These notes pick up from chapter three. Notes on the content just before this are in a physical notebook.

1 Distributed Memory Programming with MPI

Start with a hello world program, taking inspiration from the classic text by Kernighan and Ritchie.

```c
#include <stdio.h>
#include <string.h>
#include<mpi.h>

const int MAX_STRING = 100;

int main(void) {
        char greeting[MAX_STRING];
        int comm_sz;  // number of processes
        int my_rank;

        MPI_Init(NULL, NULL);
        MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

        if (my_rank != 0) {
                sprintf(greeting, "Greetings from process %d of %d!",\
                         my_rank, comm_sz);
                MPI_Send(
                        greeting, /* message buffer*/
                        strlen(greeting)+1,  /* message size */
                        MPI_CHAR,  /* message type */
                        0, 0,  /* destination (rank) and tag
                        MPI_COMM_WORLD  /* communicator */);
        } else {
                printf("Greetings from process %d of %d!", my_rank,\
                        comm_sz);
                for (int q = 1; q < comm_sz; q++) {
                        MPI_Recv(greeting, MAX_STRING, MPI_CHAR,\
                                q,\  /* source */
                                0,\  /* tag */
                                MPI_COMM_WORLD,\  /* communicator */
                                MPI_STATUS_IGNORE /* status_p */);
                        printf("%s\n", greeting);
                }
        }
}
```

I have already installed all the required libraries and binaries. For the remainder of this chapter, `mpicc` will be used for compiling assembling and linking everything. It's our GCC for programs that use `#include<mpi.h>`.

Compile using

```
$ mpicc -g -Wall -o mpi_hello mpi_hello.c
```

I checked `man mpicc` and it says nothing about optimization flags. mpicc is a wrapper for some C compilation pipeline, and most likely handles that for us. Note that we didn't have to use `-lmpi` or something like that to link `mpi.h`.

So, if the generated executable object file is executed simply like you would normally as in `./mpi_hello`, it'll just treat it as a single process task. Some systems provide alternative program startup methods e.g. `mpiexec`

```
$ mpiexec -n <number of processes> ./mpi_hello
```

So, if I try running with six processes, I get this on my terminal

```
Greetings from process 0 of 6!
Greetings fomr process 1 of 6!
Greetings fomr process 2 of 6!
Greetings fomr process 3 of 6!
Greetings fomr process 4 of 6!
Greetings fomr process 5 of 6!
```

BTW, my machine has 1 socket, 6 physical cores and two threads per core. So, if I load up bashtop, it shows me a total of twelve cores available and in use. But, when I try to `mpi_hello` for with more than six processes, I get the following error

```
There are not enough slots available in the system to satisfy the 7
slots that were requested by the application:

  ./mpi_hello

Either request fewer slots for your application, or make more slots
available for use.

A "slot" is the Open MPI term for an allocatable unit where we can
launch a process. The number of slots available are defined by the
environment in which Open MPI processes are run:

  1. Hostfile, via "slots=N" clauses (N defaults to number of
     processor cores if not provided)
  2. The --host command line parameter, via a ":N" suffix on the
     hostname (N defaults to 1 if not provided)
  3. Resource manager (e.g., SLURM, PBS/Torque, LSF, etc.)
  4. If none of a hostfile, the --host command line parameter, or an
    RM is present, Open MPI defaults to the number of processor cores

In all the above cases, if you want Open MPI to default to the number
```

```
of hardware threads instead of the number of processor cores, use the
--use-hwthread-cpus option.

Alternatively, you can use the --oversubscribe option to ignore the
number of available slots when deciding the number of processes to
launch.
```

The second to the last paragraph seems interesting. Let's try it out for 12 processes

```
Greetings from process 0 of 12!
Greetings from process 1 of 12!
Greetings from process 2 of 12!
Greetings from process 3 of 12!
Greetings from process 4 of 12!
Greetings from process 5 of 12!
Greetings from process 6 of 12!
Greetings from process 7 of 12!
Greetings from process 8 of 12!
Greetings from process 9 of 12!
Greetings from process 10 of 12!
Greetings from process 11 of 12!
```

Fuck yeah!

So, back to what we were doing. While we are using `mpiexec` to execute a specific number of instances of the program, it can also be used to tell the system which core should run each instance of the program.

Now of course we should take a look at what exactly the code is doing. Note that all the identifiers in `mpi.h` start with the prefix `MPI_` and I really hope that there are no exceptions to this. Macros and constants are all upper case, and other constructs are a mix of snake_case and CamelCase.

1.1 `MPI_Init` and `MPI_Finalize`

No other MPI function should be called before the initializing function, and no MPI function should be called after the finalizing function.

These functions don't necessarily have to be in `main`

The initializer does the necessary setup like storage allocation for message buffers, rank allocation etc.

The syntax is

```c
int MPI_Init(
        int* argc_p,   // in/out
        char *** argv_p);   //in/out
```

These are essentially pointers to `argc` and `argv`. We can pass `NULL` when our program doesn't use these. The `int` retval is error codes, i.e. in the same vein as main returning `int`. We won't really need those error codes. Hopefully.

`MPI_Finalize` is to `MPI_Initialize` what `free` is to `malloc`.

1.2 Communicators, MPI_Comm_size and MPI_Comm_rank

A **communicator** is a collection of processes that can send messages to each other. One of the purposes of MPI_Init is to define a communicator (MPI_COMM_WORLD) that consists of all the processes started when the program begins execution. Their signature is

```
int MPI_Comm_size(
        MPI_Comm comm,  /* in */
        int *comm_sz_p  /* out */);
int MPI_Comm_rank(
        MPI_Comm comm,  /* in */
        int *my_rank_p  /* out */);
)
```

Again, `int` retval because error codes, and the desired values, the number of processes and the calling process' rank, are stored in the variable pointed to by the second argument.

1.3 Communication

The first branch of the `if-else` block generates a message and sends it to process 0. The next branch is executed only for process 0 and receives and prints the messages received from processes `1..=p`.

1.3.1 MPI_Send

It can be seen that the signature is somewhat complex. The first three arguments are the message buffer, size and type in that order. The remaining arguments are desitnation, tag and communicator, in that order. `strlen+1` is used to make space for the terminating null character.

Now, why do we need a special datatypes for the MPI functions? Because standard CeeLanguage datatypes can't be passed to functions. So, I guess that they used structs or enums to implement these types. The type of these datatypes is MPI_Datatype, which perhaps would have been called MPI_Datatype_t if it were up to some other people to name it. In addition to the standard CeeLanguage datatypes, MPI also has an MPI_Packed.

Tag is just an argument used for distinguishing different types of messages from the same process. Kind of like identifiers for buffers in FORTRAN.

The last argument is the communicator. So, a message sent by a process via one communicator can't be received by a process using another communicator.

1.3.2 MPI_Recv

The first six arguments correspond to the six arguments of the sender. Instead of destination, there's a source. There's another parameter called `status_p`, which will be discussed later.

1.3.3 Message Matching

The message sent by rank q will be received by a process ranked p using the appropriate function calls only if

- `recv_comm = send_comm`
- `recv_tag = send_tag`
- `dest = r`
- `src = q`

Yup, makes sense to me. The first three parameters also play an important role, because conflicts over there would basically mean conflicting datatypes, buffer spaces etc.. If they don't match, then there is a risk of segfaulting or stack smashing. For now, one just has to make sure that the received and sent types are the same and the receiving buffer is larger than (or same size as) the sender's buffer.

The receiver function can also accept some wildcard arguments in the cases when it doesn't matter in which order or from what tags messages are received. So, the `src` could be `MPI_ANY_SOURCE`, with the `status_p` argument set to `MPI_STATUS_IGNORE`. Similarly there's also a `MPI_ANY_STATUS` etc.

The `status_p` argument

The only contraints for receiving messages are the four specified above. Messages can still be received if the first three arguments differ between corresponding send and receive.

The last argument `MPI_Recv` has type `MPI_Status*`, where `MPI_Status` is a struct with at least the three members `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`. So, if a status argument is passed to the receiver function, then we can determine the sender and tag using

```
status.MPI_SOURCE
status.MPI_TAG
```

This leaves the data size, which can be retrieved with a call to `MPI_Get_count`. The call goes like

```
MPI_Get_count(&status, recv_type, &count),
```

where `recv_type` is the type of the receive buffer in the `MPI_Recv` call. `count` then identifies the message length.

So, basically they're just calculating the buffer length in terms of the number of data entries of the given datatype. Nothing fancy going on here.

Example - Trapezoidal Rule

Remember the Trapezoidal rule? A quick and dirty way of numerically integrating nice functions. We are going to write the sequential code before moving on to the MPI implementation.

The trapezoidal rule

We want to numerically evaluate

$$\int_a^b f(x)\mathrm{d}x \tag{1}$$

by partitioning the interval $[a, b]$ uniformly into n subintervals. Let $\{x_i\}\,\forall i \in n$ do the partitioning. (*Note: if $n \in \mathbb{N}$, then $i \in n$ means $i \in \{0, 1, 2, \ldots n\}$, a notation that I have seen in some mathematics literature*). This also partitions the area under the curve and you know what the trapezoids in the trapezoid rule are.

Area of one trapezoid is

$$\frac{h}{2}\left[f(x_i) + f(x_{i+1})\right]] \tag{2}$$

where $h = \frac{b-a}{n}$, so that

$$\begin{aligned} x_0 &= a \\ x_1 &= a + h \\ x_2 &= a + 2h \\ &\vdots \\ x_n &= a + nh = b \end{aligned} \tag{3}$$

So, the approximate area under the curve is given by

$$h\left[\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \ldots + f(x_{n-1}) + \frac{f(x_n)}{2}\right] \tag{4}$$

So, a sequential code would look something like this

```
/* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i < n; i++) {
        x_i = a + i*h;
        approx += f(x_i);
}
appox = h*approx;

```

EZ.

Parellelizing the trapezoidal rule

Recall the four basic steps

1. Partition the solution into tasks

2. Identify the communication channels between the tasks

3. Aggregate the tasks into composite tasks

4. Map the composite tasks to cores

Say we have some `comm_sz` cores at our disposal. If `comm_sz` evenly divides n, then great, else we will have to stick to floor division.
Here is what the parallel pseudo code looks like

```
h = (b - a)/n;
local_n = n/comm_sz;
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
local_integral = Trap(local_a, local_b, local_n, h);
if (my_rank != 0) {
        send(local_intergal, process0);
}
else {
        for (proc = 1; proc < comm_sz; proc++) {
                receive(local_intragral, proc);
                total_integral += local_integral;
        }
}

if (my_rank == 0)
        print(result);
```

Which translates to the following `main` in the MPI program with hardcoded values instead of inputs

```c
int main() {
        int my_rank, comm_sz = 1024, local_n;
        double a = 0.0, b = 3.0, h, local_a, local_b;
        double local_int, total_int;
        int source;

        MPI_Init(NULL, NULL);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

        h = (b-a)/n;
        local_n = n/comm_sz;
        local_a = h;
        local_b = local_a + local_n*h;
        local_int = Trap(local_a, local_b, local_n, h);

        if (my_rank != 0 ) {
                MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        } else {
                total_int = local_int;
                for (source = 1; source < comm_sz; source++) {
                        MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0, PMI_CC
                total_int += local_int;
                }
        }

        if (my_rank == 0 ) {
                printf("With n = %d trapezoids, our estimate\n", n);
                printf("of the integral from %f to %f = %.15e\n", a, b, total
        }
        return 0;
}

double Trap {...} {...};
```

Semantics of `MPI_Send` and `MPI_Recv`

Once a sending process assembles a message, with the message and the metadata (e.g. sending process rank, destination rank, datatype and all that), it can either **buffer** the message, or **block**.

- **buffer**: MPI places the assembled message into some internal storage and the sending process can move on to the next instruction, i.e. the call to `MPI_Send` returns.
- **block**: process waits until it can begin transmitting the message, i.e. till a matching receiver is found.

When a call to `MPI_Send` returns, it cannot be said whether the message has been transmitted or not, it simply means that the send buffer is now available for use again. There are separate functions for knowing whether the message has been transmitted or not, or if we want to explicitly buffer et cetera.

Quite sensibly, `MPI_Recv` always blocks. MPI messages are non-overtaking, i.e. if q and r are two processes and q sends two messages to r in succession, then r must receive them in the same order in which they were sent, but if messages from different processes can be received by a single receiving process in any order.

The system can **hang** if there is no matching receive to a **blocking** send. This could happen if there are errors in values of tags, process ranks et cetera. Alternatively, if a **buffering** send does not have a matching receive, then the message is lost forever. RIP. A mismatched receive could also end up receiving from the wrong process, which might cause all sorts of havoc.

Dealing with I/O

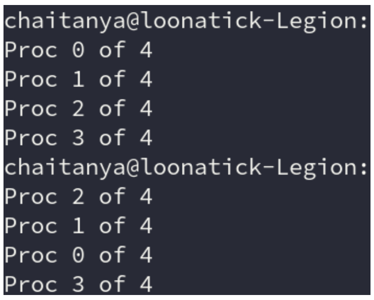
The MPI standard does not define which processes have access to I/O devices, so it is a good practice to pick a good convention and follow it. E.g. process 0 can write to stdout and we try to use nothing else to print to stdout.

Pretty much all the processes in MPI_COMM_WORLD have full access to stdout and stderr. But, you cannot expect an MPI implementation to also do the access scheduling to these devices for you; the OS will handle that and the output for runs with identical inputs will be 'nondeterministic'.

As an example, consider the simple program

```c
#include <stdio.h>
#include <mpi.h>

int main(void) {
        int my_rank, comm_sz;

        MPI_Init(NULL, NULL);
        MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

        printf("Proc %d of %d > Does anyone have a toothpick?\n",
                my_rank, comm_sz);
        MPI_Finalize();

        return 0;
}
```

Just another hello world program, right? Well, see the screenshot in Figure 1.



```
chaitanya@loonatick-Legion:
Proc 0 of 4
Proc 1 of 4
Proc 2 of 4
Proc 3 of 4
chaitanya@loonatick-Legion:
Proc 2 of 4
Proc 1 of 4
Proc 0 of 4
Proc 3 of 4
```

Figure 1: Allowing all processes to use stdout

Most MPI implementations allow only process 0 to access stdin. So, values from stdin will have to be shared with other prcesses through message passing from process 0.

So, if it is statically known that all processes must receive the input, one could write a function that does the input receiving from stdin, the MPI_Send to other processes and `MPI_Recv` from process 0. Writing the funcion down for practice

```c
void Get_input(
        int my_rank, int comm_sz,
        double *a_p, double *b_p,
        int *n_p) {
                int dest;

                if (my_rank == 0) {
                printf("Enter a, b, and n\n");
                scanf("%lf %lf %d", a_p, b_p, n_p);
                for (dest = 1; dest < comm_sz; dest++) {
                        MPI_Send(a_p, 1, MPI_DOUBLE,
                                0, 0, MPI_COMM_WORLD);
                        MPI_Send(b_p, 1, MPI_DOUBLE,
                                0, 0, MPI_COMM_WORLD);
                        MPI_Send(n_p, 1, MPI_INT,
                                0, 0, MPI_COMM_WORLD);
                        }
                } else {
                        MPI_Recv(a_p, 1, MPI_DOUBLE,
                                0, 0, MPI_COMM_WORLD);
                        MPI_Recv(b_p, 1, MPI_DOUBLE,
                                0, 0, MPI_COMM_WORLD);
                        MPI_Recv(n_p, 1, MPI_INT,
                                0, 0, MPI_COMM_WORLD);
                }
        }
```

Yeah, this is a funcion with a lot of side effects, but I do not have enought development experience to know how good or bad this practice is and whether there is a recommended alternative.

Collective communicaion

The parallelization proposed is for the trapezoidal rule is not the most efficient one that we can think of; having process 0 do all the sequential work and the rest quit is not optimum. Remember Amdahl's law? For a sequential program of which a proportion p can be parallelized, the maximum possible speedup (i.e. assuming all the good stuff like load balancing, low latency, low communication overheads et cetera) is

$$S(N) = \frac{1}{\frac{p}{N} + 1 - p} \tag{5}$$

where N is the number of nodes/cores at our disposal. So, for $n > N$ intervals, the length of each parallel branch is n/N, while the final sequential branch has length N, giving a total length of $n/N + N$. (Here the loosely used term 'length' is a measure of running time)

The same program when executed sequentialy has length n. The speedup is then loosely speaking

$$S(N) = \frac{n}{\frac{n}{N} + N} \tag{6}$$

It is possible to improve the program by further parallelizing the sequential part. Better yet, we could come up with a different algorithm altogether, because naïve parallelization can make the program slower instead.

Tree structured communication

Remember fast adders from EE 224? This kind of reminds me of those. I'm sure that this will be unfounded if I sit down to revise Professor Viru's notes LÖL. Say, we have 8 processes, 0 through 7. Each of them has computed a local sum of its own.

The even numbered processes then receive the local sum of the next odd numbered process and add it to their local sum to create a new local sum. The number of partial sums is halved. Repeat for the remaining partial sums to get the complete sum.

Figure 2 shows this

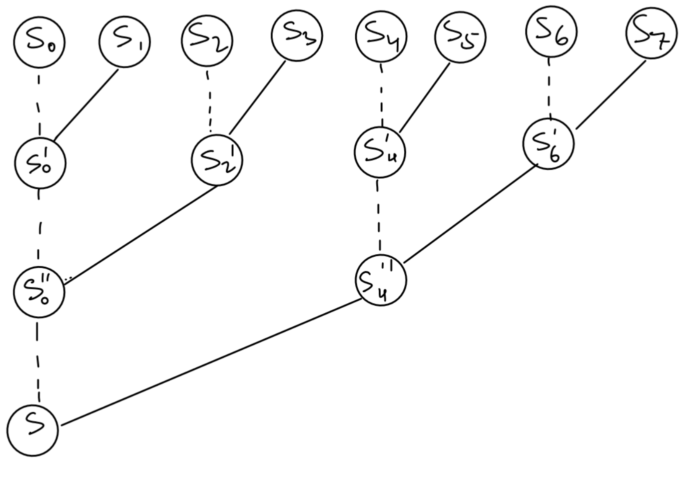


Figure 2: A better parallel algorithm for getting completet sum from partial sums

So, instead of a single set of parallel branches, we have multiple parallel branches. The length of the first set is still n/N, while the sequential part goes down from n to $\log n$. This is a significant speedup of the sequential section for larger values of n.

Implementing such tree structured stuff is not as easy as the naïve method of course. Multiple topologies are possible. E.g. Figure 3

Smells like lots of boilerplate incoming. Unless. . .

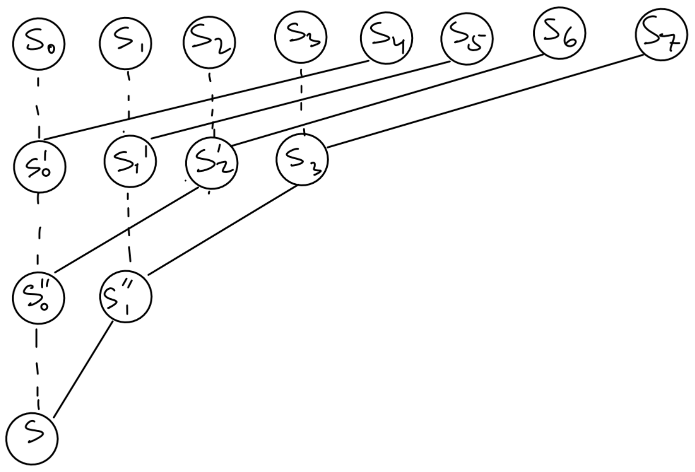


Figure 3: Alternative tree topology that gives the same runtime

`MPI_Reduce`

MPI standard includes implementations of global sums. GG. The application programmer hopes that the developer of the MPI implementation knows what they are doing.

Such global functions that involve the communication of more than two processes (i.e. unlike `MPI_Send`, `MPI_Recv` pairings) are called **collective communications** (as opposed to **point-to-point** communication).

The function `MPI_Reduce` takes in addition to the usual parameters, an operator parameter. The previous code would be equivalent to the function call

```
double local_x[N], sum[N];
/*
...
*/
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, /* operator */
                0, MPI_COMM_WORLD);
```

Some important points about collective and one-to-one communcations

- All the processes in the communicator must call the same collective function; a program cannot have a call to `MPI_Reduce` in one process and an equivalent call to `MPI_Recv` on another
- placeholder

For more information check out `man MPI_Reduce`. There are some other topologies as well, e.g. butterfly et cetera. Not documenting everything here. Check out the man pages `SEE ALSO` sections to get to know them.

Derived Datatypes in MPI

Sending a series of builtin datatypes using a series of Send-Receives is a lot more expensive than sending all the data in a single send-receive. This can be done by defining your own datatype that is optimized for use in MPI. It is not as straightforward as just packaging everything in a struct.

We use `MPI_Type_create_struct` to create such derived datatypes.

```c
int MPI_Type_create_struct(
        int count   /* in */,
        int array_of_block_lengths[]   /* in */,
        MPI_Aint array_of_displacements[]   /* in */,
        MPI_Datatype array_of_types[]   /* in */,
        MPI_Datatype* new_type_p   /* out */)
```

- `count`: Number of elements in datatype
- `array_of_block_lengths`: in case individual elements are themselves arrays, pass `1, 1, 1, ...` if all scalars.
- `array_of_displacements`: displacement of each type from beginning of message
- `array_of_types`: types of individual elements

The second argument essentially means that arrays in the datatype must have fixed lengths. To find what to use for the third argument, `MPI` provides `MPI_Get_address`.

```c
int MPI_Get_address(
        void* location_p   /* in */,
        MPI_Aint* address_p   /* out */);
```

So, if we want a derived datatype made of some three datatypes, the i^{th} element of the displacements array is the difference between the output of `MPI_Get_address` for the i^{th} data and the output for the first data.

Before the new datatype is ready for use, you first have to **commit** it using `MPI_Type_commit`. This allows for optimization of internal representation of the derived datatype for use in message passing. It can then be sent and received like any other MPI datatype.

Parallel Sorting Algorithm

First things first, what is the input? What is the output? The answer depends on where the keys are stored. We can start or finish with the keys distributed in different processes, or we can collect them in the same process.

If we have a total of n keys and $p =$`comm_sz` processes, our algo will start and finish with n/p keys assigned to each process. When the algorithm terminates,

- the keys assigned to each process will be sorted
- if $0 \leq q < r < p$, then each key assigned to process q should be less than equal to each key assigned to process r

Consider bubble sort. It is as sequential as it gets and it doesn't make sense parallelizing it. But, there is a variant of the bubble sort called the *odd-even transposition sort*, the key idea being decoupling of the compare-swap.

- Even phase: compare-swap at indices, $2j$, $2j + 1$
- Odd phase: compare-swap at indices $2j + 1$, $2j + 2$

It can be shown that the array will be sorted in at most n phases. An even phase followed by an odd phase constitutes one pass. So, So, each pass has a sequence of two parallel stages in series. For a parallelized version with p processes, the array will be sorted in p phases (assuming p divides n).

In general, during even phases, odd-reanked partners exchange with `my_rank-1` and even-ranked partners exchange with `my_rank+1`. There are going to be edge cases for ranks 0 and `comm_sz-1`.

Safety

The MPI standard allows `MPI_Send` to behave in two ways - blocking and buffering. Many systems set a threshold at which the system switches from buffering to blocking; small messages are buffered while larger ones are blocking. So, we can easily have hangs and deadlocks if we are not careful. In conclusion, `MPI_Send` is *unsafe* because it relies on MPI provided buffering Thankfully, MPI provides safer alternatives in the form of synchronous send - `MPI_Ssend`.

```c
void Merge_low(
                int my_keys[],  /* in/out */
                int recv_keys[],  /* in */
                int temp_keys[],  /* scratch */
                int local_n  /* = n/p, in */) {
        int m_i, r_i, t_i;
        m_i = r_i = t_i = 0;
        while (t_i < local_n) {
                if (my_keys[m_i] <= recv_keys[r_i]) {
                        temp_keys[t_i] = my_keys[m_i];
                        t_i++; m_i++;
                } else {
                        temp_keys[t_i] = recv_keys[r_i];
                        t_i++; r_i++;
                }
        }

        for (m_i = 0; m_i < local_n; m_i++) {
                my_keys[m_i] = temp_keys[m_i];
        }
}
```