35 Years of Research: + Results; - Results

Lawrence Snyder

www.cs.washington.edu/homes/snyder

9 April 2009

The Situation Today …

- We have ~35 years of research results to draw upon --
  - Many, who have worked in the area, think much of it is important
  - Many, who have NOT worked in the area, think none of it is useful
  - Many, who are encountering parallelism now for the first time, assume nothing came before
- Where is the truth?

Outline Of This Talk

- In The Beginning: Let The Compiler Do It
- Parallel Prefix Abstraction
- The Parallel Programming Problem
- A Parallel Machine Model
- Maybe An Easier Model Would Be Better
- Applying CTA Model In Programming -- Derive an Algorithm

Parallel Programming State-of-the-Art

- The “parallel programming problem” has been studied for 35+ years and we learned:
  - Parallelism is best when it’s invisible
  - Parallelizing compilers do not suffice (more later)
  - Hardware has not yet given SW sufficient support
  - “Silver Bullet” programming approaches do not suffice
  - “Blue Collar” parallel programs are built with library-based tools: MPI/PVM, threads, OpenMP

The First Approach Was Easiest

- The original plan for programming Illiac IV, was to have a compiler process Fortran IV programs, producing parallel code
- Dave Kuck led the Illinois team
  - Extremely Ambitious
    - Fortran IV programs notoriously badly structured
    - Compilation technology still very primitive
    - Program analysis techniques nearly non-existent
  - Ultimately, “production programs” written in assembler
- Kuck + many others continued to pursue the goal:

```
parallel_code = compilation(serial_code)
```

One (Of Many) Problems

- Loop carried dependences can get in way
- Decide if parentheses are balanced

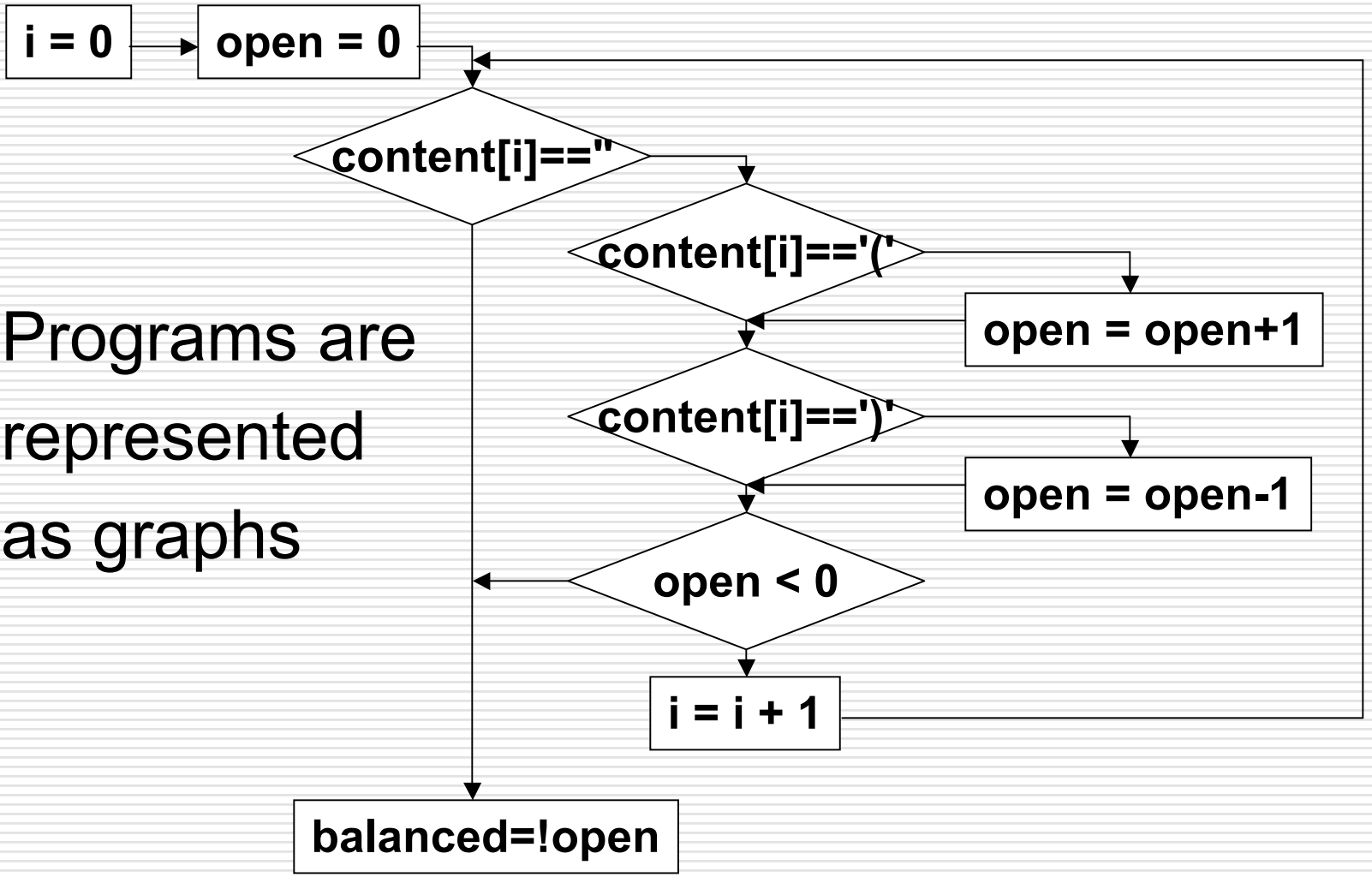
```
i = 0;
open = 0;
while (content[i] != '') {
   if (content[i] == '(')
      open = open + 1;
   if (content[i] == ')')
      open = open - 1;
   if (open < 0)
      break;
   i++;
}
balanced = ! open;
```

```
content: a-f(c)*(d+f(e))
   open:0000110011112210
```

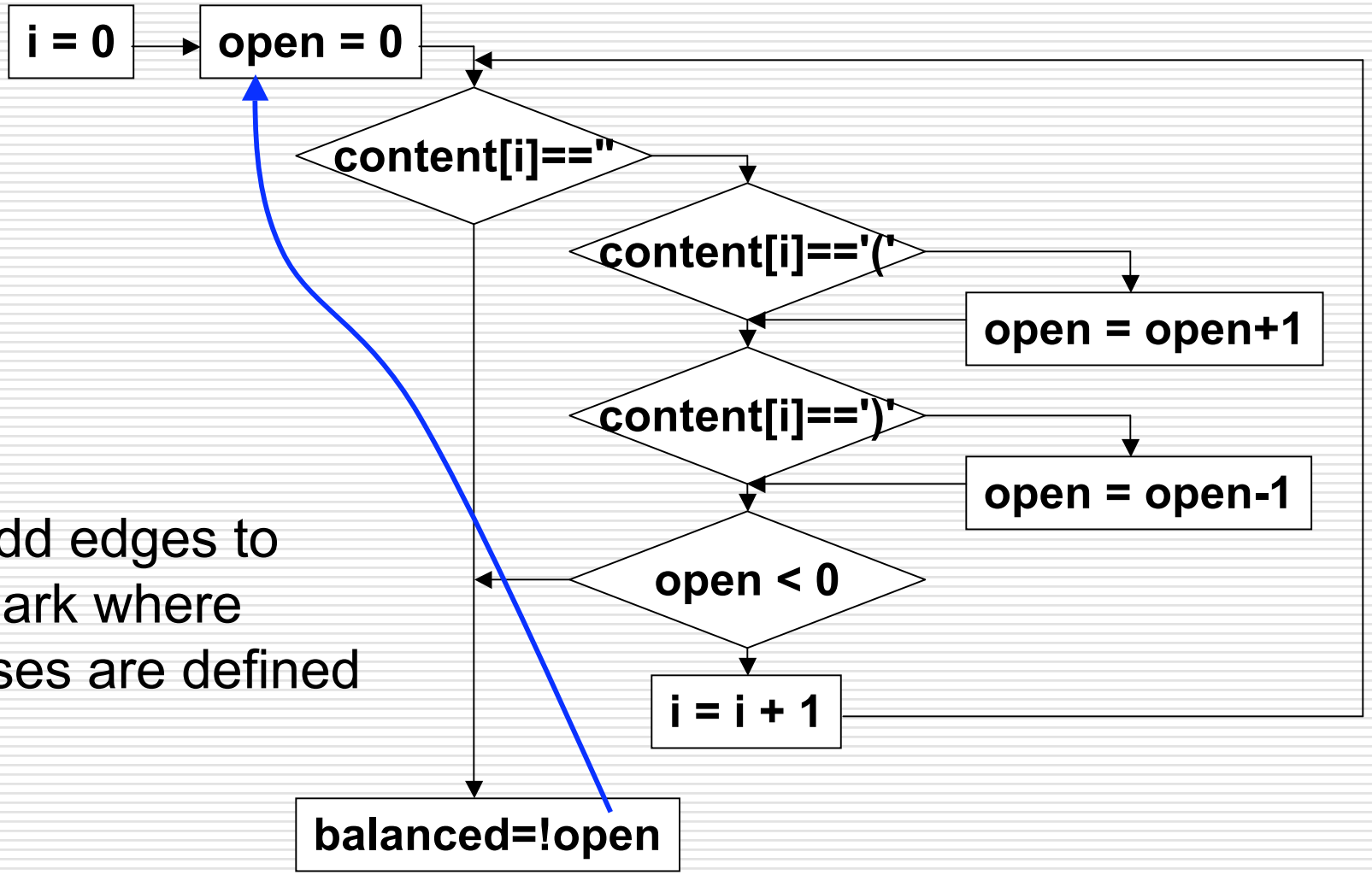
```
content: a-f)c)*(d+f(e))
   open:0000-1
```

Compiling: Graph Abstractions

Programs are represented as graphs

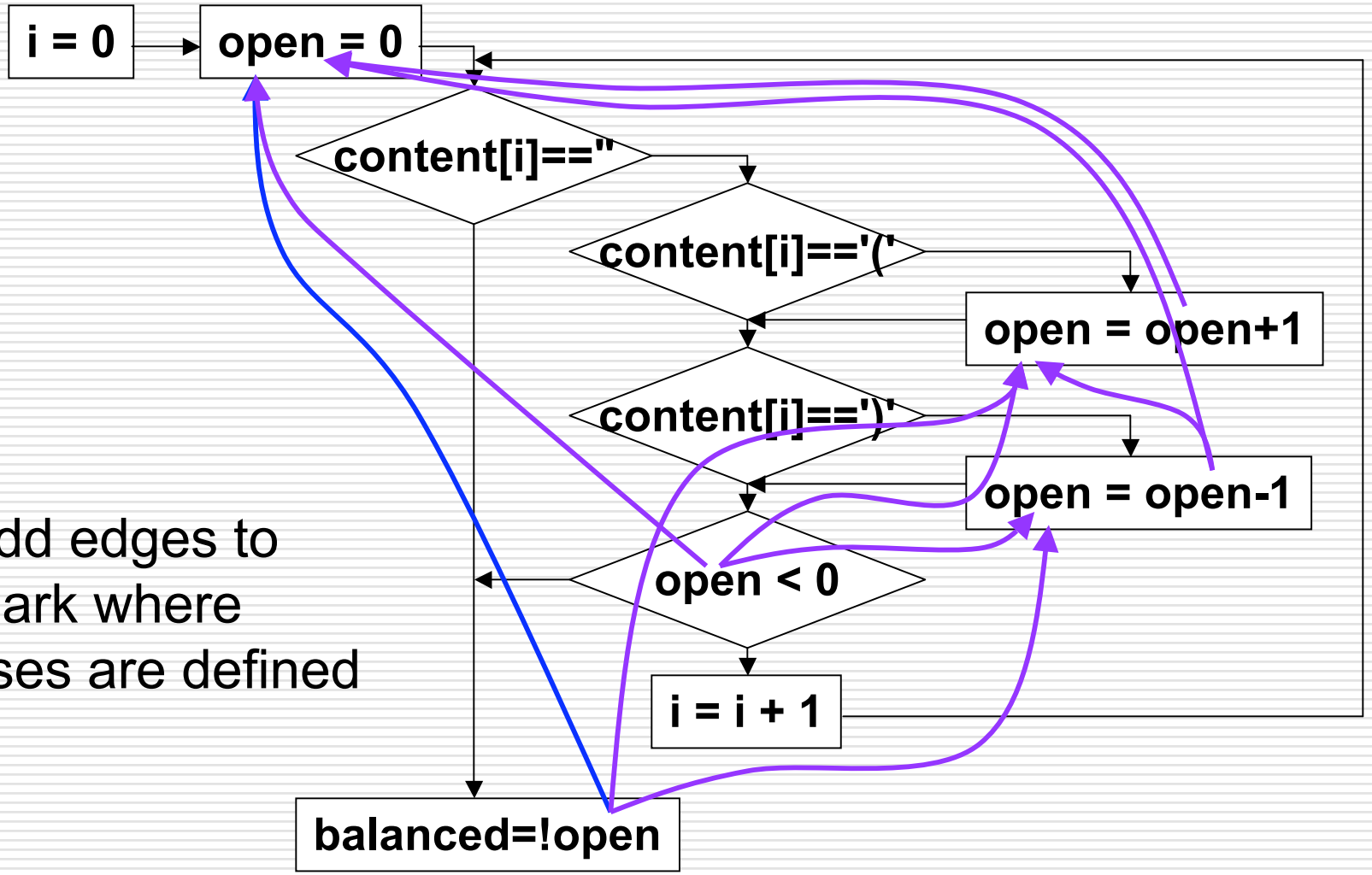


Compiling: Iterations - 0



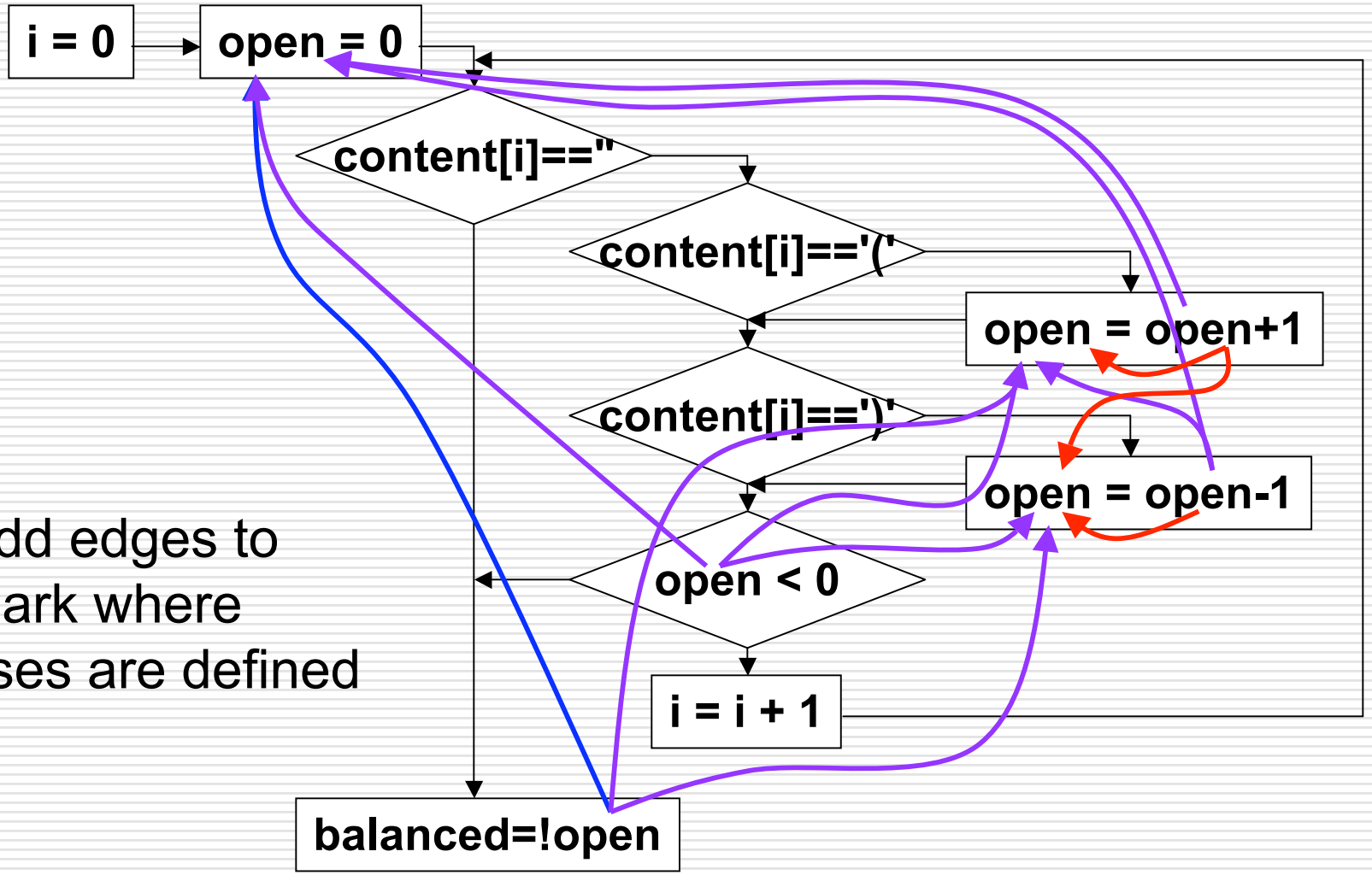
Add edges to mark where uses are defined

Compiling: Iterations - 1



Add edges to mark where uses are defined

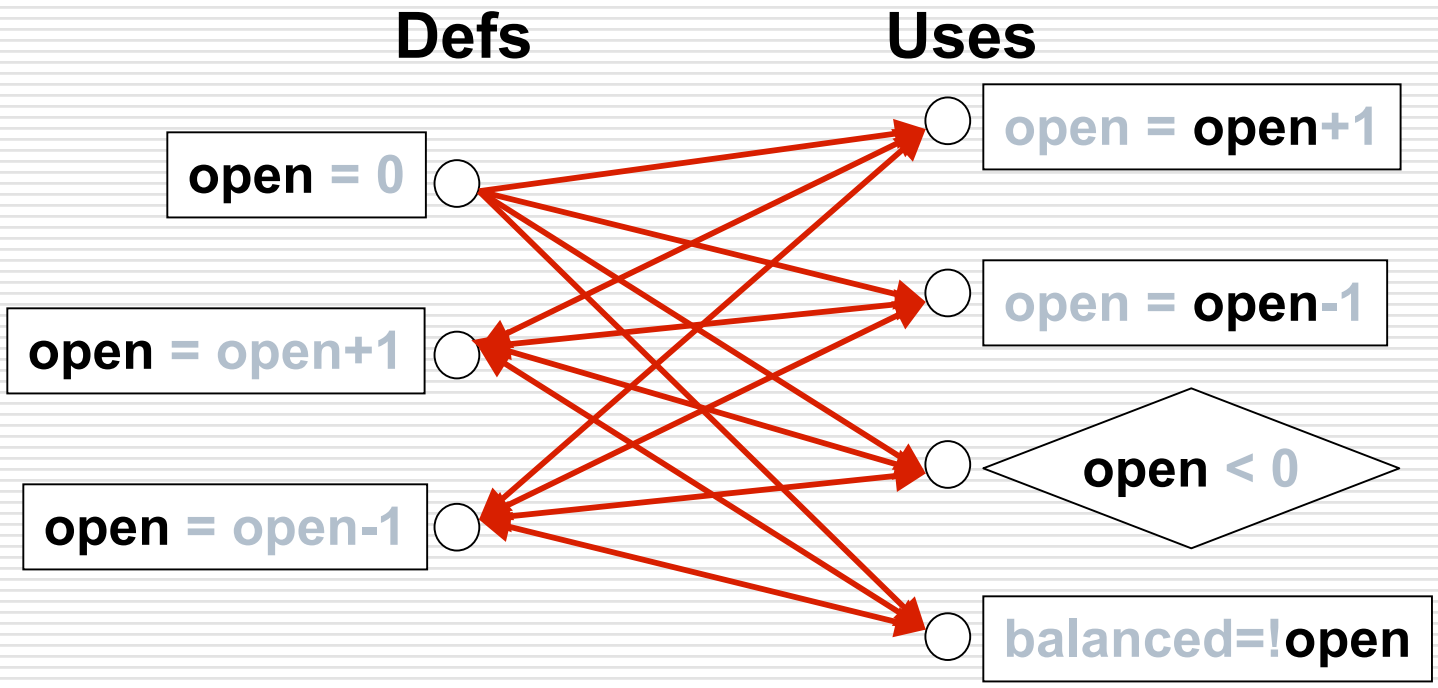
Compiling: Iterations - 2



Add edges to mark where uses are defined

Difficult To Analyze

- The computation is embodied in the “flow dependences” and the truth of predicates that specify it … it is difficult to analyze



After 35 years of research …

- In 2006 Kuck summed up the situation at a workshop, “I still think it [parallelizing sequential code] is possible, but it can’t be done yet.”
- In the intervening years
  - Architectures kept changing … a problem; not serious
  - Compiler transformations “worked” for vector machines
  - Enormous funding was available
  - Many of best researchers worked on the problem
  - Program analysis became a sophisticated technology
  - Source languages got much better, more structured
  - Progress was made in many aspects of the task
  - Commercial products were deployed

Why Hasn't It Worked?

- In many restricted cases it has worked … but those cases are not sufficient to compile "general" programs
- I think the problem is fundamentally not solvable based on the observation:
  - The best sequential program and the best parallel program for a task tend to be different
  - The difference is not a simple transformation or series of transformations
  - The two programs embody different solution paradigms

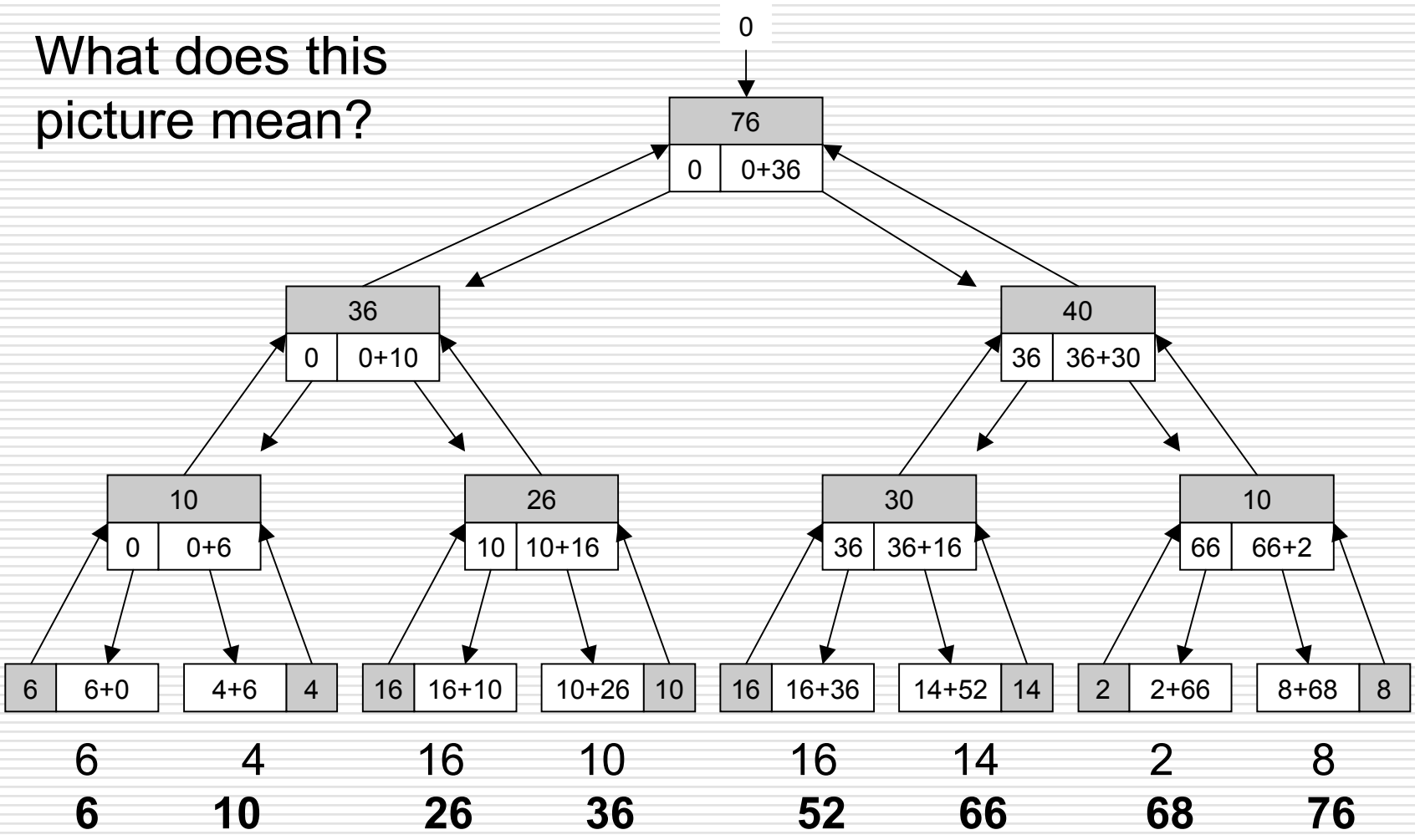
This seems to be a fundamental barrier

Outline Of This Talk

- In The Beginning: Let The Compiler Do It
- Parallel Prefix Abstraction
- The Parallel Programming Problem
- A Parallel Machine Model
- Maybe An Easier Model Would Be Better
- Applying CTA Model In Programming -- Derive an Algorithm

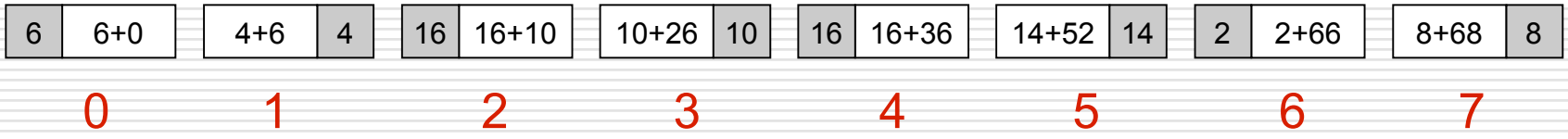
Recall Parallel Prefix Algorithm

What does this picture mean?



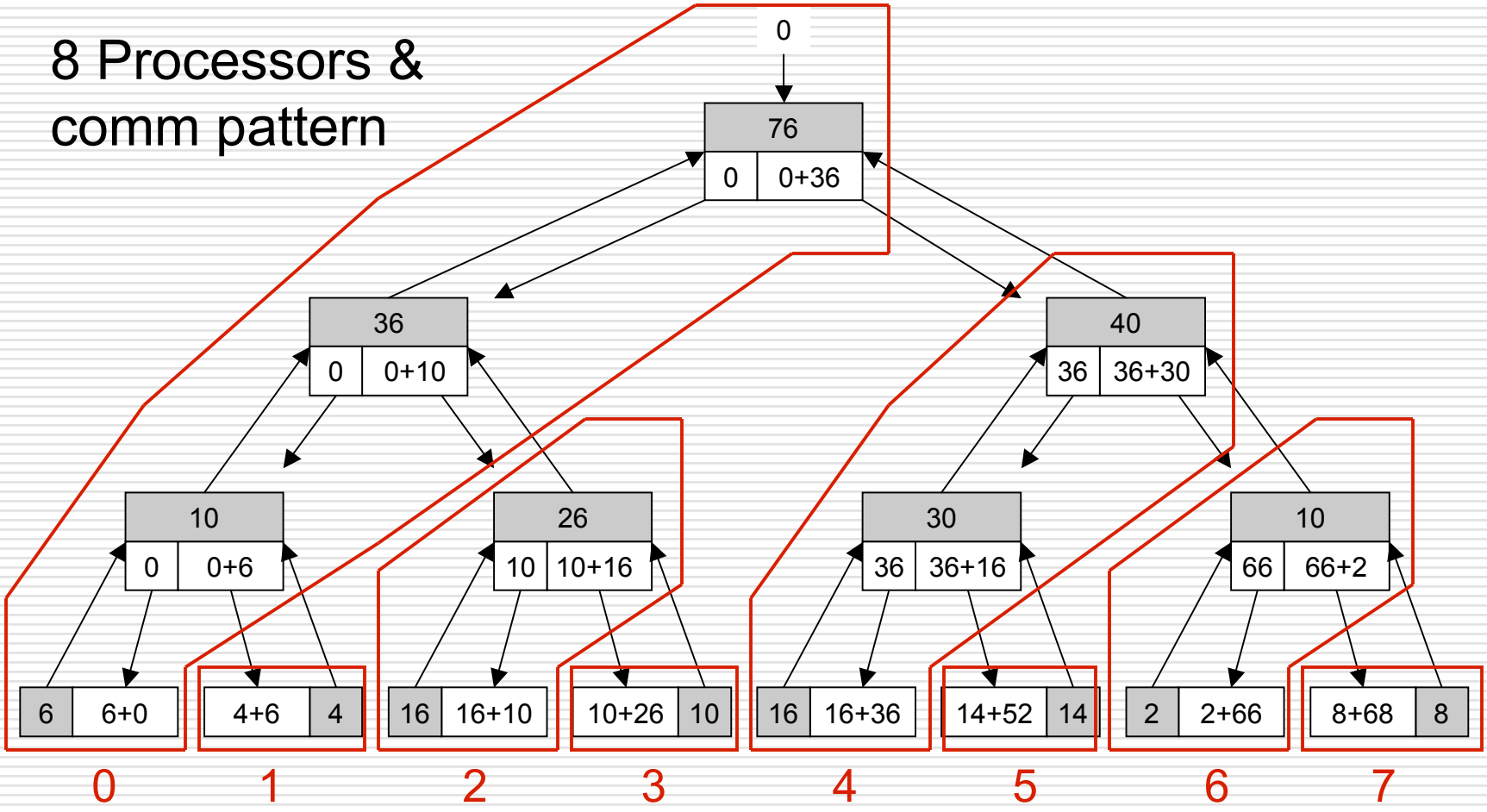
Recall Parallel Prefix Algorithm

8 Processors &
comm pattern



Recall Parallel Prefix Algorithm

8 Processors &
comm pattern



Use Parallel Prefix Abstraction …

- Thinking abstractly of "parallel prefix computations"
  - Abstract as combining adjacent pairs of sequences: `xxxxxxxx = llll + rrrr`
  - 3 parts to the abstraction --
    - initialize a descriptor (tally) for a singleton (i)
    - combine tallies of 2 adjacent sequences (c)
    - finish by extracting the answer from tally (f)

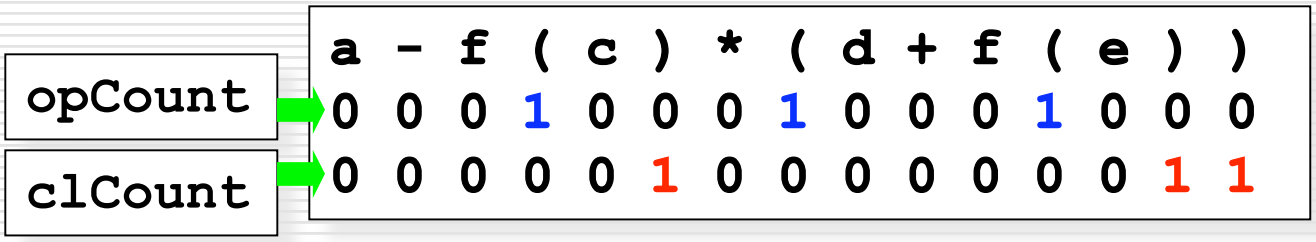
f(c(c(c(i(x) i(x)) c(i(x) i(x))) c(c(i(x) i(x)) c(i(x) i(x)))))

Applying || Prefix To Balanced ()s

- Consider
  - Information to be carried along: tally
  - How to join tallies of two independently computed subsequences
  - Consider what the output must be from tally
- The tally for “balanced parens” is two ints, excess open parens `opCount` and excess closed parents `clCount`

A || Prefix Solution

- Visualize a processor per point (not really*)
  - Each point is initialized to a *tally* data structure
  - Pairs are combined in some way
  - Process continues until there is one tally
  - Compute the final result
- Initialize on this problem: `a-f(c)*(d+f(e))`



*compilers produce “coarse-grain” code

Tri-Partite Parallel Prefix

Initialize a tally:

```
if (inval == '(' )
    int tally.opCount = 1;
else
    int tally.opCount = 0;
if (inval == ')' )
    int tally.clCount = 1;
else
    int tally.clCount = 0;
```

xxxxxxxx = llll + rrrr

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;
```

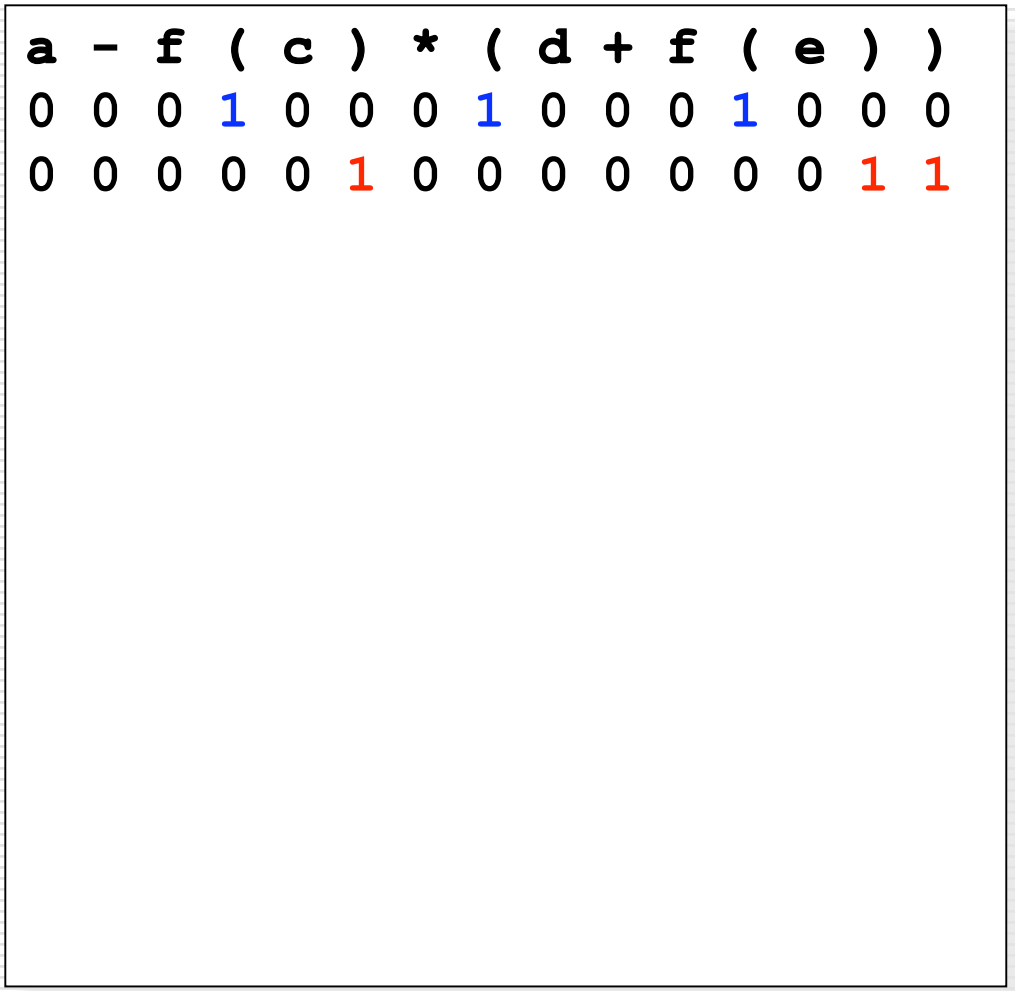
Finalize result from tally:

```
outval = (tally.opCount == 0) && (tally.clCount == 0);
```

Matching Parens

- Working out the details

  Matching



Matching Parens

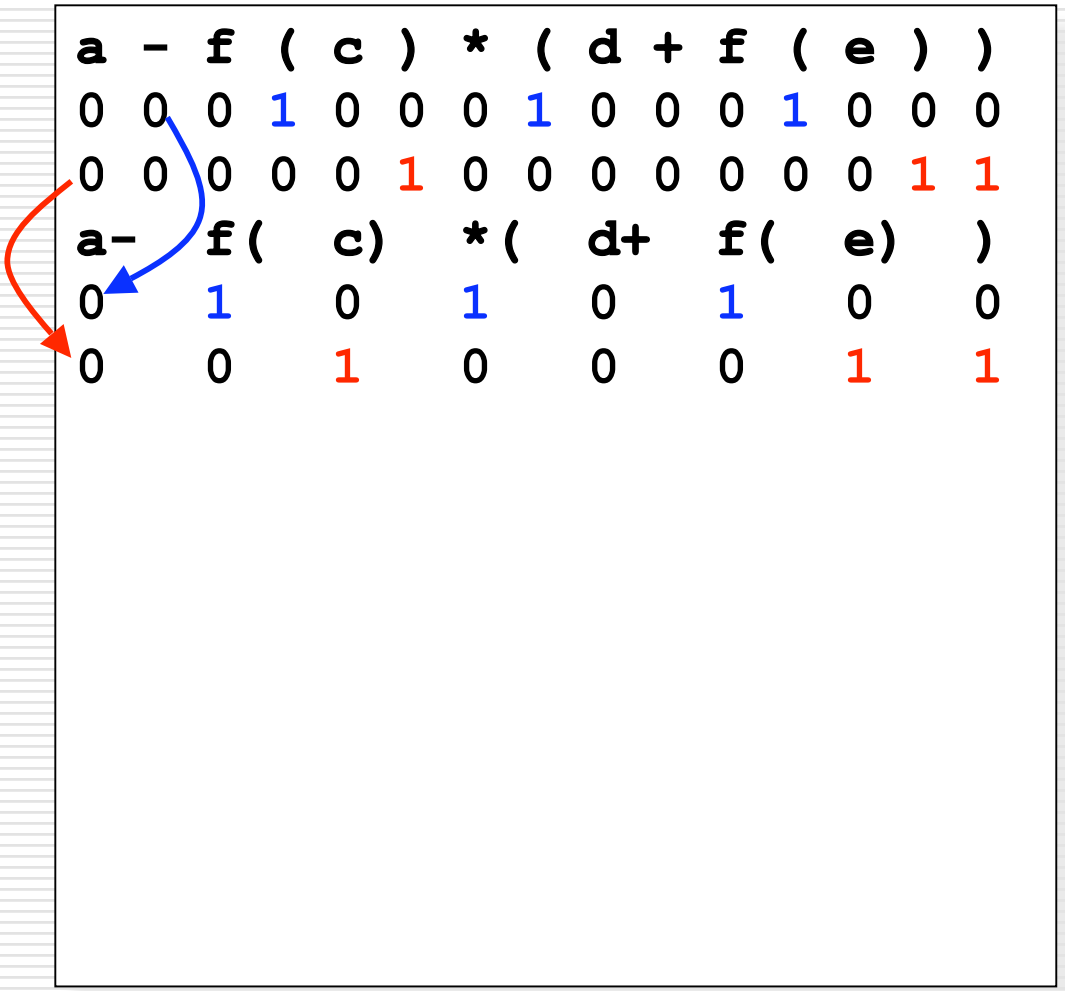
- Working out the details

  Matching

```
a - f ( c ) * ( d + f ( e ) )
0 0 0 1 0 0 0 1 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 1 1
a-  f(  c)  *(  d+  f(  e)  )
0   1   0   1   0   1   0   0
0   0   1   0   0   0   1   1
```

Matching Parens

- Working out the details
- Matching



Matching Parens

- Working out the details

  Matching

```
a - f ( c ) * ( d + f ( e ) )
0 0 0 1 0 0 0 1 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 1 1
a-  f(  c)  *(  d+  f(  e)  )
0   1   0   1   0   1   0   0
0   0   1   0   0   0   1   1
a-f(    c)*(    d+f(    e))
1       1       1       0
0       1       0       2
```

Matching Parens

- Working out the details

  Matching

```
a - f ( c ) * ( d + f ( e ) )
0 0 0 1 0 0 0 1 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 1 1
a-  f(  c)  *(  d+  f(  e)  )
0   1   0   1   0   1   0   0
0   0   1   0   0   0   1   1
a-f(    c)*(    d+f(    e))
1       1       1       0
0       1       0       2
a-f(c)*(        d+f(e))
1               0
0               1
a-f(c)*(d+f(e))
0
0
```

Tri-Partite Parallel Prefix

Initialize a tally:

```
if (inval == '(' )
    int tally.opCount = 1;
else
    int tally.opCount = 0;
if (inval == ')' ) {
    int tally.clCount = 1;
else
    int tally.clCount = 0;
```

```
                                    xxxxxxxx = llll + rrrr
```

Combine two tallies:

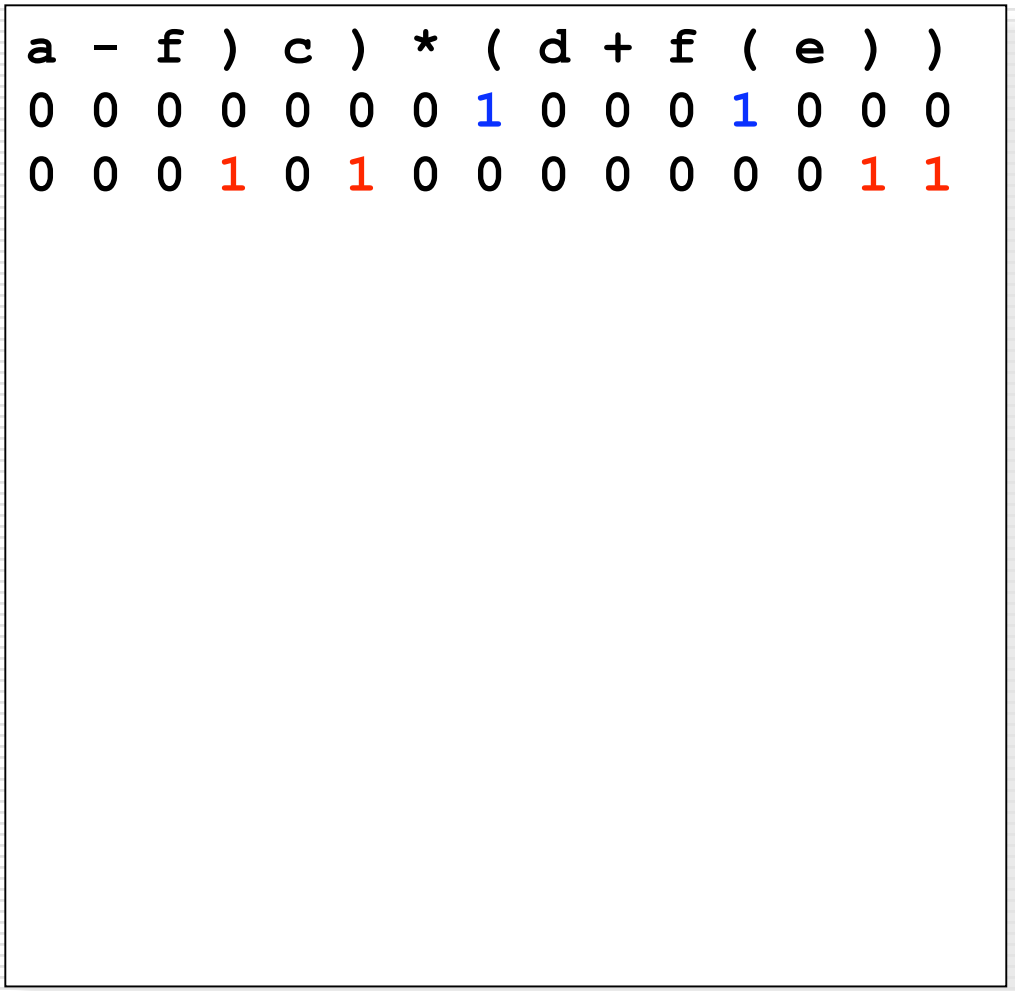
```
tally.clCount = ltally.clCount;
tally.opCount = rtally.opCount;
int temp = ltally.opCount - rtally.clCount;
if (temp < 0)
   tally.clCount += abs(temp);
else
   tally.opCount += temp;
```

Finalize result from tally:

```
outval = (tally.opCount == 0) && (tally.clCount == 0);
```

Matching Parens

- Working out the details
- Mismatching



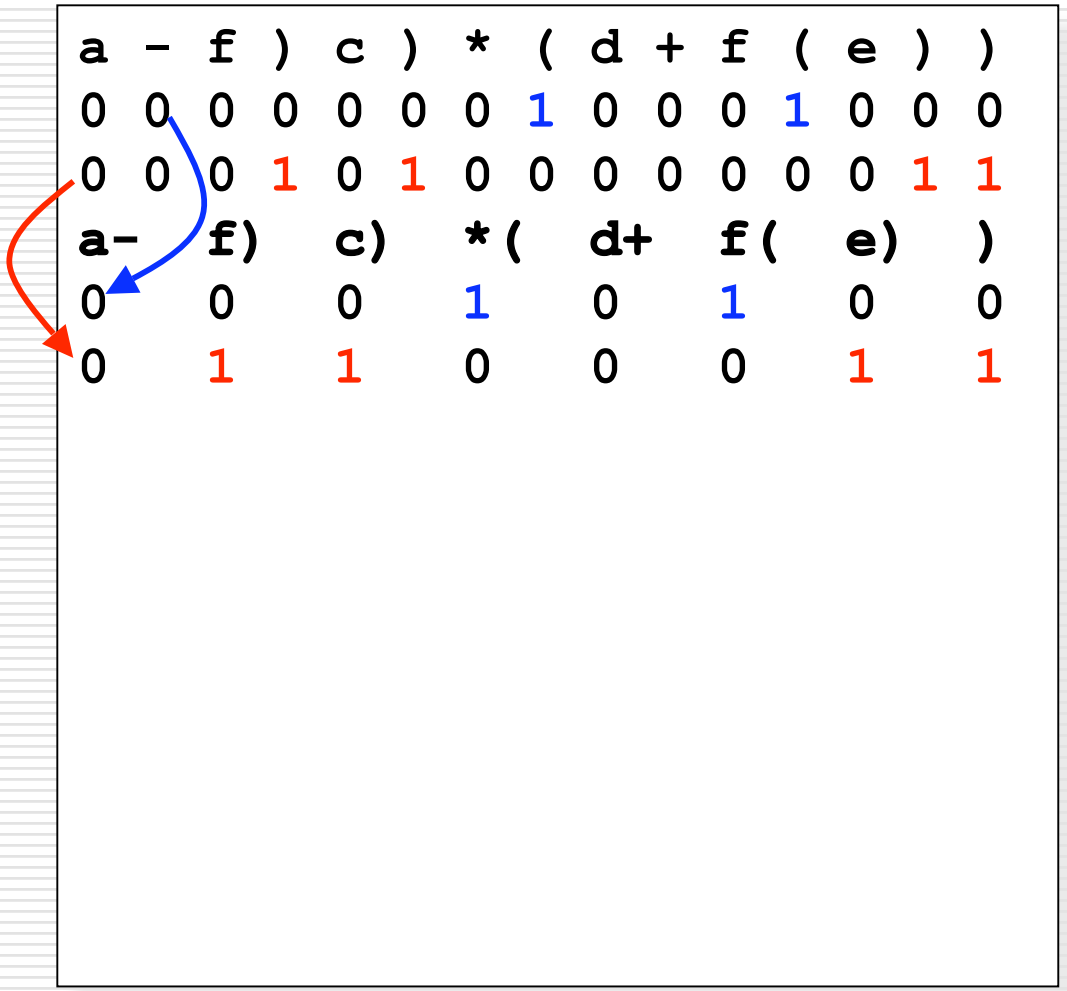
Matching Parens

- 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 1 1
a-  f)  c)  *(  d+  f(  e)  )
0   0   0   1   0   1   0   0
0   1   1   0   0   0   1   1
```

Matching Parens

- Working out the details
- Mismatching



Matching Parens

- 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 1 1
a-  f)  c)  *(  d+  f(  e)  )
0   0   0   1   0   1   0   0
0   1   1   0   0   0   1   1
a-f)    c)*(    d+f(    e))
0       1       1       0
1       1       0       2
```

Matching Parens

- 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 1 1
a-  f)  c)  *(  d+  f(  e)  )
0   0   0   1   0   1   0   0
0   1   1   0   0   0   1   1
a-f)    c)*(    d+f(    e))
0       1       1       0
1       1       0       2
a-f)c)*(        d+f(e))
1               0
2               1
a-f)c)*(d+f(e))
0
2
```

Summary on || Prefix

- 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 computations

Outline Of This Talk

- In The Beginning: Let The Compiler Do It
- Parallel Prefix Abstraction
- The Parallel Programming Problem
- A Parallel Machine Model
- Maybe An Easier Model Would Be Better
- Applying CTA Model In Programming -- Derive an Algorithm

Parallel Programming Problem

- Virtually all || programs should be platform independent
  - Recall: Machines have 1/2 life of years; programs have 1/2 life of decades
- Parallel Software Development Problem: How do we neutralize the machine differences given that
  - Some knowledge of execution behavior is needed to write programs that perform … the only interesting || programs?
  - Programs must port across platforms effortlessly, meaning, by at most recompilation
- Compare Sequential and Parallel Cases

RAM or von Neumann Model

- The Random Access Machine is our friend
  - Control, ALU, (Unlimited) Memory, [Input, Output]
  - Fetch/execute cycle runs 1 inst. pointed at by PC
  - Memory references are “unit time” independent of location
    - Gives RAM it’s name in preference to von Neumann
    - “Unit time” is not literally true, but caches fake it
  - Executes “3-address” instructions
  - Most other details (caches, I/O, etc.) ignored

It’s so intuitive, it seems like there’s no other way to compute!

Machine Model **Problem** (|| Case)

- In || programming, there is no single, logical machine to imagine to be executing the program; parallel variations are numerous
  - SIMD | MIMD
  - Shared memory | Distributed Memory
  - Shared memory | Shared address space
  - Narrow bisection bandwidth | Wide BiBW
  - Cluster | MPP
  - Etc.
- It's not likely the perfect || machine invented soon
- How we think about parallel execution is the *Parallel Programming Problem*

Options for Solving the PPP

- Adopt a very abstract language that can target to any platform & ignore details …

Options for Solving the PPP

- Adopt a very abstract language that can target to any platform & ignore details …
  - How does a programmer know how efficient or effective his/her code is? Interpreted code?
  - What are the "right" abstractions and statement forms for such a language?
    - Emphasize programmer convenience?
    - Emphasize compiler translation effectiveness?
  - No one wants to learn a new language, no matter how cool

Options for Solving the PPP

- Agree on a set of parallel primitives (spawn process, lock location, etc.) and create libraries that work w/ sequential code …

Options for Solving the PPP

- Agree on a set of parallel primitives (spawn process, lock location, etc.) and create libraries that work w/ sequential code …
  - Libraries are a mature technology
  - To work with multiple languages, limit base language assumptions … L.C.D. facilities
  - Libraries use a stylized interface (fcn call) limiting possible parallelism-specific abstractions
  - Achieving consistent semantics is difficult

Options for Solving the PPP

- Create an abstract machine model that accurately describes common capabilities and let the language facilities catch up …

Options for Solving the PPP

- Create an abstract machine model that accurately describes common capabilities and let the language facilities catch up …
  - Not a full solution until languages are available
  - The solution works in sequential world (RAM)
  - Requires discovering (and predicting) what the common capabilities are
  - Solution needs to be (continually) validated against actual experience

Summary of Options for PPP

- Adopt a very abstract language that can target to any platform & ignore details …
- Agree on a set of parallel primitives (spawn process, lock location, etc.) and create libraries that work w/ sequential code …
- Create an abstract machine model that accurately describes common capabilities and let the language facilities catch up …

Outline Of This Talk

- In The Beginning: Let The Compiler Do It
- Parallel Prefix Abstraction
- The Parallel Programming Problem
- A Parallel Machine Model
- Maybe An Easier Model Would Be Better
- Applying CTA Model In Programming -- Derive an Algorithm

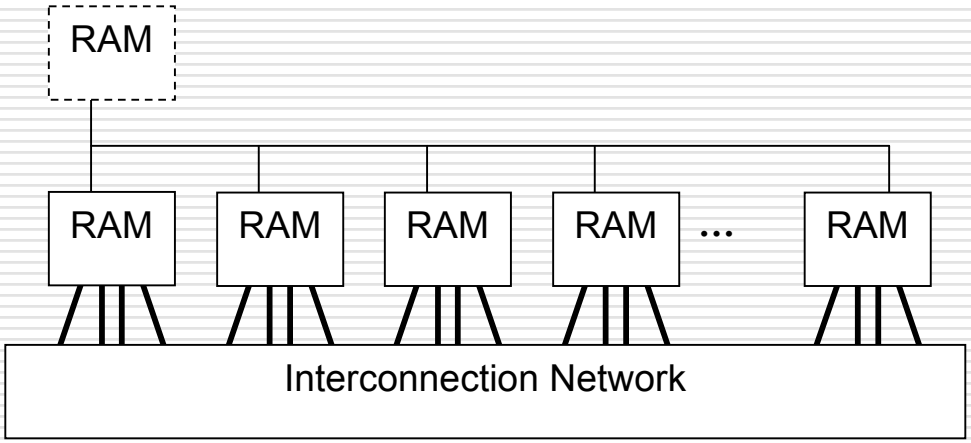
Generalization of RAM: PRAM

- Parallel Random Access Machine (PRAM)
  - Unlimited number of processors
  - Processors are standard RAM machines, executing **synchronously**
  - Memory reference is **unit time**
  - Outcome of collisions at memory specified
    - CRCW, CREW, EREW …
- The PRAM *might* approximate a multicore or SMP, but it doesn't abstract larger machines, so not suitable for algorithm analysis

PRAM is too abstract, though not entirely irrelevant

CTA Model

- Candidate Type Architecture (CTA): A || model with *P* standard processors, *d* degree, λ latency



- Node == processor + memory + NIC

Key Property: Local memory ref is 1, global memory is λ

What CTA Doesn't Describe

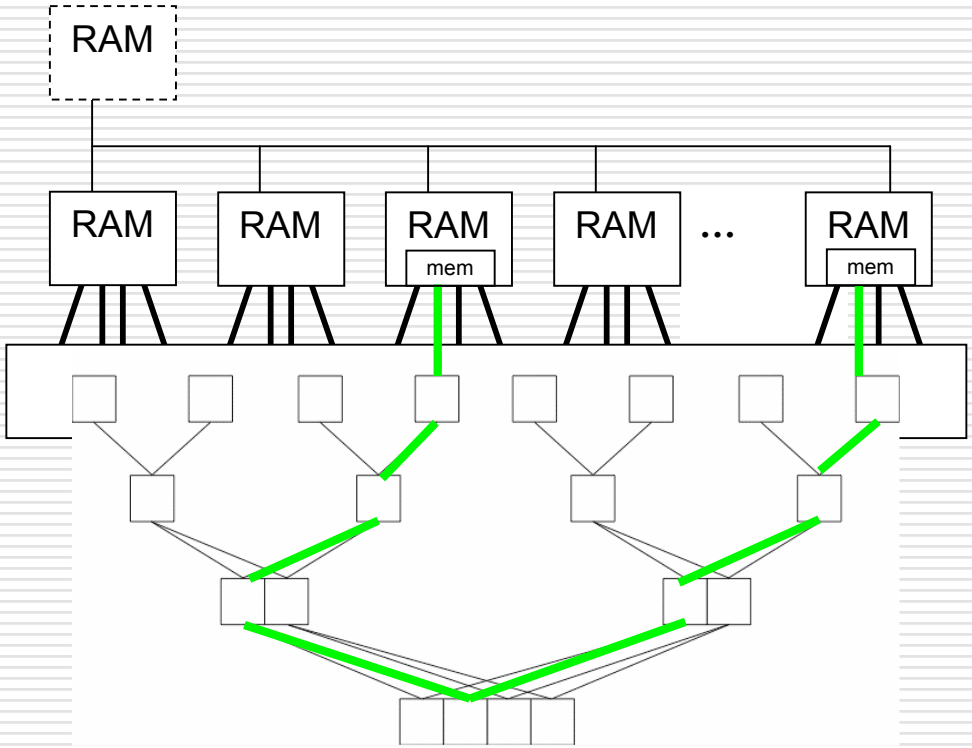
- CTA has no global memory … but memory could be globally *addressed*
- Mechanism for referencing memory not specified: shared, message passing, 1-side
- Interconnection network not specified, but there are some details
- λ is not specified beyond λ>>1 -- cannot be because every machine is different
- *d* is not specified except that it is "small", one digit
- Controller, combining network "optional"

Interprocessor Communication

- The communication mechanism (shared memory, 1-sided, message passing) is not specified
- Implications
  - Principles of operation are independent of communication choice
  - For shared memory machine, interpret “local memory” as L1/L2 cache, privately used L3/RAM
    - Avoid hazards of “memory consistency” by “staying local”
    - Could simplify hardware by relaxing cache coherence
  - Scalability potentially spans from multi-core to largest MPPs

More On the CTA

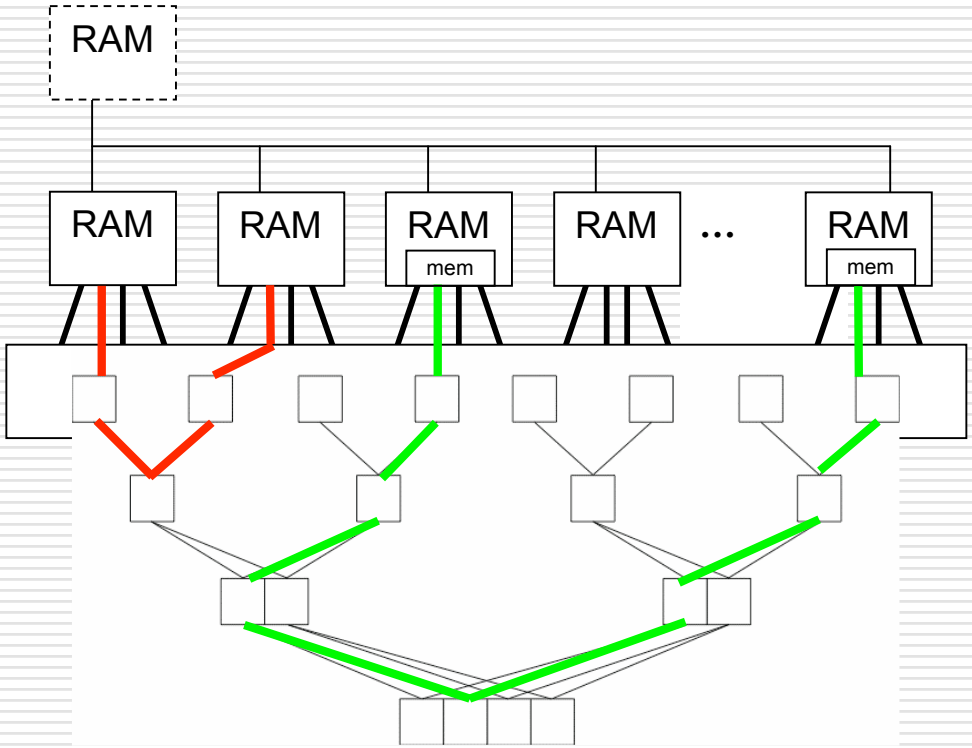
- Consider what the diagram means…



Fat Tree

More On the CTA

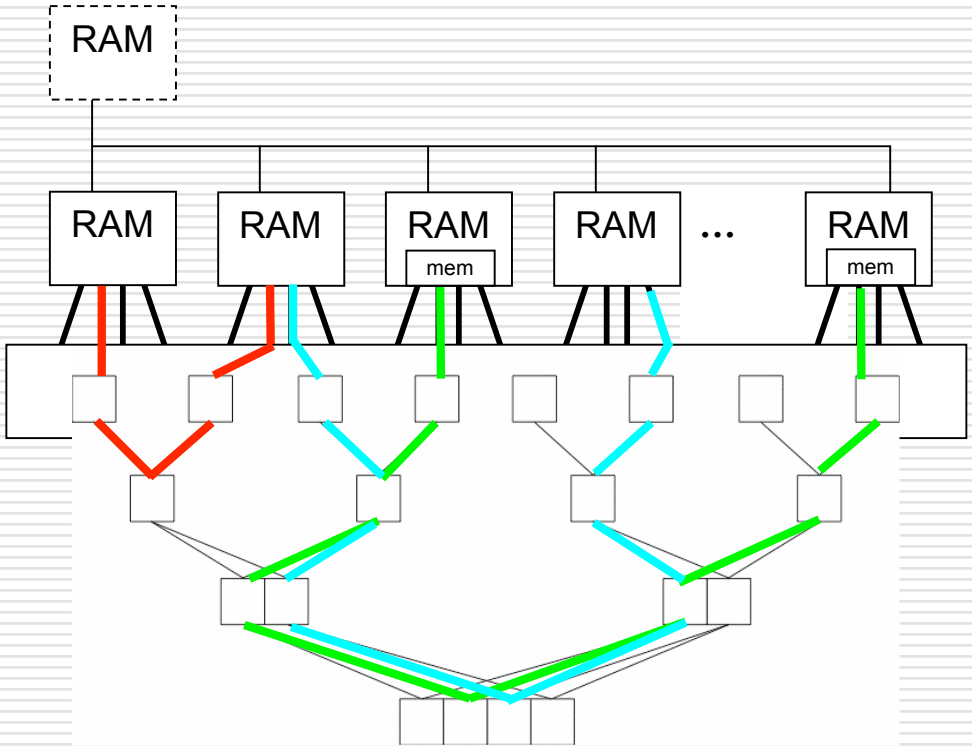
- Consider what the diagram means…



Fat Tree

More On the CTA

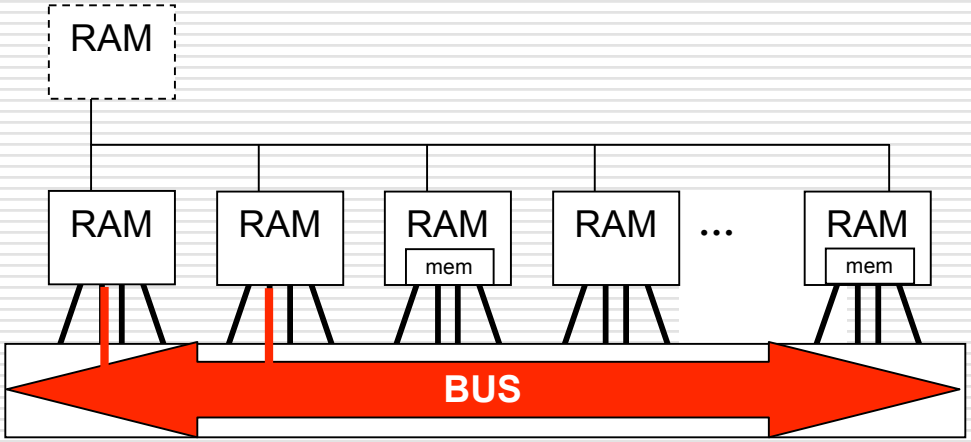
- Consider what the diagram means…



Fat Tree

More On the CTA

- Consider what the diagram doesn't mean…



- After ACKing that CTA doesn't model buses, accept that it could be a good approximation

Typical Values for λ

- Lambda can be estimated for any machine (given numbers include no contention or congestion)

CMP	AMD	100
SMP	Sun Fire E25K	400-660
Cluster	Itanium + Myrinet	4100-5100
Super	BlueGene/L	5000

Lg λ range => cannot be ignored

As with merchandizing: It's location, location, location!

Programming Implications

- How does CTA influence programming?
- Consider …
  - Expression evaluation: Same/Different?
  - Relationship among processors?
  - Data structures?
  - Organization of work?
  - …

Expression Evaluation

- The CTA processors are vN machines, so normal sequential execution is unaffected if the operands are local
  - This stresses locality
  - Insures caches work
  - Exploits the architectural progress in processor design over last 20 years … "basic blocks" that stay in L1/L2 cache execute at maximum speed
  - Therefore: Basic scalar computation preserved with customary performance

Relationship Among Processors

- The processors are autonomous
- Memory is not coherent -- it's a problem for the programmer
- Each process has it's own view on the computation -- must synchronize as needed
- Synchronization carries double cost, affecting both "sides" of the handshake
  - Reducing interactions speeds processing (locality again!)
  - Managing interactions (they're necessary) to lower their impact is essential to success

Implications for Data Structures

- A key implication is that a process cannot be oblivious to which portion of a data structure that it has local
- Implications of decomposed data structures
  - Working on partitioned data structures is more complicated than working one unitary ones
  - Load balancing and data structure partitioning are intimately connected
  - Races are easier to identify, handle
  - Automatic isolation is beneficial

Organization of Work

- The computation must be explicitly parallel
- Implications
  - Algorithms need to be rethought
  - Very fine grain parallelism is a waste of time
  - Minimize frequency of thread interaction
  - Expect to compute things redundantly
  - “First thing to try” is determine if existing (seq) techniques can solve a local portion of the problem, then combine results

Adopting the CTA implies significant change to algorithm development

Outline Of This Talk

- In The Beginning: Let The Compiler Do It
- Parallel Prefix Abstraction
- The Parallel Programming Problem
- A Parallel Machine Model
- Maybe An Easier Model Would Be Better
- Applying CTA Model In Programming -- Derive an Algorithm

Why CTA Is Right & PRAM is Wrong

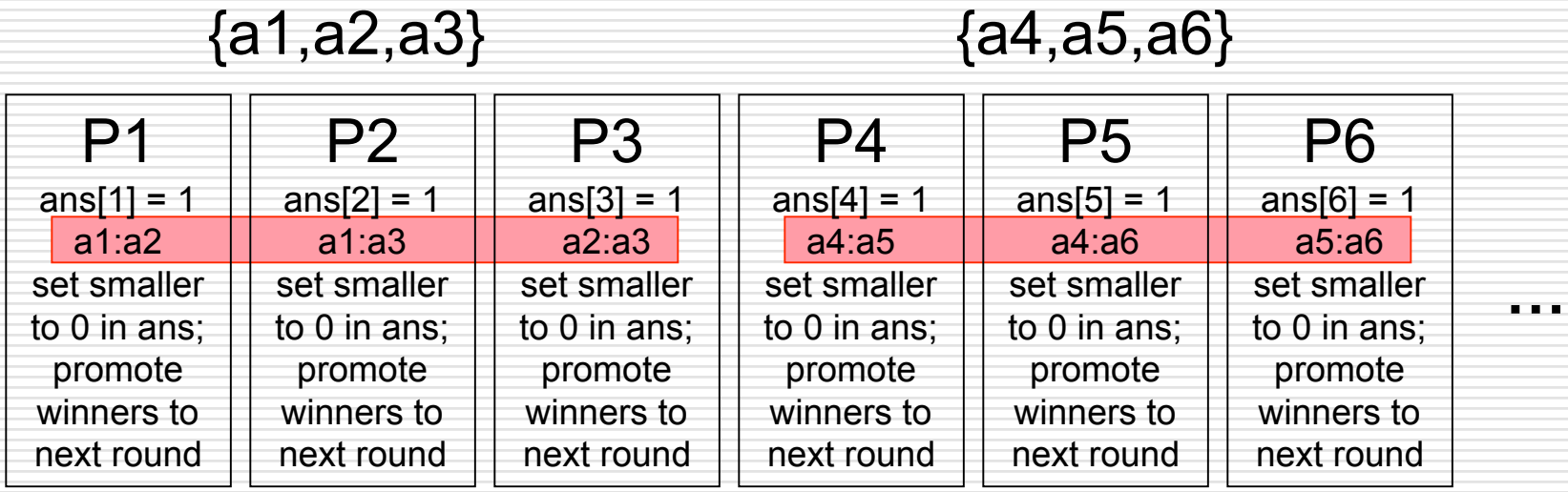
- The machine model drives our thinking on algorithms: It's the basis for choosing solutions to programming problems
  - Compare the CTA and PRAM in this role
  - Both models are simple
  - Both models take RAM machines as nodes
  - Both models have sequentially consistent memory, but PRAM is global, CTA is local
  - They differ in two notable ways
    - PRAM is synchronous
    - PRAM has a "unit cost" memory reference time
- Consider the consequences of this difference

Compare The Models

- Programming decision: What is the best solution to finding the maximum of *n* values?
  - For PRAM, the best Find_Max is Valiant's Alg
  - For CTA, the best Find_Max is Tournament Alg
  - Time bounds for $n = P$ processors
    - Valiant (CRCW) $O(\log \log n)$
    - Tournament $O(\log n)$
  - Choice of model determines program written
  - Consider both solutions

Valiant's Find_Max

- At the high level, Valiant's algorithm
  - Operates in rounds
  - In each round a processor is used to make just one comparison
  - Groups of processors can decide for a set of values which is largest; winner moves on to next round



Second Round

- Fewer elements implies larger groups, say 7

  {a1, a2, a3, a4, a5, a6, a7}

  a1:a2
  a1:a3 a2:a3
  a1:a4 a2:a4 a3:a4
  a1:a5 a2:a5 a3:a5 a4:a5
  a1:a6 a2:a6 a3:a6 a4:a6 a5:a6
  a1:a7 a2:a7 a3:a7 a4:a7 a5:a7 a6:a7

- Requires 6+5+4+3+2+1 = 21 processors to find largest in one step
- … until all that remains is a set with one element

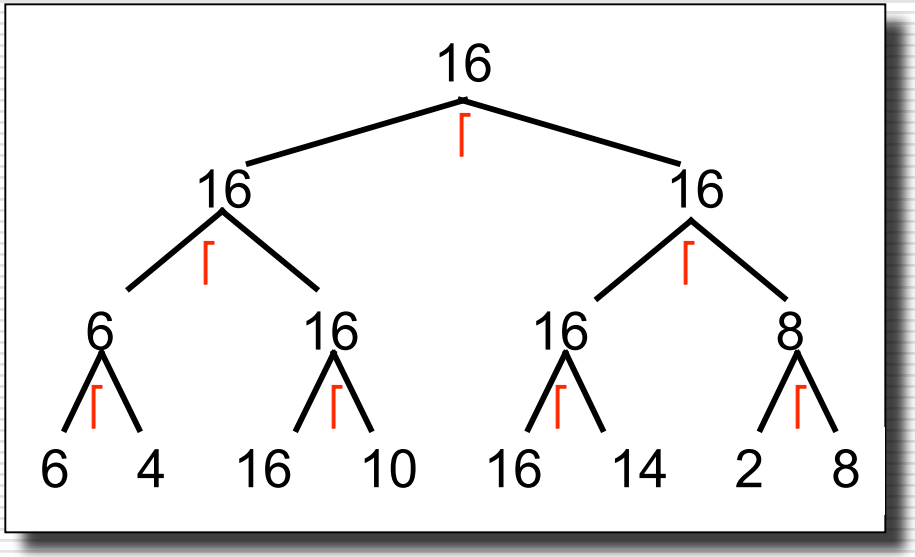
Analysis

- In the first round, there are *n* values, and *n*=*P* processors …
  - 3 = 2+1 processors can find the largest of 3 values
  - Round 1: Split *n* inputs into *n*/3 sets of 3 items each, requires *P* processors to find *n*/3 maxes
  - Now, fewer elements (*n*/3) and same number processors
  - 21 = 6+5+…+1 processors can find the largest of 7 values
  - Round 2: Split *n*/3 inputs into *n*/(3*7) sets of 7 items each, requires 21**n*/(3*7) = *P* processors to find the maxes …
  - Per round overhead is constant
  - Analysis shows a double exponential reduction in problem size, thus O(*log log n*) phases

Beautiful!

CTA Find_Max

- The CTA embeds a tree in the interconnection network, and implements the familiar tournament



PRAM Mispredicts Preferred Alg

- For task of finding maximum of *n* numbers
- Best algorithm
  - CRCW PRAM: Valiant's algorithm O(log log *n*) ★
  - CTA Model: Tournament algorithm O(log *n*)
- What's observed performance real implementation?
- PRAM communication on physical machine takes *at least* Ω (log *n*) time, but generally much more
- Observed performance
  - Valiant: O(log *n* log log *n*)
  - Tournament: O(log *n*) ★
  - The PRAM's guidance is wrong

Drawing Conclusions

- Predicting the wrong solution is not a happy outcome for a model
- The “problem” is the model ignores cost of communication, so Valiant’s solution can have unachievably fast memory reference
- Conclusions
  - Model must be accurate on important costs
  - Ignoring communication cost (λ) is dangerous in parallel programming

Outline Of This Talk

- In The Beginning: Let The Compiler Do It
- Parallel Prefix Abstraction
- The Parallel Programming Problem
- A Parallel Machine Model
- Maybe An Easier Model Would Be Better
- Applying CTA Model In Programming -- Derive an Algorithm

Consider Using CTA Model

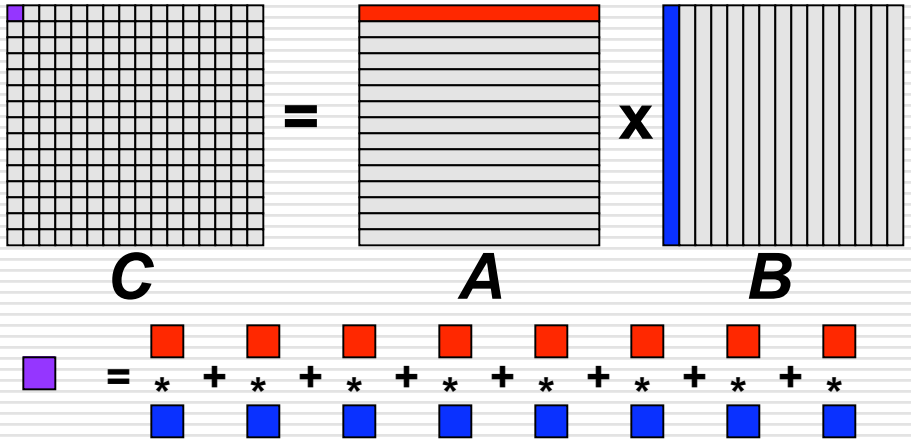
- Write a dense matrix multiplication program
- Standard sequential code is a place to begin

```
for (i = 0; i < m; i++) {
 for (j = 0; j < p; j++) {
    C[i][j]=0;
    for (k = 0; k < n; k++) {
        C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Can we use this for parallel solution?

C = AB

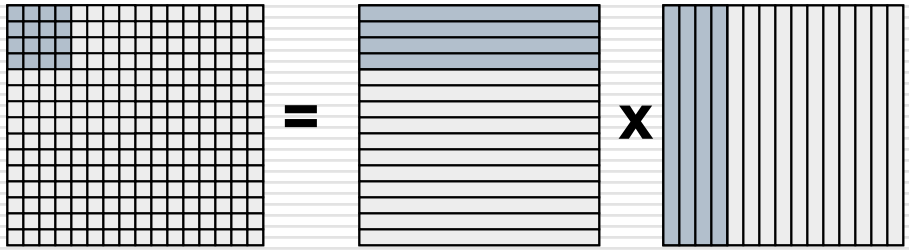
- Dense Matrix Multiplication



- Could spawn thread to compute "dot product"
  - No collisions on writing values [almost]
  - Cached values evicted for (at least one of the) operands
  - Most data is resident on other processor: communication
  - … better consider how data is stored -- blocks

Blocked View

- Tiling: a standard (sequential) reformulation

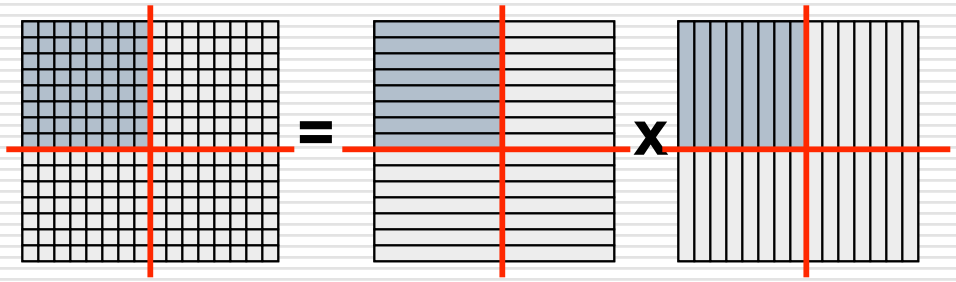


- Size of tile determined by the amount of each matrix that fits in the cache … tend to be "small"
  - Sensitive to policy for getting next tile: strip mining
  - Replicates arrays multiple times
  - *P* processor = *T* tiles? *P* < *T*, probably; *P* < $T^{1/2}$?
- Tiles are weak abstraction; they focus on reuse (that's good) but not data transfer (more critical)

Message: The cost is data *motion* != reuse

Shift Interest To Operands

- Tiles focus on which processor owns result, but ignores the cost of data transfer
- Consider which processors own operands

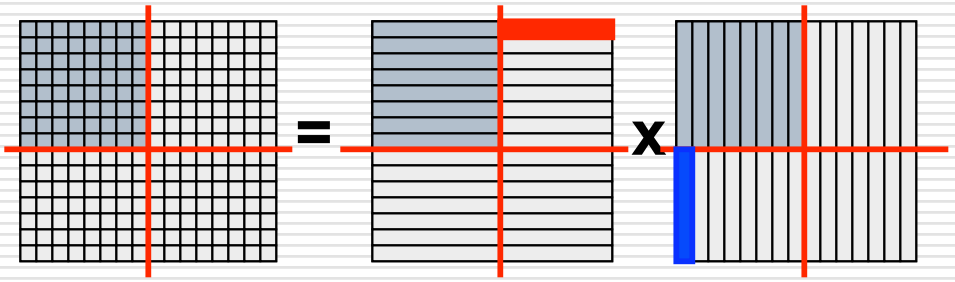


- As stated earlier, blocks avoid bias towards rows/columns … then consider data motion
  - Local portion computed without data motion
  - Completing computation requires blocks of arrays owned by other processors -- a large communication load

Consider Data Motion

- To complete dot-products already started requires considerable data transfer

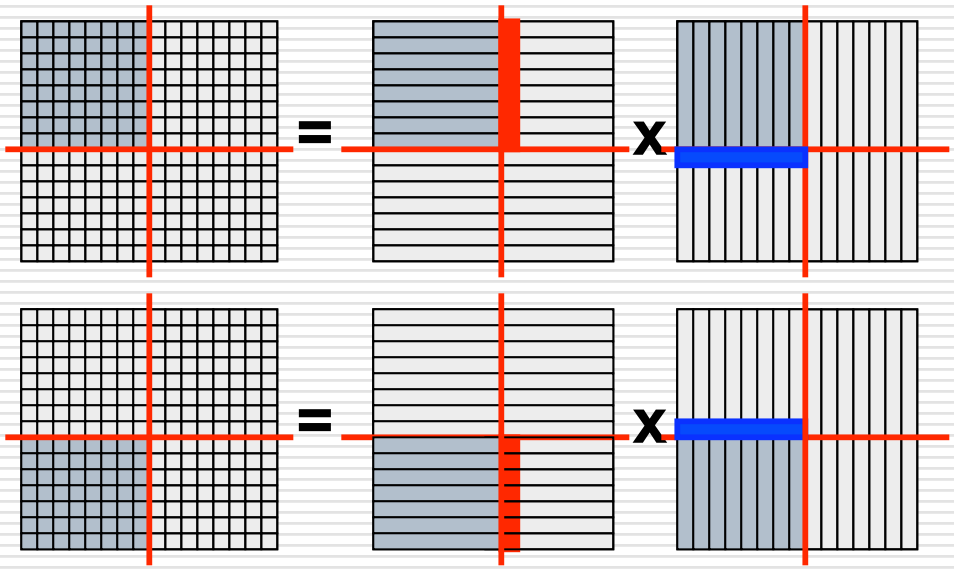
- Slamming network harms contention
- Consider ways of reducing data blast …
  - Send portions of subarrays



Obvious X-fer but not very useful

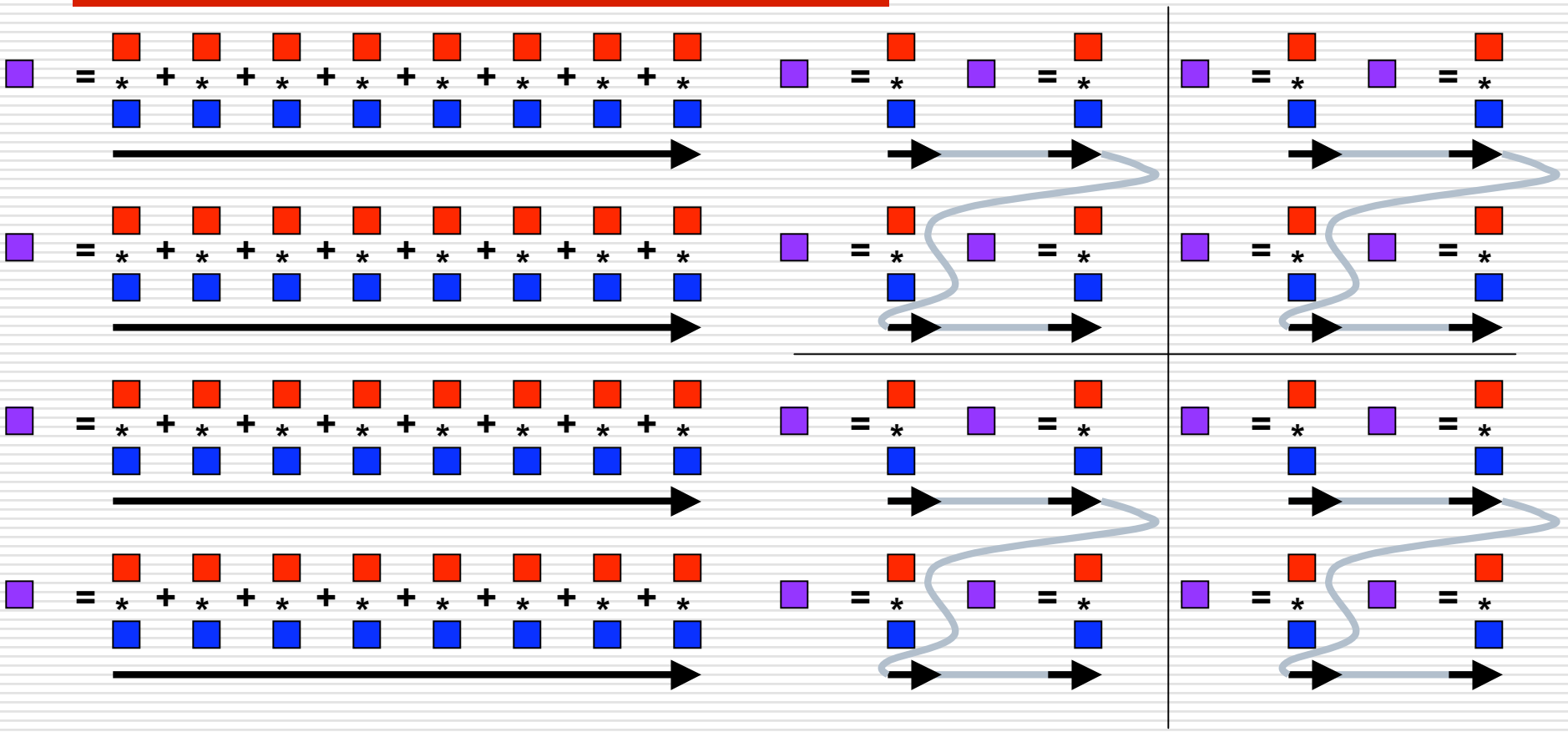
Adopt Compute-As-You-Go

- Algorithm has two phases
  - Local computation -- no communication
  - Iterative finish -- send-compute-receive
  - Data moved only once
- There's complexity here because of 2 phases



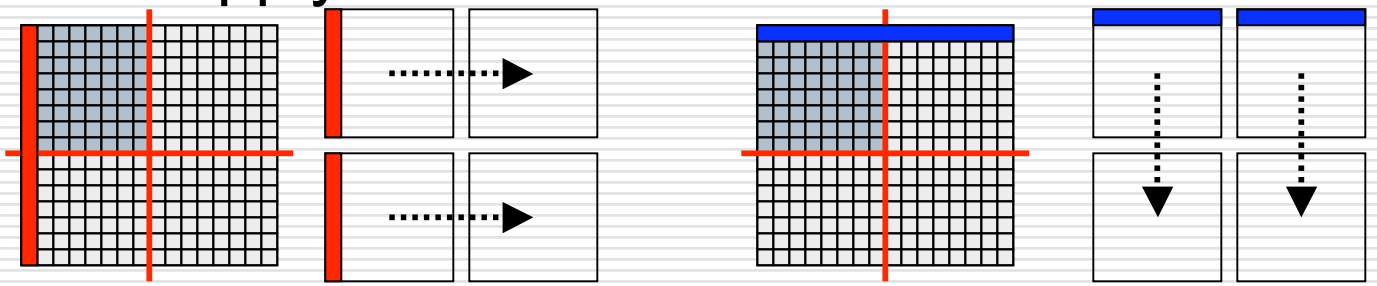
Change point of view away from dot-prod

Change In View Point



Reorder Computation

- Eliminate the 2-phase property by making second phase apply to all items

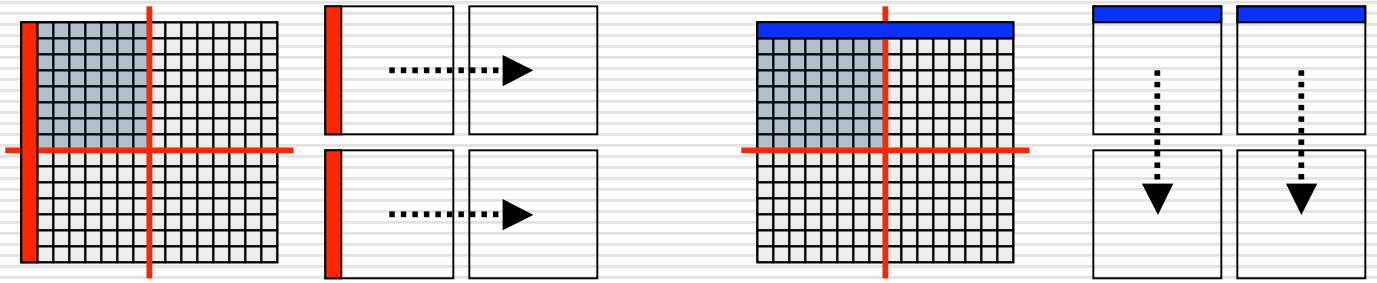


- Benefits
  - Simplify algorithm
  - Improve communication when broadcast is available
  - Reduce communication load by making all processors work on the same portion of row/column
- Can add tiling as needed …

New Algorithm

- The logic goes as follows:

```
initialize C array to 0
for (i=1; i <= n; i++) {
   bdcast col (portion) of A to horizontal peers
   bdcast row (portion) of B to vertical peers
   accumulate all local ‘next terms’ of dot prod
}
```



Voila! We Found SUMMA

- The iterative redesign of the standard “triply nested loops” algorithm resulted in the SUMMA algorithm (van de Geijn & Watts)
- Important milestones in thinking
  - Noticed lack of locality
  - Noticed tiles (std solution) focus on reuse and limit flexibility of applying processors to algorithm
  - Noticed block allocation unbiased
  - Noticed full locality on local portion
  - Noticed that continuing computation moved too much data, but move of part of row/column per processor OK
  - Noticed that “part move” could solve whole problem

Could a compiler have made these transformations?

Apply CTA to SUMMA

- How does CTA guide us understanding SUMMA algorithm is best?
  - Arrays will be partitioned and allocated to local memories
  - Each processor performs its computations locally
  - Send row and column parts via the interconnection net
  - Performance is
    - Full parallelism for local processing
    - λ for each row/col portion sent; broadcast
    - Overlapping communication/computation may remove λ charge

But isn't programming SUMMA a total headache?

Summary

- As with most research areas, in parallel computation some things worked, some things didn't

> *The parallel approach to computing … does require that some original thinking be done about numerical analysis and data management in order to secure efficient use. In an environment which has represented the absence of the need to think as the highest virtue, this is a decided disadvantage.*
>
> -- Dan Slotnick, 1967