Parallel Scientific Computation

MPI 4

J.-H. Parq
IPCST
Seoul National University

Persistent P2P Communication

- Non-blocking pipeline
- Advantage
  - Reduction of communication overheads

- Command order
  1. MPI_Send_init & MPI_Recv_init
  2. MPI_Start / MPI_Startall
  3. MPI_Wait / MPI_Waitall
  4. MPI_Request_free

Persistent P2P Communication

- Example (Fortran)

```
Call MPI_SEND_INIT(sbuf, count, dtype, next, tag, comm, req(1), ier)
Call MPI_RECV_INIT(rbuf, count, dtype, prev, tag, comm, req(2), ier)
……
Do
  Call MPI_STARTALL(2, req, ier)
  ! or Call MPI_START(req(2), ier) & Call MPI_START(req(1), ier)
……
  Call MPI_WAITALL(2, req, statuses, ier)
……
End Do
Call MPI_REQUEST_FREE(req, ier)
```

Derived Datatypes

- You can make your own MPI datatype.
  - Size: Length of actual data
  - Extent: Length of occupied memory (rounded)

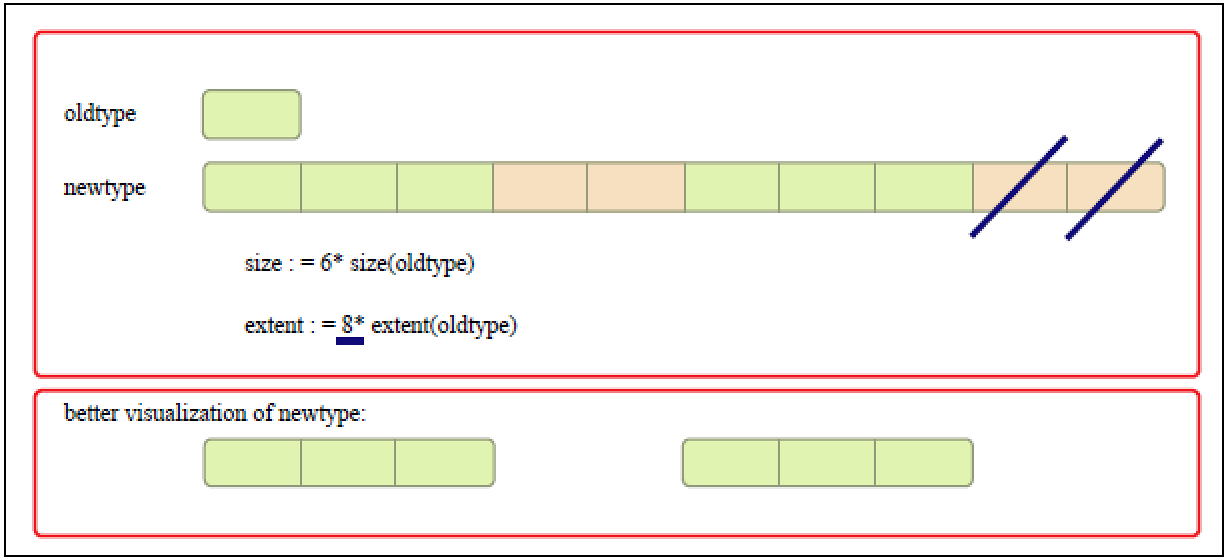


Figure by MIT OpenCourseWare.

Derived Datatypes

- You can make your own MPI datatype.
  - Contiguous: Length-fixed 1-D array (size = extent)
  - Vector: Elements are separated by increasing the stride (extent) of old datatype as a unit
  - Hvector: Similar to vector but using bytes instead of old datatype unit
  - Indexed: Strides can differ element by element
  - Hindexed: Similar to indexed but using bytes instead of old datatype unit
  - Struct: Generalized creation of a new datatype

Derived Datatypes

- Contiguous
  - MPI_TYPE_CONTIGUOUS(count, oldtype, newtype, ierror)
  - int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
- Indexed
  - int MPI_Type_indexed(int count, int *blocklength_array, int *displacement_array, MPI_Datatype oldtype, MPI_Datatype *newtype)
  - Similar structure for Fortran
- Hindexed
  - int MPI_Type_hindexed(int count, int *blocklength_array, MPI_Aint *displacement_array, MPI_Datatype oldtype, MPI_Datatype *newtype)
  - Similar structure for Fortran

Derived Datatypes

- Vector
  - MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype, ierror)
  - int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

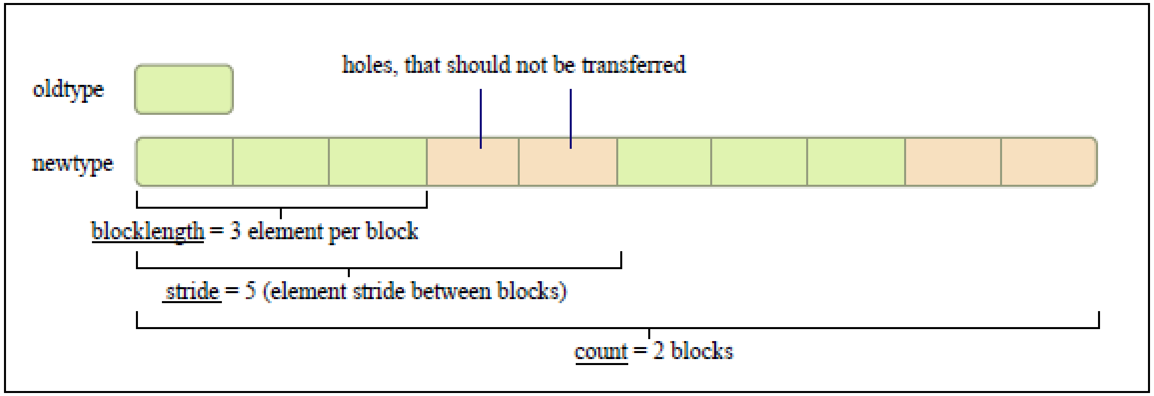


Figure by MIT OpenCourseWare.

Derived Datatypes

- Hvector
  - int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
  - Similar structure for Fortran
- Struct
  - C: structure type
  - Fortran: common block or derived type
  - MPI_TYPE_STRUCT(count, blocklength_array, displacement_array, type_array, newtype, ierror)
  - int MPI_Type_struct(int count, int *blocklength_array, MPI_Aint *displacement_array, MPI_Datatype *type_array, MPI_Datatype *newtype)

Derived Datatypes

- Procedure
  1. Construction: defining
     - MPI_Type_contiguous / MPI_Type_vector / MPI_Type_hvector / MPI_Type_indexed / MPI_Type_hindexed / MPI_Type_struct
  2. Commission to the system
     - MPI_Type_commit(newtype)
  3. Use of the new datatype
  4. Destruction: cleaning
     - MPI_Type_free(newtype)

Derived Datatypes

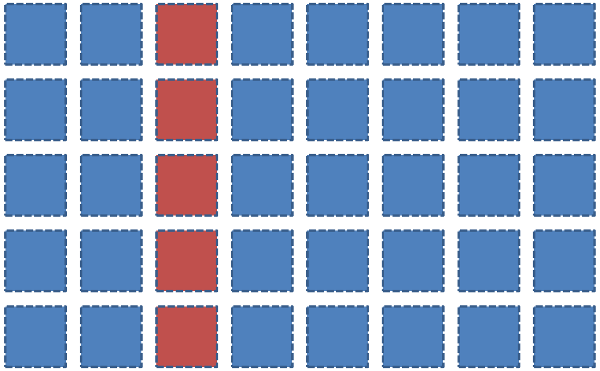
- Example (Fortran)

```
Integer, parameter :: m = 5, n = 3
Double precision  matrix(m,n)
……
Call MPI_TYPE_CONTIGUOUS(m*n, MPI_DOUBLE_PRECISON, oneD, ier)
Call MPI_TYPE_COMMIT(oneD, ier)
……
If (rank == source) Then
  Call MPI_SEND(matrix, 1, oneD, dest, tag, comm, ier)
End If
……
Call MPI_TYPE_FREE(oneD, ier)
```

Derived Datatypes

- Example (C)

```
int col;  double A[5][8];
MPI_Type_vector(5, 1, 8, MPI_DOUBLE, &coltype);
MPI_Type_commit(&coltype);
……
  MPI_Send(&A[0][col], 1, coltype, receiver, tag, comm);
```



Derived Datatypes

- Example (Fortran)

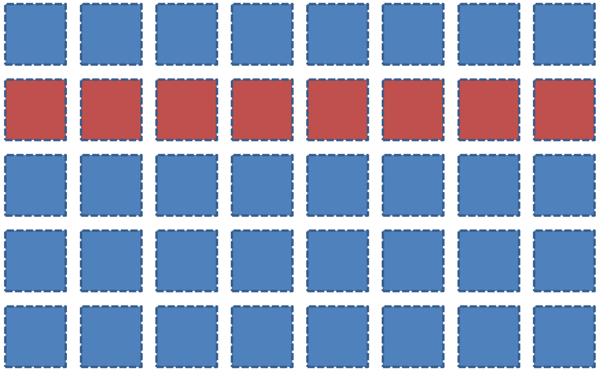
Double precision A(5,8)

Call MPI_TYPE_VECTOR(8,1,5, MPI_DOUBLE_PRECISION, rowtype, ier)

Call MPI_TYPE_COMMIT(rowtype, ier)

......

Call MPI_SEND(A(row,1), 1, rowtype, receiver, tag, comm, ier)



Derived Datatypes

- Example (C)

```
struct {
  int num;    float x;   double y[10];
} data;
int blocklength[3] = {1, 1, 10};
MPI_Datatype newstruct, types[3] = {MPI_INT, MPI_FLOAT, MPI_DOUBLE};
MPI_Aint intext, floatext, displace[3];
……
MPI_Type_extent(MPI_INT, &intext);
MPI_Type_extent(MPI_FLOAT, &floatext);
displace[0] = (MPI_Aint)0; displace[1] = intext;
displace[2] = intext + floatext;
MPI_Type_struct(3, blocklengths, displace, types, &newstruct);
```

You can use MPI_Type_contiguous if these are of the same type.

Derived Datatypes

- Example (Fortran)

```
Integer num
Real x
Double precision y(10)
Common /data/ num, x, y
integer blocklength(3)
Data blocklength /1, 1, 10/
Integer displace(3), types(3), newstruct, intext, realext
……
Call MPI_Type_extent(MPI_INTEGER, intext, ier)
Call MPI_Type_extent(MPI_REAL, realext, ier)
displace(1) = 0
……
Call MPI_Type_struct(3, blocklengths, displace, types, newstruct, ier)
```

Derived Datatypes

- Finding the size (in bytes)
  - MPI_TYPE_SIZE(datatype, size, ierror)
  - int MPI_Type_size(MPI_Datatype datatype, MPI_Aint *size)
- Finding the extent (in bytes)
  - MPI_TYPE_EXTENT(datatype, extent, ierror)
  - int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
  - You can use MPI_Address instead.
    - See also the constant MPI_BOTTOM
  - MPI_TYPE_EXTENT and MPI_Address can be used for dynamically allocated arrays.

Derived Datatypes

- In MPI-2 or MPI-3,
  - MPI_Type_extent → MPI_Type_get_extent
  - MPI_Address → MPI_Get_address
  - MPI_Type_hvector → MPI_Type_create_hvector
  - MPI_Type_hindexed → MPI_Type_create_hindexed
  - MPI_Type_struct → MPI_Type_create_struct

Derived Datatypes

- In MPI-2 or MPI-3,
  - MPI_Type_create_resized
  - MPI_Type_create_indexed_block
  - MPI_Type_get_true_extent
  - MPI_Type_create_subarray
  - ……

  ❖ Find more at OpenMPI or MPICH homepage

Counting

- MPI_Get_count
  - Finding the count from the status
  - Can be used for preparing the receiver
  - int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
- Example (Fortran)

```
Call MPI_PROBE(source, tag, comm, status, ier)
Call MPI_GET_COUNT(status, newtype, number, ier)
……
Call MPI_RECV(buffer, number, newtype, source, tag, comm, status, ier)
```

Network Architectures

- Speedup by MPI depends on the network.
  - Network speed is the essential factor of the communication speed.
- Old network architectures produce bottlenecks due to bandwidth $\lesssim$ 1 Gbit/sec
- Networks for high-performance computing
  - Infiniband (IB)
  - High-speed ethernet (HSE)
  - ……

Network Architectures

- Infiniband
  - Dominant in the top500 supercomputers now
  - High stability. High availability.
  - **Bandwidth: up to 600 Gbps for 12 links (50 Gbps per link) now**
  - Low latency
  - Switch-based serial I/O
  - Switched fabric topology
- High-speed ethernet
  - Excellent for small data transfer
  - Easy management
  - Bandwidth: up to order of 400 Gbps now
  - Relatively simple structure. Hierarchical topology.

Network Architectures

- Tofu interconnect D
  - Torus fusion
  - For supercomputers
- Aries, Slingshot
  - For supercomputers
  - Dragonfly topology
- Omni-path
  - Bandwidth up to 400 Gbps (100 Gbps per port).
  - Low latency.
- NUMAlink
  - ccNUMA

References

- W. Gropp, E. Lusk, and A. Skjellum,
  Using MPI
- C. Evangelinos,
  Parallel Programming for Multicore Machines Using OpenMP and MPI
- Bogdan Pasca,
  Derived Datatypes
- TOP500 Supercomputer Sites (www.top500.org)