Part V: Algorithms & Data Structs

Goal: Focus more closely on scalable parallel techniques, both computation and data
Notice on the calendar that next week’s class (normally 5/4) is rescheduled for Thursday (5/6), same time, same place.
Commentary on Homework

- Are there any further comments on the Red/Blue thread program?
- How was the Peril-L sample sort exercise?
  - Randomizing
  - Finding Cut-points
  - Global Exchange
  - Scooch
Recall from last week … the balanced ( ) code

```c
6    for (i=start; i<start+len_per_th; i++) {
7        temp = symb[i];
8        if (temp == "(" )
9            o++;
10       if (temp == ")")
11          o--;
12       if (o < 0) {
13            c++; o = 0;
14        }
15    }
```

The question was raised, could we move symb[i] into a local variable before the if’s
Can it?

- The answer was ‘yes, though a modern compiler could do this for us’
- That answer’s correct, but I missed the opportunity to say why
  - This move would not be legal in our assumed sequentially consistent shared memory model UNLESS the compiler could establish the global fact that the array is read only
  - It is legal in the Peril-$L$ model, which has no coherency commitments at all
Reconceptualizing a Computation

- Good parallel solutions result from rethinking a computation...
  - Sometimes that amounts to reordering scalar operations
  - Sometimes it requires starting from scratch
- The SUMMA matrix multiplication algorithm is the poster computation for rethinking!

This computation is part of homework assignment
Matrix Multiplication on Processor Grid

- Matrices $A$ and $B$ producing $n \times n$ result $C$ where
  
  \[ C_{rs} = \sum_{1 \leq k \leq n} A_{rk} \times B_{ks} \]
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?

\[
P_0 = \sum_{i=1}^{n} P_i + \sum_{j=1}^{n} P_j + \ldots + \sum_{k=1}^{n} P_k
\]
If all rows/columns are present, it’s local

- Each element requires $O(n)$ operations
- Modern pipelined processors benefit from large blocks of work
- But memory space and BW are issues
Process \( t \times t \) Blocks

- Use that solution, but incrementally
- Referring to local storage

```c
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];
        }
    }
}
```

Only move a \( t \times t \) block at a time

Sweeter caching
Don’t think of row-times-column

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

- Scalable Universal Matrix Multiplication Alg
  - Invented by van de Geijn & Watts of UT Austin
  - Claimed to be the best machine independent MM
- 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

Strange. But fast!
Threads have two indices, handle $t \times t$ block

Let $p = P^{1/2}$, then thread $u,v$

- reads all columns of $A$ for indices $u*t:(u+1)*t-1,j$
- reads all rows of $B$ for indices $i,v*t:(v+1)*t-1$
- The arrays will be in “global” memory and referenced as needed
Higher Level SUMMA View

- See SUMMA as an iteration multicasting columns and rows
- Each processor is responsible for sending/recving its column/row portion at proper time
- Followed by a step of computing next term locally

www.cs.utexas.edu/users/rvdg/abstracts/SUMMA.html
Summary of SUMMA

- Facts:
  - vdG & W advocate blocking for msg passing
  - Works for $A$ being $m \times n$ and $B$ being $n \times p$
  - Works fine when local region is not square
  - Load is balanced esp. of Ceiling/Floor is used

- Fastest machine independent MM algorithm!
- Key algorithm for 524: Reconceptualizes MM to handle high $\lambda$, balance work, use BW well, exploit efficiencies like multicast, ...
Jack Schwartz (NYU) asked: What is the optimal number of processors to combine \( n \) values?

- Reasonable Answer: binary tree w/ values at leaves has \( O(\log n) \) complexity
- To this solution add \( \log n \) values into each leaf
- Same complexity (\( O(\log n) \)), but \( n\log n \) values!
- Asymptotically, the advantage is small, but the tree edges require communication
Jack Schwartz (NYU) asked: What is optimal number of processors to combine $n$ values?

- Reasonable Answer: binary tree w/ values at leaves has $O(\log n)$ complexity
- To this solution add $\log n$ values into each leaf
- Same complexity ($O(\log n)$), but $n\log n$ values!
- Asymptotically, the advantage is small, but the tree edges require communication
Generally $P$ is not a variable, and $P << n$

Use **Schwartz as heuristic**: Prefer to work at leaves (no matter how much smaller $n$ 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 -- in cases of parallel communication (CTA) some of the communication may overlap
The Red/Blue computation illustrated a 2D-block data parallel allocation of the problem. Generally block allocations are better for data transmission: surface to volume advantage ... since only edges are x-mitted.
Different Regimens

- Though block is generally a good allocation it’s not absolute:

  P=1, all comm wasted
  
  P=2, row-wise saves column comm
  
  P=4, rows and blocks are a wash
  
  vs
  
  Where is the point of dim. return?
To simplify local computation in cases where nearest neighbor’s values x-mitted, allocate in-place memory (fluff) to store values:

- Array can be referenced as if it’s all local
Generally $P$ and $n$ do not allow for a perfectly balanced allocation ...

Several ways to assign arrays to processors

- **Quotient + remainder**
- **Ceiling + floor**
- **Generally a small effect**

13x13 on 4x4 process array
Assigning Processor o Work

- $p_0$ is often assigned “other duties”, such as
  - Orchestrate I/O
  - Root node for combining trees
  - Work Queue Manager ...
- Assigning $p_0$ the smallest quantum of work helps it avoid becoming a bottleneck
  - For either quotient + remainder or ceiling/floor $p_0$ should be the last processor

This is a late-stage tuning matter
Array computations on CMPs

- Dense Allocation vs Fluff
- Issue is cache invalidation
- Keeping MM managed intermediate buffers keeps array and fluff local (L1)
- Sharing causes elements at edge to repeatedly invalidate harming locality

False sharing an issue, too
Load Balancing

- Certain computations are inherently imbalanced ... LU Decomposition is one
  - gray is balanced work, white & black are finished
- Standard block decomposition quickly becomes very biased
  - Cyclic and block cyclic allocation are one fix
Cyclic allocation means “to deal” the elements to the processes like cards

- Allocating 64 elements to five processes: black, white, three shades of gray

- Block cyclic is the same idea, but rather with regular shaped blocks
Consider the LU matrix allocated in $3 \times 2$ blocks to four processes:

Then check it midway in the computation.
Opportunities To Apply Cyclic

- The technique applies to work allocation as well as memory allocation

Julia Set from http://alepho.clarku.edu/~djoyce/
Generalized Reduce and Scan

- The importance of reduce/scan has been repeated so often, it is by now our mantra
- In nearly all languages the only available operators are $+, \times, \min, \max, \&\&, | |$
- The concepts apply much more broadly
- Goal: Understand how to make user-defined variants of reduce/scan specialized to specific situations

Seemingly sequential looping code can be UD-scan
Recall scan specifics

+ scan of: 1 2 3 4 5 6 7 8

is either: 1 3 6 10 15 21 28 36 [inclusive]

or it is: 0 1 3 6 10 15 21 28 [exclusive]

Important fact about standard scans

\( \alpha - \text{scan}_{\text{inclusive}}(x) = \alpha - \text{scan}_{\text{exclusive}}(x) \alpha x \)

For technical reasons prefer exclusive, for today, think inclusive
Examples Applicable Computations

- **Reduce**
  - Second smallest, or generally, kth smallest
  - Histogram, counts items in k buckets
  - Length of longest run of value 1s
  - Index of first occurrence of x

- **Scan**
  - Team standings
  - Find the longest sequence of 1s
  - Index of most recent occurrence

**Associativity, but not commutativity, is key**
Structure of Computation

- Begin by applying Schwartz idea to problem
  - Local computation
  - Global $\log_d P$ tree

More computation at nodes is OK
Recall Parallel Prefix Algorithm

- Compute sum going up: reduce
- Compute prefixes going down

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

```
Compute sum going up: reduce
Compute prefixes going down
```

```
Introduce a virtual parent, the sum of values to tree’s left: 0
```
Parallel Prefix Algorithm

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

Compute sum going up: reduce
Compute prefixes going down
Parallel Prefix Algorithm

Compute sum going up
Figure prefixes going down

Invariant: Parent data is sum of elements to left of subtree
Parallel Prefix Algorithm

Compute sum going up
Figure prefixes going down

Invariant: Parent data is sum of elements to left of subtree
Each prefix is computed in $2\log n$ time, if $P = n$
Make four non-communication operations

- `init()` initialize the reduce/scan
- `accum()` perform local computation
- `combine()` perform tree combining
- `x_gen()` produce the final result for either op
  - `x = reduce`
  - `x = scan`

Incorporate into Schwartz-type logic

Think of: `reduce(fi, fa, fc, fg)`
Assignments of Functions

- **Init:** Each leaf
- **Accum:** Aggregate each array value
- **Combine:** Each tree node
- **reduceGen:** Root
Example: +<<<A Definitions

- Sum reduce uses a temporary value, called a tally, to hold items during processing.
- Four reduce functions:
  - `tally init()` {tal = new tally; tal=0; return tal;}
  - `tally accum(int op_val, tally tal)` {tal += op_val; return tal; }
  - `tally combine(tally left, tally right)` {return left + right; }
  - `int reduce_gen(tally ans)` {return ans;}

© 2010 Larry Snyder, CSE
Consider Second Smallest -- useful, perhaps for finding smallest nonzero among non-negative values

- **tally** is a struct of the smallest and next smallest found so far {float sm, nsm}

- Four functions:
  ```
tally init() {
    pair = new tally;
    pair.sm = maxFloat;
    pair.nsm = maxFloat;
    return pair;
  }
```
Accumulate

tally accum(float op_val, tally tal) {
    if (op_val < tal.sm) {
        tal.nsm = tal.sm;
        tal.sm = op_val;
    } else {
        if (op_val > tal.sm && op_val < tal.nsm)
            tal.nsm = op_val;
    }
    return tal;
}
tally combine(tally left, tally right) {
  return
  accum(left.nsm, accum(left.sm, right));
}

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

- Notice that the signatures are all different
- Conceptually easy to write equivalent code, but reduction abstraction clarifies
PoPP presents the state of the art of user-defined scans.

The conclusion must be, that generally it is:
- inconvenient, cumbersome, difficult
- requires low-level knowledge and interface

But, custom scan has wide application.

Take a moment to think “outside the box” on adding UD Scan to a programmer’s tool belt.
Because the definition of the computation is in terms of prefixes we usually see scan as a *sequential left to right operation*.

But studying the implementational or compiler view of the computation, we notice ...

> From the backbone logic of the tree evaluation that the crux is combining adjacent sequences.
The Main Idea

Add scan to languages with semantics of a *user defined* INFIX operator rather than as a LEFT ASSOCIATIVE operator, i.e. prefer

\[ (((\oplus) \oplus (\oplus)) \oplus ((\oplus) \oplus (\oplus)) \]

to

\[ ((((\oplus) \oplus (\oplus) \oplus (\oplus) \oplus (\oplus)) \oplus (\oplus) \oplus (\oplus)) \]

\[ \oplus (\oplus) \oplus (\oplus) \]

Accordingly, think of the operation as

\[ x_r \ldots x_s \oplus x_{s+1} \ldots x_t \]

where

- the sequences are contiguous
- begin anywhere, end anywhere
- any nonzero length

Additionally, think about

- The data to be merged from the two halves
- The basis case starting with initial data
- The completion processing
Consequences of $\oplus$ view

- To make the new view concrete, notice that
  - The substrings need a descriptor for state: **tally**
  - The basis case is an initial tally value: $\text{Initial}(\text{inval}_i)$ in each position $i$
  - The result of $x_1 \ldots x_s \oplus x_{s+1} \ldots x_n$ is the root value of the implementation tree, but the computation may not be finished [down sweep] implying that there is a finalize step: $\text{outval}_i = \text{Final}( )$
  - Defining the tally, $\text{Initial}( )$, $l_{\text{tally}} \oplus r_{\text{tally}}$ and $\text{Finalize}()$ suffices
Three Parts of + reduce

- The tally is a single float
  
  **Initialize:**
  
  - float tally = inval;

  **Complete:**
  
  - outval = tally;

  **Combine:** ltally ⊕ rtally
  
  - float tally = ltally + rtally;
Three Parts of + Scan

Initialize [each item in sequence]:
- pair tally = new Pair()  //descriptor is a pair
- float tally.pre = 0; float tally.sum = inval;  //initialize

Complete [each item in sequence]:
- outval = tally.pre + tally.sum  //final output

Combine: ltally \oplus rtally
- pair tally = new Pair()  //describe combin’n
- float tally.pre = ltally.pre;  //prefix is left prefix
- float tally.sum=ltally.sum+rtally.sum;  //sum is left+right
- THEN: ltally.pre = tally.pre;  //left prefix is prefix
- rtally.pre = tally.pre+left.sum  //right is prefix+l.sum
Three Parts of +scan [cartoon]

tally –
pre:  o
sum: inval
Three Parts of +scan [combine]

\[
\begin{array}{cccccc}
3 & 7 & -2 & 8 & \oplus & 5 \\
3 & 7 & -2 & 8 & 5 & 3 \\
3 & 7 & -2 & 8 & 5 & 3 \\
\end{array}
\]

\[
\begin{array}{cccccc}
3 & 7 & -2 & 8 & \oplus & 5 \\
3 & 7 & -2 & 8 & 5 & 3 \\
3 & 7 & -2 & 8 & 5 & 3 \\
\end{array}
\]

\[
\begin{array}{cccccc}
3 & 7 & -2 & 8 & \oplus & 5 \\
3 & 7 & -2 & 8 & 5 & 3 \\
3 & 7 & -2 & 8 & 5 & 3 \\
\end{array}
\]
## Three Parts of `+scan` [downsweep]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>-2</td>
<td>8</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>-2</td>
<td>8</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>tally –</td>
<td>tally –</td>
<td>tally –</td>
<td>tally –</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pre: 0</td>
<td>pre: 0</td>
<td>pre: 100</td>
<td>pre: 100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum: 38</td>
<td>sum: 16</td>
<td>sum: 22</td>
<td>sum: 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tally –</td>
<td>tally –</td>
<td>tally –</td>
<td>tally –</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pre: 0</td>
<td>pre: 0</td>
<td>pre: 100</td>
<td>pre: 116</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum: 16</td>
<td>sum: 22</td>
<td>sum: 16</td>
<td>sum: 22</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Three Parts of +scan [final]

\[\text{outval} = \text{pre} + \text{sum}\]

tally –
pre: 103
sum: 7

\[
\begin{array}{ccccccccccc}
3 & 7 & -2 & 8 & \oplus & 5 & 3 & 6 & 4 & 2 & 2 \\
103 & 110 & 108 & 116 & 121 & 124 & 130 & 134 & 136 & 138 \\
\end{array}
\]
Parts of + Scan

Initialize [each item in sequence]:

- pair tally = new Pair() //descriptor is a pair
- float tally.pre = 0; float tally.sum = inval; //initialize

Complete [each item in sequence]:

- outval = tally.pre + tally.sum //final output
Parts of + Scan

Initialize [each item in sequence]:

- pair tally = new Pair()  //descriptor is a pair
- float tally.pre = 0; float tally.sum = inval;  //initialize

Complete [each item in sequence]:

- outage = tally.pre + tally.sum  //final output

Combine: ltally ⊕ rtally

- pair tally = new Pair()  //describe combin’n
- float tally.pre = ltally.pre;  //prefix is left prefix
- float tally.sum=ltally.sum+rtally.sum;  //sum is left+right
- THEN: ltally.pre = tally.pre;  //left prefix is prefix
- rtally.pre = tally.pre+left.sum  //right is prefix+l.sum

© 2010 Larry Snyder, CSE
Another Ex.: Longest Run of x

- How do we think of this computation as combining two subcomputations

  \[ xx0000x0xxxx \oplus x0xxxxxxxx000 \]

- Obviously
  - x runs can be at the start, interior, or end
  - Combining will merge a start and end run
  - ... Making it an interior run

- The tally needs to keep this information
Longest Run of x [a *reduce* cartoon]

```
tally - in == x
  from start: 1
  inside: 0
  from end: 1

xx0000x0xxxxx ⊕ x0xxxxxxxx000
xx0000x0xxxxxx0xxxxxxxx000
```
Longest Run of x [a reduce cartoon]

tally – in == x
from start: 1
inside: 0
from end: 1

tally – in != x
from start: 0
inside: 0
from end: 0

\[ xx0000x0xxxx \oplus x0xxxxxxx000 \]
\[ xx0000x0xxxxx0xxxxxxx000 \]
Longest Run of x [a *reduce* cartoon]

\[
\begin{align*}
tally -- \\
from start: & 2 \\
inside: & 1 \\
from end: & 4
\end{align*}
\]

\[
\begin{align*}
tally -- \\
from start: & 1 \\
inside: & 6 \\
from end: & 0
\end{align*}
\]

\[
xx0000x0xxxx \oplus x0xxxxxxx000
\]

\[
xx0000x0xxxxx0xxxxxx000
\]
Longest Run of x [a reduce cartoon]

\[
\begin{align*}
\text{tally --} & \\
\text{from start: 2} & \\
\text{inside: 6} & \\
\text{from end: 0} & \\
\end{align*}
\]

\[
xx0000x0xxxx \oplus x0xxxxxxxx000
\]

\[
xx0000x0xxxxx0xxxxxxxx000000
\]
Longest Run of $x$ [a *reduce* cartoon]

tally --
from start: 2
inside: 6
from end: 0

$\text{outval}$

$\text{max}$

$xx0000x0xxxx \oplus x0xxxxxxx000$

$xx0000x0xxxxx0xxxxxx000$
Illustrate for the matching parentheses
- Carry along the count of excess of opens/closes
- Cancel if matched, else record the excess
- Output “yes” if excess is 0

Descriptor for “balanced parens” is two ints, excess open parens \( \text{opCount} \) and excess closed parents \( \text{clCount} \)
A || Prefix Solution

- Visualize a processor per point (not really)
  - Each point is initialized to its data structure
  - Pairs are combined in some way
  - Process continues until there is one descriptor
  - Compute the final result
- Illustrate on this problem:

\[
a - f(c) \times (d + f(e))
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th></th>
<th></th>
<th>1</th>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Create a tally:
if (inval == '(' )
    int tally.opCount = 1;
else
    int tally.opCount = 0;
if (inval == ')') {
    int tally.clCount = 1;
else
    int tally.clCount = 0;

Combine two tallies:
tally.clCount = ltally.clCount;
tally.opCount = rtally.opCount;
int temp = ltally.opCount - rtally.clCount;
if (temp < 0)
    tally.clCount += abs(temp);
else
    tally.opCount += temp;

Finalize result from tally:
outval = (tally.opCount == 0) && (tally.clCount == 0);
Matching Parens

- Working out the details

Matching

\[
a - f(c) * (d + f(e))
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}
\]
Matching Parentheses

- Working out the details
Matching Parentheses

- Working out the details

Matching Parentheses

\[
\begin{align*}
\text{a} &\quad -\quad f(\text{c})\quad \ast\quad (\text{d} &\quad +\quad f(\text{e})) \\
0 &\quad 0 &\quad 0 &\quad 1 &\quad 0 &\quad 0 &\quad 0 &\quad 1 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 1 &\quad 1 \\
0 &\quad 0 &\quad 0 &\quad 0 &\quad 1 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 1 &\quad 1 \\
\text{a} &\quad -\quad f(\text{c})\quad \ast\quad (\text{d} &\quad +\quad f(\text{e})) \\
0 &\quad 1 &\quad 0 &\quad 1 &\quad 0 &\quad 1 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 0 &\quad 1 &\quad 1 &\quad 1
\end{align*}
\]
### Matching Parentheses

- **Working out the details**

<table>
<thead>
<tr>
<th>a - f(c) * ( d + f(e) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 0 0 0 1 0 0 0 1 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a - f(c) * ( d + f(e) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 0 1 0 0 0</td>
</tr>
<tr>
<td>0 0 1 0 0 0 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a - f(c) * ( d + f(e) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
</tr>
<tr>
<td>0 1 0 2</td>
</tr>
</tbody>
</table>
Matching Parentheses

- Working out the details

\[
\begin{array}{cccccccc}
    a - f(c) \times (d + f(e)) \\
 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 \\
 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 \\
    a - f(c) \times (d + f(e)) \\
 0 1 0 1 0 1 0 0 0 1 0 0 1 1 1 \\
 0 0 1 0 0 0 0 1 1 1 \\
    a - f(c) \times (d + f(e)) \\
 1 1 1 0 0 2 \\
 0 1 0 2 \\
    a - f(c) \times (d + f(e)) \\
 1 0 1 \\
 0 1 \\
    a - f(c) \times (d + f(e)) \\
 0 0 \\
 0
\end{array}
\]
Matching Parens

- Working out the details
  Mismatching
Matching Parentheses

- Working out the details

Mismatching

\[
\begin{align*}
\text{a - f)} \text{ c) * ( d + f( e ) )}
\end{align*}
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

© 2010 Larry Snyder, CSE
Matching Parentheses

- Working out the details

Mismatching

\[
(a - f) c) * (d + f(e))
\]

0 0 0 0 0 0 0 1 0 0 0 1 0 0 0
0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 1

Mismatching

\[
(a - f) c) * (d + f(e))
\]

0 0 0 0 1 0 1 0 0 0 1 1
0 1 1 0 0 0 1 1

Mismatching

\[
(a - f) c) * (d + f(e))
\]

0 1 1 0 0 1 0 2
Matching Parenthesis

- Working out the details

Mismatching:

```
\[ a - f ) c ) * ( d + f ( e ) ) \]
```

```
\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 2 \\
1 & 1 & 0 & 0 & 2 & 1 \\
1 & 1 & 0 & 0 & 1 & 2 \\
\end{array}
\]
```

4/27/2010

© 2010 Larry Snyder, CSE
One last question concerned how the 3 parts of the || prefix specification fit into the tree model shown for prefix sum & Schwartz?

- Short answer, they don’t have to
- Compilers can produce excellent code from spec
At the start of class we cited bal-parens – the leaf code for a Schwartz approach

```c
for (i=start; i<start+len_per_th; i++) {
    if (symb[i] == "(" )
        o++;
    if (symb[i] == ")") {
        o--;
        if (o < 0) {
            c++; o = 0;
        }
    }
}
```

Combining required entirely different code

The Infix approach captures the whole thing, except for pre- and post-operations
By thinking abstractly of carrying along information that describes the sequence, combining adjacent subsequences, and finally extracting a value, it is possible to move directly to a $\|\ $ prefix solution.

Using the abstraction is an intellectually different way of thinking about sequential computations.
Think of a “sequential computation” that can be expressed as a UD reduce or scan
- Examples from this lecture are off limits
- Prefer a scan; it’s often easy to convert a reduce into a scan: A 10-bucket histogram (a reduce) is related to a 10-team “league standings” (a scan) that gives won/loss for game input, team t beat u
- Turn in a document giving an infix formulation of the computation together with a worked example
Write an MPI program for the SUMMA alg

- Create rectangular arrays A, B, C, filling A, B
- Send portions of A, B to worker processes
- Iterate over common dimension,
  - send columns of A, rows of B to other processes
  - for each, multiply A elements times B elements and accumulate into local portion of C
- Measure time, except for initialization, and report the “usual stuff” for different numbers of processes