CIS 631
Parallel Processing

Lecture 8: MPI Programming and Applications

Allen D. Malony
malony@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon
Owner Computes Rule

- The process that owns the data is the process that will compute new data values
- Ownership is determined by data distribution
  - Domain decomposition
- Ownership may be implicitly determined
  - Could depend on other aspects of the program
- Why is the owner computes rule useful in distributed memory parallel programming?
Buffering Issues

- Where does data go when you send it?
- One possibility is:

  ![Diagram showing data flow between two processes](image)

- This is not very efficient
  - Three copies in addition to the exchange of data
  - Copies are “bad”
Better Buffering

- Prefer:

- Requires either
  - MPI_SEND does not return until data delivered
  - Or allow a send to return before completing transfer
- In the latter case, we need to test for completion later
Blocking vs. Non-Blocking Send and Receive

- The semantics of blocking/non-blocking has nothing to do with when messages are sent or received
- The difference is when the buffer is free to be re-used
- Is this a good way to think about it? Why?

Blocking

- A blocking send routine will only “return” if it is safe to modify the application buffer (your send data)
- A blocking send is \textit{synchronous} when the receiver confirms a safe send (again, we are talking semantics)
- A blocking send is \textit{asynchronous} if a system buffer is used to hold the data for eventual delivery to the receiver
- A blocking receive only “returns” after the data has arrived and is ready for use by the program
Non-Blocking

- Non-blocking
  - Send and receive routines do not wait for any communication events to complete, either
    - message copying from user memory to system buffer space
    - actual arrival of message
  - Simply “request” the MPI library to perform the operation when it is able, but cannot predict when
  - It is unsafe to modify the application buffer until you know the non-blocking operation was actually performed
    - use “wait” routines to do this
  - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gain (How?)
Send-Receive

Send-receive operation combine in one function call the sending of a message to one process and receiving of a message from another:

\[
\text{MPI\_Sendrecv}(\text{sendbuf}, \text{sendcount}, \text{sendtype}, \text{dest}, \text{sendtag}, \text{recvbuf}, \text{recvcount}, \text{recvtype}, \text{source}, \text{recvtag}, \text{comm}, \text{status})
\]

Use same buffer for sending and receiving:

\[
\text{MPI\_Sendrecv\_replace}(\text{buf}, \text{count}, \text{datatype}, \text{dest}, \text{sendtag}, \text{source}, \text{recvtag}, \text{comm}, \text{status})
\]

Exchange data with neighbors

\[
\text{MPI\_Sendrecv}(\&a,n,\text{MPI\_DOUBLE},\text{right},\text{TAG},\&b,n, \text{MPI\_DOUBLE},\text{left},\text{TAG},\text{MPI\_COMM\_WORLD},\&\text{Status})
\]
Non-Blocking MPI Send and Receive Operations

- Saw how non-blocking operations help to avoid deadlock
- Important for overlapping communication / computation
- Non-blocking send and receive routines

```
MPI_Isend(buf, count, datatype, dest, tag, comm, request)
MPI_Irecv(buf, count, datatype, source, tag, comm, request)
```

- Request represents the particular Isend or Irecv
  - Used as an argument in test and wait routines
MPI Test and Wait

- Routines to test status of non-blocking send and receive
  - MPI_Test(request, flag, status)
  - MPI_Wait(request, status)

- MPI_Test tests whether or not the non-blocking operation identified by request has finished
  - Returns true or false in flag
  - If true, request object deallocated

- MPI_Wait blocks until the non-blocking operation identified by request completes
  - request object deallocated on return
Waiting on Several Completions

- It is often desirable to wait on multiple requests
  - `MPI_Waitall(count, request_array, status_array)`
  - `MPI_Waitany(count, requests_array, index, status)`
  - `MPI_Waitsome(incount, requests_array, outcount, indices_array, status_array)`

- There are corresponding versions of test for each of these
- Why is this useful?
# Exchanging Data with Neighbors

<table>
<thead>
<tr>
<th></th>
<th>Blocking Recv</th>
<th>Non-blocking Recv</th>
</tr>
</thead>
</table>
| **Blocking Send** | `MPI_Send(A,...,left,...);`  
`MPI_Recv(B,...,right,...);` | `MPI_Send(A,...,left,...);`  
`MPI_Irecv(B,...,right,..., &req);`  
`MPI_Wait(&req,&status);` |
| **Non-blocking Send** | `MPI_Isend(A,...,left,..., &req);`  
`MPI_Recv(B,...,right...);`  
`MPI_Wait(&req,&status);` | `MPI_Isend(A,...,left,..., &req1);`  
`MPI_Irecv(B,...,right..., &req2);`  
`MPI_Wait(&req1,&status);`  
`MPI_Wait(&req2,&status);` |
**Probe and Cancel**

- MPI_Probe() and MPI_Iprobe() allow incoming messages to be checked for, without receiving them
  - MPI_Probe(source, tag, comm, status)
  - MPI_Iprobe(source, tag, comm, flag, status)

- MPI_Cancel() cancels pending communication
  - MPI_Cancel(request)
MPI Global Operations

- Often, it is useful to have one-to-many or many-to-one message communication.
- This is what MPI’s global operations do:
  - MPI_Barrier
  - MPI_Bcast
  - MPI_Gather
  - MPI_Scatter
  - MPI_Reduce
  - MPI_Allreduce
- MPI does not specify how any of these are implemented.
- Global operations should be as efficient as possible.
**Barrier**

- **MPI_Barrier(comm)**
  - Global barrier synchronization
  - All processes in communicator wait at barrier
  - Released when all have arrived
Broadcast

- `MPI_Bcast(inbuf, incnt, intype, root, comm)`
  - `inbuf`: address of input buffer on root
  - `inbuf`: address of output buffer elsewhere
  - `incnt`: number of elements
  - `intype`: type of elements
  - `root`: process id of root process
Before Broadcast

proc0
proc1
proc2
proc3

root

inbuf
After Broadcast

proc0  proc1  proc2  proc3  inbuf

root
**MPI Scatter**

- **MPI_Scatter**(inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)
  - inbuf: address of input buffer
  - incnt: number of input elements
  - intype: type of input elements
  - outbuf: address of output buffer
  - outcnt: number of output elements
  - outtype: type of output elements
  - root: process id of root process
Before Scatter

*proc0*
*proc1*
*proc2*
*proc3*

*root*

*inbuf*
*outbuf*
After Scatter

proc0

proc1

proc2

proc3

inbuf

outbuf

root
MPI Gather

\[ \text{MPI\_Gather}(\text{inbuf}, \text{incnt}, \text{intype}, \text{outbuf}, \text{outcnt}, \text{outtype}, \text{root}, \text{comm})} \]

- \text{inbuf}: address of input buffer
- \text{incnt}: number of input elements
- \text{intype}: type of input elements
- \text{outbuf}: address of output buffer
- \text{outcnt}: number of output elements
- \text{outtype}: type of output elements
- \text{root}: process id of root process
Before Gather

- proc0
- proc1
- proc2
- proc3

root

inbuf
outbuf
After Gather

proc0  proc1  proc2  proc3

inbuf  outbuf

root
Broadcast / Gather / Scatter

- These three primitives combine sends and receives
  - May be confusing
  - Collective operations
- Perhaps un-intended consequence
  - Requires global agreement on layout of array
Groups, Communicators, and Contexts

- A group is an ordered set of processes
  - Each process in a group has a unique integer rank
  - Rank values are from 0 to N-1, where N is the group size
  - A group is accessible to the programmer by a “handle”
- A communicator is a group of processes that are allowed to communicate between themselves
  - All MPI messages must specify a communicator
  - Communicators accessible to programmer by “handles”
- A context is a system-defined object that uniquely identifies a communicator
More Groups and Communicators

- Groups/communicators are dynamic
- Processes may be in more than one group/communicator
  - They will have a unique rank within group/communicator
- Typical usage:
  1. Extract handle using `MPI_Comm_group`
  2. Form new group using `MPI_Group_incl`
  3. Create new communicator using `MPI_Comm_create`
  4. Determine new rank in communicator using `MPI_Comm_rank`
  5. Conduct communications using any MPI routine
  6. Free up communicator and group
Logical View

Groups are derived from communicators

Communicators are created from groups

MPI_COMM_WORLD

comm1  comm2

communications
Predefined Communicator

- **MPI_COMM_WORLD**
  - Contains all processes available at start of the program

- **MPI_COMM_NULL**
  - An invalid communicator

- **MPI_COMM_SELF**
  - Contains only the local process

- **MPI_COMM_EMPTY**
  - There is no such thing as `MPI_COMM_EMPTY`
    - Why not?
MPI Parallelization Process

- Divide program in parallel parts
- Create and destroy processes to do above
- Partition and distribute the data
- Communicate data at the right time
- (Sometimes) perform index translation
- Still need to do synchronization?
  - Sometimes, but many times goes hand in hand with data communication
Parallel Libraries and MPI

- Many libraries developed using MPI
  - Powerful parallelization and programming model
  - Gain portability advantages
- Scientific linear algebra, numerical, … libraries
  - ScaLAPACK, SuperLU, PETSc, Trilinos, PMTL (Parallel Matrix Template Library), …
- Graph, mesh, data, … libraries
  - PBGL (Parallel Boost Graph Library), ParMetis, PPM (Parallel Particle Mesh), HDF5, …
- Data, communication, computational, … libraries
  - GA (Global Arrays), ADLB (Asynchronous Dynamic Load Balancing), ARMCI (Aggregate Remote Memory Copy Interface), AP (Active Pebbles), LibNBC (Non-Blocking Collectives), AM++ (Active Message), …
The GAS / PGAS Model

- A parallel program consists of a set of *threads* and at least one *address space*
- A program is said to have a *global view* if all threads share a single address space
  - GAS – *Global Address Space*
- A program is said to have a *local view* if the threads have distinct address spaces
  - PGAS – *Partitioned Global Address Space*
- How are global and local views supported in reality
  - Shared memory provides by default
  - Distributed memory requires something more
GAS / PGAS Motivation and Implementation

- Need to support GAS and PGAS model in distributed memory systems
  - Threads with partitioned shared space
  - Datum may reference data in other partitions
  - Global array fragments in multiple partitions

- Advantages
  - Shared memory programming … why?
  - Performance … why?

- Disadvantages
  - Shared memory programming … why?
  - Performance … why?

- Library approach: GA, …
- Language approach: UPC, CAF, X10, Chapel
Realizing Dynamic Parallelism in PGAS

- Dynamic PGAS library ...
  - C, Fortran, Java, ...
  - Co-habiting with MPI
- ... implementing ...
  - Remote references
  - Global data structures
  - Inter-place messaging
  - Global and/or collective operations
  - Intra-place concurrency
  - Atomic operations
- ... through languages ...
  - Asynchronous CAF
  - Asynchronous UPC
  - X10
  - Chapel
- ... leveraging runtimes
  - GASNet, ARMCI, LAPI
  - UPC runtime
  - Chapel runtime
- Libraries reduce cost of adoption, languages offer enhanced productivity
**PGAS vs. Others**

<table>
<thead>
<tr>
<th></th>
<th>UPC, X10, Chapel, CAF, Titanium</th>
<th>MPI</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory model</td>
<td>PGAS (Partitioned Global Address Space)</td>
<td>Distributed Memory</td>
<td>Shared Memory</td>
</tr>
<tr>
<td>Notation</td>
<td>Language</td>
<td>Library</td>
<td>Annotations</td>
</tr>
<tr>
<td>Global arrays?</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Global pointers/references?</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Locality Exploitation</td>
<td>Yes</td>
<td>Yes, necessarily</td>
<td>No</td>
</tr>
</tbody>
</table>

Dynamic parallelism
Next Class

- Shared memory parallel programming
- Threads