HPCmatlab: User Guide

Version 1.0

Last Updated: 11/28/2016

Xinchen Guo, Mukul Dave, Ayush Mishra and Mohamed Sayeed

GitHub: https://github.com/xinchenguo/HPCmatlab

ASU Research Computing
Arizona State University
Tempe, AZ 85287

Contents

1 Introduction

HPCmatlab is a framework developed for prototyping of parallel applications on Matlab. HPCmatlab enables Shared Memory Programming using Pthreads, Distributed Memory Programming using the Message Passing Interface (MPI) and Parallel I/O using the ADIOS package. We are currently releasing just the MPI part of the framework in version 1.0 and this user guide focusses on how to use it.

1.1 Motivation for HPCmatlab

Matlab is now a ubiquitous programming tool known for its ease of programmability and it's Integrated Development Environment. Users who are new to programming often take to Matlab for prototyping their applications. However, very soon they may realize that their applications are computationally very intensive. Matlab, being an interpreted language is inherently supposed to have relatively larger run times. Hence, the only way out for users would be to use parallel computing.

Mathworks' Parallel Computing Toolbox (PCT) and the Matlab Distributed Computing Server (MDCS) have many features that may help users carry out Shared Memory Programming or Distributed Memory Programming using Matlab. But, they require purchasing a separate license. Matlab licenses provided by Academic and Research institutions often do not include these licenses. The user also has to overcome the additional learning curve, especially for functions like message passing.

Having these issues in mind, we have developed this framework through which Matlab users can create parallel programs in Matlab without the need of purchasing any additional licenses. Benchmarking results show that the performance of this framework is better than MDCS or PCT in most cases and comparable to corresponding features in a lower level language like C.

1.2 Message Passing through HPCmatlab

Message Passing Interface enables the movement of data from the address space of one process to that of another process. One can think of it as a way for processes to communicate with each other when different parts of a program are running on different processes. Without the ability to communicate, parallel programs would be able to efficiently address only a small class of applications which are embarrassingly parallel.

The semantics of MPI functions used in HPCmatlab are same as the **C language** semantics of MPI with some minor changes as listed at the start of Section 3. Hence, learning these semantics enables users to easily incorporate their prototype parallel applications in Matlab to a low-level language later if they desire. The MPI standard is a good reference document for referring to the syntax for different MPI functions and knowing more about the Message Passing Interface: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. Please be informed that only a subset of functions described in this document are supported by HPCmatlab and the syntax to refer to for use in HPCmatlab would be the C Language syntax provided in it and not the Fortran syntax. This is because the MEX functions that have been used to wrap the MPI functions have been written in C Language. **Appendix A** lists all the MPI functions which are a part of HPCmatlab along with their syntax in HPCmatlab.

2 Getting Started

The current version of HPCmatlab (Version 1.0) has been installed on the Saguaro and Ocotillo computing clusters at ASU Research Computing. Users who do not already have an account on the cluster can request an account through the ASU Research Computing website: https://researchcomputing.asu.edu/. HPCmatlab has been installed in a pre-compiled form. One can start using the MPI functions in their Matlab programs simply by loading the HPCmatlab module. The path of the directory where the executable files of the framework lie are added to the Matlab search paths by loading the module. More information on how to submit jobs on Saguaro and Ocotillo, monitoring the cluster status, software modules and system documentation can be found here: http://saguaro.a2c2.asu.edu/, http://ocotillo1.a2c2.asu.edu/.

2.1 Matlab and MPI modules support

The version of Matlab used for HPCmatlab is **R2015a**. The MPI library module on Saguaro used for compiling the framework is **mvapich2/2.1-gnu-4.9.2**. The MPI module which is loaded when loading the HPCmatlab module is **intel-mpi/5.0** (along with the Matlab environment module). This is because we have had good success using this. If the user wishes, he/she can switch to a different MPI module like the MVAPICH module mentioned above by first unloading the Intel MPI module and then loading the other one. Using older versions of Matlab (that support running MEX files) also should not be a problem. Similarly, using other latest MPI implementations like OpenMPI also should not be a problem as the MPI syntax remains the same except if the version is a bit old and some of the semantics and structures are different. In general, try to use the latest versions. Please feel free to report any issues to us while running with other implementations so that we can look into it and improve the framework and make it more versatile.

2.2 Running Interactively on Saguaro/Ocotillo

For example, we wish to interactively run the simple send-receive script from Section 3 using 2 processes. We need to ask for 2 cores on a compute node for this, which can be achieved by the following command. We can also mention other details such as wall-time, etc.

```
$ interactive –n 2
```

Once we are on a compute node, we load the HPCmatlab module. As Saguaro and Ocotillo use SLURM as the resource management system, we use `srun` for running MPI jobs. For our example, where we want to launch **2** MPI processes and run a script named **test.m** on them, we use the following command at the shell prompt. This will run our parallel program in MATLAB and the user can monitor the output as it runs.

```
$ module load hpcmatlab/1.0
$ srun -n 2 --mpi=pmi2 matlab -r “test,exit”
```

2.3 Running in Batch Mode

It is sometimes useful to have a parallel program run interactively, say while debugging or with less number of processes. However, batch mode is the preferred method to run highly computationally intensive jobs on a cluster requiring more number of processes. This is because we can submit jobs to the queue and the resource manager will start running it as soon as the resources that we have asked for become available. Here is an example bash script written in a file **matlabtest.sh** to submit a job for running the parallel program script **test.m** from Section 3 on 2 cores.

```
#!/bin/bash
#SBATCH -n 2                                # request allocation for 2 tasks
#SBATCH -t 00:05:00                         # maximum wall-time
#SBATCH -o test.out                         # standard output file
#SBATCH -e test.err                         # standard error file
module load hpcmatlab/1.0                   # load HPCmatlab module
srun -n 2 --mpi=pmi2 matlab -r "test,exit"  # launch program on 2 processes
```

The statements on the right after the # sign are comments. The `#SBATCH` directive is used by SLURM to allocate resources according to the details specified. Finally, to submit a job using the above script file **matlabtest.sh**, `sbatch` command can be used at the shell prompt as under.

```
$ sbatch matlabtest.sh
```

The above commands are valid for the SLURM resource management system. If a cluster uses some other resource management system, the directives and commands may be different. Here is more information on creating and using sbatch scripts:
http://saguaro.a2c2.asu.edu/howto/sbatch_files.php

2.4 Using HPCmatlab on a Different Cluster

HPCmatlab files have been installed on Saguaro and Ocotillo and it has been included as an environment module too, which will not be the case on other clusters. To use this framework on a different platform than Saguaro or Ocotillo, please follow these directions:

1. Download the HPCmatlab package archive to the desired location.
2. Unpack the archive. The README file can be referred to for basic information.
3. Make sure Matlab's "mex" compiler is in your $PATH. Also, mpicc and mpicxx should be in $PATH with all environment variables set up for MPI libraries.
4. Run "make" to compile the MEX functions from source code.
5. For running your Matlab scripts which use HPCmatlab functions, manually load the Matlab and MPI environment modules.
6. The path to HPCmatlab MEX files has to be added to the Matlab search paths:

- **addpath** function can be used from **inside** Matlab, OR
- The **MATLABPATH** environment variable can be set to the required path.
- Note that executable files of the HPCmatlab functions lie in the directory 'matlab'.

7 Instead of `srun`, `mpirun` or `mpiexec` may have to be used according to the MPI module and cluster configurations.

8 The example script for running a Matlab script would be something like this:

```
$ module load <MATLAB module>
$ module load <MPI module>
$ export MATLABPATH = ${HPCmatlab}/matlab
$ mpirun -n 2 matlab -r "test,exit"
```

2.5 Contact Us

We hope that HPCmatlab helps in significantly speeding up your applications. We would be glad to provide additional guidance to you and even work with you for parallelizing your applications using HPCmatlab.

As they say, no software is bug free. If you encounter any issues and feel this might be a bug in the software, do report it to us.

Not all routines and features of MPI are supported in HPCmatlab. If you think a particular function or feature is essential to your application and should be supported, write to us and we will try our best to include the functionality in our next version.

Please use our GitHub page for your comments, suggestions and feedback:

https://github.com/xinchenguo/HPCmatlab/issues

3 Demonstration of HPCmatlab using Examples

We will demonstrate the use of MPI routines using some basic examples. Some explanation and rationale behind the function calls is provided or inserted as comments in the example scripts. The example scripts are included in the directory 'examples' in the root directory of the package. Here is the path to the module files on Saguaro: /packages/6x/hpcmatlab/1.0.

The MPI routines along with their syntax are listed in Appendix A. The syntax to be used for MPI routines is same as the C Language syntax and can also be looked up in the several references available throughout the web like the MPI standard document. Users can also use these resources for getting advanced information about usage and implementation of particular functions. Here are some of the key points and departures from the usual MPI syntax to be kept in mind while using the MPI routines in HPCmatlab. Note the following usages in the example scripts that follow.

- All MPI functions in HPCmatlab **have** to be called with **one output** in the form of the error code. It will show up as an error, if the function is called without an output. The error codes can be used for dynamic error detection.
- In C language, pointers to the buffers have to be passed in the communication calls. However, note that in the following examples, no operators are used to pass any pointers as Matlab has no such thing. It is sufficient that the buffer name itself is mentioned in the relevant fields without any other operators. The pointer to the buffers are accessed by internal manipulation within the HPCmatlab MEX functions (one less thing to worry about). So, **no pointers, just array names**!
- MPI uses structures to store information about certain things. E.g. structures storing information about the status of a communication call like MPI_Recv are of the type MPI_Status. Specific instances of these structure types have to be defined using an **assignment** operation, e.g. `status = MPI_Status;`

3.1 Initializing MPI Environment

Before starting to communicate data between processes, a few basic things are to be taken care of, like initializing MPI, getting the rank of process, etc. These things are also necessary in a normal MPI program but there may be minor changes while using HPCmatlab. Here is a code section which will be a part of all programs using HPCmatlab MPI, although with some changes, depending upon what the user intends to do further in the program.

```
GetMPIEnvironment;      % Set up HPCmatlab MPI Environment
status=MPI_Status;      % Define a 'struct' for storing status of MPI_Recv
window=MPI_Win;         % Define a 'struct' for storing window information
commsize=0;             % Initialize variable to store size of communicator
rank=0;                 % Initialize variable to store rank of the process
err=MPI_Init(0,0);      % Initialize MPI
err=MPI_Comm_size(MPI_COMM_WORLD,commsize);     % Get size
err=MPI_Comm_rank(MPI_COMM_WORLD,rank);         % Get rank
disp(['I am rank ' num2str(rank) ' out of ' num2str(commsize)]); % Display
```

Notes on the above code section:

1. Note the use of assignment operations to define different structure variables which would be required throughout the program. More of them can be defined as needed. Those not needed can be omitted. E.g. If the user does not intend to use One-Sided Communication, he/she does not have to define the window structure.
2. `MPI_Init` currently does not support arguments.
3. Variables used to store size of the communicator and rank of the process have to be pre-allocated. Size is the total number of processes. If the number of processes are *n*, then the ranks range from *0* to *n-1*.
4. After the use of MPI is no longer needed in the program, it has to be cleaned up using the finalize command:

```
err=MPI_Finalize;
```

3.2 Point-to-Point Communication

This section provides examples of point-to-point communication using HPCmatlab. The list of routines supported in HPCmatlab is as follows.

- MPI_Send - Blocking Send
- MPI_Recv - Blocking Receive
- MPI_Isend - Non-blocking Send
- MPI_Irecv - Non-blocking Receive
- MPI_Sendrecv - Simultaneous Send and Receive
- MPI_Bsend - Buffered Send

3.2.1 Send and Receive

Here is a simple script showing a send-receive operation between two processes. 'If' statements are used with the ranks to run a particular set of instructions on a given process.

```
N=5;
tag=123;
% We will send data from rank '0' to rank '1'
if rank==0
    data=1:N;
    err=MPI_Send(data,N,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
end

if rank==1
    % Initialize buffer (array) on rank 1 which will receive data
    recv_buff=zeros(1,N);
    disp('Before communication:');
    recv_buff
    % Now receive N 'double' type elements from rank '0'
    err=MPI_Recv(recv_buff,N,MPI_DOUBLE,0,tag,MPI_COMM_WORLD,status);
    disp('After communication:');
    recv_buff
end
```

Here is what the output of the above program looks like:

```
I am rank 0 out of 2
I am rank 1 out of 2
Before communication:

recv_buff =

     0     0     0     0     0

After communication:

recv_buff =

     1     2     3     4     5
```

3.2.2 Non-blocking Send and Receive

The `MPI_Send` operation is a blocking one, which means that it does not return until the corresponding `MPI_Recv` operation is completed and so, the calling process has to wait till then. If we wish to overlap communication with computation, we can use non-blocking send and receive operations. However, it has to be used with caution when we wish to access the memory region from or to which data is being sent or received after the call.

In the following code section, if we had used blocking sends and receives, it would have resulted in a **deadlock** where both processes are waiting at the `MPI_Recv` call until the other process starts the `MPI_Send` call. It is the user's responsibility to avoid such situations. However, if we are sure that we are not going to work with the data being sent or received immediately after the send-receive call, we can use non-blocking send and receive as below. The following code would perfectly run through without any hitches. We can also use the status object to know at a particular point in the program whether the send-receive calls have completed or not. In the following example, say if we performed some computation immediately after the send-receive calls, the computation time would overlap with the communication time, resulting in less overall run-time. Users are encouraged to learn more about non-blocking communication from other sources, so that they can take advantage of this feature.

```
N=5;
tag1=123;
tag2=456;
req1=MPI_Request;
req2=MPI_Request;
status1=MPI_Status;
status2=MPI_Status;
```

```
if rank==0
    data=1:N;                          % Data to be sent from rank '0' to rank '1'
    recv_buff=zeros(1,N);              % Buffer to receive data from rank '1'
    err=MPI_Irecv(recv_buff,N,MPI_DOUBLE,1,tag2,MPI_COMM_WORLD,req2);
    err=MPI_Isend(data,N,MPI_DOUBLE,1,tag1,MPI_COMM_WORLD,req1);
    % Do some computation which DOES NOT use data to be received above
    err=MPI_Waitall(2,[req1 req2],[status1 status2]);      % Wait for completion
    disp('After communication, on rank 0:');
    recv_buff
end
if rank==1
    data=(1:N)*2;                      % Data to be sent from rank '1' to rank '0'
    recv_buff=zeros(1,N);              % Buffer to receive data from rank '0'
    disp('Before communication:');
    recv_buff
    err=MPI_Irecv(recv_buff,N,MPI_DOUBLE,0,tag1,MPI_COMM_WORLD,req1);
    err=MPI_Isend(data,N,MPI_DOUBLE,0,tag2,MPI_COMM_WORLD,req2);
    % Do some computation which DOES NOT use data to be received above
    err=MPI_Waitall(2,[req1 req2],[status1 status2]);      % Wait for completion
    disp('After communication, on rank 1:');
    recv_buff
end
```

Output:

```
I am rank 0 out of 2
I am rank 1 out of 2
Before communication:

recv_buff =

     0     0     0     0     0

After communication, on rank 1:

recv_buff =

     1     2     3     4     5

After communication, on rank 0:

recv_buff =

     2     4     6     8    10
```

3.2.3 Using MPI_Sendrecv

We used consecutive send and receive operations in the previous example for each process. Instead, we can use MPI_Sendrecv for a simultaneous call to both.

```
N=5;
tag1=123;
tag2=456;
status=MPI_Status;

if rank==0
    data=1:N;
    recv_buff=zeros(1,N);
err=MPI_Sendrecv(data,N,MPI_DOUBLE,1,tag1,recv_buff,N,MPI_DOUBLE,1,tag2,MPI_COMM_WORL
D,status);                              % Simultaneous send-receive using one call
    disp('After communication, on rank 0:');
    recv_buff
end

if rank==1
    data=(1:N)*2;
    recv_buff=zeros(1,N);
    disp('Before communication:');
    recv_buff
err=MPI_Sendrecv(data,N,MPI_DOUBLE,0,tag2,recv_buff,N,MPI_DOUBLE,0,tag1,MPI_COMM_WORL
D,status);                              % Simultaneous send-receive using one call
    disp('After communication, on rank 1:');
    recv_buff
end
```

The above program would produce the same output as the one before that in 3.2.2. But, note that here the send-receive are **blocking** and so they cannot be overlapped with computation as with the non-blocking case. The benefit is that the exchange of information requires half the number of function calls.

3.3 Collective Communication

When we wish to send or receive data to multiple processes at a time, instead of using individual send and receive calls, it is more efficient to use collective calls. Those supported in HPCmatlab are listed below. More information about collective calls can be gained from several available sources, like this one:
http://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node64.html#Node64

- MPI_Bcast - Broadcast same data to all processes
- MPI_Reduce - Reduce data from all processes to one
- MPI_Scatter - Scatter different parts of data to processes in rank order
- MPI_Gather - Gather data from different processes to one in rank order
- MPI_Alltoall - works somewhat like combined MPI_Scatter and MPI_Gather
- MPI_Allgather - like MPI_Gather with all processes receiving the data
- MPI_Allreduce - like MPI_Reduce with all processes receiving the data
- MPI_Barrier - blocks the caller until all group members have called it

The following program incorporates three of the operations listed above. We will go through each part one by one. First, initializing the MPI environment.

```
GetMPIEnvironment;
commsize=0;
rank=0;
err=MPI_Init(0,0);
err=MPI_Comm_size(MPI_COMM_WORLD,commsize);
err=MPI_Comm_rank(MPI_COMM_WORLD,rank);
% The HPCmatlab MPI Environment is set up
disp(['I am rank ' num2str(rank) ' out of ' num2str(commsize)]);
```

1 The data is first scattered from the root process (here, rank 0) to all processes. If `N` is the total number of elements to be scattered and `commsize` is the number of processes, each process gets `n=N/commsize` number of elements in rank order.

```
N=32;               % Total length of array to be scattered
n=N/commsize;       % Number of elements that each process will receive
buff=zeros(1,n);    % Allocate the receive buffer on each process
```

```
if rank==0
    data=rand(1,N);              % Initialize data to be scattered from root process
    disp('starting scatter');
    avg_check = mean(data);      % For validating the Reduce operation later
    tic
    err=MPI_Scatter(data, n, MPI_DOUBLE, buff, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
end

if rank>0
    err=MPI_Scatter(0, n, MPI_DOUBLE, buff, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
end
```

2 After scattering, some operation is performed on each process (e.g. filtering). Remember: this is the ultimate goal i.e. to have each process simultaneously perform operations on different parts of the data to speed up the overall program.

```
disp(['filtering on rank' num2str(rank)]);
avg = mean(buff);
buff = buff - avg;        % Perform some operation on buffer on each process
```

3 We need the newly computed values back together in rank order but now, on all processes and not just the original root process. This is accomplished by using `MPI_Allgather`.

4 Also, note the use of `MPI_Barrier` here. It acts as a “barrier” to the process and does not let it progress until all other processes enter the call. This is called synchronization and is very useful in cases where we want all processes to reach a particular point before going further. Here we use it for a trivial purpose - to make sure the output from different processes does not get mixed up. Barrier synchronization should be used with care as it affects the performance when used unnecessarily.

```
data=zeros(1,N);          % Allocate receive buffer on all processes
if rank==0
    disp('starting gather');
end
err=MPI_Allgather(buff, n, MPI_DOUBLE, data, n, MPI_DOUBLE, MPI_COMM_WORLD);
clear buff

if rank==0
    data                  % Print received array values on any of the processes
end                       % it should be same for all processes
err=MPI_Barrier(MPI_COMM_WORLD); % Synchronization to ensure output is not messed up
if rank==commsize-1
    data                  % Print received array values on any of the processes
end                       % it should be same for all processes
```

5 The use of `MPI_Reduce` is also demonstrated on a single element. It allows us to perform different operations on the data elements while reducing them. Here we use the `MPI_Sum` operation to calculate the overall mean of our data from the mean calculated on each process.

```
avg_sum=0;          % Initialize buffer to receive reduced value
                    % (significant only at root process)
err=MPI_Reduce(avg, avg_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if rank==0
    avg_all=avg_sum/commsize        % Calculate mean from sum
    avg_check                       % Validate using pre-calculated mean
    toc
end
```

At last, we finalize MPI and exit from the process.

```
err=MPI_Finalize;
exit
```

Note that we can still use vast library of convenient built-in functions offered by MATLAB while using MPI to run parallel programs and communicate among processes! Users are encouraged to play around with the program to see how the collective communication works. The batch script for running the above program is also included in the same directory as the script file.

3.4 One-sided Communication

The communication routines that we have seen so far here are two-sided communication routines, i.e. each send call should have a matching receive call. Instead we can use one-sided communication so that only one of the processes (origin) has to call the communication routine. This has advantages in the form of synchronization, less data movement and ease of programming. This topic will not be discussed in detail here as it can be considered as an advanced topic. Here is a good source of information on one-sided communication (Remote Memory Access): https://cvw.cac.cornell.edu/MPIoneSided/default

```
% This example shows the use of MPI one-sided communication calls from HPCmatlab
% Set up and initialize MPI environment
% Initialize a MPI Window object
win = MPI_Win;

n=5;
data=zeros(n,n);
bytes=8*n*n;        % 8 bytes per element for 'double'

if rank==0
    for j=1:n
        for i=1:n
            data(i,j)=n*(j-1)+i;
        end
    end
    disp('On rank 0:');
    data
    err=MPI_Win_create(data, bytes, 8, MPI_INFO_NULL, MPI_COMM_WORLD, win)
    err=MPI_Win_fence(0,win);
    err=MPI_Win_fence(0,win);
end

if rank==1
    err=MPI_Win_create(0, 0, 8, MPI_INFO_NULL, MPI_COMM_WORLD, win)
    err=MPI_Win_fence(0,win);
    disp('At rank 1, before Get from 0:');
    data
    err=MPI_Get(data, n*n, MPI_DOUBLE, 0, 0, n*n, MPI_DOUBLE, win);
    err=MPI_Win_fence(0,win);
    disp('At rank 1, after Get from 0');
    data
end

err=MPI_Win_free(win);
err=MPI_Finalize;
exit
```

4 Best Practices

We believe that Matlab users wanting to speed up their computationally intensive programs will benefit a lot from using this framework. At the same time, users familiar with MPI should surely try out using MPI on Matlab and take advantage of the huge library of in-built functions that it provides.

Both type of users would be expecting that all the salient features of Matlab as well as MPI could also be used while using the MPI module of HPCmatlab. But, unfortunately this is not true. There are limitations on what features or commands or routines can be used and also, how they can be used. However, we feel that these limitations will not deter users from doing some serious computation using the framework.

On the other hand, there are things which do not HAVE to be used in a certain way, but rather should be used in that way to optimize the applications. This section first provides basic explanation of how Matlab's memory model and MPI work and then using that information to explain the limitations and best practices while using our framework.

4.1 Understanding Matlab's memory model

As is common knowledge, arrays in a single buffer element are arranged in memory as contiguous blocks. In Matlab they are arranged in a column-major order. This means that for a two-dimensional array, first all the elements of the first column are arranged together contiguously followed by the second column and so on. This is why it is recommended to go in a column-wise order when looping over all the elements of a matrix in Matlab as this is more efficient as compared to going row-wise.

If we dynamically increase the size of an array by assigning new values which are out of bounds of the original array, then a new memory region is allocated to the buffer and all the elements are copied to the new buffer. This is why pre-allocation is important to prevent applications from slowing down when dealing with very large amount of data. Matlab also gives a warning for this.

4.2 Comments on how MPI communication works

While sending or receiving data using MPI, what data is to be sent is specified by passing the pointer to the first element of the buffer and then the number of elements to be sent along with the size in bytes of each element. This works because as mentioned before, the elements in the same buffer are arranged contiguously. In HPCmatlab, this is enforced by accessing the pointers from within the MEX functions and so users don't have to worry about it.

In one-sided communication, a region of memory is defined as a window which is remotely accessible by other processes. The origin process which issues a one-sided communication call just mentions the displacement of the target buffer from start of the window region and the count and datatype of elements. So, on the target process, it is important that our data of interest remains in the memory region declared as a window.

What happens if for some reason, our data on the target process to be communicated moves to a region outside the window? This is addressed in the following sub-section.

4.3 Limitations on use of windows in RMA

Let's say if we have pre-allocated an array on the target process. We declare a window using that buffer. Now, if we increase the size of the array after creating the window, the array will be moved to a totally new memory location most probably outside the window and so is no longer remotely accessible.

Similarly, there may be other cases when the buffer is assigned a new location in memory like if there is a brand new assignment to the buffer name altogether. We do not inspect all such cases here, but this is something which the users should be aware of.

Example: Consider the following code section. After creating the window, we change values in 'data' in two ways. First we use indices to change values of individual elements. This is perfectly **okay** as this does not change the memory location as far as we do not go out of bounds of the original array.

Next, we use the 'rand' function to allocate new values to 'data' after creating a window. Doing this will allocate new memory region to the buffer 'data'. Now, if we use `MPI_Get` from another process to get values from 'data', it will obtain the values assigned **before** this new allocation. This is because the new values are at a **different** memory region, **not** inside the window.

```
win = MPI_Win;

n=5;
data=zeros(n,1);
bytes=8*n;          % 8 bytes per element for 'double'

% Now create a window using above buffer, making it remotely accessible
err=MPI_Win_create(data, bytes, 8, MPI_INFO_NULL, MPI_COMM_WORLD, win)

for i=1:n
    data(i)=i;      % assigning values to elements
end

data=rand(n,1);     % re-allocation of array 'data'
```

4.4 Limitations on use of partial arrays for communication

Matlab allows you to work with parts of an array, say certain columns, rows or any other random parts. An obvious question for users would then be, can partial arrays be used for sending and receiving data while using HPCmatlab? The answer is Yes and No, respectively.

The parts of an array accessed using partial indices for assigning to another variable or passing to a function are copied to a different memory location. This is because it is completely possible that the part of array may not be contiguous in memory, e.g. a particular row of the array and so it is necessary to copy those values to make sure they are contiguous. This copying works great while sending the data as we can still access the pointer to the first element of partial array from the MEX function and the rest of the data to be sent is now arranged contiguously, irrespective of whether it originally was or not.

However, this same factor works against receiving the data in a partial array. If we use partial array to receive the data, the received data is stored at the memory location where the partial array was copied and not in the original buffer. Hence we cannot access the received part using the original buffer name. The user then has to use a separate temporary array for receiving the data and then assign the contents to the desired part of the buffer which is actually supposed to receive the data.

Example: Following code section and its output show how **sending** using partial arrays is **okay**, but **receiving** is **not**.

```
N=5;
tag=123;
% We will send data from rank '0' to rank '1'
if rank==0
    data=1:10;
    err=MPI_Send(data(6:10),N,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
    err=MPI_Send(data(6:10),N,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
end                    ✓

if rank==1
    % Initialize buffer (array) on rank 1 which will receive data
    recv_buff1=zeros(1,5);
    recv_buff2=zeros(1,10);
    % Now receive N 'double' type elements from rank '0'
    err=MPI_Recv(recv_buff1,N,MPI_DOUBLE,0,tag,MPI_COMM_WORLD,status);
    err=MPI_Recv(recv_buff2(6:10),N,MPI_DOUBLE,0,tag,MPI_COMM_WORLD,status);
                           ✗
    disp('After communication:');
    recv_buff1
    recv_buff2
end
```

Output:

```
After communication:

recv_buff1 =

     6     7     8     9    10      ✓


recv_buff2 =

     0     0     0     0     0     0     0     0     0     ✗
```

Appendix A: Syntax of HPCmatlab MPI functions

A.1 General MPI Functions

1) Initialize MPI

```
error = MPI_Init(0,0)
```

2) Get rank of the process

```
error = MPI_Comm_rank(MPI_COMM_WORLD, rank)
```

3) Get size of the communicator (total number of processes)

```
error = MPI_Comm_size(MPI_COMM_WORLD, size)
```

4) Abort all processes

```
error = MPI_Abort(MPI_COMM_WORLD, errorcode)
```

5) Terminate MPI environment

```
error = MPI_Finalize()
```

6) Make all processes wait for particular communication calls to return

```
error = MPI_Waitall(count, array_of_requests[],
            array_of_statuses[])
```

A.2 Point to Point Communications

7) Send data

```
error = MPI_Send(buf, count, datatype, dest, tag, MPI_COMM_WORLD)
```

8) Receive data

```
error = MPI_Recv(buf, count, datatype, source, tag,
            MPI_COMM_WORLD, status)
```

9) Non-blocking send

```
error = MPI_Isend(buf, count, datatype, dest, tag,
            MPI_COMM_WORLD, request)
```

10) Non-blocking receive

```
error = MPI_Irecv(buf, count, datatype, source, tag,
                  MPI_COMM_WORLD, request)
```

11) Buffered send

```
error = MPI_Bsend(buf, count, datatype, dest, tag,
            MPI_COMM_WORLD)
```

12) Combined send and receive operations

```
error = MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag,
            recvbuf, recvcount, recvtype, source, recvtag,
            MPI_COMM_WORLD, status)
```

A.3 Collective Communications

13) Gather data from all processes in rank order; all processes receive the result

```
error = MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf,
            recvcount, recvtype, MPI_COMM_WORLD)
```

14) Reduce data from all processes using an operation; all processes receive the result

```
error = MPI_Allreduce(sendbuf, recvbuf, count, datatype, op,
            MPI_COMM_WORLD)
```

15) Process j sends the k-th block of its local *sendbuf* to process k, which places the data in the j-th block of its local *recvbuf*

```
error = MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf,
            recvcount, recvtype, MPI_COMM_WORLD)
```

16) Block the caller until all processes enter the call

```
error = MPI_Barrier(MPI_COMM_WORLD)
```

17) Broadcast message from root process to all processes

```
error = MPI_Bcast(buffer, count, datatype, root, MPI_COMM_WORLD)
```

18) Gather data from all processes in rank order to the root process

```
error = MPI_Gather(sendbuf, sendcount, sendtype, recvbuf,
            recvcount, recvtype, root, MPI_COMM_WORLD)
```

19) Reduce data to the root process from all processes using an operation

```
error = MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root,
            MPI_COMM_WORLD)
```

20) Split message into parts and send i^{th} segment to the i^{th} process

```
error = MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf,
            recvcount, recvtype, root, MPI_COMM_WORLD)
```

A.4 One-Sided Communications (RMA)

21) Create window object which can be used for RMA

```
error = MPI_Win_create(base, size, disp_unit, info,
            MPI_COMM_WORLD, win)
```

22) Free the window object

```
error = MPI_Win_free(win)
```

23) Synchronize RMA calls

```
error = MPI_Win_fence(assert, win)
```

24) Complete all outstanding RMA operations initiated by the calling process to the target rank on the specified window

```
error = MPI_Win_flush(rank, win)
```

25) Start an RMA access epoch

```
error = MPI_Win_lock(lock_type, rank, assert, win)
```

26) Complete an RMA access epoch

```
error = MPI_Win_unlock(rank, win)
```

27) Transfer data from origin node to target node

```
error = MPI_Put(origin_addr, origin_count, origin_datatype,
            target_rank, target_disp, target_count,
            target_datatype, win)
```

28) Transfer data from target node to origin node

```
error = MPI_Get(origin_addr, origin_count, origin_datatype,
            target_rank, target_disp, target_count,
            target_datatype, win)
```

A.5 Details of different fields in the syntax

- `error`
  - error code returned by the function
- `rank, size`
  - passed as ‘double’ type variables
  - the function will set the values of these variables to the rank/size of processes
- `MPI_COMM_WORLD`
  - as we have not included the functionality to create new groups of processes, users will be using this default communicator for MPI processes
- `errorcode, count, sendcount, recvcount, origin_count, target_count, dest, source, target_rank, tag, sendtag, recvtag, root, size, disp_unit, target_disp`

- these can be all passed as 'double' type values which is the default datatype for numbers in Matlab; the functions will convert them to the required datatype
- counts are the number of data elements to be communicated
- tags are like identifiers for matching the send and receive calls
- `root` is the root process for the broadcast, scatter, reduce or gather operations
- `size` is the size of window in 'bytes'
- `disp_unit` is the local unit size for displacements
- `target_disp` is the displacement from start of the window to target buffer

➢ `request, array_of_requests[]`

- of the type `MPI_Request`, can be declared using an assignment operation

➢ `status, array_of_statuses[]`

- of the type `MPI_Status`, can be declared using an assignment operation

➢ `buf, sendbuf, recvbuf, origin_addr`

- data buffers from or to which data is to be sent or received

➢ `datatype, sendtype, recvtype, origin_datatype, target_datatype`

- MPI datatypes
- options available:
  `MPI_CHAR, MPI_SIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_BYTE, MPI_WCHAR, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_INT, MPI_UNSIGNED, MPI_LONG, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_LONG_LONG_INT, MPI_UNSIGNED_LONG_LONG, MPI_LONG_LONG`

➢ `op`

- MPI operations
- options available:
  `MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MINLOC, MPI_MAXLOC, MPI_REPLACE, MPI_NO_OP`

➢ `win`

- window object of the type `MPI_Win`, can be declared using an assignment operation i.e. `win = MPI_Win`

➢ `info`

- `MPI_Info` object

- options available:
  `MPI_INFO_NULL`

- `base`
  - data buffer to be declared as an RMA window
- `assert`
  - assertions on the context of the call, used to optimize performance
  - support for assertions has not been added, hence users may just pass `0`
- `lock_type`
  - state of the lock
  - options available:
    `MPI_LOCK_SHARED, MPI_LOCK_EXCLUSIVE`