

# Principles of High Performance Computing (ICS 632)



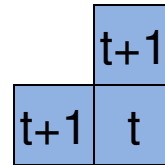
---

Algorithms on a Ring (II)

# Stencil Application (Section 4.3)

- We've talked about stencil applications in the context of shared-memory programs

0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12



$\text{new} = \text{update}(\text{old}, W, N)$

- We found that we had to cut the matrix in “small” blocks
  - On a ring the same basic idea applies, but let's do it step-by-step



# Stencil Application

---

- Let us, for now, consider that the domain is of size  $n \times n$  and that we have  $p=n$  processors
  - Classic way to first approach a problem
- Each processor is responsible for computing one row of the domain (at each iteration)
- Each processor holds one row of the domain and has the following declaration:

```
var A: array[0..n-1] of real
```
- One first simple idea is to have each processor send each cell value to its neighbor as soon as that cell value is computed
- **Basic principle:** do communication as early as possible to get your “neighbors” started as early as possible
  - Remember that one of the goals of a parallel program is to reduce idle time on the processors
- We call this algorithm the Greedy algorithm, and seek an evaluation of its performance



# The Greedy Algorithm

---

```
q = MY_NUM()
p = NUM_PROCS
if (q == 0) then
    A[0] = Update(A[0],nil,nil)
    Send(A[0],1)
else
    Recv(v,1)
    A[0] = Update(A[0],nil,v)
endif
for j = 1 to n-1
    if (q == 0) then
        A[j] = Update(A[j], A[j-1], nil)
        Send(A[j],1)
    elsif (q == p-1) then
        Recv(v,1)
        A[j] = Update(A[j], A[j-1], v)
    else
        Send(A[j-1], 1) || Recv(v,1)
        A[j] = Update(A[j], A[j-1], v)
    endif
endfor
```

First element of the row

Other elements

note the use of "nil"  
for borders and corners



# Greedy Algorithm

---

- This is all well and good, but typically we have  $n > p$
- Assuming that  $p$  divides  $n$ , each processor will hold  $n/p$  rows
  - Good for load balancing
- The goal of a greedy algorithm is always to allow processors to start computing as early as possible
- This suggests a **cyclic** allocation of rows among processors

P0
P1
P2
P0
P1
P2
P0
P1
P2

- P1 can start computing after P0 has computed its first cell



# Greedy Algorithm

---

- Each processor holds  $n/p$  rows of the domain
- Thus it declares:  

```
var A[0..n/p-1,n] of real
```
- Which is a contiguous array of rows, with these rows not contiguous in the domain
  - Therefore we have a non-trivial mapping between global indices and local indices, but we'll see that they don't appear in the code
- Let us rewrite the algorithm



# The Greedy Algorithm

---

```
p = MY_NUM()
q = NUM_PROCS
For i = 0 to n/p -1
    if (q == 0) and (i == 0) then
        A[0,0] = Update(A[0,0],nil,nil)
        Send(A[0],1)
    else
        Recv(v,1)
        A[i,0] = Update(A[i,0],nil,v)
    endif
    for j = 1 to n-1
        if (q == 0) and (i == 0) then
            A[i,j] = Update(A[i,j], A[i,j-1], nil)
            Send(A[i,j],1)
        elsif (q == p-1) and (i = n/p-1) then
            Recv(v,1)
            A[i,j] = Update(A[i,j], A[i-1,j], v)
        else
            Send(A[i,j-1], 1) || Recv(v,1)
            A[i,j] = Update(A[i,j], A[i-1,j-1], v)
        endif
    endfor
endfor
```



# Performance Analysis

---

- Let  $T(n,p)$  denote the computation time of the algorithm for a  $n \times n$  domain and with  $p$  processors
- At each step a processor does at most three things
  - Receive a cell
  - Send a cell
  - Update a cell
- The algorithm is “clever” because at each step  $k$ , the sending of messages from step  $k$  is overlapped with the receiving of messages at step  $k+1$
- Therefore, the time needed to compute one algorithm step is the sum of
  - Time to send/receive a cell:  $L + b$
  - Time to perform a cell update:  $w$
- So, if we can count the number of steps, we can simply multiply and get the overall execution time





# Performance Analysis

---

- It takes  $p-1$  steps before processor  $P_{p-1}$  can start computing its first cell
- Thereafter, this processor can compute one cell at every step
- The processor holds  $n*n/p$  cells
- Therefore, the whole program takes:  $p-1+n*n/p$  steps
- And the overall execution time:  
$$T(n,p) = (p - 1 + n^2/p) (w + L + b)$$
- The sequential time is:  $n^2w$
- The Speedup,  $S(n,p) = n^2w / T(n,p)$
- When  $n$  gets large,  $T(n,p) \sim n^2/p (w + L + b)$
- Therefore,  $\text{Eff}(n,p) \sim w / (w + L + b)$
- This could be WAY below one
  - In practice, and often,  $L + b \gg w$
- Therefore, this greedy algorithm is probably not a good idea at all!



# Granularity

---

- How do we improve on performance?
- What really kills performance is that we have to do so much communication
  - Many bytes of data
  - Many individual messages
- So we we want is to augment the **granularity** of the algorithm
  - Our “tasks” are not going to be “update one cell” but instead “update multiple cells”
- This will allow us to reduce both the amount of data communicated and the number of messages exchanged



# Reducing the Granularity

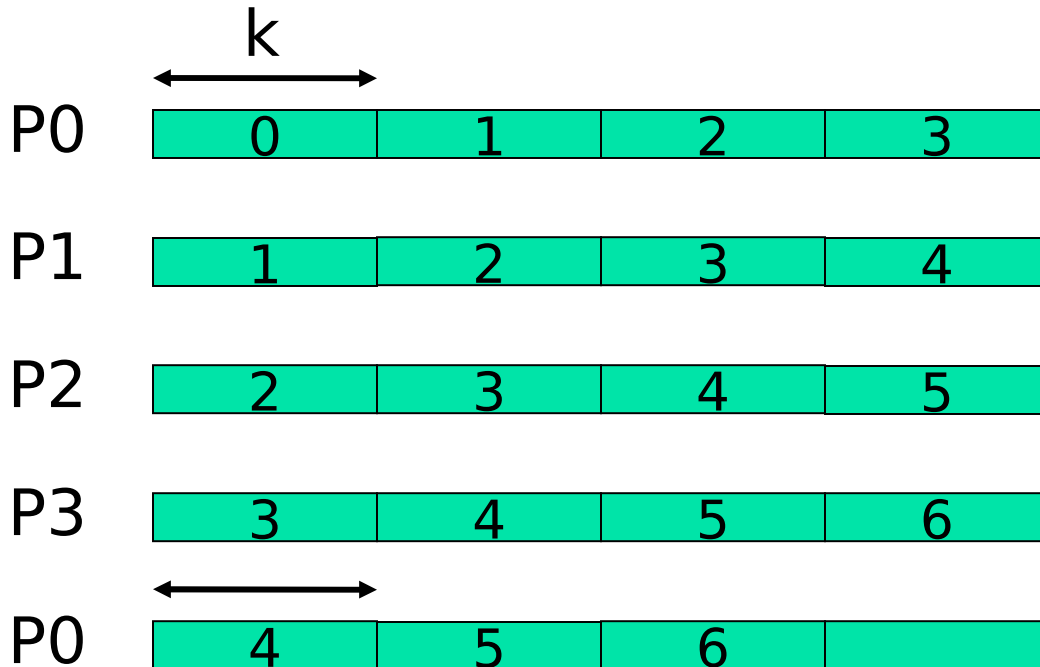
---

- A simple approach: have a processor compute  $k$  cells in sequence before sending them
- This is in conflict with the “get processors to compute as early as possible” principle we based our initial greedy algorithm on
  - So we will reduce communication cost, but will increase idle time
- Let us assume that  $k$  divides  $n$
- Each row now consists of  $n/k$  segments
  - If  $k$  does not divide  $n$  we have left over cells and it complicates the program and the performance analysis and as usual doesn't change the asymptotic performance analysis



# Reducing the Granularity

---



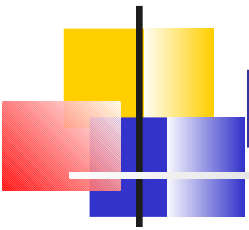
- The algorithm computes segment after segment
- The time before P1 can start computing is the time for P0 to compute a whole segment
- Therefore, it will take longer until  $P_{p-1}$  can start computing

# Reducing the Granularity

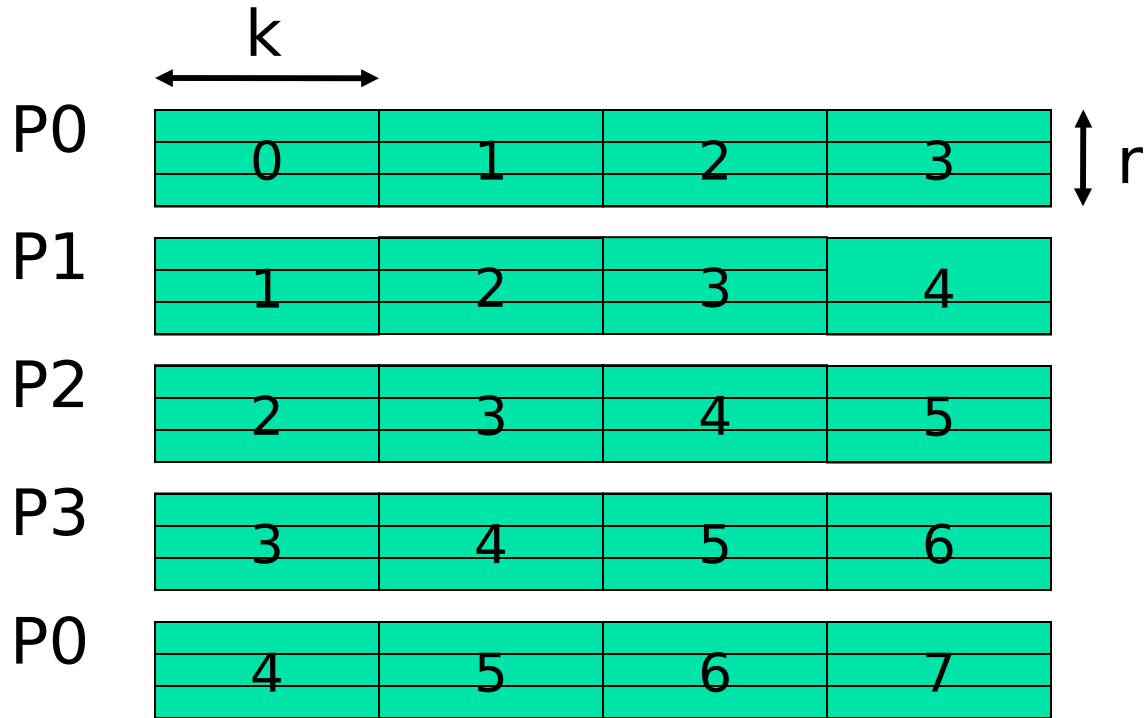
## More

---

- So far, we've allocated non-contiguous rows of the domain to each processor
- But we can reduce communication by allocating processors groups of contiguous rows
  - If two contiguous rows are on the same processors, there is no communication involved to update the cells of the second row
- Let us use say that we allocate blocks of rows of size  $r$  to each processor
  - We assume that  $r \cdot p$  divides  $n$
- Processor  $P_i$  holds rows  $j$  such that
$$i = \text{floor}(j/r) \bmod p$$
- This is really a “block cyclic” allocation



# Reducing the Granularity





# Idle Time?

---

- One question is: does any processor stay idle?
- Processor  $P_0$  computes all values in its first block of rows in  $n/k$  algorithm steps
- After that, processor  $P_0$  must wait for cell values from processor  $P_{p-1}$
- But  $P_{p-1}$  cannot start computing before  $p$  steps
- Therefore:
  - If  $p \geq n/k$ ,  $P_0$  is idle
  - If  $p < n/k$ ,  $P_1$  is not idle
- If  $p < n/k$ , then processors had better be able to buffer received cells while they are still computing
  - Possible increase in memory consumption



# Performance Analysis

---

- It is actually very simple
- At each step a processor is involved at most in
  - Receiving  $k$  cells from its predecessor
  - Sending  $k$  cells to its successor
  - Updating  $k \cdot r$  cells
- Since sending and receiving are overlapped, the time to perform a step is  $L + k b + k r w$
- Question: How many steps?
- Answer: It takes  $p-1$  steps before  $P_{p-1}$  can start doing any thing.  $P_{p-1}$  holds  $n^2/(pkr)$  blocks
- Execution time:

$$T(n,p,r,k) = (p-1 + n^2/(pkr)) (L + kb + k r w)$$





# Performance Analysis

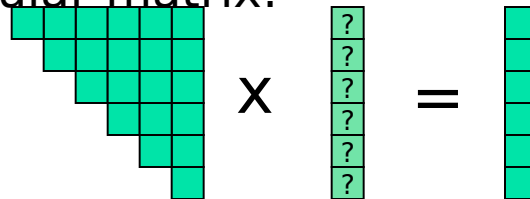
---

- Our naïve greedy algorithm had asymptotic efficiency equal to  $w / (w + L + b)$
- This algorithm does better: Assympt. Eff =  $w / (w + L/rk + b/r)$ 
  - Divide  $n^2w$  by  $p$   $T(n,p,r,k)$
  - And make  $n$  large
- In the formula for the efficiency we clearly see the effect of the granularity increase
- Asymptotic efficiency is higher
- But not equal to 1
- Therefore, this is a “difficult” application to parallelize
  - We can try to do the best we can by increasing  $r$  and  $k$ , but it’s never going to be perfect
- One can compute the optimal values of  $r$  and  $k$  using numerical solving
  - See the book for details



# 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:


$$\begin{bmatrix} & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{bmatrix} \times \begin{bmatrix} ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix} = \begin{bmatrix} \\ \\ \\ \\ \end{bmatrix}$$

equation  $n-i$  has  $i$  unknowns, with

# Gaussian Elimination

$$\begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & -2 & 2 \\ \hline 1 & 2 & -1 \\ \hline \end{array} \mathbf{x} = \begin{array}{|c|} \hline 0 \\ \hline 4 \\ \hline 2 \\ \hline \end{array}$$

Subtract row 1 from rows 2 and 3

$$\begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 0 & -3 & 1 \\ \hline 0 & 1 & -2 \\ \hline \end{array} \mathbf{x} = \begin{array}{|c|} \hline 0 \\ \hline 4 \\ \hline 2 \\ \hline \end{array}$$

Multiple row 3 by 3 and add row 2

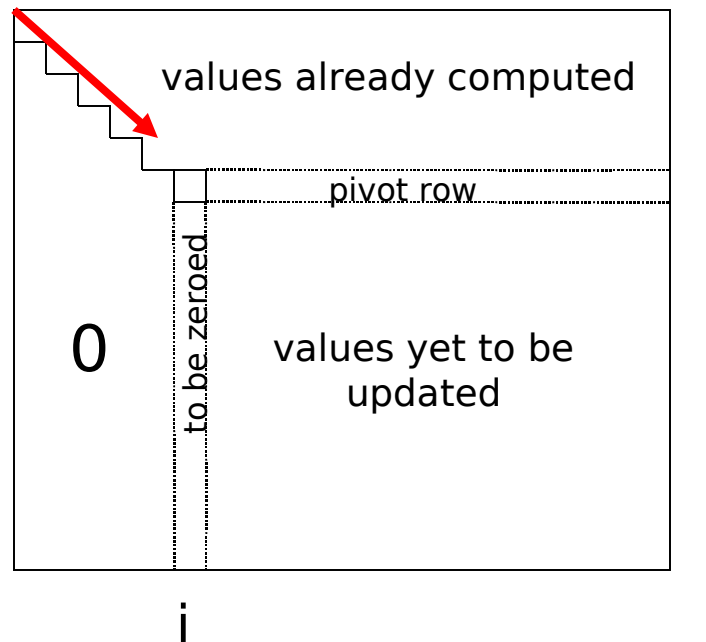
$$\begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 0 & -3 & 1 \\ \hline 0 & 0 & -5 \\ \hline \end{array} \mathbf{x} = \begin{array}{|c|} \hline 0 \\ \hline 4 \\ \hline 1 \\ \hline 0 \\ \hline \end{array}$$

Solving equations in reverse order (backsolving)

$$\begin{array}{l} -5x_3 = 10 \\ \xrightarrow{\quad} 3x_2 + x_3 = 4 \\ \xrightarrow{\quad} x_1 + x_2 + x_3 = 0 \end{array} \quad \begin{array}{l} x_3 = -2 \\ \xrightarrow{\quad} x_2 = -2 \\ \quad \quad x_1 = 4 \end{array}$$

# Gaussian Elimination

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





# 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

---

0	1
1	1

- A few pathological cases
- Division by small numbers → round-off error in computer arithmetic
- Consider the following system
$$\begin{aligned}0.0001x_1 + x_2 &= 1.000 \\ x_1 + x_2 &= 2.000\end{aligned}$$
- exact solution:  $x_1=1.00010$  and  $x_2 = 0.99990$
- 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
- $(1 - 1)x_1 + (1 - 10^4)x_2 = 2 - 10^4$
- But, in finite precision with only 3 digits:
  - $1 - 10^4 = -0.9999 \text{ E}+4 \sim -0.999 \text{ E}+4$
  - $2 - 10^4 = -0.9998 \text{ E}+4 \sim -0.999 \text{ E}+4$
- Therefore,  $x_2 = 1$  and  $x_1 = 0$  (from the first equation)



# Partial Pivoting

---

- One can just swap rows

$$x_1 + x_2 = 2.000$$

$$0.0001x_1 + x_2 = 1.000$$

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

$$(1 - 0.0001)x_2 = 1 - 0.0001$$

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

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

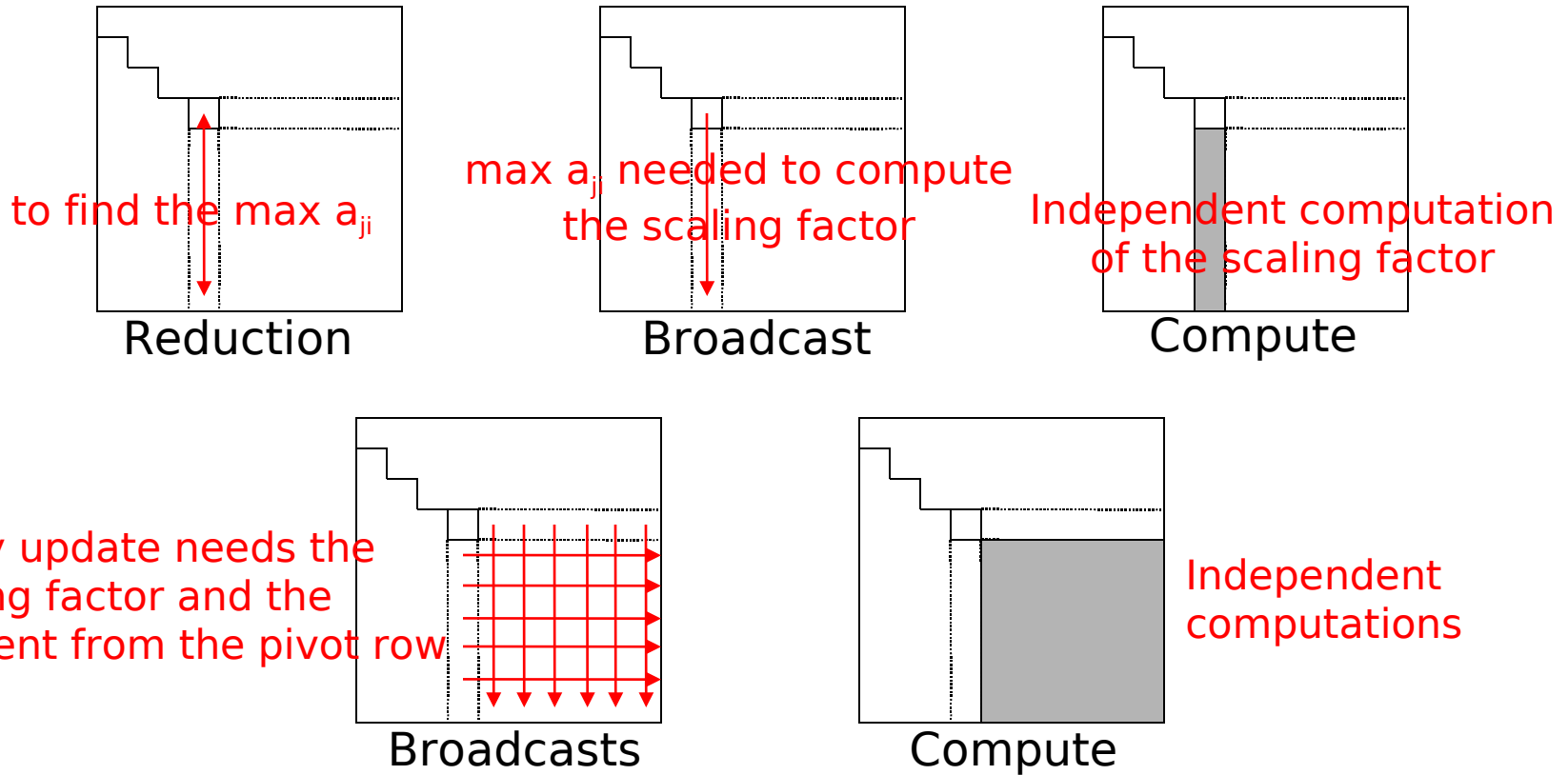
- Total Pivoting

- Look for the greatest element ANYWHERE in the matrix
- Swap columns
- Swap rows

- Numerical stability is really a difficult field

# Parallel Gaussian Elimination?

- Assume that we have one processor per matrix element

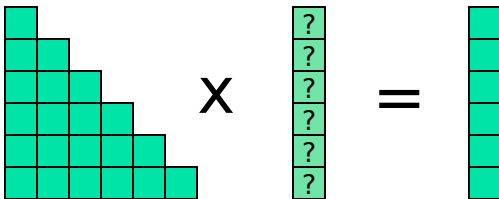




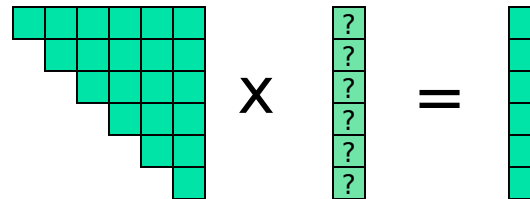
# LU Factorization (Section 4.4)

- 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  $A = LU$ , where  $L$  is a lower triangular matrix, and  $U$  is an upper triangular matrix  $O(n^3)$
- Then  $Ax = b$  is written  $LUx = b$
- Solve  $Ly = b$   $O(n^2)$
- Solve  $Ux = y$   $O(n^2)$

triangular system solves are easy



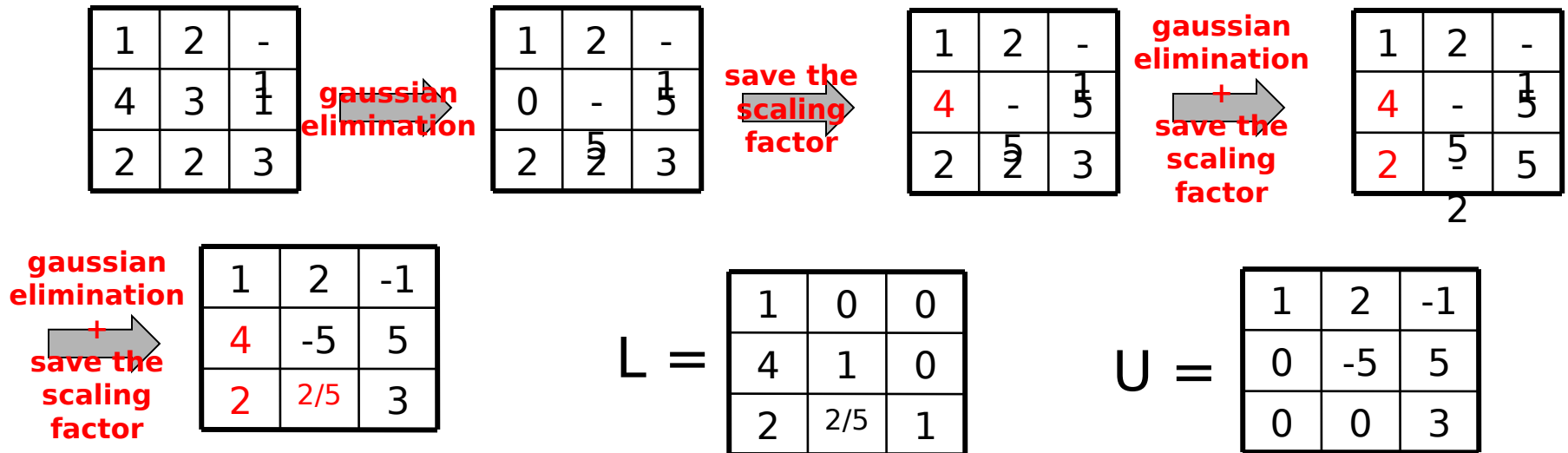
equation  $i$  has  $i$  unknowns



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.



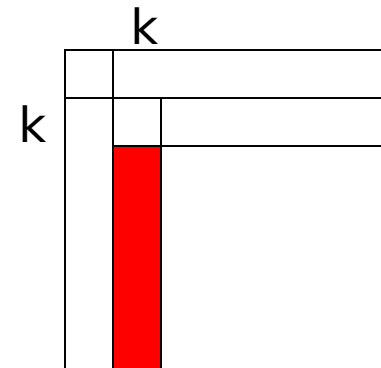
- Magically,  $A = L \times U$  !
- Should be done with pivoting as well

# 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  
       $a_{ik} \leftarrow -a_{ik} / a_{kk}$   
    for j = k+1 to n-1  
      // Task  $T_{kj}$ : update of column j  
      for i=k+1 to n-1  
         $a_{ij} \leftarrow a_{ij} + a_{ik} * a_{kj}$   
      }  
    }  
}
```

stores the scaling factors







# 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  $\text{alloc}(k)$  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] ←  $a_{ik} \leftarrow -a_{ik} / a_{kk}$ 
    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
           $a_{ij} \leftarrow a_{ij} + \text{buffer}[i-k-1] * a_{kj}$ 
  }
}
```



# 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  $\text{alloc}(k)$  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 = 1 to r-1
```

```
            for i=k+1 to n-1
```

```
                a[i,j] ← a[i,j] + buffer[i-k-1] * a[k,j]
```

```
    }
```

```
}
```

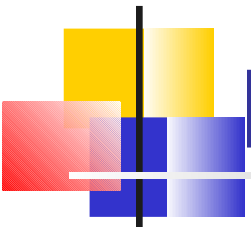




# What about the Alloc function?

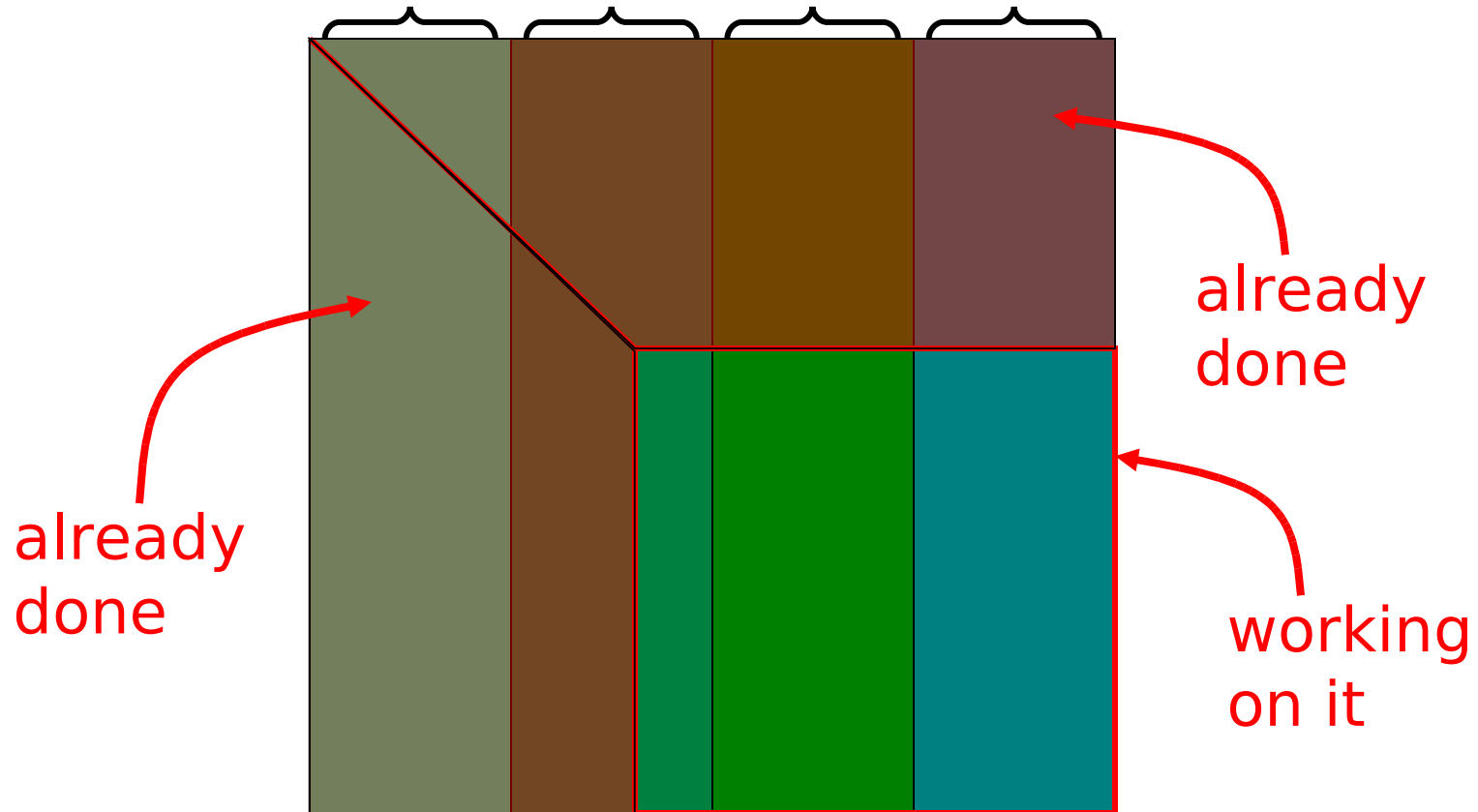
---

- 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

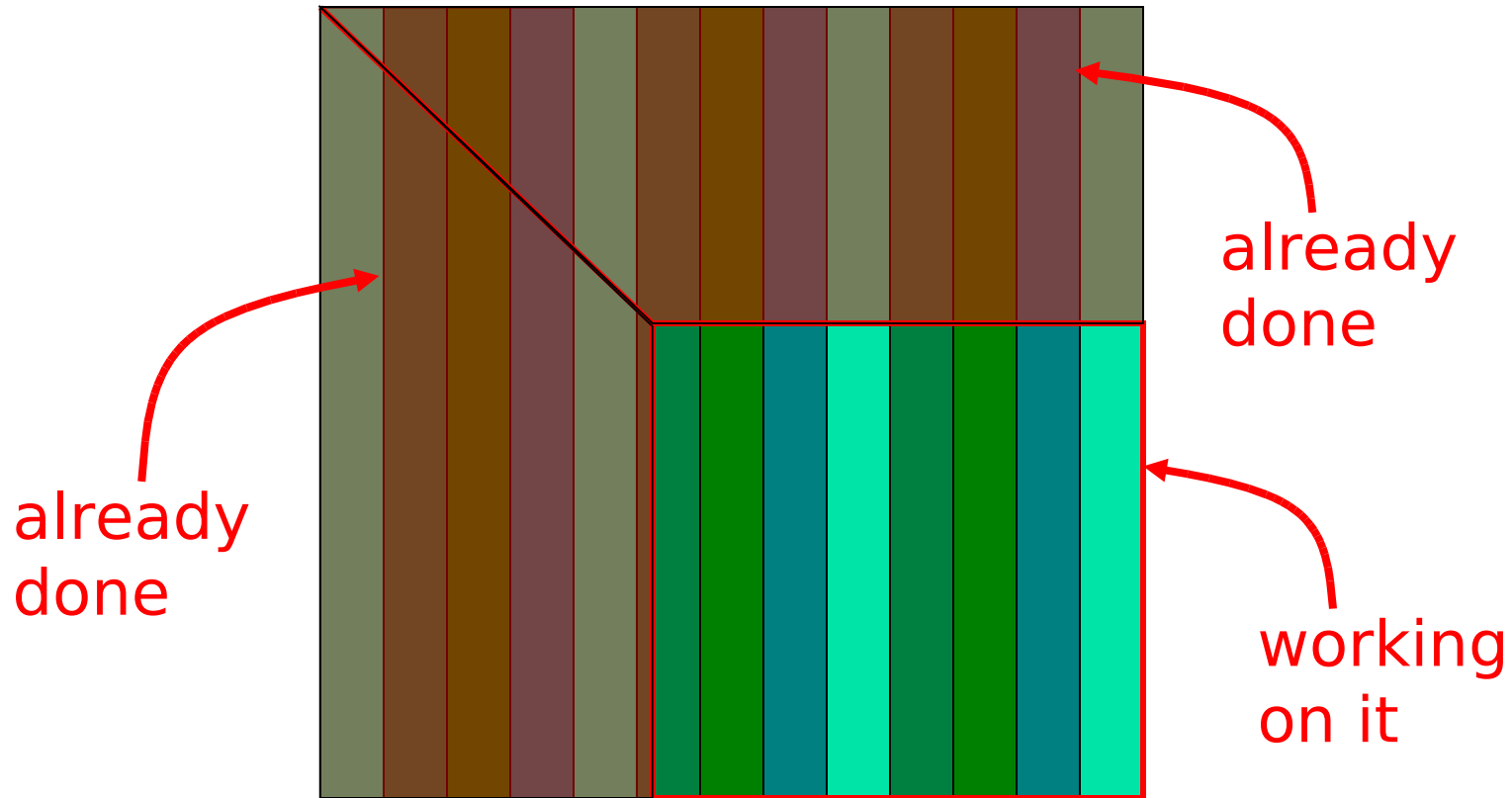


# Bad load balancing

P1 P2 P3 P4



# 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  $k=0, \dots, j-1$
- At step  $k$ , elements  $i=k+1, \dots, n-1$  are updates
  - indices start at 0
- Therefore, at step  $k$ , the update of column  $j$  entails  $n-k-1$  updates
- The total number of updates for column  $j$  in the execution is:

$$\sum_{k=0}^{j-1} (n - k - 1) = j(n - 1) - \frac{j(j - 1)}{2}$$

# Proof that load balancing is good

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

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

- 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,1] / a[k,1]
```

```
      l ← l+1
```

```
  broadcast(alloc(k), buffer, n-k-1)
```

```
  for j = 1 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





# 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,1] / a[k,1]
```

```
            l ← l+1
```

```
        broadcast(alloc(k), buffer, n-k-1)
```

```
        for j = 1 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 = 1 to r-1
```

```
    for i=k+1 to n-1
```

```
      a[i,j] ← a[i,j] + buffer[i-k-1] * a[k,j]
```

```
  }
```

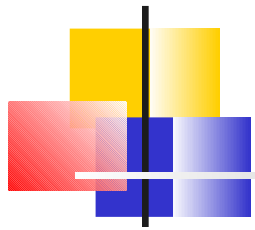
```
}
```



# 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

First four stages

Some communication occurs in parallel with computation

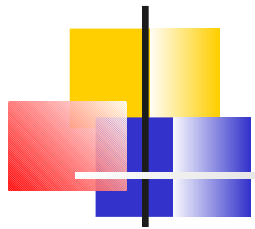
Prep(0)			
Send(0)	Recv(0)		
Update(0,4)	Send(0)	Recv(0)	
Update(0,8)	Update(0,1)	Send(0)	Recv(0)
Update(0,12)	Update(0,5)	Update(0,2)	Update(0,3)
	Update(0,9)	Update(0,6)	Update(0,7)
	Update(0,13)	Update(0,10)	Update(0,11)
	Prep(1)	Update(0,14)	Update(0,15)
	Send(1)	Recv(1)	
	Update(1,5)	Send(1)	Recv(1)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



# 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

First four stages

Many communications occur in parallel with computation

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



# Further improving performance

---

- One can use local overlap of communication and computation
  - multi-threading, good MPI non-blocking implementation, etc.
- There is much more to be said about parallel LU factorization
  - Many research articles
  - Many libraries available
- It's a good example of an application for which one can think hard about operation orderings and try to find improved sequences
  - The basic principle is always the same: sends things as early as possible

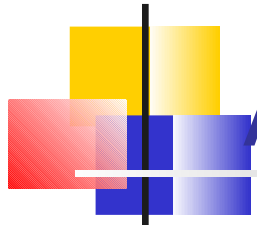


# Another Stencil Application

---

- Let us consider a simple stencil:
  - $C_{\text{new}} = \text{Update}(C_{\text{old}}, W_{\text{old}}, E_{\text{old}}, N_{\text{old}}, S_{\text{old}})$
- To implement this stencil (in sequential or in parallel), one need to keep two arrays around:
  - The original one: A
  - The new one: B
- To run multiple iterations on can just swap these pointers
- The simplest way to partition the domain among processors on the ring is to give a block of  $r = n/p$  consecutive rows to each processor
- Declaration: `var A,B: array[0..r-1, 0..n-1] of real;`
- Each processor can update rows `1..r-2` easily, but for its top row and its bottom row, it needs to receive elements from its neighbors
- For simplicity we assume that P0 and Pp-1 exchange rows
  - We have a “wrap-around domain”





# Another Stencil Application

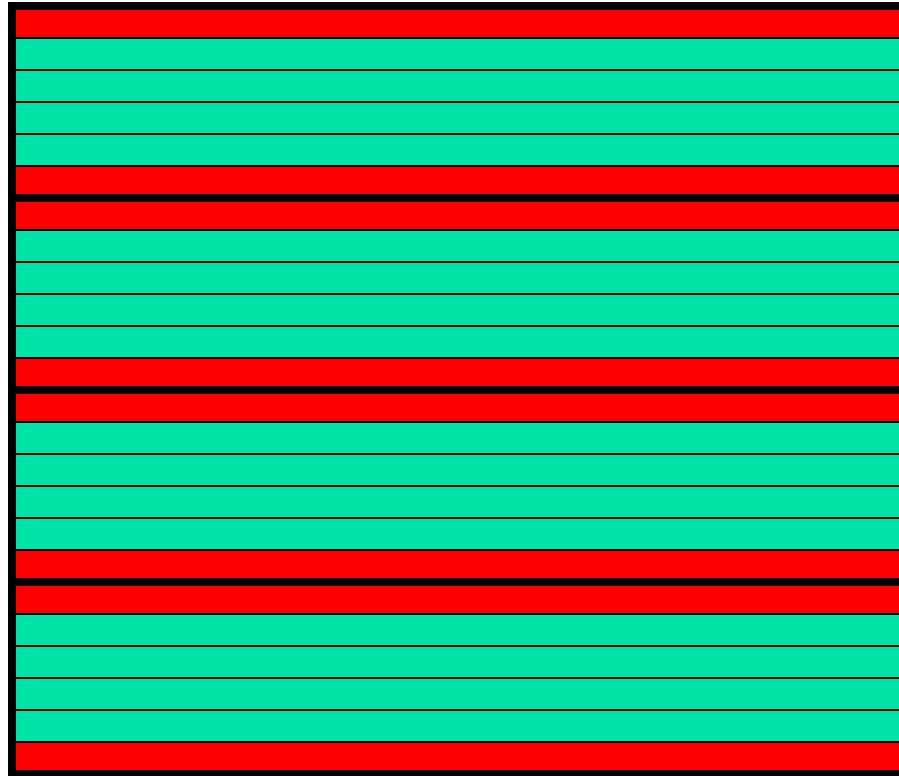
---

P0

P1

P2

P3





# Communication Pattern

---

- To “swap” rows with a neighbor is a bit complicated
  - Sending to a Successor is easy
  - Sending to a predecessor requires  $p-1$  hops
- The structure of the algorithm is:
  1. send/recv border rows || compute green cells
  2. compute red cells
- We assume that each processor declares two “buffer” arrays
  - var fromPred, fromSucc: array[0.. $n-1$ ] of real
- Let us write the communication part
  - See full algorithm on page 132



# Communication Pattern

---

- Each processor does:

```
tempS = &(A[0,0])
```

```
for k = 1 to p-2:
```

```
    Send(tempS, n) || Recv(tempR, n)
```

```
    swap(tempS, tempR)
```

```
endfor
```

```
Send(tempS, n) || Recv(fromSucc, n)
```

```
Send(&(A[r-1,0]), n) || Recv(fromPred,n)
```

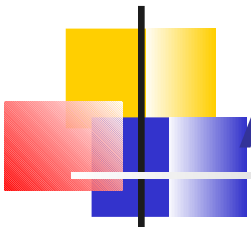
- At this point, every processor has the fromPred and fromSucc arrays filled with the needed cell values



# Performance Analysis

---

- The communication phase consists in a sequence of  $p$  concurrent sends and receives of a row of  $n$  cells
  - It takes time  $pL + pnb$
- It occurs concurrently with a computation phase that computes  $r-2$  rows
  - It takes time  $(r-2)nw = (n/p - 2)nw$
- Then, we have a computation phase that computes 2 rows:
  - It takes time  $2nw$
- The overall execution time is:
  - $T(n,p) = \max\{pL + pnb, (n/p - 2)nw\} + 2nw$
- When  $n$  becomes large:  $T(n,p) \sim n^2w/p$
- Therefore we have perfect asymptotic efficiency



# Another virtual topology?

---

- The previous code is asymptotically optimal, so we're essentially done
  - There is not really way to reduce communication overhead when  $n$  isn't very large
- But the communication phase is a bit cumbersome
- How about using a **bidirectional** ring?
- Again, we can choose whatever we want really
  - As long as the physical platform "supports" it
- With a bidirectional ring, the communication phase is written as:  
Send(pred, &(A[0,0],n) || Recv(succ, fromSucc,n)  
Send(succ, &(A[r-1,0], n) || Recv(pred, fromPred, n)
- Much simpler. much more readable



# Conclusion

---

- We can do a lot of things with a ring
  - We saw our first example that a modification to the virtual topology can make the code much simpler (albeit in a trivial case)
- Next, we'll look at a 2-D grid topology, which induces 2-D data distributions