

PRAM and Other Models

Jingke Li

Portland State University

The RAM Model

RAM = Random Access Machine

A commonly used model for sequential computation.

- ▶ *Machine Model:*
 - ▶ A processor operating under the control of a sequential program.
 - ▶ A memory with M cells (M can be unbounded).
- ▶ *Basic Operations:*
 - ▶ **Read** — the processor reads a datum from an arbitrary location in memory into one of its internal registers.
 - ▶ **Compute** — the processor performs an (arithmetic) operation on data in register(s).
 - ▶ **Write** — the processor writes the content of one register into an arbitrary memory cell.
- ▶ *Cost Model:*

Each basic operation takes one time unit to execute.

RAM Algorithm Examples

- ▶ *Global Sum* — Compute $s = \sum_{i=1}^n a_i$, $1 \leq i \leq n$

```
s ← a1
for i = 2 to n do
    s ← s + ai
endfor
```

Total $3n$ time units:

- n reads (for a_1, \dots, a_n);
- n computes (for s);
- n writes (for s).

- ▶ *Prefix Sums* — Compute $s_i = \sum_{j=1}^i a_j$, $1 \leq i \leq n$

```
s1 ← a1
for i = 2 to n do
    si ← si-1 + ai
endfor
```

Total $3n$ time units:

- n reads (for a_1, \dots, a_n);
- n computes (for s_1, \dots, s_n);
- n writes (for s_1, \dots, s_n).

The Parallel RAM (PRAM) Model

- ▶ *Machine Model:*
 - ▶ A number of identical processors, P_1, P_2, \dots, P_N .
 - ▶ A common memory with M cells, shared by the N processors.
 - ▶ The processors work in a synchronous fashion.
- ▶ *Basic Operations:*
 - ▶ **Read** — (Up to N) processors read simultaneously from memory. Each reads from one memory cell and stores the value in a local register.
 - ▶ **Compute** — (Up to N) processors perform an (arithmetic) operation on their local data in register(s).
 - ▶ **Write** — (Up to N) processors write simultaneously into register into an arbitrary memory cell.

Inter-processor communication is *not* explicitly modeled.

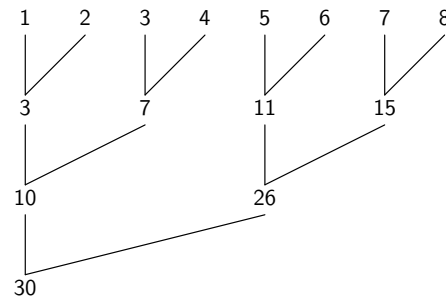
- ▶ *Cost Model:*
 - ▶ The compute operation takes one time unit to execute.
 - ▶ The reads and writes each takes one time unit to execute when there is no conflicts. (Otherwise, more info is needed.)

PRAM Algorithm Examples

Complexity is measured by $time \times space$ (#processors).

► *Global Sum* — final result in s_1

```
spawn( $P_1, P_2, \dots, P_{\lceil n/2 \rceil}$ )
for all  $P_i$  do
   $s_i \leftarrow a_i$ 
  for  $j \leftarrow 1$  to  $\lceil \log n \rceil$  do
    if  $i \% 2^j = 1$  and  $i + 2^{j-1} \leq n$ 
    then
       $s_i \leftarrow s_i + s_{i+2^{j-1}}$ 
    endif
  endfor
endfor
```

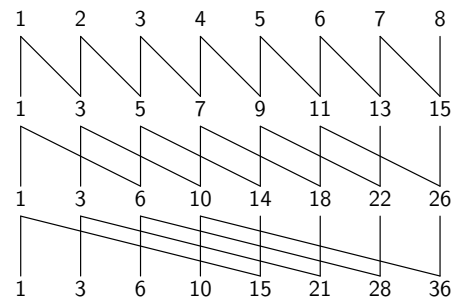


No read or write conflicts, so $Time = \lceil \log n \rceil$, $Space = \lceil \frac{n}{2} \rceil$

PRAM Algorithm Examples (cont.)

► *Prefix Sum*:

```
spawn( $P_1, P_2, \dots, P_n$ )
for all  $P_i$  do
   $s_i \leftarrow a_i$ 
  for  $j \leftarrow 1$  to  $\lceil \log n \rceil$  do
    if  $i - 2^{j-1} > 0$  then
       $s_i \leftarrow s_i + s_{i-2^{j-1}}$ 
    endif
  endfor
endfor
```



There is no write conflict, but there are potential read conflicts.
Assume it's OK for now. Then, $Time = \lceil \log n \rceil$, $Space = n$

Cost-Optimal PRAM Algorithms

How can we compare performance between a parallel algorithm and a sequential algorithm? Or between two parallel algorithms?

The *cost-optimal* metric can help:

- ▶ *Cost* — The cost of a PRAM algorithm is the product of its time complexity and space complexity.
- ▶ *Cost-Optimal* — A cost-optimal PRAM algorithm is one in which the cost is in the same complexity class as the optimal sequential algorithm.

Optimality Analysis

- ▶ *Global Sum (PRAM)*:
 - Time = $\lceil \log n \rceil$, Space = $\lceil \frac{n}{2} \rceil$
 - Cost = $\lceil \log n \rceil \times \lceil \frac{n}{2} \rceil = O(n \log n)$
 - Sequential lower bound = $\Theta(n)$

Conclusion: Not optimal!

Reason: Some processors have idling steps — the total number of operations performed is $n - 1$. *How to improve?*

- ▶ *Prefix Sums (PRAM)*:
 - Time = $\lceil \log n \rceil$, Space = n
 - Cost = $\lceil \log n \rceil \times n = O(n \log n)$
 - Sequential lower bound = $\Theta(n)$

Conclusion: Not optimal!

But there is almost no idle steps — the total number of operations performed is $n \lceil \log n \rceil$! *How to explain?*

Brent's Theorem

Key Idea: Any PRAM algorithm can be simulated by a different PRAM with fewer processors.

Theorem. Let A be a PRAM algorithm with time complexity t . If A performs m total operations, then p processors can simulate the PRAM and execute A in time $t + (m - t)/p$.

Proof:

- ▶ Let s_i be the number of operations performed by A at step i , i.e. i varies from 1 to t .
- ▶ By definition, $\sum_{i=1}^t s_i = m$.
- ▶ Using p processors, we can simulate step i in time $\lceil s_i/p \rceil$.
- ▶ The entire computation can then be done in time

$$\sum_{i=1}^t \lceil \frac{s_i}{p} \rceil \leq \sum_{i=1}^t \frac{s_i - 1 + p}{p} = \sum_{i=1}^t \frac{p}{p} + \sum_{i=1}^t \frac{s_i - 1}{p} = t + \frac{m - t}{p}$$

Applying Brent's Theorem

Global Sum:

- ▶ Choose a new PRAM with $p = \lfloor n / \log n \rfloor$ processors
- ▶ Total number of operations performed is $n - 1$
- ▶ The new time is

$$\lceil \log n \rceil + \frac{n - 1 - \lceil \log n \rceil}{\lfloor n / \log n \rfloor} = \Theta(2 \log n - \frac{\log n}{n} - \frac{\log^2 n}{n}) = \Theta(\log n)$$

- ▶ Cost = $\lfloor n / \log n \rfloor \times \Theta(\log n) = \Theta(n)$ — It is optimal!

Applying Brent's Theorem (cont.)

Prefix Sums (PRAM):

- ▶ Choose a new PRAM with $p = \lfloor n / \log n \rfloor$ processors
- ▶ Total number of operations performed is $n \lceil \log n \rceil$
- ▶ The new time is

$$\lceil \log n \rceil + \frac{n \lceil \log n \rceil - \lceil \log n \rceil}{\lfloor n / \log n \rfloor} = \Theta(\log n + \log^2 n - \frac{\log^2 n}{n}) = \Theta(\log^2 n)$$

- ▶ Cost = $\lfloor n / \log n \rfloor \times \Theta(\log^2 n) = \Theta(n \log n)$ — *Still not optimal!*

Reason: The total number of PRAM operations is higher than that of the optimal sequential algorithm.

“Coarse-Grain” Algorithms

Another approach for improving performance.

Prefix Sums (Coarse-Grain) — use $< n$ processors

- Divide the n values into p sets, each containing $\leq \lceil n/p \rceil$ values.
- The first $p - 1$ processors each uses the optimal sequential algorithm to do a local prefix computation. (*time*: $\lceil n/p \rceil - 1$)
- Then the processors run the PRAM parallel prefix algorithm on the subtotals. (*time*: $\log(p - 1)$)
- Each processor then goes back and updates the local prefix values with the results from the global comp. (*time*: $\lceil n/p \rceil$)

Total Cost: $(2\lceil n/p \rceil + \log(p - 1))p = \Theta(n + p \log p)$

For small p , this algorithm is *cost-optimal*!

PRAM Memory Access Models

- ▶ *Exclusive Read (ER)*
 - For any memory cell, only a single read is allowed at any given time.
- ▶ *Concurrent Read (CR)*
 - Simultaneous multiple reads to the same memory cell are allowed.
- ▶ *Exclusive Write (EW)*
 - For any memory cell, only a single write is allowed at any given time.
- ▶ *Concurrent Write (CW)*
 - Simultaneous multiple writes to the same memory cell are allowed.
 - *Question:* What value gets written?

Concurrent Write Sub Models

- ▶ *Priority CW*
 - The processors are assigned certain priorities; the processor with the highest priority wins.
- ▶ *Common CW*
 - The processors are allowed to write to the same memory cell only if they are attempting to write the *same* value; otherwise a special flag will be raised.
- ▶ *Arbitrary CW*
 - *Any* of the attempting processors can succeed; the selection is according to an algorithm.
- ▶ *Random CW*
 - A processor is chosen by a random process to succeed.
- ▶ *Combining CW*
 - *All* the values from the attempting processors are combined into a single value, which is then stored in the memory cell.
 - Combining operators: sum, product, and, max, min, etc.

Three Specific PRAM Models

These three models are commonly used in parallel algorithm analysis:

- ▶ *EREW PRAM* — The weakest among the three models. It takes $O(\log n)$ time units to spread a value from one processor to n other processors.
- ▶ *CREW PRAM* — A processor can spread a value to n other processors in $O(1)$ time units.
- ▶ *CRCW PRAM* — The strongest model. But CW needs special handling for resolving memory write conflicts.

Concurrent-Write Algorithms

- ▶ *Global Sum (CR+Combing CW PRAM):*

```
spawn( $P_1, P_2, \dots, P_n$ )  
for all  $P_i$  do  
     $s \leftarrow a_i$   
endfor
```

$Time = O(1)$
 $Space = n$

- ▶ *Prefix Sums (CR+Combining CW PRAM):*

```
spawn( $P_{i,j} : 1 \leq i \leq n, 1 \leq j \leq n - i$ )  
for all  $P_{i,j}$  do  
     $t_{i,j} = a_j$ ;  
     $s_i \leftarrow t_{i,j}$   
endfor
```

$Time = O(1)$
 $Space = n^2$

Two Rank Sort Examples

► *Rank Sort (CREW PRAM):*

```
spawn( $P_1, P_2, \dots, P_n$ )
for all  $P_i$  do
   $k = 0$ 
  for  $j = 0$  to  $n$  do
    if  $a[i] > a[j]$  then  $k \leftarrow k + 1$ 
   $b[k] \leftarrow a[i]$ 
endfor
```

$Time = O(n)$
 $Space = n$

► *Rank Sort (CR+Combining CW PRAM):*

```
spawn( $P_1, P_2, \dots, P_n$ )
for all  $P_i$  do
   $k = 0$ ; spawn( $Q_1, Q_2, \dots, Q_n$ )
  for all  $j = 1$  to  $n$  do
    if  $a[i] > a[j]$  then  $k \leftarrow 1$ 
   $b[k] \leftarrow a[i]$ 
endfor
```

$Time = O(1)$
 $Space = n^2$

Critiques on the PRAM Model

- It assumes the processors operate in a fully-synchronous mode.
- It assumes a single shared memory in which each processor can access any cell in unit time.
- It neglects the issue of contention caused by concurrent access to different cells within the same memory module.
- It assumes the interprocessor communication has infinite bandwidth, zero latency, and zero overhead.
- It assumes the number of processors can increase with the problem size.

In conclusion, the PRAM model is good for a gross classification of algorithms, but not very useful for describing realistic algorithms or predicting the performance of algorithms.

Extensions of the PRAM Model

- ▶ *Phase PRAM* — Introduces asynchrony.
- ▶ *Module Parallel Computer* — Divides memory into modules.
- ▶ *Local-Memory PRAM* — Divides memory into local and global.
- ▶ *Memory Hierarchy Model* — Views the memory as a hierarchy.
- ▶ *Delay Model* — Introduces communication delay.

Valiant's BSP Model

BSP = Bulk Synchronous Parallel

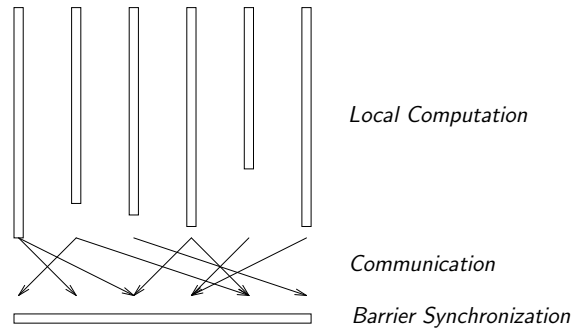
The BSP model is developed to model message-passing multicomputers. It consists of three components:

- ▶ A group of p *processors* each with local memory.
- ▶ An interconnection network for *point-to-point communication* between the processors.
- ▶ A *mechanism for synchronizing* all the processors at defined intervals.

Supersteps

In BSP, a computation consists of a sequence of *supersteps*. Each superstep is further subdivided into three ordered phases:

- ▶ *Local Computation* — each processor performs computation using only data stored in the local memory.
- ▶ *Communication* — processors send/receive messages to each other.
- ▶ *Barrier Synchronization* — this global synchronization waits for all of the communication actions to complete.



BSP Architectural Parameters

BSP model has three architectural parameters:

p — the number of processors

l — the latency of a barrier synchronization

g — the communication overhead

They can be computed for a particular real target machine.

With these parameters, the time of a superstep can be expressed by

$$T_{\text{superstep}} = w_{\text{max}} + gh_{\text{max}} + l$$

where w_{max} is the maximum time caused by local operations by any processor and h_{max} is the maximum number of messages being sent or received by a processor.

BSP Model Properties

- ▶ *High-level* — architecture-independent.
- ▶ *Simple to use* — BSP programs look much the same as sequential programs.
- ▶ *Can predict performance* – With a small set of parameters used to describe a given architecture, the performance of a BSP program on the target architecture is predictable.

A BSP Program

The following function calculates the partial sums of p integers stored on p processors. (The code is written in C with Oxford BSP library.)

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