Introduction to MPI

Thomas Hauser, thomas.hauser@colorado.edu
Timothy Brown, timothy.brown-1.colorado.edu
University of Colorado Boulder

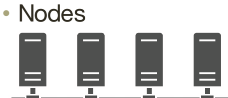
Feb 9-10, 2016

Outline

- Background
- Message Passing Interface
- Communicator
- Sending and receiving
- Collective operations

Parallelism on Many Levels

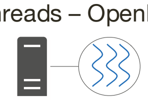
- Nodes



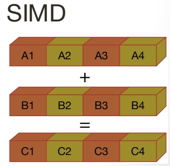
- Instructions – ILP

I1: add R1, R2, R3
I2: sub R4, R1, R5
I3: xor R10, R2, R11

- Threads – OpenMP

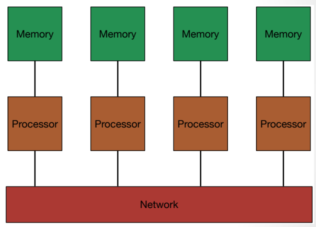


- Data - SIMD



Distributed Memory Computer

- Processors have different content in memory
- Data exchange by message passing



Message passing

- Most natural and efficient paradigm for distributed-memory systems
- Two-sided, send and receive communication between processes
- Efficiently portable to shared-memory or almost any other parallel architecture:
  "assembly language of parallel computing" due to universality and detailed, low-level control of parallelism

More on message passing

- Provides natural synchronization among processes (through blocking receives, for example), so explicit synchronization of memory access is unnecessary
- Sometimes deemed tedious and low-level, but thinking about locality promotes
  - good performance,
  - scalability,
  - portability
- Dominant paradigm for developing portable and scalable applications for massively parallel systems

Programming a distributed-memory computer

- MPI (Message Passing Interface)
  also PVM (Parallel Virtual Machine) and others
- Message passing standard, universally adopted
  =
  library of communication routines
  callable from C, C++, Fortran, (Python)
- 125+ functions—we will study small subset
  may be possible to improve performance with more

MPI standard

- MPI has been developed in three major stages MPI 1 – 1994
  - MPI 2 – 1996
  - MPI 3 – 2012
- MPI Forum
  http://www.mpi-forum.org/docs/docs.html
- MPI Standard
  http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf
- Using MPI and Using Advanced MPI
  http://www.mcs.anl.gov/research/projects/mpi/usingmpi/
- Online MPI tutorial
  http://mpitutorial.com/beginner-mpi-tutorial/

MPI-1

- Features of MPI-1 include
  - Point-to-point communication
  - Collective communication process
  - Groups and communication domains
  - Virtual process topologies
  - Environmental management and inquiry
  - Profiling interface bindings for Fortran and C

MPI-2

- Additional features of MPI-2 include:
  - Dynamic process management input/output
  - One-sided operations for remote memory access (update or interrogate)
  - Memory access bindings for C++
  - Parallel I/O

MPI-3

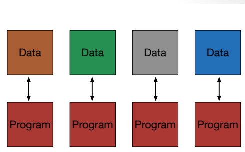
- Non-blocking collectives
- New one-sided communication operations
- Fortran 2008 bindings

MPI Implementations

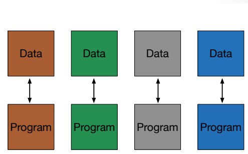
- MPICH
  ftp://ftp.mcs.anl.gov/pub/mpi
- OpenMPI
  http://www.open-mpi.org
- Intel MPI
  https://software.intel.com/en-us/intel-mpi-library
- SGI
- Cray
- IBM

Programming Models

- Single Program Multiple Data (SPMD)
  - Same program runs on each process.



- Multiple Programs Multiple Data (MPMD)
  - Different programs runs on each process.



Compiling MPI Programs

- Wrapper scripts for the compiler

C
```
mpicc –o a.out a.c
```

Fortran
```
mpifc –o a.out a.f90
```

- Automatically sets
  - Include path
  - Library path
  - Links the MPI library

MPI programs use SPMD model

- Same program runs on each process
- Build executable and link with MPI library
- User determines number of processes and on which processors they will run

Execution

- You can run a MPI program with the following commands
  - mpiexec –n 48 ./a.out
- With SLURM
  - srun –N 4 –ntasks-per-node=12 ./a.out

Programming in MPI

```
use mpi

integer :: ierr
call MPI_init(ierr)
.
.
.
call MPI_Finalize(ierr)
```

```
#include "mpi.h"

int ierr;
ierr = MPI_Init(&argc, &argv);
.
.
.
ierr = MPI_Finalize();
```

C returns error codes as function values,
Fortran requires arguments (ierr)

MPI Communicator

- A collection of processors of an MPI program
- Used as a parameter for most MPI calls.
- Processors with in a communicator have a number
  - Rank: 0 to n-1
- `MPI_COMM_WORLD`
  - Contains all processors of your program run
- You can create new communicators that are subsets
  - All even processors
  - The first processor
  - All but the first processor

Programming in MPI

```
use mpi
integer ierr

call MPI_init(ierr)
call MPI_COMM_RANK( MPI_COMM_WORLD, id, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )
.
.
.
call MPI_Finalize(ierr)
```

Determine process id or *rank* (here = id)
And number of processes (here = nprocs)

Determine the processor running on

- ierr = MPI_Get_processor_name(proc_name, &length);

MPI Scientific Hello world

- Write a Scientific hello world program
  - Compute: exp(rank)
- Output should be:
  - Hello from process %d on node %s
  - Exp(%d) = %f
  - Number of mpi processes = %d

Message sending and receiving

- Which process is sending the message?
- Where is the data on the sending process?
- What kind of data is being sent?
- How much data is there?
- Which process is going to receive the message?
- Where should the data be stored on the receiving process?
- What amount of data is the receiving process prepared to accept?

Blocking send

- call MPI_SEND(

message,	e.g., my_partial_sum,
count,	number of values in msg
data_type,	e.g, MPI_DOUBLE_PRECISION,
destination,	e.g., myid + 1
tag,	some info about msg, e.g., store it
communicator,	e.g., MPI_COMM_WORLD,
ierr	

)

All arguments are inputs.

Message envelop

- Source – implicitly determined by identity of sender
- Destination - argument
- Tag – argument
- Communicator – argument

Blocking?

- MPI_send
  - does not return until the message data and envelope have been buffered in matching receive buffer or temporary system buffer.
  - can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver.
  - MPI buffers or not, depending on availability of space
  - **non-local**: successful completion of the send operation may depend on the occurrence of a matching receive.

Fortran MPI Data Types

MPI_CHARACTER
MPI_COMPLEX, MPI_COMPLEX8, also 16 and 32
MPI_DOUBLE_COMPLEX
MPI_DOUBLE_PRECISION
MPI_INTEGER
MPI_INTEGER1, MPI_INTEGER2, also 4 and 8
MPI_LOGICAL
MPI_LOGICAL1, MPI_LOGICAL2, also 4 and 8
MPI_REAL
MPI_REAL4, MPI_REAL8, MPI_REAL16

Numbers = numbers of bytes
Somewhat different in C—see text or Google it

C MPI Datatypes

MPI_CHAR	8-bit character
MPI_DOUBLE	64-bit floating point
MPI_FLOAT	32-bit floating point
MPI_INT	32-bit integer
MPI_LONG	32-bit integer
MPI_LONG_DOUBLE	64-bit floating point
MPI_LONG_LONG	64-bit integer
MPI_LONG_LONG_INT	64-bit integer
MPI_SHORT	16-bit integer
MPI_SIGNED_CHAR	8-bit signed character
MPI_UNSIGNED	32-bit unsigned integer
MPI_UNSIGNED_CHAR	8-bit unsigned character
MPI_UNSIGNED_LONG	32-bit unsigned integer
MPI_UNSIGNED_LONG_LONG	64-bit unsigned integer
MPI_UNSIGNED_SHORT	16-bit unsigned integer
MPI_WCHAR	Wide (16-bit) unsigned character

Blocking receive

- call MPI_RECV(

message,	e.g., my_partial_sum,
count,	number of values in msg
data_type,	e.g, MPI_DOUBLE_PRECISION,
source,	e.g., myid - 1
tag,	some info about msg, e.g., store it
communicator,	e.g., MPI_COMM_WORLD,
status,	info on size of message received
ierr	

)

Blocking receive

- Process must wait until message is received to return from call.
- Stalls progress of program BUT
  - blocking sends and receives enforce process synchronization
  - so enforce consistency of data

The arguments

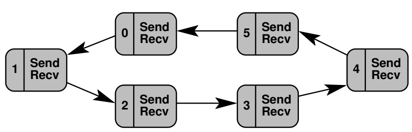
- outputs: message, status
- count*size of data_type determines size of receive buffer:
  --too large message received gives error,
  --too small message is ok
- status must be decoded if needed
  - MPI_Get_Count(status, datatype, ierror)
  - status(MPI_SOURCE) status.MPI_SOURCE
  - status(MPI_TAG) status.MPI_TAG
  - status(MPI_ERROR) status.MPI_ERROR

Wildcards

- MPI_ANY_SOURCE
- MPI_ANY_TAG

- Send must send to specific receiver
- Receive can receive from arbitrary sender

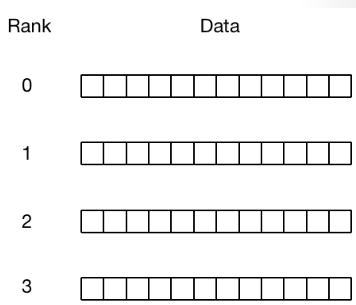
Sending data in a ring



- Store the data in array of size nprocs x n
- Each process sends message to neighbor with higher rank
  - N elements to id+1
- Receives values from neighbor with lower rank
  - N elements from id -1
- At the end sum up and print local results

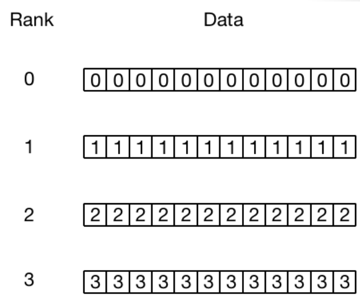
Ring Data Layout

- Nprocs=4
- N=3



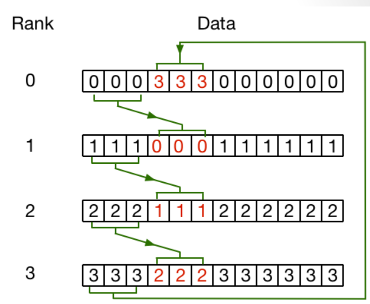
Ring Data Layout

- Nprocs=4
- N=3



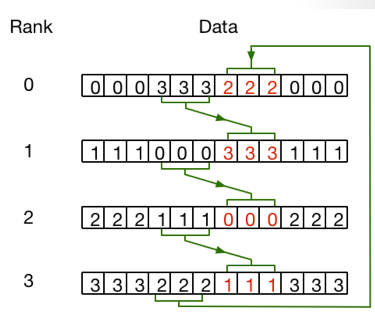
Ring Data Layout

- Nprocs=4
- N=3



Ring Data Layout

- Nprocs=4
- N=3



Ring Data Layout

- Nprocs=4
- N=3

Rank	Data
0	0 0 0 3 3 3 2 2 2 1 1 1
1	1 1 1 0 0 0 3 3 3 2 2 2
2	2 2 2 1 1 1 0 0 0 3 3 3
3	3 3 3 2 2 2 1 1 1 0 0 0

Ring Data Layout

- Nprocs=4
- N=3

Rank	Data
0	0 0 0 3 3 3 2 2 2 1 1 1
1	1 1 1 0 0 0 3 3 3 2 2 2
2	2 2 2 1 1 1 0 0 0 3 3 3
3	3 3 3 2 2 2 1 1 1 0 0 0

Deadlock

- Deadlock: process waiting for a condition that will never become true
- Easy to write send/receive code that deadlocks
  - Two processes: both receive before send
  - Send tag doesn't match receive tag
  - Process sends message to wrong destination process

Non-Blocking Send & Receive

- Same syntax as MPI_Send() and MPI_Recv()
  - Addition of a request handle argument.
- Calls return immediately
- Data in the buffer (send and receive) may not be accessed until operations is complete.
- Send and receive are completed by
  - MPI_Test
  - MPI_Wait

MPI_ISEND (buf, cnt, dtype, dest, tag, comm, request, ierr)

- Same syntax as MPI_SEND with the addition of a request handle
- Request is a handle (int in Fortran; MPI_Request in C) used to check for completeness of the send
- This call returns immediately
- Data in buf may not be accessed until the user has completed the send operation
- The send is completed by a successful call to MPI_TEST or a call to MPI_WAIT

MPI_IRECV(buf, cnt, dtype, source, tag, comm, request, ierr)

- Same syntax as MPI_RECV except status is replaced with a request handle
- Request is a handle (int in Fortran MPI_Request in C) used to check for completeness of the recv
- This call returns immediately
- Data in buf may not be accessed until the user has completed the receive operation
- The receive is completed by a successful call to MPI_TEST or a call to MPI_WAIT

MPI_WAIT (request, status, ierr)

- Request is the handle returned by the non-blocking send or receive call
- Upon return, status holds source, tag, and error code information
- This call does not return until the non-blocking call referenced by *request* has completed
- Upon return, the request handle is freed
- If *request* was returned by a call to MPI_ISEND, return of this call indicates nothing about the destination process

MPI_WAITANY (count, requests, index, status, ierr)

- Requests is an array of handles returned by non-blocking send or receive calls
- Count is the number of requests
- This call does not return until a non-blocking call referenced by one of the *requests* has completed
- Upon return, index holds the index into the array of requests of the call that completed
- Upon return, status holds source, tag, and error code information for the call that completed
- Upon return, the request handle stored in requests[index] is freed

MPI_WAITALL (count, requests, statuses, ierr)

- *requests* is an array of handles returned by non-blocking send or receive calls
- *count* is the number of requests
- This call does not return until all non-blocking call referenced by *requests* have completed
- Upon return, *statuses* hold source, tag, and error code information for all the calls that completed
- Upon return, the request handles stored in *requests* are all freed

MPI_TEST (request, flag, status, ierr)

- *request* is a handle returned by a non-blocking send or receive call
- Upon return, *flag* will have been set to true if the associated non-blocking call has completed. Otherwise it is set to false
- If *flag* returns true, the request handle is freed and *status* contains source, tag, and error code information
- If *request* was returned by a call to MPI_ISEND, return with *flag* set to true indicates nothing about the destination process

MPI_TESTANY (count, requests, index, flag, status, ierr)

- *requests* is an array of handles returned by non-blocking send or receive calls
- *count* is the number of requests
- Upon return, *flag* will have been set to true if one of the associated non-blocking call has completed. Otherwise it is set to false
- If *flag* returns true, *index* holds the index of the call that completed, the request handle is freed, and *status* contains source, tag, and error code information

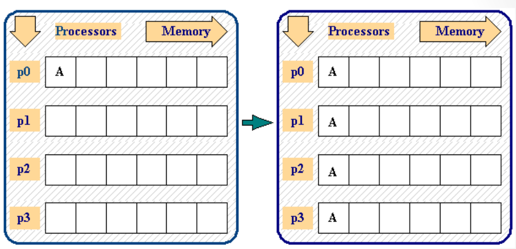
MPI_TESTALL (count, requests, flag, statuses, ierr)

- *requests* is an array of handles returned by non-blocking send or receive calls
- *count* is the number of requests
- Upon return, *flag* will have been set to true if ALL of the associated non-blocking call have completed. Otherwise it is set to false
- If *flag* returns true, all the request handles are freed, and *statuses* contains source, tag, and error code information for each operation

Collective communication

- Other
  - MPI_Barrier()
- One-To-All
  - MPI_Bcast(), MPI_Scatter(), MPI_Scatterv()
- All-To-One
  - MPI_Gather(), MPI_Gatherv(), MPI_Reduce()
- All-To-All
  - MPI_Allgather(), MPI_Allgatherv(), MPI_Allreduce()

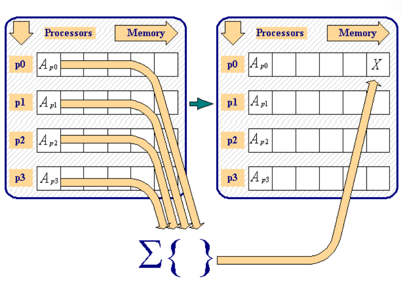
Broadcast



send_count = 1;
root = 0;
MPI_Bcast (&a, send_count, MPI_INT, root, comm)

Figure from MPI-tutor: http://www.citutor.org/index.php

Reduction



count = 1;
rank = 0;
MPI_Reduce (&a, &x, count, MPI_REAL, MPI_SUM, rank, MPI_COMM_WORLD);

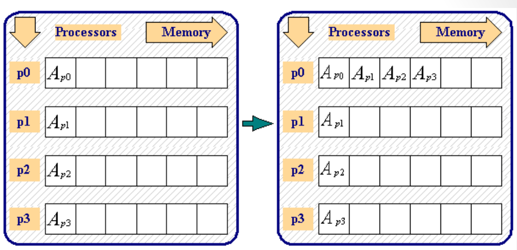
Figure from MPI-tutor: http://www.citutor.org/index.php

Reduction operations

Operation	Description
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bitwise xor

MPI_MINLOC computes a global minimum and an index attached to the minimum value – can be used to determine the rank of the process containing the minimum value
MPI_MAXLOC computes a global maximum and an index attached to the rank of the process containing the minimum value

Gather



send_count = 1;
recv_count = 1;
recv_rank = 0;
MPI_Gather (&a, send_count, MPI_REAL, &a, recv_count, MPI_REAL, recv_rank, MPI_COMM_WORLD);
Figure from MPI-tutor: http://www.citutor.org/index.php

All-gather

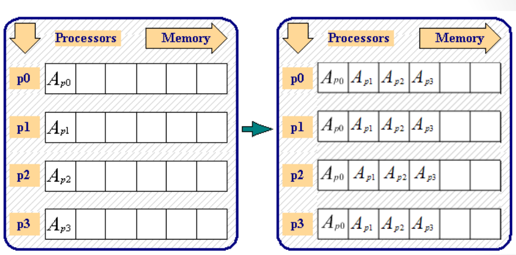
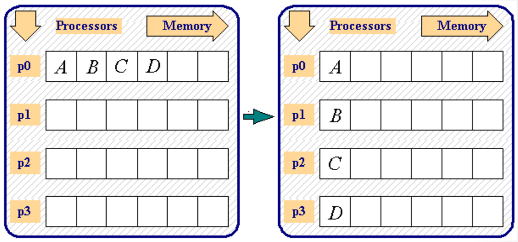


Figure from MPI-tutor: http://www.citutor.org/index.php

Scatter




```
recv_count = 1;
send_rank = 0;
MPI_Scatter ( &a, send_count, MPI_REAL,
          &a, recv_count, MPI_REAL,
          send_rank, MPI_COMM_WORLD );
```

Figure from MPI-tutor: http://www.citutor.org/index.php

Research Computing @ CU Boulder Introduction to MPI - USGS 2/9/16