CS420 – Lecture 12

Marc Snir

Fall 2018

Which of the following programs has a data race?

```
int sum = 0;
#pragma omp parallel
{
    sum = omp_get_thread_num();
    ++sum;
}
```

```
#pragma omp parallel
{
    int sum = 0;
    ++sum;
}
```

```
int sum = 5;
++sum;
#pragma omp parallel
{
    int a = sum;
}
```

```
#pragma omp parallel
{
    int sum = 0;
    sum = omp_get_thread_num();
    ++sum;
}
```

Which of the following programs has a data race?

```
int sum = 0;
#pragma omp parallel
{
    sum = omp_get_thread_num();
    ++sum;
}
```

Yes

```
#pragma omp parallel
{
    int sum = 0;
    ++sum;
}
```

```
int sum = 5;
++sum;
#pragma omp parallel
{
    int a = sum;
}
```

```
#pragma omp parallel
{
    int sum = 0;
    sum = omp_get_thread_num();
    ++sum;
}
```

Which of the following programs has a data race?

```
int sum = 0;
#pragma omp parallel
{
    sum = omp_get_thread_num();
    ++sum;
}
```

Yes

```
#pragma omp parallel
{
    int sum = 0;
    ++sum;
}
```

No

```
int sum = 5;
++sum;
#pragma omp parallel
{
    int a = sum;
}
```

```
#pragma omp parallel
{
    int sum = 0;
    sum = omp_get_thread_num();
    ++sum;
}
```

Which of the following programs has a data race?

```
int sum = 0;
#pragma omp parallel
{
    sum = omp_get_thread_num();
    ++sum;
}
```

Yes

```
#pragma omp parallel
{
    int sum = 0;
    ++sum;
}
```

No

```
int sum = 5;
++sum;
#pragma omp parallel
{
    int a = sum;
}
```

No

```
#pragma omp parallel
{
    int sum = 0;
    sum = omp_get_thread_num();
    ++sum;
}
```

Which of the following programs has a data race?

```
int sum = 0;
#pragma omp parallel
{
    sum = omp_get_thread_num();
    ++sum;
}
```

Yes

```
#pragma omp parallel
{
    int sum = 0;
    ++sum;
}
```

No

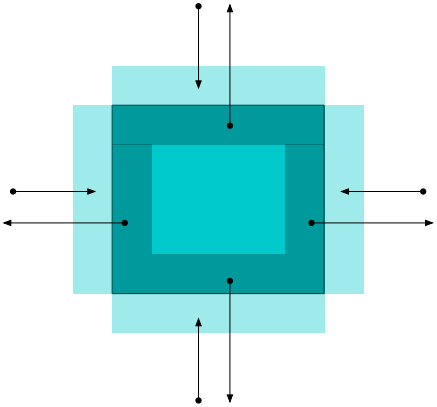
```
int sum = 5;
++sum;
#pragma omp parallel
{
    int a = sum;
}
```

No

```
#pragma omp parallel
{
    int sum = 0;
    sum = omp_get_thread_num();
    ++sum;
}
```

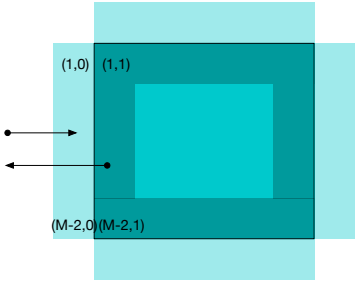
No

Jacobi – 2D decomposition



- Problem: Columns are no stored in consecutive locations
- Solution 1: Pack to consecutive locations to send; unpack after receiving
- Solution 2: Define a suitable datatype for the send and receive locations
- Solution 2 is better *if the MPI implementation does a good job*

Option 1




```
...
MPI_Irecv(fromleft,M-2,MPI_DOUBLE,myrank-1,
    0,MPI_COMM_WORLD,&req[0]);
for(i=1;i<M-1;i++)
 toleft[i]=a[1][i][1];
MPI_Isend(toleft,M-2,MPI_DOUBLE,myrank-1,
    0,MPI_COMM_WORLD,&req[1]);
...
MPI_Waitall(...)
for(i=1;i<M-1;i++)
 a[1][i][0]=fromleft[i];
 ...
```

Derived datatypes

count=3

blocklength=2

stride=6

```
MPI_Type_vector(3,2,6,MPI_DOUBLE,&vector-type);
MPI_TYPE_VECTOR(count,blocklength,stride,
oldtype,newtype)
```

Derived datatypes

count=3

blocklength=2

stride=6

```
MPI_Type_vector(3,2,6,MPI_DOUBLE,&vector-type);
MPI_TYPE_VECTOR(count,blocklength,stride,
oldtype,newtype)
```

count: number of blocks

Derived datatypes


```
MPI_Type_vector(3,2,6,MPI_DOUBLE,&vector-type);
MPI_TYPE_VECTOR(count,blocklength,stride,
oldtype,newtype)
```

count: number of blocks

blocklength: number of elements per block (usually 1)

Derived datatypes

count=3

blocklength=2

stride=6

```
MPI_Type_vector(3,2,6,MPI_DOUBLE,&vector-type);
MPI_TYPE_VECTOR(count,blocklength,stride,
oldtype,newtype)
```

count: number of blocks

blocklength: number of elements per block (usually 1)

stride: distance from start of block to start of next block

Derived datatypes


```
MPI_Type_vector(3,2,6,MPI_DOUBLE,&vector-type);
MPI_TYPE_VECTOR(count,blocklength,stride,
oldtype,newtype)
```

- count: number of blocks
- blocklength: number of elements per block (usually 1)
- stride: distance from start of block to start of next block
- oldtype: type of each element

Derived datatypes


```
MPI_Type_vector(3,2,6,MPI_DOUBLE,&vector-type);
MPI_TYPE_VECTOR(count,blocklength,stride,
oldtype,newtype)
```

count: number of blocks

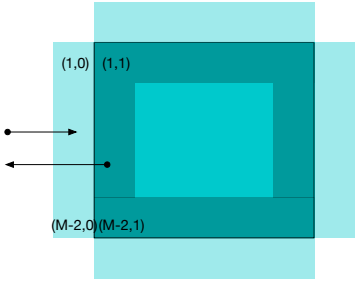
blocklength: number of elements per block (usually 1)

stride: distance from start of block to start of next block

oldtype: type of each element

newtype: name of new derived datatype

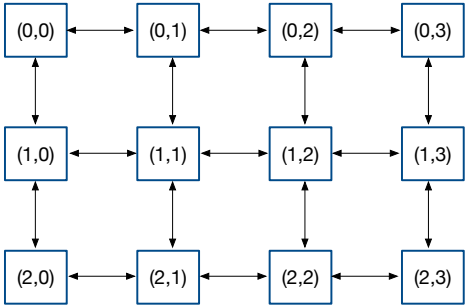
Option 2




```
...
MPI_Type_vector(M-2,1,N,MPI_DOUBLE,&vector_type
MPI_Type_commit(&vector_type);
MPI_Irecv(&a[1][1][0],1,vector_type,myrank-1,
  0,MPI_COMM_WORLD,&req[0]);
MPI_Isend(&a[1][1][1],1,vector_type,myrank-1,
   0,MPI_COMM_WORLD,&req[1]);
...
MPI_Waitall(...)
...
```

Cartesian grid of processes

- For the 2D Jacobi, it is convenient to think of the processes as organized in a 2D mesh
- Can do this with `MPI_Cart_create`




```
dims[0]=4; dims[1]=3;
periods[0]=false; periods[1]=false;
reorder= true;
ndim=2;
MPI_Cart_create(MPI_COMM_WORLD,ndim,
dims,periods,reorder,&comm2d);
```

- May want to know my Cartesian coordinates: `MPI_Cart_Coords()`

- May want to know my Cartesian coordinates: `MPI_Cart_Coords()`
- May want to know my rank and the rank of one of my neighbors: `MPI_Cart_shift(comm2d,direction,disp,&rank_source,&rank_dest)`

- May want to know my Cartesian coordinates: `MPI_Cart_Coords()`
- May want to know my rank and the rank of one of my neighbors: `MPI_Cart_shift(comm2d,direction,disp,&rank_source,&rank_dest)`
- What happens if there is no neighbor in the chosen direction? The call returns `MPI_PROC_NULL`; a send to `MPI_PROC_NULL` or a receive from `MPI_PROC_NULL` is a noop.

2D Jacobi, in all its glory...

```
...
/* initialization */
MPI_Comm_size(MPI_Comm_World,&size)
/* pick grid dimensions */
MPI_Dims_create(size,2,dims)
/* dims[0]*dims[1] = size */
period[0]=period[1]=false;
MP_Cart_create(MPI_COMM_WORLD,2,dims,periods,true,&comm2d);
MPI_Cart_shift(comm2d,0,-1,&myrank,&up);
MPI_Cart_shift(comm2d,0,1,&myrank,&down);
MPI_Cart_shift(comm2d,1,-1,&myrank,&left);
MPI_Cart_shift(comm2d,1,1,&myrank,&right);

MPI_Type_vector(M-2,1,N,MPI_DOUBLE,&vector_type);
MPI_Type_commit(&vector_type);
```

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

```
/* left */
for(i=1;i<M-1;i++) {
 a[1-l][i][1]=0.25*(a[l][i-1][1]
  +a[l][i+1][1]+a[l][i][0]+a[l][i][2]);
 local_err = fmax(local_err,fabs(a[1][i][1]-a[0][i][1]));
}
/* right */
for(i=1;i<M-1;i++) {
 a[1-l][i][N-2]=0.25*(a[l][i-1][N-3]
  +a[l][i+1][N-1]+a[l][i][N-3]+a[l][i][N-1]);
 local_err = fmax(local_err,fabs(a[1][i][N-2]-a[0][i][N-2]));
}
/* start communications */
MPI_Irecv(&a[l][0][1],N-2,MPI_DOUBLE,up,0,MPI_COMM_WORLD,&req[1]);
MPI_Irecv(&a[l][M-1][1],N-2,MPI_DOUBLE,down,0,MPI_COMM_WORLD,&req[2]);
MPI_Irecv(&a[l][1][0],1,vector_type,left,MPI_COMM_WORLD,&req[3]);
MPI_Irecv(&a[l][1][N-1],1,vector_type,right,MPI_COMM_WORLD,&req[4])
MPI_Isend(&a[l][1][1],N-2,MPI_DOUBLE,up,MPI_COMM_WORLD,&req[5]);
MPI_Isend(&a[l][M-2][1],N-2,MPI_DOUBLE,down,MPI_COMM_WORLD,&req[6]);
MPI_Isend(&a[l][1][1],1,vector_type,left,MPI_COMM_WORLD,&req[7]);
MPI_Isend(&a[l][1][N-2],1,vector_type,right,MPI_COMM_WORLD,&req[8]);
```

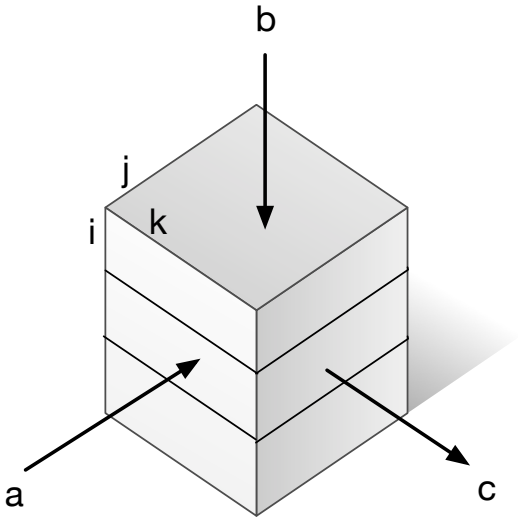
```
/* compute interior */
for (i=2;i<M-2; i++)
 for(j=2;j<N-2;j++) {
  a[1-l][i][j]=0.25*(a[l][i-1][j]
   +a[l][i+1][j]+a[l][i][j-1]+a[l][i][j+1]);
  local_err = fmax(local_err,fabs(a[1][i][j]-a[0][i][j]));
}
l=1-l;

/* start convergence test */
MPI_Iallreduce(&local_err,&global_err,1,MPI_DOUBLE,MPI_MAX,
  MPI_COMM_WORLD,&req[9]);

/* end communications */
MPI_Waitall(9,req,MPI_STATUSES_IGNORE);

}  while(global_err>maxerr);
```

Matrix product 1D tiling



Will parallelize with MPI, by assigning a tile to each process

Code

```
...
int matmult(int nra, int nca, int ncb, double a[nra][nca],
double b[nca][ncb], double c[nra][ncb], MPI_Comm comm) {
    /*******************
    Computes the product c = a*b using the processes in the group of
    communicator comm
    The three matrices are stored on the process with rank 0 in comm
    ************************/
    int numprocs,                /* number of processes in group */
    procid,                  /* a process identifier */
    numworkers,              /* number of worker processes */
    source,                  /* process id of message source */
    dest,                    /* process id of message destination */
    rows,                    /* rows of matrix A sent to each worker */
    averow, extra, offset,   /* used to determine rows sent to each worker */
    i, j, k, rc;             /* misc */

    MPI_Comm_rank(comm,&procid);
    MPI_Comm_size(comm,&numprocs);
```

```
if (numprocs < 2 ) {
    /* run sequential code */
    for (i=0;i<nra;i++)
    for (j=0;j<ncb;j++)
    for(k=0;k<nca;k++)
    c[i][j] += a[i][k]*b[k][j];
    exit(1);
}
```

```
if(procid==0) {
/**************  master process ***************/
numworkers = numprocs-1;
/* Send matrix data to the worker tasks */
averow = nra/numworkers;
extra = nra%numworkers;
offset = 0;
for (dest=1; dest<=numworkers; dest++)
{
    rows = (dest <= extra) ? averow+1 : averow;
    MPI_Send(&offset, 1, MPI_INT, dest, 0, comm);
    MPI_Send(&rows, 1, MPI_INT, dest, 0, comm);
    MPI_Send(&a[offset][0], rows*nca, MPI_DOUBLE, dest, 0,
    comm);
    MPI_Send(b, nca*ncb, MPI_DOUBLE, dest, 0, comm);
    offset = offset + rows;
}
```

```
/* Receive results from worker processes */
for (source=1; source<=numworkers; source++)
{
MPI_Recv(&offset, 1, MPI_INT, source, 0, comm, MPI_STATUS_IGNORE);
MPI_Recv(&rows, 1, MPI_INT, source, 0, comm, MPI_STATUS_IGNORE);
MPI_Recv(&c[offset][0], rows*ncb, MPI_DOUBLE, source, 0,
comm, MPI_STATUS_IGNORE);
}
}
```

```
/**************************** worker task ************************************/
else {
MPI_Recv(&offset, 1, MPI_INT, 0, 0, comm, MPI_STATUS_IGNORE);
MPI_Recv(&rows, 1, MPI_INT, 0, 0, comm, MPI_STATUS_IGNORE);
MPI_Recv(a, rows*nca, MPI_DOUBLE, 0, 0, comm, MPI_STATUS_IGNORE);
MPI_Recv(b, nca*ncb, MPI_DOUBLE, 0, 0, comm, MPI_STATUS_IGNORE);

for (i=0; i<rows; i++)
for (j=0; j<ncb; j++) {
c[i][j] = 0;
for (k=0;k<nca;k++)
c[i][j] += a[i][j] * b[j][k];
}
MPI_Send(&offset, 1, MPI_INT, 0, 0, comm);
MPI_Send(&rows, 1, MPI_INT, 0, 0, comm);
MPI_Send(&c, rows*ncb, MPI_DOUBLE, 0, 0, comm);
}
return(1);
}
```

- Matmult is invoked on all processes – should be a collective invocation
- Matrices `a`,`b` and `c` are allocated on all processes – not a big problem?
- (Usually matmult is invoked while matrices are already distributed)

Possible improvements:

- Matmult is invoked on all processes – should be a collective invocation
- Matrices `a`,`b` and `c` are allocated on all processes – not a big problem?
- (Usually matmult is invoked while matrices are already distributed)

Possible improvements:

- Involve process 0 in computation as well

- Matmult is invoked on all processes – should be a collective invocation
- Matrices `a`,`b` and `c` are allocated on all processes – not a big problem?
- (Usually matmult is invoked while matrices are already distributed)

Possible improvements:

- Involve process 0 in computation as well
- Avoid communicating `offset` and `rows`

- Matmult is invoked on all processes – should be a collective invocation
- Matrices `a`,`b` and `c` are allocated on all processes – not a big problem?
- (Usually matmult is invoked while matrices are already distributed)

Possible improvements:

- Involve process 0 in computation as well
- Avoid communicating `offset` and `rows`
- Use collective communications (scatter `a`, broadcast `b`, gather `c`)

uneven scatter-gather

Problem: Not each worker receives same number of rows; need to use scatterv and gatherv functions.

```
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)
```

uneven scatter-gather

Problem: Not each worker receives same number of rows; need to use scatterv and gatherv functions.

```
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)
```

sendcounts Number of elements sent to each process

uneven scatter-gather

Problem: Not each worker receives same number of rows; need to use scatterv and gatherv functions.

`MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)`

sendcounts Number of elements sent to each process

displs `displs[i]` is displacement from start of buffer to start of elements sent to process `i`

uneven scatter-gather

Problem: Not each worker receives same number of rows; need to use scatterv and gatherv functions.

`MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)`

- sendcounts: Number of elements sent to each process
- displs: `displs[i]` is displacement from start of buffer to start of elements sent to process `i`

`MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)`

Arguments that are not needed can be NULL

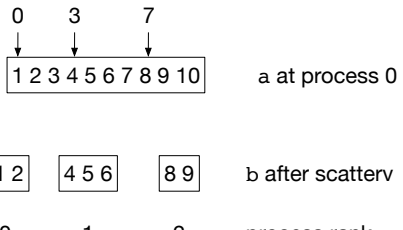
Example

Assume 3 processes

```
/* array arguments needed at root */
int a[10] ={1,2,3,4,5,6,7,8,9,10};
int counts ={2,3,2}
int displs={0,3,7}]

int b[3];

MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Scatterv(&a, counts, displs, MPI_INT, &b,
count[rank], MPI_INT,0 , MPI_COMM_WORLD);
```



Improved code

```
...
int matmult(int nra, int nca, int ncb, double a[nra][nca],
double b[nca][ncb], double c[nra][ncb], MPI_Comm comm) {

int numprocs,                /* number of processes in group */
procid,                   /* a process identifier */
source,                   /* process id of message source */
dest,                     /* process id of message destination */
averow, extra, offset,    /* used to determine rows sent to each worker */
i, j, k;                   /* misc */

MPI_Comm_rank(comm,&procid);
MPI_Comm_size(comm,&numprocs);

int  rows[numprocs],            /* # rows sent to each process */
countsa[numprocs],          /* # elements of a sent to each process */
countsc[numprocs],          /* # elements of c received form each process */
displsa[numprocs],          /* displacements in matrix a */
displsc[numprocs];          /* displacements in matrix c */
```

```
if (numprocs < 2 ) {
/* run sequential code */
for (i=0;i<nra;i++)
for (j=0;j<ncb;j++)
for(k=0;k<nca;k++)
c[i][j] += a[i][k]*b[k][j];
exit(1);
}


/* computed by all processes */
averow = nra/numprocs;
extra = nra%numprocs;
offset=0;
for (i=0; i<numprocs; i++) {
rows[i] = (i < extra) ? averow+1 : averow;
countsa[i] = rows[i]*nca;
countsc[i] = rows[i]*ncb;
displsa[i] = offset*nca;
displsc[i] = offset*ncb;
offset += rows[i];
}
```

```
/* Scatter matrix data to all processes */

double aa[rows[procid]][nca];  /* local tile of a */
double cc[rows[procid]][ncb]; /* local tile of c */

MPI_Scatterv(&a[0][0], countsa, displsa, MPI_DOUBLE, &aa[0][0],
countsa[procid], MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&b[0][0], nca*ncb, MPI_DOUBLE, 0, MPI_COMM_WORLD);

/* local computation */

for (i=0; i<rows[procid]; i++)
for (j=0; j<ncb; j++) {
cc[i][j] = 0;
for (k=0;k<nca;k++)
cc[i][j] += aa[i][k] * b[k][j];
}
/* gather results */
MPI_Gatherv(&cc[0][0], countsc[procid], MPI_DOUBLE,
&c[0][0], countsc, displsc, MPI_DOUBLE, 0, MPI_COMM_WORLD);
return(1);
}
```

Further Optimizations

Can use 2D or 3D tiling; if 3D can use collective reduce
Assume communication time is $\ell + n/b$ for message of length n.

Broadcast of n items: time $\approx (\lg p)(\ell + n/b)$

Reduce of n items ibid

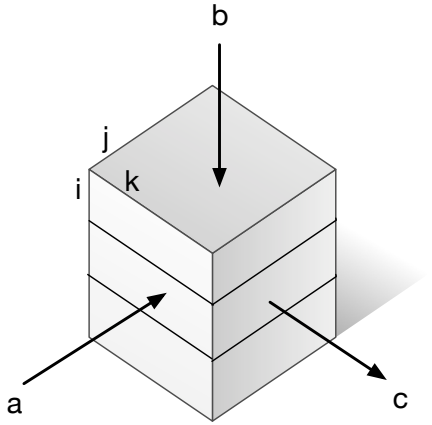
Scatter of n items time $\approx (\lg p)\ell + n/b$

Gather of n items ibid

Communication time (all have about same computation time)

Assume all matrices are $N \times N$ 1D tiling:

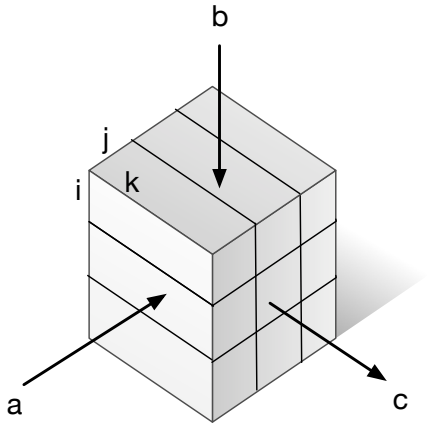
- Scatter a: $t \approx \ell \lg p + N^2/b$
- Broadcast b: $t \approx \ell \lg p + N^2 \lg p/b$
- Gather c: $t \approx \ell \lg p + N^2/b$
- Total: $t = \Theta(\lg p(\ell + N^2/b)$
  Communication time dominated by broadcast of b.



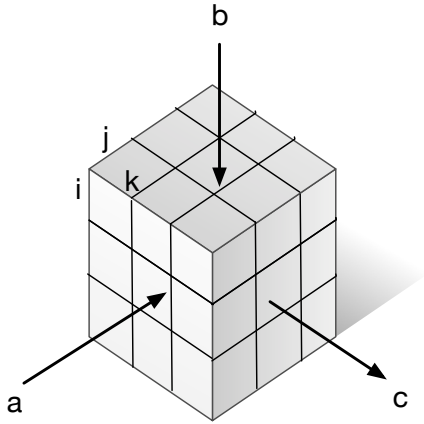
2D tiling

a and b are partitioned into $\sqrt{p}$ tiles; c is partitioned into p tiles.

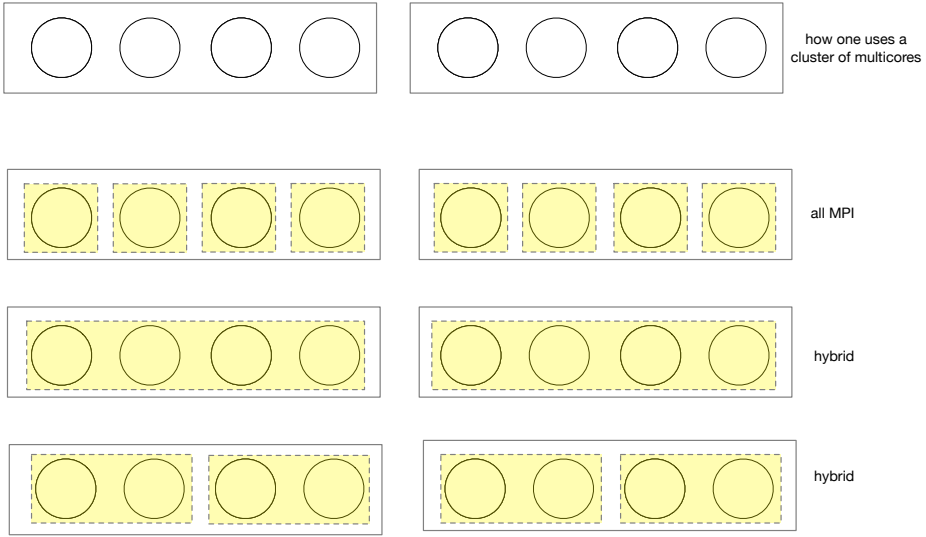
- Scatter a to $\sqrt{p}$ processes, next broadcast each tile to $\sqrt{p}$ processes: $t \approx (0.5\ell \lg p + N^2/b) + (0.5 \lg p(\ell + N^2/(b\sqrt{p}) = \Theta(\ell \lg p + N^2/b)$
- Scatter and broadcast b: ibid
- Gather c: $t \approx \ell \lg p + N^2/b$
- Total: $t = \Theta(\ell \lg p + N^2/b)$



- 1D tiling: $t = \Theta(\ell \lg p(N^2/b) \lg p)$
- 2D tiling $t = \Theta(\ell \lg p + N^2/b)$
- 3D tiling only changes constants – communicatioin time is dominated by scattter time
- 3D tiling has better communication time when matrices are already distributed



MPI+Threads


how one uses a
cluster of multicores
all MPI
hybrid
hybrid

pros and cons

- "All MPI" model is simpler
  - MPI uses shared memory for communication within node (faster)
  - But benefit is limited in a bulk -synchronous model

pros and cons

- "All MPI" model is simpler
  - MPI uses shared memory for communication within node (faster)
  - But benefit is limited in a bulk -synchronous model
- No data sharing across processes – usually increases memory consumption
- More ranks – lower MPI performance

pros and cons

- "All MPI" model is simpler
  - MPI uses shared memory for communication within node (faster)
  - But benefit is limited in a bulk -synchronous model
- No data sharing across processes – usually increases memory consumption
  - MPI 3 introduced ways of sharing data among processes on same node
- More ranks – lower MPI performance

pros and cons

- "All MPI" model is simpler
  - MPI uses shared memory for communication within node (faster)
  - But benefit is limited in a bulk -synchronous model
- No data sharing across processes – usually increases memory consumption
  - MPI 3 introduced ways of sharing data among processes on same node
- More ranks – lower MPI performance
- Hybrid model is more complicated
- May reduce memory consumption and improve performance

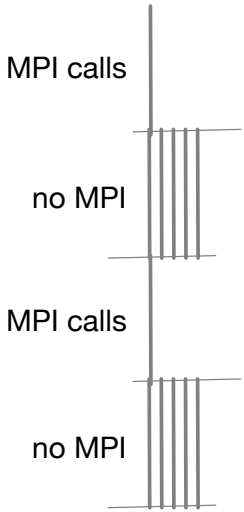
pros and cons

- "All MPI" model is simpler
  - MPI uses shared memory for communication within node (faster)
  - But benefit is limited in a bulk -synchronous model
- No data sharing across processes – usually increases memory consumption
  - MPI 3 introduced ways of sharing data among processes on same node
- More ranks – lower MPI performance
- Hybrid model is more complicated
- May reduce memory consumption and improve performance
- Hybrid 2 is often a good compromise – especially for NUMA nodes

Two ways of using MPI with OpenMP

1. MPI calls occur only when OpenMP code executes on one thread
2. MPI calls can occur concurrently on multiple threads

- Second choice provides more flexibility and "should" perform better in many cases
- MPI implementations do not support well multithreading – first choice often better, in practice.



MPI+OpenMP

Replace `MPI_Init` with `MPI_Init_thread(required,&provided)`

required: what you ask for.

provided: what you actually get.

`required` can be one of the following:

MPI_THREAD_SINGLE: MPI process is single threaded

MPI_THREAD_FUNNELED: Only the master thread (the thread that initialized MPI) executes MPI calls.

- Used when MPI always called from master thread in OpenMP

MPI_THREAD_SERIALIZED: Multiple threads can call MPI, but the calls ar not concurrent.

MPI_THREAD_MULTIPLE: No constraints on concurrent MPI calls.

- Used when multiple OpenMP threads may call MPI

Jacobi, yet again – 1D decomposition, funneled mode

```
...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if(rank==0) {firstreq=2; nreq=2;}
else if(rank==(size-1)) {firstreq=0; nreq=2;}
else {firstreq=0; nreq=4}
for(iter=0; iter<MAX; iter++) {

/* compute first and last row */
#pragma omp parallel for
 for(j=1; j<N-1; j++)
  a[1-k][1][j] = 0.25*(a[k][0][j]+a[k][2][j]
   +a[k][1][j-1]+a[k][1][j+1]);
#pragma omp parallel for
for(j=1; j<N-1; j++)
 a[1-k][M-2][j] = 0.25*(a[k][M-3][j]+a[k][M-1][j]
   +a[k][M-2][j-1]+a[k][M-2][j+1]);
```

```
/* start communication */
if (rank>0) { /* communicate with neighbor above */
 MPI_Isend(&a[k][1][1], N-2, MPI_DOUBLE, rank-1, 0,
    MPI_COMM_WORLD, &req[0]);
 MPI_Irecv(&a[k][0][1], N-2, MPI_DOUBLE, rank-1, 0,
    MPI_COMM_WORLD, &req[1]);
}
if (rank<size-1) { /* communicate with neighbor below */
 MPI_Isend(&a[k][M-1][1], N-2, MPI_DOUBLE, rank+1, 0,
    MPI_COMM_WORLD, &req[2]);
 MPI_Irecv(&a[k][M][1], N-2, MPI_DOUBLE, rank+1, 0,
MPI_COMM_WORLD, &req[3]);
}
/* compute interior */
#pragma omp parallel for collapse(2)
 for(i=1;i<M-2;i++)
  for(j=1;j<N-2;j++)
   a[1-k][i][j] = 0.25*(a[j][i-1][j]+a[k][i+1][j]
     +a[k][i][j-1]+a[k][i][j+1]);

/* complete communication */
MPI_Waitall(&reqs[freq],nreq)
k=1-k;
```

Thread multiple mode- parallelize communication

```
/* compute and communicate first and last row */
#pragma omp parallel sections
{
 #pragma omp section
  {
  for(j=1; j<N-1; j++)
   a[1-k][1][j] = 0.25*(a[k][0][j]+a[k][2][j]
   +a[k][1][j-1]+a[k][1][j+1]);
  if (rank>0) { /* communicate with neighbor above */
   MPI_Isend(&a[k][1][1], N-2, MPI_DOUBLE, rank-1, 0,
      MPI_COMM_WORLD, &req[0]);
   MPI_Irecv(&a[k][0][1], N-2, MPI_DOUBLE, rank-1, 0,
     MPI_COMM_WORLD, &req[1]);
   }
  }
```

```
 #pragma omp section
 {
 for(j=1; j<N-1; j++)
  a[1-k][M-2][j] = 0.25*(a[k][M-3][j]+a[k][M-1][j]
    +a[k][M-2][j-1]+a[k][M-2][j+1]);
  if (rank<size-1) { /* communicate with neighbor below */
   MPI_Isend(&a[k][1][1], N-2, MPI_DOUBLE, rank-1, 0,
     MPI_COMM_WORLD, &req[0]);
   MPI_Irecv(&a[k][0][1], N-2, MPI_DOUBLE, rank-1, 0,
    MPI_COMM_WORLD, &req[1]);
  }
 }
}
```

```
#pragma omp parallel sections
 {
  #pragma omp section
   block1
  #pragma omp section
   block2
   ...
}
```

Each section is executed in parallel with the other ones

Attributes

User-defined Collective Library

- Collective invocation by all processes in a communicator.
- It is desirable to ensure that communications inside the library cannot possible match communications outside the library (isolation).
- Can do so by duplicating communicator

```
matmult(...,MPI_Comm comm) {
    ...
    MPI_Comm newcomm;
    MPI_Comm_dup(comm, &newcomm)

    ...
    /* all communications use newcomm */
```

- Creating communicators is expensive; can we avoid doing so each time?
- Usually, libraries require an initialization call; can duplicate communicator at initialization.
- How do we check that the library was initialized?
- How do we check that it is invoked with the communicator used at initialization?
- Use “caching”: Communicators can carry extra information in the form of a list of attributes (key-value pairs)
- Can have multiple libraries and invoke them with multiple communicators; only coordination required is to use a unique name for the variable that holds the library key
- Can cache copy of communicator

Caching

```
/* declare a key for the matmult library; it is initially invalid.
 (will probably appear in a header file) */

int matmult_key = MPI_KEYVAL_INVALID;
...
int matmult(MPI_Comm comm) {
 MPI_Comm *private_comm;
 int flag;
/* check if library was initialized. If not, generate new (unique) key value */
 if (matmult_key == MPI_KEYVAL_INVALID)
  MPI_Comm_create_keyval(NULL, NULL, &matmult_key, NULL);
```

```
/* Check if matmult has already been invoked with comm
(if yes, a value is cached with the library key) */
 MPI_Comm_get_attr(comm, matmult_key, (void **)&private_comm, &flag);
 if (flag == 0) { /* Nothing cached */
  private_comm = (MPI_Comm *)malloc(sizeof(MPI_Comm)); /* ''New'' */
  MPI_Comm_dup(comm, private_comm); /* copy communicator */
/* attach private communicator to external one */
  MPI_Comm_set_attr(comm, matmult_key, (void *)private_comm);
  }

/* execution now continues using private_comm */
...
}
```

New function

```
MPI_Comm_create_keyval(MPI_Copy_function, MPI_Delete_function, keyval,
extra_state)
```

New function

```
MPI_Comm_create_keyval(MPI_Copy_function, MPI_Delete_function, keyval,
extra_state)
```

MPI_Copy_function: "Constructor" – function invoked when a communicator caching this key is duplicated.

New function

```
MPI_Comm_create_keyval(MPI_Copy_function, MPI_Delete_function, keyval,
extra_state)
```

MPI_Copy_function: "Constructor" – function invoked when a communicator caching this key is duplicated.

MPI_Delete_function: "Destructor" – function invoked when a communicator caching this key is freed (destructed)

New function

`MPI_Comm_create_keyval(MPI_Copy_function, MPI_Delete_function, keyval, extra_state)`

MPI_Copy_function: "Constructor" – function invoked when a communicator caching this key is duplicated.

MPI_Delete_function: "Destructor" – function invoked when a communicator caching this key is freed (destructed)

keyval: returned key value

New function

`MPI_Comm_create_keyval(MPI_Copy_function, MPI_Delete_function, keyval, extra_state)`

MPI_Copy_function: "Constructor" – function invoked when a communicator caching this key is duplicated.

MPI_Delete_function: "Destructor" – function invoked when a communicator caching this key is freed (destructed)

keyval: returned key value

extra_state: extra argument passed to the two functions

```
MPI_Comm_set_attr(comm, keyval, attribute_val)
MPI_Comm_get_attr(comm, keyval, attribute_val, flag)
```

keyval key of attribute

attribute_val value of attribute (pointer)

flag 0 if no value is associated with key, 1 otherwise

```
MPI_Comm_dup(comm,*newcomm)
```