

Parallel Algorithms with MPI

SPMD

Data Decomposition and Partitioning

Example: Vector addition

The Stencil Pattern with MPI

Example: Heat2D

Bibliography

- [Pacheco]: Peter Pacheco, Matthew Malensek, *Introduction to Parallel Programming*, 2nd Edition, Morgan Kaufmann Publisher, March 2020, Chapter 3
- [GGKK] Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, Chapter 6, Chapter 3
- <https://hpc-tutorials.llnl.gov/mpi/>

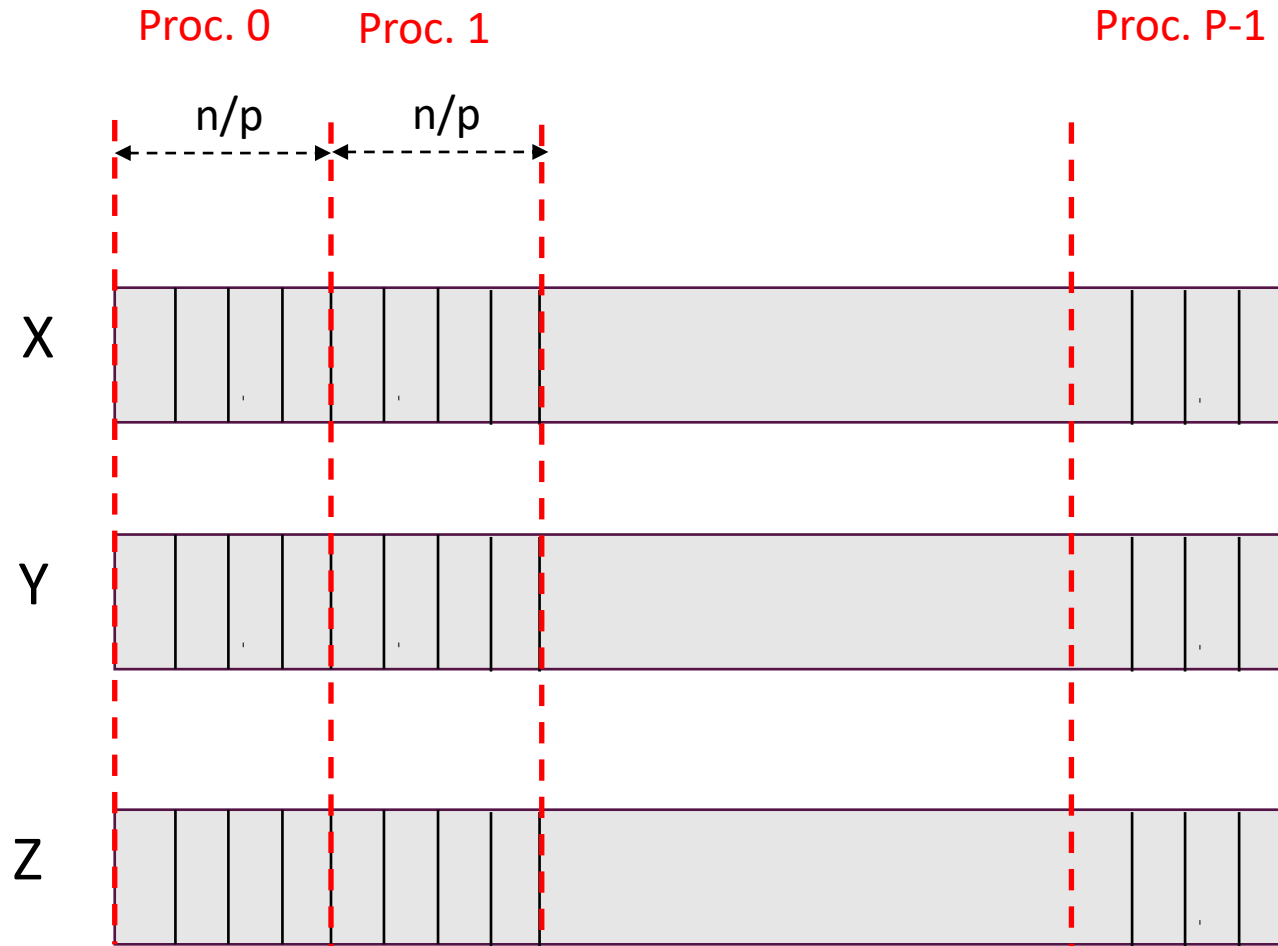
Typical MPI Parallel Programs

- Typical MPI programs are SPMD
- **SPMD** (Single Program Multiple Data): multiple processors simultaneously execute the same program asynchronously
 - This is different than SIMD (Single Instruction Multiple Data) when the same operation (*instruction*) is applied *synchronously* on multiple data to manipulate data streams (a version of SIMD are vectorial machines or GPU-CUDA)
- SPMD programs are designed by applying **Data decomposition**: dividing a data set into discrete chunks that can be operated on in parallel
- Most often SPMD programs belong to the **Data Parallel Model**: Tasks are *statically* mapped to processes and each task performs similar operations on different chunks of data.
- **Data Parallel Model=Data decomposition + static mapping**

Example: Vector addition

- Two vectors x and y of n elements are added and the result is vector z with n elements, $z[i]=x[i]+y[i]$
- Every element of z can be computed independently of other elements of z . It only needs one corresponding element of x and one corresponding element of y ->we partition all 3 vectors across processes
- If number of processes $p \ll n$, each process gets a block of n/p consecutive elements
- Simplifying assumption: size n is divisible by number of processes p . If this condition is not fulfilled, terminate.
- With MPI: can find out the number of processes from the communicator size ($p==comm_sz$)

Vector addition - Partitioning



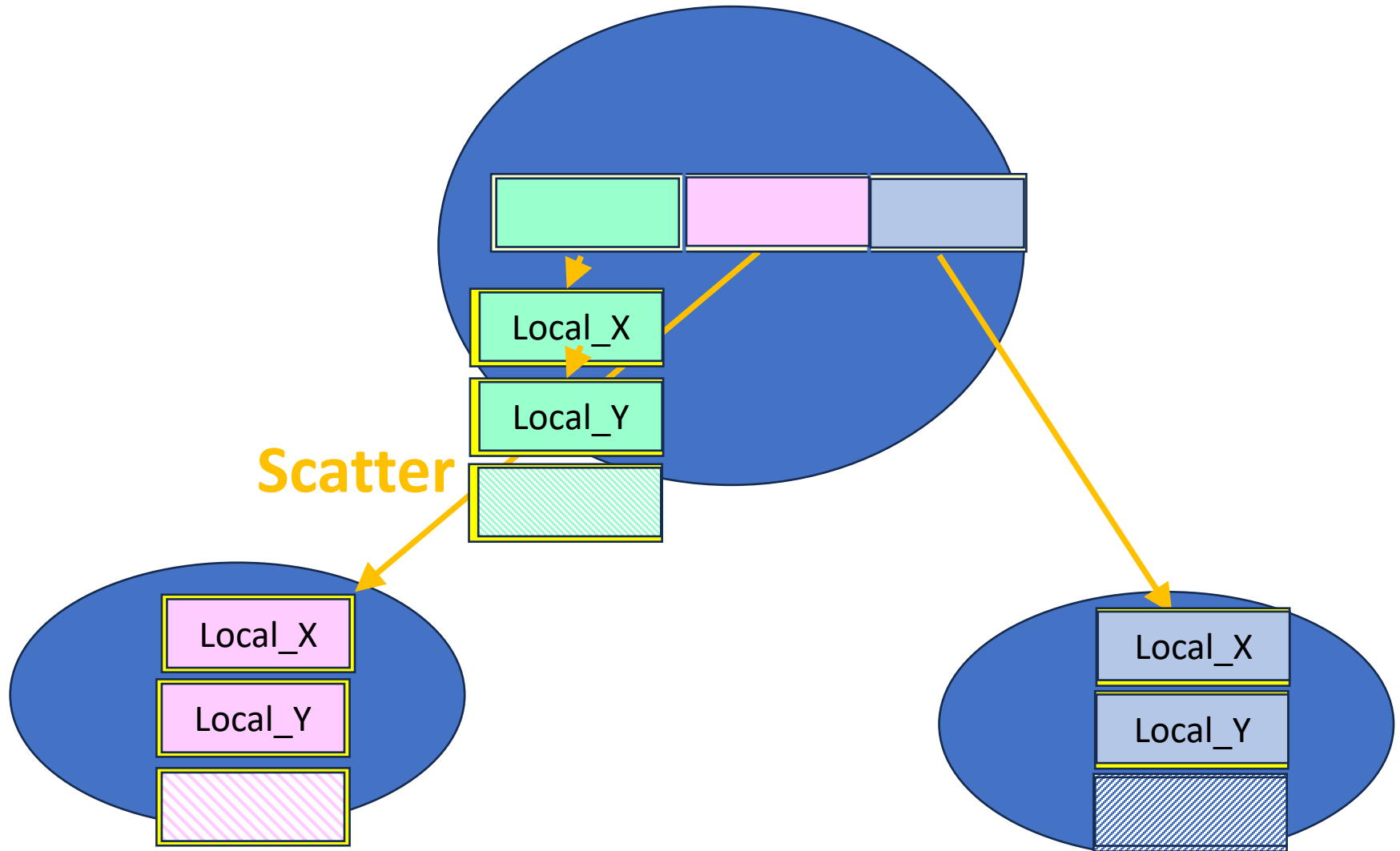
Vector addition – parallel with Distributed Memory

- Master Process reads input data(size) and reads or generates vectors x and y
- Master Process sends data (size information) to all processes
- Master Process sends a chunk of size n/p from vectors x and y to each process.
- Each Process i receives a chunk of size n/p
- Each Process i computes vector addition on its local chunks
- Each Process i sends its chunk of result vector z to master process
- Master Process receives chunks of result vector from all processes
- Master Process may print result vector

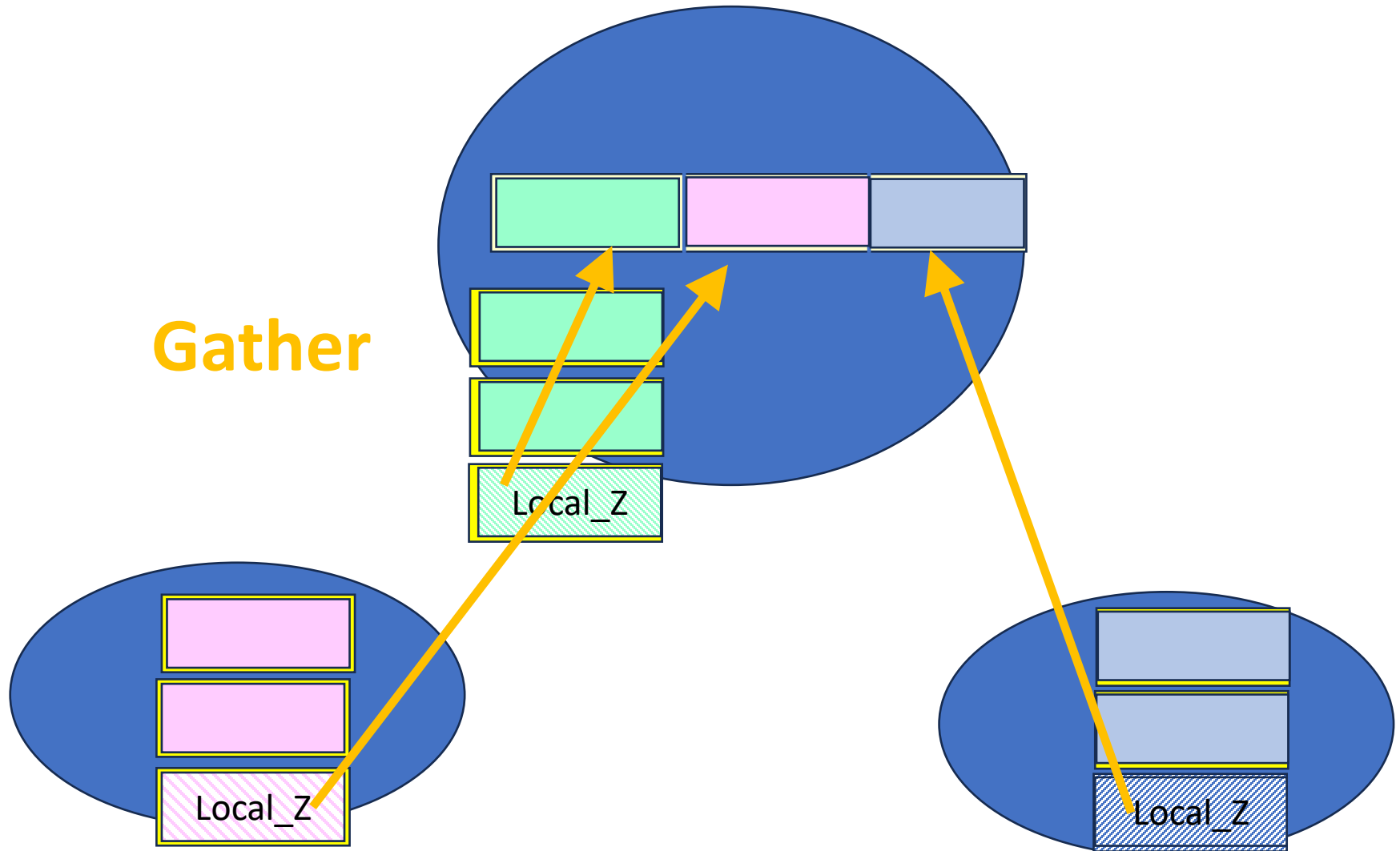
Vector addition – Efficient use of collective communications

- Master Process reads input data(size) and reads or generates vectors x and y
- Master Process sends data (size information) to all processes - **MPI_Bcast**
- Master Process sends a chunk of size n/p from vectors x and y to each process – **MPI_Scatter**
- Each Process i receives a chunk of size n/p (as part of the **MPI_Scatter operation**)
- Each Process i computes vector addition on its local chunk of z
- Each Process i sends its chunk of result vector z to master process – **MPI_Gather**
- Master Process receives chunks of result vector from all processes (as part of the **MPI_Gather operation**)
- Master Process may print result vector

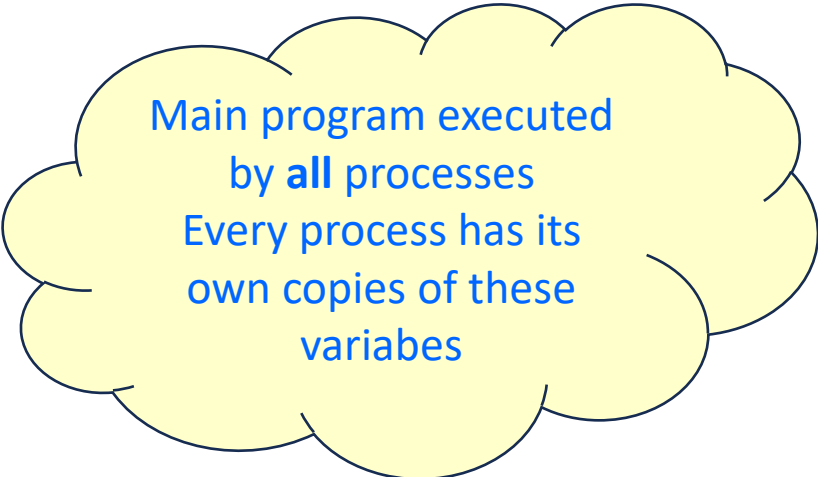
Adding Vectors with Scatter and Gather



Adding Vectors with Scatter and Gather



```
int main(void) {  
    int n, local_n;  
    int comm_sz, my_rank;  
    double *local_x, *local_y, *local_z;  
    MPI_Comm comm;  
  
    MPI_Init(NULL, NULL);  
    comm = MPI_COMM_WORLD;  
    MPI_Comm_size(comm, &comm_sz);  
    MPI_Comm_rank(comm, &my_rank);  
  
    Read_n(&n, &local_n, my_rank, comm_sz, comm);  
  
    Allocate_vectors(&local_x, &local_y, &local_z, local_n, comm);  
  
    Read_vector(local_x, local_n, n, "x", my_rank, comm);  
  
    Read_vector(local_y, local_n, n, "y", my_rank, comm);  
  
    Parallel_vector_sum(local_x, local_y, local_z, local_n);  
  
    Print_vector(local_z, local_n, n, "The sum is", my_rank, comm);  
  
    MPI_Finalize();  
  
    return 0;  
} /* main */
```



Main program executed
by **all** processes
Every process has its
own copies of these
variables

```

void Read_n(
    int*      n_p      /* out */,
    int*      local_n_p /* out */,
    int       my_rank   /* in  */,
    int       comm_sz   /* in  */,
    MPI_Comm  comm      /* in  */) {
    int local_ok = 1;
    char *fname = "Read_n";

```

```

    if (my_rank == 0) { /* Master process reads data */
        printf("What's the order of the vectors?\n");
        scanf("%d", n_p);
    }

```

```

    /* Master broadcasts and all others receive */
    MPI_Bcast(n_p, 1, MPI_INT, 0, comm);

```

```

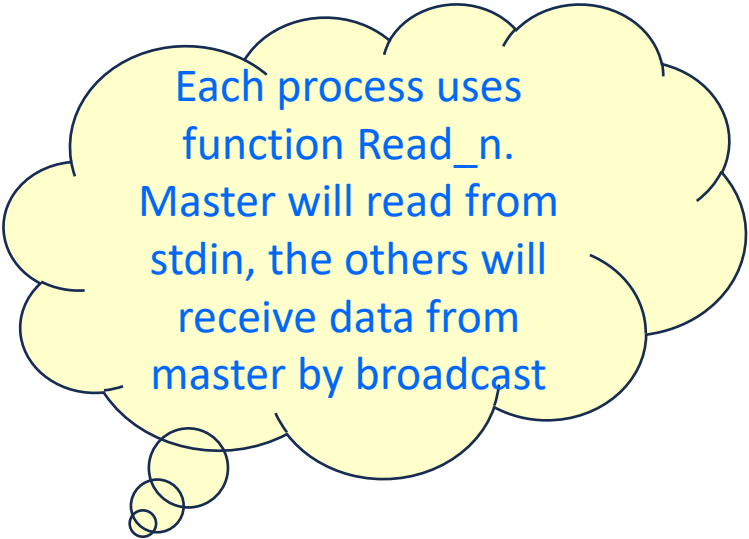
    if (*n_p <= 0 || *n_p % comm_sz != 0) local_ok = 0;
    Check_for_error(local_ok, fname,
        "n should be > 0 and evenly divisible by comm_sz", comm);

```

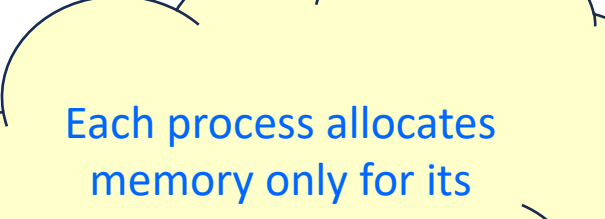
```

    *local_n_p = *n_p/comm_sz;
} /* Read_n */

```



Each process uses
function Read_n.
Master will read from
stdin, the others will
receive data from
master by broadcast



Each process allocates
memory only for its
own vector chunks

```
if (*local_x_pp == NULL || *local_y_pp == NULL ||
    *local_z_pp == NULL) local_ok = 0;
Check_for_error(local_ok, fname, "Can't allocate local vector(s)",
    comm);
/* Allocate_vectors */
```

```

void Read_vector(
    double    local_a[]    /* out */,
    int       local_n      /* in  */,
    int       n            /* in  */,
    char      vec_name[]   /* in  */,
    int       my_rank      /* in  */,
    MPI_Comm  comm         /* in  */) {

```

```

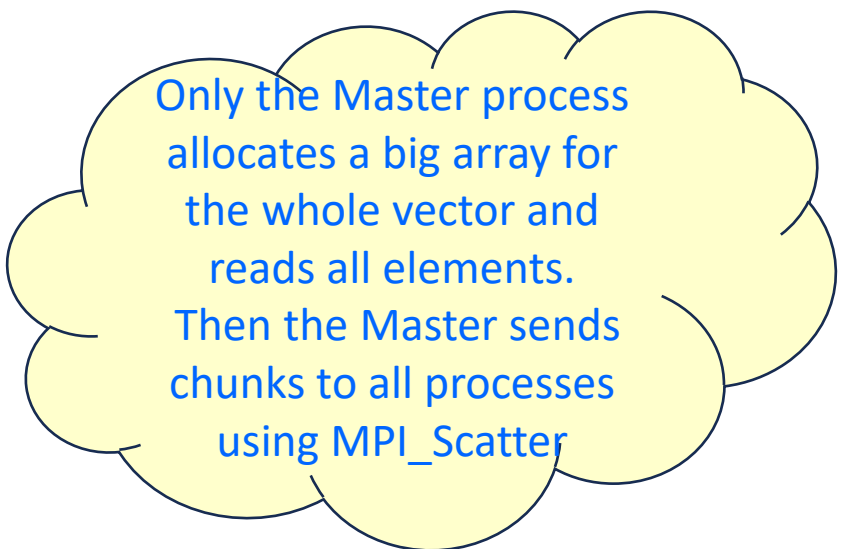
double* a = NULL;
int i;
int local_ok = 1;
char* fname = "Read_vector";

```

```

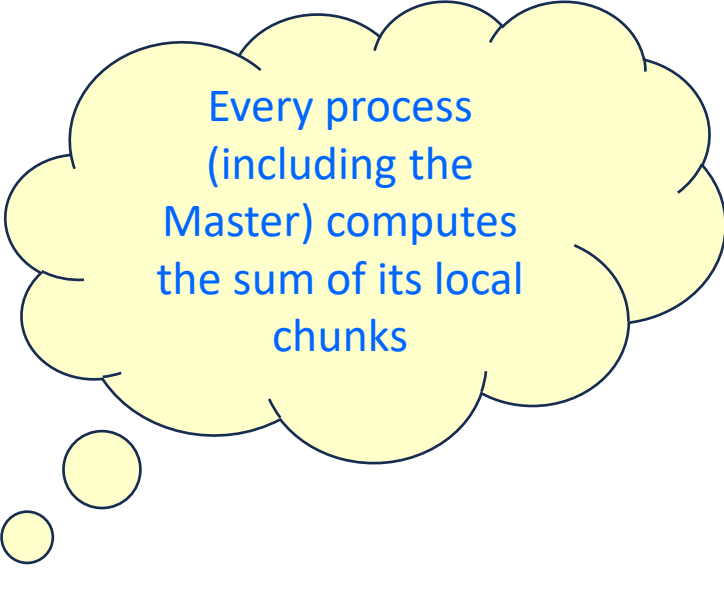
if (my_rank == 0) {
    a = malloc(n*sizeof(double));
    printf("Enter the vector %s\n", vec_name);
    for (i = 0; i < n; i++)
        scanf("%lf", &a[i]);
    MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0,
               comm);
    free(a);
} else {
    MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0,
               comm);
}
} /* Read_vector */

```



Only the Master process
allocates a big array for
the whole vector and
reads all elements.
Then the Master sends
chunks to all processes
using MPI_Scatter

```
void Parallel_vector_sum(  
    double local_x[] /* in */,  
    double local_y[] /* in */,  
    double local_z[] /* out */,  
    int local_n /* in */) {  
    int local_i;  
  
    for (local_i = 0; local_i < local_n; local_i++)  
        local_z[local_i] = local_x[local_i] + local_y[local_i];  
  
} /* Parallel_vector_sum */
```



Every process
(including the
Master) computes
the sum of its local
chunks

```

void Print_vector(
    double    local_b[]    /* in */,
    int       local_n      /* in */,
    int       n            /* in */,
    char      title[]      /* in */,
    int       my_rank      /* in */,
    MPI_Comm  comm         /* in */) {

    double* b = NULL;
    int i;
    int local_ok = 1;
    char* fname = "Print_vector".

    if (my_rank == 0) {
        b = malloc(n*sizeof(double));
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
                    0, comm);
        printf("%s\n", title);
        for (i = 0; i < n; i++)
            printf("%f ", b[i]);
        printf("\n");
        free(b);
    } else {
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0,
                    comm);
    }
} /* Print_vector */

```

Only the Master process
allocates a big array for
the whole vector.

Then all processes send
their chunks to the
master process using
MPI_Gather

Source code

- staff.cs.upt.ro/~ioana/apd/mpi/mpi_vector_add.c

Data Decomposition and Partitioning

Data Decomposition

- Identify the data on which computations are performed.
- Partition this data across various tasks.
- This partitioning induces a decomposition of the problem.
- Data can be partitioned in various ways - this critically impacts performance of a parallel algorithm.
- Different categories of data can be partitioned:
 - Output data
 - Input data
 - Input and output data

Output Data Decomposition

- Often, each element of the output can be computed independently of others (but simply as a function of the input)
- A partition of the output across tasks decomposes the problem naturally.
- Examples:
 - Matrix Multiplication: Each output element of a matrix multiplication is obtained by multiplying a row by a column of the input matrices

Input Data Decomposition

- Input data decomposition can be applied when the algorithm is a one-to many function
- Example:
 - A search function may take an array as input and look for the occurrence of a certain element. Input array is divided in chunks. Partial search results must be added.
- In many cases, this is the only natural decomposition because the output is not clearly known a-priori
- A task is associated with each input data partition. The task performs as much of the computation with its part of the data. Subsequent processing is needed to combine these partial results.

Input and Output Data Decomposition

- Both input and output data can be partitioned: ideally when a subset of the output can be computed as a function of a subset of the input
- Example:
 - *Vector addition*: both the vector operands and the vector result are partitioned in chunks
 - Image processing: Each output pixel of an image convolution is obtained by applying a filter to a region of input pixels

The Owner Computes Rule

- The *Owner Computes Rule* generally states that the process assigned a particular data item is responsible for all computation associated with it.
- In the case of input data decomposition, the owner computes rule implies that all computations that use the input data are performed by the process.
- In the case of output data decomposition, the owner computes rule implies that the output is computed by the process to which the output data is assigned.

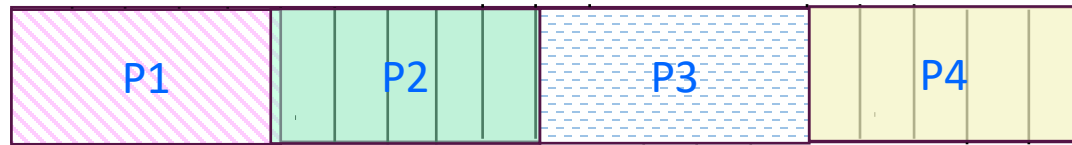
Static Mapping based on Data Partitioning

- We can combine data partitioning with the ``owner-computes'' rule to partition the computation into subtasks.

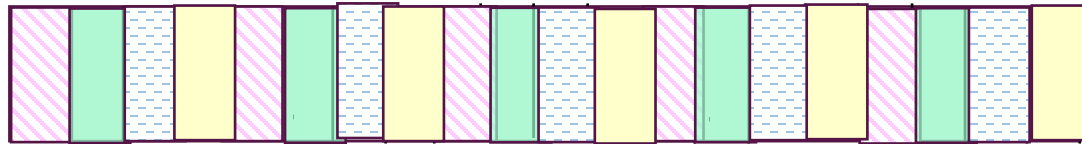
Partitioning options: how we distribute data partitions to processes

- Block partitioning
 - Assign blocks of consecutive components to each process.
- Cyclic partitioning
 - Assign components in a round robin fashion. In this way, a process gets “samples” from different regions. This option is useful when the amount of computations is not the same in all points of the data block. Example: a vector where every element at index k is processed in a loop of k iterations
- Block-cyclic partitioning
 - Use a cyclic distribution of blocks of components.

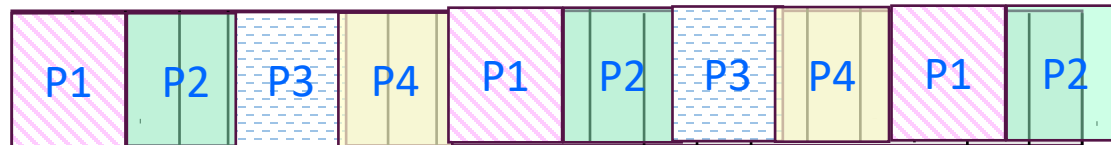
Data Partitioning variants for a vector



Block



Cyclic



Block-Cyclic

Data Partitioning variants for a matrix (a 2D-array)

- For a matrix (a 2D-array), we can do a 1D or 2D decomposition scheme
- All options are valid: Block, Cyclic, BlockCyclic

Block Partitioning for a matrix: 1D block array distribution schemes

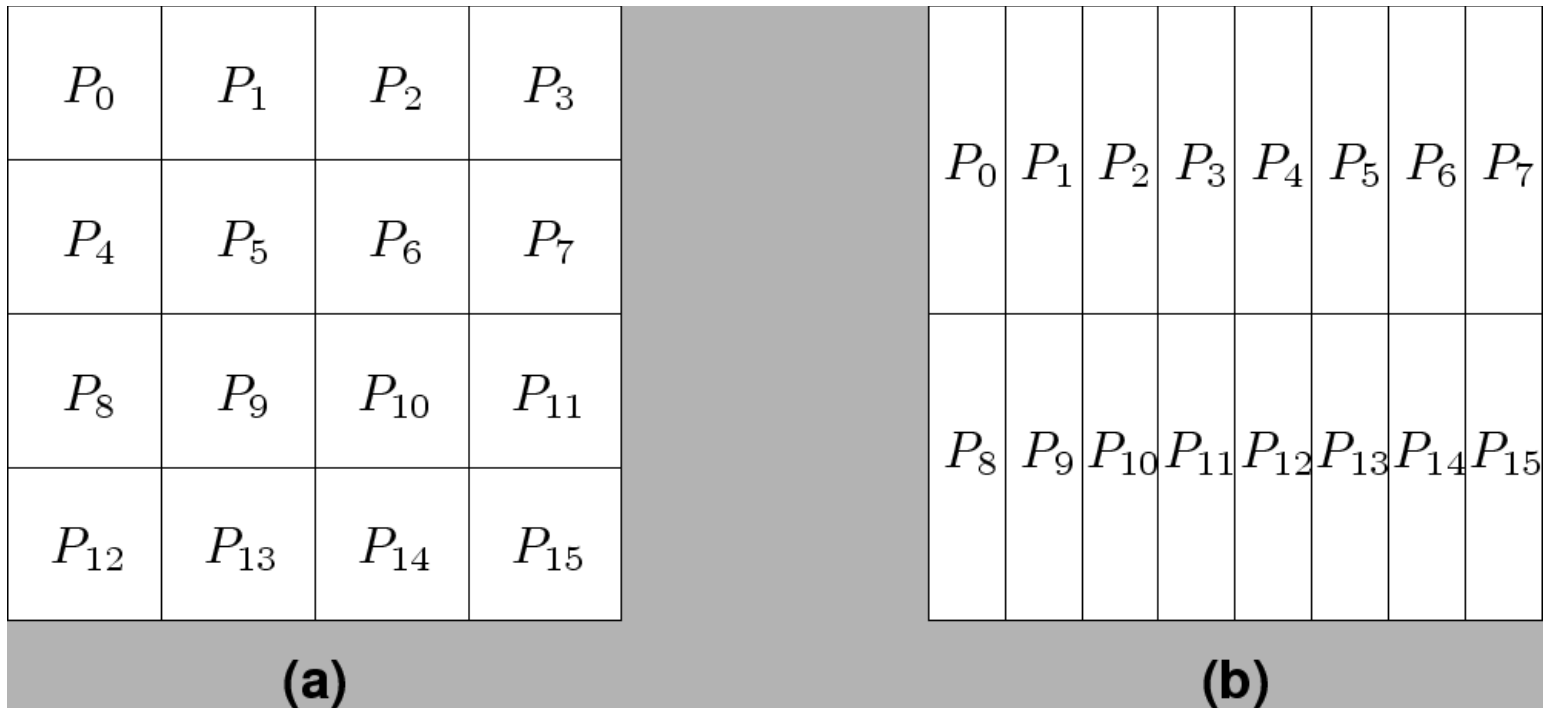
row-wise distribution

P_0
P_1
P_2
P_3
P_4
P_5
P_6
P_7

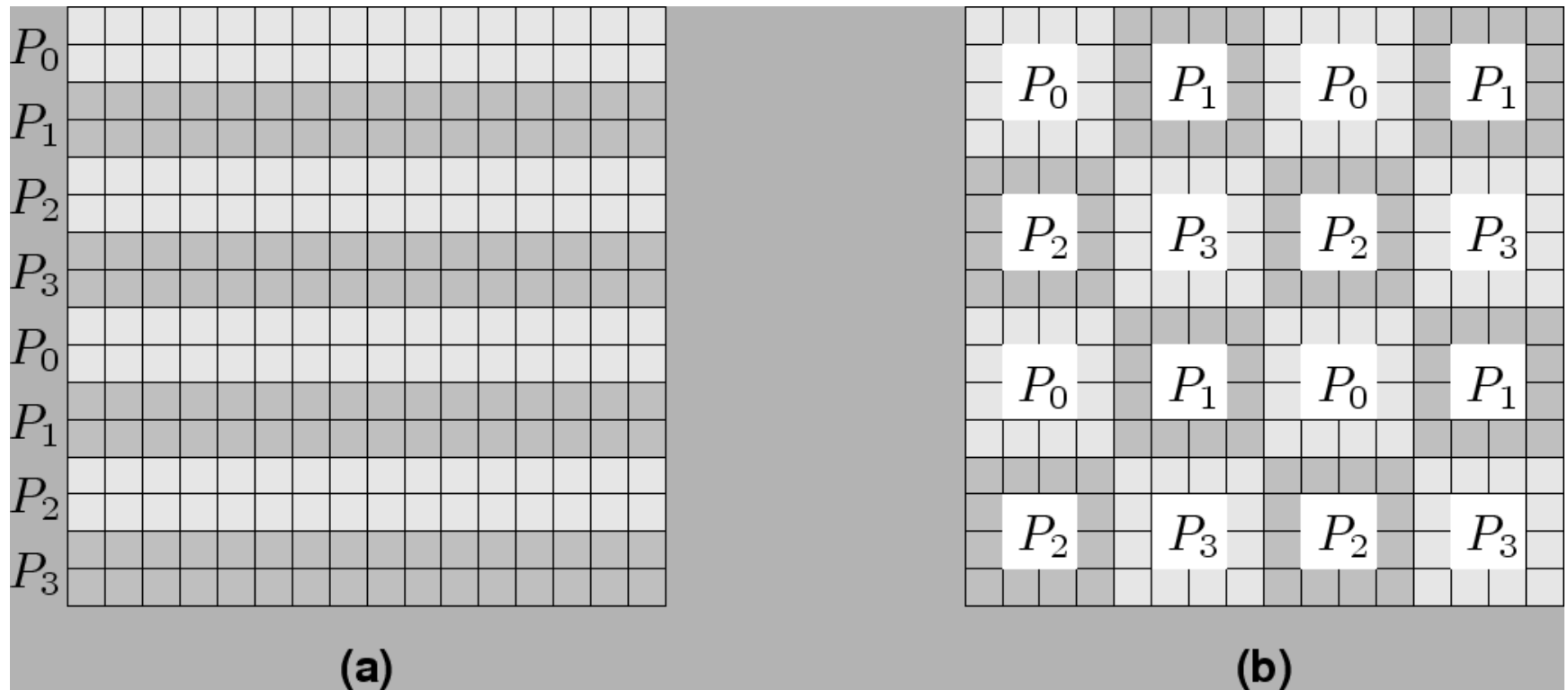
column-wise distribution

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
-------	-------	-------	-------	-------	-------	-------	-------

Block Partitioning for a matrix: 2D block array distribution schemes



Block Partitioning for a matrix: 1D and 2D Block-cyclic array distribution schemes



Block vs Block-cyclic distributions

- Block distribution: a process owns a contiguous area of the matrix
 - Easy to distribute blocks to processes
 - A process can easily access all its data
- Block-cyclic: a process owns samples from different regions of the matrix
 - Good for load-balancing IF different regions of the matrix require different kinds of processing
 - Difficult for data access: if a process needs to access “neighbor” elements in the matrix they will be owned by another process

The Stencil Pattern with MPI

The Stencil pattern

- *Stencils* compute each element of an output array as a function of neighbor elements from the input array
- For 2D array: the output element (x,y) is computed as a function of the input elements *above, below, to the left, and to the right*, that is, $(x,y-1)$, $(x,y+1)$, $(x-1,y)$, $(x+1,y)$, although any pattern is possible involving more neighbors
- Stencil loop: the stencil is applied to all elements of the array
- Stencil loops are often embedded in nested loops (array is processed in many successive generations)
- Stencil loops are very frequent in a number of applications:
 - scientific computing: for solving PDEs discretized with Finite Difference (FD) or Finite Volume (FV) methods.
 - Image processing: blur, edge detection

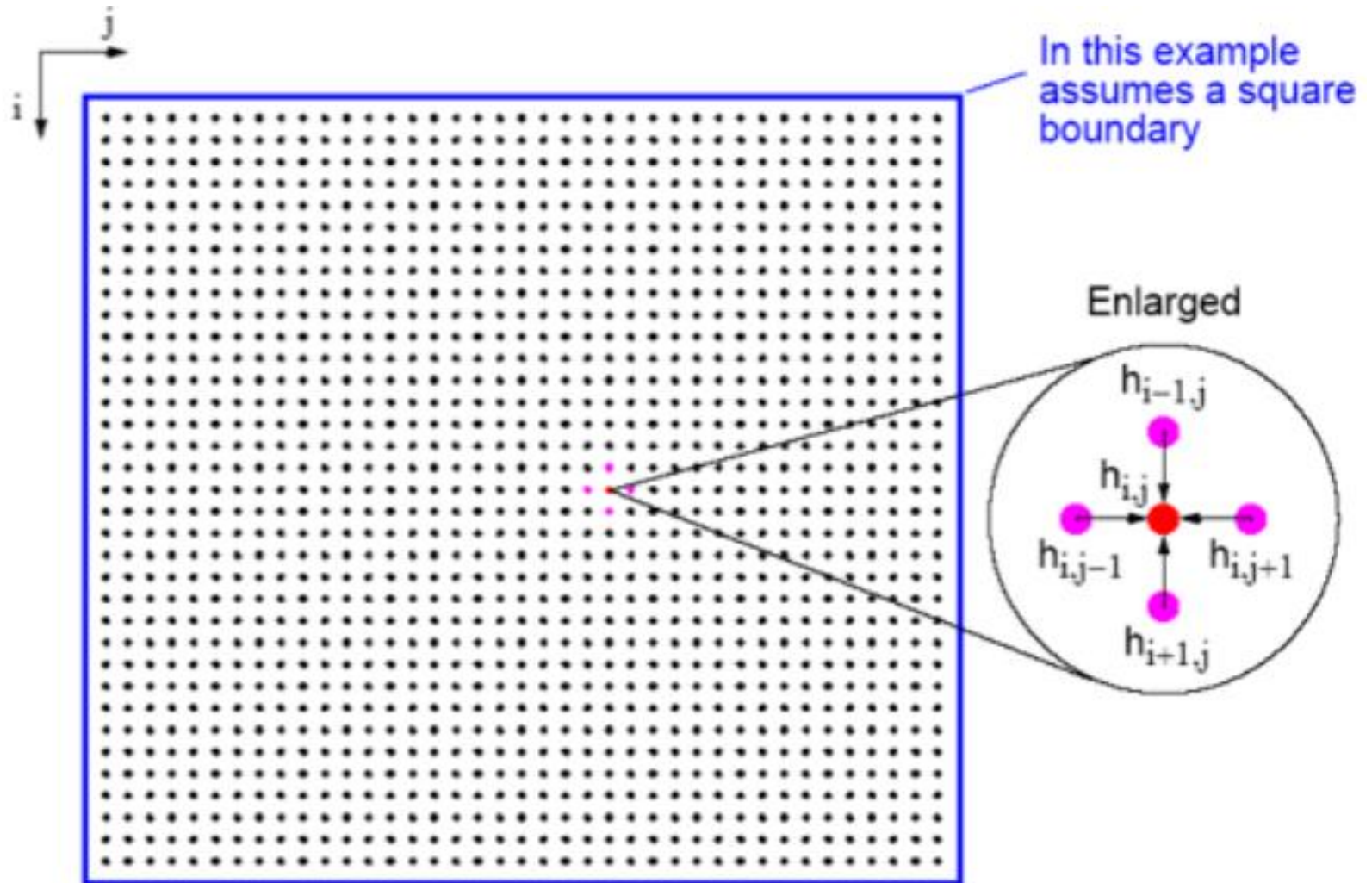
The Heat Distribution Problem - Serial

```
void serial_temp()
{
    int i, j, time;
    int current, next;
    current = 0;
    next = 1;

    for (time = 0; time < MAXITER; time++)
    {
        for (i = 1; i < N - 1; i++) // iterate grid but skip boundary
            for (j = 1; j < N - 1; j++)
                temp[next][i][j] = (temp[current][i + 1][j] +
                                     temp[current][i - 1][j] +
                                     temp[current][i][j + 1] +
                                     temp[current][i][j - 1]) *
                                     0.25;

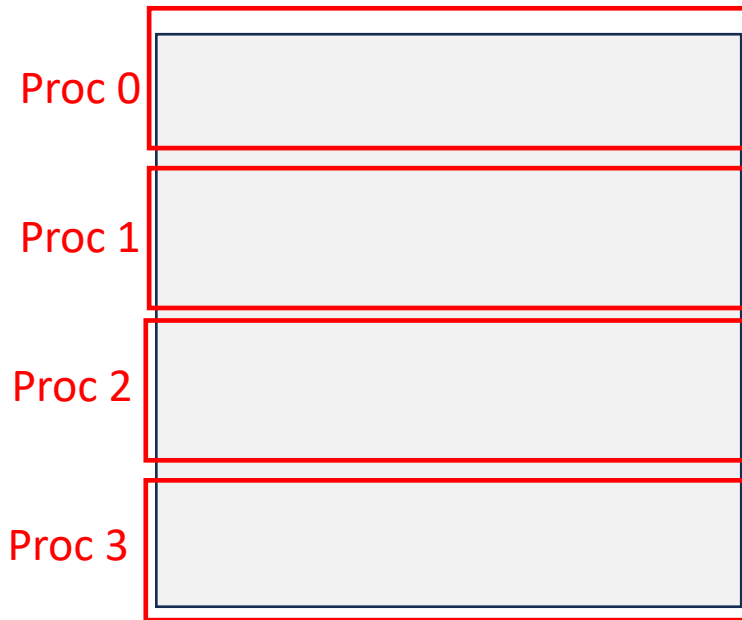
        current = next; // toggle between current and next grid
        if (current == 0) next = 1; else next = 0; //avoid copying arrays!!
    }
}
```


Heat 2D

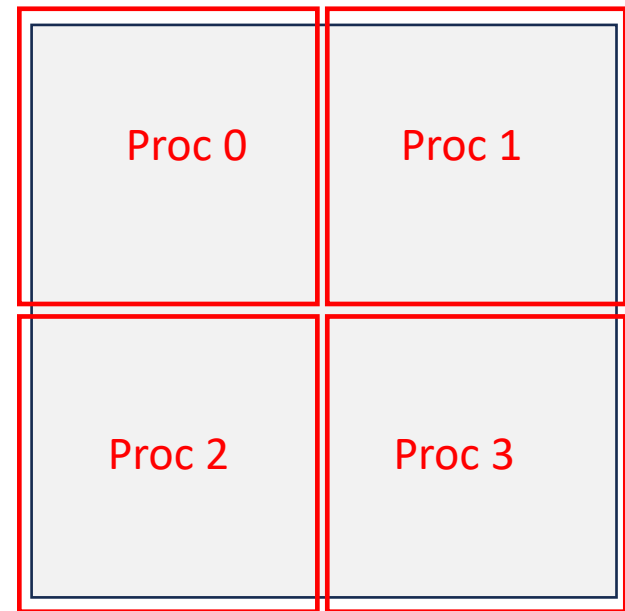


Block Distribution variants:

1D Decomposition



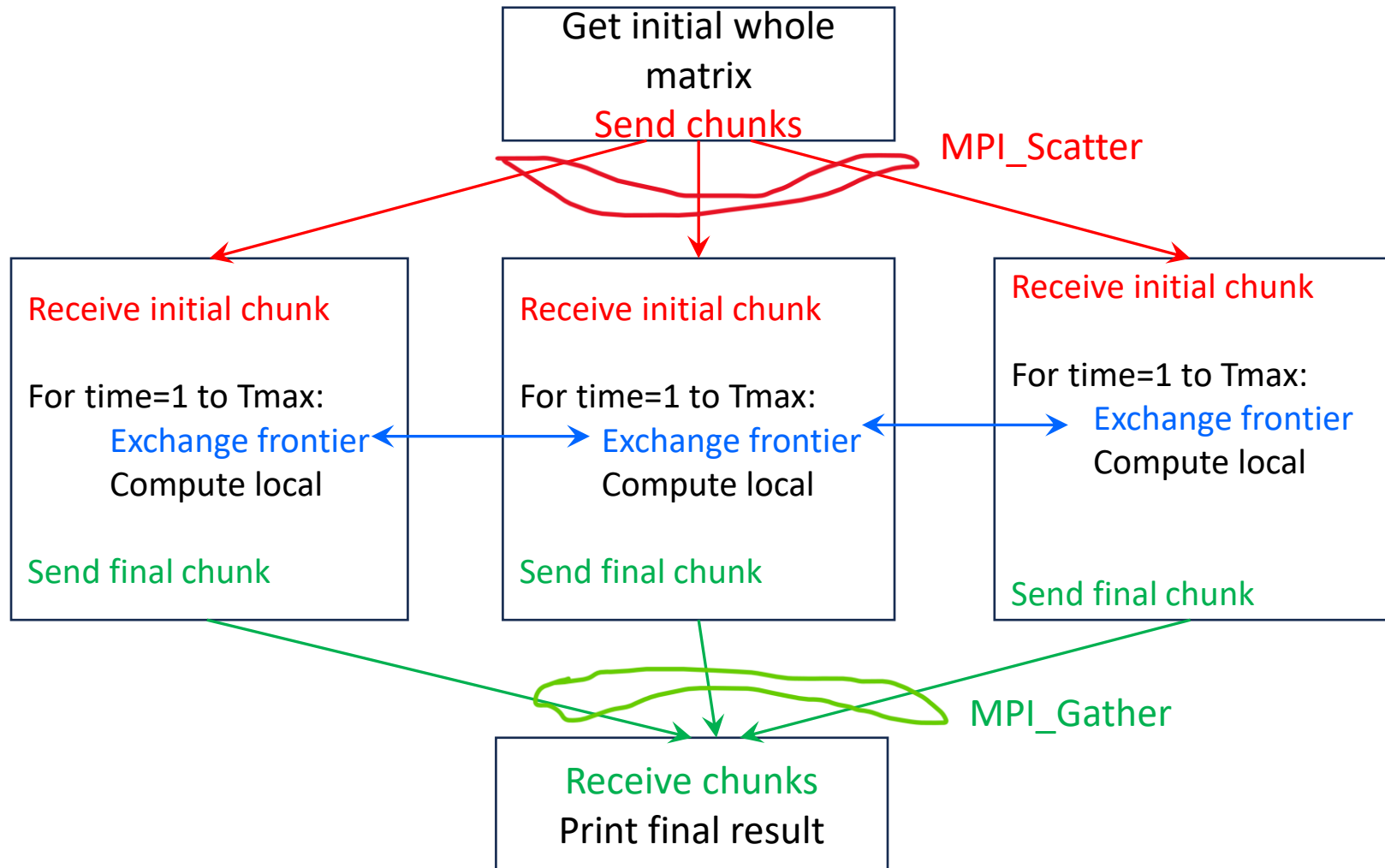
2D Decomposition



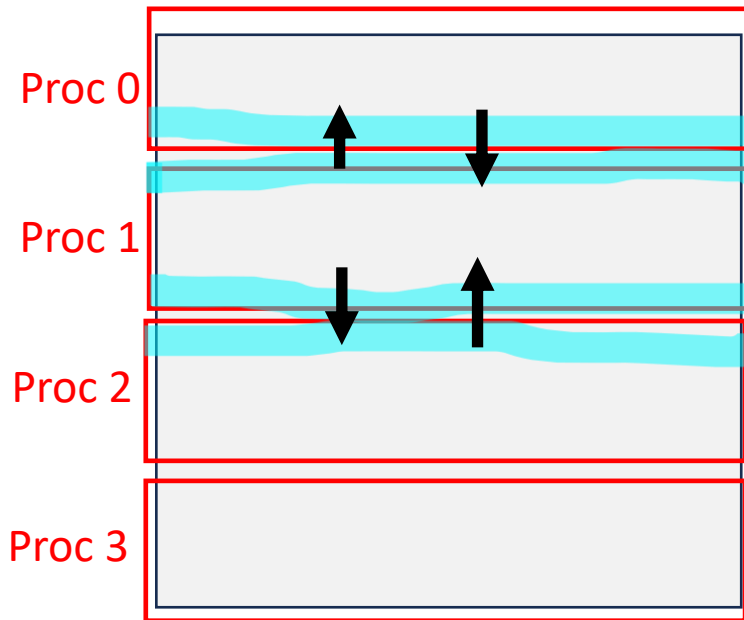
Parallel Stencil Pattern

- Each process owns a chunk of data and performs computations on it
- For the elements on the frontier of its chunk, every process needs neighbour elements from other processes
- Send-Receive operations in order to exchange data should be done "in bulk" due to performance reasons: exchange all neighbours on a frontier at once, not each element in an individual message
- If the stencil loop must be repeated for a number of time iterations, all processes must synchronize with a barrier at the end of every generation

Stencil Pattern with MPI – Task Interactions



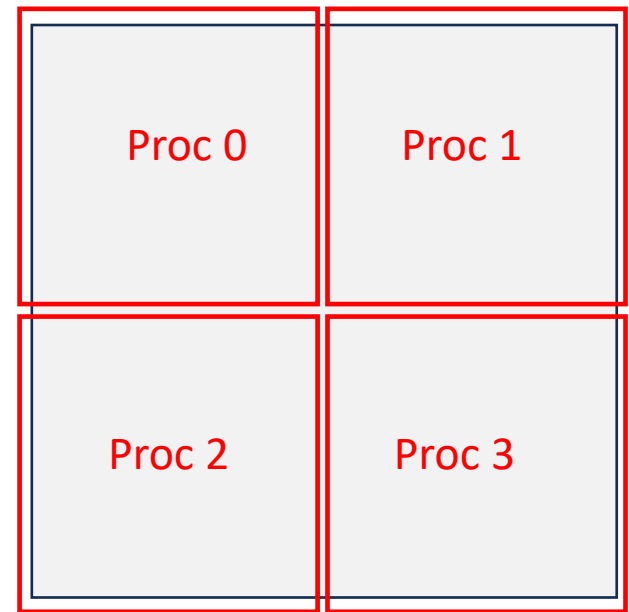
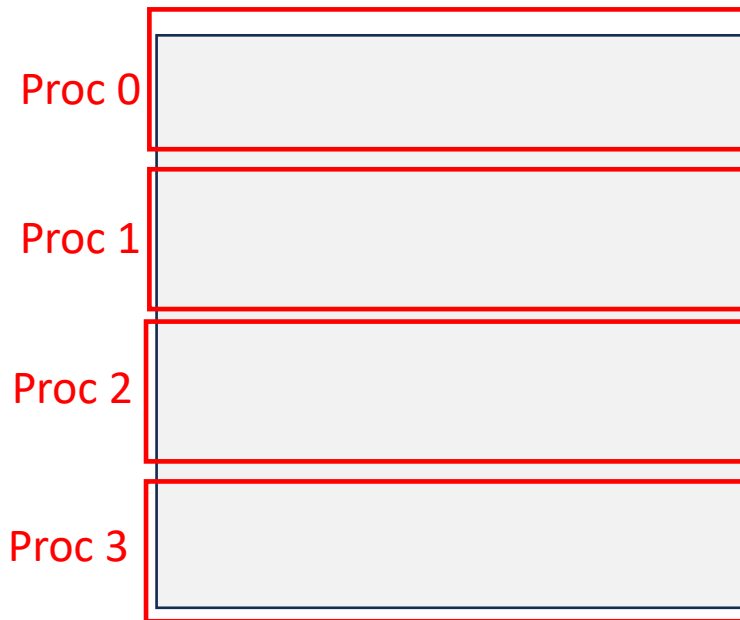
Frontier exchanges



- Frontier exchanges must be done directly between neighbour processes – do NOT involve master process in these exchanges!
- The chunk owned by each process can be extended with “ghost” cells to hold values from neighbour processes
- `MPI_Sendrecv` – simultaneously exchanges data between 2 processes

Which decomposition is better?

- We can have p processes. The matrix is $N \times N$.
- Is there a difference in performance: 1D vs 2D decomposition?
- Each process gets *the same number of elements*, $N \times N / p$ elements, but on a region of a different shape with a *different frontier length*
- 1D: size of chunk: $N \times N / p$. Exchange data of size N with 2 neighbours
- 2D: size of chunk: $(N / \sqrt{p}) \times (N / \sqrt{p})$. Exchange data of size N / \sqrt{p} with 4 neighbours



Performance model of message passing communication

- We need analytical models to estimate impact of message passing
- A simplified linear cost model for point-to-point message passing: $T_{\text{comm}} = a + b * N$
- The communication time is linear proportional with length of message N
- Coefficient a = startup time
- Coefficient b = bandwidth
- Values of a, b are system constants, but usually $a \gg b \Rightarrow$ *if we must transmit N bytes, it is better to transmit them in one message, and NOT to initiate N independent messages of length 1 (“it is better to transmit data in bulk”)*

Which decomposition is better?

- We can have p processes. The matrix is $N \times N$.
- Is there a difference in performance: 1D vs 2D decomposition?
- Each process gets $N \times N / p$ elements
- 1D: size of chunk: $N \times N / p$. Exchange data of size N with 2 neighbours
- 2D: size of chunk: $(N / \sqrt{p}) \times (N / \sqrt{p})$. Exchange data of size N / \sqrt{p} with 4 neighbours
- $\text{TimeParallel} = T_{\text{comp}} + T_{\text{comm}}$
- $T_{\text{comp}} = N \times N / p$, the same in both cases
- $T_{\text{comm-1D}} = 2 \times (a + b \times N)$
- $T_{\text{comm-2D}} = 4 \times (a + b \times N / \sqrt{p})$
- If p is small \rightarrow better to use 1D decomposition
- If p is large \rightarrow better to use 2D decomposition

Heat2D example

- See https://staff.cs.upt.ro/~ioana/apd/mpi/mpi_heat2D.c