Distributed Memory Programming

ICS632: Principles of High Performance Computing

Henri Casanova (henric@hawaii.edu)

Fall 2015

Foreword

- This set of lecture notes will not be a theoretical treatment of parallel algorithms
  - But we will have pragmatic performance analyses
- Instead we take a more engineering approach, showcasing useful techniques for well-known parallel algorithms
  - I won't write MPI code here, just C-looking pseudo-code
- But first, let's talk a bit about "Gustafson's law"

Beyond Amdahl's Law

- Now that we're talking about distributed memory computing, we have the option to achieve massive scales (e.g., hundreds of thousands of processors)
- Amdahl's Law is bad news: your application just won't scale
- The law assumes that the work of the application is fixed
- But in practice, if one gives you a lot of processors you'll be running a larger application
  - Give a scientist more compute power, they'll try to solve bigger problems
  - Especially to use the whole aggregate memory
- Gustafson's law: Allow the total amount of to work scale linearly with the number of processors

Gustafson's Law

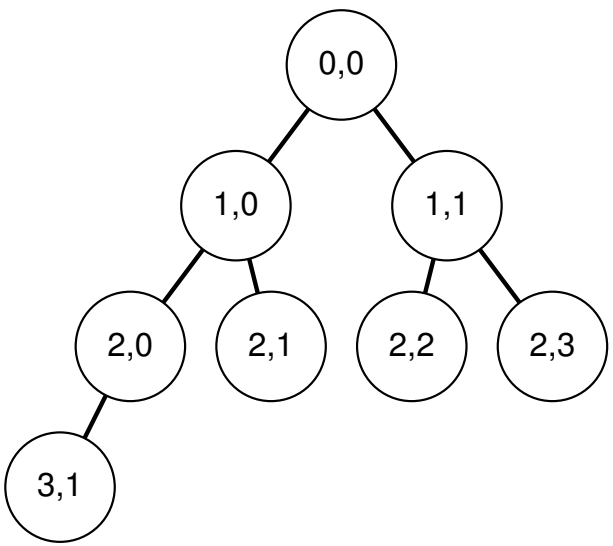
- Let p be the number of processors
- We assume that each time we add a processor, we also add some work to do
- Total work $= a + pb$ (a: sequential time; pb: parallelizable time)
- Parallel time $= a + b$
- Speedup $= (a + pb)/(a + b)$
- The Gustafson constant: $\alpha = a/(a + b)$
- Speedup $= p - \alpha(p - 1)$
- This is called the Scaled Speedup
- It is much higher than the regular speedup
- So if you're application can scale its work, you should be ok

Outline

Parallel Algorithms

- When designing a parallel(ized) algorithm, one typically reasons based on some logical structure imposed on the processors
- This structure corresponds to particular communication patterns of the algorithm
- For instance, it may be very convenient to think of the processors arranged in a ring (rank i communicates with rank $i + 1$)
  - The underlying platform may not be a physical ring
- MPI only gives us linear ranks between 0 and $p - 1$
- So often in an algorithm we *re-number* the processes to impose our own logical structure
  - Your own abstraction to enforce notions of “neighbors"

Virtual Topology Example



- $(i,j) = (\lfloor \log_2(rank) \rfloor, rank \mod 2^i)$
- $rank = 2^i + j$
- my_parent(i,j) = $(i-1, \lfloor j/2 \rfloor)$
- my_lchild(i,j) = $(i+1, 2j)$, if any
- my_rchild(i,j) = $(i+1, 2j+1)$, if any

```
MPI_Send(...,my_parent(i,j),...)
MPI_Recv(...,my_rchild(i,j),...)
```

Typical Topologies

- Many logical topologies have been proposed/used
  - Rings, Grids, Tori
  - Binary Trees, k-Nomial Trees, One-level Trees
  - Particular Graphs
- Often picked to suit the program's objectives
  - In Programming Assignment #3, we implemented a bintree broadcast using a bintree virtual topology
- How well does the logical topology fit the physical topology?
  - Back to the graph-embedding problem...
- Luckily, many physical topologies are switch-based and many virtual topologies work pretty well in practice

Data Distribution

- Given a virtual topology, one must "distribute" the data
  - Decide for each data item which process holds it
  - Involves arithmetic to "map" data item indices to process indices in the virtual topology
  - Looks like what we did in Assignment #2
- We often assume that the data is *already distributed*
  - Could be generated in-place
  - Could be actually distributed via data movements
- This is what many parallel libraries assume, so that the user is responsible for getting data items in the right place before calling library functions

Logical Ring Topology

- In these lecture notes we assume a bidirectional p-processor ring topology
- What this means for our implementation:
  - Process $rank$ can communicate only with processes $(rank + 1) \mod p$ and $(rank - 1) \mod p$.
- Of course the underlying physical topology is not a ring
- And MPI doesn't limit communicating rank pairs
- But we can enforce that limitation ourselves hoping that:
  1. It makes the code simple
  2. It still allows good performance

Outline

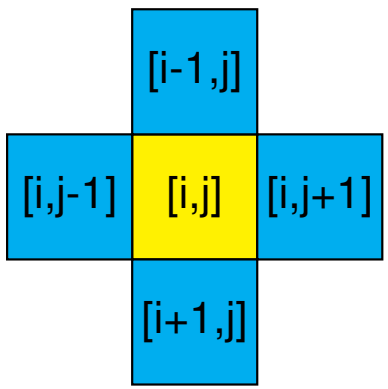
Outline

Easy Stencil Application

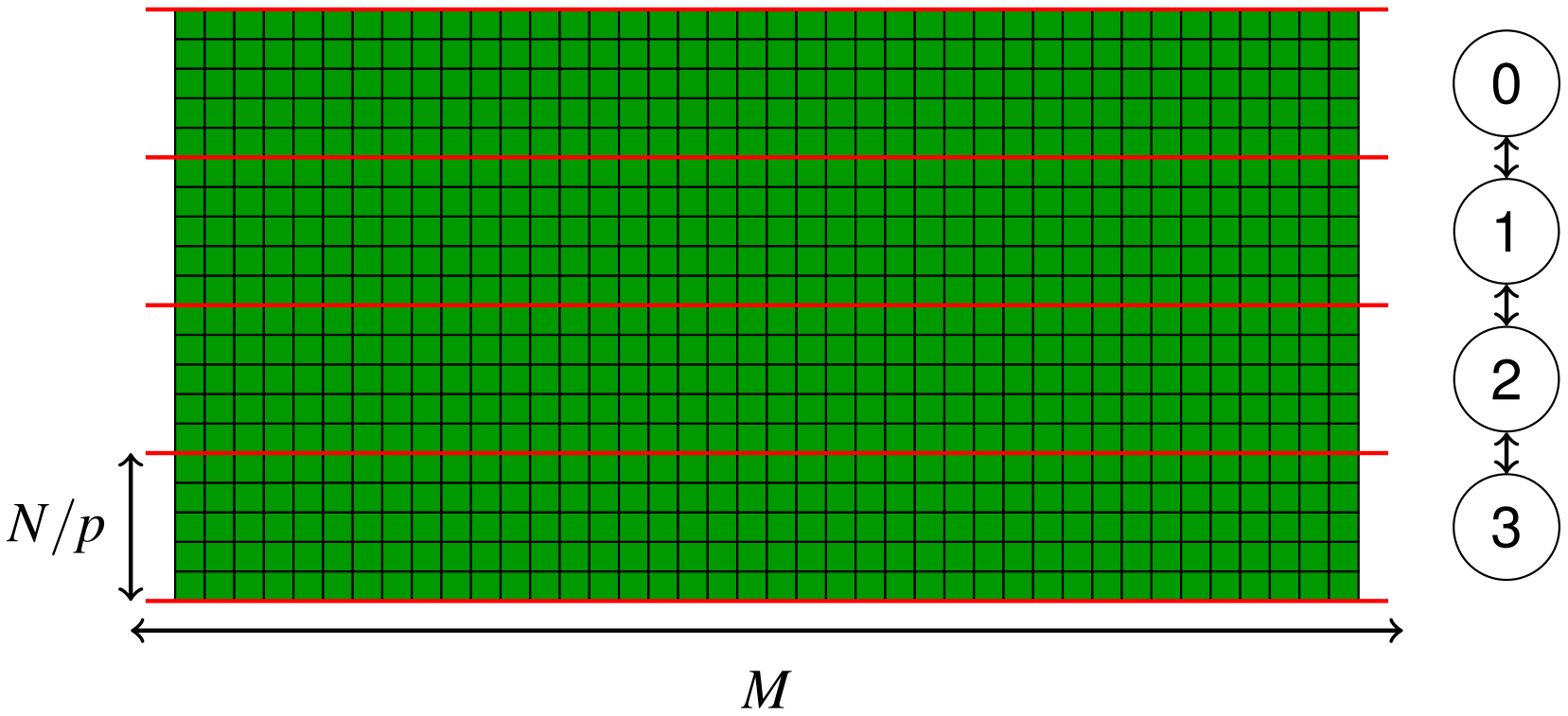
Stencil

```
int A[N][M], B[N][M];
int *tmp, *cur, int *new;
int t,i,j;

cur = &(A[0][0]); // cur points to A
new = &(B[0][0]); // new points to B

// Initialize A's content
...

// 1000 iterations
for (t=0; t < 1000; t++) {
  for (i=1; i < N-1; i++) {
    for (j=1; j < M-1; j++) {
      new[i*N+j] = update(cur[i*N+j],cur[i*N+j-1],cur[i*N+j+1],
                          cur[(i-1)*N+j],cur[(i+1)*N+j]);
    }
  }
  // Swap array pointers
  tmp = cur; cur = new; new = tmp;
}
```

Update based on current value and **old** values of neighbors

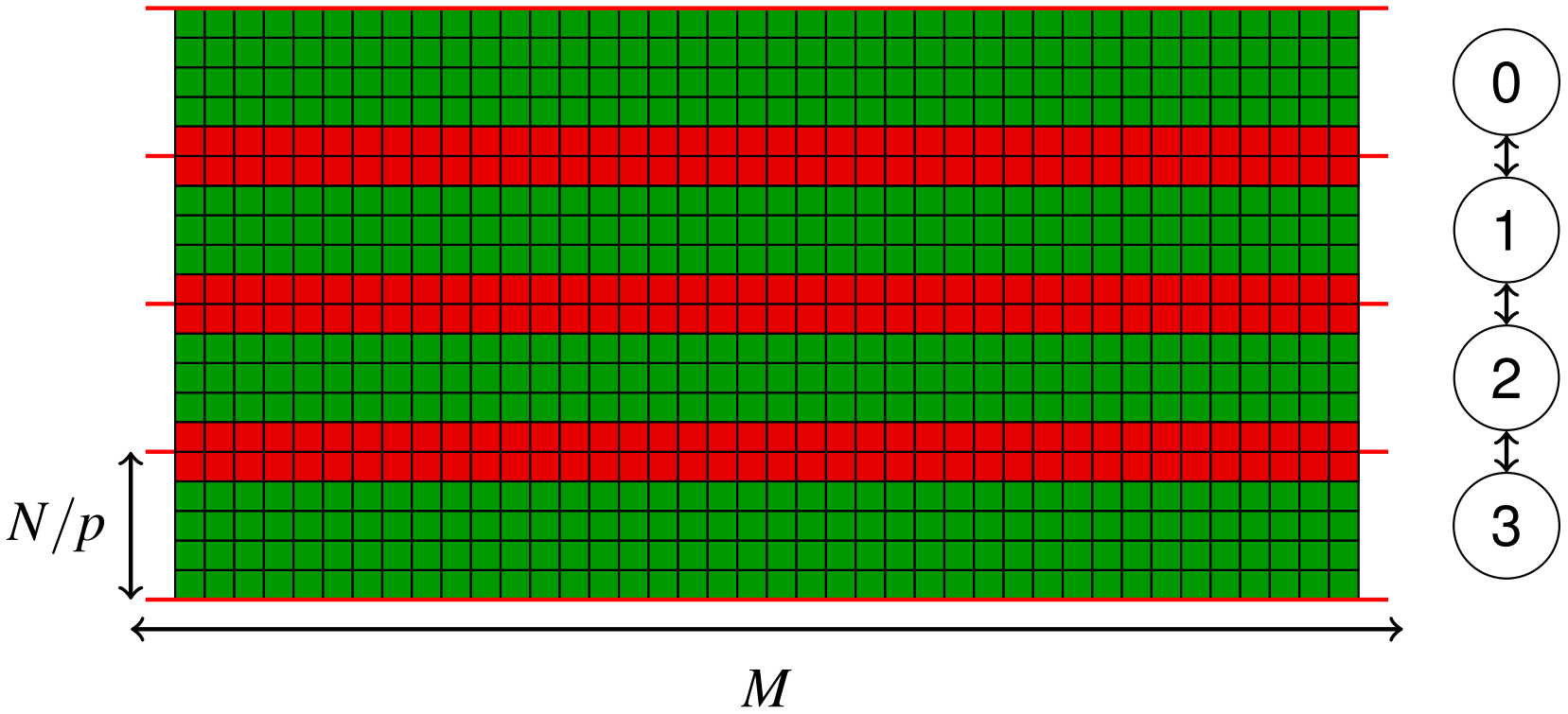


Data Distribution



- Each of the p processes allocates a $N/p \times M$ array

Data Distribution



- These red cells require values from neighbors

Distributed Memory Code

Parallel stencil sketch

```
int A[N*M/p], B[N*M/p];
...

for (t=0; t < 1000; t++) {
  [ send row 0 to rank-1]
  [ recv row from rank-1 ]
  [ send row N/p-1 to rank+1 ]
  [ recv red row from rank+1 ]
  < update my green row(s) >
  < update my red row(s) >
  < swap buffers as in sequential version >
}
```

- Note that the real code would be more complex because some processes have only one neighbor
- We assume a bi-directional ring, so if links are not full-duplex we will have contention, which may be ok

Using Non-Blocking Communications

Parallel stencil sketch

```
int A[N*M/p], B[N*M/p];
...

for (t=0; t < 1000; t++) {
  [ send row 0 to rank-1, asynchronously ]
  [ send row N/p-1 to rank+1, asynchronously ]
  < update my green row(s) >
  [ wait to receive red row from rank-1 ]
  [ wait to receive red row from rank+1 ]
  < update my red row(s) >
  < swap buffers as in sequential version >
}
```

- One can use asynchronous communication (`MPI_Isend`, `MPI_Irecv`) for these communications
- If the time to send/receive rows is shorter than the time to update green cells at a processor, then *communication is fully hidden*
- Depends on network speed, computing speed, size of the domain, and number of processors
- With fully hidden communication one can hope for 100% parallel efficiency

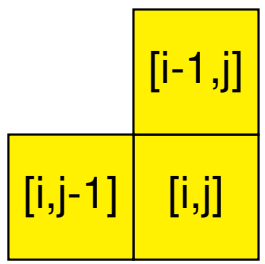
Outline

Less Easy Stencil Application

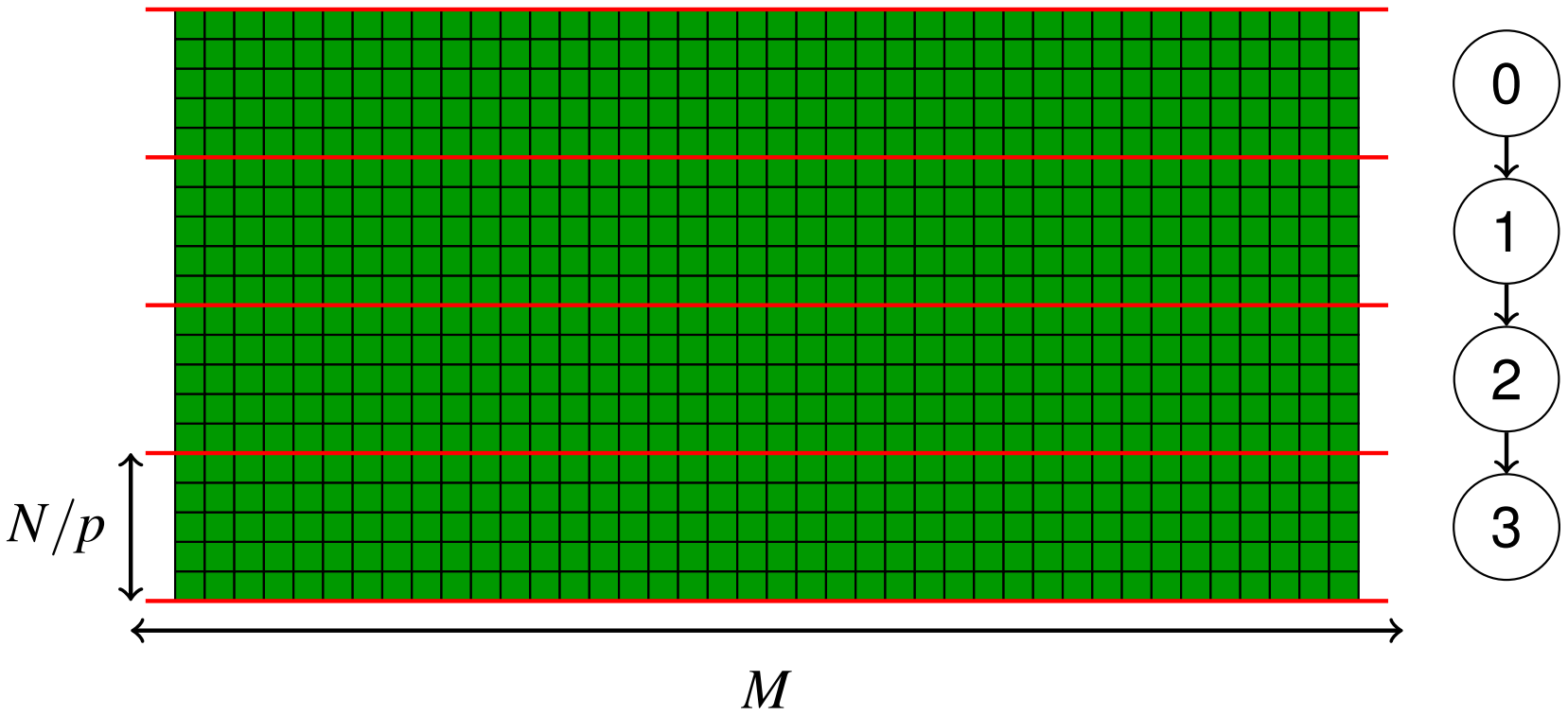
Stencil

```
int A[N*M];
int t,i,j;

// Initialize A's content
...

// 1000 iterations
for (t=0; t < 1000; t++) {
  for (i=1; i < N; i++) {
    for (j=1; j < M; j++) {
      A[i*N+j] = update(A[i*N+j],A[i*N+j-1],A[i*N+j+1],
                        A[(i-1)*N+j],A[(i+1)*N+j]);
    }
  }
}
```

Update based on current value and current values of West and North neighbors

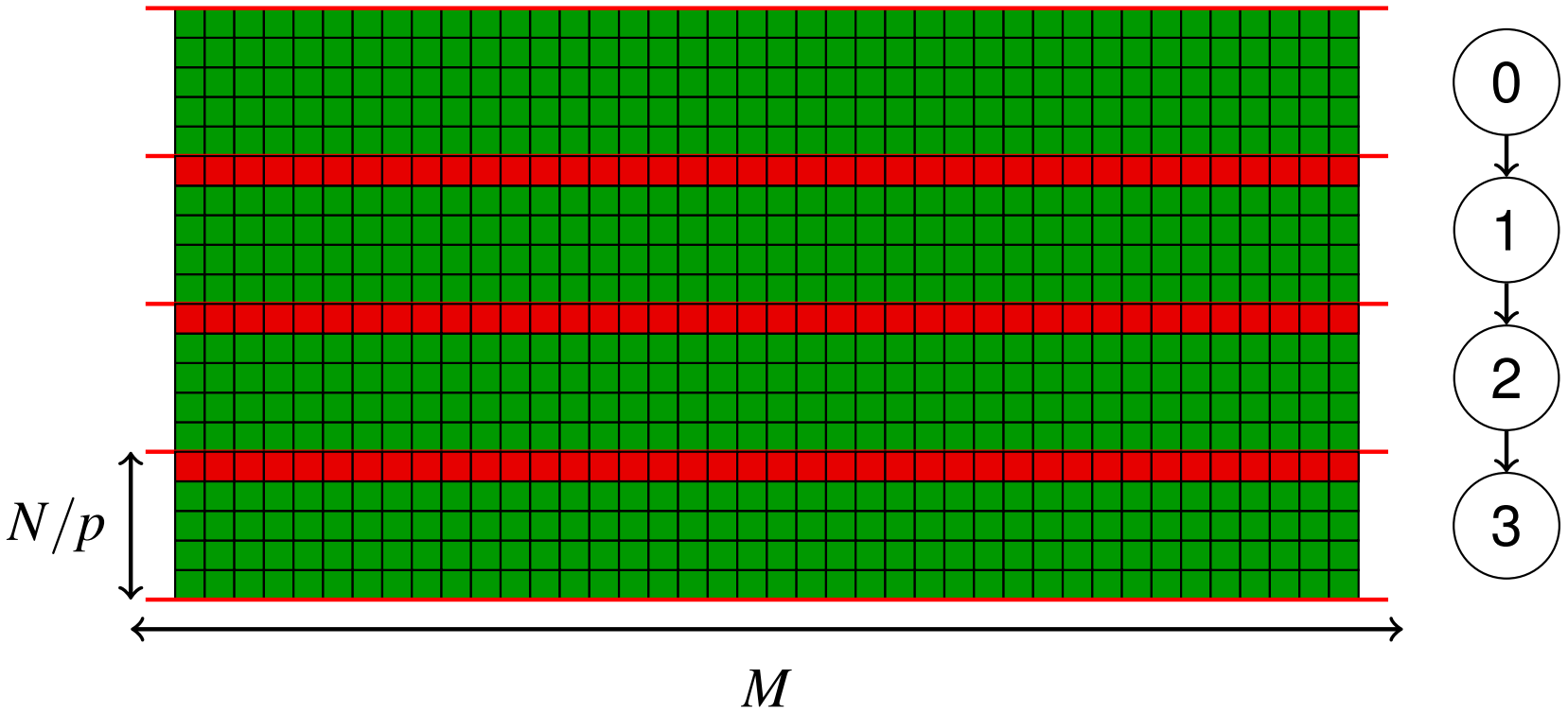


Same Data Distribution as Before



- Each of the p processes allocates a $N/p \times M$ array

Same Data Distribution as Before



- These red cells require values from neighbors

Naïve Algorithm (one iteration)

Stencil

```
p = num_procs();
rank = my_rank();
int A[N/p+1][M]; // One extra row at each process to hold
                 // the received row from neighbor

if (rank != 0) {
  // Receive my predecessor's last row
  receive(&(A[0][0]),N)
}
// Update all my green cells
for (i=1; i < N/p; i++) {
  for (j=0; j < M; j++) {
    update(i,j)
  }
}
if (rank != p-1) {
  // Send my last row to rank r+1
  send(&(A[N/p-1][0]),N)
}
```

It this code good?

Naïve Algorithm (one iteration)

Stencil

```
p = num_procs();
rank = my_rank();
int A[N/p+1][M]; // One extra row at each process to hold
                 // the received row from neighbor

if (rank != 0) {
  // Receive my predecessor's last row
  receive(&(A[0][0]),N)
}
// Update all my green cells
for (i=1; i < N/p; i++) {
  for (j=0; j < M; j++) {
    update(i,j)
  }
}
if (rank != p-1) {
  // Send my last row to rank r+1
  send(&(A[N/p-1][0]),N)
}
```

It this code good?

No!!! It's **sequential**

Making it parallel

- This code is sequential because process $r + 1$ has to wait for process r to finish computing all its rows
- What we need:
  - Process r should compute the elements of its last row as early as possible
  - Each element should be sent to process $r + 1$ *at once*, without waiting for the whole row to be computed
- One option is to have each process go down columns first rather than rows
- Let's try this....

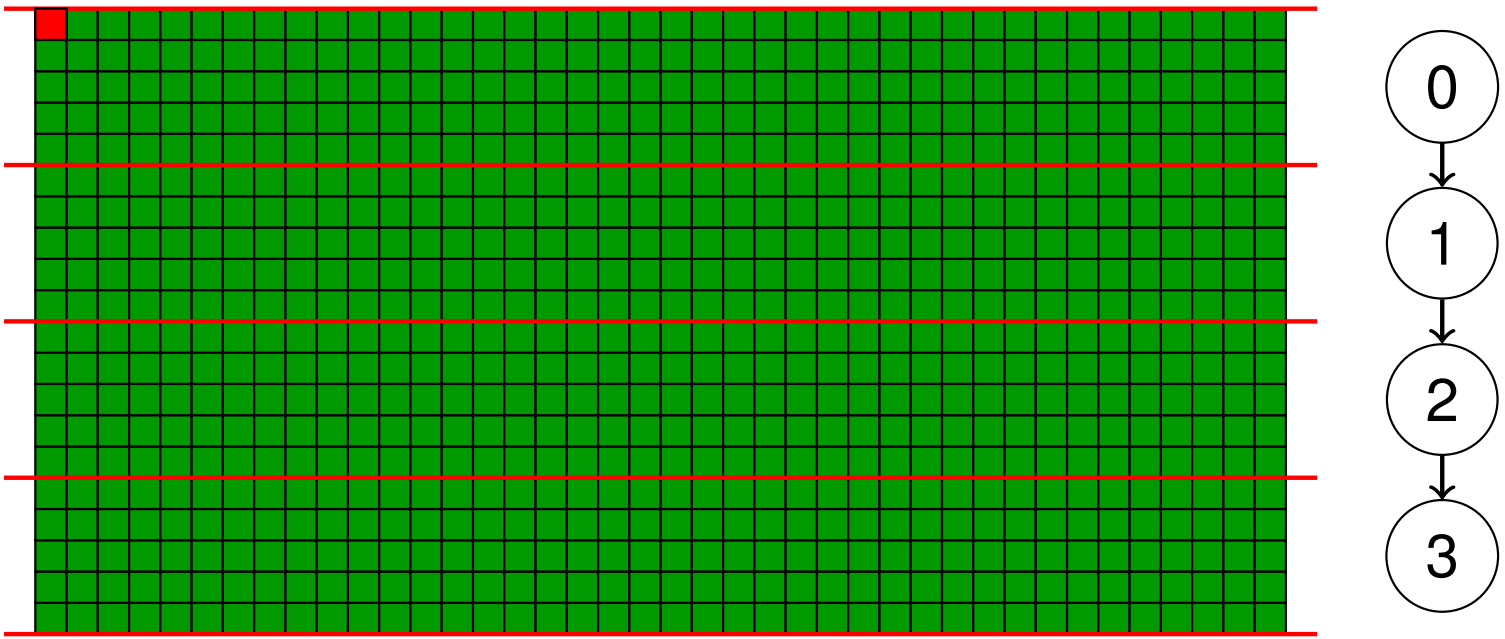
Less Naïve Algorithm (one iteration)

Stencil

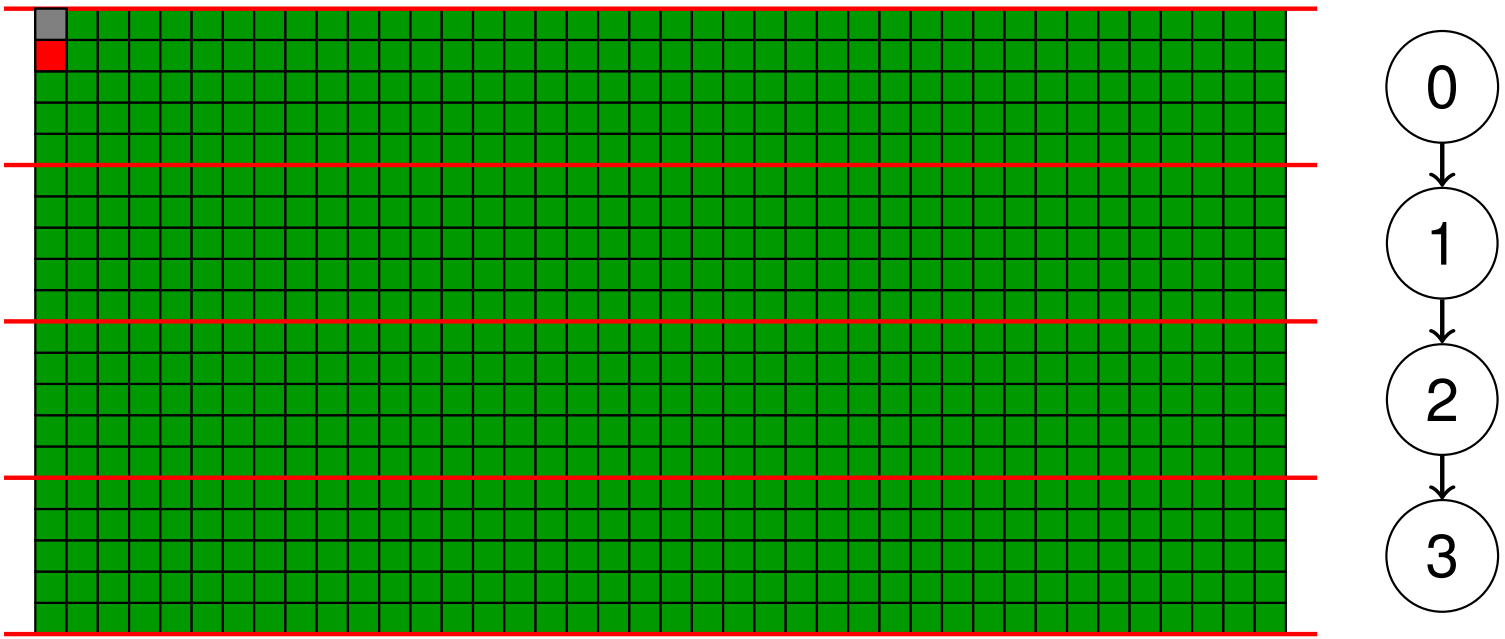
```
p = num_procs();
rank = my_rank();
int A[N/p+1][M]; // One extra row at each process to hold
                 // the received row from neighbor

for (j=0; j < M; j++) {
  for (i=0; i < N/p; i++) {
    if (rank != 0) {
      // Receive my predecessor's last element in column j
      receive(&(A[0][j]))
    }
    // Update all my green cells in column j
    for (i=1; i < N/p; i++) {
      update(i,j)
    }
    if (rank != p-1) {
      // Send my last element in column j to rank r+1
      send(A[N/p-1][j])
    }
  }
}
```

Let's visualize the order of computation step by step...

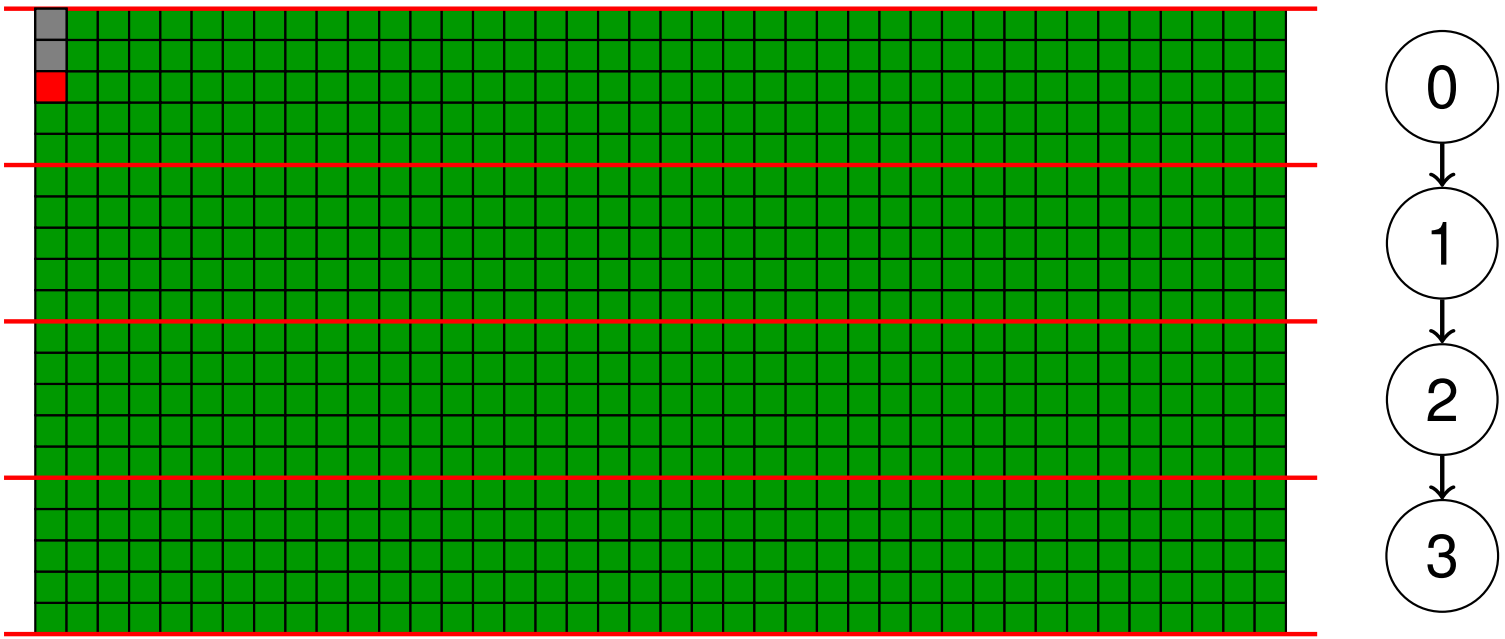
Parallel Execution



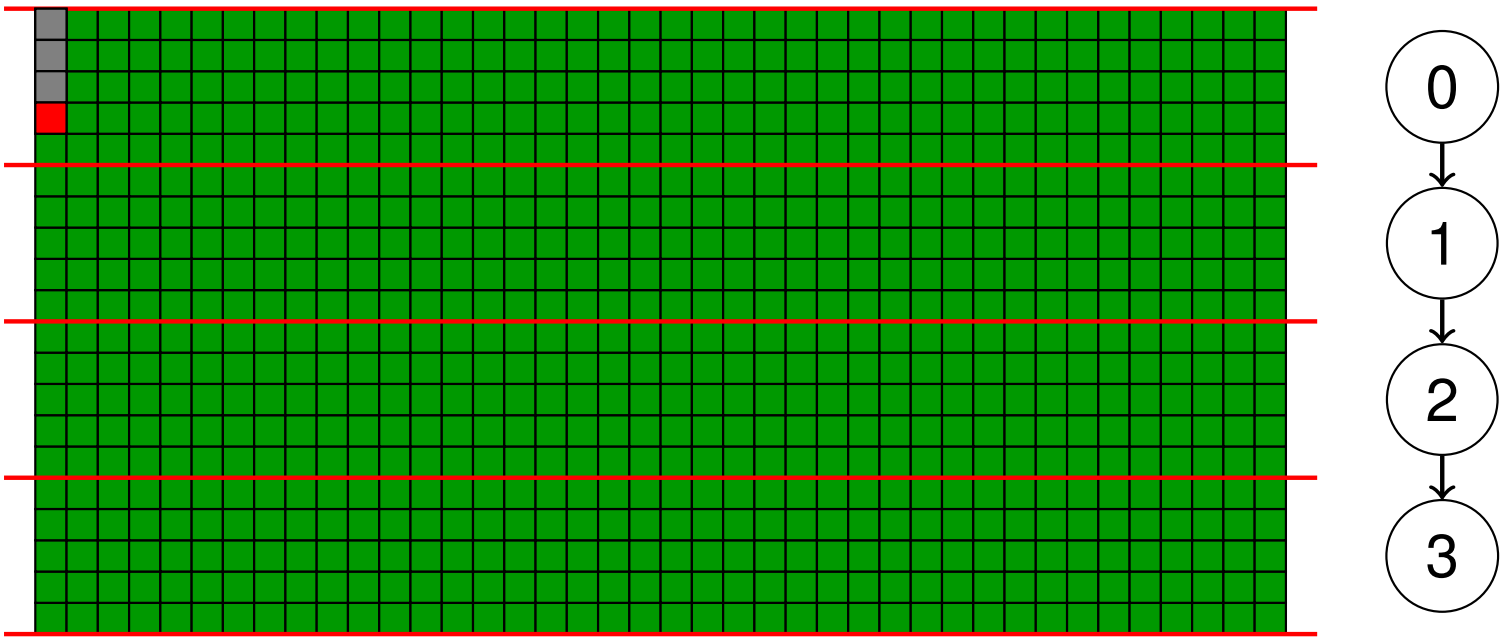
Parallel Execution



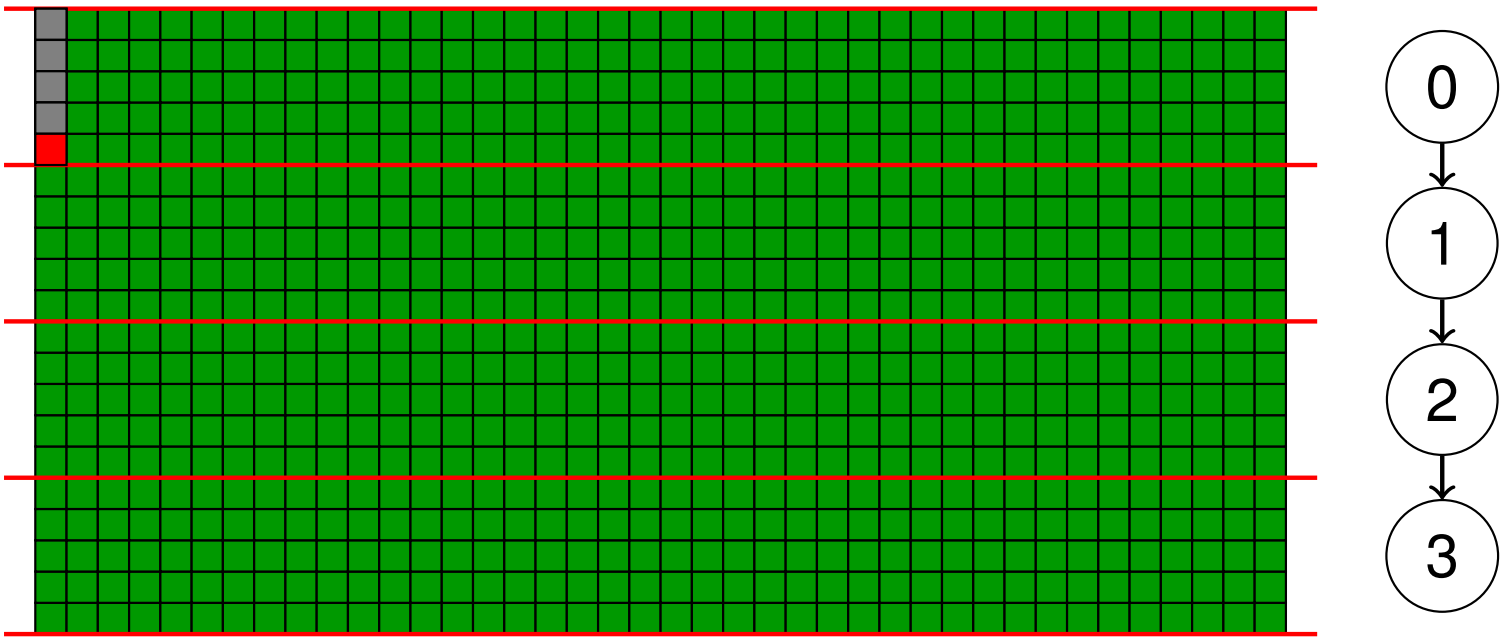
Parallel Execution



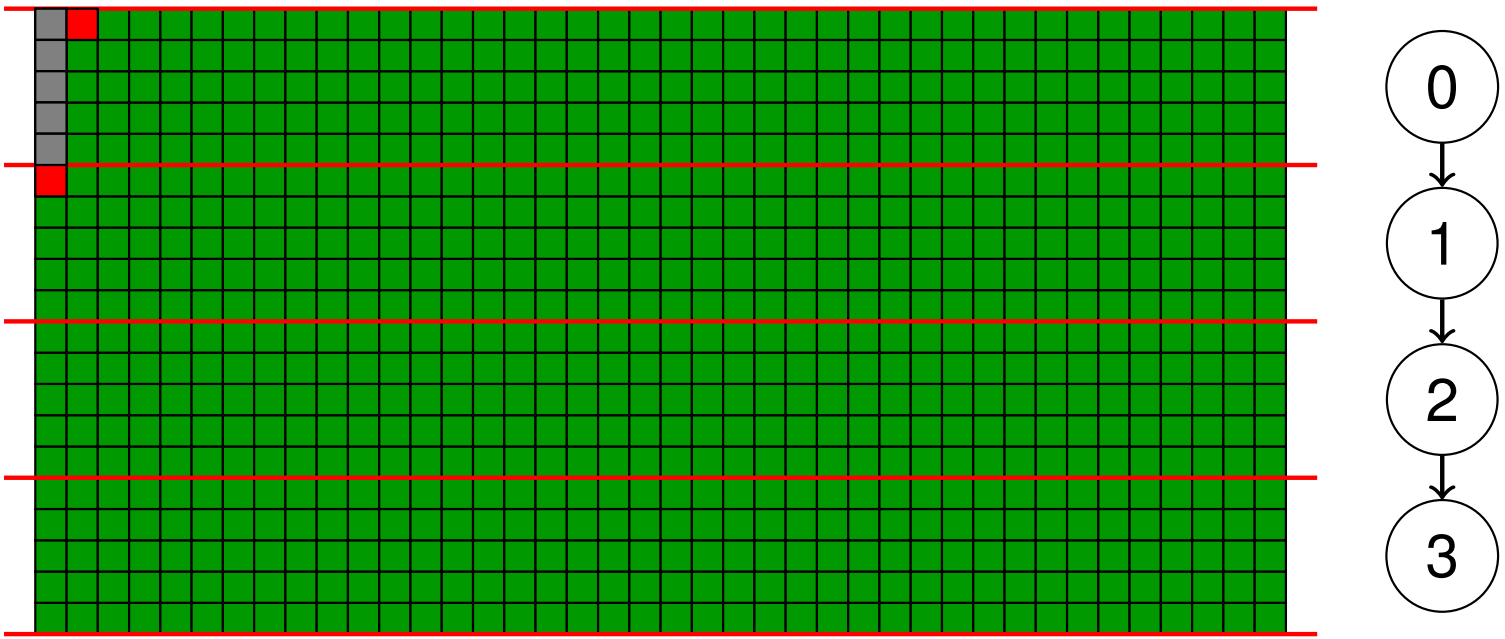
Parallel Execution



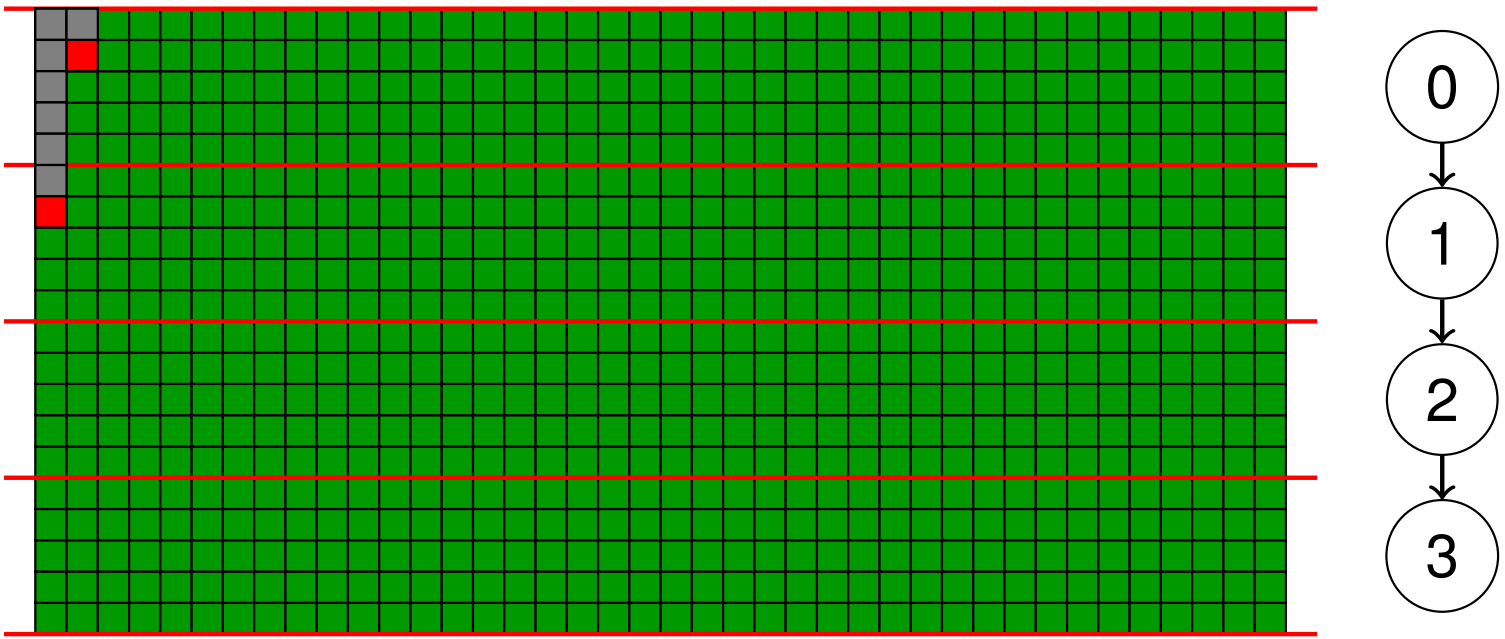
Parallel Execution



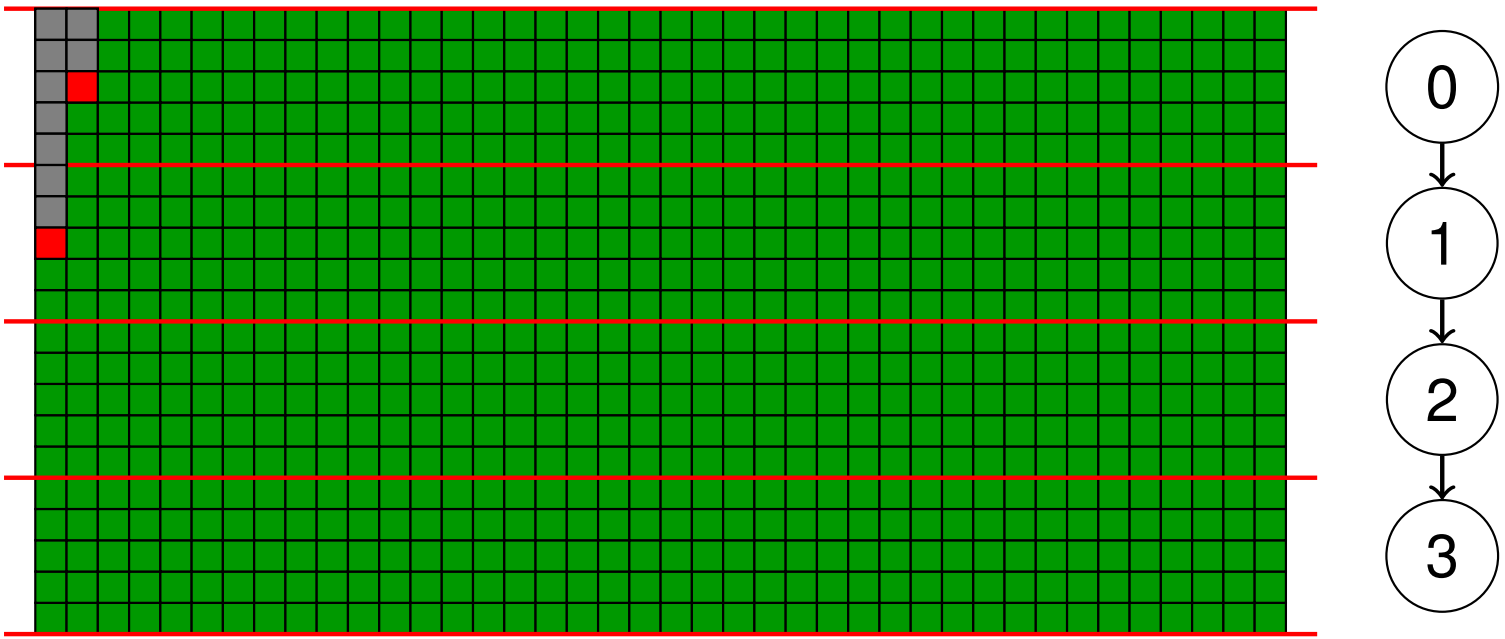
Parallel Execution



Parallel Execution



Parallel Execution



Parallel Speedup

- Let's assume an infinitely fast network so that communicating cells takes zero time
- Let c be the time to update a cell
- Let's figure out the parallel execution time... any ideas?

Parallel Speedup

- Let's assume an infinitely fast network so that communicating cells takes zero time
- Let c be the time to update a cell
- The last process begins computing at time: $(p-1) \times (N/p) \times c$
- It then computes for time $(N/p) \times M \times c$ time units
- It is the last one to finish computing, so the overall parallel execution time if $(p-1) \times (N/p) \times c + (N/p) \times M \times c$
- The sequential execution time is $N \times M \times c$
- So the parallel speedup is: $pM/(p-1+M)$
- If $M \to +\infty$, then speedup $\to p$

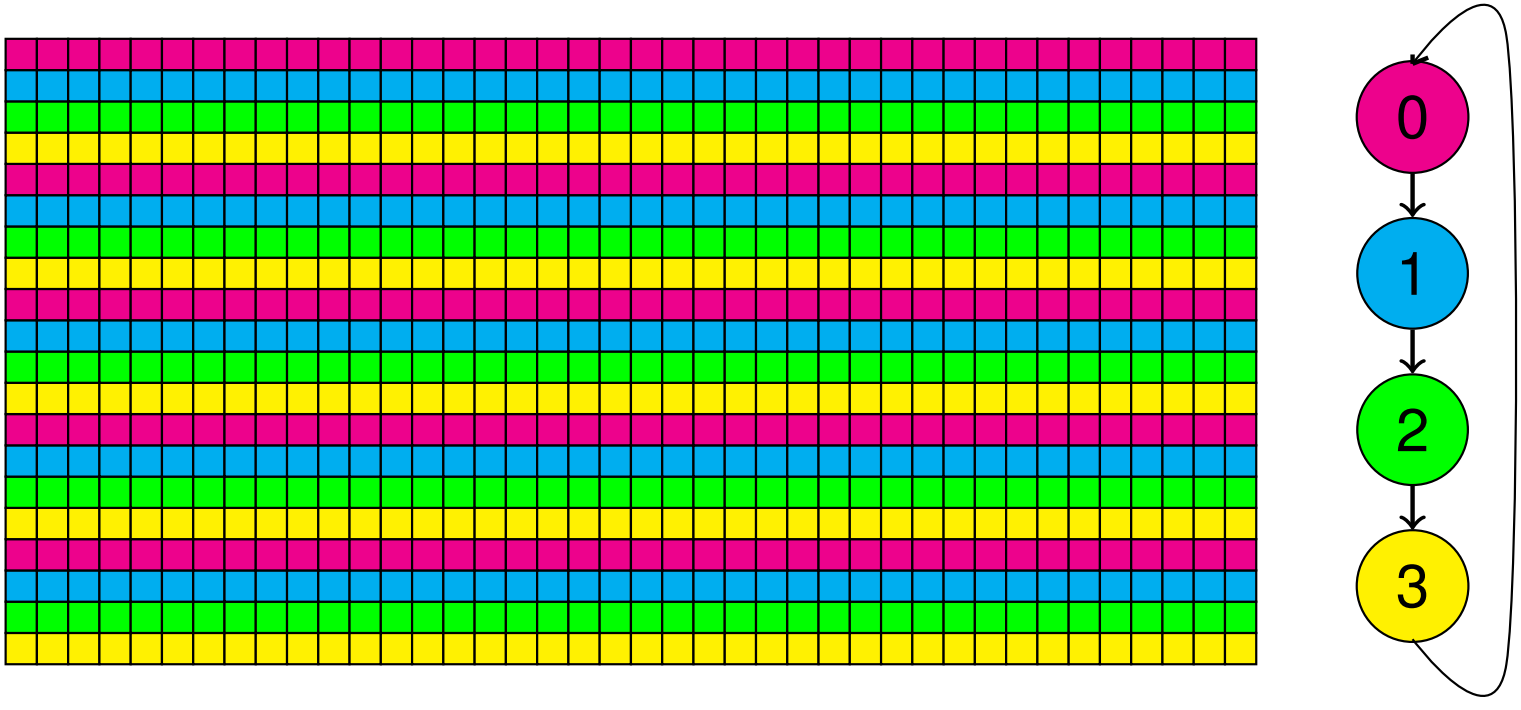
Can we do better?

- Our algorithm is *asymptotically optimal*
  - It has asymptotically optimal parallelism
- But if M isn't very large compared to p, then we're not in great shape
  - Parallelism is not great because the last processor starts computation "late"
- How can we do better?
- How can we have each process start computing as early as possible? Any idea?

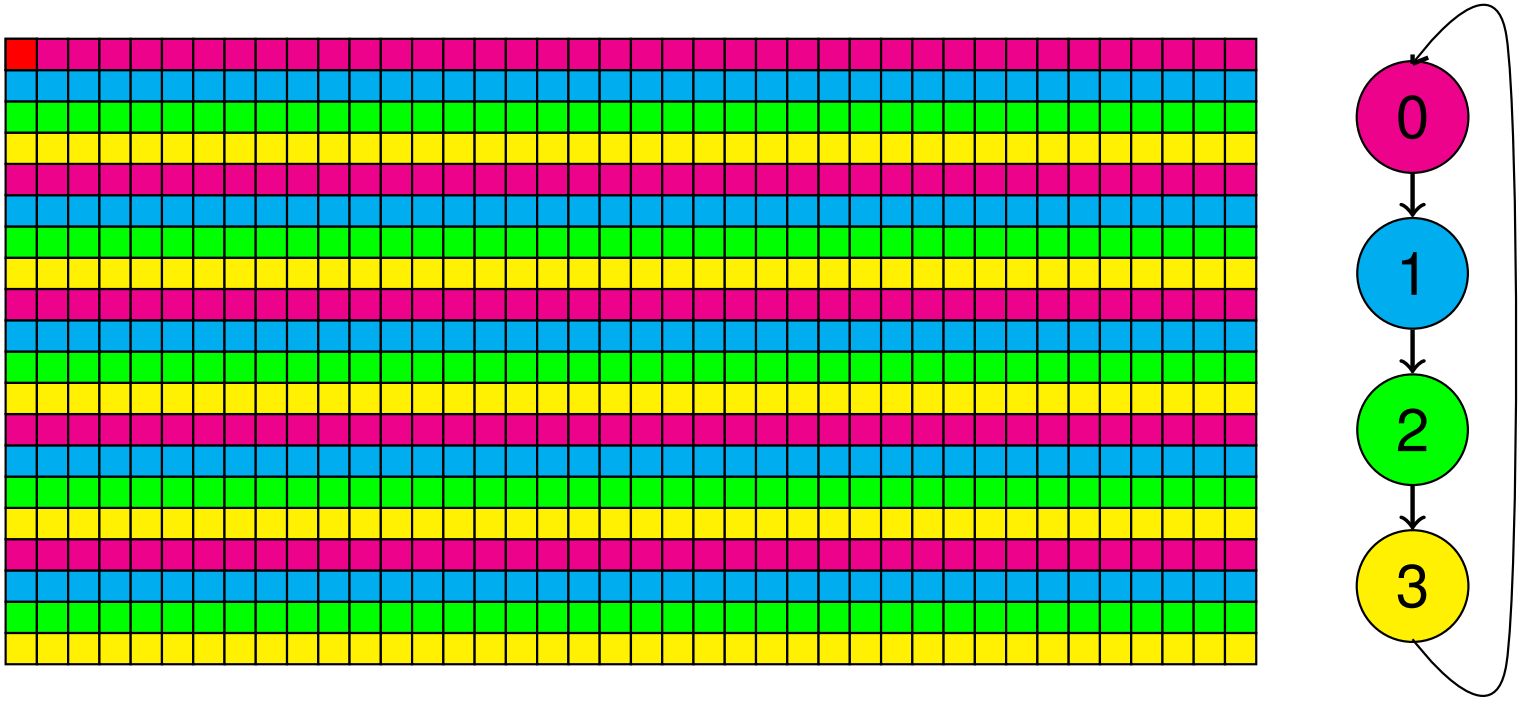
Can we do better?

- Our algorithm is *asymptotically optimal*
  - It has asymptotically optimal parallelism
- But if M isn't very large compared to p, then we're not in great shape
  - Parallelism is not great because the last processor starts computation "late"
- How can we do better?
- We can use a cyclic data distribution
- Processor r is assigned row i if $i \mod p = r$

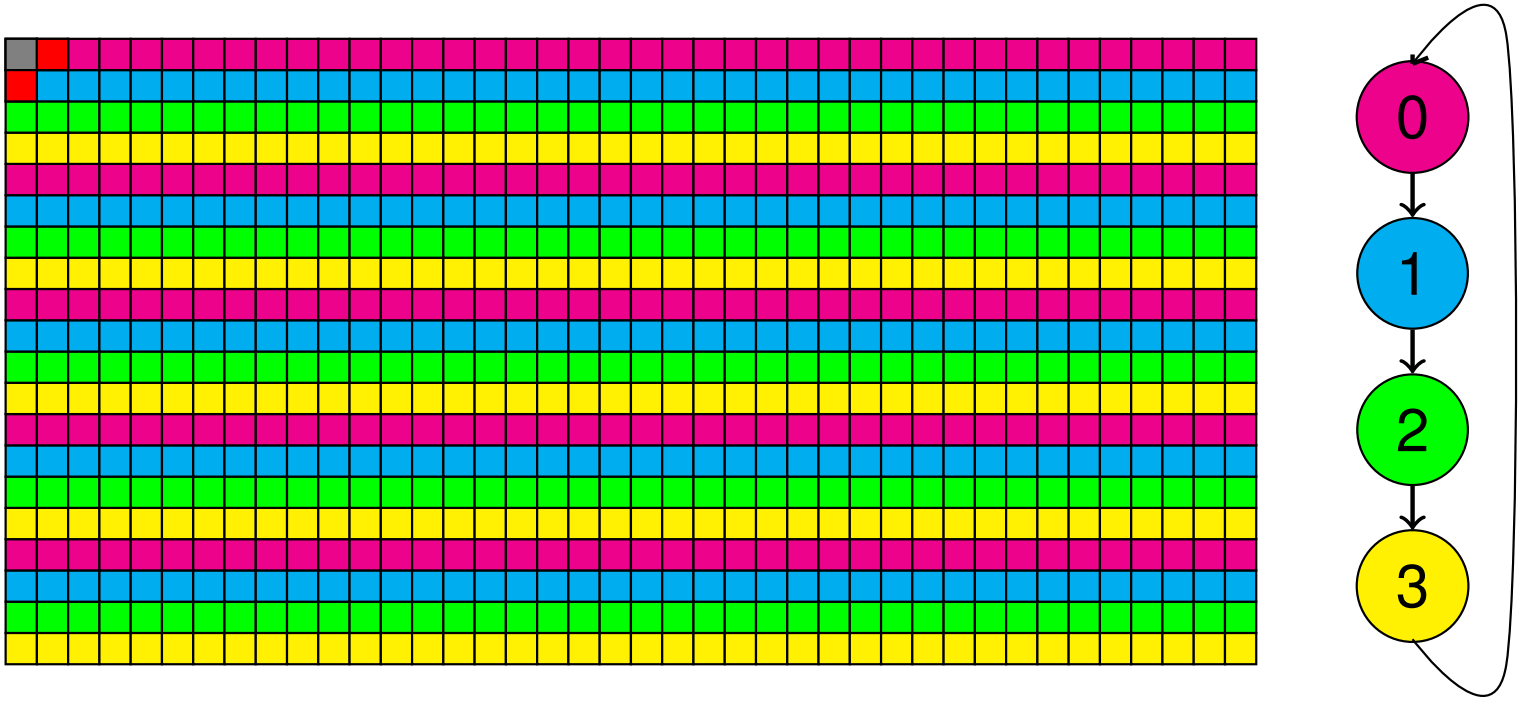
Cyclic Data Distribution



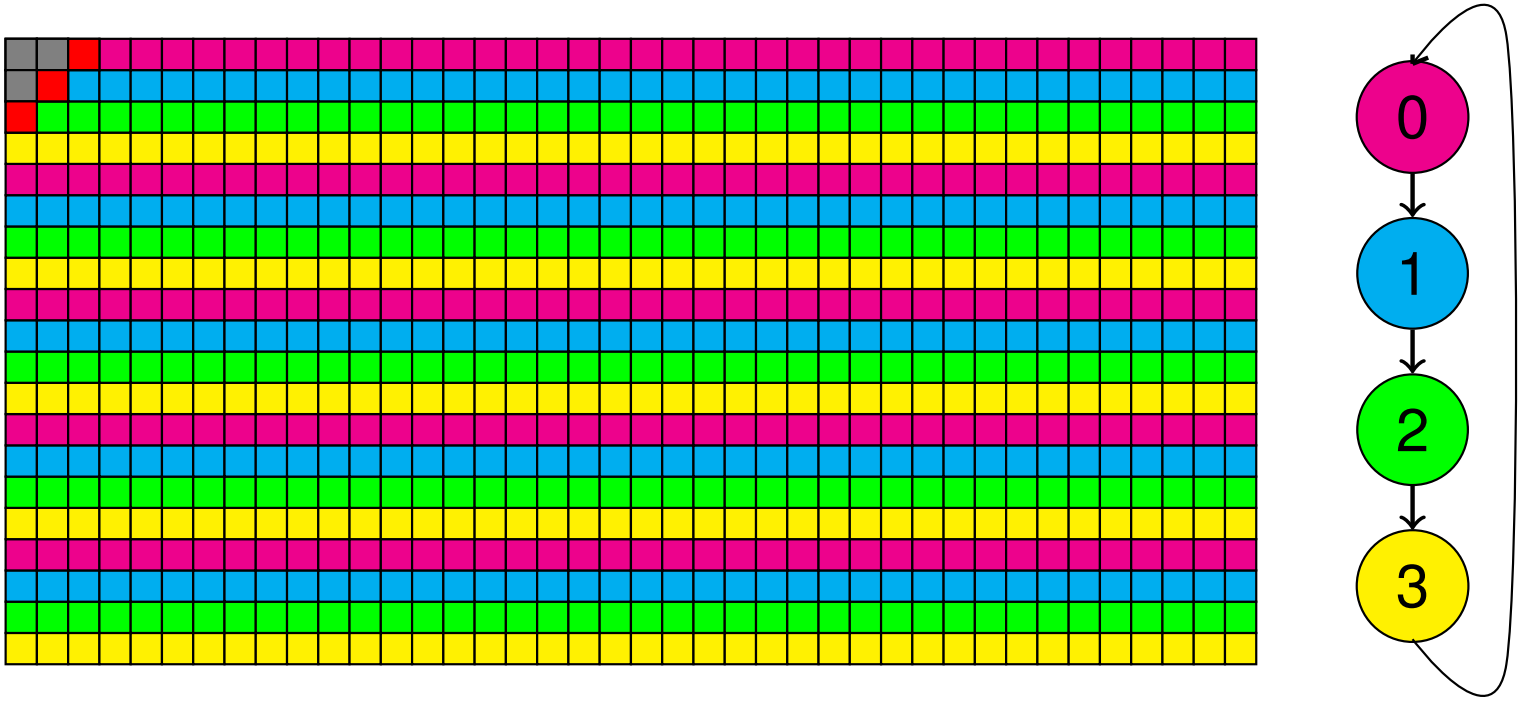
Cyclic Data Distribution



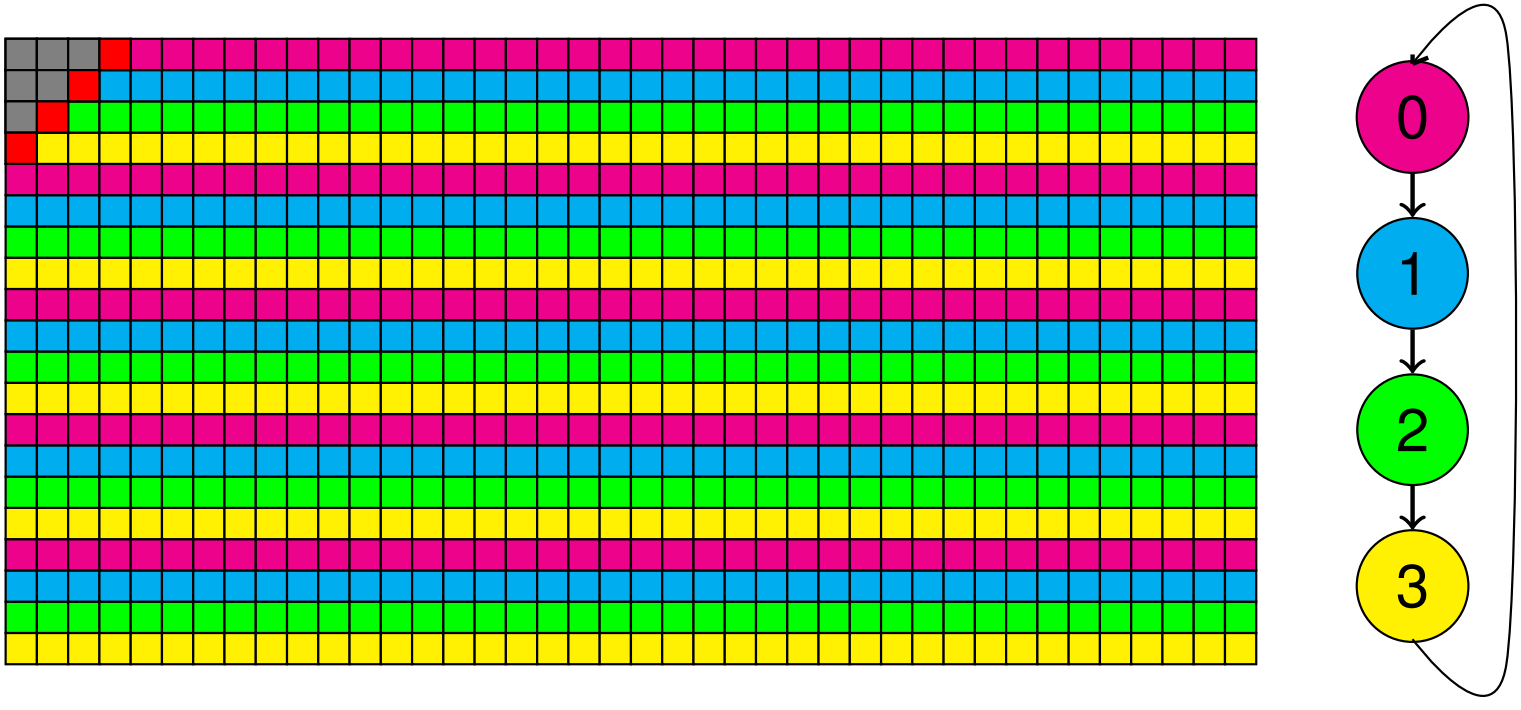
Cyclic Data Distribution



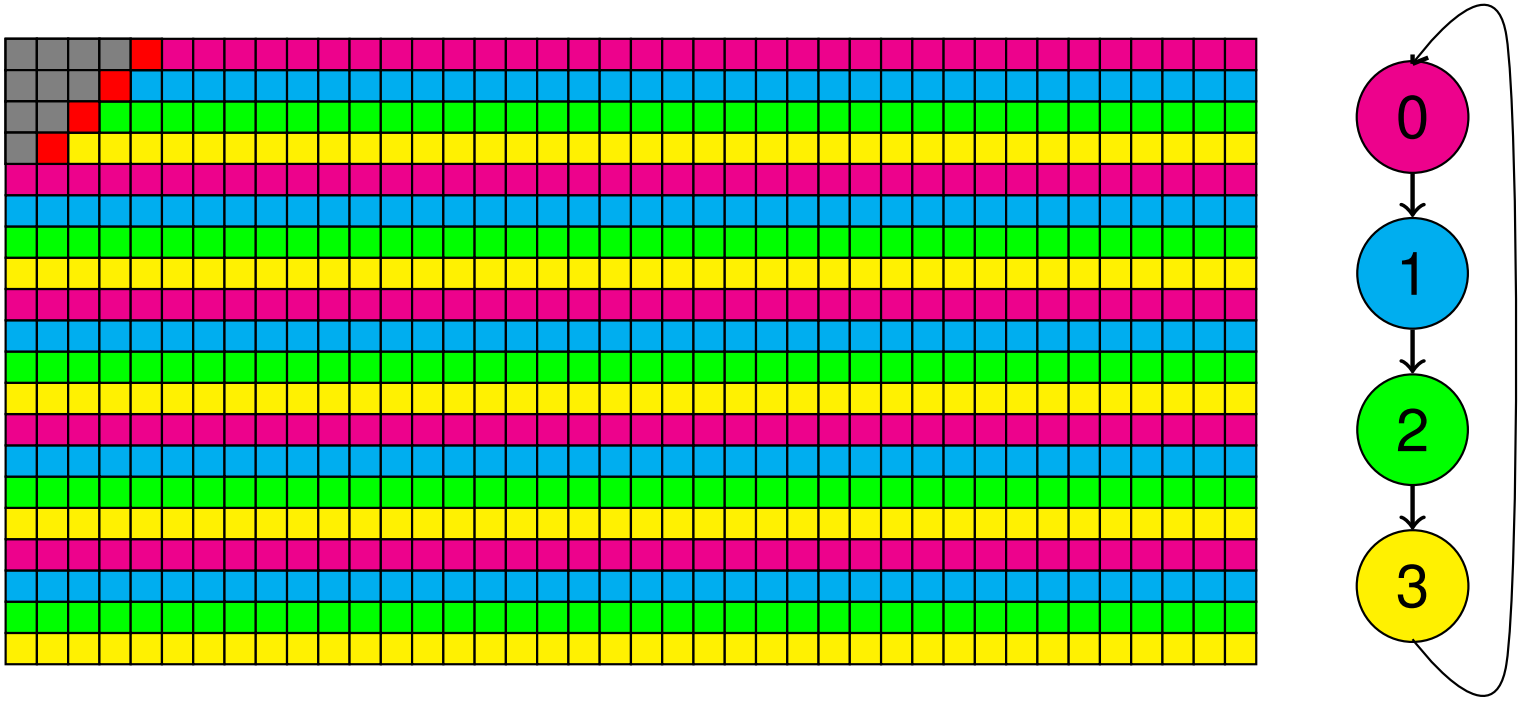
Cyclic Data Distribution



Cyclic Data Distribution



Cyclic Data Distribution



Parallel Speedup

- Let's again assume an infinitely fast network so that communicating cells takes zero time
- The last process begins computing at time: $(p-1) \times c$
- It then computes for time $(N/p) \times M \times c$ time units (one cell computed each time unit)
- It is the last one to finish computing, so the overall parallel execution time if $(p-1) \times c + (N/p) \times M \times c$
- The sequential execution time is $N \times M \times c$
- So the parallel speedup is: $pM/(p(p-1)/N + M)$
  - Was $pM/(p-1+M)$
- We have made the denominator smaller (because $N > p$). We've improved parallelism as much as possible

Cyclic Algorithm (one iteration)

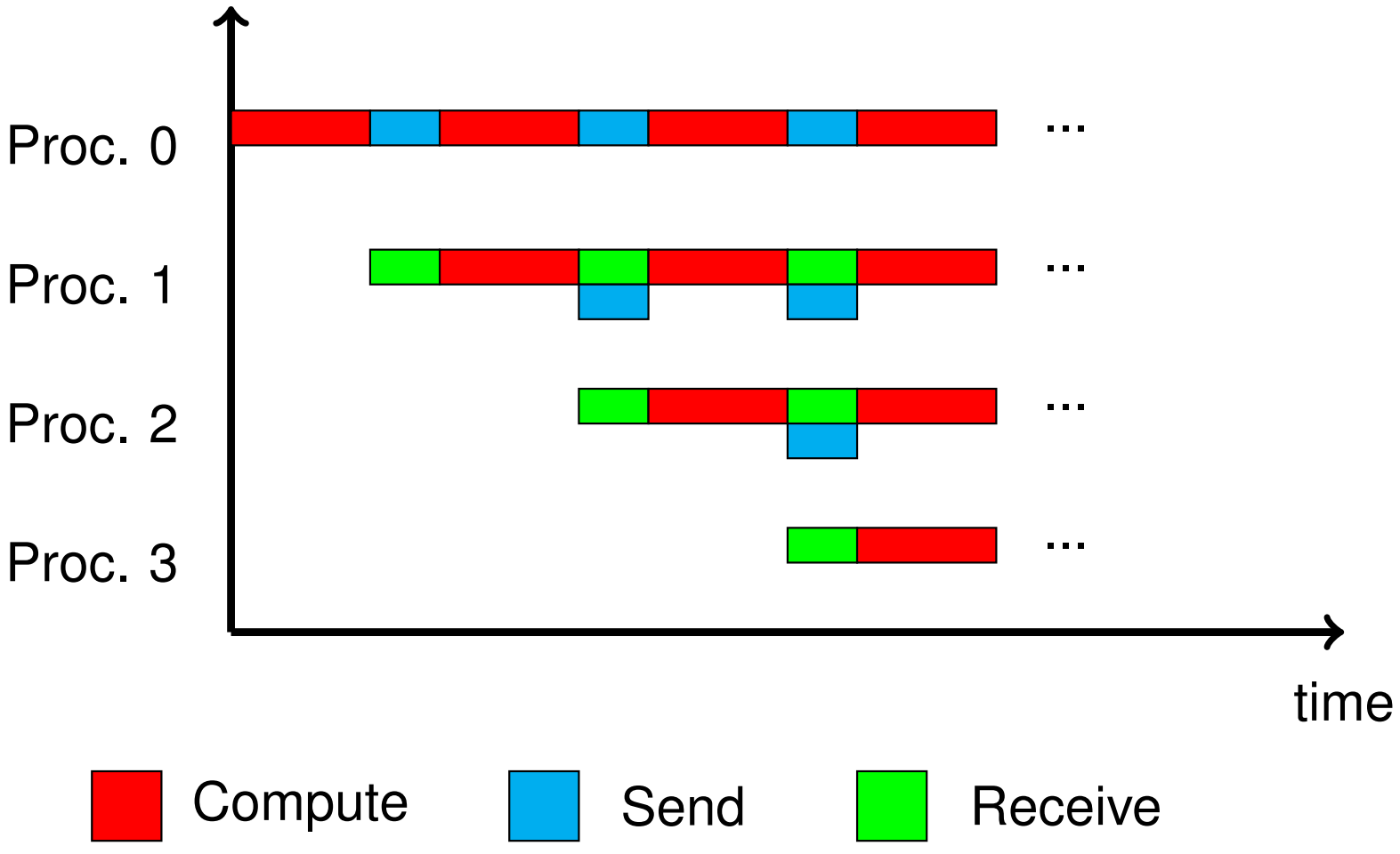
Stencil

```
p = num_procs();
rank = my_rank();
int A[N/p][M];
int cell_above;

for (i=0; i < N/p; i++) {
  for (j=0; j < M; j++) {
    if ((i > 0) && (rank > 0)) {
      // Receive my predecessor's last element in column j
      receive(&cell_above)
    }
    // Update my current cell
    update(i,j,cell_above)
    if ((i < N/p-1) && (rank < p-1)) {
      // Send my current cell to my successor
      send(A[i][j])
    }
  }
}
```

Network Overhead

- Network communication isn't zero-overhead
- Typical model of time to send x bytes: $\alpha + \beta x$
  - α: latency
  - β: inverse of the data rate
- Let s be the size of a cell value, in bytes
- The last process begins computing at time: $(p-1) \times (c + \alpha + \beta s)$
- It then computes for time $(N/p) \times M \times (c + \alpha + \beta s)$ time units (one cell computed and communicated each time unit)
- This assumes that a process can send and receive at the same time
- This is called the "two-port model"
- Let's see this on a Gantt chart..

The Two-Port Model



Parallel Speedup

- Parallel execution time:
  $(p-1) \times (c + \alpha + \beta s) + (N/p) \times M \times (c + \alpha + \beta s)$
- Sequential time: $N \times M \times c$ (no communication!)
- So the parallel speedup is the ratio of the two
- When $NM \to +\infty$, speedup $\to pc/(c + \alpha + \beta s)$
- This could be bad if communications are expensive
- If $c = \alpha + \beta s$, then speedup $\to p/2$ (50% parallel efficiency)
- In practice α could be huge compared to c
  - CPU Clock rate is high, network latencies can be high
- Our parallelism is great
- But our overhead is terrible!

Reducing Overhead

- Each time we send one message, we incur an α overhead!
- This is the typical “parallel application that sends tons of tiny messages" problems
- Idea: send groups of cell together
- Initially we sent a whole row, that was too many cells
- But sending one cell is too few
- So let’s send m cells, where we choose m
  - We assume m divides M, for simplicity
- Let’s look at the code...

Cyclic Algorithm, m cells (one iteration)

Stencil

```
p = num_procs();
rank = my_rank();
int A[N/p][M];
int cells_above[m];

for (i=0; i < N/p; i++) {
  for (j=0; j < M; j+=m) {
    if ((i > 0) && (rank != 0)) {
      // Receive my predecessor's last m cells
      receive(&cells_above,m)
    }
    // Update my current @m@ cells
    for (k=0; k < m; k++)
      update(i,j+k,cell_above)

    if ((i < N/p-1) && (rank != p-1)) {
      // Send my current m cells to my successor
      send(&(A[i][j]),m)
    }
  }
}
```

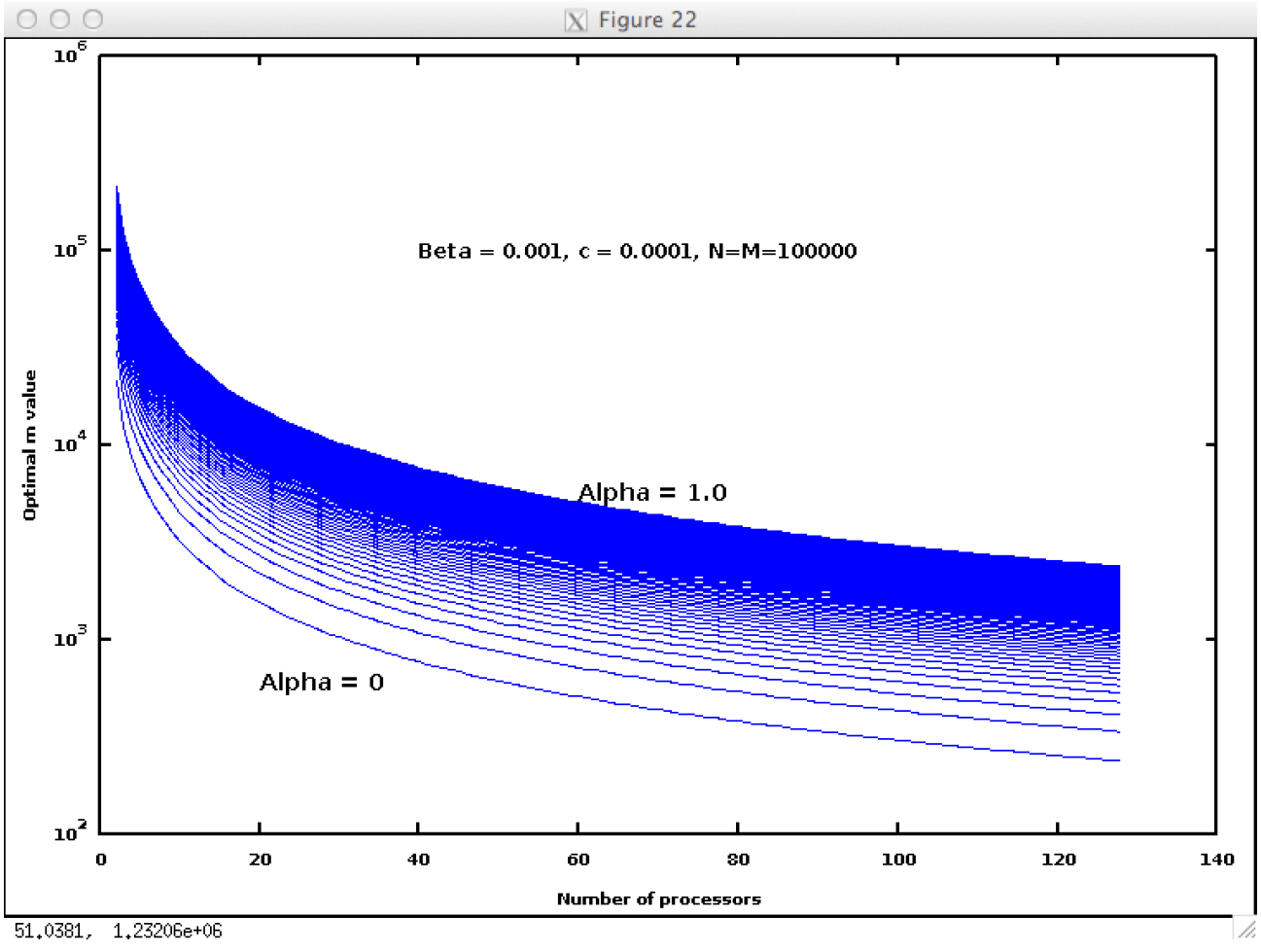
Parallel Speedup

- The last processor begins computing at time: $(p-1) \times (mc + \alpha + \beta m)$
- Then it computes for: $(NM/mp)(mc + \alpha + \beta m)$
- Parallel time is the sum of the two
- Sequential time: $N \times M \times c$
- Parallel speedup: the ratio of the two
  - For $m = 1$ we get our previous speedup
- When $NM \to +\infty$, speedup $\to pc/(c + \alpha/m + \beta)$
- Compared to before we've divided α my m
- We've decreased parallelism
- But we've also decreased overhead
- What's a good value of m?
- Let's find out...

Best m value

- Parallel time:
  $$T = (p-1) \times (mc + \alpha + \beta m) + (NM/mp)(mc + \alpha + \beta m)$$
  $$\frac{\partial T}{\partial m} = (p-1)(c+\beta) - \frac{NM\alpha}{pm^2}$$
  $$\frac{\partial T}{\partial m} = 0 \implies m = \sqrt{\frac{NM\alpha}{p(p-1)(c+\beta)}}$$
- We should select the divisor of M that's the closest to the above (real) value
- Let's plot the (not rounded off) optimal value above...

Best m vs. p and α



Stencil Application

- If we plug in the best m into the *asymptotic* parallel speedup we get:

  $$p \times \frac{c}{c+\sqrt{\frac{\alpha p(p-1)(c+\beta)}{NM}}+\beta}$$

- But this formula is for $NM \to +\infty$, so we get $p \times \frac{c}{c+\beta}$
- So we're not asymptotically optimal because of β
  - Makes sense: for each c you have to do a β
- And if p^2 is large or comparable to NM, then the speedup gets really poor
- In the end, this is just a difficult application to parallelize, and one shouldn't expect great parallel efficiency
  - Unless c is large, which could happen for a complicated stencil, but then that stencil may involve more neighbors...
- Side note: there is a yearly "HPC Stencil" conference, there are "stencil" research groups, etc.

Outline

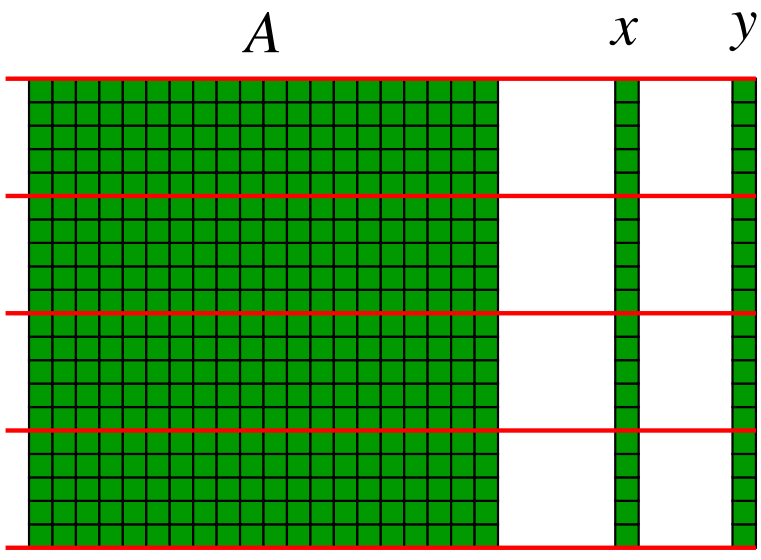
Matrix-Vector Multiplication

MatVec

```
int A[N][N];
int x[N];
int y[N];

// y = A * x

for (int i=0; i < N; i++) {
  int dotproduct = 0;
  for (int j=0; j < N; j++) {
    dotproduct += A[i][j] * x[j];
  }
  y[i] = dotproduct;
}
```

- Classic $O(N^2)$ algorithm
- Let's assume a 1-D data distribution...

Data Distribution

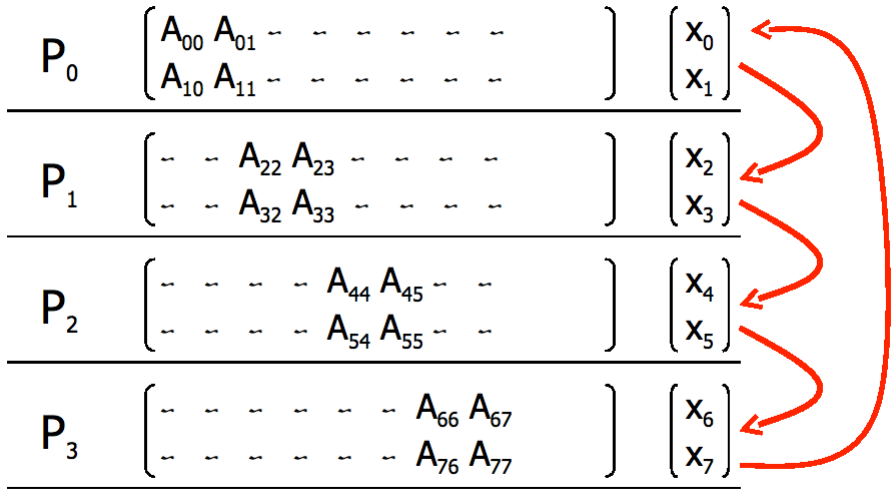


- Each processor holds a slice of A, a slice of x, and a (unitialized) slice of y.
- We go "fully distributed" (easier options would have x replicated, but that's really not standard at all)

Initial State ($p = 4, N = 8$)

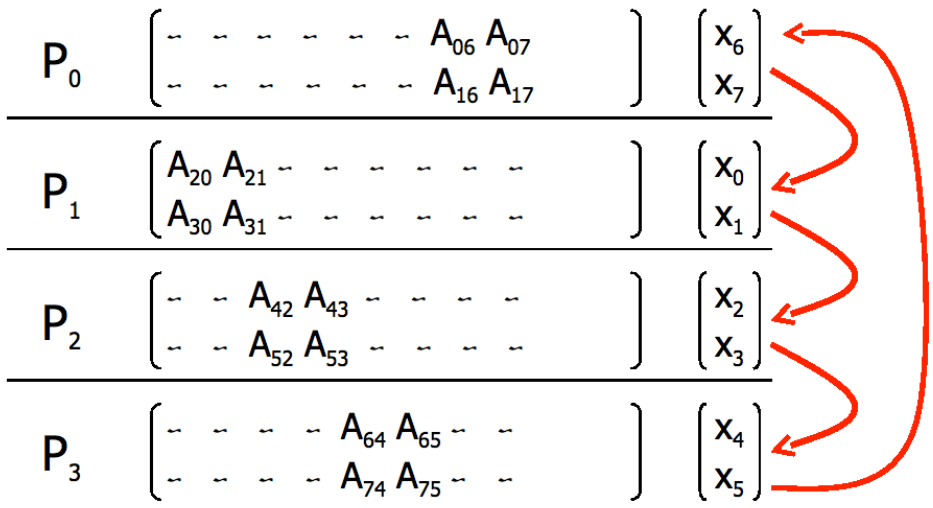
$$
\begin{array}{c|c|c}
P_0 & \begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} & A_{05} & A_{06} & A_{07} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} & A_{15} & A_{16} & A_{17} \end{pmatrix} & \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \\
\hline
P_1 & \begin{pmatrix} A_{20} & A_{21} & A_{22} & A_{23} & A_{24} & A_{25} & A_{26} & A_{27} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & A_{36} & A_{37} \end{pmatrix} & \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \\
\hline
P_2 & \begin{pmatrix} A_{40} & A_{41} & A_{42} & A_{43} & A_{44} & A_{45} & A_{46} & A_{47} \\ A_{50} & A_{51} & A_{52} & A_{53} & A_{54} & A_{55} & A_{56} & A_{57} \end{pmatrix} & \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \\
\hline
P_3 & \begin{pmatrix} A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} & A_{67} \\ A_{70} & A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{pmatrix} & \begin{pmatrix} x_6 \\ x_7 \end{pmatrix}
\end{array}
$$

Step 1: Compute + Send



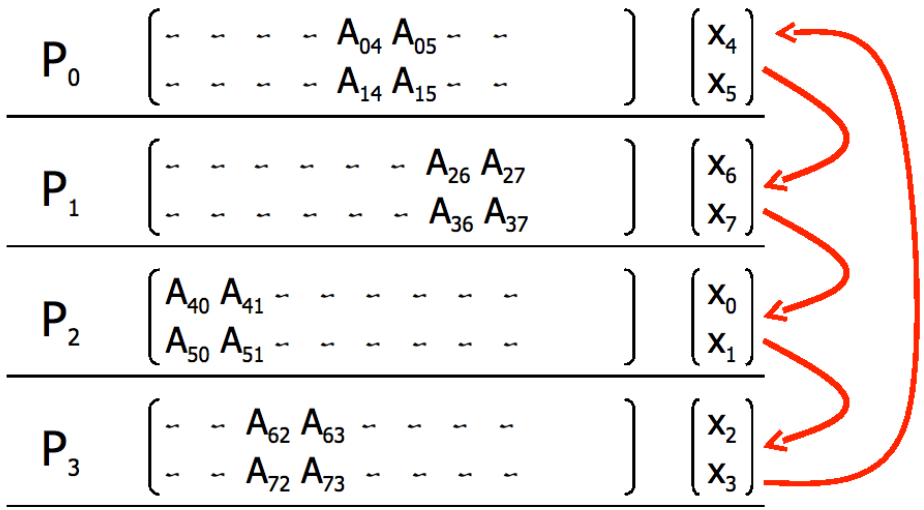
- Each process computes a tiny 2×2 matrix-vector multiplication
- e.g., y[0] += A[0][0] * x[0] + A[0][1] * x[2]
- Each process sends its chunk of x to its successor

Step 2: Compute + Send



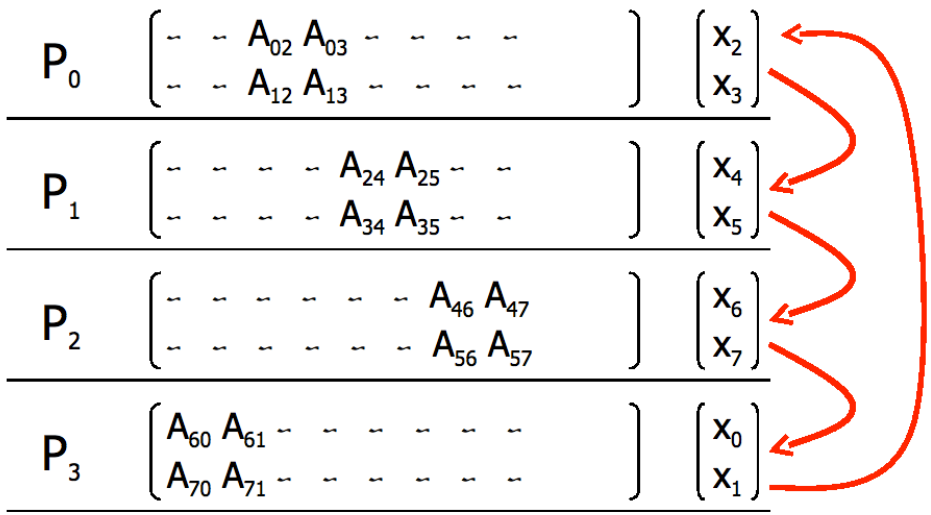
- e.g., y[0] += A[0][6] * x[6] + A[0][7] * x[7]

Step 3: Compute + Send



- e.g., y[0] += A[0][4] * x[4] + A[0][5] * x[5]

Step 4: Compute + Send



- e.g., y[0] += A[0][2] * x[2] + A[0][3] * x[3]

Step 5: Send

$$
\begin{array}{l|l|l}
P_0 & \begin{pmatrix} A_{00} & A_{01} & - & - & - & - & - & - \\ A_{10} & A_{11} & - & - & - & - & - & - \end{pmatrix} & \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \\ \hline
P_1 & \begin{pmatrix} - & - & A_{22} & A_{23} & - & - & - & - \\ - & - & A_{32} & A_{33} & - & - & - & - \end{pmatrix} & \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \\ \hline
P_2 & \begin{pmatrix} - & - & - & - & A_{44} & A_{45} & - & - \\ - & - & - & - & A_{54} & A_{55} & - & - \end{pmatrix} & \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \\ \hline
P_3 & \begin{pmatrix} - & - & - & - & - & - & A_{66} & A_{67} \\ - & - & - & - & - & - & A_{76} & A_{77} \end{pmatrix} & \begin{pmatrix} x_6 \\ x_7 \end{pmatrix}
\end{array}
$$

- Do one last communication to get x back to its initial distribution

Matrix-Vector Multiplication

Stencil

```
rank = my_rank(); // rank
p = num_procs();  // # processors

int A[N/p][N], x[N/p], y[N/p];    // array slices
int buffer[N/p];      //receive buffer

int *tempR = buffer; // receive buffer
int *tempS = x;          // pointer to my slice of x

for (step=0; step<p; step++) {   // p steps
  send(tempS,N/p);
  recv(tempR,N/p);

  for (i=0; i<N/p; i++)
    for (j=0; j <N/p; j++)
      y[i] = y[i] + a[i,(rank - step mod p) * N/p + j] * tempS[j]

  // Swap pointers
  tmp = tempS; tempS = tempR; tmpR = tmp;
}
```

Performance Analysis

- Each processor goes through p steps
- Using the same notations as before (α, β, c)
- Any thoughts about the performance analysis?

Performance Analysis

- Each processor goes through p steps
- Using the same notations as before (α, β, c)
- Each steps involves:
  - Send N/p elements: $\alpha + \beta N/p$ (assuming 1-byte elements)
  - Recv N/p elements: $\alpha + \beta N/p$ (assuming 1-byte elements)
  - Compute: $(N/p)^2 c$ (c: time to perform one update)
- Parallel time: $p((N/p)^2 c + 2\alpha + 2\beta N/p)$
- Sequential time: $N^2 c$
- Speedup: p fixed, $N \to +\infty \implies$ speedup $\to$ p
- The algorithm is asymptotically optimal!

Overlapping communication/computation

MatVec

```
rank = my_rank(); // rank
p = num_procs();  // # processors

int A[N/p][N], x[N/p], y[N/p];    // array slices
int buffer[N/p];      //receive buffer

int *tempR = buffer; // receive buffer
int *tempS = x;          // pointer to my slice of x

for (step=0; step<p; step++) {   // p steps
  isend(tempS,N/p);
  irecv(tempR,N/p);

  for (i=0; i<N/p; i++)
    for (j=0; j <N/p; j++)
      y[i] = y[i] + a[i,(rank - step mod p) * N/p + j] * tempS[j]

  // Swap pointers
  tmp = tempS; tempS = tempR; tmpR = tmp;
}
```

Performance Analysis

- Each processor goes through p steps
  - Send N/p elements: $\alpha + \beta N/p$ (assuming 1-byte elements)
  - Recv N/p elements: $\alpha + \beta N/p$ (assuming 1-byte elements)
  - Compute: $(N/p)^2 c$ (c: time to perform one update)
- Parallel time: $p \times \max\left((N/p)^2 c, \alpha + \beta N/p\right)$
- The algorithm is still asymptotically optimal, but will be better in practice for smaller values of N

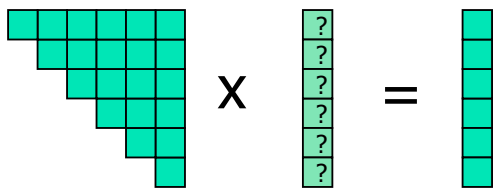
Matrix Multiplication

- It turns out we can use the exact same ideas for a 1-D Matrix multiplication!
- Just give “horizontal" slices of the matrices to each processor
- Perform local matrix-matrix multiplication
- We’ll leave this as a project/exercise...

Outline

Solving Linear Systems of Eq.

- Method for solving Linear Systems
  - The need to solve linear systems arises in an estimated 75% of all scientific computing problems [Dahlquist 1974]
- Gaussian Elimination is perhaps the most well-known method
  - based on the fact that the solution of a linear system is invariant under scaling and under row additions
    - One can multiply a row of the matrix by a constant as long as one multiplies the corresponding element of the right-hand side by the same constant
    - One can add a row of the matrix to another one as long as one adds the corresponding elements of the right-hand side
  - Idea: scale and add equations so as to transform matrix A in an upper triangular matrix:



equation n-i has i unknowns, with

Gaussian Elimination

1	1	1		0
1	-2	2	x =	4
1	2	-1		2

Subtract row 1 from rows 2 and 3

1	1	1		0
0	-3	1	x =	4
0	1	-2		2

Multiple row 3 by 3 and add row 2

1	1	1		0
0	-3	1	x =	4
0	0	-5		10

Solving equations in reverse order (backsolving)

$-5x_3 = 10$

$-3x_2 + x_3 = 4$

$x_1 + x_2 + x_3 = 0$

$= 4$

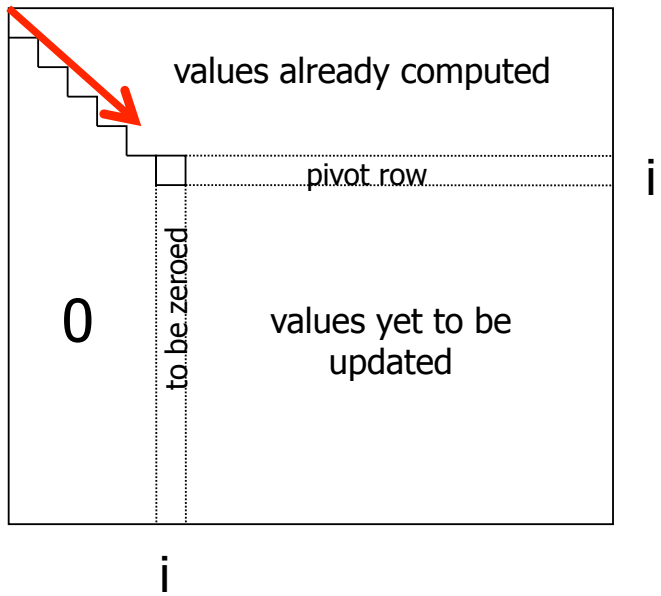
$x_3 = -2$

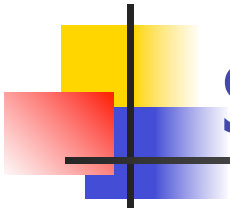
$x_2 = -2$

x_1

Gaussian Elimination

- The algorithm goes through the matrix from the top-left corner to the bottom-right corner
- the ith step eliminates non-zero sub-diagonal elements in column i, substracting the ith row scaled by a_{ji}/a_{ii} from row j, for j=i+1,..,n.





Sequential Gaussian Elimination

Simple sequential algorithm

```
// for each column i
// zero it out below the diagonal by adding
// multiples of row i to later rows
for i = 1 to n-1
    // for each row j below row i
    for j = i+1 to n
         // add a multiple of row i to row j
         for k = i to n
              A(j,k) = A(j,k) - (A(j,i)/A(i,i)) * A(i,k)
```

- Several "tricks" that do not change the spirit of the algorithm but make implementation easier and/or more efficient
  - Right-hand side is typically kept in column n+1 of the matrix and one speaks of an augmented matrix
  - Compute the A(i,j)/A(i,i) term outside of the loop

Pivoting: Motivation

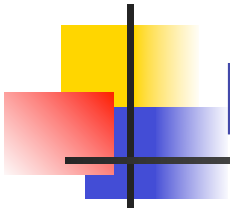
- A few pathological cases

0	1
1	1

- Division by small numbers → round-off error in computer arithmetic
- Consider the following system

  $0.0001x_1 + x_2 = 1.000$

  $x_1 + x_2 = 2.000$
- exact solution: $x_1=1.00010$ and $x2 = 0.99990$
- say we round off after 3 digits after the decimal point
- Multiply the first equation by 10^4 and subtract it from the second equation
- $(1 - 1)x_1 + (1 - 10^4)x_2 = 2 - 10^4$



Partial Pivoting

- One can just swap rows

  $x_1 + x_2 = 2.000$

  $0.0001x_1 + x_2 = 1.000$

- Multiple the first equation my 0.0001 and subtract it from the second equation gives:

  $(1 - 0.0001)x2 = 1 - 0.0001$

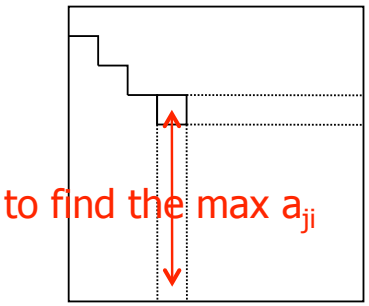
  $0.9999\ x_2 = 0.9999 \Rightarrow x_2 = 1$

  and then $x_1 = 1$

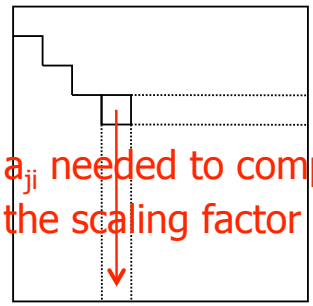
- Final solution is closer to the real solution. (Magical?)
- Partial Pivoting
  - For numerical stability, one doesn't go in order, but pick the next row in rows i to n that has the largest element in row i
  - This row is swapped with row i (along with elements of the right hand side) before the subtractions
    - the swap is not done in memory but rather one keeps an indirection array
- Total Pivoting
  - Look for the greatest element ANYWHERE in the matrix
  - Swap columns
  - Swap rows

Parallel Gaussian Elimination?

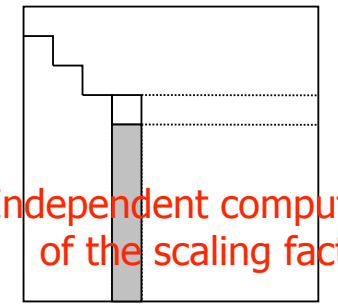
- Assume that we have one processor per matrix element



Reduction

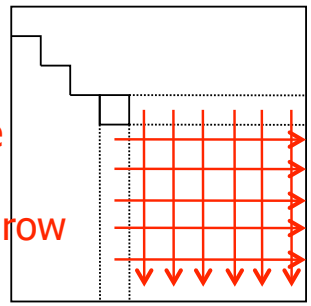


Broadcast

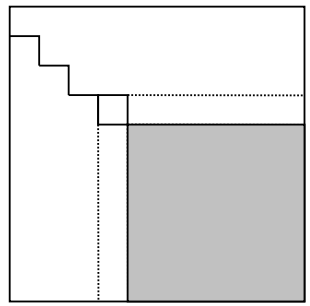


Compute

Every update needs the scaling factor and the element from the pivot row



Broadcasts



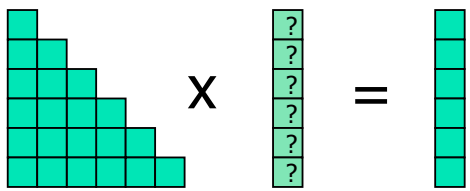
Compute

Independent computations

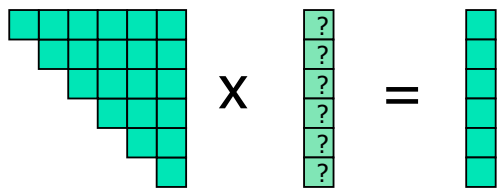
LU Factorization (Section 4.4)

- Gaussian Elimination is simple but
  - What if we have to solve many Ax = b systems for different values of b?
    - This happens a LOT in real applications
- Another method is the "LU Factorization"
- Ax = b
- Say we could rewrite A = L U, where L is a lower triangular matrix, and U is an upper triangular matrix $O(n^3)$
- Then Ax = b is written L U x = b
- Solve L y = b $O(n^2)$
- Solve U x = y $O(n^2)$

triangular system solves are easy



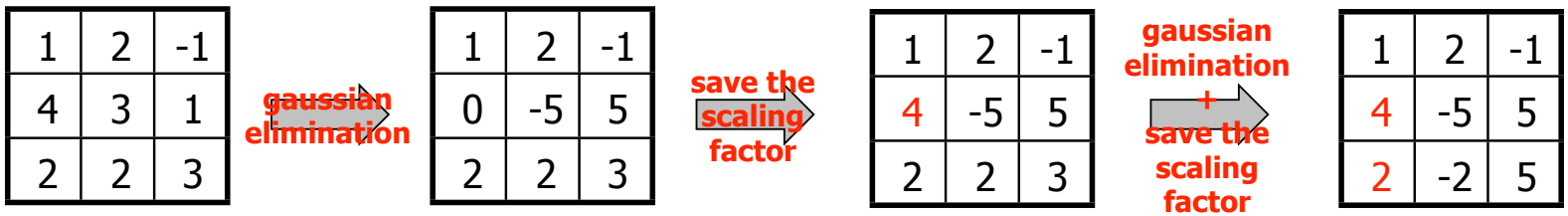
equation i has i unknowns

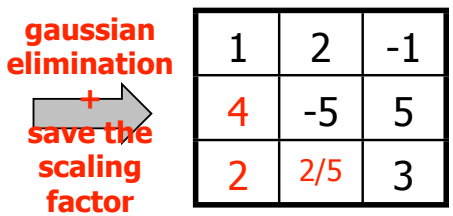


equation n-i has i unknowns

LU Factorization: Principle

- It works just like the Gaussian Elimination, but instead of zeroing out elements, one "saves" scaling coefficients.





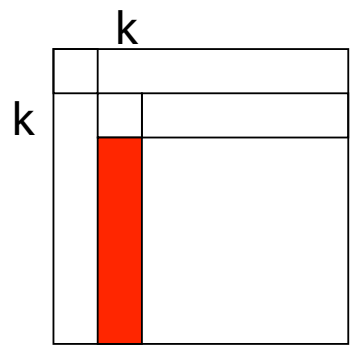
$$L = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 2 & 2/5 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 1 & 2 & -1 \\ 0 & -5 & 5 \\ 0 & 0 & 3 \end{bmatrix}$$

LU Factorization

- We're going to look at the simplest possible version
  - No pivoting:just creates a bunch of indirections that are easy but make the code look complicated without changing the overall principle

```
LU-sequential(A,n) {
  for k = 0 to n-2 {
     // preparing column k
     for i = k+1 to n-1
        a_ik ← -a_ik / a_kk
     for j = k+1 to n-1
        // Task T_kj: update of column j
        for i=k+1 to n-1
           a_ij ← a_ij + a_ik * a_kj
  }
}
```

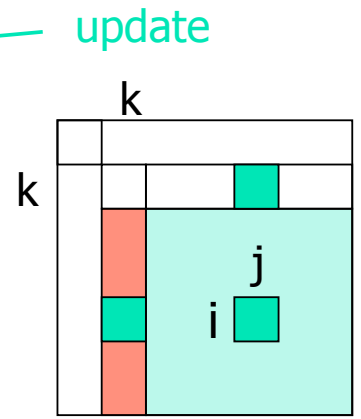
stores the scaling factors

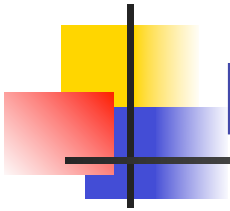


LU Factorization

- We're going to look at the simplest possible version
  - No pivoting:just creates a bunch of indirections that are easy but make the code look complicated without changing the overall principle

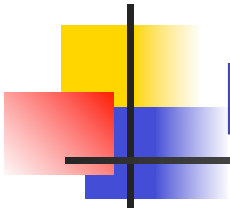
```
LU-sequential(A,n) {
  for k = 0 to n-2 {
     // preparing column k
     for i = k+1 to n-1
        a_ik ← -a_ik / a_kk
     for j = k+1 to n-1
        // Task T_kj: update of column j
        for i=k+1 to n-1
           a_ij ← a_ij + a_ik * a_kj
  }
}
```





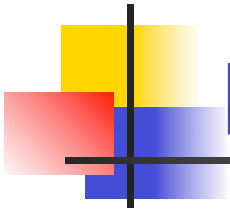
Parallel LU on a ring

- Since the algorithm operates by columns from left to right, we should distribute columns to processors
- Principle of the algorithm
  - At each step, the processor that owns column k does the "prepare" task and then broadcasts the bottom part of column k to all others
    - Annoying if the matrix is stored in row-major fashion
    - Remember that one is free to store the matrix in anyway one wants, as long as it's coherent and that the right output is generated
  - After the broadcast, the other processors can then update their data.
- Assume there is a function alloc(k) that returns the rank of the processor that owns column k
  - Basically so that we don't clutter our program with too many global-to-local index translations
- In fact, we will first write everything in terms of global indices, as to avoid all annoying index arithmetic



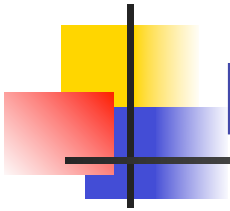
LU-broadcast algorithm

```
LU-broadcast(A,n) {
  q ← MY_NUM()
  p ← NUM_PROCS()
  for k = 0 to n-2 {
    if (alloc(k) == q)
       // preparing column k
       for i = k+1 to n-1
          buffer[i-k-1] ← a_ik ← -a_ik / a_kk
    broadcast(alloc(k),buffer,n-k-1)
    for j = k+1 to n-1
      if (alloc(j) == q)
         // update of column j
         for i=k+1 to n-1
            a_ij ← a_ij + buffer[i-k-1] * a_kj
  }
}
```



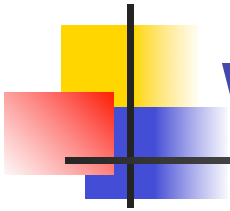
Dealing with local indices

- Assume that p divides n
- Each processor needs to store r=n/p columns and its local indices go from 0 to r-1
- After step k, only columns with indices greater than k will be used
- Simple idea: use a local index, l, that everyone initializes to 0
- At step k, processor alloc(k) increases its local index so that next time it will point to its next local column



LU-broadcast algorithm

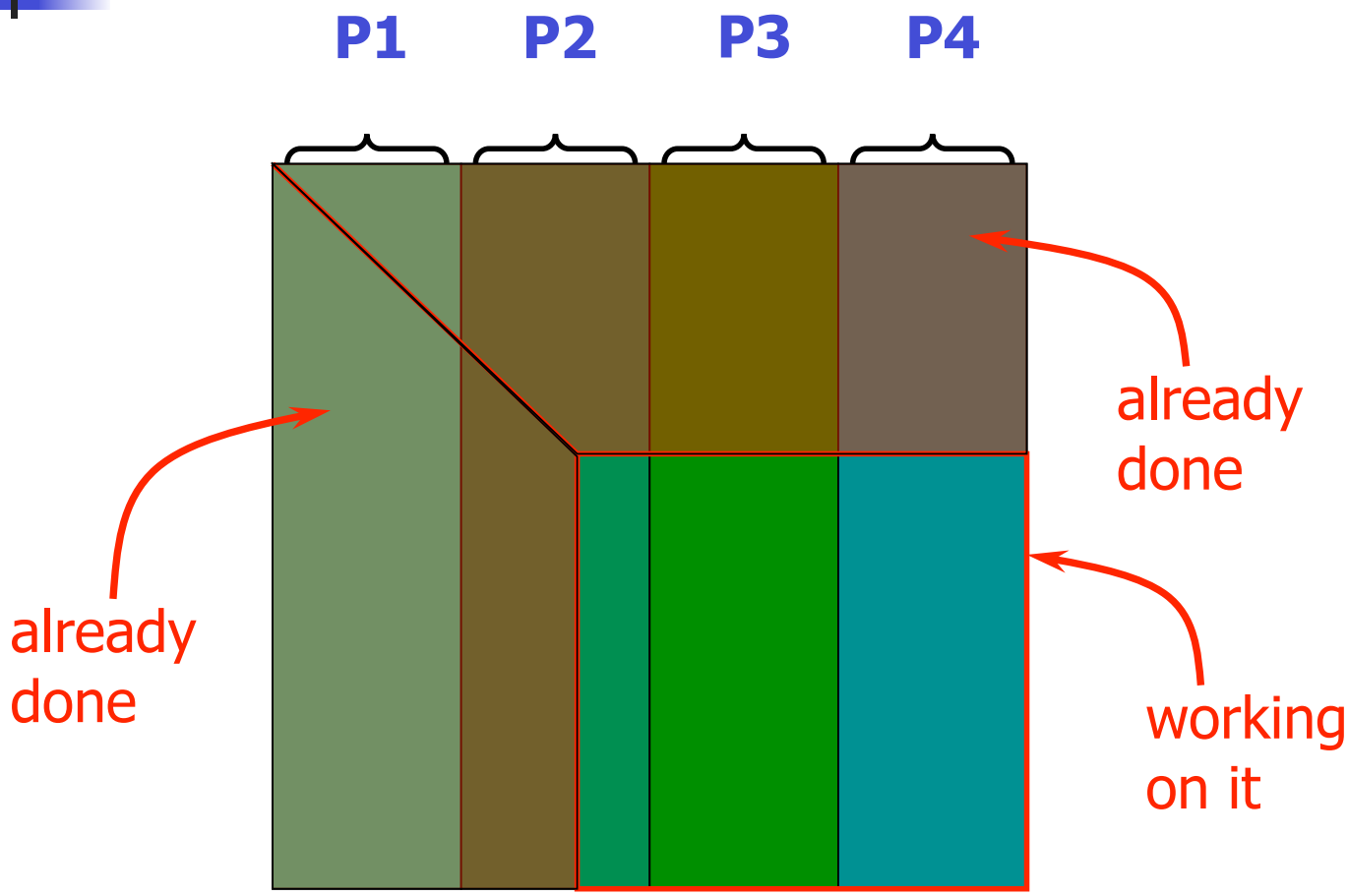
```
...
  double a[n-1][r-1];


  q ← MY_NUM()
  p ← NUM_PROCS()
  l ← 0
  for k = 0 to n-2 {
    if (alloc(k) == q)
        for i = k+1 to n-1
          buffer[i-k-1] ← a[i,k] ← -a[i,l] / a[k,l]
        l ← l+1
    broadcast(alloc(k),buffer,n-k-1)
    for j = l to r-1
        for i=k+1 to n-1
          a[i,j] ← a[i,j] + buffer[i-k-1] * a[k,j]
  }
```



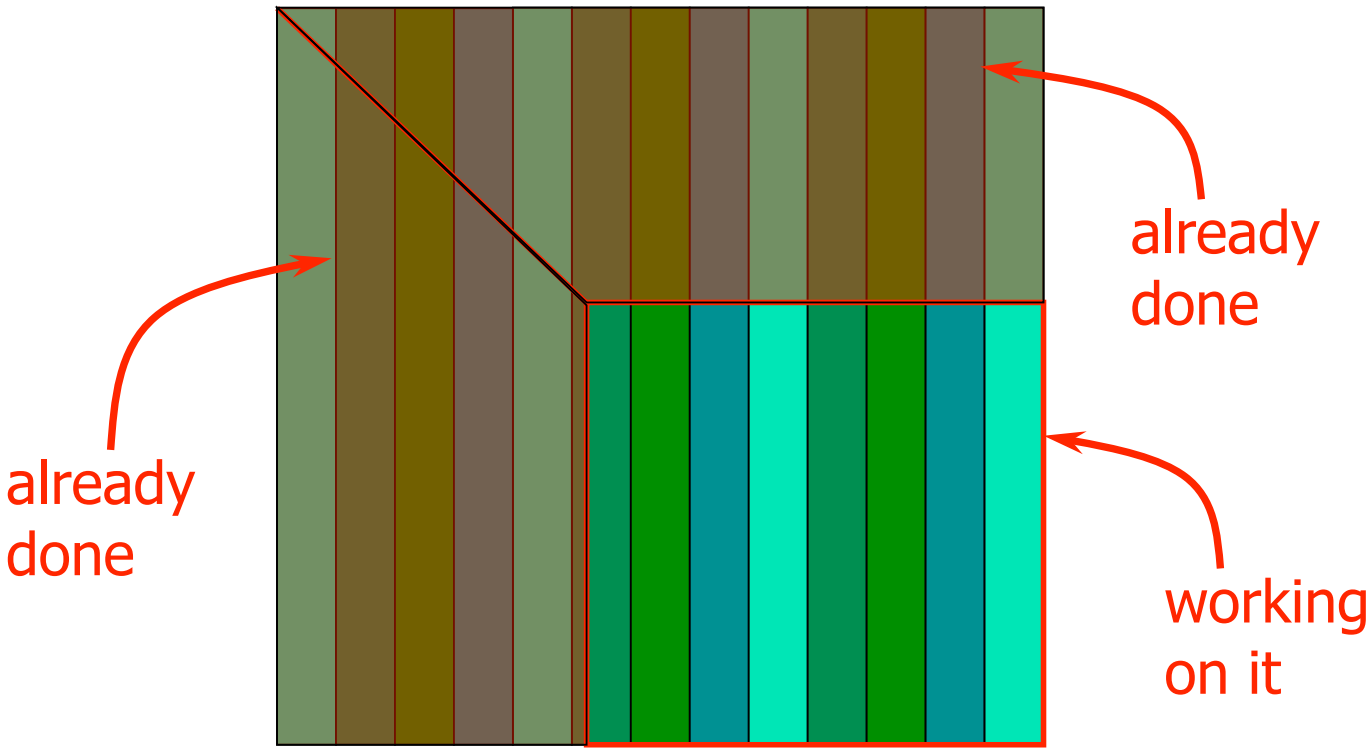
What about the Alloc function?

- One thing we have left completely unspecified is how to write the alloc function: how are columns distributed among processors
- There are two complications:
  - The amount of data to process varies throughout the algorithm's execution
    - At step k, columns k+1 to n-1 are updated
    - Fewer and fewer columns to update
  - The amount of computation varies among columns
    - e.g., column n-1 is updated more often than column 2
    - Holding columns on the right of the matrix leads to much more work
- There is a strong need for load balancing
  - All processes should do the same amount of work

Bad load balancing



Good Load Balancing?



Cyclic distribution

Proof that load balancing is good

- The computation consists of two types of operations
  - column preparations
  - matrix element updates
- There are many more updates than preparations, so we really care about good balancing of the preparations
- Consider column j
- Let's count the number of updates performed by the processor holding column j
- Column j is updated at steps k=0, ..., j-1
- At step k, elements i=k+1, ..., n-1 are updates
  - indices start at 0
- Therefore, at step k, the update of column j entails n-k-1 updates
- The tota$\sum_{k=0}^{j-1}(n-k-1) = j(n-1) - \frac{j(j-1)}{2}$tion is:

Proof that load balancing is good

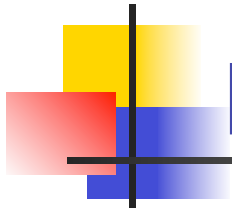
- Consider processor P_i, which holds columns lp+i for l=0, ... , n/p -1
- Processor P_i needs to perform this many updates:

$$\sum_{l=0}^{n/p-1} ((lp+i)(n-1) - \frac{(lp+i)(lp+i-1)}{2})$$

- Turns out this can be computed
  - separate terms
  - use formulas for sums of integers and sums of squares
- What it all boils down to is:

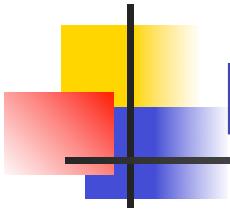
$$\frac{n^3}{3p} + O(n^2)$$

- This does not depend on i !!
- Therefore it is (asymptotically) the same for all P_i processors
- Therefore we have (asymptotically) perfect load balancing!



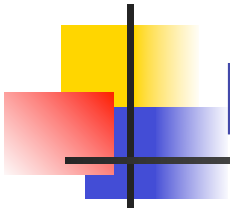
Load-balanced program

```
...
  double a[n-1][r-1];


  q ← MY_NUM()
  p ← NUM_PROCS()
  l ← 0
  for k = 0 to n-2 {
    if (k mod p == q)
        for i = k+1 to n-1
          buffer[i-k-1] ← a[i,k] ← -a[i,l] / a[k,l]
        l ← l+1
    broadcast(alloc(k),buffer,n-k-1)
    for j = l to r-1
        for i=k+1 to n-1
          a[i,j] ← a[i,j] + buffer[i-k-1] * a[k,j]
  }
```



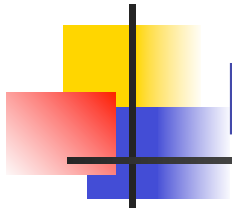
Performance Analysis

- How long does this code take to run?
- This is not an easy question because there are many tasks and many communications
- A little bit of analysis shows that the execution time is the sum of three terms
  - n-1 communications: $n\,L + (n^2/2)\,b + O(1)$
  - n-1 column preparations: $(n^2/2)\,w' + O(1)$
  - column updates: $(n^3/3p)\,w + O(n^2)$
- Therefore, the execution time is $\sim (n^3/3p)\,w$
- Note that the sequential time is: $(n^3/3)\,w$
- Therefore, we have perfect asymptotic efficiency!
- This is good, but isn't always the best in practice
- How can we improve this algorithm?



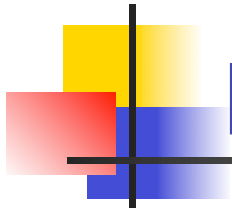
Pipelining on the Ring

- So far, the algorithm we've used a simple broadcast
- Nothing was specific to being on a ring of processors and it's portable
  - in fact you could just write raw MPI that just looks like our pseudo-code and have a very limited, inefficient for small n, LU factorization that works only for some number of processors
- But it's not efficient
  - The n-1 communication steps are not overlapped with computations
  - Therefore Amdahl's law, etc.
- Turns out that on a ring, with a cyclic distribution of the columns, one can interleave pieces of the broadcast with the computation
  - It almost looks like inserting the source code from the broadcast code we saw at the very beginning throughout the LU code



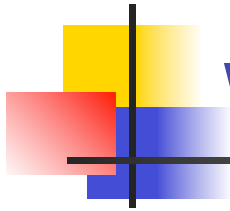
Previous program

```
...
  double a[n-1][r-1];


  q ← MY_NUM()
  p ← NUM_PROCS()
  l ← 0
  for k = 0 to n-2 {
    if (k == q  mod p)
        for i = k+1 to n-1
          buffer[i-k-1] ← a[i,k] ← -a[i,l] / a[k,l]
        l ← l+1
    broadcast(alloc(k),buffer,n-k-1)
    for j = l to r-1
        for i=k+1 to n-1
          a[i,j] ← a[i,j] + buffer[i-k-1] * a[k,j]
  }
```



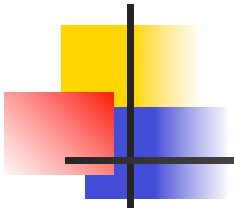
LU-pipeline algorithm

```
double a[n-1][r-1];


q ← MY_NUM()
p ← NUM_PROCS()
l ← 0
for k = 0 to n-2 {
  if (k == q mod p)
     for i = k+1 to n-1
        buffer[i-k-1] ← a[i,k] ← -a[i,l] / a[k,l]
     l ← l+1
     send(buffer,n-k-1)
  else
     recv(buffer,n-k-1)
     if (q ≠ k-1 mod p) send(buffer, n-k-1)
  for j = l to r-1
     for i=k+1 to n-1
        a[i,j] ← a[i,j] + buffer[i-k-1] * a[k,j]
```



Why is it better?

- During a broadcast the root's successor just sits idle while the message goes along the ring
- This is because of the way we have implemented broadcast, partially
  - With a better broadcast on a general topology the wait may be smaller
  - But there is still a wait
- What we have done is allow each processor to move on to other business after receiving and forwarding the message
- Possible by writing the code with just sends and receive
  - More complicated, more efficient: usual trade-off
- Let's look at a (idealized) time-line

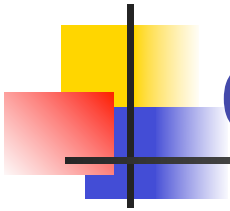


A processor sends out data as soon as it receives it

Some communication occurs in parallel with computation

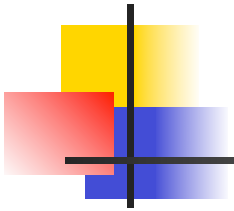
First four stages

Prep(0)	Recv(0)	Recv(0)	Recv(0)
Send(0)	Send(0)	Send(0)	Update(0,3)
Update(0,4)	Update(0,1)	Update(0,2)	Update(0,7)
Update(0,8)	Update(0,5)	Update(0,6)	Update(0,11)
Update(0,12)	Update(0,9)	Update(0,10)	Update(0,15)
	Update(0,13)	Update(0,14)	
	Prep(1)	Recv(1)	
	Send(1)	Send(1)	Recv(1)
	Update(1,5)	Update(1,2)	Send(1)
Recv(1)	Update(1,9)	Update(1,6)	Update(1,3)
Update(1,4)	Update(1,13)	Update(1,10)	Update(1,7)
Update(1,8)		Update(1,14)	Update(1,11)
Update(1,12)		Prep(2)	Update(1,15)
		Send(2)	Recv(2)
Recv(2)		Update(2,6)	Send(2)
Send(2)	Recv(2)	Update(2,10)	Update(2,3)
Update(2,4)	Update(2,5)	Update(2,14)	Update(2,7)
Update(2,8)	Update(2,9)		Update(2,11)
Update(2,12)	Update(2,13)		Update(2,15)
			Prep(3)
Recv(3)			Send(3)
Send(3)	Recv(3)		Update(3,7)
Update(3,4)	Send(3)	Recv(3)	Update(3,11)
Update(3,8)	Update(3,5)	Update(3,6)	Update(3,15)
Update(3,12)	Update(3,9)	Update(3,10)	
	Update(3,13)	Update(3,14)	



Can we do better?

- In the previous algorithm, a processor does all its updates before doing a Prep() computation that then leads to a communication
- But in fact, some of these updates can be done later
- Idea: Send out pivot as soon as possible
- Example:
  - In the previous algorithm
    - P1: Receive(0), Send(0)
    - P1: Update(0,1), Update(0,5), Update(0,9), Update(0,13)
    - P1: Prep(1)
    - P1: Send(1)
    - …
  - In the new algorithm (see page 130)
    - P1: Receive(0), Send(0)
    - P1: Update(0,1)
    - P1: Prep(1)
    - P1: Send(1)
    - P1: Update(0,5), Update(0,9), Update(0,13)
    - …



A processor sends out data as soon as it receives it

Many communications occur in parallel with computation

First four stages

Prep(0)	Recv(0)	Recv(0)	Recv(0)
Send(0)	Send(0)	Send(0)	Update(0,3)
Update(0,4)	Update(0,1)	Update(0,2)	Update(0,7)
Update(0,8)	Prep(1)	Recv(1)	Recv(1)
Update(0,12)	Send(1)	Send(1)	Send(1)
	Update(0,5)	Update(1,2)	Update(1,3)
Recv(1)	Update(0,9)	Prep(2)	Recv(2)
Update(1,4)	Update(0,13)	Send(2)	Send(2)
Update(1,8)	Update(1,5)	Update(0,6)	Update(2,3)
Recv(2)	Update(1,9)	Update(0,10)	Prep(3)
Send(2)	Recv(2)	Update(0,14)	Send(3)
Update(1,12)	Update(1,13)	Update(1,6)	Update(0,11)
Recv(3)	Update(2,5)	Update(1,10)	Update(0,15)
Send(3)	Recv(3)	Recv(3)	Update(1,7)
Update(2,4)	Send(3)	Update(1,14)	Update(1,11)
Update(2,8)	Update(2,9)	Update(2,6)	Update(1,15)
Update(2,12)	Update(2,13)	Update(2,10)	Update(2,7)
Update(3,4)	Update(3,5)	Update(2,14)	Update(2,11)
Update(3,8)	Update(3,9)	Update(3,6)	Update(2,15)
Update(3,12)	Update(3,13)	Update(3,10)	Update(3,7)
		Update(3,14)	Update(3,11)
			Update(3,15)

Processors, Nodes, Cores?

- In the description of the algorithms I always assume that we have “processors”
  - This term is vague, “Processing Element" (PE) is better
  - It means: something on which I can run sequential code
- Our real machines have multi-core compute nodes
- It’s up to you to decide what you mean by “processors"
- Let’s take an example...

Example

- On our Cray you got 4 nodes to yourself after waiting in the batch queue
- You need to run your MPI code, and you have 2 commonplace options
- **Option #1:** Each core is a “processor”: 80 MPI processes
- **Option #2:** Each node is a “processor”: 4 MPI processes, each using 20 threads (Hybrid Parallelism)
- But each group of 4 cores could be a “processor": 20 MPI processes, each using 4 threads
- Using MPI+OpenMP is generally a better idea
- Many users still don’t multi-thread (21st century, anyone? legacy... sigh)
  - Which is why you can request a single core on our cluster

Conclusion

- Onward to 2-D Data Distributions in another set of lecture notes...