Distributed Memory Programming (2D)

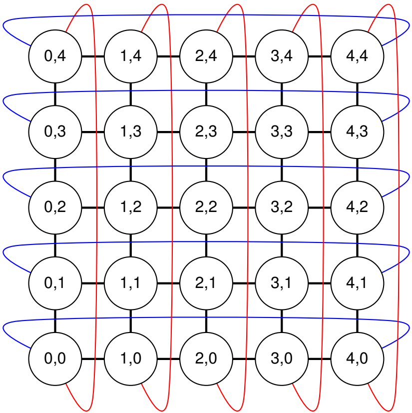
ICS632: Principles of High Performance Computing

Henri Casanova (henric@hawaii.edu)

Fall 2015

Outline

Grid/Torus of Processors



2-D Data Distribution

- We'll only consider 2-D *square* matrices
- There is thus a natural "block" data distribution:

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$
$A_{4,0}$	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$

- Each of the p processes holds a $N/p \times N/p$ matrix block of an $N \times N$ matrix

How do Matrices Get Distributed?

- You can do whatever you want, but what about libraries?
- Option #1 - Centralized: when calling a function (e.g., matrix multiplication) the input data is available on a single “master" machine (perhaps in a file) the input data must then be distributed among workers the output data must be undistributed and returned to the “master" machine (perhaps in a file)
  - More natural/easy for the user
  - The library makes data distribution decisions transparently
  - Prohibitively expensive if one does sequences of operations!!
- Option #2 - Distributed: When calling a function (e.g., matrix multiplication), one assumes that the input is already distributed and the output is left distributed
  - More work for the user
  - May lead to “redistribution" of data in between calls, which is harder for the user and may be costly
- Most current software adopt the distributed approach
- We always assume that the data is magically already distributed

Outline

Outer-Product Algorithm

- Let's see one classic matrix-multiplication algorithm
- Consider the k-i-j order:

k-i-j Matrix Multiplication

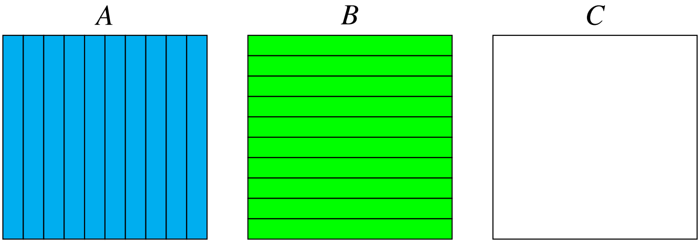
```
for (k=0; k < N; k++)
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      C[i][j] += A[i][k] * B[k][i];
```

- This is a sequence of N outer-products!
  - Multiply a column vector by a row vector

Matrix Multiplication: N Outer-Products

k-j-j Matrix Multiplication

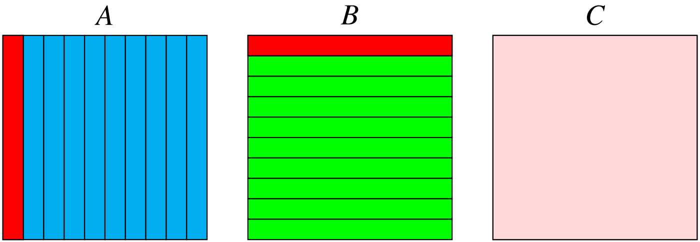
```
for (k=0; k < N; k++)
  // Multiply a column of A by a row of B
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      C[i][j] += A[i][k] * B[k][i];
```



Matrix Multiplication: N Outer-Products

k-j-j Matrix Multiplication

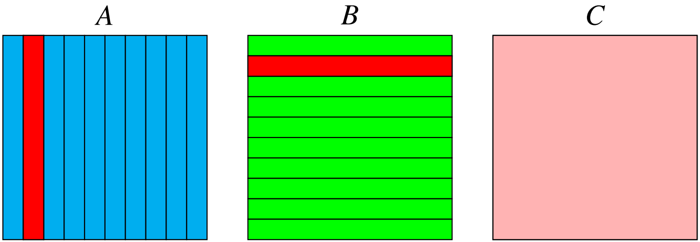
```
for (k=0; k < N; k++)
  // Multiply a column of A by a row of B
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      C[i][j] += A[i][k] * B[k][i];
```



Matrix Multiplication: N Outer-Products

k-j-j Matrix Multiplication

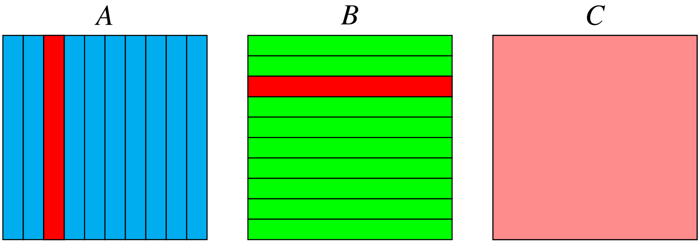
```
for (k=0; k < N; k++)
  // Multiply a column of A by a row of B
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      C[i][j] += A[i][k] * B[k][i];
```



Matrix Multiplication: N Outer-Products

k-j-j Matrix Multiplication

```
for (k=0; k < N; k++)
  // Multiply a column of A by a row of B
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      C[i][j] += A[i][k] * B[k][i];
```

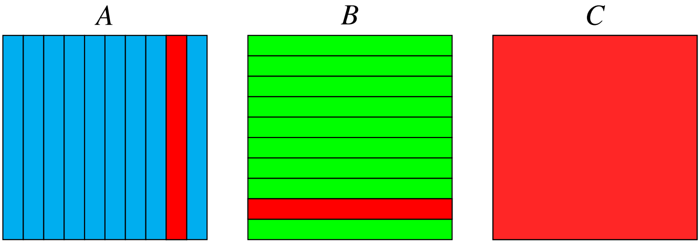


Matrix Multiplication: N Outer-Products

k-j-j Matrix Multiplication

for (k=0; k < N; k++)

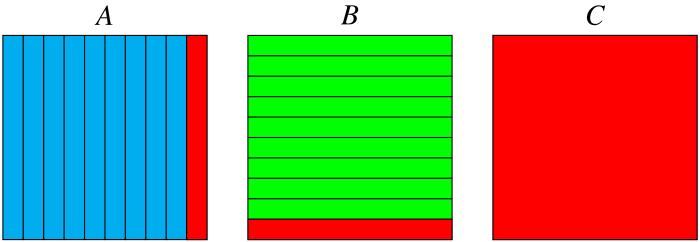
```
  // Multiply a column of A by a row of B
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      C[i][j] += A[i][k] * B[k][i];
```



Matrix Multiplication: N Outer-Products

k-j-j Matrix Multiplication

```
for (k=0; k < N; k++)
  // Multiply a column of A by a row of B
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      C[i][j] += A[i][k] * B[k][i];
```



So What??

- Why do we care about thinking of matrix multiplication this way???
  - Note that in principles there are $n^3!$ possible sequential algorithms
- Because it's possible to have a very elegant parallel algorithm
- Let's see a small example for a 4×4 grid of processors...

The Outer-Product Algorithm

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$
$A_{4,0}$	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$	$B_{0,4}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$	$B_{1,4}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$	$B_{2,4}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$	$B_{3,4}$
$B_{4,0}$	$B_{4,1}$	$B_{4,2}$	$B_{4,3}$	$B_{4,4}$

$C_{0,0}$	$C_{0,1}$	$C_{0,2}$	$C_{0,3}$	$C_{0,4}$
$C_{1,0}$	$C_{1,1}$	$C_{1,2}$	$C_{1,3}$	$C_{1,4}$
$C_{2,0}$	$C_{2,1}$	$C_{2,2}$	$C_{2,3}$	$C_{2,4}$
$C_{3,0}$	$C_{3,1}$	$C_{3,2}$	$C_{3,3}$	$C_{3,4}$
$C_{4,0}$	$C_{4,1}$	$C_{4,2}$	$C_{4,3}$	$C_{4,4}$

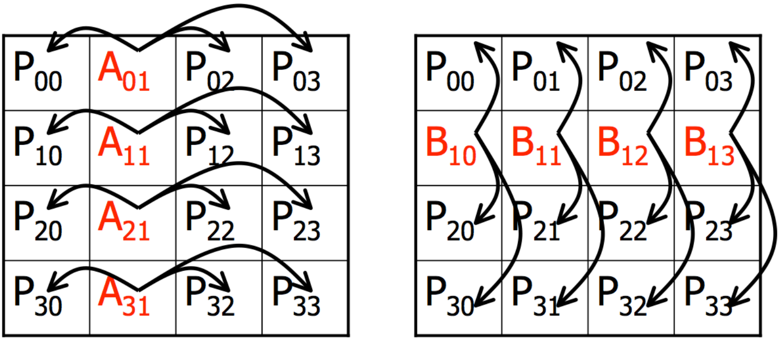
```
for (k=0; k < N; k++)
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      // Block operations
      C_{i,j} += A_{i,k} * B_{k,j});
```

- At step k, processor (i, j) needs $A_{i,k}$ and $B_{k,j}$
- If $j = k$, processor (i, j) already has $A_{i,k}$, otherwise it must receive it from processor (i, k)
- If $i = k$, processor (i, j) already has $B_{k,j}$, otherwise it must receive it from processor (k, j)

Communication Pattern

- From the previous slide:
  - At step k, processor (i,j) needs $A_{i,k}$ and $B_{k,j}$
  - If $j = k$, processor (i,j) already has $A_{i,k}$, otherwise it must receive it from processor (i,k)
  - If $i = k$, processor (i,j) already has $B_{k,j}$, otherwise it must receive it from processor (k,j)
- Therefore, at step $k = 0, \ldots, p-1$:
  - $\forall i$, processor (i,k) broadcasts its block of A to all processors in row i
  - $\forall j$, processor (k,j) broadcasts its block of B to all processors in column j
- Let's see it on a picture...

Communication Pattern: $k = 1$



The Outer-Product Algorithm

```
p = sqrt(num_procs());
int A[N/p][N/p],B[N/p][N/p],C[N/p][N/p];
int bufferA[N/p][N/p], bufferB[N/p][N/p];

(myrow,mycol) = my_2D_rank()

for (int k=0; k < p; k++) {
  // Broadcast A along rows
  BroadcastRow((myrow,k), A, bufferA, N/p * N/p);
  // Broadcast B along columns
  BroadcastColumn((k,mycol), B, bufferB, N/p * N/p);

  // Multiply Matrix blocks (assuming a convenient MatrixMultiplyAdd() function)
  if ((myrow == k) && (mycol == k))
    MatrixMultiplyAdd(C, A, B, N/p, N/p);
  else if (myrow == k)
    MatrixMultiplyAdd(C, bufferA, B, N/p, N/p);
  else if (mycol == k)
    MatrixMultiplyAdd(C, A, bufferB, N/p, N/p);)
  else
    MatrixMultiplyAdd(C, bufferA, bufferB, N/p, N/p);)
}
```

Performance Analysis

- β: time to do a row/column broadcast
- γ: time to compute a block
- No overlap: time = $p \times (\beta + \beta + \gamma)$
- Some overlap: time = $\beta + \beta + (p - 1)\max(\beta + \beta, \gamma) + \gamma$

- This algorithm is in fact asymptotically optimal

Grid vs. Ring

- On a Ring, with a 1-D data distribution we already have an asymptotically optimal algorithm (same ideas as for the 1-D Matrix Vector multiply)
- So who cares about this more complex algorithm?
- If N is huge, we don't care
- But in fact, using a 2-D distribution reduces communication costs
- The algorithm sends less data overall
- And it can be proven that even if the underlying platform is not a torus, the 2-D algorithm is still better than the 1-D algorithm

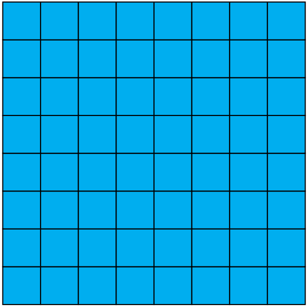
Other Matrix Multiplication Algorithms

- People have come up with many algorithms
  - Cannon (1969)
  - Fox (1987)
  - Snyder (1992)
  - ...
- They all correspond to “cruising” through the operations in different (possibly really confusing) order
- Some begin by shuffling things around in each matrix!

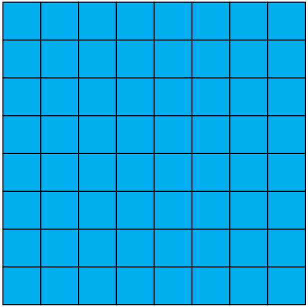
2-D Block Cyclic Distribution

- If $N >> p$, then blocks are large
- This can be a problem as it limits parallelism
- One “swiss army knife" solution is to use a 2-D Block Cyclic Distribution
  - Doesn’t matter which way the computation “moves", we should be ok...
- Let’s see what that looks like...

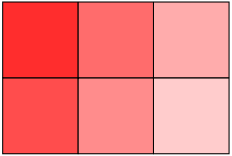
2-D Block Cyclic Distribution



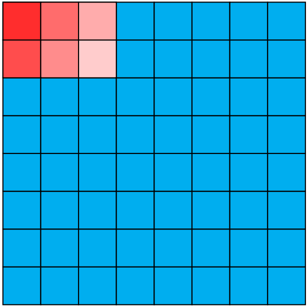
2-D Block Cyclic Distribution



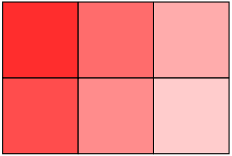
2×3 processor grid



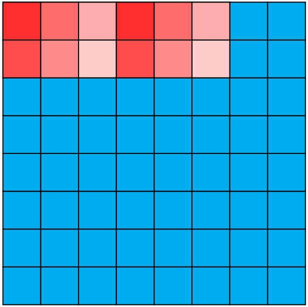
2-D Block Cyclic Distribution



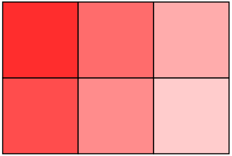
2×3 processor grid



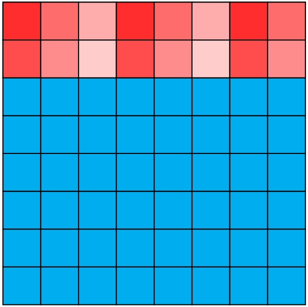
2-D Block Cyclic Distribution



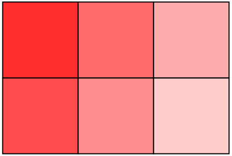
2×3 processor grid



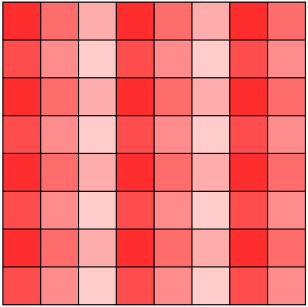
2-D Block Cyclic Distribution



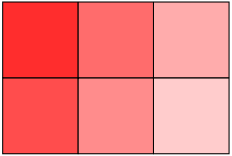
2×3 processor grid



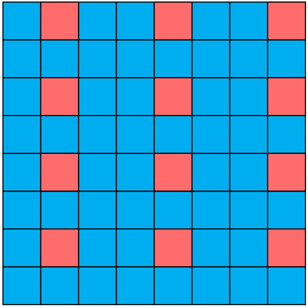
2-D Block Cyclic Distribution



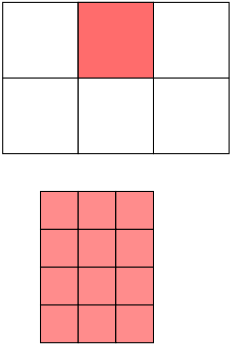
2×3 processor grid



What One Processor Holds



Processor (0,1)



Fun local/global indices computations!

```
int A[N/2][N/3];
```

2-D Block Cyclic Distribution

- All algorithms we've seen so far can be implemented with the 2-D Block Cyclic distribution
- If you don't abstract it away a bit, the code becomes horrendous
- But you'd get performance benefits from it
- For instance, the ScaLAPACK library recommends 2-D block cyclic distributions
    - It actually supports other distributions

Outline

What was all this????

- These lecture notes are representative of “traditional” parallel computing
- Similar algorithms have been studied for decades, their performance analyzed in depth
  - Using various models/assumptions
- We have a programming assignment on this topic...
- The main caveat of all this material is that when the assumptions break down, or when the platform becomes is very complex, then many difficulties arise

- For instance, when the platform is heterogeneous...