MPI

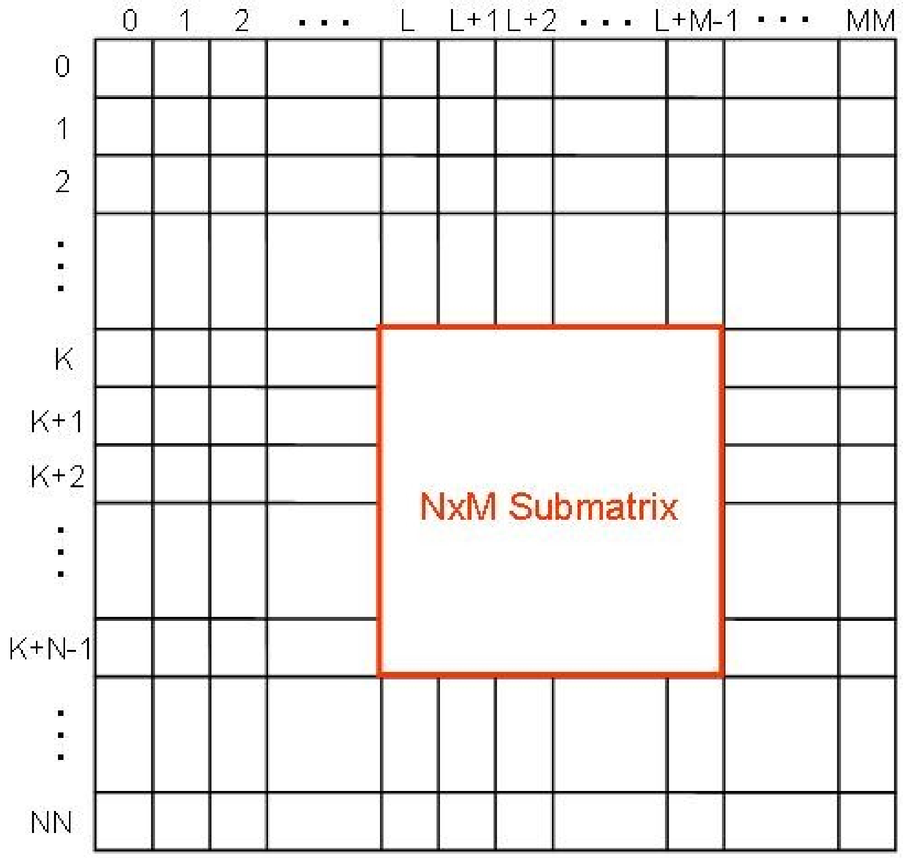
- Communicating non-contiguous data or mixed datatypes
- Collective communications
- Virtual topologies

Communicating Non-contiguous Data or Mixed Datatypes

- The following strategies and features in MPI enable you to transmit collections of data of mixed types or data that are scattered within an array:
  - Sending Multiple Messages
  - Buffering
  - Pack and Unpack
  - Derived Datatypes

Sending Multiple Messages

- Identify the largest pieces of your data that individually meet the requirements of being of homogeneous intrinsic type and contiguous, and send each of those pieces as a separate message.
- You have a large matrix stored in a two-dimensional array, and you want to send a rectangular submatrix to another processor.
  - In C, successive elements of a row are contiguous, with the last element in one row followed by the first element of the next column.
  - The rows of your submatrix will not be contiguous.
  - The elements within a row will be contiguous, so you can send one message for each row of the submatrix.
- If the receiving processor does not know the values of N, M, K, or L , they can be sent in a separate message.



```
for (i=0; i<n; ++i) {
MPI_Send(
 &a[k+i][l], m, MPI_DOUBLE,
 dest, tag, MPI_COMM_WORLD);
}
```

Sending Multiple Messages

- The principal advantage of this approach is that you do not need to learn anything new to apply it.
- The principal disadvantage is its overhead.
  - A fixed overhead is associated with the sending and receiving of a message, however long or short it is.
  - If you replace one long message with multiple short messages, you slow down your program by greatly increasing your overhead. Also, this method can be error-prone.
- If you are working in a portion of your program that will be executed infrequently and the number of additional messages is small, the total amount of added overhead may be small enough to ignore.
  - For most programs there is no advantage in limiting yourself to this strategy.
  - You will have to learn other techniques for the heavily executed portions.

Buffering

- When the data you need to communicate is not contiguous, one approach might be to copy them into a contiguous buffer.

```
p = &buffer;
for (i=k; i<k+n; ++i) {
   for(j=l; j<l+m; ++j) {
      *(p++) = a[i][j];
   }
}
MPI_Send(p, n*m, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
```

- Copying to a buffer eliminates the excessive number of messages generated by the multiple messages approach at the cost of extra memory for the buffer and extra CPU time to perform the copy into the buffer.
- The obvious limitation of this approach is that it still handles only one type of data at a time.

A Tempting Wrong Way to Extend Buffering

- It is often possible to encode the values of one type as values of another type. We could convert the values of N, M, K, and L to floating point in order to include them in our buffer.
  - Such conversions generally take more CPU time than a simple copy and, in most cases, the result will occupy more memory.
- At this point, you may be tempted to use a programming trick (e.g.casting the type of a pointer) to put the bit patterns for values of one type into a buffer declared to be of some other type.
  - This approach can be very dangerous.
  - If you write a test program to try it out, it is highly likely that it will appear to work for you. However, if you use this approach extensively, especially in programs that are run in multiple environments, it is almost inevitable that it will eventually fail.
  - If you are lucky, it will fail spectacularly.
  - If you are not lucky, you may just get incorrect results without realizing what has gone wrong.

A Tempting Wrong Way to Extend Buffering

- The fundamental problem with this approach is that MPI transmits values, not just bit patterns.
  - As long as you are using a set of processors that all represent values the same way, MPI optimizes its communications by simply transmitting the bit patterns in your message and tricks like this will work.
  - If there is any chance of communicating with a processor that uses a different representation for some or all of the values, MPI:
    - Translates the values in your message into a standard intermediate representation
    - Transmits the bits of the intermediate representation
    - Translates the intermediate representation back into values on the other processor.
  - This extra translation ensures that the same value is received as was sent.
  - That value on the receiving processor may no longer have the same bit pattern as the value in the original type.

Buffering the Right Way - Pack and Unpack

- The right way to fill a buffer is to use the MPI_PACK routine.
  - You call MPI_PACK with arguments that describe the buffer you are filling and most of the arguments you would have provided when using MPI_SEND.
  - MPI_PACK copies your data into the buffer and, if necessary, translates it into a standard intermediate representation.
  - After all the data you want to send have been placed in the buffer by MPI_PACK, you can send the buffer (giving its type as MPI_PACKED) and no further translations will be performed.

```
count = 0;
for(i=0; i<n; i++){
  MPI_Pack(&a[k+i][l], m, MPI_DOUBLE, buffer,
           bufsize, count, MPI_COMM_WORLD);
}
MPI_Send(buffer, count, MPI_PACKED, dest, tag,
MPI_COMM_WORLD);
```

Buffering the Right Way - Pack and Unpack

- The successive calls to MPI_PACK update the count variable to reflect the data that have been added to the buffer and the final value is used in the call to MPI_SEND to specify the amount of data to send.
- On the receiving side, you can similarly specify type MPI_PACKED to receive a buffer without translation and then use MPI_UNPACK to translate and copy the data from the buffer to where you really want it.
- Because of translation, data may occupy a different amount of space in the buffer than it does natively.
  - You can make your buffer large enough by using the routine MPI_PACK_SIZE to calculate how much buffer space is required for the different types of data you plan to place in your buffer.

Buffering the Right Way - Pack and Unpack

- Nothing in the content of a message indicates it was or was not built with MPI_PACK.
  - If, as in our example, all the data packed into the buffer is of a single type, the message could be received in a buffer of that type rather than receiving it as MPI_PACKED and using MPI_UNPACK to decode it.
  - Conversely, a message that was sent as an ordinary intrinsic type could be received as MPI_PACKED and distributed using calls to MPI_UNPACK.

Buffering the Right Way - Pack and Unpack

- Use of MPI_PACK and MPI_UNPACK provides great flexibility.
  - In addition to allowing messages that include arbitrary mixtures of datatypes its incremental construction and interpretation of messages allows the values of data early in a message to affect the type, size, or destination of data appearing later in the same message.
  - The principal costs of this flexibility are the memory used for the buffers and CPU time used in copying data to and from those buffers.
  - If constructing a message requires a large number of calls to MPI_PACK (or interpreting a message requires a large number of calls to MPI_UNPACK), the added procedure call overhead may also be significant.

Derived Datatypes

- MPI has a more efficient approach to handling mixed datatypes and noncontiguous data called derived datatypes.
- Derived datatypes are datatypes that are built from the basic MPI datatypes.
  - You can view them as a way to get MPI to do packing and unpacking "on-the-fly" as part of the send and receive operations.
  - The packing and unpacking can then be done directly to, and from, its internal communications buffers, generally making your program more efficient by eliminating the need for:
    - The explicit intermediate buffer used when you do the packing and unpacking
    - The copying between the intermediate buffer and the communications buffer

Derived Datatypes

- There are various MPI routines that can be used to create derived datatypes.
- We will use MPI_TYPE_VECTOR, which creates a datatype consisting of uniformly spaced blocks of data, each block having the same size.
  - We need to send segments of n rows of the matrix, each containing m values.
  - In defining the derived datatype, we need to specify the stride, which is the distance between the starting locations of adjacent blocks of data.
  - The rows of the full matrix each have mm values, so mm will be the stride between the beginning of one row segment and an adjacent row segment.

Derived Datatypes

```
MPI_Type_vector(n,m, mm, MPI_DOUBLE,
&my_mpi_type);
MPI_Type_commit(&my_mpi_type);
MPI_Send(&a[k][l], 1, my_mpi_type, dest, tag,
MPI_COMM_WORLD);
```

- The call to MPI_TYPE_VECTOR indicates that we are creating a new derived datatype called my_mpi_type. It contains n blocks of data, each containing m MPI_DOUBLE values, with a stride of mm between each block.
- This datatype defines the whole submatrix.
- The call to MPI_TYPE_COMMIT makes the derived datatype available for use in communication routines.
  - Sometimes derived datatypes are used to construct other derived datatypes, so the intermediate types do not have to be committed.

Derived Datatypes

- Finally, the MPI_SEND routine sends the data.
  - The send is quite simple, since we are sending a single instance of a my_mpi_type, starting at location (k,l), containing the whole submatrix.
  - The data are received in an analogous way, receiving a single instance of the derived datatype my_mpi_type, resulting in the values arranged properly in the submatrix.
- The derived datatype only needs to be created one time, and can be used any number of times in communication routines.
- If our submatrix must be transferred additional times throughout the code, only a single call to MPI_SEND will be required each time.

Derived Datatypes

- As a direct replacement for pack and unpack operations, MPI-derived datatype operations are usually more efficient.
- If you are packing data that do not remain in existence until the time the packed buffer is sent (e.g., if successive values to be packed are computed into the same variables), derived datatype operations lose their efficiency advantage because you must institute some other form of buffering to retain those values until they can be sent.

Derived Datatypes

- Similarly, if the locations to which data are to be unpacked are not disjointed (e.g., if successive values from a message are processed in the same variables), derived datatype operations may lose their efficiency advantage.
  - This loss is because you may need to buffer those values somewhere else until they can be processed.
  - Buffering is also necessary if the locations to receive the data cannot be determined in advance.
- Derived datatype operations cannot be used to replace unpacking in those cases where values in the early part of a message determine the structure of the later part of the message.
  - In such cases, explicitly typed buffering will not work, and you need the flexibility of piecemeal unpacking of an MPI_PACKED buffer.

User-Defined Datatypes

- Creating a derived datatype to use it just once before freeing it can be a bit wasteful.
- It is far more effective to create derived datatypes that describe recurring patterns of access and then reuse them for each occurrence of that pattern of access.
- The classic example of this is the use of a derived datatype to describe a user-defined datatype in the language you are using.
- This technique is called mapping.

User-Defined Datatypes

```
struct SparseElt {      /* representation of a sparse matrix element */
      int    location[2]; /* where the element belongs in the overall matrix */
      double value;       /* the value of the element */
   };

   /* a representative variable of this type */
   struct SparseElt anElement;

   /* length, displacement, and type arrays used to describe an MPI derived type */
   /* their size reflects the number of components in SparseElt */
   int          lena[2];
   MPI_Aint     loca[2];
   MPI_Datatype typa[2];

   MPI_Aint     baseaddress;
   /* a variable to hold the MPI type indicator for SparseElt */
   MPI_Datatype MPI_SparseElt;
   /* set up the MPI description of SparseElt */

   MPI_Address(&anElement, &baseaddress);
   lena[0] = 2;    /* anElement.location has length of 2 ints*/
   MPI_Address(&anElement.location,&loca[0]);
   loca[0] -= baseaddress;  /* byte address relative to start of structure */
   typa[0] = MPI_INT;
   lena[1] = 1;     /* anElement.value has length of 1 double*/
   MPI_Address(&anElement.value   ,&loca[1]);
   loca[1] -= baseaddress;
   typa[1] = MPI_DOUBLE;
   MPI_Type_struct(2, lena, loca, typa, &MPI_SparseElt);
   MPI_Type_commit(&MPI_SparseElt);
```

User-Defined Datatypes

- In this example, we construct three arrays containing the length, location, and types of the components to be transferred when this type is used.
- We then subtract the address of the whole variable from the addresses of its components; so the locations are specified relative to the variable rather than relative to MPI_BOTTOM.
  - This construct allows us to use the type indicator MPI_SparseElt to describe a variable of type SparseElt anywhere in the program.
- Once a type has been created and committed, it may be used anywhere an intrinsic type indicator can be used such as send and receive operations, including collective communications and reduction MPI functions.
- It can also be used for purposes other than communication, such as defining another MPI-derived datatypes that might have a SparseElt component or in performing pack or unpack operations on variables of type SparseElt.

Constructing Derived Datatypes

- We have already shown the use of MPI_TYPE_VECTOR for constructing derived datatypes.
- The most general way to construct a derived datatype is through the use of MPI_TYPE_STRUCT, which allows the length, location, and type of each component to be specified independently.
- There are several other procedures available to describe common patterns of access, primarily within arrays. These are:
  - MPI_TYPE_CONTIGUOUS
  - MPI_TYPE_HVECTOR
  - MPI_TYPE_INDEXED
  - MPI_TYPE_HINDEXED

Constructing Derived Datatypes

- MPI_TYPE_CONTIGUOUS is the simplest, describing a contiguous sequence of values in memory. For example, using it as follows in C:

  ```
  MPI_Type_contiguous(2,MPI_DOUBLE,&MPI_2D_POINT);
  MPI_Type_contiguous(3,MPI_DOUBLE,&MPI_3D_POINT);
  ```

  creates new type indicators MPI_2D_POINT and MPI_3D_POINT.
- These type indicators allow you to treat consecutive pairs of doubles as point coordinates in a 2-dimensional space and sequences of three doubles as point coordinates in a 3-dimensional space.

Constructing Derived Datatypes

- MPI_TYPE_HVECTOR is similar to MPI_TYPE_VECTOR except that the distance between successive blocks is specified in bytes rather than elements.
  - The most common reason for specifying this distance in bytes would be that elements of some other type are interspersed in memory with the elements of interest.
    - For example, if you had an array of type SparseElt, you could use MPI_TYPE_HVECTOR to describe the array of value components.
- MPI_TYPE_INDEXED describes sequences that may vary both in length and in spacing.
  - Because the location of these sequences is measured in elements rather than bytes, it is most appropriate for identifying arbitrary parts of a single array.
- MPI_TYPE_HINDEXED is similar to MPI_TYPE_INDEXED except that the locations are specified in bytes rather than elements.
  - It allows the identification of arbitrary parts of arbitrary arrays, subject only to the requirement that they all have the same type.

Message Matching and Mismatching

- Just as there is nothing in the content of a message to indicate whether it was built with MPI_PACK, there is nothing to indicate whether or what kind of derived datatypes may have been used in its construction.
  - All that matters is that the sender and receiver agree on the nature of the sequence of primitive values the message represents.
  - Thus, a message constructed and sent using MPI_PACK could be received using derived datatypes, or a message sent using derived datatypes could be received as MPI_PACKED and distributed using MPI_UNPACK.
  - Similarly, a message could be sent using one derived datatype and received using a different derived datatype.

Message Matching and Mismatching

- This leads to one significant difference between derived datatypes and primitive datatypes.
  - If you send data using the same derived datatype with which it will be received, the message will contain an integral number of that datatype, and MPI_GET_COUNT will work for that datatype in the same way it does for primitive datatypes.
  - If the datatypes are different, you could end up with a partial value at the end.

Message Matching and Mismatching

- For example, if the sending processor sends an array of four MPI_3D_POINTs (or a total of twelve MPI_DOUBLEs) and the receiving processor receives the message as an array of MPI_2D_POINTs, MPI_GET_COUNT will report that six MPI_2D_POINTs were received.
- If instead five MPI_3D_POINTs were sent (i.e., fifteen MPI_DOUBLEs), seven and a half MPI_2D_POINTs will be changed on the receiving side, but MPI_GET_COUNT cannot return "7.5."
  - Rather than return a potentially misleading value such as "7" or "8", MPI_GET_COUNT returns the flag value MPI_UNDEFINED.
  - If you need more information about the size of the transfer, you can still use MPI_GET_ELEMENTS to learn that nine primitive values were transferred.

Collective Communications

- Collective communication involves the sending and receiving of data among processes.
  - In general, all movement of data among processes can be accomplished using MPI send and receive routines.
  - However, some sequences of communication operations are so common that MPI provides a set of collective communication routines to handle them.
  - These routines are built using point-to-point communication routines. Even though you could build your own collective communication routines, these "blackbox" routines hide a lot of the messy details and often implement the most efficient algorithm known for that operation.

Collective Communications

- Collective communication routines transmit data among all processes in a group.
  - Collective communication calls do not use the tag mechanism of send/receive for associating calls.
  - They are associated by order of program execution and because of this you must ensure that all processors execute a given collective communication call.
- The collective communication routines allow data movement among all processors or just a specified set of processors.
  - You may define your own communicator that provides for collective communication between a subset of processors.

Barrier Synchronization

- There are occasions when some processors cannot proceed until other processors have completed their current instructions.
  - This commonly occurs when the root process reads data and then transmits these data to other processors.
  - The other processors must wait until the I/O is completed and the data are moved.
- The MPI_BARRIER routine blocks the calling process until all group processes have called the function.
  - When MPI_BARRIER returns, all processes are synchronized at the barrier.
  - MPI_BARRIER synchronizes the processes but does not pass data.
  - It is nevertheless categorized as one of the collective communications routines.

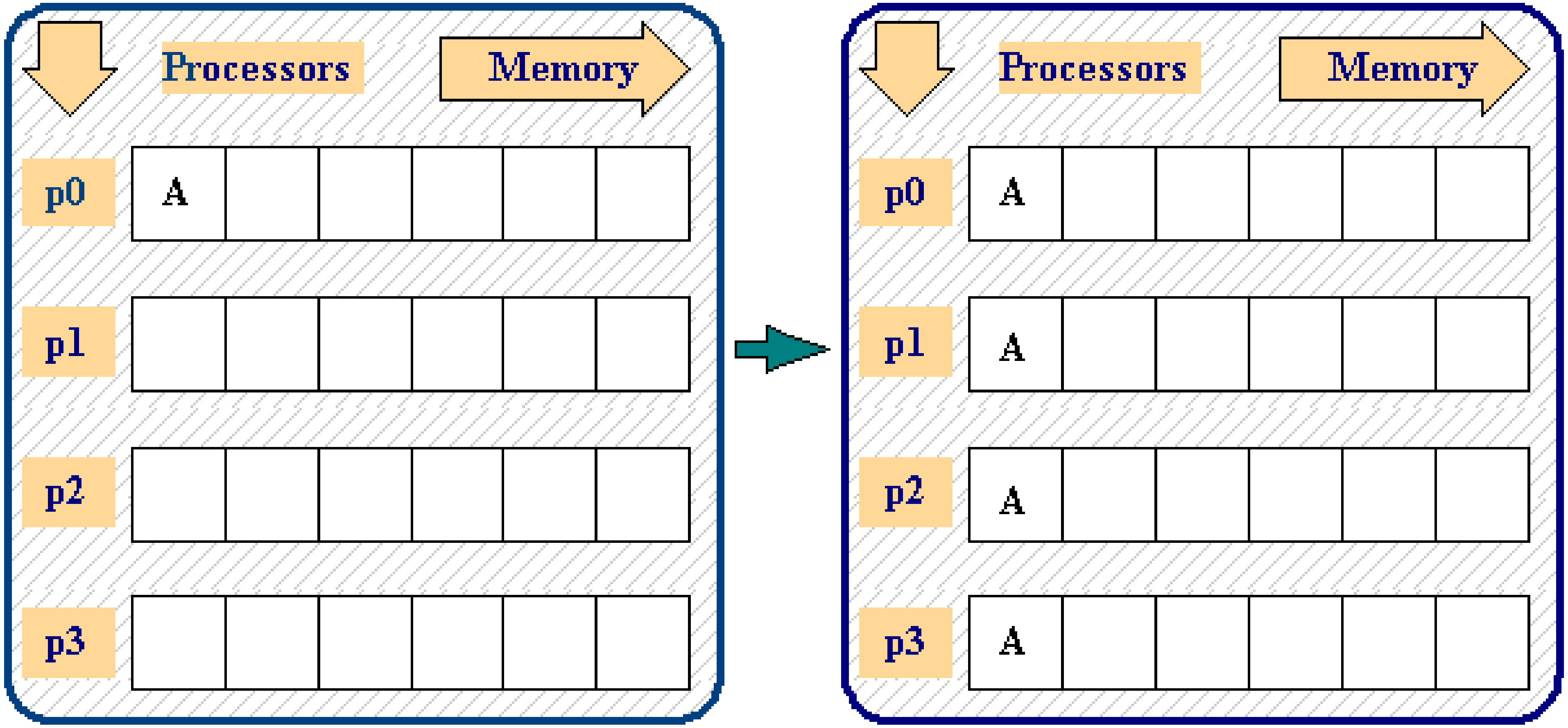
Barrier Synchronization

- MPI_BARRIER is done in software and can incur a substantial overhead on some machines. In general, you should only insert barriers when they are truly needed.

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

Broadcast

- The MPI_BCAST routine enables you to copy data from the memory of the root processor to the same memory locations for other processors in the communicator.



Broadcast

- In this example, one data value in processor 0 is broadcast to the same memory locations in the other three processors.
  - Clearly, you could send data to each processor with multiple calls to one of the send routines, but the broadcast routine makes this data motion a bit easier.
- All processes call the following:

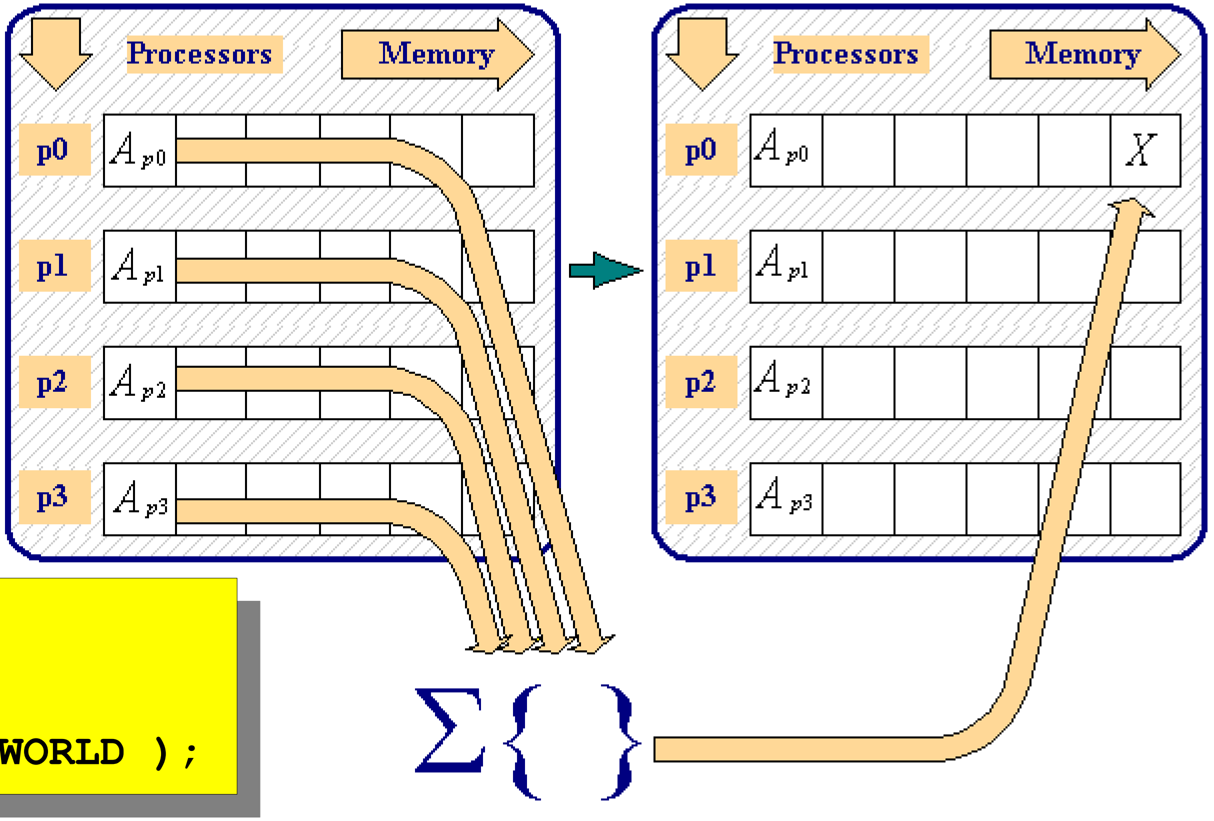
```
...
send_count = 1;
root = 0;
MPI_Bcast ( &a, send_count, MPI_INT, root,
               comm )
...
```

Broadcast Example

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

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

 int rank;
 double param;

 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 if(rank==5) param=23.0;
 MPI_Bcast(&param,1,MPI_DOUBLE,5,MPI_COMM_WORLD);
 printf("P:%d after broadcast parameter is %f \n",rank,param);
 MPI_Finalize();
 return 0;
}
```

Reduction

- The MPI_REDUCE routine enables you to:
  - Collect data from each processor
  - Reduce these data to a single value (such as a sum or max) Store the reduced result on the root processor
- The example shown below sums the values of A on each processor and stores the results in X on processor 0.



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

Reduction

- In general, the calling sequence is:

```
MPI_Reduce( send_buffer, recv_buffer, count, data_type,
reduction_operation, rank_of_receiving_process, communicator )
```

- MPI_REDUCE combines the elements provided in the send buffer, applies the specified operation (sum, min, max, etc.), and returns the result to the receive buffer of the root process where:
  - The send buffer is defined by the arguments send_buffer, count, and datatype.
  - The receive buffer is defined by the arguments recv_buffer, count, and datatype.
  - Both buffers have the same number of elements with the same type.
  - The arguments count and datatype must have identical values in all processes.
  - The argument rank, which is the location of the reduced result, must also be the same in all processes.

Reduction

Operation	Description
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bitwise and
MPI_LOR	logical or
MPI_BOR	bitwise 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 maximum value

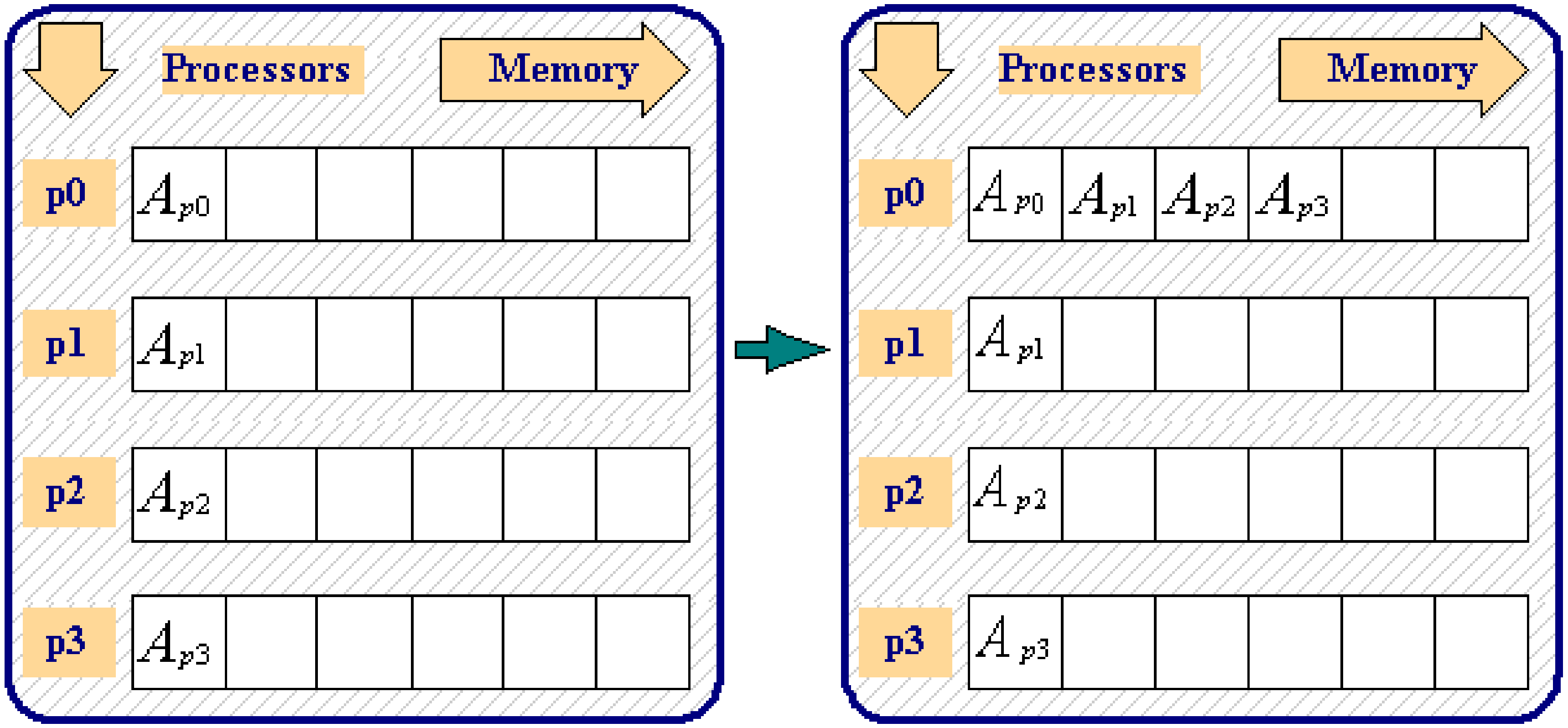
Reduction Example

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

int main(int argc, char *argv[])
{
   int rank;
   int source,result,root;
/* run on 10 processors */
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

root=7;
source=rank+1;
MPI_Barrier(MPI_COMM_WORLD);

MPI_Reduce(&source,&result,1,MPI_INT,MPI_PROD,root,MPI_COMM_WORLD);
if(rank==root) printf("P:%d MPI_PROD result is %d \n",rank,result);

MPI_Finalize();

return 0;
}
```

Gather

- The MPI_GATHER routine is an all-to-one communication.
  - MPI_GATHER has the same arguments as the matching scatter routines.
  - The receive arguments are only meaningful to the root process.
- When MPI_GATHER is called, each process (including the root process) sends the contents of its send buffer to the root process.
  - The root process receives the messages and stores them in rank order.
- The gather also could be accomplished by each process calling MPI_SEND and the root process calling MPI_RECV N times to receive all of the messages.

Gather



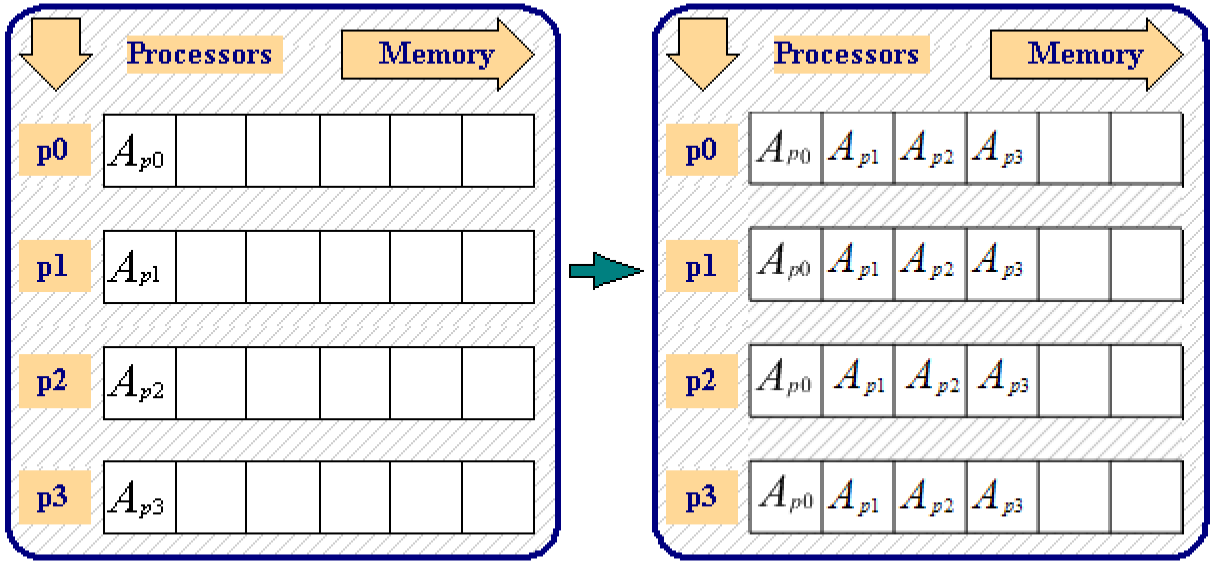
```
send_count = 1;
recv_count = 1;
recv_rank = 0;
MPI_Gather ( &a, send_count, MPI_REAL,
             &a, recv_count, MPI_REAL,
             recv_rank, MPI_COMM_WORLD );
```

Gather Example

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

int main(int argc, char *argv[])
{
   int rank,size;
   double param[16],mine;
   int sndcnt,rcvcnt;
   int i;

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

   sndcnt=1;
   mine=23.0+rank;
   if(rank==7) rcvcnt=1;

   MPI_Gather(&mine,sndcnt,MPI_DOUBLE,param,rcvcnt,MPI_DOUBLE,7,MPI_COMM_WORLD);

   if(rank==7)
    for(i=0;i<size;++i) printf("PE:%d param[%d] is %f \n",rank,i,param[i]);

   MPI_Finalize();
   return 0;
}
```

MPI_ALLGATHER

- In the MPI_GATHER example, after the data are gathered into processor 0, you could then MPI_BCAST the gathered data to all of the other processors.
- It is more convenient and efficient to gather and broadcast with the single MPI_ALLGATHER operation, which will result in the following:

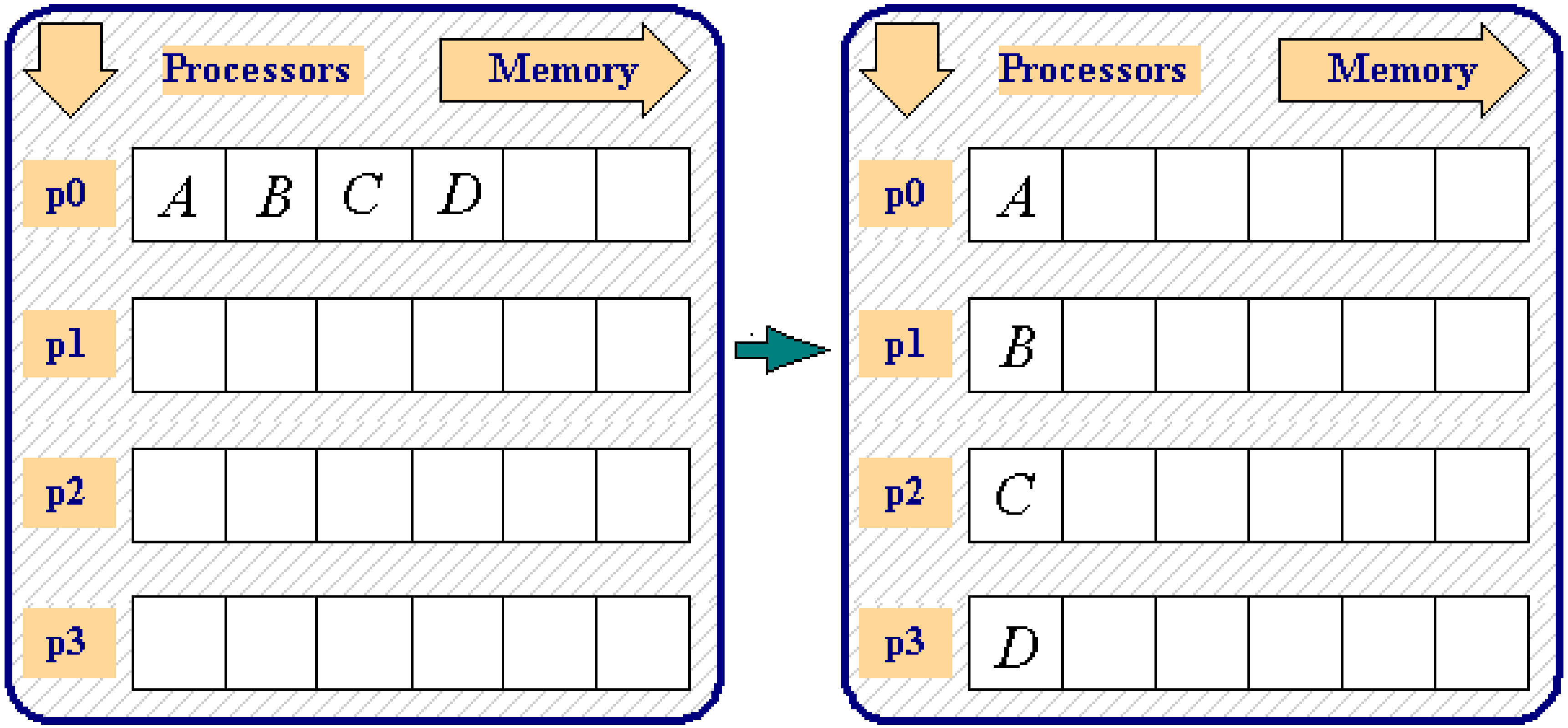


- The calling sequence for MPI_ALLGATHER is exactly the same as the calling sequence for MPI_GATHER.

Scatter

- The MPI_SCATTER routine is a one-to-all communication.
  - Different data are sent from the root process to each process (in rank order).
- When MPI_SCATTER is called, the root process breaks up a set of contiguous memory locations into equal chunks and sends one chunk to each processor.
  - The outcome is the same as if the root executed N MPI_SEND operations and each process executed an MPI_RECV.
  - The send arguments are only meaningful to the root process.

Scatter



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

Scatter Example

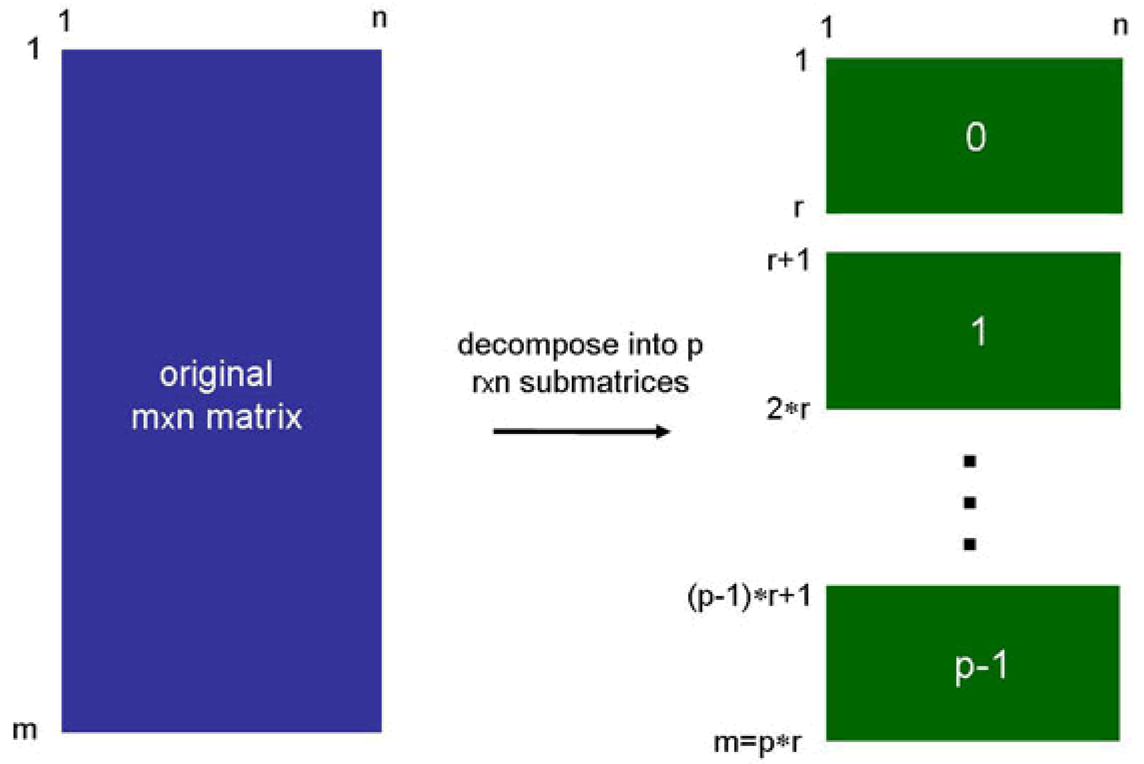
```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
int rank,size,i;
double param[8],mine;
int sndcnt,rcvcnt;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
rcvcnt=1;
if(rank==3) {
  for(i=0;i<8;++i) param[i]=23.0+i;
  sndcnt=1;
}
MPI_Scatter(param,sndcnt,MPI_DOUBLE,&mine,rcvcnt,MPI_DOUBLE,3,MPI_COMM_WORLD);
for(i=0;i<size;++i)  {
  if(rank==i) printf("P:%d mine is %f \n",rank,mine);
  fflush(stdout);
  MPI_Barrier(MPI_COMM_WORLD);
}
MPI_Finalize();
return 0;
}
```

Advanced Operations

- MPI_ALLREDUCE - used to combine the elements of each process's input buffer and stores the combined value on the receive buffer of all group members.
- User-Defined Reduction Operations - enable reduction to be defined as an arbitrary operation.
- Gather / Scatter Vector Operations - MPI_GATHERV and MPI_SCATTERV allow a varying count of data from/to each process.
- Other Gather / Scatter Variations - MPI_ALLGATHER and MPI_ALLTOALL
  - No root process specified: all processes get gathered or scattered data.
  - Send and receive arguments are meaningful to all processes.
- MPI_SCAN - used to carry out a prefix reduction on data throughout the group and returns the reduction of the values of all of the processes.
- MPI_REDUCE_SCATTER combines an MPI_REDUCE and an MPI_SCATTERV.

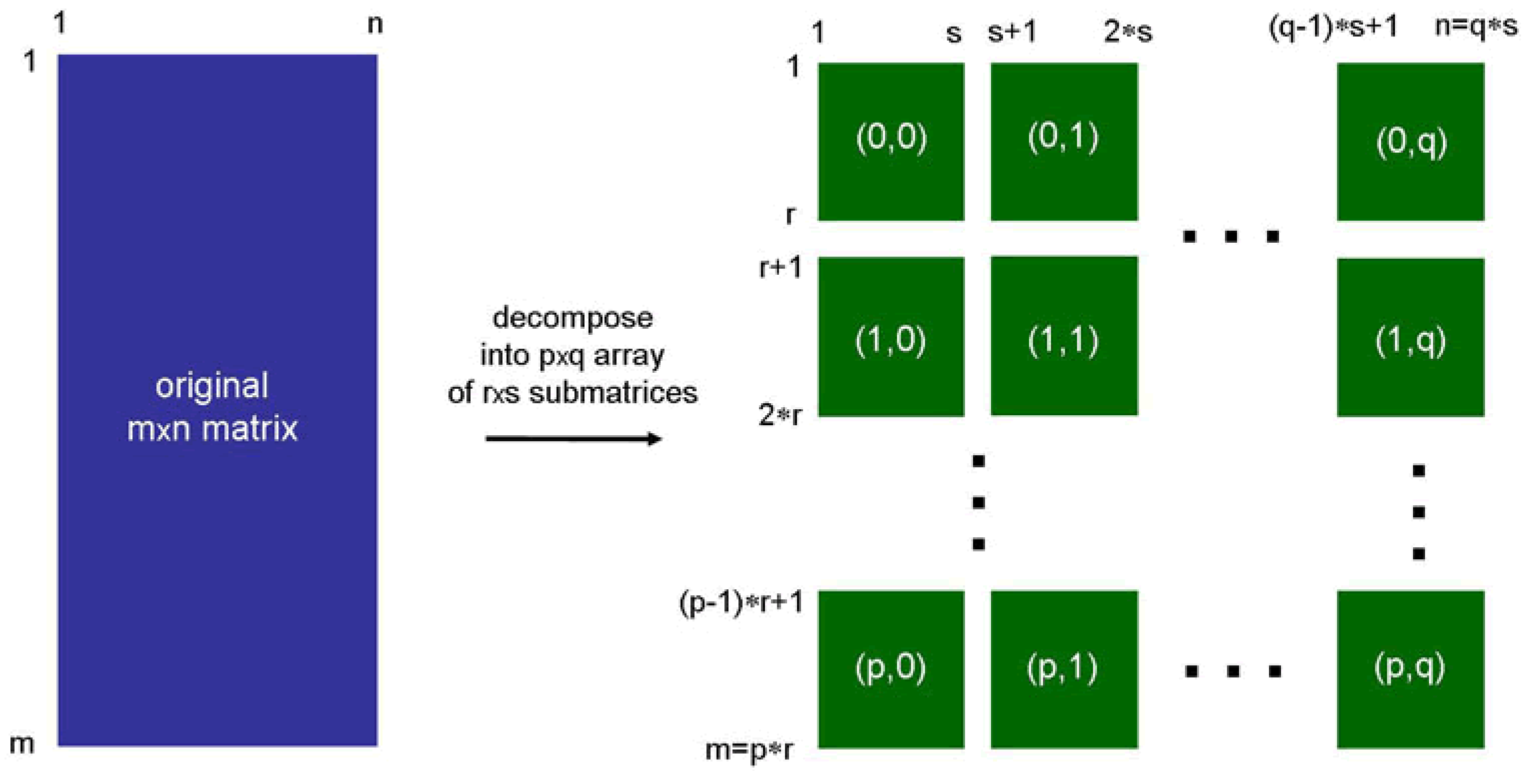
Virtual Topologies

- Many computational science and engineering problems involve the manipulation of large matrices.
  - Frequently in multiprocessing, these matrices are partitioned, or domain-decomposed, so that each partition (or subdomain) is assigned to a process.
  - One such example is an m x n matrix decomposed into p q x n submatrices, with a submatrix assigned to each p process



Virtual Topologies

- In this example it is easy to identify and keep track of the submatrices, since they can be numbered sequentially from top to bottom.
  - An algorithm might dictate that the matrix be decomposed into a p x q logical grid, whose elements are themselves each an r x s matrix



Virtual Topologies

- This requirement might be due to efficiency considerations, ease in code implementation, or code clarity.
- Although it is possible to refer to each of these p x q subdomains by a sequential rank number, a 2-dimensional rank numbering system results in a much clearer and natural computational representation.
- To address the needs of this and other topological layouts, the MPI library provides two types of topology routines:
  - Cartesian
  - Graph
- The graph topologies allow more flexibility.

MPI Topology Routines

- MPI includes routines for working with Cartesian Topologies. Some of these routines are
  - MPI_CART_CREATE
  - MPI_CART_COORDS
  - MPI_CART_RANK
  - MPI_CART_SUB
  - MPI_CARTDIM_GET
  - MPI_CART_GET
  - MPI_CART_SHIFT

MPI_CART_CREATE

- The MPI_CART_CREATE routine creates a new communicator using a Cartesian topology. The calls to MPI_CART_CREATE is as follows:

```
int MPI_Cart_create(MPI_Comm old_comm, int ndims,
int  *dim_size, int *periods, int reorder,
MPI_Comm *new_comm)
```

- The arguments for this routine are:
  - ndims – number of dimensions
  - dim_size – array of size ndims specifying number of processors in each dimension
  - periods - array of size ndims specifying periodicity status of each dimension
  - reorder - whether process rank reordering by MPI is permitted
  - new_comm - communicator handle

MPI_CART_CREATE

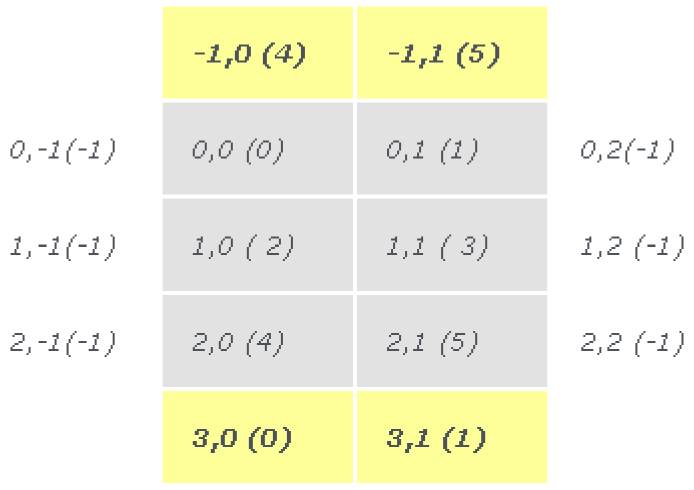
```
#include "mpi.h"
MPI_Comm old_comm, new_comm;
int ndims, reorder, periods[2], dim_size[2];

old_comm = MPI_COMM_WORLD;
ndims = 2;          /*  2-D matrix/grid */
dim_size[0] = 3;    /* rows */
dim_size[1] = 2;    /* columns */
periods[0] = 1;     /* row periodic (each column forms a ring) */
periods[1] = 0;     /* columns nonperiodic */
reorder = 1;        /* allows processes reordered for efficiency */

MPI_Cart_create(old_comm, ndims, dim_size,
             periods, reorder, &new_comm);
```

MPI_CART_CREATE

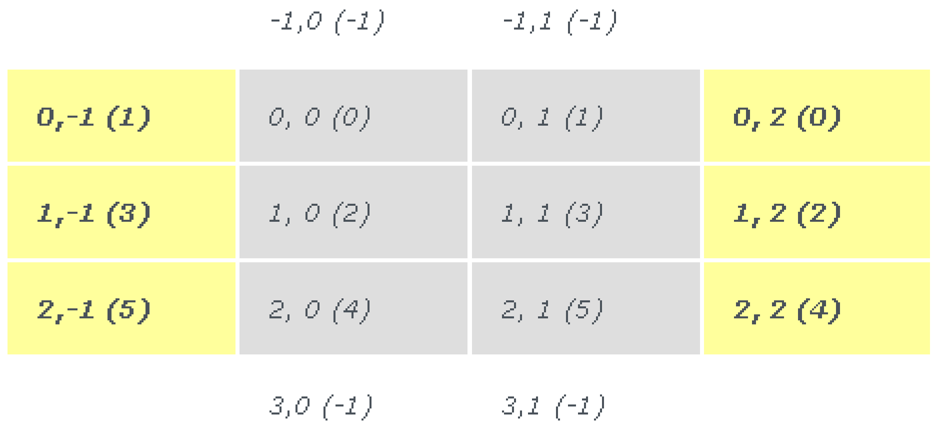
- In the example we use MPI_CART_CREATE to map (or rename) 6 processes from a linear ordering (i.e., 0,1,2,3,4,5) into a two-dimensional matrix ordering of 3 rows by 2 columns (i.e., (0,0), (0,1), ..., (2,1)).
- Figure below depicts the resulting Cartesian grid representation for the processes.
  - The index pair i,j represents row i and column j.
  - The corresponding (linear) rank number is enclosed in parentheses.

0,0 (0)	0,1 (1)
1,0 (2)	1,1 (3)
2,0 (4)	2,1 (5)

MPI_CART_CREATE - Periodicity and Reorder



periods[0]=1.;periods[1]=0



periods[0]=0;periods[1]=1

- While the processes are arranged logically as a Cartesian topology, the processors corresponding to these processes may in fact be scattered physically - even within a shared-memory machine.
  - If reorder is set to "1", MPI may reorder the process ranks in the new communicator (for potential gain in performance due to, say, the physical proximities of the processes assigned).
  - If reorder is "0", the process rank in the new communicator is identical to its rank in the old communicator.

MPI_CART_CREATE - Summary

- MPI_CART_CREATE is a collective communication function so it must be called by all processes in the group.
  - Like other collective communication routines, MPI_CART_CREATE uses blocking communication.
  - It is not required to be synchronized among processes in the group and hence is implementation dependent.
- If the total size of the Cartesian grid is smaller than available processes, those processes not included in the new communicator will return MPI_COMM_NULL.
- If the total size of the Cartesian grid is larger than available processes, the call results in error.

MPI_CART_COORDS

- The MPI_CART_COORDS routine returns the corresponding Cartesian coordinates of a (linear) rank in a Cartesian communicator. The calls to MPI_CART_COORDS is as follows:

```
int MPI_Cart_coords( MPI_Comm comm, int rank,
int maxdims, int *coords )
```

- The arguments for this routine are:
- comm - communicator handle
- rank - process rank in linear coordinates
- maxdims - number of dimensions in cartesian topology
- coords   - corresponding cartesian coordinates of rank

MPI_CART_COORDS Example

```
MPI_Cart_create(old_comm, ndims, dim_size,
         periods, reorder, &new_comm);  /* creates communicator */

if(Iam == root) {   /* only want to do this on one process */
   for (rank=0; rank<p; rank++) {
     MPI_Cart_coords(new_comm, rank, ndims, &coords);
     printf("%d, %d\n ",rank, coords[0]);
   }
 }
```

0,0 (0)	0,1 (1)
1,0 (2)	1,1 (3)
2,0 (4)	2,1 (5)

MPI_CART_RANK

- The MPI_CART_RANK routine returns the corresponding process rank of the Cartesian coordinates of a Cartesian communicator.
- The call to MPI_CART_RANK is as follows:
  - **`int MPI_Cart_rank( MPI_Comm comm, int *coords, int *rank )`**
- The arguments for this routine are:
  - comm - Cartesian communicator handle
  - coords - array specifying Cartesian coordinates
  - rank - process rank of process specified by its Cartesian coordinates, coords

MPI_CART_RANK Example

```
MPI_Cart_create(old_comm, ndims, dim_size,
        periods, reorder, &new_comm);

if(Iam == root) {         /* only want to do this on one process */
  for (i=0; i<nv; i++) {
    for (j=0; j<mv; j++)  {
      coords[0] = i;
      coords[1] = j;
      MPI_Cart_rank(new_comm, coords, &rank);
      printf("%d, %d, %d\n",coords[0],coords[1],rank);
    }
  }
}
```

0,0 (0)	0,1 (1)
1,0 (2)	1,1 (3)
2,0 (4)	2,1 (5)

MPI_CART_SUB

- The MPI_CART_SUB routine creates new communicators for subgrids of up to (N-1) dimensions from an N-dimensional Cartesian grid.
- Often, after we have created a Cartesian grid, we wish to further group elements of this grid into subgrids of lower dimensions.
  - Typical operations requiring subgrids include reduction operations such as the computation of row sums, column extremums.
  - For example, the subgrids of a 2-dimensional Cartesian grid are 1-dimensional grids of the individual rows or columns.
  - Similarly, for a 3-dimensional Cartesian grid, the subgrids can either be 2- or 1-dimensional.

MPI_CART_SUB

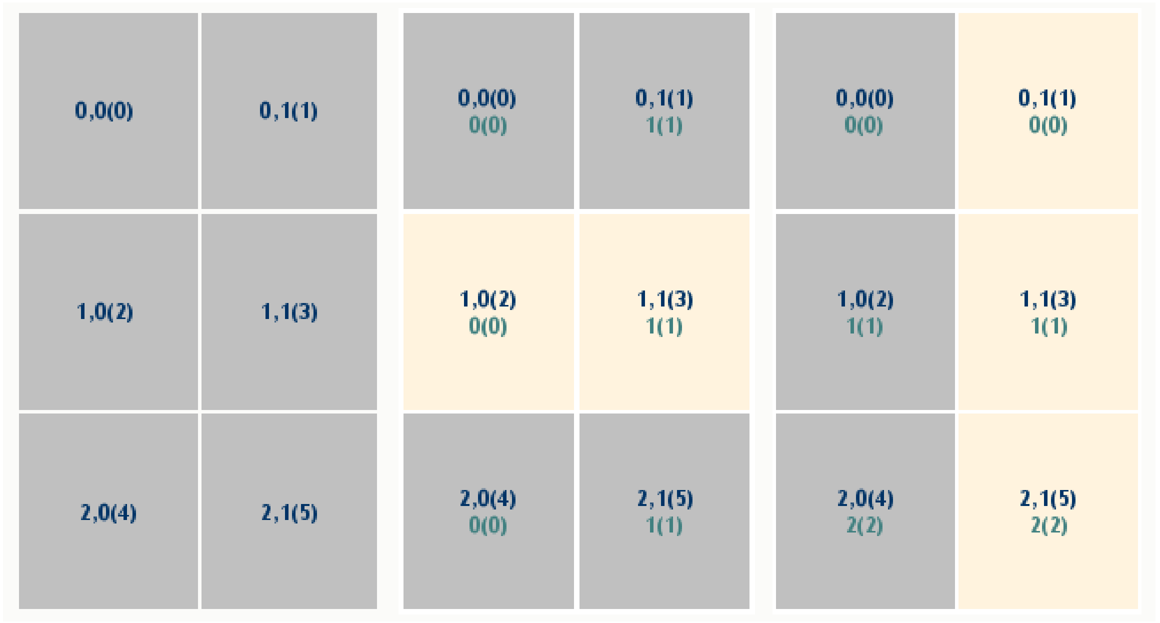
- The calls to MPI_CART_SUB in both C and Fortran are as follows:

```
int MPI_Cart_sub( MPI_Comm old_comm,
            int *belongs, MPI_Comm *new_comm )
```

- The arguments for this routine are:
  - old_comm - Cartesian communicator handle
  - belongs - array of size ndims specifying whether a dimension belongs to new_comm
  - new_comm - Cartesian communicator handle

MPI_CART_SUB Example

```
/* Create 2D Cartesian topology for processes */
  MPI_Cart_create(MPI_COMM_WORLD, ndim, dims,
              period, reorder, &comm2D);
  MPI_Comm_rank(comm2D, &id2D);
  MPI_Cart_coords(comm2D, id2D, ndim, coords2D);
/* Create 1D row subgrids */
  belongs[0] = 0;
  belongs[1] = 1;      ! this dimension belongs to subgrid
  MPI_Cart_sub(comm2D, belongs, &commrow);
/* Create 1D column subgrids */
  belongs[0] = 1;       /* this dimension belongs to subgrid */
  belongs[1] = 0;
  MPI_Cart_sub(comm2D, belongs, &commcol);
```



2D Cartesian Grid   3 Row Subgrids   2 Column Subgrid

MPI_CART_SUB

- MPI_CART_SUB is a collective routine. It must be called by all processes in old_comm.
- MPI_CART_SUB-generated subgrid communicators are derived from Cartesian grid created with MPI_CART_CREATE.
- Full length of each dimension of the original Cartesian grid is used in the subgrids.
- Each subgrid has a corresponding communicator. It inherits properties of the parent Cartesian grid; it remains a Cartesian grid.
- It returns the communicator to which the calling process belongs.
- There is a comparable MPI_COMM_SPLIT to perform similar function.
- MPI_CARTDIM_GET and MPI_CART_GET can be used to acquire structural information of a grid (such as dimension, size, periodicity).

MPI_CARTDIM_GET

- The MPI_CARTDIM_GET routine determines the number of dimensions of a subgrid communicator.
  - On occasions, a subgrid communicator may be created in one routine and subsequently used in another routine.
  - If the dimension of the subgrid is not available, it can be determined by MPI_CARTDIM_GET.
- The call to MPI_CARTDIM is as follows:

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

- The arguments for this routine are:
  - comm - Cartesian communicator handle
  - ndims - number of dimensions

MPI_CART_GET

- The MPI_CART_GET routine retrieves properties such as periodicity and size of a subgrid.
  - On occasions, a subgrid communicator may be created in one routine and subsequently used in another routine.
  - If only the communicator is available in the latter, this routine, along with MPI_CARTDIM_GET, may be used to determine the size and other pertinent information about the subgrid.
- The call to MPI_CART_GET is as follows:

```
int MPI_Cart_get(MPI_Comm subgrid_comm, int ndims,
            int *dims, int *periods, int *coords )
```

- The arguments for this routine are:
  - subgrid_comm - communicator handle
  - ndims - Number of dimensions
  - dims - array of size ndims providing length in each dimension
  - periods - array of size ndims specifying periodicity status of each dimension
  - coords -array of size ndims providing Cartesian coordinates of calling process

MPI_CART_GET Example

```
   /* create Cartesian topology for processes */
     dims[0] = nrow;
     dims[1] = mcol;
     MPI_Cart_create(MPI_COMM_WORLD, ndim, dims,
                period, reorder, &grid_comm);
     MPI_Comm_rank(grid_comm, &me);
     MPI_Cart_coords(grid_comm, me, ndim, coords);
   /* create column subgrids */
     remain[0] = 1;
     remain[1] = 0;
     MPI_Cart_sub(grid_comm, remain, &row_comm);
   /* Retrieve subgrid dimensions and other info */
     MPI_Cartdim_get(row_comm, &mdims);
     MPI_Cart_get(row_comm, mdims, dims, period, row_coords);
```

0,0 (0)	0,1 (1)
1,0 (2)	1,1 (3)
2,0 (4)	2,1 (5)

MPI_CART_SHIFT

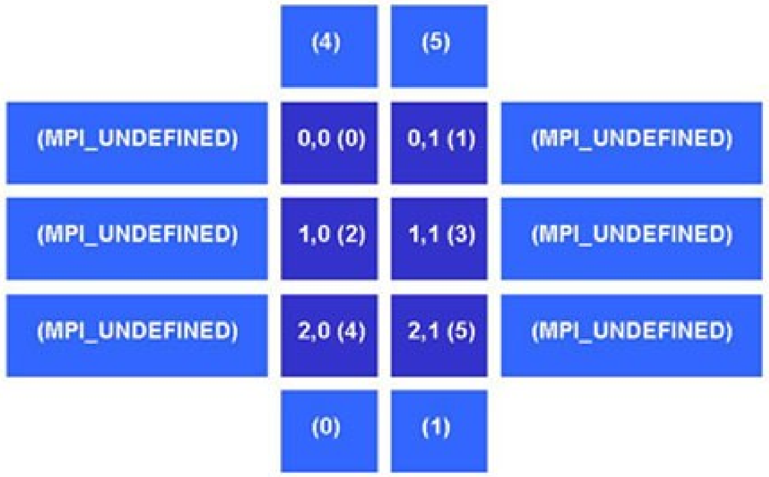
- The MPI_CART_SHIFT routine finds the resulting source and destination ranks, given a shift direction and amount. The call to MPI_CART_SHIFT is as follows:

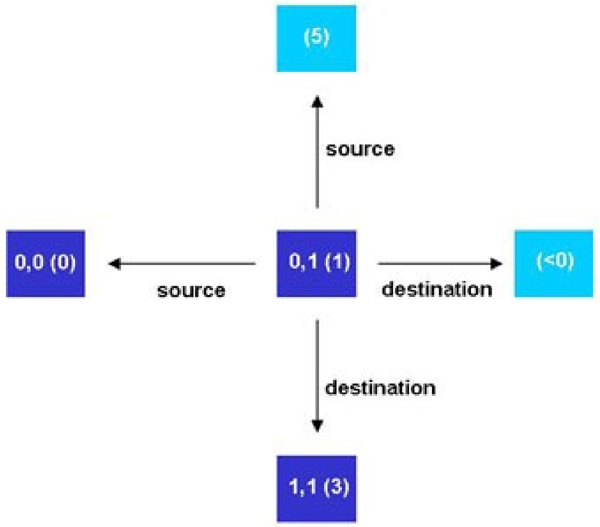
  ```
  int MPI_Cart_shift( MPI_Comm comm, int
  direction, int displ, int *source, int *dest )
  ```

- The arguments for this routine are:
  - comm - communicator handle
  - direction - the dimension along which shift is to be in effect
  - displ - amount and sense of shift (<0; >0; or 0)
  - source - the source of shift (a rank number)
  - dest - the destination of shift (a rank number)

MPI_CART_SHIFT

```
/* create Cartesian topology for processes */
  ndim = 2;             /* number of dimensions */
  dims[0] = nrow;       /* number of rows     */
  dims[1] = mcol;       /* number of columns  */
  period[0] = 1;        /* cyclic in this direction */
  period[1] = 0;        /* not cyclic in this direction */
  MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, period, reorder, &comm2D);
  MPI_Comm_rank(comm2D, &me);
  MPI_Cart_coords(comm2D, me, ndim, coords);

  displ =  1;    /* shift by  1 */
  index =  0;    /* shift along the 1st index (out of 2) */
  MPI_Cart_shift(comm2D, index, displ, &source0, &dest0);
  index =  1;    /* shift along the 2nd index (out of 2) */
  MPI_Cart_shift(comm2D, index, displ, &source1, &dest1);
```





MPI_CART_SHIFT

- Direction, in the Cartesian grid dimension index, has range (0, 1, ..., ndim-1).
  - For a 2-dimensional grid, the two choices for direction are 0 and 1.
- MPI_CART_SHIFT is a query function.
  - No action results from its call.
- A negative returned value (MPI_UNDEFINED) of source or destination signifies the respective value is out of bounds.
  - It also implies that there is no periodicity along that direction.
- If periodic condition is enabled along the shift direction, an out of bounds condition will not occur.

The Laplace Equation

- The Laplace equation has the form:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=0$$

where *u=u(x,y)* is an unknown scalar potential subjected to some boundary conditions

$$\begin{aligned} u(x,0)&=\sin(\pi x) && 0\leqslant x\leqslant 1\\ u(x,1)&=\sin(\pi x)\mathrm{e}^{-\pi} && 0\leqslant x\leqslant 1\\ u(0,y)&=u(1,y)=0 && 0\leqslant y\leqslant 1 \end{aligned}$$

The Laplace Equation Discretization

- Making the equation discrete numerically with centered difference results in the algebraic equation:

$$u_{i,j}=\frac{u_{i+1,j}+u_{i-1,j}+u_{i,j-1}+u_{i,j+1}}{4};i=i,\ldots,m;\ j=1,\ldots,m$$

where $u_{i-1,j}$ represents the following:

$$u_{i-1,j}=u(x_{i-1},y_j);i=1,\ldots,m;\ j=1,\ldots,m$$
$$u_{i-1,j}=u((i-1)\Delta x,j\Delta y)$$

and for simplicity, we define:

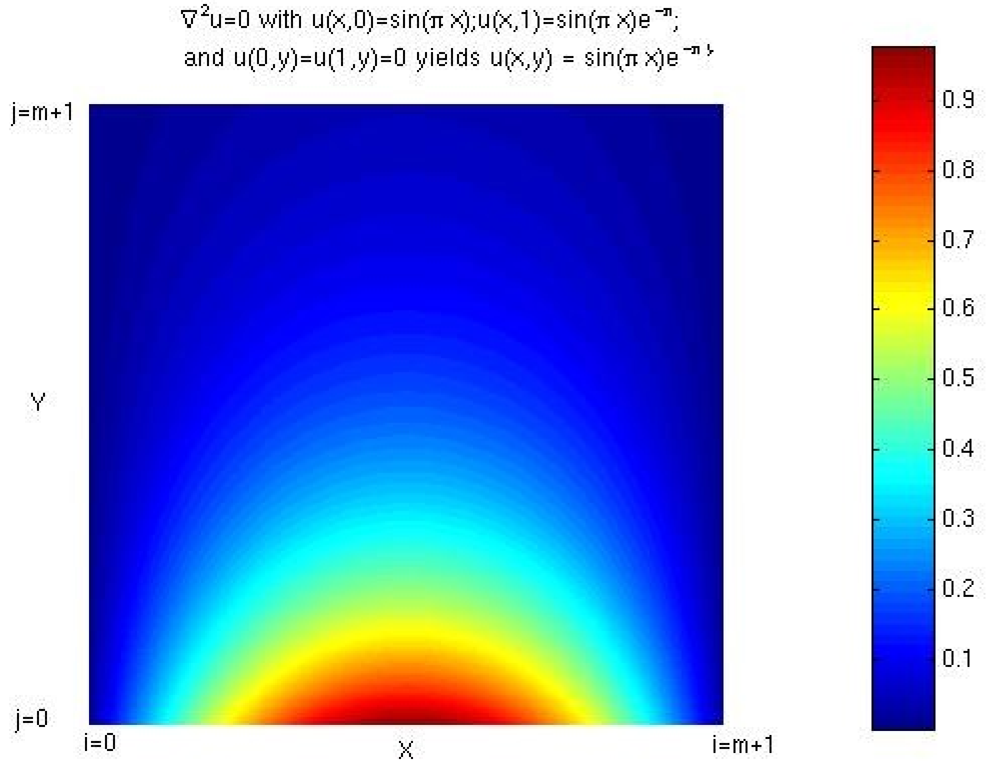
$$\Delta x=\Delta y=\frac{1}{m+1}$$

The Laplace Equation Solution

- The analytical solution for this boundary value problem can easily be verified to be:

$$u(x,y)=\sin(\pi x)e^{-\pi y};0\leqslant x\leqslant 1;0\leqslant y\leqslant 1$$

The solution is shown below in a contour plot with x pointing from left to right and y going from bottom to top.



Jacobi Scheme

- We will focus on the use of two iterative methods to solve the equation.
  - These methods will be shown to be readily parallelizable, as well as lending themselves to the opportunity to apply the MPI Cartesian topology we introduced earlier.
  - The simplest of iterative techniques is the Jacobi scheme, which may be stated as follows:

    1. Make initial guess for $u_{i,j}$ at all interior points (i,j) for all i=1:m and j=1:m.
    2. Use equation below to compute $u^{n+1}_{i,j}$ at all interior points (i,j)

    $$u^{n+1}_{i,j} = \frac{u^{n}_{i+1,j} + u^{n}_{i-1,j} + u^{n}_{i,j-1} + u^{n}_{i,j+1}}{4} ; i = i, \ldots, m ; j = 1, \ldots, m$$

    3. Stop if the prescribed convergence threshold is reached, otherwise continue on to the next step.
    4. $u^{n}_{i,j} = u^{n+1}_{i,j}$
    5. Go to Step 2.

Parallel Jacobi Scheme

- To enable parallelism, the work must first be divided among the individual processes; this is commonly known as domain decomposition.
- The governing equation is two-dimensional
  - 1-dimensional or 2-dimensional decomposition possible
  - We focus on a 1-dimensional decomposition
  - p processes will be used
  - The computational domain is split into p horizontal strips, each assigned to one process, along the north-south or y-direction

Parallel Jacobi Scheme

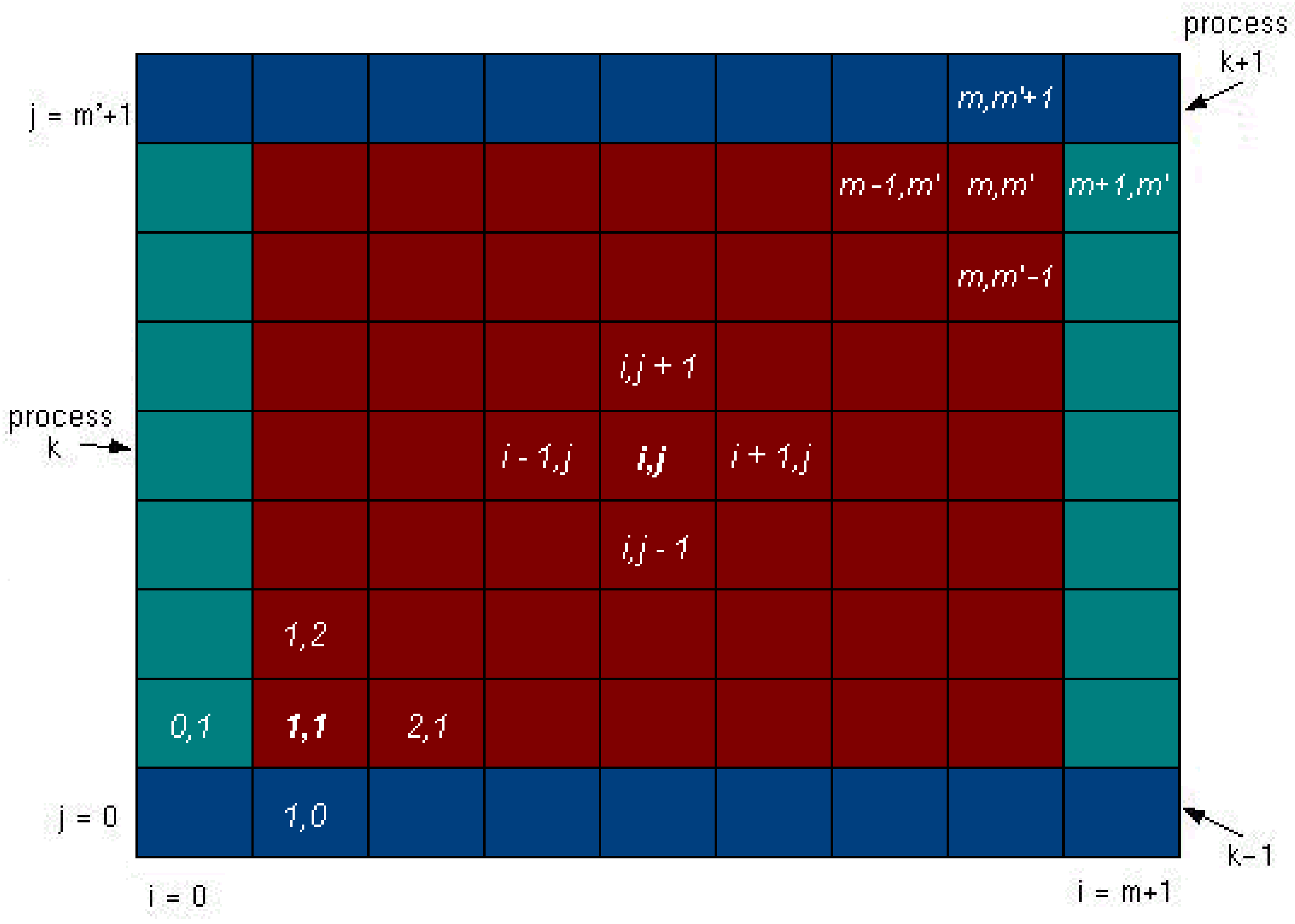
- For the obvious reason of better load-balancing, we will divide the amount of work, in this case proportional to the grid size, evenly among the processes (m x m / p).
- For convenience, m' = m/p is defined as the number of cells in the y-direction for each process.
- Iterative equation is restated for a process k as follows:

$$v_{i,j}^{n+1,k} = \frac{v_{i+1,j}^{n,k} + v_{i-1,j}^{n,k} + v_{i,j-1}^{n,k} + v_{i,j+1}^{n,k}}{4} ; i = i, \ldots, m ; j = 1, \ldots, m' ; k = 0, \ldots, p-1$$

- where v denotes the local solution corresponding to the process k with m'=m/p.

Parallel Jacobi Scheme

- Figure below depicts the grid of a typical process k, as well as part of the adjoining grids of k-1, k+1.



Parallel Jacobi Scheme

- The red cells represent process k's grid cells for which the solution u is sought through iterative equation.
- The blue cells on the top row represent cells belonging to the first row (j = 1) of cells of process k+1. The blue cells on the bottom row represent the last row (j = m') of cells of process k-1.
- It is important to note that the u at the blue cells of k belong to adjoining processes (k-1 and k+1) and hence must be "imported" via MPI message passing routines.
  - Similarly, process k's first and last rows of cells must be "exported" to adjoining processes for the same reason.
- For i = 1 and i = m, an iterative equation again requires an extra cell beyond these two locations. These cells contain the prescribed boundary conditions (u(0,y) = u(1,y) = 0) and are colored green to distinguish them from the red and blue cells.
  - No message passing operations are needed for these green cells as they are fixed boundary conditions and are known a priori.

Parallel Jacobi Scheme

- From the standpoint of process k, the blue and green cells may be considered as additional "boundary" cells around it.
- As a result, the range of the strip becomes (0:m+1,0:m'+1).
- Physical boundary conditions are imposed on its green cells, while u is imported to its blue "boundary" cells from two adjoining processes.
  - With the boundary conditions in place, iterative equation can be applied to all of its interior points.
  - Concurrently, all other processes proceed following the same procedure.
  - It is interesting to note that the grid layout for a typical process k is completely analogous to that of the original undivided grid.
  - Whereas the original problem has fixed boundary conditions, the problem for a process k is subjected to variable boundary conditions.

Successive Over Relaxation (SOR)

- While the Jacobi iteration scheme is very simple and easily parallelized, its slow convergent rate renders it impractical for any real-world applications.
  - One way to speed up the convergent rate would be to "over predict" the new solution by linear extrapolation.
  - This approach leads to the Successive Over Relaxation (SOR) scheme shown below:

Successive Over Relaxation (SOR)

1. Make initial guess for $u_{i,j}$ at all interior points (i,j).
2. Define a scalar w_n ($0 < w_n < 2$).
3. Apply equation below to all interior points (i,j)

$$u'_{i,j} = \frac{u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j-1} + u^n_{i,j+1}}{4} ; i = i, \ldots, m ; j = 1, \ldots, m$$

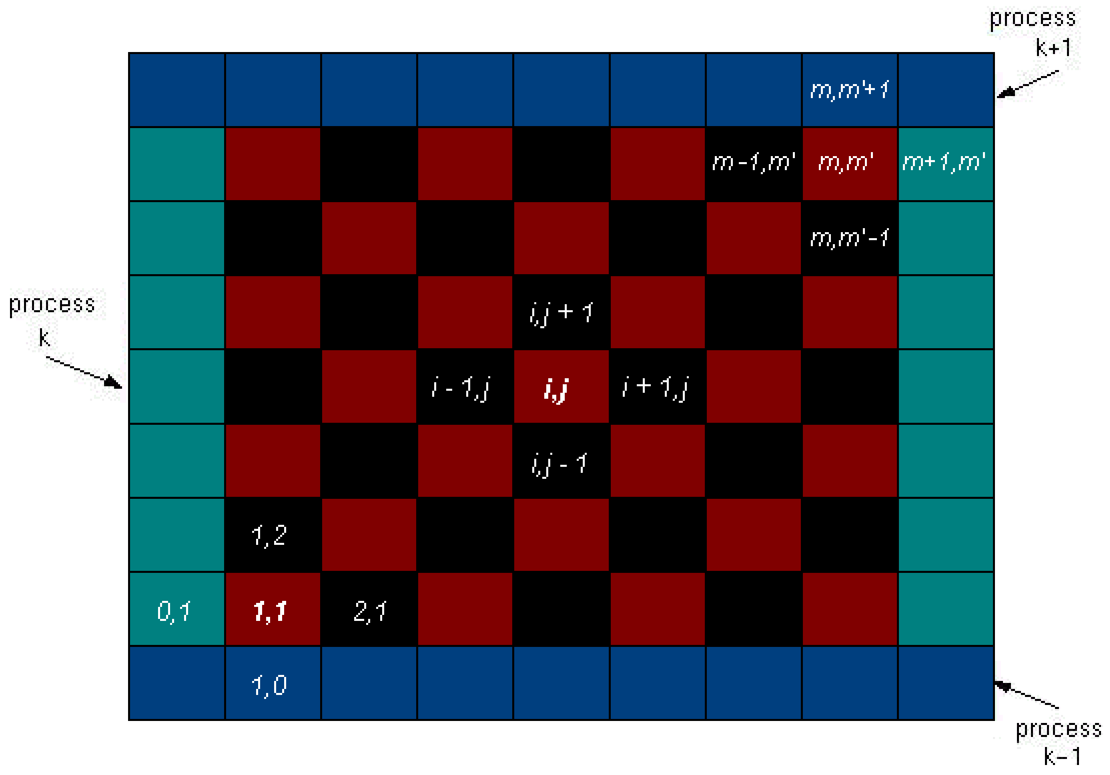
4. $u^{n+1}_{i,j} = w_n \, u'_{i,j} + (1 - w_n) \, u^n_{i,j}$
5. Stop if the prescribed convergence threshold is reached, otherwise continue to the next step.
6. $u^n_{i,j} = u^{n+1}_{i,j}$
7. Go to Step 2.

Successive Over Relaxation (SOR)

- In the above setting $w_n = 1$ recovers the Jacobi scheme while $w_n < 1$ under-relaxes the solution.
- Ideally, the choice of w_n should provide the optimal rate of convergence and is not restricted to a fixed constant.
- We can further speed up the rate of convergence by using u from iteration n+1 for any or all terms on the right hand side as soon as they become available.
  - This approach is the essence of the Gauss-Seidel scheme.
  - A conceptually similar red-black scheme will be used here, which is best understood visually by painting the interior cells alternately in red and black to yield a checkerboard-like pattern.

Successive Over Relaxation (SOR)

- By using this red-black group identification strategy and applying the five-point finite-difference stencil to a point (i,j) located at a red cell, it is immediately apparent that the solution at the red cell depends only on its four immediate black neighbors to the north, east, west, and south.
- Similiarly a point (i,j) located at a black cell depends only on its north, east, west, and south red neighbors.



Successive Over Relaxation (SOR)

- The finite-difference stencil in effects an uncoupling of the solution at interior cells such that the solution at the red cells depends only on the solution at the black cells and vice versa.
- In a typical iteration, if we first perform an update on all red (i,j) cells, then when we perform the remaining update on black (i,j) cells, the red cells that have just been updated could be used.
- Otherwise, everything that we described about the Jacobi scheme applies equally well here; i.e., the green cells represent the physical boundary conditions while the solutions from the first and last rows of the grid of each process are deposited into the blue cells of respective process grids to be used as the remaining boundary conditions.