MPI Usage and Techniques

Programming Models for Emerging Platforms

MPI Standard

- Just about anything can be built with 6 MPI functions
  - MPI_Init
  - MPI_Comm_rank
  - MPI_Comm_size
  - MPI_Recv
  - MPI_Send
  - MPI_Finalize

MPI Standard

- Just about anything can be built with 6 MPI functions
  - MPI_Init
  - MPI_Comm_rank
  - MPI_Comm_size
  - **MPI_Recv**
  - **MPI_Send**
  - MPI_Finalize

MPI Standard

- Just about anything can be built with 6 MPI functions
  - MPI_Init
  - MPI_Comm_rank
  - MPI_Comm_size
  - **MPI_Recv**
  - **MPI_Send**
  - MPI_Finalize
- Speaks to the simplicity of MPI design

Let’s build a “parallel” sort with MPI

```c
#define TAG_DATA 0
#define TAG_WORK 1
#define NINTS 1024

void send_sort(int rank, int size) {
  int tosort[NINTS];
  for (int i = 0; i < NINTS; i++) {
    tosort[i] = NINTS - i - 1;
  }

  int split = NINTS / (size - 1);
  for (int i = 1; i < size; i++) {
    int start = (split * (i-1));
    MPI_Send(tosort + start, split, MPI_INT, i,
     TAG_DATA, MPI_COMM_WORLD);
  }

  int **bufs = malloc(sizeof(int*) * (size-1));
  int *mi = malloc(sizeof(int) * (size-1));
  for (int i = 0; i < size-1; i++) {
    bufs[i] = malloc(sizeof(int) * split);
    mi[i] = 0;
  }

  for (int i = 1; i < size; i++) {
    MPI_Recv(bufs[i-1], split, MPI_INT, i, TAG_DATA,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }

  // merge
```

Distributing work for sort

Gathering sorted “chunks”

```c
int main(…) {
  // ...
  if (rank == 0) {
    send_sort(rank, size);
  } else {
    do_sort(rank, size);
  }
  // …
}
```

```c
#define TAG_DATA 0
#define TAG_WORK 1
#define NINTS 1024

int int_compare(const void *e1, const void *e2) {
  return (*(int*)e1) - (*(int*)e2);
}

void do_sort(int rank, int size) {
  int split = NINTS / (size - 1);
  int *buf = malloc(sizeof(int) * split);

  MPI_Recv(buf, split, MPI_INT, 0, TAG_DATA,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  qsort(buf, split, sizeof(int), int_compare);
  MPI_Send(buf, split, MPI_INT, 0, TAG_DATA,
    MPI_COMM_WORLD);
}
```

C int comparator

C “generic” quick sort

```c
int main(…) {
  // ...
  if (rank == 0) {
    send_sort(rank, size);
  } else {
    do_sort(rank, size);
  }
  // …
}
```

```
#define TAG_DATA 0
#define TAG_WORK 1
#define NINTS 1024

void send_sort(int rank, int size) {
  // ...

  // merge

  int ni = 0;
  while (ni < NINTS) {
    // Find minimum val and index
    int minv = INT_MAX;
    int mini = -1;
    for (int i = 1; i < size; i++) {
      if (mi[i-1] < split && bufs[i-1][mi[i-1]] < minv) {
        minv = bufs[i-1][mi[i-1]];
        mini = i;
      }
    }
    tosort[ni] = bufs[mini-1][mi[mini-1]];
    mi[mini-1]++;
    ni++;
  }
```

Next “merging” index for sorted array

Arbitrary number of processes, have to find the “min” (instead of just left, right etc)

i-1 is an unfortunate side effect of index / rank mismatch

Assuming number of items to sort is very large, do we have bottlenecks?

```c
void send_sort(int rank, int size) {
  // ...
  int split = NINTS / (size - 1);
  for (int i = 1; i < size; i++) {
    int start = (split * (i-1));
    MPI_Send(tosort + start, split, MPI_INT, i,
     TAG_DATA, MPI_COMM_WORLD);
  }
  // ...
  for (int i = 1; i < size; i++) {
    MPI_Recv(bufs[i-1], split, MPI_INT, i, TAG_DATA,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }

  int ni = 0;
  while (ni < NINTS) {
    // Find minimum val and index
    int minv = INT_MAX;
    int mini = -1;
    for (int i = 1; i < size; i++) {
      if (mi[i-1] < split && bufs[i-1][mi[i-1]] < minv) {
        minv = bufs[i-1][mi[i-1]];
        mini = i;
      }
    }
    tosort[ni] = bufs[mini-1][mi[mini-1]];
    mi[mini-1]++;
    ni++;
  }
}
```

Single slow send halts
all computation for workers

Single slow recv halts all
communication for workers

```
void send_sort(int rank, int size) {
  // ...
  int split = NINTS / (size - 1);
  for (int i = 1; i < size; i++) {
    int start = (split * (i-1));
    MPI_Send(tosort + start, split, MPI_INT, i,
     TAG_DATA, MPI_COMM_WORLD);
  }
  // ...
  for (int i = 1; i < size; i++) {
    MPI_Recv(bufs[i-1], split, MPI_INT, i, TAG_DATA,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }

  int ni = 0;
  while (ni < NINTS) {
    // Find minimum val and index
    int minv = INT_MAX;
    int mini = -1;
    for (int i = 1; i < size; i++) {
      if (mi[i-1] < split && bufs[i-1][mi[i-1]] < minv) {
        minv = bufs[i-1][mi[i-1]];
        mini = i;
      }
    }
    tosort[ni] = bufs[mini-1][mi[mini-1]];
    mi[mini-1]++;
    ni++;
  }
}
```

Single slow send halts
all computation for workers

Single slow recv halts all
communication for workers

Non-blocking communication

int MPI_ISSend(**void** *b, **int** count, **MPI_Datatype** type, **int** src, **int** tag,
MPI_Comm com, **MPI_Request** *req)

Non-blocking communication

int MPI_ISSend(**void** *b, **int** count, **MPI_Datatype** type, **int** src, **int** tag,
MPI_Comm com, **MPI_Request** *req)

non-blocking, synchronous send

Create a request handle to later check (output param)

Non-blocking communication

int MPI_ISSend(**void** *b, **int** count, **MPI_Datatype** type, **int** src, **int** tag,
MPI_Comm com, **MPI_Request** *req)

non-blocking, synchronous
send

Create a request handle to
later check (output param)

int MPI_Wait(**MPI_Request** *req, MPI_Status *stat)

Non-blocking communication

int MPI_ISSend(**void** *b, **int** count, **MPI_Datatype** type, **int** src, **int** tag,
MPI_Comm com, **MPI_Request** *req)

non-blocking, synchronous
send

Create a request handle to
later check (output param)

Way to kick off multiple communication

int MPI_Wait(**MPI_Request** *req, MPI_Status *stat)

Non-blocking communication

int MPI_IRecv(**void** *b, **int** count, **MPI_Datatype** type, **int** src, **int** tag, **MPI_Comm** com, **MPI_Request** *req)

non-blocking, synchronous recv

Create a request handle to later check (output param)

int MPI_Wait(**MPI_Request** *req, MPI_Status *stat)

Non-blocking communication

int MPI_Wait(**MPI_Request** *req, **MPI_Status** *stat)

int MPI_Waitall(**int** count, **MPI_Request** reqs[], **MPI_Status** stats[])

int MPI_Waitany(**int** count, **MPI_Request** reqs[], **int** *idx, **MPI_Status** stats[])

int MPI_Waitsome(**int** count, **MPI_Request** reqs[],
int *ocount, **int** idxs[], **MPI_Status** stats[])

Non-blocking communication

int MPI_Test(**MPI_Request** *req, **int** *flag, **MPI_Status** *stat)

Returns true if req has finished,
non-blocking test

Analogous functions to Wait:
MPI_Testany
MPI_Testall
MPI_Testsome

MPI Parallel Sort

- Example (sort.c, wait-sort.c)

MPI Collecive Communication

- Optimized communication routines across a MPI Communicator (MPI_COMM_WORLD default)
- All processes must call the communication function (“logical sync point”)
- Some useful ones
  - MPI_Barrier
  - MPI_Bcast
  - MPI_Scatter / MPI_Gather
  - MPI_Reduce

MPI_Barrier

int MPI_Barrier(**MPI_Comm** comm)

Synchronize all processes
in a communicator

Communicator to sync on
(MPI_COMM_WORLD)

Useful for debugging purposes,
not production code

MPI_Bcast

int MPI_Bcast(**void** *b, **int** count, **MPI_Datatype** type, **int** root, **MPI_Comm** comm)

MPI_Bcast

int MPI_Bcast(**void** *b, **int** count, **MPI_Datatype** type, **int** root, **MPI_Comm** comm)

Rank of the process that
performs the broadcast

Broadcast to all processes
in communicator

MPI_Bcast

int MPI_Bcast(**void** *b, **int** count, **MPI_Datatype** type, **int** root, **MPI_Comm** comm)

Rank of the process that
performs the broadcast

If root, broadcast data in b,
If not root, recv data in b

Broadcast to all processes
in communicator

```
int main(int argc, char **argv) {
  // MPI_Init, rank, size etc

  char buf[5];
  strcpy(buf,"ping");

  if (rank == 0) {
    MPI_Bcast(buf,5,MPI_CHAR,0,MPI_COMM_WORLD);
    for (int i = 1; i < size; i++) {
      MPI_Recv(buf,5,MPI_CHAR,i,0,
        MPI_COMM_WORLD,MPI_STATUS_IGNORE);
      printf("Got %s\n", buf);
    }
  } else {
    MPI_Bcast(buf,5,MPI_CHAR,0,MPI_COMM_WORLD);
    buf[0] = 'a' + rank;
    MPI_Send(buf,5,MPI_CHAR,0,0,MPI_COMM_WORLD);
  }

  // …

  return 0;
}
```

ping/pong with BCast

Both sender and receivers call MPI_Bcast

Sending and Receiving buffer must be same size

MPI_Scatter

```
int MPI_Scatter(
  void *sbuf,
  int scount,
  MPI_Datatypes stype,
  void *rbuf,
  int rcount,
  MPI_Datatypes rtype,
  int root,
  MPI_Comm comm)
```

MPI_Scatter

int MPI_Scatter(
  void *sbuf,
  int scount,
  MPI_Datatypes stype,
  void *rbuf,
  int rcount,
  MPI_Datatypes rtype,
  int root,
  MPI_Comm comm)

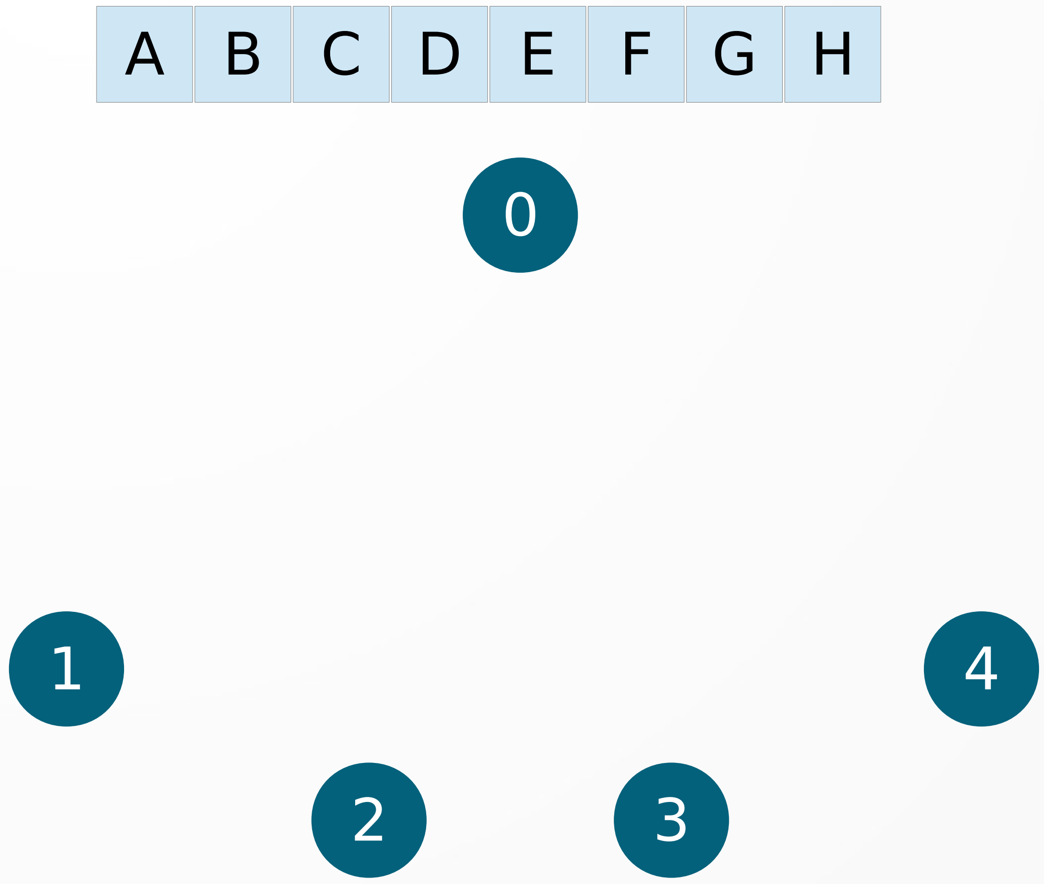
Sending buffer, count, and data type

Count is number of items to send to *each* process

Process performing scatter

MPI_Scatter

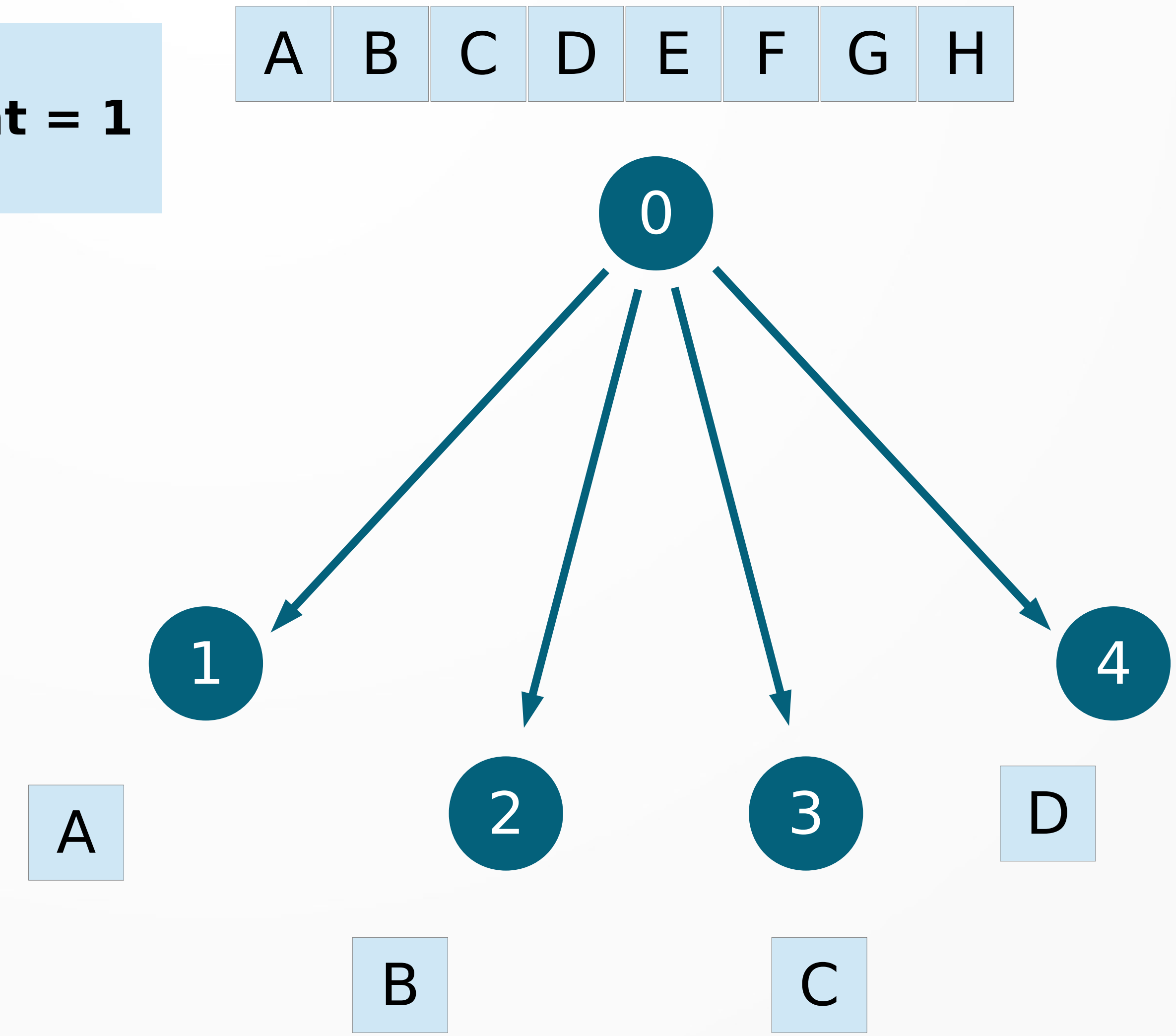
```
int MPI_Scatter(
  void *sbuf,
  int scount,
  MPI_Datatypes stype,
  void *rbuf,
  int rcount,
  MPI_Datatypes rtype,
  int root,
  MPI_Comm comm)
```



MPI_Scatter

int MPI_Scatter(
  void *sbuf,
  int scount,
  MPI_Datatypes stype,
  void *rbuf,
  int rcount,
  MPI_Datatypes **r**type,
  int root,
  MPI_Comm comm)

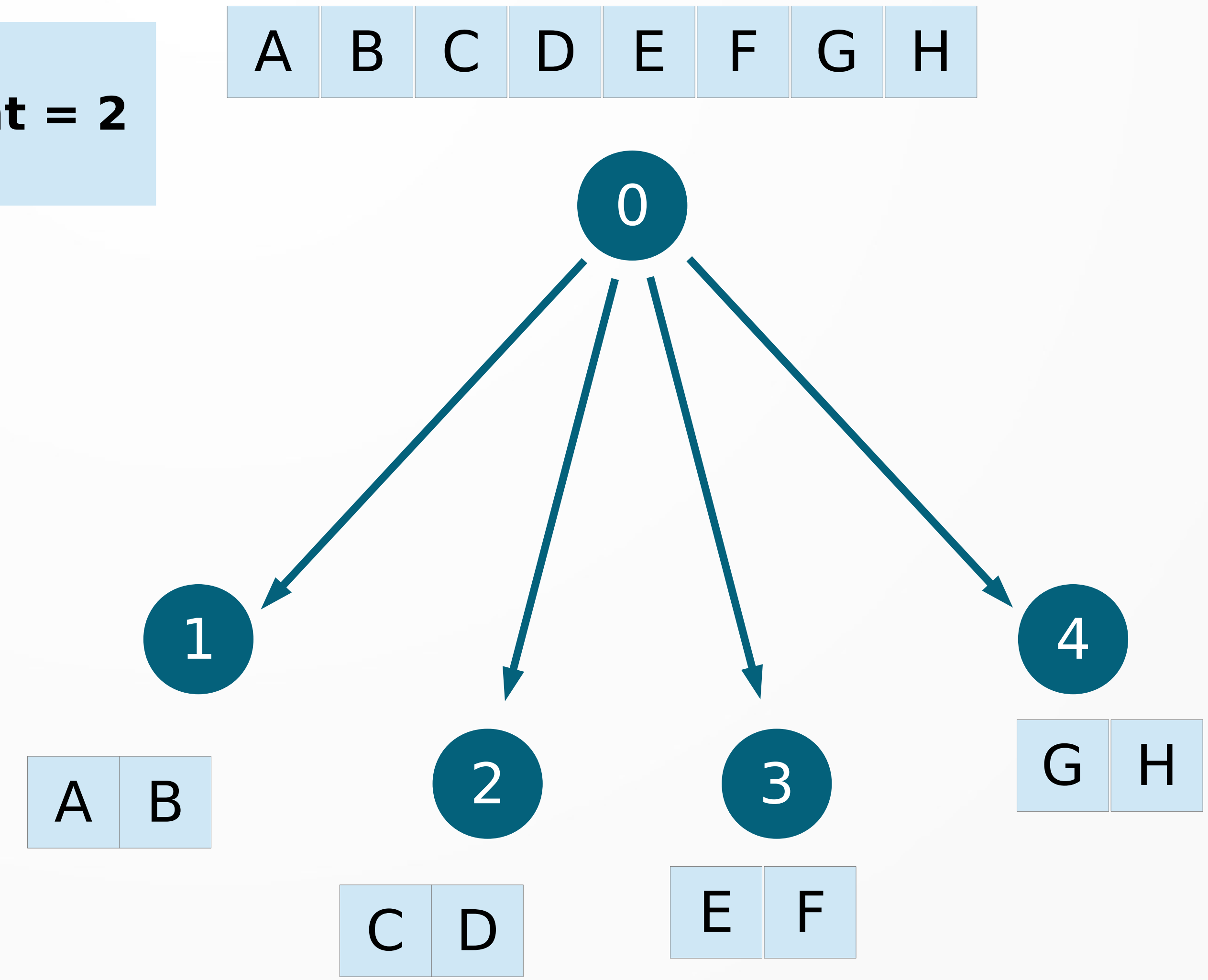
scount = 1



MPI_Scatter

```
int MPI_Scatter(
  void *sbuf,
  int scount,
  MPI_Datatypes stype,
  void *rbuf,
  int rcount,
  MPI_Datatypes rtype,
  int root,
  MPI_Comm comm)
```

scount = 2



MPI_Gather

```
int MPI_Gather(
  void *sbuf,
  int scount,
  MPI_Datatypes stype,
  void *rbuf,
  int rcount,
  MPI_Datatypes rtype,
  int root,
  MPI_Comm comm)
```

MPI_Gather

```
int MPI_Gather(
  void *sbuf,
  int scount,
  MPI_Datatypes stype,
  void *rbuf,
  int rcount,
  MPI_Datatypes rtype,
  int root,
  MPI_Comm comm)
```

Recv parameters associated with the root

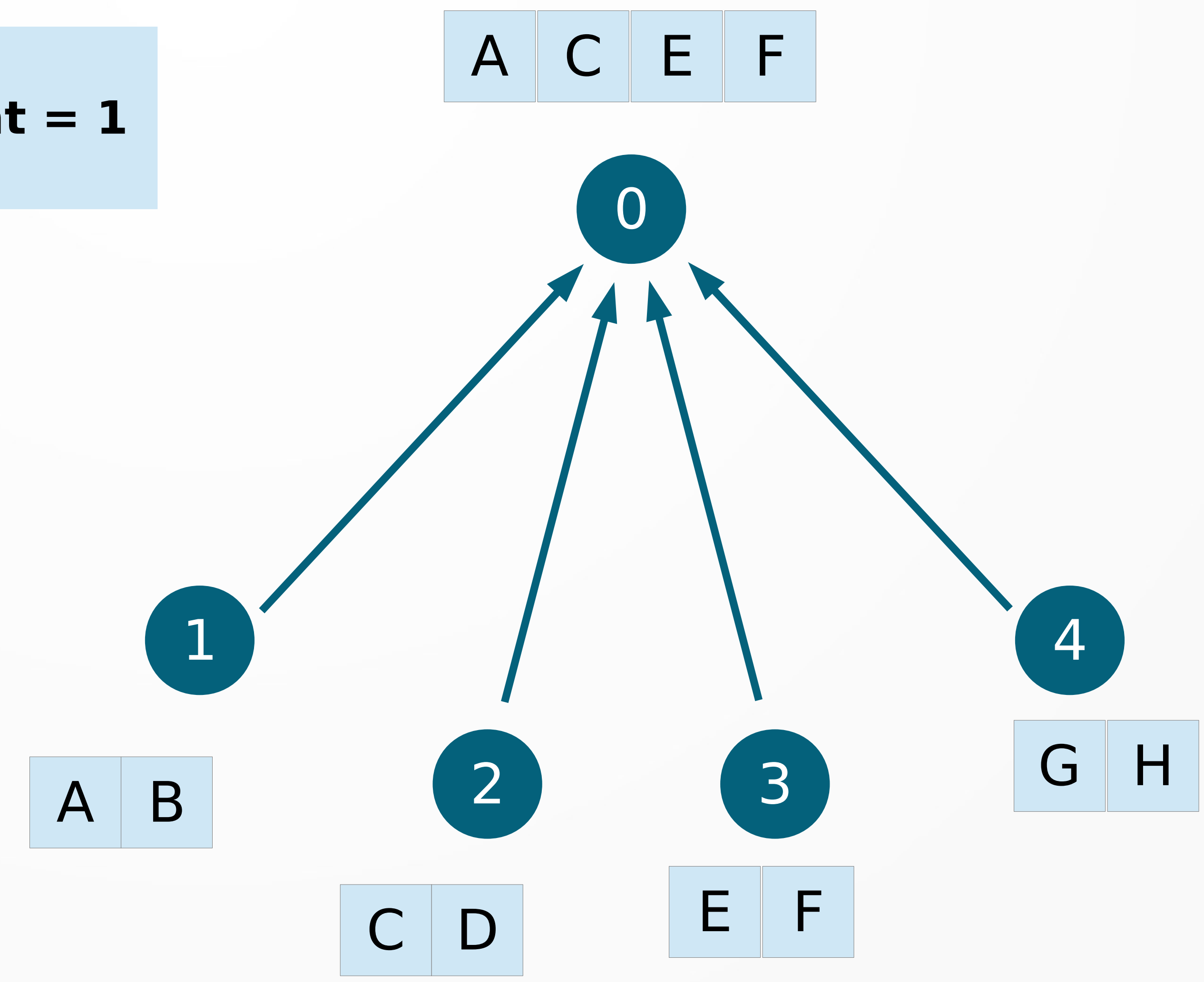
Count is number of items to recv from *each* process

Process to receive the gather

MPI_Gather

```
int MPI_Gather(
  void *sbuf,
  int scount,
  MPI_Datatypes stype,
  void *rbuf,
  int rcount,
  MPI_Datatypes rtype,
  int root,
  MPI_Comm comm)
```

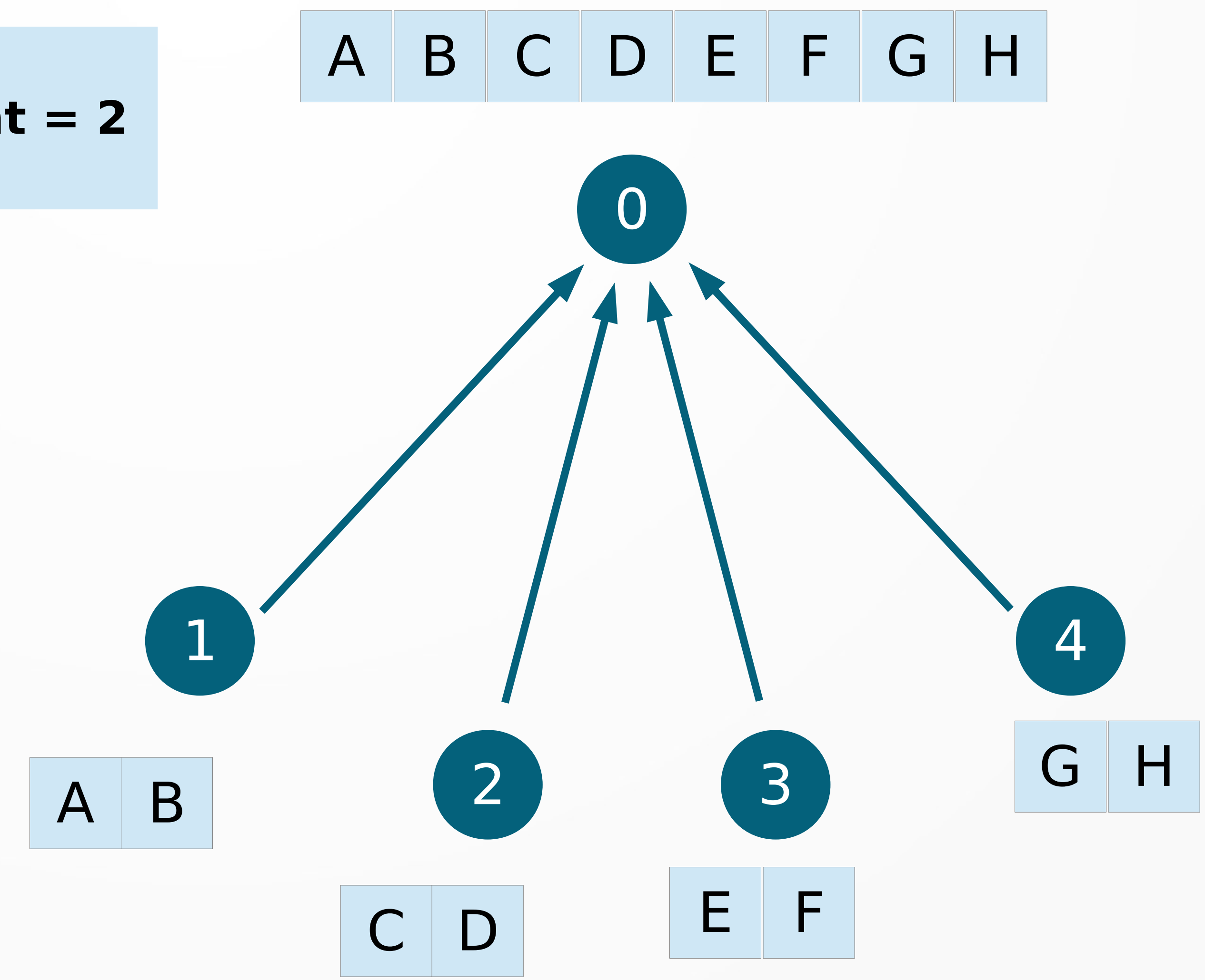
rcount = 1



MPI_Gather

```
int MPI_Gather(
  void *sbuf,
  int scount,
  MPI_Datatypes stype,
  void *rbuf,
  int rcount,
  MPI_Datatypes rtype,
  int root,
  MPI_Comm comm)
```

rcount = 2



MPI_Reduce

```
int MPI_Reduce(
  void *sbuf,
  void *rbuf,
  int count,
  MPI_Datatypes type,
  MPI_Op op,
  int root,
  MPI_Comm comm)
```

MPI_Reduce

```
int MPI_Reduce(
  void *sbuf,
  void *rbuf,
  int count,
  MPI_Datatypes type,
  MPI_Op op,
  int root,
  MPI_Comm comm)
```

Recv parameters associated with the root

Count is number of items to recv from *each* process

MPI_Reduce

```
int MPI_Reduce(
  void *sbuf,
  void *rbuf,
  int count,
  MPI_Datatypes type,
  MPI_Op op,
  int root,
  MPI_Comm comm)
```

Recv parameters associated with the root

Count is number of items to recv from *each* process

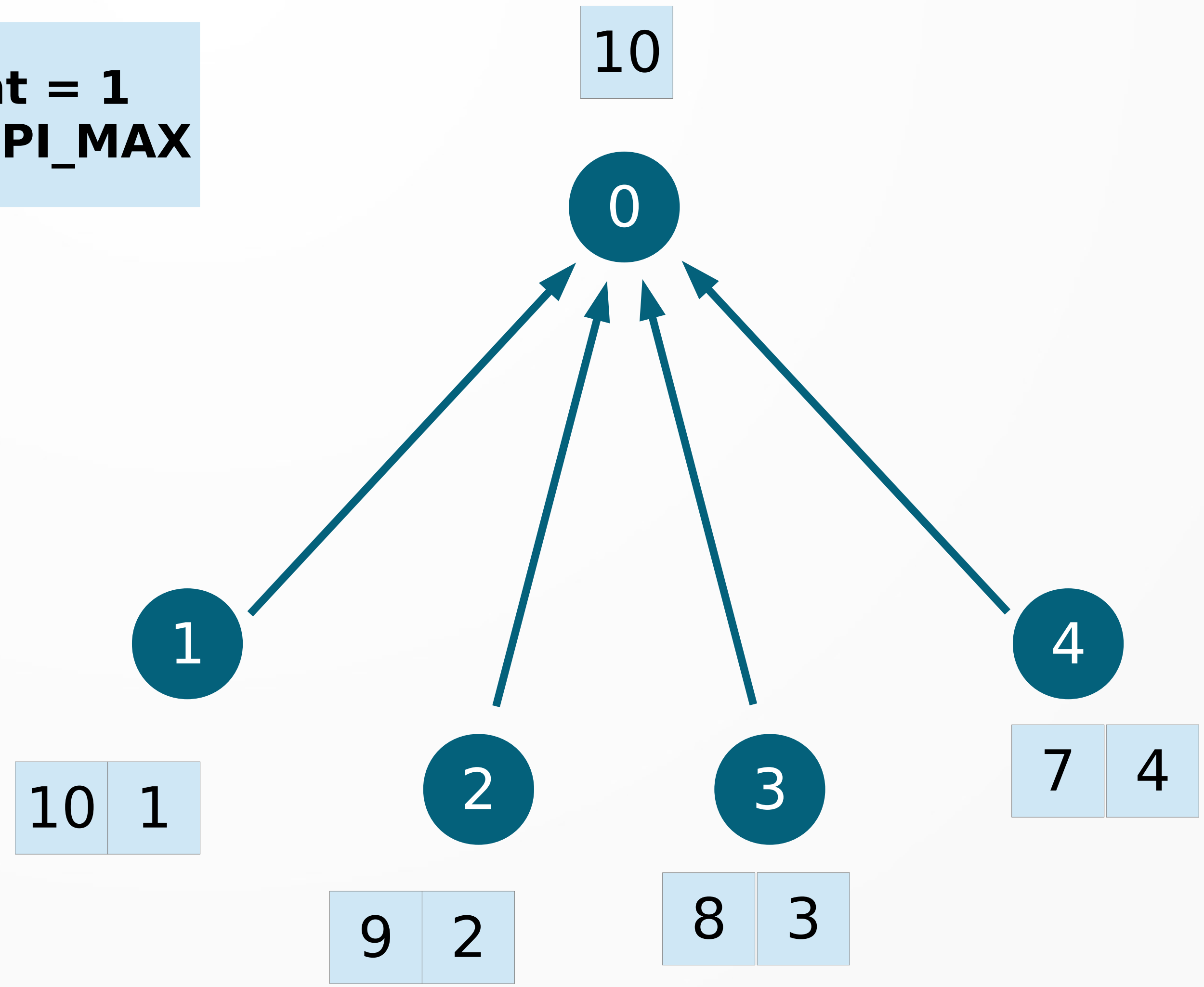
Operation applied to each received item before storing in rbuf

MPI_MAX
MPI_SUM
MPI_LOR

MPI_Reduce

```
int MPI_Reduce(
  void *sbuf,
  void *rbuf,
  int count,
  MPI_Datatypes type,
  MPI_Op op,
  int root,
  MPI_Comm comm)
```

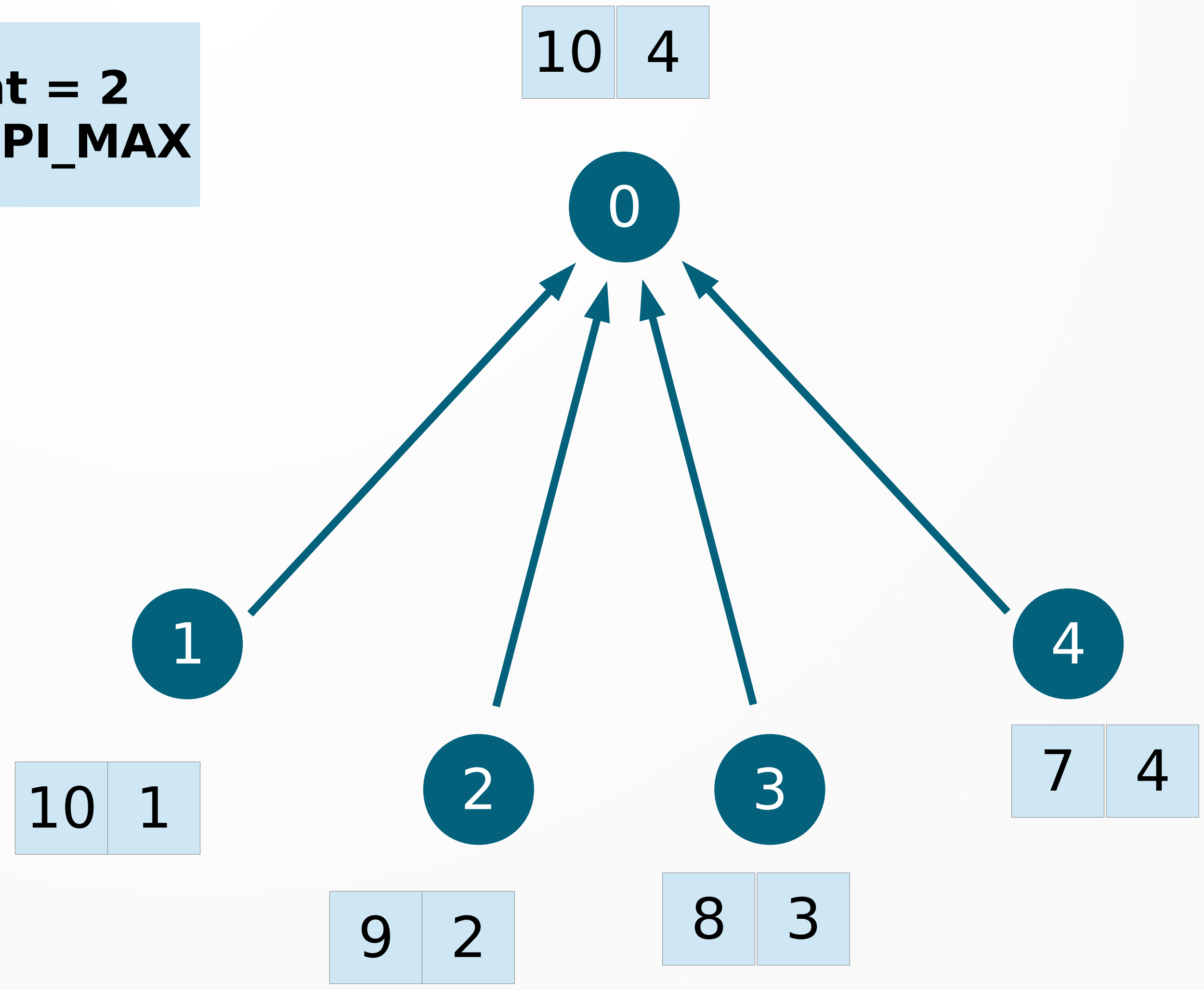
count = 1
op = MPI_MAX



MPI_Reduce

```
int MPI_Reduce(
  void *sbuf,
  void *rbuf,
  int count,
  MPI_Datatypes type,
  MPI_Op op,
  int root,
  MPI_Comm comm)
```

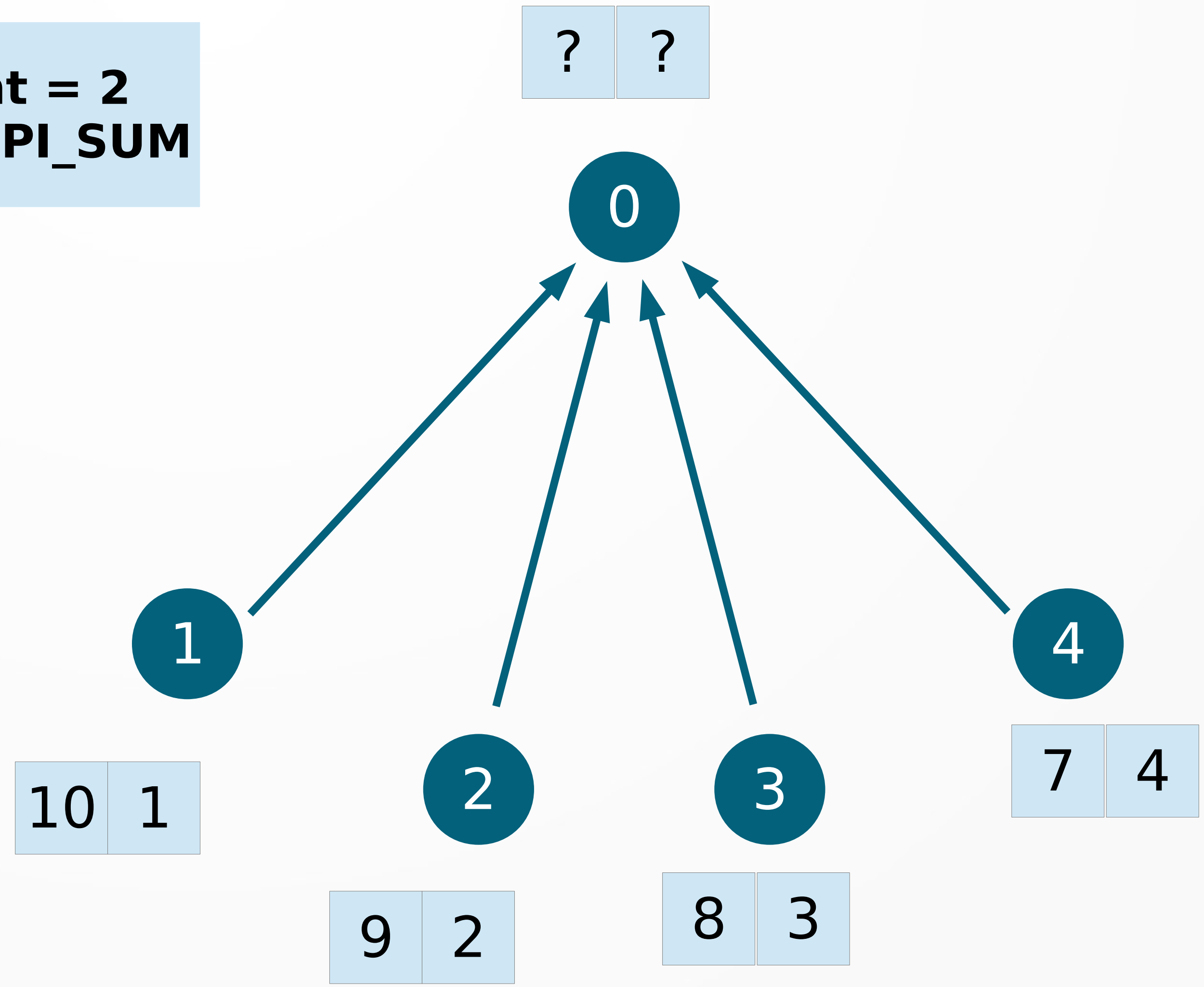
count = 2
op = MPI_MAX



MPI_Reduce

```
int MPI_Reduce(
  void *sbuf,
  void *rbuf,
  int count,
  MPI_Datatypes type,
  MPI_Op op,
  int root,
  MPI_Comm comm)
```

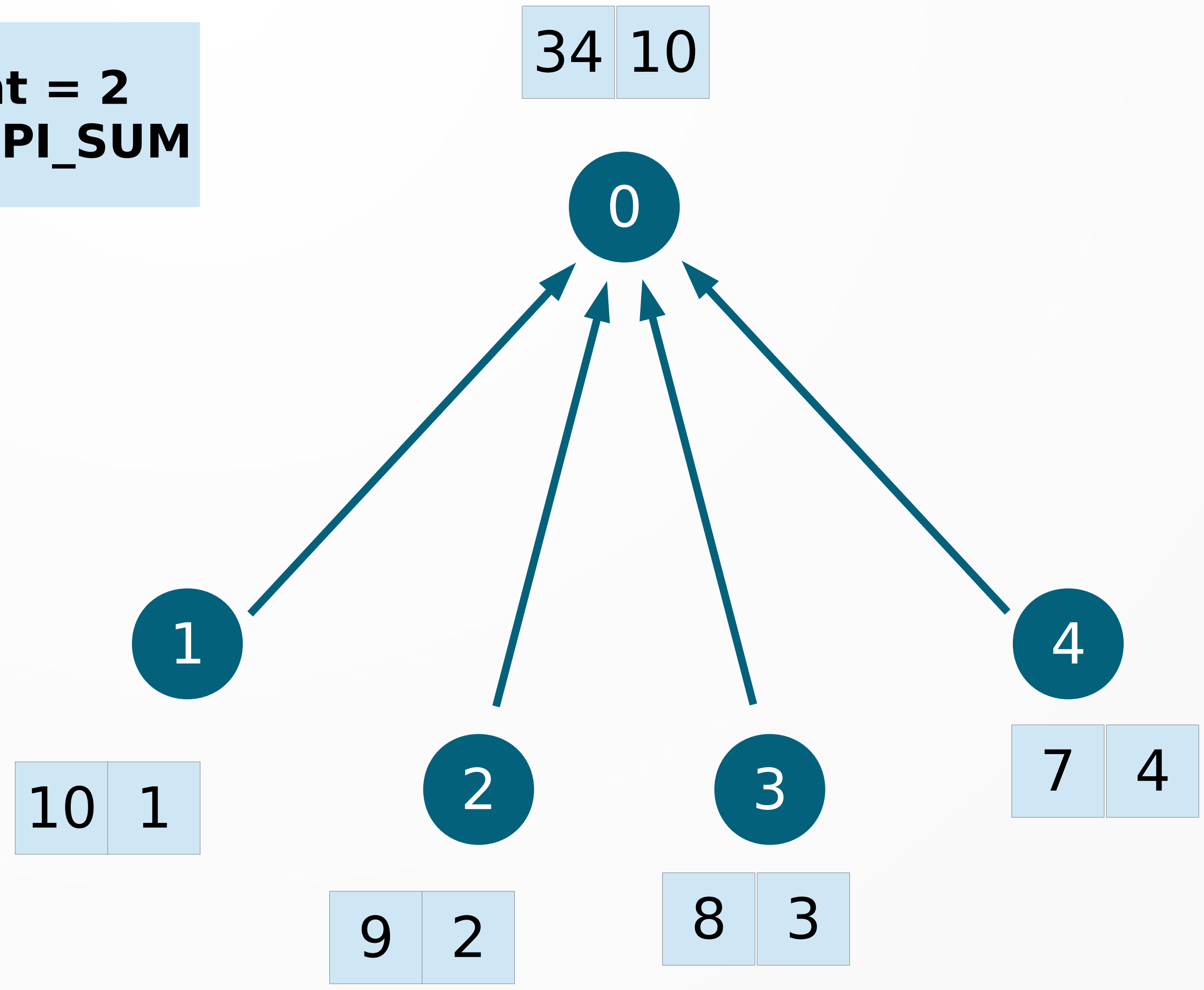
count = 2
op = MPI_SUM



MPI_Reduce

int MPI_Reduce(
  void *sbuf,
  void ***rbuf,
  int count,
  MPI_Datatypes type,
  MPI_Op op,
  int root,
  MPI_Comm comm)

count = 2
op = MPI_SUM



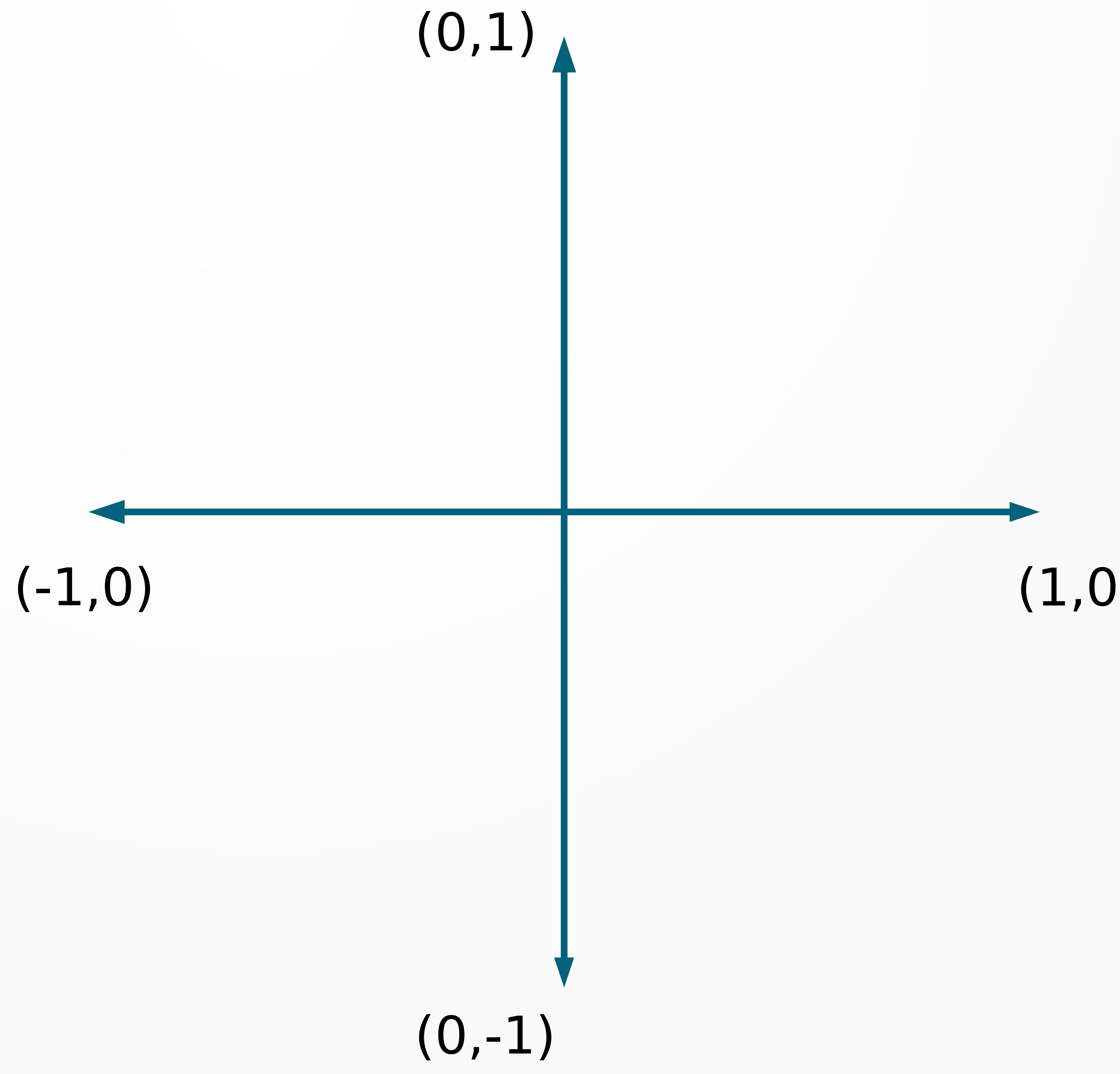
MPI Collecive Communication

- Many, many more routies
  - MPI_Allgather (each process gets the gather)
  - MPI_Allreduce (each process gets the reduce)
  - MPI_Scatterv (control which items go to which processes)
  - ...

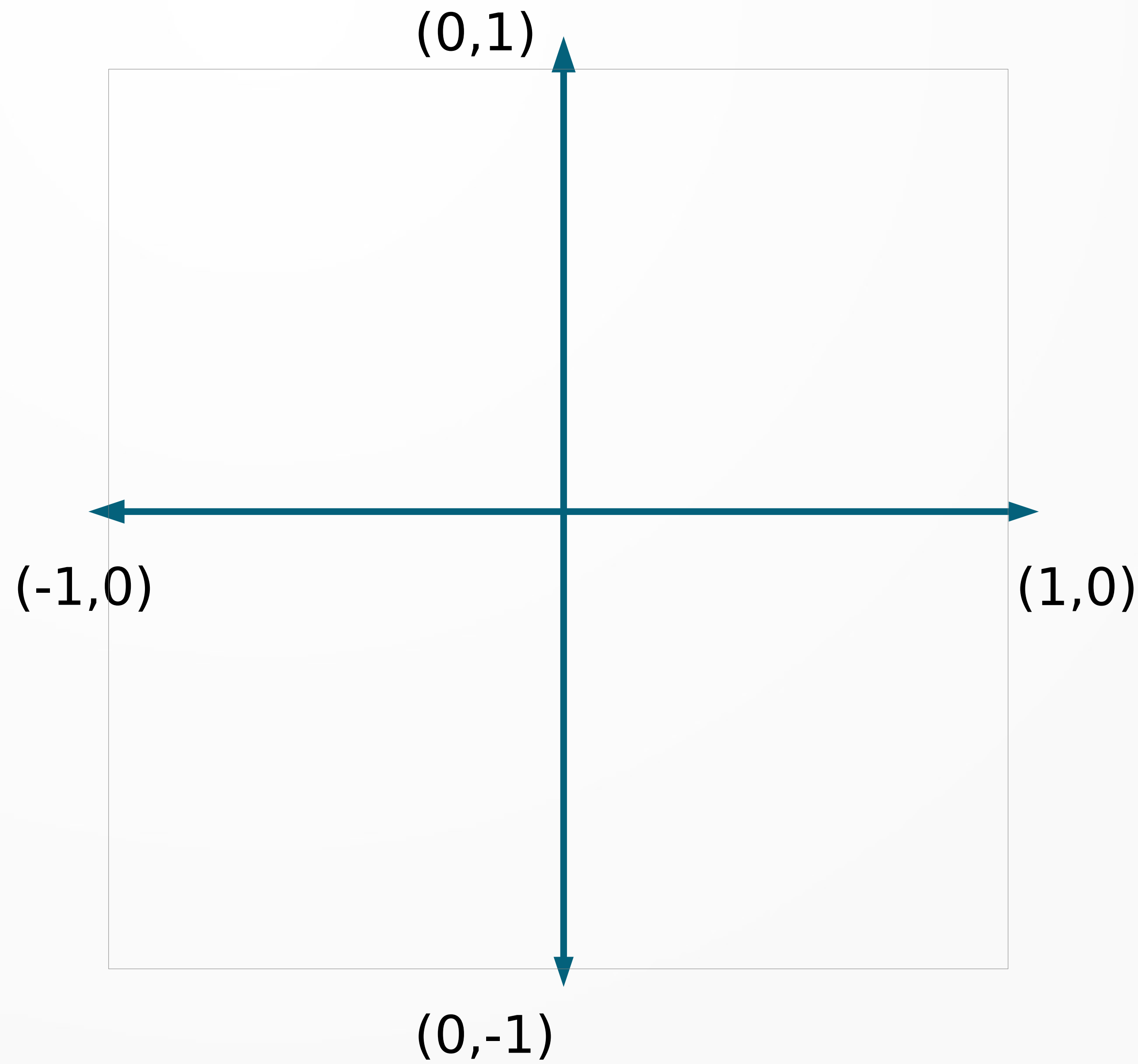
Monte Carlo

- Monte Carlo Simulation
  - Uses random sampling to perform trials in a simulation
  - Approximation that relies on the law of large number (if you draw enough samples, you eventually converge)
  - Topic of study in its own right, but a simple case is useful for us

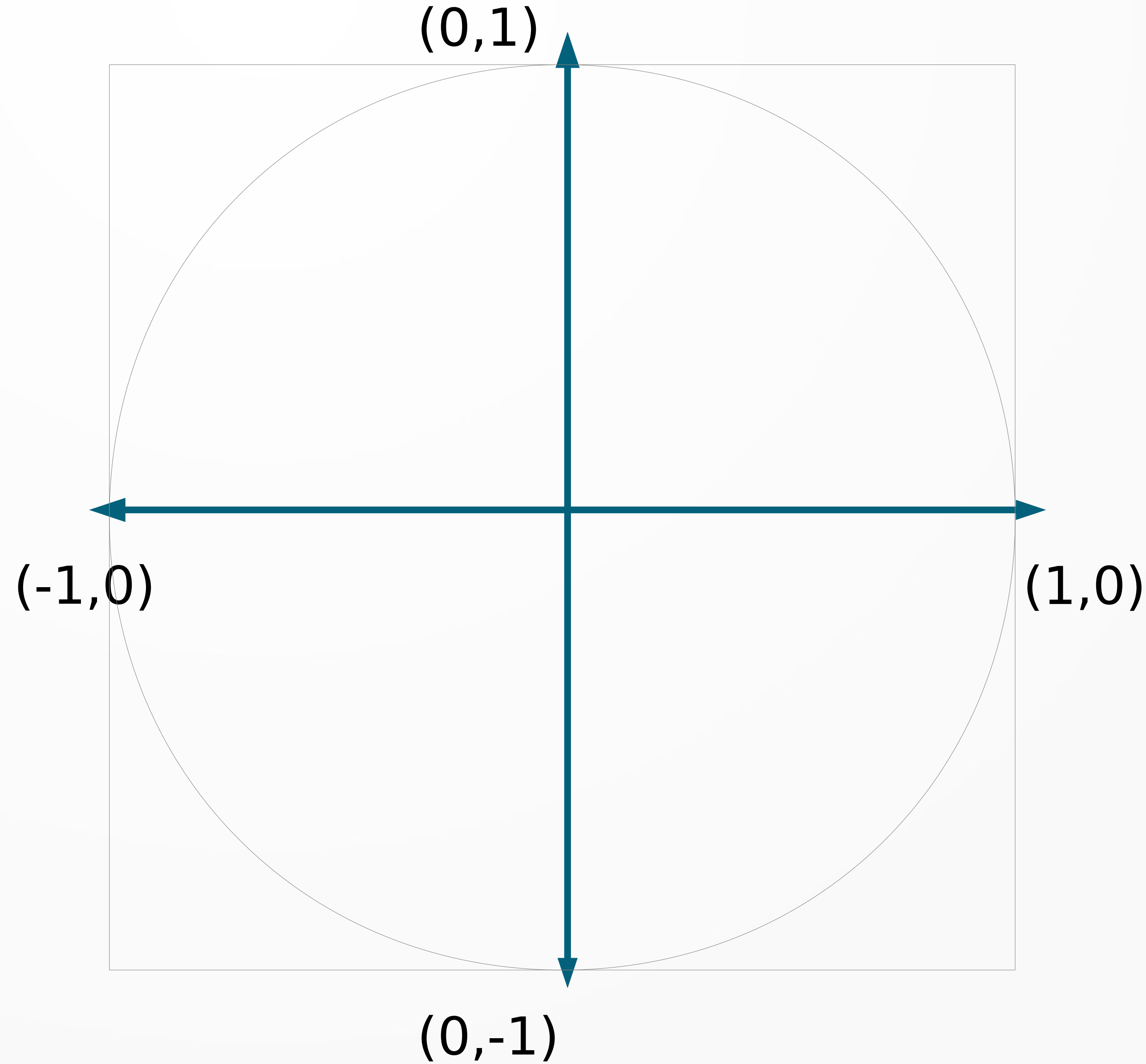
Monte Carlo PI



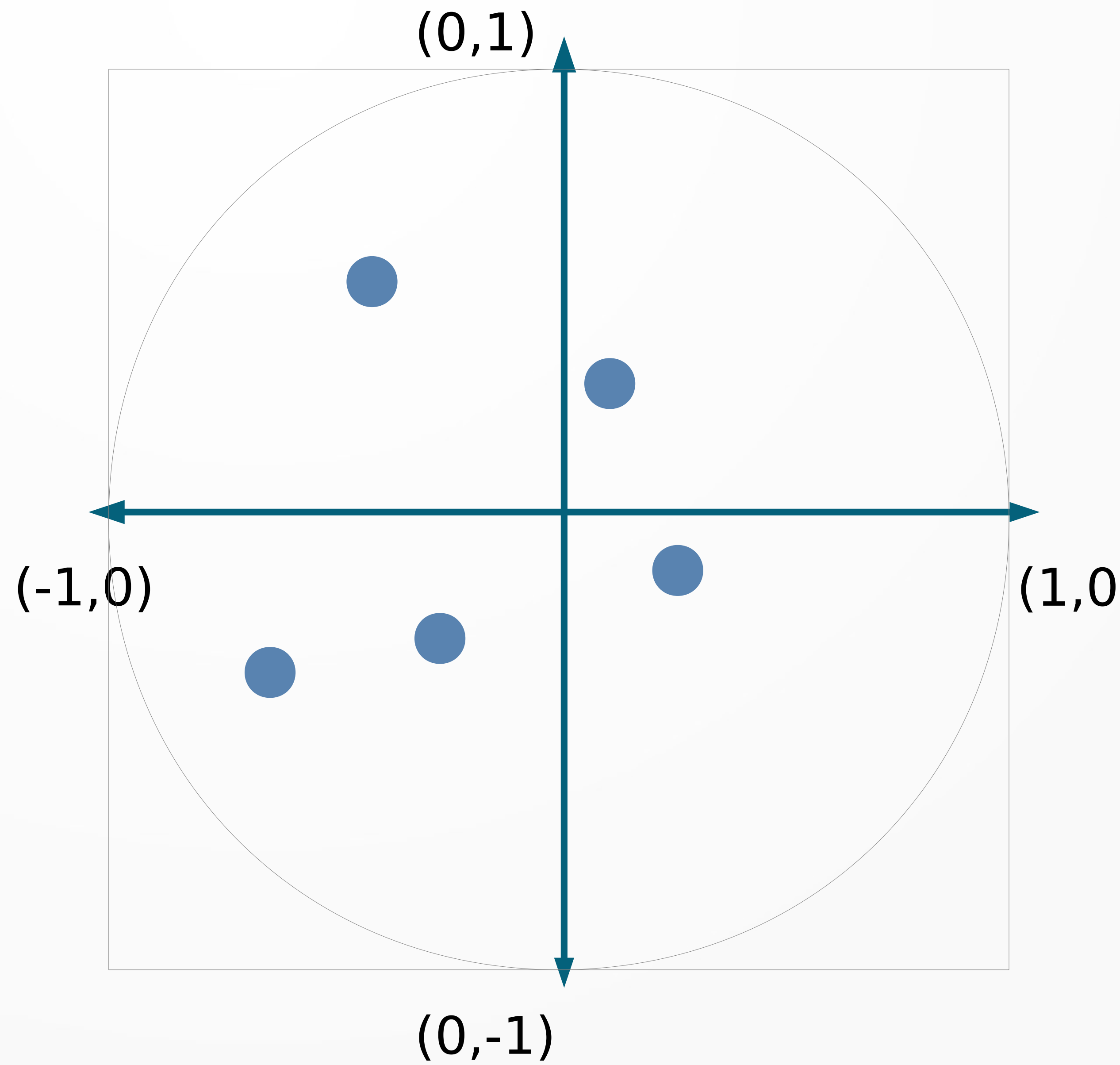
Monte Carlo PI



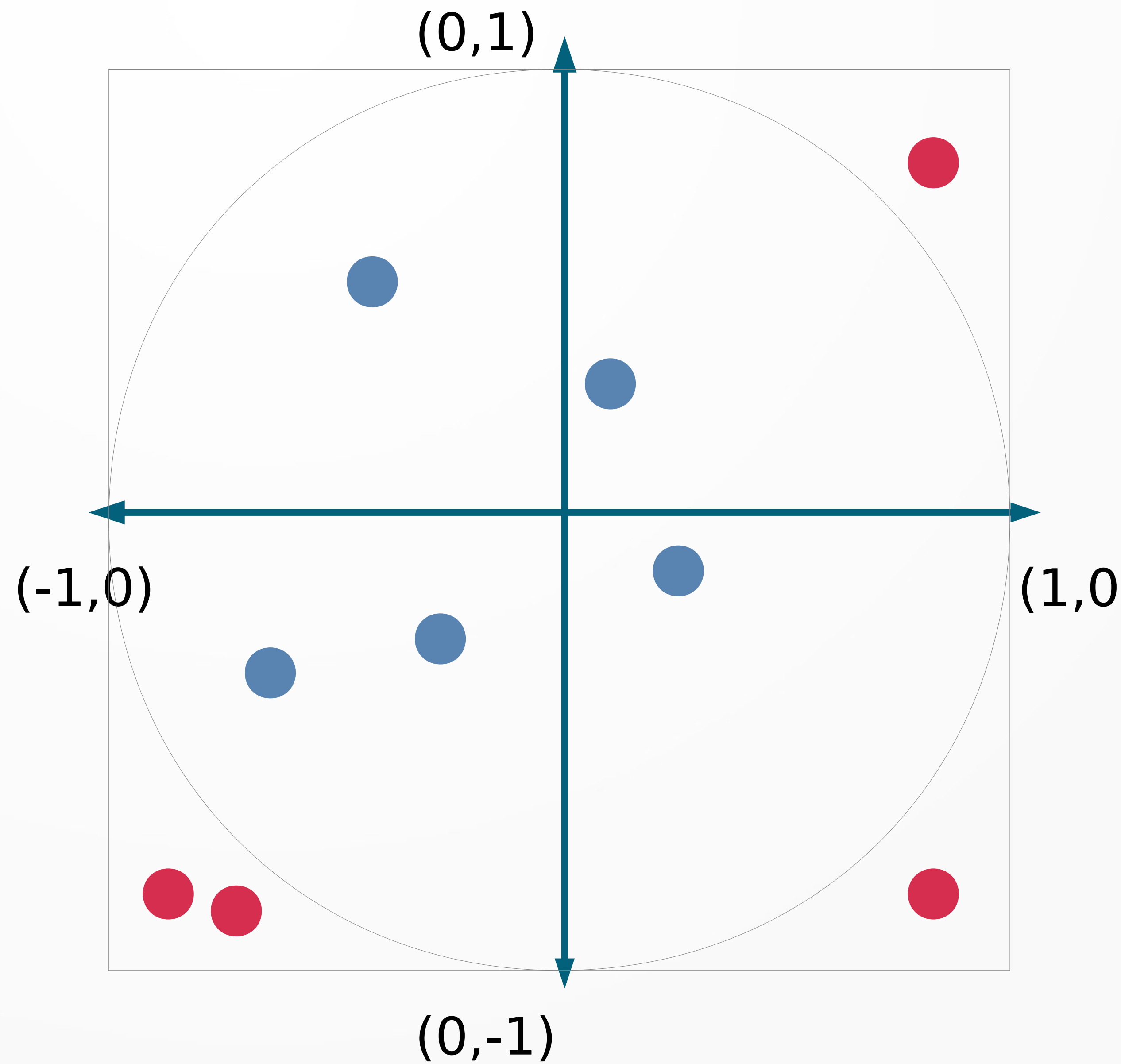
Monte Carlo PI



Monte Carlo PI

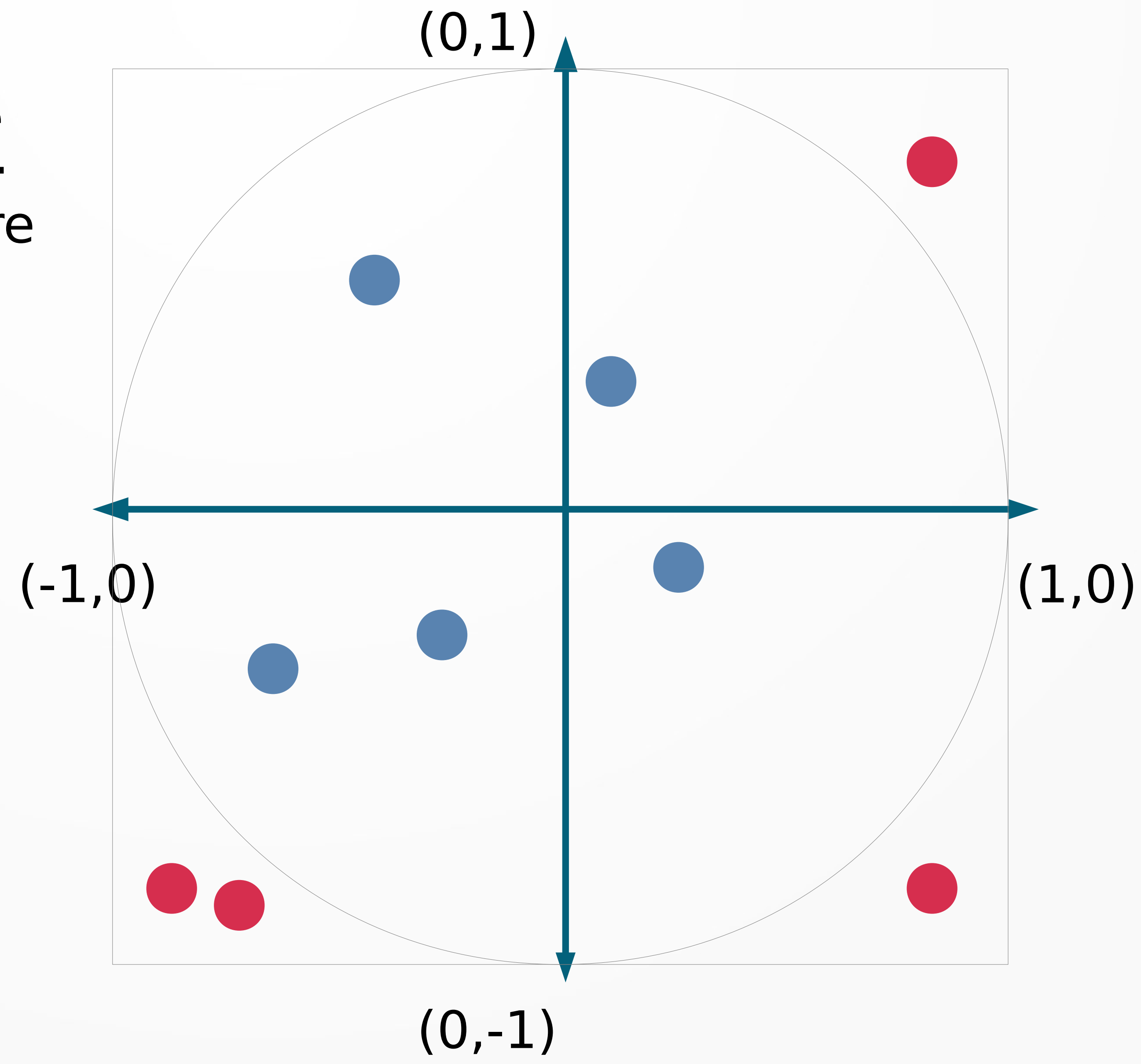


Monte Carlo PI



Monte Carlo PI

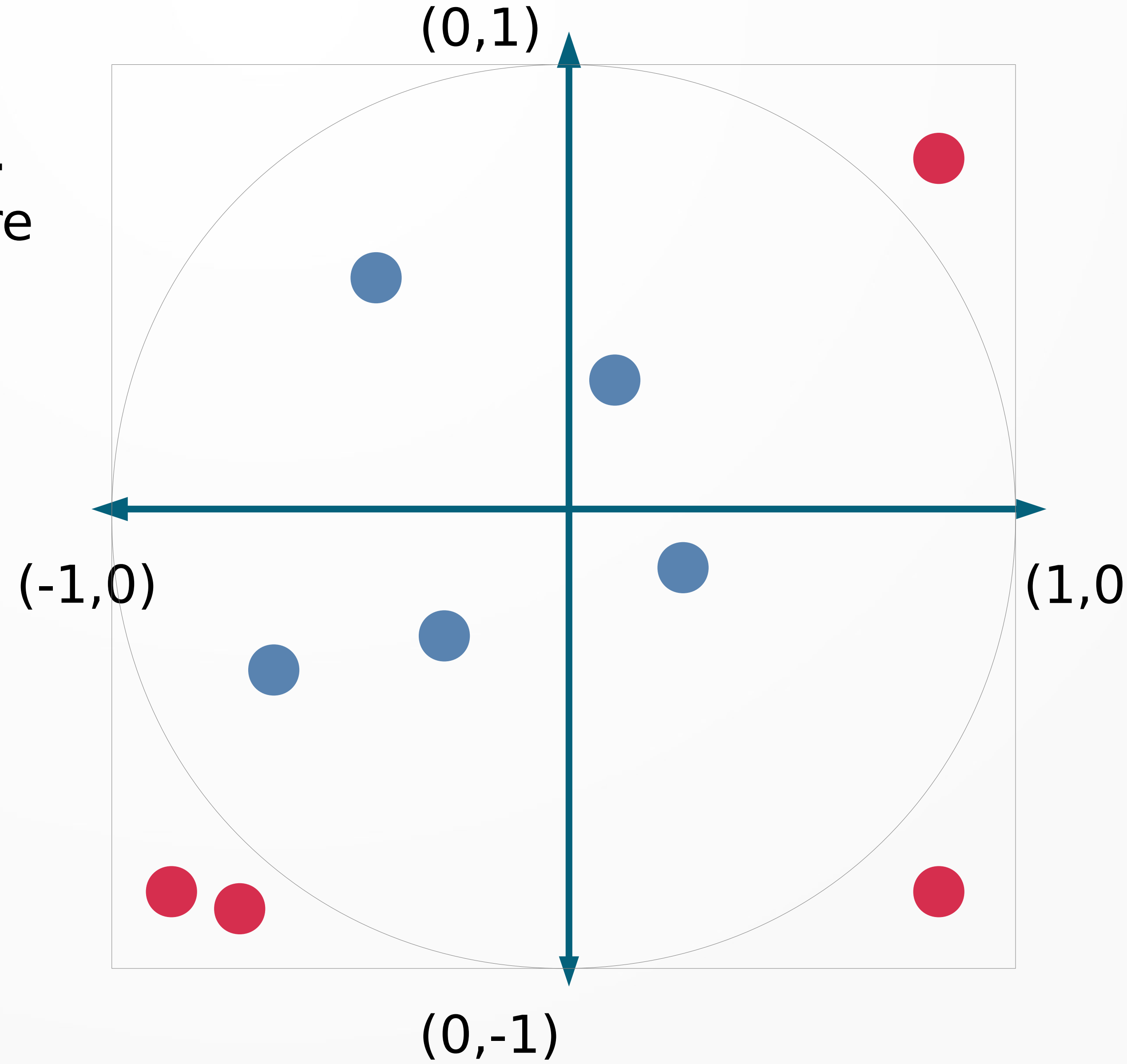
$$pi = area_{square} * \frac{\text{points in circle}}{\text{points in square}}$$



Monte Carlo PI

$$pi = area_{square} * \frac{\text{points in circle}}{\text{points in square}}$$

Need to draw a lot of samples to be accurate



```c
double sample() {
  double s = -1.0+2.0*((double)rand())/RAND_MAX;
  return s;
}

int main(int argc, char **argv) {
  if (argc < 2) {
    fprintf(stderr, "%s usage: [NUM-SAMPLES]\n", argv[0]);
    exit(1);
  }

  srand(time(NULL));

  long long nsamples = strtoll(argv[1],NULL,10);

  long long incircle = 0;

  for (long long i = 0; i < nsamples; i++) {
    double x = sample();
    double y = sample();
    if (x * x + y * y < 1.0) incircle++;
  }

  double pi = 4 * ((double) incircle / (double) nsamples);
  printf("%lld samples gives pi: %.16g\n", nsamples, pi);
  return 0;
}
```

C hack to sample from [-1,1]

Sample and check constraint

PI calculation

Parallel Monte Carlo

- Example (monte.c)

For Reference / Self Study

- MPI Collective Communication
  - MPI_Barrier
  - MPI_Bcast
  - MPI_Scatter
  - MPI_Gather
  - MPI_Reduce
  - MPI_All(gather,reduce)
  - Many more
- Compiling and linking
  - Compile: mpicc hello.c
  - Run: mpirun -n 4 ./a.out
- MPI Ops
  - MPI_MAX
  - MPI_MIN
  - MPI_SUM
  - Many more

Acknowledgments

- Introduction to the Message Passing Interface (Irish Centre for High-End Computing (ICHEC) www.ichec.ie
- A Comprehensive MPI Tutorial Resource (http://mpitutorial.com/)
- Parallel Monte Carlo Computation
- Module for Monte Parlo PI