INTRODUCTION TO PARALLEL COMPUTING

ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
UNIVERSITY OF WEST ATTICA

DEPARTMENT OF INFORMATION AND COMPUTER ENGINEERING

INTRO IN MPI WITHOUT COLLECTIVE COMMUNICATION

STUDENT DETAILS

NAME: ATHANASIOU VASILEIOS EVANGELOS
STUDENT ID: 19390005
STUDENT **SEMESTER:** 7th
STUDENT STATUS : UNDERGRADUATE
STUDY PROGRAM : UNIWA
LABORATORY DEPARTMENT : E3 MONDAY 14:00 – 16:00
LABORATORY TEACHER : MICHAEL IORDANAKIS
DELIVERY DATE : 11/12/2022

STUDENT PHOTO:

The point

The exercise aims to achieve " point - to - point " communication between " p " processes having as its occasion the classification control of a sequence " Seq " of size " N ". More specifically, the elements of the sequence must be equally distributed among the processors, having first ensured that the size of the sequence will be an integer multiple of the number of processors, and then checking in parallel whether their elements are sorted in ascending order. If it is not, then return to process P 0 which will be the main one, the first element for which the sorting of the sequence breaks down. Finally, P 0 will print the string " yes " in the case that the sequence is sorted in ascending order or the string " no " in the case that the sequence is not sorted in ascending order and will additionally print the first element that breaks the classification. This process will run iteratively with a menu of options to continue or exit the process. The choice will be given by the user.

The problem and the implementation in natural language

The process " P 0" takes on the duties of "manager", that is, it will distribute the elements equally to the other processes, so that it also gets the same number of elements. In the first phase the processes will check if their elements are sorted in ascending order. For example, we have 4 processors (P 0, P 1, P 2, P 3) and the sequence "1, 2, 3, 4, 5, 7, 6 ,8" of size 8. To check the sorting each processor it will take 8 / 4 = 2 elements. As shown in Figure 1, " P 0" will take elements 1, 2, " P 1" 3, 4, " P 2" 5, 7 and " P 3" 6, 8.

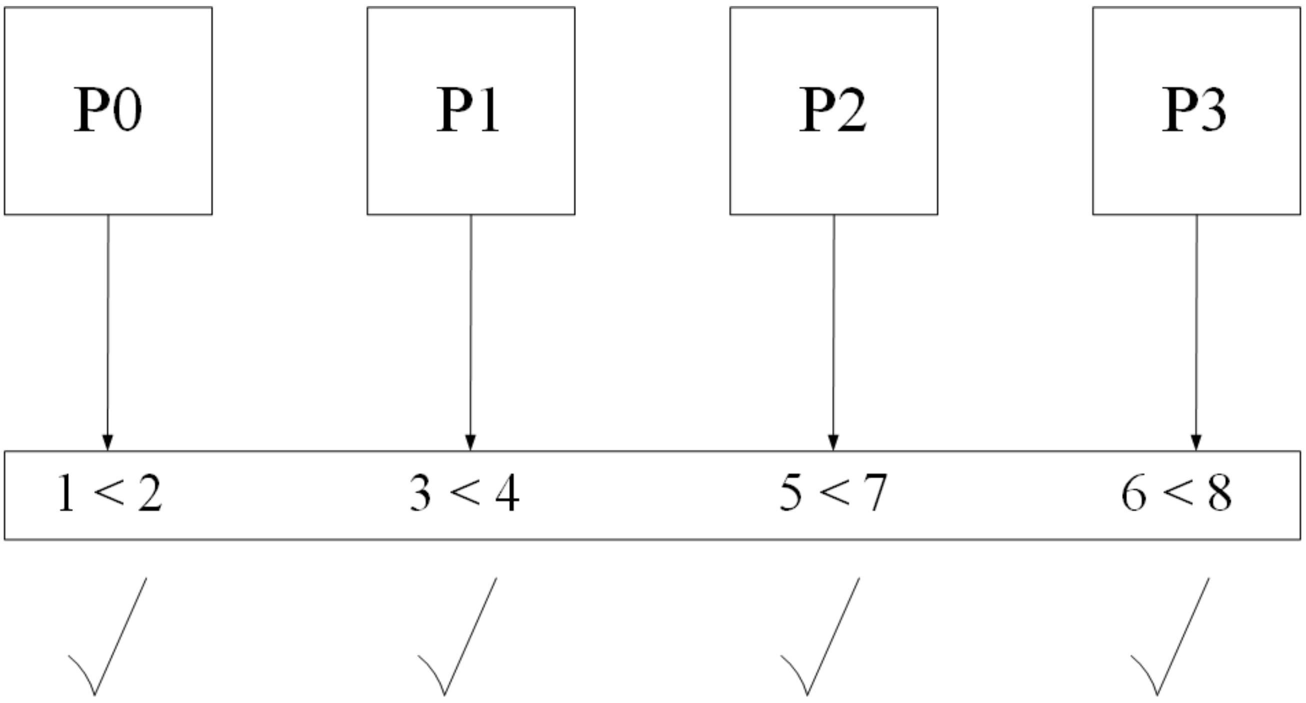

Figure 1. Checking the details of each processor

The targets of all four processes are sorted in ascending order individually . However, this does not lead to the conclusion that the entire sequence is ordered, as there are elements that have not been checked against each other, creating this problem in Figure 2.

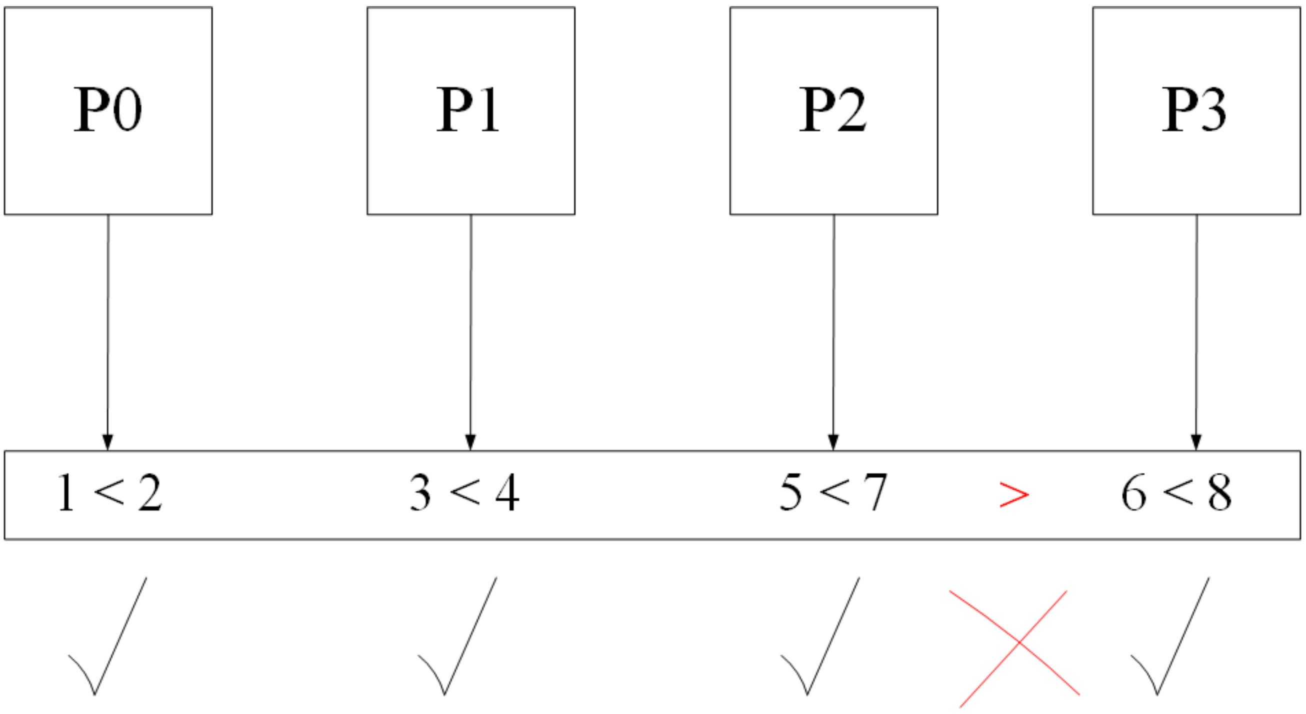

Figure 2. The problem with the absence of control of intermediate elements

Although the processes did not find an element that spoils the sorting of its elements, the sequence is not sorted in ascending order. This is because process communication is based only between " P 0" and the rest as shown in Figure 3.

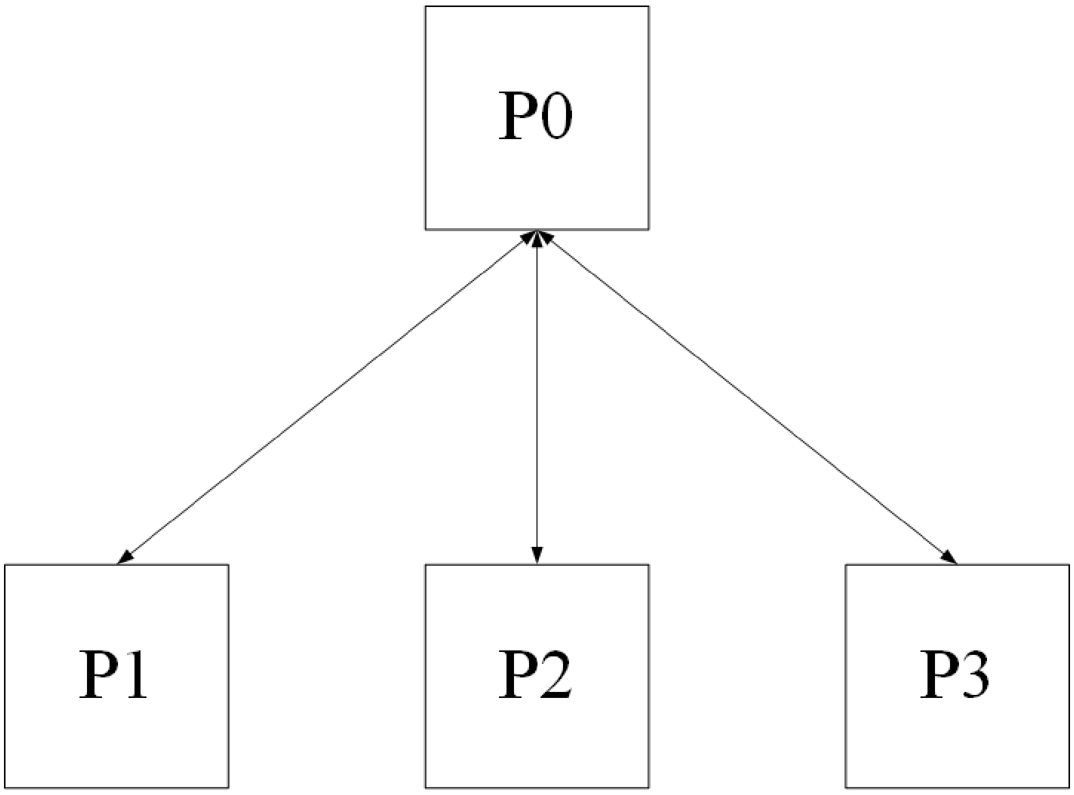

Figure 3. The problem with the absence of control of intermediate elements with a diagram

Therefore, it is not only a question of " P 0" communicating with the other processes, but also that the processes themselves communicate with each other to inform about their own element (the latter). Therefore, the correct " point - to - point " communication to solve the intermediate elements problem is that of Figure 4.

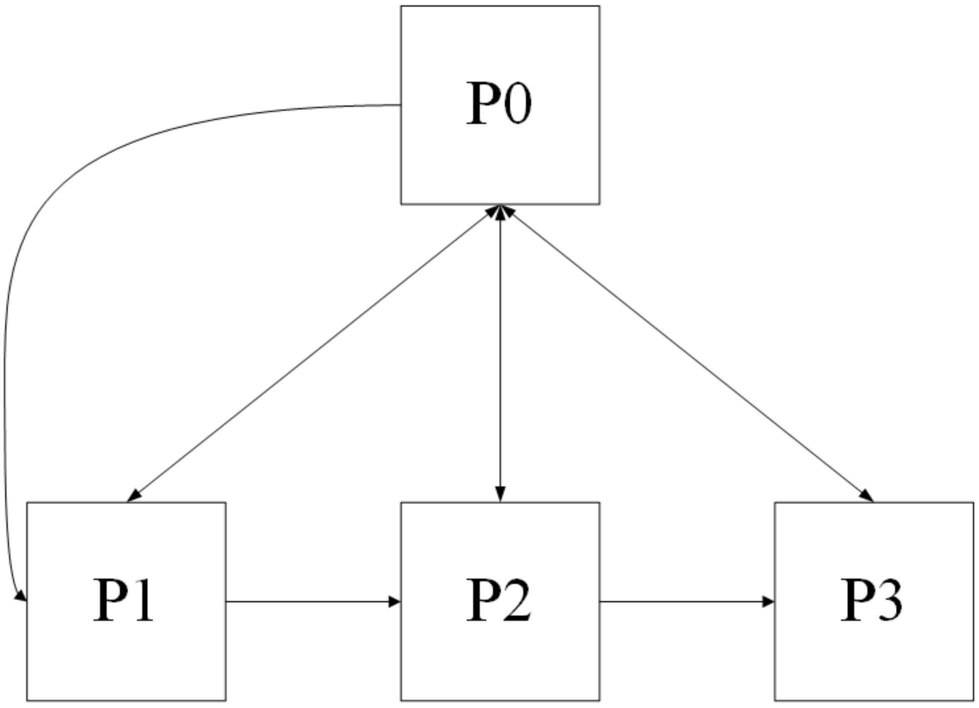

Figure 4. The solution to the problem with the absence of control of intermediate elements with a diagram

Now, it is now correctly concluded that the sequence is not sorted (Figure 5) and even the first element that spoils the sorting is found (Figure 6).

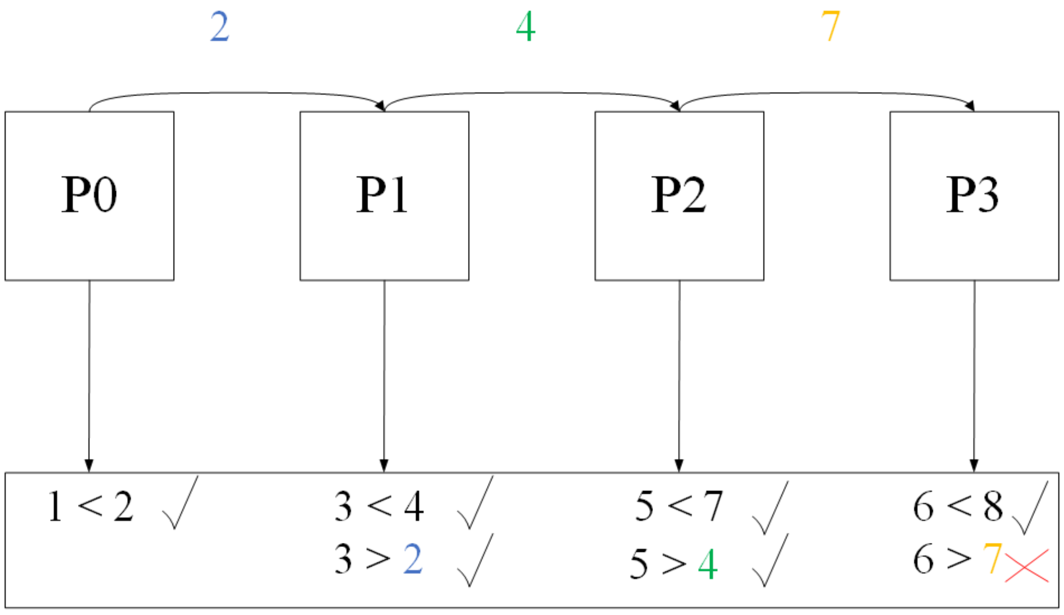

Figure 5. The solution to the problem with the absence of control of intermediate elements

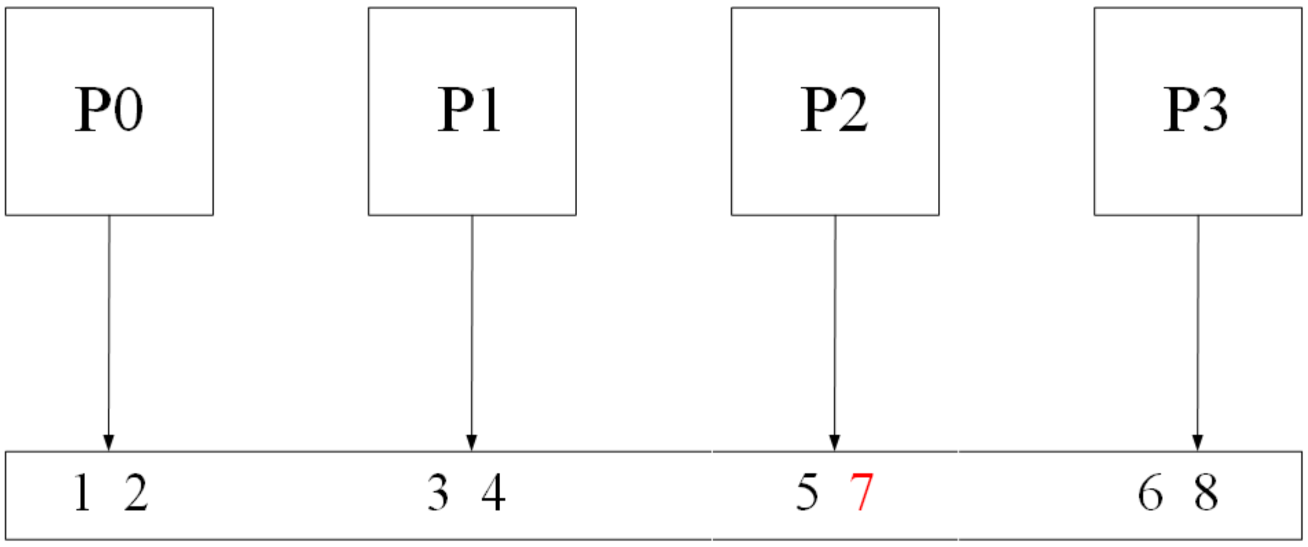

Figure 6. The first element that breaks the classification

The computational load of each process

The table below shows the computing load that each process will have.

P0	P1…P(p-1)	Shared computing load
1) Reads "N" (sequence size). 2) Reads " Seq " (the elements of the sequence one by one). 3) Sends "N" to other processes. 4) It calculates how many " n " elements each process should get, so that the data is evenly distributed. 5) It distributes the elements equally to the remaining processes starting from position " n " of the sequence, so that P 0 also gets its own elements which will be the first " n " of the sequence. 6) She gets her own data. 7) It sends its last element to the next process (P1). 8) It checks from its own elements if it found an element that breaks the sort and stores it, since it is also the first of the entire sequence. 9) Receives messages from other processes to sort their items. 10) Once it receives a message that there is an item that breaks the	1) They receive from P0 the " N ". 2) They calculate based on the " N " they just received how many elements of the sequence they expect to receive from " P 0 " (n). 3) They receive from " P 0" their " n " elements. 4) "P1" receives from " P0 " its last element to check with its own elements. The same routine is repeated with the other processes (eg " P 1" sends " P 2" its own last element, etc.) <u>with the exception of the last process " P (p -1)" which receives the last element of " P (p -2)", but doesn't send hers somewhere.</u> 5) They check if the last element received from the immediately preceding process is	1) They check if their items are sorted in ascending order . 2) If they find an element that breaks the classification, they save it.

sort, it retrieves the item itself from the process that found it and stores it. 11) Prints with a message the sorting status of the entire sequence, as well as the first element that breaks the sorting, if any. 12) Reads the user's selection from the options menu. 13) Sends the selection to other processes as well	greater than their first element 6) They send " P 0 " a message to sort their items. 7) In the event that they find an element that spoils the classification, they also send it to " P 0". 8) They receive from " P 0" the user's choice from the options menu.	

The problem and implementation in C language

To implement the request, the MPI environment that achieves parallel calculation and the C language was used . The MPI routines used are MPI_Init , MPI_Comm_rank , MPI_Comm_size , MPI_Send , MPI_Recv and MPI_Finalize . Communication between processes is point - to - point and asynchronous. The program is broken down into these major chapters:

Declaration of variables (Lines 13-29)

The variables are described in detail in the comments of “ Check _ Sort _ Seq . c »

Start the MPI environment (Lines 31-33)

The process is described in detail in the comments of “ Check _ Sort _ Seq . c »

Endless loop (Lines 35-184)

The infinite loop was used because the process will repeat itself until the user selects the exit option from the options menu. Point - to - point communication is achieved using the MPI_Send and MPI_Recv routines . These routines are blocking, which means that the process that calls one of them is blocked until the corresponding routine is called by the other process with which there is a desire to communicate. The point that might seem complicated and needs some documentation is how to store the first element that breaks the sort, if it exists. Initially, regardless of finding a point that breaks the classification, the process is not interrupted, since the goal is to find the first one. All processes will check their elements and the last element received from the immediately preceding process and if they find an element that breaks the sorting of their elements they will send it to " P 0". The variable " sort_breaker_exist " is the indicator variable that breaks the sort and also helps on line 88 by calling " Check_Sort_Breaker " (with the help of a pointer pointing to this variable) to save the first element from the processes other than " P 0" which breaks the ordering of the elements owned by one process. In lines 101-105 it is used by processes P 1… P (p -1) to compare the first element of one process with the last element of the immediately preceding one, so that intermediate elements are also checked. In lines 125-131, it is used by " P 0" for its own elements that it had concluded, since if an element was found that breaks the classification, it is also the first of the entire sequence. The “ first _ sort _ breaker _ stored ” from its name is the variable indicating that the first element that breaks the sequence sort has been stored. So pointer-variables are used so that I can hold the first element that breaks the sorting, since the processes, if found, will correctly send their own sequence-breaking elements to " P 0 ". The whole process is described in detail in the comments of “ Check _ Sort _ Seq . c ».

Menu (Lines 159-183)

The options menu contains two options. 1) Continue the process, 2) Exit the program. Any other option entered by the user is taken as invalid and prompted to enter a valid one. The reason that " P 0" sends the user's choice to the rest of the processes is so that they understand that if it is the exit choice they should also exit with a " break " command from the endless loop (lines 181-182) , as " P 0" is output in lines 173-174. So all processes will call the MPI _ Finalize routine on line 186 and the program will terminate smoothly. The purpose was to avoid the easy solution of " exit (1)" which would not allow the calling of the MPI _ Finalize routine and the smooth termination of the program, since without the user's choice distributed to all processes but only to " P 0 ” would not allow the other processes a way to escape the infinite loop. The whole process is described in detail in the comments of “ Check _ Sort _ Seq . c ».

Check _ MPI _ Routines (Lines 192-199)

The function takes as arguments:
int return _ value : The value returned by the MPI routine that calls it
char * mpi _ routine : T the name of the MPI routine that calls it

The whole procedure is described in detail in the comments of " Check _ Sort _ Seq . c ».

Check _ Memory (Lines 201-208)

The function takes as arguments:
int * seq : The dynamically bound array that stores the sequence of the process that calls it.

The whole process is described in detail in the comments of " Check _ Sort _ Seq . c ».

Check_Sort_Break (Lines 211-228)

The function takes as arguments :
int * seq : T the part of the sequence owned by the calling process
int n : T the number of elements owned by the process that calls it
int *ptr_sort_breaker: Pointer pointing to the variable " sort _ breaker "
int *ptr_sort_breaker_exist: Pointer pointing to the variable " sot _ breaker _ exist "

The whole process is described in detail in the comments of " Check _ Sort _ Seq . " c ».

Difficulties

My difficulty was in the intermediate elements problem, as I had thought of several ways that violated the specification, so I rejected them. To begin with, I was thinking in the 2nd phase to remove " P 0" and the first element of the sequence from the process and to distribute the N – 1 data equally to p – 1 processors (the 1st phase the N data equally distributed to p processors), so that to check intermediate elements and somehow "rest" " P 0 " which is loaded with other jobs. I didn't know to what extent this first thought of mine violated the specifications, so to be sure I came up with the methodology of one process sending the other its last element and comparing it with the first one they already have from the equalization. The result is correct and from the chapter "The computational load of each process" it is observed that the computational load is also approximately equally distributed.

Indicative Runs

Compile : mpicc -o Check_Sort_Seq Check_Sort_Seq.c
mpicc -o mpi mpi.c

Example 1 (mpirun -np 4 ./ Check_Sort_Seq) mpirun -np 4 ./mpi

```
Number of processors are 4

Size of integers' sequence must be integer multiple of number of processors (N mod processors ==
0).

Input the size of integers' sequence : 8

Input the integers' sequence

Seq[ 0 ] : 1

Seq[ 1 ] : 2

Seq[ 2] : 3

Seq[ 3] : 4

Seq[ 4] : 5

Seq[ 5] : 7

Seq[ 6] : 6

Seq[ 7] : 8



------------------- P0 ----------------------------
```

```
Is sequence sorted? no

Which is the 1st sort breaker? 7


[1] Continue...

[2] Exit...

Input a choice:
```

Example 2 (mpirun -np 4 ./ Check_Sort_Seq) mpirun -np 4 ./mpi

```
Number of processors are 4

Size of integers' sequence must be integer multiple of number of processors (N mod processors ==
0).

Input the size of integers' sequence : 24

Input the integers' sequence

Seq[ 0 ] : 1

Seq[ 1 ] : 2

Seq[ 2] : 3

Seq[ 3] : 4

Seq[ 4] : 5

Seq[ 5] : 6

Seq[ 6] : 7

Seq[ 7] : 12

Seq[ 8] : 34

Seq[ 9] : 56

Seq[ 10] : 55

Seq[ 11] : 23

Seq[ 12] : 111

Seq[ 13] : -9

Seq[ 14 ] : 0

Seq[ 15] : 123

Seq[ 16] : 45
```

```
Seq[ 17] : 56

Seq[ 18] : 7

Seq[ 19] : 8

Seq[ 20] : 1

Seq[ 21] : 3

Seq[ 22] : 45

Seq[ 23] : 24



------------------- P0 -----------------------------


Is sequence sorted? no

Which is the 1st sort breaker? 56


[1] Continue...

[2] Exit...

Input a choice:
```

Example 3 (mpirun -np 25 ./ Check_Sort_Seq) mpirun -np 25 ./mpi

```
Number of processors are 25

Size of integers' sequence must be integer multiple of number of processors (N mod processors ==
0).

Input the size of integers' sequence : 50

Input the integers' sequence

Seq[ 0 ] : 1

Seq[ 1 ] : 2

Seq[ 2] : 3

Seq[ 3] : 4

Seq[ 4] : 5

Seq[ 5] : 6
```

```
Seq[ 6] : 7

Seq[ 7] : 8

Seq[ 8] : 9

Seq[ 9] : 10

Seq[ 10] : 11

Seq[ 11] : 12

Seq[ 12] : 13

Seq[ 13] : 14

Seq[ 14] : 15

Seq[ 15] : 16

Seq[ 16] : 17

Seq[ 17] : 18

Seq[ 18] : 29

Seq[ 19] : 45

Seq[ 20] : 21

Seq[ 21] : 32

Seq[ 22] : 31

Seq[ 23] : 44

Seq[ 24] : 2

Seq[ 25 ] : 0

Seq[ 26] : 13

Seq[ 27] : 46

Seq[ 28] : 7

Seq[ 29] : 4

Seq[ 30] : 35

Seq[ 31] : 67

Seq[ 32] : 8

Seq[ 33] : 4

Seq[ 34] : 68

Seq[ 35] : 4

Seq[ 36] : 8
```

```
Seq[ 37] : 4

Seq[ 38 ] : 0

Seq[ 39] : 2

Seq[ 40] : -3

Seq[ 41 ] : -56

Seq[ 42] : 3

Seq[ 43] : 45

Seq[ 44] : -7

Seq[ 45] : 100

Seq[ 46] : 99

Seq[ 47] : 1

Seq[ 48] : 2

Seq[ 49] : 3



------------------- P0 -----------------------------


Is sequence sorted? no

Which is the 1st sort breaker? 45


[1] Continue...

[2] Exit...
Input a choice :
```

bpy.context.scene.objects.active = modifier_ob
print("Selected" + str(modifier_ob)) # modifier

Thank you for your attention.