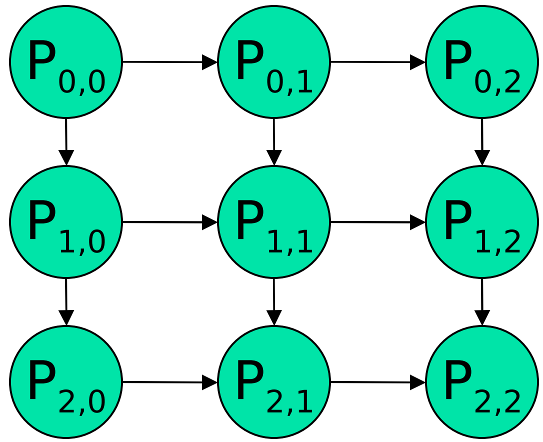
Principles of High Performance Computing (ICS 632)

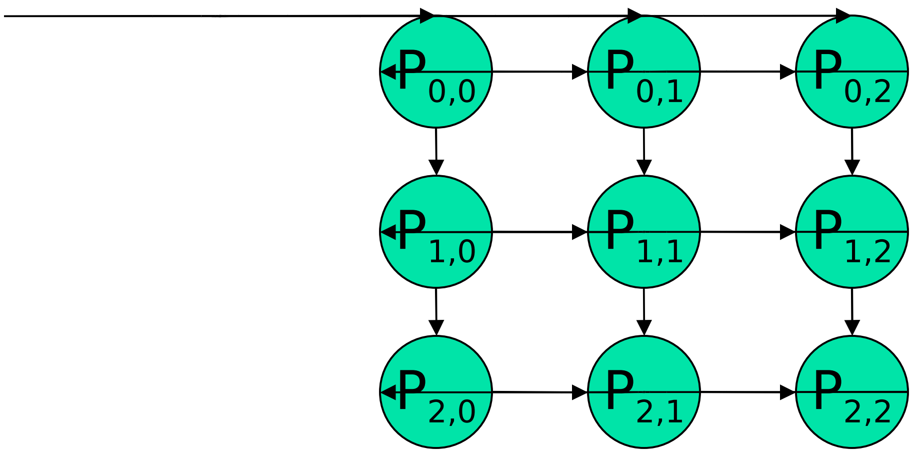
Algorithms on a Grid of Processors

2-D Grid (Chapter 5)



- Consider $p=q^2$ processors
- We can think of them arranged in a square grid
  - A rectangular grid is also possible, but we'll stick to square grids for most of our algorithms
- Each processor is identified as $P_{i,j}$
  - i: processor row
  - J: processor column

2-D Torus



- Wrap-around links from edge to edge
- Each processor belongs to 2 different rings
  - Will make it possible to reuse algorithms we develop for the ring topology
- Mono-directional links OR Bi-directional links
  - Depending on what we need the algorithm to do and on what makes sense for the physical platform

Concurrency of Comm. and Comp.

- When developing performance models we will assume that a processor can do all three activities in parallel
  - Compute
  - Send
  - Receive
- What about the bi-directional assumption?
  - Two models
    - Half-duplex: two messages on the same link going in opposite directions contend for the link’s bandwidth
    - Full-duplex: it’s as if we had two links in between each neighbor processors
  - The validity of the assumption depends on the platform

Multiple concurrent communications?

- Now that we have 4 (logical) links at each processor, we need to decide how many concurrent communications can happen at the same time
  - There could be 4 sends and 4 receives in the bi-directional link model
- If we assume that 4 sends and 4 receives can happened concurrently without loss of performance, we have a *multi-port* model
- If we only allow 1 send and 1 receive to occur concurrently we have a *single-port* model

So what?

- We have many options:
  - Grid or torus
  - Mono- or bi-directional
  - Single-or multi-port
  - Half- or full-duplex
- We'll mostly use the torus, bi-directional, full-duplex assumption
- We'll discuss the multi-port and the single-port assumptions
- As usual, it's straightforward to modify the performance analyses to match with whichever assumption makes sense for the physical platform

How realistic is a grid topology?

- Some parallel computers are built as physical grids (2-D or 3-D)
  - Example: IBM’s Blue Gene/L
- If the platform uses a switch with all-to-all communication links, then a grid is actually not a bad assumption
  - Although the full-duplex or multi-port assumptions may not hold
- We will see that even if the physical platform is a shared single medium (e.g., a non-switched Ethernet), it’s sometimes preferable to think of it as a grid when developing algorithms!

Communication on a Grid

- As usual we won’t write MPI here, but some pseudo code
- A processor can call two functions to known where it is in the grid:
  - My_Proc_Row()
  - My_Proc_Col()
- A processor can find out how many processors there are in total by:
  - Num_Procs()
  - Recall that here we assume we have a square grid
  - In programming assignment we may need to use a rectangular grid

Communication on the Grid

- We have two point-to-point functions
  - Send(dest, addr, L)
  - Recv(src, addr, L)
- We will see that it's convenient to have broadcast algorithms within processor rows or processor columns
  - BroadcastRow(i, j, srcaddr, dstaddr, L)
  - BroadcastCol(i, j, srcaddr, dstaddr, L)
- We assume that a a call to these functions by a processor not on the relevant processor row or column simply returns immediately
- How do we implement these broadcasts?

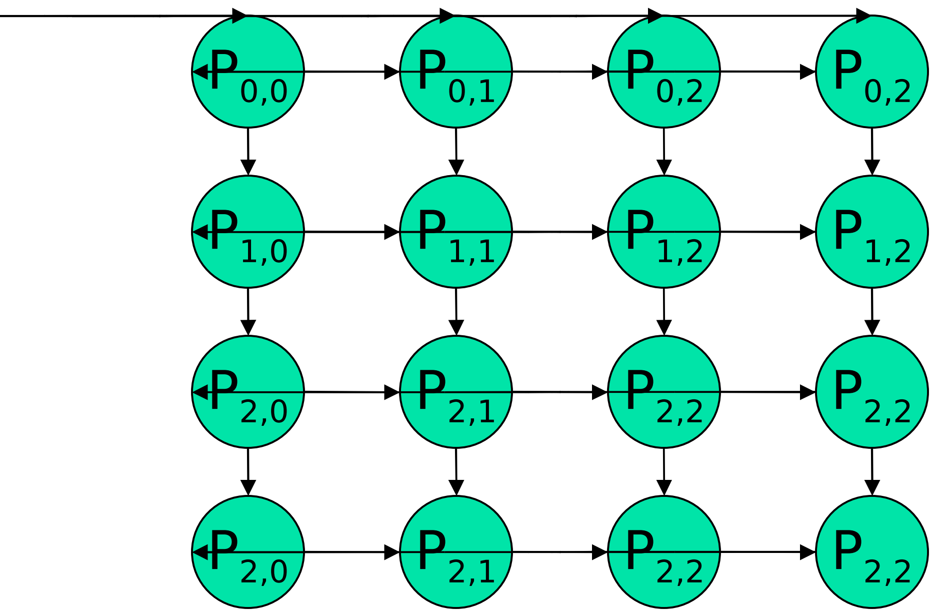
Row and Col Broadcasts?

- If we have a torus
  - If we have mono-directional links, then we can reuse the broadcast that we developed on a ring of processors
    - Either pipelined or not
  - It we have bi-directional links AND a multi-port model, we can improved performance by going both-ways simultaneously on the ring
    - We’ll see that the asymptotic performance is not changed
- If we have a grid
  - If links are bi-directional then messages can be sent both ways from the source processor
    - Either concurrently or not depending on whether we have a one-port or multi-port model
  - If links are mono-directional, then we can’t implement the broadcasts at all

Matrix Multiplication on a Grid

- Matrix multiplication on a Grid has been studied a lot because
  - Multiplying huge matrices fast is always important in many, many fields
    - Each year there is at least a new paper on the topic
  - It's a really good way to look at and learn many different issues with a grid topology
- Let's look at the natural matrix distribution scheme induced by a grid/torus

2-D Matrix Distribution



- We denote by $a_{i,j}$ an element of the matrix
- We denote by $A_{i,j}$ (or A_{ij}) the block of the matrix allocated to $P_{i,j}$

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

How do Matrices Get Distributed? (Sec. 4.7)

- Data distribution can be completely ad-hoc
- But what about when developing a library that will be used by others?
- There are two main options:
- 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
  - Allows for the library to make data distribution decisions transparently to the user
  - Prohibitively expensive if one does sequences of operations
    - and one almost always does so
- Distributed
  - when calling a function (e.g., matrix multiplication)
    - Assume that the input is already distributed
    - Leave the output distributed
  - May lead to having to “redistribute” data in between calls so that distributions match, which is harder for the user and may be costly as well
    - For instance one may want to change the block size between calls, or go from a non-cyclic to a cyclic distribution
- Most current software adopt the distributed approach
  - more work for the user
  - more flexibility and control
- We’ll always assume that the data is magically already distributed by the user

Four Matrix Multiplication Algorithms

- We’ll look at four algorithms
  - Outer-Product
  - Cannon
  - Fox
  - Snyder
- The first one is used in practice
- The other three are more “historical” but are really interesting to discuss
  - We’ll have a somewhat hand-wavy discussion here, rather than look at very detailed code

The Outer-Product Algorithm

- Consider the “natural” sequential matrix multiplication algorithm

```
for i=0 to n-1
    for j=0 to n-1
        for k=0 to n-1
            c[i,j] += a[i,k] * b[k,j]
```

  - This algorithm is a sequence of inner-products (also called scalar products)
- We have seen that we can switch loops around
- Let’s consider this version

```
for k=0 to n-1
    for i=0 to n-1
        for j=0 to n-1
            c[i,j] += a[i,k] * b[k,j]
```

  - This algorithm is a sequence of outer-products!

The Outer-Product Algorithm

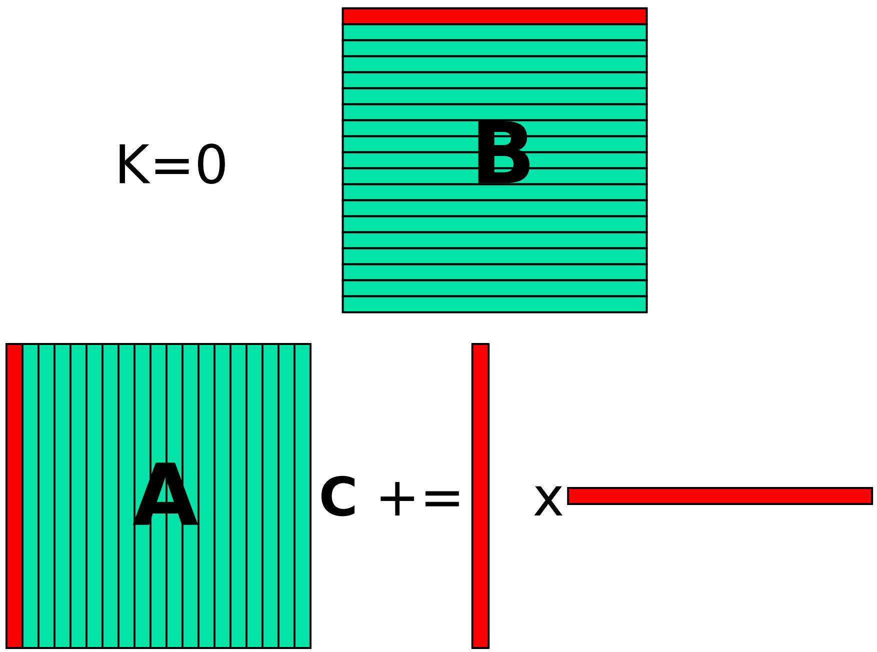
for k=0 to n-1

  for i=0 to n-1

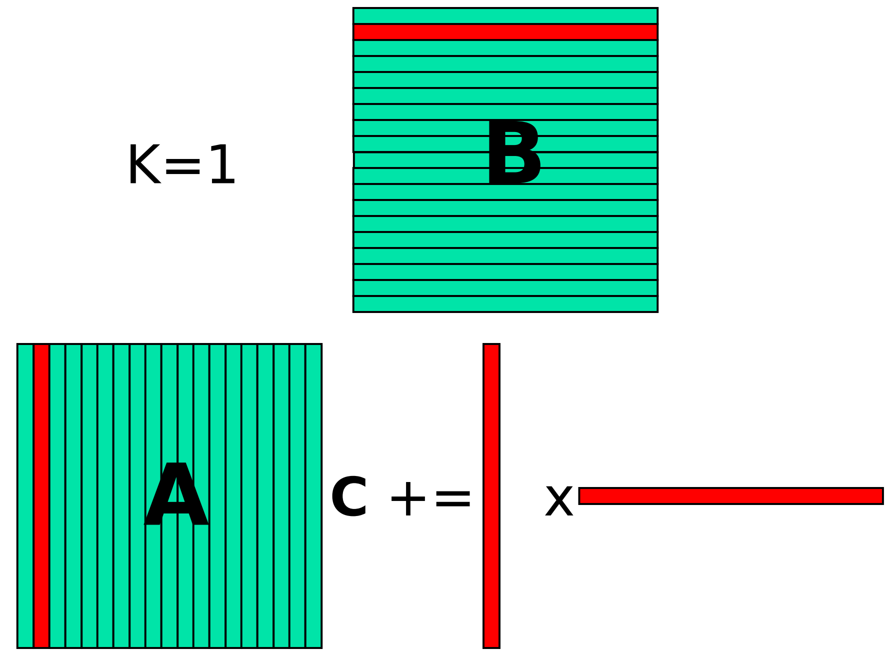
    for j=0 to n-1

      $c_{i,j} \mathrel{+}= a_{i,k} * b_{k,j}$

K=0



K=1



The outer-product algorithm

- Why do we care about switching the loops around to view the matrix multiplication as a sequence of outer products?
- Because it makes it possible to design a very simple parallel algorithm on a grid of processors!
- First step: view the algorithm in terms of the blocks assigned to the processors

```
for k=0 to q-1
  for i=0 to q-1
    for j=0 to q-1
      C_{i,j} += A_{i,k} * B_{k,j}
```

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

The Outer-Product Algorithm

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_0	B_0	B_0	B_0
B_1	B_1	B_1	B_1
B_2	B_2	B_2	B_2
B_3	B_3	B_3	B_3
0	1	2	3

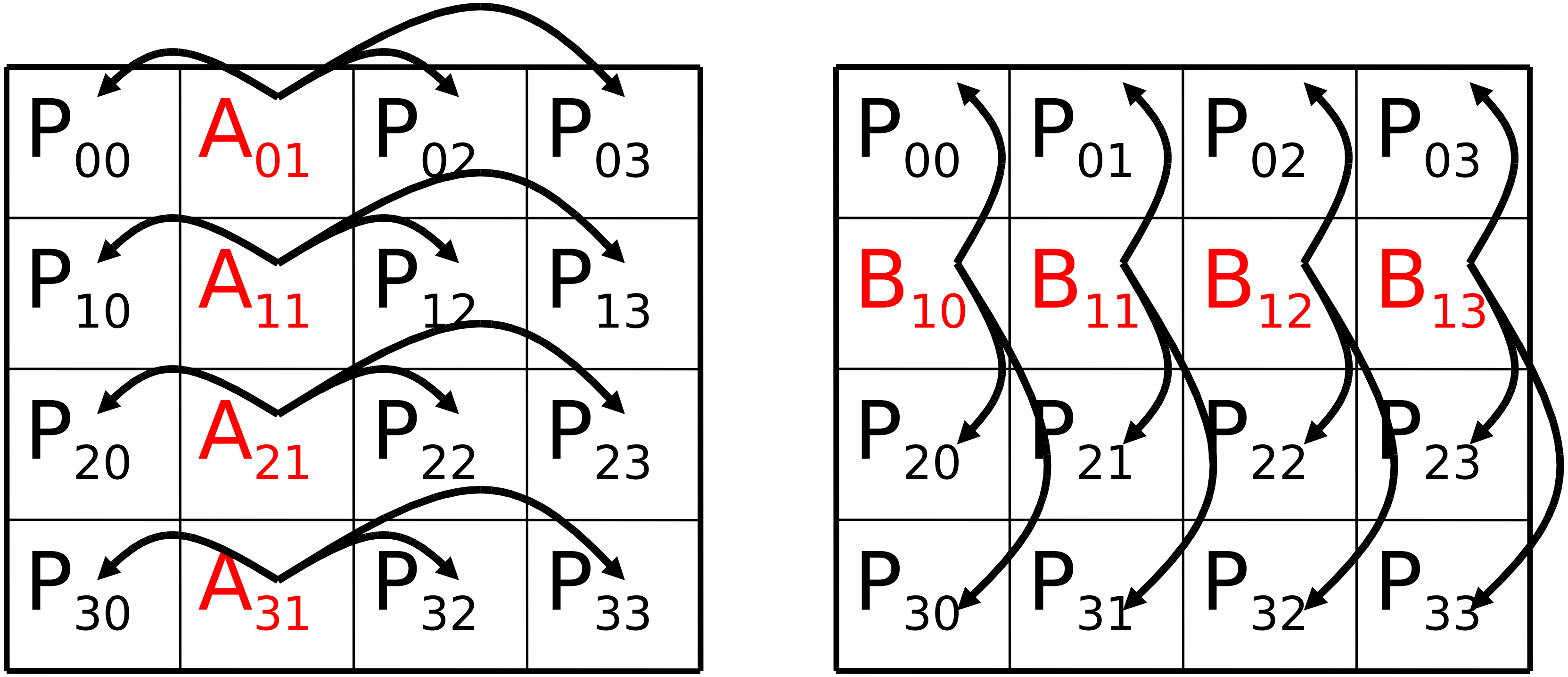
for k=0 to q-1
    for i=0 to q-1
        for j=0 to q-1
            $C_{i,j} += A_{i,k} * B_{k,j}$

- At step k, processor $P_{i,j}$ needs $A_{i,k}$ and $B_{k,j}$
  - If k = j, then the processor already has the needed block of A
    - Otherwise, it needs to get it from $P_{i,k}$
  - If k = I, then the processor already has the needed block of B
    - Otherwise, it needs to get it from $P_{k,j}$

The Outer-Product Algorithm

- Based on the previous statements, we can now see how the algorithm works
- At step k
  - Processor $P_{i,k}$ broadcasts its block of matrix A to all processors in processor row i
    - True for all i
  - Processor $P_{k,j}$ broadcasts its block of matrix B to all processor in processor column j
    - True for all j
- There are q-1 steps

The Outer Product Algorithm



Step k=1 of the algorithm

The Outer-Product Algorithm

```
// m = n/q
var A, B, C: array[0..m-1, 0..m-1] of real
var bufferA, bufferB: array[0..m-1, 0..m-1] of real
var myrow, mycol
myrow = My_Proc_Row()
mycol = My_Proc_Col()
for k = 0 to q-1
    // Broadcast A along rows
    for i = 0 to q-1
        BroadcastRow(i,k,A,bufferA,m*m)
    // Broadcast B along columns
    for j=0 to q-1
        BroadcastCol(k,j,B,bufferB,m*m)
    // Multiply Matrix blocks (assuming a convenient MatrixMultiplyAdd()
    function)
    if (myrow == k) and (mycol == k)
        MatrixMultiplyAdd(C,A,B,m)
    else if (myrow == k)
        MatrixMultiplyAdd(C,bufferA,B,m)
    else if (mycol == k)
        MatrixMultiplyAdd(C, A, bufferB, m)
    else
        MatrixMultiplyAdd(C, bufferA, bufferB, m)
```

Performance Analysis

- The performance analysis is straightforward
- With a one-port model:
  - The matrix multiplication at step k can occur in parallel with the broadcasts at step k+1
  - Both broadcasts happen in sequence
  - Therefore, the execution time is equal to:

$T(m,q) = 2\ Tbcast + (q-1)\ \max(2\ Tbcast,\ m^3 w) + m^3 w$

  - w: elementary += * operation
  - Tbcast: time necessary for the broadcast
- With a multi-port model:
  - Both broadcasts can happen at the same time

$T(m,q) = Tbcast + (q-1)\ \max(Tbcast,\ m^3 w) + m^3 w$

- The time for a broadcast, using the pipelined broadcast:

  $Tbcast = \left(\mathrm{sqrt}((q-2)L) + \mathrm{sqrt}(m^2 b) \right)^2$
- When n gets large: $T(m,q) \sim q\ m^3 = n^3 / q^2$
- Thus, asymptotic parallel efficiency is 1!

So what?

- On a ring platform we had already given an asymptotically optimal matrix multiplication algorithm on a ring in an earlier set of slides
- So what’s the big deal about another asymptotically optimal algorithm?
- Once again, when n is huge, indeed we don’t care
- But communication costs are often non-negligible and do matter
    - When n is “moderate”
    - When w/b is low
- It turns out, that the grid topology is advantageous for reducing communication costs!

Ring vs. Grid

- When we discussed the ring, we found that the communication cost of the matrix multiplication algorithm was: $n^2 b$
  - A each step, the algorithm sends n^2/p matrix elements among neighboring processors
  - There are p steps
- For the algorithm on a grid:
  - Each step involves 2 broadcasts of n^2/p matrix elements
    - Assuming a one-port model, not to give an “unfair” advantage to the grid topology
  - Using a pipelined broadcast, this can be done in approximately the same time as sending n^2/p matrix elements between neighboring processors on each ring (unless n is really small)
  - Therefore, at each step, the algorithm on a grid spends twice as much time communicating as the algorithm on a ring
  - But it does sqrt(p) fewer steps!
- **Conclusion**: the algorithm on a grid spends at least sqrt(p) less time in communication than the algorithm on a ring

Grid vs. Ring

- Why was the algorithm on a Grid much better?
- Reason: More communication links can be used in parallel
  - Point-to-point communication replaced by broadcasts
  - Horizontal and vertical communications may be concurrent
  - More network links used at each step
- Of course, this advantage isn't really an advantage if the underlying physical platform does not really look like a grid
- But, it turns out that the 2-D distribution is inherently superior to the 1-D distribution, no matter what the underlying platform is!

Grid vs. Ring

- On a ring
  - The algorithm communicates p matrix block rows that each contain n^2/p elements, p times
  - Total number of elements communicated: pn^2
- On a grid
  - Each step, 2sqrt(p) blocks of n^2/p elements are sent, each to sqrt(p)-1 processors, sqrt(p) times
  - Total number of elements communicated: $2\text{sqrt}(p)n^2$
- Conclusion: the algorithm with a grid in mind inherently sends less data around than the algorithm on a ring
- Using a 2-D data distribution would be better than using a 1-D data distribution even if the underlying platform were a non-switched Ethernet for instance!
  - Which is really 1 network link, and one may argue is closer to a ring (p comm links) than a grid (p^2 comm links)

Conclusion

- Writing algorithms on a grid topology is a little bit more complicated than in a ring topology
- But there is often a payoff in practice and grid topologies are very popular