Data reorganization patterns

UROŠ LOTRIČ

Introduction

Data transfer is many times a bottleneck

For data-intensive applications primary design focus should be on data movement and add computation later

Parallel systems add additional cost

- For efficient vectorization it is important to properly declare sructures
- Effect of cache size on scalability (avoid false sharing)

Gather

Gather collects all data from a collection of location indices and source arrays and places them into an output collection

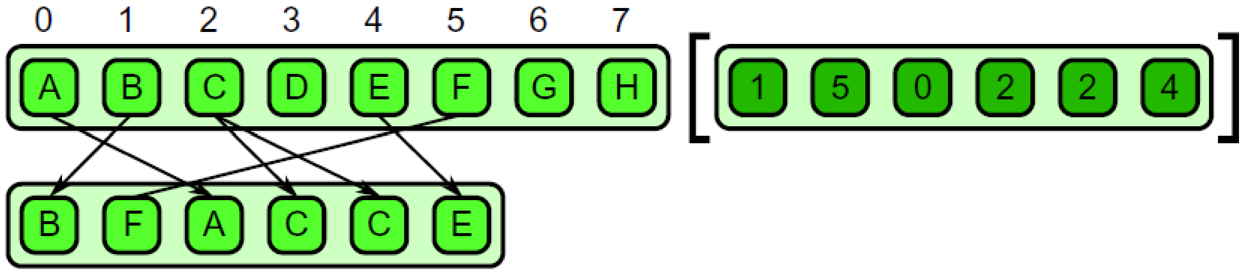
Combination of a random read and map

Output data has

- the same number of elements as the number of indices in input collections
- the same dimensionality as location index collection

MPI_Gather

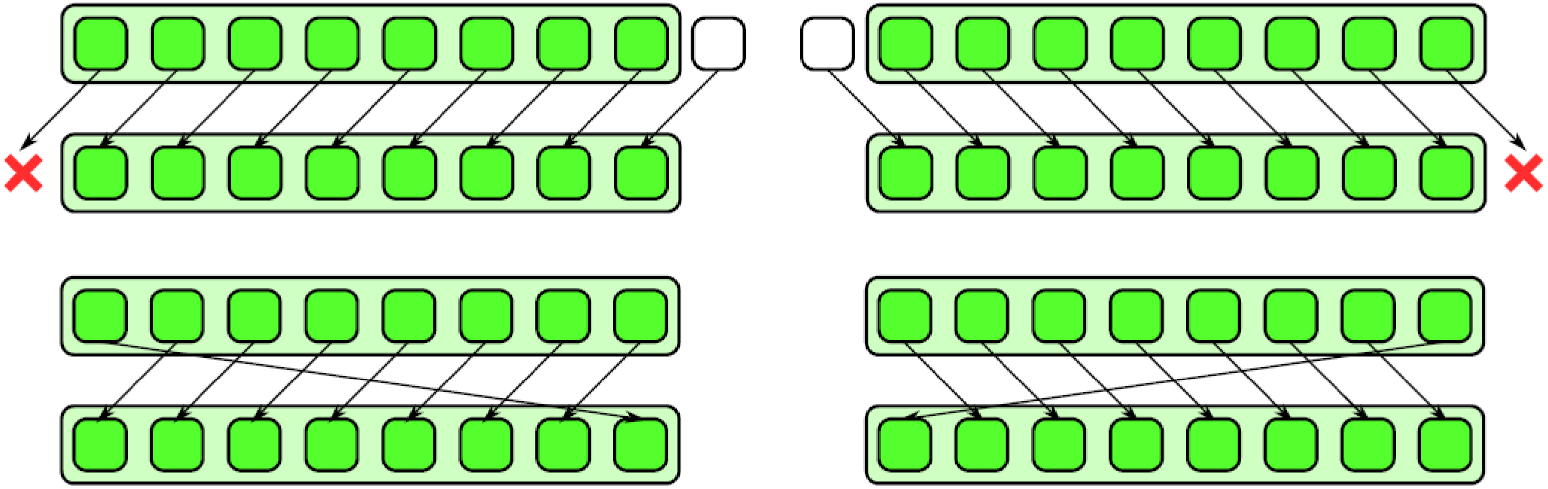
- Less general
- A lot can be gained with derived datatypes



Gather

Shift / Rotate

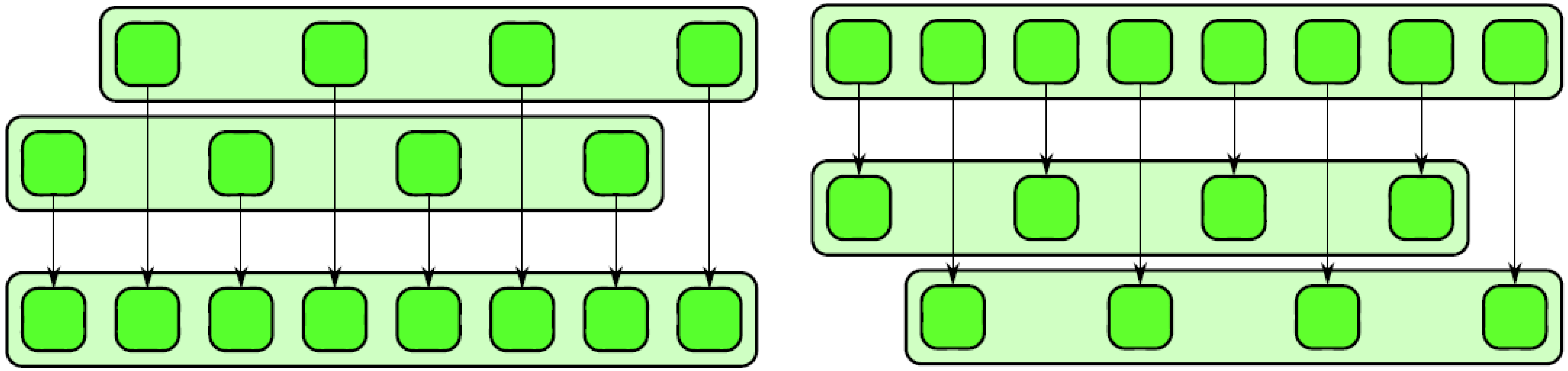
- Special gathers
- Has regular data access pattern
- Can be efficiently implemented using vector instructions
- In multi-dimensions shift/rotate offsets may differ
- Leads to coalesced data access
- Efficient implementations using vector operations
- Boundary conditions handling



Gather

Zip / Unzip

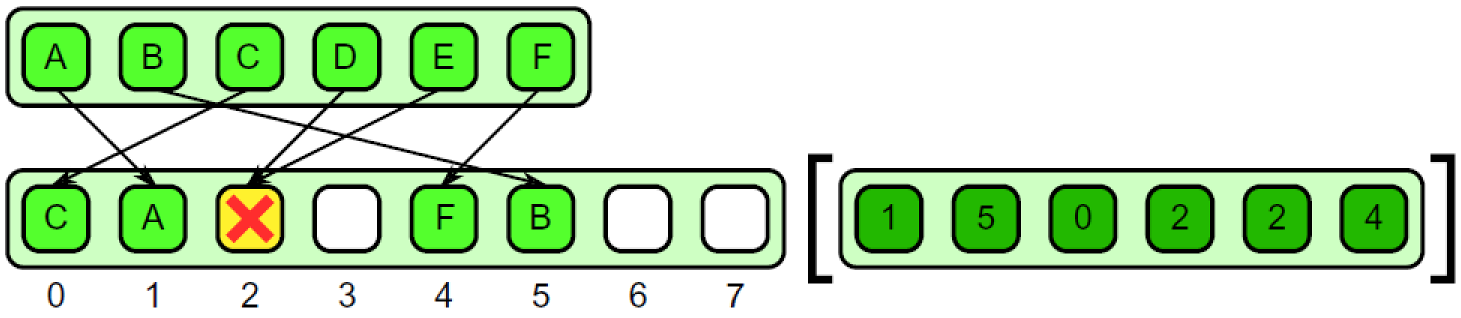
- Interleaves data
- Example: assemble complex data by interleaving real and imaginary parts
- Convert from structure of arrays to array of structures
- Unzip reverses zip operation



Scatter

A collection of input data is written to specified write locations

Multiple writes to the same location are possible



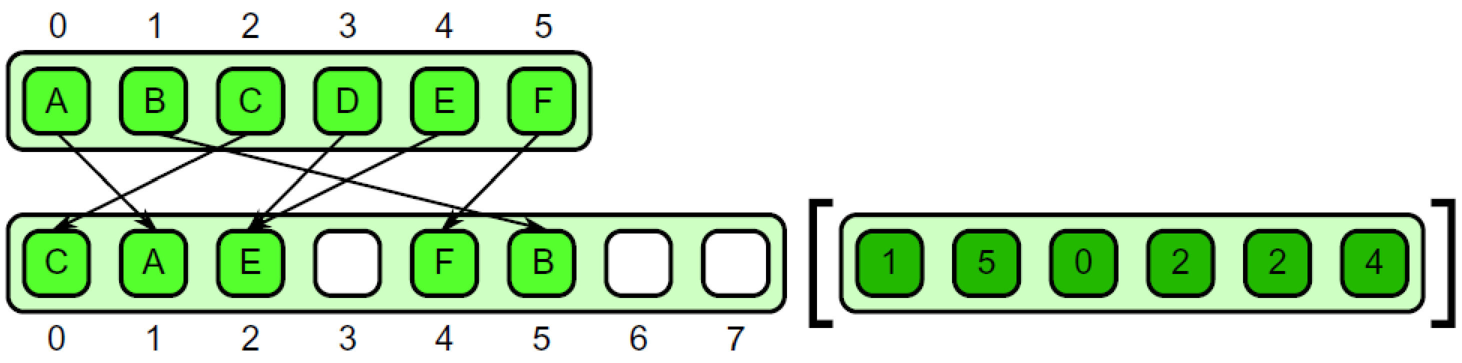
Resolutions

- Permutation scatter
  - Collisions are illegal, array of indices should not have duplicates
  - Can be always turned into gather when addresses are known in advance
  - Example: matrix transpose
  - MPI_Scatter
- Merge scatter
  - Combines values
  - Only works with associative and commutative operators
  - Example: histogram computation

Scatter

Resolutions

- Atomic scatter
  - Atomic writes, non-deterministic
  - Can be deterministic when written input elements gave the same value
  - Example: parallel disjunction (output array is initially cleared, writing true is actually OR operation)
- Priority scatter (deterministic using priorities)
  - Priority based on a position of an element in input array
  - Higher priority for elements at the end of the array is consistent with serial code



Gather vs Scatter

Scatter is more expensive

- Gather reads versus scatter reads & writes (whole cache line)
- Scatter on shared memory systems requires cores synchronization to keep cache coherent, false sharing may occur

If addresses are known in advance, scatter can be converted to gathers

One option is also to scatter the addresses first and later gather data

Conversion takes resources

- Makes sense when it is used repeatedly

Suitable for shared-memory systems

MPI_Gather and MPI_Scatter

- Optimized, not so general
- No need for conversion

Pack and unpack

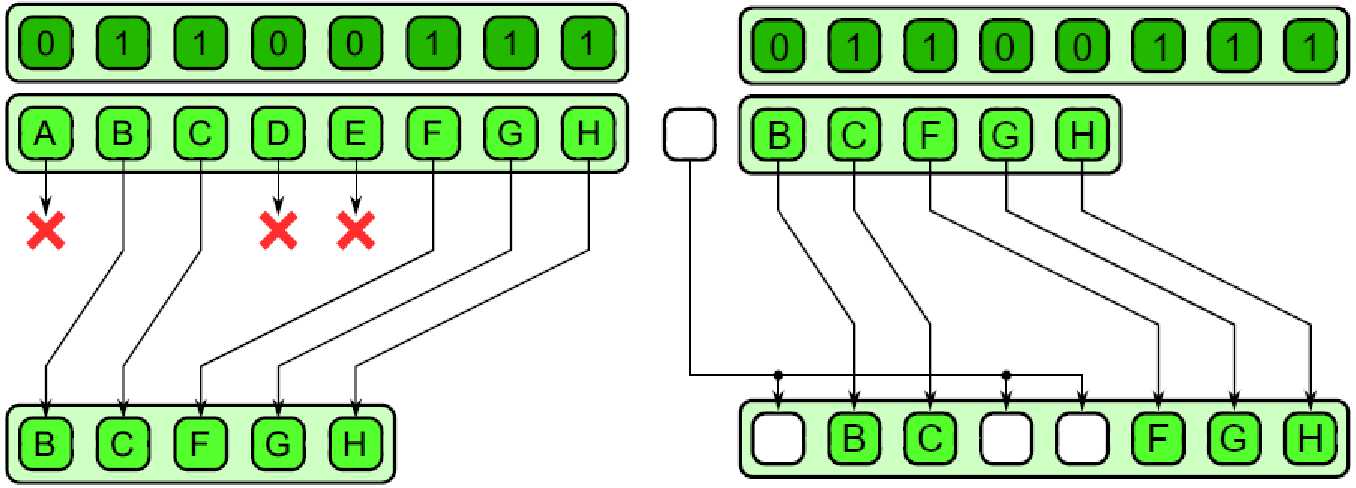
Eliminates unused elements from collection

Output is contiguous in memory, which leads to better memory access and vectorization

Pack is combination of scan with conditional scatter

Pack can be fused with map

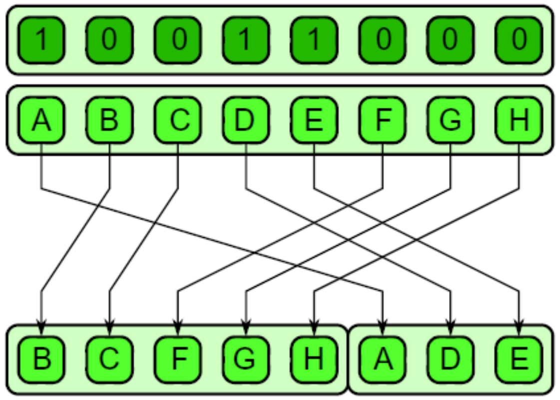
- Useful when small number of elements is discarded

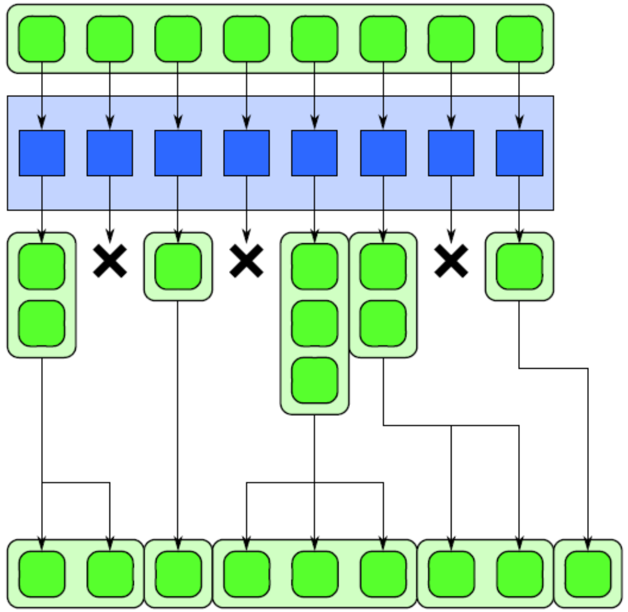


Pack and unpack

Generalizations

- Split
  - separate elements to two or more sets
- Expand
  - In combination with map
  - When map can produce arbitrary number of elements





MPI_Pack and MPI_Unpack

- Packs a datatype into a contiguous memory
- Useful for combining data of different datatypes to reduce number of sends
- MPI_Pack_size gives size of data in bytes
  - Useful to dynamically allocate size of pack structure
- Copies data to new location (better to use datatypes)

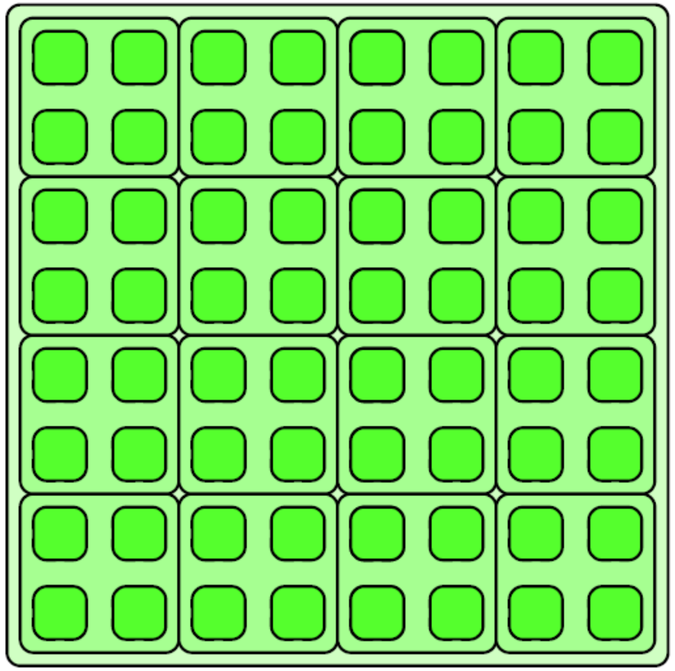
Geometric decomposition

Common parallelization strategy

- Divide computational domain to sections
- Work on sections individually
- Combine the results
- Divide-and-conquer

Geometric decomposition

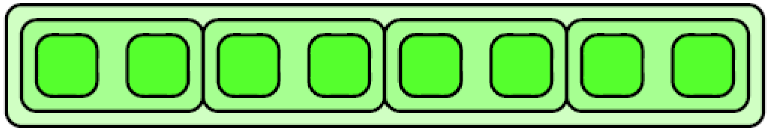
- Spatially regular structure
- Image, grid, also sorting and graphs



Partition

Partition

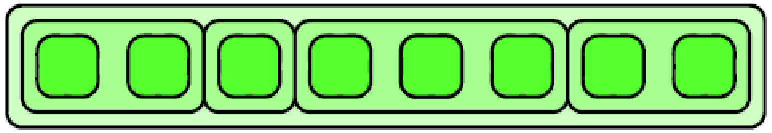
- Non-overlapping sections to avoid write conflicts and race conditions
- Partitions are of equal size
- 1D or multi dimensions
- Combined with map – no problems as it has exclusive access to partition
- Can be further split to allow for nested (hierarchical) parallelism
- Boundary conditions require special treatment
  - partial sections along the edges, special code, but can be commonly parallelized/vectorized
- Cache line size, vector-unit size
  - Related to stencil strip-mining



Segmentation

Segmentation

- Like partition, but sections vary in size
- More complex functions for data manipulation must be used
  - MPI_Scatterv instead of MPI_Scatter, …
- Segmentation along each dimension is possible (kD-tree)
- How to assign segments to processes?



Segmentation

How to distribute N elements to S segments

Problem when S does not divide N

Larger-first approach

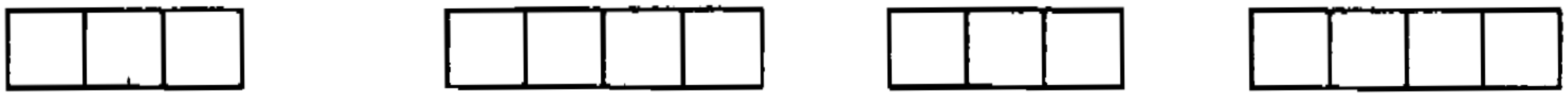
- First $r = N \bmod \mathrm{S}$ segments have one element more, $\lceil N/S \rceil$,
- Other segments are of size $\lfloor N/S \rfloor$
- Index of first element in segment s: $i_L = \lfloor N/S \rfloor s + \min(s, r)$
- Index of last element in segment s: $i_H = \lfloor N/S \rfloor (s+1) + \min(s+1, r) - 1$
- Complex function to determine to which segment belongs element i:

$$s = \min(\lfloor i/(\lfloor N/S \rfloor + 1) \rfloor, \lfloor (i-r)/\lfloor N/S \rfloor \rfloor)$$

Segmentation

Mixed approach

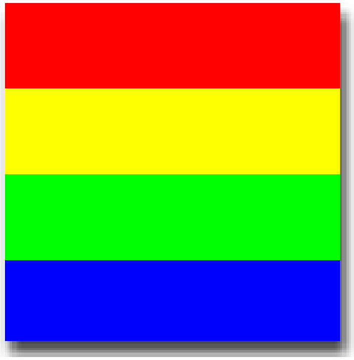
- Larger and smaller segments are mixed
- Index of first element in segment s: $i_L = \lfloor sN/S \rfloor$
- Index of last element in segment s: $i_H = \lfloor (s+1)N/S \rfloor - 1$
- Element i belongs to segment $s = \lfloor (S(i+1)-1)/N \rfloor$

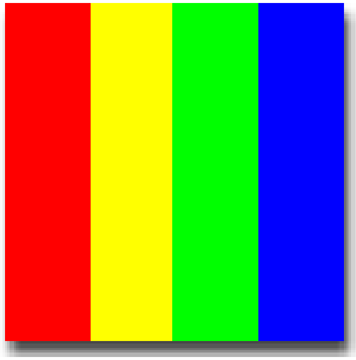


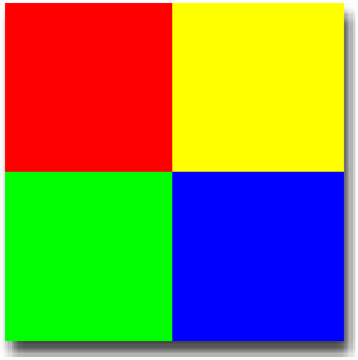
Segmentation

Two dimensions

- Row-wise stripped
- Column-wise stripped
- Checkerboard







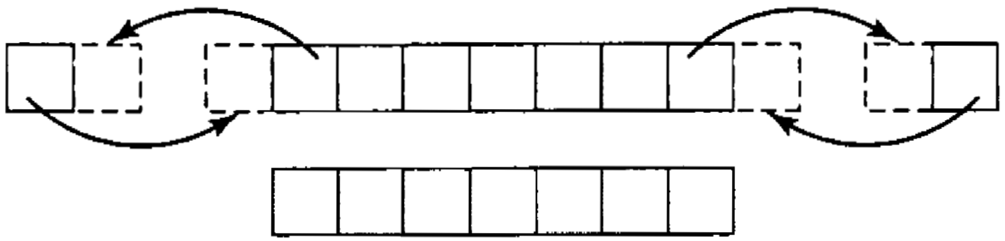
Example: halo exchange

- Square matrix of size N x N
- Exchange of edge elements between neighbouring segments
- Row stripped and column stripped: $2 \times N$
- Checkerboard: $4 \times [N/\sqrt{S}]$

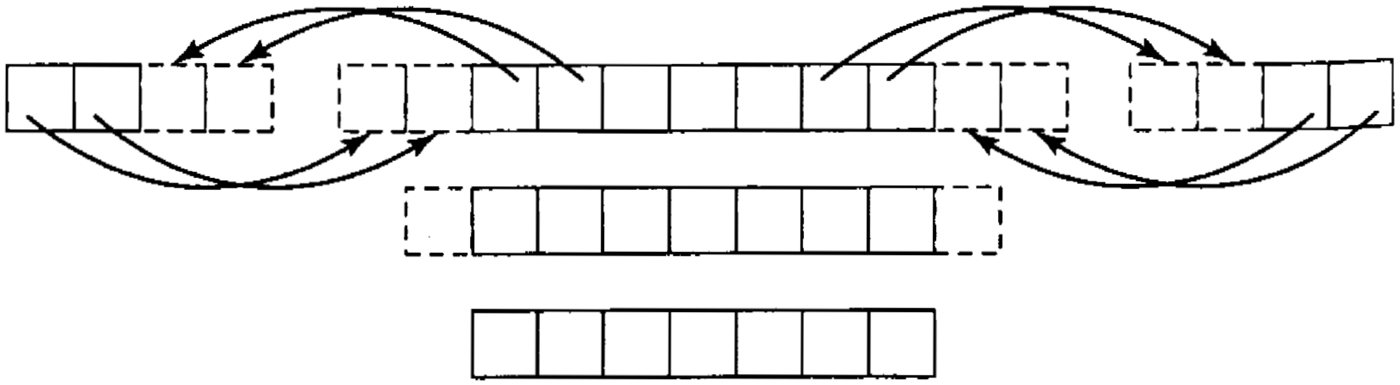
Segmentation

Example: halo exchange 2

- New cell value depends on the values of its neighbours



- Exchanging one element needed for next step od communication
- Exchanging two elements
  - Exchange is needed only on every second step
  - Some additional computation

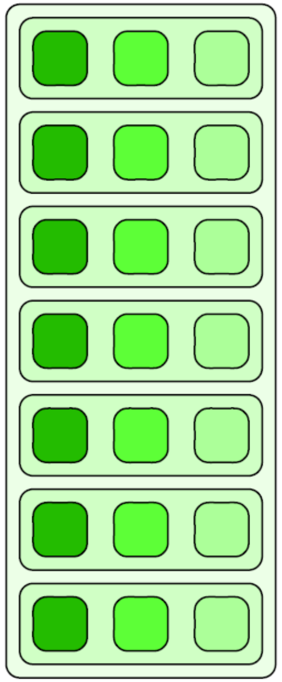


- Latency hiding
  - Initialization of communication cost more than some additional data transfer and computation

Array of structures vs structures of arrays

Common data representation approach (AoS)

- Object-oriented programming
- Declare structures representing some object
  - Vehicle has mass, position, velocity, acceleration, …
- Create collection of that structure
  - Vehicles can be presented as array of vehicle
- Data is not aligned well for transfer, vectorization
- Nice for writing code, also beneficial when data is randomly read

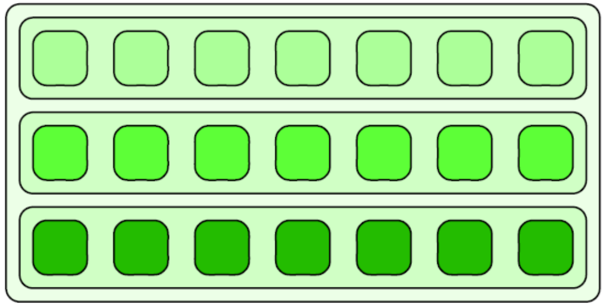


Array of structures vs structures of arrays

For data transfer and vectorization data layout may have to be modified for better performance

Alternative approach (SoA)

- Declare structure of collections
  - Collection of masses, positions, velocities, accelerations, …
- Data is now contiguous, better aligned
- Better way of representing data when majority of data is used



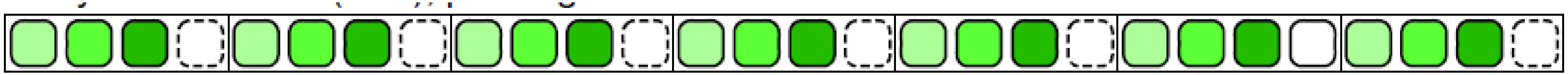
Array of structures vs structures of arrays

Conversion between AoS and SoA is not an easy task

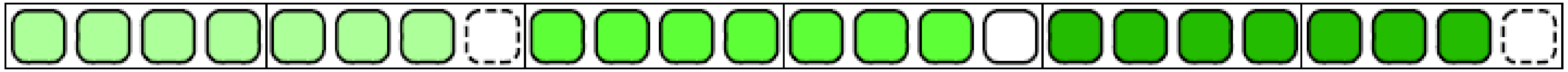
- Significant changes in data structures
- Brakes data encapsulation

Data can be padded for alignment

- Improves data transfer, can be adjusted to cache line, simplifies vectorization
- Important for AoS



- For SoA can be added, but is usually not really needed



MPI: Derived datatypes

Idea

Any data layout can be described with them

Derived from basic MPI datatypes

- For passing data organized as AoS

Allow for efficient transfer of non-contiguous and heterogeneous data

- Example: halo exchange
- During communication MPI datatype tells MPI system where to get the data and where to put it

Both solutions help user to avoid hand-coding

Idea

Libraries should have efficient implementations

- More general datatypes are slower
- No need for MPI_Pack and MPI_Unpack
- Overhead is reduced as only one long message is sent

Derived datatype

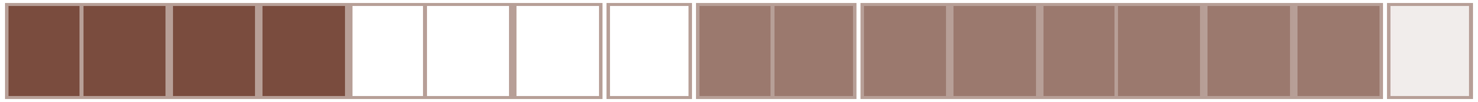
An object used to describe a data layout in memory by

- A sequence of basic datatypes
- A sequence of displacements

Constructed and destroyed during runtime

Structures

- Typemap: pairs of basic MPI datatypes and displacements
  - {(type 0, displacement 0), …, (type N-1, displacement N-1)}
  - {(int, 0}, (double, 8), (char, 16)}



Datatype routines

Construction

- MPI_Type_contiguous: contiguous datatype
- MPI_Type_vector: regularly spaced datatype
- MPI_Type_indexed: variably spaced datatype
- MPI_Type_create_subarray: describes subarray of an array
- MPI_Type_create_struct: general datatype

Commit

- MPI_Type_commit: must be called before new datatype can be used

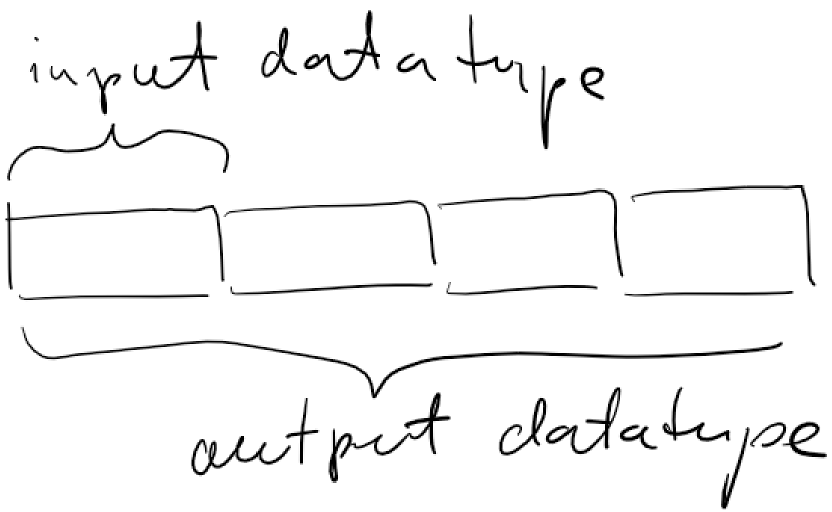
Free

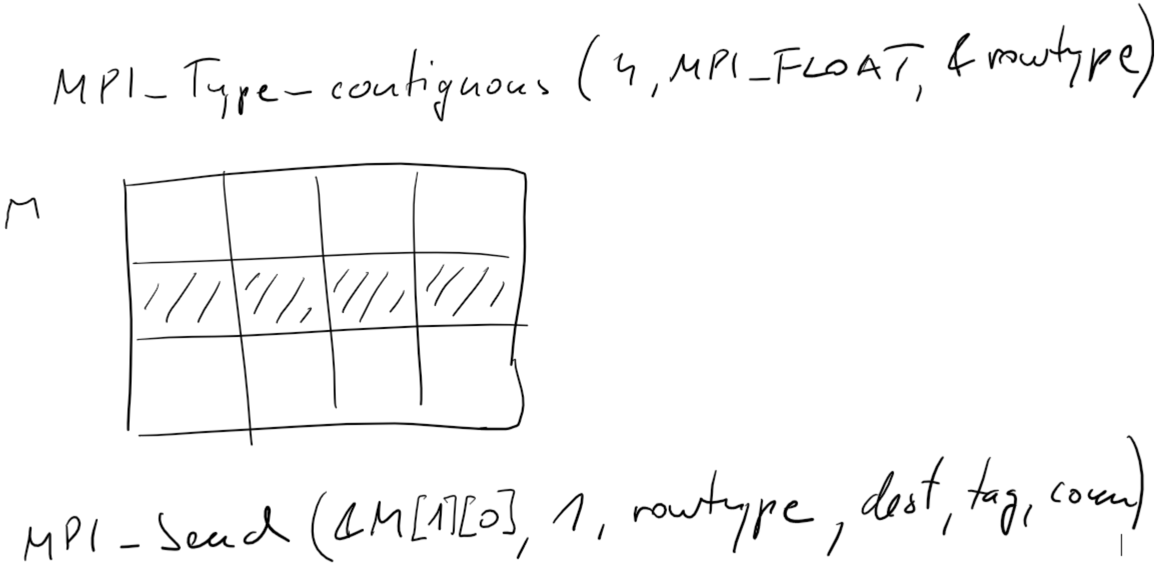
- MPI_Type_free: marks a datatype for deallocation

MPI_Type_contiguous

Output datatype is obtained by concatenating defined number of copies of input datatype.

Constructs a typemap for output datatype consisting of replications of input datatype

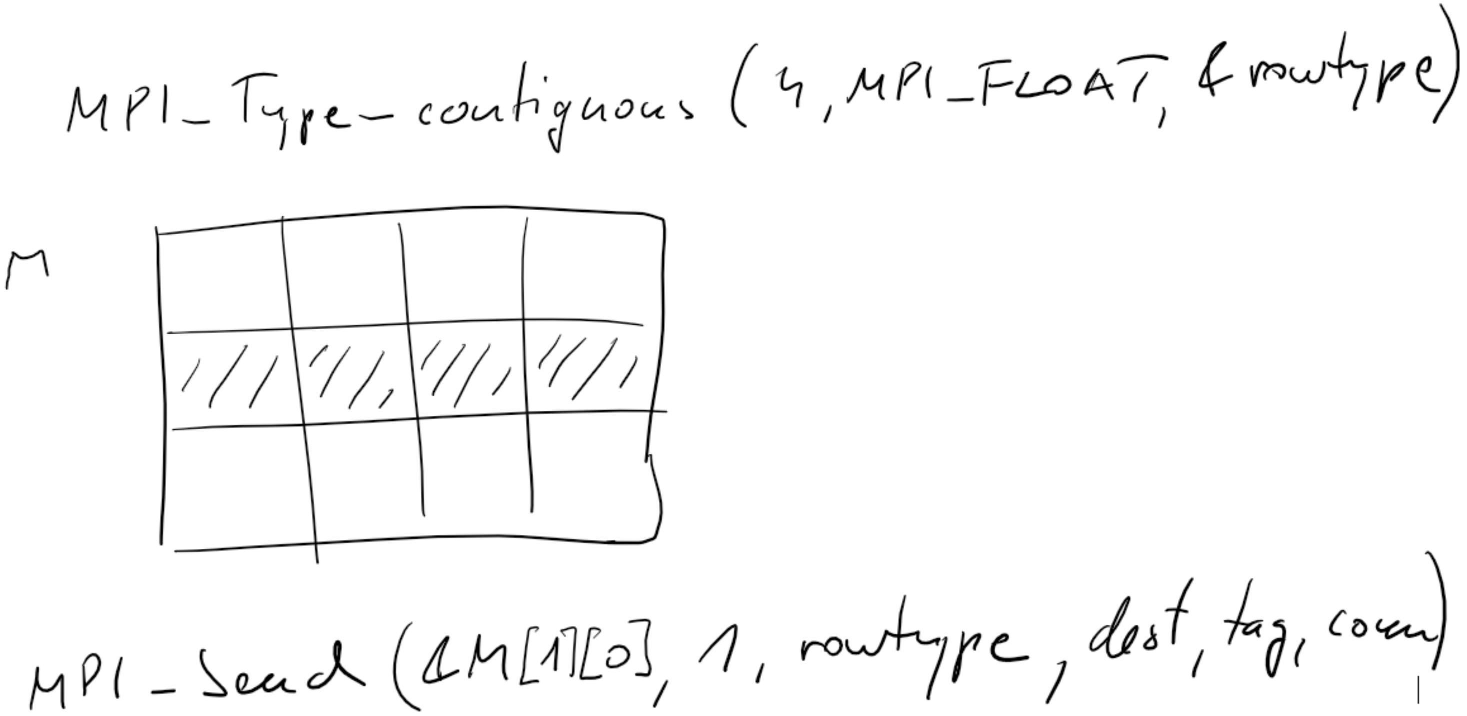




MPI_Type_contiguous

Example: matrix row as a datatype

- Matrix M of size 3 x 4

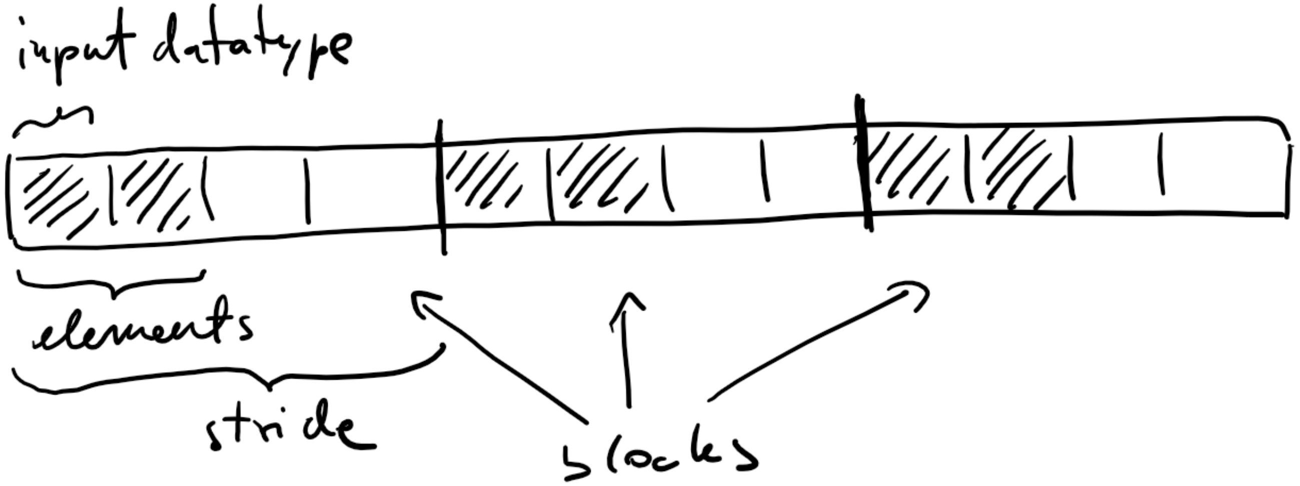


MPI_Type_vector

Similar to MPI_Type_contiuous but with self-defined stride

Input

- Number of blocks
- Number of elements of input datatype in each block
- Stride - number of elements between beginnings of neighbouring blocks

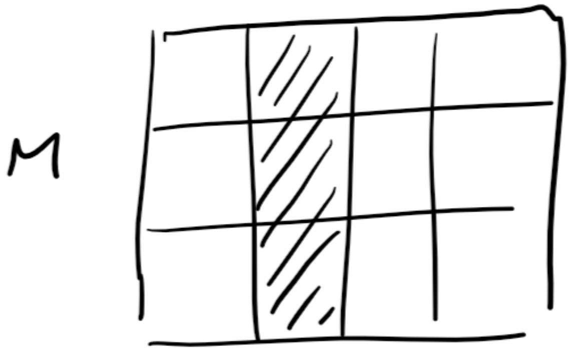


MPI_Type_vector

Example: matrix column as a datatype

- Matrix M of size 3 x 4

MPI_Type_vector(3, 1, 4, MPI_Float, &coltype)

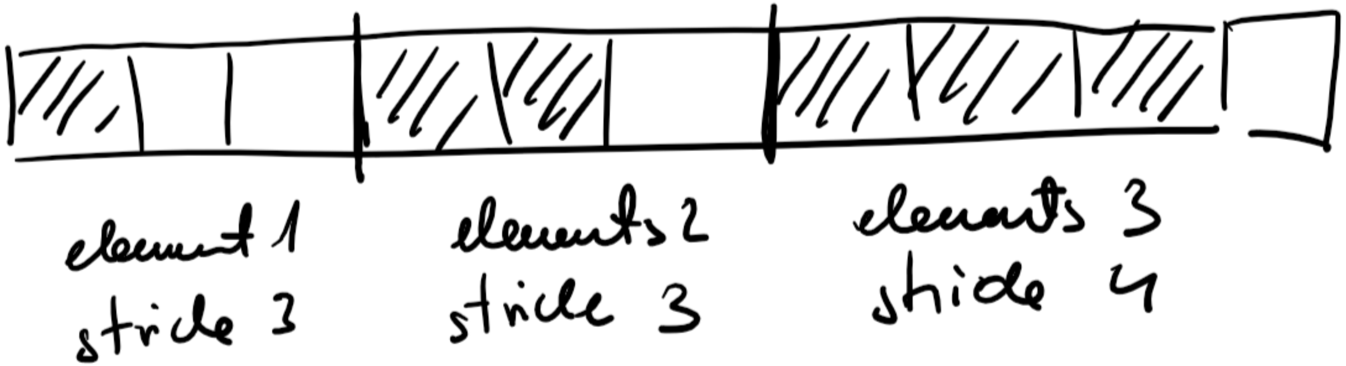


MPI_Send(&M[0][1], 1, coltype, dest, tag, comm)

MPI_Type_indexed

Generalization of MPI_Type_vector

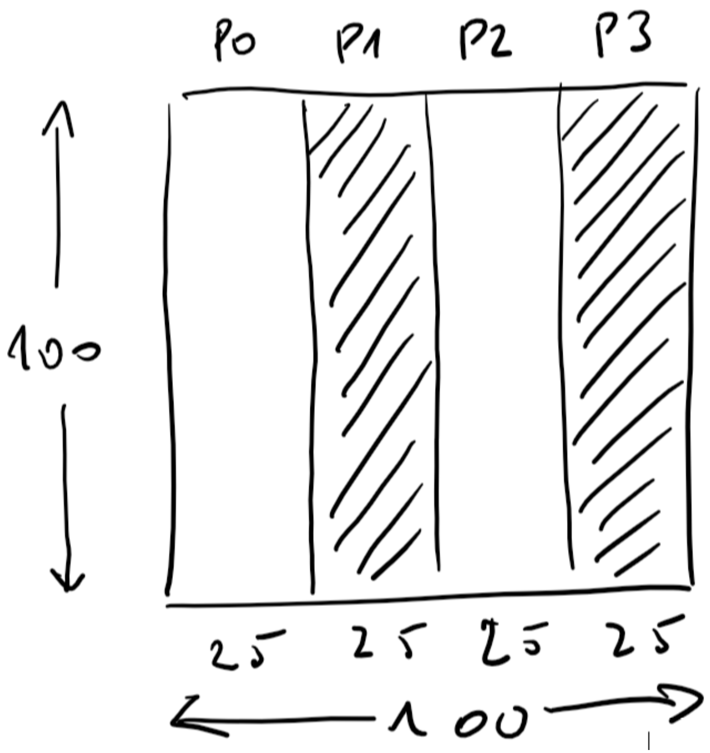
- Number of blocks
- For each block we specify number of elements and stride



MPI_Type_create_subarray

Creates a datatype which is subarray of an array

Useful for column-wise distribution of data




```
double subarray[100][25];
MPI_Datatype filetype;
int sizes[2], subsizes[2], starts[2];
int rank;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

sizes[0]=100; sizes[1]=100;
subsizes[0]=100; subsizes[1]=25;
starts[0]=0; starts[1]=rank*subsizes[1];

MPI_Type_create_subarray(2, sizes, subsizes,
starts, MPI_ORDER_C, MPI_DOUBLE, &filetype);

MPI_Type_commit(&filetype);
```

MPI_Type_create_struct

Typemap

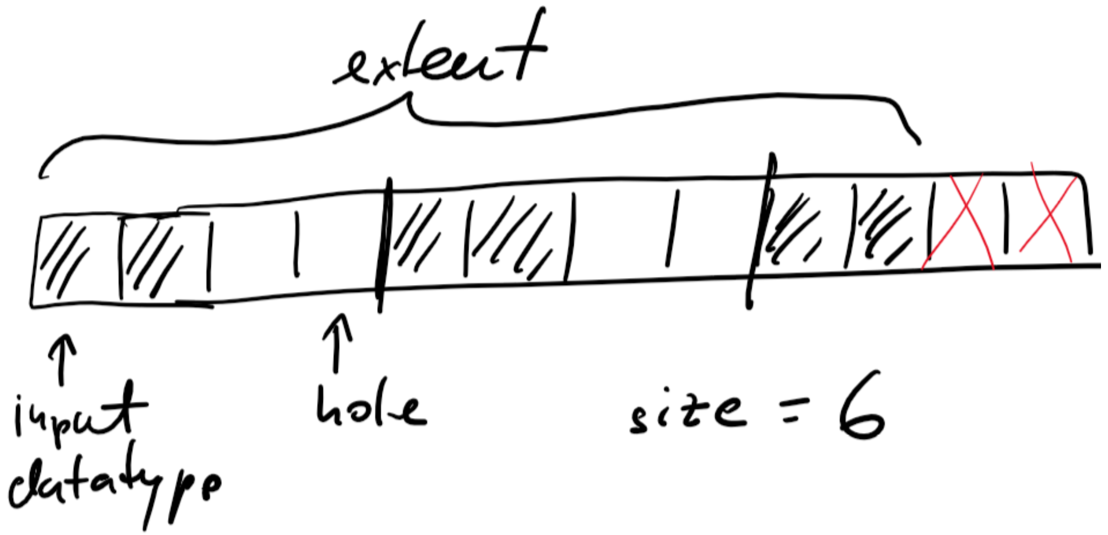
- pairs of basic datatypes and displacements

Extent

- span from the lower to the upper bound
- Inner holes are counted, holes at the end are not!
- Important for alignment of datatypes to data, not for construction and memory allocation
- Query: MPI_Type_get_extent

Size

- Number of bytes that has to be transferred
- Holes are not counted!
- Query: MPI_Type_size



MPI_Type_create_struct

Example

- To get displacements
  - MPI_Type_get_extent
  - MPI_Get_address

```
struct vehicle {
    int mass;
    float position, velocity, acceleration;
};

vehicle Vehicles[NUM];

MPI_Datatype    inputtype[2];
int             blocks[2];
MPI_Aint        displacement[2];
MPI_Aint        lbint, extentint;
MPI_Datatype    typevehicle

inputtype[0] = MPI_INT;
inputtype[1] = MPI_FLOAT;

blocks[0] = 1;
blocks[1] = 3;

MPI_Type_get_extent(MPI_INT, &lbint, &extentint);
displacement[0] = 0;
displacement[1] = blocks[0]*extentint;

MPI_Type_create_struct(2, blocks, displacement, inputtype, &typevehicle);

MPI_Type_commit(&typevehicle);

MPI_Send(Vehicles, NUM, typevehicle, dest, tag, comm);

MPI_Type_free(&typevehicle);
```

MPI_Type_create_resized

Output datatype is identical to the input datatype but lower bound and extent are changed

Useful to correct stride for communication

- Example: zip

When size of MPI datatype and system datatype are not equal, the MPI datatype can be corrected for portability