Principles of High Performance Computing (ICS 632)

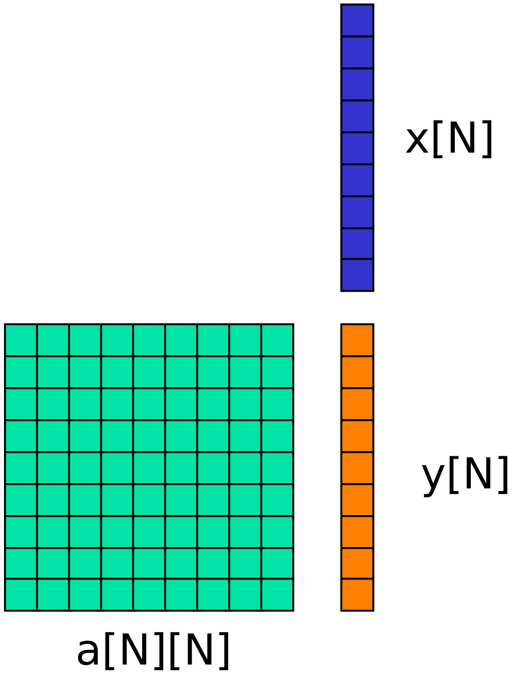
Algorithms on a Ring

Parallel Matrix-Vector product

- y = A x
- Let n be the size of the matrix

```
int a[n][n];
int x[n];
for i = 0 to n-1 {
  y[i] = 0;
  for j = 0 to n-1
    y[i] = y[i] + a[i,j] * x[j];
}
```

- How do we do this in parallel?

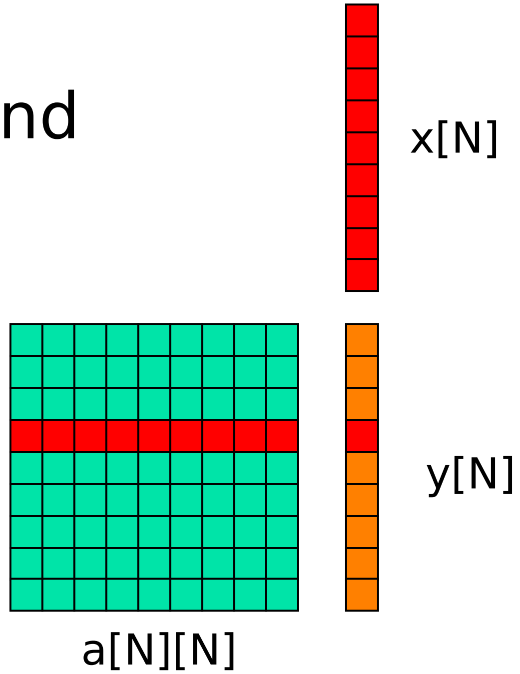


Section 4.1 in the book

Parallel Matrix-Vector product

- How do we do this in parallel?
- For example:
  - Computations of elements of vector y are independent
  - Each of these computations requires one row of matrix a and vector x
- In shared-memory:

```
#pragma omp parallel for private(i,j)
for i = 0 to n-1 {
    y[i] = 0;
    for j = 0 to n-1
        y[i] = y[i] + a[i,j] * x[j];
}
```



Parallel Matrix-Vector Product

- In distributed memory, one possibility is that each process has a full copy of matrix a and of vector x
- Each processor declares a vector y of size n/p
  - We assume that p divides n
- Therefore, the code can just be

```
load(a); load(x)
p = NUM_PROCS(); r = MY_RANK();
for (i=r*n/p; i<(r+1)*n/p; i++) {
      for (j=0;j<n;j++)
      y[i-r*n/p] = a[i][j] * x[j];
}
```

- It's embarrassingly parallel
- What about the result?

What about the result?

- After the processes complete the computation, each process has a piece of the result
- One probably wants to, say, write the result to a file
  - Requires synchronization so that the I/O is done correctly
- For example

```
 . . .
if (r != 0) {
      recv(&token,1);
}
open(file, "append");
for (j=0; j<n/p ; j++)
      write(file, y[j]);
send(&token,1);
close(file)
barrier();   // optional
```

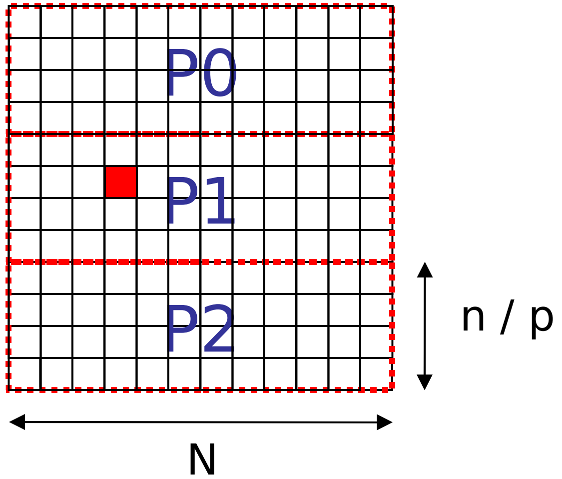
- Could also use a "gather" so that the entire vector is returned to processor 0
  - vector y fits in the memory of a single node

What if matrix a is too big?

- Matrix a may not fit in memory
  - Which is a motivation to use distributed memory implementations
- In this case, each processor can store only a piece of matrix a
- For the matrix-vector multiply, each processor can just store n/p rows of the matrix
  - Conceptually: A[n][n]
  - But the program declares a[n/p][n]
- This raises the (annoying) issue of global indices versus local indices

Global vs. Local indices

- When an array is split among processes
  - global index (I,J) that references an element of the matrix
  - local index (i,j) that references an element of the local array that stores a piece of the matrix
  - Translation between global and local indices
    - think of the algorithm in terms of global indices
    - implement it in terms of local indices



Global: A[5][3]
Local: a[1][3] on process P1

a[i,j] = A[(n/p)*rank + i][j]

Global Index Computation

- Real-world parallel code often implements actual translation functions
  - GlobalToLocal()
  - LocalToGlobal()
- This may be a good idea in your code, although for the ring topology the computation is pretty easy, and writing functions may be overkill
- We'll see more complex topologies with more complex associated data distributions and then it's probably better to implement such functions

Distributions of arrays

- At this point we have
  - 2-D array a distributed
  - 1-D array y distributed
  - 1-D array x replicated
- Having distributed arrays makes it possible to partition work among processes
  - But it makes the code more complex due to global/local indices translations
  - It may require synchronization to load/save the array elements to file

All vector distributed?

- So far we have array x replicated
- It is usual to try to have all arrays involved in the same computation be distributed in the same way
  - makes it easier to read the code without constantly keeping track of what's distributed and what's not
    - e.g., "local indices for array y are different from the global ones, but local indices for array x are the same as the global ones" will lead to bugs
- What one would like it for each process to have
  - N/n rows of matrix A in an array a[n/p][n]
  - N/n components of vector x in an array x[n/p]
  - N/n components of vector y in an array y[n/p]
- Turns out there is an elegant solution to do this

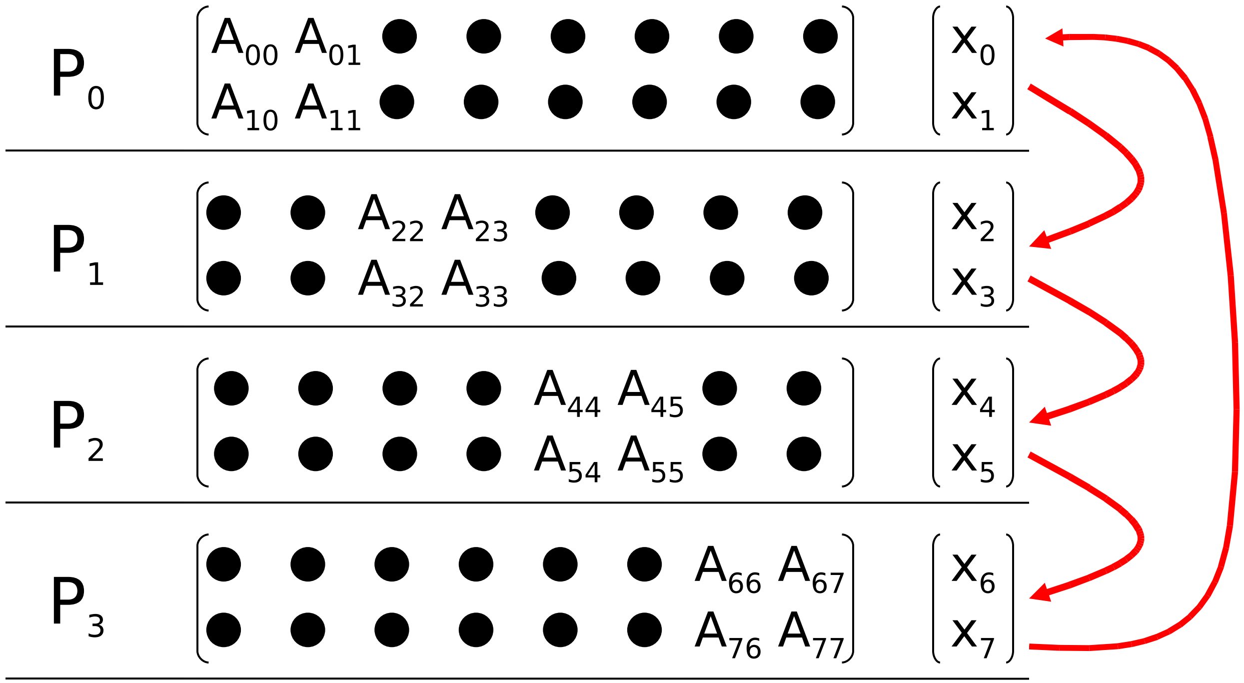
Principle of the Algorithm

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

Initial data distribution for:

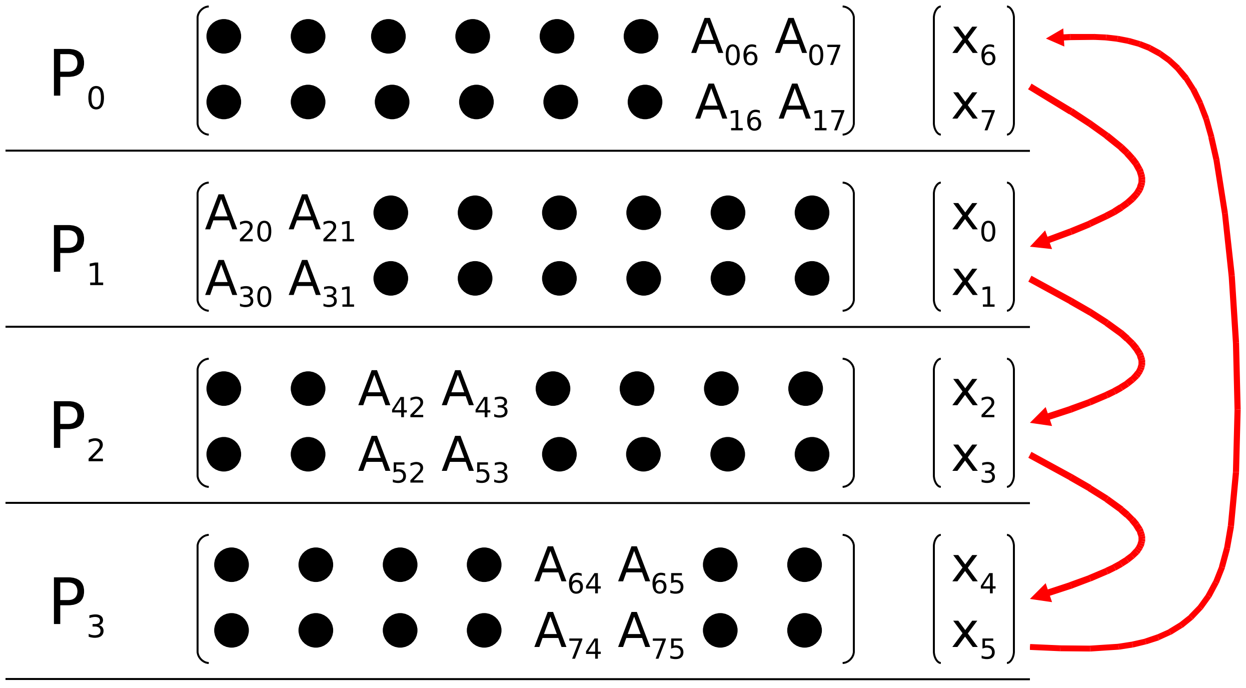
- $n = 8$
- $p = 4$
- $n/p = 2$

Principle of the Algorithm



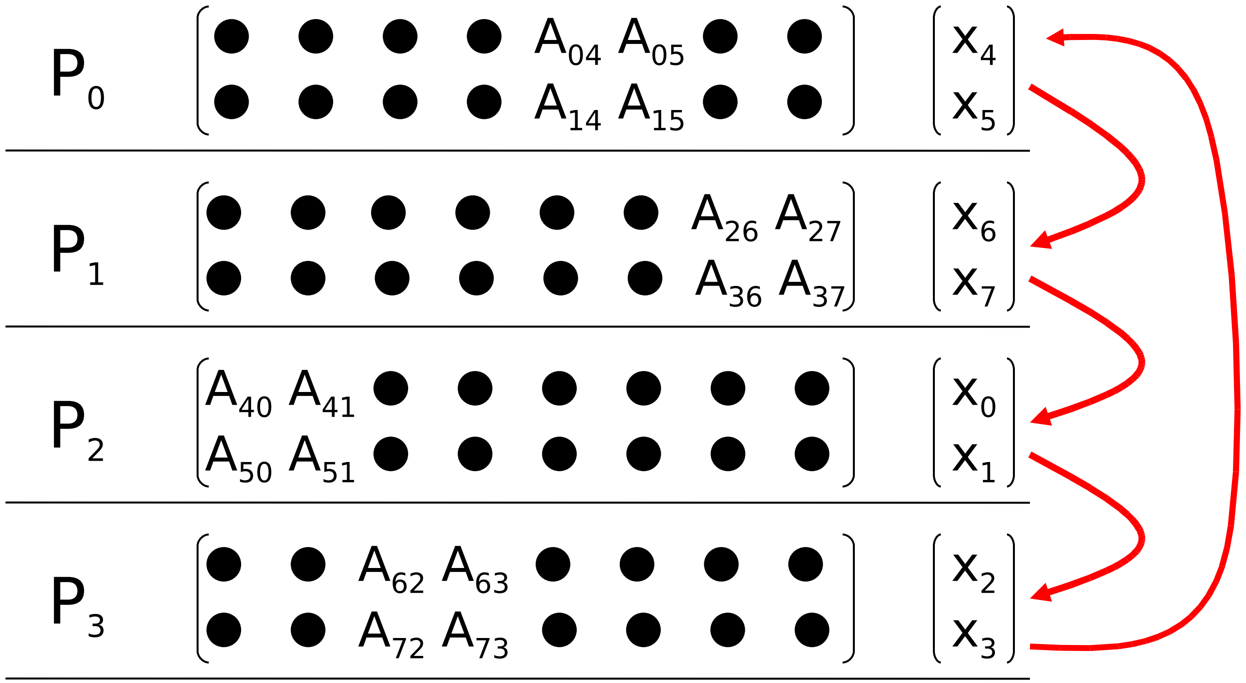
Step 0

Principle of the Algorithm



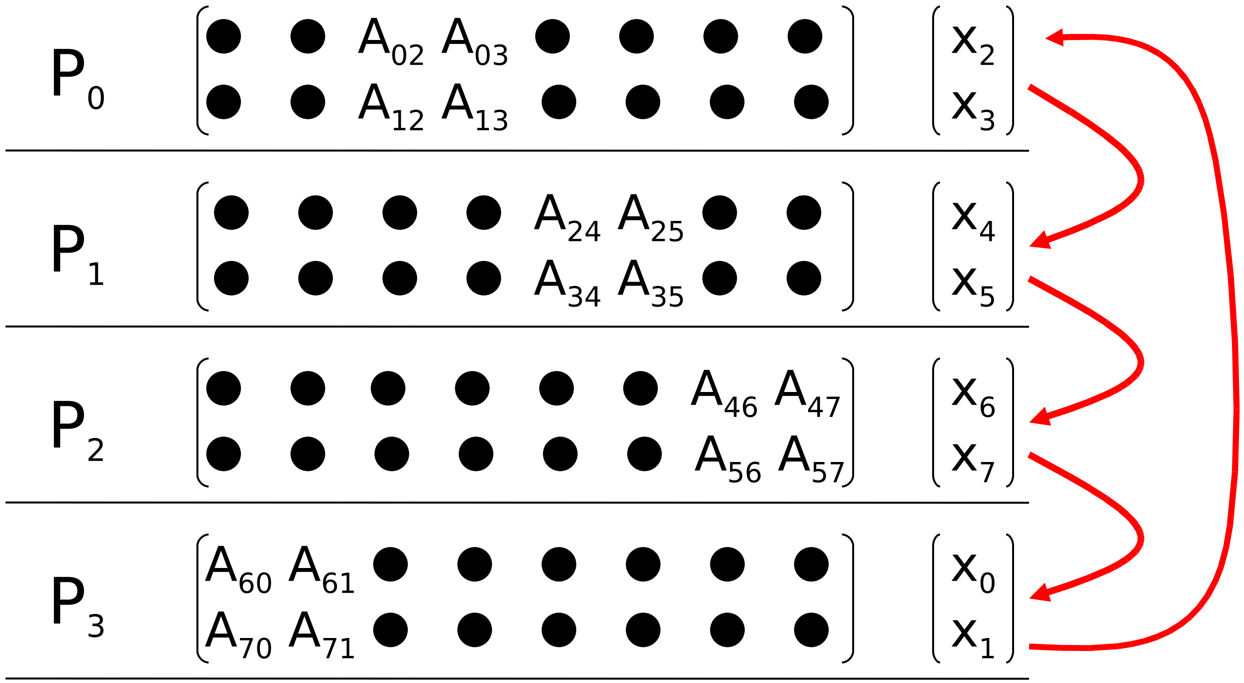
Step 1

Principle of the Algorithm



Step 2

Principle of the Algorithm



Step 3

Principle of the Algorithm

$$P_0 \quad \begin{bmatrix} A_{00} & A_{01} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ A_{10} & A_{11} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix} \quad \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

$$P_1 \quad \begin{bmatrix} \bullet & \bullet & A_{22} & A_{23} & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & A_{32} & A_{33} & \bullet & \bullet & \bullet & \bullet \end{bmatrix} \quad \begin{bmatrix} x_2 \\ x_3 \end{bmatrix}$$

$$P_2 \quad \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & A_{44} & A_{45} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & A_{54} & A_{55} & \bullet & \bullet \end{bmatrix} \quad \begin{bmatrix} x_4 \\ x_5 \end{bmatrix}$$

$$P_3 \quad \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{66} & A_{67} \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{76} & A_{77} \end{bmatrix} \quad \begin{bmatrix} x_6 \\ x_7 \end{bmatrix}$$

Final state

The final exchange of vector x is not strictly necessary, but one may want to have it distributed as the end of the computation like it was distributed at the beginning.

Algorithm

- Uses two buffers
  - tempS for sending and tempR to receiving

```
float A[n/p][n], x[n/p], y[n/p];
r ← n/p
tempS ← x   /* My piece of the vector (n/p elements) */
for (step=0; step<p; step++) {  /* p steps */
  SEND(tempS,r)
  RECV(tempR,r)
  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]
  tempS ↔ tempR
}
```

- In our example, process of rank 2 at step 3 would work with the 2x2 matrix block starting at column ((2 - 3) mod 4)*8/4 = 3 * 8 / 4 = 6;

A few General Principles

- Large data needs to be distributed among processes (running on different nodes of a cluster for instance)
  - causes many arithmetic expressions for index computation
  - People who do this for a leaving always end up writing local_to_global() and global_to_local() functions
- Data may need to be loaded/written before/after the computation
  - requires some type of synchronization among processes
- Typically a good idea to have all data structures distributed similarly to avoid confusion about which indices are global and which ones are local
  - In our case, all indices are local
- In the end the code looks much more complex than the equivalent OpenMP implementation

Performance

- There are p identical steps
- During each step each processor performs three activities: computation, receive, and sending
  - Computation: $r^2 w$
    - w: time to perform one += * operation
  - Receiving: L + r b
  - Sending: L + r b

$$T(p) = p\,(r^2 w + 2L + 2rb)$$

Asymptotic Performance

- $T(p) = p(r^2w + 2L + 2rb)$
  - $Speedup(p) = n^2w \; / \; p\,(r^2w + 2L + 2rb)$

    $= n^2w \; / \; (n^2w/p + 2pL + 2nb)$
  - $Eff(p) = n^2w \; / \; (n^2w + 2p^2L + 2pnb)$
  - For p fixed, when n is large, Eff(p) ~ 1

- Conclusion: the algorithm is asymptotically optimal

Performance (2)

- Note that an algorithm that initially broadcasts the entire vector to all processors and then have every processor compute independently would be in time

  $$(p-1)(L + n\,b) + pr^2 w$$

  - Could use the pipelined broadcast
- which:
  - has the same asymptotic performance
  - is a simpler algorithm
  - wastes only a tiny little bit of memory
  - is arguably much less elegant
- It is important to think of simple solutions and see what works best given expected matrix size, etc.

Back to the Algorithm

```
float A[n/p][n], x[n/p], y[n/p];
r ← n/p
tempS ← x   /* My piece of the vector (n/p elements) */
for (step=0; step<p; step++) {  /* p steps */
  SEND(tempS,r)
  RECV(tempR,r)
  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]
  tempS ↔ tempR
}
```

- In the above code, at each iteration, the SEND, the RECV, and the computation can all be done in parallel
- Therefore, one can overlap communication and computation by using non-blocking SEND and RECV if available
- MPI provides MPI_ISend() and MPI_IRecv() for this purpose

Nore Concurrent Algorithm

- Notation for concurrent activities:

```
float A[n/p][n], x[n/p], y[n/p];
tempS ← x   /* My piece of the vector (n/p elements) */
r ← n/p
for (step=0; step<p; step++) {  /* p steps */
     SEND(tempS,r)
  || RECV(tempR,r)
  || 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]
  tempS ↔ tempR
}
```

Better Performance

- There are p identical steps
- During each step each processor performs three activities: computation, receive, and sending
  - Computation: r^2w
  - Receiving: $L + rb$
  - Sending: $L + rb$

$$T(p) = p \, \max(r^2w, L + rb)$$

Same asymptotic performance as above, but better performance for smaller values of n

Hybrid parallelism

- We have said many times that multi-core architectures are about to become the standard
- When building a cluster, the nodes you will buy will be multi-core
- Question: how to exploit the multiple cores?
  - Or in our case how to exploit the multiple processors in each node
- Option #1: Run multiple processes per node
  - Causes more overhead and more communication
  - In fact will cause network communication among processes within a node!
    - MPI will not know that processes are co-located

OpenMP MPI Program

- Option #2: Run a single multi-threaded process per node
  - Much lower overhead, fast communication within a node
  - Done by combining MPI with OpenMP!
- Just write your MPI program
- Add OpenMP pragmas around loops
- Let’s look back at our Matrix-Vector multiplication example

Hybrid Parallelism

```
float A[n/p][n], x[n/p], y[n/p];
tempS ← x    /* My piece of the vector (n/p elements) */
for (step=0; step<p; step++) {  /* p steps */
     SEND(tempS,r)
   || RECV(tempR,r)
   || #pragma omp parallel for private(i,j)
     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]
  tempS ↔ tempR
}
```

- This is called Hybrid Parallelism
- Communication via the network among nodes
- Communication via the shared memory within nodes

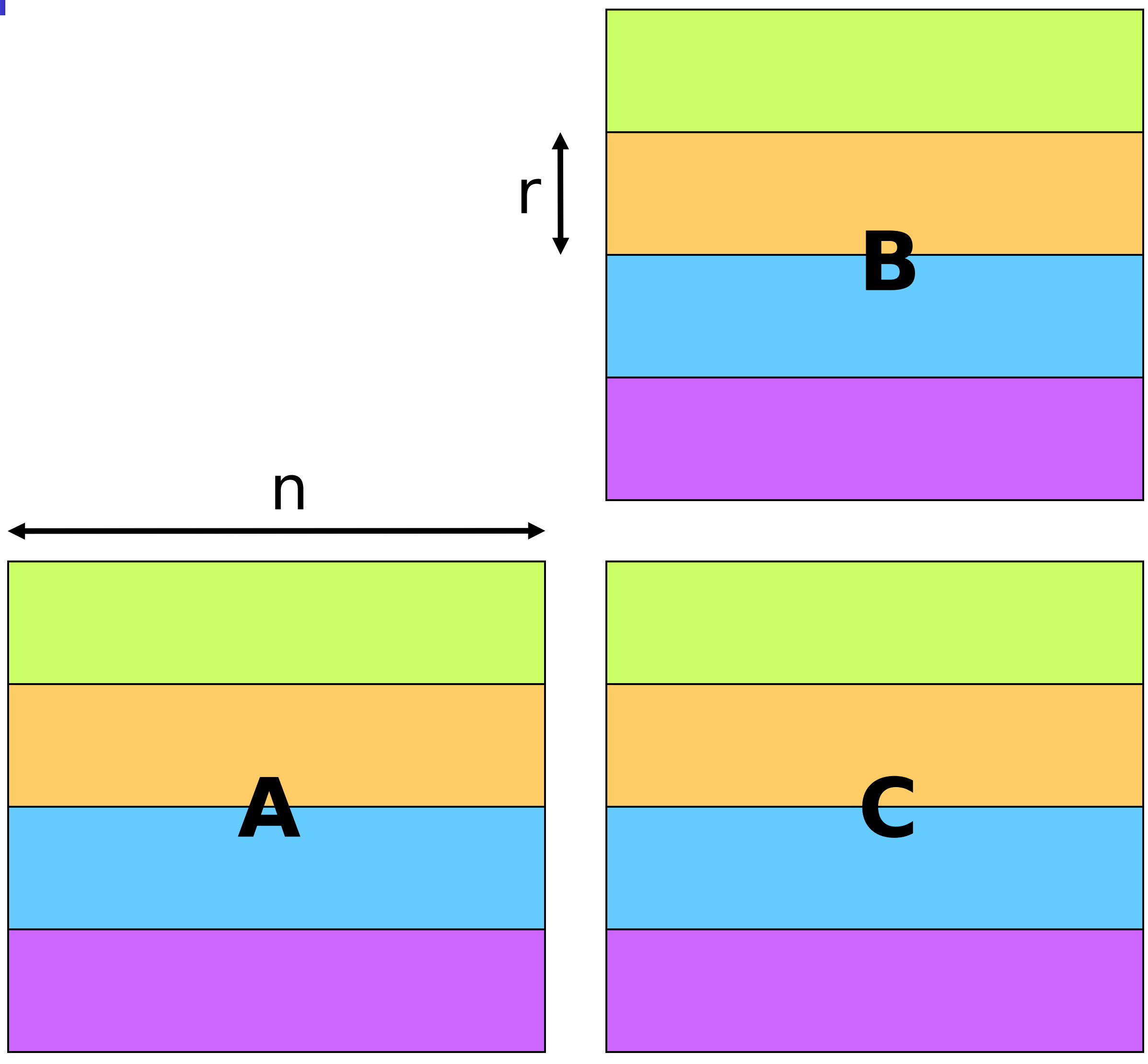
Getting it Compiled and Linked

- It can be tricky to compile and link a hybrid program
  - Because mpicc and ompcc do their own things to make our lives simple, they don’t play well with each other
- My solution: use any gcc after 4.2
- The cluster has gcc 3.4 installed by default
  - Because the cluster is managed using a software that rolls out particular RedHat distributions, and so far, we’re stuck with this
- BUT, any gcc after 4.2 supports openMP:
  - gcc whatever.c -o whatever -fopenmp
  - We could’ve used it for HW #1
- So I installed gcc 4.2 in **/home/casanova/public/bin/gcc**
- Compiling with mpicc is however no longer possible
- So I put an example Makefile in **/home/casanova/public/Makefile.hybrid**
  - Let’s look at it…

Matrix Multiplication on the Ring

- See Section 4.2
- Turns out one can do matrix multiplication in a way very similar to matrix-vector multiplication
  - A matrix multiplication is just the computation of n^2 scalar products, not just n
- We have three matrices, A, B, and C
- We want to compute C = A*B
- We distribute the matrices to that each processor “owns” a block row of each matrix
  - Easy to do if row-major is used because all matrix elements owned by a processor are contiguous in memory

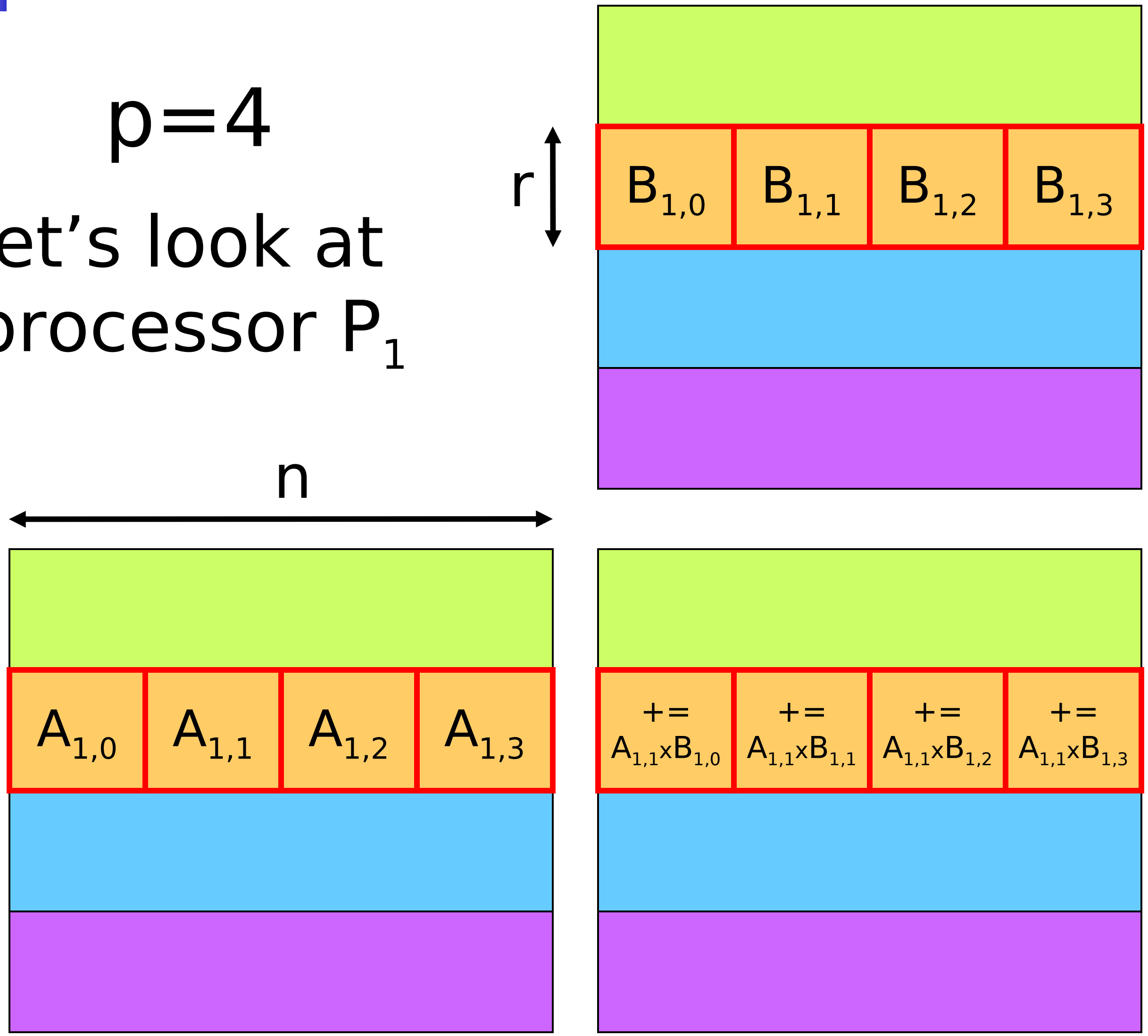
Data Distribution



First Step

p=4

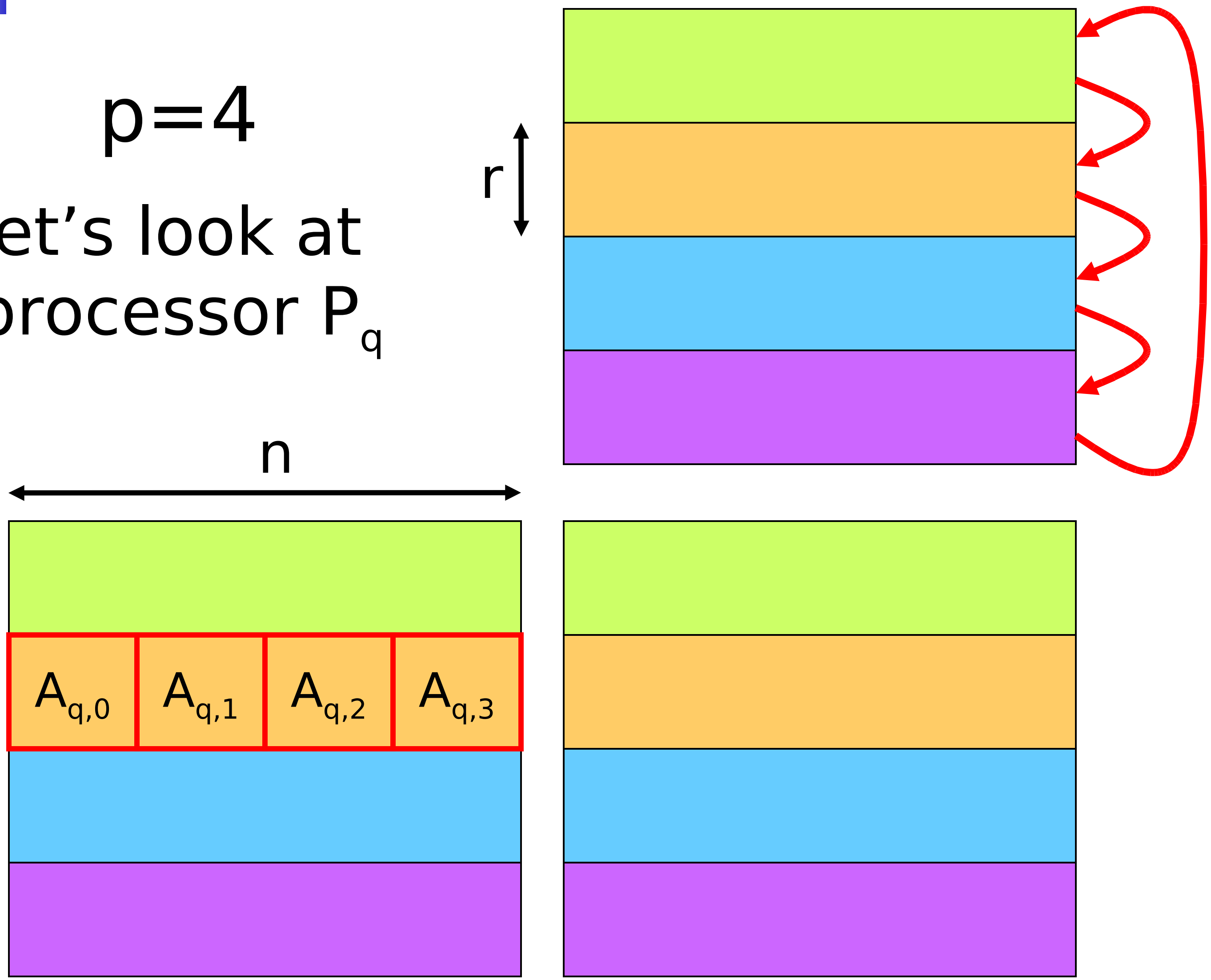
let's look at processor P_1



Shifting of block rows of B

p=4

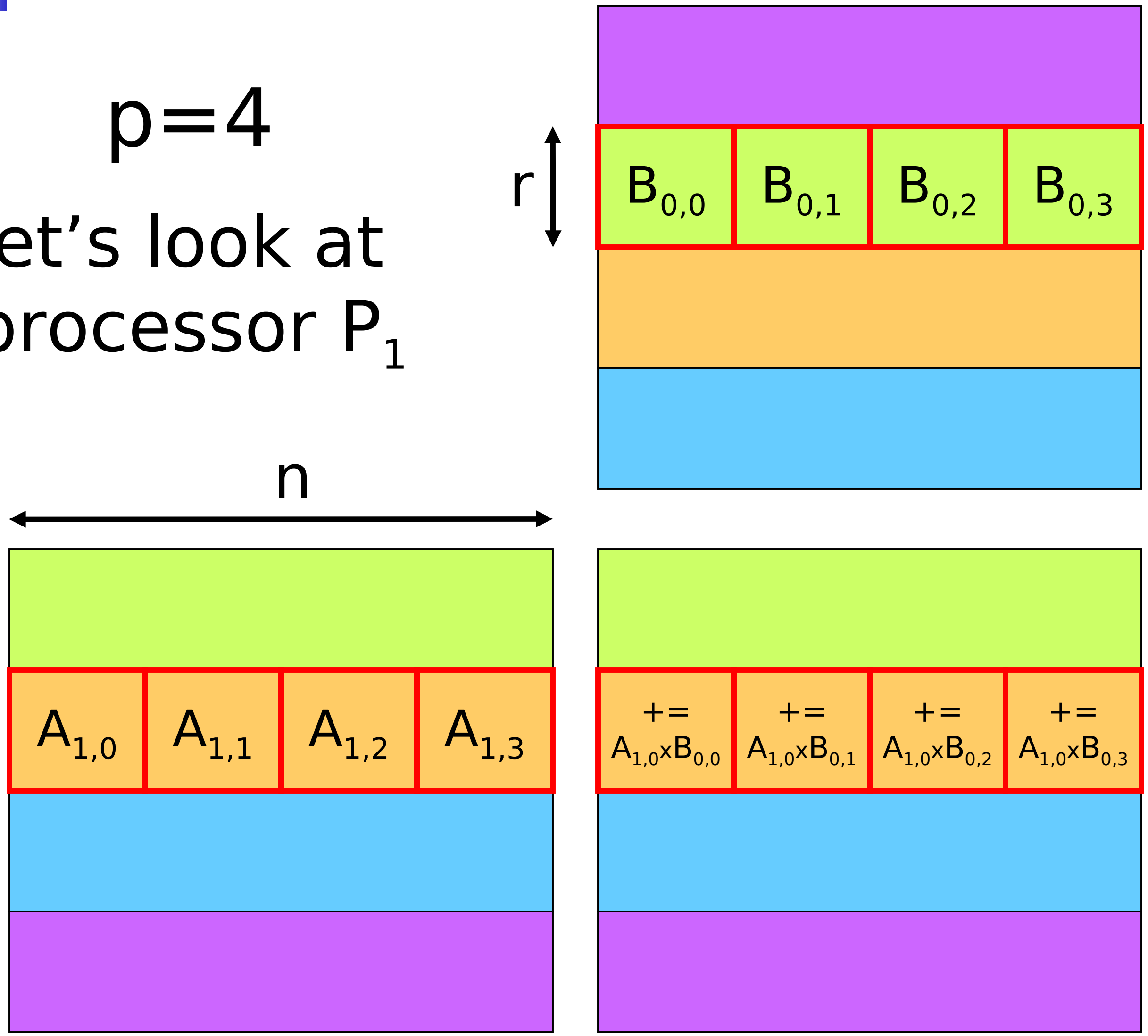
let's look at processor P_q



Second step

p=4

let's look at processor P_1



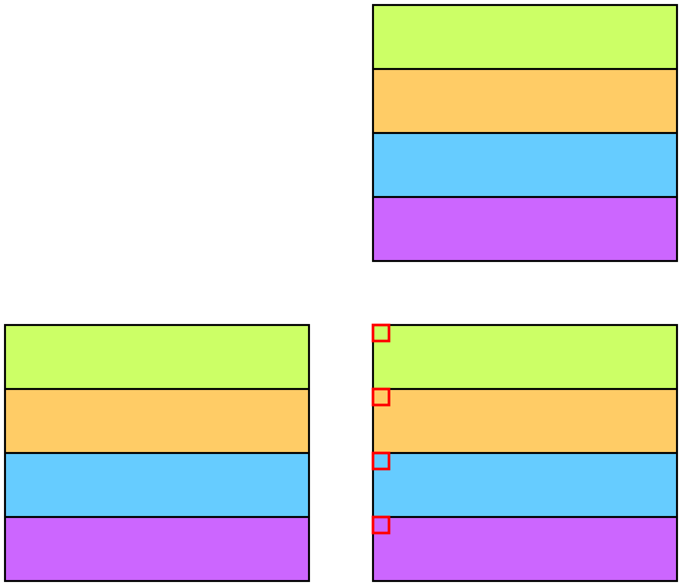
Algorithm

- In the end, every Ci,j block has the correct value: $A_{i,0}B_{0,j} + A_{i,1}B_{1,j} + \ldots$
- Basically, this is the same algorithm as for matrix-vector multiplication, replacing the partial scalar products by submatrix products (gets tricky with loops and indices)

```
float A[N/p][N], B[N/p][N], C[N/p][N];
r ← N/p
tempS ← B
q ← MY_RANK()
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,l*r+j] ← C[i,l*r+j] + A[i,r((q - step)%p)+k] * tempS[k,l*r+j]
  tempS ↔ tempR
}
```

Algorithm

```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,lr+j] ← C[i,lr+j] + A[i,r((rank - step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```



step=0

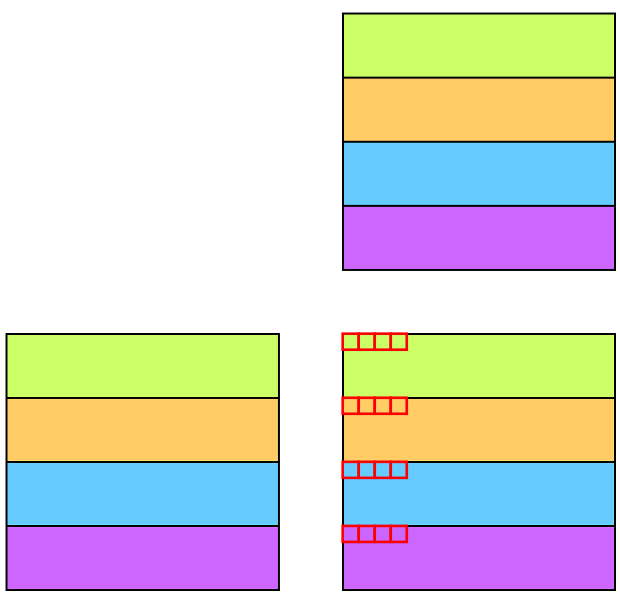
l=0

i=0

j=0

Algorithm

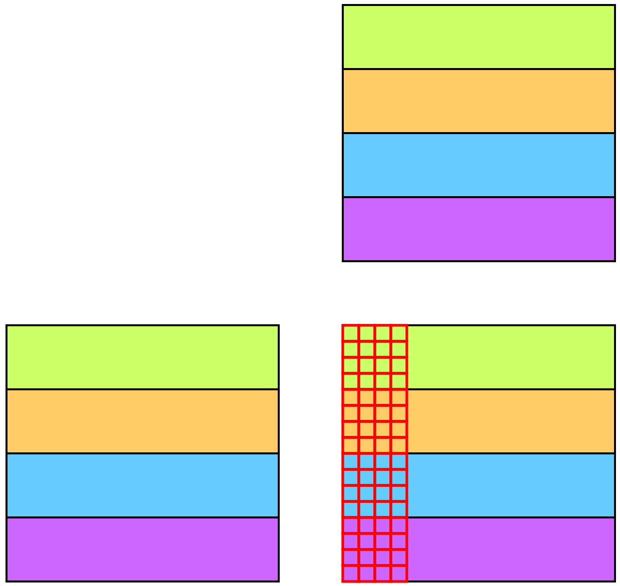
```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,lr+j] ← C[i,lr+j] + A[i,r((rank - step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```



step=0
l=0
i=0
j=*

Algorithm

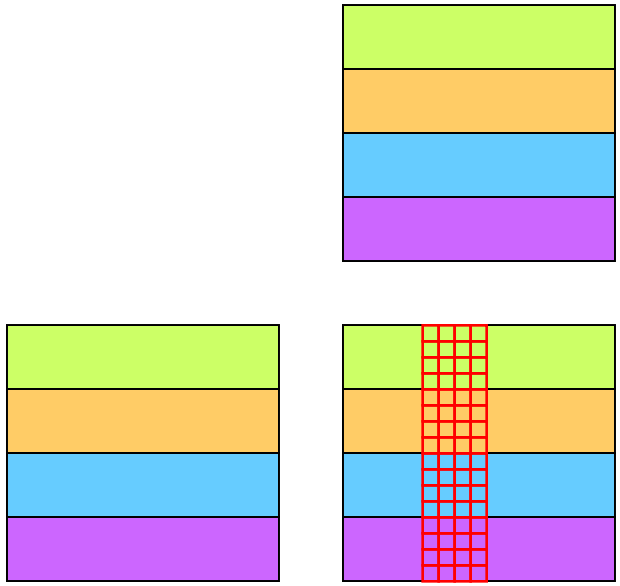
```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,lr+j] ← C[i,lr+j] + A[i,r((rank - step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```



step=0
l=0
i=*
j=*

Algorithm

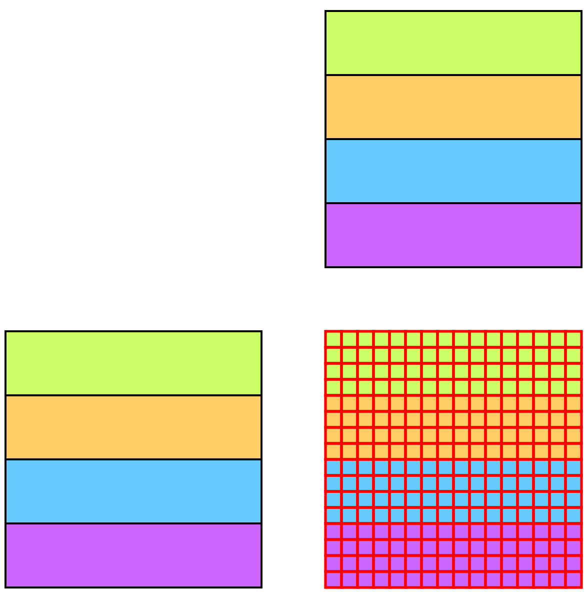
```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,lr+j] ← C[i,lr+j] + A[i,r((rank - step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```



step=0
l=1
i=*
j=*

Algorithm

```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,lr+j] ← C[i,lr+j] + A[i,r((rank - step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```



step=0

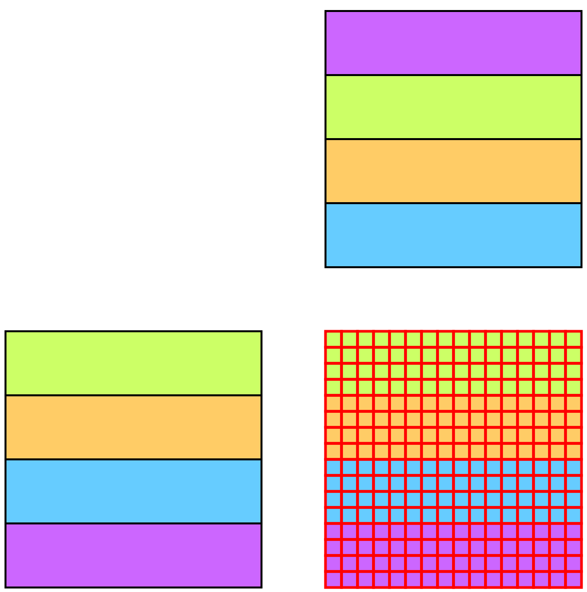
l=*

i=*

j=*

Algorithm

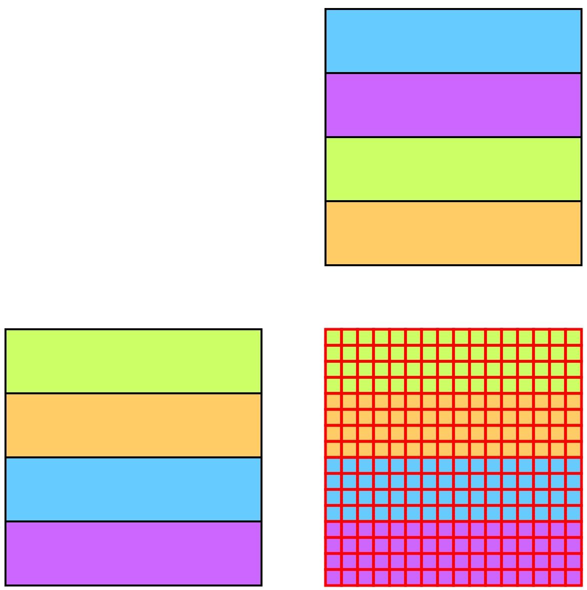
```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,lr+j] ← C[i,lr+j] + A[i,r((rank - step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```



step=1
l=*
i=*
j=*

Algorithm

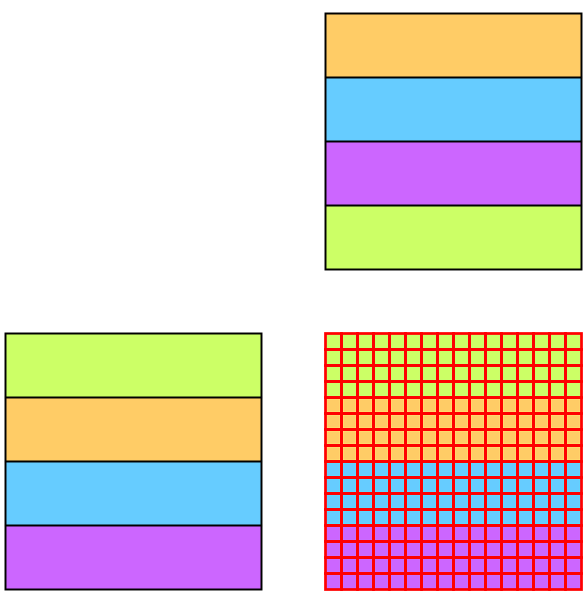
```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,lr+j] ← C[i,lr+j] + A[i,r((rank - step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```



step=2
l=*
i=*
j=*

Algorithm

```
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N)
 || RECV(tempR,r*N)
 || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
      for (j=0; j<N/p; j++)
       for (k=0; k<N/p; k++)
         C[i,lr+j] ← C[i,lr+j] + A[i,r((rank - step)%p)+k] * tempS[k,lr+j]
  tempS ↔ tempR
}
```



step=3

l=*

i=*

j=*

Performance

- Performance Analysis is straightforward
- p steps and each step takes time:

  max $(nr^2 w, L + nrb)$

  - p r x r matrix products = $pr^3 = nr^2$ operations
- Hence, the running time is:

  $T(p) = p \max(nr^2 w, L + nrb)$
- Note that a naive algorithm computing n Matrix-vector products in sequence using our previous algorithm would take time

  $T(p) = p \max(nr^2 w, \mathbf{n}L + nrb)$
- We just saved network latencies!

Conclusion

- This was our first foray in the realm of distributed memory parallel algorithms
- In a programming assignment you'll write things like these in MPI and see what happens
- In the next set of slides we'll look at more complex algorithms that involve interesting performance trade-offs