Chapter 2
Introduction to MPI: The Message Passing Interface

2.1 MPI for Parallel Programming: Communicating with Messages

Programming parallel algorithms is far more delicate than programming sequential algorithms. And so is debugging parallel programs too! Indeed, there exists several *abstract models* of "parallel machines" (parallel computations, distributed computations) with different kinds of *parallel programming paradigms*: For example, let us mention:

- *Vector super-computers* that rely on the programming model called *Single Instruction Multiple Data* (*SIMD*) with their optimized code based on pipelined operations,
- *Multi-core machines* with shared memory and their programming model using multi-threading, with all threads potentially accessing the shared memory. Programs can be easily crashing and it is difficult to debug sometimes due to potential conflicts when accessing concurrently a shared resource,
- *Clusters of computer machines* interconnected by a high-speed network that have a distributed memory.

It is precisely this last category of "parallel machines", the clusters of machines, that we are focusing on in this textbook: namely, parallel programming paradigm with distributed memory. Each computer can execute programs using its own local memory. Executed programs can be the same on all computers or can be different. Cooperation takes place by sending and receiving messages among these interconnected computers to accomplish their overall task.

Speaking on the size of these clusters, we can distinguish between:

- small-size to mid-size clusters of computers (say, dozens to hundreds, sometimes thousands, of computer nodes) that communicate with each other by sending and receiving messages, and
- large-size clusters (thousands to hundreds of thousands, sometimes millions computers) that execute rather simpler codes targeting *Big Data* processing.

F. Nielsen, *Introduction to HPC with MPI for Data Science*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-21903-5_2

Usually, these large-size clusters are programmed using the MapReduce/Hadoop programming model.

The Message Passing Interface (or MPI for short) standard is a programming interface: namely, an *Application Programming Interface* (*API*) that defines properly the syntax and full semantic of a software library that provides standardized basic routines to build complex programs thereof. Thus the MPI interface allows one to code parallel programs exchanging data by sending and receiving messages encapsulating those data. Using an API has the advantage of leaving the programmer free of the many details concerning the implementation of the fine details of implementing from scratch network procedures, and allows the ecosystem (academy, industry, programmers) to benefit of interoperability and portability of source codes. It is important to emphasize the fact that the MPI API *does not depend* on the underlying programming language it uses. Thus we can use MPI commands with the most common (sequential) programming languages like C, C++, Java, Fortran, Python and so on. That is, several *language bindings* of the MPI API are available.

MPI historically got initiated from a workshop organized in 1991 on distributed memory environments. Nowadays, we use the third version of the standard, *MPI-3*, which standardization has been completed and published openly in 2008. We shall choose *OpenMPI* (http://www.open-mpi.org/) to illustrate the programming examples in this book.

Let us emphasize that the MPI interface is the dominant programming interface for parallel algorithms with distributed memory in the HPC community. The strong argument in favor of MPI is the standardization of *many* (i) global routines of communication (like broadcasting, the routine that consists in sending a message to all other machines) and (ii) many primitives to perform global calculations (like computing a cumulative sum of data distributed among all machines using an aggregation mechanism). In practice, the complexity of these global communications and calculation operations depend on the underlying topology of the interconnection network of the machines of the cluster.

2.2 Parallel Programming Models, Threads and Processes

Modern operating systems are *multi-tasks*: from the user viewpoint, several non-blocking applications seem to be executed (run) "simultaneously". This is merely an illusion since on a single Central Processing Unit (CPU) there can be only one program instruction at a time being executed. In other words, on the CPU, there is a current process being executed while the others are blocked (suspended or waiting to be waked up) and wait their turn to be executed on the CPU. It is the rôle of the *task scheduler* to allocate dynamically processes to CPU.

Modern CPUs have several *cores* that are independent *Processing Units* (PUs) that can execute truly in parallel on each core a thread. Multi-core architectures yield the multi-threading programming paradigm that allows for *concurrency*. For example,

your favorite Internet WEB browser allows you to visualize simultaneously several pages in their own tabs: each HTML page is rendered using an independent thread that retrieves from the network the page contents in HTML[1]/XML and displays it. The resources allocated to a process are *shared* between the different threads, and at least one thread should have a `main` calling function.

We can characterize the threads as follows:

- Threads of a same process share the same memory area, and can therefore access both the data area but also the code area in memory. It is therefore easy to access data between the threads of a same process, but it can also raise some difficulties in case of simultaneous access to the memory: In the worst case, it yields a system crash! A theoretical abstraction of this model is the so-called *Parallel Random-Access Machine* (or *PRAM*). On the PRAM model, we can classify the various conflicts that can happen when reading or writing simultaneously on the local memory. We have the *Exclusive Read Exclusive Write* sub-model (EREW), the *Concurrent Read Exclusive Write* (CREW) and the *Concurrent Read Concurrent Write* models.
- This multi-threading programming model is very well suited to multi-core processors, and allows applications to be ran faster (for example, for encoding a MPEG4 video or a MP3 music file) or using non-blocking applications (like a web multi-tab browser with a mail application).
- Processes are different from threads because they have their own *non-overlapping* memory area. Therefore, communications between processes have to be done careful, in particular using the MPI standard.

We can also distinguish the parallel programming paradigm *Single Program Multiple Data* (SPMD) from the paradigm called Multiple Program Multiple Data (MPMD). Finally, let us notice that we can run several processes either on a same processor (in parallel when the processor is multi-core) or on a set of processors interconnected by a network. We can also program processes to use several multi-core processors (in that case, using both the MPI and OpenMP standards).

2.3 Global Communications Between Processes

By executing a MPI program on a cluster of machines, we launch a set of processes, and we have for each process traditional local computations (like ordinary sequential programs) but also:

- some data transfers: for example, some data broadcasted to all other processes using a message,
- some synchronization barriers where all processes are required to wait for each other before proceeding,

[1] *Hypertext Markup Language*.

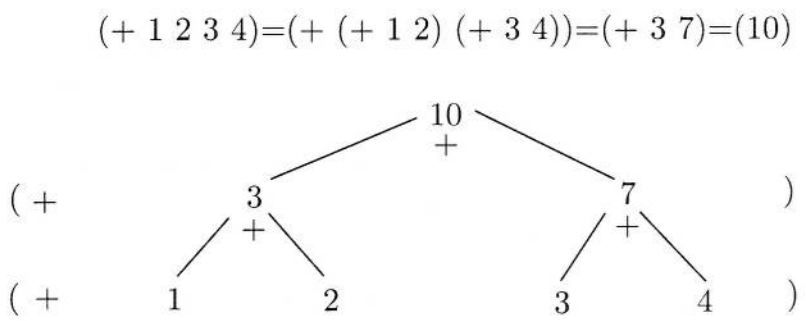

Fig. 2.1 Example of a global reduce computation: calculating the global cumulative sum of the local values of a process variable. We illustrate here the reduction tree performed when invoking the reduce primitive

- global computations: for example, a reduce operation that calculates, say, the sum or the minimum of a distributed variable x belonging to all the processes (with a local value stored on the local memory of each process). Figure 2.1 illustrates a reduce cumulative sum operation. The global computation depends on the underlying topology of the interconnected cluster machines.

Global communication primitives are carried out on all processes belonging to the same *group of communication*. By default, once MPI got initialized, all processes belong to the same group of communication called `MPI_COMM_WORLD`.

2.3.1 Four Basic MPI Primitives: Broadcast, Gather, Reduce, and Total Exchange

The MPI broadcasting primitive, `MPI_Bcast`, sends a message from a *root process* (the calling process of the communication group) to all other processes (belonging to the communication group). Conversely, the reduce operation aggregates all corresponding values of a variable into a single value that is returned to the calling process. When a different personalized message is send to each other process, we get a scatter operation called in MPI by `MPI_Scatter`.

Aggregating primitives can either be for communication of for computing globally: gather is the converse of the scatter operation where a calling process receives from all other processes a personalized message. In MPI, `MPI_Reduce` allows one to perform a *global calculation* by aggregating (reducing) the values of a variable using a commutative binary operator.[2] Typical such examples are the cumulative sum (`MPI_SUM`) or the cumulative product (`MPI_PROD`), etc. A list of such binary operators used in the reduce primitives is given in Table 2.1. Those four basic MPI primitives are illustrated in Fig. 2.2. Last but not least, we can also call a global

[2] An example of binary operator that is not commutative is the division since $p/q \neq q/p$.

Table 2.1 Global calculation: predefined (commutative) binary operators for MPI reduce operations

MPI name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bit-wise and
MPI_LOR	Logical or
MPI_BOR	Bit-wise or
MPI_LXOR	Logical xor
MPI_BXOR	Bit-wise xor
MPI_MAXLOC	Max value and location
MPI_MINLOC	Min value and location

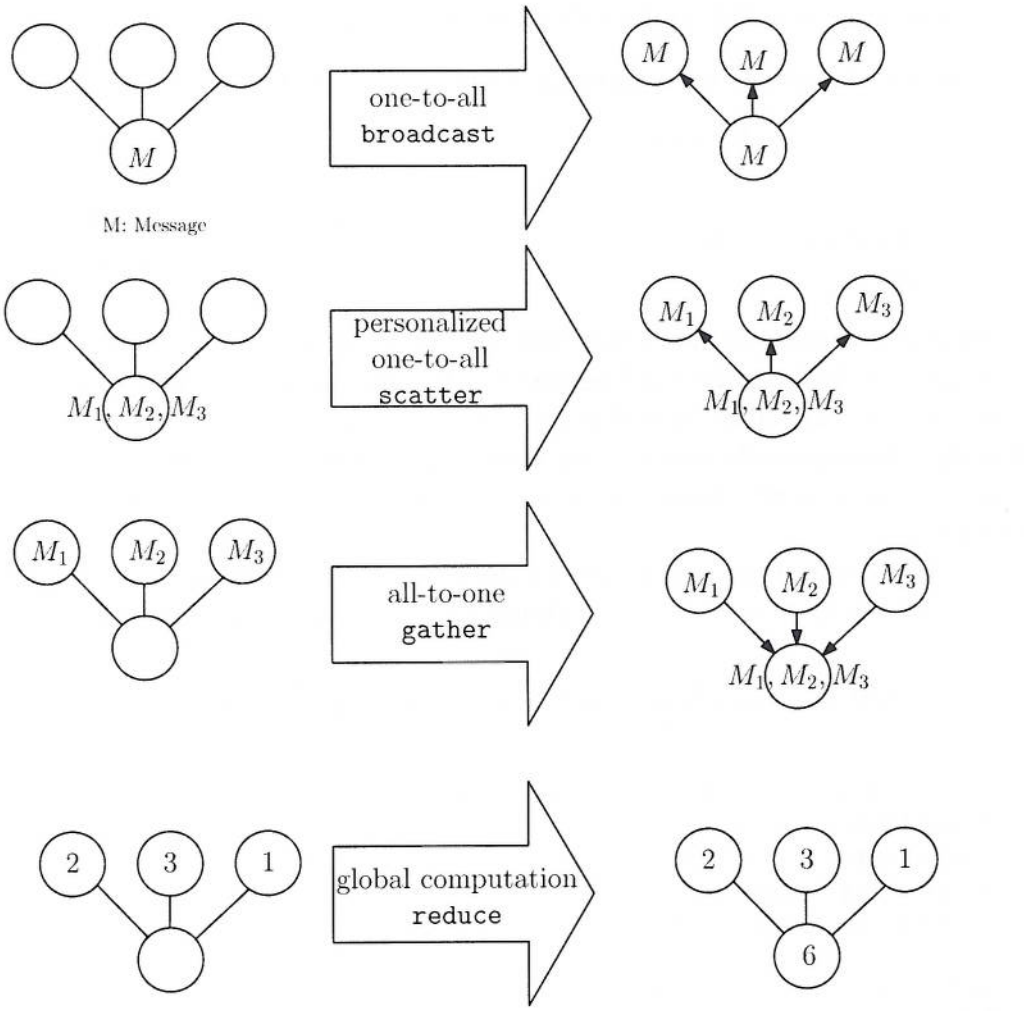

Fig. 2.2 Four basic collective communication primitives: broadcast (one to all), scatter (personalized broadcast or personalized one-to-all communication), gather (the inverse of a scatter primitive, or personalized all-to-one communication) and reduce (a global collective computation)

all-to-all communication primitive (`MPI_Alltoall`, also called *total exchange*) where each process sends a personalized message to all other processes.

2.3.2 Blocking Versus Non-blocking and Synchronous Versus Asynchronous Communications

MPI has several send modes depending on whether data are buffered or not and whether synchronization is required or not. First, let us start by denoting by `send` and `receive` the two basic communication primitives. We describe the syntax and semantic of these two primitives as follows:

- `send(&data, n, Pdest)`: Send an array of n data starting at memory address `&data` to process `Pdest`
- `receive(&data, n, Psrc)`: Receive n data from process `Psrc` and store them in an array which starts a local memory address &data

Now, let us examine what happens in this example below:

Process P0	Process P1
`...`	`...`
`a=442;`	`receive(&a, 1, P0);`
`send(&a, 1, P1);`	`cout << a << endl;`
`a=0;`	

Blocking communications (not buffered) yield a *waiting time* situation: that is, a *idling time*. Indeed, the sending process and the receiving process need to wait mutually for each other: it is the communication mode commonly termed hand-shaking. This mode allows one to perform synchronous communications. Figure 2.3 illustrates these synchronous communications by hand-shaking and indicates the idling periods.

The C program below gives an elementary example of blocking communication in MPI (using the *C binding* of the OpenMPI vendor implementation of MPI):

WWW source code: `MPIBlockingCommunication.cpp`

```
// filename: MPIBlockingCommunication.cpp
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs;
```

```
    int tag,source,destination,count;
    int buffer;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    tag=2312; /* any integer to tag messages */
    source=0;
    destination=1;
    count=1;
    if(myid == source){
      buffer=2015;
      MPI_Send(&buffer,count,MPI_INT,destination,tag
          ,MPI_COMM_WORLD);
      printf("processor %d received %d \n", myid,
          buffer)
    }
    if(myid == destination){
        MPI_Recv(&buffer,count,MPI_INT,source,tag,
            MPI_COMM_WORLD,&status);
        printf("processor %d received %d \n",myid,
            buffer);
    }
    MPI_Finalize();
}
```

Clearly, for blocking communications, one seeks to minimize the overall idling time. We shall see later on how to perform this optimization using a load-balancing technique to balance fairly the local computations among the processes.

We report the syntax and describe the arguments of the `send`[3] primitive in MPI:

- Syntax using the C binding:

```
#include <mpi.h>
int MPI_Send(void *buf, int count, MPI_Datatype
    datatype, int dest, int tag, MPI_Comm comm)
```

- Syntax in C++ (Deprecated. That is, it is not regularly updated since MPI-2 and we do not recommend using it):

```
#include <mpi.h>
void Comm::Send(const void* buf, int count, const
    Datatype& datatype, int dest, int tag) const
```

The `tag` argument in `send` assigns an integer to a message (its label) so that processes can specify which type of message to wait for. Tags are useful in practice to filter communication operations, and ensures for example that messages sent/received are pairwise matching using blocking communications.

The C data types in MPI are summarized in Table 2.2.

[3]See the manual online: https://www.open-mpi.org/doc/v1.4/man3/MPI_Send.3.php.

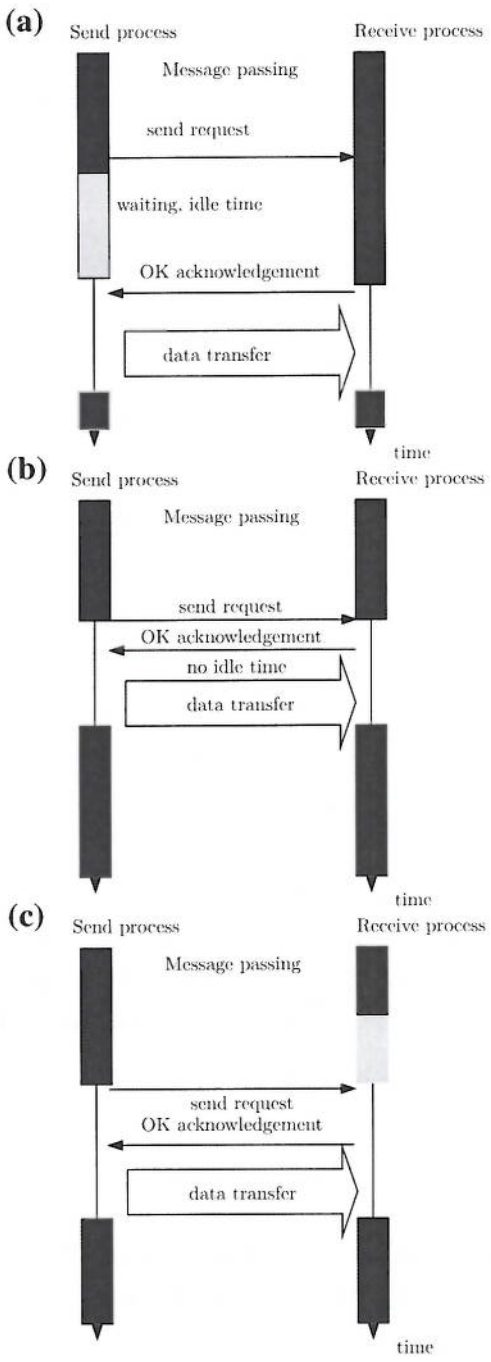

Fig. 2.3 Blocking communications by hand-shaking: **a** sending process waits for the "OK" of the receiver and thus provoke a waiting situation, **b** one seeks to minimize this idling time, and **c** case where it is the receiving process that needs to wait

2.3.3 Deadlocks from Blocking Communications

Using blocking communications allows one to properly match "send queries" with "receive queries", but can unfortunately also yields deadlocks.[4]

[4]In that case, either a time-out signal can be emitted externally to kill all the processes, or we need to manually kill the processes using their process identity number using Shell command line instructions.

Table 2.2 Basic data types in MPI when using the C language binding

MPI type	Corresponding type in the C language
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
MPI_BYTE	
MPI_PACKED	

Let us consider this toy example to understand whether there is a deadlock situation occurring or not:

Process P0	Process P1
`send(&a, 1, P1);`	`send(&a, 1, P0);`
`receive(&b, 1, P1);`	`receive(&b, 1, P0);`

Process $P0$ send a message and then waits the green light "OK for sending" of receiver process $P1$, but the send query of $P1$ also waits for the green light "OK for sending" of process $P0$. That is typically a *deadlock* situation. In this toy example, we have highlighted the deadlock problem when using blocking communication primitives. However in practice, it is not so easy to track them as process programs can take various execution paths.

In practice, in MPI, each send/receive operation concerns a group of communication and has a tag attribute (an integer). From an algorithmic viewpoint, blocking communications are a highly desirable feature to ensure consistency (or the semantic) of programs (for example, to avoid that messages arrive in wrong orders) but they can yield difficulties to detect deadlocks.

In order to remove (or at least minimize!) these deadlock situations, we can preallocate to each process a dedicated memory space for buffering data: the *data buffer* (bearing the acronym DB). We then send data in two steps:

- First, the send process sends a message on the data buffer, and
- Second, the receive process copies the data buffer on its local memory area indicated by the address `&data`.

This buffered communication can either be implemented by hardware mechanisms or by appropriate software. However, there still remains potential deadlocks when the data buffers become full (raising a "buffer overflow" exception). Even if

we correctly manage the `send` primitives, there can still be remaining deadlocks, even with buffered communications, because of the blocking receive primitive. This scenario is illustrated as follows:

Process P0	Process P1
`receive(&a, 1, P1);`	`receive(&a, 1, P0);`
`send(&b, 1, P1);`	`send(&b, 1, P0);`

Each process waits for a message before being able to send its message! Again, this is a deadlock state! In summary, blocking communications are very useful when we consider global communication like broadcasting in order to ensure the correct arrival order of messages, but one has to take care of potential deadlocks when implementing these communication algorithms.

A solution to avoid deadlocks is to consider both the `send` and `receive` primitives being non-blocking. These *non-blocking communication routines* (not buffered) are denoted by `Isend` and `Ireceive` in MPI: There are *asynchronous communications*. In that case, the send process posts a message "Send authorization request" (a pending message) and continues the execution of its program. When the receiver process posts a "OK for sending" approval, data transfers are initiated. All these mechanics are internally managed using signals of the operating system. When the data transfer is completed, a check status let indicate whether processes can proceed to read/write data safely. The C program below illustrates such a non-blocking communication using the C binding of OpenMPI. Let us notice that the primitive `MPI_Wait(&request,&status);` waits until the transfer is completed (or interrupted) and indicates whether that transfer has been successful or not, using a state variable called `status`.

WWW source code: `MPINonBlockingCommunication.cpp`

```
// filename: MPINonBlockingCommunication.cpp
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
     int myid, numprocs;
     int tag,source,destination,count;
     int buffer;
     MPI_Status status;
     MPI_Request request;

     MPI_Init(&argc,&argv);
     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
     tag=2312;
```

```
    source=0;
    destination=1;
    count=1;
    request=MPI_REQUEST_NULL;
    if(myid == source){
      buffer=2015;
      MPI_Isend(&buffer,count,MPI_INT,destination,
          tag,MPI_COMM_WORLD,&request);
    }
    if(myid == destination){
        MPI_Irecv(&buffer,count,MPI_INT,source,tag,
            MPI_COMM_WORLD,&request);
    }
    MPI_Wait(&request,&status);
    if(myid == source){
      printf("processor %d  sent %d\n",myid,buffer);
    }
    if(myid == destination){
      printf("processor %d  received %d\n",myid,
          buffer);
    }
    MPI_Finalize();
}
```

We summarize the calling syntax in the C binding of the non-blocking primitives `Isend` and `Irecv`:

```
int MPI_Isend( void *buf, int count, MPI_Datatype
    datatype, int dest, int tag, MPI_Comm comm,
    MPI_Request *req )

int MPI_Irecv( void *buf, int count, MPI_Datatype
    datatype, int src, int tag, MPI_Comm comm,
    MPI_Request *req )
```

The structure `MPI_Request` is often used in programs: it returns `*flag=1` when the operation `*req` has completed, and 0 otherwise.

```
int MPI_Test( MPI_Request *req, int *flag,
    MPI_Status *status )
```

The primitive MPI_Wait waits until the operation indicated by `*req` has been completed

```
int MPI_Wait( MPI_Request *req, MPI_Status *status )
```

We summarize the various communication protocols (blocking/non-blocking send with blocking/non-blocking receive) in Table 2.3.

Table 2.3 Comparisons of the various send/receive operation protocols

	Blocking operation	Non-blocking operation
Bufferized	`send` completes after data have been copied to the data buffer	`send` completes after having initialized DMA (*Direct Memory Access*) transfer to the data buffer. The operation is not necessarily completed after it returns
Not-bufferized	Blocking `send` until it meets a corresponding `receive`	To define
Meaning	Semantic of `send` and `receive` by matching operations	Semantic must be explicitly specified by the programmer that needs to check the operation status

The program listings so far highlighted eight common procedures of MPI (among a rich set of MPI instructions):

`MPI_Init`	Initialize the MPI library
`MPI_Finalize`	Terminate MPI
`MPI_Comm_size`	Return the number of processes
`MPI_Comm_rank`	Rank of the calling process
`MPI_Send`	send a message (blocking)
`MPI_Recv`	receive message (blocking)
`MPI_Isend`	send a message (non-blocking)
`MPI_Irecv`	receive message (non-blocking)

All these procedures return `MPI_SUCCESS` when they are completed with success, or otherwise an error code depending on the problems. Data types and constants are prefixed with `MPI_` (we invite the reader to explore the header file `mpi.h` for more information).

2.3.4 *Concurrency: Local Computations Can Overlap with Communications*

It is usual to assume that the processors (or Processing Elements, PEs) can perform several tasks at the same time: for example, a typical scenario is to use non-blocking communications (`MPI_IRecv` and `MPI_ISend`) at the same time they perform some local computations. Thus we require that those three operations are not interferring with each other. In one stage, we can therefore not send the result of a calculation and we can not send what has been concurrently received (meaning

forwarding). In parallel algorithmic, we denote by the double vertical bar || these concurrent operations:

```
IRecv||ISend||Local_Computation
```

2.3.5 Unidirectional Versus Bidirectional Communications

We distinguish between *one-way communication* and *two-way communication* as follows: in one-way communication, we authorize communications over communication channels in one direction only: that is, either we send a message or we receive a message (`MPI_Send`/`MPI_Recv`) but not both at the same time. In a two-way communication setting, we can communicate using both directions: in MPI, this can be done by calling the procedure MPI_Sendrecv.[5]

2.3.6 Global Computations in MPI: Reduce and Parallel Prefix (Scan)

In MPI, one can perform global computations like the cumulative sum $V = \sum_{i=0}^{P-1} v_i$ where v_i is a local variable stored in the memory of process P_i (or the cumulative product $V = \prod_{i=0}^{P-1} v_i$). The result of this global computation V is then available in the local memory of the process that has called this reduce primitive: the calling process, also called the root process. We describe below the usage of the `reduce`[6] primitive using the C binding of OpenMPI:

```
#include <mpi.h>

int MPI_Reduce ( // Reduce routine
 void* sendBuffer, // Address of local val
 void* recvBuffer, // Place to receive into
 int count, // No. of elements
 MPI_Datatype datatype, // Type of each element
 MPI_OP op, // MPI operator
 int root, // Process to get result
 MPI_Comm comm // MPI communicator
 );
```

Reduction operations are predefined and can be selected using one of the keywords among this list (see also Table 2.1).

---

[5] https://www.open-mpi.org/doc/v1.8/man3/MPI_Sendrecv.3.php.

[6] See manual online at https://www.open-mpi.org/doc/v1.5/man3/MPI_Reduce.3.php.

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum $\sum$
MPI_PROD	product $\prod$
MPI_LAND	logic AND
MPI_BAND	bitwise AND
MPI_LOR	logic OR
MPI_BOR	bitwise OR
MPI_LXOR	logic XOR
MPI_BXOR	bitwise COR
MPI_MAXLOC	maximal value and corresponding index of the maximal element
MPI_MINLOC	minimal value and corresponding index of the minimal element

In MPI, one can also build its own data type and define the associative and commutative binary operator for reduction.

A second kind of global computation are *parallel prefix* also called *scan*. A `scan` operation calculates all the partial reductions on the data stored locally on the processes.

Syntax in MPI is the following:

```
int MPI_Scan( void *sendbuf,void *recvbuf,int count,
    MPI_Datatype datatype,MPI_Op op,MPI_Comm comm )
```

Calling this procedure allows one to perform a *prefix reduction* on the data located in `sendbuf` on each process with the result available in the memory address `recvbuf`. Figure 2.4 illustrates graphically the difference between these two global computation primitives: `reduce` and `scan`.

```
MPI_Scan( vals, cumsum, 4, MPI_INT, MPI_SUM,
    MPI_COMM_WORLD )
```

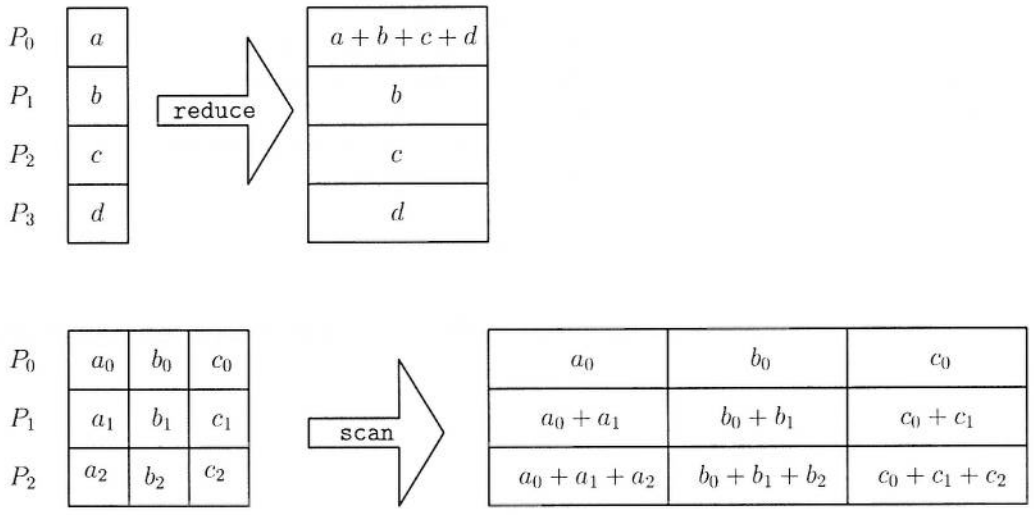

Fig. 2.4 Visualizing a reduce operation and a parallel prefix (or scan) operation

We described by syntax of `reduce` and `scan` using the C binding because the C++ binding is not any longer updated since MPI-2. In practice, we often program using the modern oriented-object C++ language and call the MPI primitives using the C interface. Recall that the C language [1] is a precursor of C++ [2] that is *not* an oriented-object language, and manipulates instead data structures defined by the keyword `struct`.

These global computations are often implemented internally using spanning trees of the underlying topology of the interconnection network.

2.3.7 *Defining Communication Groups with Communicators*

In MPI, communicators allow one to group processes into various groups of communications. Each process is included in a communication and is indexed by its *rank* inside this communication group. By default, `MPI_COMM_WORLD` includes all the P processes with the rank being an integer ranging from 0 to $P-1$. To get the number of processes inside its communication group or its rank inside the communication, we use the following primitives in MPI: `int MPI_Comm_size` (MPI_Comm comm, int *size) and `int MPI_Comm_rank` (MPI_Comm comm, int *size).

For example, we create a new communicator by removing the first process as follows:

WWW source code: `MPICommunicatorRemoveFirstProcess.cpp`

```
// filename: MPICommunicatorRemoveFirstProcess.cpp
#include <mpi.h>

int main(int argc, char *argv[])
    {
    MPI_Comm comm_world, comm_worker;
    MPI_Group group_world, group_worker;
    comm_world = MPI_COMM_WORLD;

    MPI_Comm_group(comm_world, &group_world);
    MPI_Group_excl(group_world, 1, 0, &group_worker)
        ;

    /* process 0 is removed from the communication
       group */

    MPI_Comm_create(comm_world, group_worker, &
       comm_worker);
}
```

In this second listing, we illustrate how to use communicators:

WWW source code: MPICommunicatorSplitProcess.cpp

```
// filename: MPICommunicatorSplitProcess.cpp
#include <mpi.h>
#include <stdio.h>
#define NPROCS 8

int main(int argc, char *argv[])
    {
    int *ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};

    MPI_Group orig_group, new_group;
    MPI_Comm new_comm

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    sendbuf = rank;

    // Retrieve the intial group
    MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

    if (rank < NPROCS/2)
        MPI_Group_incl(orig_group, NPROCS/2, ranks1,
            &new_group);
 else
    MPI_Group_incl(orig_group, NPROCS/2, ranks2, &
        new_group);

// create new communicator
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm
    );

// global computation primitive
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT,
    MPI_SUM, new_comm);
MPI_Group_rank (new_group, &new_rank);
printf("rank= %d newrank= %d recvbuf= %d\n", rank,
    newrank, recvbuf);

MPI_Finalize();
}
```

`MPI_Comm_create` is a collective operation. All processes of the former communication group need to call it, even those who do not belong to the new communication group.

2.4 Synchronization Barriers: Meeting Points of Processes

In the coarse-grained parallelism mode, processes execute large chunks of computations independently from each other. Then they wait for each other at a *synchronization barrier* (see Fig. 2.5, `MPI_Barrier` in MPI), perform some send/receive messages, and proceed their program execution.

2.4.1 A Synchronization Example in MPI: Measuring the Execution Time

For example, let us illustrate a way to measure the parallel time of a MPI program with a synchronization barrier. We shall use the procedure `MPI_Wtime` to measure time in MPI. Consider this master/slave code:

WWW source code: `MPISynchronizeTime.cpp`

```
// filename: MPISynchronizeTime.cpp
double start, end;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

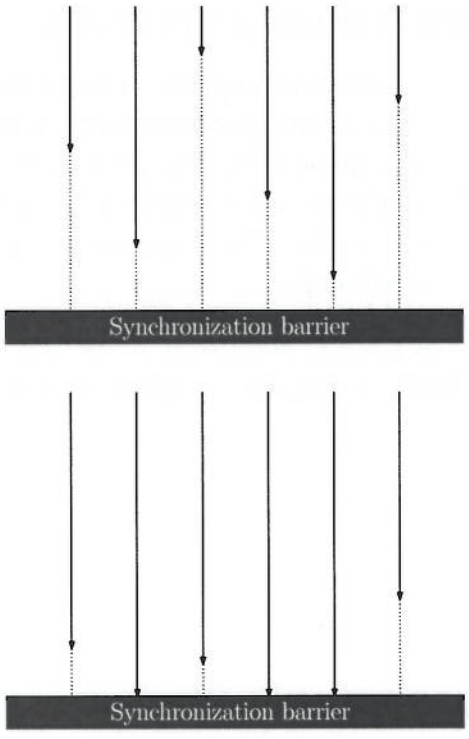

Fig. 2.5 Conceptual illustration of a synchronization barrier: processes wait for each other at a synchronization barrier before carrying on the program execution

```
MPI_Barrier(MPI_COMM_WORLD); /* IMPORTANT */
start = MPI_Wtime();   .

/* some local computations here */
LocalComputation();

MPI_Barrier(MPI_COMM_WORLD); /* IMPORTANT */
end = MPI_Wtime(); /* measure the worst-case time of
     a process */

MPI_Finalize();

if (rank == 0)
        { /* use time on master node */
      cout<< end-start <<endl; // here we use C++ syntax
        }
```

We can also use a `MPI_Reduce()` procedure to compute the minimum, maximum, and overall sum of all the process times. But this eventually requires to add an extra step for perform the global computation with a reduce operation.

2.4.2 The Bulk Synchronous Parallel (BSP) Model

One of the high-level parallel programming model is called the *Bulk Synchronous Parallel* (or *BSP* for short). This abstract model has been conceived by Leslie G. Valiant (Turing award, 2010) and facilitates the design of parallel algorithms using three fundamental steps that form a "super-step":

1. concurrent computation step: processes locally and asynchronously compute, and those local computation can overlap with communications,
2. communication step: processes exchange data between themselves,
3. synchronization barrier step: when a process reaches a synchronization barrier, it waits for all the other processes to reach this barrier before proceeding another super-step.

A parallel algorithm on the BSP model is a sequence of super-steps. A software library, `BSPonMPI`,[7] allows one to use this programming model easily with MPI.

[7] http://bsponmpi.sourceforge.net/.