Lecture 5: Elementary Parallel Algorithms

Lecture Outline

(Reading: Chapter 6 of textbook)

1. Classfying MIMD algorithms
2. General approaches
3. Solving the reduction problem
4. Designing algorithms on processor arrays using reduction
5. Broadcast
6. Prefix sums on multicomputers

1. Classfying MIMD algorithms

(1) **Pipelined Algorithms**. A pipelined algorithm is an ordered set of segments in which the output of one segment is the input of its successor. The input to the algorithm is the input to the first segment. the output of the last segment is the output of the algorithm.
All segments should produce outputs at the same rate. Or the slowest segment becomes the bottleneck.
Example. Parallel compilers: scanning, parsing, code generation, code optimization.

(2) **Partitioning**. A problem is divided into subproblems that are solved by individual processors. The solutions to the subproblems are then combined to form the problem solution. Synchronization is essential. So also callled **synchronous algorithms**.
Example. Adding kp numbers on a multiprocessor with p processors. Initialize to 0 a global variable S. Create p processes, one for each processor. Each process adds add k numbers and then add the subtotal to the grand total. Each process must have exclusive access to S, which requires synchronization.
Partitioning algorithms can be further divided into **prescheduled algorithm** and **self-scheduled algorithms**.

a. **Prescheduled algorithms**. Each process is allocated its share of computation task at compile time. (The above example belongs to this category). Prescheduling is the norm when a computation consists a large number of subtasks, each with a known complexity. If a computation consists of large number of subtasks, and the exact amount of time needed to perform each subtask varies dramatically, we can either use self-scheduling, or still use prescheduling, but creating more subtasks than processes. If each process is executing a large number of subtasks and this set of subtasks is a random selection from the entire set of subtasks, then there is a good probability that every process will end up doing the same amount of work.

b. **Self-scheduled algorithms**. The work to each process is not assigned until run-time. A global list of work to be done is kept and when a process needs work, a subtask is removed from the list.

(3) **Asynchronous**. A control-parallel algorithm that is not pipelined.

2. General approaches

Given a problem Π to be solved, how to obtain a parallel algorithm solving it?

(1) Come up with a new parallel algorithm;

(2) Modify an existing sequential algorithm;

(3) Adapt an existing parallel/PRAM algorithm.

3. Solving the reduction problem

(0) The problem: perform a reduction on a set of n values, where n is much larger than the number of available processors p.

(1) When is it appropriate to implement a PRAM-style algorithm on a real parallel computer?

> **Design Strategy 1**. If a cost optimal CREW PRAM algorithm exists, and the way the PRAM processors interact through shared variables maps onto the target architeture, a PRAM algorithm is a reasonable starting point.

(2) A cost-optimal PRAM algorithm (in Ch. 2): $n/\log n$ processors add n numbers in $\Theta(\log n)$ time.

4. Designing algorithms on processor arrays using reduction

(0) Features: no need to concern the synchronization problems. Communication costs are not zero. In fact it may be dominating. Assume adding $n = 2^m = l^2$, $m, l > 0$, n values to be added are $A = (a_0, a_1, \cdots, a_{n-1})$.

(1) Algorithm for SIMD-CC model: adds $n = 2^m$ value (Fig.6-1,6-2,6-3, p.161,162,163).

a. The pseudo code: Fig.6-1, p.161. At the end the algorithm P_0 holds the sum of all values.

b. Illustration: Fig.6-2, p.162

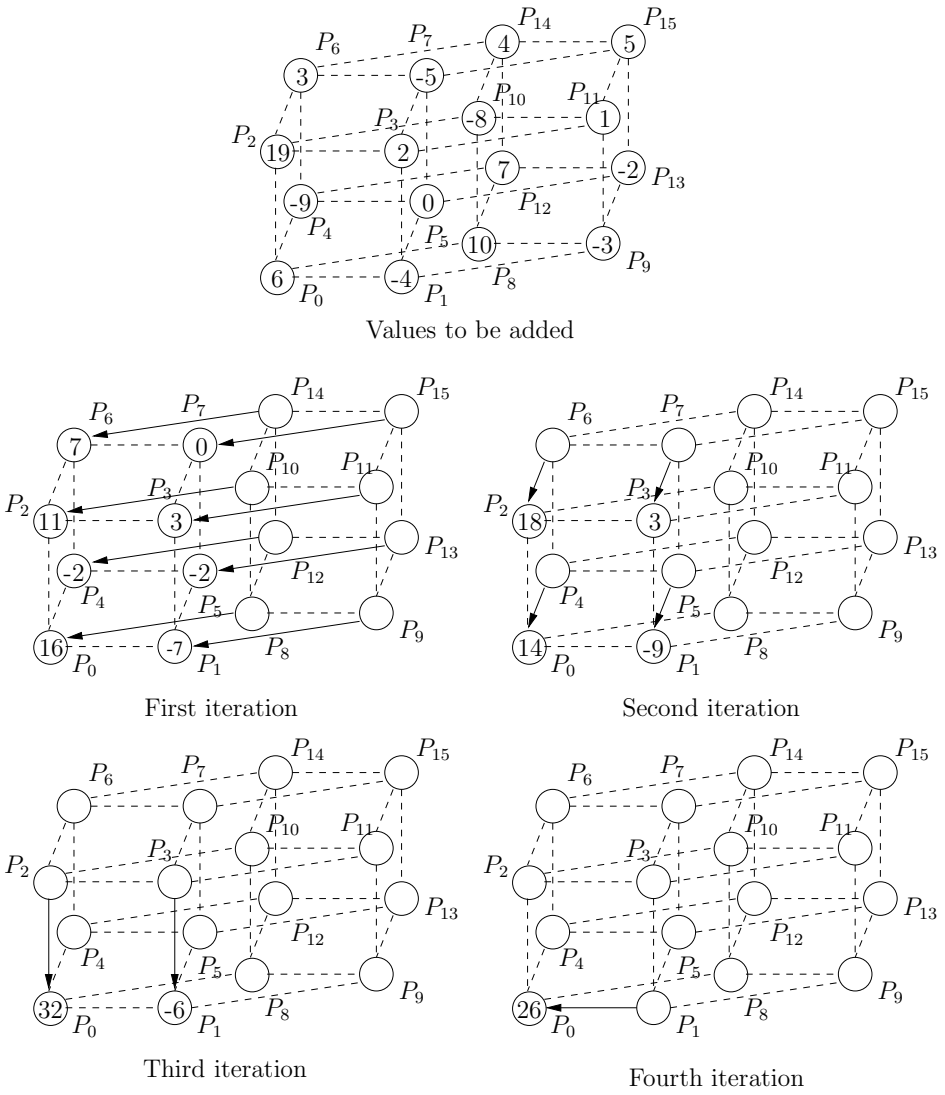

Figure 66: Parallel summation on the hypercube SIMD model (Fig.6-2,p.162).

SUMMATION(HYPERCUBE SIMD):
Parameters: n: {Number of elements to add}
p: {Number of processing elements}
Global: j
Local: *local.set.size*, *local.value*$[1...\lceil n/p \rceil]$, *sum, tmp*
begin
for all P_i, where $0 \leq i \leq p-1$ **do**
if $i <$ (n modulo p) **then**
local.set.size $\leftarrow \lceil n/p \rceil$
else
local.set.size $\leftarrow \lfloor n/p \rfloor$

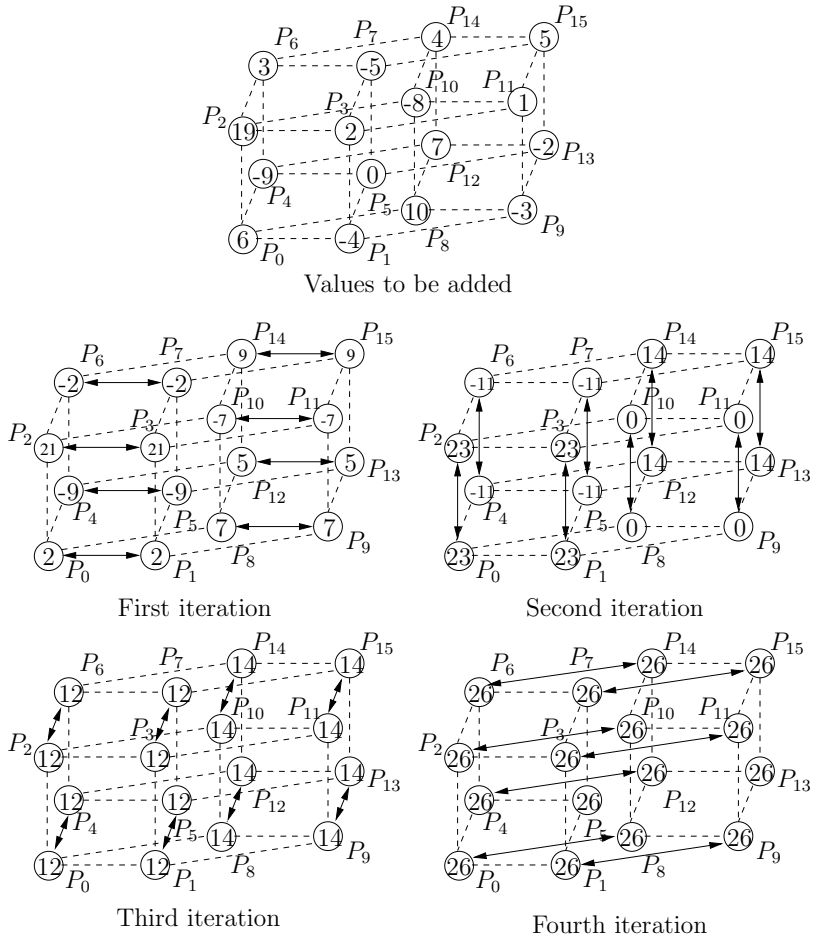

Figure 67: Another parallel summation for the hypercube SIMD model. After the processing elements have found the sum of their local values, they perform $\log p$ swap-and-accumulate steps, one for each dimension of the hypercube (Fig.6-3,p.163).

```
    endif
    sum ← 0
endfor
for j ← 1 to ⌈n/p⌉ do
    for all P_i, where 0 ≤ i ≤ p − 1 do
        if local.set.size ≥ j then
            sum ← sum + local.value[j]
        endif
    endfor
endfor
for j ← log p − 1 downto 0 do
    for all P_i, where 0 ≤ i ≤ p − 1 do
```

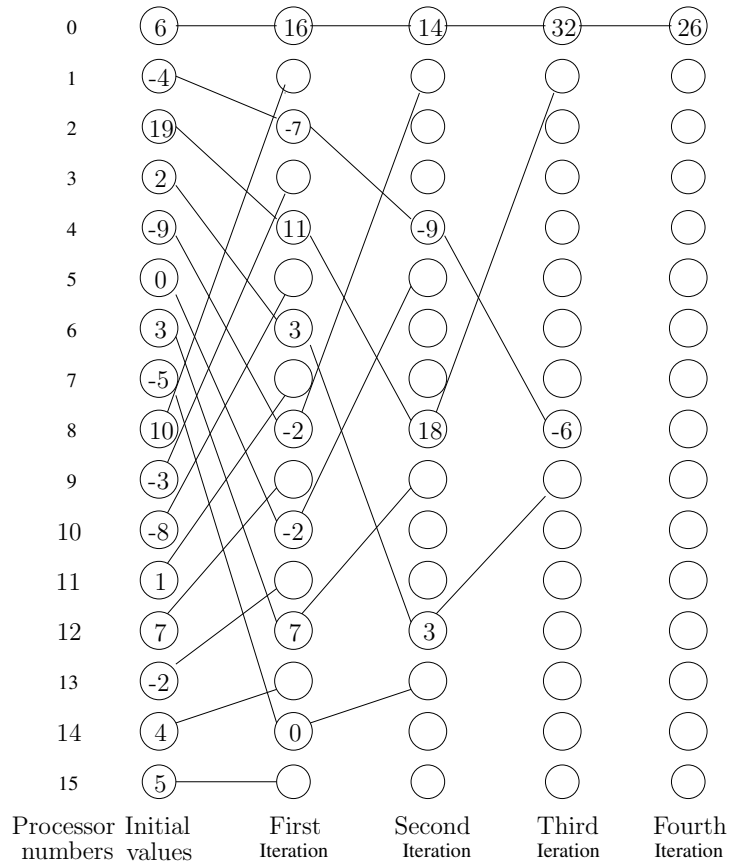

Figure 68: Finding sum of 16 values on the shuffle-exchange SIMD model (Fig.6-5,p.165).

if $i < 2^j$ **then**
$tmp \Leftarrow [i + 2^j]sum$
$sum \leftarrow sum + tmp$
endif
endfor
endfor
end

Fig.6-1 Algorithm for the hypercube SIMD model to sum n values. Upon completion of the algorithm processor 0's value of *sum* is the global sum.

c. Another algorithm: Fig.6-3, p.163. At the end of this algorithm every node has the sum of all values.

d. Complexity: $\Theta(n/p + \log p)$ (easy to see).

(2) Algorithm for SIMD-PS model: adds $n = 2^m$ values (Fig.6-4,6-5, p.164,165).

a. Two importance procedures: **shuffle** and **exchange**, which will perform the *shuffle* and *exchange* operations respectively.

b. The pseudo code: Fig.6-4, p.164.

c. Illustration: Fig.6-5, p.165

d. Question: Fig.6-5 shows that at the end of the algorithm P_0 holds of sum of all values. What values are in the local variable *sum* for all other processors?

```
SUMMATION(SHUFFLE-EXCHANGE SIMD):
Parameters: n: {Number of elements to add}
            p: {Number of processing elements}
Global:     j
Local:      local.set.size, local.value[1...⌈n/p⌉], sum, tmp
begin
   for all P_i, where 0 ≤ i ≤ p − 1 do
      if i < (n modulo p) then
         local.set.size ← ⌈n/p⌉
      else
         local.set.size ← ⌊n/p⌋
      endif
      sum ← 0
   endfor
   for j ← 1 to ⌈n/p⌉ do
      for all P_i, where 0 ≤ i ≤ p − 1 do
         if local.set.size ≥ j then
            sum ← sum + local.value[j]
         endif
      endfor
   endfor
   for j ← 0 to log p − 1 do
      for all P_i, where 0 ≤ i ≤ p − 1 do
         shuffle(sum) ⇐sum
         exchange(tmp) ⇐sum
         sum ← sum + tmp
      endfor
   endfor
end
```

Fig.6-4 Algorithm for the shuffle-exchange processor array model.

e. Complexity: $\Theta(n/p + \log p)$ again (easy to see).

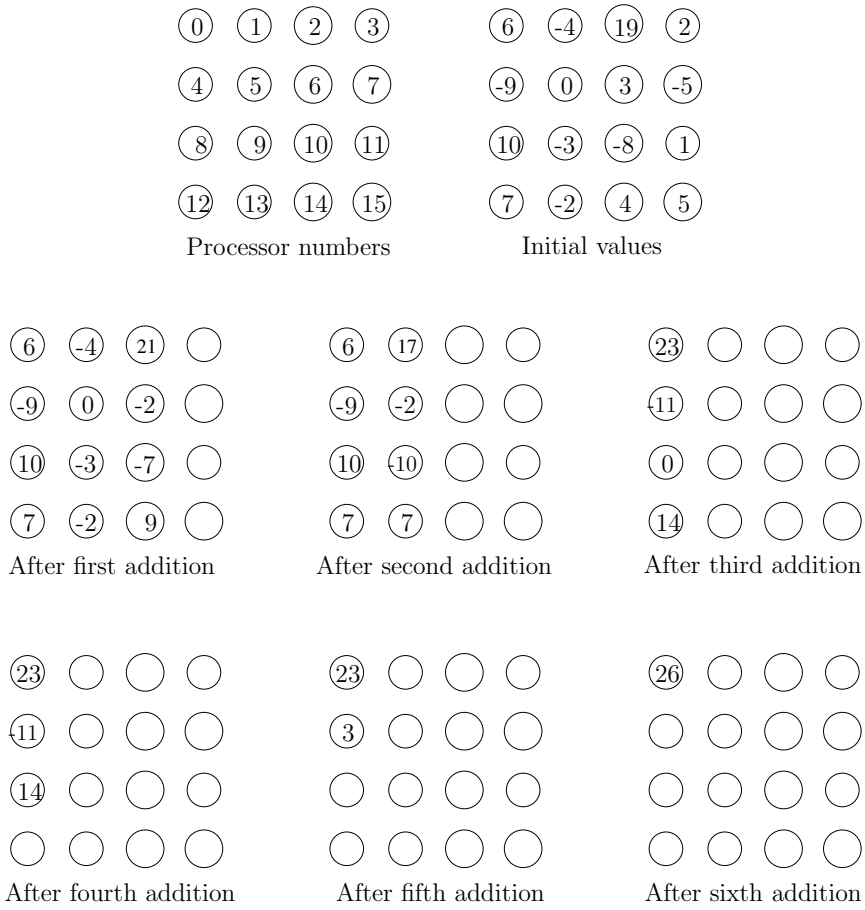

Figure 69: Finding sum of 16 values on a processor array organized as a 2-D mesh (Fig.6-7,p.167).

(3) Algorithm for SIMD-MC^2 model

a. There are n matrix elements distributed evenly among the l^2 2-D mesh processors.

b. The algorithm consists of three phases.

- Phase 1: as usual, add local sums at each node sequentially;
- Phase 2: starting from *right most* column, the computation moves toward the *left most* column. At each iteration, only processors at one column are active.
- Phase 2: starting from *lowest* row, the computation moves toward the *top* row. However, during each iteration only one node in the left most column is active.

c. Fig.6-7 illustrattes the basic ideas. Assume that the local sums have been computed there at the initial value distribution configuration.

```
SUMMATION(2-D MESH SIMD):
Parameters:  l:   {Mesh has size l × l}
Global:      i
Local:       sum, tmp
begin
    {Each processor finds sum of its local values – code not shown}

    {Processors elements in column i active }
    for i ← l − 1 downto 1 do
        for all P_{j,i}, where 1 ≤ j ≤ l do
            {P_{j,i} receives value sum from P_{j,i+1}}
            tmp ⇐ east(sum)
            sum ← sum + tmp
        endfor
    endfor
    {Move up on rows. P_{i,1} receives sum from P_{i−1,1}}
    for i ← l − 1 downto 1 do
        for all P_{i,1} do
            tmp ⇐ south(sum)
            sum ← sum + tmp
        endfor
    endfor
end
```

Fig.6-6 Parallel summation on a processor array organized as a 2-D mesh. Upon completion of algorithm, variable $[1,1]sum$ contains the global sum.

```
SUMMATION(2D MESH SIMD)
begin
    {Each processor finds sum of its local values – code not shown}

    for i ← l − 1 downto 1 do
        for all P_{j,i}, where 1 ≤ j ≤ l do {Column i active }
            t_{j,i} ⇐ a_{j,i+1}
            a_{j,i} ← a_{j,i} + t_{j,i}
        endfor
    endfor
    for i ← l − 1 downto 1 do
        for all P_{i,1} do {Only a single processing element active }
            t_{i,1} ⇐ a_{i+1,1}
            a_{i,1} ← a_{i,1} + t_{i,1}
        endfor
```

endfor
end

Fig.6-6′ Another parallel summation on a processor array organized as a 2-D mesh. There is only one local variable t that is expressed with subscripts. $A[1,1]$ contains the global sum at the end of algorithm.

d. Fig.6-6′ (not from textbook) shows the same algorithm with only one local variables t that is expressed using processor subscipts.

e. Time complexity. It requires $2(l-1) = 2\sqrt{n} - 2$ iterations. Hence complexity is $\Theta(\sqrt{n})$.
This result is optimal. Therefore SIMD-MC^2 cannot add a series of numbers as quickly as SIMD-PS or SIMD-CC models. The inefficiency comes from interprocessor communications.

(4) UMA multiprocessor Model

a. Review of the UMA multiprocessor model (Sect.3.3.1 and Fig.3-15)

(a) Like PRAM model, UMA multiprocessors also have a shared memory that is accessible by individual processors through a switching network. However processors in a UMA multiprocessor computer executes asynchronously. Recall that in PRAM model processors are activated gradually and are synchronized.

(b) Problem. Synchronization among individual processors is critical. For the summation problem, it must be ensured that no processor accesses a variable containing the partial sum until the variable is set.

(c) The number of processors p is assumed to be a power of 2.

b. First version (Fig.6-8,p.168). The algorithm has two phases.

(a) Each processor is assigned (or responsible for) n/p elements. In the first phase, each processor computes the sum of those elements. Notice that the computation actions performed by individual processors cannot be synchronized.

(b) Each processor maintains a boolean variable *flags* that is initialized to 0. This variable is used to signal if the partial sum is ready to be used in the final total sum. A processor enters its second phase after it finishes computing sum of its locally assigned values.

(c) When a processor *in the second half* finishes its calculation of its own sum of assigned elements, it will turn the variable *flags* to 1, indicating that the partial sum is available for others to include into the final total sum.

(d) The algorithm then uses a binary search style control to compute the total sum. More specifically, a processor in the first half, say $P_i, 0 \leq i \leq p/2$ will retrieve and add the partial sum from a corresponding processor $P_{i+p/2}$ at the second half. This processor P_i will then change the value of its variable *flags* to 1. Then processors at the first quarter will retrieve and add the partial sum from a corresponding processor at the second quarter. It then set the value of its variable *flags* to 1. And so on.

```
SUMMATION(UMA MULTIPROCESSOR, VERSION 1):
Global   a[0...(n − 1)]          {Values to be added}
         p                       {Number of processors, a power of 2}
         flags[0...(p − 1)]      {Set to 1 when partial sum available}
         partial[0...(p − 1)]    {Contains partial sum}
         global.sum              {Result stored here}
Local    local.sum
begin
   for k ← 0 to p − 1 do
      flags[k] ← 0
   endfor
   for all P_i, where 0 ≤ i ≤ p − 1 do
      local.sum ← 0
      for j ← i to n − 1 step p do
         local.sum ← local.sum + a[j]
      endfor
      j ← p
      while j > 0 do
         if i ≥ j/2 then
            partial[i] ← local.sum
            flags[i] ← 1
            break
         else
            while (flags[i + j/2] = 0) do endwhile; { spin }
            local.sum ← local.sum + partial[i + j/2]
         endif
         j ← j/2
      endwhile
      if i = 0 then
         global.sum ← local.sum
      endif
   endfor
```

end

Fig.6-8 UMA multiprocessor summation program that uses fan-in stragety to compute global sum from subtotals.

(e) Worst-case complexity:

i. We have to realize that a process has to be created to run on each of the p processors.

ii. Process creation could take $\Theta(p)$ time. However this will normally not be counted as cost because a process is only created once and because $p \ll n$.

iii. $\Theta(p)$ time is needed to initialize the variable *flags*.

iv. $\Theta(n/p)$ time is needed to add the n/p values sequentially at each process (Assume that memory bank conflicts does not increase the complexity by more than a constant factor).

v. The **while** loop takes $\Theta(\log p)$ time.

vi. Synchronization occurs at the last **endfor** (in order for the whole algorithm to terminate). Assume a single global semaphore is used, it will take $\Theta(p)$ time.

vii. Therefore overall worst-case complexity:

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

c. Second version (Fig.6-9,p.169).

(a) Motivation: because the complexity of the first version is already $\Theta(n/p + p)$, the second version is simpler yet having the same complexity.

(b) Basic ideas. The second version also has two phases.

i. Each processor is still responsible for n/p values. In the first phase each processor still computes partial sum of its assigned values. This phase takes $\Theta(n/p)$ time.

ii. In the second phase the p processors will enter its critical section attempting to lock the global variable *global.sum* in order to add its local sum to it. The phase will takes $\Theta(p)$ time.

Therefore the complexity of the second version is still $\Theta(n/p + p)$.

SUMMATION(UMA MULTIPROCESSOR, VERSION 2):

Global	$a[0...(n-1)]$	{Elements to be added}
	global.sum	{Result stored here}
Local	*local.sum*	{Sum of processor's share of values}

```
        j                                {Processor's private loop index}
begin
    global.sum ← 0
    for all P_i, where 0 ≤ i ≤ p − 1 do
        local.sum ← 0
        for j ← i to n − 1 step p do
            local.sum ← local.sum + a[j]
        endfor
        lock(global.sum)
        global.sum ← global.sum + local.sum
        unlock(global.sum)
    endfor
end
```

Fig.6-9 UMA multiprocessor summation program in which each process enters a critical section to adds its subtotal to the variable accumulating the global sum.

d. Comparison and discussion of the two versions (Fig.6-10,p.169).

(a) The diagram shows that the first version algorithm is superior over the critical-section style algorithm. This is because the constant of proportionality associated with the $\Theta(p)$ term is smaller for the 1st algorithm.

(b) The second algorithm forces each of the p processors executes its critical section sequentially, while the first algorithm allows multiple processors active in the same time in second phase.

(c) Notice that the first algorithm is obtained by direct application of the PRAM reduction strategy in the second phase. Because each processor applies same strategy in parallel in the second phase the first version is more like a data parallel algorithm. Notice also that PRAM algorithms are data-parallel. This motivates the next design strategy.

> **Design Strategy 2**. Look for a data-parallel algorithm before considering a control-parallel algorithm.

(5) Data-parallel algorithm on MIMD machines: SPMD programs

a. Data parallelism is the only parallelism available on SIMD machines.

b. Data-parallel algorithms: more common, easier to design and debug, better scalibility.

c. Data-parallel algorithms on MIMD: SPMD (single program multiple data). Easier to write and debug than arbitrary MIMD algorithms

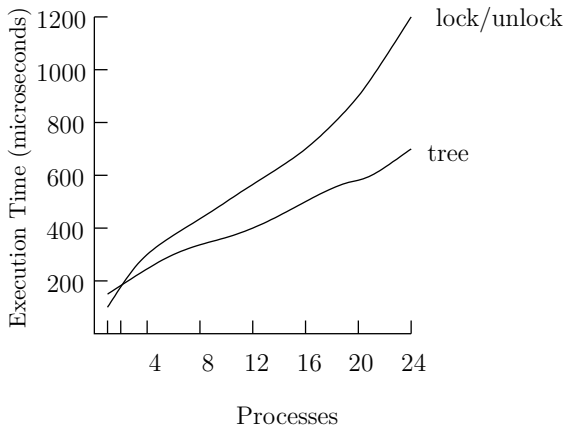

Figure 70: Comparison of the execution times of the reduction phases of two sum-finding algorithms, implemented on the Sequent Balance. (Data courtesy Bradley K. Seevers.) (Fig.6-10, p.169)

5. Broadcast

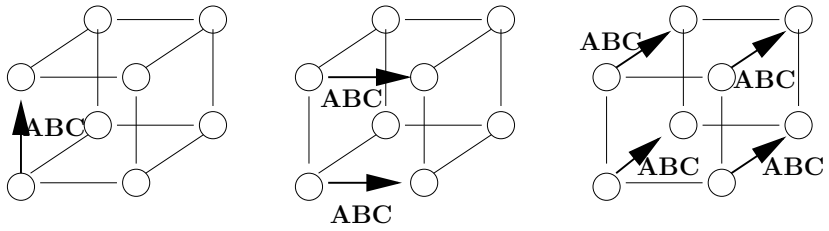

Figure 71: Hypercube broadcast algorithm based on binomial tree communication pattern. Message elements ABC are sent together, to avoid repeated message startup costs (Fig.6-11,p.170)

(1) The eventual size of the applications must be considered during the design stage.
Eventually one of the machine resources (CPU speed, communication speed, memory size, and etc.) will become a bottleneck.

(2) Example problem: broadcast a list of values to other processors on hypercube multicomputer.

a. Two major components of time: message startup time (also called message-passing overhead, or message latency), and data-transfer time.

b. For small of amount data, message-passing overhead dominates.

(a) Minimize the number of communications performed by any processor.

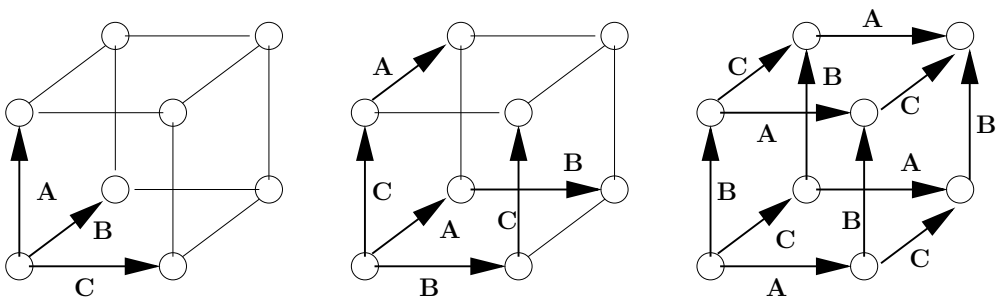

Figure 72: Johnson and Ho's (1989) hypercube broadcast algorithm makes better use of the available communication links to improve performance when the message is long. (Fig.6-13,p.171)

(b) Binomial tree is suitable pattern because there is a dilation-1 embedding of a binomial tree into a hypercube.

(c) $\log p$ communications needed (Fig.6-11, p.170).

(d) Fig.6-12,p.171 shows pseudocode.

i. The operation $\otimes$ is logical *exclusive or*.

ii. The execution of the statement

$$[partner]\text{value} \Leftarrow \text{value}$$

by processor P_i will send its value in *value* to processor $P_{partner}$

c. For large amount of data, data-transfer time is dominating.

(a) Binomial tree-based algorithms have a serious weakness – at any one time at most $p/2$ processors out of $p \log p$ communication links are in use.

(b) If it takes M time to pass a message from one processor to another processor, the total broadcast time will be $M \log p$.

d. A more efficient algo by Johnsson and Ho (1989): upto $\log p$ times faster than a binomial-tree algorithm (Fig.6-13,p.171).

(a) Based on the fact that every hypercube contains $\log p$ edge-disjoint spanning trees with the same root node.

(b) The algorithm breaks the messages into $\log p$ parts and broadcast each part to the other nodes through a different binomial spanning tree.

(c) Because the spanning trees have no edges in common, all data flow concurrently – the entire algorithm takes $M \log p / \log p = M$ time.

(d) In a hypercube multiprocessor, the most constrained resource for broadcast is the network capacity. The second algo makes better use of this resource.

```
HYPERCUBE.BROADCAST(id, source, value):
Parameters:  p:          {Number of processors}
Value:       id          {Processor's ID number}
             source      {ID of source processor}
Reference:   value       {Value to be broadcast}
Local:       i           {Loop iteration}
             partner     {Partner processor}
             position    {Position in broadcast tree}
{This procedure is called from inside a for all statement}
begin
   position ← id ⊗ source
   for i ← 0 to log p − 1 do
      if position < 2^i then
         partner ← id ⊗ 2^i
         [partner]value ⇐ value
      endif
   endfor
end
```

Fig.6-12 Pseudocode for hypercube broadcast algorithm based on binomial tree communication pattern.

(3) **Design Strategy 3**. As problem size grows, use the algorithm that makes best use of the available resouces.

6. Prefix sums on multicomputers

(1) Review of the cost-optimal PRAM algorithm in Sec.2-4 (Fig.2-21)

a. The operation $\oplus$ is generic.

b. Requires $n/\log n$ processors to solve the problem in time $\Theta(\log n)$ time.

c. Each processor uses the optimal sequential algorithm to manipulate its own $\log n$ elements.

(2) **Design Strategy 4**. Let each processor perform the most efficient sequential algorithm on its share of data.

(3) Prefix sum on mesh and hypercube networks

a. Assumptions

(a) There are p processors and n numbers (in array A) such that n/p is an integer.

(b) Let

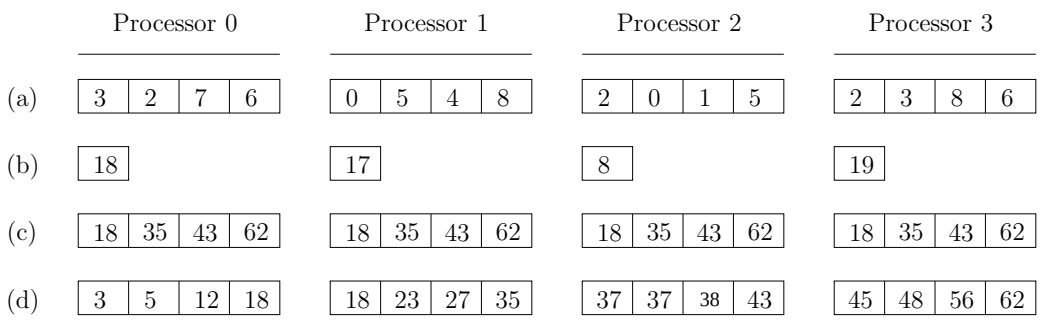

Figure 73: Computing prefix sums of 16 values on a four-processor multicomputer. (a) Each processor is allocated its share of the values. (b) In step one each processor finds the sum of its local elements. (c) In step two the prefix sums of the local sums are computed and distributed to all processors. (d) In step three each processor computes the presix sums of its own elements and adds to each result the sum of the values held in lower-numbered processors (Fig.6-14, p.173)

- χ be the time to perform the operation $\oplus$
- λ be the time needed for one processor to initiate a message to another
- β be the message transmission time per value

Hence, sending k elements from one processor to another requires $\lambda + k\beta$ time.

b. The basic ideas of the method are illustrated in Fig.6-14, p.173
   (a) Initially, the n values are distributed evenly among the local memories of the p processors, n/p values per processor
   (b) Step 1: each processor finds the sum of its n/p elements
   (c) Step 2: the processors cooperate to find the p prefix sums of their local sums
   (d) Step 3: Each processor computes the prefix sums of its n/p values, adding to each result to the sum of the values held in lower-numbered processors

c. Computation time for step 1 and 3 are independent of the topology:
   * Step 1: $(n/p - 1)\chi$ time units to add n/p values.
   * Step 3: Processor 0 computes prefix sums of its n/p values in $(n/p-1)\chi$ time; processor 1 to processor $p-1$ computes taking $(n/p)\chi$ time.

d. The communication time taken by step 2 depends on the topology.

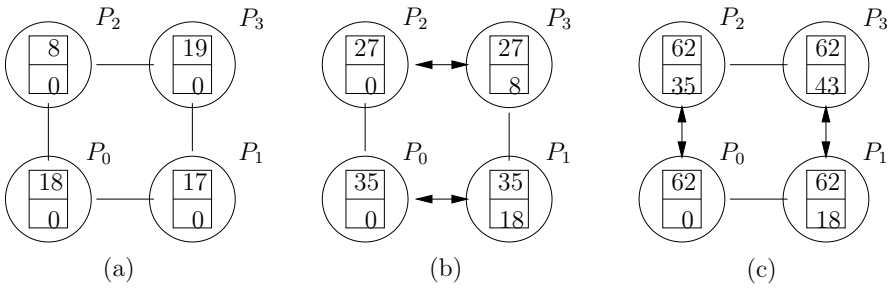

Figure 74: Computing prefix sums of 16 values on a four-processor multicomputer. This is step (b) of the hypercube prefix sums algorithm illusrated in Fig.6-14. (a) Every processor has two variables. Upper variable accumulates global sum. Lower variable accumulates prefix sum. The upper variable is initialized to the processor's local value. The lower variable is initialized to zero. (b) Swap value of upper variable across dimension 0 of hypercube. Every processor adds this value to the value of its upper variable. Higher-numbered processor adds lower-numbered processor's value to the value of its lower variable. (c) Swap value of upper variable across dimension 1 of hypercube. Every processor adds this value to the value of its upper variable. Higher-numbered processor adds lower-numbered processor's value to the value of its lower variable. (Fig.6-15, p.174)

- The memory access pattern of the PRAM algorithm shown in Fig.2-10 does not directly translate into comm. pattern having a dilation-1 embedding in a hypercube.
- Therefore the above ideas cannot be directed implemented on a mesh or hypercube based multicomputer.

e. To implement the above discussed ideas in a multicomputer with processor organizations like mesh or hypercube, we have to consider and take advantage its special connection topology. For hypercube multicomputers, we can modify the reduction algorithm on Fig.6-3 (which is for adding numbers on hypercubes) for prefix sums.

(a) Each processor contains two variables:
   - One contains the total of all values received;
   - Another contains the total of all values received from lower-numbered processors.

(b) At the end of $\log p$ swap-and-add steps, the second variable associated with each processor contains the prefix sum of for that processor.

Fig.6-15, p.174, illustrated the basic operations of the algorithm.

f. Complexity of the algorithm:

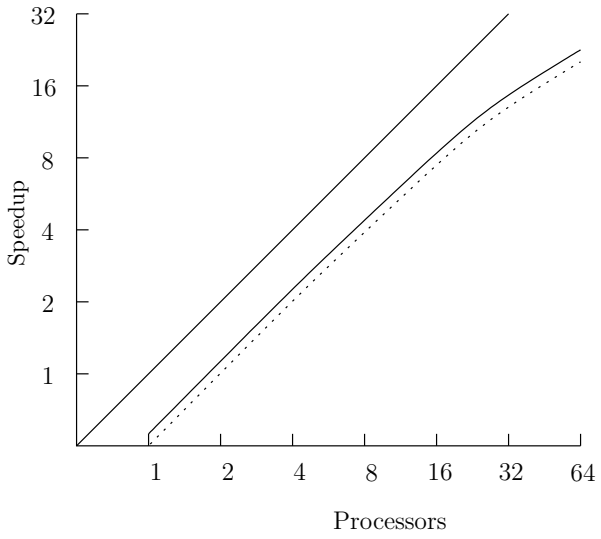

Figure 75: Predicated speedup (dotted line) and actual speedup (solid line) of paralle prefix sums algorithm implemented on an nCUBE 3200 hypercube multicomputer, where $n = 65,536$. (Performance data provided by Bradley K. Seevers) (Fig.6-16, p.175)

(a) Total $\log p$ phases.

(b) Computation time:
During each phase, a processor performs at most two $\oplus$ operations
$\Rightarrow 2\chi \log p$ time for computation.

(c) Communication time: during each phase each processor sends one value to a neighbor and receives a value from that neighbor.
$\Rightarrow 2(\lambda + \beta) \log p$ time for comm.

(d) Total time: $2(\chi + \lambda + \beta) \log p$ for step 2.

g. Total estimated time of the algorithm

* Recall, the time needed by step 1 and 3 are independent of processor topology. The total time for step 1 and 3 is

$$(n/p - 1)\chi + (n/p)\chi$$

i.e.

$$(2n/p - 1)\chi$$

* Because step 2 takes time

$$2(\chi + \lambda + \beta) \log p$$

the total time of all three steps is:

$$(2\frac{n}{p} + 2\log p - 1)\chi + 2(\lambda + \beta) \log p$$

Note: the book has a typo for the above total time.

h. Speedup: estimated sequential time is $(n-1)\chi$. So speedup is:

$$\frac{(n-1)\chi}{(2\frac{n}{p}+2\log p-1)\chi+2(\lambda+\beta)\log p}$$

i. Notes:

$$\lim_{n\to\infty}\frac{(n-1)\chi}{(2\frac{n}{p}+2\log p-1)\chi+2(\lambda+\beta)\log p}\leq\lim_{n\to\infty}\frac{n}{\frac{2n}{p}}=\frac{p}{2}$$

viz. the efficiency of this algorithm cannot exceed 50% (regardless how large the problem size or how small the message latency).

j. Fig.6-16,p.175 compares the estimated speedup with the actual speedup obtained by the algorithm on the nCUBE 3200. $n = 65,536$, $\oplus$ is integer addition, $\chi = 414$ nanoseconds, $\lambda = 363$ microseconds, and $\beta = 4.5$ microseconds.