1997

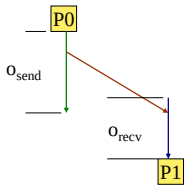
Data from Kathy Yelick, UCB and NERSC

LogP Parameters: Overhead & Latency

- Non-overlapping overhead



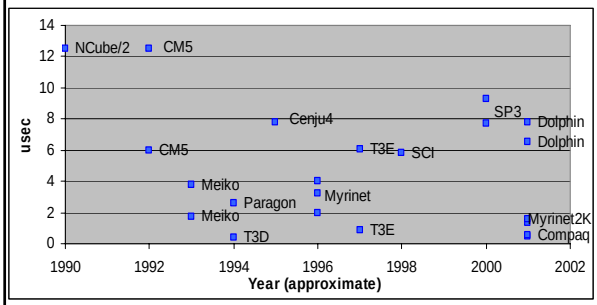
- Send and recv overhead can overlap



EEL = End-to-End Latency
= o_{send} + L + o_{recv}

EEL = f(o_{send}, L, o_{recv})
≥ max(o_{send}, L, o_{recv})

Send Overhead Over Time

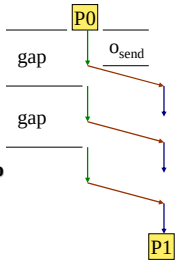


- Overhead has not improved significantly; T3D was best
  - Lack of integration; lack of attention in software

Data from Kathy Yelick, UCB and NERSC

LogP Parameters: gap

- The Gap is the delay between sending messages
- Gap could be larger than send ovhd
  - NIC may be busy finishing the processing of last message and cannot accept a new one.
  - Flow control or backpressure on the network may prevent the NIC from accepting the next message to send.
  - No overlap ⇒
    time to send n messages =

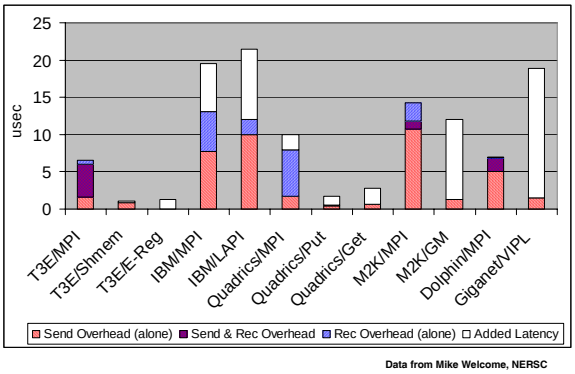


$(o_{send} + L + o_{recv} - gap) + n*gap = \alpha + n*\beta$

Limitations of the LogP Model

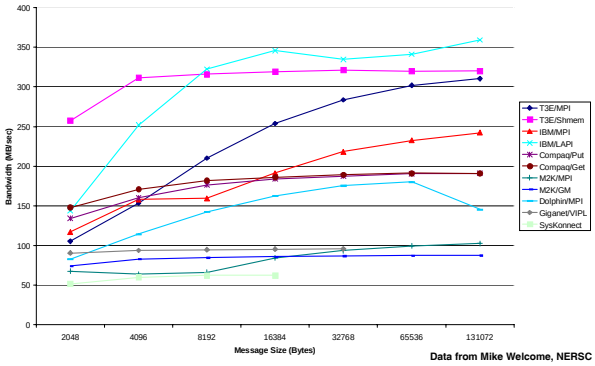
- The LogP model has a fixed cost for each messages
  - This is useful in showing how to quickly broadcast a single word
  - Other examples also in the LogP papers
- For larger messages, there is a variation LogGP
  - Two gap parameters, one for small and one for large message
  - The large message gap is the β in our previous model
- No topology considerations (including no limits for bisection bandwidth)
  - Assumes a fully connected network
  - For some algorithms with nearest neighbor communication, but with "all-to-all" communication we need to refine this further
- This is a flat model, i.e., each processor is connected to the network
  - Clusters of SMPs are not accurately modeled

Results: EEL and Overhead



Data from Mike Welcome, NERSC

Bandwidth Chart



Data from Mike Welcome, NERSC

logP模型

- 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

1. S. E. Hambrusch, "Models for Parallel Computation", *Proceedings of Workshop on Challenges for Parallel Processing, International Conference on Parallel Processing, 1996.* [Good overview]
2. "LogP Performance Assessment of Fast Network Interfaces", D. E. Culler, Lok Tin Liu, Richard P. Martin, and Chad Yoshikawa, IEEE Micro 1996.
3. "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture" R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson, *Technical Report CSD-96-925*, Berkeley, Nov. 1996.
4. "LogGP: Incorporating Long Messages into the LogP Model --- One step closer towards a realistic model for parallel computation", Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, Chris Scheiman, *TRCS95-09*, Computer Science Department, University of California, Santa Barbara, April 1995.
5. "LogP:Towards a Realistic Model of Parallel Computation", D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken, , 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, May 1993.
6. P. M. B. Vitanyi, "Locality, communication and interconnect length in multicomputers", *SIAM J. on Computing*, **17**:659-672 (1988). [The web version is apparently a modified version of the Journal paper]