CSE 6220/CX 4220
Introduction to HPC

Lecture 16: MPI Topologies and Datatypes

Helen Xu
hxu615@gatech.edu

Slide credits to Prof. Srinivas Aluru

MPI Communicators

Recall that:

- MPI processes are grouped into a communicator
- Each process within a communicator has a unique rank
- Ranks are integers from 0 to $(p - 1)$, where p is the total number of processes in the communicator
- Initially all processes belong to MPI_COMM_WORLD

Sometimes (when?) it may be desirable to create a custom communicator...

Communicator Constructors

MPI provides functions to create a new communicator from an existing one:

- `int MPI_Comm_dup(MPI_Comm comm, MPI_Comm* newcomm);`
- `int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm* newcomm);`

Communicators can be released when no longer needed:

- `int MPI_Comm_free(MPI_Comm* comm);`

Recall: Communicator Split Example

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

int main(int argc, char* argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int color = rank % 2;
    MPI_Comm comm;

    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &comm);
    int S = 0;
    MPI_Reduce(&rank, &S, 1, MPI_INT, MPI_SUM, 0, comm);
    MPI_Comm_free(&comm);

    std::cout << rank << " " << S << std::endl;
    return MPI_Finalize();
}
```

New communicator

Reduce on even and odd

Recall: Communicator Split Example

MPI_WORLD = {P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7}

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
rank	0	1	2	3	4	5	6	7
color	0	1	0	1	0	1	0	1
key	0	1	2	3	4	5	6	7

comm = {P_0, P_2, P_4, P_6} or { P_1, P_3, P_5, P_7}

	P_0	P_2	P_4	P_6	P_1	P_3	P_5	P_7
rank	0	1	2	3	0	1	2	3

From Srinivas Aluru

Communicator Split Example

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

int main(int argc, char* argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int color = rank % 2;
    MPI_Comm comm;

    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &comm);
    int S = 0;
    MPI_Reduce(&rank, &S, 1, MPI_INT, MPI_SUM, 0, comm);
    MPI_Comm_free(&comm);

    std::cout << rank << " " << S << std::endl;
    return MPI_Finalize();
}
```

```
5 0
2 0
7 0
3 0
6 0
4 0
0 12
1 16
```

What is the output for p

Other Default Communicators

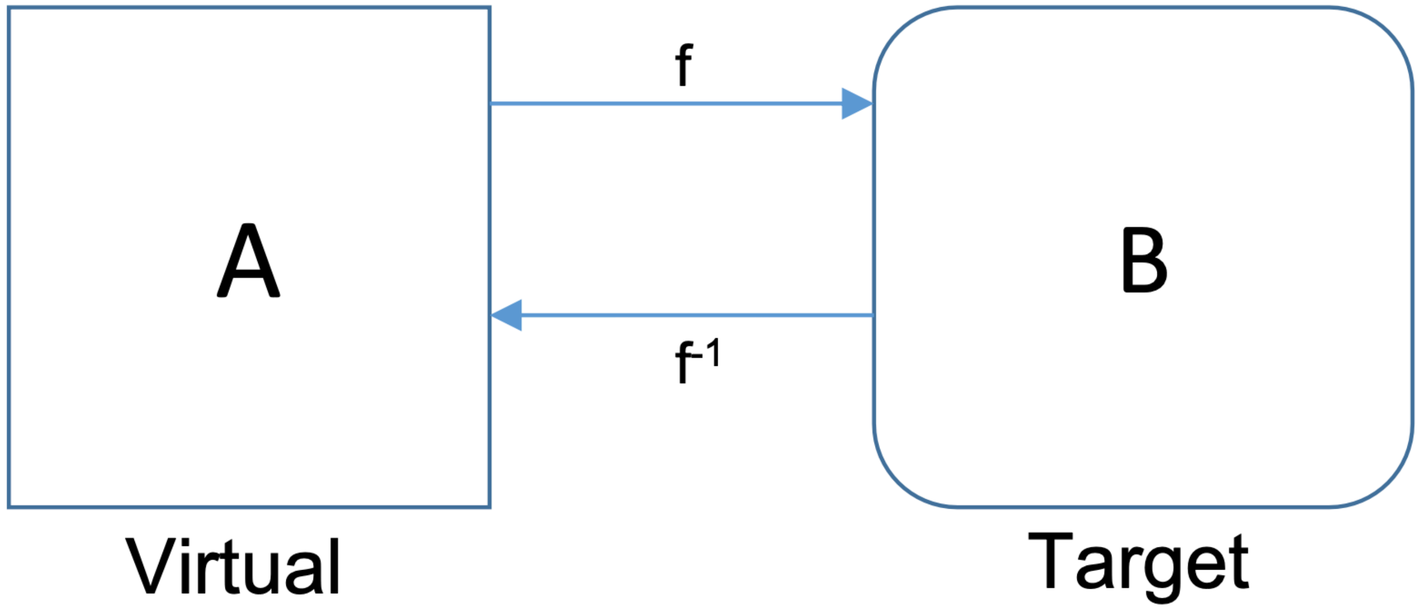
- Communicator constructors are collective operations
- MPI provides two additional predefined communicators `MPI_COMM_NULL` and `MPI_COMM_SELF`

If you want to avoid some ranks participating in collective operations

Virtual Topologies

- MPI provides a mechanism to describe **logical organization of processors**
- Such virtual topologies are **independent from actual physical topology**
- MPI implementation may, but does not have to, use virtual topologies to assign MPI processes to physical processors
- Regardless of actual physical mapping, in many cases **virtual topologies simplify implementation of algorithms**
- Virtual topologies are represented by MPI communicators (which internally store information about topology)

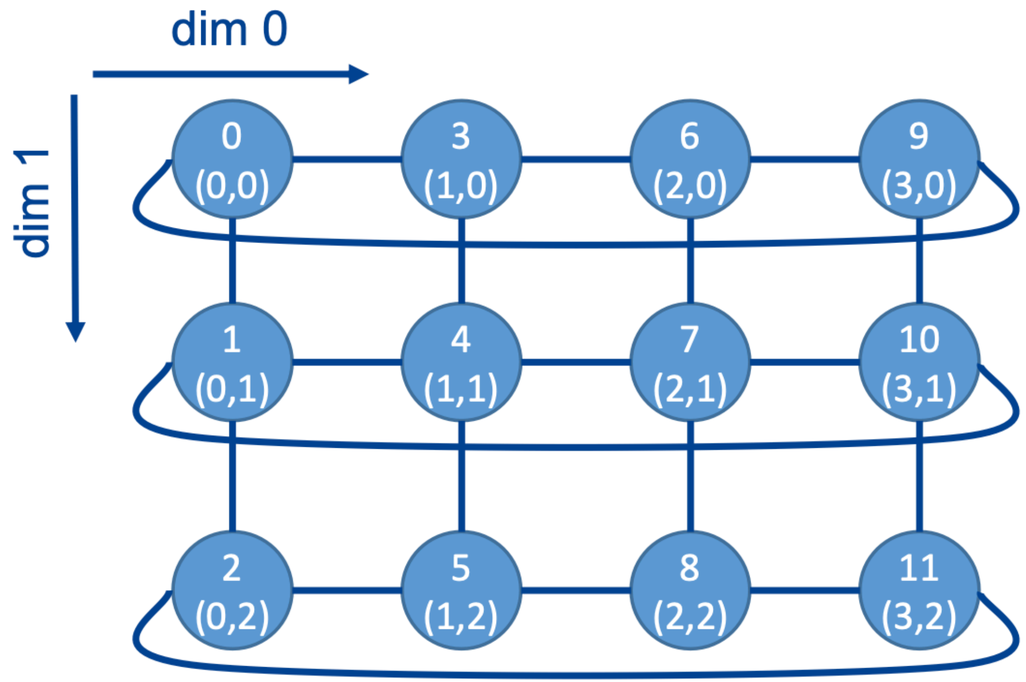
Embedding



- Algorithm designed for “Virtual” network. The system you have is “Target” network.
- We will call them Virtual Topology and Physical Topology
- Virtual and Physical Topologies can be modeled as graphs.

Cartesian Topology

- MPI provides specialized constructors for **Cartesian topologies**
- In general MPI vendors provide optimized constructors for specific hardware (e.g., IBM BlueGene)



Cartesian Topology

- Cartesian topology with `ndims` dimensions:

num. dimensions

dimension lengths

```
int MPI_Cart_create(MPI_Comm comm, int ndims, int* dims,
                    int* periods, int reorder,
                    MPI_Comm* newcomm);
```

array or ring in dimension

output

can MPI reorder ranks in assignment?

- How to find how many nodes to put in any dimension?

- MPI defines a function to find “optimal” dimensions (including user defined constraints):

output

```
int MPI_Dims_create(int nnodes, int ndims, int* dims);
```

input

input

Cartesian Topology

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

int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int dims[3] = { 0, 0, 0 };
    // dims[1] = 4;
    MPI_Dims_create(size, 3, dims);
    if (rank == 0) {
        std::cout << dims[0] << " " << dims[1]
                  << " " << dims[2] << std::endl;
    }
    return MPI_Finalize();
}
```

MPI gives dimension lengths given the total number of ranks

Questions from last time

Does making a new communicator overwrite the old one?
- short answer - no
- technically, the documentation recommends that libraries should use duplicated version of `MPI_COMM_WORLD` rather than using it directly

Can ranks be the same in comm split?
- yes, they need different colors

Cartesian Topology

- In Cartesian topology each processor is mapped into `ndims` dimensional space
- MPI provides functions to get coordinates of a processor:

```
int MPI_Cart_rank(MPI_Comm comm, int* coords, int* rank);
```

Determine process rank given Cartesian location

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int ndims,
                    int* coords);
```

Determine process coords in Cartesian topology given the rank in group

Cartesian Topology

- In Cartesian topologies shifting (via simultaneous send and receive) is a frequent operation:

dimension in which the shift will be made

Number of units to virtually move the topology

```
int MPI_Cart_shift(MPI_Comm comm, int dim, int disp,
                   int* srank, int* drank);
```

Output: store the source rank that would be sending to this rank

Output: store the dest rank that would be sending to this rank

- This function does not perform actual data transfer!

```
int dims[2] = {0, 0};
MPI_Dims_create(size, 2, dims);

// Make both dimensions non-periodic
int periods[2] = {false, false};
int reorder = true;

// Create a communicator with a cartesian topology.
MPI_Comm new_communicator;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, new_communicator);

// Declare our neighbours
enum DIRECTIONS {DOWN, UP, LEFT, RIGHT};
int neighbours_ranks[4];

// Let consider dims[0] = X, so the shift tells us our left and right neighbours
MPI_Cart_shift(new_communicator, 0, 1, &neighbours_ranks[LEFT], neighbours_ranks[RIGHT]);

// Let consider dims[1] = Y, so the shift tells us our up and down neighbours
MPI_Cart_shift(new_communicator, 1, 1, &neighbours_ranks[DOWN], neighbours_ranks[UP]);
```

https://rookiehpc.org/mpi/docs/mpi_cart_shift/index.html

Recall: `MPI_Sendrecv`

```
int MPI_Sendrecv(const void* buffer_send,
                 int count_send,
                 MPI_Datatype datatype_send,
                 int recipient,
                 int tag_send,
                 void* buffer_recv,
                 int count_recv,
                 MPI_Datatype datatype_recv,
                 int sender,
                 int tag_recv,
                 MPI_Comm communicator,
                 MPI_Status* status);
```

- `MPI_Sendrecv` is a combination of an `MPI_send` and `MPI_Recv`.
- It can be seen as both subroutines executing concurrently.
- The buffers for send and receive must be different.

https://rookiehpc.org/mpi/docs/mpi_sendrecv/index.html

Recall: MPI_Sendrecv Example

```
// NOT SHOWN - INIT AND MAKE SURE YOU HAVE TWO PROCESSES

    // Prepare parameters
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    int buffer_send = (my_rank == 0) ? 12345 : 67890;
    int buffer_recv;
    int tag_send = 0;
    int tag_recv = tag_send;
    int peer = (my_rank == 0) ? 1 : 0;

    // Issue the send + receive at the same time
    printf("MPI process %d sends value %d to MPI process %d.\n", my_rank, buffer_send, peer);
    MPI_Sendrecv(&buffer_send, 1, MPI_INT, peer, tag_send,
                 &buffer_recv, 1, MPI_INT, peer, tag_recv, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("MPI process %d received value %d from MPI process %d.\n", my_rank, buffer_recv,
peer);

// NOT SHOWN - FINALIZE
```

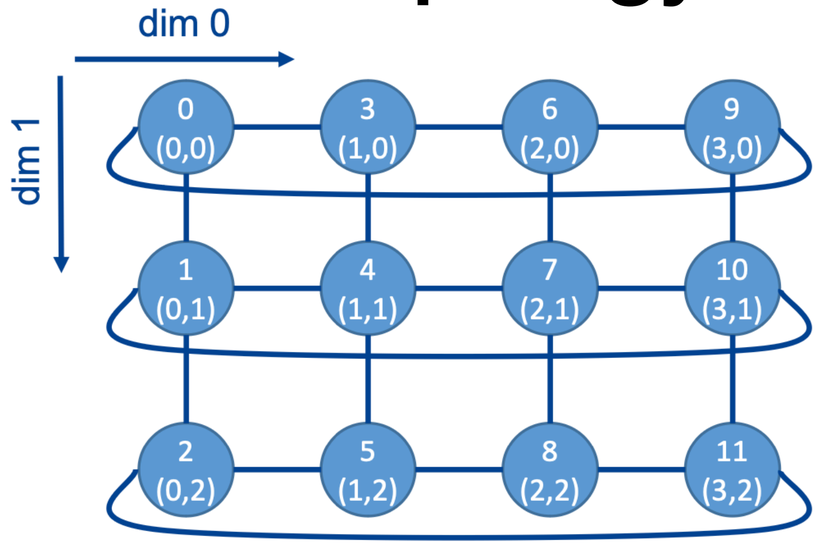
P0 sends 12345, receives 67890
P1 sends 67890, receives 12345

https://rookiehpc.org/mpi/docs/mpi_sendrecv/index.html

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

#define NDIM 2

int main(int argc, char* argv[]) {
    int wrank, crank, size;
    int dims[NDIM] = {0, 0};
    int period[NDIM] = {1, 0};
    int coords[NDIM];
    MPI_Comm comm;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &wrank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Dims_create(size, NDIM, dims);

    if (wrank==0)
        printf("p: %2d Dims (%d x %d)\n",
            size, dims[0], dims[1]);

    MPI_Cart_create(MPI_COMM_WORLD, NDIM, dims,
        period, 1, &comm);
    MPI_Comm_rank(comm, &crank);
    MPI_Cart_coords(comm, crank, NDIM, coords);

    int src[NDIM], dst[NDIM];

    MPI_Cart_shift(comm, 0, 1, &src[0], &dst[0]);
    MPI_Cart_shift(comm, 1, 1, &src[1], &dst[1]);

    int sum[NDIM]= {0, 0};
    MPI_Status stat;

    for (int j=0; j<NDIM; ++j) {
        int in=0, out=crank;

        for (int i = 0; i < dims[j]; ++i) {
            MPI_Sendrecv(&out, 1, MPI_INT,
                         dst[j], 111,
                         &in, 1, MPI_INT,
                         src[j], 111,
                         comm, &stat);
            sum[j] += in;
            out = in;
        }
    }
    printf("W: %2d C: %2d (%d, %d) "
           "dim-0: s: %2d d: %2d SUM= %2d "
           "dim-1: s: %2d d: %2d SUM= %2d\n",
           wrank, crank, coords[0], coords[1],
           src[0], dst[0], sum[0],
           src[1], dst[1], sum[1]);

    MPI_Comm_free(&comm);
    return MPI_Finalize();
} // main
```

What is the output for p=12?

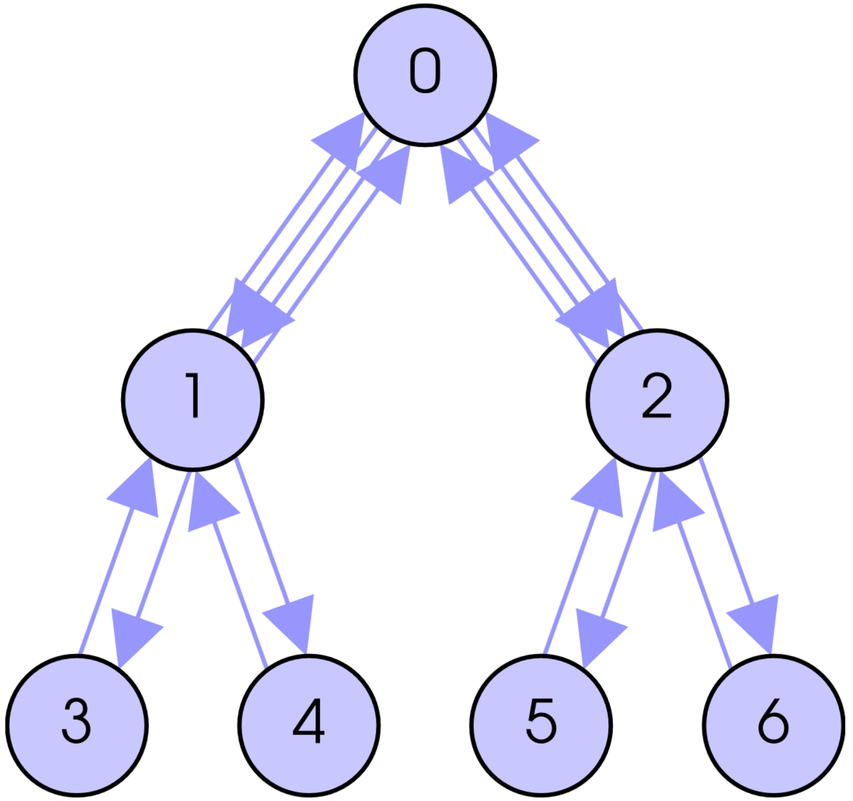
Cartesian Topology: Example




```
p: 12 Dims (4 x 3)
W:  0 C:  0 (0, 0) dim-0: s:  9 d:  3 SUM= 18 dim-1: s: -2 d:  1 SUM=  0
W:  1 C:  1 (0, 1) dim-0: s: 10 d:  4 SUM= 22 dim-1: s:  0 d:  2 SUM=  0
W:  2 C:  2 (0, 2) dim-0: s: 11 d:  5 SUM= 26 dim-1: s:  1 d: -2 SUM=  1
W:  3 C:  3 (1, 0) dim-0: s:  0 d:  6 SUM= 18 dim-1: s: -2 d:  4 SUM=  0
W:  4 C:  4 (1, 1) dim-0: s:  1 d:  7 SUM= 22 dim-1: s:  3 d:  5 SUM=  3
W:  5 C:  5 (1, 2) dim-0: s:  2 d:  8 SUM= 26 dim-1: s:  4 d: -2 SUM=  7
W:  6 C:  6 (2, 0) dim-0: s:  3 d:  9 SUM= 18 dim-1: s: -2 d:  7 SUM=  0
W:  7 C:  7 (2, 1) dim-0: s:  4 d: 10 SUM= 22 dim-1: s:  6 d:  8 SUM=  6
W:  8 C:  8 (2, 2) dim-0: s:  5 d: 11 SUM= 26 dim-1: s:  7 d: -2 SUM= 13
W:  9 C:  9 (3, 0) dim-0: s:  6 d:  0 SUM= 18 dim-1: s: -2 d: 10 SUM=  0
W: 10 C: 10 (3, 1) dim-0: s:  7 d:  1 SUM= 22 dim-1: s:  9 d: 11 SUM=  9
W: 11 C: 11 (3, 2) dim-0: s:  8 d:  2 SUM= 26 dim-1: s: 10 d: -2 SUM= 19
```

How to Describe a Topology

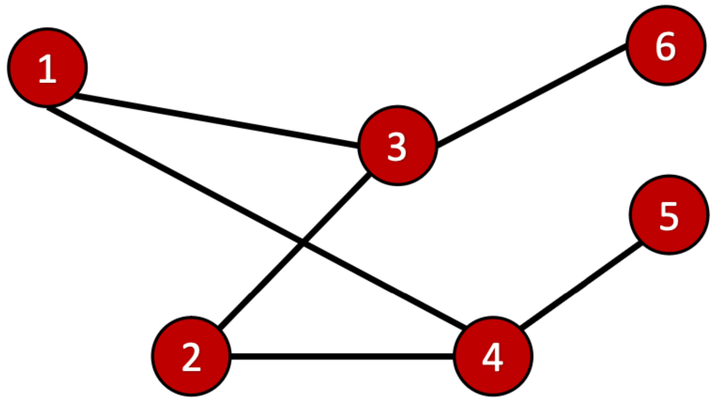
In the most general case, each topology can be represented as a directed graph with multi-edges,



Graph Formats

Adjacency Matrix Representation for Graphs

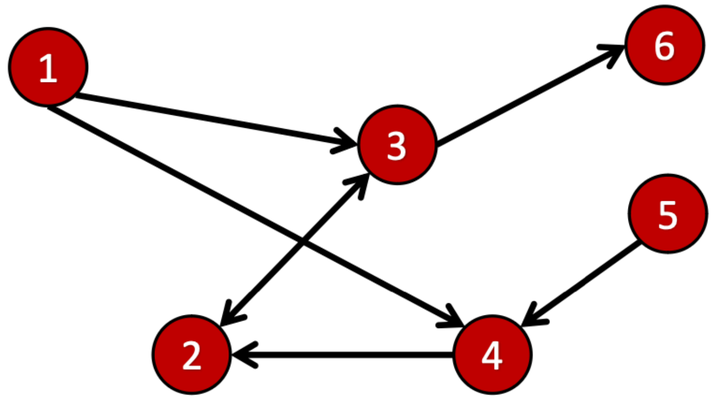
- Adjacency matrix
- For n vertices
  - Square matrix: $n \times n$
  - Undirected -> binary, symmetric



0	0	1	1	0	0
0	0	1	1	0	0
1	1	0	0	0	1
1	1	0	0	1	0
0	0	0	1	0	0
0	0	1	0	0	0

Adjacency Matrix Representation for Graphs

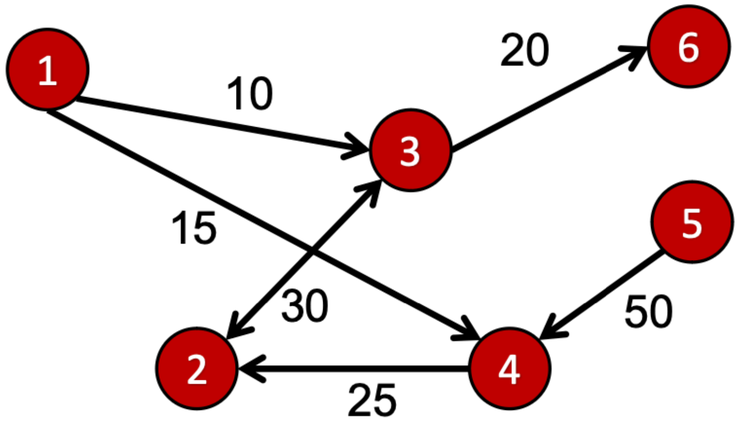
- Adjacency matrix
- For n vertices
  - Square matrix: $n \times n$
  - Undirected -> binary, symmetric
  - Directed -> binary, not symmetric



0	0	1	1	0	0
0	0	1	0	0	0
0	1	0	0	0	1
0	1	0	0	0	0
0	0	0	1	0	0
0	0	0	0	0	0

Adjacency Matrix Representation for Graphs

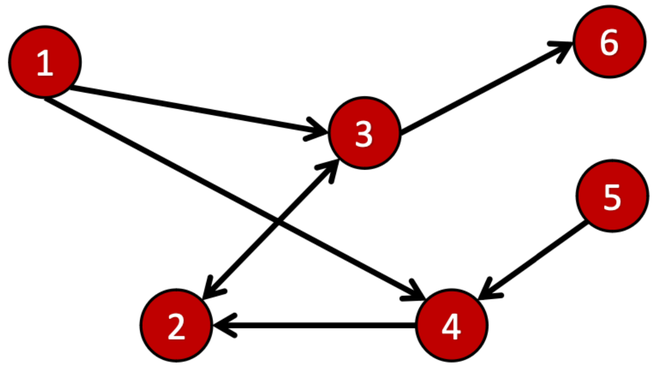
- Adjacency matrix
- For n vertices
  - Square matrix: $n \times n$
  - Undirected -> binary, symmetric
  - Directed -> binary, not symmetric
  - Weighted -> not binary



0	0	10	15	0	0
0	0	30	0	0	0
0	30	0	0	0	20
0	25	0	0	0	0
0	0	0	50	0	0
0	0	0	0	0	0

Adjacency List Representation for Graphs

- In practice, many real-world graphs are **sparse** - they have many less than n^2 edges
- Sparse representations take space proportional to $O(m+n)$



1	3, 4
2	3
3	2, 6
4	2
5	4
6	

Adjacency List

	1	2	3	4	5	6
1			1	1		
2			1			
3		1				1
4		1				
5				1		
6						

Adjacency Sparse Matrix

	1	2	3	4	5	6
1	0	0	1	1	0	0
2	0	0	1	0	0	0
3	0	1	0	0	0	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	0

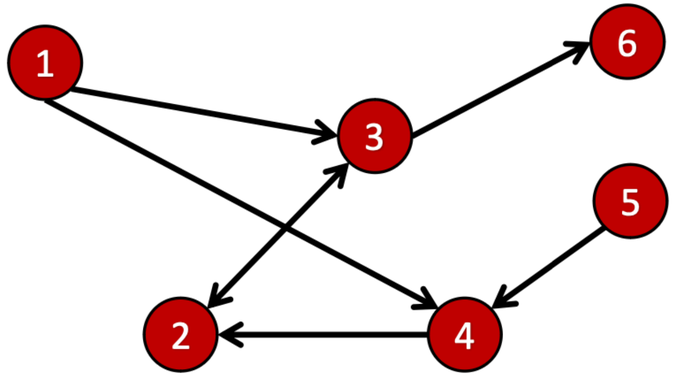
Adjacency Matrix

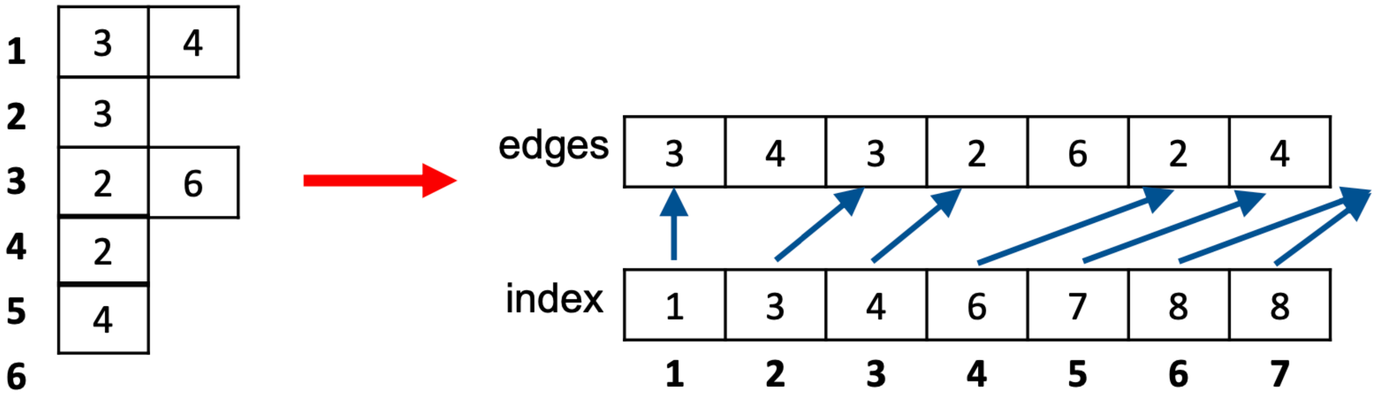
Space: $O(n+m)$

Space: $O(n^2)$

Compressed Sparse Row (CSR) Representation

- Compressed Sparse Row (CSR) is a classical sparse matrix/graph representation.
- It supports efficient traversal through the neighbors of a vertex.





Adjacency List

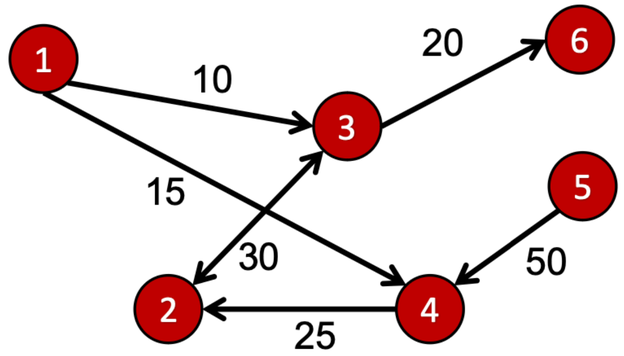
Compressed Adjacency List/
Compressed Sparse Row

Space: $O(n+m)$

Space: $O(n+m)$

Coordinate (COO) Representation

- Coordinate format (COO) stores the edges as tuples (src, dest, weight)
- It uses space proportional to the number of edges



1	3	4
2	3	
3	2	6
4	2	
5	4	
6		

Adjacency List

Space: $O(n+m)$

weights	10	15	30	30	20	25	50
edges	3	4	3	2	6	2	4
index	1	3	4	6	7	8	8
	1	2	3	4	5	6	7

Compressed Adjacency List / Compressed Sparse Row (CSR)

Space: $O(n+m)$

Space = $2m+n+1$

1	3	10
1	4	15
2	3	30
3	2	30
3	6	20
4	2	25
5	4	50

Edge List / Coordinate (COO)

Space: $O(m)$

Space = $3m$

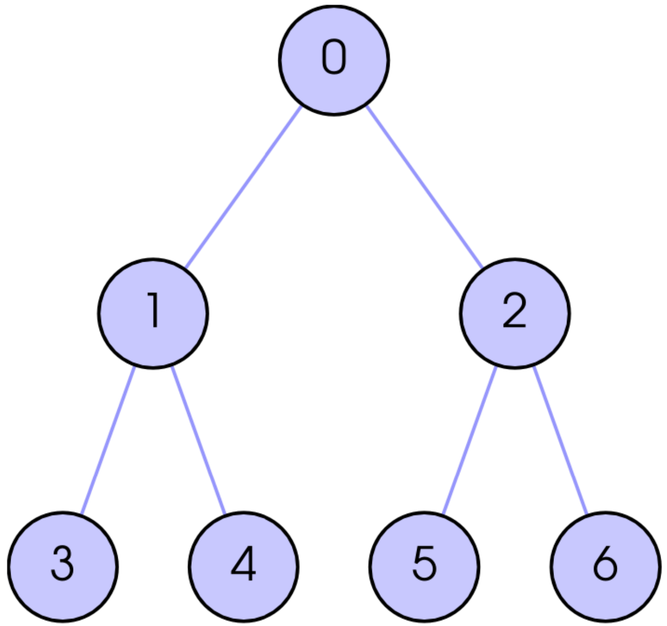
MPI Graph Topology

- MPI provides a general function to create an arbitrary topology:

```
int MPI_Graph_create(MPI_Comm comm, int nnodes,
                     int* index, int* edges,
                     int reorder, MPI_Comm* newcomm);
```

- No performance guarantees for parallel edges
- Edge direction does not imply a direction of the communication

MPI Graph Topology



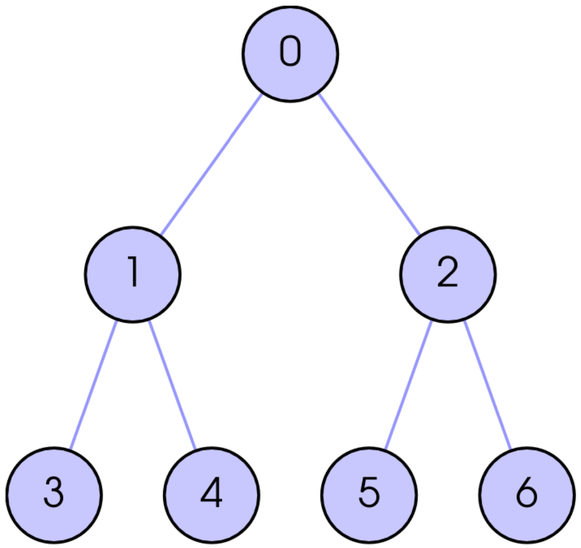

```
nnodes = 7
index = [2,5,8,9,10,11,12]
edges = [1,2,0,3,4,0,5,6,1,1,2,2]
```

```
MPI_Comm hypercube_create(MPI_Comm comm) {
    int nnodes = 0;
    MPI_Comm_size(comm, &nnodes);

    int k = static_cast<int>(log2(nnodes));
    int* index = new int[nnodes];
    int* edges = new int[nnodes * k];

    for (int i = 0; i < nnodes; ++i)
         index[i] = (i + 1) * k;

    unsigned int pos = 0;
    for (int i = 0; i < nnodes; ++i) {
        unsigned int mask = (1 << k);
        for (int j = 0; j < k; ++j) {
            mask >>= 1;
            unsigned int p = (i ^ mask);
            edges[pos++] = p;
        }
    } // for

    MPI_Comm newcomm;
    MPI_Graph_create(comm, nnodes, index, edges, 1, &newcomm);

    delete[] edges;    delete[] index;
    return newcomm;
} // hypercube_create
```

MPI Graph Topology

MPI provides routines to create arbitrary topologies as general graph virtual topology

- **Vertices** represent MPI processes
- **Edges** indicate important connections
- Edge **weights** provide additional info, such as volume of communication



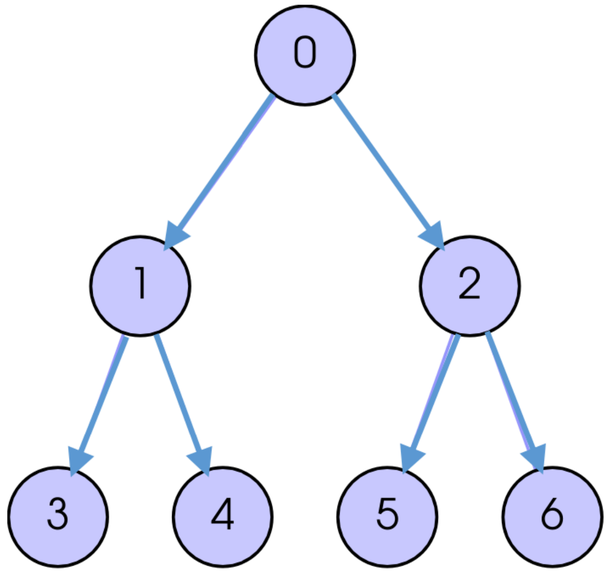
Distributed Graph Topology

- Distributed, directed, graph create function:

```
int MPI_Dist_graph_create_adjacent(
        MPI_Comm comm_old, int indegree, const int sources[],
        const int sourceweights[],
        int outdegree, const int destinations[],
        const int destweights[],
         MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
```

- Describes only the graph vertex corresponding to the calling process
- `sourceweights, destweights`: array of non-negative integers or `MPI_UNWEIGHTED`

MPI_Dist_graph_create_adjacent



Process 0

```
indegree  = 0
sources = []
outdegree = 2
destinations = [1,2]
```

Process 1

```
indegree  = 1
sources = [0]
outdegree = 2
destinations = [3,4]
```

...

P6

```
ideg = 1
srcs = [2]
odeg = 0
dests = []
```

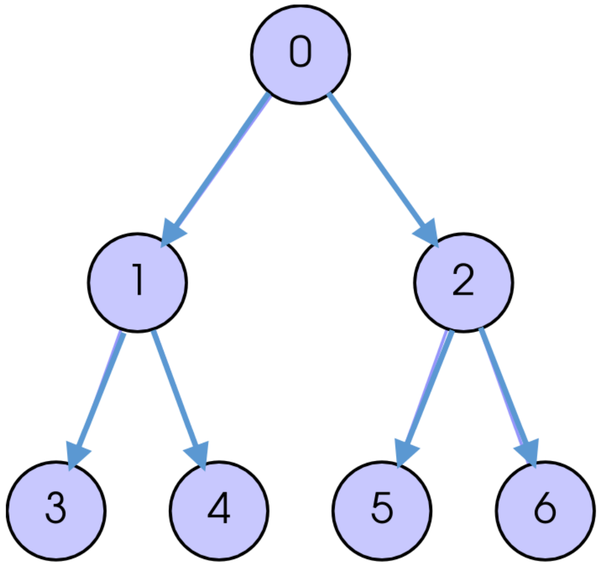
Distributed Graph Topology

- More general:

```
int MPI_Dist_graph_create(
              MPI_Comm comm_old, int n, const int sources[],
              const int degrees[], const int destinations[],
              const int weights[],
              MPI_Info info, int reorder,
              MPI_Comm *comm_dist_graph)
```

- Allows multiple graph vertices per process
- Requires global communication, hence slightly more expensive
- `n` : number of source nodes
- `sources` : n source nodes
- `degrees` : number of edges for each source
- `destinations` : destination processors
- `weights`: array of non-negative integers or `MPI_UNWEIGHTED`

MPI_Dist_graph_create



Process 0	Process 1	...	P6
n = 2	n = 2	...	n = 0
sources = [0,1]	sources = [2,5]		...
degrees = [2,2]	degrees = [2,0]		
dests = [1,2,3,4]	dests = [5,6]		

MPI Derived Datatypes

MPI Datatyles

Built-in datatypes: char, int, float, double, ...
What about structs, array of structs?

```
struct A {
    char c;
    double d;
    char e[3];
};
A arr[10];
```

MPI Datatyles

Why not just send `sizeof(A) * MPI_BYTE`?

```
struct A {
    char c;
    double d;
    char e[3];
};
A arr[10];
```

sizeof(A)?

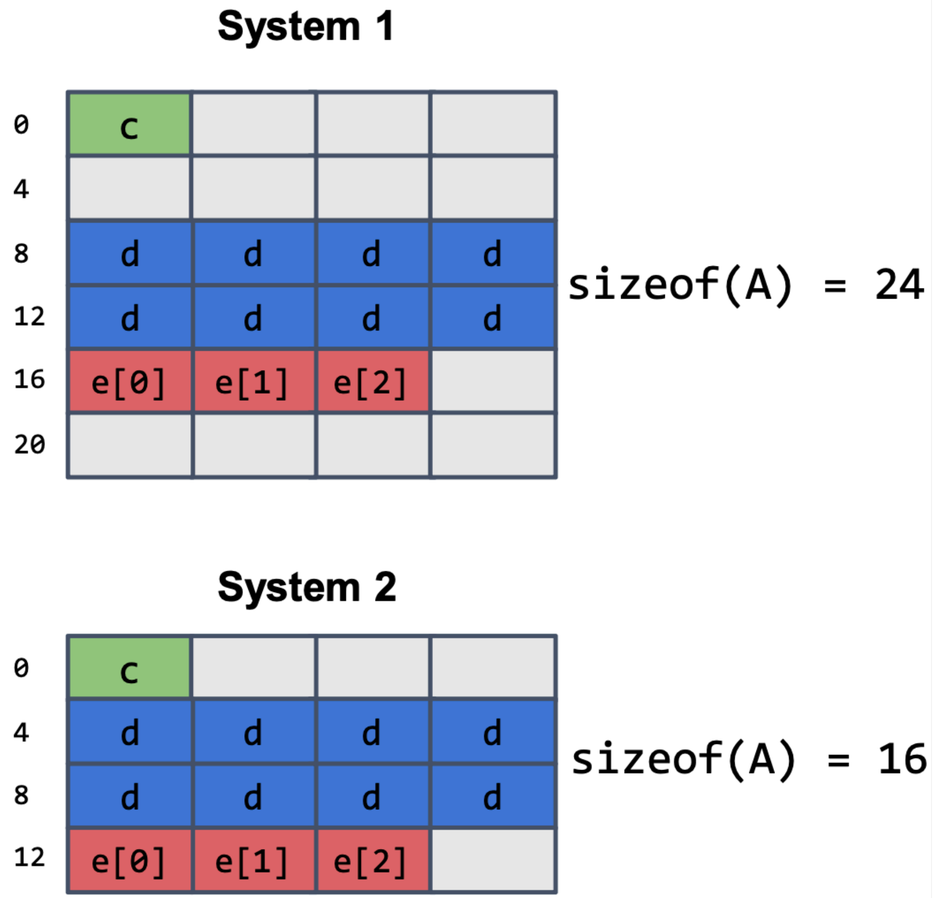
MPI Datatyles

Why not just send `sizeof(A) * MPI_BYTE`?

```
struct A {
    char c;
    double d;
    char e[3];
};
A arr[10];
```

sizeof(A)?

Heterogeneous systems!



MPI Datatypes

- MPI's generalized type model:

  Type map = {($type_0$,$disp_0$),($type_1$,$disp_1$),...,($type_{n-1}$,$disp_{n-1}$)}

where:

  $type_i$: base (built-in) type (**int**, **char**, **double**,...)

  $disp_i$: displacement (byte offset)

- Example:

```
MPI_DOUBLE -> {(double, 0)}
int[3]     -> {(int, 0), (int, 4), (int, 8)}
```

MPI Datatypes

- MPI's generalized type model:

Type map = $\{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}$

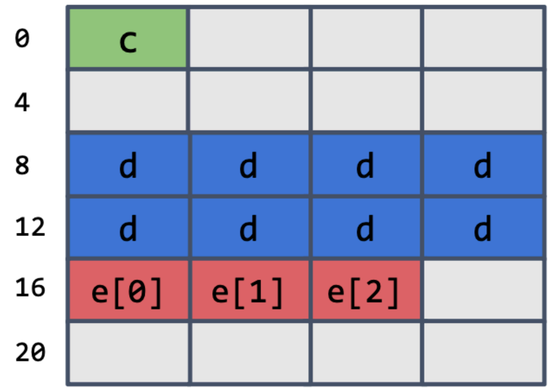
where:

$type_i$: base (built-in) type (**int**, **char**, **double**,...)

$disp_i$: displacement (byte offset)

- Example:

```
struct A {
    char c;
    double d;
    char e[3];
};
```




```
struct A -> {(char, 0),
             (double, 8),
             (char, 16), (char, 17), (char, 18)}
```

MPI Datatypes

MPI provides functions to create new MPI Datatypes (= **type maps**)

- `MPI_Type_contiguous`
- `MPI_Type_vector`
- `MPI_Type_create_hvector`
- `MPI_Type_indexed`
- `MPI_Type_create_hindexed`
- `MPI_Type_create_indexed_block`
- `MPI_Type_create_hindexed_block`

subset of `MPI_Type_create_struct`

- `MPI_Type_create_struct`

most general

MPI_Type_create_struct

```
MPI_Type_create_struct(
      int                 count,
      const int           array_of_blocklengths[],
      const MPI_Aint      array_of_displacements[],
      const MPI_Datatype  array_of_types[],
      MPI_Datatype        *newtype)
```

size of arrays

$disp_i$

$type_i$

Type map = $\{(type_0,disp_0),(type_1,disp_1),...,(type_{n-1},disp_{n-1})\}$

-> count = n

-> array_of_types = $\{type_0,type_1,...,type_{n-1}\}$

-> array_of_displacements = $\{disp_0,disp_1,...,disp_{n-1}\}$

MPI_Type_create_struct

```
MPI_Type_create_struct(
      int                 count,
      const int           array_of_blocklengths[],
      const MPI_Aint      array_of_displacements[],
      const MPI_Datatype  array_of_types[],
      MPI_Datatype        *newtype)
```

size of arrays

number of contiguous elements of equal type

$disp_i$

$type_i$

Example:

```
struct A {
    char c;
    double d;
    char e[3];
};
```

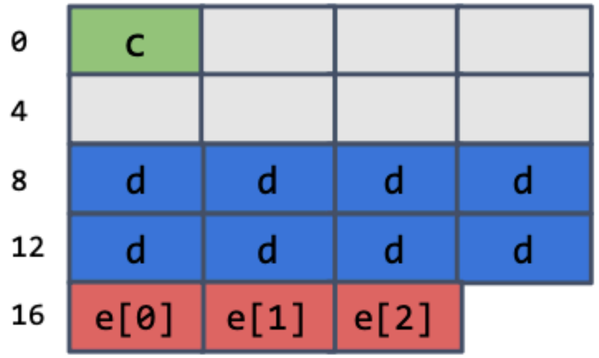
```
struct A: {(char, 0), (double, 8), (char, 16), (char, 17), (char, 18)}
      -> count = 3
      -> array_of_blocklength = {1, 1, 3}
      -> array_of_types = {MPI_CHAR,MPI_DOUBLE,MPI_CHAR}
      -> array_of_displacements = {0, 8, 16}
```

Example: Array of Structs

```
struct A: {(char, 0), (double, 8), (char, 16), (char, 17), (char, 18)}
```

struct A according to type map:

```
struct A {
    char c;
    double d;
    char e[3];
};
```

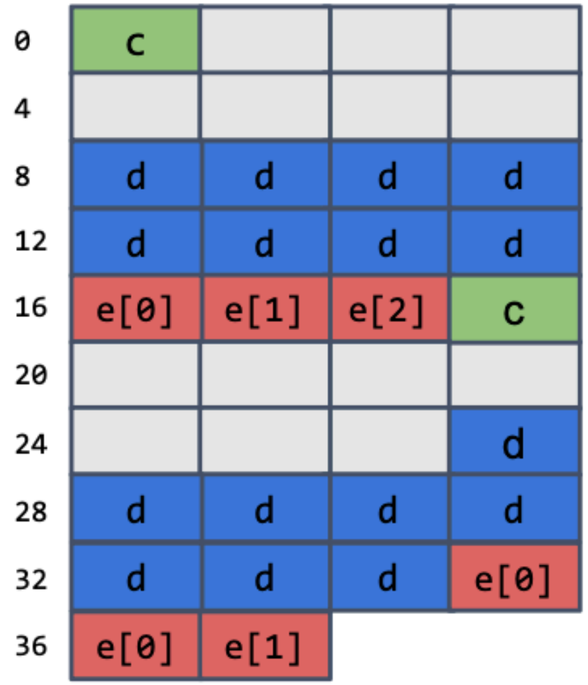


Example: Array of Structs

`struct A: {(char, 0), (double, 8), (char, 16), (char, 17), (char, 18)}`

```
struct A {
    char c;
    double d;
    char e[3];
};
```

Array of struct A?

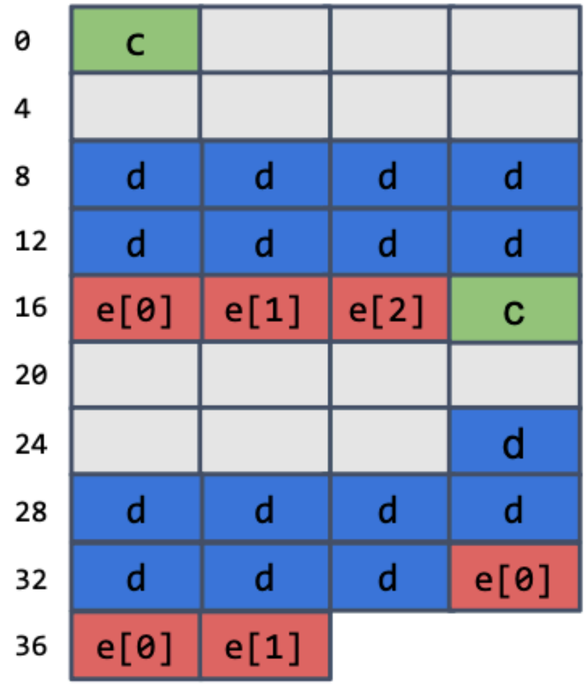


Example: Array of Structs

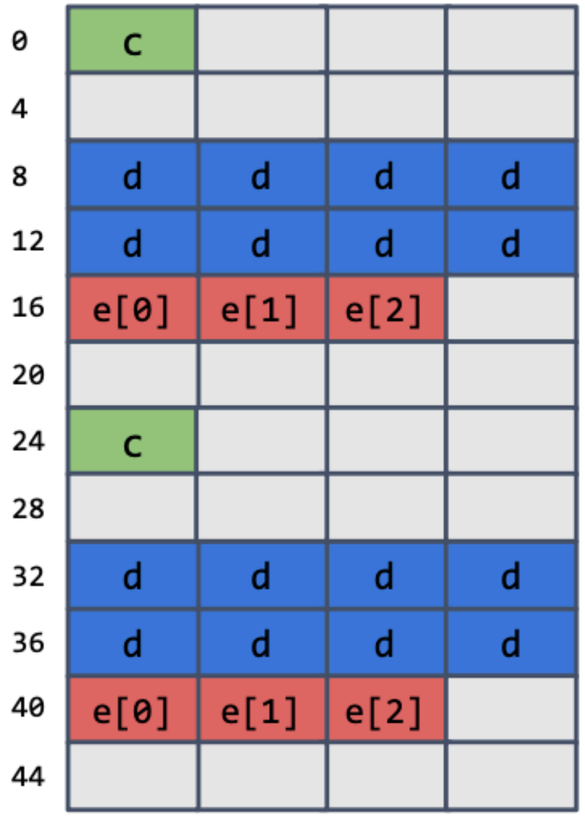
`struct A: {(char, 0), (double, 8), (char, 16), (char, 17), (char, 18)}`

```
struct A {
    char c;
    double d;
    char e[3];
};
```

Array of struct A?



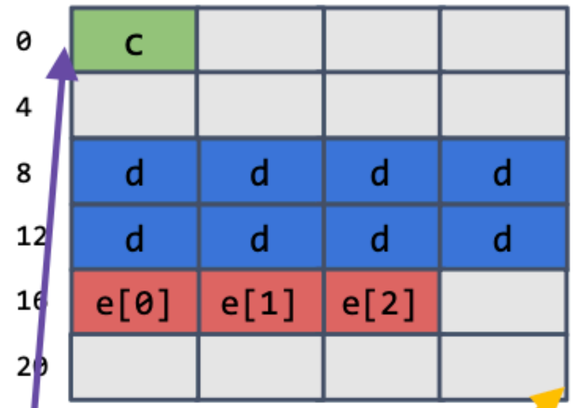
Actual memory layout:



Example: Array of Structs

```
struct A: {(char, 0), (double, 8), (char, 16), (char, 17), (char, 18)}
```

```
struct A {
    char c;
    double d;
    char e[3];
};
```



Solution: **lower** and **upper** bound

- new types: `lb` and `ub`
- size: 0
- define:

`extend = ub - lb`

```
struct A: {(lb, 0),
           (char, 0), (double, 8), (char, 16), (char, 17), (char, 18),
           (ub, 24)}
```

Statically assigning displacements

Creating MPI datatype for `struct A`:

What's wrong?

```
struct A {
    char c;
    double d;
    char e[3];
};

int          blens[3] = {1, 1, 3};
MPI_Datatype types[3] = {MPI_CHAR,MPI_DOUBLE,MPI_CHAR};
MPI_Aint     disps[3] = {0, 8, 16};
MPI_Datatype mpi_tmp_t, mpi_a_t;
MPI_Type_create_struct(3, blens, types, disps, &mpi_tmp_t);
MPI_Type_create_resized(mpi_tmp_t, 0, 24, &mpi_a_t);
MPI_Type_commit(&mpi_a_t);
```

Statically assigning displacements

Creating MPI datatype for `struct A`:

Problem: Not Portable!

```
struct A {
    char c;
    double d;
    char e[3];
};

int          blens[3] = {1, 1, 3};
MPI_Datatype types[3] = {MPI_CHAR,MPI_DOUBLE,MPI_CHAR};
MPI_Aint     disps[3] = {0, 8, 16};
MPI_Datatype mpi_tmp_t, mpi_a_t;
MPI_Type_create_struct(3, blens, types, disps, &mpi_tmp_t);
MPI_Type_create_resized(mpi_tmp_t, 0, 24, &mpi_a_t);
MPI_Type_commit(&mpi_a_t);
```

Determine displacements dynamically

Correctly determining displacements:

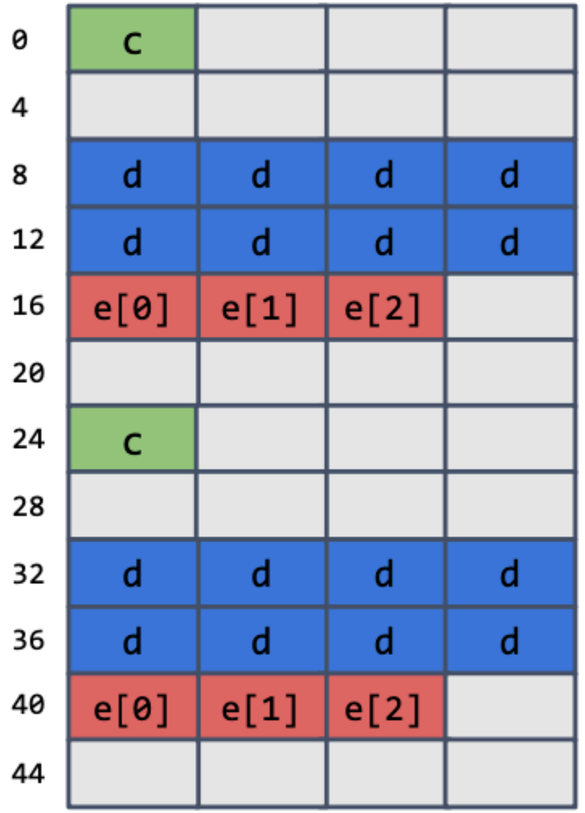
```
struct A {
    char c;
    double d;
    char e[3];
};

A a;
MPI_Aint base, adr_c, adr_d, adr_e;
MPI_Get_address(&a, &base);
MPI_Get_address(&a.c, &adr_c);
MPI_Get_address(&a.d, &adr_d);
MPI_Get_address(&a.e[0], &adr_e);
MPI_Aint disps[3] = {adr_c - base, adr_d - base, adr_e - base};
MPI_Aint extend = sizeof(a);
...
MPI_Type_create_struct(3, blens, types, disps, &mpi_tmp_t);
MPI_Type_create_resized(mpi_tmp_t, 0, extend, &mpi_a_t);
MPI_Type_commit(&mpi_a_t);
```

calculating offsets in an architecture independent manner

setting the true size of the data type

Communicating MPI structs

- So far:
  - we can communicate complex structs
  - between heterogeneous architectures
- BUT
  - sending fragmented data with MPI is slow …
  - MPI might create its own buffers to “linearize” the data (no gaps)
  - we are doing HPC!
  - we care about performance

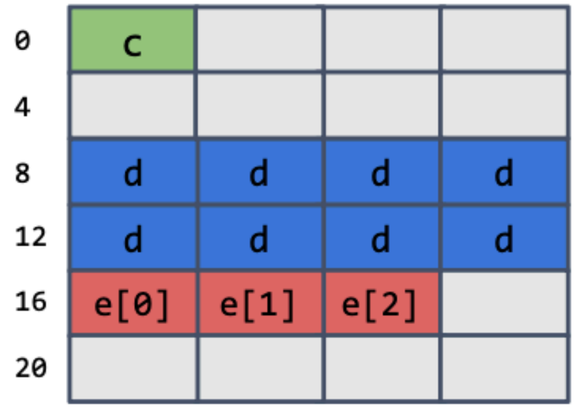


Optimizing MPI datatypes

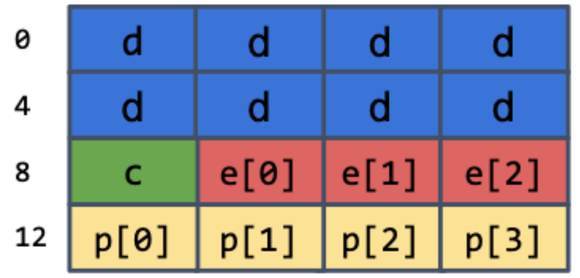
- Optimization
  - re-arrange data
  - add ***explicit padding***
  - include explicit padding in MPI datatype

- Now MPI can send contiguous data when sending an array of type A
  - MPI communication gets faster
  - plus we saved space due to the reordering!

```
struct A {
    char c;
    double d;
    char e[3];
};
```




```
struct A {
    double d;
    char c;
    char e[3];
    // padding
    char p[4];
};
```



Creating MPI datatypes

- `MPI_Type_create_struct`:
  - most general function for creating MPI datatypes

- Two other useful ones:

```
int MPI_Type_contiguous(int count,
                        MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Example:

```
struct mypair {
    int x;
    int y;
};

MPI_Datatype pair_type;
MPI_Type_contiguous(2, MPI_INT, &pair_type);
```

MPI_Type_vector Example

Matrix:

1	2	3
4	5	6
7	8	9

In row major:

1	2	3	4	5	6	7	8	9

Suppose we only want to send the second column:

-	2	-
-	5	-
-	8	-

-	2	-	-	5	-	-	8	-

```
case SENDER:
{
  // Create the datatype
  MPI_Datatype column_type;
  MPI_Type_vector(3, 1, 3, MPI_INT, &column_type);
  MPI_Type_commit(&column_type);

  // Send the message
  int buffer[3][3] = { 0, 1, 2, 3, 4, 5, 6, 7, 8 };
  MPI_Request request;
  printf("MPI process %d sends values %d, %d and %d.\n", my_rank, buffer[0][1], buffer[1]
[1], buffer[2][1]);
  MPI_Send(&buffer[0][1], 1, column_type, RECEIVER, 0, MPI_COMM_WORLD);
  break;
}
case RECEIVER:
{
   // Receive the message
   int received[3];
   MPI_Recv(&received, 3, MPI_INT, SENDER, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   printf("MPI process %d received values: %d, %d and %d.\n", my_rank, received[0],
received[1], received[2]);
   break;
}
```

MPI_Type_create_resized Example

Matrix:

1	2	3
4	5	6
7	8	9

In row major:

1	2	3	4	5	6	7	8	9

Problem: MPI_Scatter takes contiguous data

Suppose we want to scatter the columns onto the processors:

P1	P2	P3
1	2	3
4	5	6
7	8	9

```
    // Create the vector datatype
    MPI_Datatype column_not_resized;
    MPI_Type_vector(CELLS_PER_COLUMN, 1, CELLS_PER_ROW, MPI_INT, &column_not_resized);

    // Resize it to make sure it is interleaved when repeated
    MPI_Datatype column_resized;
    MPI_Type_create_resized(column_not_resized, 0, sizeof(int), &column_resized);
    MPI_Type_commit(&column_resized);

    // Get my rank and do the corresponding job
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    int my_column[CELLS_PER_COLUMN];
    if(my_rank == 0)
    {
        // NOT SHOWN: Declare and initialise the full array
        // Send the column
        MPI_Scatter(full_array, 1, column_resized, my_column, CELLS_PER_COLUMN, MPI_INT, 0,
MPI_COMM_WORLD);
    }
    else
    {
        // Receive the column
        MPI_Scatter(NULL, 1, column_resized, my_column, CELLS_PER_COLUMN, MPI_INT, 0,
MPI_COMM_WORLD);
    }
```

https://rookiehpc.org/mpi/docs/mpi_type_create_resized/index.html

MPI Custom Operators

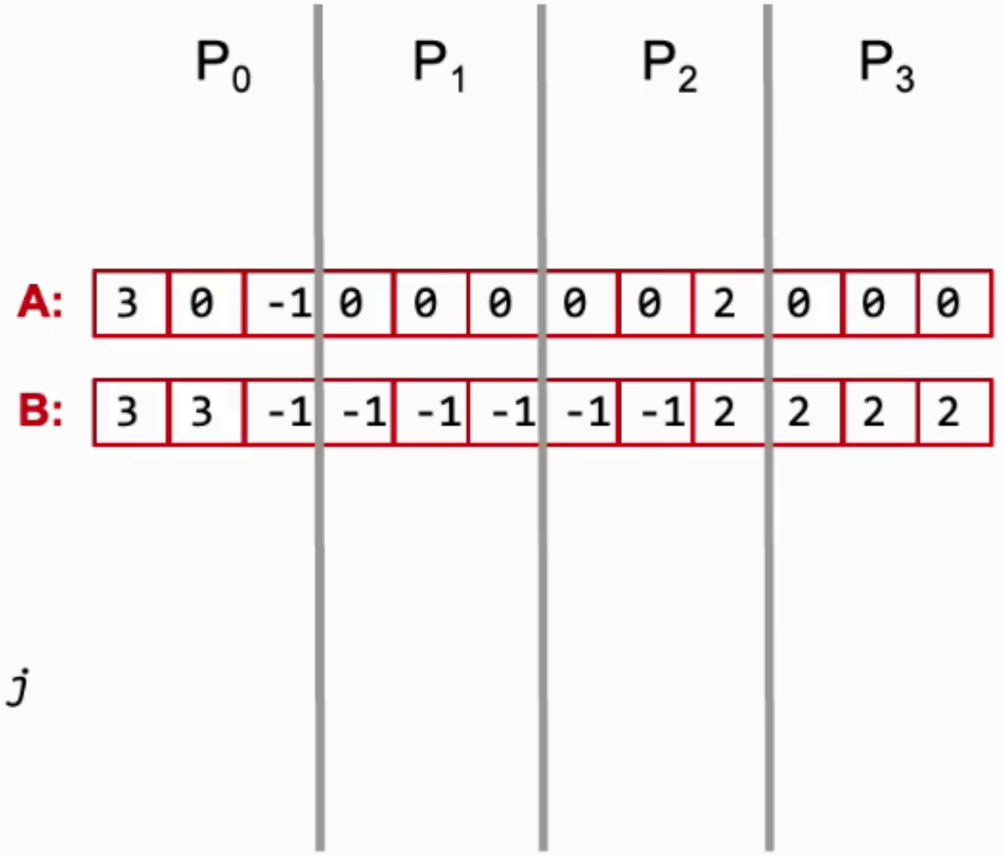
MPI Custom Operators

Let's consider the following problem:

- For each entry of an array A that is 0, replace it with the nearest non-zero value to the left
- Solution:
  - initialize tuples
    - (i, A[i]), if A[i] ≠ 0
    - (-1,0), if A[i] = 0
  - prefix scan with modified *max*:

$$max\{(i,A[i]),\ (j,A[j])\} = \begin{cases} (i,A[i]) & \text{if } i > j \\ (j,A[j]) & \text{else} \end{cases}$$

- In MPI? -> custom reduction operators



MPI Custom Operators

- MPI functions:
  - `MPI_Reduce`
  - `MPI_Allreduce`
  - `MPI_Scan`
  - `MPI_Exscan`
- Builtin operators **`MPI_Op`**:
  - `MPI_SUM, MPI_PROD,`
  - `MPI_MIN, MPI_MAX,`
  - `MPI_LOR, MPI_LAND,`
  - `MPI_BOR, MPI_BAND,…`

MPI Custom Operators

Creating a custom `MPI_Op`:

```
void myop(void *invec, void *inoutvec,
          int *len, MPI_Datatype *datatype){
    int* in = (int*) invec;
    int* inout = (int*) inoutvec;
    for (int i = 0; i < *len; ++i) {
        if (in[2*i] > inout[2*i]) {
            inout[2*i] = in[2*i];
            inout[2*i+1] = in[2*i+1];
        }
    }
}
```

`inout[i] = in[i] ⊙ inout[i]`

`len`: number of elements of type `` `datatype` ``

$$max\{(i,A[i]), (j,A[j])\} = \begin{cases} (i,A[i]) \text{ if } i > j \\ (j,A[j]) \text{ else} \end{cases}$$

MPI Custom Operators

Creating a custom `MPI_Op`:

```
void myop(void *invec, void *inoutvec,
          int *len, MPI_Datatype *datatype){
    int* in = (int*) invec;
    int* inout = (int*) inoutvec;
    for (int i = 0; i < *len; ++i) {
        if (in[2*i] > inout[2*i]) {
            inout[2*i] = in[2*i];
            inout[2*i+1] = in[2*i+1];
        }
    }
}
...
MPI_Op mpi_myop;
MPI_Op_create(&myop, 1, &mpi_myop);
```

commutative?

function pointer

new MPI_Op

`inout[i] = in[i] ⊙ inout[i]`

`len`: number of elements of type `` `datatype` ``

$$max\{(i,A[i]),\ (j,A[j])\} = \begin{cases} (i,A[i]) & \text{if } i > j \\ (j,A[j]) & \text{else} \end{cases}$$