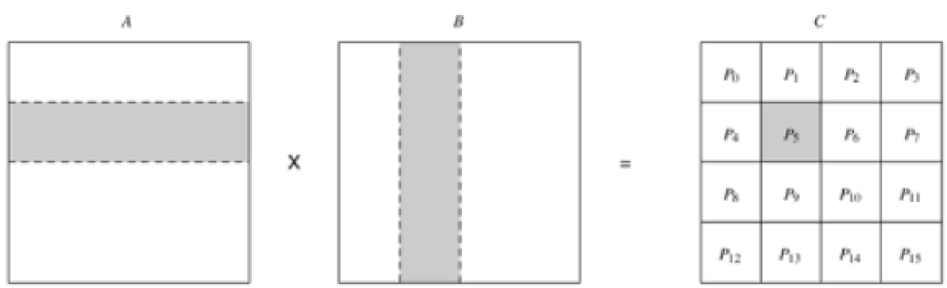
Lecture 5: Parallel Matrix Algorithms (part 3)

A Simple Parallel Dense Matrix-Matrix Multiplication

Let $A = [a_{ij}]_{n\times n}$ and $B = [b_{ij}]_{n\times n}$ be $\mathrm{n} \times \mathrm{n}$ matrices. Compute $C = AB$

- Computational complexity of sequential algorithm: $O(n^3)$
- Partition A and B into p square blocks $A_{i,j}$ and $B_{i,j}$ $(0 \le i, j < \sqrt{p})$ of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each.
- Use Cartesian topology to set up process grid. Process $P_{i,j}$ initially stores $A_{i,j}$ and $B_{i,j}$ and computes block $C_{i,j}$ of the result matrix.
- Remark: Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $0 \le k < \sqrt{p}$.



- Algorithm:
  - Perform all-to-all broadcast of blocks of A in each row of processes
  - Perform all-to-all broadcast of blocks of B in each column of processes
  - Each process $P_{i,j}$ perform $C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} B_{k,j}$

Performance Analysis

- $\sqrt{p}$ rows of all-to-all broadcasts, each is among a group of $\sqrt{p}$ processes. A message size is $\frac{n^2}{p}$, communication time: $t_s log\sqrt{p} + t_w \frac{n^2}{p}\left(\sqrt{p}-1\right)$
- $\sqrt{p}$ columns of all-to-all broadcasts, communication time: $t_s log\sqrt{p} + t_w \frac{n^2}{p}\left(\sqrt{p}-1\right)$
- Computation time: $\sqrt{p} \times (n/\sqrt{p})^3 = n^3/p$
- Parallel time: $T_p = \frac{n^3}{p} + 2\left(t_s log\sqrt{p} + t_w \frac{n^2}{p}\left(\sqrt{p}-1\right)\right)$

Memory Efficiency of the Simple Parallel Algorithm

- Not memory efficient
  - Each process $P_{i,j}$ has $2\sqrt{p}$ blocks of $A_{i,k}$ and $B_{k,j}$
  - Each process needs $\Theta(n^2/\sqrt{p})$ memory
  - Total memory over all the processes is $\Theta(n^2 \times \sqrt{p})$, i.e., $\sqrt{p}$ times the memory of the sequential algorithm.

Cannon's Algorithm of Matrix-Matrix Multiplication

Goal: to improve the memory efficiency.

Let $A = [a_{ij}]_{n \times n}$ and $B = [b_{ij}]_{n \times n}$ be $\mathrm{n \times n}$ matrices. Compute $C = AB$

- Partition A and B into p square blocks $A_{i,j}$ and $B_{i,j}$ $(0 \leq i, j < \sqrt{p})$ of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each.
- Use Cartesian topology to set up process grid. Process $P_{i,j}$ initially stores $A_{i,j}$ and $B_{i,j}$ and computes block $C_{i,j}$ of the result matrix.
- Remark: Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $0 \leq k < \sqrt{p}$.
- **The contention-free formula**:

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k)\%\sqrt{p}} B_{(i+j+k)\%\sqrt{p},j}$$

Cannon's Algorithm

// make initial alignment

for i, j :=0 **to** $\sqrt{p} - 1$ **do**

Send block $A_{i,j}$ to process $\left(i, \left(j - i + \sqrt{p}\right) mod \sqrt{p}\right)$ and block $B_{i,j}$ to process $\left(\left(i - j + \sqrt{p}\right) mod \sqrt{p}, j\right)$;

endfor;

Process $P_{i,j}$ multiply received submatrices together and add the result to $C_{i,j}$;

// compute-and-shift. A sequence of one-step shifts pairs up $A_{i,k}$ and $B_{k,j}$

// on process $P_{i,j}$. $C_{i,j} = C_{i,j} + A_{i,k} B_{k,j}$

for step :=1 **to** $\sqrt{p} - 1$ **do**

Shift $A_{i,j}$ one step left (with wraparound) and $B_{i,j}$ one step up (with wraparound);

Process $P_{i,j}$ multiply received submatrices together and add the result to $C_{i,j}$;

Endfor;

Remark: In the initial alignment, the send operation is to: shift $A_{i,j}$ to the left (with wraparound) by i steps, and shift $B_{i,j}$ to the up (with wraparound) by j steps.

Cannon's Algorithm for 3 × 3 Matrices

A(0,0)	A(0,1)	A(0,2)
A(1,0)	A(1,1)	A(1,2)
A(2,0)	A(2,1)	A(2,2)

B(0,0)	B(0,1)	B(0,2)
B(1,0)	B(1,1)	B(1,2)
B(2,0)	B(2,1)	B(2,2)

Initial A, B

A(0,0)	A(0,1)	A(0,2)
A(1,1)	A(1,2)	A(1,0)
A(2,2)	A(2,0)	A(2,1)

B(0,0)	B(1,1)	B(2,2)
B(1,0)	B(2,1)	B(0,2)
B(2,0)	B(0,1)	B(1,2)

A, B initial alignment

A(0,1)	A(0,2)	A(0,0)
A(1,2)	A(1,0)	A(1,1)
A(2,0)	A(2,1)	A(2,2)

B(1,0)	B(2,1)	B(0,2)
B(2,0)	B(0,1)	B(1,2)
B(0,0)	B(1,1)	B(2,2)

A, B after shift step 1

A(0,2)	A(0,0)	A(0,1)
A(1,0)	A(1,1)	A(1,2)
A(2,1)	A(2,2)	A(2,0)

B(2,0)	B(0,1)	B(1,2)
B(0,0)	B(1,1)	B(2,2)
B(1,0)	B(2,1)	B(0,2)

A, B after shift step 2

Performance Analysis

- In the initial alignment step, the maximum distance over which block shifts is $\sqrt{p}\ -1$
  - The circular shift operations in row and column directions take time: $t_{comm} = 2(t_s + \frac{t_w n^2}{p})$
- Each of the $\sqrt{p}$ single-step shifts in the compute-and-shift phase takes time: $t_s + \frac{t_w n^2}{p}$.
- Multiplying $\sqrt{p}$ submatrices of size $(\frac{\mathrm{n}}{\sqrt{p}}) \times (\frac{\mathrm{n}}{\sqrt{p}})$ takes time: n^3/p.
- Parallel time: $T_p = \frac{n^3}{p} + 2\sqrt{p}\left(t_s + \frac{t_w n^2}{p}\right) + 2(t_s + \frac{t_w n^2}{p})$

int MPI_Sendrecv_replace(void **buf*, **int** *count*, **MPI_Datatype** *datatype*, **int** *dest*, **int** *sendtag*, **int** *source*, **int** *recvtag*, **MPI_Comm** *comm*, **MPI_Status** **status* **);**

- Execute a blocking send and receive. The same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received.
- *buf*[in/out]: initial address of send and receive buffer

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])
{
   int    myid, numprocs, left, right;
   int    buffer[10];
   MPI_Request request;
   MPI_Status status;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);

   right = (myid + 1) % numprocs;
   left = myid - 1;
   if (left < 0)
      left = numprocs - 1;

   MPI_Sendrecv_replace(buffer, 10, MPI_INT, left, 123, right, 123, MPI_COMM_WORLD,
&status);

   MPI_Finalize();
   return 0;
}
```

DNS Algorithm

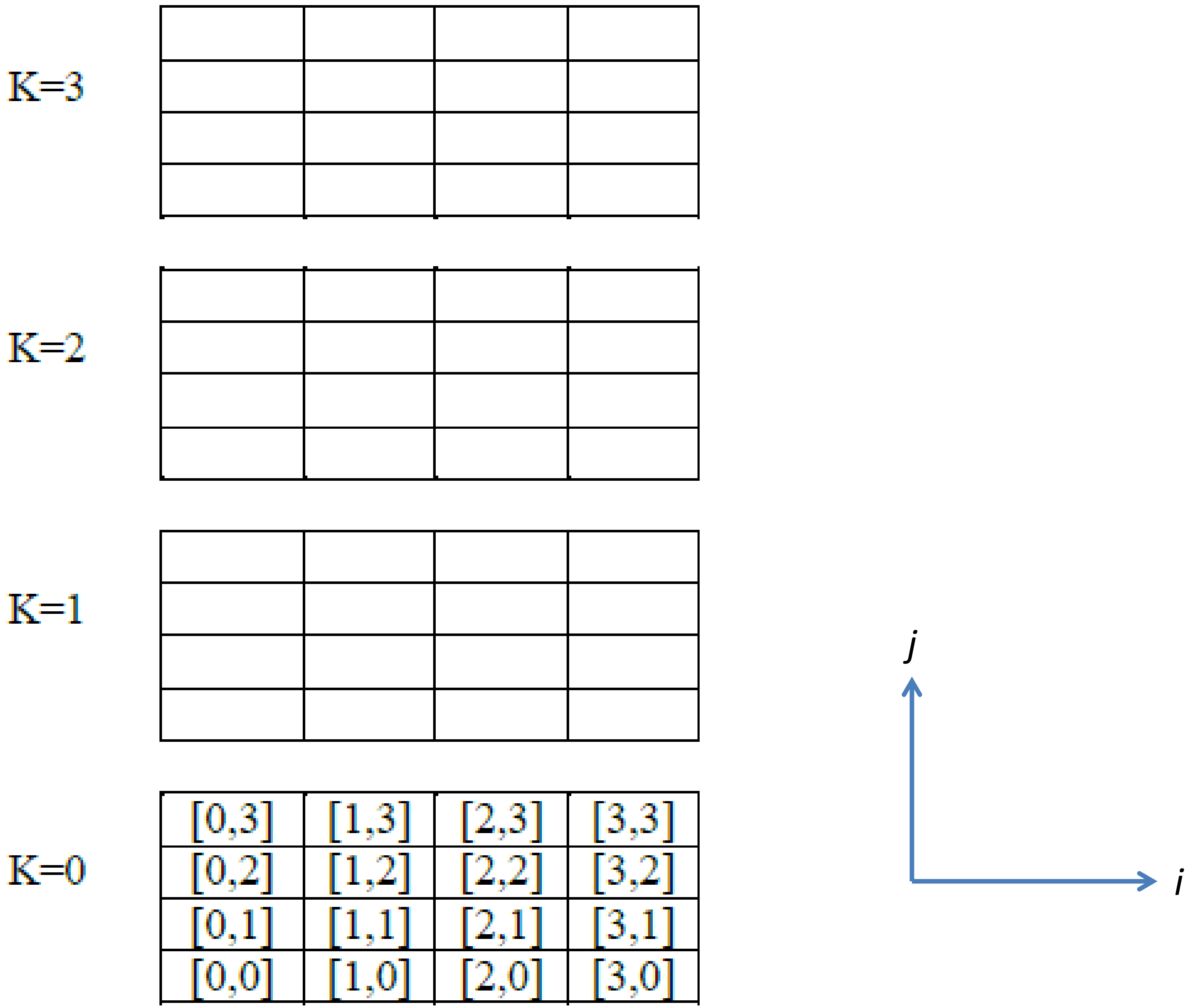
- The algorithm is named after Dekel, Nassimi and Aahni
- It is based on partitioning intermediate data
- It performs matrix multiplication in time $O(\log n)$ by using $O(n^3/\log n)$ processes

The sequential algorithm for $C = A \times B$

$$C_{ij} = 0$$
$$for(i = 0; i < n; i++)$$
$$for(j = 0; j < n; j++)$$
$$for(k = 0; k < n; k++)$$
$$C_{ij} = C_{ij} + A_{ik} \times B_{kj}$$

Remark: The algorithm performs n^3 scalar multiplications

- Assume that n^3 processes are available for multiplying two $n \times n$ matrices.
- Then each of the n^3 processes is assigned a single scalar multiplication.
- The additions for all C_{ij} can be carried out simultaneously in $\log n$ steps each.
- Arrange n^3 processes in a three-dimensional $n \times n \times n$ logical array.
  - The processes are labeled according to their location in the array, and the multiplication $A_{ik}B_{kj}$ is assigned to process P[i,j,k] ($0 \leq i, j, k < n$).
  - After each process performs a single multiplication, the contents of P[i,j,0],P[i,j,1],…,P[i,j,n-1] are added to obtain C_{ij}.



(a) Initial distribution of A and B

A[0,3]	A[1,3]	A[2,3]	A[3,3]

A[0,2]	A[1,2]	A[2,2]	A[3,2]

A[0,1]	A[1,1]	A[2,1]	A[3,1]

A[0,0]	A[1,0]	A[2,0]	A[3,0]

(b) After moving A[i,j] from P[i,j0] to P[i,j,j]

K=3

			B[3,3]
			B[3,2]
			B[3,1]
			B[3,0]

K=2

		B[2,3]	
		B[2,2]	
		B[2,1]	
		B[2,0]	

K=1

	B[1,3]		
	B[1,2]		
	B[1,1]		
	B[1,0]		

K=0

B[0,3]			
B[0,2]			
B[0,1]			
B[0,0]			

(b) After moving B[i,j] from P[i,j,0] to P[i,j,i]

	A[0,3]	A[1,3]	A[2,3]	A[3,3]	C[0,0]	B[3,3]	B[3,3]	B[3,3]	B[3,3]
K=3	A[0,3]	A[1,3]	A[2,3]	A[3,3]	=	B[3,2]	B[3,2]	B[3,2]	B[3,2]
	A[0,3]	A[1,3]	A[2,3]	A[3,3]	A[0,3] × B[3,0]	B[3,1]	B[3,1]	B[3,1]	B[3,1]
	A[0,3]	A[1,3]	A[2,3]	A[3,3]		B[3,0]	B[3,0]	B[3,0]	B[3,0]

	A[0,2]	A[1,2]	A[2,2]	A[3,2]		B[2,3]	B[2,3]	B[2,3]	B[2,3]
K=2	A[0,2]	A[1,2]	A[2,2]	A[3,2]	+	B[2,2]	B[2,2]	B[2,2]	B[2,2]
	A[0,2]	A[1,2]	A[2,2]	A[3,2]	A[0,2] × B[2,0]	B[2,1]	B[2,1]	B[2,1]	B[2,1]
	A[0,2]	A[1,2]	A[2,2]	A[3,2]		B[2,0]	B[2,0]	B[2,0]	B[2,0]

	A[0,1]	A[1,1]	A[2,1]	A[3,1]		B[1,3]	B[1,3]	B[1,3]	B[1,3]
K=1	A[0,1]	A[1,1]	A[2,1]	A[3,1]	+	B[1,2]	B[1,2]	B[1,2]	B[1,2]
	A[0,1]	A[1,1]	A[2,1]	A[3,1]	A[0,1] × B[1,0]	B[1,1]	B[1,1]	B[1,1]	B[1,1]
	A[0,1]	A[1,1]	A[2,1]	A[3,1]		B[1,0]	B[1,0]	B[1,0]	B[1,0]

	A[0,0]	A[1,0]	A[2,0]	A[3,0]		B[0,3]	B[0,3]	B[0,3]	B[0,3]
K=0	A[0,0]	A[1,0]	A[2,0]	A[3,0]	+	B[0,2]	B[0,2]	B[0,2]	B[0,2]
	A[0,0]	A[1,0]	A[2,0]	A[3,0]	A[0,0] × B[0,0]	B[0,1]	B[0,1]	B[0,1]	B[0,1]
	A[0,0]	A[1,0]	A[2,0]	A[3,0]		B[0,0]	B[0,0]	B[0,0]	B[0,0]

(c) After broadcasting **A[i,j]** along j axis

(d) Corresponding distribution of **B**

- The vertical column of processes P[i,j,*] computes the dot product of row A_{i*} and column B_{*j}

- The DNS algorithm has three main communication steps:
    1. moving the rows of A and the columns of B to their respective places,
    2. performing one-to-all broadcast along the j axis for A and along the i axis for B
    3. all-to-one reduction along the k axis