Cannon Algorithm

Problem Statement:
Let A = [aij] and B = [bij] be n × n matrices. Compute C = AB

- Cannon’s algorithm shifts data along rows and columns of processors
- Three matrices (A, B, C) are allocated memory location. The sizes of the matrices are assumed to be N x N
- AB = C
- A and B are given random values
- C is initially called with all the values 0, using calloc function
- Assume p processors are used for multiplication.
- The matrices A and B are divided into blocks of size n x n = [N $\sqrt{p}$] × [N $\sqrt{p}$], where n is the size of the block matrices
- In the first shifting, the blocks of first matrix shift left based on the number of column they reside in. And the blocks of second matrix shift upwards based on the number of row they reside in (as shown in the table below – INITIAL SHIFTING)
- Each processor forms the product of the two local matrices adding into the accumulated sum
- Then we do the circular shifting, depending on the number of processors.
- Product of the two local matrices is added to the accumulated sum after very shifting

e.g. : C[1,2] = A[1,0]*B[0,2] + A[1,1]*B[1,2] + A[1,2]*B[2,2]

ALIGNMENT	A[0,0]	A[0,1]	A[0,2]		B[0,0]	B[0,1]	B[0,2]
	A[1,0]	A[1,1]	A[1,2]		B[1,0]	B[1,1]	B[1,2]
	A[2,0]	A[2,1]	A[2,2]		B[2,0]	B[2,1]	B[2,2]

INITIAL SHIFTING							
A[1,0] and B[0,2] on the same processor	A[0,0]	A[0,1]	A[0,2]		B[0,0]	B[1,1]	B[2,2]
	A[1,1]	A[1,2]	A[1,0]		B[1,0]	B[2,1]	B[0,2]
	A[2,0]	A[2,1]	A[2,2]		B[2,0]	B[0,1]	B[1,2]
NEXT COMES THE CIRCULAR SHIFTING							
A[1,1] and B[1,2] on the same processor	A[0,1]	A[0,2]	A[0,0]		B[1,0]	B[2,1]	B[0,2]
	A[1,2]	A[1,0]	A[1,1]		B[2,0]	B[0,1]	B[1,2]
	A[2,1]	A[2,2]	A[2,0]		B[0,0]	B[1,1]	B[2,2]
Takes √p steps							

A[1,2] and B[2,2] on the same processor	A[0,2]	A[0,0]	A[0,1]
	A[1,0]	A[1,1]	A[1,2]
	A[2,2]	A[2,0]	A[2,1]

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]

<u>For communication between processors: The following commands are used</u>

```
MPI_Init(&argc, &argv); //pointers to argc and argv
MPI_Comm_size(MPI_COMM_WORLD, &nprocs); //number of processors in MPI_COMM_WORLD
MPI_Comm_rank(MPI_COMM_WORLD, &myid); //id of the processor in MPI_COMM_WORLD
MPI_Comm comm;
```

MPI_Sendrecv : sends and receives a message

```
int MPI_Sendrecv[const void *sendbuf, int sendcount, MPI_Datatype sendtype,
        int dest, int sendtag,
        void *recvbuf, int recvcount, MPI_Datatype recvtype,
        int source, int recvtag,
        comm, MPI_Status *status]
```

sendbuf : initial address of send buffer (choice)
sendcount : number of elements in send buffer (integer)
sendtype : type of elements in send buffer (handle)
dest : rank of destination (integer)
sendtag : send tag (integer)
recvcount : number of elements in receive buffer (integer)
recvtype : type of elements in receive buffer (handle)
source : rank of source (integer)
recvtag : receive tag (integer)
comm : communicator (handle)
recvbuf: initial address of receive buffer (choice)
status : status object (Status). This refers to the receive operation.

MPI_Finalize(); ends the communication