CPA NOTES 2

UNIT 3: MESSAGE PASSING. ADVANCED PARALLEL ALGORITHMS DESIGN

1.-MESSAGE PASSING MODEL

Tasks manage their **own private memory space**. **Data** are **exchanged** through **messages**. Communication normally requires **coordinated operations**. It is a **complex** and **costly** programming but provides a **total control of the parallelization**.

The parallel program comprises **several processes**. Usually correspond to O.S. processes. Typically, there is one process per processor and each one has an **index or identifier** (integer number).

Process creation can be:

- **Static**: at the startup of the program. Details are defined in the command line (mpiexec). They live during the whole execution. It is the most common approach.
- **Dynamic**: during the execution. It is done using the spawn() primitive.

Processes are organized into **groups**. It is key in collective operations, such as broadcast. It is defined by using indexes or operations on sets.

More general concept: **communicator = group + context**. The communication in a communicator cannot interfere with the communication taking place in another. It is useful for isolating the communication within a library. Communicators are defined from groups or other communicators. The **predefined communicators** are:

- **World**: includes all processes created by mpiexec.
- **Self**: includes only the calling process.

Basic send/receive operations are the most common in point-to-point communication:

- One process sends a message (send), another receives it (recv).
- Each send needs a corresponding recv.
- The message includes the content of one or more variables.

In the **synchronous mode**, the send operation does not conclude until the other process has done the matching recv:

- Along with the data transfer, processes get synchronized.
- It requires a protocol so that sender and receiver know that transmission can begin.

Buffered send/synchronous send:

- A buffer stores a temporary copy of the message.
- The buffered send finishes when the message has been copied from the program memory to a system buffer.
- The synchronous send does not finish until a matching recv has been done in the other process.

Blocking/nonblocking operations:

- When a call to a blocking send returns it is safe to modify the sent variable.
- When a call to a blocking recv returns it is guaranteed that the variable contains the message.
- Nonblocking calls simply initiate the operation.

In **nonblocking operations**, we need to know if the operation is complete:

- In recv in order to start reading the message.
- In send in order to start overwriting the variable.

Nonblocking send and recv provide a **request number** (req).

Primitives:

- **wait(req)**: the process gets blocked until the operation req is finished.
- **test(req)**: indicates if the operation has finished or not.
- **waitany** and **waitall** can be used when there are several pending operations.

This can be used to **overlap communications and computing**.

The recv operations **requires a process identifier id**:

- It does not finish until a message from id arrives.
- Messages from other processes are ignored.

For more flexibility, it is possible to use a **wildcard** value to receive from any process. Moreover, a tag is used to distinguish among messages. A **wildcard** is also possible to match any tag.

An incorrect usage of send and recv can lead to deadlocks. Another problem may arise when each process has to send data to its right neighbour. The potential solutions are:

- **Odd-even protocol**: odd processes perform one variant, even processes the other.
- **Nonblocking send or recv**.
- **Combined operations**: sendrecv.

Collective operations involve all processes in a communicator:

- **Synchronization (barrier)**: all processes wait for the rest to arrive.
- **Data transfer**: one or more processes send to one or more.
- **Reductions**: along with the communication an operation is performed on data.

These operations can be realized using point-to-point communication, but it is recommended to use the corresponding primitive:

- There are several algorithms for each case (linear, tree).
- The optimal solution often depends on the architecture.

The **types of collective communication** are:

- **One-to-all broadcast**: all processes receive what the root process send.
- **All-to-one reduction**: the dual of broadcast. Data are combined using an associative operator
- **Scatter**: the root sends an individualized message to the rest
- **Gather or concatenation**: the dual of scatter. Similar to reduction but without operation
- **All-to-all broadcast**: p simultaneous broadcasts, with different root processes. At the end, all processes will have received all the data.
- **All-to-all reduction**: the dual of all-to-all broadcast.

2.-ALGORITHM SCHEMES(II)

2.1.-DATA PARALLELISM

Data parallelism are used in algorithms with **many data treated in a similar way** (typically, matrix algorithms):

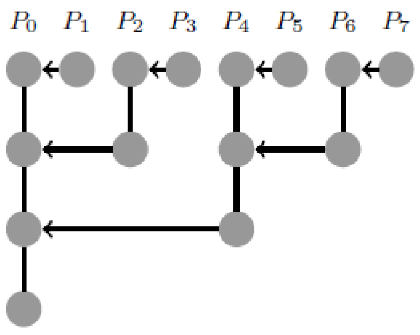
- In **shared memory**, loops are parallelized (each thread works on a part of the data).
- In **message passing**, an explicit data partitioning is performed.

In message passing it may be **inconvenient to parallelize**:

- The computational volume must be at least one order of magnitude higher than the communication cost.
- Often data are already distributed.

2.2.-TREE SCHEMES

Reduction using **Recursive Doubling**. There are $\log_2(p)$ communication stages. The number of participating processes is divided by two after each stage. The receiver accumulates the value on its local sum s.



2.3.-OTHER SCHEMES

Task parallelism is used in cases where **decomposition creates more tasks than processes**, or when **solving a task generates new tasks**:

- **Static assignment** of tasks is not feasible or leads to load imbalance.
- **Dynamic assignment**: tasks are being assigned to processes as they become idle.

It is usually implemented by means of an **asymmetric scheme**: manager-worker (or master-slave).

- The master keeps a count of finished and pending tasks.
- Workers receive tasks and notify the master when they have finished them.

Sometimes, a **symmetric solution** is feasible: **replicated workers**.

3.-PERFORMANCE EVALUATION (II)

3.1.-PARALLEL TIME

The **parallel execution time** is the time spent by a parallel algorithm with p processors, from the start of the first one until the last finishes.

It is composed of **arithmetic** and **communication** time:

$$t(n, p) = t_a(n, p) + t_c(n, p)$$

t_a corresponds to all computing times:

- All processors compute concurrently.
- It is equal or higher than the maximum arithmetic time.

t_c corresponds to times associated with data transfers:

- In distributed memory t_c = time of sending the messages.
- In shared memory t_c = synchronization time.

In practice:

- There is no clear separation between computation and communication stages.
- Often communication and computation can be overlapped.

Assuming message passing, P0 and P1 running on two different nodes with direct link. The Time needed to send a message of n bytes: $t_s + t_w n$.

- Communication set-up time, t_s.
- Bandwidth, w (maximum number of bytes per second).
- Sending time for 1 byte, $t_w = 1/w$.

3.2.-RELATIVE PARAMETERS

Relative parameters are used to compare different parallel algorithms:

- **Speedup**: S(n, p).
- **Efficiency**: E(n, p).

The **speedup** indicates the speed gain of a parallel algorithm with respect to its sequential version:

$$S(n,p) = \frac{t(n)}{t(n,p)}$$

The **reference time t(n)** could be:

- The best sequential algorithm.
- The parallel algorithm run on 1 processor.

The **efficiency** measures the degree of utilization of the parallel computer by the parallel algorithm:

$$E(n,p) = \frac{S(n,p)}{p}$$

Possible cases of speedup:

- **Speed-down**: the parallel algorithm is slower than the sequential algorithm.

$$S(n, p) < 1$$

- **Sublinear case**: the parallel algorithm is faster than the sequential one, but does not exploit all the capacity of procs.

$$1 < S(n, p) < p$$

- **Lineal case**: the parallel algorithm is as fast as possible, exploiting all the processors up to 100%.

$$S(n, p) = p$$

- **Superlinear case**: anomalous situation, when the parallel algorithm has less cost than the sequential one.

$$S(n, p) > p$$

Usually, the **efficiency decreases** as the **number of processors is increased**. The effect is normally less important for larger problem sizes.

Often, a part of the problem cannot be executed in parallel. Amdahl's Law estimates the maximum attainable speedup.

Given a sequential algorithm, split $t(n) = t_s + t_p$, where:

- t_s is the time of the intrinsically sequential part.
- t_p is the time of the perfectly parallelizable part (can be solved using p processors).

The **minimum attainable parallel time** will be:

$$t(n,p) = t_s + \frac{t_p}{p}$$

The **maximum speedup** is:

$$\lim_{p\to\infty} S(n,p) = \lim_{p\to\infty} \frac{t(n)}{t(n,p)} = \lim_{p\to\infty} \frac{t_s + t_p}{t_s + \frac{t_p}{p}} = 1 + \frac{t_p}{t_s}$$

4.-ALGORITHM DESIGN: TASK ASSIGNMENT

The decomposition phase has produced a set of tasks. An abstract and **platform-independent** parallel algorithm is obtained, potentially **inefficient**. The obtained decomposition must be **adapted** to a specific architecture.

Task assignment and **task scheduling** consist in determining the **processing units** and the **order** in which the tasks will be executed.

A **process** is a logical computing agent that performs tasks.

A **processor** is a hardware unit that physically performs computations.

A **parallel algorithm** is composed of processes that execute tasks:

- The assignment establishes the correspondence between tasks and processes in the design phase.
- The correspondence between processes and processors is done at the end and typically at execution time.

4.1.-THE PROBLEM OF ASSIGNMENT

The **main objective** is the assignment is to minimize execution time.

The **factors of the execution time** of a parallel algorithm and minimization strategies are:

- **Computing time**: to maximize concurrency by assigning independent tasks to distinct processes.
- **Communication time**: to assign tasks that communicate between them a lot to the same process.
- **Idle time**: to minimize the two main causes:
  - **Load imbalance**: computation and communication costs should be balanced among processes (previous diagram).
  - **Waiting time**: to minimize the waiting time of tasks that are not yet ready.

Static assignment or **deterministic scheduling**: the assignment decisions are taken before the execution time. Steps:

1. The number of tasks, their execution time and their communication costs are estimated.
2. Tasks are merged into larger ones to reduce communication costs.
3. Tasks are associated to processes.

The optimal static assignment problem is **NP-complete** in the general case.

The main **advantages** of static assignment:

- Does not add any overhead at run time.
- Design and implementation are generally simpler.

Dynamic assignment: the distribution of computational workload is done at execution time.

This kind of assignment is used when:

- Tasks are dynamically generated.
- The task size is not known a priori.

In general, **dynamic techniques are more complex**. The main drawback is the induced **overhead** due to:

- Information transfer among processes regarding load and work.
- Decision making to move load among processes (done at run time).

The main **advantage** is that there is no need to know the behaviour a priori, they are flexible and appropriate for heterogeneous architectures.

4.2.-STRATEGIES FOR MERGING AND REPLICATION

Merging is used to reduce the number of tasks aiming at:

- **Limiting the task** creation and termination costs.
- **Minimizing the delays** due to the interaction among tasks.

Merging strategies:

- **Volume minimization of the transferred data**. Distribution of tasks based on data blocks (matrix algorithms), merging of non-concurrent tasks (static task graphs), temporal storage of intermediate results.
- **Reduction of the interaction frequency**. To minimize the number of transfers and to increase the volume of data to be exchanged.
    - In **distributed memory**, it means to reduce latency (number of messages) and to increase the volume of data per message.
    - In **shared memory**, it means to reduce the number of cache misses.

Replication implies that part of the computations or data from a problem are not split but executed or managed by all or several processes:

- **Data replication**: consists in copying commonly accessed data in the different processes aiming at reducing communication.
  - In **shared memory** it is implicit since it only affects cache memory.
  - In **distributed memory** it may lead to a considerable performance improvement and design simplification.
- **Computation and communication replication**: consists in repeating a computation in each one of the processes that need the result. It is convenient in the case that the computation cost is smaller than the communication cost.

5.-ASSIGNMENT SCHEMES

5.1.-SCHEMES FOR STATIC ASSIGNMENT

Static schemes for domain decomposition focus on large-scale data structures. The assignment of tasks to processes consists in distributing the data among the processes. There are two types:

- Block-oriented matrix distributions.
- Static splitting of graphs.

Schemes on static dependency graphs are normally obtained by a functional decomposition of the data flow or recursive decomposition.

In matrix computations, typically the computation of an entry depends on the neighbouring entries (**spatial locality**). The assignment considers contiguous portions (**blocks**) in the data domain (**matrix**).

The most typical **block distributions** are:

- Uni-dimensional block distribution of a vector.
- Uni-dimensional distribution by blocks of **rows** of a matrix.
- Uni-dimensional distribution by blocks of **columns** of a matrix.
- **Bi-dimensional** block distribution of a matrix.

They also exist the **cyclic variants**.

In the **uni-dimensional block distribution**, the **global index i** is assigned to process **$[i/m_b]$** where **$m_b = [n/p]$** is the **block size**. The local index is i mod m_b (reminder of integer division).

In the **bi-dimensional block distribution**, each process has a **block of size $m_b \times n_b = [m/p_m] \times [n/p_n]$**, where p_m and p_n are the first and second dimension of the process grid, respectively.

Task merging improves locality, since it reduces the volume of communication. The goal is to maximize computation and minimize communication.

Volume-surface effect:

- The computational load increases proportionally to the number of elements assigned to a task.
- The communication cost increases proportionally to the perimeter of the task.

This effect grows as the number of dimensions of the matrix is increased.

In **cyclic distributions**, the objective is to balance the load during all the execution time:

- Larger communication cost since locality is reduced
- Usually combined with block schemes
- An equilibrium between load balancing and communication costs should be kept: **most appropriate block size**.

In case of **functional decomposition**, we assume a static dependency graph and task costs known a priori. The problem of finding optimal assignments is NP-complete. However, there are cases in which optimal algorithms or heuristic approaches are known.

5.2.-DYNAMIC WORKLOAD BALANCING

When the **static assignment** is **not feasible**:

- The tasks obtained in the decomposition are data structures that represent sub-problems.
- Sub-problems are kept in a collection and dispatched to the different processes.
- The solution of a sub-problem may lead to the dynamic creation of more sub-problems.
- These schemes require a mechanism for **detecting termination**.

There are different **types of dynamic workload balancing**:

- **Centralized scheme**, a master process manages the collection of sub-problems and dispatches them.
- **Distributed scheme, without a master**; load balancing is not trivial.

SEMINAR 3: PROGRAMMING WITH MPI

1.-BASIC CONCEPTS

1.1.-MESSAGE-PASSING MODEL

Exchange of information by **explicitly sending and receiving messages**.

Its **advantages** are:

- Universality.
- Easy understanding.
- High expressivity.
- Higher efficiency.

Its **drawbacks** are:

- Complex programming.
- Total control of communications.

1.2.-THE MPI STANDARD

MPI is a specification proposed by a committee of researchers, users and companies. **Main features**:

- It is portable to any parallel platform.
- It is simple.
- It is powerful.

1.3.-MPI PROGRAMMING MODEL

MPI programming is based on **library functions**. To use them, an initialization is required. **MPI_Init** and **MPI_Finalize** are compulsory. Once initialized, different operations can be performed.

Operations can be classified in:

- **Point-to-point communication**: exchange of information between two processes.
- **Collective communications**: exchange of information among groups of processes.
- **Data management**: derived datatypes (e.g. data stored non-contiguously in memory).
- **High-level communications**: groups, communicators, attributes, topologies.
- **Advanced operations (MPI-2, MPI-3)**: input-output, process creation, one-sided communication.
- **Utilities**: interaction with the environment.

Most communication operations work on **communicators**.

A **communicator** is an abstraction that comprises the following concepts:

- **Group**: group of processes.
- **Context**: to avoid interferences among different messages.

A communicator groups together p processes:

int MPI_Comm_size(MPI_Comm comm, int *size)

Each process has an identifier (rank), a number between 0 and p – 1:

int MPI_Comm_rank(MPI_Comm comm, int *rank)

The MPI execution model follows a scheme of simultaneous process creation when launching the application. Applications are normally executed by a launcher command:

mpiexec -n p program [arguments]

When an application is executed:

- p copies of the same executable are spawn.
- A communicator is created (MPI_COMM_WORLD) that groups all the processes.

2.-POINT-TO-POINT COMMUNICATION

2.1.-SEMANTICS

Messages must be explicitly issued by the sender and explicitly received by the receiver

Standard send:

MPI_Send(buf, count, datatype, dest, tag, comm)

Standard receive:

MPI_Recv(buf, count, datatype, src, tag, comm, stat)

The **contents of the message** are defined by the first three arguments:

- A memory buffer where data is stored.
- The message length (number of elements from the buffer).
- Datatype of the elements.

To perform the communication, it is necessary to indicate the destination **(dest) and the origin (src)**:

- The communication is allowed only within the same communicator, comm.
- The origin and destination are specified via process identifiers.
- In the reception it is allowed to use src=MPI_ANY_SOURCE.

An **integer number** can be used (tag) to **distinguish among messages** of different type. In the reception it is allowed to use tag=MPI_ANY_TAG.

In the reception, **the status (stat)** contains information:

- Source process (stat.MPI_SOURCE), tag (stat.MPI_TAG).
- Message length.

Note: pass **MPI_STATUS_IGNORE** if **not required**.

There are several **send modes**:

- **Synchronous send mode**.
- **Buffered send mode**.
- **Standard send mode**.

The most commonly used one is the standard mode. The rest of the modes could be useful to obtain better performance or increased robustness. For each mode, there are blocking and nonblocking primitives.

2.2.-BLOCKING PRIMITIVES

Synchronous send mode with **blocking primitives**:

MPI_Ssend(buf, count, datatype, dest, tag, comm)

It implements the send model with "**rendezvous**": the sender gets blocked until receiver posts the receive operation. It is **inefficient** since the sender remains blocked doing no useful work.

Buffer-based send mode with **blocking primitives**:

MPI_Bsend(buf, count, datatype, dest, tag, comm)

The message is copied to an intermediate memory and the sending process continues its execution. Its main **drawbacks** are the **additional copy** and the **risk of failure**. A buffer may be provided (MPI_Buffer_attach).

Standard send mode with **blocking primitives**:

MPI_Send(buf, count, datatype, dest, tag, comm)

Completion is guaranteed in any kind of systems, since it avoids storage problems. **Short messages** are usually sent using **MPI_Bsend**. **Long messages** are usually sent with **MPI_Ssend**

Standard reception with **blocking primitives**:

MPI_Recv(buf, count, datatype, src, tag, comm, stat)

It implements the reception model with "**rendezvous**": the receiver gets blocked until the message arrives. It is **inefficient** since the receiver process gets blocked and idle.

2.3.-OTHER PRIMITIVES

Nonblocking send:

MPI_Isend(buf, count, datatype, dest, tag, comm, req)

The send operation is started, but **the sender is not blocked**:

- It has an additional argument (req).
- Before reusing the buffer, one must make sure that the send operation has been completed.

It **overlaps communication/computation with no extra copy**. Its main **drawback** is that it is a **more complex programming**.

Nonblocking reception:

MPI_Irecv(buf, count, type, src, tag, comm, req)

Reception is initiated, but the **receiver does not get blocked**:

- The stat argument is replaced by req.
- It is necessary to check the effective arrival of the message.

Its main **advantage** is that it **overlaps communication and computation**. Its main **drawback** is that is a **more complex programming**.

Combined operations:

MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

It performs a send and receive operation in the same call (not necessarily involving the same process).

MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag, comm, status)

It performs a send and receive operation at the same time on the **same variable**.

3.-COLLECTIVE COMMUNICATION

They involve all processes in a group (communicator) – all of them must execute the operation. **Available operations**:

- **Synchronization** (Barrier)
- **Broadcast** (Bcast).
- **Distribution** (Scatter).
- **Concatenation** (Gather).
- **Conc.-broadcast** (Allgather).
- **All-to-all comm.** (Alltoall).
- **Reduction** (Reduce).
- **Prefix reduction** (Scan).

These operations usually take as an argument a process (root) who plays a special role. **“All” prefix**: Every process receives the result. **“v” suffix**: The size of data received or sent by each process may be different.

3.1.-SYNCHRONIZATION

Pure synchronization operation:

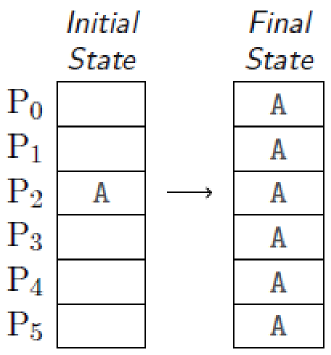
MPI_Barrier(comm)

All the processes of the communicator comm wait until all of them have reached this call.

3.2.-BROADCAST

The root process broadcasts the message indicated by the first three arguments to the rest of processes:

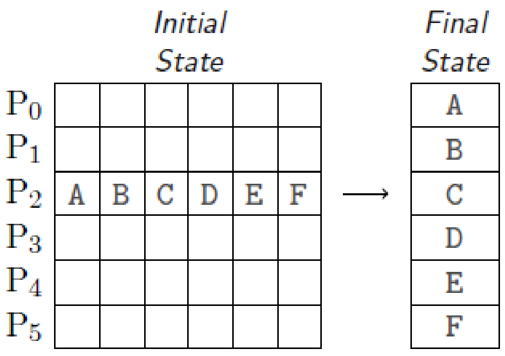
MPI_Bcast(buffer, count, datatype, root, comm)



3.3.-SCATTER

The root process distributes a series of consecutive fragments of the buffer to the rest of processes (including itself):

MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

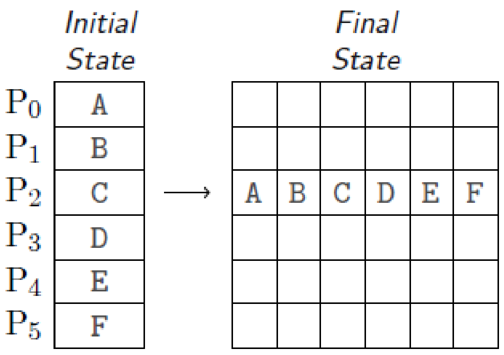


It exists an asymmetric version: MPI_Scatterv.

3.4.-GATHER

It is the reverse operation of MPI_Scatter: Each process sends amessage to root, who stores them in an ordered way according to the index of the process owning each fragment:

MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

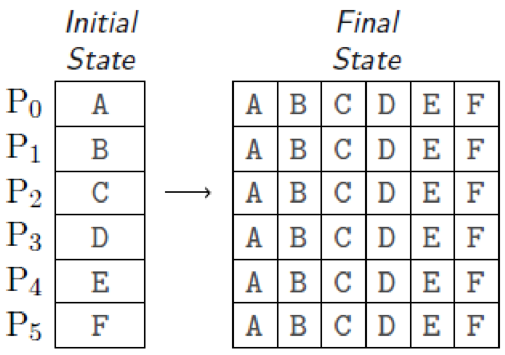


It exists an asymmetric version: MPI_Gatherv.

3.5.-ALLGATHER

Similar to the MPI_Gather operation, but in this case all processes get the result:

MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

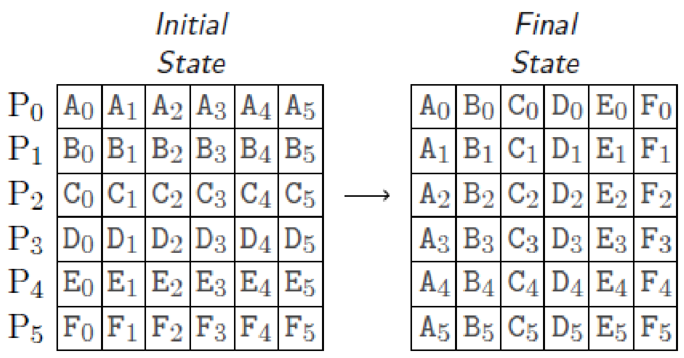


It exists an asymmetric version: MPI_Allgatherv.

3.6.-ALL-TO-ALL COMUNICATION

An extension of MPI_Allgather, where each process sends different data and receives (ordered) data from the rest:

MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

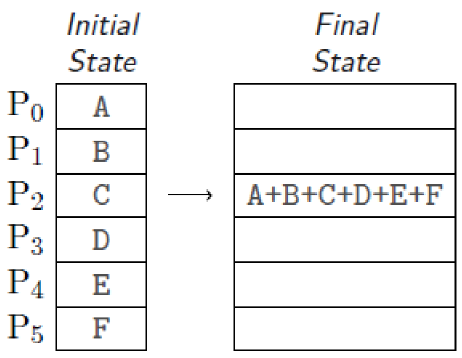


It exists an asymmetric version: MPI_Alltoallv.

3.7.-REDUCTION

Similar to MPI_Gather, but instead of concatenation, an arithmetic or logic operation is applied to the data (sum, max, and, ..., or user-defined). The final result is stored in the root process:

MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)



3.8.-MULTI-REDUCTION

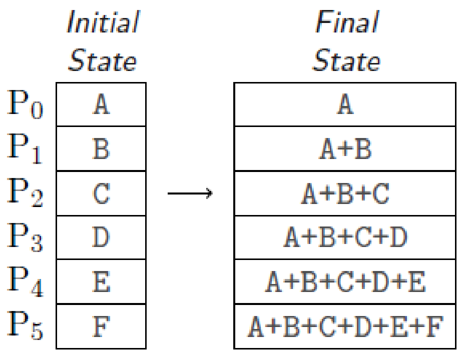
Extension of MPI_Reduce in which all processes get the result:

MPI_Allreduce(sendbuf, recvbuf, count, type, op, comm)

3.9.-SCAN

Extension of the reduction operations in which each process receives the result of processing all the elements from process 0 to itself:

MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm)



4.-OTHER FUNCTIONALITIES

The **basic datatypes in the C language** are:

`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`

Along with the previous ones, there are the special types **MPI_BYTE** and **MPI_PACKED**.

It is possible to send/receive **multiple data in one message**:

- The sender indicates the number of data to be sent using argument **count**.
- The message is formed by the first **count** elements **contiguous in memory**.
- In the receiver, argument **count** indicates the buffer size.

The actual size of the received message can be obtained with:

MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

4.1.-DERIVED DATATYPES

MPI allows defining **new types** from other types. The procedure goes through the following steps:

1. The programmer defines the new type, indicating:
    a. The datatypes of its different constituents.
    b. The number for elements of each type.
    c. The relative displacements for each element.
2. It is registered as a new MPI datatype (commit).
3. From then on, it can be used in any communication operation as it was a basic datatype.
4. If no longer needed, the type must be destroyed (free).

Its **main advantages** are:

- It simplifies programming when it is used several times.
- There is no intermediate copy, since data is compressed only at the time of sending.

4.1.1.-REGULAR DERIVED DATATYPES

It creates an homogeneous datatype from the evenly distributed elements of an array:

1. How many blocks are included (count).
2. Which is the length of the blocks (length).
3. Which is the separation from an element of a block and the same element in the next block (stride).
4. Of which type are the individual blocks (type).

MPI_Type_vector(count, length, stride, type, newtype)

There are some related constructors:

- **MPI_Type_contiguous**: contiguous elements.
- **MPI_Type_indexed**: variable length and displacement.

4.1.2.-IRREGULAR DERIVED DATATYPES

It creates an heterogeneous datatype:

MPI_Type_struct(count, lens, displs, types, newtype)