Parallel dense linear algebra computations (1)

Prof. Richard Vuduc
Georgia Institute of Technology
CSE/CS 8803 PNA, Spring 2008
[L.07] Tuesday, January 29, 2008
Sources for today’s material

- Mike Heath at UIUC
- CS 267 (Yelick & Demmel, UCB)
- Robert Numrich at Minnesota Supercomputing Institute
Review: Cilk, MPI, UPC/CAF

- **Cilk**: Language extensions to C for shared-memory dynamic multithreading
  - “spawn” / “sync”, with API for locks
  - “Optimal” work-stealing scheduler
- **MPI**: de facto standard message-passing API
- **UPC / Co-Array Fortran**: Partitioned global address space languages
  - Shared memory SPMD
  - Language extensions for processor-locality data layout control
UPC collectives

- Usual suspects, **untyped**: broadcast, scatter, gather, reduce, prefix, …
- Interface has synchronization modes
  - Avoid over-synchronizing (barrier before/after is simplest, but may be unnecessary)
  - Data collected may be read/written by any thread
- Simple interface for collecting scalar values (i.e., **typed**)
- Berkeley UPC value-based collectives
- Reference: [http://upc.lbl.gov/docs/user/README-collectivev.txt](http://upc.lbl.gov/docs/user/README-collectivev.txt)
Recall: Shared arrays in UPC

```c
shared int x[THREADS];    /* 1 elt per thread */
shared int y[3][THREADS]; /* 3 elt per thread */
shared int z[3][3];       /* 2 or 3 per thread */
```

**Example:**

```c
threads = 4
```

- `x` = "lives" on thread 0
- `y` = 5
- `z` = 4

**Distribution rule:**

1. Linearize
2. Distribute cyclically
Recall: Shared arrays in UPC

Example: Vector addition using `upc_forall`

```c
shared int A[N], B[N], C[N]; /* distributed cyclically */
... {
    int i;
    upc_forall (i = 0; i < N; ++i; i) /* Note affinity */
        C[i] = A[i] + B[i];
... }
```

A

B

S
Blocked arrays in UPC

**Example**: Vector addition using `upc_forall`

```c
shared int [*] A[N], B[N], C[N]; /* distributed by blocks */
... {
    int i;
    upc_forall (i = 0; i < N; ++i; &C[i]) /* Note affinity */
        C[i] = A[i] + B[i];
... }
```

A

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

B

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

S

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
Data layout in UPC

- All non-arrays bound to thread 0
- Variety of layout specifiers exist
  - No specifier (default): **Cyclic**
  - [*]: **Blocked**
  - [0] or [:]: **Indefinite**, all on 1 thread
  - [b] or [b1][b2]…[bn] = [b1*b2*…*bn]: **Fixed** block size
- **Affinity** of element i = floor(i / block-size) % THREADS
- Dynamic allocation also possible (upc_alloc)
2-D array layouts in UPC

Example: n x m array

```
shared int [m] a1[n][m];
```

```
shared int [k][m] a2[n][m];
```
Co-Array Fortran (CAF)

- Extends Fortran 95 to support PGAS programming model
  - Program == collection of images (i.e., threads)
  - Array “co-dimension” type extension to specify data distribution

References:

- http://www.co-array.org
- http://www.hipersoft.rice.edu/caf/index.html
Co-array data type

- Declare real array, locally of length n, globally distributed

  Example: \( n = 3 \), \texttt{num\_images()} = 4

  \[
  \text{real :: A(n)[*]}
  \]

- Compare to UPC

  \[
  \text{shared float [*] A\_upc[n*THREADS];}
  \]

  \[
  \text{shared float [3] A\_upc[THREADS][3];}
  \]
Communication in CAF

- Example: Every image copies from an image, p

\[
\begin{align*}
\text{real} & :: A(n)[*] \\
\text{...} & \\
A( :) & = A( :) [p]
\end{align*}
\]

- Syntax “[p]” is a visual flag to user
More CAF examples

```
real :: s  ! Scalar
real :: z[*]  ! “co-scalar”
real, dimension(n)[*] :: X, Y  ! Co-arrays
integer :: p, list(m)  ! Image IDs

...  

X       = Y[p]  ! 1. get
Y[p]     = X  ! 2. put
Y[::]    = X  ! 3. broadcast
Y[list]  = X  ! 4. broadcast over subset
X(list)  = z[list]  ! 5. gather
s = minval(Y[::])  ! 6. min (reduce) all Y
X(:)[::] = s  ! 7. initialize whole co-array
```
Multiple co-dimensions

- Organizes images in logical 3-D grid
- Grid size: \( p \times q \times k \), where \( p \times q \times k = \text{num\_images}() \)
Network topology
Network topology

- Of great interest historically, particularly in mapping algorithms to networks
  - Key metric: Minimize hops
  - Modern networks hide hop cost, so topology less important
- Gap in hardware/software latency: On IBM SP, cf. 1.5 usec to 36 usec
- Topology affects **bisection bandwidth**, so still relevant
Bisection bandwidth

- Bandwidth across smallest cut that divides network in two equal halves
- Important for **all-to-all** communication patterns

\[
bisection \text{ bw } = \text{ link bw} \quad \text{and} \quad bisection \text{ bw } = \sqrt{n} \times \text{ link bw}
\]
Linear and ring networks

**Linear**
- Diameter $\sim n/3$
- Bisection = 1

**Ring/Torus**
- Diameter $\sim n/4$
- Bisection = 2
Multidimensional meshes and tori

2-D mesh
Diameter $\sim 2\sqrt{n}$
Bisection $= \sqrt{n}$

2-D torus
Diameter $\sim \sqrt{n}$
Bisection $= 2\sqrt{n}$
Hypercubes

- No. of nodes = $2^d$ for dimension $d$
  - Diameter = $d$
  - Bisection = $n/2$
Trees

- Diameter = log $n$
- Bisection bandwidth = 1
- **Fat trees**: Avoid bisection problem using fatter links at top
Butterfly networks

- Diameter = $\log n$
- Bisection = $n$
- Cost: Wiring
## Topologies in real machines

<table>
<thead>
<tr>
<th>Machine</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray XT3, XT4</td>
<td>3D torus</td>
</tr>
<tr>
<td>BG/L</td>
<td>3D torus</td>
</tr>
<tr>
<td>SGI Altix</td>
<td>Fat tree</td>
</tr>
<tr>
<td>Cray X1</td>
<td>4D hypercube*</td>
</tr>
<tr>
<td>Millennium (UCB, Myricom)</td>
<td>Arbitrary*</td>
</tr>
<tr>
<td>HP Alphaserver (Quadrics)</td>
<td>Fat tree</td>
</tr>
<tr>
<td>IBM SP</td>
<td>~ Fat tree</td>
</tr>
<tr>
<td>SGI Origin</td>
<td>Hypercube</td>
</tr>
<tr>
<td>Intel Paragon</td>
<td>2D mesh</td>
</tr>
<tr>
<td>BBN Butterfly</td>
<td>Butterfly</td>
</tr>
</tbody>
</table>
Administrivia
Administrative stuff

- **New room** (dumpier, but cozier?): College of Computing Building (CCB) 101
- **Accounts**: Apparently, you already have them
- Front-end login node: **ccil.cc.gatech.edu** (CoC Unix account)
  - We “own” **warp43—warp56**
  - **Sign-up** for mailing list: [https://mailman.cc.gatech.edu/mailman/listinfo/ihpc-lab](https://mailman.cc.gatech.edu/mailman/listinfo/ihpc-lab)
Homework 1: Parallel conjugate gradients

- Implement a parallel solver for $Ax = b$ (serial C version provided)
- Evaluate on three matrices: 27-pt stencil, and two application matrices
- “Simplified:” No preconditioning
- **Bonus**: Reorder, precondition
- Performance models to understand scalability of your implementation
  - Make measurements
  - Build predictive models
- Collaboration encouraged: Compare programming models or platforms
Parallel matrix-matrix multiplication
Summary: Parallelization process
Latency and bandwidth model

- Model time to send a message in terms of latency and bandwidth

\[ t(n) = \alpha + \frac{n}{\beta} \]

- Usually have cost(flop) \( \ll \frac{1}{\beta} \ll \alpha \)

- One long message cheaper than many short ones

- Can do hundreds or thousands of flops for each message

- Efficiency demands large computation-to-communication ratio
Matrix multiply computation

\[ c_{i,j} \leftarrow \sum_k a_{i,k} \cdot b_{k,j} \]
1-D block row-based algorithm

- Consider \( n \times n \) matrix multiply on \( p \) processors (\( p \) divides \( n \))
- Group computation by block row (block size \( b = \frac{n}{p} \))
  - At any time, processor owns same block row of \( A, C \)
  - Owns some block row of \( B \) (passed along)
  - Must eventually see all of \( B \)
- Assume communication in a ring network (no contention)
- First, suppose no overlap of computation and communication
Time, speedup, and efficiency

\[ T_p = \frac{2n^3}{p} + \alpha p + \frac{n^2}{\beta} \]

\[ \text{Speedup} = \frac{1}{\frac{1}{p} + \frac{\alpha p}{2n^3} + \frac{1}{2n\beta}} \]

\[ E_p \equiv \frac{C_1}{C_p} = \frac{1}{1 + \frac{\alpha p^2}{2n^3} + \frac{p}{2n\beta}} \]

⇒ Perfect speedup

⇒ Scales with n/p
Is this a “good” algorithm?

- Speedup?
- Efficiency?
  - Time as \( p \) increases?
  - Memory as \( p \) increases?
- In each iteration, what is the flop-to-memory ratio?
2-D block layout

- Observation: Block-based algorithm may have better flop-to-memory ratio
- Simplifying assumptions
  - $p = \text{integer}^2$ and 2-D torus network
  - Broadcast along rows and columns

\[
\begin{array}{ccc}
p(0,0) & p(0,1) & p(0,2) \\
p(1,0) & p(1,1) & p(1,2) \\
p(2,0) & p(2,1) & p(2,2)
\end{array}
= 
\begin{array}{ccc}
p(0,0) & p(0,1) & p(0,2) \\
p(1,0) & p(1,1) & p(1,2) \\
p(2,0) & p(2,1) & p(2,2)
\end{array}
\times
\begin{array}{ccc}
p(0,0) & p(0,1) & p(0,2) \\
p(1,0) & p(1,1) & p(1,2) \\
p(2,0) & p(2,1) & p(2,2)
\end{array}
\]
Cannon’s algorithm, initial step: “Skew” A & B
Cannon’s algorithm, iteration step: Local multiply + circular shift (1)

<table>
<thead>
<tr>
<th></th>
<th>0,0</th>
<th>1,1</th>
<th>2,2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,0</td>
<td>1,1</td>
<td>2,2</td>
</tr>
<tr>
<td>1,0</td>
<td>2,1</td>
<td>0,2</td>
<td></td>
</tr>
<tr>
<td>2,0</td>
<td>0,1</td>
<td>1,2</td>
<td></td>
</tr>
</tbody>
</table>

\[ j \]

\[ i \]

\[ A \]

\[ B \]

\[ C \]
Cannon’s algorithm, iteration step: Local multiply + circular shift (2)
Cannon’s algorithm

// Skew A & N
for i = 0 to s-1 // s = sqrt (p)
   left-circular-shift row i of A by i
for i = 0 to s-1
   up-circular-shift column i of B by i

// Multiply and shift
for k = 0 to s-1
   local-multiply
   left-circular-shift each row of A by 1
   up-circular-shift each column of B by 1
The costs of Cannon’s algorithm

// Skew A & N
for i = 0 to s-1 // s = sqrt (p)
    left-circular-shift row i of A by i // cost = s*(α + n²/p/β)
for i = 0 to s-1
    up-circular-shift column i of B by i // cost = s*(α + n²/p/β)

// Multiply and shift
for k = 0 to s-1
    local-multiply // cost = 2*(n/s)³ = 2*n³/p³/2
    left-circular-shift each row of A by 1 // cost = α + n²/p/β
    up-circular-shift each column of B by 1 // cost = α + n²/p/β
Time, speedup, and efficiency of Cannon’s algorithm

\[ T_p = \frac{2n^3}{p} + 4\alpha \sqrt{p} + \frac{4n^2}{\beta \sqrt{p}} \]

\[ S(n) \equiv \frac{T_1}{T_p} = \frac{1}{\frac{1}{p} + 2\alpha \frac{\sqrt{p}}{n^3} + \frac{2}{\beta} \frac{1}{\sqrt{p}}} \]

\[ E_p \equiv \frac{T_1}{p \cdot T_p} = \frac{1}{1 + 2\alpha \left(\frac{\sqrt{p}}{n}\right)^3 + \frac{2}{\beta} \frac{\sqrt{p}}{n}} \]
“In conclusion...”
Backup slides
2-D array layouts in UPC

```
assert (THREADES == r*c);

shared [b1][b2] int A[n][m][r][c][b1][b2];

&A[i][j][u][v][x][y]
== A + i*m*r*c*b1*b2 + j*r*c*b1*b2 + u*c*b1*b2 + v*b1*b2 + x*b2 + y

(i*m*r*c*b1*b2 + j*r*c*b1*b2 + u*c*b1*b2 + v*b1*b2 + x*b2 + y) % (r*c)
== (u*c*b1*b2 + v*b1*b2 + x*b2 + y) % (r*c)
```
Evolution of distributed memory machine networks

- Message queues replaced by direct memory access (DMA)
- **Wormhole** routing: Processor packs/copies, initiates transfer, then goes on
- Message passing libraries provide store-and-forward abstraction
  - May send/receive between any pair of nodes
  - Time proportional to distance since each processor along path participates