Algorithms on rings (contd.)

Some references

Parallel Programming – For Multicore and Cluster System

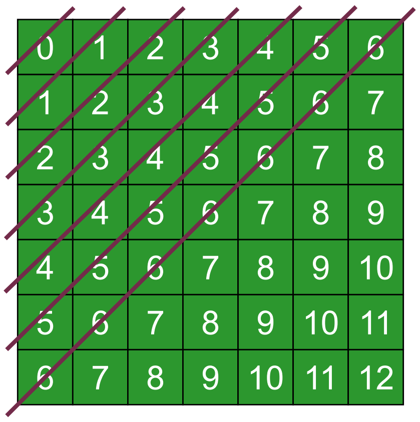
T. Rauber, G. Rünger

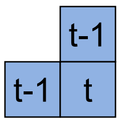
Parallel Algorithms

H. Casanova, A. Legrand, Y. Robert

Stencil applications

- Discrete domain with cells. Each cell holds some value and has neighbor cells
- Update the values of a cell using the values of the neighbor cells → stencil
- Used in many areas of science and engineering
    - image processing, approximate solutions to differential equations, simulation of complex cellular automata, …





```
new = update(old,W,N)
```

Stencil applications, contd.

- We assume that
    - we have a domain of size `nxn` and
    - we run the program on `p = n` processors
- Each processor is responsible for computing a domain row at each iteration
- Each processor owns one row of the domain

```
var A: array[0..n-1] of real
```

- **First simple idea**
    - Each processor sends each cell as soon as it is computed
    - Start communications as soon as possible so neighbors start as soon as possible
    - Reduce the processor wait time!
    - This algorithm is called a **greedy algorithm**

Greedy algorithm

```
q = MY_NUM()
p = NUM_PROCS
if (q == 0) then
    A[0] = Update(A[0], nil, nil)
    Send(A[0],1)
elseif (q == p-1) then
    Recv(v,1)
    A[0] = Update(A[0], nil, v)
else
    Recv(v,1)
    A[0] = Update(A[0], nil, v)
    Send(A[0],1)
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
```

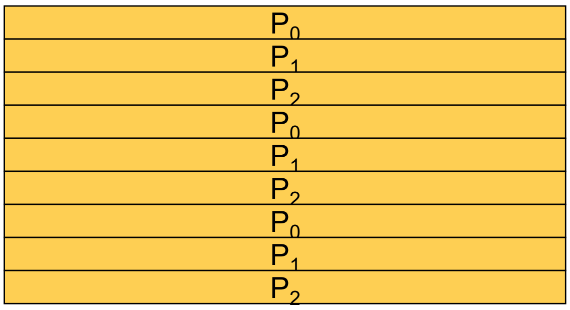
First element of each row

Other elements

Use of “nil” for borders and corners

Greedy algorithm, contd.

- Usually, we have n>p
- If we suppose that p divides n, each processor owns n/p rows
    - Good load balancing
- The purpose of the algorithm is always to allow the processors to start as soon as possible
- This suggests a cyclic placement of the lines on the processors



- P_1 can start its computation after P_0 has started calculating its first cell

Greedy algorithm, contd.

- Each processor owns `n/p` lines of the domain
- Each processor declares

  `var A[0..n/p-1,n] of real`

- Which is a contiguous array of rows with its rows not contiguous in the domain
  - We have a non-trivial mapping between global indexes and local ones

Greedy algorithm, contd.

```
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[0,0], 1)
    elseif (q == p-1) and (i = n/p-1)
          Recv(v,1)
          A[i,0] = Update(A[i,0], nil, v)
    else
          Recv(v,1)
          A[i,0] = Update(A[i,0], nil, v)
          Send(A[i,0], 1)
    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 $T(n,p)$ be the execution time of the algorithm for a domain of size $n \times n$ on p processors
- At each step, a processor performs at least three operations
  - Receive one cell
  - Send one cell
  - Update one cell
- The algorithm is somehow optimized because at each step k the sending of the messages of step k is overlapped with the reception of the messages of step $k+1$
- Then, the time required to calculate a step of the algorithm is the sum of
  - Time to send/receive one cell: $L + b$
  - Time to perform an update: w
- If one can have the number of steps, multiply it to the sum to have the overall execution time

Performance analysis, contd.

- It takes $p-1$ steps before the P_{p-1} processor can start calculating its first cell
- Then, it can compute a cell at each step
- The processor has $n*n/p$ cells
- So the program takes $p-1 + n*n/p$ steps
- And the total execution time is

$$T(n,p) = (p-1 + n^2/p)(w + L + b)$$

- Sequential time: n^2w
- Acceleration: $S(n,p) = n^2w/T(n,p)$
- When n grows: $T(n,p) \sim n^2/p(w + L + b)$
- Efficiency: $Eff(n,p) \sim w/(w + L + b)$
  - This can be much less than 1 ($L + b >> w$ in practice)
  - The algorithm is not really efficient!

Granularity

- How to improve performance?

- Too much communication in the greedy algorithm
  - Large number of bytes sent
  - Large number of individual messages

- **Idea:** increase the granularity of the algorithm
  - Do not update **a cell** but **several cells** at each step
  - This will reduce both the **volume of communicated data** and the **number of messages** exchanged

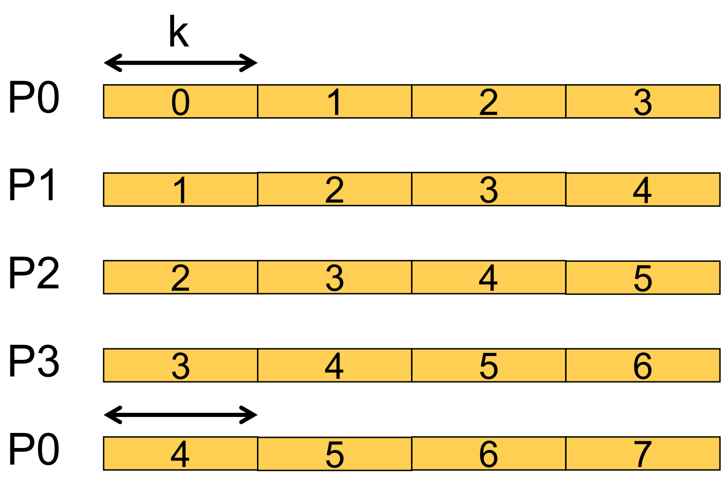
Increasing granularity

- **A simple approach**
  - Compute `k` cells sequentially before sending them

- Conflict with the sending principle "as soon as possible" used for the greedy algorithm
  - Reduced cost of communications
  - We will increase the waiting time

- If we suppose that `k` divides `n`
- Each row is composed of `n/k` segments
  - If `k` does not divide `n`, it complicates the algorithm and the performance analysis but not the asymptotic performance

Increasing granularity, contd.

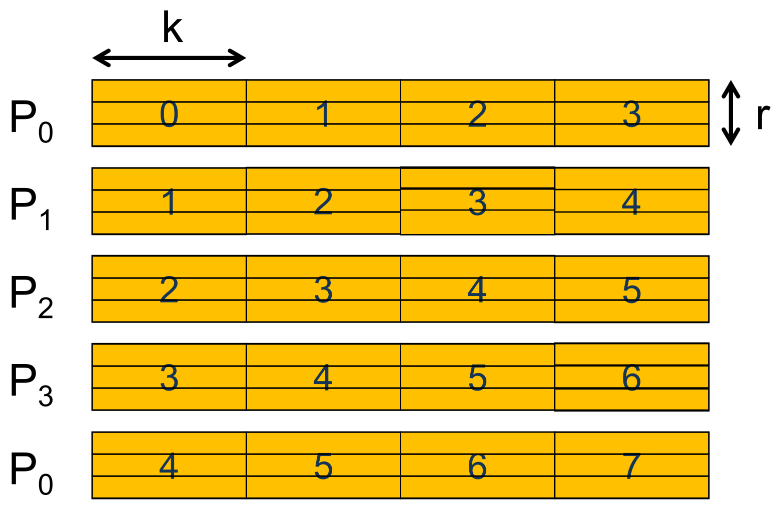


- The algorithm computes the segments one after the other
- The start time of P_1 is the time of P_0 to compute a segment (and send it!)
- This time increases with the length of the segment

Increasing granularity, contd.

- Up to now, non-contiguous rows of the domain have been allocated on each processor
- Communications can be further reduced by allocating blocks of rows to the processors
- If two contiguous rows are on the same processor, no communication for updating
- Assume that blocks of `r` rows are allocated to each processor
  - We suppose that `r*p` divides `n`
  - The processor P_i has the lines `j` such that `I = floor(j/r) mod p`
  - We have a block-cyclic allocation

Increasing granularity, contd.



Waiting time

- One question is: does any processor stay idle?
- Processor P_0 computes all the values in its first block in `n/k` steps
- Then it must wait for the cell values of the P_{p-1} processor
- But P_{p-1} can not start before `p-1` steps
- Then
  - If `p >= n/k`, P_0 waits
  - If `p < n/k`, P_0 does not wait
- If `p < n/k`, processors must buffer the received data while computing
  - Increases memory consumption

Performance analysis

- At each step, each processor
  - Receives `k` cells from its predecessor
  - Sends `k` cells to its successor
  - Updates `k*r` cells
- As each communications are overlapped, the time to execute one step is:

$$L + kb + krw$$

- How many steps do we have?
  - We need `p-1` steps before P_{p-1} can start
  - P_{p-1} owns $n^2/(pkr)$ blocks

- **Execution time**

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

Performance analysis, contd.

- The **greedy** algorithm had a asymptotic efficiency of

$$w/(w + L + b)$$

- The **block cyclic** performs better
  - Asymptotic efficiency:

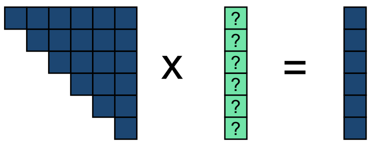
$$Eff = n^2w/p\ T(n,p,r,k) = w / (w + L/rk + b/r)$$

- Better efficiency but still not equal 1
- Application “difficult” to parallelize efficiently
- By increasing `r` and `k` we improve the efficiency
- We can compute the optimal `r` and `k`!

Resolution of linear systems of equations

- **Method to solve linear systems**
  - Used in 75% of scientific problems `[Dahlquist 1974]`

- **Most classic Gauss method**
  - 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:** transform the matrix A into a triangular matrix using these principles



Gaussian elimination

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -2 & 2 \\ 1 & 2 & -1 \end{pmatrix} x = \begin{pmatrix} 0 \\ 4 \\ 2 \end{pmatrix}$$

Subtract row 1 from rows 2 and 3

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & -3 & 1 \\ 0 & 1 & -2 \end{pmatrix} x = \begin{pmatrix} 0 \\ 4 \\ 2 \end{pmatrix}$$

Multiply row 3 by 3 and add row 2

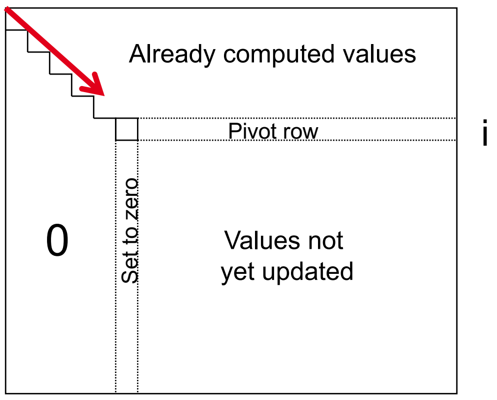
$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & -3 & 1 \\ 0 & 0 & -5 \end{pmatrix} x = \begin{pmatrix} 0 \\ 4 \\ 10 \end{pmatrix}$$

Solve the equations in reverse order (backsolving) $\longrightarrow$

$$\begin{aligned} -5x_3 &= 10 \\ -3x_2 + x_3 &= 4 \\ x_1 + x_2 + x_3 &= 0 \end{aligned} \quad \longrightarrow \quad \begin{aligned} x_3 &= -2 \\ x_2 &= -2 \\ x_1 &= 4 \end{aligned}$$

Gaussian elimination, contd.

- The Gaussian elimination passes through the matrix of the corner at the top left at the bottom right corner
- `i`-th step eliminates the non-zero sub-diagonal elements of column `i` by subtracting the `i`-th line multiplied by a_{ji}/a_{ii} at row `j`, for `j=i+1,..,n`



Gaussian elimination, contd.

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)
```

Optimizations that do not alter the algorithm but simplify the implementation and make it more efficient

- The right-hand part is generally conserved in the `n+1` column of the matrix (we speak of an augmented matrix)
- Computation of the term `A(i,j)/A(i,i)` outside the loop

Motivation for the pivoting

0	1
1	1

- Some pathological cases
- Divisions by small numbers → rounding errors
- Consider the following system

$$\begin{aligned} 0.0001x_1 + x_2 &= 1.000 \\ x_1 \quad + x_2 &= 2.000 \end{aligned}$$

- Exact solution: x_1=1.00010 and x_2 = 0.99990
- If you round up 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 we have only 3 digits:

$$\begin{aligned} 1 - 10^4 &= -0.9999 \text{ E+4} \sim -0.999 \text{ E+4} \\ 2 - 10^4 &= -0.9998 \text{ E+4} \sim -0.999 \text{ E+4} \end{aligned}$$

- Then, x_2 = 1 and x_1 = 0 (of the first equation)
- Far enough from the truth!

Partial pivoting

- We just exchange the rows

$$\begin{aligned} x_1 \quad + x_2 &= 2.000 \\ 0.0001x_1 + x_2 &= 1.000 \end{aligned}$$

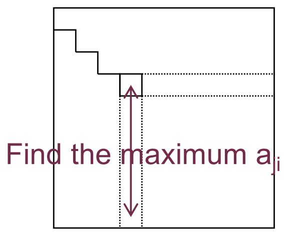
- Multiplying the first equation by 0.0001 and subtracting it from the second equation gives:

$$(1 - 0.0001)x_2 = 1 - 0.0001$$
$$0.9999\ x_2 = 0.9999 \Rightarrow x_2 = 1 \text{ then } x_1 = 1$$

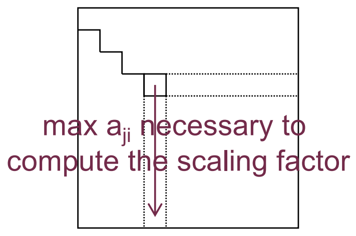
- The solution is close to the real solution
- **Partial pivoting**
  - For numerical stability, we do not follow the order of the lines but we take the next line (from i to n) which has the largest element in column I
  - This line is exchanged with line I (and the elements on the right side) before the subtractions
    - Not actually done but we keep an indirection table
- **Total pivoting**
  - Find the largest item anywhere and exchange rows and columns
- Numerical stability is a topic of research in its own right

Parallel Gaussian Elimination

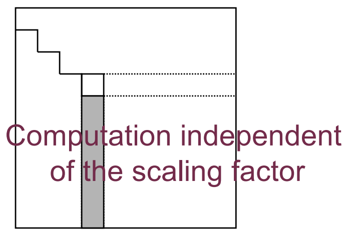
It is assumed that we have one element per processor



Reduction

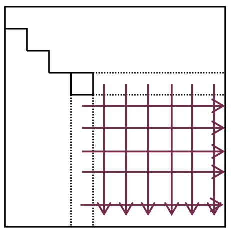


Broadcast

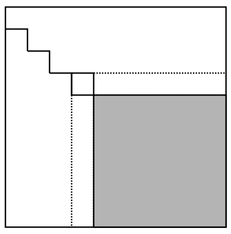


Computation

Each update needs the scaling factor and an element of the pivot row



Broadcasts



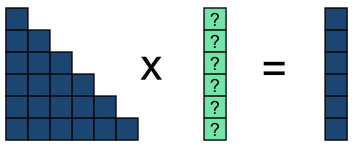
Computations

Independent computations

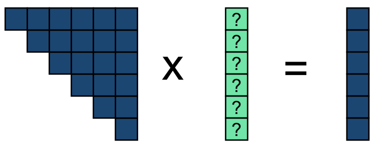
LU factorization

- The elimination of Gauss is simple but
  - If one has to solve several systems `Ax = b` for different values of `b` (appears in a large number of applications)?
- Other method: LU factorization
- `Ax = b`
- We transform `A` into `LU` where `L` is a lower triangular matrix and `U` is a top triangular matrix: $O(n^3)$
- Then Ax = b is written as `LU x = b`
- Solve `L y = b` $O(n^2)$
- Solve `U x = y` $O(n^2)$

Simple solve of triangular systems



Equation I has i unknowns



Equation n-I has i unknowns

LU factorization: principle

- It works like the Gaussian elimination, but instead of zeroing the elements, we "save" the scaling coefficients
- It is necessary to apply the pivoting
- At the end, `A = LU`

$$\begin{bmatrix}1&2&-1\\4&3&1\\2&2&3\end{bmatrix} \xrightarrow{\text{Gaussian elimination}} \begin{bmatrix}1&2&-1\\0&-5&5\\2&2&3\end{bmatrix} \xrightarrow{\text{Saving the scaling factors}} \begin{bmatrix}1&2&-1\\4&-5&5\\2&2&3\end{bmatrix} \xrightarrow{\text{Gaussian elimination + Saving the scaling factors}} \begin{bmatrix}1&2&-1\\4&-5&5\\2&-2&5\end{bmatrix}$$

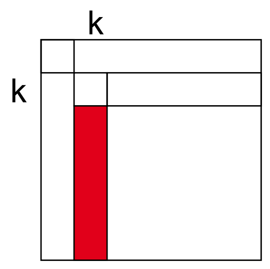
$$\xrightarrow{\text{Gaussian elimination + Saving the scaling factors}} \begin{bmatrix}1&2&-1\\4&-5&5\\2&2/5&3\end{bmatrix} \qquad L = \begin{bmatrix}1&0&0\\4&1&0\\2&2/5&1\end{bmatrix} \qquad U = \begin{bmatrix}1&2&-1\\0&-5&5\\0&0&3\end{bmatrix}$$

LU factorization, contd.

- We will take a look a the simplest version
  - No pivoting: we create only a set of indirections that are simple but make the code complicated without changing the global 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
           $a_{ij} \leftarrow a_{ij} + a_{ik} * a_{kj}$
  }
}
```

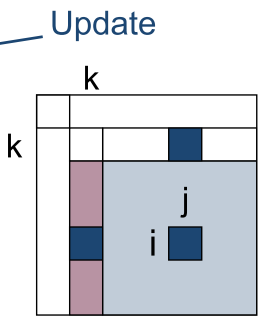
Stores the scaling factors



LU factorization, contd.

- We will take a look a the simplest version
  - No pivoting: we create only a set of indirections that are simple but make the code complicated without changing the global 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
           $a_{ij} \leftarrow a_{ij} + a_{ik} * a_{kj}$
  }
}
```



LU factorization on a ring

- Since the algorithm operates in columns from left to right, we must distribute the columns on the processors
- **Principle of the algorithm**
  - At each step, the processor which has the column `k` performs the task of preparing the task and broadcasts the lower part of the column `k` to all the others processors
    - Issue if the matrix is stored by rows
    - It should be remembered that the matrix can be stored as desired, as long as it is coherent and the correct output is generated
  - After the broadcast, the other processors can update their data
- Assume that there is a function `alloc(k)` which returns the rank of the processor which has column `k`
- First, we write first in terms of global indices, to avoid the arithmetic of complex indices

LU algorithm with broadcast

```
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] * $a_{kj}$
  }
}
```

Manage local indices

- Suppose that `p` divides `n`
- Each processor needs to store `r = n/p` columns and its local indices range from `0` to `r-1`
- After step `k`, only columns with indices greater than `k` will be used
- Simple idea: use a local index that everyone initializes to 0
- In step `k`, the processor `alloc(k)` increases its local index so that at the next step it points to the next local column

LU algorithm with broadcast, contd.

```
...
  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]
  }
}
```

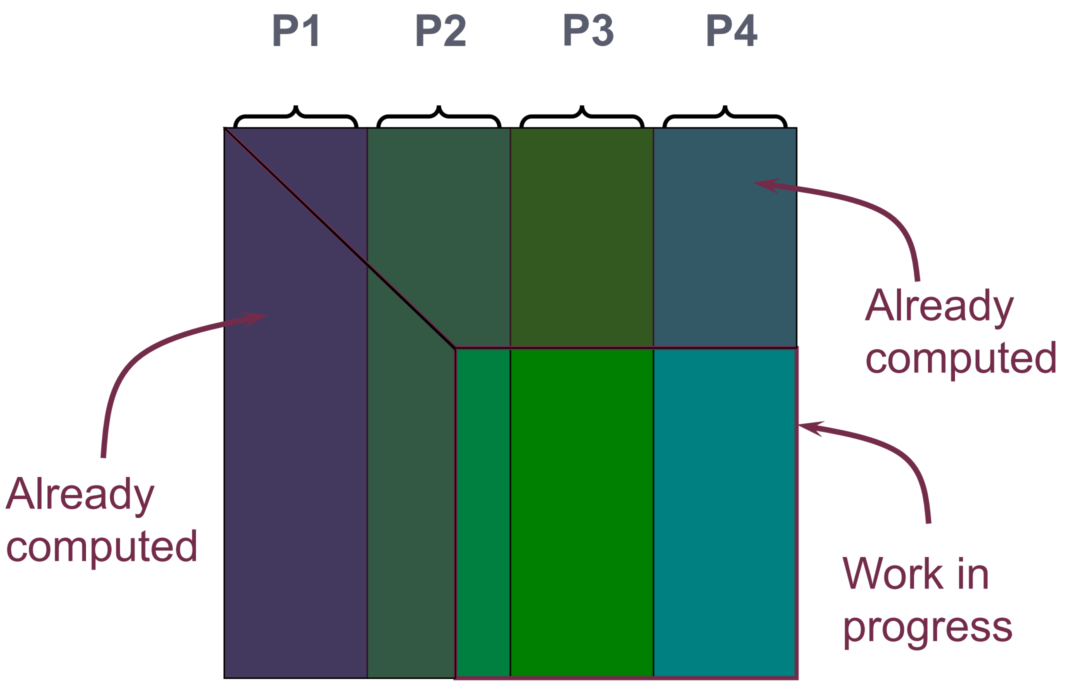
What's about the Alloc function?

- We did not specify how to write the `alloc` function:
  - How are the columns distributed on the processors?

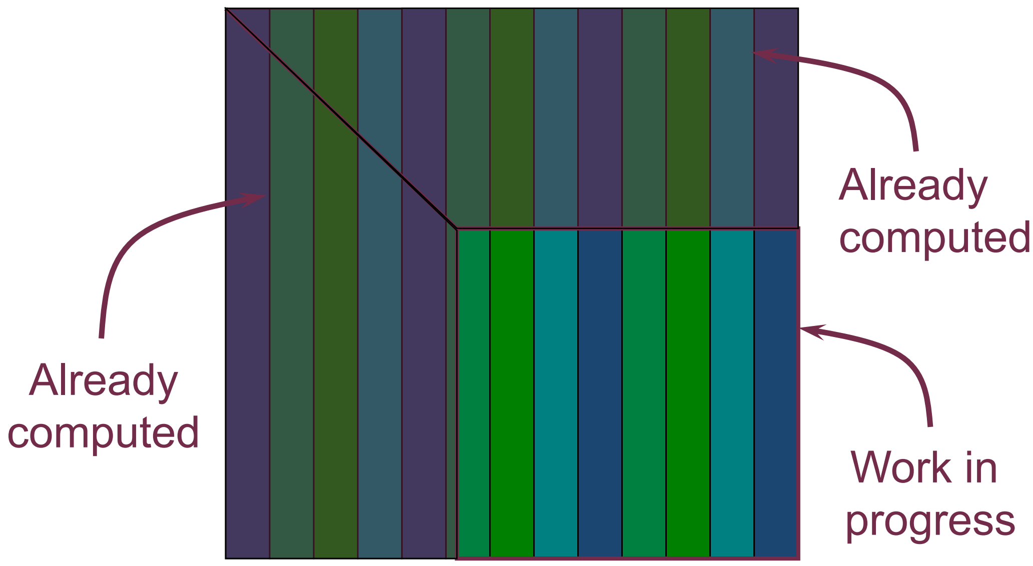
- **There are two complications**
  - The volume of the data to be calculated varies during the execution of the algorithm
    - At step k, columns k+1 to n-1 are updated
    - Fewer columns to update
  - The computation volume varies according to the columns
    - for example, column `n-1` is updated more often than column 2
    - Keeping the columns to the right of the matrix gives more work

- There is a great need for load balancing
  - All processors must have the same volume of work

Bad load-balancing



Good load balancing?



Cyclic distribution

Proof that load-balancing is good

- The computation consists of **two types of operations**
    - **Preparation** of columns
    - **Update** matrix elements
- There is **more updating than preparations** and so we must be very careful balancing preparations
- Consider column j
- Let's count the number of updates performed by the processor with column j
- The column j is updated at steps k = 0, ..., j-1
- At step k, the elements i = k + 1,..., N-1 are updates
    - Indexes start at 0
- Thus, in step k, the updating of the column j gives n-k-1 updates
- The total number of updates for column j during execution is:

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

Proof that load-balancing is good, contd.

- Consider the processor P_i, which contains the columns lp + i for l = 0, ..., n / p -1
- Processor P_i needs to perform this number of updates

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

- Good news, it can be computed!
    - Separate Terms
    - Use formulas for sums of integers and sums of squares
    - This gives:

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

- Does not depend of `i` !
- This is asymptotically the same on all processors
- So we have an 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 it take to run this code?
- Difficult to analyze because many tasks and communications
- The execution time is made of three terms terms
  - n-1 communications: $n L + (n^2/2)\ b + O(1)$
  - n-1 columns preparations: $(n^2/2)\ w' + O(1)$
  - Columns update: $(n^3/3p)\ w + O(n^2)$
- The execution time is then $\sim (n^3/3p)\ w$
- The sequential execution time is: $(n^3/3)\ w$

- We thus have a perfect asymptotic efficiency
- Not always the case in practice

- How to improve the algorithm?

Pipeline on a ring

- In the previous code, the algorithm uses a "classical" broadcast (`MPI_BROADCAST`)
- Nothing is done to take advantage of the ring and it's portable
  - Average performance for small sizes of n
- But it is ineffective
  - The n-1 steps of communications are not overlapped by the computations
  - Overhead !
- In fact, on a ring, with a cyclic distribution of the columns, the diffusion can be mixed with the computations

Previous version

```
...
  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]
  }
}
```

Pipelined *LU* factorization 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]
  }
}
```

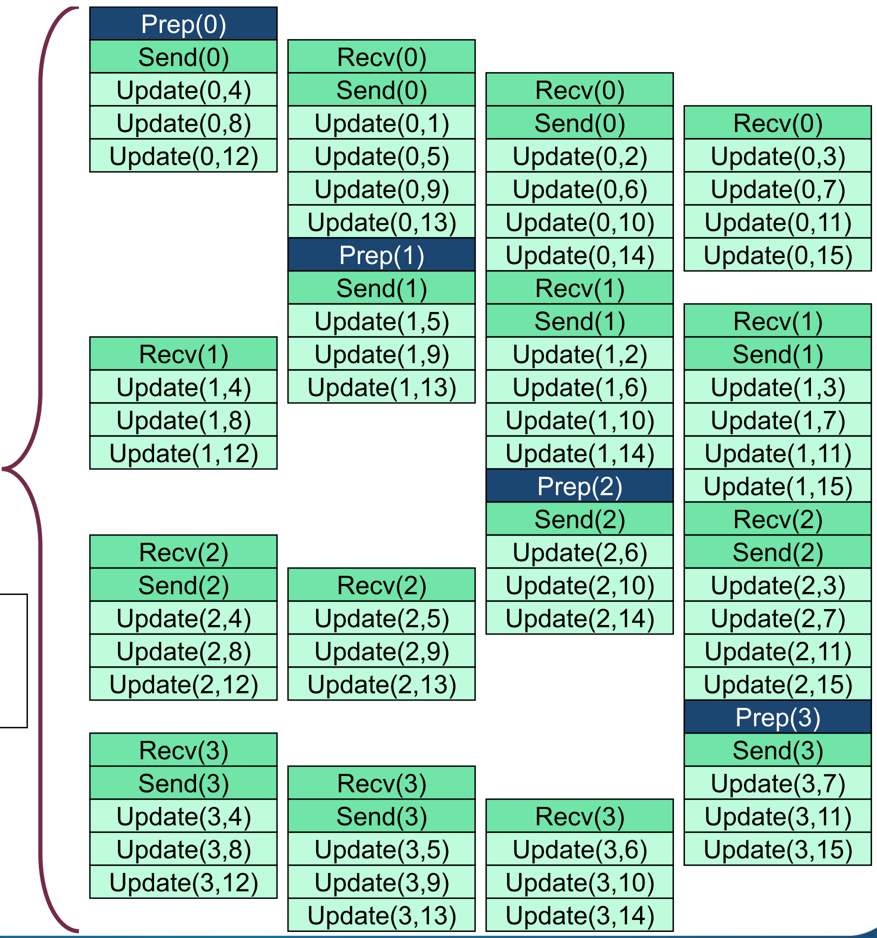
What’s better?

- During the broadcast, the successor of the root waits while the message is broadcast in the ring
  - With better distribution on a general topology, the wait would be shorter
  - But there would still be a wait
- With the pipeline algorithm, each processor goes on after receiving and forwarding the message
- Possible by writing the code only with send and receive
  - More complicated, more effective: **tradeoff**

A processor sends its data as soon as it receives it

First four steps

Many communications are performed in parallel with computations



Can we do even better?

- In the preceding algorithms, a processor performs all its updates before performing a `Prep()` computation which leads to a communication
- In fact, some of these updates can be done later
- **Idea:** send the 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)
…
```

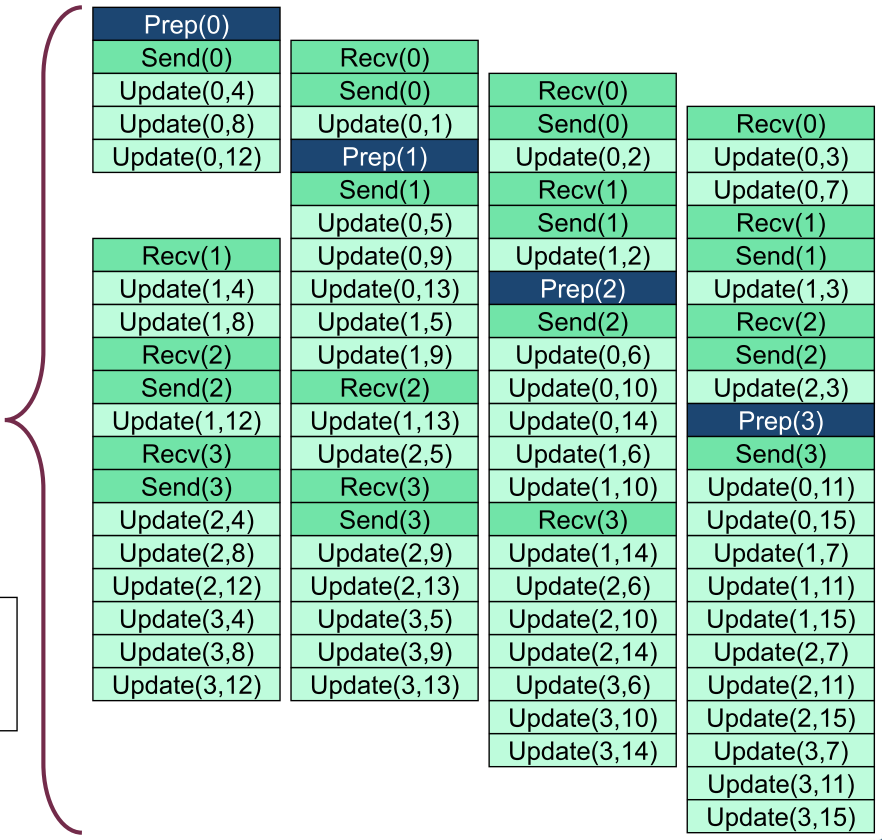
    - In the new algorithm

```
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 its data as soon as it receives them

First four steps

Many communications are performed in parallel with computations



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)	Prep(1)	Update(0,2)	Update(0,3)
	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)

Still increasing performance

- We can use communication/computation overlap
  - Multi-threading, non-blocking MPI implementation (effective), ...
  - Numerous other optimizations in the literature
    - Numerous research papers
    - Many libraries available

- This is a good example of an application for which you can reorganize operations to gain performance (find the right sequences of operations)
  - The general principle remains the same: send the data as soon as possible

Another application of “stencil” type

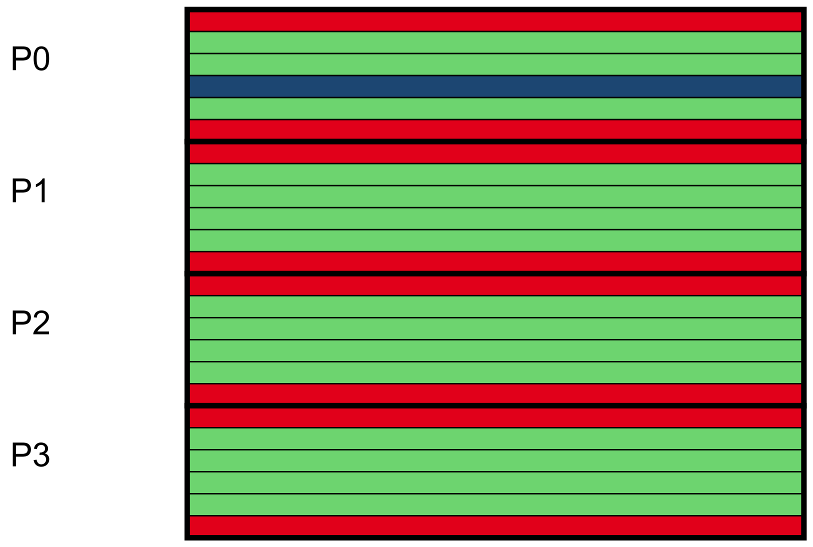
- Simple operation

  C_{new} = Update(C_{old}, W_{old}, E_{old}, N_{old}, S_{old})
- To implement this operation (in parallel or sequentially), two tables must be kept:
  - The original array: `A`
  - The new one: `B`
- To execute several iterations, we just exchange the pointers
- The simplest way to partition the data on the ring is to give a block of `n = N/p` consecutive lines to each processor

```
var A,B: array[0..r-1, 0..n-1] of real;
```

- Each processor can update lines `1..r-2` easily but for up and down lines, it is necessary to exchange elements with neighbors
- We will assume that even P_0 and P_{p-1} exchange rows

Another application of “stencil” type, contd.



Communication scheme (unidirectional ring)

- Complex exchange of rows:
  - Simple send to the successor
  - Send to the predecessor needs `p-1` steps
- The structure of the algorithm is as follows:
  - Send the borders to the neighbors ll compute the internal cells
  - Compute external cells
- It is assumed that each processor declares two buffer arrays

```
var fromPred, fromSucc: array[0..n-1] of real
```

- Each processor executes

```
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)
// Each processor has retrieved the values necessary for the
// computations
```

Performance analysis

- The communication phase consists in a sequence of p concurrent sends and receives of a row of n cells
  - It takes $pL + pnb$
- This runs concurrently with the compute phase on r-2 rows
  - It takes $(r-2)nw = (n/p - 2)nw$
- Then we have a calculation phase that computes 2 rows
  - It takes $2nw$
- The overall execution time is:

$$T(n,p) = \max\{pL + pnb, (n/p - 2)nw\} + 2nw$$

- When n becomes large: $T(n,p) \sim n^2w/p$

- We thus have a perfect asymptotic efficiency

Another virtual topology?

- The previous code is asymptotically optimal so normally we are good
- There is no way to reduce the overhead communications when n is not large
- Communication phase a bit complicated
- Use a bidirectional ring?
- Communication phase

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

- Simpler, more readable

Summary of optimizations

- **Aggregating communications**
  - Reduces the overhead of communications due to network latencies
- **Send data in advance**
  - Other processors start earlier, reduces waiting times
- **Overlapping communications and computations**
  - Hide the overhead of communications
- **Block distribution of the data**
  - Reduces the overhead of communications and may improve of the use of caches
- **Cyclic data distribution**
  - Better load-balancing between processors, reduces the waiting times

INVENTEURS DU MONDE NUMÉRIQUE