OpenMP and MPI parallelization

http://tiny.cc/ncsa-pire

Roland Haas (NCSA / University of Illinois)
Email: rhaas@ncsa.illinois.edu

ILLINOIS

NCSA | National Center for Supercomputing Applications

OpenMP standard

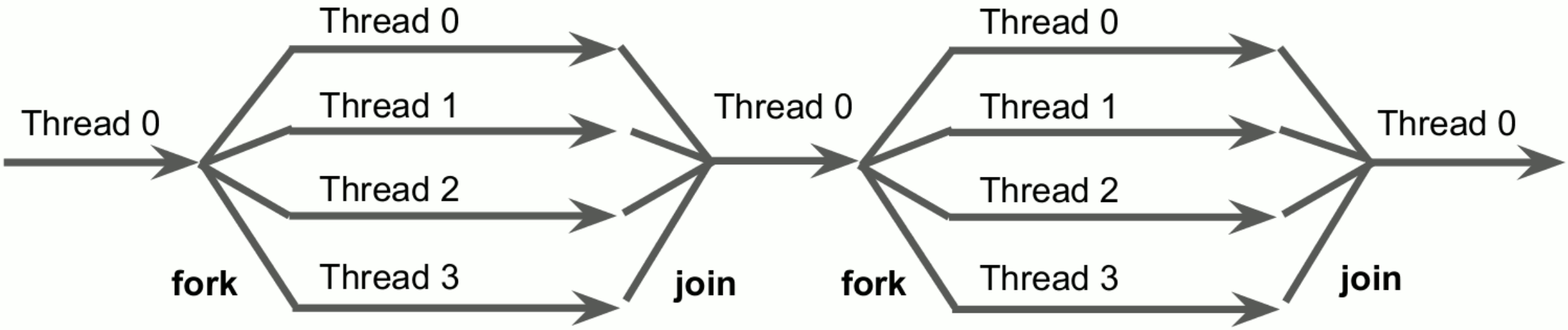
- OpenMP is a language extension for C/C++/Fortran to develop shared-memory, thread based algorithms
- Current version is OpenMP 5.0 from 2018 http://openmp.org
- Targets CPU and accelerator code
- Multiple parallelism schemes
  - Domain decomposition / data parallelism
  - Task parallelism
  - Anything in between using lock primitives
- Supported by all major compiler vendors: GNU, Intel, PGI, MS Visual Studio, Cray, ...
- Designed to be easily added to legacy code
- High level approach hiding pthreads and hardware specifics
- Single source, all code is one stream of source file, no breakout routines
- Compilers ignore OpenMP directives if OpenMP is not supported or unknown
- Hide complexity of threading / accelerator libraries from user

```
#pragma omp parallel for
for(i=1; i<n-1; i++) {
  a[i] = b[i+1] - b[i-1];
}
```

OpenMP threading model

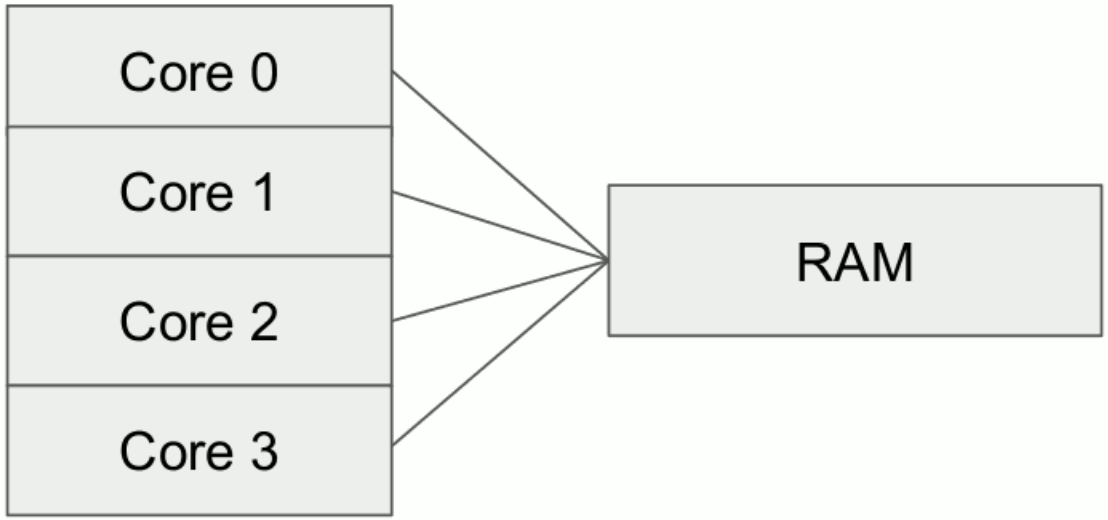
- OpenMP uses a (logical) fork-join model where sections not inside of a an "`omp parallel`" execute on only a single thread
- Usually build on top of pthreads, optimized for multiple short lived tasks
- Each join contains an implicit barrier to wait for all threads

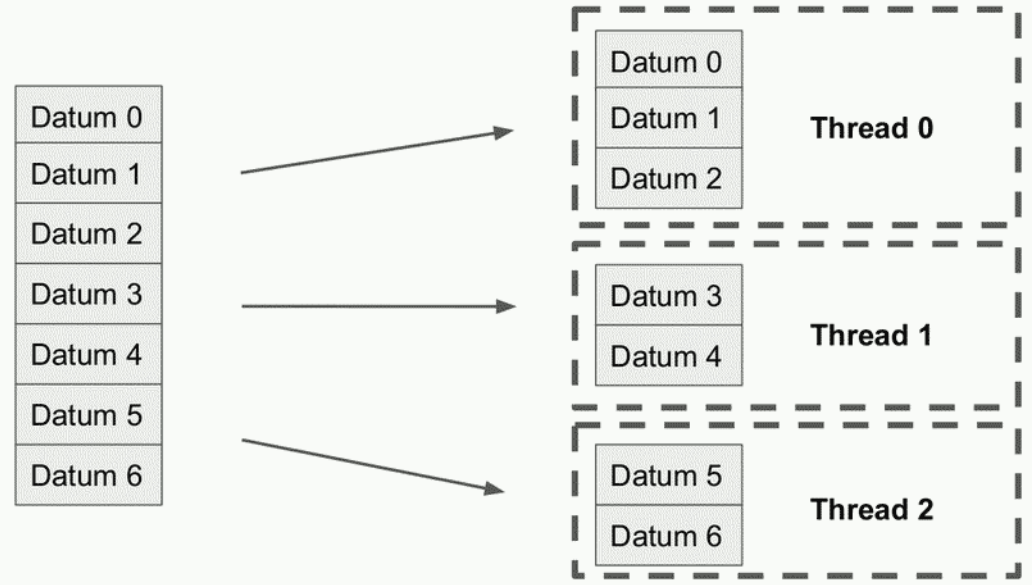
- Data reductions are possible as part of the join
- Directives for
  - `critical` sections (no concurrency)
  - `single` sections (execute once only)
  - `atomic` operations



OpenMP memory model

- Variables residing in memory are (usually) shared by default
- OpenMP defines explicit synchronization points at which point variables are flushed to memory
- Explicit `lock` / `flush` directives for synchronization (will not cover)
- `atomic` operations to safely update shared variables
- 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 Hello world

```
int main(int argc, char **argv)
{
 #pragma omp parallel
 {
  printf("Hello, world!\n");
 }
 return 0;
}
```

- Compile using

```
cd examples
gcc -fopenmp -o hello hello.c
```

- Run via

```
export OMP_NUM_THREADS=4
./hello
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

- OpenMP pragma control parallelism
- Compilers ignore unknown pragmas so typos generate only warnings!
  `#pragma openmp parallel`
- OpenMP code is compiled by passing `-fopenmp` (GNU, newer versions of Intel), `-openmp` (older versions of Intel), `/openmp` (MSVC), `-mp` (PGI)
- Control how many threads are used via the `OMP_NUM_THREADS` environment variable
- No special starter program required to run code
- Can oversubscribe number of cores, but this is very slow

General directive layout

```
#pragma omp parallel [clause …]
{
 Structured block
 #pragma omp [workshare construct] [clause …]
 {
  Structured block
 }
}
```

- A parallel directive starts a parallel code block
- Inside the `parallel` section **worksharing** constructs indicate how threads split up the available work
  - `omp for`: parallel loops
  - `omp single`: serial section
  - `omp critical [(name)]`: protect shared variables, can be given a name for finer grained control
- omp clauses control eg. data sharing
  - `private(var1, var2)`
  - `shared(var1, var2)`
  - `firstprivate(var1, var2)`
- or how loops are split
  - `schedule(dynamic)`
  - `schedule(ordered)`
  - `schedule(static)`

Typical OpenMP code

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

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

- this is the stream benchmark used to measure memory bandwidth
- `parallel for` creates threads which each operate on subset of the `i` range
- `a`, `b`, `c` as well as `q` and `n` are shared among the threads, but each has a private `i` counter

- Compiling the example code

```
cd examples/
gcc -fopenmp -o stream stream.c
```

- Run the example code

```
export OMP_NUM_THREADS=1
./stream

export OMP_NUM_THREADS=2
./stream

export OMP_NUM_THREADS=4
./stream
```

- Compare timing results. Why is there so little speedup beyond 2 threads?
  - Does adding `-O3` help?
  - At which value of `OMP_NUM_THREADS` does scaling break down?

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

```
#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

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

- `reduction` clause

```
#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
- `reduction`s can be one of
  - min, max
  - +, -, *, &, |, ^, &&, ||

  belong with `omp parallel`

Data sharing clauses

- make all variables private

```
#pragma omp parallel default(private)
```

- explicitly share variables

```
#pragma omp parallel shared(data_ptr)
```

- explicitly make variables private

```
#pragma omp parallel private(i,j)
```

- make a private copy of a variable

```
#pragma omp parallel firstprivate(arg)
```

- or declare in block

```
#pragma omp parallel
{
  double tempvar;
}
```

- persistent per thread variables

```
static void *state = NULL;
#pragma omp threadprivate(state)
```

- 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++
- `firstprivate` useful for `omp tasks` (more on this later)
- making a pointer private, still keeps he pointed to memory shared
  - making an array private creates a new instance of the array

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

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

$$\mathbf{G} = \sqrt{\mathbf{G}_x^{\,2} + \mathbf{G}_y^{\,2}}$$

- which is basically a stencil operation taking a gradient

- multiple passes over data
  - Compute convolution
  - Normalize
  - Convert to output format
- illustrates
  - `private` variables
  - `collapse` clauses
  - `reduction` clauses
  - multiple `parallel` regions
- scales well to ~4 threads
- output written to file edges.pgm

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));
 }
}
```

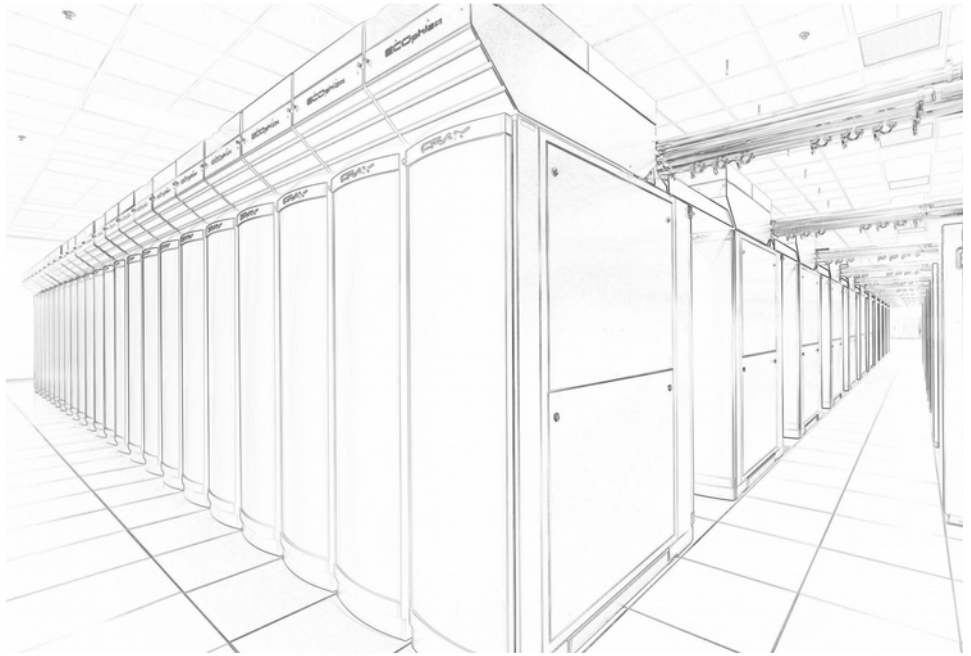
Running the sobel example

- Compiling the example code

```
cd examples/
make sobel
```

- Run the example code

```
export OMP_NUM_THREADS=1
./sobel

export OMP_NUM_THREADS=4
./sobel
```



- View results

```
convert edges.pgm edges.png
scp $USER@pire-vm:ncsa-pire/examples/edges.png
open edges.png
```

Controlling parallelism

`master`	executes block only on master thread
`single`	executes block only on one, arbitraty 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

```
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

```
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, can specify chunk size
`guided`	each thread operates on fraction of **remaining** iterations
`runtime`	decide based on OMP_SCHEDULE env variable

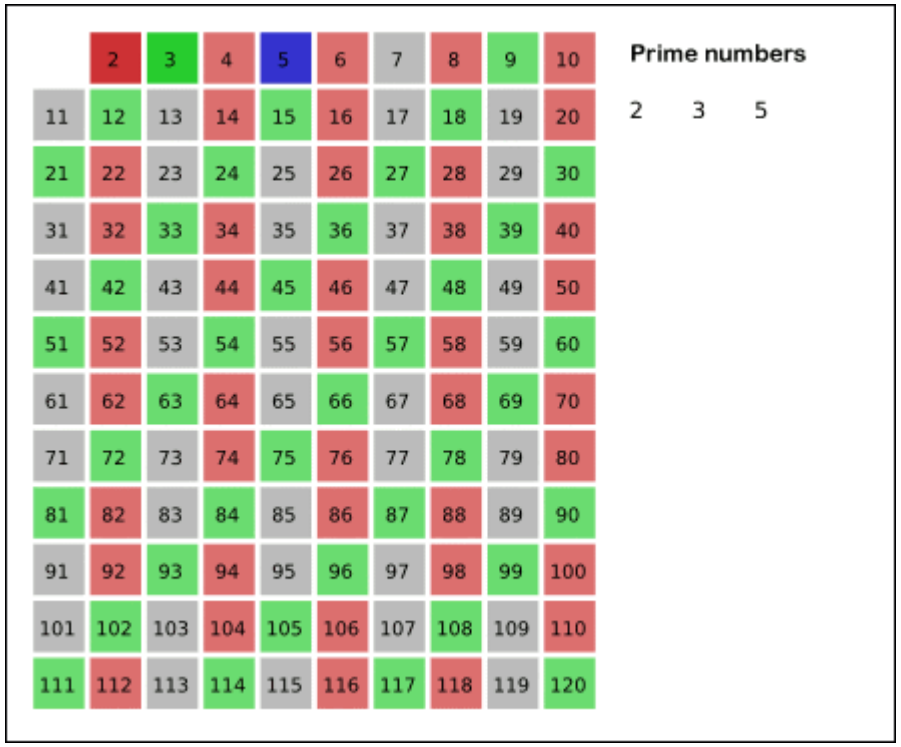
Running the primes example

- Compiling the example code

```
cd examples/
make primes
```

- Run the example code

```
export OMP_NUM_THREADS=1
./primes

Primes: 41539
Took 26327.7 millisecond

export OMP_NUM_THREADS=2
./primes

Primes: 41539
Took 10168.1 milliseconds
```



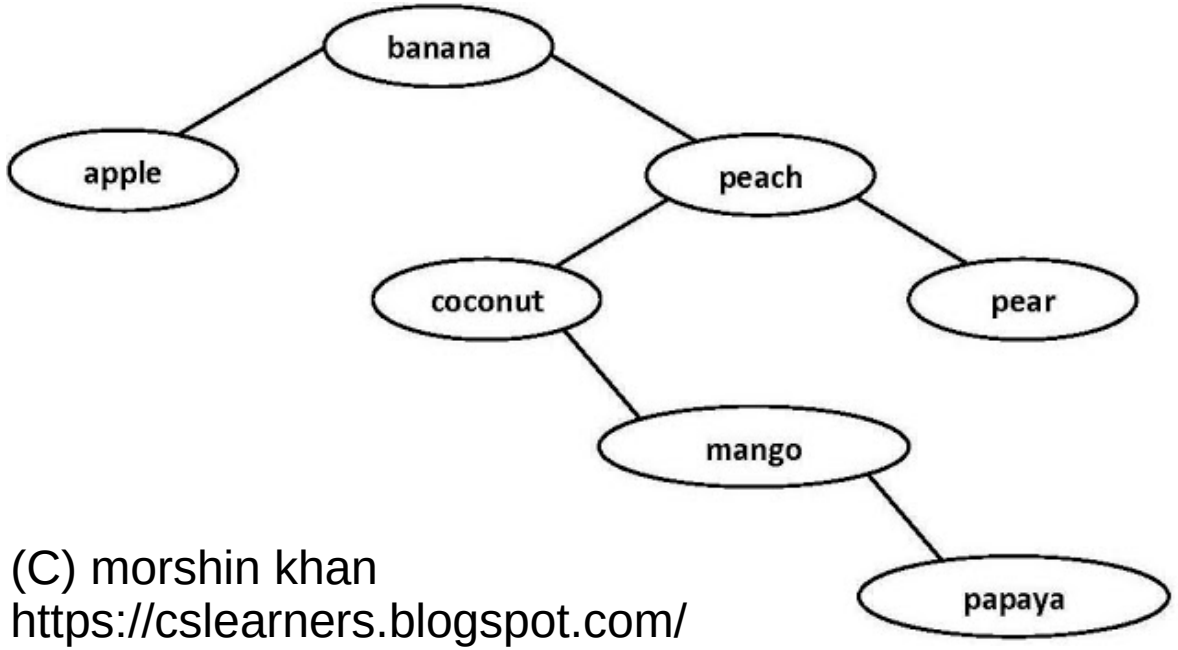
SKopp at German Wikipedia CC BY-SA 3.0

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 no sufficient
- not useful for
  - concurrency, tasks end at end of `omp parallel` region

```
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



(C) morshin khan
https://cslearners.blogspot.com/

```
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;
}
```

- count frequencies of words in Hamlet
- each word is stored as a node in a binary tree structure

```
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

Graph traversal using tasks

```
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 = findMostCommonNode(root);
 }
 printf("The most common word in %s is '%s'"
        " with %d occurances\n", fn,
        maxfreq->word, maxfreq->count);
}
```

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

```
cd examples
make wordcount

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

- we already used `OMP_NUM_THREADS` to control the number of threads used by the code
- `OMP_STACK_SIZE` controls the stack available to store local variables, SEGFAULTs happen if it is too small

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()` the current time

```
#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 %g s\n",
  end-start);
 return 0;
}
```

Further reading

- OpenMP 5.0 specs: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
- XSEDE OpenMP training material https://www.psc.edu/hpc-workshop-series/openmp-august-2018

OpenMP and MPI parallelization

http://tiny.cc/ncsa-pire

Roland Haas (NCSA / University of Illinois)
Email: rhaas@ncsa.illinois.edu

ILLINOIS

NCSA | National Center for Supercomputing Applications

What is MPI?

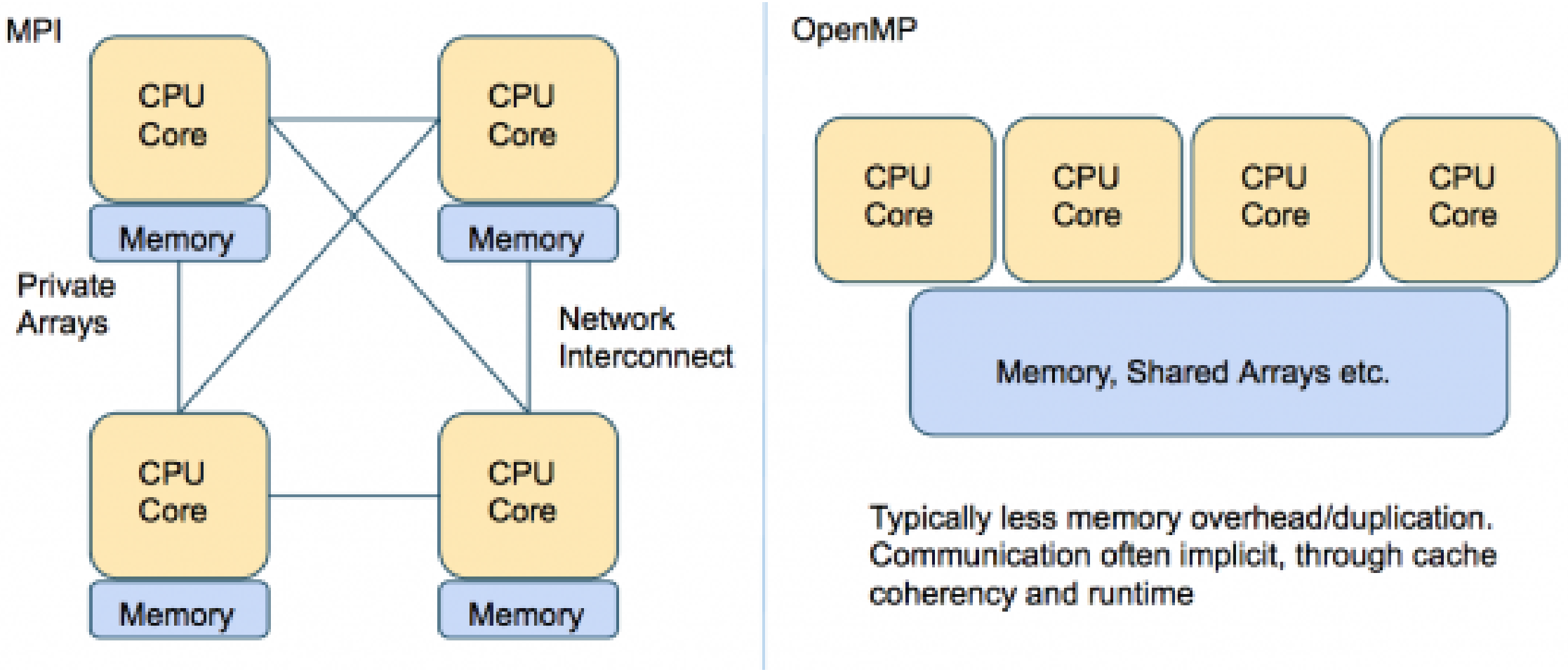
- MPI is the de-facto standard for inter-node communication on distributed memory systems
- uses explicit function calls and manual parallelization
- has been around for a long time (20+ years), will be around for a while longer
- updated regularly to address new hardware developments

- 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, ...

C vs. python code

- MPI was originally designed with compiled languages (Fortran 77, C) in mind
  - MPI is (mostly) object oriented, (almost) all identifiers are for opaque objects, but uses a function based interface
  - all MPI routines, preprocessor constants and data types start with `MPI_` to form a namespace
  - all functions operate on arrays, taking an array pointer, an array length and a symbolic constant describing the array datatype as arguments
- mpi4py provides pythonic interface to MPI
  - fully object oriented
  - deduce array type and length
  - provide defaults for parameters
- use python to prototype algorithm, C for production
- mpi4py provides SWIG bindings to interact with C code
  - pass communicators between languages
  - can mix C / python code in communicatioin
- multiple books cover C / Fortran API (Amazon finds 595 books for MPI)
- standards documents (complex) https://www.mpi-forum.org/docs/
- MPI API reference https://www.open-mpi.org/doc/v4.0/
- mpi4py docs at readthedocs https://mpi4py.readthedocs.io/en/stable/

Distributed vs. shared memory



http://www.nersc.gov/assets/Uploads/_resampled/ResizedImage540228-MPIVSOPENMP.png

Hello, MPI!

C code

```c
#include <mpi.h>
int main(int argc, char **argv){
 int rank;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD,
               &rank);
 printf("Hello from %d\n", rank);
 MPI_Finalize();
}
```

Python code

```python
import mpi4py.MPI as MPI
rank = MPI.COMM_WORLD.Get_rank()
print ("Hello from %d" % rank)
```

- MPI_Init, MPI_Finalize should be first and last lines in code
- (almost) all MPI functions return a status code `ierr` which is typically ignored
- MPI is **defined** in its **standard**, not via a canonical implementation

- all processes execute the same binary and code
- processes are assigned a unique `rank` unique among all nodes in a job
- `mpi.h` provides C prototypes
- `import mpi4py` for python

Compiling and running

```
cd examples
mpicc -o mpihello mpihello.c

mpirun -n 1 ./mpihello

mpirun -n 4 ./mpihello

mpirun -n 1 python mpihello.py

mpirun -n 4 python mpihello.py
```

- compiler wrappers take care of linking against MPI runtime library
  - mpicc, mpicxx, mpif90 (usually)
  - mpiicc, mpiicpc, mpiifort (Intel compiler)
  - cc, CC, ftn (Cray machines...)

- executable must be run via mpirun
  - some MPI stacks (OpenMPI) let you run a serial version without using mpirun
  - some cluster won't let you run on the login nodes
- each MPI rank corresponds to a UNIX process
- when one of the ranks terminates, (usually) mpirun will terminate the whole run
- console output from all ranks is collected by mpirun
- rank 0 receives mpirun's input stream

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 and length of arrays used
- messages from the **same** rank are received in the order sent
  - between ranks, explicit synchronization is required
- messages can be tagged by a (small) integer and cherry-picked
- MPI operates on data, not byte streams
- MPI calls come in two 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
  - Send calls return to sender when last bit of data is handed over to the network
  - use `MPI_Barrier` to synchronize ranks

Assigning work when all are equal...

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

```
if (my_pe_num == 0)
  Routine_SpaceInvaders();
else if (my_pe_num == 1)
  Routine_CrackPasswords();
else
  Routine_WeatherForecast();
```

- 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

```
MPI_Comm_rank(MPI_Comm comm,
 int *rank)

MPI_Comm_size(MPI_Comm comm,
 int *size)

MPI_Barrier(MPI_Comm comm)
```

Passing the buck – how to serialize code

```
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\n",
    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

Running the serialization example

```
cd examples
make serialmpi mpihello

export OMPI_MCA_rmaps_base_oversubscribe=yes

mpirun -n 1 ./mpihello

mpirun -n 12 ./mpihello

mpirun -n 1 ./serialmpi

mpirun -n 12 ./serialmpi
```

- randomness of output is more obvious if one runs more MPI ranks than there are CPUs
  - need to allow OpenMPI to do so (some clusters may never let you)

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
- same MPI function for all data types, takes type argument to distinguish types
  - `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE` (C)
  - python deduces type and length automatically
- Reductions of arrays are done **horizontally** (element-by-element)

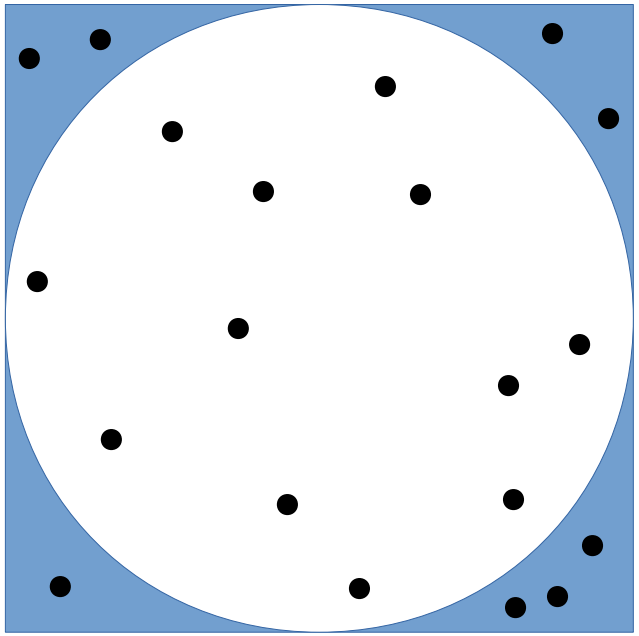
```
MPI_Bcast(void *buffer,
 int count,
 MPI_Datatype datatype,
 int root,
 MPI_Comm comm)
```

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

Computing π

- use Monte-Carlo integration to compute π from the area of a disk

$$A = \pi r^2$$



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

…

// 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) /
 (sz * local_points);

if (rank == 0)
 printf("pi is %g\n", my_pi);
```

Computing π output

```
cd examples
make pi

mpirun -n 1 ./pi

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

mpirun -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` as communication primitives

```
float numbertosend = 4.0, numbertoreceive = -1;

if (rank == 0)
 MPI_Recv(&numbertoreceive, 1, MPI_FLOAT, MPI_ANY_SOURCE,
  MPI_ANY_TAG, MPI_COMM_WORLD, &status);

if (rank == 1)
  MPI_Send(&numbertosend, 1, MPI_FLOAT, 0, dummy_tag,
   MPI_COMM_WORLD);
```

- can be used to build arbitrary communication patterns

```
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

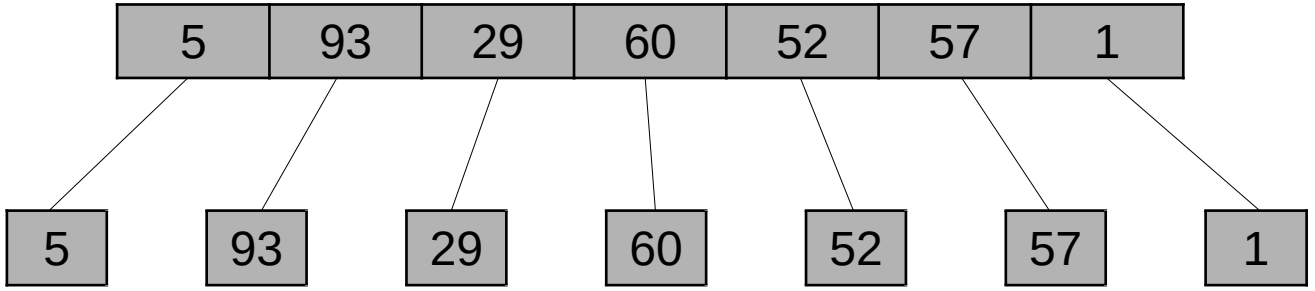
```
cd examples
make worker

mpirun -n 3 ./worker

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

```
if (rank == 0) {
 float numbertosend = 4;
 for(i=1 ; i<sz ; i++) {
  MPI_Send(&numbertosend, 1, MPI_FLOAT, i,
  dummy_tag, MPI_COMM_WORLD);
 }
} else {
 MPI_Recv(&numbertoreceive, 1, MPI_FLOAT,
  0, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
 result = numbertoreceive * rank;
}
```

```
if (rank == 0) {
 float result = 0;
 for(i=1 ; i<sz ; i++) {
  MPI_Recv(&numbertoreceive, 1, MPI_FLOAT,
   MPI_ANY_SOURCE, MPI_ANY_TAG,
   MPI_COMM_WORLD, &stat);
  result += numbertoreceive;
 }
 printf("Total is %f\n", result);
} else {
 MPI_Send(&result, 1, MPI_FLOAT, 0,
  dummy_tag, MPI_COMM_WORLD);
}
```

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,
 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);
}
```

```
cd examples
make gather

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

mpirun -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
```

- large arrays only exist on root rank
  - length of array must be multiple of `sz`
  - use `MPI_Scatterv` otherwise (complicated)
- and similar for gather...

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

Further reading

- more MPI functions
  - non-blocking communication
  - multiple communicators
  - MPI shared memory
  - custom data types
  - MPI-IO
- multiple books cover C / Fortran API (Amazon finds 595 books for MPI)
- standards documents (complex) https://www.mpi-forum.org/docs/
- MPI API reference https://www.open-mpi.org/doc/v4.0/
- mpi4py docs at readthedocs https://mpi4py.readthedocs.io/en/stable/
- XSEDE MPI training material https://www.psc.edu/hpc-workshop-series/mpi
- MPI for grid based algorithms, e.g. the Laplace equation

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

ILLINOIS

NCSA | National Center for Supercomputing Applications

OpenMP task constructs

- `omp task` spins off a new task
  - variables are `private` by default
  - use `firstprivate` to pass arguments, `shared` causes in a race condition
  - use `shared` to return values
- `omp single` to spin off recursion
- `omp taskwait` waits for immediate child tasks to finish
- `omp critical` with a name if shared variables are modified
- `omp taskgroup` if result of tasks is not required
- many more clauses and constructs
  - `taskyield`
  - `final, untied, mergable, depend`

- Compiling the example code

```
cd examples/
make fib
```

- Run the example code

```
export OMP_NUM_THREADS=1
./fib

fib(30)=832040
Took 248.556 ms

export OMP_NUM_THREADS=2
./fib

fib(30)=832040
Took 1036.49 ms
```

- this is indeed slower in parallel