Today's agenda

- Homework discussion
- Bandwidth and latency in theory and in practice
- Paired and Nonblocking Pt2Pt Communications
- Other Point to Point routines
- Collective Communications: One-with-All
- Collective Communications: All-with-All
A bit of theory

- Let “zero-message” latency be $l$ and asymptotic bandwidth be $BW$:
  - Then most simplistic (w/o contention) linear model for the time to communicate message of size $L$ is $T_c = l + L / BW$
  - In fact the model should be piecewise linear to distinguish between (small,) eager and rendezvous protocols.
  - Moreover, the $BW$ that should be used is independent of $L$.
    - Cost of memory copies can be a factor in $BW$.
      - For small enough $L$, cache effects increase $BW$
      - For very large $L$, TLB misses decrease $BW$
Latency Examples

- y-axis intercept is zero message latency
- Note the difference between Gigabit Ethernet, IPoIB, DDR Infiniband and Shared Memory (same and different socket) performance
Bandwidth Examples

- Plot of the “effective bandwidth” \( B_{\text{ef}} = \frac{L}{T_c} = 2L/\text{RTT} \)
- Note cache effects, noise.

Parallel Programming for Multicore Machines Using OpenMP and MPI
Deadlock around the ring

- Consider a ring communication scenario where everyone talks to one's neighbour to the right around a circle. If synchronous blocking communications are used (MPI_Ssend or MPI_Send for large messages) the messages never get delivered as everybody needs to send before receiving!

Figure by MIT OpenCourseWare.

Parallel Programming for Multicore Machines Using OpenMP and MPI
Bidirectional Exchange

- Consider the cases where two processes are exchanging messages concurrently or a process is sending a message while receiving another.

- There are two routines that provide an optimized deadlock free macro for this operation:
  - `MPI_Sendrecv(int *sendbuf, int sendcnt, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvnt, MPI_Datatype recvtype, int src, int recvtag, MPI_Comm comm, MPI_Status *stat)`
  - `MPI_Sendrecv_replace` does not use the `recvbuf`, `recvnt` and `recvtype` arguments as the source array gets overwritten (like a swap)
Nonblocking communications

- The situation can be rectified by using nonblocking communication routines that return immediately, without making sure that the data has been safely taken care of.
  - That way after the send the receive can be posted and the deadlock is avoided
  - But **beware**: Until such a time that the communication is successfully completed, no pointer input arguments to the routines may be modified as they wrong data/parameters will be used when the communication does take place.
  - This is unlike the situation with the blocking comms where upon return from the call, one is free to reuse the args.
  - MPI_Ixxxx instead of MPI_xxxx for the names
MPI nonblocking standard send

- `MPI_Isend(void *buf, int cnt, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req)`
- `MPI_ISEND(buf, cnt, type, dest, tag, comm, req, ierr)`
- `MPI_Wait(MPI_Request *req, MPI_Status *stat)`
- `MPI_WAIT(req, stat, ier)`

Call `MPI_Isend`, store the request handle, do some work to keep busy and then call `MPI_Wait` with the handle to complete the send.

- `MPI_Isend` produces the request handle, `MPI_Wait` consumes it.
- The status handle is not actually used.
MPI nonblocking receive

- **MPI_Irecv**(void *buf, int cnt, MPI_Datatype type, int src, int tag, MPI_Comm comm, MPI_Request *req)
- **MPI_IRecv**(buf, cnt, type, src, tag, comm, req, ier)
- **MPI_Wait**(MPI_Request *req, MPI_Status *stat)
- **MPI_WAIT**(req, stat, ier)

Call **MPI_Irecv**, store the request handle, do some work to keep busy and then call **MPI_Wait** with the handle to complete the receive.

- **MPI_Irecv** produces the request handle, **MPI_Wait** consumes it.

- In this case the status handle is actually used.
Deadlock avoidance

- If the nonblocking sends or receives are called back-to-back with MPI_Wait we basically retrieve the blocking behavior as MPI_Wait is a blocking call.

- To avoid deadlock we need to interlace nonblocking sends with blocking receives, or nonblocking receives with blocking sends; the nonblocking calls always precede the blocking ones. Using both nonblocking calls may land us in trouble again unless we reverse the order of Wait calls, or interlace the order of send and receive calls (even #P).

Figure by MIT OpenCourseWare.
Other nonblocking sends

- For each blocking send, a nonblocking equivalent:
  - MPI_Issend: nonblocking synchronous send
  - MPI_Ibsend: nonblocking asynchronous send
  - MPI_Irsend: nonblocking ready send
- Take care not to confuse *nonblocking* send with *asynchronous* send although the terminology has been used interchangeably in the past!
  - A successful blocking asynchronous send returns very quickly and the send buffer can be reused.
  - Any *nonblocking call* returns immediately and the buffer cannot be tampered with until the corresponding blocking MPI_Wait call has returned!
Nonblocking Synchronous Send

• For example, an MPI_Issend() works like using an unattended fax machine. You set up the fax to be sent, go away but need to come and check if all's been sent.

Figure by MIT OpenCourseWare.
Implementing nonblocking comms

- The actual communication in the case of the nonblocking calls can take place at any given time between the call to the MPI_Isend/Irecv operation and the corresponding MPI_Wait. Fortran 90 issues!

- The moment it actually happens is implementation dependent and in many cases it is coded to take place mostly within MPI_Wait.

- On systems with "intelligent" network interfaces it is possible for communications to be truly taking place concurrently with the computational work the sending process is performing, thus allowing for computation to "hide" communication; otherwise nonblocking calls just help avoid deadlock without helping performance.
Testing instead of Waiting

- **MPI_Test**(MPI_Request *req, int *flag, MPI_Status *stat)
- **MPI_TEST**(req,flag,stat,ier), logical flag

Instead of calling the blocking **MPI_Wait** call **MPI_Test**; if flag is true, the message has been received with more information in stat and ier. Otherwise the routine returns and can be called after a while to test for message reception again.

- On systems where one can have real overlap of communication with computation **MPI_Test** allows finer control over communication completion times.
All, some and any

- Suppose one has a large number of outstanding nonblocking calls to wait or test for. MPI provides us with special shorthand routines for this case:

  - **MPI_Waitall/MPI_Testall** deal with arrays of requests and statuses as their arguments. They test for all pending communication requests.

  - Ditto for **MPI_Waitsome/MPI_Testsome** but they also mark the locations of successful operations in another index array. They return after some time at least one completion on pending requests (usually more).

  - Finally **MPI_Waitany/MPI_Testany** will test for all the pending requests and return if they come across one that is completed or if none are completed.
Other comms routines & handles

- **MPI_Probe/MPI_IProbe** will check for a message awaiting to be received but will not actually receive it - one needs to call MPI_Recv/MPI_Irecv for that.

- **MPI_Cancel(MPI_Request *req)** will mark a pending send or receive for cancellation. One still needs to call MPI_Wait or MPI_Test or MPI_Request_free to free the request handle. The message may still be delivered at that time! MPICH based implementations beware!

- **MPI_Test_cancelled**

- **MPI_Send_init, MPI_Recv_init and MPI_Start, MPI_Startall**: Persistent comms

- **MPI_PROC_NULL, MPI_REQUEST_NULL**
Probing

IF (rank.EQ.0) THEN

    CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)

ELSE IF(rank.EQ.1) THEN

    CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)

ELSE  ! rank.EQ.2

    DO i=1, 2

        CALL MPI_PROBE(MPI_ANY_SOURCE, 0, comm, status, ierr)

        IF (status(MPI_SOURCE) = 0) THEN

            CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, status, ierr)

        ELSE

            CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, status, ierr)

        END IF

    END DO

END IF
Persistent Communications

- In the case of very regular communications (say inside a loop), some communication overhead can be avoided by setting up a persistent communication request ("half" channel or port). **There is no binding of receiver to sender!**

- `MPI_Send_init(buf, count, datatype, dest, tag, comm, request)` sets up persistent sends. The request is inactive and corresponds to an `Isend()`. Corresponding initialization calls for `Bsend`, `Ssend` and `Rsend` exist.

- `MPI_Recv_init(buf, count, datatype, source, tag, comm, request)` sets up persistent receives. The request is inactive and corresponds to an `Irecv()`.
Persistent Communications (cont)

- To activate a persistent send or receive pass the request handle to `MPI_Start(request)`.

- For multiple persistent communications employ `MPI_Startall(count, array_of_requests)`. This processes request handles in some arbitrary order.

- To complete the communication, `MPI_Wait()/Test()` and friends are needed. Once they return, the request handle is once again inactive but allocated. To deallocate it `MPI_Request_free()` is needed. Make sure it operates on an inactive request handle.

- Persistent sends can be matched with blocking or non-blocking receives and vice-versa for the receives.
Wildcards & Constants

- **MPI_PROC_NULL**: operations specifying this do not actually execute. Useful for not treating boundary cases separately to keep code cleaner.

- **MPI_REQUEST_NULL**: The value of a null handle, after it is released by the MPI_Wait()/Test() family of calls or by MPI_Request_free()

- **MPI_ANY_SOURCE**: Wild card for source

- **MPI_ANY_TAG**: Wildcard for tag

- **MPI_UNDEFINED**: Any undefined return value
Collective Comms

- Collective communications involve a group of processes, namely all processes in a communicator

- All processes call the routine which is blocking and has no tag; **assumed to be implementor optimized**!

- Any receive buffers all have to be the same size and be distinct from send buffers (Fortran semantics)

- Operations can be one-to-all, all-to-one and all-to-all in nature and combinations thereof.

- They can involve data exchange and combination as well as reduction operations

- Many routines have a "vector" variant MPI_xxxxv
Synchronization

- **MPI_Barrier(MPI_Comm comm)**
- **MPI_BARRIER(comm, ier)**
- Forces synchronization for:
  - timing purposes
  - non-parallel I/O purposes
  - debugging
- Costly for large numbers of processes, try to avoid.

Figure by MIT OpenCourseWare.
Broadcast

- `MPI_Bcast(void *buf, int cnt, MPI_Datatype type, int root, MPI_Comm comm)`
- `MPI_BCAST(buf, cnt, type, root, comm, ier)`
- root has to be the same on all procs, can be nonzero

![Diagram showing broadcast process](https://example.com/diagram.png)

Figure by MIT OpenCourseWare.
Gather

- **MPI_Gather**(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvnt, MPI_Datatype recvtype, int root, MPI_Comm comm)

- Make sure recvbuf is large enough on root where it matters, elsewhere it is ignored

- **MPI_Gatherv** has additional arguments for variable recvnt, and output stride

---

Parallel Programming for Multicore Machines Using OpenMP and MPI
MPI Scatter

**MPI_Scatter**

- `MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)`

- Make sure `recvbuf` is large enough on all procs, `sendbuf` matters only on root

- **MPI_Scatterv** has additional arguments for variable `sendcnt`, and input stride

---

**Figure**

- Before scatter: `A B C D E`
- After scatter: `A B C D E`, `E`

- E.g., root = 1

---

Parallel Programming for Multicore Machines Using OpenMP and MPI

Figure by MIT OpenCourseWare.
Gather to all

- MPI_Allgather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recv cnt, MPI_Datatype recvtype, MPI_Comm comm)
- Make sure recvbuf is large enough on all procs
- MPI_Allgatherv has additional arguments for variable recvcnt, and output stride
- Can be thought of as an MPI_Gather followed by an MPI_Bcast, with an unspecified root process
Vector Variants

```
offset = 0;
for (i=0; i<gsze; ++i) {
    displs[i] = offset;
    offset += stride[i];
    scounts[i] = 100 - i;
}
```

Figure by MIT OpenCourseWare.

Parallel Programming for Multicore Machines Using OpenMP and MPI
Differences in scattering

• Non-contiguous parts, irregularly spaced apart, all end at the head of the receive buffers.
Binary trees

Figure by MIT OpenCourseWare.
Tree Variants

**Sequential tree**
Root process to all others in (n-1) steps for n processes

**Binary tree**
All processes (apart from root) receive and send to at most 2 others. Use 2l.log2(n+1).l-1.5±0.5 steps for n processes

**Chain**
Message passed in a chain to all others in (n-1) steps for n processes

**Binomial tree**
In each step a process with data sends to its “half-the-remaining-size” neighbor. Use l.log2(n-1).l+1 steps for n processes

Parallel Programming for Multicore Machines Using OpenMP and MPI
All to All Personalized Comm

- **MPI_Alltoall**(`void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm`)

- Everybody sends something different to everyone else, like a scatter/gather for all. If what was getting sent was the same it would be like a bcast/gather for all.

- This is the most stressful communication pattern for a communication network as it floods it with messages: \( P*(P-1) \) for \( P \) procs for certain direct implementations.

- **MPI_Alltoallv** has additional arguments for variable sendcnt and recvnt, and input and output strides

- Other MPI-2 variants, at the heart of matrix transpose!
Matrix transpose

- Consider the simplest case of 1 element per proc.
- Transpose accomplished with 1 call.
- Otherwise use derived datatypes or pack/unpack

<table>
<thead>
<tr>
<th>Send Buffer</th>
<th>Receive Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data</strong></td>
<td><strong>Data</strong></td>
</tr>
<tr>
<td>A₀ B₀ C₀ D₀ E₀ F₀</td>
<td>A₀ A₁ A₂ A₃ A₄ A₅</td>
</tr>
<tr>
<td>A₁ B₁ C₁ D₁ E₁ F₁</td>
<td>B₀ B₁ B₂ B₃ B₄ B₅</td>
</tr>
<tr>
<td>A₂ B₂ C₂ D₂ E₂ F₂</td>
<td>C₀ C₁ C₂ C₃ C₄ C₅</td>
</tr>
<tr>
<td>A₃ B₃ C₃ D₃ E₃ F₃</td>
<td>D₀ D₁ D₂ D₃ D₄ D₅</td>
</tr>
<tr>
<td>A₄ B₄ C₄ D₄ E₄ F₄</td>
<td>E₀ E₁ E₂ E₃ E₄ E₅</td>
</tr>
<tr>
<td>A₅ B₅ C₅ D₅ E₅ F₅</td>
<td>F₀ F₁ F₂ F₃ F₄ F₅</td>
</tr>
</tbody>
</table>

Figure by MIT OpenCourseWare.
Global Reductions

- An efficient way to perform an **associative binary operation** on a set of data spread over processes in a communicator

- Operation can have an MPI defined handle:
  - **MPI_MAX, MPI_MIN, MPI_MAXLOC, MPI_MINLOC**
  - **MPI_SUM, MPI_PROD**
  - **MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR**

  Operating on datatypes that make sense

- **MPI_MAXLOC** and **MPI_MINLOC** require special datatypes, already predefined (**note the Fortran ones and for more information look up the standard**).
User defined binary ops

- MPI provides for user defined binary operations for the reduction routines:

- `MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)`

- `MPI_OP_CREATE(function, commute, op, ierr)`, external function, logical commute

- If commute is true, then the operation is assumed to be commutative. Otherwise only the required associativity rule applies.

- The function (*non-MPI*) needs to be of the form:
  
  - `typedef void MPI_User_function (void *invec, void *inoutvec, int *len, MPI_Datatype *type)`
Reduce

- MPI_Reduce(void *sendbuf, void *recvbuff, int cnt, MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm)
- MPI_REDUCE(sendbuf, recvbuf, cnt, type, op, root, comm, ier)

Figure by MIT OpenCourseWare.
Reduce to all

- **MPI_Allreduce**(void *sendbuf, void *recvbuff, int cnt, MPI_Datatype type, MPI_Op op, MPI_Comm comm)
- **MPI_ALLREDUCE**(sendbuf, recvbuf, cnt, type, op, comm, ier)

Figure by MIT OpenCourseWare.
Reduce - Scatter

- `MPI_Reduce_scatter(void *sendbuf, void *recvbuff, int *recvnt, MPI_Datatype type, MPI_Op op, MPI_Comm comm)`

- `MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvnt, type, op, comm, ier)`

- Can be considered as a

  ```c
  MPI_Reduce(sendbuf, tmpbuf, cnt, type, op, root, comm);
  MPI_Scatterv(tmpbuf, recvnt, displs, type, recvbuff, recvnt[myid], type, root, comm);
  ```

  where `cnt` is the total sum of the `recvnt` values and `displs[k]` is the sum of the `recvnt` for up to processor `k-1`.

- Implementations may use a more optimal approach
Prefix operations

- \texttt{MPI\_Scan(void *sendbuf, void *recvbuf, int cnt, MPI\_Datatype type, MPI\_Op op, MPI\_Comm comm)}
- \texttt{MPI\_SCAN(sendbuf, recvbuf, cnt, type, op, comm, ier)}
- The scan is inclusive: result on proc P includes its data

![Diagram showing before and after MPI_SCAN](image-url)
Prefix operations in action

MPI_Scan

• Computes the partial reduction (scan) of input data in a communicator

• count=1;
  MPI_Scan(send,recv,count,MPI_INT,MPI_PROD, MPI_COMM_WORLD)

![Diagram of prefix operations using MPI_Scan](image)

Figure by MIT OpenCourseWare.