Technical University of Denmark

F19
02346
DTU DIPLOM
DISTRIBUTED AND PARALLEL SYSTEMS

MANDATORY ASSIGNMENT 2

GRUPPE 17

17. MARTS 2019

JOACHIM S. MORTENSEN
S175179

MIKAEL KRISTENSEN
S094766

NICOLAI KAMMERSGÅRD
S143780

Problem 1 (group): Basic MPI Communication

Handed-in C File: ***communication.c***

Report on your work with Databar Exercises 3.4 and 3.5

The program partners up 2 or more (Must not be an odd amount) of processes, so that the partner with the lowest id sends a message containing its id to the partner first and then receives the partners id afterwards.

The program is expected to somewhat output the different prints in a random order, but we insert barriers in the hopes of receiving first a message from the Master regarding how many processes are used. After this a 'Hello' from each of the processes will be outputtet in a random order. Lastly the output is expected to be sorted in regards to the taskid, with the master as the first one.

When we use more than one node and inspect the output, we see the message from the master first, with the randomized hellos afterwards, so as expected. However the last part which should have been a number of messages sorted with the master first, is not the sorted as expected. the output is incrementing in regard to the taskid overall, but its not in the order we expected, its kindoff randomly incrementing over the taskid.

When we use just one node but multiple processes, the messages outputs from each process one at the time, so first the master with the number of tasks, then the hello from master and lastly the message at the end. after this the hello from another process and the last message i outputtet.

It seems the output have a buffer between the print command and the actual writing to output, since it is gone through sequentially for one node and randomized for multiple nodes.

Our messages (the last output per program) should be outputtet fairly ordered since each process except the master is in a receiving state waiting for another proces to send the "go ahead"message, making them output the message. So the turn signal is actually working, but the intermediate buffer between the program and the output obscures our order.

Problem 2 (group): Basic Collective Communication

Handed-in C File: ***routempi.c***

The functionality of the program is to calculate the length of a route given by a .csv file containing geographic coordinates of latitude longitude and elevation. The given implementation is sequential, our job is to rewrite it to do its calculations in parallel via MPI.

The route is divided up into sections based on how many tasks have been started, with a minimum size of 1. A slight overlap is necessary to get the correct route length, meaning the end point of section 1 is the start point of section 2.
The local length is determined using the given function which is based on the Haversine formula.

The total route length is calculated by adding up the route lengths from each individual section.
Distance from the start to every point on the route is trivial as the entire `incr` array containing every single distance is synchronised over all processes using a Gather/Broadcast.

The local extremes are found using the given find_extremes() function, although with altered `first` and `last` variables. The local extremes are collected using MPI_Reduce() with the MPI_MAX and MPI_MIN operators meaning we get the global extremes in the collection process, since only the highest/lowest value of these extremes will arrive on the master.

Problem 3 (Group): Global coordination I

Handed-in C File: ***routempi.c***

The longest ascent in a local section will not necessarily be the longest ascent in the context of the full route, as an ascent might start at the end of one section, span the entire length of the second section, and end in the third section. So we have to stitch these lengths together.
The implementation that has been handed in stores three things, the length of the first ascent in a section if the section starts with an ascent, the longest ascent in the center, and the length of the final ascent, if the section ends with an ascent.
These values are calculated in the find_longest_ascent() function.
These three values are then gathered together with MPI_Gather() and stitched together properly by the master process so we get the correct length of the full ascent.
The values are stitched together in the process_longest_ascent() function.

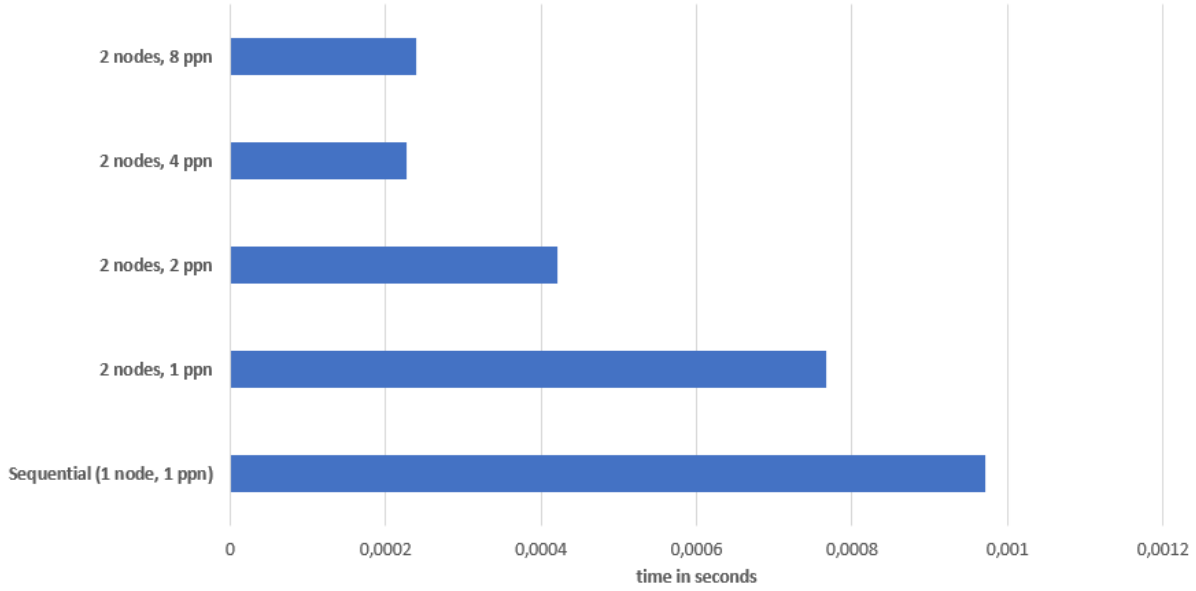
The first ascent is decided by checking if we're rising on the first interval (from `first` to `first+1`), setting it to 0.0001m's in length if the section does not start with an ascent. If the section does start with an ascent it is set to the length of this first ascent.
The program then runs along the middle of the section, checking the lengths of all ascents, and notes the longest one.
If the section ends with an ascent, the length of this is noted.
If the entire section consists of a single ascent (from `first` to `last`), the start and end ascent variables are set to be equal, so we can process this accordingly on the master.

Problem 5 (individual): Performance evaluation

Handed-in C File: ***routempitimed.c***
s175179

Instrumenting the program to be timed was done through usage of the clock_gettime() with the CLOCK_MONOTONIC_RAW as implemented in the first mandatory assignment.

Route used: tdf-2018-stage-01.csv



Looking at the plot we can see that the MPI program is always faster than the sequential implementation, but reaches an equilibrium around 2n4ppn where adding more tasks simply slows down the program because of communication costs.

The length of the route should decide how many nodes should be used.

Problem 6 (individual): Distance information

Handed-in C File: ***routempi.c***
s094766

To find the coordinates of the first occurrence of the extreme, each process must first find its local extremes. MPI_Reduce is then used with the operators `MPI_MAXLOC` and `MPI_MINLOC`, in order to compare and find the global extremes.
By looking at the full specification of MPI_Reduce as suggested and researching the possible operands of the MPI_Reduce, I found the following math description explaining the functionality of `MPI_MAXLOC` and `MPI_MINLOC`:

The operation that defines MPI_MAXLOC is:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = \max(u, v)$$

and

$$k = \begin{cases} i & \text{if } u > v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

MPI_MINLOC is defined similarly:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = \min(u, v)$$

and

$$k = \begin{cases} i & \text{if } u < v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}$$

Source: https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node79.html

I.e. to find the distance to the first maximum elevation: The function find_extremes() was rewritten, to use a `struct`, which beside the value of the maximum elevation value, also contains the index of the position in the arrays.

When MPI_Reduce gathers the values from the different processes with operand `MPI_MAXLOC` it first compares the elevation values to find the largest and then takes the associated index. If the two largest values are equal, the one with the smallest index value is selected.

That way, it is ensured that you always get the lowest index of the maximum value. The distance is then calculated via the function calc_total_dist([index of the position in the arrays]).

Problem 8 (individual): Communication pattern

s143780

The following table shows the five different processes (P_i) and which other two processes (P_{j1} & P_{j2}) they should communicate to. The value P_{j1} is calculated by the formular $P(i-1)\%5$ and the value P_{j2} by the formular $P(i+2)\%5$. The % equals to the mathmatical function modulo.

P_i	P_{j1}	P_{j2}
0	4	2
1	0	3
2	1	4
3	2	0
4	3	1

Since the MPI framework requires a process to either send or recieve and not both at the same time, we can only send two messages at the time. This is because two processes must be sending and the two receiving processes must be in the receiving state. This leaves one process waiting, and thats the downside of having an odd number of processes. The table below shows an example of making the communication exchange in five time units:

Time	P_0	P_1	P_2	P_3	P_4
1	Send to P_2	Send to P_3	Receive from P_0	Receive from P_1	Wait
2	Send to P_4	Wait	Receive from P_3	Send to P_2	Receive from P_0
3	Receive from P_1	Send to P_0	Send to P_4	Wait	Receive from P_2
4	Receive from P_3	Receive from P_4	Wait	Send to P_0	Send to P_1
5	Wait	Receive from P_2	Send to P_1	Receive from P_4	Send to P_3

Conclusion

Through this mandatory assignment, the group got an expanded view into the world of the MPI library. It has provided the group with a greater understanding of how to coordinate different processes and have them work on the same problem in parallel, and that parallelisation isn't always the best solution for any given problem, given that the problem has to be somewhat scaleable.