CSEP 524 – Parallel Computation
University of Washington

Lecture 4: Parallel Algorithms and Abstractions

Michael Ringenburg
Spring 2015

Announcement

- The class on Tuesday, May 19 has been rescheduled to Thursday, May 21.
  - Same time (6:30pm), same place (CSE 305, MS building 99)
- Next class: Guest lecture from Brad Chamberlain, Chapel lead.
  - Should be the most interesting lecture of the class – please don't miss it!
- No homework due next week.
  - Work on project!
  - May turn in Problem 3 of last homework next week.

Bitonic Sort: Setup

Let's walk through Figure 4.7 in text – should help with HW:

```
int t;                  Number of threads - 2^m
rec L[n];               Records to be sorted
int size = n/t;         Local size - assume t divides m
key BufK[t][size];      Buffer for passing data to partners
bool free'[t] = false; ready'[t];  synchronization variables
forall(index in(0..t-1) {
  int i,d,p; bool stall;
  rec LocL[size] = localize(L[]); Local piece of L
  rec inputCopy[size];            Simplifes copy at end
  key Kn[size]=localize(BufK[]);  Local piece of BufK
  key K[size];
  for (i=0; i<size; i++) {
    K[i].x=LocL[i].x;    Copy string to sort into work buffer
    K[i].home=localToGlobal(LocL,I,0); Remember global index
  }
```

Bitonic Sort: Data Movement

Let's walk through Figure 4.7 in text – should help with HW:

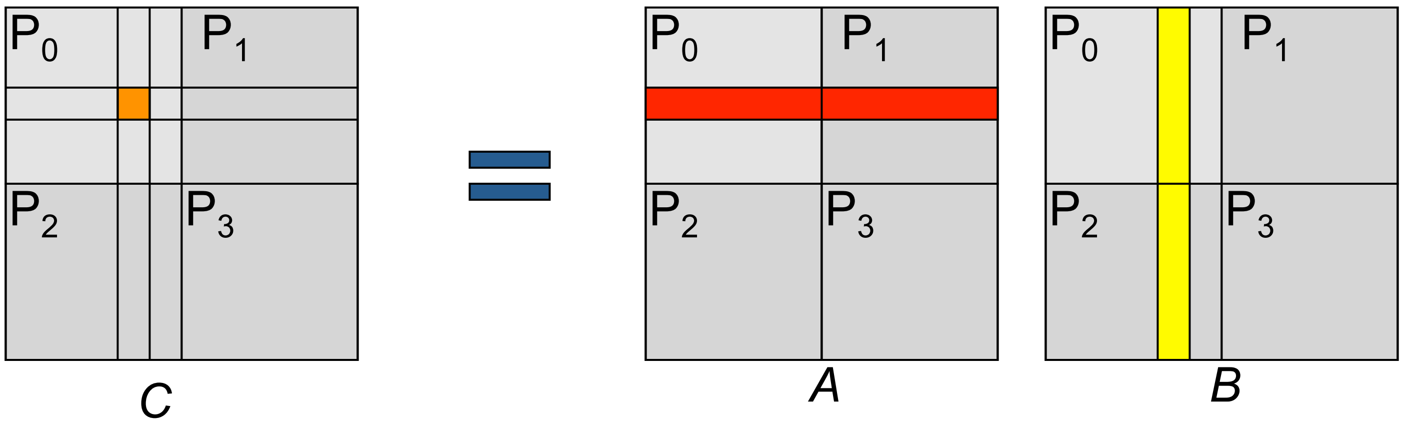
```
alphabetizeInPlace(K[],bit(index,0));     Local sort, up or
                                          down based on bit 0
for(d=1; d<=m; d++)  {                    Main loop, m phases
  for(p=d-1; p<0; p--)  {         Define p for each sub-phase
    stall=free'[neigh(index,p)];     Stall till can give data
    for(i=0; i<size; i++)  {          Send my data to partner
      BufK[neigh(index,p)][i]=K[i];     neigh() finds partner
    }
    ready'[neigh(index,p)]=true;      Release neighbor to go
    stall=ready'[index];          Stall till my data is ready
    … Bitonic merge two buffers (mine in K, partner's in my
    local piece of BufK), I keep half, partner keeps other
    … Barrier
  }
}
… Copy back into L (via inputCopy)
```

Agenda

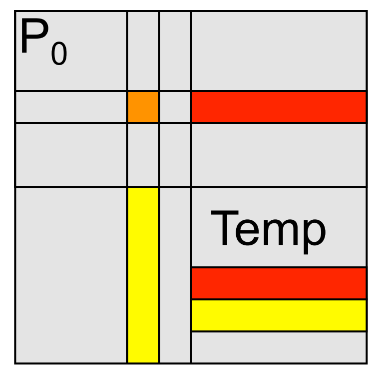
- Discuss parallel algorithms
  - Huge topic, could spend an entire quarter (and more)
- We will just give some highlights
  - Re-conceptualizing computation – classic example of SUMMA matrix multiplication
  - Formulating algorithms as generalized reduces and scans

Recall From Lecture 1

- Matrix Multiplication on Processor Grid

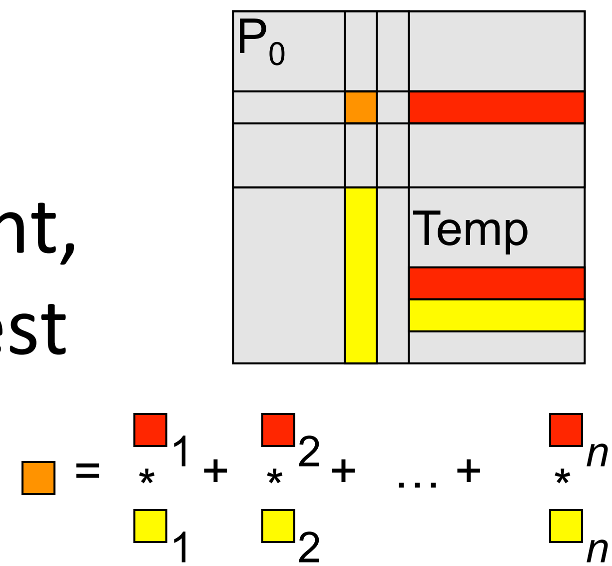


  - Matrices ***A*** and ***B*** producing *n* x *n* result ***C*** where $C_{rs} = \sum_{1 \le k \le n} A_{rk} * B_{ks}$
  - Need to copy partial row from A and partial column from B.
    - In this example, row from P_1, column from P_2



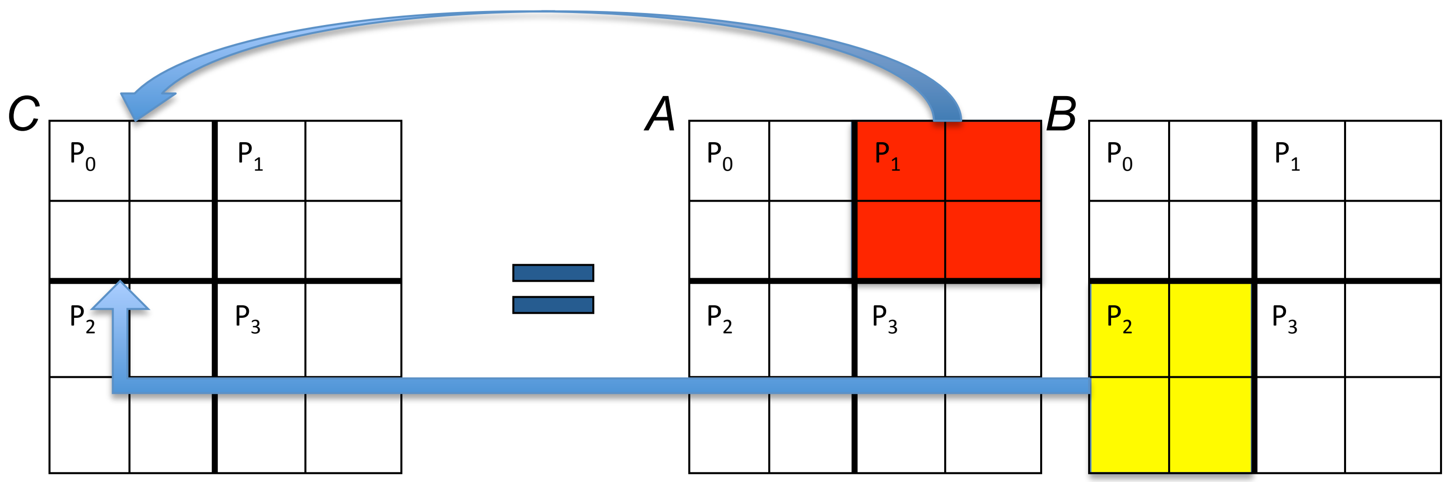
Applying Scalable Techniques

- Assume each processor stores block of ***C***, ***A***, ***B***; assume "can't" store all of any matrix
- To compute c_{rs} a processor needs all of row r of ***A*** and column s of ***B***
- Consider strategies for minimizing data movement, because that is the greatest cost – what are they?



Grab All Rows/Columns At Once

- Send each processor all of rows and columns it needs at the beginning – rest is all local.

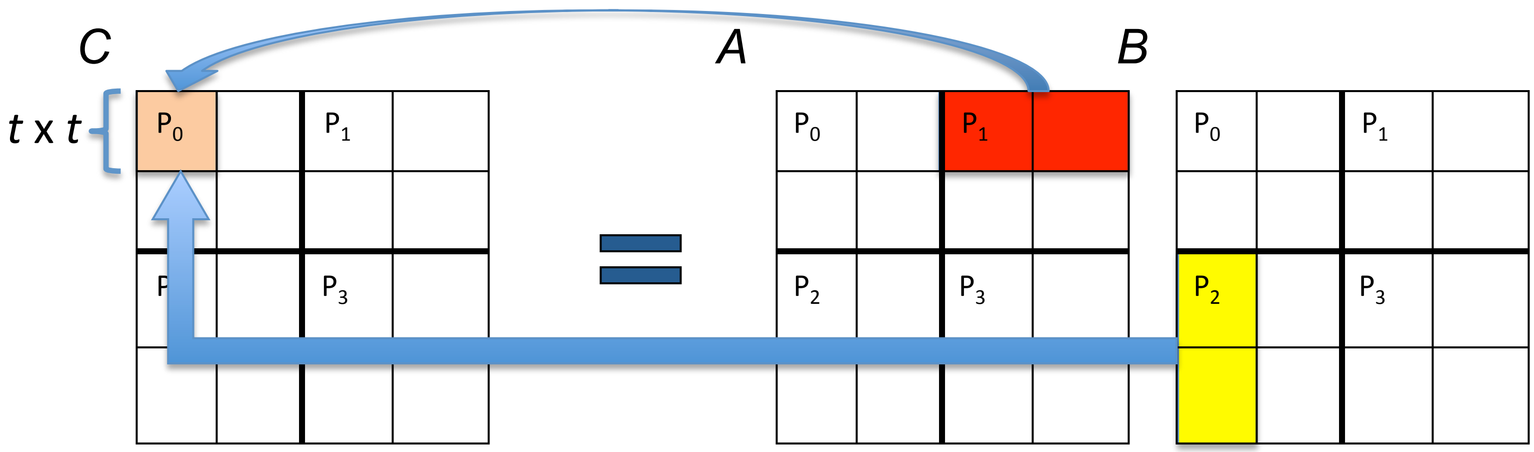


- If there was that much space, why aren't we using bigger blocks?
- Network congestion – all threads doing this in parallel?

Process *t* x *t* Blocks

- What if, instead of processing entire *m* x *m* block we process smaller *t* x *t* chunks?

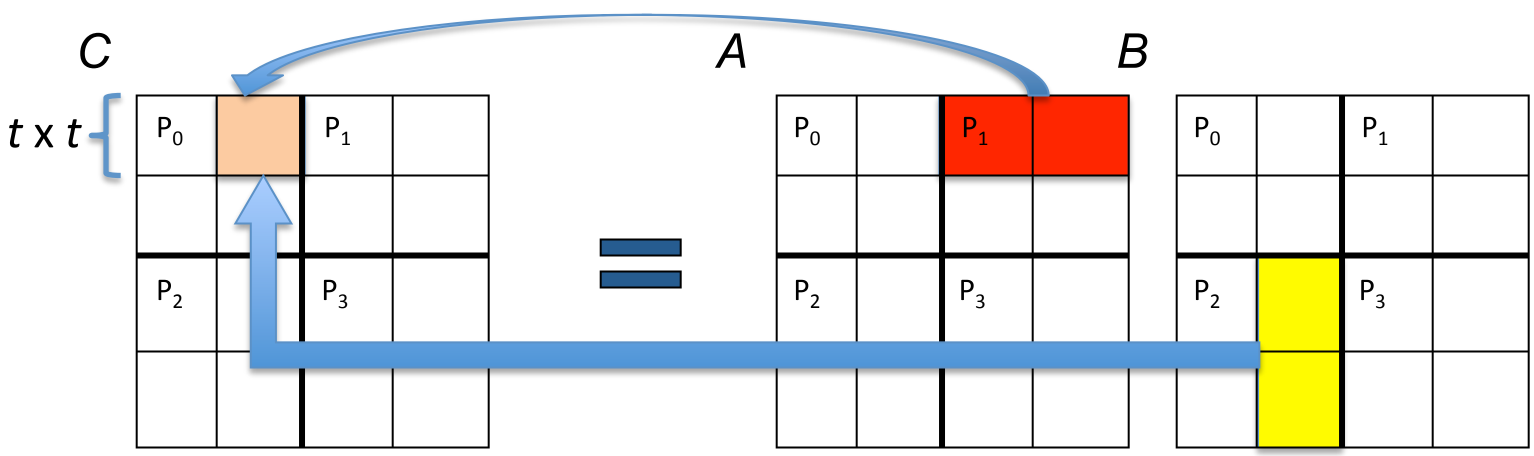
```
for (r=0; r < t; r++)
  for (s=0; s < t; s++){
    c[r][s] = 0.0;
    for (k=0; k < n; k++)
      c[r][s] += a[r][k]*b[k][s];
  }
```



Process *t* x *t* Blocks

- What if, instead of processing entire *m* x *m* block we process smaller *t* x *t* chunks?

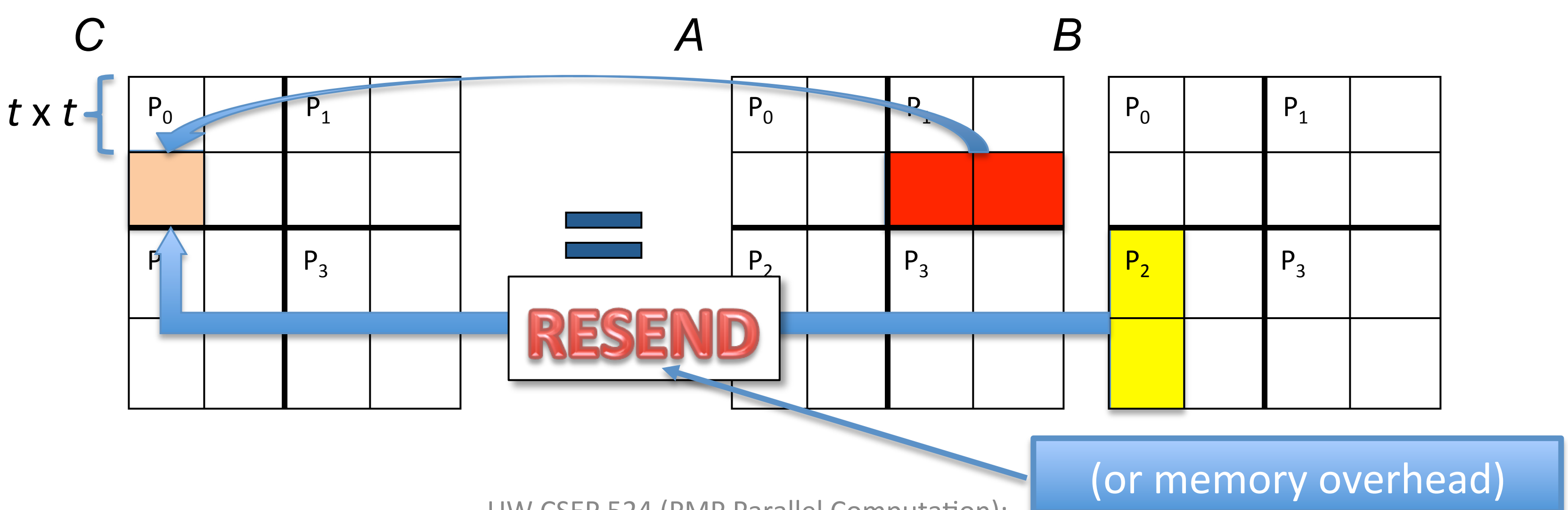
```
for (r=0; r < t; r++)
  for (s=0; s < t; s++){
    c[r][s] = 0.0;
    for (k=0; k < n; k++)
       c[r][s] += a[r][k]*b[k][s];
  }
```



Process *t* x *t* Blocks

- What if, instead of processing entire *m* x *m* block we process smaller *t* x *t* chunks?

```
for (r=0; r < t; r++)
  for (s=0; s < t; s++){
    c[r][s] = 0.0;
    for (k=0; k < n; k++)
       c[r][s] += a[r][k]*b[k][s];
  }
```



Revisit The Formula

$$C_{rs} = \sum_{1 \le k \le n} A_{rk} * B_{ks}$$

```
// Assume c[][] initialized to 0s
for (r=0; r < n; r++){
  for (s=0; s < n; s++){
    for (k=0; k < n; k++){
      c[r][s] += a[r][k]*b[k][s];
    }
  }
}
```

Revisit The Formula

$$C_{rs} = \sum_{1 \le k \le n} A_{rk} * B_{ks}$$

```
// Assume c[][] initialized to 0s
for (r=0; r < n; r++){
  for (s=0; s < n; s++){
    for (k=0; k < n; k++){
      c[r][s] += a[r][k]*b[k][s];
    }
  }
}
```

What if we lift the *k*-loop out of the nest?

Revisit The Formula

$$C_{rs} = \sum_{1 \le k \le n} A_{rk} * B_{ks}$$

```
// Assume c[][] initialized to 0s
for (k=0; k < n; k++){
  for (r=0; r < n; r++){
    for (s=0; s < n; s++){
      c[r][s] += a[r][k]*b[k][s];
    }
  }
}
```

Does this still compute the same values?
What have we done?

Revisit The Formula

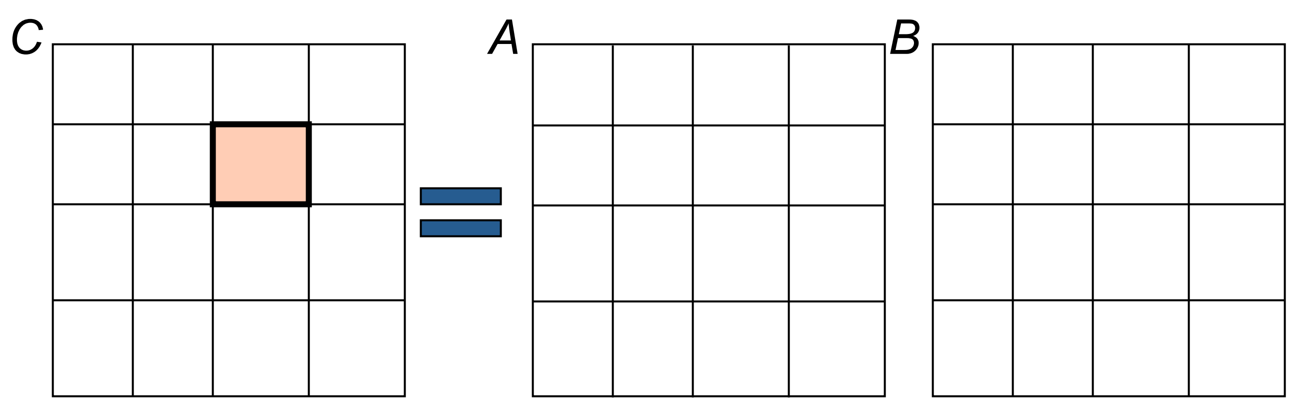
$$C_{rs} = \sum_{1 \le k \le n} A_{rk} * B_{ks}$$

```
// Assume c[][] initialized to 0s
for (k=0; k < n; k++){
  for (r=0; r < n; r++){
    for (s=0; s < n; s++){
      c[r][s] += a[r][k]*b[k][s];
    }
  }
}
```

Computing C term-by-term rather than element-by-element (all 1st terms, all 2nd terms, etc.)

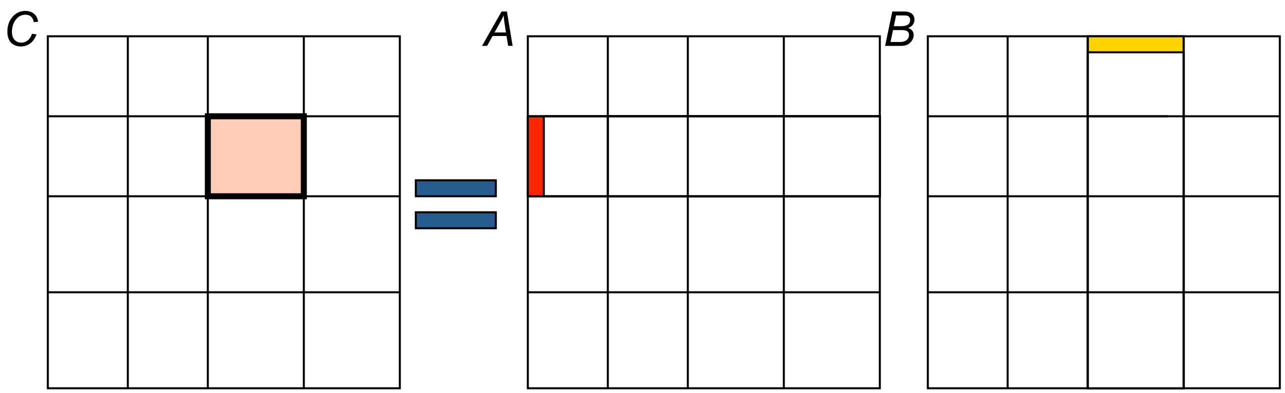
Change Of View Point

- Consider this *m* x *m* block – what do we need to compute 1st terms?



Change Of View Point

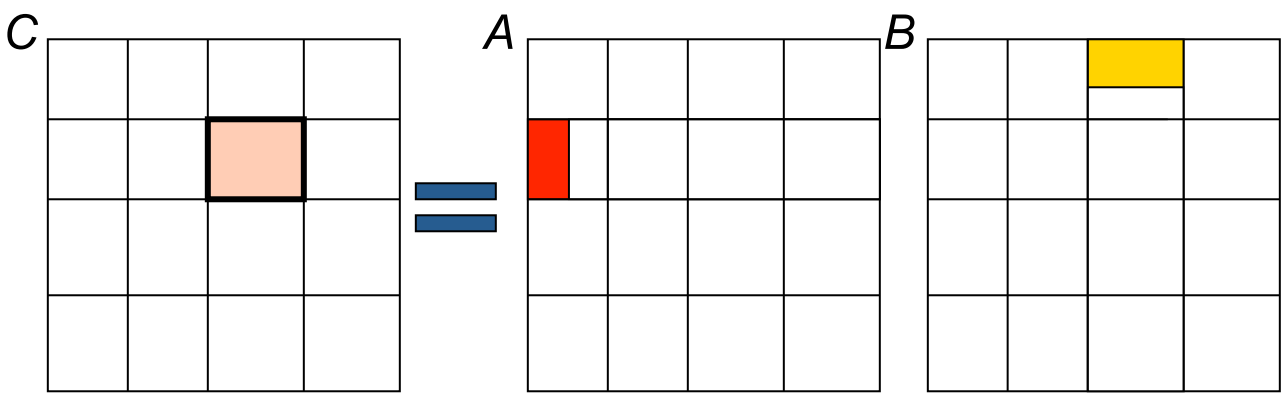
- Consider this *m* x *m* block – what do we need to compute 1st terms?



Switch orientation -- by using a *column* of *A* and a *row* of *B* compute all 1st terms of the dot products

Change Of View Point

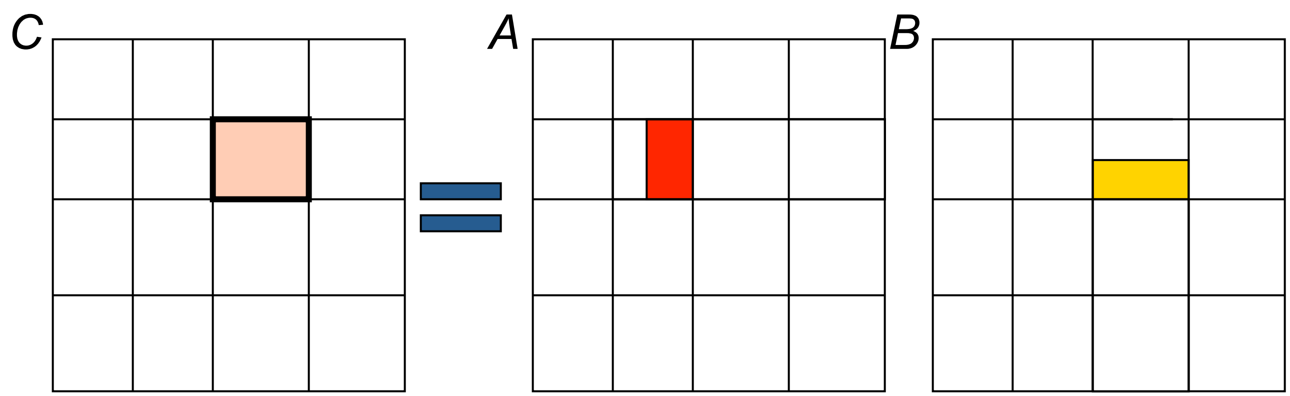
- Consider this m x m block – what do we need to compute 1st t terms?



Need *t* ***columns*** of *A* and *t* ***rows*** of *B* ...

Change Of View Point

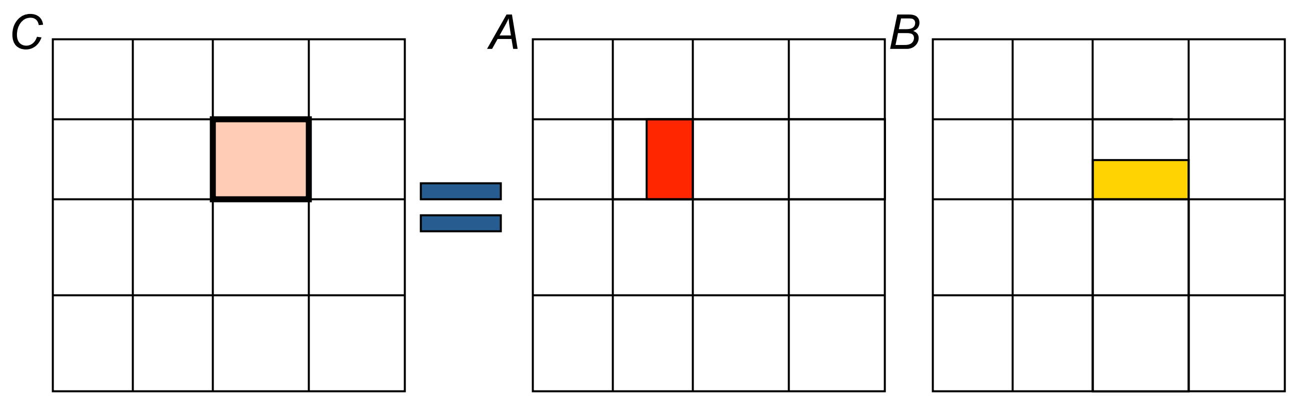
- Consider this *m* x *m* block – what do we need to compute arbitrary set of the same *t* terms?



Need different *t* ***columns*** of *A* and *t* ***rows*** of *B* …

Change Of View Point

- Consider this *m* x *m* block – what do we need to compute arbitrary set of *t* terms?

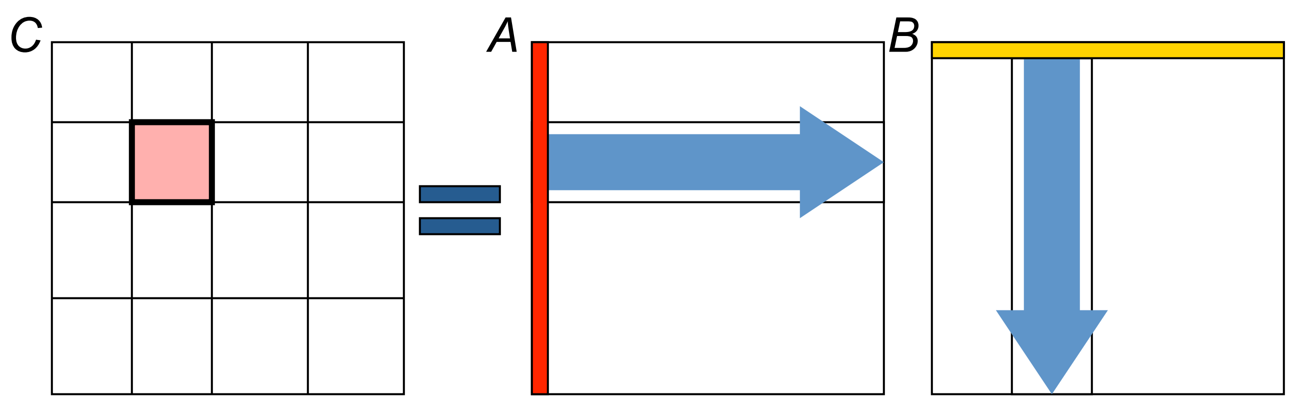


Need different *t* ***columns*** of *A* and *t* ***rows*** of *B* …

- Key: each block only needs each value once, can compute all terms that depend on it

Higher Level SUMMA View

- SUMMA communication: send my portion of row (or block of rows) to everyone in my column, my portion of column (or block of columns) to everyone in my row
- Followed by a step of computing next term(s) locally
- Repeat with next (block of) partial row(s)/column(s)…

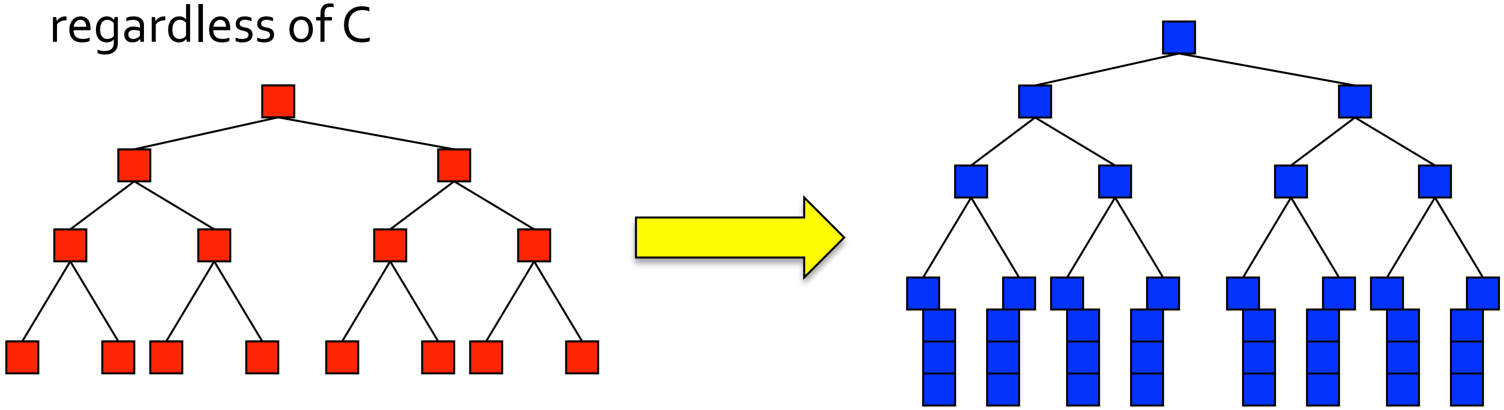


SUMMA

- Scalable Universal Matrix Multiplication Algorithm
  - Invented by van de Geijn & Watts of UT Austin
  - Generally considered best machine independent Matrix Multiplication
  - Many linear algebra libraries implement variations of this
- Whereas MM is usually A row x B column, SUMMA is A column x B row because computation switches sense
  - Normal: Compute all terms of a dot product
  - SUMMA: Computer a term of all dot products
- Key: Don't have to send data twice!
  - By computing term-by-term, and "flipping the sense", each processor can do all computations from a received block at once .

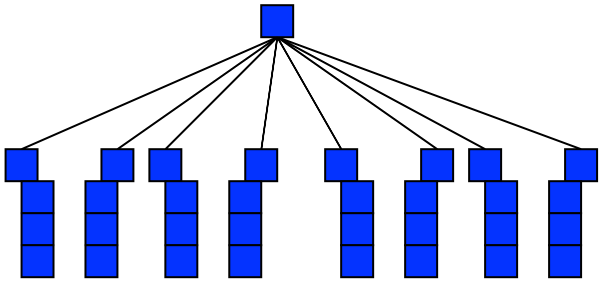
Schwartz's Algorithm

- Recall our observation earlier that it made sense to locally sum numbers before combining them in a tree.
- The generalized version of this is due to Jack Schwartz. Idea:
  - Can combine *N* items on *P*= *N* threads/processors in log *P* (=log *N*) time
  - If we first combine O(log *N*) values at each leaf, we end up with the same time complexity (O(log *N*)), but *CN* log *N* values!
  - In practice, communication $>>_{cost}$ local computation, so this is a big win regardless of C



Schwartz

- Generally *P* is not a variable, and *P* << *N*
- Use Schwartz as heuristic: Prefer to work at leaves (no matter how much bigger *N* is than *P*) rather than enlarge (make a deeper) tree, implying tree will have no more than $log_2 P$ height
- Also, consider higher degree tree – especially if communication can be overlapped (multiple outstanding fetches/receives)

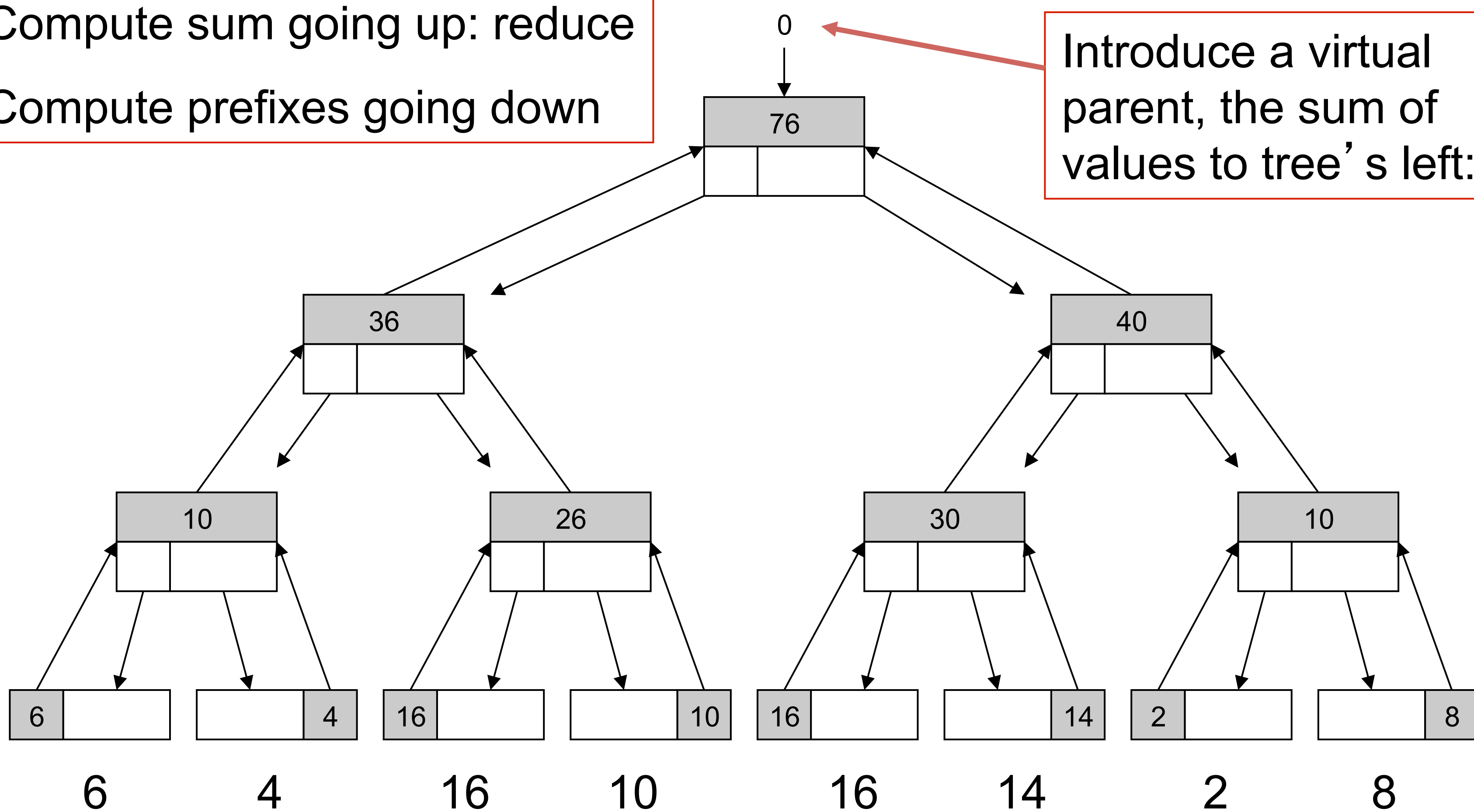


Recall Parallel Prefix Algorithm – Canonical Scan

Compute sum going up: reduce

Compute prefixes going down

Introduce a virtual parent, the sum of values to tree's left: 0

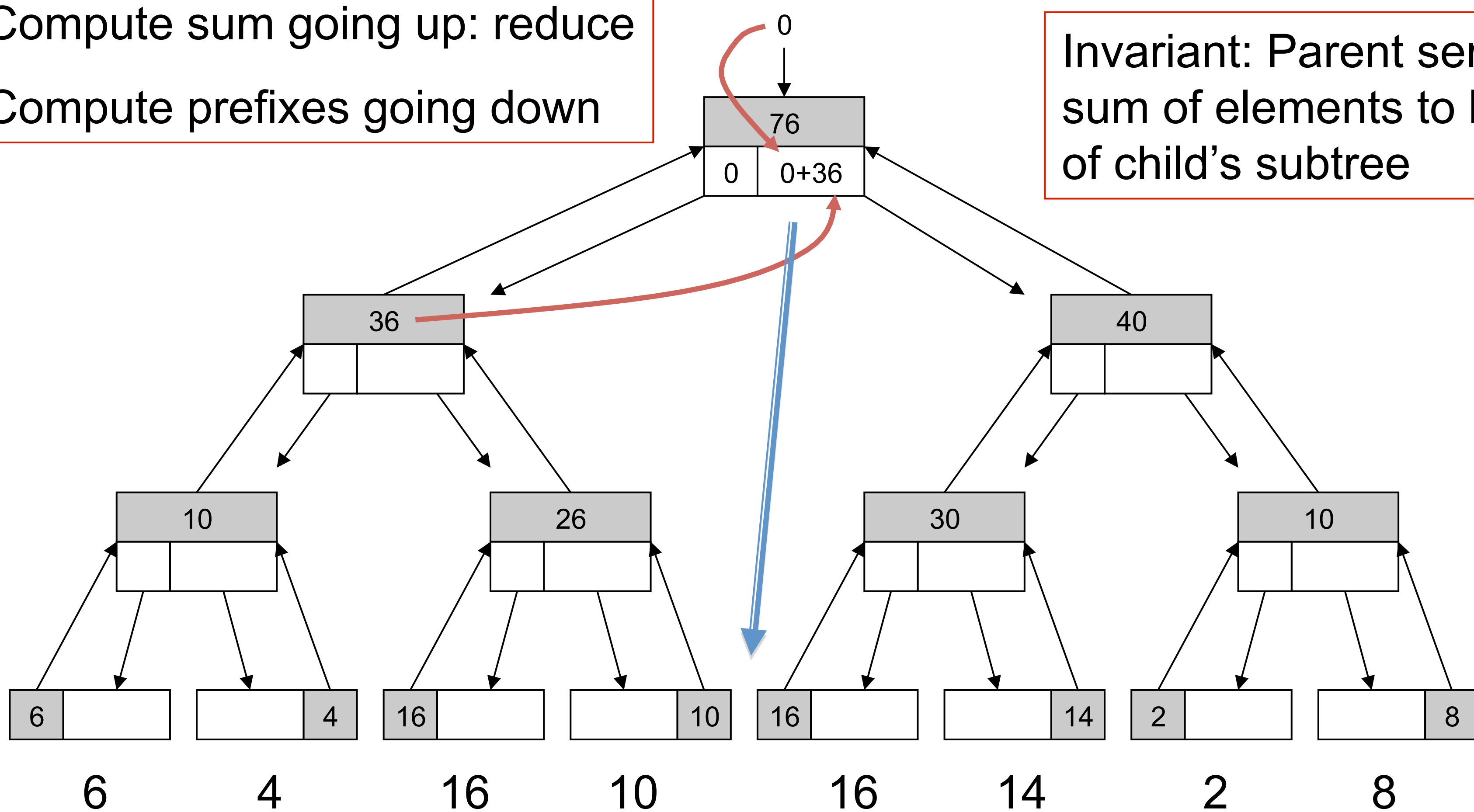


Recall Parallel Prefix Algorithm – Canonical Scan

Compute sum going up: reduce

Compute prefixes going down

Invariant: Parent sends sum of elements to left of child's subtree

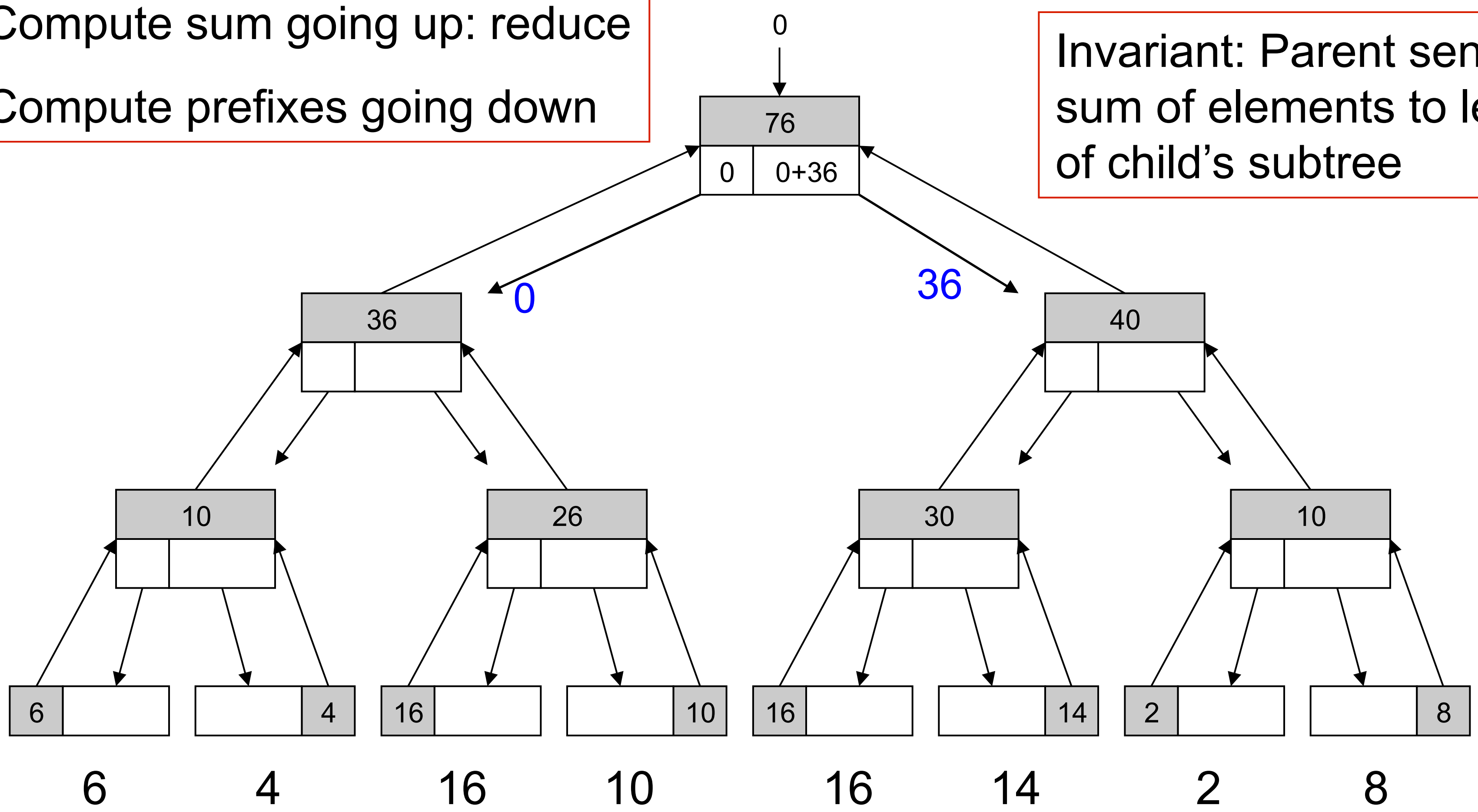


Recall Parallel Prefix Algorithm – Canonical Scan

Compute sum going up: reduce

Compute prefixes going down

Invariant: Parent sends sum of elements to left of child's subtree

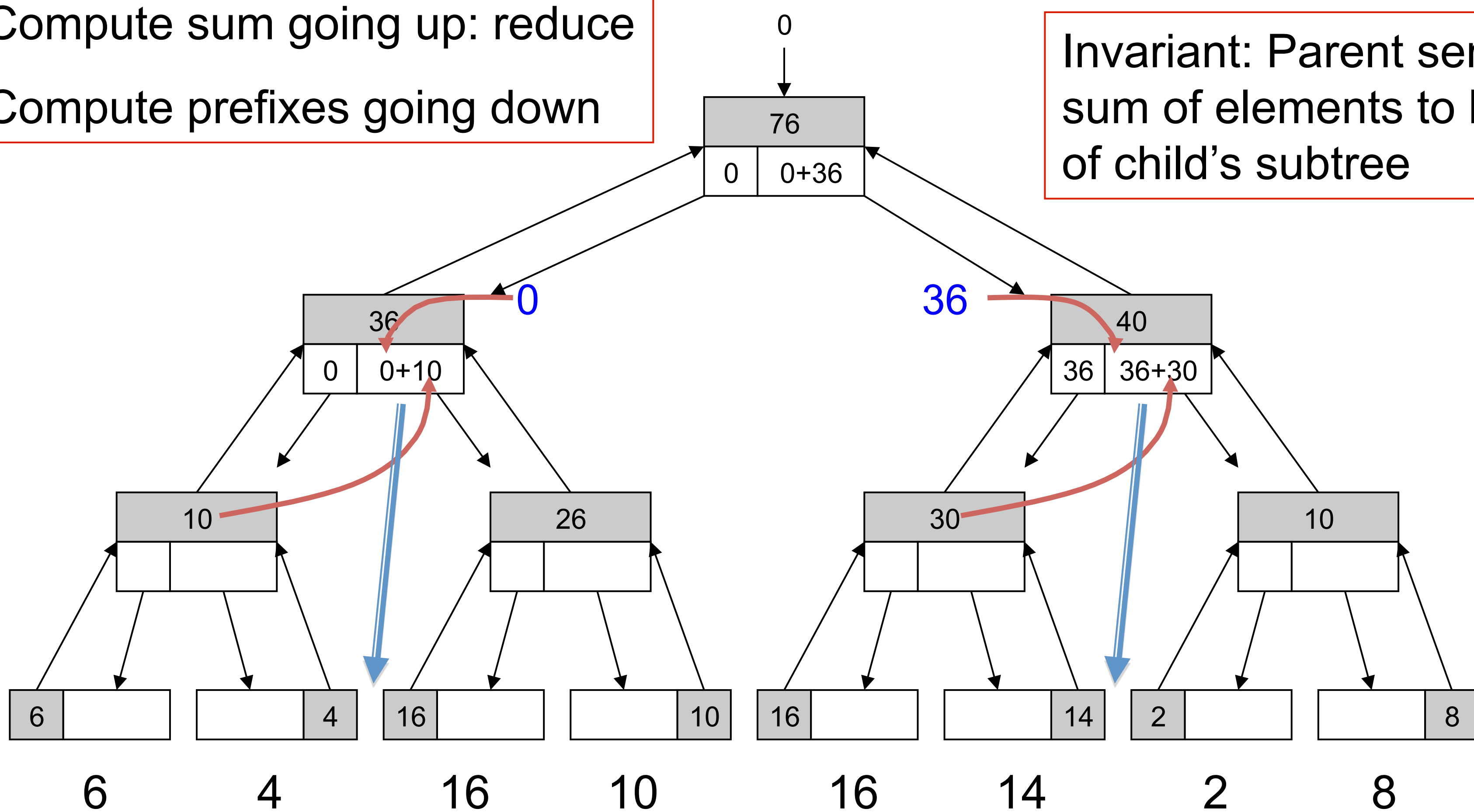


Recall Parallel Prefix Algorithm – Canonical Scan

Compute sum going up: reduce

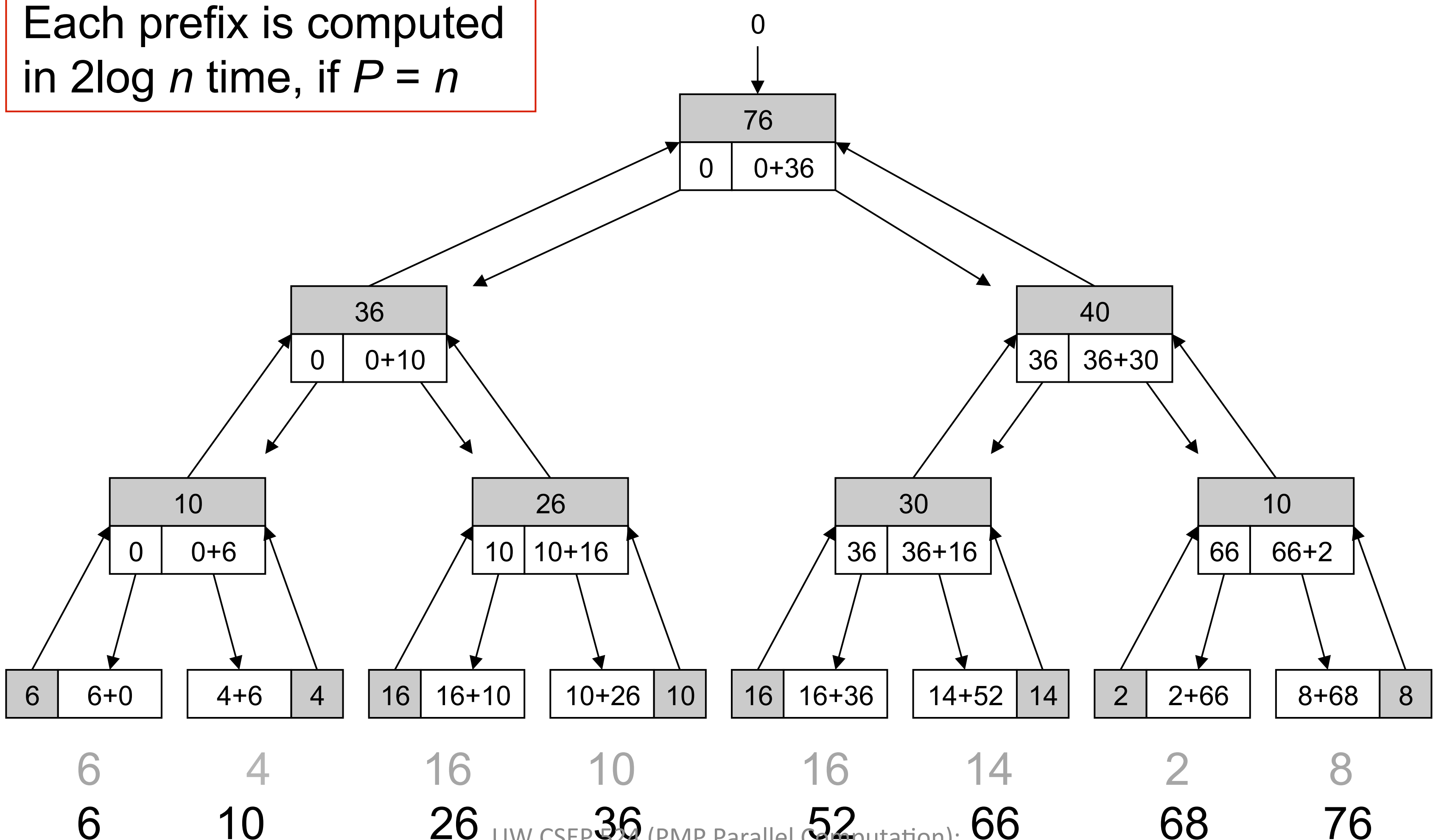
Compute prefixes going down

Invariant: Parent sends sum of elements to left of child's subtree



Recall Parallel Prefix Algorithm – Canonical Scan

Each prefix is computed in 2log n time, if $P = n$



Generalized Reduce and Scan

- We've seen the notions of tree-based reduce and scan pop up repeatedly
  - *Reduce* aggregates elements into a single result (e.g., sum)
  - *Scan* also computes all "partial results" (e.g., prefix sum)
- Language-level support for +, *, min, max, &&, || is common
- Turns out that many algorithms can be formulated (and parallelized) as *generalized* reduces or scans
- If so, can practically apply to "recipe" to achieve efficient tree-based (Schwartz) parallelization
- **Note**: Scans can be inclusive (output[0] = input[0]) or exclusive (output[0] = identity, output[1] = input[0])
  - Exclusive is more flexible, as we will see…

Examples

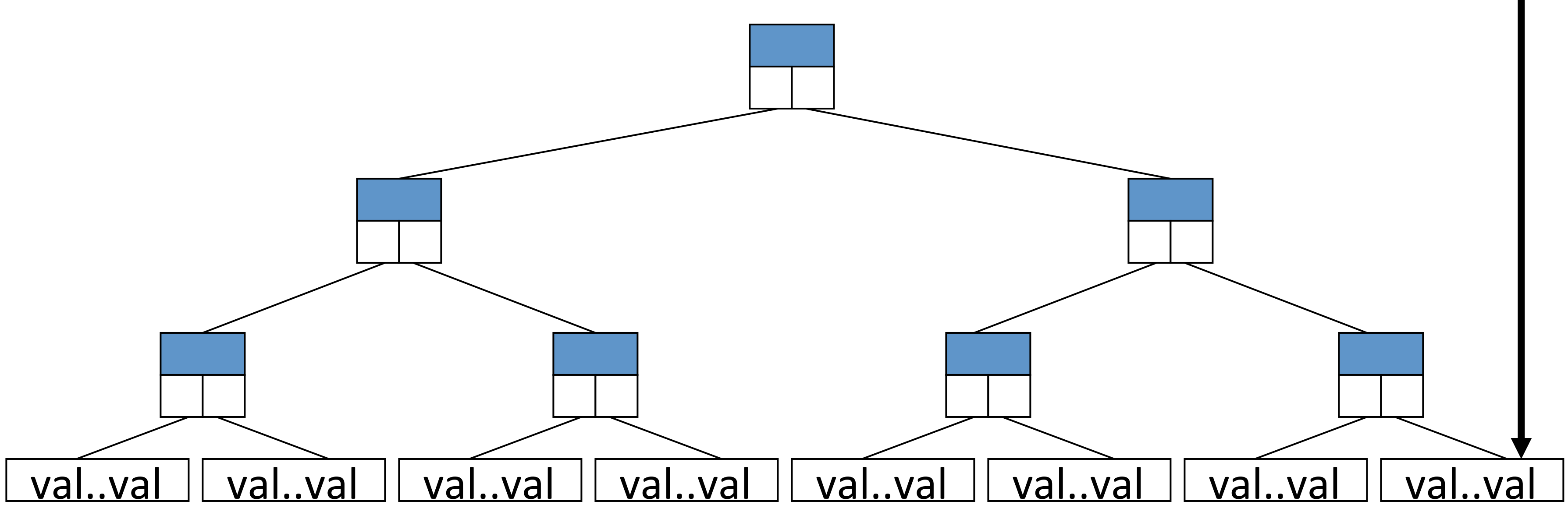
- Reduce examples
  - Second smallest value (!= smallest): send two smallest to parent, parent combines by keeping two smallest across children.
  - Length of longest run of 1's: compute longest in each leaf, take max at parent. Requires edge cases to track/handle 1s that cross child boundaries
  - Histogram, counts items in k buckets: how would you?
  - Index of first occurrence of x: how would you?

Examples

- Scan examples
  - Team standings at every point from list of game results:
    - Instead of prefix sum of scalar, do a prefix sum of vector *v*, where v_i is number of wins of team *i*.
    - Treat each game element as a vector with a 1 in the winning team's position.
  - Index of most recent occurrence of a character:
    - Locally compute *last* occurance of each character in term of global indices.
    - Combine at parents by taking max for each character
    - On the way down, we will receive the last occurrence to the left of our leaf – use to initialize local rescan

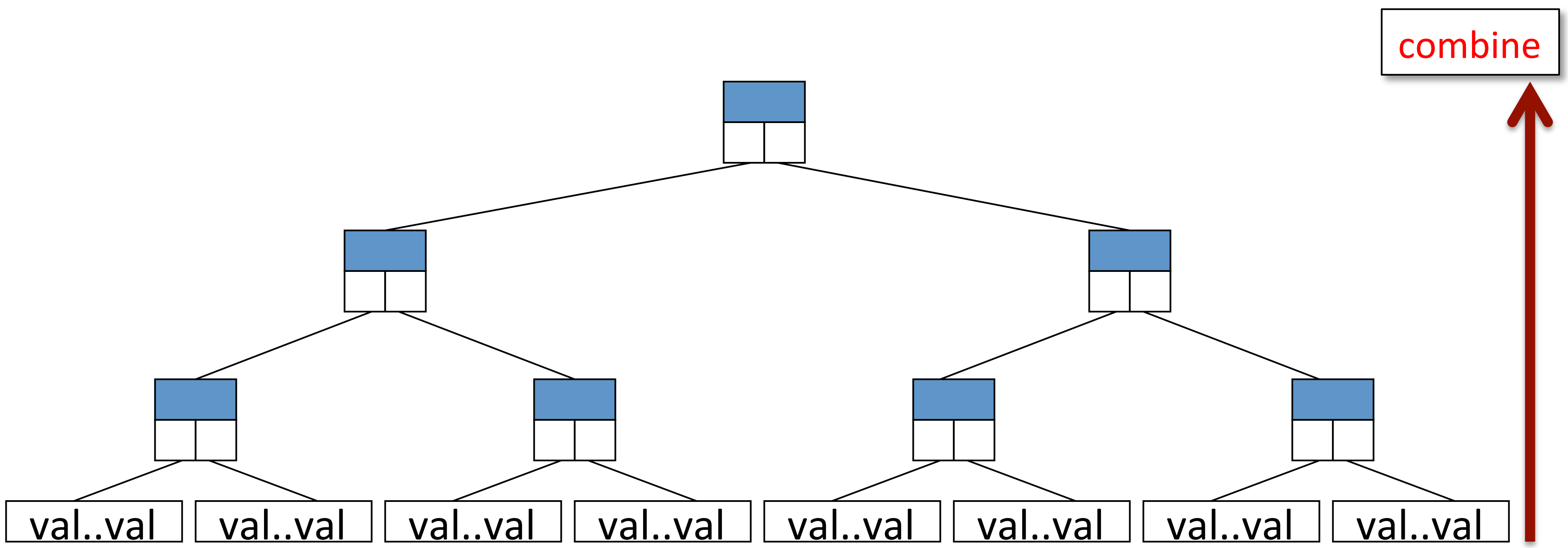
Structure of Computation

- Begin by applying Schwartz idea to problem
  - Local accumulate at leaves



Structure of Computation

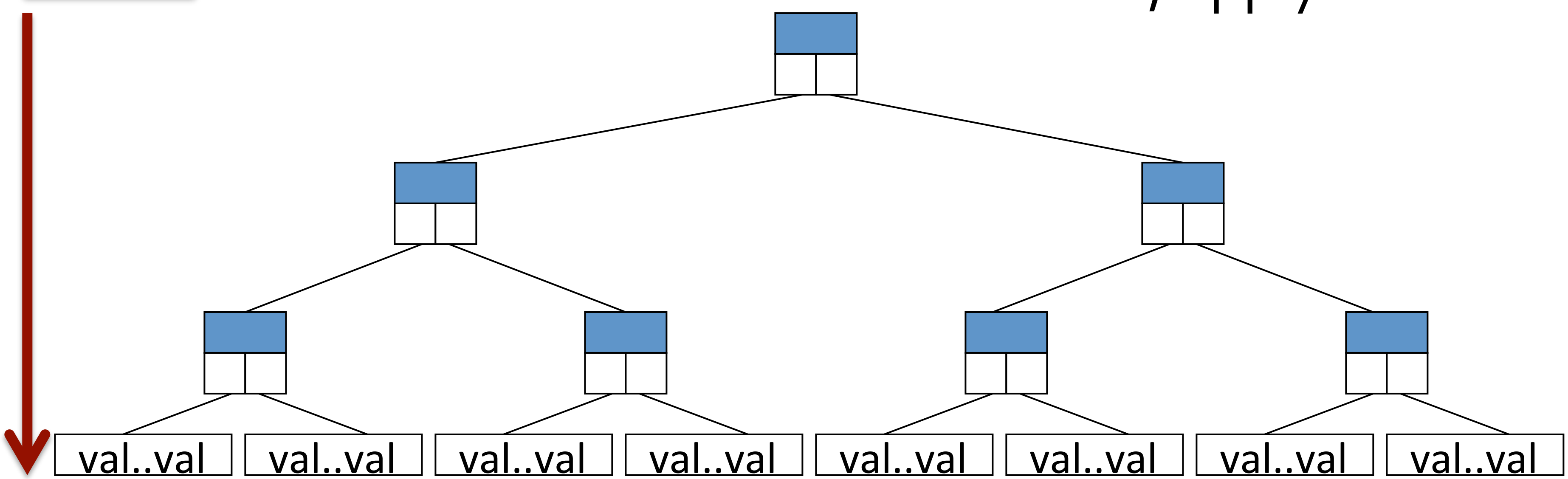
- Begin by applying Schwartz idea to problem
  - Local computation
  - Combine leaf results at parents



Structure of Computation

- Begin by applying Schwartz idea to problem
  - Local computation
  - Combine leaf results at parents
  - If scan: send down “values to left”, apply at leaves

“to-left”

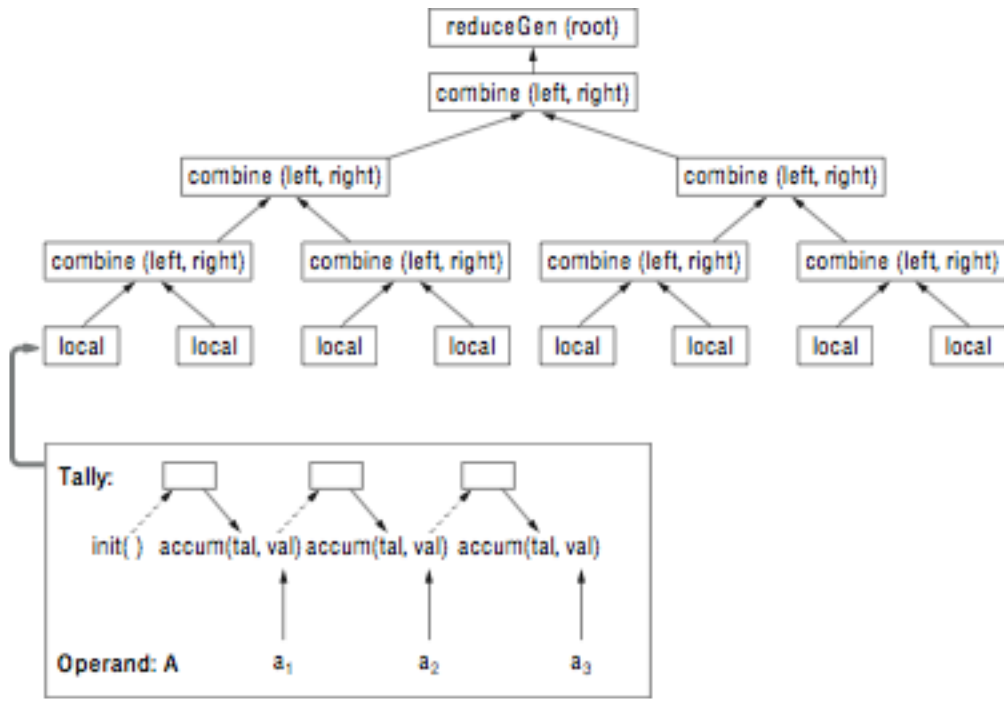


Generalizing R & S

- Goal: come up with a recipe for parallel reduces and scans.
- Attempt to define in terms of four *sequential* functions:
  - `init()` initialize data structures
  - `accum()` perform local computation
  - `combine()` perform tree combining
  - *x*`_gen()` produce the final result(s)
    - *x* = reduce
    - *x* = scan

Reduce illustration from Textbook

- `init()`: Initialize tally at each leaf
- `accum()`: Aggregate each array value into tally
- `combine()`: Combine child tallys at each parent
- `reduceGen()`: Return root



Reduce Recipe Pseudocode

```
tally nodeval'[P];             Global full/empty variables
tally result;                  tally represents result datatype
forall(index in (0..P-1)) {
  int myData[size] = localize(dataarray[]);  Local portion
  tally tal;
  int stride = 1;
  tal = init();                                  Initialization
  for (int i = 0; i < size; i++)
    tally = accum(tally, myData[i]);     Local accumulation
  while(stride < P) {
    if(index % (2*stride) == 0) {
      tally = combine(tally, nodeval'[index+stride]);
      stride = 2 * stride;
    } else {
      nodeval'[index] = tally;          Done: fill for parent
      break;
    }
  }
}
result = reduceGen(nodeval'[0]);      Generate final result
```

Example: Sum Reduce

```
typedef int tally;
```

```
tally init() {
  tally tal = new tally;
  tal=0;
  return tal;
}
```

```
tally combine(tally left,
              tally right) {
  return left + right;
}
```

```
tally accum(int op_val,
            tally tal) {
  tal += op_val;
  return tal;
}
```

```
int reduce_gen(tally ans) {
  return ans;
}
```

More Involved Case

- Consider Second Smallest – find second smallest unique value
- **`tally`** tracks smallest and next smallest found so far:

```
struct tally {
  float sm;  // smallest
  flost nsm; // next smallest
};
```

- Initialization:

```
tally init() {
   pair = new tally;
   pair.sm = maxFloat;
   pair.nsm = maxFloat;
   return pair;
}
```

Second Smallest (Continued)

- Accumulate

```
tally accum(float op_val, tally tal) {
  // Check if op_val less than smallest
  if (op_val < tal.sm) {
    tal.nsm = tal.sm;
    tal.sm = op_val;
  } else {
    // Otherwise, check if op_val betweeen
    // smallest and second smallest
    if (op_val > tal.sm && op_val < tal.nsm) {
      tal.nsm = op_val;
    }
  }
  return tal;
}
```

Second Smallest (Continued)

- Combining children

```
tally combine(tally left, tally right) {
  return accum(left.nsm, accum(left.sm, right));
}
```

- Generating final result

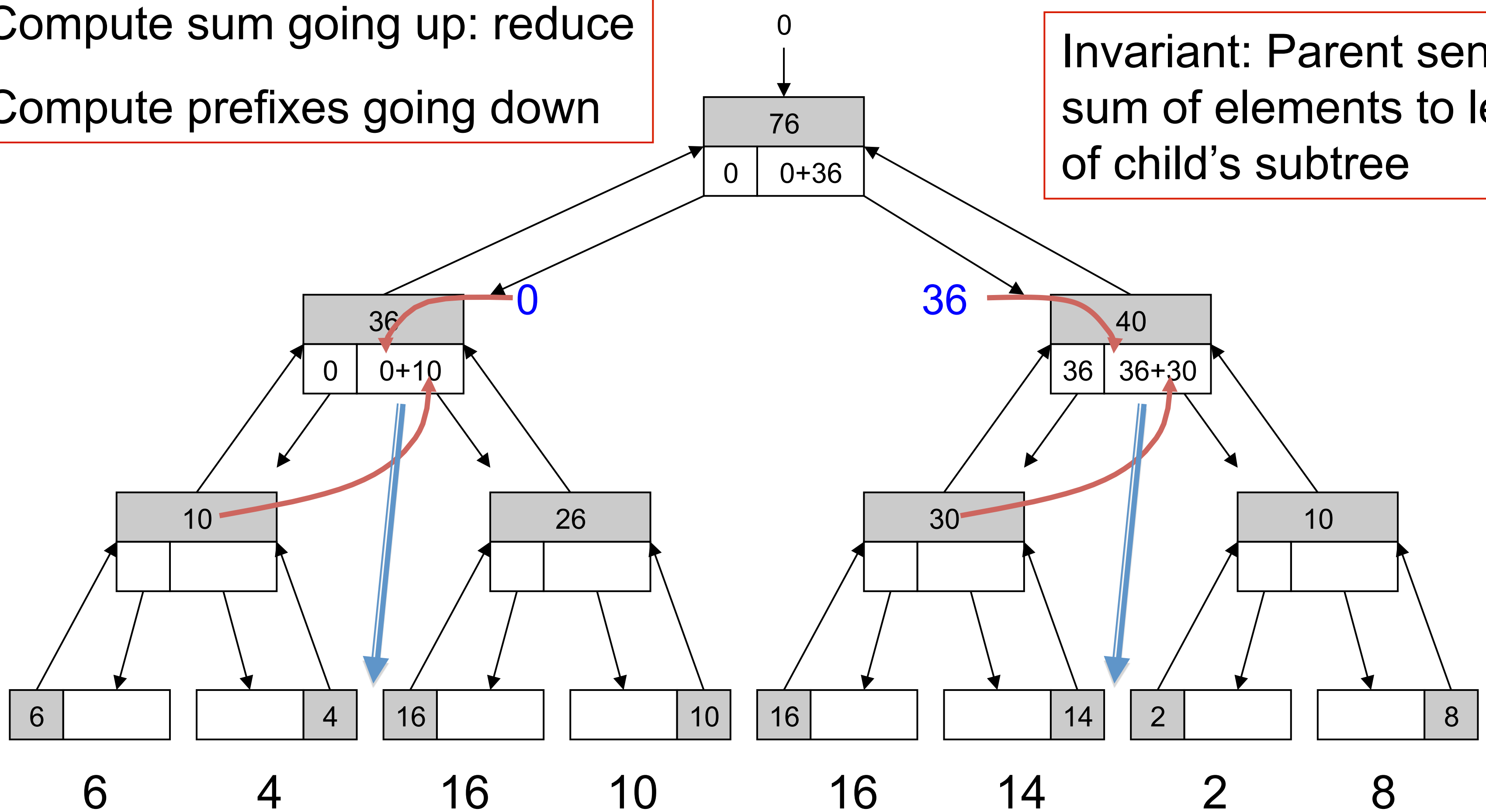
```
int reduce_gen(tally ans){
  return ans.nsm;
}
```

Recall Parallel Prefix Algorithm – Canonical Scan

Compute sum going up: reduce

Compute prefixes going down

Invariant: Parent sends sum of elements to left of child's subtree



Generalized scan

- See textbook errata for full code.
  - In combining loop, track left tally – store it with sibling that will need to add it to parent tally on downsweep:

```
while(stride < P) {
   if(index % (2*stride) == 0) {
     ltally[index + stride] = tally;
     tally = combine(tally, nodeval'[index+stride]);
     stride = 2 * stride;
   } else {
     nodeval'[index] = tally;   Done: fill for parent
     break;
   }
 }
```

Generalized scan

- See textbook errata for full code.
  - Then, add downsweep after upsweep. Ensures leaves have combined value of everything to their left. Recompute local accumulation using total to left to initialize.

```
if (index == 0) {
  dummy = nodeval'[0]; nodeval'[0] = init();
}
for(stride = P/2; stride >= 1; stride = stride/2)
  if(index % (2*stride) == 0) {
    ptally = nodeval'[index];
    nodeval'[index] = ptally;  // Left child gets parent tally,
    nodeval'[index+stride] =   // right gets parent + left tally
      combine(ptally, ltally[index+stride]);
  }
ptally = nodeval'[index]
for(int i = 0; i < size i++) {
  // Re-accumulate using tally of data to left, apply to data
  myResult[i] = scanGen(ptally, myData[i],
                        localToGlobal(myData, i, 0));
  ptally = accum(ptally, myData[i], localToGlobal(myData,i,0));
}
```

Example: Prefix Sum

```
typedef int tally;
tally ltally[P]
```

```
tally init() {
  return 0;
}
```

```
tally accum(tally t,
            int elem,
            int i) {
  return t + elem;
}
```

```
tally combine(tally left,
              tally right) {
  return left + right;
}
```

```
// Already computed prefix
// sum into t when applying
// parent tally to elements
int scan_gen(tally t,
             int elem,
             int i) {
  // + elem makes inclusive
  return t + elem;
}
```

Example: Last Occurrence

```
//S = # of possible symbols
typedef int[S] tally;
tally ltally[P]
```

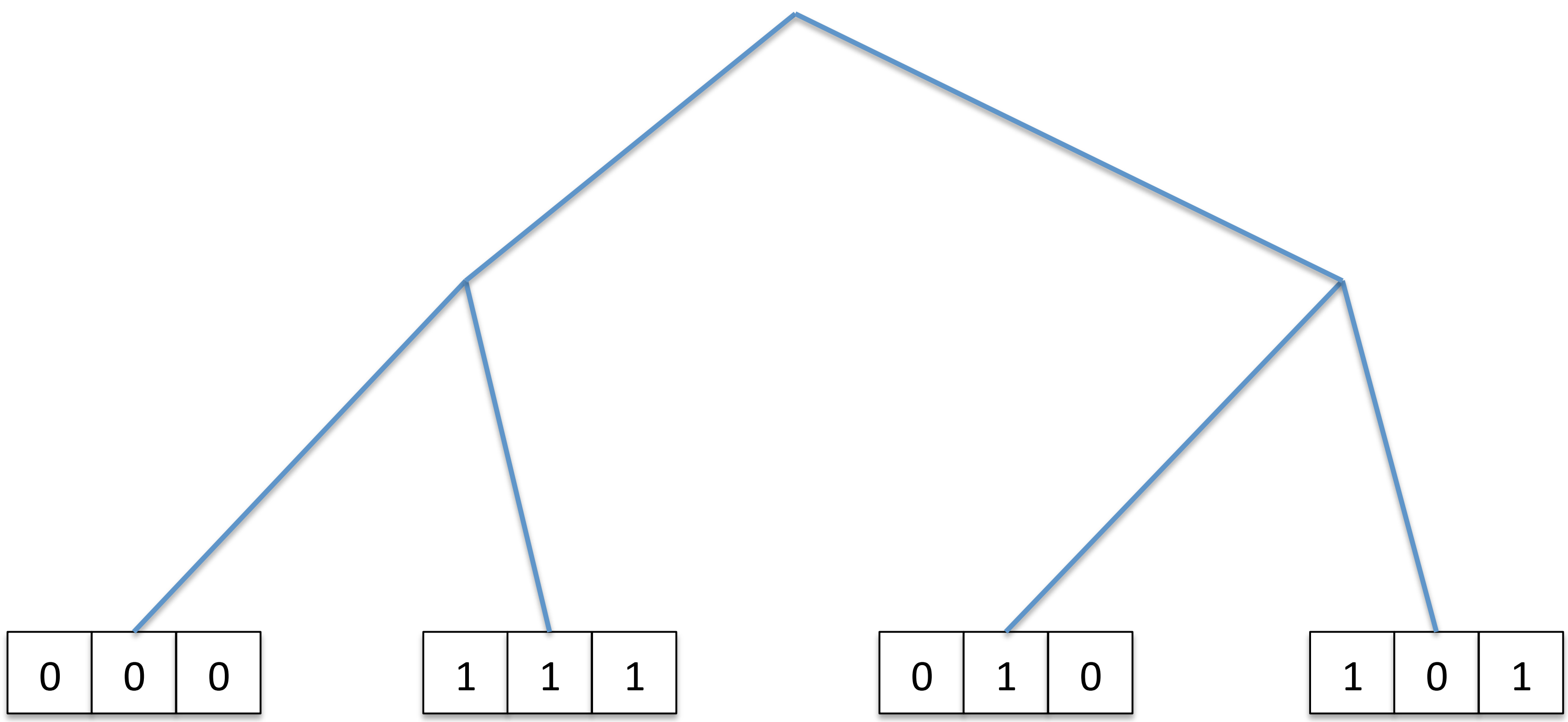
```
tally init() {
  t = new tally;
  for(int i=0; i<S; i++)
    t[i] = -1;
  return t;
}
```

```
tally accum(tally t, int sym,
            int i) {
  t[sym] = i;
  return t;
}
```

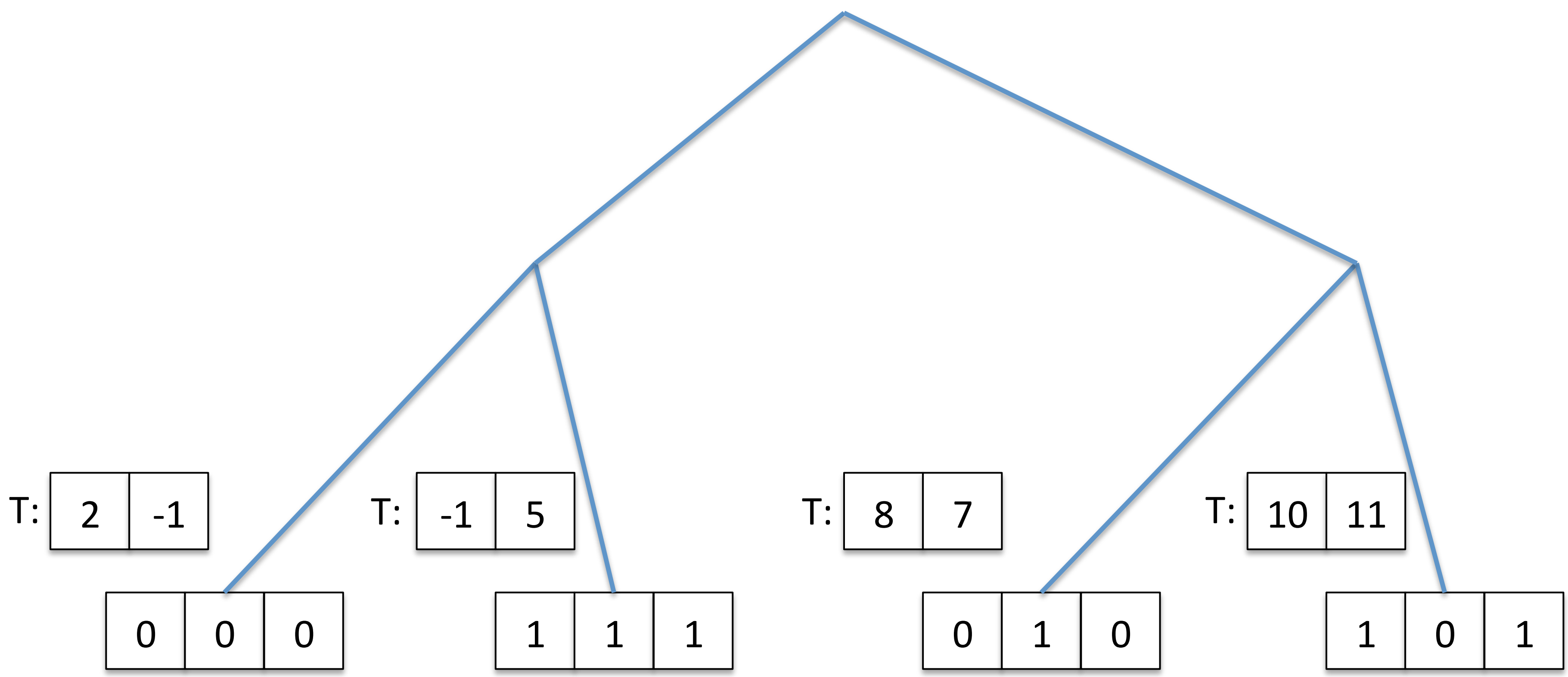
```
tally combine(tally left,
              tally right) {
  for(int i=0; i<S; i++)
    max(left[i],right[i])
}
```

```
// Passed in tally will have
// index of last occurrence of
// each symbol to left of elem
// This is why we prefer
// exclusive ordering in
// recipe – can't undo tally
// updates if non-invertable
int scan_gen(tally t, int sym,
             int i) {
  return t[sym];
}
```

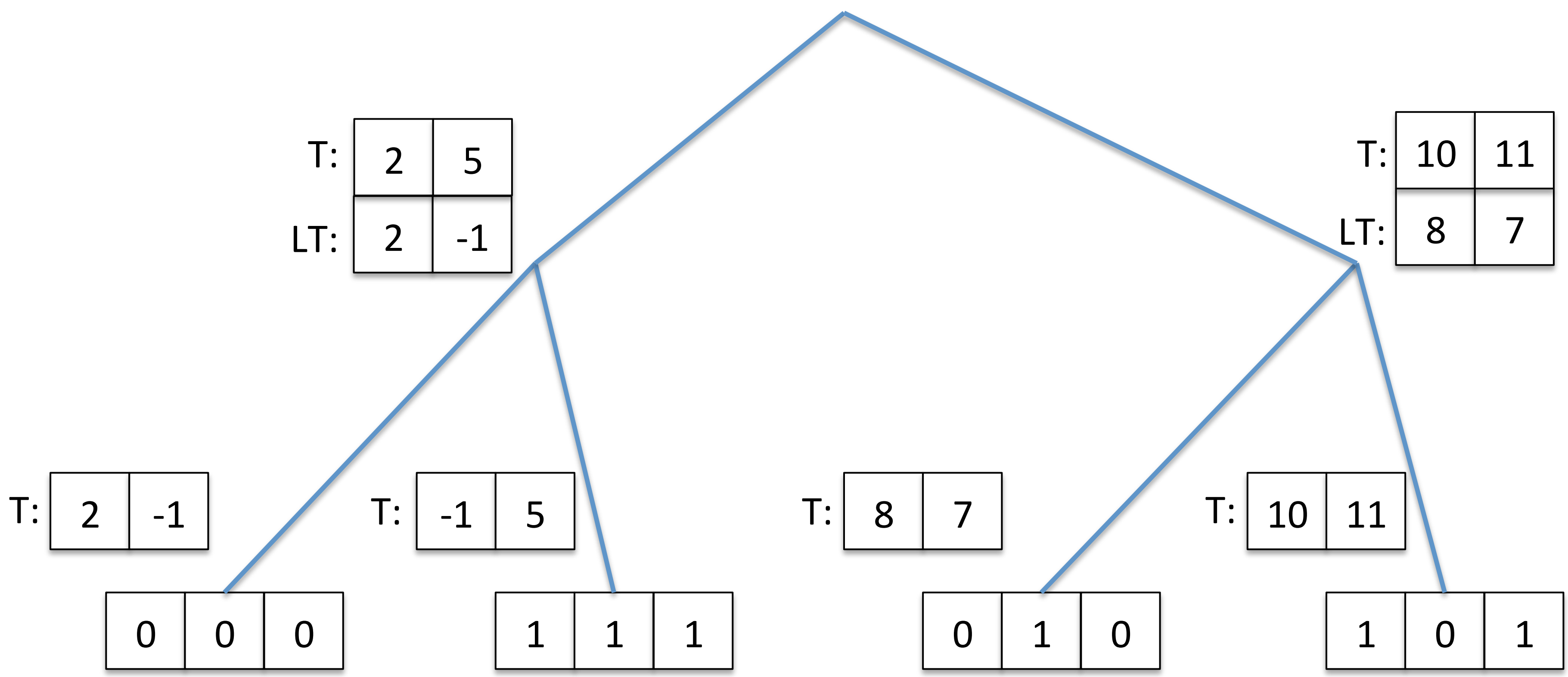
Illustration



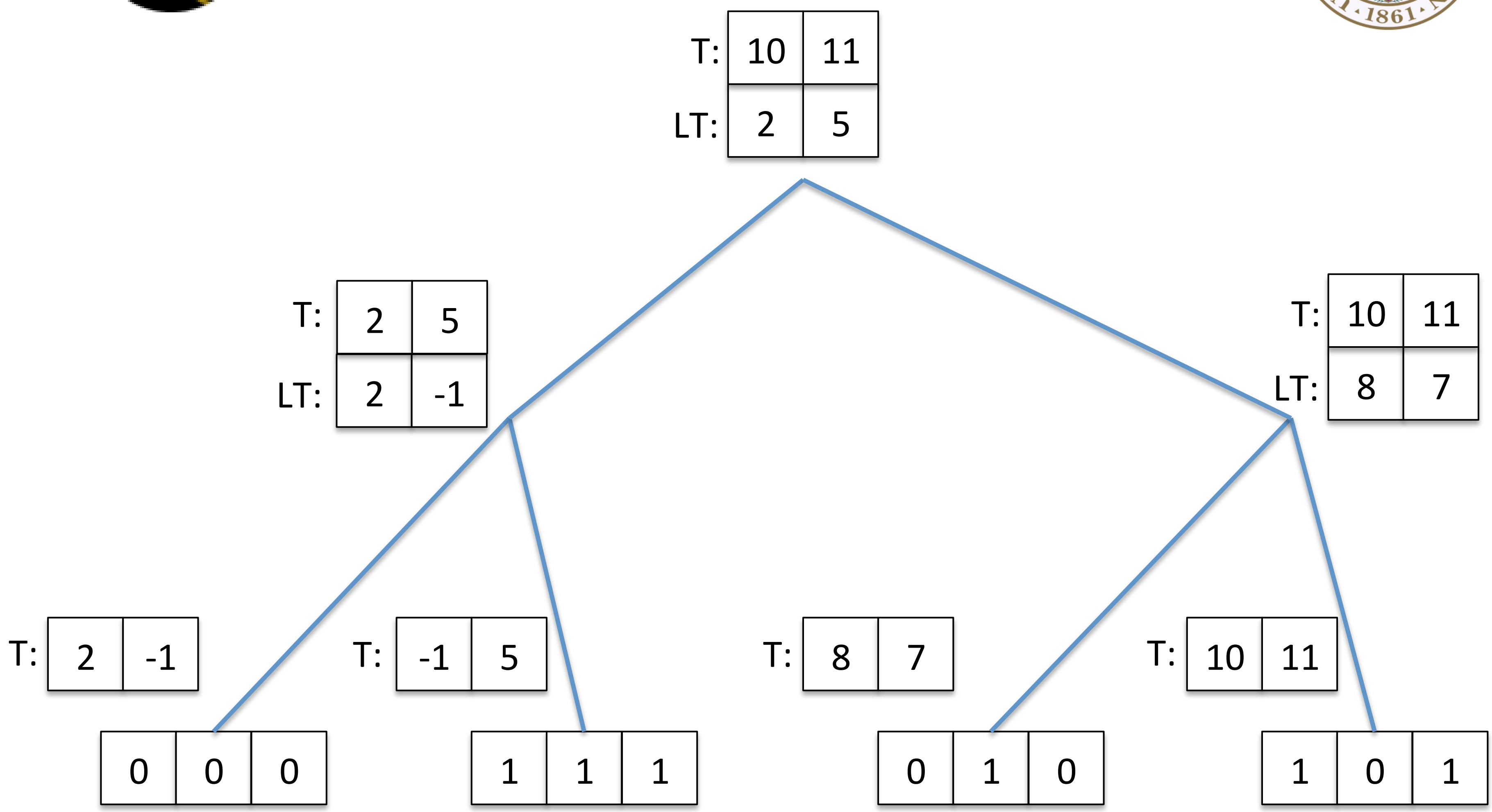
Illustration



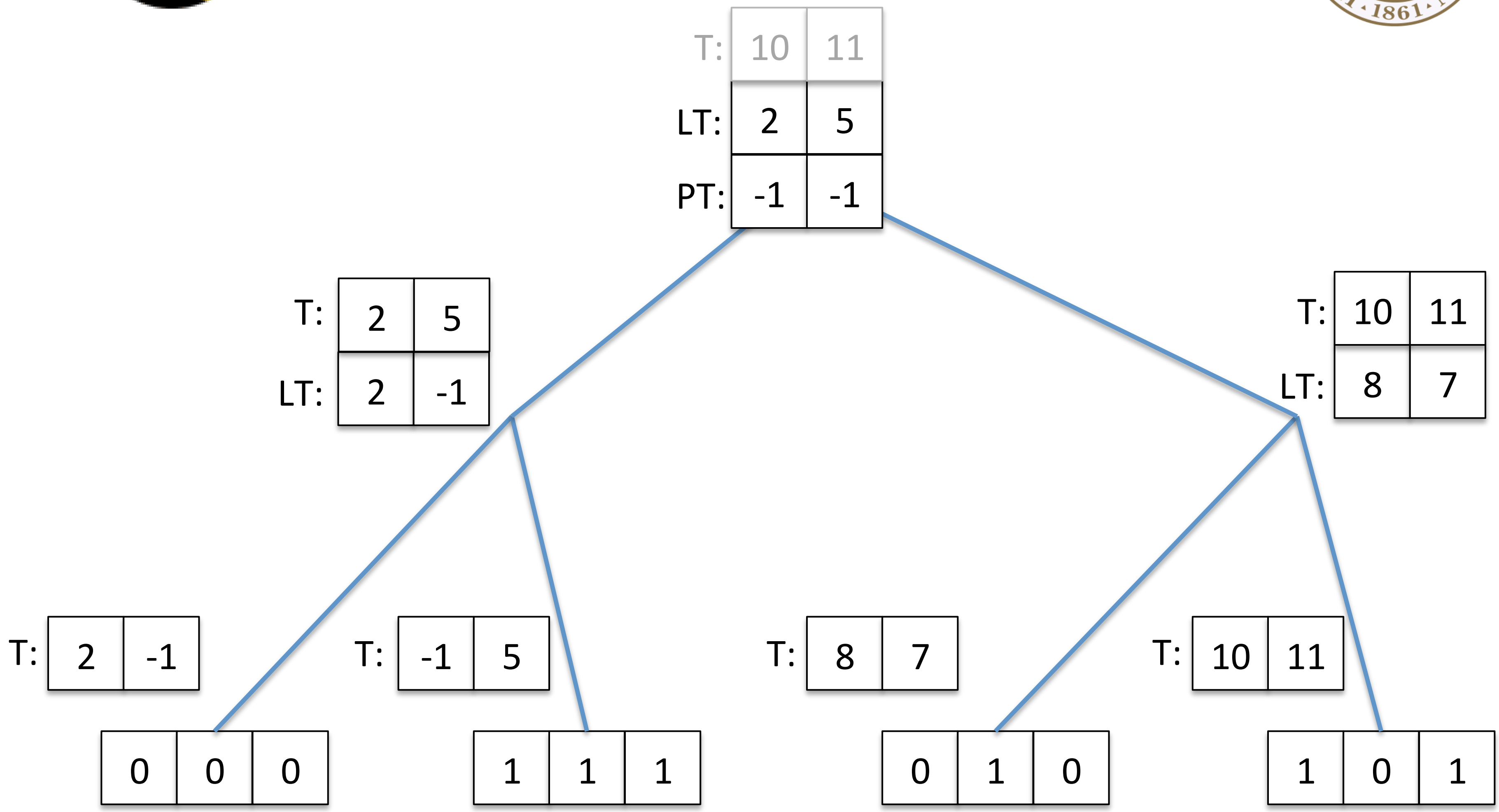
Illustration



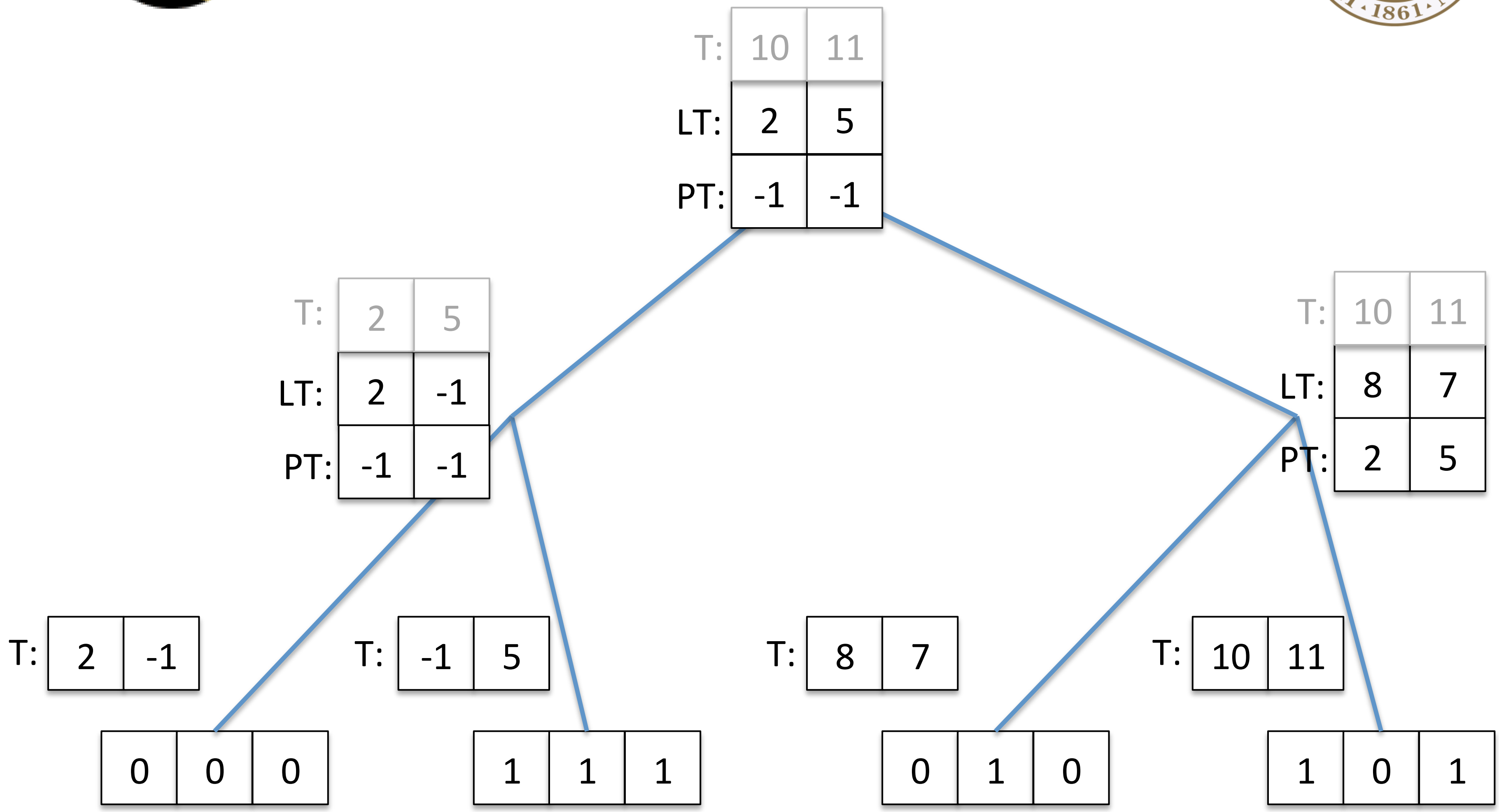
Illustration



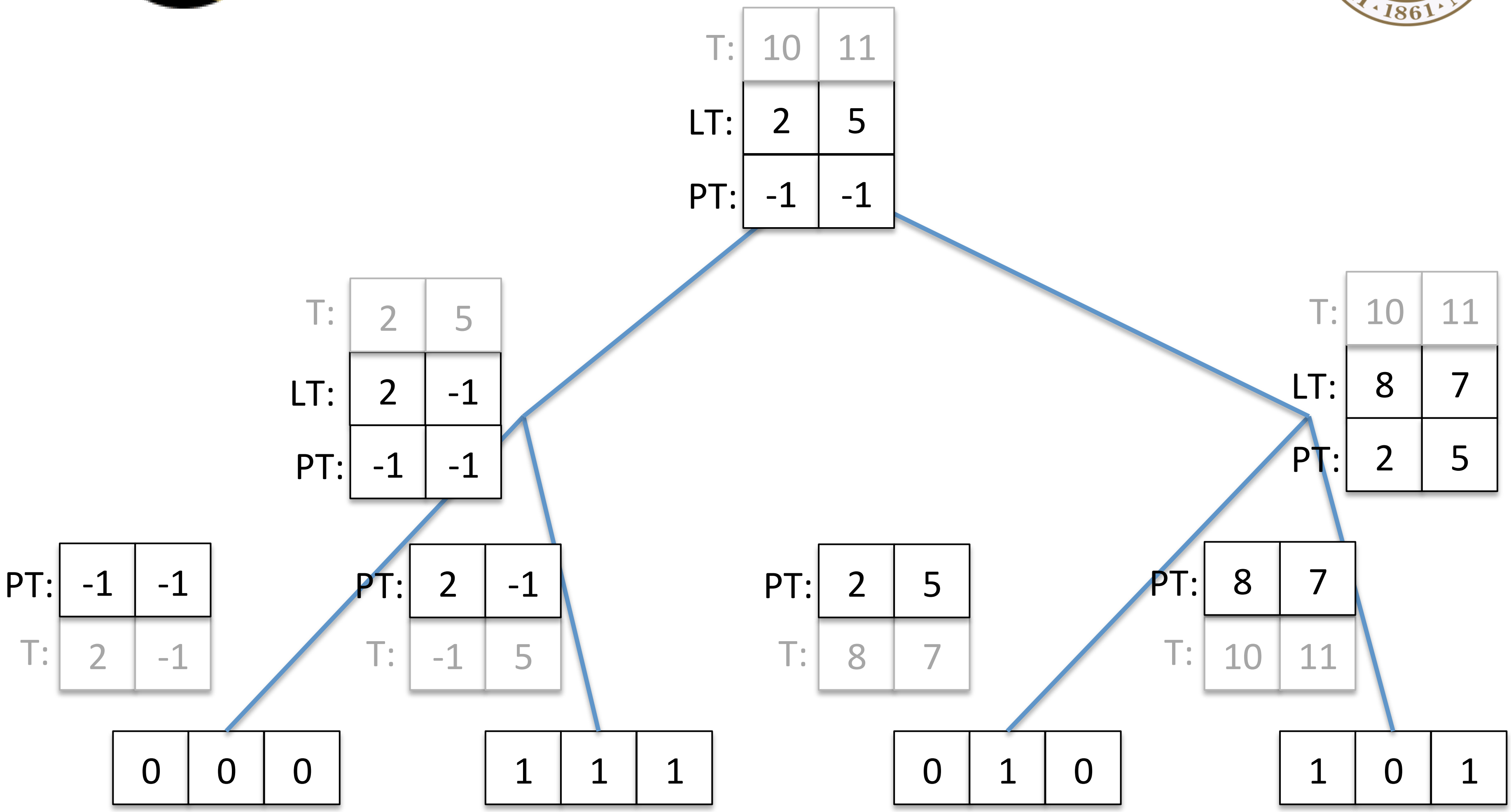
Illustration



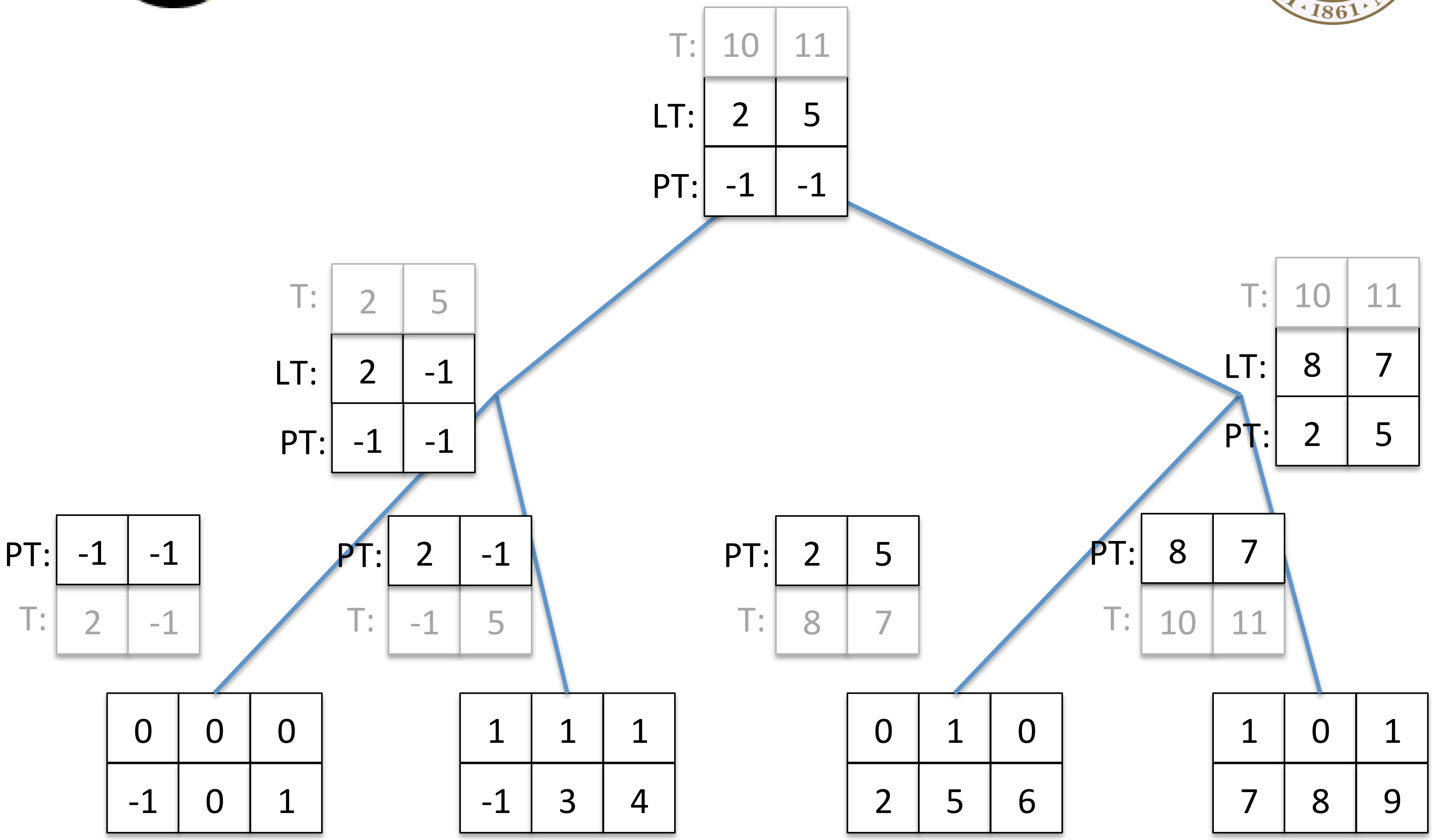
Illustration



Illustration



Illustration



What's the idea?

- Many computations can be reformulated as reduces or scans
- You can then apply these techniques + Schwartz's algorithm as a recipe for solving them in parallel
- Some high-level parallel languages have built-in support for this concept – e.g., Chapel
  - Still valuable to understand how it could be done

Discussion Session

- What did you think of the paper, and the MapReduce paradigm?
  - Flexibility? Can you implement everything you'd want to?
  - Ease of use?
  - Robustness?
- We've reached the midpoint of the class, and will be switching gears, to cover languages and more "applied" topics.
  - Anything specifically you want to see covered (no promises, but I'm open to suggestions)
  - Any thoughts about what we've learned, and the papers you've read?