Mark Oskin …

- Wave Scalar Architecture
Announcements

- Thanks for your patience with UID/PW mess
- Thanks also for constructive ZPL comments
- Running ZPL … perhaps w/MPI on laptop?
- Recall that you need to turn in a brief (paragraph) description ON PAPER of your progress on the project this week
- Out of email contact from last lecture to project turn-in

Are the projects fun yet?

Review and Extend ZPL Concepts

- Several key ideas have not been covered
  - Applying WYSIWYG (better than last time)
  - Shattered Control Flow -- a simply thread idea
    - Illustrate with Red/Blue
  - Understanding/Reducing Dependences
  - Problem Space Promotion -- New algorithmic technique for parallelism, based on flood
  - Final comments on programming systems
Applying WYSIWYG in Alg Design

WYSIWYG, a key tool for parallel algorithm design … work through the logic of balancing costs

- There are dozens (hundreds?) of matrix product algorithms … which do you want?
  MM is a common building block, so someone else should have done this (vdG&W did!), but we use it as an example of process

- Two popular choices are
  - Cannon’s algorithm
  - SUMMA (vdG&W)

- Which is better as a ZPL program, i.e. better for scalable parallel machines, clusters, CTA model

Cannon’s Algorithm, A Classic

Compute: \( C = AB \) as follows …

\( C \) is initialized to 0.0

Arrays \( A, B \) are skewed

\( A, B \) move “across” \( C \) one step at a time

Elements arriving at a place are multiplied, added in
**Motion of Cannon’s, First Step**

\[
c_{43} = c_{43} + a_{41}b_{13}
\]

**Second steps ...**

\[
c_{43} = c_{43} + a_{42}b_{23}
\]
\[
c_{33} = c_{33} + a_{31}b_{13}
\]
\[
c_{42} = c_{42} + a_{41}b_{12}
\]

---

**Programming Cannon’s In ZPL**

\[
b_{13}
\]

\[
b_{12} b_{23}
\]

\[
b_{11} b_{22} b_{33}
\]

\[
b_{21} b_{32} b_{43}
\]

\[
b_{31} b_{42}
\]

\[
b_{41}
\]

Pack skewed arrays into dense arrays by rotation; process all n² vals at once
Four Steps of Skewing A

for i := 2 to m do
    [i..m, 1..n] A := A[^right];  Shift last m-i rows left
end;

a11 a12 a13 a14            a11 a12 a13 a14
a21 a22 a23 a24            a22 a23 a24 a21
a31 a32 a33 a34            a32 a33 a34 a31
a41 a42 a43 a44            a42 a43 a44 a41

Initial   i = 2 step
a11 a12 a13 a14      a11 a12 a13 a14
a22 a23 a24 a21      a22 a23 a24 a21
a33 a34 a31 a32      a33 a34 a31 a32
a43 a44 a41 a42      a44 a41 a42 a43

i = 3 step   i = 4 step

... And Skew B vertically

Cannon’s Declarations

For completeness, if A is m×n and B is n×p, the declarations are ...

region    Lop = [1..m, 1..n];
Rop = [1..n, 1..p];
Res = [1..m, 1..p];
direction right = [0, 1];
below = [-1, 0];
var           A : [Lop] double;
B : [Rop] double;
C : [Res] double;
Cannon's Algorithm

Skew $A$, Skew $B$, {Multiply, Accumulate, Rotate}^n

\[
\begin{align*}
&\text{for } i := 2 \text{ to } m \text{ do} \quad \text{Skew } A \\
&[i..m, 1..n] A := A@^\text{right}; \quad \text{end;} \\
&\text{for } i := 2 \text{ to } p \text{ do} \quad \text{Skew } B \\
&[1..n, i..p] B := B@^\text{below}; \quad \text{end;} \\
&[\text{Res}] \quad C := 0.0; \quad \text{Initialize } C \\
&\text{for } i := 1 \text{ to } n \text{ do} \quad \text{For common dim} \\
&[\text{Res}] \quad C := C + A*B; \quad \text{For product} \\
&[\text{Lop}] \quad A := A@^\text{right}; \quad \text{Rotate } A \\
&[\text{Rop}] \quad B := B@^\text{below}; \quad \text{Rotate } B \\
&\text{end;} 
\end{align*}
\]

SUMMA Algorithm To Compare To

\[
\begin{align*}
&\text{var} \quad \text{Col} : [1..m,*] \text{ double; } \quad \text{Col flood array} \\
&\text{Row} : [* ,1..p] \text{ double; } \quad \text{Row flood array} \\
&\text{A} : [1..m,1..n] \text{ double; } \\
&\text{B} : [1..n,1..p] \text{ double; } \\
&\text{C} : [1..m,1..p] \text{ double; } \\
&[1..m,1..p] \quad C := 0.0; \quad \text{Initialize } C \\
&\text{for } k := 1 \text{ to } n \text{ do} \quad \text{Flood kth col of } A \\
&[1..m,*] \quad \text{Col} := >>[ , k] \text{ A}; \quad \text{Flood kth row of } B \\
&[* ,1..p] \quad \text{Row} := >>[k, ] \text{ B}; \quad \text{Combine elements} \\
&[1..m,1..p] \quad C += \text{Col}*\text{Row}; \\
&\text{end;}
\end{align*}
\]
Compare Cannon’s & SUMMA MM

- Analyze the choices with WYSIWYG …
  - SUMMA has shortest code [so what?]
  - Cannon’s uses only local communication

- The two algorithms abstractly:

<table>
<thead>
<tr>
<th>Cannon’s</th>
<th>SUMMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declare</td>
<td>Declare</td>
</tr>
<tr>
<td>Skew A</td>
<td>Initialize</td>
</tr>
<tr>
<td>Skew B</td>
<td>loop til n</td>
</tr>
<tr>
<td>Initialize</td>
<td>Flood A</td>
</tr>
<tr>
<td>loop til n</td>
<td>Flood B</td>
</tr>
<tr>
<td>C+=A*B</td>
<td>C+=A*B</td>
</tr>
<tr>
<td>Rotate A,B</td>
<td></td>
</tr>
</tbody>
</table>

Compare Cannon’s & SUMMA MM

- Step one is to cancel out the equivalent parts of the two solutions … they’ll work the same
- For MM the comparison reduces to whether the initial skews and the iterated rotates are more/less expensive than iterated floods
Cannon’s Algorithm

<table>
<thead>
<tr>
<th>Skew A, Skew B, {Multiply, Accumulate, Rotate}</th>
</tr>
</thead>
<tbody>
<tr>
<td>for i := 2 to m do</td>
</tr>
<tr>
<td>[i..m, 1..n] A := A@^right;</td>
</tr>
<tr>
<td>end;</td>
</tr>
<tr>
<td>for i := 2 to p do</td>
</tr>
<tr>
<td>[1..n, i..p] B := B@^below;</td>
</tr>
<tr>
<td>end;</td>
</tr>
<tr>
<td>[Res] C := 0.0;</td>
</tr>
<tr>
<td>for i := 1 to n do</td>
</tr>
<tr>
<td>[Res] C := C + A*B;</td>
</tr>
<tr>
<td>[Lop] A := A@^right;</td>
</tr>
<tr>
<td>[Rop] B := B@^below;</td>
</tr>
<tr>
<td>end;</td>
</tr>
</tbody>
</table>

Comms have λ latency, but much data motion

SUMMA Algorithm Analysis

The flood is (likely) more expensive than λ time, but less that λ(log P) ... probably much less, and there are fewer of them

| [1..m, 1..p] C := 0.0;                        |
| Initialize C                                 |
| for k := 1 to n do                           |
| [1..m,*] Col := >>[ ,k] A;                   |
| Flood kth col of A                           |
| [*,,1..p] Row := >>[k, ] B;                  |
| Flood kth row of B                           |
| [1..m,1..p] C += Col*Row;                    |
| Combine elements                             |
| end;                                         |

SUMMA does not require as much comm or data motion as Cannon’s, nor does it “touch” the array as much
Bottom Line ...

- We assert that SUMMA is the better algorithm
  - Though it does “potentially more expensive” communication, it does less of it
  - It’s “nonredundant” flood arrays cache well
  - There is less data motion

- Analytically ...

<table>
<thead>
<tr>
<th>algorithm</th>
<th>number of communications</th>
<th>communication complexity</th>
<th>communication volume</th>
<th>flops</th>
<th>elements referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cannon</td>
<td>$4n$</td>
<td>1</td>
<td>$n$</td>
<td>$2n^2 - n^3$</td>
<td>$n \cdot (\frac{n^2}{p} + 3n^2)$</td>
</tr>
<tr>
<td>SUMMA</td>
<td>$2n$</td>
<td>$\log p$</td>
<td>$n$</td>
<td>$2n^3$</td>
<td>$n \cdot (n^2 + 2n)$</td>
</tr>
</tbody>
</table>

- Test the assertion experimentally...
Shattered Control Flow

- ZPL executes one statement at a time, to completion, implying predicates are scalars
- If a predicate is an array, split into threads
  
  ```
  if A < 0 then A := -A; end;  
  ```
  Compute absolute value
  
  n
  The statements still execute alone, but each index is treated separately
- Constraints: communication is prohibited to avoid synching

Red/Blue As Shattered Control

```language=plaintext
program redBlue;
region R = [1..n, 1..n];
var RWB : [R] ubyte = 1; Mv:[R] ubyte;
direction e = [0,1]; s = [1,0];
procedure redBlue();
/* Initialize RWB w/colors: white=0;red=1;blue=2 */
while (true) do
  Mv := (RWB = 1 & RWB@^e = 0);  
  if Mv then RWB := 0; end;  
  Mv@^e := Mv;  
  if Mv then RWB := 1; end;  
/* Figure moving reds */
if Mv then RWB := 0; end;  
/* Move, by killing red */
Mv@^e := Mv;  
/* finding new position */
if Mv then RWB := 1; end;  
/* and setting red */
```
Blue Half Step

\[ Mv := (RWB = 2 \& RWB^s = 0); \quad \text{Figure moving blues} \]
\[
\text{if } Mv \text{ then } RWB := 0; \quad \text{Move, by killing blue}
\]
\[
Mv^s := Mv; \quad \text{finding new position}
\]
\[
\text{if } Mv \text{ then } RWB := 2; \quad \text{and setting blue}
\]
end;
end;

Red/Blue Data Motion

- When is I/O performed? Consider def/use

procedure redBlue();
/* Initialize RWB: white=0; red=1; blue=2 */
while (true) do
\[
Mv := (RWB = 1 \& RWB^e = 0); \quad \text{Figure moving reds}
\]
\[
\text{if } Mv \text{ then } RWB := 0; \quad \text{Move, by killing red}
\]
\[
Mv^e := Mv; \quad \text{finding new position}
\]
\[
\text{if } Mv \text{ then } RWB := 1; \quad \text{and setting red}
\]
\[
Mv := RWB = 2 \& RWB^s = 0; \quad \text{Figure moving blues}
\]
\[
\text{if } Mv \text{ then } RWB := 0; \quad \text{Move, by killing blue}
\]
\[
Mv^s := Mv; \quad \text{finding new position}
\]
\[
\text{if } Mv \text{ then } RWB := 2; \quad \text{and setting blue}
\]
end;
end; Can we do better?
Do the Logic …

- Figure actual data motion ... reduce dependences!

```plaintext
var Rnew, Bnew, Mv : [R] ubyte; Bit arrays

[R]while (true) do

Mv := (RWB = 1 & RWB^e = 0); OK 4 red 2 move
Rnew := (RWB = 0 & RWB^w = 1); New location
Mv := Mv | (RWB = 2 & (RWB^s = 0 | Direct blue move
  (RWB^s = 1 & RWB^se=0)))); Vacated move
Bnew := (RWB^n = 2 & (RWB = 0 | New location
  (RWB = 1 & RWB^e = 0)))); by either means

[R with Mv] RWB := 0; Clear vacated
[R with Rnew] RWB := 1; Set red
[R with Bnew] RWB := 2; Set blue

end; Shattered is equally good
```

---

Problem Space Promotion (PSP)

- PSP is a new parallel programming idea deriving from the power of flood
- Recall SUMMA inner loop in C

```plaintext
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    C[i][j] += Acol[i]*Brow[j]; ZPL uses flood
  }
}
```

- This is an all pairs (2D compute) over 1D data
- Generally, PSP is d-dim compute on (d-1)-dim data

Ideal for "all pairs"
Concept, In 2D on 1D data

- Imagine 1D array A & an all pairs compute
- Thinking of A as a row, there are 5 steps:
  1. Transpose A to AT
  2. Flood A
  3. Flood AT
  4. Compute all pairs
  5. Partial reduce back into 1D row

```
A: AT:
Af: ATf:

[1..n,1..n] ... Af != ATf ... 0110
1011
1101
0110

[*,1..n] Ans := &<< [1..n,1..n] ...
```

“All items the same” should be: &<< (A = A@e)

A “Little” Sort Using PSP

- Assume items distinct; sort by counting inequalities
- region R = [1,1..n]; Row of indices
- var Keys: [R] integer; Keys to sort
- Perm : [R] integer; Permutation to sort ’em
- FlR : [*1..n] integer; Flood array for rows
- FlC : [1..n,*] integer; Flood array for cols

```
procedure sortDistinct();
  [R] begin
  [*1..n] FlR := >>[1,1..n] Keys; Flood
  [1..n,*] FlC := >>[1..n,1] Keys#1,Index1]; Transpose and flood
  Perm := 1 + +<<[1..n,1..n] (FlC < FlR); Figure perm
  Keys#1,Perm := Keys; Reorder keys
  end;
```
Example of Sorting with PSP

Keys data

| 1 | 7 | 9 | 4 | 3 | 5 | 6 | 0 |

Compare <:

| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 7 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 3 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 5 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 6 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

Perm:

| 2 | 7 | 8 | 4 | 3 | 5 | 6 | 1 |

Key Features of PSP

- PSP creates a logical 2D array
  
  \[
  \begin{align*}
  \text{[R]} \text{ begin} \\
  [*,1..n] \text{ FLR} := >>[1,1..n] \text{ Keys;} \quad & \text{Flood} \\
  [1..n,*] \text{ FLC} := >>[1..n,1] \text{ Keys#}[1,\text{Index1}]; \quad & \text{Trans \& flood} \\
  \text{Perm} := 1 + +<<[1..n,1..n] (\text{FLC} < \text{FLR}); \quad & \text{Find perm} \\
  \text{Keys#}[1,\text{Perm}] := \text{Keys}; \quad & \text{Reorder keys} \\
  \text{end;}
  \end{align*}
  
- The only 2D structure is < test \ldots only logical
- Multiple 2D computations likely fused, so no 2D array is ever created
Examples of PSP Computations

- “Small” Sorting
- Matrix product
  - 2D data/3D comp
- N-body computations
- Mode of set of values
- ...

Expect any “all pairs” problem to be PSP

ZPL Classic

- So far we’ve learned only ZPL ‘Classic’
- ZPL has many other features
  - Sparse regions/arrays, multigrid regions/arrays
  - “Mighty scan” to support pipelining
  - Quad regions (::) to write processor local code
  - Control over processor arrangements (grid) and distribution of regions to processors
- Many features not well understood … much more research is needed
Parallel Programming Facilities

- Taxonomy of popular systems
  - Threading
    - Pthreads
    - OpenMP
    - Java Threads
  - Local focus
    - MPI*, PVM
    - Co-array Fortran
    - GAS (Global Addr Space) Languages: UPC, Ti
  - Global focus
    - ZPL

*Accounts for 95+% of production parallel code

Co-Array Fortran

Developed within Cray (originally F--) by Numrich & Reed

- Motivated to use T3D/T3E’s shmem facilities
- Add’s a processor “co-dimension” to F95 arrays

REAL, DIMENSION (N) [*] :: X,Y  !Declare 2 size n vectors
X(:) = Y(:) [PE]  !If PE is same on all vectors, copy Y to X

- Has a few collective operations, synch. primitives
- CAF provides a clean way to manage (shmem) communication in a “local view” language …
  machine model is CTA

Cray supports CAF
Co-Array Fortran

real dimension(n,n)[p,*] :: a,b,c

do k=1,n
  do q=1,p
    c(i,j)[myP,myQ] = c(i,j)[myP,myQ] + a(i,k)[myP, q]*b(k,j)[q,myQ]
  enddo
enddo

GAS Languages

- The idea with GAS languages is to present a global address space (like Peril-L), but not to commit to memory consistency
  - CTA model, no WYSIWYG, however
  - Programmers must worry about consistency
  - Programmers write local code a la MPI
  - With CAF, Universal Parallel C (UPC), Titanium
- We will not cover UPC or Ti … check’em out

GAS may be future, but details are tough
Homework

- No Textbook Reading For Next Week

- Project
  - Bring paper statement of progress to class