CS420 – Lecture 10

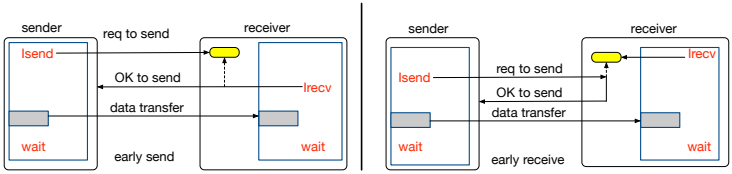
Marc Snir

Fall 2018

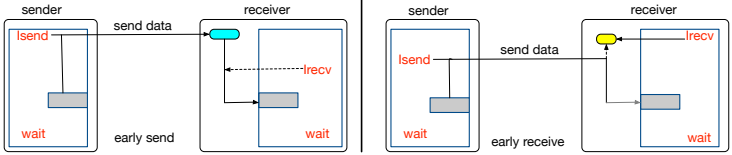
Nonblocking communication

- If wait is too early, process blocks until communication is complete.
- Start communication as soon as possible and wait as late as possible

Rendezvous



Eager



Back to Jacobi

```
...
double a[2][M][N];
MPI_Request req[4];
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
...
/* assume (M-2)*size=N */
for(iter=0;iter<MAX;iter++) {
/* up */
 if(rank>0) {
  MPI_Isend(&a[1][1][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,&req[0]);
  MPI_Irecv(&a[1][0][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,&req[1]);
  }
```

```
/*down */
 if(rank<size-1) {
  MPI_Isend(&a[l][M-2][1],N-2,MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,&req[2]);
  MPI_Irecv(&a[l][M-1][1],N-2,MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,&req[3]);
  }
 if(rank==0) MPI_Waitall(2,&req[2],MPI_STATUSES_IGNORE);
 else if(rank==(size-1)) MPI_Waitall(2,&req[0],MPI_STATUSES_IGNORE);
 else MPI_Waitall(4,req,MPI_STATUSES_IGNORE);

 for (i=1;i<N-1; i++)
  for(j=1;j<N-1;j++)
   a[1-l][i][j]=0.25*(a[l][i-1][j]+a[l][i+1][j]+a[l][i][j-1]+a[l][i][j+1]);
 l=1-l;
}
```

Reduction

- Convergence check: Need to compute the max of the local errors.
- Can use *collective communication*

```
...
int rank,maxrank,sumrank,size;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Reduce(&rank,&maxrank,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
MPI_Reduce(&rank,&sumrank,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
if(rank==0) printf('' %d %d \n'', maxrank,sumrank);
...
```

```
MPI_Reduce(&rank,&maxrank,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
```

```
int MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm)
```

```
MPI_Reduce(&rank,&maxrank,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
```

```
int MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm)
```

sendbuf: the location where my input comes from

```
MPI_Reduce(&rank,&maxrank,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
```

```
int MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm)
```

sendbuf: the location where my input comes from

recvbuf: the location where the result goes to

```
MPI_Reduce(&rank,&maxrank,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
```

```
int MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm)
```

sendbuf: the location where my input comes from

recvbuf: the location where the result goes to

count: the length of the vector being reduced element-wise

```
MPI_Reduce(&rank,&maxrank,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
```

```
int MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm)
```

- sendbuf: the location where my input comes from
- recvbuf: the location where the result goes to
- count: the length of the vector being reduced element-wise
- datatype: the type of each vector element

```
MPI_Reduce(&rank,&maxrank,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
```

```
int MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm)
```

- sendbuf: the location where my input comes from
- recvbuf: the location where the result goes to
- count: the length of the vector being reduced element-wise
- datatype: the type of each vector element
- op: the reduction operation

```
MPI_Reduce(&rank,&maxrank,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
```

```
int MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm)
```

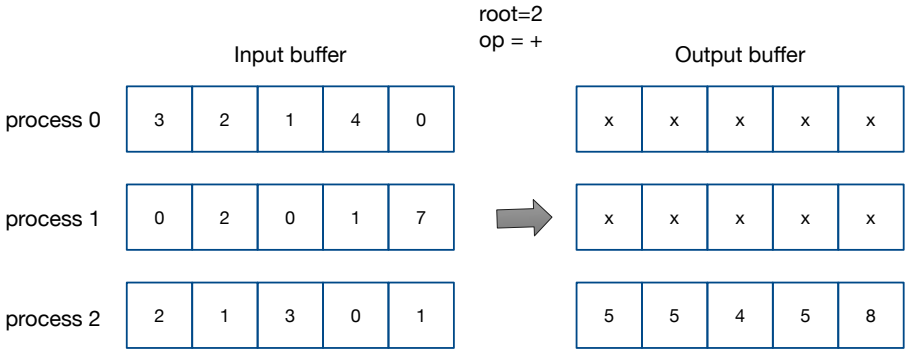
- sendbuf: the location where my input comes from
- recvbuf: the location where the result goes to
- count: the length of the vector being reduced element-wise
- datatype: the type of each vector element
- op: the reduction operation
- root: the rank of the process that is gathering the result

```
MPI_Reduce(&rank,&maxrank,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
```

```
int MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm)
```

- sendbuf: the location where my input comes from
- recvbuf: the location where the result goes to
- count: the length of the vector being reduced element-wise
- datatype: the type of each vector element
- op: the reduction operation
- root: the rank of the process that is gathering the result
- comm: the group of processes involved in the reduction

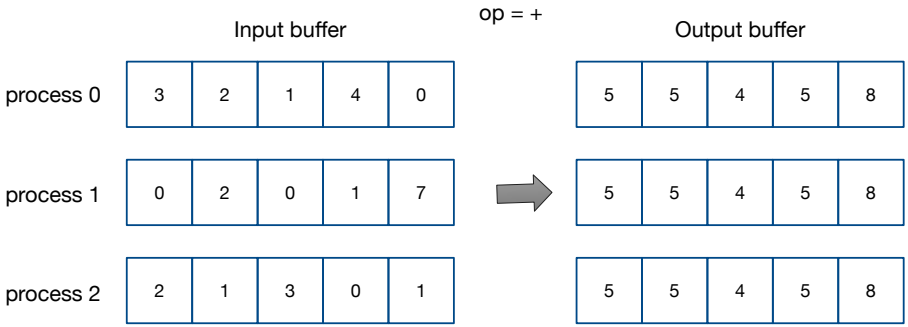
MPI_Reduce


root=2
op = +
Input buffer
Output buffer
process 0
3 2 1 4 0
x x x x x
process 1
0 2 0 1 7
x x x x x
process 2
2 1 3 0 1
5 5 4 5 8

MPI_Allreduce

- With reduce one process gets the result of the reduction and can decide whether the iteration converged. But we need *all* processes to break out of the loop.
- Use `allreduce`, instead: The result of the reduction is broadcast to all participating processes
- `MPI_Allreduce(sendbuf,recvbuf,count,datatype,op,comm)`
- Same as `MPI_Reduce`, except that there is no `root` argument

MPI_Allreduce



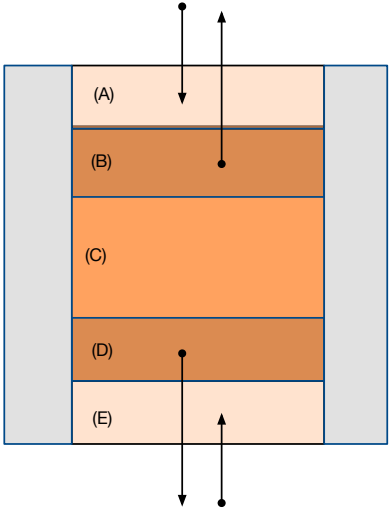
Jacobi, again

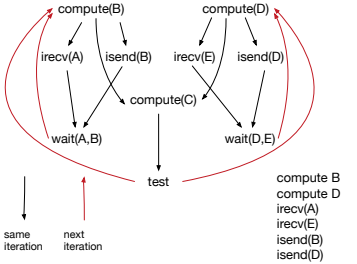
```
double a[2][M][N];
double local_err,global_err;
MPI_Request req[4];
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
...
/* assume (M-2)*size=N */
do {
/* up */
 if(rank>0) {
  MPI_Isend(&a[1][1][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,&req[0]);
  MPI_Irecv(&a[0][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,&req[1]);
  }
/*down */
 if(rank < size-1) {
  MPI_Isend(&a[1][M-1][1],N-2,MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,&req[2]);
  MPI_Irecv(&a[1][M-1][1],N-2,MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,&req[3]);
  }
```

```
if(rank==0) MPI_Waitall(2,&req[2],MPI_STATUSES_IGNORE);
else if(rank==(size-1)) MPI_Waitall(2,&req[0],MPI_STATUSES_IGNORE);
else MPI_Waitall(4,req,MPI_STATUSES_IGNORE);

local_err=0;
for (i=1;i<N-1; i++)
 for(j=1;j<N-1;j++) {
  a[1-l][i][j]=0.25*(a[l][i-1][j]
      +a[l][i+1][j]+a[l][i][j-1]+a[l][i][j+1]);
 local_err = fmax(local_err,fabs(a[1][i][j]-a[0][i][j]));
 }
l=1-l;
MPI_Allreduce(&local_err,&global_err,1,MPI_DOUBLE,MPI_MAX,MPI_COMM_WORLD);
} while(global_err>maxerr);
```

Optimize: overlap computation and communication





start communication as early as possible; end it as late as possible

Optimized jacobi

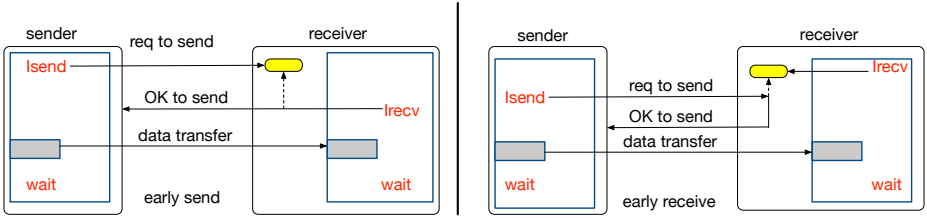
```
...
MPI_Request req[5];
/* loop */
do {
 local_err=0; l=1-l;
/* compute top, bottom rows */
 for(j=1;j<N-1;j++) {
  a[1-l][1][j]=0.25*(a[l][0][j]
     +a[l][2][j]+a[l][1][j-1]+a[l][1][j+1]);
  local_err = fmax(local_err,fabs(a[1][1][j]-a[0][1][j]));
  }
 for(j=1;j<N-1;j++) {
  a[1-l][M-2][j]=0.25*(a[l][M-3][j]
   +a[l][M-1][j]+a[l][M-2][j-1]+a[l][M-2][j+1]);
  local_err = fmax(local_err,fabs(a[1][M-2][j]-a[0][M-2][j]));
  }
```

```
/* start communications */
 if(rank>0) {
  MPI_Isend(&a[1][1][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,&req[0]);
  MPI_Irecv(&a[0][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,&req[1]);
  }
 if(rank < size-1) {
  MPI_Isend(&a[1][M-1][1],N-2, MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,&req[2]);
  MPI_Irecv(&a[1][M-1][1],N-2, MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,&req[3]);
  }
/* compute interior */
 for (i=2;i<M-2; i++)
  for(j=1;j<N-1;j++) {
   a[1-l][i][j]=0.25*(a[l][i-1][j]
        +a[l][i+1][j]+a[l][i][j-1]+a[l][i][j+1]);
   local_err = fmax(local_err,fabs(a[1][i][j]-a[0][i][j]));
   }
```

```
/* start reduction */
 MPI_Iallreduce(&local_err,&global_err,1,MPI_DOUBLE,MPI_MAX,
      MPI_COMM_WORLD,&req[4]);

/* end pt-2-pt communications */
 if(rank==0) MPI_Waitall(2,&req[2],MPI_STATUSES_IGNORE);
 else if(rank==(size-1)) MPI_Waitall(2,&req[0], MPI_STATUSES_IGNORE);
 else MPI_Waitall(4, req, MPI_STATUSES_IGNORE);
/* end reduction */
 MPI_Wait(req[4], MPI_STASTUS_IGNORE)
 }  while(global_err>maxerr);
```

To what extent is communication overlapped with computation?

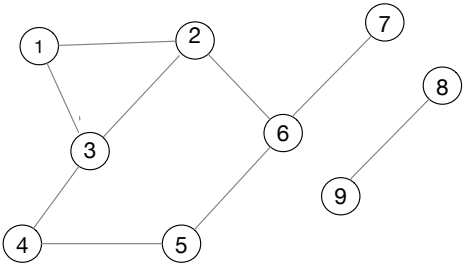


- Most MPI libraries require software polling for communication to progress
- Can dedicate a thread as "progress engine"
- Otherwise, progress occurs when MPI is called

Shared memory vs. distributed memory Jacobi

- Computation partitioned in similar ways (tiling)
- Dynamic partitioning and overpartitioning seldom used for distributed memory
- Distributed memory (with MPI) adds the burden of
  - Distributing data, and using local indices/pointers
  - Communicating data (ghost cells)
  - Replicating sequential code

Distributed memory graph traversal



1	2 3
2	1 3 6
3	1 2 4
4	3 5
5	4 6
6	2 5 7
7	6
8	9
9	8

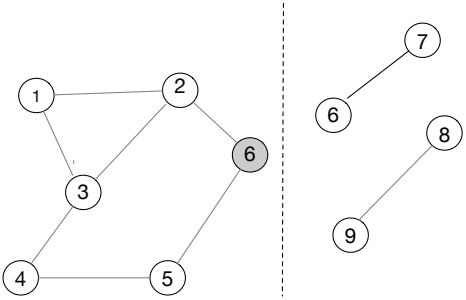
Distributed graph representation

- Nodes are distributed across processes
- If node u owned by a process connects to a node v owned by another process then a "ghost copy" of v is stored at the first process

Algorithm:

- Process 0 starts by traversing its nodes, starting from node 0
- If traversal encounters a ghost node, then a request is send to node owner to start a traversal from that node

Distributed graph representation (2 processes)



1	2	3	
2	1	3	6
3	1	2	4
4	3	5	
5	4	6	

6	2	5	7	
7	6			
8	9			
9	8			

Partial, inefficient and incomplete traversal code

```
...
typedef struct {
 bool visited;
 int degree; // number of neighbors
 int neighbor[];// indices of neighbors; use reversed sign to mark ghosts
} Node;

Node *node; // local array of nodes
int nnodes; // number of nodes
MPI_Requests reqs[max];
int reqcount // index for requests;

void graph_init() {}
```

```
void visit(int i) {
 int j,k,m;
 if (!node[i]->visited) {
  node[i]->visited=true;
  for(j=0;j<node[i]->degree;j++) {
   k=node[i]->neighbor[j];
   if(k<0)
    MPI_Isend(-k, 1, MPI_INT,
      -k*size/nnodes, 0, MPI_COMM_WORLD,reqs[reqcount++]);
   else
    visit(k)}
 }
}
```

```
int main(int argc, char **argv) {
 int i,j;
 MPI_Request recvreq;

 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 graph_init();

 MPI_Irecv(&i, 1, MPI_INT, MPI_ANY_SOURCE,0, MPI_COMM_WORLD, &req);
 if(rank==0)
  visit(0);

 while(notdone) {
  MPI_Wait(&req,MPI_STATUS_IGNORE);
  j=i;
  MPI_Irecv(&i, 1, MPI_INT, MPI_ANY_SOURCE,0, MPI_COMM_WORLD, &recvreq);
  visit(j);
 // complete all sends of visit
  MPI_Waitall(reqs,reqcount,MPI_STATUSES_IGNORE);
  reqcount=0;
  }

 MPI_Finalize();
```

Problems

- Correctness: How do we know computation has completed?
  - Need to periodically test for completion; i.e., that all processes are done
- Performance: Sending too many small messages
  - Need to aggregate messages and send them in bulk

Bulk-synchronous code – (not debugged!)

- Computation proceeds in phases, with communication at end of each phase
- At each phase do as much work as possible without communicating

```
typedef struct {
  bool visited;
  int degree;
  int neighbor[]; // mark ghost cell with negative index
} Node;

Node *node[nnodes];

void graph_init() {}

int rank, size;
int **out;     // send buffers (one per destination)
int *out_ptr; // points to first empty slot in output buffer
int *in;       // receive buffer
int in_ptr;    // points to first empty slot in input buffer
MPI_Request *reqs;
MPI_Status status;
MPI_Request *reqs;
```

```
void visit(int i) {
  int j,k,m;
  if (!node[i]->visited) {
    node[i]->visited=true;
    for(j=0;j<node[i]->degree;j++) {
      k= node[i]->neighbor[j];
      if(k<0) { // ghost node; append to  appropiate output list
        k=-k;
        m = k*size/nnodes; // destination
        out[m][out_ptr[m]++]=k;
      }
      else
      visit(k);
    }
  }
}
```

```
int main(int argc, char **argv) {
  int i,j,k;
  bool done, global_done;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  graph_init();
  for(i=0;i<size;i++) out[size] = (int*)malloc(max*sizeof(int));
  in = (int *)malloc(max*size*sizeof(int));
  reqs = (MPI_Request *)malloc(size*sizeof(MPI_Request));
  out_ptr = (int *)malloc(size*sizeof(int));
  if(rank==0)
     in[in_ptr++] =0;
```

```
while (1) {
 for(i=0;i<size;i++)
  out_ptr[i]=0;

// visit all newly marked nodes
 for(i=0; i<in_ptr ;i++)
  visit(in[i]);
 in_ptr=0;

// send out marked ghost nodes
 k=0;
 for(i=0;i<size;i++)
  if (i!=rank)
 MPI_Isend(out[i],out_ptr[i], MPI_INT, i, 0, MPI_COMM_WORLD, &reqs[k++]);
```

```
 // receive marked ghost nodes
  done=true;
  for(i=0;i<size;i++)
   if(i !=rank) {
    MPI_Recv(&in[in_ptr], max, MPI_INT, MPI_ANY_SOURCE, 0,
    MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_INT,&j);
    if(j>0) done=false;
    in_ptr+=j;
    }

 // complete sends
  MPI_Waitall(reqs,size-1,MPI_STATUSES_IGNORE)

 // test for completion
  MPI_Allreduce(&done,&global_done, 1, MPI_C_BOOL,MPI_LAND, MPI_COMM_WORLD);
  if(global_done) break;
  }
}
```

New Constructs

- `MPI_ANY_SOURCE` – wildcard source
- Also have `MPI_ANY_TAG`
- `status` object, of type `MPI_Status`
- `MPI_Get_count(status, datatype, count)`
- Also can directly access `status.MPI_SOURCE` and `status.MPI_TAG`
- Why getting count is more complicated?