CISC 372: Parallel Computing

MPI Collectives

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

The collective model of computation

- a collective operation is invoked by all processes in a communicator

The collective model of computation

- a collective operation is invoked by all processes in a communicator
- some processes contribute data, some receive data, some operations may be performed

The collective model of computation

- a collective operation is invoked by all processes in a communicator
- some processes contribute data, some receive data, some operations may be performed
- collective operations capture many commonly used parallel patterns
  - one process broadcasts data to all
  - add up values across processes and return the result to one process
  - gather data from all processes into one big array on one process

The collective model of computation

- a collective operation is invoked by all processes in a communicator
- some processes contribute data, some receive data, some operations may be performed
- collective operations capture many commonly used parallel patterns
  - one process broadcasts data to all
  - add up values across processes and return the result to one process
  - gather data from all processes into one big array on one process
- many parallel algorithms can be expressed most easily using collectives

The collective model of computation

- a collective operation is invoked by all processes in a communicator
- some processes contribute data, some receive data, some operations may be performed
- collective operations capture many commonly used parallel patterns
  - one process broadcasts data to all
  - add up values across processes and return the result to one process
  - gather data from all processes into one big array on one process
- many parallel algorithms can be expressed most easily using collectives
- collective programs are easier to understand than point-to-point
  - a higher-level, abstract, "big step" view of an algorithm
  - all procs do this, then all procs do that, then all procs do, ...
  - you can reason about them much like a sequential program

The collective model of computation

- a collective operation is invoked by all processes in a communicator
- some processes contribute data, some receive data, some operations may be performed
- collective operations capture many commonly used parallel patterns
  - one process broadcasts data to all
  - add up values across processes and return the result to one process
  - gather data from all processes into one big array on one process
- many parallel algorithms can be expressed most easily using collectives
- collective programs are easier to understand than point-to-point
  - a higher-level, abstract, "big step" view of an algorithm
  - all procs do this, then all procs do that, then all procs do, ...
  - you can reason about them much like a sequential program
- MPI implementations have very optimized implementations of collectives
  - these can be quite complicated, but the implementation details are hidden from the user

MPI_Reduce

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

`sendbuf`	address of send buffer (`void*`)
`recvbuf`	address of recv buffer (root only, `void*`)
`count`	number of elements in send buffer (`int`)
`datatype`	data type of elements in send buffer (`MPI_Datatype`)
`op`	reduce operation (`MPI_Op`)
`root`	rank of root process (`int`)
`comm`	communicator (`MPI_Comm`)

Rank 0 `sendbuf`	x_{00}	x_{01}	x_{02}
Rank 1 `sendbuf`	x_{10}	x_{11}	x_{12}
Rank 2 `sendbuf`	x_{20}	x_{21}	x_{22}

Root `recvbuf`	$x_{00} + x_{10} + x_{20}$	$x_{01} + x_{11} + x_{21}$	$x_{02} + x_{12} + x_{22}$

- see `reduce.c`

Creating global synchronization points: `MPI_Barrier`

```
MPI_Barrier(comm)
```

`comm` communicator (`MPI_Comm`)

Creating global synchronization points: `MPI_Barrier`

`MPI_Barrier(comm)`

`comm` communicator (`MPI_Comm`)

- blocks calling process until all processes in `comm` call `MPI_Barrier`

Creating global synchronization points: `MPI_Barrier`

`MPI_Barrier(comm)`

`comm` communicator (`MPI_Comm`)

- blocks calling process until all processes in `comm` call `MPI_Barrier`
- if one process in `comm` calls `MPI_Barrier(comm)`, all should
  - else deadlock ensues

Creating global synchronization points: MPI_Barrier

```
MPI_Barrier(comm)
```

`comm` communicator (`MPI_Comm`)

- blocks calling process until all processes in `comm` call `MPI_Barrier`
- if one process in `comm` calls `MPI_Barrier(comm)`, all should
  - else deadlock ensues
- explicit barriers are rarely needed; some exceptions...
  - good practice: use barriers before calling `MPI_Wtime`
    - ensures all processes have reached that point
  - programs that use `MPI_ANY_SOURCE` (coming soon)
    - barriers may be necessary to control how sends are matched with received

Creating global synchronization points: MPI_Barrier

`MPI_Barrier(comm)`

`comm` communicator (`MPI_Comm`)

- blocks calling process until all processes in `comm` call `MPI_Barrier`
- if one process in `comm` calls `MPI_Barrier(comm)`, all should
  - else deadlock ensues
- explicit barriers are rarely needed; some exceptions...
  - good practice: use barriers before calling `MPI_Wtime`
    - ensures all processes have reached that point
  - programs that use `MPI_ANY_SOURCE` (coming soon)
    - barriers may be necessary to control how sends are matched with received
- using barriers to control order of printing from different procs is not reliable
  - sometimes it works, sometimes it doesn't

Broadcast: MPI_Bcast

```
MPI_Bcast(buffer, count, datatype, root, comm)
```

`buffer`	address of buffer (`void*`)
`count`	number of elements in buffer (`int`)
`datatype`	data type of elements in buffer (`MPI_Datatype`)
`root`	rank of root process (`int`)
`comm`	communicator (`MPI_Comm`)

- broadcasts a message from a single process (`root`) to all other processes in `comm`
- on root, `buffer` acts as a send buffer; on non-root procs, `buffer` acts as a receive buffer
- after return, buffer will contain the same data as that on root
- is a barrier (global synchronization point) induced?
  - all procs must enter before any proc can exit ?

Broadcast: MPI_Bcast

```
MPI_Bcast(buffer, count, datatype, root, comm)
```

`buffer`	address of buffer (`void*`)
`count`	number of elements in buffer (`int`)
`datatype`	data type of elements in buffer (`MPI_Datatype`)
`root`	rank of root process (`int`)
`comm`	communicator (`MPI_Comm`)

- broadcasts a message from a single process (`root`) to all other processes in `comm`
- on root, `buffer` acts as a send buffer; on non-root procs, `buffer` acts as a receive buffer
- after return, buffer will contain the same data as that on root
- is a barrier (global synchronization point) induced?
  - all procs must enter before any proc can exit ? No
- see `bcast.c`

MPI_Allreduce

```
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)
```

`sendbuf`	address of send buffer (`void*`)
`recvbuf`	address of recv buffer (`void*`)
`count`	number of elements in send buffer (`int`)
`datatype`	data type of elements in send buffer (`MPI_Datatype`)
`op`	reduce operation (`MPI_Op`)
`comm`	communicator (`MPI_Comm`)

- just like `MPI_Reduce`, but no root
- instead, result is returned to all processes in `comm`
- equivalent to `MPI_Reduce` followed by `MPI_Bcast`
- barrier?

MPI_Allreduce

```
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)
```

`sendbuf`	address of send buffer (`void*`)
`recvbuf`	address of recv buffer (`void*`)
`count`	number of elements in send buffer (`int`)
`datatype`	data type of elements in send buffer (`MPI_Datatype`)
`op`	reduce operation (`MPI_Op`)
`comm`	communicator (`MPI_Comm`)

- just like `MPI_Reduce`, but no root
- instead, result is returned to all processes in `comm`
- equivalent to `MPI_Reduce` followed by `MPI_Bcast`
- barrier? yes

MPI_Scatter

```
MPI_Scatter(sendbuf, sendcount, sendtype,
            recvbuf, recvcount, recvtype, root, comm)
```

`sendbuf`	address of send buffer (root only, `void*`)
`sendcount`	num. elements sent to each proc (root, `int`)
`sendtype`	data type of send buf. elements (root, `MPI_Datatype`)
`recvbuf`	address of receive buffer (`void*`)
`recvcount`	number of elements in recv buffer (`int`)
`recvtype`	type of data to receive (`MPI_Datatype`)
`root`	rank of sending process (`int`)
`comm`	communicator (`MPI_Comm`)

- similar to broadcast: the root sends, everyone else receives
- but: root sends a different block of `sendbuf` to each proc
  - rank 0 gets the first `sendcount` elements, rank 1 gets the next `sendcount` elements ...
- number of elements in `sendbuf` is nprocs*`sendcount`; see `scatter.c`

MPI_Scatterv

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

`sendbuf`	address of send buffer (root only, `void*`)
`sendcounts`	num. elements sent to each proc (root only, `int[nprocs]`)
`displs`	displacements for each proc (root only, `int[nprocs]`)
`sendtype`	data type of send buf. elements (root only, `MPI_Datatype`)
`recvbuf`	address of receive buffer (`void*`)
`recvcount`	number of elements in recv buffer (`int`)
`recvtype`	type of data to receive (`MPI_Datatype`)
`root`	rank of sending process (`int`)
`comm`	communicator (`MPI_Comm`)

- generalizes `MPI_Scatter`: the amount of data sent to each process can vary
- `sendcounts[i]` = number of elements to send to proc i
- `displs[i]` = offset of send buffer for proc i relative to `sendbuf`

MPI_Gather

```
MPI_Gather(sendbuf, sendcount, sendtype,
           recvbuf, recvcount, recvtype, root, comm)
```

`sendbuf`	address of send buffer (`void*`)
`sendcount`	num. elements to send (`int`)
`sendtype`	data type of send buf. elements (`MPI_Datatype`)
`recvbuf`	address of receive buffer (root only, `void*`)
`recvcount`	number of elements to recv from each proc (root, `int`)
`recvtype`	type of data to receive (root, `MPI_Datatype`)
`root`	rank of receiving process (`int`)
`comm`	communicator (`MPI_Comm`)

- inverse of `MPI_Scatter`: everyone sends to root, root receives from everyone
- root receives into a different block of `recvbuf` for each proc
  - rank 0's message goes into first `recvcount` elements
  - rank 1's message goes into next `recvcount` elements ...
- number of elements in `recvbuf` is nprocs*`recvcount`

MPI_Allgather

```
MPI_Allgather(sendbuf, sendcount, sendtype,
              recvbuf, recvcount, recvtype, comm)
```

`sendbuf`	address of send buffer (`void*`)
`sendcount`	num. elements to send (`int`)
`sendtype`	data type of send buf. elements (`MPI_Datatype`)
`recvbuf`	address of receive buffer (`void*`)
`recvcount`	number of elements to recv from each proc (`int`)
`recvtype`	type of data to receive (`MPI_Datatype`)
`comm`	communicator (`MPI_Comm`)

- like `MPI_Gather` done once for each proc

MPI_Gatherv

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

`sendbuf`	address of send buffer (`void*`)
`sendcount`	num. elements to send (`int`)
`sendtype`	data type of send buf. elements (`MPI_Datatype`)
`recvbuf`	address of receive buffer (root, `void*`)
`recvcounts`	number of elements to recv from each proc (root, `int[nprocs]`)
`displs`	displacements of receive buffers (root, `int[nprocs]`)
`recvtype`	type of data to receive (root, `MPI_Datatype`)
`root`	rank of receiving process (`int`)
`comm`	communicator (`MPI_Comm`)

- generalizes `MPI_Gather`
- the amount sent by each process can vary
- see `gatherv.c`

MPI_Alltoall

```
MPI_Alltoall(sendbuf, sendcount, sendtype,
             recvbuf, recvcount, recvtype, comm)
```

`sendbuf`	address of send buffer (`void*`)
`sendcount`	num. elements to send (`int`)
`sendtype`	data type of send buf. elements (`MPI_Datatype`)
`recvbuf`	address of receive buffer (`void*`)
`recvcount`	number of elements to receive on each proc (`int`)
`recvtype`	type of element to receive (`MPI_Datatype`)
`comm`	communicator (`MPI_Comm`)

- every process sends distinct buffers to all others
- amount of data sent to each process is the same
- symmetric (no root)

MPI_Alltoallv

```
MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype,
              recvbuf, recvcounts, rdispls, recvtype, comm)
```

`sendbuf`	address of send buffer (`void*`)
`sendcounts`	num. elements to send to others (`int[nprocs]`)
`sdispls`	displacements of send buffers (`int[nprocs]`)
`sendtype`	data type of send buf. elements (`MPI_Datatype`)
`recvbuf`	address of receive buffer (`void*`)
`recvcounts`	number of elements to receive from others (`int[nprocs]`)
`rdispls`	displacements of receive buffers (`int[nprocs]`)
`recvtype`	type of element to receive (`MPI_Datatype`)
`comm`	communicator (`MPI_Comm`)

- generalizes `MPI_Alltoall`
- see `emily.c`