Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

MPI in Python

vorgelegt von

Alexander Michael Gerlach

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Software-System-Entwicklung
Matrikelnummer:	7216258
Veranstaltung:	Python im Hochleistungsrechnen
Betreuer:	Jannek Squar

Hamburg, 2019-08-26

Contents

1 Introduction

In the past few years, the popularity of Python increased tremendously, resulting in it becoming one of the most popular programming languages to date. According to the TIOBE index, which ranks the popularity of programming languages based on the number of search results for queries containing the name of the language, Python ranks third, directly ahead of C++ (TIOBE, 2019).There are several factors that contributed to its success. Not only is it relatively easy to learn compared to other languages such as Java or C, but it also can be applied to a wide variety of different contexts, ranging from simple automation scripts to full-fetched web- and machine learning applications. What is more, there is an abundance of open-source packages available online, which facilitates the development of new solutions substantially.
However, one of the main applications Python is used for today is scientific computing. Scientific computing often deals with problems that cannot be solved by traditional methods or where conducting experiments is simply not possible, such as astrophysics or predicting the weather (Mehta, 2015). When dealing with scientific computing, speed and efficiency are becoming crucial factors that need to be considered (Eijkhout, Chow, & van de Geijn, 2014). This is particularly true when a problem involves complex calculations or the processing of large amounts of data.
One of many ways to achieve a boost in computation performance is the Message-Passing Interface (MPI). The goal of this paper is to give an overview of how the Message-Passing Interface works and how it can be utilized in Python.

2 Fundamental Concepts of MPI

The Message-Passing Interface is a message-passing library standard put forward by the MPI Forum which comprises of over 40 organizations, including parallel computing vendors, computer scientists, and application developers (MPI Forum, 2015). Prior to the initial release of MPI-1 in 1992, no standardized way of developing parallel applications existed, making the development of such applications even more difficult than it already was at that time (Kendall, 2019).
Hence, the MPI Forum tried to address these problems by developing a standardized approach based on the message-passing programming model, which was most commonly used by the existing libraries (Kendall, 2019). The main goal was to develop a standard for designing message-passing programs that is practical, portable, efficient, and flexible (MPI Forum, 2015). The standard itself, however, is not an implementation, but rather a specification of what a corresponding implementation

should look like.
This chapter is going to elaborate the fundamentals of the MPI standard as well as the setup of MPI in order to use it with Python.

2.1 Distributed Memory Model

As already mentioned above, the MPI standard is based on the message-passing programming model, also known as the distributed memory programming model, where data is moved from the address space of one process to the one of the other (MPI Forum, 2015). This is done by allowing the processes to send and receive messages containing the data that needs to be transmitted. Figure 1 illustrates the basic structure of the distributed memory model. It is important to be aware of the differences between a process and a thread. A process is a self-contained execution environment and has its own address space, whereas a thread exists within a process and shares the memory of the process with other threads (Oracle, 2019).

In contrast to the shared memory model, where all processes access the same address space, the distributed memory model is characterized by each process having its own local memory (Blaise, 2019). This approach yields many advantages, but also disadvantages.
Firstly, with each process operating independently, cache coherency is not of concern (Blaise, 2019). The problem of cache coherency especially occurs in shared memory architectures based on the non-uniform memory access (NUMA) approach. In the case of NUMA, every process has its own local memory, but the physically distributed memory is treated as a logically shared memory (Eijkhout et al., 2014). Hence, every process is able to access the memory of the other processes. Nonetheless, if two different processes are holding a reference to the same memory location and are trying to change it at the same time, it becomes difficult to determine the behaviour of the program (Eijkhout et al., 2014).
Secondly, it can be easily scaled as additional processes and memory do not increase traffic as it would in a shared memory system (Blaise, 2019).
On the downside, while the shared memory model allows data to be easily shared among processes, with the distributed memory model it is up to the programmer to define how and when data is exchanged between processes, which may be more difficult to work with (Eijkhout et al., 2014; Blaise, 2019).
What is more, even though each process can access its local memory relatively fast, retrieving data from the memory of a remote process takes longer, as the process trying to access the data may have to wait for the other process to be ready (Blaise, 2019).

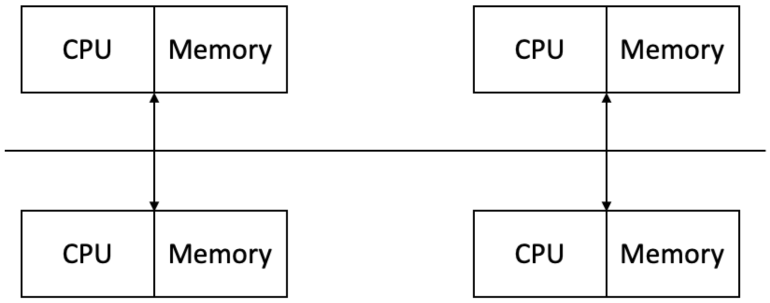

Figure 1: Distributed Memory Model

2.2 MPI in Python

There are several packages available that add MPI functionality to Python. Nevertheless, the probably most common package used for writing MPI programs in Python is mpi4py. mpi4py is based on the latest MPI-3 standard and, thus, facilitates switching to Python from another scientific language, such as C or C++, since the semantics are quite similar (Dalcin, 2019). In addition, mpi4py not only supports the communication of buffer-like objects, but also the transfer of Python objects. However, this is not recommended due to the increase in overhead resulting from pickeling and unpickeling the objects prior to their transmission (Dalcin, 2019).
In order for mpi4py to work properly, an actual implementation of the MPI standard, such as MPICH or OpenMPI, and Cython have to be installed on the system in addition to the package itself. The terminal commands for installing the required packages can be seen in Listing 1.

```
brew install mpich # Mac OS
apt install mpich # Ubuntu
sudo pip install mpi4py
sudo pip install cython
```

Listing 1: Installing MPI and its Dependencies

2.3 Groups and Communicators

In order for the processes to able to send and receive messages, a communication context needs to be defined to identify the receiver and sender of a message. This is done by assigning a unique identifier, called a rank, in the form of an integer between 0 and $n - 1$ to each process, whereas n is the total number of processes (GWDG, 2019; MPI Forum, 2015). In accordance with MPI taxonomy, the communication context is called a communicator (GWDG, 2019; MPI Forum, 2015).

Principally, two communicator types can be differentiated.
Intra-communicators are used for communicating within a group and are the most commonly used type of communicator (MPI Forum, 2015). A group is an ordered collection of processes, which are able to communicate with each other using their ranks and is defined by each intra-communicator (MPI Forum, 2015).
Inter-communicators, on the other hand, define a communication context between two non-overlapping groups of processes that allows them to communicate with each other (MPI Forum, 2015). This is particularly helpful in situations where multiple parallel modules need exchange messages (MPI Forum, 2015).

In mpi4py, the base intra-communicator is `COMM_WORLD`. It initializes `COMM_WORLD` with an instance of the `MPI.Comm` class after executing the mpirun command, which is the base class for intra-communicators (Dalcin, 2019; GWDG, 2019). It contains the number of processes defined in the -n argument of the mpirun command and can be used to create additional communicators (Dalcin, 2019). The methods `Get_size()` and `Get_rank()` can be used to inquire the number of processes in the communicator and the rank of each process. Listing 2 shows the terminal command for executing a MPI program with two processes. Listing 3 shows a simple MPI program that queries the size of the communicator as well as rank of each process.

```
# Executes example.py using a communicator with 4 processes
mpirun -n 2 python example.py
```

Listing 2: Executing a MPI program in Python

Upon executing the mpirun command on Listing 3, the program will be executed two times. The print statement in line 6 will only be executed once, since the if-statement in line 5 restricts it to the processor with rank 0. However, the print statement in line 7 will be executed twice, as it is not restricted to any processor. Listing 4 shows the output after executing the mpirun command on the example.

```python
import mpi4py as MPI

comm = MPI.Comm_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    print(f'Communicator size: {size}')
print(f'Rank {rank} reporting for duty')
```

Listing 3: Simple MPI Program

```
Communicator size: 4
Rank 0 reporting for duty
Rank 1 reporting for duty
```

Listing 4: Output of Example Program

3 Communication Types

This chapter delineates the basic communication types defined in the MPI standard. When sending data from one process to another, the data is transmitted as a stream of bytes. mpi4py offers two different methods for sending and receiving data. Upper-case functions are used to send and receive buffer-like objects, that is, objects that do not have to be serialized prior to their transmission, e.g. Numpy arrays (Dalcin, 2019). Lower-case functions on the other hand can be used to communicate Python objects (Dalcin, 2019). However, it is strongly recommended to stick to the upper-case functions, as python objects need to be serialized prior to their transmission, which may impose significant overheads on memory and processor usage (Dalcin, 2019). Hence, the following sections are mainly focusing on the the use of the upper-case functions.

3.1 Point-To-Point Communications

Point-to-point communications are the simplest form of communication between two processes. It involves sending data from one process to another. Regardless of whether the lower- or upper-case functions are being used, sending data to another process typically involves specifying the data that needs to be transmitted, the destination process, as well as a message tag, whereas the latter two may be omitted. If that is the case, both will be set to the default value, which is 0 (GWDG, 2019). Respectively, the receiving process needs to specify the source of the message, the

message tag, as well as a buffer of sufficient size where the data of the message can be stored (GWDG, 2019). The size of the buffer must be at least equal to the size of the transmitted data (GWDG, 2019).
If the actual message tag and source are not of importance, MPI.ANY_TAG and MPI.ANY_SOURCE can be used instead, allowing the process to receive any message from any source.
Listing 5 illustrates the basic syntax for sending and receiving buffer-like objects in mpi4py.

```python
buf = [data, count, type]
comm.Send(buf, dest=RANK, tag=MESSAGE_TAG)
comm.Recv(buf, source=RANK, tag=MESSAGE_TAG, status=STATUS_OBJ)
```

Listing 5: Send and Recv Syntax

As Listing 5 shows, the actual data is being transmitted as a list containing the data itself as well as additional information about the data.
count defines the number of elements in the buffer and type defines the type of the data (MPI Forum, 2015). In some cases, both count and type can be omitted if they are implied by the data itself, for instance when sending Numpy array objects. The example given in Listing 6 shows how a numpy array is sent from one process to another. The output of the program after executing it with two or more processes can be seen in Listing 7.

```python
import numpy as np
import mpi4py

a_size = 5
if rank == 0:
    send_buf = np.random.randint(-100, 100, a_size, dtype='i')
    print(f'Data to be transmitted: {send_buf}')
    comm.Send(send_buf, dest=1, tag=14)
elif rank == 1:
    recv_buf = np.zeros(a_size, dtype='i')
    print(f'Data before transmission: {recv_buf}')
    comm.Recv(recv_buf, source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG)
    print(f'Data after transmission: {recv_buf}')
```

Listing 6: Point-to-Point Example

```
Data to be transmitted: [ 0 42 72 71 98]
Data before transmission: [0 0 0 0 0]
Data after transmission: [ 0 42 72 71 98]
```

Listing 7: Output of Point-to-Point Example

3.1.1 Blocking and Non-Blocking Communications

Listing 6 is an example for a so-called blocking or synchronous communication. The print statement in line 9 will only be executed, if a corresponding receive method has been posted and all of the transmitted data has been copied to the local memory of the receiving process.
The MPI standard also defines a non-blocking or asynchronous communication mode. Instead of waiting for the message transfer to complete, non-blocking send and receive calls return immediately, regardless of whether or not the transmission has been finished (MPI Forum, 2015). This allows the process to perform other tasks in the meantime.
In mpi4py, the non-blocking send and receive methods have the same names as the blocking methods, only with a preceeding I, that is, `ISend()` and `IRecv()`. After initiating a send or receive call, the process inquires the status of the transfer to verify whether it has been completed or not (MPI Forum, 2015). In mpi4py, the `IRecv()` and `ISend()` methods always return an instance of the Request class, which can then be used to get the current status of the transmission using `MPI.Request.Test()` (Dalcin, 2019).

3.1.2 Message Probing

In the example given in Listing 6, the size of the Numpy array is visible to both processes, which made it quite easy for process 1 to define a buffer of sufficient size. In some cases, however, the receiving process may not know what the actual size of an incoming message is. Thus, it needs to query the message size before saving it to its local memory. This is done by probing the message before executing the receive method (GWDG, 2019).
Listing 8 shows the syntax of the Probe method.

```
comm.Probe(source, tag, status)
```

Listing 8: Probe Syntax

The source and tag parameters have already been discussed above. The status parameter expects an instance of the Status class, which offers several methods that can be used to gain information about the message, such as the source of the message. The size of the data transmitted by the message can be queried with the Get_elements(datatype) method (GWDG, 2019). Using this method, a buffer can be initialized before the receive method is executed. Listing 9 illustrates how this can be achieved.

```
if rank == 0:
    a_size = 3
    send_buf = np.random.randint(-100, 100, a_size, dtype='i')
elif rank == 1:
    info = MPI.Status()
    comm.Probe(MPI.ANY_SOURCE, MPI.ANY_TAG, info)
    sz = info.Get_elements(MPI.INT)
    recv_buf = np.zeros(sz, dtype='i')
    comm.Recv(recv_buf, source=0, tag=14)
```

Listing 9: Probe Example

3.1.3 Deadlocks

If a process tries to send (receive) data without another process calling the corresponding receive (send) method, a deadlock will occur. Listing 10 exemplifies this scenario. Since the message tags defined in the send and receive methods are different, trying to run the code snippet in Listing 10 will result in a deadlock, as neither the send nor the receive method will return due to the absence of a matching send / receive call.

```
if rank == 0:
    send_buf = np.random.randint(-100, 100, 5, dtype='i')
    comm.Send(send_buf, dest=1, tag=11)
elif rank == 1:
    recv_buf = np.zeros(5, dtype='i')
    comm.Recv(recv_buf, source=0, tag=4)
```

Listing 10: Deadlock Example

To avoid deadlocks, it is recommended to use the MPI.Comm.Sendrecv() method offered by mpi4py. It combines the send and receive methods in a single method to avoid the situation encountered in Listing 10 (Dalcin, 2019; GWDG, 2019). Listing 11 shows the syntax of the method.

```
comm.Sendrecv(send_data, dest=RANK, recv_data, source=RANK)
```

Listing 11: Sendrecv Syntax

3.2 Collective Communications

Collective communications are used to communicate data between multiple processes of a group or groups of processes at the same time (MPI Forum, 2015; Dalcin, 2019). While the MPI standard also defines non-blocking collective communications, the mpi4py package offers blocking versions only (MPI Forum, 2015; Dalcin, 2019).
In general, two types of collective communications can be differentiated, namely collective send and collective receive operations.

3.2.1 Collective Send Operations

Two of the most commonly used collective send operations in MPI are `Bcast()` and `Scatter()`, whereas Bcast stands for broadcast. With `Bcast()`, the same data is distributed from a root process to all the other processes, including the root process itself. As illustrated by Listing 12, `Bcast()` requires two parameters, namely a send buffer containing the data to be sent as well as the rank of the root process. It does not require a separate buffer to store the received data, as the size of the send and receive buffers is the same. Hence, `send_buffer` is used to store the data after the transmission.
Listing 13 exemplifies the broadcast operation. The output of the program after executing it with 2 processes can be seen in Listing 14.

```
comm.Bcast(send_buffer=[data, count, MPI.DATA_TYPE],  root=RANK)
```

Listing 12: Broadcast Syntax

```
send_buf = rank * np.arange(5, dtype='i')
print(f'Process {rank} - Data before transmission: {send_buf}')
if rank == 0:
    send_buf = np.random.randint(-100, 100, 5, dtype='i')
comm.Bcast(send_buf, root=0)
print(f'Process {rank} - Data after transmission: {send_buf}')
```

Listing 13: Broadcast Example

```
Process 0 - Data before transmission: [0 0 0 0 0]
Process 0 - Data after transmission: [-54   3  71  -2 -67]
Process 1 - Data before transmission: [0 1 2 3 4]
Process 1 - Data after transmission: [-54   3  71  -2 -67]
```

Listing 14: Output of Broadcast Example

Scatter() works quite similar to Bcast(). In contrast to the broadcast operation, each process receives a different portion of the data with each portion being equal in size. In other words, the process with rank k receives the k-th portion of the data. Thus, the size of the data that is supposed to be distributed to the other processes needs to be $n * data\_size$, whereas n is the total number of processes the data is being scattered to and $data\_size$ represents the size of the data each process receives (GWDG, 2019).
As Listing 15 shows, the syntax of Bcast() and Scatter() are quite similar. Since each process, including the root process, only receives a portion of the transmitted data, Bcast() requires a separate receive buffer of size $data\_size$.
Listing 16 gives an example of how to use Scatter() and Listing 17 shows the output of the example after executing it with 2 processes.

```
comm.Scatter(send_buffer=[data, count, MPI.DATA_TYPE], recv_buffer=[data,
 ↪  count, MPI.DATA_TYPE], root=RANK)
```

Listing 15: Scatter Syntax

```
recv_buf = np.zeros(5, dtype='i')
send_buf = None
if rank == 0:
    send_buf = np.arange(5 * size, dtype='i')
    print(f'Data to be transmitted {send_buf}')
comm.Scatter(send_buf, recv_buf, root=0)
print(f'Process {rank} received {recv_buf}')
```

Listing 16: Scatter Example

```
Data to be transmitted [0 1 2 3 4 5 6 7 8 9]
Process 0 received [0 1 2 3 4]
Process 1 received [5 6 7 8 9]
```

Listing 17: Output of Scatter Example

3.2.2 Collective Receive Operations

The simplest collective receive operation is `Gather()`. `Gather()` does quite the opposite of what `Bcast()` does. It collects an equal portion of data from every process in a group and saves it into the local memory of the root process. As Listing 18 shows, `Gather()` expects three parameters: the rank of the root process, a send buffer as well as a receive buffer.
The size of the receive buffer must be $n * data\_size$, where n represents the number of processes data is gathered from and $data\_size$ is the size of the data portion retrieved from each process (GWDG, 2019).
Listing 19 gives an example of how `Gather()` can be used with the output displayed in Listing 20.

```
comm.Gather(send_buffer=[data, count, MPI.DATA_TYPE], recv_buffer=[data, count,
 ↪  MPI.DATA_TYPE], root=RANK)
```

Listing 18: Gather Syntax

```
recv_buf = None
send_buf = (rank + 1) * np.arange(5, dtype='i')
print(f'Process {rank} data {send_buf}')
if rank == 0:
    recv_buf = np.zeros(5 * size, dtype='i')
comm.Gather(send_buf, recv_buf, root=0)
if rank == 0:
    print(f'Data received: {recv_buf}')
```

Listing 19: Gather Example

```
Process 1 data [0 2 4 6 8]
Process 0 data [0 1 2 3 4]
Data received: [0 1 2 3 4 0 2 4 6 8]
```

Listing 20: Output of Gather Example

3.2.3 Other Collective Communications

There are several additional collective communication modes defined in the MPI standard apart from the ones discussed above. For instance, Scatter() as well as Gather() offer a vector based variant, known as Scatterv() and Gatherv(), where the portion of each data section can be defined individually (GWDG, 2019).
Other interesting communication modes are the so-called Reduce() and Alltoall() functions. Reduce() gathers an equal portion of data from each process within a group, performs a reduction operation on each element of the data, and saves it to the receive buffer of the root process. Alltoall() on the other hand is a mixture of both collective send and receive operations. It collects data from all processes within a group and sends the k-th portion of every data block gathered to the process with rank k. However, due to the scope of this paper, they will not be discussed in detail.

3.3 One-Sided Communication

Both Point-to-Point and collective communications are synchronous, two-sided communication modes, meaning that each communication call requires a sender and receiver. Requiring two parties to participate in the communication has some disadvantages, as it can for instance cause a delay when one of the parties needs to wait for the counterpart to be ready (Nguyen, 2014).
To address these issues, the MPI Forum implemented one-sided communications with the release of the MPI-2 standard (Nguyen, 2014).

One-sided communication, also known as Remote Memory Access (RMA), allows a process to access the memory of a remote process without interrupting the remote process of its current task and requiring it to interact (Nguyen, 2014).
For one-sided communications to work, the remote process, also known as the target process, needs to define a so-called window, which is a shared memory region (Dalcin, 2019; Nguyen, 2014). Windows define a specific memory area on the target that can then be accessed by a so-called origin process to transfer data to or retrieve data from the target process. (Eijkhout et al., 2014).
In mpi4py, windows can be created on several different ways. With MPI.Win.Create() a window is created by passing a buffer to the the function (Eijkhout et al., 2014). MPI.Win.Allocate() on the other hand lets MPI handle the memory allocation (Eijkhout et al., 2014). This method has also been used in Listing 21. Lastly, MPI.Win.Create_dynamic() creates a window and postpones the memory allocation (Eijkhout et al., 2014).

The MPI standard defines two different modes to access the window of a target process. With active RMA, the window can only be accessed in a specific time period, a so-called epoch (Eijkhout et al., 2014). An epoch is typically initiated by calling the `MPI.Win.Fence()` method and lasts until the `MPI.Win.Fence()` method has been called a second time (Eijkhout et al., 2014). During this time period, the window can be accessed by the origin process. In passive RMA, the window can be accessed at any time by the origin process (Eijkhout et al., 2014). The origin process locks the target window, performs some transfer operations, and unlocks the window again to make it accessible by other processes again (Eijkhout et al., 2014).

To actually transfer data to a window, the MPI standard defined three methods: `MPI.Win.Put()`, `MPI.Win.Get()`, and `MPI.Win.Accumulate()`.
As the name already suggests, `MPI.Win.Put()` is used to transfer data to the target window. It requires two parameters: a buffer-like object containing the data that should be saved to the remote window and the rank of the target process.
Secondly, `MPI.Win.Get()` is used to retrieve data from the window. It too requires a buffer-like object where the data can be saved to as well as the rank of the target process.
Both `MPI.Win.Put()` and `MPI.Win.Get()` also accept an offset as a parameter that can be used to access a different portion of the window.
Lastly, `MPI.Win.Accumulate()` performs a reduction operation on the data that is transferred to the window of the target process (Eijkhout et al., 2014).
Listing 21 exemplifies the put and get operations and Listing 22 shows the output of the example.

```
comm: MPI.Comm = COMM_WORLD
rank: int = comm.Get_rank()

# Defining the size of the window for process 0
window_size = 100
dtype_size = MPI.INT.Get_size()
if rank == 0:
 nbytes = dtype_size * window_size
else:
 nbytes = 0

# Creating the actual window
window: MPI.Win = MPI.Win.Allocate(nbytes, itemsize, comm=comm)


window.Fence()  # Initializing epoch
if rank == 1:
```

```python
    localmem = rank * np.arange(15, dtype='i')
    window.Put(localmem, 0)
window.Fence()  # Ending epoch

window.Fence()  # Initializing second epoch
if rank == 2:
    localmem = np.zeros(10, dtype='i')
    print(f'Initial content of localmem: {localmem}')
    window.Get(localmem, 0, 5). # Offset set to 5
    print(f'Content after executing get method: {localmem}')
window.Fence()  # Ending second epoch
```

Listing 21: Example of using the Put and Get methods

```
Initial content of localmem: [0 0 0 0 0 0 0 0 0 0]
Content after executing get method: [ 5  6  7  8  9 10 11 12 13 14]
```

Listing 22: Output of the One-Sided Communication Example

4 Discussion and Conclusion

The goal of this paper was to give a brief overview of the Message-Passing Interface and how it can be used in Python using the mpi4py package. The first chapter discussed the basic structure of MPI and how to set up MPI to use it with Python. The second chapter focused on the different communication modes defined in the MPI standard and how they can be used in Python with the mpi4py package.
It is safe to say that the mpi4py package adds a great set of tools to Python that might come in handy in many different situations, especially in combination with other Python packages, such as Numpy. For instance, instead of executing a sorting algorithm in serial, with mpi4py the work can easily be split up among multiple different processes, resulting in a substantial boost in performance. This is particularly helpful when dealing with large amounts of data.
What is more, mpi4py makes it significantly easier to write MPI programs, as it automates a lot of work that has to be done manually when working with MPI in other languages. For example, it is not necessary to call MPI.Init() and MPI.Finalize() when working with mpi4py.
Still, the documentation of the mpi4py package is rather insufficient, specifically when dealing with advanced topics, such as one-sided communications. Hence, it often takes quite a lot of research to understand the way an MPI function works in Python and it often involves reading the documentation of other MPI libraries.

This consumes a lot of time and adds yet another hurdle to the rather complex nature of the MPI framework.
Nevertheless, mpi4py offers a lot and it is strongly recommended to take a closer look at the package and how it can maybe be utilized for existing or upcoming projects.

References

Blaise, B. (2019). *Introduction to parallel computing.* Retrieved 2019-06-01, from `https://computing.llnl.gov/tutorials/parallel_comp`

Dalcin, L. (2019). *Mpi for python.* Retrieved 2019-06-01, from `https://mpi4py.readthedocs.io/en/stable/index.html`

Eijkhout, V., Chow, E., & van de Geijn, R. (2014). *Introduction to high performance scientific computing.* Retrieved from `http://pages.tacc.utexas.edu/~eijkhout/Articles/EijkhoutIntroToHPC.pdf`

GWDG. (2019). *Mpi4py.* Retrieved 2019-06-01, from `https://info.gwdg.de/wiki/doku.php?id=wiki:hpc:mpi4py`

Kendall, L. (2019). *A comprehensive mpi tutorial resource.* Retrieved 2019-06-01, from `https://mpitutorial.com`

Mehta, H. K. (2015). *Mastering python scientific computing.* Packt Publishing Limited. Retrieved from `https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781783288823`

MPI Forum. (2015). *Mpi: A message-passing interface standard.* Message Passing Interface Forum. Retrieved from `https://www.mpi-forum.org/docs/`

Nguyen, L. Q. (2014). *Mpi one-sided communication.* Retrieved 2019-06-19, from `https://software.intel.com/en-us/blogs/2014/08/06/one-sided-communication`

Oracle. (2019). *Processes and threads.* Retrieved 2019-08-25, from `https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html`

TIOBE. (2019). *TIOBE Index for August 2019.* Retrieved 2010-08-16, from `https://www.tiobe.com/tiobe-index/`