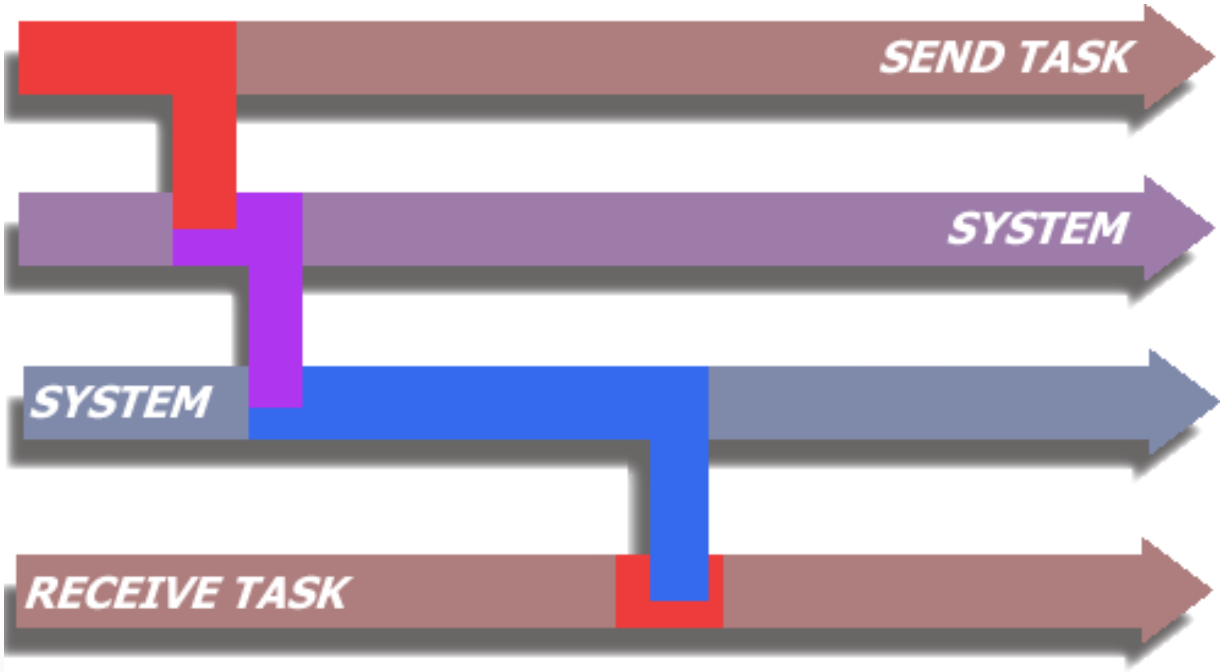
cse5441 - parallel computing

introduction to MPI

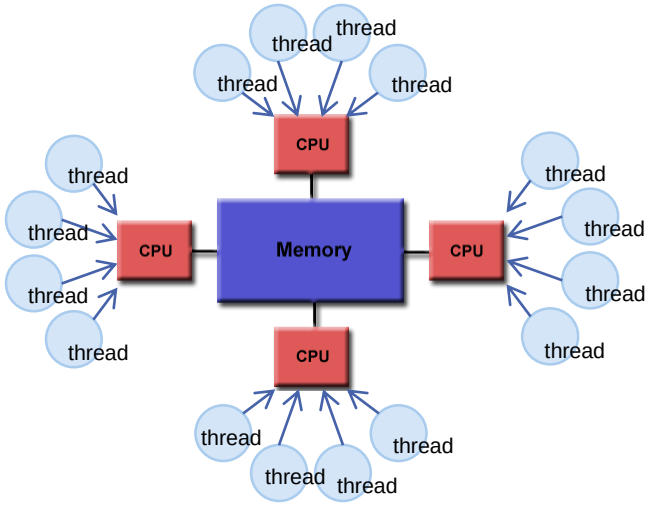


https://computing.llnl.gov/tutorials/mpi_performance/images/interrupt.gif

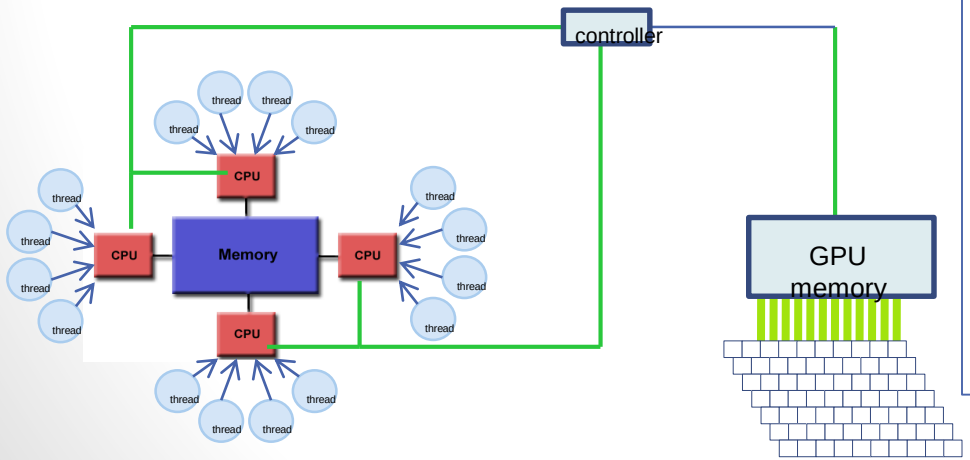
what is MPI ?

- yet another multi-processing paradigm
- a set of libraries
- an API
- Multi-Processing, literally
- communications based

MP recap



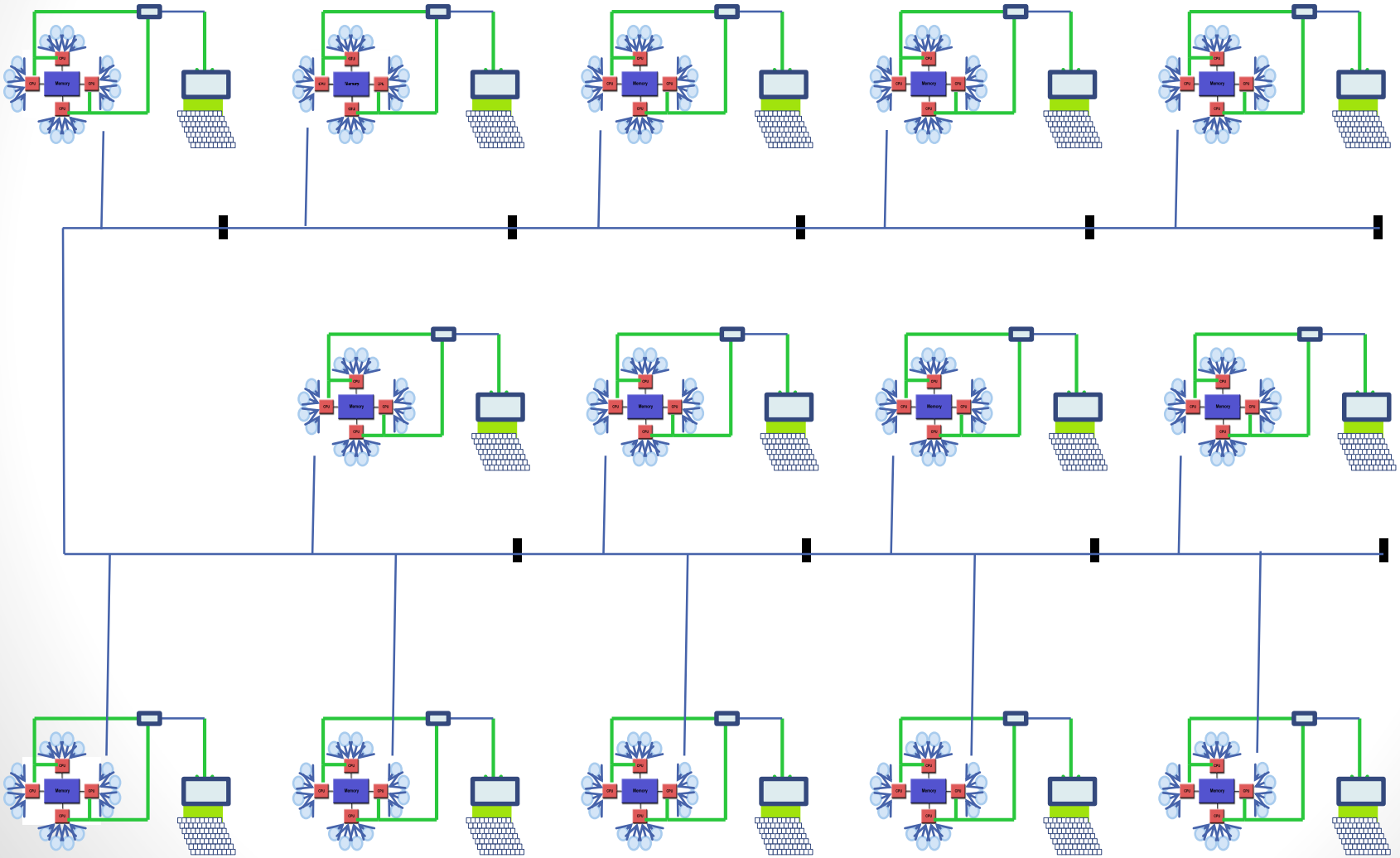
- pthreads: single global shared memory
- OpenMP: single global shared memory



cuda:

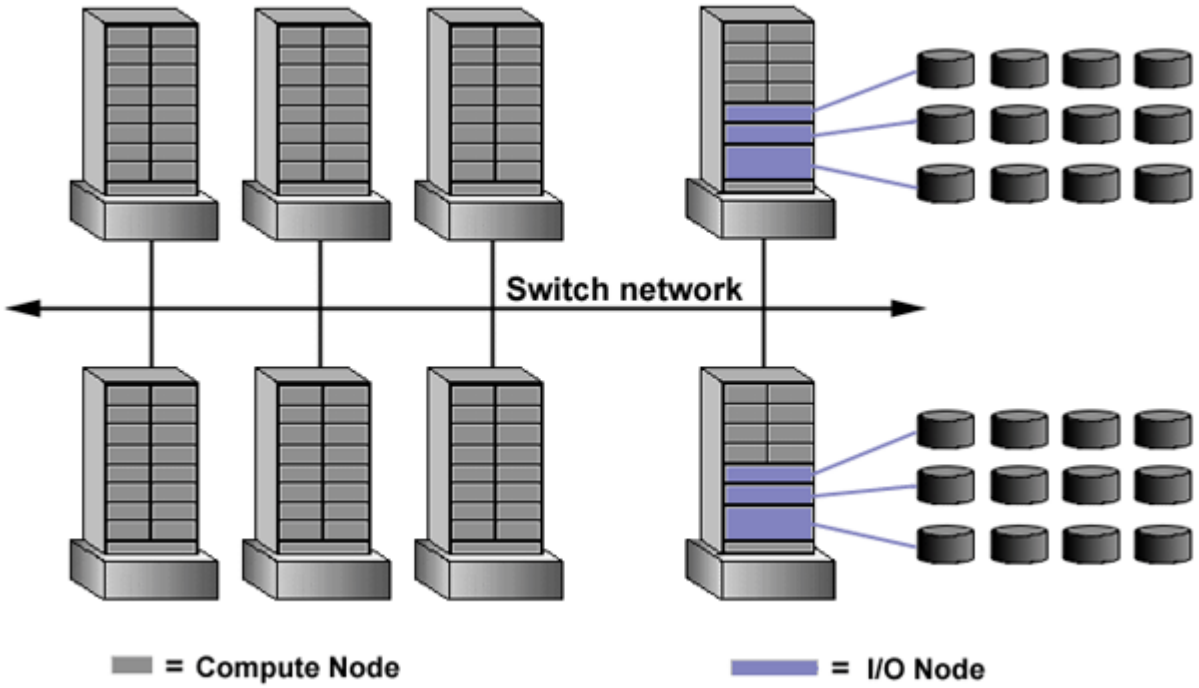
- global host shared memory
- hierarchical device shared memory

massively parallel systems



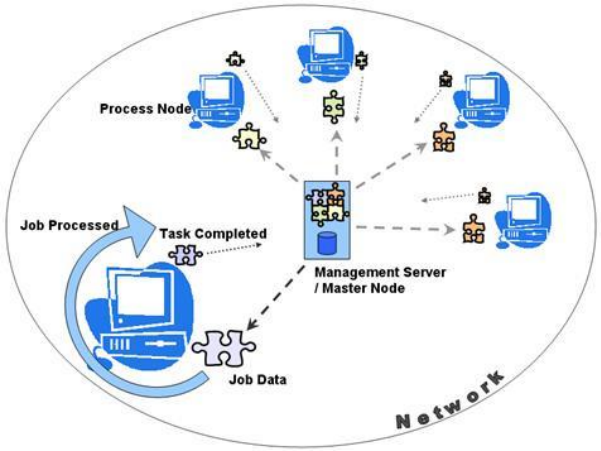
MPI architecture

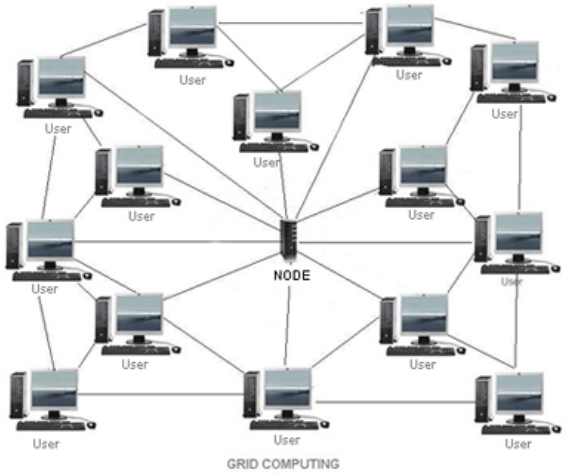
- MP at the process level
- MP with distributed nodes
  - distributed non-continguous memory
  - non-homogenous compute engines

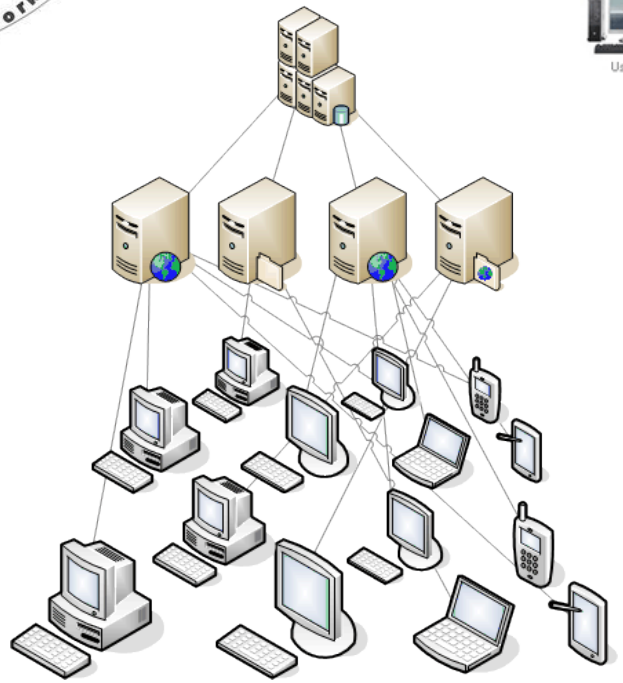


https://computing.llnl.gov/tutorials/ibm_sp/images/gpfs02.gif

distributed computing







http://chessbrain.net/LFBOF2005/images/20kview.gif
http://www.citrenz.ac.nz/bacit/0203/images/2004CaukillFigure1.jpg
http://www.electronicproducts.com/images2/fajb_grid_cloud_01_nov2011.gif

MPI history

circa 1993: MPI – 1

- a standardization effort
  - IBM Watson Research Center, Intel's NX/2,
  - Express, nCUBE's Vertex, p4,
  - PARMACS, Zipcode , Chimp,
  - PVM, Chameleon , PICL
- a “discussion standard”
- main focus point-to-point communication
- did not include communicators
- was not thread safe

MPI history

circa 1998: MPI – 2.0

- API with bindings for:
  - Fortran 77
  - C

- many new/extended features
  - new datatype constructors
  - language interoperability
  - dynamic processes
  - "one-sided communication"
  - parallel I/O
  - etc.

MPI history

circa 2009: MPI – 2.2

- additional bindings for:
  - Fortran 90
  - C++
- future view was an expanded C++ functionality to deliver enhanced parallel classes

MPI history

circa 2012: MPI – 3

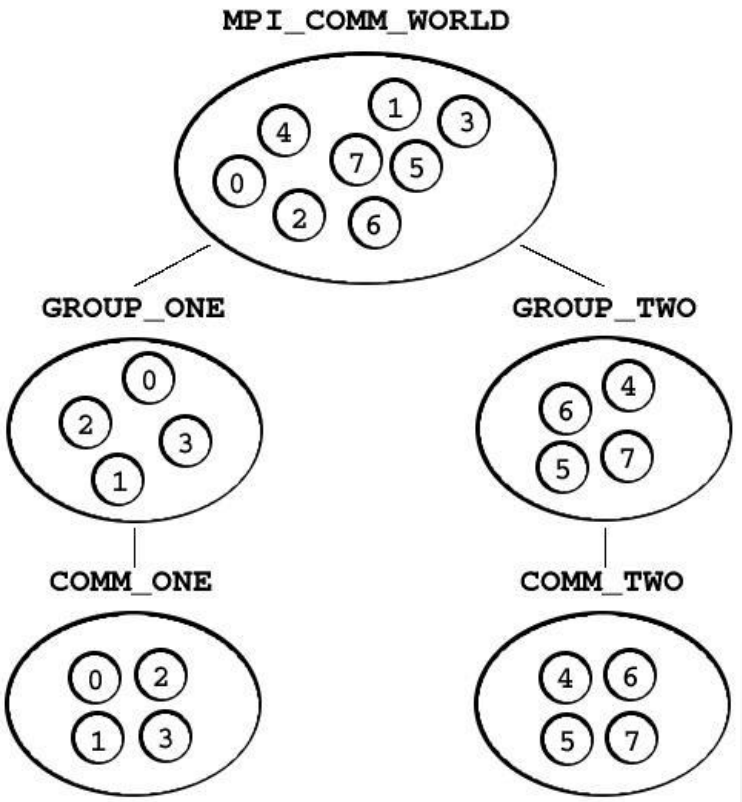
- many new features
  - nonblocking versions of collective operations
  - extensions to one-sided operations
  - new Fortran 2008 binding
  - removal of deprecated C++ bindings

MPI communication basics

- process groups
  - a system-level ordered set of processes
  - each process identified by its rank
  - processes may belong to multiple groups (rank is per-group)
  - processes may access any group lists
  - groups are “opaque”
- communicators
  - an ordered set of collaborating processes
  - each process identified by its rank
  - processes may belong to multiple communicators (rank is per-comm.)
  - may include same group in multiple communicators
  - processes may access communicators only if they are members
  - communicators are “opaque”
- both are dynamic entities

MPI communication basics

- context
  - a communicator/tag combination (or similar) defines a basis for MPI communications
- default communincator
  - MPI_COMM_WORLD



https://www.rc.usf.edu/tutorials/classes/tutorial/mpi/images/comm_group.jpg

MPI process identification

MPI_Comm_size()

- number of processes in the specified communicator

MPI_Comm_rank()

- reports the rank of the calling process in the specified communicator

hello MPI

```
#include <mpi.h>
#include <stdio.h>

int main (int argc, char *argv[ ])
{
int  rank, size;

    MPI_Init( &argc, &argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);
    MPI_Comm_size( MPI_COMM_WORLD, &size);
    printf("I am rank %d, of communicator size %d\n", rank, size);
    MPI_Finalize();
    return(0);
}
```

```
$ mpirun  -np  4  ./a.out
```

basic send / receive

MPI_Send(start*, count, mpi_datatype, dest, tag, comm);

start*	address of beginning of data buffer
count	length (in mpi_datatype units) of data buffer
mpi_datatype	system- or user-defined type
dest	rank of receiver within communicator
tag	message identifier (for filtering)
comm	MPI communicator

basic send / receive

MPI_Recv(start*, count, mpi_datatype, source, tag, comm, status);

start*	address of beginning of data buffer
count	length (in mpi_datatype units) of data buffer
mpi_datatype	system- or user-defined type
source	rank of sender within communicator
tag	message identifier (for filtering)
comm	MPI communicator
status	transaction information

MPI datatypes

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

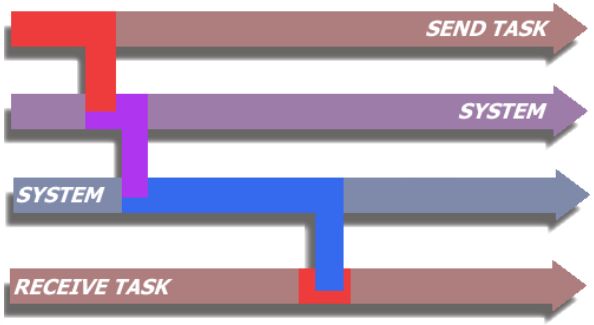
plus, user-defined extensions ⇐ evil

basic send / receive

blocking

MPI_Send(start*, count, mpi_datatype, dest, tag, comm);
MPI_Recv(start*, count, mpi_datatype, source, tag, comm, status*);

how will data be represented?	start, count, datatype
how will processes be identified?	dest, source, comm
how will receiver recognize message?	tag, comm
status upon completion?	send - buffer may be re-used recv – buffer contents valid



https://computing.llnl.gov/tutorials/mpi_performance/images/interrupt.gif

send / receive example

single value

```
#include <mpi.h>
#include <stdio.h>

int main (int argc, char *argv[ ])
{
int             rank, msg;
MPI_Status      status;

   MPI_Init( &argc, &argv);
   MPI_Comm_rank( MPI_COMM_WORLD, &rank);
   // process 0 sending to process 1
   if ( rank == 0 )
   {
      msg= 42;
      MPI_Send( &msg, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
   }
   else if ( rank == 1 )
   {
      MPI_Recv( &msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status)
      printf("Received %d\n", buf);
   }
   MPI_Finalize();
   return(0);
}
```

send / receive example

array

```
#include <mpi.h>
#include <stdio.h>

int main (int argc, char *argv[ ])
{
int             rank;
float           buffer[size];
MPI_Status      status;

   MPI_Init( &argc, &argv);
   MPI_Comm_rank( MPI_COMM_WORLD, &rank);
   // process 0 sending to process 1
   if ( rank == 0 )
   {
      //set buffer = something interesting
      MPI_Send( buffer, size, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
   }
   else if ( rank == 1 )
   {
      MPI_Recv( buffer, size, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status)
      printf("Received %d\n", buf);
   }
   MPI_Finalize();
   return(0);
}
```

receive status

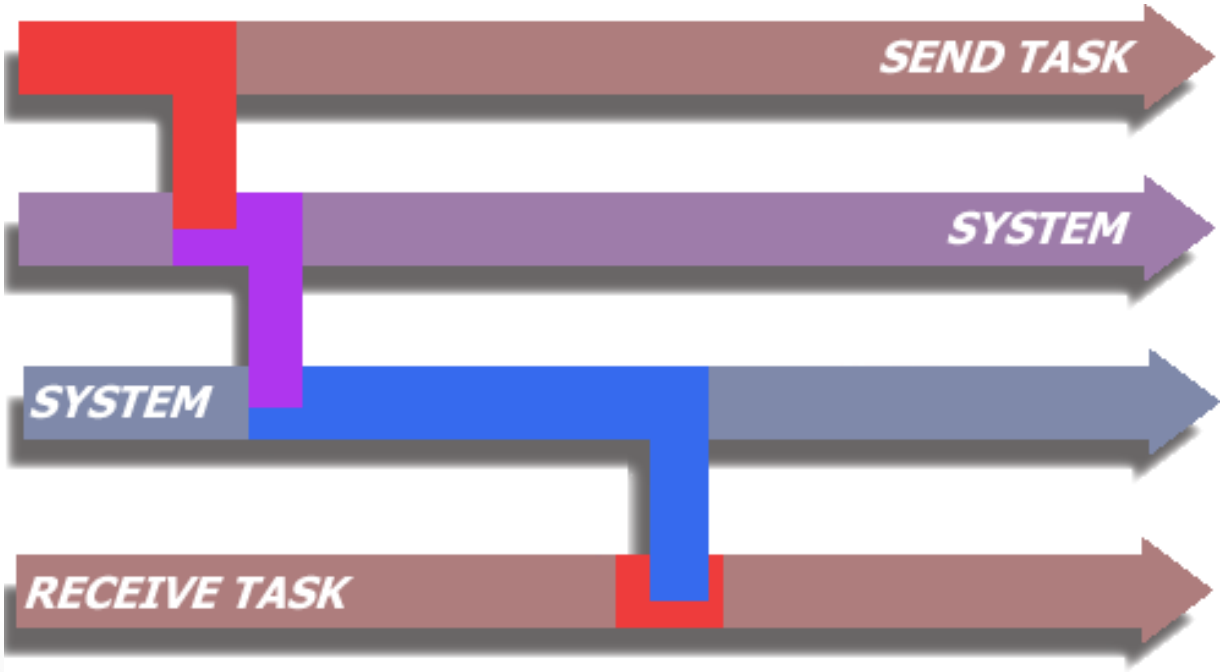
```
int             recvd_tag, recvd_from, recvd_count, error_code;
MPI_Status      status;

. . .
MPI_Recv( . . ., &status);
recvd_tag   = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
error_code  = status.MPI_ERROR;
MPI_Get_count( &status, datatype, &recvd_count);
```

basic MPI commands

```
MPI_Init( &argc, &argv);

MPI_Finalize();

MPI_Comm_size( communicator, &size);

MPI_Comm_rank( communicator, &rank);

MPI_Send( &buf, size, data_type, receiver, tag, communicator);

MPI_Recv( &buf, size, data_type, sender, tag, communicator, &status)
```

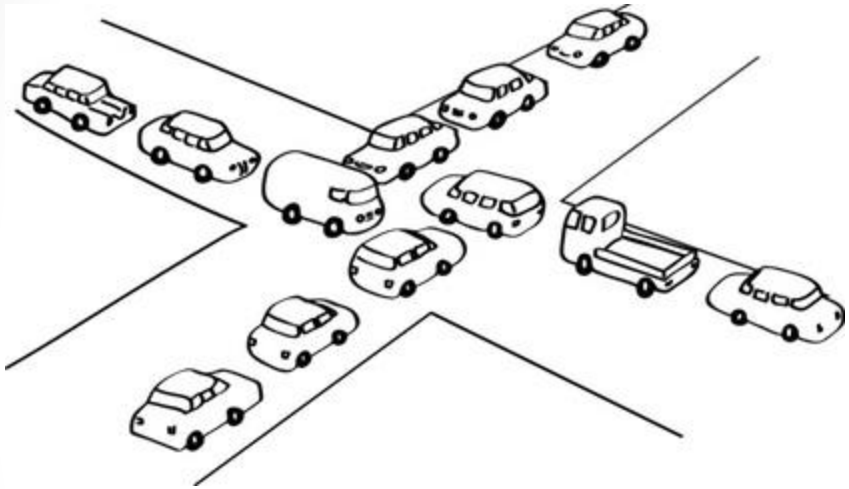
cse5441 - parallel computing

introduction to MPI



https://computing.llnl.gov/tutorials/mpi_performance/images/interrupt.gif

deadlock



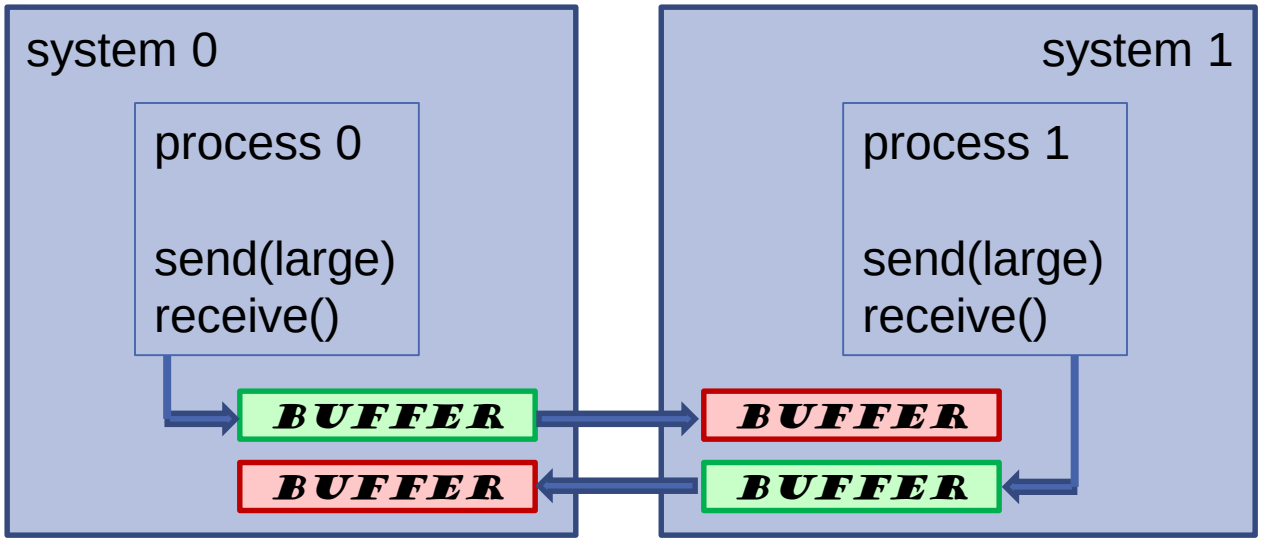
```
#include <mpi.h>
#include <stdio.h>

int main (int argc, char *argv[ ])
{
int                         rank, buf;
MPI_status    status;

   MPI_Init( &argc, &argv);
   MPI_Comm_rank( MPI_COMM_WORLD, &rank);
   // process 0 sending to process 1
   if ( rank == 0 )
   {
      buf = 123456;
      MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
   }
   else if ( rank == 1 )
   {
      MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status)
      printf("Received %d\n", buf);
   }
   MPI_Finalize();
   return(0);
}
```

very small buffer, this should be fine ...

- a situation where two or more threads are blocked forever, waiting for each other.
- a state in which progress is impossible

http://csunplugged.org/routing-and-deadlock/

buffering deadlock



unsafe

- process 0 sends a large message
  - if destination storage insufficient, send will wait until a receive() consumes data

- process 1 sends a large message
  - if destination storage insufficient, send will wait until a receive() consumes data

non-blocking I/O

- function calls return immediately, no waiting
- data is held in system buffers until transmitted / received

```
MPI_Request     request;
MPI_Status      status;
int             flag;

MPI_Isend(start, count, datatype, dest, tab comm, &request);

MPI_Irecv(start, count datatype, source, tag, comm, &request);

MPI_Wait(&request, &status);
      error_code  = status.MPI_ERROR;

MPI_Test(&request, &flag, &status);
      flag = true if operation has completed
      error_code  = status.MPI_ERROR
```

MPI error codes

provisional

MPI_SUCCESS	operation has completed successfully
MPI_ERR_REQUEST	requested operation was invalid
MPI_ERR_ARG	request had invalid arguments
MPI_ERR_PENDING	request has neither completed nor failed
MPI_ERROR	sytem-related failure

collective communications

- synchronization
- data movement
- collective computation

all collective operations are blocking

http://www.gsba.com/portals/0/Advocacy&Comm/GroupChat.jpg
http://bloguin.com/buckeyebattlecry/wp-content/uploads/sites/166/2013/06/Script-Ohio.gif

reduction

LARGE DATA SET

partial result

partial result

partial result

partial result

partial result

the final answer

https://www.coursehero.com/assets/img/star_ratings/ch_star_on.svg
https://cardiff.imgix.net/__data/assets/image/0010/498349/Distributed-and-Parallel-Systems.jpg?w=570&fit=clip

custom communicators

note: all MPI processes running the same program

```
#include <mpi.h>
#include <stdio.h>
#define NPROCS 8

main(int argc, char *argv[])
{
int             rank, new_rank, sendbuf, recvbuf, numtasks, ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
MPI_Group   orig_group, new_group;
MPI_Comm    new_comm;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

   sendbuf = rank;
/* Extract the original group handle */
   MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
/* Divide tasks into two distinct groups based upon rank */
   if (rank < NPROCS/2)
   {
      MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
   }
   else
   {
      MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
   }
/* Create new new communicator and then perform collective communications */
   MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
   MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);
   MPI_Group_rank (new_group, &new_rank);
   printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);
   MPI_Finalize();
}
```

```
int MPI_Group_incl(MPI_Group  group,   int n,
            const int ranks[],   MPI_Group *newgroup)

Input Parameters
   group       handle returned from MPI_Comm_group
   n           size of newgroup
   ranks       new processes rank map

Output Parameters
   newgroup  new group derived from above
```

```
output:
rank= 7 newrank= 3 recvbuf= 22
rank= 0 newrank= 0 recvbuf= 6
rank= 1 newrank= 1 recvbuf= 6
rank= 2 newrank= 2 recvbuf= 6
rank= 6 newrank= 2 recvbuf= 22
rank= 3 newrank= 3 recvbuf= 6
rank= 4 newrank= 0 recvbuf= 22
rank= 5 newrank= 1 recvbuf= 22
```

MPI synchronization

MPI_Barrier blocks until all processes in the communicator call it

```
int  MPI_Barrier( MPI_Comm   comm);
```

MPI data movement

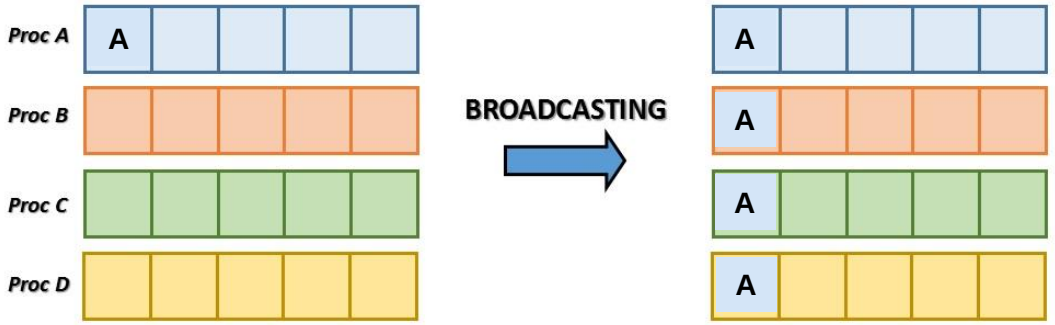
broadcast

MPI_Bcast(&buf, size, data_type, root, tag, communicator);

compare:

MPI_Send(&buf, size, data_type, receiver, tag, communicator);

```
MPI_Comm   comm;
int        A[100];
int root=0;

...

MPI_Bcast( A, 100, MPI_INT, root, tag, comm );
```

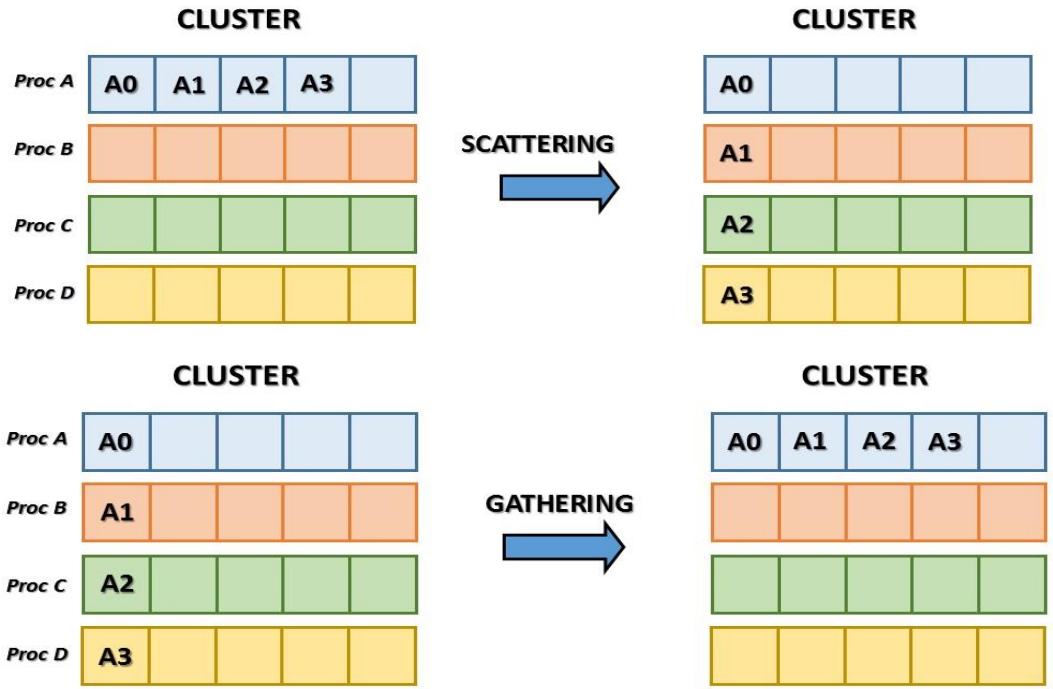


http://www.meccanismocomplesso.org/wp-content/uploads/2015/04/Meccanismo-Complesso-mpi4py-broadcasting.jpg

MPI data movement

multicast

```
MPI_Scatter( send_array, send_size, send_datatype,
             recv_array,  recv_size,  recv_datatype, root, comm );

MPI_Gather( send_array, send_size, send_datatype,
            recv_array,  recv_size,  recv_datatype, root, comm );
```



http://www.meccanismocomplesso.org/wp-content/uploads/2015/04/Meccanismo-Complesso-mpi4py-scattering.jpg
http://www.meccanismocomplesso.org/wp-content/uploads/2015/04/Meccanismo-Complesso-mpi4py-gathering.jpg

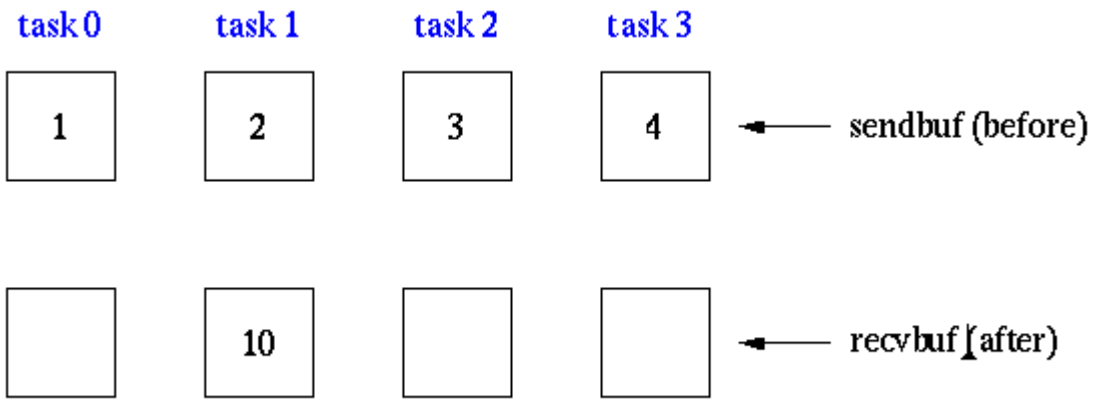
MPI reduction

general:

MPI_Reduce(sendbuf, recvbuf, size, datatype, reduce_op, root, comm);

example:

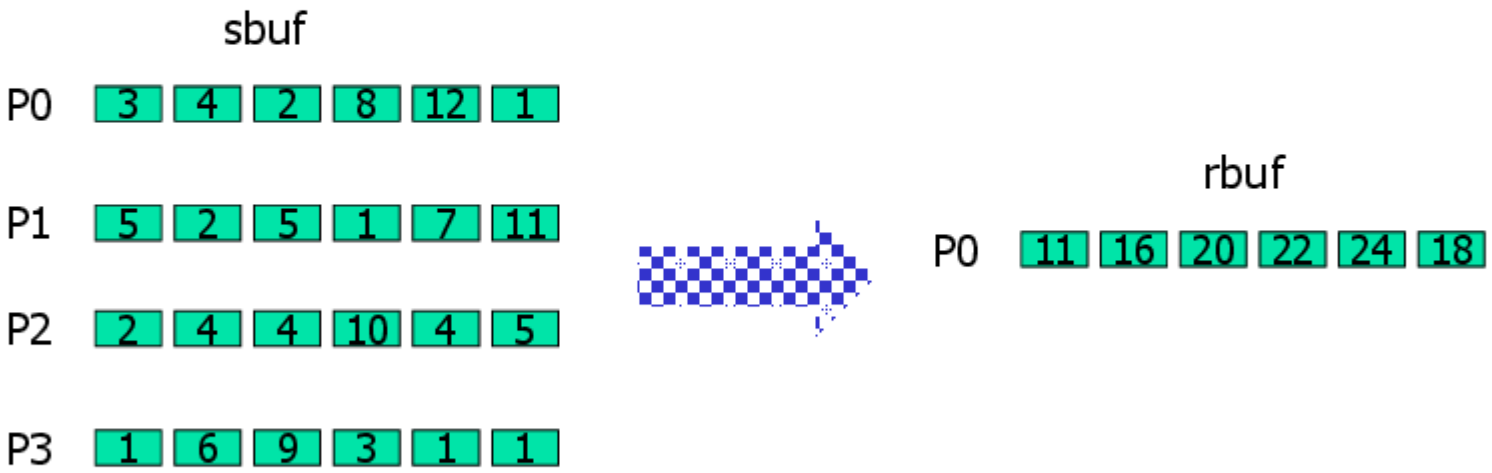
MPI_Reduce(&partial, &final, **1**, **MPI_INT**, **MPI_SUM**, **1**, MPI_COMM_WORLD);



http://phi.sinica.edu.tw/~tyuan/old.pages/pcfarm.19991228/aspac/instruct/workshop/html/mpi/MPI_Reduce.gif

MPI reduction

MPI_Reduce(sbuf, rbuf, **6**, **MPI_INT**, **MPI_SUM**, **0**, MPI_COMM_WORLD);



http://infocpp.iit.bme.hu/sites/default/files/6_ea_para/pg_0022.pdf.png

MPI reduction

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	bit-wise xor
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

MPI reduction

MPI_MAXLOC / MPI_MINLOC

slide 16:

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int
MPI_LONG_INT	long and int
MPI_2INT	pair of ints
MPI_SHORT_INT	short and int
MPI_LONG_DOUBLE_INT	long double and int

note: e.g. MPI_SHORT_INT is not a single short integer ...

MPI reduction

example sketch

```
float *rand_nums = NULL;
rand_nums = create_rand_nums(num_elements_per_proc);

. . .

// Sum the numbers locally
float local_sum = 0;
int i;
for (i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Print the random number sums and averages on each process
printf("Local sum for process %d = %f, avg = %f\n",
        world_rank,  local_sum,  local_sum  /  num_elements_per_proc);

// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum,  &global_sum, 1,  MPI_FLOAT,  MPI_SUM,  0, MY_COMMUNICATOR);
                                                    ^ result available only at root
// Print the result
if (world_rank == 0) {
  printf("Total sum = %f, avg = %f\n", global_sum,
           global_sum / (world_size * num_elements_per_proc));
}
```

MPI reduction

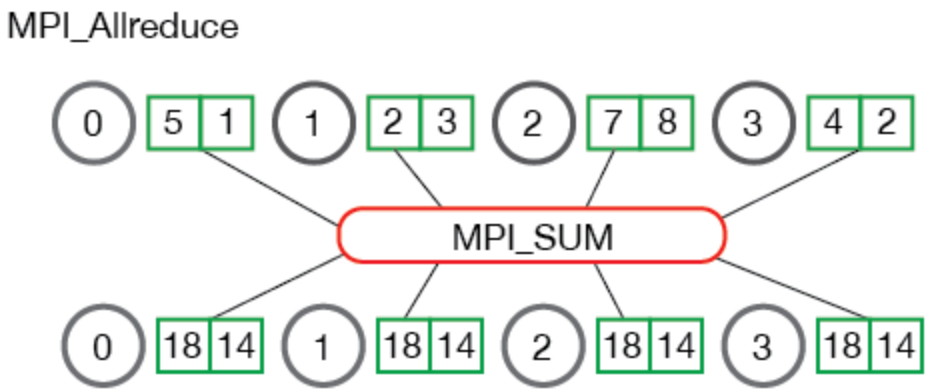
Allreduce

general:

MPI_Allreduce(send_array, recv_array, size, datatype, reduce_op, comm);

example:

MPI_Allreduce(my_work, result , **2**, **MPI_INT**, **MPI_SUM**, MPI_COMM_WORLD);



http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/mpi_allreduce_1.png

MPI limitations and concerns

summary

- highly implementation dependent
  - not all features may be available
- performance
  - latency – network
  - latency – computing nodes
  - portability / performance trade-offs
- system resources
  - buffering / deadlock
- correctness
  - debugging
  - non-deterministic execution

running MPI on OSC

log on to the Oakley cluster:

```
$ ssh osu0000@oakley.osc.edu
```

ensure MPI module is loaded:

```
-bash-4.1$ module list
Currently Loaded Modules:
  ...                ...              4) mvapich2/2.1
```

if not loaded:

```
-bash-4.1$ module load mvapich2
```

compile with MPI compiler:

```
-bash-4.1$ mpicc pingpong.c
```

run:

```
-bash-4.1$ qsub -I -l nodes=1:ppn=12 -l walltime=0:59:00
-bash-4.1$ mpiexec -np 2 a.out
```

MPI applications

examples

Basic Local Alignment Search Tool (BLAST)	DNA and protein sequence comparison algorithm
MrBayes	Bayesian estimation of phylogeny using Markov Chain Monte Carlo techniques
AVL EXCITE	dynamics, strengths, vibration and acoustics of combustion engines
Cart3D	conceptual and preliminary aerodynamic design (uses both MPI and OpenMP)
Gerris	computational fluid dynamics (CFD)
ParaView	interactive, scientific visualization
SHIPFLOW	ship hydrodynamics design
GNU Octave	numerical solution of linear and nonlinear problems
Chimera	interactive visualization and analysis of molecular structures

further study

introduction to MPI

general references

- "**Using MPI-2: Portable Parallel Programming with the Message-passing Interface**," Gropp et al., 2000

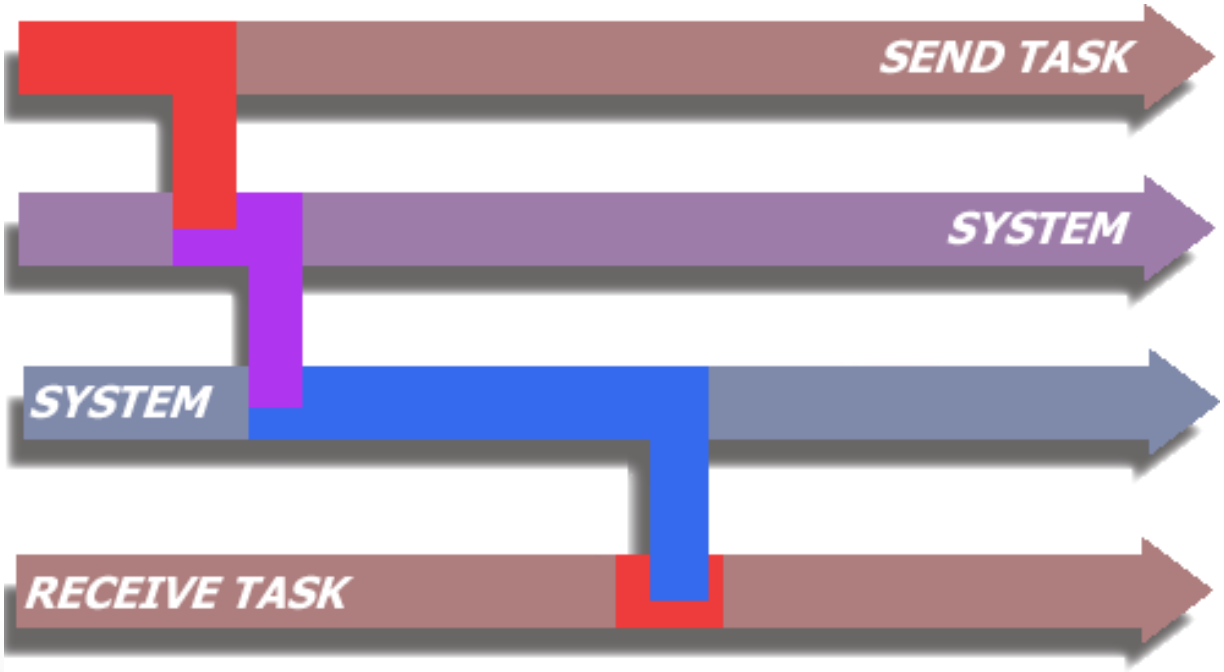
- “Programming Distributed Memory Machines with Message Passing,” Demmel, Carter, Yelick and Gropp, 2010

- http://www.mpi-forum.org

- http://www.msc.anl.gov/mpi

cse5441 - parallel computing

introduction to MPI



https://computing.llnl.gov/tutorials/mpi_performance/images/interrupt.gif