UNIVERSITÀ di VERONA

Dipartimento di INFORMATICA

PARCO Parallel Computing Lab

Part II

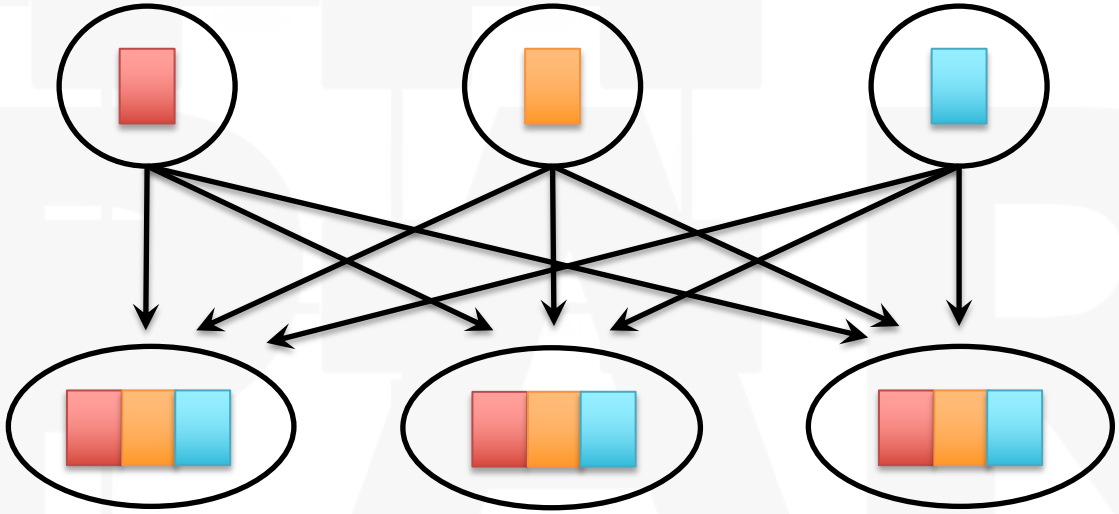
Laurea magistrale in Ingegneria e Scienze Informatiche
Laurea Magistrale in Medical Bioinformatics

Nicola Bombieri – Federico Busato

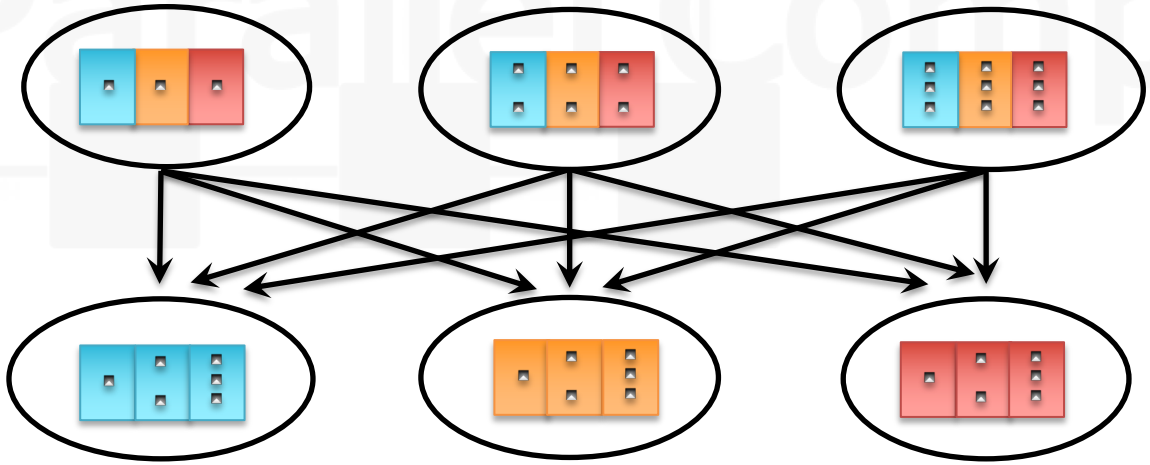
Agenda

- MPI collective communication (Intermediate)
  - All-gather, All-to-All, All-Reduce
- MPI Derivated data types:
  - Continuos, Vector, Indexed, Struct
- MPI collective communication (Advanced)
  - Blocking communication
- Advanced MPI Concept
  - Network Topologies
- MPI performance factors
- Message passing protocol
  - Eager protocol, Rendezvous protocol
- Sender-Receiver synchronization
  - Polling, Interrupt
- Conclusion
- Example
  - Derivated Types
- Exercises
  - Matrix Diagonal
  - Intermediate Collective Communication

All-Gather



All-to-All



Collective communication

Intermediate Methods (I)

```
MPI::Comm::Allgather(void* sendbuf, int sendcount,
          MPI::Datatype& sendtype,
          void* recvbuf, int recvcount,
          MPI::Datatype& recvtype)
```

- Gathers data from all processes
- ALLGATHER is similar to GATHER, except that all processes receive the result, instead of just the root (there is no *root* among parameters)

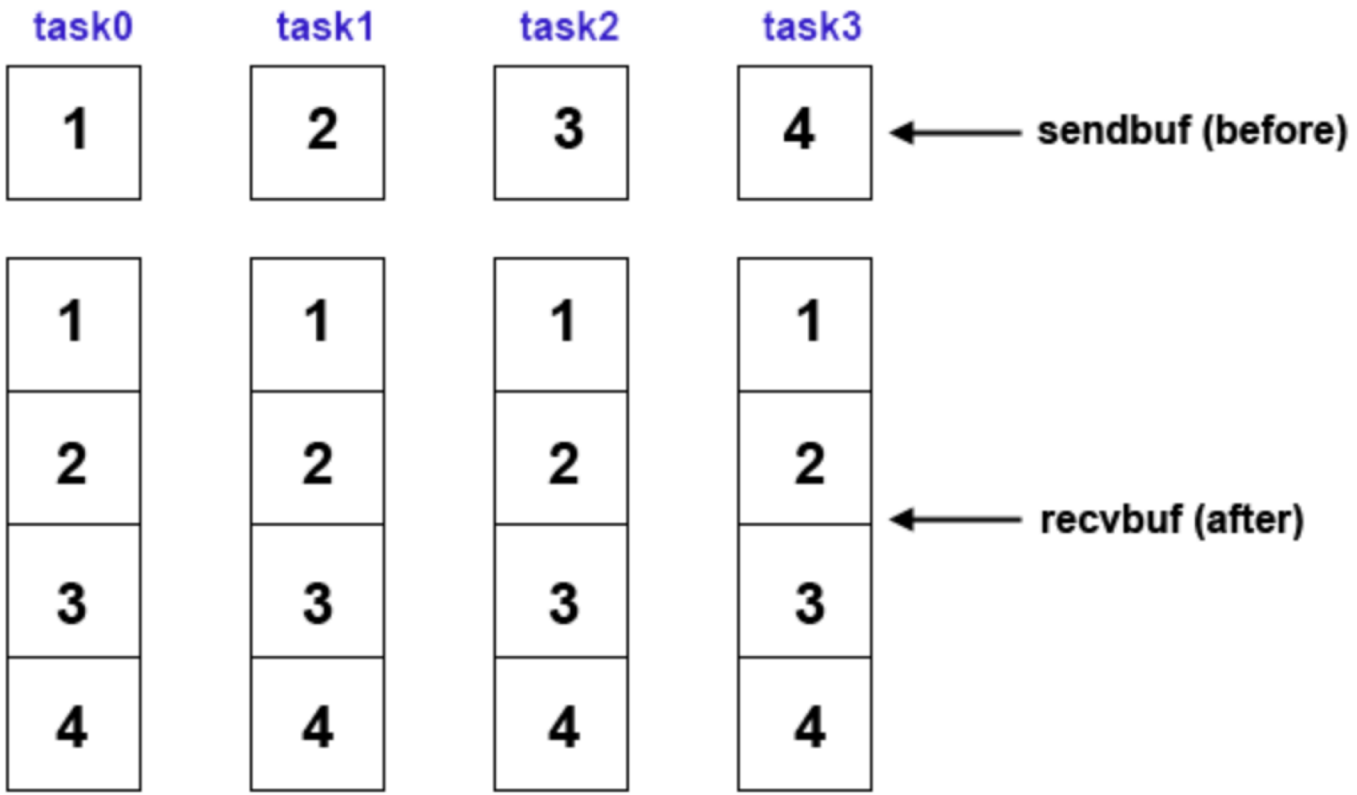
```
MPI::Comm::Alltoall(void* sendbuf, int sendcount,
          MPI::Datatype& sendtype,
          void* recvbuf, int recvcount,
          MPI::Datatype& recvtype)
```

- All processes send data to all
- All arguments on all processes are significant
- There is no *root* among parameters
- Something like matrix transpose

Allgather: example

Gathers data from all tasks and then distributes to all tasks in communicator

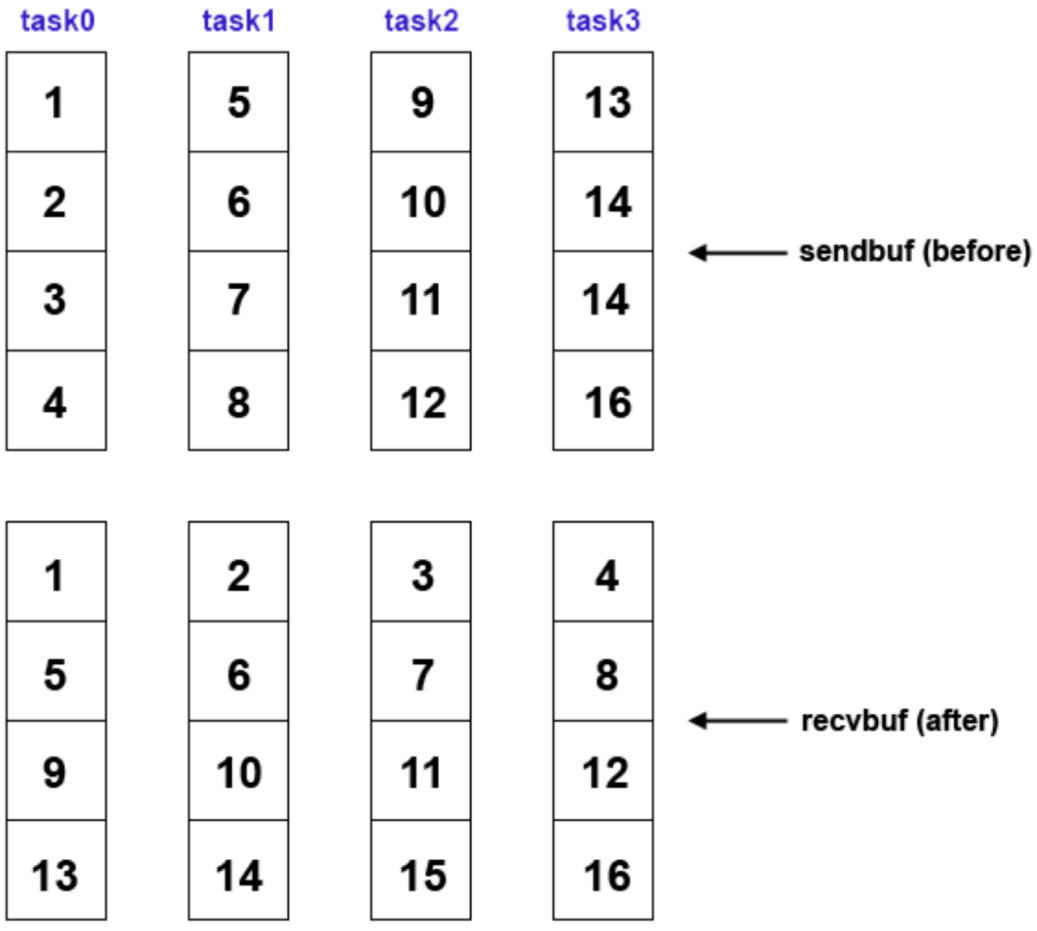
```
sendcnt = 1;
recvcnt = 1;
MPI_Allgather(sendbuf, sendcnt, MPI_INT
              recvbuf, recvcnt, MPI_INT
              MPI_COMM_WORLD);
```



Alltoall: example

Scatter data from all tasks to all tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT
             recvbuf, recvcnt, MPI_INT
             MPI_COMM_WORLD);
```

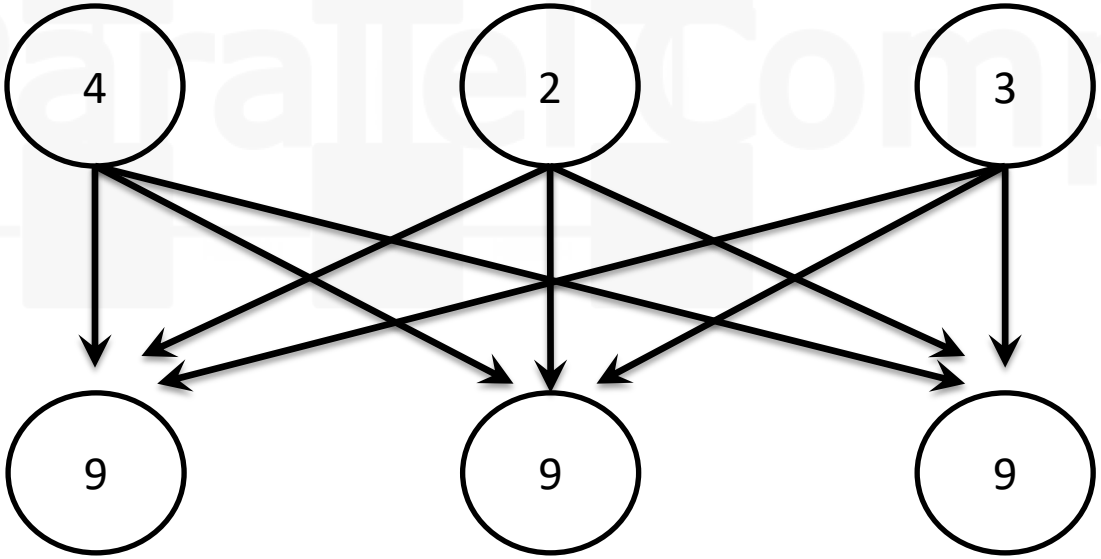


Collective communication

Intermediate Methods (II)

```
MPI::Comm::Allreduce( void* sendbuf, int sendcount,
                      MPI::Datatype& sendtype,
                      void* recvbuf, int recvcount,
                      MPI::Datatype& recvtype)
```

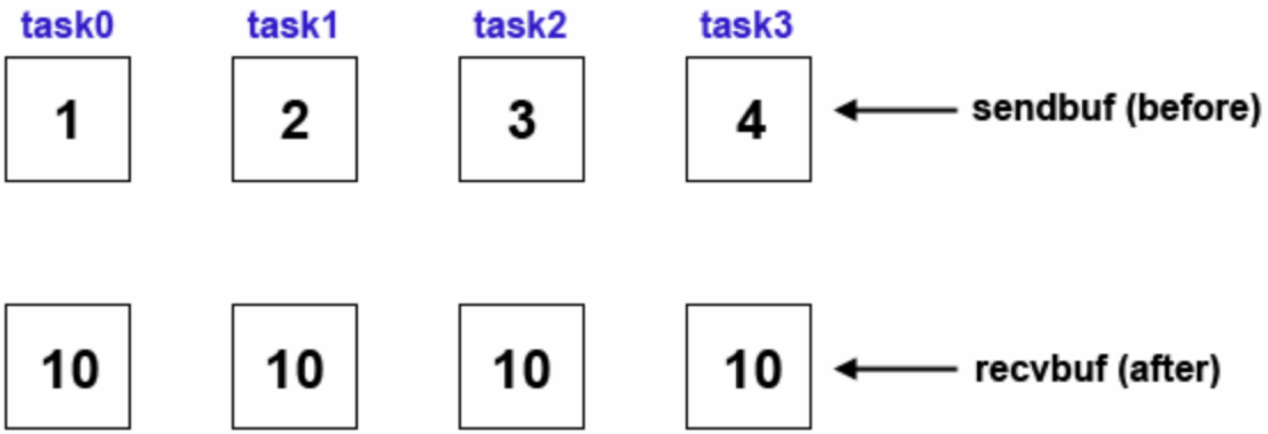
- Combines values from all processes and distributes the result back to all processes (there is no *root* among parameters).
- ALLREDUCE is the equivalent of doing REDUCE followed by an BCAST



Allreduce: example

Perform reduction and store result across all tasks in communicator

```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT,
              MPI_SUM, MPI_COMM_WORLD);
```



MPI data types

- For reasons of portability, MPI predefines its elementary data types
- MPI also supports user-defined data structures built upon MPI primitive data types
  - These user-defined data structures are called derived data types
  - Derived data types are created through dedicated MPI routines at runtime
  - Derived data type objects have MPI_Datatype as their type
  - Derived data types allow one to specify non-contiguous data and to treat it as it was contiguous

Derived data types methods

- How can we do this in MPI?

```
int B[2][3];
```

```
struct {
    int num;
    float x;
    double data[2];
} obj;
```

- MPI provides several methods for constructing derived data types:
  - Contiguous
  - Vector
  - Indexed
  - Struct

- How to Use?

  Before you can use a derived datatype, you must create it. Here are the steps you take:

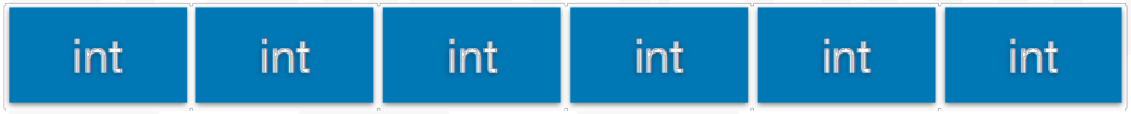
  1. Construct the datatype.
  2. Allocate the datatype.
  3. Use the datatype.
  4. Deallocate the datatype.

Derived data types methods

```
Newtype Oldtype.Create_contiguous(int count)
```

- Creates a new type that consists of the concatenation of count copies of OLDTYPE
- Allows replication of a datatype into contiguous locations

```
int B[2][3];
MPI::Datatype Matrix = MPI::INT.Create_contiguous(6);
```



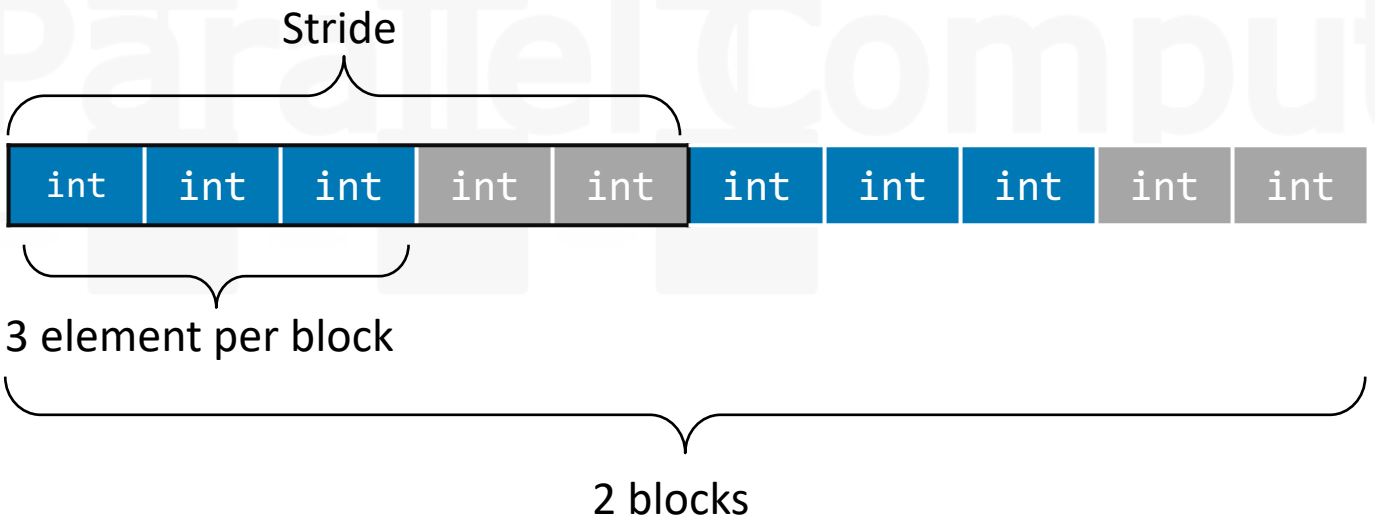

```
. . .

MPI::COMM_WORLD.Send(B, 1, Matrix, dest, tag);
```

Derived data types methods

```
Newtype Oldtype.Create_vector(int count, int blocklength, int stride)
```

- Creates a new type that consists of count blocks
- Each block is the concatenation of blocklength copies of OLDTYPE
- The number of elements between the start of each block is indicated by stride

```
MPI::INT.Create_vector(count=2, blocklength=3, stride=5);
```



Derived data types methods

- How is this useful?

A =

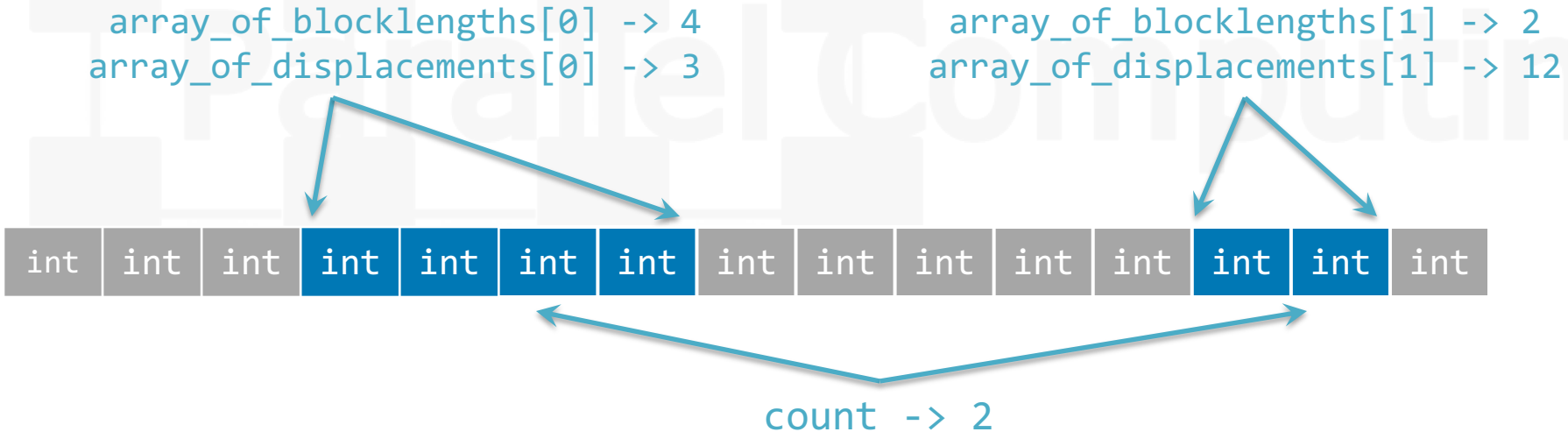
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

```
                                                ROWS        N_OF_CELLS          COLUMNS
MPI::Datatype Column = MPI::INT.Create_vector(count=4, blocklength=1, stride=4);

Comm.Send(&A[0][1], 1, column, source, tag);
```

Derived data types methods

```
Newtype Oldtype::Create_indexed(int count,
                    int array_of_blocklengths[],
                            int array_of_displacements[])
```

- Creates a new type that takes COUNT blocks of different lengths and at different offsets
- It allows one to specify data types where offsets between successive blocks are not equal



Derived data types methods

```
MPI::Datatype MPI::Datatype::Create_struct(
                        int count,
                        int array_of_blocklengths[],
                        MPI::Aint array_of_displacements[],
                        MPI::Datatype array_of_types[])
```

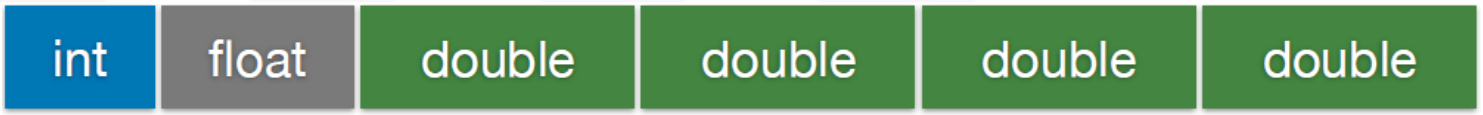
- Creates a new type that consists of a C/C++ struct, by specifying a complete map of the component data types

Count	Number of blocks. Also number of entries in arrays ARRAY_OF_TYPES, ARRAY_OF_DISPLACEMENTS, and ARRAY_OF_BLOCKLENGTHS
array_of_blocklengths	Number of elements in each block (array of integers)
array_of_displacements	Byte displacement of each block (array of MPI::Aint).
array_of_types	Type of elements in each block (array of handles to data-type objects).

Derived data types methods example:

```
struct {
    int num;
    float x;
    double data[4];
} obj;
```

Count	3
array_of_blocklengths	{ 1, 1, 4 }
array_of_displacements	{ 0, sizeof(int), sizeof(int) + sizeof(float) }
array_of_types	{ MPI::INT, MPI::FLOAT, MPI::DOUBLE }



Derived data types utility methods

```
MPI::Datatype.Get_extent(MPI::Aint& lb, MPI::Aint& extent)
```

- Returns the lower bound and extent of a data type
- Usage: think C/C++ sizeof()

```
MPI::Datatype.Commit()
```

- Commits the new data type to the system, which is required for all derived data types

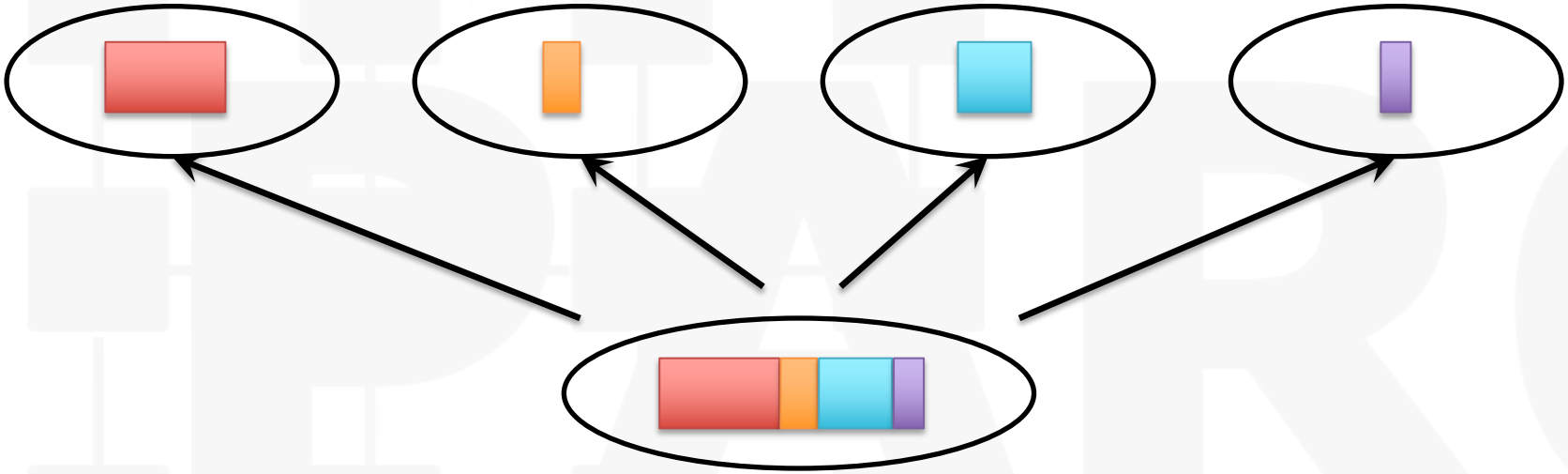
```
MPI::Datatype.free()
```

- Deallocates the specified data type object

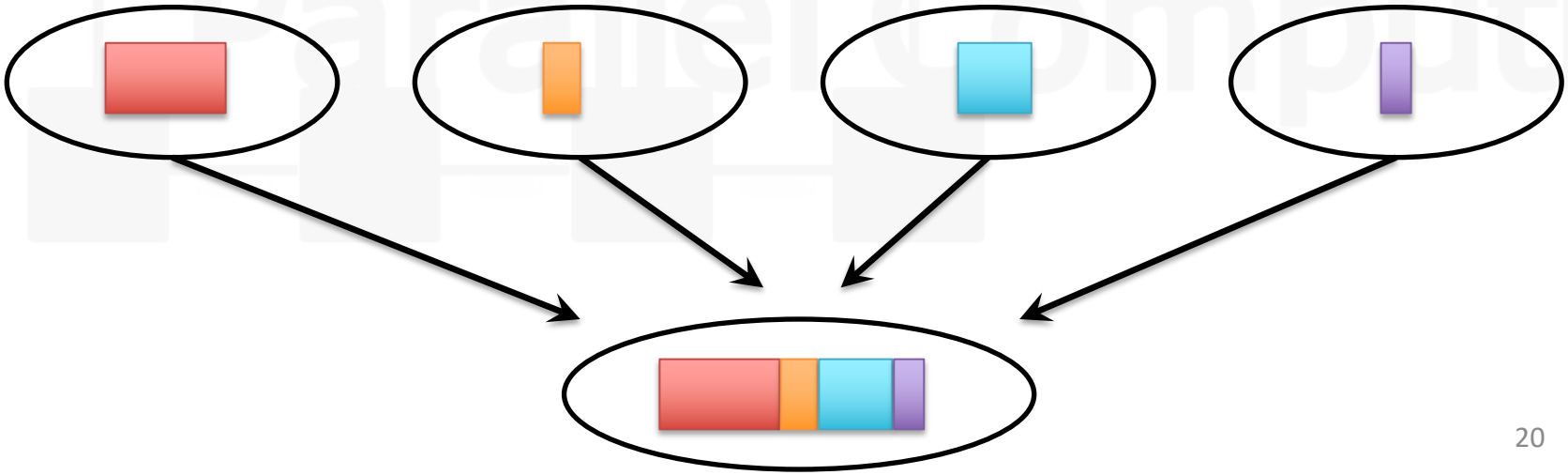
Collective Communication
Advanced Method (I)

- There are also extended MPI collectives: **vector operations**
  - A vector of counts instead of a single count
  - One count for each process
  - The scalar counts may all be different
- Four Collective Communication:
  - Scatter**v**
  - Gather**v**
  - All-gather**v**
  - All-to-all**v**
- Generally, use collectives wherever possible
  - Provide most opportunity for implementation tuning

Scatterv



Gatherv



Collective Communication
Advanced Method (II)

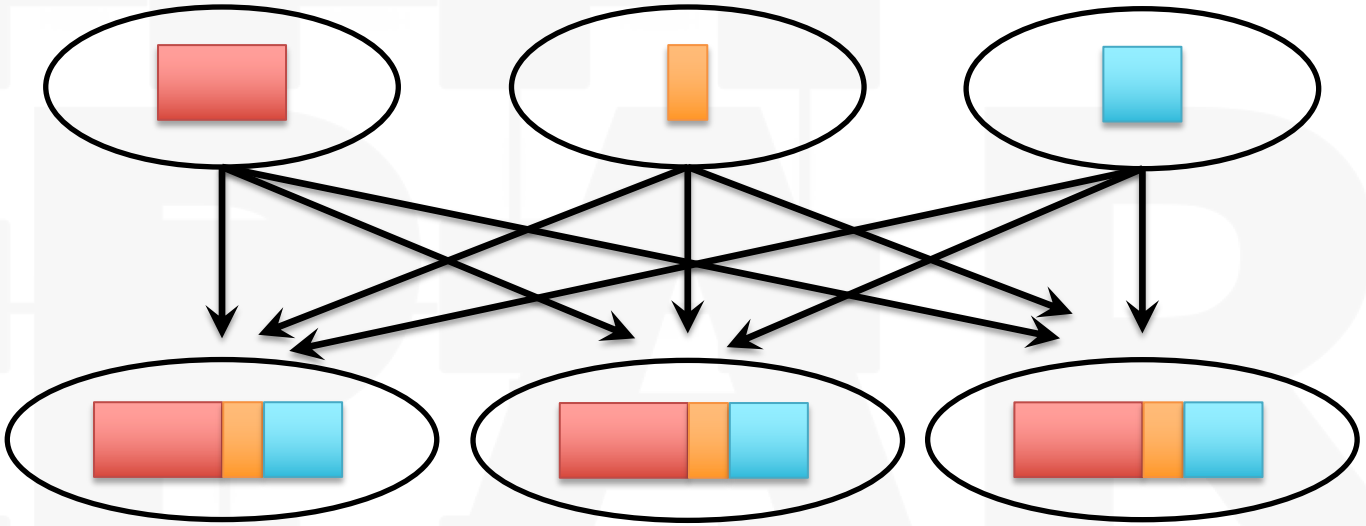
```
MPI::Comm::Scatterv( void* sendbuf,
          int array_of_blocklengths[], //this is sendcount in scatter()
          int array_of_displacements[],
          MPI::Datatype& sendtype,
          void* recvbuf, int recvcount,
          MPI::Datatype& recvtype, int root)
```

- Extends the functionality of MPI_SCATTER by allowing a varying count of data to be sent to each process, since SENDCOUNT is now an array

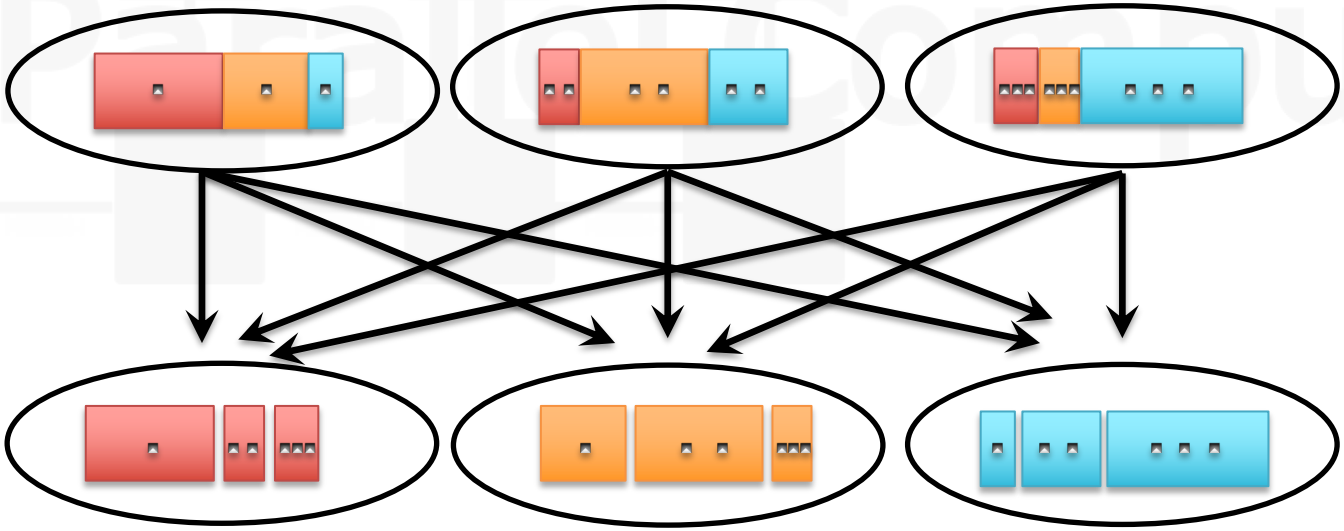
```
MPI::Comm::Gatherv( void* sendbuf, int sendcount
            MPI::Datatype& sendtype,
            void* recvbuf,
            int array_of_blocklengths[], //this is recvcount in gather()
            int array_of_displacements[],
            MPI::Datatype& recvtype, int root)
```

- Extends the functionality of MPI_GATHER by allowing a varying count of data from each process, since RECVCOUNT is now an array.

All-Gatherv



All-to-Allv



Collective Communication
Advanced Method (I)

```
MPI::COMM::Allgatherv( void* sendbuf, int sendcount,
            MPI::Datatype& sendtype,
            void* recvbuf,
                 int recvcounts[],
            int displs[],
            MPI::Datatype& recvtype)
```

- Is similar to MPI_ALLGATHER in that all processes gather data from all other processes, except that each process can send a different amount of data

```
MPI::COMM::Alltoallv( void* sendbuf,
                      int sendcounts[],
                      int displs[],
                      MPI::Datatype& sendtype,
                      void* recvbuf,
                      int recvcounts[],
                      int rdispls[],
                      MPI::Datatype& recvtype)
```

- Is a generalized collective operation in which all processes send data to and receive data from all other processes. It adds flexibility to MPI_ALLTOALL by allowing the user to specify data to send and receive vector-style

Advanced MPI

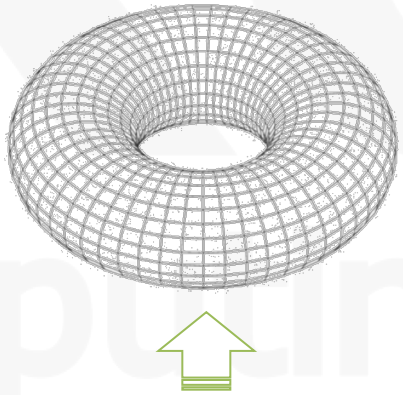
Network Topology

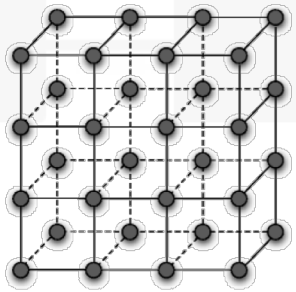
- Intra-communicator : a collection of processes that can send messages to each other and engage in collective communication operations.
- Inter-communicator: are used for sending messages between processes belonging to disjoint intra-communicators

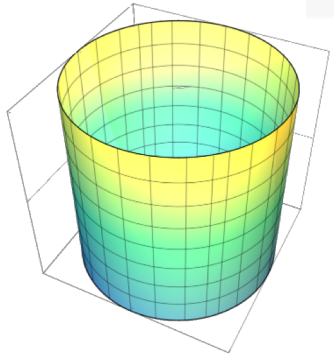
Cartesian topologies

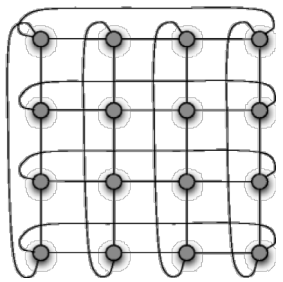
- Each process is connected to its neighbors in a virtual grid
- Boundaries can be cyclic
- Processes can be identified by cartesian coordinates
- Three topologies:
  - 2D Grid/Mesh
  - Cylinder
  - 2D Torus

Graph Topology (user defined)









MPI performance factors (I)

- Platform/architecture related
  - CPU: clock speed, number of cores
  - Memory: memory and cache configuration
  - Network: latency, bandwitch
- Application related
  - Algorithm efficiency and scalability
  - Communication-to-computation ratios
    - I/O operations
    - Message size used
  - Load balance
  - Memory usage patterns
  - Types of MPI routines used (i.e., blocking, non-blocking, collective communication)

MPI performance factors (II)

- MPI implementation related
  - Message buffering
    - Message buffering concerns the storage of data between the time a send operation begins and when the matching receive operation completes
    - Buffer space can be provided by the system, or by the user
  - Message passing protocols
    - Eager
    - Rendezvous
  - Sender-receiver synchronization
    - Polling
    - Interrupt
  - Routine internals

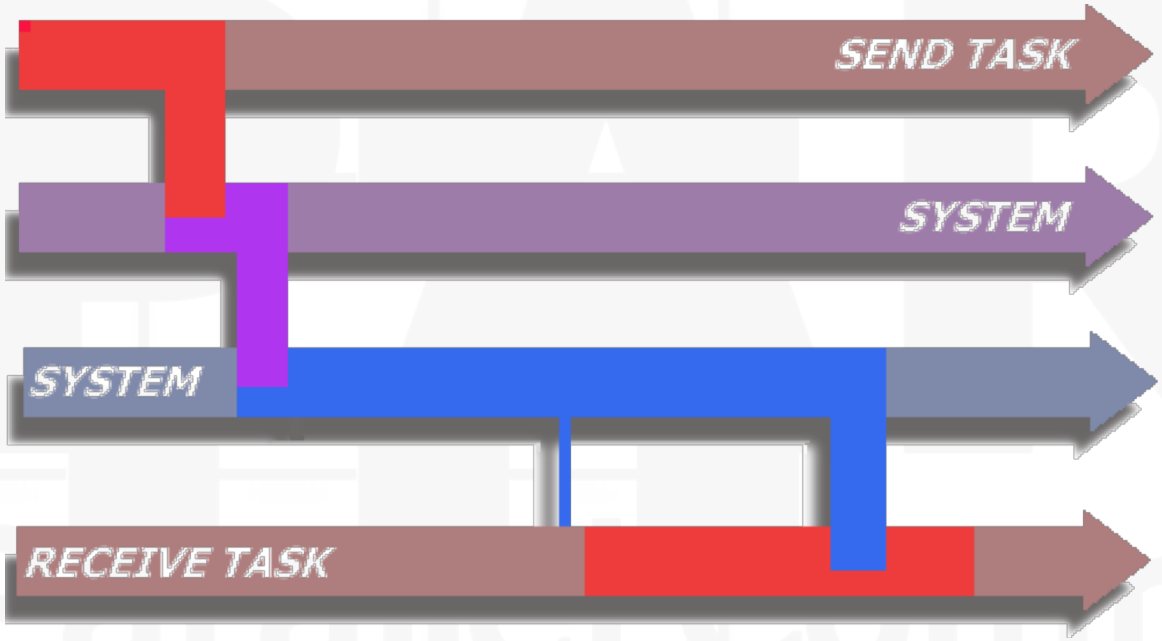
Message passing protocols

- A message passing protocol describes the internal methods and policies an MPI implementation employs to accomplish message delivery
  - Eager protocol
    - Asynchronous protocol that allows a send operation to complete without acknowledgment from a matching receive
  - Rendezvous protocol
    - Synchronous protocol which requires an acknowledgment from a matching receive in order for the send operation to complete

Sender-Receiver synchronization

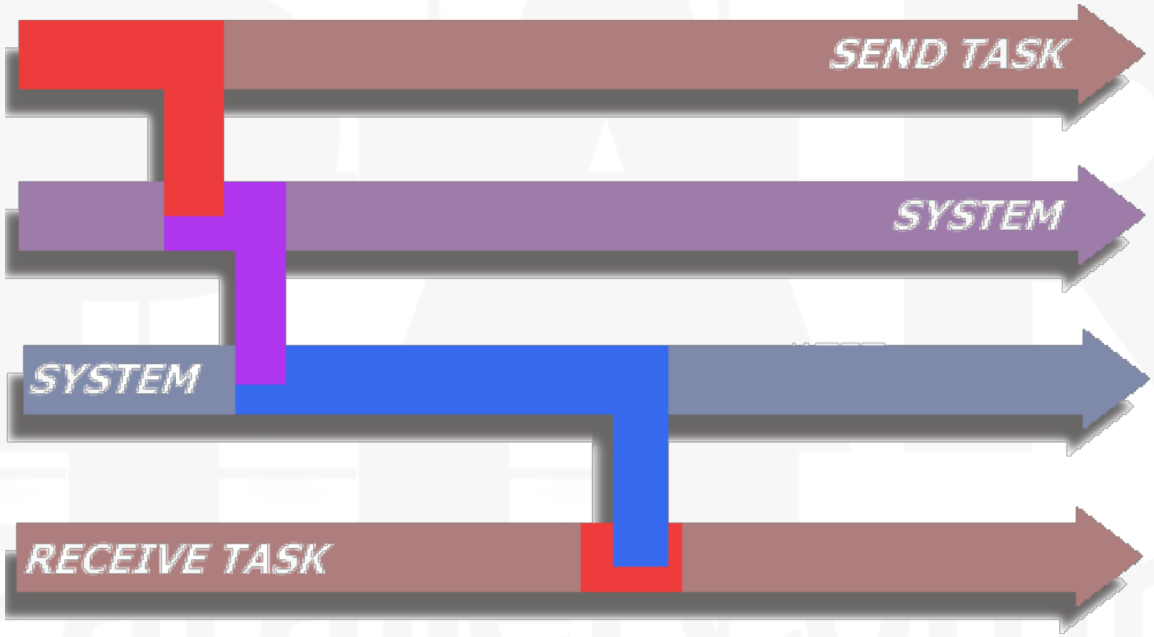
- The MPI standard does not define how this should be accomplished
  - How does a receive task know if a task is requesting to send?
  - How often should a receive task check for incoming messages?
- MPI implementations typically rely on two synchronization modes
  - Polling
  - Interrupt

Polling



- The user MPI task will be interrupted by the system to check for communication events at regular intervals
- If a communication event occurs while the user task is busy, it must wait

Interrupt



- The user MPI task will be interrupted by the system for communication events when they occur

Message size

- Message size can be a very significant contributor to MPI application performance
- In most cases, increasing the message size improves performance
- Performance can often improve significantly within a relatively small range of message sizes
- It is usually worth experimenting with different message size (if possible) to determine the best solution

Conclusions (I)

- MPI allows for explicit control over communication
  - High efficiency due to overlapping computation and communication
- MPI scales well to very large number of processors
- MPI is portable, and its current implementation are efficient and optimized
- However application development is quite difficult and time-consuming
  - Extensive modifications to the serial code are required
- Dynamic load balancing is difficult to implement

Conclusions (II)

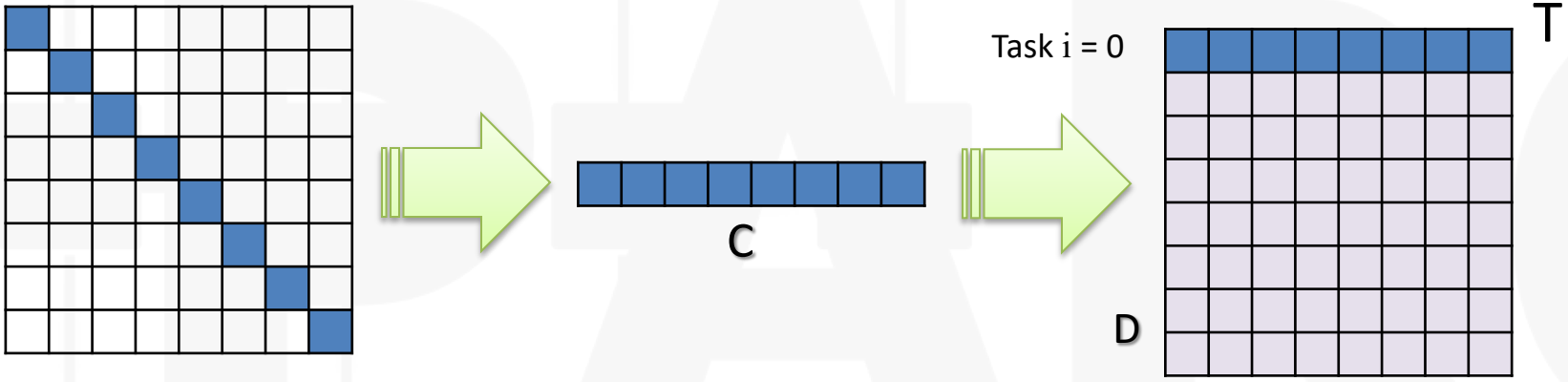
- MPI is most effective for problems with coarse-grained parallelism
  - The problem decomposes into relatively independent sub-problems
  - Communication between tasks is minimized
- It is less effective with fine-grained problems
  - Communication costs may dominate

Examples

- Test some methods for constructing derived data types:
  - Contiguous
  - Vector
  - Indexed
  - Struct

Exercises (I)

Matrix Diagonal and transpose



What is the speedup again the sequential version for large values?

1) N is a multiple of the number of tasks
2) The root has two NxN matricies A, B
3) The matrix rows of A and the columns of B are scattered to other tasks
4) Each task computes the dot product between own rows and columns to compute the diagonal elements
5) Each task takes the whole result C
6) Each task i computes i * C in the matrix D
7) Each task takes the transpose of C without local computation

Exercises (II)

Advanced Collective Communication

Work Distribution:

1) The root generates an array of N random values

2) The root sends to tasks i: $\frac{i+1}{\left(\frac{|task|^2+|task|}{2}\right)} \cdot N$ values

3) Each task compute: $values \bmod (3)$

4) For each value that the predicates is true, the task send back these values to the root

References

- Official MPI specifications

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

- Good API References:
  - OpenMPI

    https://www.open-mpi.org/doc/v1.8/

  - Microsoft MPI References

    https://msdn.microsoft.com/en-us/library/dn473458%28v=vs.85%29.aspx

- Tutorials

  http://mpitutorial.com/

  https://computing.llnl.gov/tutorials/mpi/