CSE524 Parallel Computation

Lawrence Snyder
www.cs.washington.edu/CSEp524

24 April 2007

1

Announcements

- New homework assigned at end of class
- Discuss this week's assignment in 2 parts
  - Red/Blue, now
  - Essay, later this class
- Project description circulated at break

Project thinking begins this week

2

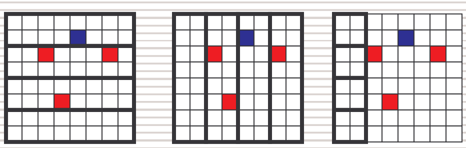
Red/Blue Problem

- Write Peril-L code to implement Red/Blue
- Goal is to get a scalable program
- Inputs:
  - *n* x *n* board, initialized and allocated
  - *t*, a small region used to sample
  - Termination condition: some *t* x *t* on left > 90%
- Rules of the game
  - Red moves 1 right on 1st half step if cell free
  - Blue moves 1 down on 2nd half step if cell free

3

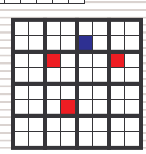
View Possible Solutions Globally

- Shared allows threads to process rows/cols



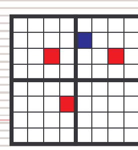
All references to global memory

- Allocate *t* x *t* threads



Must be multiplexed

- Scalable blocks



Maximize local work; multiplexing possible

4

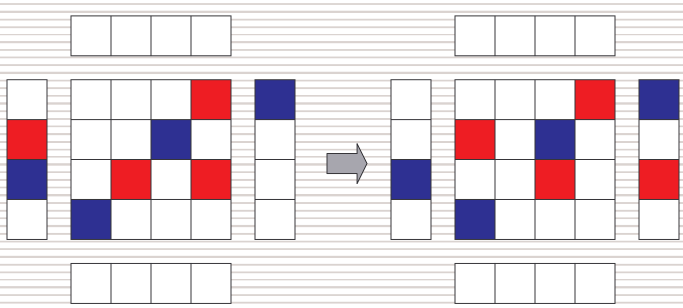
Red-Blue … One Solution

```
int RB_G[n][n];                        t|n, colors 0=w,1=r,2=b
/* Initialize RB with data */          Get external data
int size=sqrt(P); my=n/size;           Procs along row/col
int bin_GO[size][size][my];            MM for passing cols
int bou_GO[size][size][my];            MM for passing rows
int thresh=.9;                         90% threshold
forall thr,thc in (0:size-1,0..size-1) {   2D threads
  int Lrb[my][my]=is_local(RB_G);  Work on local assigned
  int lft[my],rgt[my],top[my],bot[my];Local neighbr storage
  int xl = RB_G_myLo_1; yl = RB_G_myHi_1;  Global index
  int xh = RB_G_myLo_2; yh = RB_G_myHi_2;  Global index
  int done=0,countR, countB;           Termination variables
```

5

Cartoon of Red Move

- Get adjacent context from neighbors



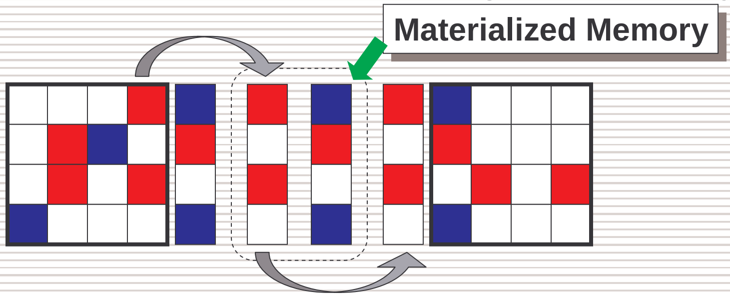
- Compute local state

Similarly for blue moves

6

Cartoon of State Transfer

- Save values in materialized global memory



7

Solution (Continued)

```
while (done==0) {                                      Go till threshold
  bou_GO[thr][thc][0:my-1]=Lrb[x1][yl:yh];  pass left col
  bin_GO[trh][thc][0:my-1]=Lrb[xh][yl:yh];  pass right col
  rgt[0:my-1]=bou_GO[thr][(thc+1)%my][0:my-1];last col+1
  lft[0:my-1]=bin_GO[thr][(thc-1)%my][0:my-1];first col-1
  moveRed(Lrb[][],rgt[],lft[]); barrier;
  bou_GO[thr][thc][0:my-1]=Lrb[x1:xh][yl];  pass top row
  bin_GO[trh][thc][0:my-1]=Lrb[x1:xh][yh];  pass bot row
  top[0:my-1]=bin_GO[(thr-1)%my][thc][0:my-1];  top row-1
  bot[0:my-1]=bou_GO[(thr+1)%my][thc][0:my-1];  last row+1
  moveBlue(Lrb[][],bot[],top[]);
```

8

Solution Continued

```
if (thc == 0)
  {for (k=0; k<my/t; k++) {
    countR = 0; countB = 0;
    for (i=0; i<t; i++) {
     for (j=0; j<t; j++) {
       if (Lrb[i][j]== 1)
           countR++;
       if (Lrb[i][j]== 2)
           countB++;
     }
    }
```

Am I on left edge?
Do all my txt blocks
Count up colors

9

Solution Continued

```
/* After a t x t block ...  */
     if (countR/(countR+countB)>=thresh ||
         countB/(countR+countB)>=thresh)
           done = 1;           1 color is over thresh
    }                          End of t x t tiles
   }                           End of if for term test
   done=+/done;                Find out how others did;synchs
  }                            End of while
}                              End of thread
```

10

Recap Solution

- Scaled by available processors; *t* x *t* is min
- Materialized memory implements a bulk exchange, benefiting from fast transfer
- Constructing context allows local compute
- Termination performed by reduction
- Why was the barrier needed?
- Is less data motion possible?

A good exercise: Analyze R/B complexity for different P

Schwartz's Algorithm

- 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* more values to leaves
  - Same complexity (O(log n)), but *nlog n* values!
- Generally *P* is not a variable, and $P << n$
- Use Schwartz as heuristic: Prefer to work at leaves rather than enlarge (make a deeper) tree, implying tree will have *log P* height

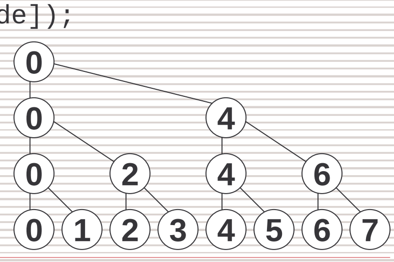
Expressing A Tree

- Consider writing tree-based sum in Peril-L
  - Processors will have several roles: leaf, intr node
  - Solve synchronization with materialized memory
- Stages:
  - Compute result at leaves
  - Threads have IDs in 0 - 2^p-1, pass if 0 != ID%2
  - Even threads add, pass value if 0 != ID%4
  - Etc.

Peril-L Code for Collective Ops

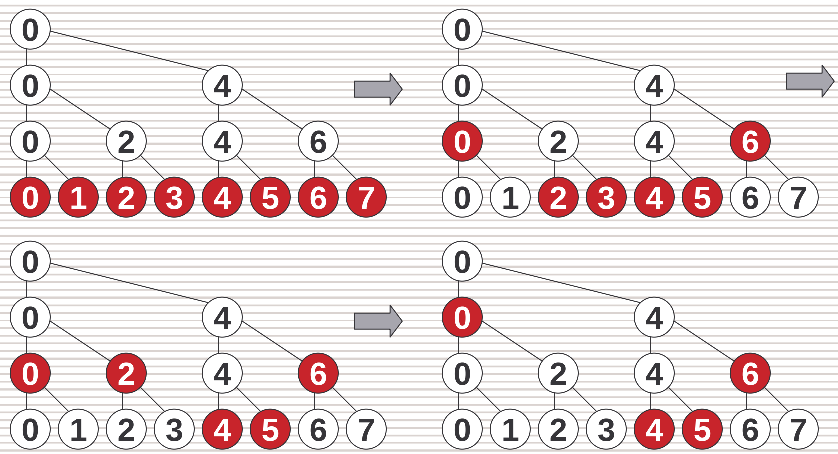
- Thread logic is easy … initialize `stride = 1`

```
nodeval_GO[tID] =  tally;          Send local val to tree node
while (stride < P) {               Begin logic for tree
 if (tID %(2*stride) == 0) {
     nodeval_GO[tID]= nodeval_GO[tID] +
     nodeval_GO[tID + stride]);
     stride = 2*stride;
 }
  else
     break;
}
```



Reverse to broadcast

Materialized Memory Self-Synchs



15

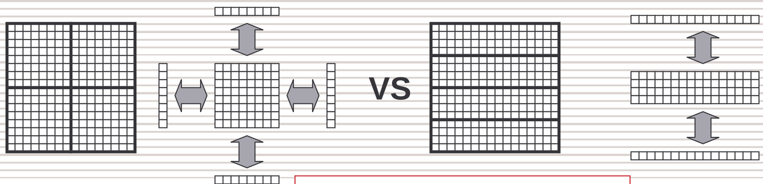
Asynchronous Trees

- MM gives asynchronous tree, preferred over a “leveled” tree using barriers
- Notice that once a thread’s role is complete it can continue computing
- For collective op **+** broadcast, a tree implemented with materialized memory allows half of the threads to continue
- **The Principle**: Relax synchronization rules wherever possible!

16

Block Allocations

- 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

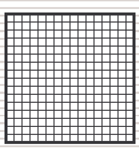


Now scale problem 4x

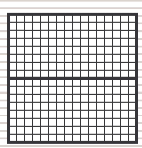
17

Different Regimens

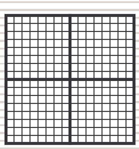
- Though block is generally a good allocation it's not absolute:



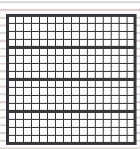
P=1, all comm wasted



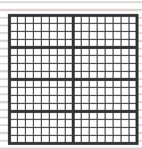
P=2, row-wise saves column comm



vs



P=4, rows and blocks are a wash

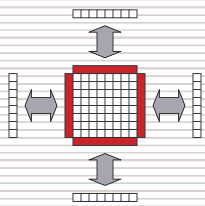


Where is the point of dim. return?

18

Shadow Buffers/Fluff

- To simplify local computation in cases where nearest neighbors values x-mitted, allocate in-place memory to store values:

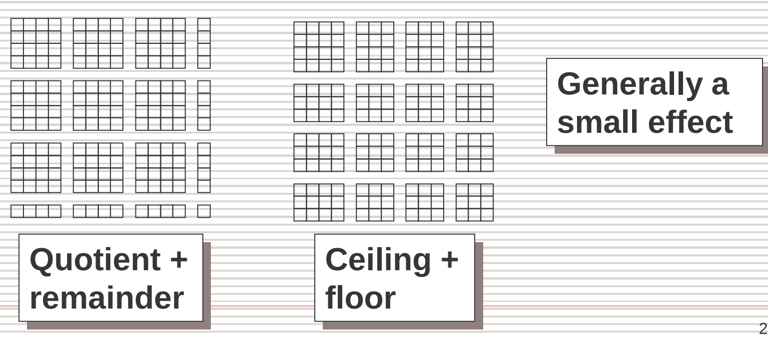


- Array can be referenced as if it’s all local

Edge storage (rgt,…) illustrated in R/B

Aspect Ratio

- Generally *P* and *n* do not allow for a perfectly balanced allocation …
- Several ways to assign arrays to processors



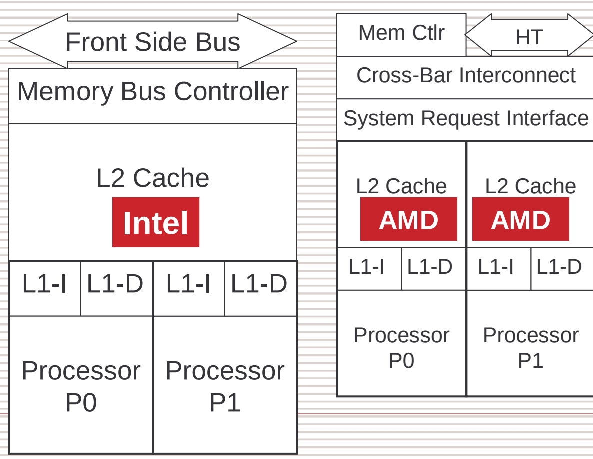
Assigning Processor 0 Work

- 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

Solving Array Problems on CMPs

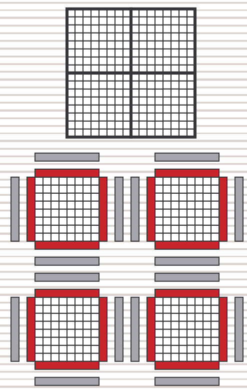
- How important are these topics for CMPs?



Speed comes from staying in L1 cache

Locality Always Matters

- 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 killing locality



False sharing an issue, too

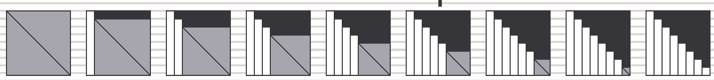
Reduce For Global Sum

- Reduce abstraction vs `sum_G += 1`
  - Reduce raises level of abstration
  - Reduce provides private var implementation
- Tree implementation of reduce
  - Irrelevant for tiny numbers of processors

Bottom Line: CTA machine model not strictly required for CMPs. Using it yields scalable programs that benefit CMPs in most ways without causing any harm.

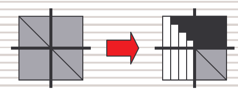
Load Balancing

- 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

Details in the appendix

25

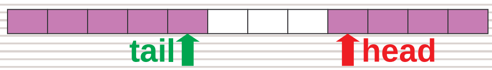
Work Queue, A Balancing Act(ivity)

- When work is generated dynamically, a work queue leads to reasonable balance
- Solution components:
  - Queue -- often a cyclically managed global array
  - Work or task descriptors -- succinct statements of the task to perform, often only index or bound
- Process:
  - As work is created, *append* to queue
  - On task completion, *remove* work from Q-head

26

Work Queue Considerations

- Synchronization
  - Threads modifying the queue do so exclusively
  - Independent modification of head (*remove*) & tail (*append*) is dangerous, tough to do right



- Work quantum (grain) must exceed the overhead of queue manipulation and comm
  - Large quantum may lead to poor “end game”

Considerations (Continued)

- Ordering, though strict at queue *remove*, may not be strict when viewed at completion
- Work assignment interacts with locality
  - Want strategies where descriptor embodies all data needed for task: chess move search
  - Avoid global memory reference by caching data: floor planning, place and route, etc.
  - When global reference inevitable, grain size should must cover comm costs too

Work quantum must be variable, not scalable

Break

Essay On ‘Quicksort’

- Discuss

  “Quicksort is an example of a sequential algorithm that is a good candidate to be incrementally transformed into a parallel algorithm”
- Where does the belief come from?
- What aspects are compatible/incompatible with the CTA?

Reconceptualizing a Computation

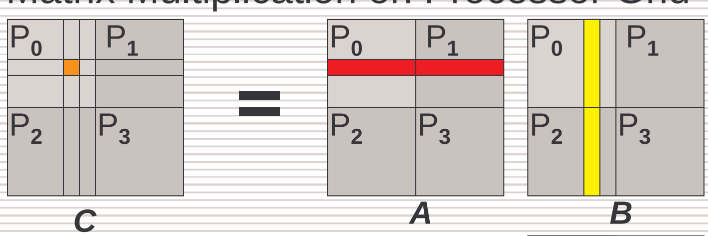
- Good parallel solutions result from rethinking a computation …
  - Sometimes that amounts to ordering scalar operations
  - Sometimes it requires starting from scratch
- The SUMMA matrix multiplication algorithm is the poster computation for rethinking!

This computation is part of homework assignment

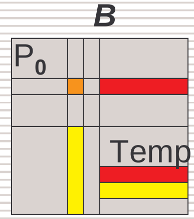
31

Return To A Lecture 1 Computation

- Matrix Multiplication on Processor Grid



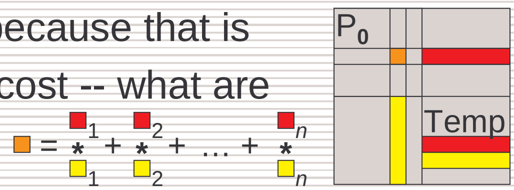
  - Matrices ***A*** and ***B*** producing n x n result ***C*** where $C_{rs} = \sum_{1 \le k \le n} A_{rk} * B_{ks}$



32

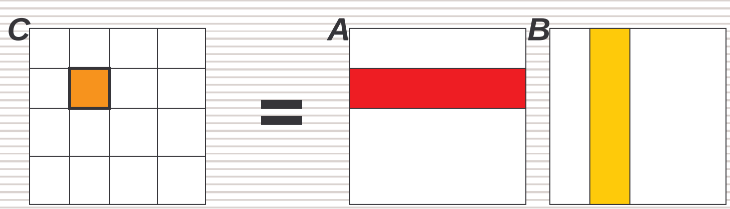
Applying Scalable Techniques

- 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

- If all rows/columns are present, it's local



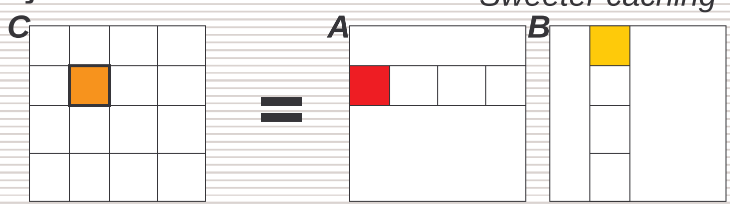
•Each element requires O(n) operations
•Modern pipelined processors benefit from large blocks of work
•But memory space and BW are issues

Process *t* x *t* Blocks

- Referring to local storage

```
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];
      }
  }
}
```

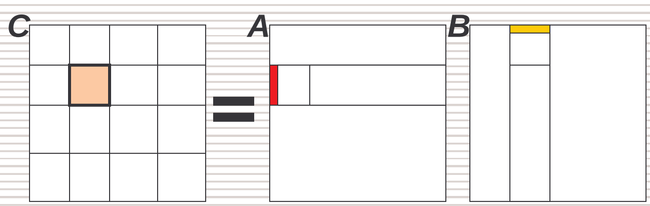
Sweeter caching



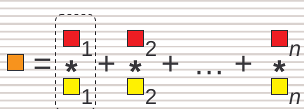
Only move a *t* x *t* block at a time

Change Of View Point

- Don't think of row-times-column



	b_{11}	b_{12}
a_{11}	$a_{11}b_{11}$	$a_{11}b_{12}$
a_{21}	$a_{21}b_{11}$	$a_{21}b_{12}$



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

SUMMA

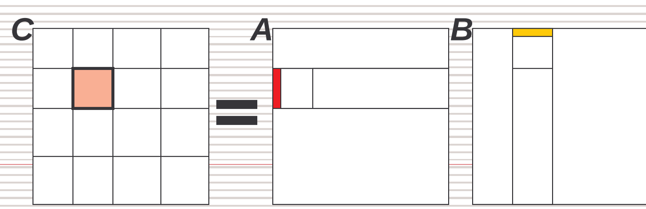
- Scalable Universal Matrix Multiplication Alg
  - Invented by van de Geijn & Watts of UT Austin
  - Claimed best machine independent MM
- Whereas MM is usually A row x B column, SUMMA is A column x B row because computation switches from sense
  - Normal: Compute all terms of dot product
  - SUMMA: Computer first term of all dot products

Strange. But fast!

37

SUMMA Assumptions

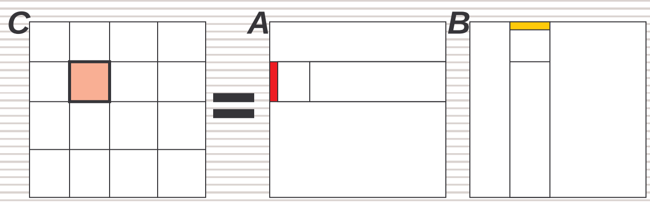
- Threads have two indices, handle t x t block
- Let $p = P^{1/2}$, then thread u,v
  - reads all columns of A for indices u*t:(u+1)*t-1,j
  - reads all rows of B for indices i,v*t:(v+1)*t-1
  - The arrays will be in global memory and referenced as needed



38

Solve SUMMA In Peril-L

- Arrays in Global Memory
- Consider how processors reference arrays
  - Where do operands come from?
  - Where are results saved?



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

Homework

- Read the remainder of Chapter 5
- Write a version of SUMMA in Peril-*L*
- Decide on Project Topic -- write a paragraph description of the computation that you expect to solve for the term project
- We will use a cluster making MPI and ZPL the most amenable languages we'll discuss