OpenMP and MPI parallelization

Roland Haas, Senior Research Programmer
rhaas@illinois.edu
Distributed vs. shared memory

MPI

- CPU Core
- Memory

Private Arrays

CPU Core
Memory

Network Interconnect

OpenMP

- CPU Core
- Memory

Memory, Shared Arrays, etc.

Typically less memory overhead / duplication. Communication often implicit through cache coherency and runtime.
OpenMP standard

- OpenMP is a language extension (only) for C / C++ / Fortran to develop shared-memory, thread based code
  - Python 3.8 can share memory via `multiprocessing.shared_memory`
- Designed to be added to legacy code
- Compilers ignore OpenMP directives if OpenMP is not supported
- Hide complexity of threading / accelerator libraries from user

- Current version is OpenMP 5.0 from 2018 [http://openmp.org](http://openmp.org)
- Targets CPU and GPU code
- Multiple parallelism schemes
  - Domain decomposition / data parallelism
  - Task parallelism
  - Anything in between using lock primitives
General directive layout

Fortran

```fortran
$omp parallel [clause ...]
  structured block
$omp end parallel
```

C/C++

```c
#pragma omp parallel [clause ...]
{
  Structured block
}
```

- A parallel directive starts a parallel code block
- All variables are shared by default, use `private` clause to change
- Inside the `parallel` section `worksharing` constructs indicate how threads split up the available work
  - `omp for` / `do`: parallel loops
  - `omp workshare` (Fortran only): parallel array operations
  - `omp critical` (not technically a workshare construct): protect shared variables
Typical OpenMP code

• **omp for** and **omp do** are by far the most common OpenMP directives encountered in grid based codes

• **omp parallel for** / **do** shorthand

• **private** declares variable thread local

• **omp critical** to serialize sections of code

• compile with **-fopenmp** (GNU, Intel) or **-mp** (PGI) or **-h thread2** (Cray)

Fortran example

double precision tmpval,maxval
integer i
maxval = -1d0
 !$OMP parallel do private(i,tmpval)
do i=1,n
 tmpval = b(i) + q*c(i)
a(i) = tmpval
 !$omp critical
 if(tmpval .gt. maxval) then
  maxval = a(i)
 endif
 !$omp end critical
end do
 !$OMP end parallel do
Data sharing clauses

- make all variables private
  
  !$omp parallel default(private)

- explicitly share variables
  
  !$omp parallel shared(maxval)

- explicitly make variables private
  
  !$omp parallel private(i,tmpval)

- or declare in block
  
  #pragma omp parallel
  {
      double tempval;
  }

- By default all variables are shared among the threads → the most common cause of errors and race conditions

- private mostly useful for Fortran code
  - use blocks in C / C++

- making a pointer private, still keeps he pointed to memory shared
  - making a C or Fortran array private creates a new instance of the array
What is MPI?

- MPI is the de-facto standard for inter-node communication on distributed memory systems
- Uses explicit function calls and manual parallelization
- has been around for a long time (25+ years), will be around for a while longer
- updated regularly to address new hardware developments

- comprehensive design
  - point-to-point communication
  - collective communication
  - remote-DMA
  - (parallel) IO
  - memory management
  - process management
- available for all major programming languages
  - C / C++, Fortran
  - Python, Java, R, ...
Hello, MPI!

```fortran
program hello
use mpi
integer rank, ierr
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,&
    rank, ierr)
print *, 'Hello from ', rank
call MPI_FINALIZE(ierr)
end program hello
```

- `MPI_INIT`, `MPI_FINALIZE` should be first and last lines in code
- `mpif.h` (Fortran 77), `use mpi` (Fortran 90), and `mpi.h` (C / C++)
- all processes execute the same binary and code
- processes are assigned a unique rank number
- rank counts from 0 even in Fortran

```fortran
MPI_Comm_rank(MPI_Comm comm, int *rank)
MPI_Comm_size(MPI_Comm comm, int *size)
MPI_Barrier(MPI_Comm comm)
```
Compiling and running

ftn -o mpihello mpihello.f90
aprun -n 1 ./mpihello
aprun -n 4 ./mpihello

- compiler wrappers take care of linking against MPI runtime library: cc, CC, ftn or mpicc, mpicxx, mpif90
- run executable via aprun or mpirun
- on BlueWaters, MPI code can run only inside a job, not on login nodes

- each MPI rank corresponds to a UNIX process
- aprun, by default, packs processes densely onto nodes, does not oversubscribe nodes
- when one of the ranks terminates, aprun will terminate the whole run
- console output from all ranks is collected by aprun
- rank 0 receives aprun's input stream
Computing $\pi$
use Monte-Carlo integration to compute $\pi$ from the area of a disk

\[ A = \pi r^2 \]

// send random seed for MC integration
if (rank == 0)
    my_seed = seed;
MPI_Bcast(&my_seed, 1, MPI_INT, 0, MPI_COMM_WORLD);
seed = my_seed + rank;
...

// combine all rank results
MPI_Reduce(&local_inside,&global_inside, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
my_pi = 4.0*global_inside/total_points;
if (rank == 0)
    printf("pi is %g\n", my_pi);
MPI_Bcast, MPI_Reduce

- MPI_Bcast broadcasts data to all ranks
- MPI_Reduce combines data from all ranks to one rank
  - MPI_SUM, MPI_PROD
  - MPI_MIN, MPI_MAX
  - MPI_LAND, MPI_LOR, MPI_LXOR
- MPI_Allreduce bcast's result to all ranks
- Reductions of arrays are done horizontally (element-by-element)

- same MPI function for all data types, takes type argument to distinguish types
  - MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE (C)
  - MPI_INTEGER, MPI_REAL MPI_DOUBLE_PRECISION (Fortran)

MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
MPI in Python

- MPI bindings provided by `mpi4py` Python module
  - module load bwpy-mpi on BW contains MPI enabled packages
- deduces array type and size from numpy array information
- contains parallel `map` function useful for bag-of-jobs workloads
- backend for `ipyparallel` parallel notebooks

```
#!/usr/bin/python

import numpy as np
import mpi4py.MPI as MPI

rank = MPI.COMM_WORLD.Get_rank()
print ('Hello from %d' % rank)

...  
my_seed = np.empty(shape=1,dtype='i')
if (rank == 0):
    my_seed[:] = seed
MPI.COMM_WORLD.Bcast(my_seed,0)
```
Further reading

- XSEDE OpenMP training material https://www.psc.edu/hpc-workshop-series/openmp-august-2018
- OpenMP `task` construct, OpenMP `target` construct
- MPI specs: https://www.mpi-forum.org/
- MPI list of functions http://www.mpich.org/static/docs/latest/www/
- XSEDE MPI training material https://www.psc.edu/hpc-workshop-series/mpi
- MPI for grid based algorithms, e.g. the Laplace equation
- non-blocking communication in MPI
- example codes in same folder as slides
Extra material
OpenMP
Before we start

- if you would like to follow along for the examples you should log into Blue Waters twice
  
  - once normally to compile

```
ssh traXXX@bwbay.ncsa.illinois.edu
```
  
  - once starting and “interactive job” to run the aprun

```
ssh traXXX@bwbay.ncsa.illinois.edu
qsub -l walltime=2:0:0 -l nodes=1:xk:ppn=16 -q high -I
```
OpenMP memory model

- Memory is shared by default
- Variables residing in memory are shared by default
- Explicit lock / flush directives for synchronization
- Control accessibility of values via data-sharing attributes
  - private: each thread creates storage
  - shared: all threads share storage
  - default: specify sharing for variables not appearing in other attributes
OpenMP threading model

- OpenMP uses a (logical) fork-join model where sections not inside of a `omp parallel` execute on only a single thread
- Each join contains an implicit barrier to wait for all threads

- Data reductions are possible as part of the join
- Directives for
  - Critical sections (no concurrency)
  - Single sections (execute once only)
  - Atomic operations
Typical OpenMP code

- **omp for** is by far the most common OpenMP directive encountered in grid based codes

```c
#pragma omp parallel
{
    #pragma omp for
    for(int i=0; i<n; i++)
    {
        a[i] = b[i] + q*c[i];
    }
}
```

- Compiling the example code
  ```
  cd examples/
  cc -fopenmp -o stream stream.c
  ```

- Run the example code
  ```
  export OMP_NUM_THREADS=1
  aprun -n 1 -d 1 ./stream
  ```

  ```
  export OMP_NUM_THREADS=4
  aprun -n 1 -d 4 ./stream
  ```

- Compare timing results. Why is there so little speedup?
A slightly more complex example

double phi[n][n], mag_grad_phi[n][n];
const double dx = 0.25, dy = 0.25;
#pragma omp parallel for
for(int j=1; j<n-1; j++) {
    for(int i=1; i<n-1; i++) {
        double dx_phi = (phi[j][i+1] - phi[j][i-1])/(2*dx);
        double dy_phi = (phi[j+1][i] - phi[j-1][i])/(2*dy);
        mag_grad_phi[j][i] = sqrt(dx_phi*dx_phi + dy_phi*dy_phi);
    }
}

- Computes the magnitude of the gradient $|\nabla \phi|$ on a 2D grid
- Nested loops for x ($i$) and y ($j$) directions
- Private variables declared in `omp parallel region`
- Shared variables declared outside of `omp parallel region`
- Loops arranged to be most continuous in memory ($i$ last)
More on OpenMP clauses

- **collapse clause**
  
  ```c
  #pragma omp for collapse(2)
  for(k = 0 ; k < nz ; k++)
      for(j = 0 ; j < ny ; j++)
          for(i = 0 ; i < nx ; i++)
              a[k][j][i] = b[k][j][i];
  ```

- **simd clause**
  
  ```c
  #pragma omp simd
  for(i = 0 ; i < nx ; i++)
      a[k][j][i] = b[k][j][i];
  ```

- **reduction clause**
  
  ```c
  #pragma omp parallel for \
  reduction(+: sum)
  for(i = 0 ; i < nx ; i++)
      sum += b[k][j][i];
  ```

- **omp for** only parallelizes the next loop, for multi-d grids. collapse fuses multiple loops into one.

- **omp simd** asserts that SIMD code can be used for the next loop.

- **reductions** can be one of:
  - Min, max
  - +, -, *, &, |, ^, &&, ||

Belong with **omp parallel**
Revisiting the example

double phi[n][n], mag_grad_phi[n][n];
const double dx = 0.25, dy = 0.25;
double dx_phi, dy_phi

#pragma omp parallel for collapse(2) private(dx_phi, dy_phi) \  
private (i,j) shared(phi, mag_grad_phi)
for(j=1; j<n-1; j++) {
    for(i=1; i<n-1; i++) {
        dx_phi = (phi[j][i+1] - phi[j][i-1])/(2*dx);
        dy_phi = (phi[j+1][i] - phi[j-1][i])/(2*dy);
        mag_grad_phi[j][i] = sqrt(dx_phi*dx_phi + dy_phi*dy_phi);
    }
}

- only the collapse(2) actually changes anything
- private and shared (in this example) replicate previous behaviour
OpenMP parallel image processing

- edge detection using a Sobel filter
- implemented as a convolution

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \ast A \quad \text{and} \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \ast A$$

$$G = \sqrt{G_x^2 + G_y^2}$$

- multiple passes over data
  - Compute convolution
  - Normalize
  - Convert to output format

- illustrates
  - **private** variables
  - collapse clauses
  - reduction clauses
  - **Multiple parallel regions**
  - Timing functions available

- **source code in**: `sobel.c`
Sobel operator code

// apply Sobel edge detection to image
int i,j;
#pragma omp parallel for private(i,j) collapse(2)
for(j = 1 ; j < ny-1 ; j++) {
    for(i = 1 ; i < nx-1 ; i++) {
        // x direction
        int gx = 0;
        for(int jj = 0 ; jj < 3 ; jj++) {
            for(int ii = 0 ; ii < 3 ; ii++) {
                gx+=Gx[jj][ii]*data[j-1+jj][i-1+ii];
            }
        }
        // y direction
        int gy = 0;
        for(int jj = 0 ; jj < 3 ; jj++) {
            for(int ii = 0 ; ii < 3 ; ii++) {
                gy+=Gy[jj][ii]*data[j-1+jj][i-1+ii];
            }
        }
        sobel[j][i] = sqrt(gx*gx + gy*gy);
    }
}

// normalize result
float maxval = -HUGE_VAL;
float minval = +HUGE_VAL;
#pragma omp parallel for collapse(2) reduction(max: maxval) reduction(min: minval)
for(int j = 1 ; j < ny-1 ; j++) {
    for(int i = 1 ; i < nx-1 ; i++) {
        if(sobel[j][i] > maxval)
            maxval = sobel[j][i];
        if(sobel[j][i] < minval)
            minval = sobel[j][i];
    }
}
#pragma omp parallel for collapse(2)
for(int j = 1 ; j < ny-1 ; j++) {
    for(int i = 1 ; i < nx-1 ; i++) {
        edges[j][i] = (unsigned char) round(255*(sobel[j][i]-minval) / (maxval-minval));
    }
}
Running the sobel example

- Compiling the example code
  
  ```
cd examples/
make sobel
  ```

- Run the example code
  
  ```
export OMP_NUM_THREADS=1
aprun -n 1 -d 1 ./sobel

export OMP_NUM_THREADS=4
aprun -n 1 -d 4 ./sobel
  ```

- Obtain results
  
  ```
convert edges.pgm edges.png
rsync -Pe 'ssh traXXX@bwbay.ncsa.illinois.edu ssh' bw:examples/edges.png ./
  ```
Controlling parallelism

**master** executes block only on master thread

**single** executes block only on one, arbitrary thread. Use to initialize shared variables.

**critical** only one thread at a time can execute the block. Protects access to shared variables, or IO.

**atomic** perform the next, simple operation atomically

```c
int global_max = 0;
int global_count = 0;
float coeffs[100];
#pragma omp parallel
{
    #pragma omp single
    precompute(coeffs);
    
    #pragma omp critical
    if(mymax > global_max)
        global_max = mymax;
    
    #pragma omp atomic
    global_count += 1;
}
Scheduling loop iterations

```c
int n = 500000, not_primes=0;
#pragma omp parallel for \
    reduction(+: not_primes) \
    schedule(dynamic, 5)
for (int i = 2; i <= n; i++){
    for (int j = 2; j < i; j++){
        if (i % j == 0){
            not_primes++;
            break;
        }
    }
}
printf("Primes: %d\n",
    n - not_primes);
```

- **static**: statically divides iterations among threads
- **dynamic**: each thread requests a new chunk once it finishes
- **guided**: each thread operates on fraction of remaining iterations
- **runtime**: decide based on OMP_SCHEDULE env variable
Running the primes example

- Compiling the example code
  cd examples/
  make primes

- Run the example code
  export OMP_NUM_THREADS=1
  aprun -n 1 -d ./primes
  Primes: 41539
  Took 26327.7 millisecond

  export OMP_NUM_THREADS=4
  aprun -n 1 -d ./primes
  Primes: 41539
  Took 10168.1 milliseconds
OpenMP task constructs

- similar to C++11 threads + Lambdas
  - any thread can spin off tasks
  - any thread can pick up a task
  - tasks are inexpensive, have many more than threads
  - explicit taskwait to join all tasks
- useful for
  - handling graphs, recursively defined data structures, recursion relations
  - task-based parallelism where the parallel workload is non-uniform and for loops are not sufficient
- not useful for
  - concurrency, tasks end at end of omp parallel region

```c
int fib(int n) {
    int i, j;
    if(n<2) {return n;} else {
        #pragma omp task shared(i) firstprivate(n)
        i=fib(n-1);
        #pragma omp task shared(j) firstprivate(n)
        j=fib(n-2);
        #pragma omp taskwait
        return i+j;
    }
}

int main(void) {
    int n = 30;
    #pragma omp parallel
    {
        #pragma omp single
        printf("fib(%d)=%d \n",n,fib(n));
    }
    return 0;
}
```
Finding the most common word in Hamlet

- each word is stored as a node in a binary tree structure

```c
struct node {
    struct node *left, *right;
    char *word;
    int count;
};
```

- search algorithm
  - start at root node
  - pick most frequent word from
    - node itself
    - left branch ← recursion
    - right branch ← recursion

```c
findMostCommonNode(struct node *this) {
    struct node *left, *right;
    left = findMostCommonNode(this->left);
    right = findMostCommonNode(this->right);
    struct node *retval = this;
    if(left->count > this->count)
        retval = left;
    if(right->count > this->count)
        retval = right;
    return retval;
}
```

(C) morshin khan
https://cslearners.blogspot.com/
Graph traversal using tasks

```c
struct node *
findMostCommonNode(struct node *this) {
    if(this == NULL) return NULL;

    struct node *left_most_common, *right_most_common;
    #pragma omp task shared(left_most_common) firstprivate(this)
    left_most_common = findMostCommonNode(this->left);
    #pragma omp task shared(right_most_common) firstprivate(this)
    right_most_common = findMostCommonNode(this->right);
    #pragma omp taskwait
    struct node *retval = this;
    if(left_most_common && left_most_common->count > this->count)
        retval = left_most_common;
    if(right_most_common && right_most_common->count > this->count)
        retval = right_most_common;
    return retval;
}

int main(void) {
    [...]  
    #pragma omp parallel
    {
        #pragma omp single maxfreq
        maxfreq = findMostCommonNode(root);
    }
    printf("The most common word in %s is '%s' with %d occurrences\n", fn, maxfreq->word, maxfreq->count);
}
```

- nodes form a binary tree of words found
- each node processed by a task

```
OMP_NUMT_THREADS=1 ./wordcount
Took 1.12209 ms to compute

OMP_NUMT_THREADS=2 ./wordcount
Took 4.13777 ms to compute
```
OpenMP runtime

Environment variables

- OMP_NUM_THREADS per process
- OMP_STACK_SIZE for each thread

Runtime library

- header file omp.h
  - defines _OPENMP preprocessor variable
- omp_get_num_threads() number of threads
- omp_get_thread_num() thread id
- omp_get_wtime() current time

```c
#include <omp.h>
int main(int c, char **v){
    double start = omp_get_wtime();
    #pragma omp parallel
    {
        int mythread = omp_get_thread_num();
        int numthreads = omp_get_num_threads();
        printf("Hello, world from "
               "thread %d of %d!\n",
               mythread, numthreads);
    }
    double end = omp_get_wtime();
    printf("Took %gs\n", end-start);
    return 0;
}
```
Key concepts

- MPI uses communicators to group ranks
  - MPI_COMM_WORLD
  - new subgroups can be created
- calls to MPI routines must match in all involved MPI ranks
  - deadlocks (may) occur if violated
  - this does include the data type used
- messages from the same rank are received in the order sent
  - explicit synchronization otherwise

- MPI operates on values, not byte streams
- MPI calls come in three flavors
  - collective calls
  - point-to-point communication
  - remote-memory access
- one-to-many and many-to-one communications designate a “root” rank
- MPI calls do not wait for the receiver to confirm receipt
Assigning work when all are equal...

- all ranks execute the same code
- unless explicitly request, no synchronization takes place
- they are distinguishable **only** by their rank number

```plaintext
if (my_pe_num .eq. 0) then
call Routine_SpaceInvaders
else if (my_pe_num .eq. 1) then
call Routine_CrackPasswords
else
  call Routine_WeatherForecast
end if
```

- more commonly use rank to distribute spatial domain
- or have one controller (rank 0) assign work to multiple workers
- MPI even offers help to choose work based on the physical location of the rank in the network
- for many problems load-balancing becomes an issue
Passing the buck – how to serialize code

```c
int main(int argc, char **argv) {
    int rank, sz;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &sz);
    for(int i = 0 ; i < sz ; i++){
        if(i == rank)
            printf("Hello from %d
", rank);
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

- serializing code is useful to
  - disentangle output to screen or file
  - reduce load on IO system when reading or writing data
  - manipulating global state (typically files)
  - reduce required buffers for many-to-one communication

- code shown is not very scalable
  - size^2 communications
  - run time increases linearly with size

- can be sped up by passing a “token” (or “baton” or the “buck”) to your neighbour once done
  - only size communications
  - same runtime
π output

cd examples
make pi

aprun -n 1 ./pi

pi is 3.1415185555555554
real pi is 3.1415926535897931 diff 7.4098034237746191E-005
Took 1392.6311291288584 milliseconds

aprun -n 3 ./pi

pi is 3.1414653333333336
real pi is 3.1415926535897931 diff 1.2732025645956213E-004
Took 665.79209803603590 milliseconds
Point-to-point communication

- **MPI provides** `MPI_Send` and `MPI_Recv` communication primitives

```c
if (rank.eq.0) then
  call MPI_Recv(recv, 1, MPI_INTEGER, &
                MPI_ANY_SOURCE, MPI_ANY_TAG, &
                MPI_COMM_WORLD, status, ierr)
  print *, 'received:', recv
else
  send = 4
  call MPI_Send(send, 1, MPI_INTEGER, &
                0, tag, MPI_COMM_WORLD, ierr)
endif
```

- can be used to build arbitrary communication patterns

```c
MPI_Send(void *buf, int count,
         MPI_Datatype datatype,
         int dest, int tag,
         MPI_Comm comm)

MPI_Recv(void *buf, int count,
         MPI_Datatype datatype,
         int source, int tag,
         MPI_Comm comm,
         MPI_Status *status)

typedef struct {
  int MPI_SOURCE, MPI_TAG,
  MPI_ERROR;
} MPI_Status;
```
Manager - worker schemes using MPI

- one rank, typically rank 0, is designated the manager and hands out work orders to the workers
- workers process each order and return result to master
- master collects the results
- use special “terminate” order to signal no-more work to workers

```c
MPI_Send(void *buf, int count,
         MPI_Datatype datatype,
         int dest, int tag,
         MPI_Comm comm)

MPI_Recv(void *buf, int count,
         MPI_Datatype datatype,
         int source, int tag,
         MPI_Comm comm,
         MPI_Status *status)

typedef struct {
    int MPI_SOURCE, MPI_TAG,
    MPI_ERROR;
} MPI_Status;
```
Master - worker schemes using MPI

- one rank, typically rank 0, is designated the master and hands out work orders to the workers
- workers process each order and return result to master
- master aggregates the results

cd examples
make worker
aprun -n 3 ./worker

PE 1 's result is 4.00000000
PE 2 's result is 8.00000000
Total is 12.0000000

if (rank .eq. 0) then
    numbertosend = 4
    do i=1,sz-1
        call MPI_Send(numbertosend, 1, MPI_REAL, i, dummy,&
                    MPI_COMM_WORLD, ierr)
    end do
else
    call MPI_Recv(numbertoreceive, 1, MPI_REAL, 0, &
                  MPI_ANY_TAG, MPI_COMM_WORLD, stat, ierr)
    result = numbertoreceive * rank
end if

do i=1,sz-1
    if (i .eq. rank) then
        print *, "PE ",rank,"'s result is ", result
    end if
    call MPI_Barrier(MPI_COMM_WORLD, ierr)
end do

if (rank .eq. 0) then
    result = 0
    do i=1,sz-1
        call MPI_Recv(numbertoreceive, 1, MPI_REAL, i, &
                      MPI_ANY_TAG, MPI_COMM_WORLD, stat, ierr)
        result = result + numbertoreceive
    end do
    print *, "Total is ", result
else
    call MPI_Send(result, 1, MPI_REAL, 0, dummy, &
                   MPI_COMM_WORLD, ierr)
end if
Scatter and gather data

- **MPI_Scatter** and **MPI_Gather** are similar to **MPI_Bcast** and **MPI_Reduce**
- scatter or gather the components of an array among the MPI ranks
- **MPI_Allgather** broadcasts the resulting array to all ranks (like **MPI_Allreduce**)
- useful if one rank prepares work assignments then scatters them to the others

```
MPI_Scatter(const void *sendbuf,
            int sendcount,
            MPI_Datatype sendtype,
            void *recvbuf, int recvcount,
            MPI_Datatype recvtype,
            int root, MPI_Comm comm)

MPI_Gather(const void *sendbuf,
           int sendcount,
           MPI_Datatype sendtype,
           void *recvbuf, int recvcount,
           MPI_Datatype recvtype,
           int root, MPI_Comm comm)

MPI_Allgather(
    const void *sendbuf,
    int sendcount,
    MPI_Datatype sendtype,
    void *recvbuf, int recvcount,
    MPI_Datatype recvtype,
    int root, MPI_Comm comm)
```
MPI_Gather, MPI_Scatter examples

double myvals[2];
if(rank == 0) {
    double values[2*sz];
    for(int i = 0; i < 2*sz; i++)
        values[i] = i;
    MPI_Scatter(values, 2,
                MPI_DOUBLE, myvals, 2,
                MPI_DOUBLE, 0, MPI_COMM_WORLD);
} else {
    MPI_Scatter(NULL, 0,
                MPI_DOUBLE, myvals, 2,
                MPI_DOUBLE, 0, MPI_COMM_WORLD);
}

• large arrays only exist on root rank
  - length of array must be multiple of sz
  - use MPI_Scatterv otherwise (complicated)

• and similar for gather...

cd examples
make gather

aprun -n 1 ./gather
got squares:
0^2=0 1^2=1

aprun -n 4 ./gather
got squares:
0^2=0 1^2=1 2^2=4 3^2=9 4^2=16
5^2=25 6^2=36 7^2=49
Why not use MPI?

- You will likely have to rewrite portions in all areas of your code.
  - Old, dusty subroutines written by a long-departed grad student.

- You will have to understand almost all of your code.
  - Old, dusty subroutines written by a long-departed grad student.

- You can’t do it incrementally.
  - Major data structures have to be decomposed up front.

- Debugging will be “different”.
  - You aren’t just finding the bad line of code. You sometimes need to find the bad PE.

taken from https://www.psc.edu/hpc-workshop-series/mpi