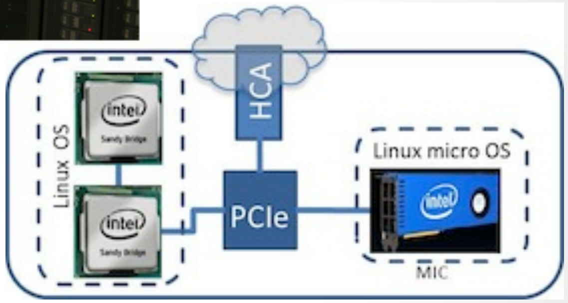
Introduction to distributed parallel programming

MPI programming

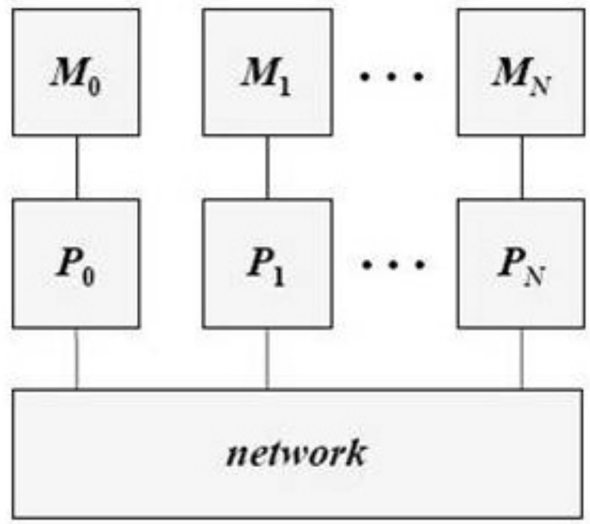
Parallelism at All Levels

- Parallelism across multiple nodes or processes - MPI
- Parallelism across threads - OpenMP
- Parallelism across instructions
- Parallelism on data – SIMD Single Instruction Multiple Data



www.scan.co.uk

Distributed Memory Programming



distributed-memory multicomputer

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

Programming a distributed-memory computer

- MPI (Message Passing Interface)
- Message passing standard, universally adopted library of communication routines callable from C, C++, Fortran, (Python)
- 125+ functions—I will introduce a small subset of functions

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

MPI-1

- MPI was developed in two major stages, MPI-1 and MPI-2
- 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++

MPI-3

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

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

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)

Programming in MPI

```
use mpi
integer ierr

call MPI_init(ierr)
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
.
.
.
call MPI_Finalize(ierr)
```

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

Determine the processor running on

- Ierr = MPI_Get_processor_name(proc_name, &length);

MPI_COMM_WORLD

- Is a *communicator*
- Predefined in MPI
- Consists of all processes running at start of program execution
- Process rank and number of processors determined from MPI_COMM_WORLD
- Possible to create new communicators

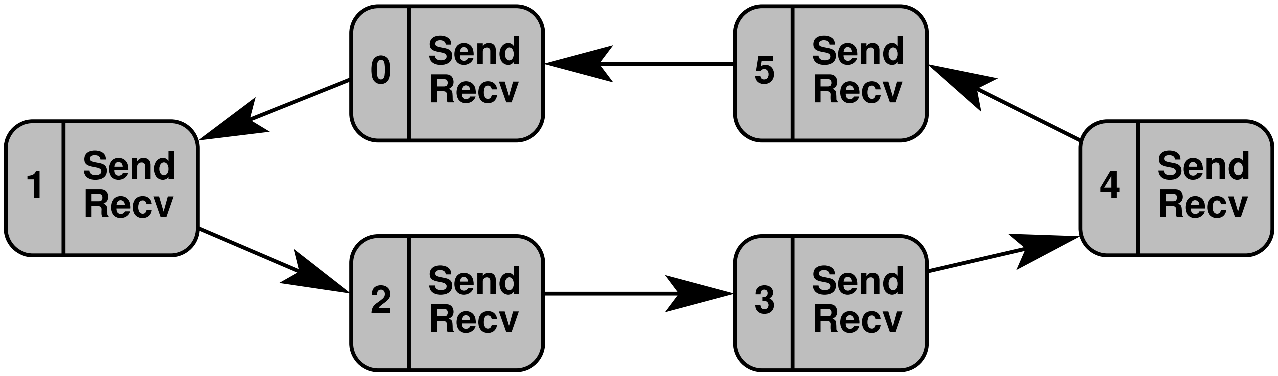
MPI Hello world

- Write a program similar to the OMP hello world
- Output should be:
  - Hello World from process = %d on node %s
  - Number of mpi processes = %d

Compiling and Running an MPI program

- See mpi_intro.ipynb

Sending data in a ring



Example program

- Summing numbers on ring of processors
  initially, N single numbers per processor

- If I am processor myid,
- Store my number in x(1:n)
- For number of steps = numprocs – 1
  - Send my n numbers to process myid + 1 (mod numprocs)
  - Receive N x from process myid – 1 (also mod numprocs)
  - Once all values have been received, sum x(1)+…+x(numprocs)

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.

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?

- 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.

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	

)

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)

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

Our program

```
integer ierr  (and other dimension statements)
include “mpi.h”
call MPI_init(ierr), MPI_COMM_RANK, MPI_COMM_SIZE
< Processor myid has x(1), x(2) to begin>
count = 1
do j = 1, numprocs-1
   call MPI_send(x(count), 2, …,mod(myid+1,numprocs),…)
   count = count + 2
   call MPI_recv(x(count), 2, …, mod(myid-1,numprocs),…)
enddo
print*,’here is my answer’,sum(x)
Call MPI_finalize(ierr)
```

Point-to-Point Communication Modes

Standard Mode:

blocking:

MPI_SEND (buf, count, datatype, dest, tag, comm, ierr)

MPI_RECV (buf, count, datatype, source, tag, comm, status, ierr)

- Generally **ONLY** use if you cannot call earlier **AND** there is no other work that can be done!
- Standard **ONLY** states that buffers can be used once calls return. It is implementation dependent on when blocking calls return.
- Blocking sends **MAY** block until a matching receive is posted. This is not required behavior, but the standard does not prohibit this behavior either. Further, a blocking send may have to wait for system resources such as system managed message buffers.

Be VERY careful of deadlock when using blocking calls!

Requirements for Point to Point Communications

- For a communication to succeed:
  - Sender must specify a valid destination rank.
  - Receiver must specify a valid source rank.
  - The communicator must be the same.
  - Tags must match.
  - Message data types must match.
  - Receiver's buffer must be large enough.

Wildcarding

- Receiver can wildcard.
- To receive from any source
  - source = MPI_ANY_SOURCE
- To receive from any tag
  - tag = MPI_ANY_TAG
- Actual source and tag are returned in the receiver's status parameter.

Communication Envelope

- Envelope information is returned from MPI_RECV in status.
- C:
  - status.MPI_SOURCE
  - status.MPI_TAG
  - count via MPI_Get_count()
- Fortran:
  - status(MPI_SOURCE)
  - status(MPI_TAG)
  - count via MPI_GET_COUNT()

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

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 <u>the</u> 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

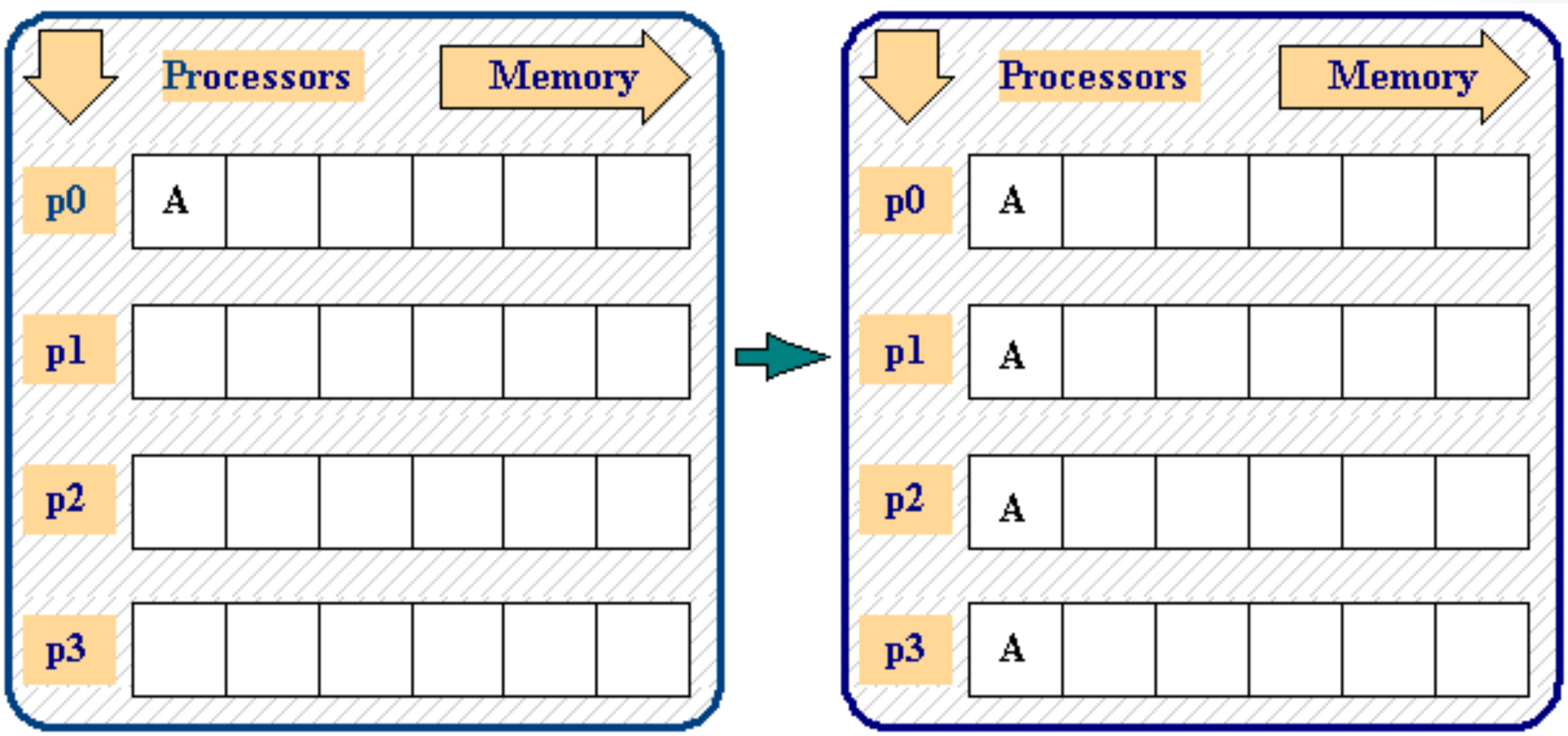
MPI_TESTALL (count, requests, flag, statuses, ierr)

- *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

- Synchronization
  - 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()

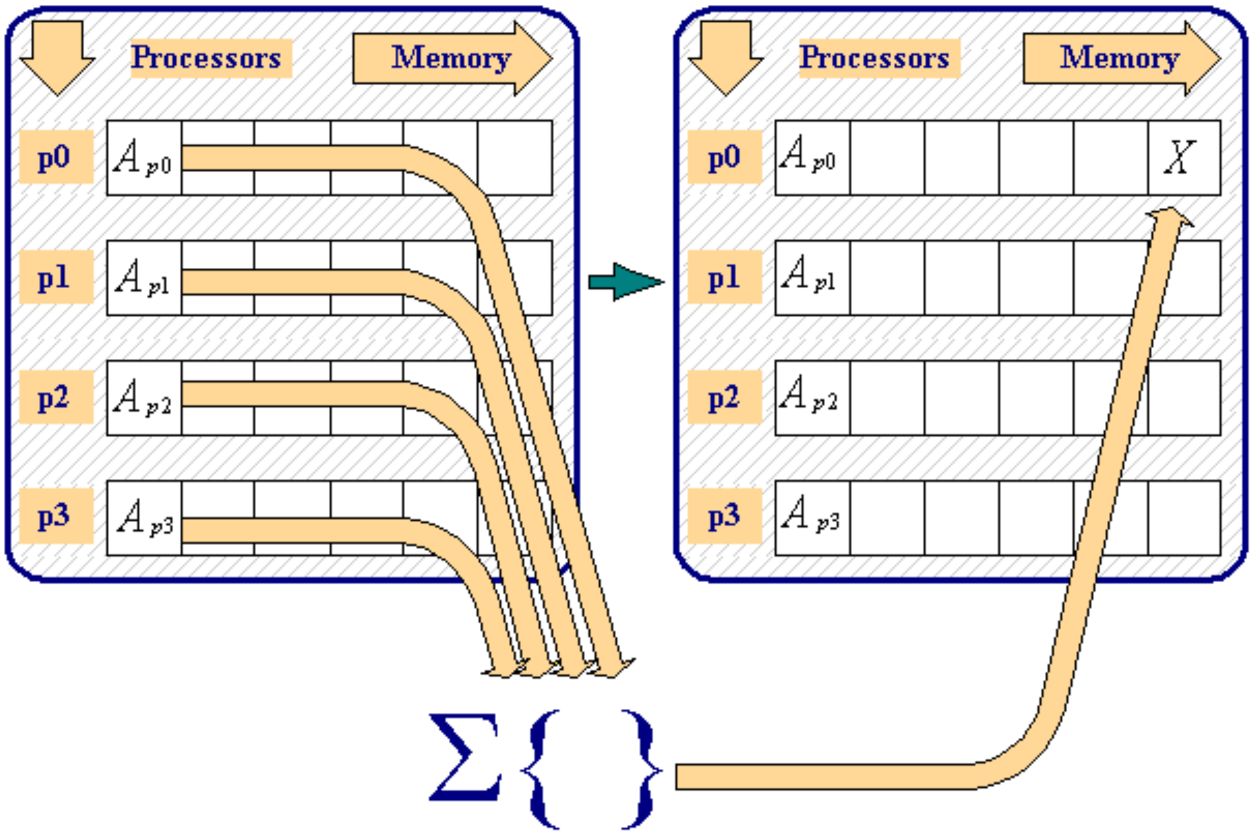
Broadcast




```
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;

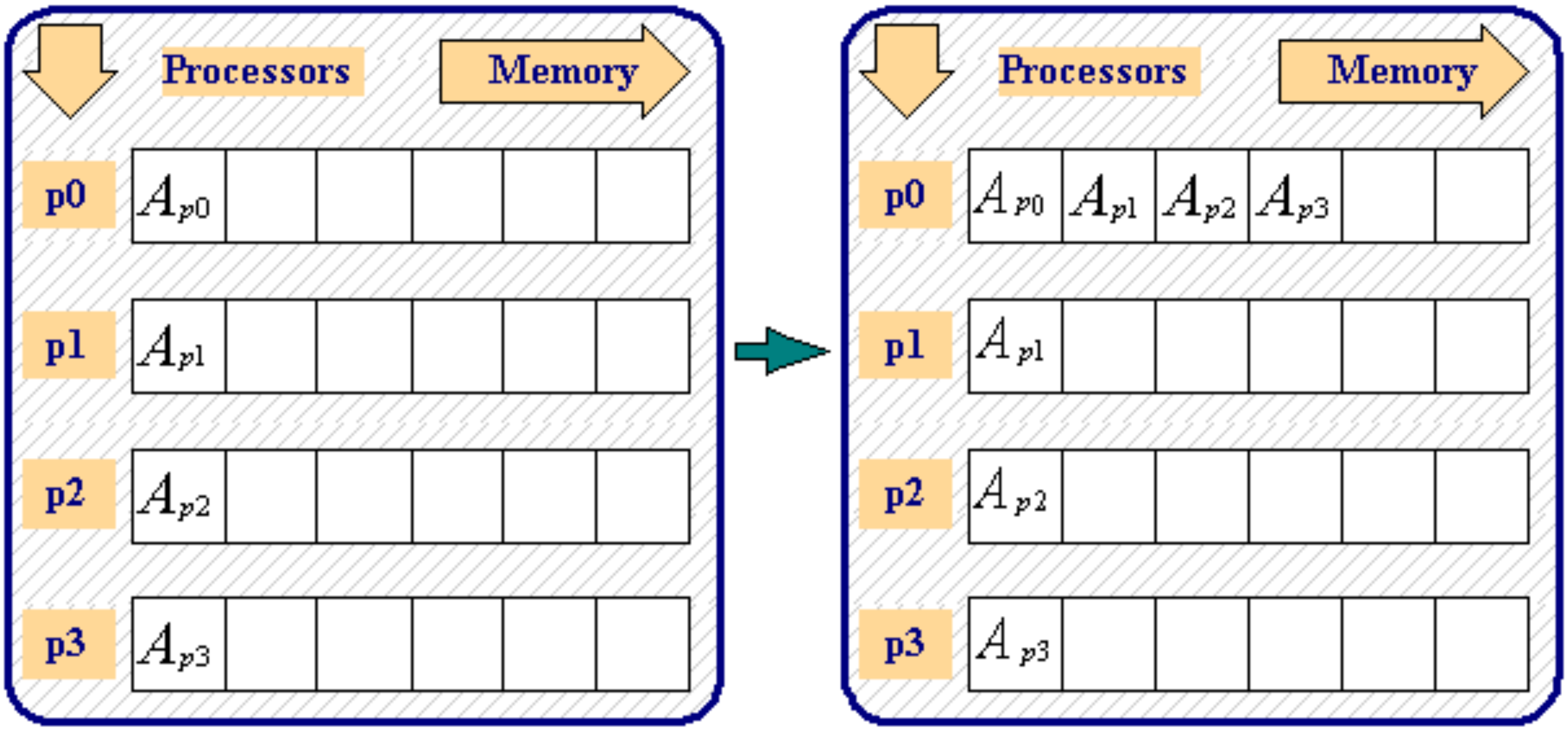
MPI_Reduce (&a, &x, count, MPI_REAL, MPI_SUM, rank, MPI_COMM_WORLD);

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

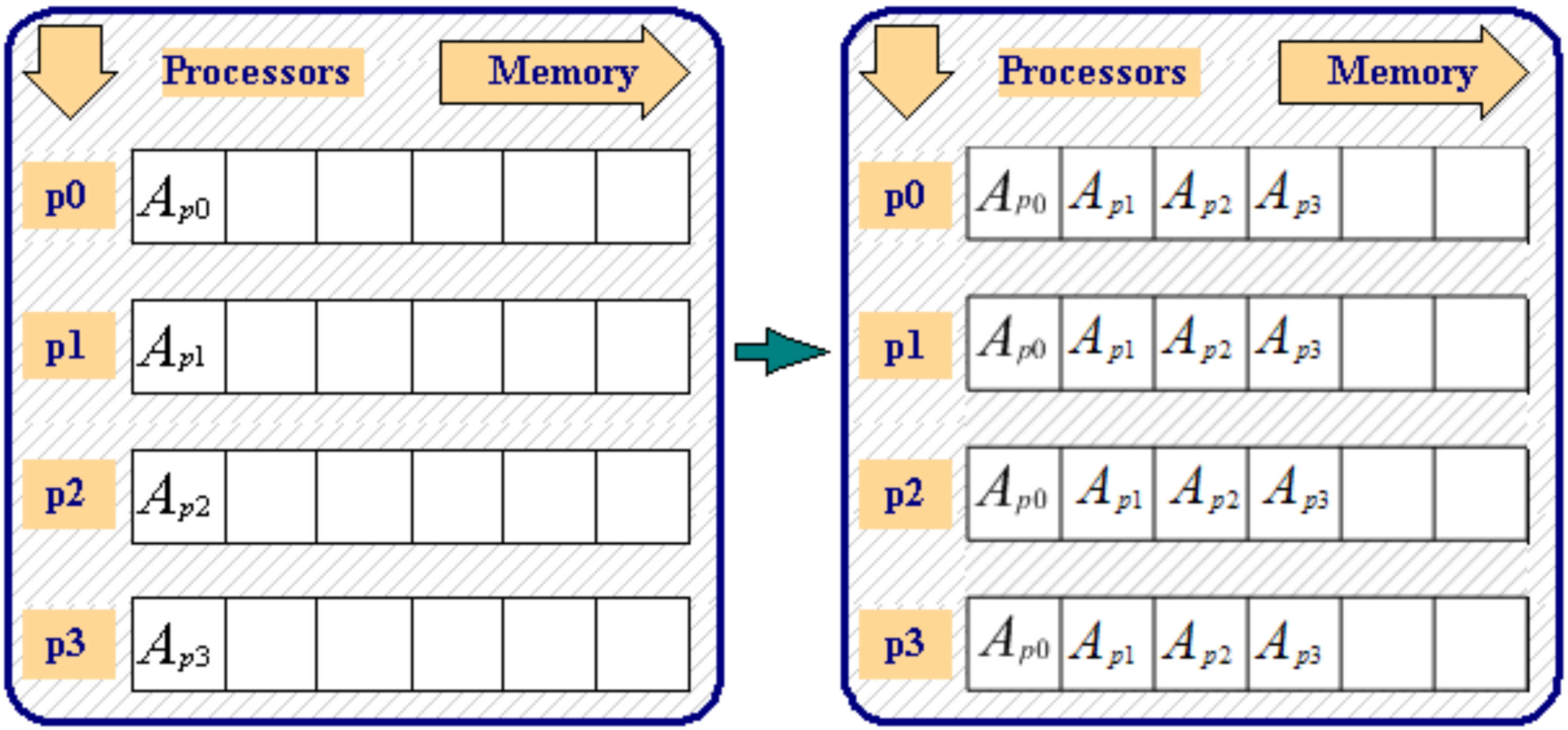
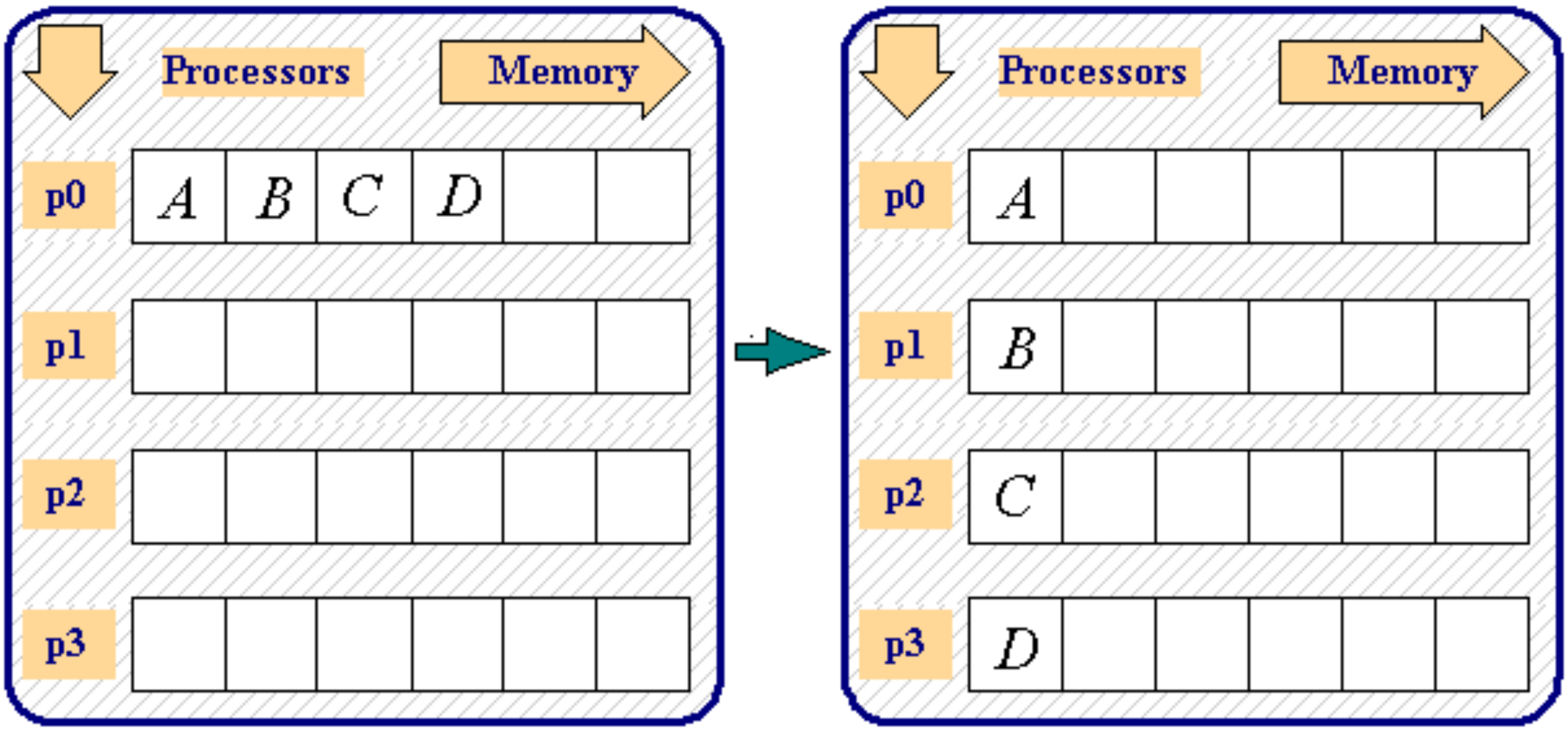


Figure from MPI-tutor: http://www.citutor.org/index.php

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

Summary

- MPI is the standard for distributed parallel programming
- Best approach is probably hybrid
  - MPI for inter node communication
  - OpenMP for and other directives for parallelism within a node
- If possible use exisiting libraries
  - Global Arrays http://hpc.pnl.gov/globalarrays/
  - PETSc http://www.mcs.anl.gov/petsc/