LAU
الجامعة اللبنانية الأميركية
Lebanese American University

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Shared-Memory Programming: Parallel Prefix, Matrix Multiplication

Instructor: Haidar M. Harmanani

Spring 2020

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Parallel Prefix

Consider A Simple Task …

- Adding a sequence of numbers A[0],…,A[n-1]
- Standard way to express it

```
sum = 0;
for (i=0; i<n; i++) {
   sum += A[i];
}
```

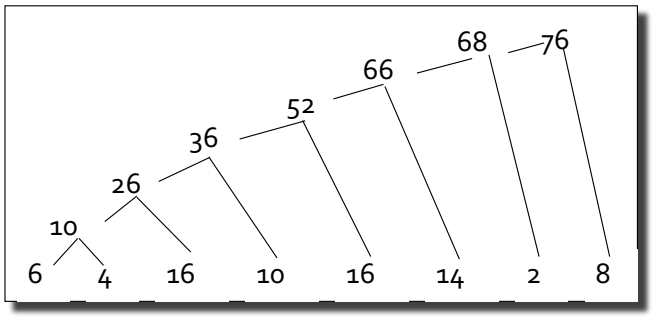
- Semantics require: (…((sum+A[0])+A[1])+…)+A[n-1]
  - That is, sequential
- Can it be executed in parallel?

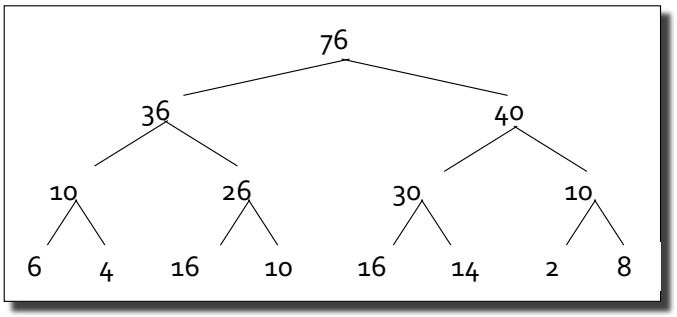
Parallel Summation

- To sum a sequence in parallel
  - add pairs of values producing 1st level results,
  - add pairs of 1st level results producing 2nd level results,
  - sum pairs of 2nd level results …
- That is, (…((A[0]+A[1]) + (A[2]+A[3])) + … + (A[n-2]+A[n-1]))…)

Express the Two Formulations

- Graphic representation makes difference clear





  - Same number of operations; different order

The Dream …

- Since 70s (Illiac IV days) the dream has been to compile sequential programs into parallel object code
  - Three decades of continual, well-funded research by smart people implies it's hopeless
    - For a tight loop summing numbers, its doable
    - For other computations it has proved extremely challenging to generate parallel code, even with pragmas or other assistance from programmers

What's the Problem?

- It's not likely a compiler will produce parallel code from a C specification any time soon…
- Fact
  - For most computations, a "best" sequential solution (practically, not theoretically) and a "best" parallel solution are usually fundamentally different …
    - Different solution paradigms imply computations are not "simply" related
    - Compiler transformations generally preserve the solution paradigm

Therefore… the programmer must discover the || solution

A Related Computation

- Consider computing the prefix sums

```
for (i=1; i<n; i++) {
   A[i] += A[i-1];
}
```

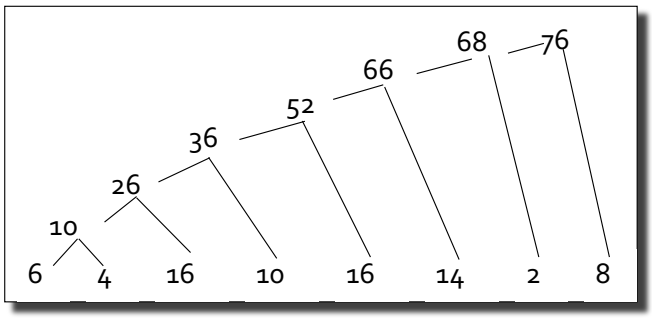
A[i] is the sum of the first i + 1 elements

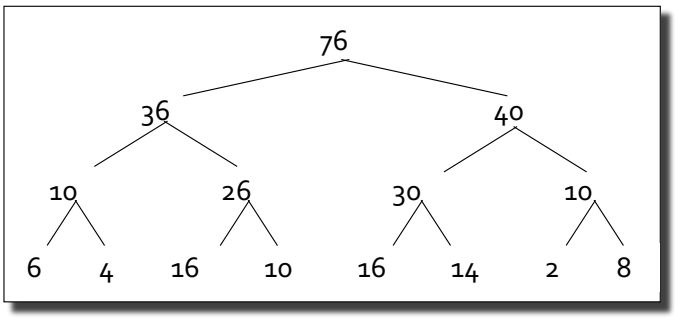
- Semantics …
  - A[0] is unchanged
  - A[1] = A[1] + A[0]
  - A[2] = A[2] + (A[1] + A[0])
  - …
  - A[n-1] = A[n-1] + (A[n-2] + (… (A[1] + A[0]) …)

What advantage can ||ism give?

Comparison of Paradigms

- The sequential solution computes the prefixes … the parallel solution computes only the last





- Or does it?

Parallel Prefix

- Assume that *n* operands are input to the leaves of the complete binary tree
- Algorithm
  - Compute the grand total at the root by pair-wise sum;
  - On completion, root receives a 0 from its (nonexistent) parent;
  - All non-leaf nodes receive a value from their parent, relay that value to their left child, and send their right child the sum of the parent's value and their left child's value that was computed on the way up;
  - Leaves add the prefixes from computed above values and saved input.
- Time: Phase1 : log n - 1, Phase2: 2 , Phase 3: 2, Phase 4: log n - 1

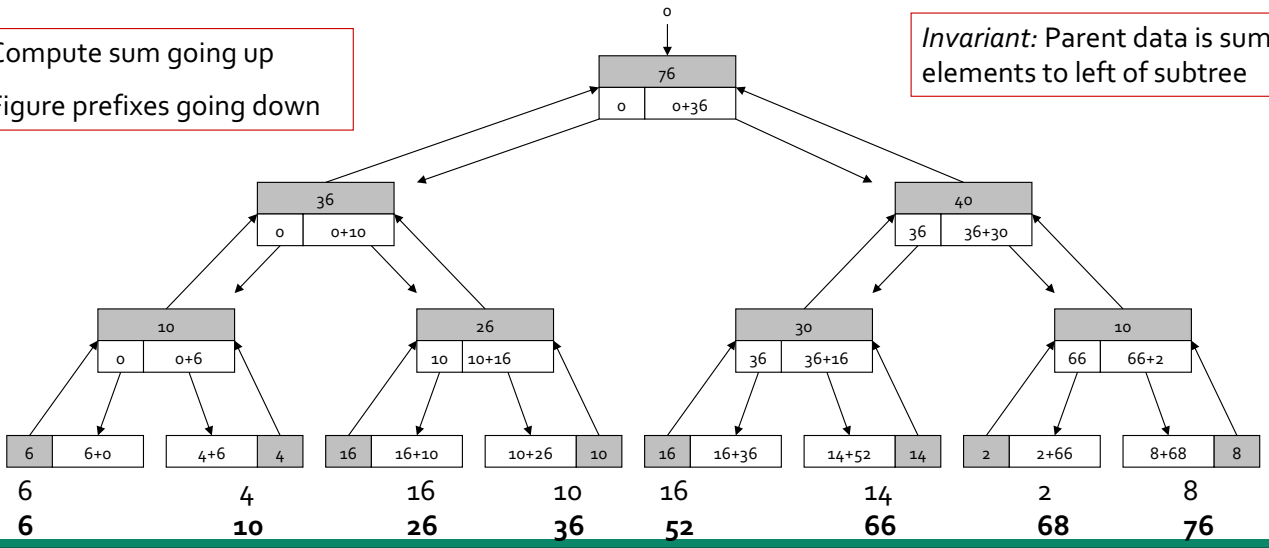
non-leaf nodes receive a value from their parent, relay that value to their left child, and send their right child the sum of the parent's value and their left child's value that was computed on the way up;

Parallel Prefix Algorithm

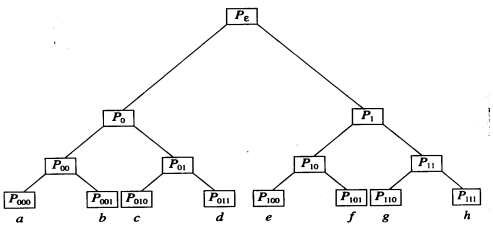
Compute sum going up

Figure prefixes going down

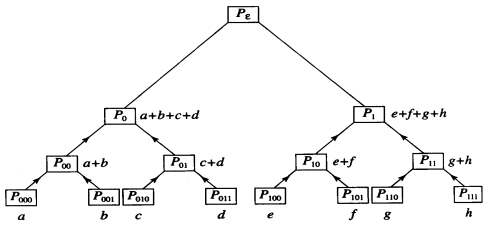
Invariant: Parent data is sum of elements to left of subtree



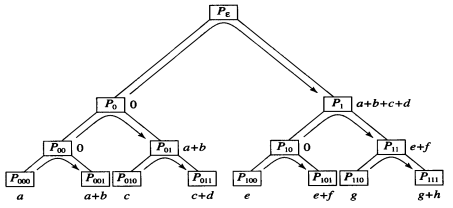
Parallel Prefix : On the complete binary tree



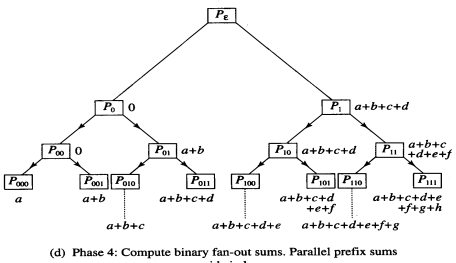
(a) Phase 1: Input the numbers in the leaves of PT_{2n-1}.



(b) Compute binary fan-in sums.



(c) Phase 2: For leaves, add sums in siblings and leave resulting sum in right child sibling. Phase 3: For non-root, non-leaf, left children, transfer binary fan-in sum to sibling then zero out own sum.



(d) Phase 4: Compute binary fan-out sums. Parallel prefix sums now reside in leaves.

Fundamental Tool of || Pgmming

- Original research on parallel prefix algorithm published by

  R. E. Ladner and M. J. Fischer
  Parallel Prefix Computation
  Journal of the ACM 27(4):831-838, 1980

The Ladner-Fischer algorithm requires *2log n* time, twice as much as simple tournament global sum, not linear time

Applies to a wide class of operations

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Matrix Multiplication

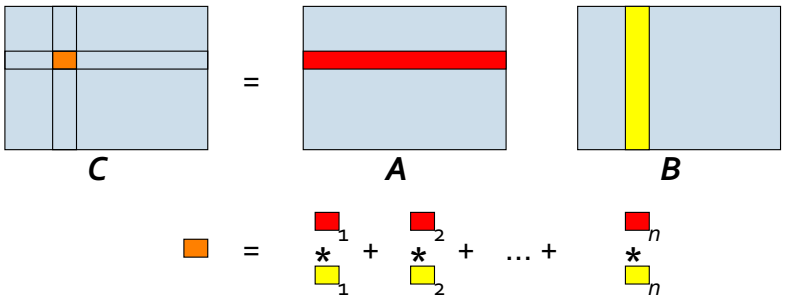
Matrix Product: || Poster Algorithm

- Matrix multiplication is most studied parallel algorithm (analogous to sequential sorting)
- Many solutions known
  - Illustrate a variety of complications
  - Demonstrate great solutions
- Our goal: explore variety of issues
  - Amount of concurrency
  - Data placement
  - Granularity

Exceptional by requiring O(n^3) ops on O(n^2) data

Recall the computation…

- Matrix multiplication of (square n x n) matrices ***A*** and ***B*** producing n x n result ***C*** where $C_{rs} = \sum_{1 \le k \le n} A_{rk} * B_{ks}$

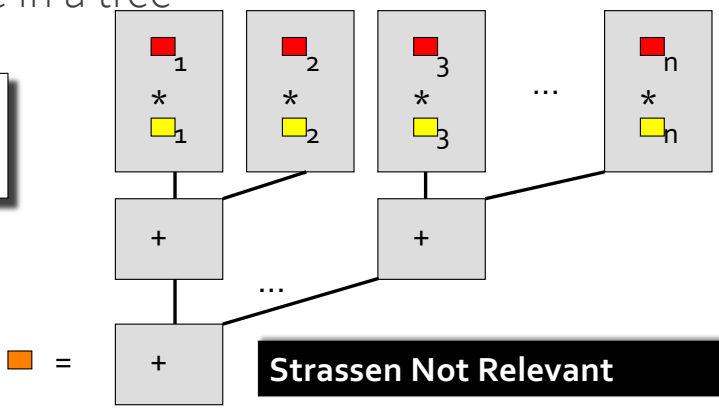


Extreme Matrix Multiplication

- The multiplications are independent (do in any order) and the adds can be done in a tree

O(*n*) processors for each result element implies O(n^3) total

Time: O(*log n*)



Strassen Not Relevant

O(*log n*) MM in the real world …

Good properties

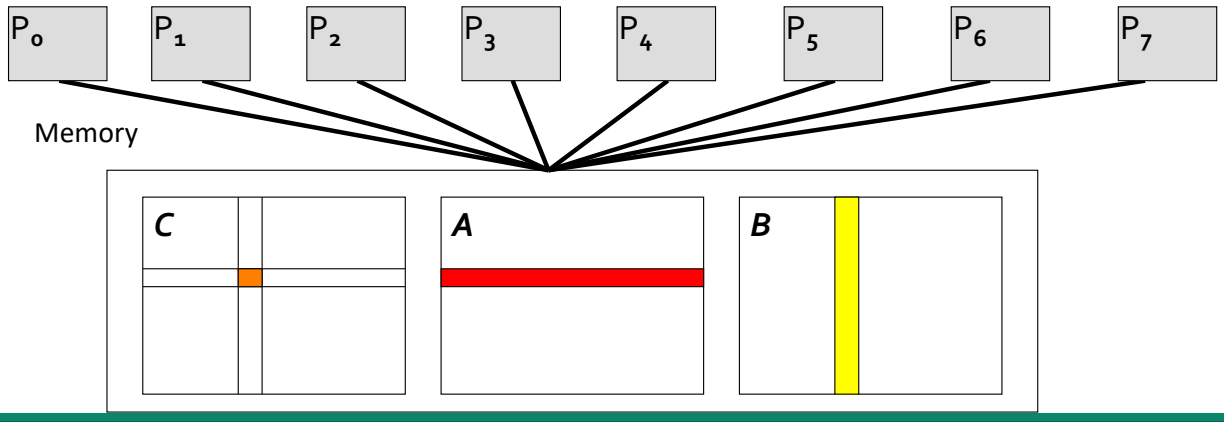
- Extremely parallel … shows limit of concurrency
- Very fast -- $log_2 n$ is a good bound … faster?

Bad properties

- Ignores memory structure and reference collisions
- Ignores data motion and communication costs
- Under-uses processors -- half of the processors do only 1 operation

Where is the data?

- Data references collisions and communication costs are important to final result … need a model … can generalize the standard RAM to get PRAM



Parallel Random Access Machine

- Any number of processors
- Any processor can reference any memory in "unit time"
- Resolve Memory Collisions
  - Read Collisions -- simultaneous reads to location are OK
  - Write Collisions -- simultaneous writes to loc need a rule:
    - Allowed, but must all write the same value
    - Allowed, but value from highest indexed processor wins
    - Allowed, but a random value wins
    - Prohibited

Caution: The PRAM is *not* a model we advocate

PRAM says O(log n) MM is good

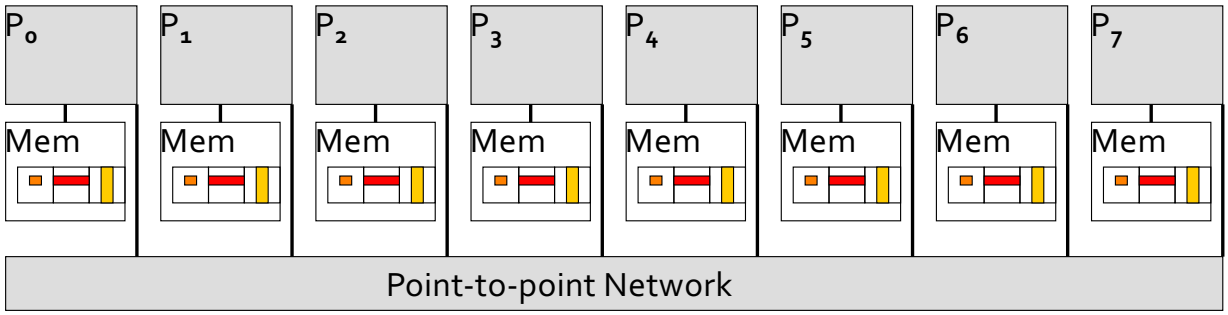
- PRAM allows any # processors => O(n^3) processors are OK
- ***A*** and ***B*** matrices are read simultaneously, but that's OK
- ***C*** is written simultaneously, but no location is written by more than 1 processor => OK

PRAM model implies O(*log n*) algorithm is best … but in real world, we suspect not

We return to this point later

Where else could data be?

- Local memories of separate processors …

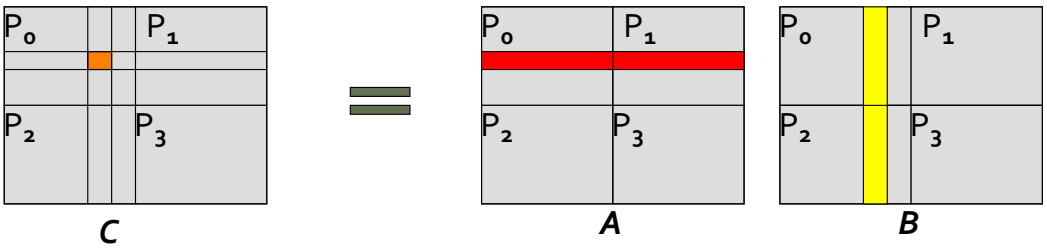


- Each processor could compute block of C
  - Avoid keeping multiple copies of A and B

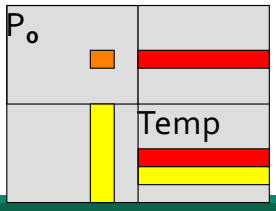
Architecture common for servers

Data Motion

- Getting rows and columns to processors

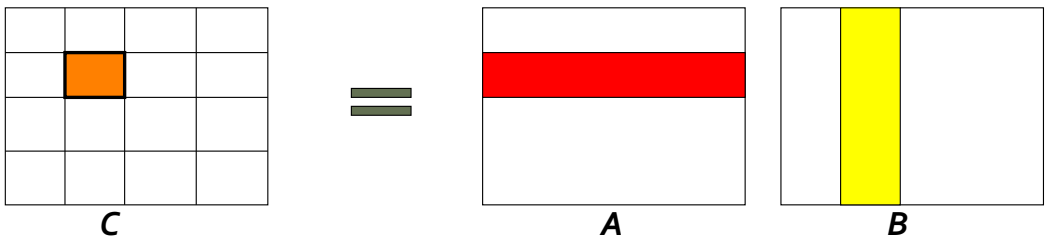


  - Allocate matrices in blocks
  - Ship only portion being used



Blocking Improves Locality

- Compute a b x b block of the result



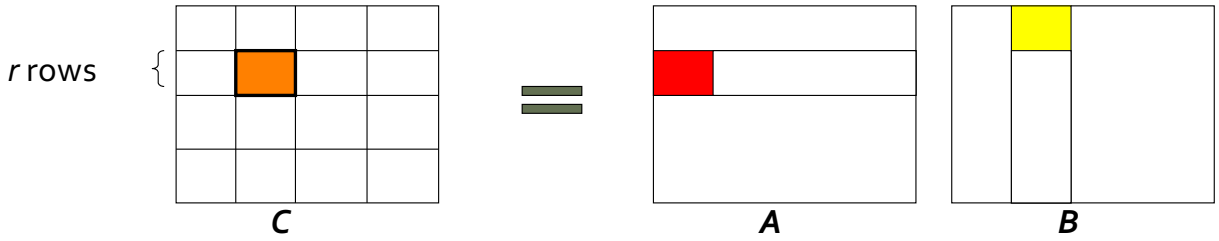
- Advantages
  - Reuse of rows, columns = caching effect
  - Larger blocks of local computation = hi locality

Caching in Parallel Computers

- Blocking = caching … why not automatic?
  - Blocking improves locality, but it is generally a manual optimization in sequential computation
  - Caching exploits two forms of locality
    - Temporal locality -- refs clustered in time
    - Spatial locality -- refs clustered by address
- When multiple threads touch the data, global reference sequence may not exhibit clustering features typical of one thread -- thrashing

Sweeter Blocking

- It's possible to do even better blocking …



- Completely use the cached values before reloading

Best MM Algorithm?

- We haven't decided on a good MM solution
- A variety of factors have emerged
  - A processor's connection to memory, unknown
  - Number of processors available, unknown
  - Locality--always important in computing--
    - Using caching is complicated by multiple threads
    - Contrary to high levels of parallelism
- Conclusion: Need a better understanding of the constraints of parallelism

Next week, architectural details + model of ||ism