BITS Pilani
Pilani Campus

PRAM Model

K Hari Babu (Slides adapted from Prof. Shan's)
Department of Computer Science & Information Systems

RAM Model

Why Machine Models?

- What is a machine model?
  - An abstraction that describes the operation of a machine
  - Associates a value (cost) with each machine operation
- Why do we need models?
  - Makes it easier to analyze and develop algorithms
  - Hides the machine implementation details so that general results that apply to a broad class of machines are obtainable
  - Analyzes the achievable complexity (time, space, etc.) bounds
  - Analyzes maximum parallelism
  - Conversely, models are directly related to algorithms.

RAM (Random Access Machine) Model

- Memory consists of infinite array (memory cells).
- Instructions executed sequentially, one at a time
- All instructions take unit time:
  - Load/store
  - Arithmetic
  - Logic
- Running time of an algorithm: the number of instructions executed
- Memory requirement: the number of memory cells used in the algorithm

RAM (Random Access Machine) Model

- The RAM model is the base of algorithm analysis for sequential algorithms although it is not perfect:
  - Memory is not infinite
  - Not all memory accesses take the same time
  - Not all arithmetic operations take the same time
  - Instruction pipelining is not taken into consideration
- The RAM model (with asymptotic analysis) often gives relatively realistic results

PRAM (Parallel RAM)

- An unbounded collection of processors
- Each process has an infinite number of registers
- An unbounded collection of shared memory cells
- All processors can access all memory cells in unit time (when there is no memory conflict)
- All processors execute PRAM instructions synchronously (some processors may be idle)
- Each PRAM instruction executes in a 3-phase cycle:
  - Read from a share memory cell (if needed)
  - Computation
  - Write to a share memory cell (if needed)
- PRAM is an abstract machine model
  - multiple execution units, single control and clock, local (private) memory units, and global (shared) memory.

PRAM (Parallel RAM)

BITS Pilani

- The only way processors exchange data is through the shared memory.
- Individual processor
  - Time: number of instructions executed
  - Space: number of memory cells accessed
- Parallel time complexity:
  - the number of synchronous steps in the algorithm
  - Time taken by the longest running process
- Space complexity:
  - the number of shared memory cells
- Parallelism:
  - the number of processors used

PRAM Model - variants

BITS Pilani

- Shared Memory Model:
  - Can processes – *running in different processors* –access locations in global shared memory concurrently?
    - If yes, is concurrent access allowed for ***read*** as well as for ***write*** operations?
    - If yes, how are concurrent ***write***s governed?

- Variants of PRAM are defined on the basis of answers to these questions.

PRAM Model - variants

BITS Pilani

- Variants based on concurrent operations:
- Exclusive Read Exclusive Write (EREW)
  - Shared memory locations cannot be read / written concurrently.
- Concurrent Read Exclusive Write (CREW)
  - Shared memory locations can be read but not written concurrently.
- Concurrent Read Concurrent Write (CRCW)
  - Shared memory locations can be read / written concurrently.

PRAM variants: Interpretation at software level

BITS Pilani

- EREW model
  - Contract between the processor and the program:
    - processor does not support concurrent memory operations (read or write);
    - program must devise its own mechanism for ensuring memory is accessed serially.
- CREW model
  - Contract between the processor and the program:
    - processor supports concurrent reads;
    - program must devise its own mechanism for concurrent writes to get processed serially.

PRAM Model: CRCW - variants

BITS Pilani

- Variants of CRCW based on handling of write conflicts:
  - Shared memory locations can be read / written concurrently:
  - Common: All concurrent writes must write the same value
  - Arbitrary: Concurrent writes may write different values but which of those values gets stored is arbitrary.
  - Priority: Concurrent writes may write different values and the value that gets written stored is decided by priority
    - priority is the id of the writer i.e. the processor

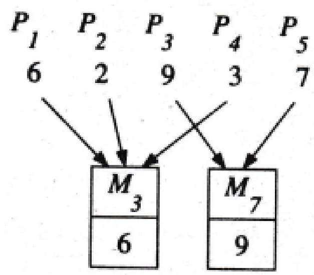
Relative Strengths of Models

BITS Pilani

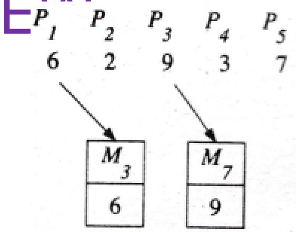
- Model A is computationally stronger than model B if and only if any algorithm written in B will run unchanged in A. We can prove,
  - EREW <= CREW <= CRCW (common) <= CRCW (arbitrary) <= CRCW (Priority)
  - Weakest model ………………………………. Strongest Model
  - Most realistic ………………………………….Least Realistic
    - CREW PRAM can execute any EREW PRAM algorithm in the same amount of time. Concurrent read facility is not used.
    - CRCW PRAM can execute any EREW PRAM algorithm
    - Priority PRAM model is the strongest
  - An algorithm to solve a problem on the EREW PRAM model can have higher time complexity than an algorithm on Priority model

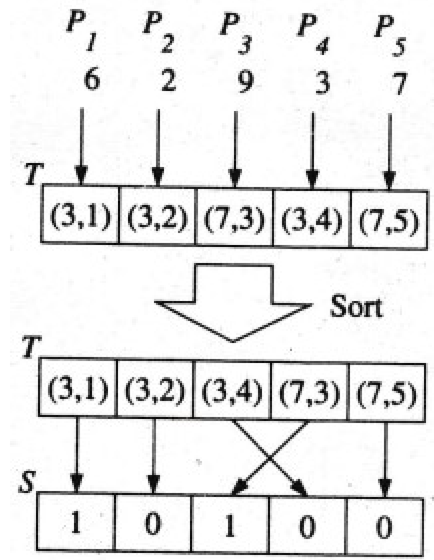
Priority PRAM vs EREW PRAM

- A p-processor EREW PRAM can sort a p-element array stored in global memory in Θ(log p) time
- Theorem: A p-processor Priority PRAM can be simulated by a p-processor EREW PRAM with the time complexity increased by a factor of Θ(log p)
  - Handling of concurrent write in Priority PRAM



  - Simulating concurrent write on EREW





Priority PRAM vs EREW PRAM

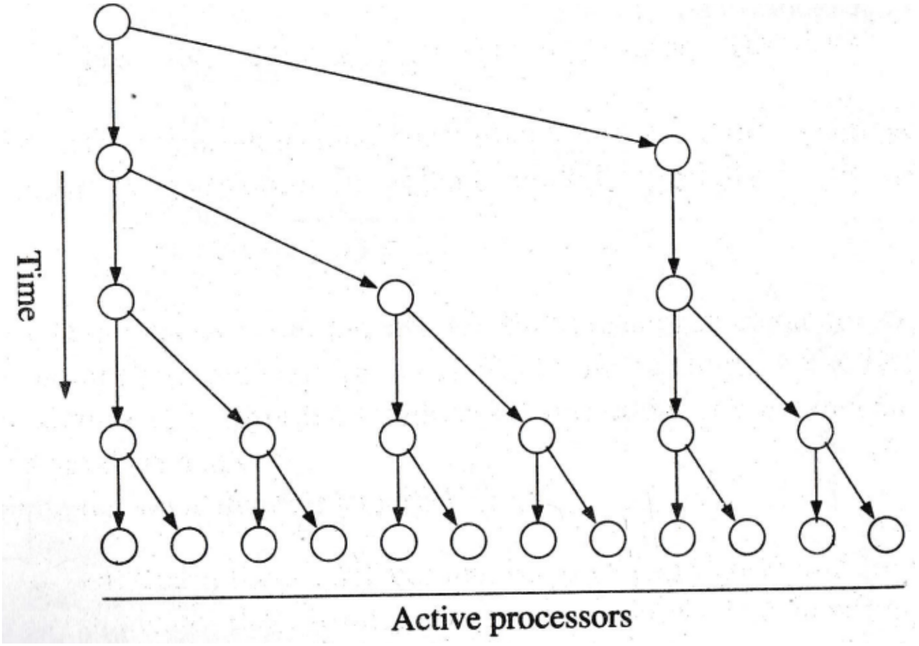
- A p-processor EREW PRAM can sort a p-element array stored in global memory in $\Theta(\log p)$ time
- Theorem: A p-processor Priority PRAM can be simulated by a p-processor EREW PRAM with the time complexity increased by a factor of $\Theta(\log p)$
  - Proof:
  - All write operations at a time by the Priority PRAM are stored in an array T of pairs (M[j], Pi).
  - Sort T in lexicographic order
    - $\Theta(\log p)$ time. (parallel sorting with p processors)
  - Find the highest priority processor writing into a specific location
    - $\Theta(1)$ time

PRAM: Algorithmic Model

- Time Complexity vs. Cost
  - Cost = Time Complexity * #processors
- Since a PRAM algorithm begins only with a single processor active, PRAM algorithms have two phases
- 2-phase model:
  - Activation Phase and Computation Phase
  - In the first phase sufficient number of processors are activated (spawned)
  - In the second phase these activated processors perform the computation in parallel

Activation Phase

- Exactly ceil(logp) processor steps are necessary and sufficient to change from 1 active processor to p active processors
  - Because the number of active processors can double by executing a single instruction



All logarithms are base 2 unless stated

Computation Phase

- After the necessary processes are spawned, each processor executes same segment of code in parallel

```
for all <processor list>
 do
  <statement list>
endfor
```

- This is essentially similar to programming on SIMD architectures:
  - also referred to as the SPMD model

PRAM: Algorithmic Model - Computation

BITS Pilani

- If all processors execute the same set of statements, how do you obtain variations?
  - e.g. different data to be processed in each processor
  - e.g. different choices to be made in each processor
- Use the processor id (rank) for data access or control:

```
for all Pi where i = 0 to p-1
do
        x = M[i];
        if (i%2 == 0)
                { M[i] = x * x; }
        else
                {  M[i] = x * x * x; }
endfor
```

PRAM Algorithms – Example 1

BITS Pilani

- Algorithm for Adding Two Matrices [Mat-Add] :
  - How many processors?
  - Consider the two algorithms below.

- Algorithm 1 [for Mat-Add]:

```
// Input: Matrices A and B of size m*n
for all Pi,j in i = 1 to m, j = 1 to n
{C[i,j] = A[i,j] + B[i,j]  }
```

  - m*n processors, O(1) time, Cost = O(m*n)

- Algorithm 2 [for Mat-Add]:

```
for all Pi in i = 1 to m {
      for j = 1 to n { C[i,j] = A[i,j] + B[i,j]  }}
```

  - m processors, O(n) time, Cost = O(m*n)

PRAM Algorithms – Example 2

BITS Pilani

- Algorithm for Vector Product [Vec-Pro]:
  - How many processors?
- Algorithm [for Vec-Pro]:
  - // Input: Vectors A and B of size n

```
for all Pi in i = 1 to n
C[i] = A[i] * B[i]
```

  - // Compute sum C[i] for all i
  - // How do you do this in parallel ?

PRAM Algorithms – Example 3

BITS Pilani

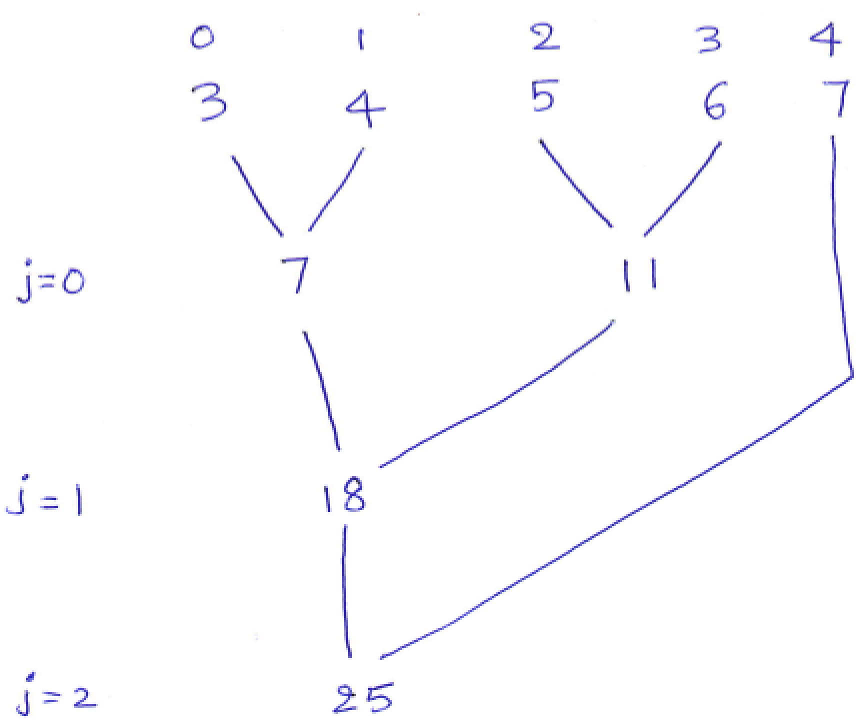
- Recall the second phase of the algorithm for Vector Product:
  - A list (i.e. an array) of values has to summed up in one value
- How many parallel tasks are possible?
  - In the first step – *given* **n** *values to be added* :
    - **n/2** additions can be performed in parallel
    - resulting in n/2 values to be added
    - which in turn is the same as the original problem

Algorithm Design – Reduction: Example

BITS Pilani

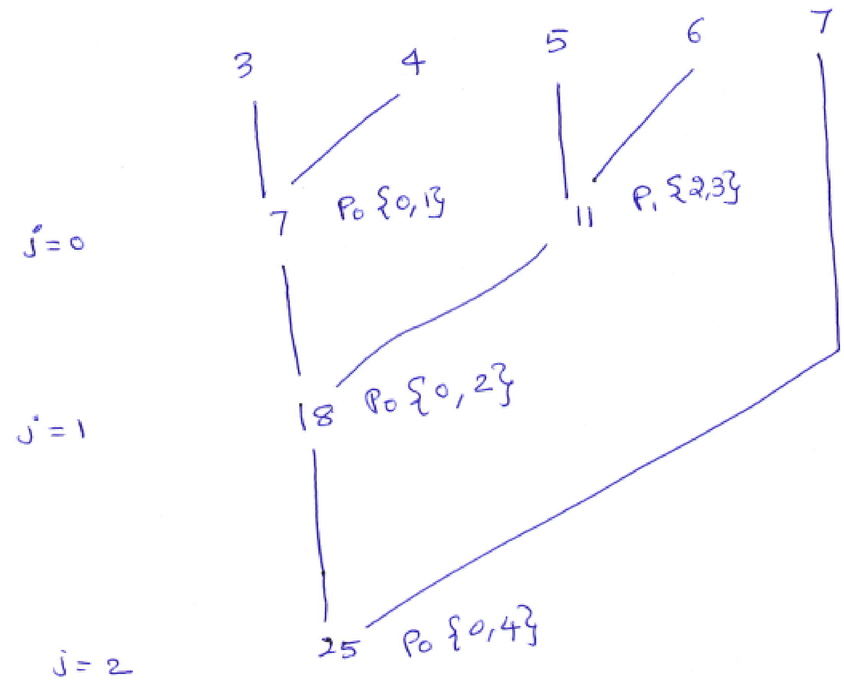
- SUM (EREW PRAM)
  - Precondition: List A[0..n-1] in global memory
  - Postcondition: sum A[0] in global memory
  - Global variables, A, n, and j
- Parallel reductions
  - Parallel summation is an example of divide and conquer as well as parallel reduction
  - Parallel reduction can be represented by a binary tree
    - At a time two values are added
  - A group of n values can be added in CEIL(log n) steps

```
for all Pi where 0 <= i <= FLOOR(n/2)-1 {
    for j = 0 to CEIL(log(n))-1 {
    …
}}
```



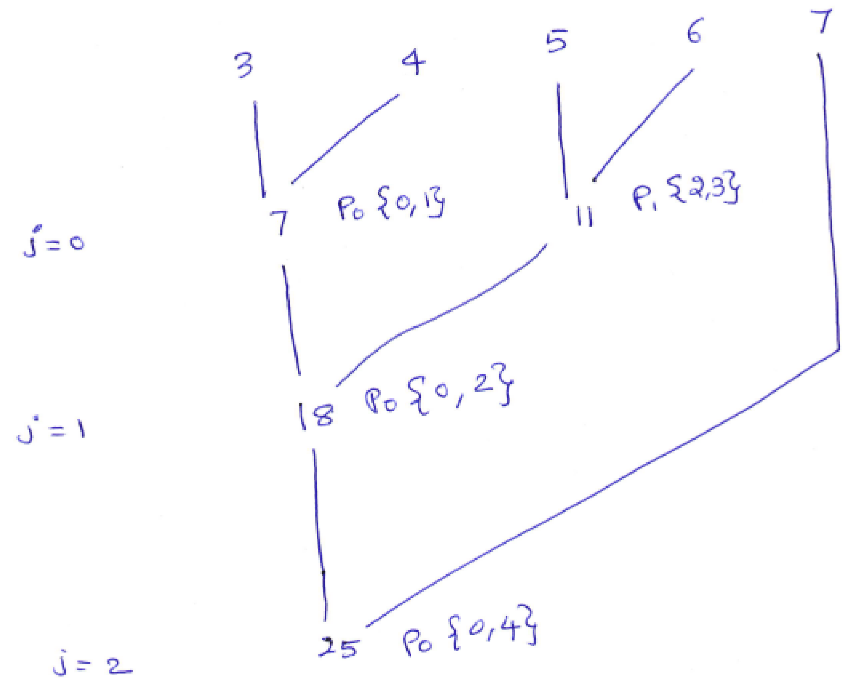
Algorithm Design – Reduction: Example

- How many processes are required?
  - n elements
  - Sum up two values at a time
    - No of processors=FLOOR(n)
- How do we identify elements per process and per step?
  - What should be the relationship between i (rank) and j?
    - A[2i]=A[2i] + A[2i + 2^j]
    - 2^j is the distance between self and (processor holding) other data



Algorithm Design – Reduction: Example

- Are all processors participating in each step?
  - No
  - How do we decide which process should participate?
  - i module 2^j ==0
    - j=0, {0,0}
    - j=1, {0,1}
    - j=2,{0,1}



Algorithm Design – Reduction: Example

BITS Pilani

- Notice that P1 is accessing A[6] which is invalid
  - Boundary condition is
    - 2i+2^j < n

```
spawn(P0, P1, … Pk) where k=FLOOR(n/2)-1
for all Pi where 0 <= i <= FLOOR(n/2)-1 {
   for j = 0 to CEIL(log(n))-1 {
      if (i mod 2^j = 0)  /* incomplete */ then
         A[2*i] = A[2*i] + A[2*i + 2^j]
  }}
```

```
int A[]={3,4,5,6,7};
//we are using FLOOR(n/2) processes
//NO of processes=5/2=2; i=0,1
//P0,P1
```

	P0	P1
j=0	A[0]=a[0]+A[0+2^0]	A[2]=A[2]+A[2+2^0]
	A[0]=A[0]+A[1]	A[2]=A[2]+A[3]
j=1	A[0]=a[0]+A[0+2^1]	A[2]=A[2]+A[2+2^1]
	A[0]=A[0]+A[2]	A[2]=A[2]+A[4]
j=2	A[0]=a[0]+A[0+2^2]	A[2]=A[2]+A[2+2^2]
	A[0]=A[0]+A[4]	A[2]=A[2]+A[6]

Algorithm Design – Reduction: Example

BITS Pilani

- Are there any concurrent accesses (R/W) in this algorithm?
  - EREW

```
begin
spawn (P0, P1, … Pk) where k=FLOOR(n/2)-1
for all Pi where 0 <= i <= FLOOR(n/2)-1 {
   for j = 0 to CEIL(log(n))-1 {
      if (i mod 2^j = 0) and (2*i + 2^j  < n) then
         A[2*i] = A[2*i] + A[2*i + 2^j]
   }}
end
```

```
int A[]={3,4,5,6,7};
//we are using FLOOR(n/2) processes
//NO of processes=5/2=2; i=0,1
//P0,P1
```

	P0	P1
j=0	A[0]=a[0]+A[0+2^0]	A[2]=A[2]+A[2+2^0]
	A[0]=A[0]+A[1]	A[2]=A[2]+A[3]
j=1	A[0]=a[0]+A[0+2^1]	
	A[0]=A[0]+A[2]	
j=2	A[0]=a[0]+A[0+2^2]	
	A[0]=A[0]+A[4]	

Algorithm Design – Reduction: Example

BITS Pilani

- What is the time complexity?
  - Spawn routine takes CEIL(log (FLOOR(n/2)) steps
  - For loop executes CEIL(log n) steps
    - Each iteration has constant time complexity
  - Overall time complexity is $\Theta(\log n)$
    - Given FLOOR(n/2) processors

```
begin
spawn (P0, P1, … Pk) where k=FLOOR(n/2)-1
for all Pi where 0 <= i <= FLOOR(n/2)-1 {
   for j = 0 to CEIL(log(n))-1 {
      if (i mod 2^j = 0) and (2*i + 2^j  < n) then
         A[2*i] = A[2*i] + A[2*i + 2^j]
   }}
end
```

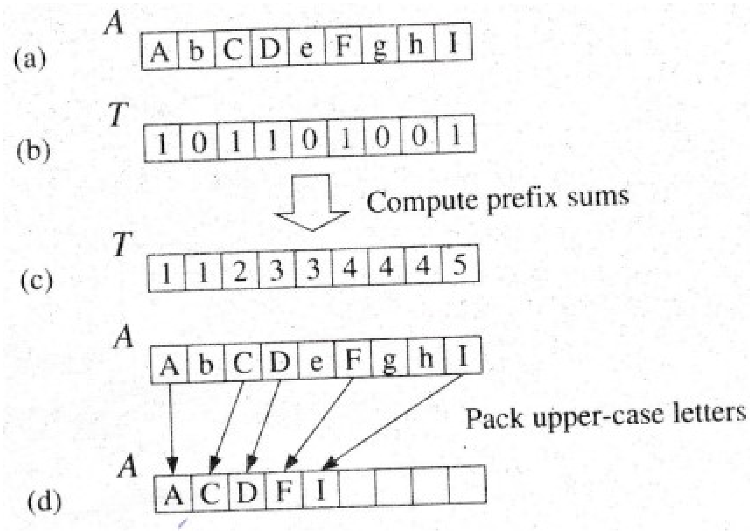
Algorithm Design – Speedup and Efficiency

- Speedup:
  - $S = T_{seq} / T_{par}$ (generic)
  - $S(p) = T(n,1) / T(n,p)$
    - Ideal speedup should be p.
- Example: <u>Speedup of *Summation by Reduction*</u>:
  - $S(n/2) = (n/2) / \log(n)$
    - This is less than ideal speedup!
- Efficiency (definition):
  - $E(n,p) = S(p)/p$
    - Ideal efficiency is 1.
- For our example:
  - $E(n,n/2) = (n/2) / \log(n)*(n/2) = 1/\log(n)$

Algorithm Design: Prefix Sum Example

BITS Pilani

- Given a set of n values a1, a2, a3, …. an, and an associative operator ⊕ the prefix sums problem is to compute n quantities:
  - a_1
  - $a_1 \oplus a_2$
  - …..
  - $a_1 \oplus a_2 \oplus a_3 \oplus \ldots \oplus a_n$
  - Operator
- Prefix sums are also called parallel prefixes or scans
  - Have many applications for instance packing elements



(a) Array A contains both upper and lower-case letters
(b) Array T contains a 1 for upper, 0 for lower-case
(c) Array T after prefix-sums
(d) For each element of A containing upper-case letter, the corresponding element of T is the element's index in the packet array

Algorithm Design: Prefix Sum Example

BITS Pilani

- PREFIX_SUM (EREW PRAM)
  - Precondition: List A[0..n-1] in global memory
  - Postcondition: Each element A[i] contains its prefix sum
  - Global variables, A, n, and j
- Does it have concurrent accesses?
  - Yes but only reads
- What is the time complexity?
  - $\Theta(\log n)\ given\ n-1\ processors$

```
begin
spawn (P0, P1, … Pk) where k=n-1
for all Pi where 0 <= i <= n-1 {
    for j = 0 to CEIL(log(n))-1 {
        if (i - 2^j >= 0) then
            A[i] = A[i] + A[i - 2^j]
    }}
end
```

Algorithm Design – Prefix Sum Example

BITS Pilani

- Speedup:
  - $S = T_{seq} / T_{par}$ (generic)
  - $S(p) = T(n,1) / T(n,p)$
    - Ideal speedup should be p.
- Example: Speedup of prefix *Summation by Reduction*:
  - $S(n/2) = (n-1) / \log(n)$
    - This is less than ideal speedup!
- Efficiency (definition):
  - $E(n,p) = S(p)/p$
    - Ideal efficiency is 1.
- For our prefix sum Example :
  - $E(n,n-1) = (n-1) / \log(n)*(n-1) = 1 / \log(n)$

Parallel Reduction - Template

BITS Pilani

- Template REDUCE
- Precondition: Inputs, G, in global memory
- Postcondition: Result in G[f(0)]
- Global variables: n and j, apart from G
- begin

  spawn (P0, P1, … Pk) where k=floor(n/2)-1

  for all Pi where 0 <= i <= floor(n/2)-1 {

  for j = 0 to ceil(log(n))-1 {

  if <u>(i mod 2^j = 0)</u> and (<u>g(i) < n</u>) then

  G[f(i)] = G[f(i)] <u>BOP</u> G[g(i)]

  }}
- end

BOP is any binary operation

index of own data

index of other data

Parallel Reduction - Template

- Reduction provides a template for
  - parallel execution of
    - *any associative binary operation*
  - extended over a list of values
- Example Instances:
  - Maximum of n values
    - BOP is max
  - Sum of n matrices
    - BOP is matrix addition
  - MergeSort
    - BOP is merging two sorted lists

References

- Chapter 2 from M.J. Quinn, *Parallel Computing : Theory & Practice*, McGraw Hill Inc. 2nd Edition 2002 innovate achieve lead

BITS Pilani
Pilani Campus

Thank You