MPI Collective Communication

CDP
MPI Collective Communication

• Reuse code for patterns that appear in different types of applications
• Built by using point-to-point communication routines
• Collective communication routines in MPI:
  • Barrier synchronization
  • Broadcast from one member to all other members
  • Scatter data from one member to all other members
  • Gather data from all members to one member
  • All-to-all exchange of data
  • Global reduction (e.g., sum, min of "common" data element to one node)
  • Scan across all members of a group
• Synchronization, data movement, and global computation
• Helpful Site
Collective communication routines characteristics

• Coordinated communication within a group of processes
  • Identified by an MPI communicator
  • No message tags are needed
• Substitute for a more complex sequence of point-to-point calls
• All routines block until they are locally completed
  • Starting from MPI 1.7, non-blocking collective communication is supported
• In some cases, a root process originates or receives all data
• Amount of data sent must exactly match amount of data specified by receiver
Barrier synchronization routine

• A synchronization primitive
  • Blocking until all the nodes in the group have called it

• MPI_Barrier()
  • comm: a communicator
Data movement routines - Broadcast

- One processor sends data to all the processes in a group
- **Bcast**(buffer, root)
  - Must be called by each node in the group, specifying the same comm, root and count

**Bcast**(buffer, root=1)
Gather and Scatter

- MPI_Scatter(sendbuf, recvbuf, root)
- MPI_Gather(sendbuf, recvbuf, root)
MPI_Scatter

• Scatter sets of 100 ints from the root to each process in the group
• At all processes
  • MPI_Scatter(None, rbuf, root=0)
• At root = 0
  • MPI_Scatter(sbuf, rbuf, root=0)
Gatherv and Scatterv

- Allow each node to send/receive a different number of elements

- `MPI_Gatherv(sbuf, [rbuf,counts,dspls,type], root)`

- `MPI_Scatterv([sbuf,counts,dspls,type], rbuf, root)`
MPI_Gatherv

• At all processes:
  • MPI_Gatherv(sbuf, None, root);

• At root = 0:
  • rcount = [30, 70, 120]
  • displs = [0, 30, 100]
  • MPI_Gatherv(sbuf, [rbuf, rcount, displs, type], root);

Displacement differences must be larger than the corresponding rcounts entry.
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

a_size = 4
recvdata = None
senddata = (rank + 1) * np.arange(a_size, dtype=np.float64)
if rank == 0:
    recvdata = np.arange(size * a_size, dtype=np.float64)

comm.Gather(senddata, recvdata, root=0)
print("on task", rank, "after Gather: data = ", recvdata)
Code Example: Gatherv

```python
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

a_size = 4
recvdata = None
senddata = (rank + 1) * np.arange(a_size, dtype=np.float64)
counts = (2, 3, 4)
dspls = (0, 2, 5)
if rank == 0:
    recvdata = np.empty(9, dtype=np.float64)

sendbuf = [senddata, counts[rank]]
recvbuf = [recvdata, counts, dspls, MPI.DOUBLE]
comm.Gatherv(sendbuf, recvbuf, root=0)
print("on task", rank, "after Gatherv: data = ", recvdata)
```
Allgather

- MPI_Allgather(sbuf, rbuf)
- MPI_Allgatherv(sbuf, [rbuf, counts, dspls, type])
All to All

- MPI_Alltoall(sbuf, rbuf)

<table>
<thead>
<tr>
<th>Processes</th>
<th>sbuf</th>
<th>rbuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A(^1)</td>
<td>A(^1)</td>
</tr>
<tr>
<td>2</td>
<td>A(^2)</td>
<td>B(^1)</td>
</tr>
<tr>
<td>3</td>
<td>A(^3)</td>
<td>C(^1)</td>
</tr>
<tr>
<td>4</td>
<td>A(^4)</td>
<td>D(^1)</td>
</tr>
<tr>
<td>1</td>
<td>B(^1)</td>
<td>A(^2)</td>
</tr>
<tr>
<td>2</td>
<td>B(^2)</td>
<td>B(^2)</td>
</tr>
<tr>
<td>3</td>
<td>C(^2)</td>
<td>C(^2)</td>
</tr>
<tr>
<td>4</td>
<td>D(^3)</td>
<td>D(^3)</td>
</tr>
<tr>
<td>1</td>
<td>C(^3)</td>
<td>A(^3)</td>
</tr>
<tr>
<td>2</td>
<td>D(^4)</td>
<td>B(^3)</td>
</tr>
<tr>
<td>3</td>
<td>D(^1)</td>
<td>C(^3)</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>D(^4)</td>
</tr>
</tbody>
</table>
Global computation routines \texttt{Reduce}

- $D(i, j)$ is the $j^{th}$ data item in process $i$
- In the root process: $D_j = D(0, j) \odot D(1, j) \odot \cdots \odot D(n-1, j)$
- Operator must be associative: $(a \odot b) \odot c = a \odot (b \odot c)$
- MPI predefined operators are also commutative: $a \odot b = b \odot a$

- \texttt{MPI\_Reduce}(sbuf, rbuf, root, op=MPI.OP)
Global computation routines
AllReduce + Reduce_Scatter

• MPI_Allreduce(sbuf, rbuf, op=MPI.OP)
  • Access the reduced data on all processors rather than the root process

• MPI_Reduce_scatter (sbuf, rbuf, recvcounts, op=MPI.SUM)
  • recvcounts indicate the number of elements in result distributed to each process (vector)
  • Array must be identical on all calling processes
Scan

- $d(k, j)$ - the $j$th data item in process $k$ before the scan
- $D(k, j)$ - the $j$th data item in process $k$ after returning from scan

For processor $k$:
- $D(k, j) = d(0, j) \odot d(1, j) \odot \cdots \odot d(k, j)$

- `MPI_Scan(sbuf, rbuf, op=MPI.OP)`
Predefined reduce operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>maximum</td>
<td>integer, float</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>minimum</td>
<td>integer, float</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>sum</td>
<td>integer, float</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>product</td>
<td>integer, float</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>logical and</td>
<td>integer</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>bit-wise and</td>
<td>integer, MPI_BYTE</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>logical or</td>
<td>integer</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>bit-wise or</td>
<td>integer, MPI_BYTE</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>logical xor</td>
<td>integer</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>bit-wise xor</td>
<td>integer, MPI_BYTE</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>max value and location</td>
<td>combination of int, float, double, and long double</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>min value and location</td>
<td>combination of int, float, double, and long double</td>
</tr>
</tbody>
</table>
Performance issues

• A great deal of hidden communication takes place with collective communication
• Performance depends greatly on the particular implementation of MPI
• Because there may be forced synchronization at the end of the used function, not always best to use collective communication