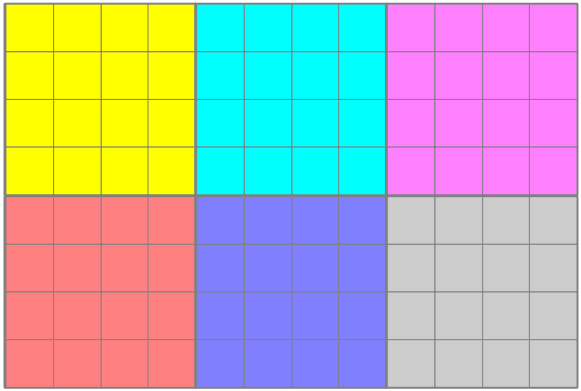
CS420 – Lecture 27

Marc Snir

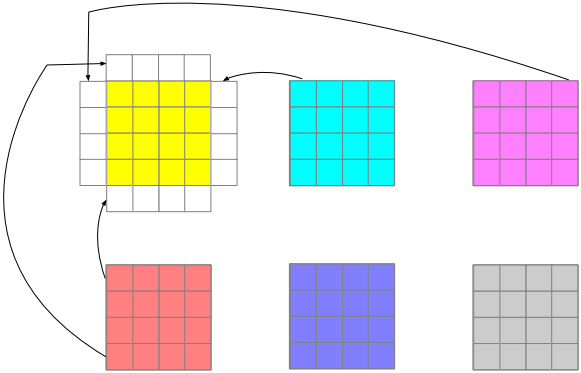
Fall 2018

MPI Cartesian Mesh and Neightorhood Collectives
aka Sparse Collectives

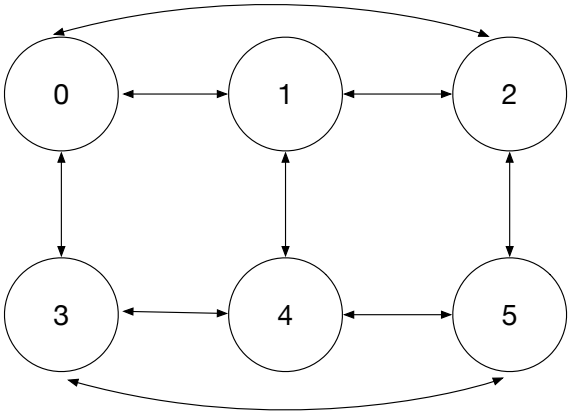
2D Jacobi



2D Jacobi – Distribute on 6 processes



Needed process topology

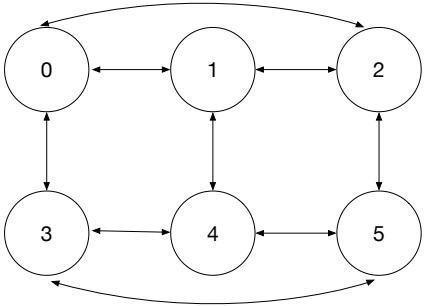


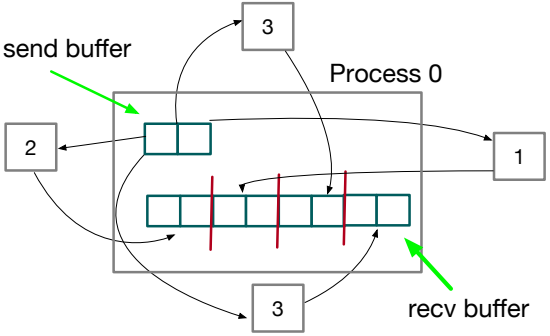
Each node (process) has four outgoing edges and four incoming edges.

Creating a Cartesian topology

```
...
int size, dims[2], periods[2];
MPI_Comm cart;

// find number of processes
MPI_Comm_size(MPI\_COMM\_WORLD, &size)

/* find 2 numbers a,b so that a*b=size, and the numbers
 are as close to one another as possible */
MPI_Dims_create(size, 2, dims)

// we want a periodic mesh
periods[0]=periods[1]=1;

/* Create Cartesian communicator
   it is a copy of MPI_COMM_WORLD, expect that processes
   also have Cartesian coordinates */
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &cart);
```

Neighborhood Collective – Allgather



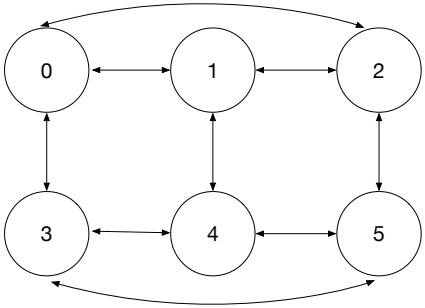


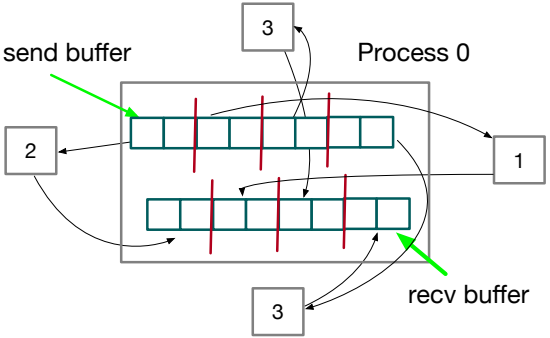
Allgather

```
MPI_Neighbor_allgather(const void* sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf, intrecvcount, MPI_Datatype recvtype, MPI_Comm
comm)
```

- Each process sends its data through all of its outgoing edges. It gathers data from all of it incoming edges.
- In a 2D Cartesian mesh, a process send its data to all of its four neighbors and receives, in one contiguous buffer, from each of them.
- Not what we need; for Jacobi, each process has to send a distinct datum to each neighbor

Neighborhood Collective – Alltoall

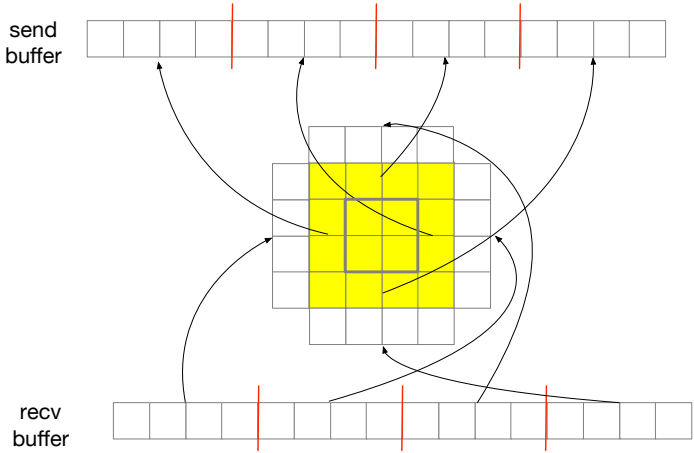




Allgather

```
MPI_Neighbor_alltoall(const void* sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf, intrecvcount, MPI_Datatype recvtype, MPI_Comm
comm)
```

- Each process sends the k-th chunck of its send buffer though its k-th outgoing edge. It receives data into the k-th chunk of its receive buffer from its k-th incoming edge.
- In a 2D Cartesian mesh, a process send data chunks to all of its four neighbors and receives, data chuncks from each of them.
- Closer to what we need – but will require us to copy the boundaries intoa contiguous send buffer and copy into the ghost cells from a contiguous recevive buffer, and requires square matrix (same number of elements sent to all neightboirs)


send
buffer
recv
buffer

Halo exchange code

```
...
float A[N+2][N+2];
float sendbuf S[4*N]
float recvbuf R[4*N];
int i;
...
// copy boundaries into send buffer
for(i=1;i<=N; i++) {
 S[i] = A[i][1];
 S[N+i] = A[i][N];
 S[2*N+i] = A[1][i];
 S[3*N+i] = A[N][i];
}
MPI_Neighbor_alltoall(S,N, MPI_FLOAT, R, N, MPI_FLOAT, cart)
```

```
// copy halo from receive buffer
for(i=1;i<=N; i++) {
 A[i][0] = R[i];
 A[i][N] = R[N+i];
 A[1][i] = R[2*N+i];
 A[N][i] = R[3*N+i];
}
```

All2all variants

Can send chunks of different sizes, starting at different displacements, to different neighbors; and same for receiving.

`MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, rectype, comm)`

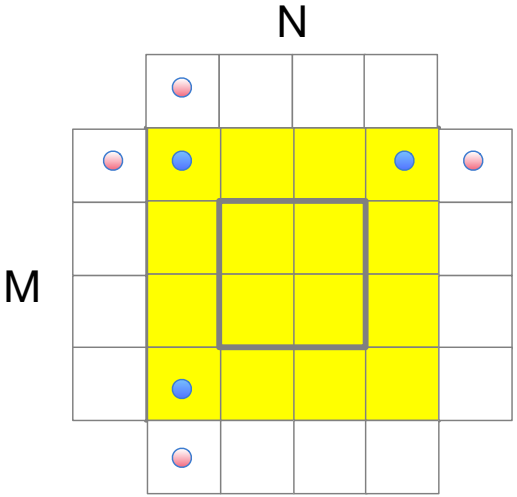
This will allow handling tiles that are not square

All2all variants

Can send chunks of different sizes, starting at different displacements, to different neighbors; and same for receiving.
`MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, rectype, comm)`
This will allow handling tiles that are not square

Can send chunks of different size, with different displacements and different datatypes to each neighbor; and same for receiving.
`MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, rectypes, comm)`
Displacements are in bytes.
This will avoid the additional copies

Datatypes



- Can use a datateyp of `MPI_FlOAT` to send and receive the top and bottom rows
- Can use a vector datatype to send and receive the left and right columns

```
...
float A[N+2][N+2];
MPI_Datatypes types[4]; // same datatypes work for sending and receiving
int sdipls[4];
int rdispls[4];
int counts[4]; // same coutns work for sending and receiving

sdispls[0] = (int)(&A[1][1]-&A[0][0])*sizeof(float);
sdispls[1] = (int)(&A[1][N]-&A[0][0])*sizeof(float);
sdispls[2] = sdispls[0];
sdipls[3] = (int)(&A[M][1]-&A[0][0])*sizeof(float);

rdispls[0] = (int)(&A[1][0]-&A[0][0])*sizeof(float);
rdipls[1] = (int)(&A[1][N+1]-&A[0][0]])*sizeof(float);
rdispls[2] = (int)(&A[0][1]-&A[0][0])*sizeof(float);
rdipls[3] = (int))&A[M+1][1]-&A[0][0])*sizeof(float);
```

```
counts[0] = counts [2] = 1; // will use one vector
counts[1] = counts[3] = N;

types[1] = types[3] = MPI_FLOAT

MPI_Type_vector(M, 1, N+2, MPI_FLOAT, &types[0]);
types[2] = types[0];

MPI_Type_commit(&types[0]);
MPI_Type_commit(&types[2]);

...
MPI_Neighbor_alltoallw(a, counts, sdipls, types, a,
     counts, rdipls, types, cart);
```

MPI and Threads

```
#include <mpi.h>
#include <omp.h>

int main(argc, argv) {
 MPI_Init_thread(argc, argv, xxx, &provided);
 ...
 MPI_Isend(...);
#pragma omp parallel
 {some code}
 MPI_Wait(...);
}
```

- The code is multithreaded
- But MPI is invoked by only one thread (the master thread)
- Required mode is `MPI_THREAD_FUNNELED`

```
#include <mpi.h>
#include <omp.h>
#include <stdlib.h>
...
int main(argc, argv) {

 MPI_Init_thread(argc, argv, MPI_THREAD_FUNNELED &provided);
 if (provided < MPI_THREAD_FUNNELED) exit(EXIT_FAILURE);
 ...
 MPI_Isend(...);
 #pragma omp parallel
 {some code}
 MPI_Wait(...);
}
```

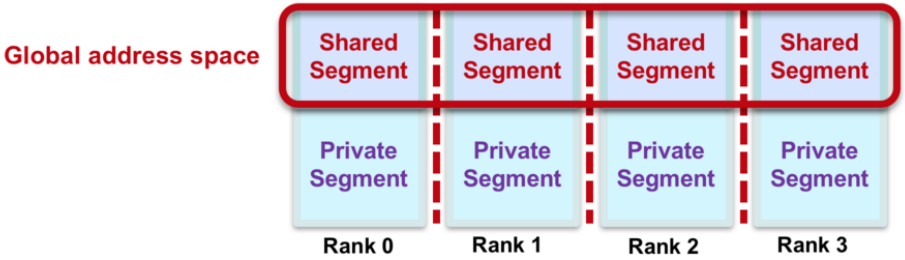
```
#include <mpi.h>
#include <omp.h>

int main(argc, argv) {
 MPI_Init_thread(argc, argv, xxx, &provided);
 ...
 #pragma omp parallel
 {MPI_Send(...);}
}
```

- The code is multithreaded
- MPI is invoked concurrently by multiple threads (each thread executes the send call)
- Required mode is `MPI_THREAD_MULTIPLE`

```
#include <mpi.h>
#include <omp.h>
#include <stdlib.h>
...

int main(argc, argv) {
 MPI_Init_thread(argc, argv, MPI_THREAD_MULTIPLE &provided);
 if (provided < MPI_THREAD_MULTIPLE) exit(EXIT_FAILURE);
  ...
 #pragma omp parallel
 {MPI_Send(...);}
}
```

PGAS – UPC

PGAS


Global address space
Shared Segment
Shared Segment
Shared Segment
Shared Segment
Private Segment
Private Segment
Private Segment
Private Segment
Rank 0
Rank 1
Rank 2
Rank 3

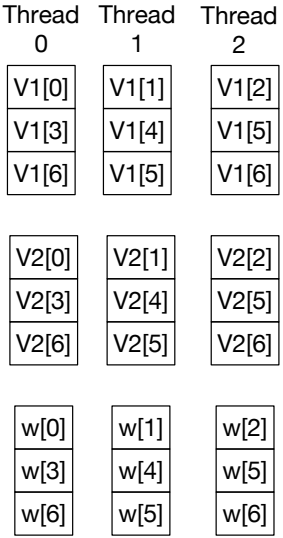
- Each process (misleadingly called thread) executes the main program.
- `THREADS` is the number of processes and `MYTHREAD` is the local process rank.
- All processes can access locations in the shared segments
- But read or write of remote location is expensive
- Arrays can be distributed across all processes

Vector addition

Bad code!

```
#include <upc_relaxed.h>
define N 10000

shared float v1[N], v2[n], w[N]; // distributed arrays
int main () {
 int i;
// each thread picks a segment of the array indices
 first = MYTHREAD*N/THREADS;
 last = (MYTHREAD+1)*N/THREADS;
 for(i=first; i<last; i++)
  w[i] = v1[i]+v2[i]:
```

Why Bad?



Good: `v1`, `v2` and `w` are distributed the same way across the processes

Bad: Most of the operations performed by each process use operands that are stored remotely.

Better code

```
#include <upc_relaxed.h>
define N 10000

shared float v1[N], v2[n], w[N]; // distributed arrays
int main () {
int i;
for(i=MYTHREAD; i<N; i+=THREADS)
w[i] = v1[i]+v2[i]:
```

- A non-optimizing compiler may still use global pointers in the loop, slowing down computation.
- `w[i]` is stored on process `i%THREADS`, at displacement `i/THREADS` – integer division and mod are expnesive operations.

Best code

```
#include <upc_relaxed.h>
define N 10000

shared float v1[N], v2[n], w[N]; // distributed arrays
int main () {
  int i;
  upc_forall(i=0; i<N; i++; i)
  w[i] = v1[i]+v2[i]:
```

- The call is collective (invoked by all processes)
- `upc_forall` execute iteration `i` on process `i%THREADS`
- The compiler generated code will loop on local members of the array.
- Similar to the OpenMP parallel for construct.

Better

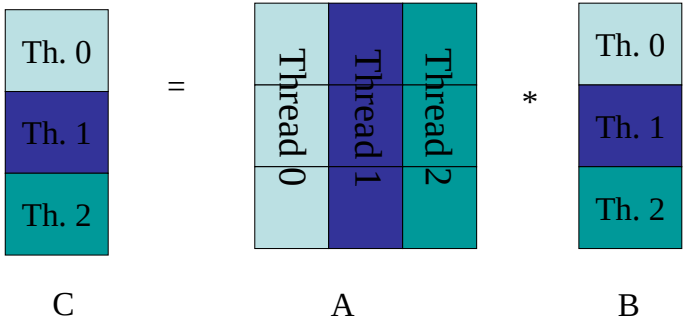
```
#include <upc_relaxed.h>
define N 10000

shared float v1[N], v2[n], w[N]; // distributed arrays
int main () {
int i;
upc_forall(i=0; i<N; i++; w[i])
w[i] = v1[i]+v2[i]:
```

- Iteration `i` is executed on the process that owns `w[i]`.
- Works fine provided that `v1`, `v2` and `w` have same distribution, irrespective of what this distribution is.

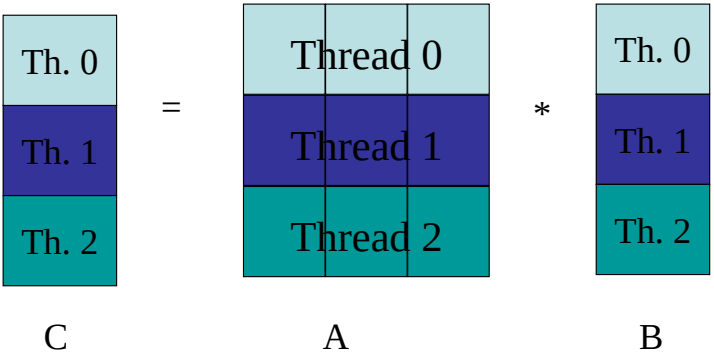
Other Distributions – matrix-vector product

```
#include <upc_relaxed.h>

shared int a[THREADS][THREADS];
shared int b[THREADS], c[THREADS] ;
int main () {
 int i, j;
 upc_forall( i = 0 ; i < THREADS ; i++; i) {
  c[i] = 0;
  for ( j= 0 ; j < THREADS ; j++)
   c[i] += a[i][j]*b[j];
  }
}
```

Bad distribution



- Process `i` computes `c[i]` and accesses elements from all the other processes.
- It is preferable to partition the matrix by rows


Th. 0
Th. 1
Th. 2
=
Thread 0
Thread 1
Thread 2
*
Th. 0
Th. 1
Th. 2
C
A
B

Code

```
#include <upc_relaxed.h>
// partition a into chunks of size THREADS
shared [THREADS] int a[THREADS][THREADS];
shared int b[THREADS], c[THREADS];

int main () {
 int i, j;

 upc_forall( i = 0 ; i < THREADS ; i++; i) {
  c[i] = 0;
  for ( j= 0 ; j< THREADS ; j++)
   c[i] += a[i][j]*b[j];
  }
}
```