Emre Kılıç 2021400015
Kristina Trajkovski 2020400384
Group 47

Contents

Emre Kılıç 2021400015
Kristina Trajkovski 2020400384
Group 47

Files

The program consists of two files: mpi.py and client.py. The python file mpi.py represents the master process and client.py represents the children factory processes. The code must be run by cellist mpi.py with two following arguments representing the input and output files. The processes created will be discussed later in this report.

Implementation

- ### Object oriented programming

The information sent to the processes is gathered in a "Node" class. It holds the total number of cycles, all the wear factors, the maintenance limit, its children, its current wear (initialized as zero), its parent, its initial state, and initial input if it is a leaf node. It also has a few simple functions such as add_child() which appends an id to its array of children, is_root() which returns true if the node has no parent, and is_leaf() which is true if the node has no children. Once the processes are initialized, the appropriate node will be sent to the related process and production will be handled according to the information in the received object.

- ### Handling input and initializing the processes

The whole simulation begins with calling mpi.py with input and output files as arguments. There will be only one instance of i.py running at a time as a master process. It then reads the input file line by line and stores the information accordingly. Some of the information such as the wearing factors, number of cycles, and maintenance limit are mutual for all nodes, the rest such as parent, children, state etc. are handled according to the input node by node. Node 0 is treated as the root of the tree which only adds the products and reports to the master.

After reading the input, the master spawns as many child processes as denoted in the first line of input. Then, it sends a node containing all of the information mentioned to each process.

- ## Child processes

Each child process besides the root has the same algorithm outline:

1. If it is a leaf node it already has its string as input, otherwise the input becomes the concatenation of products received from children nodes
2. Perform the string operation denoted by the state and change the state according to the cycles denoted in the description
3. Update the current wear and report the id, cycle, and calculated cost to the master if it exceeds the maintenance limit
4. Send information to the parent
5. Repeat until the number of repetitions equals the number of cycles denoted in the input file

The root only gathers information from its children, concatenates it and sends to the master. It also breaks the infinite loop in the master process once it is done with it cycles (since it is at the top of the tree and information flows from the bottom upwards, if the root is done then all the other processes must be done).

- ## Handling the output

After the master process sends the nodes to related processes it enters an infinite loop. It always waits for data from any of the processes. The data can either be a product – which is of type string and immediately written to the output file, or maintenance log – an array of size 3 with values of [<machine_id>, <production_cycle>, <cost>] which saved to a local array. Once all of the products have been written to the output file, the maintenance logs are sorted (first by id, then by cycle) and written to the output file. It is important to note that the order of printing a maintenance log to the output has production cycle and cost inversed. It was much more convenient to hold it otherwise because it made sorting much easier.

Emre Kılıç 2021400015
Kristina Trajkovski 2020400384
Group 47

Important Design Choices

- Multiprogramming:

  The purpose of the project is to test and see the effects of having multiple processes working towards a single result. While the algorithm could have been handled with a single process linearly, there is a big loss of time and cycles when unrelated processes are waiting for each other to execute. This way, the resources are mainly utilized, for there is always at least one complete layer of nodes processing data (for example, in the beginning all the leaf nodes are processing the input at the same time), and there are many opportunities for the whole tree to be active. The time and cycle gain from this approach is significant.

- Separating the code for master and child processes:

  It definitely made spawning easier. However, something that needed to be handled when implementing this way was making a clear distinction between communication among child processes (node to its parent) and communication between a child process and master (root reporting products or any node reporting maintenance). For example, the difference can be seen when children try to send a message to another child process, it is not enough to call comm.send(), it must be node.comm.send() because comm.send() is reserved for master process.

- Reading all output in the master dynamically:

  It aligns with the project description and makes sure that all data from children processes is recorded as soon as possible without internal blocking.

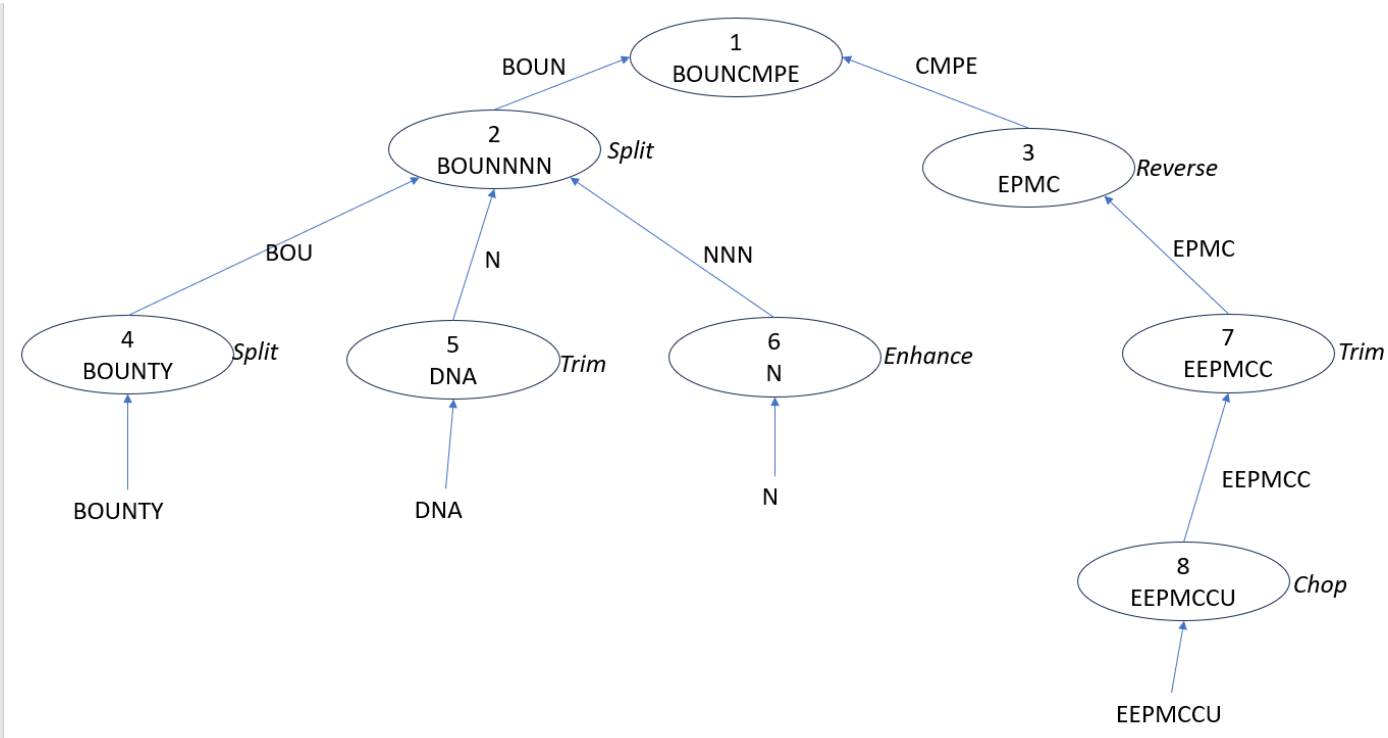
Emre Kılıç 2021400015
Kristina Trajkovski 2020400384
Group 47

Sample Input

Let the input file be filled out as such:

```
8
1
1 1 1 1 1
5
2 1 split
3 1 reverse
4 2 split
5 2 trim
6 2 enhance
7 3 trim
8 7 chop
BOUNTY
DNA
N
EEPMCCU
```

That means that there will be 8 child processes in total. Note that all of the possible string operations appear at least once. Then the diagram after one cycle (formatted in the same way as the example in the description) is:



Emre Kılıç 2021400015
Kristina Trajkovski 2020400384
Group 47

Challenges of real world implementation

If this was a real Industry 4.0 project aiming to implement a digital twin, the challenges for the implementation would mostly be related to synchronization. First, if we take the example of a car factory having different functions at each node, the time to prepare the product between nodes would differ significantly, and the processes would need to wait for each other in one way or another. For example, it would take much more time to prepare the motor than to paint the car. Considering such bottlenecks, parallel working would only be effective in subtrees with similar requirements. A good hierarchy between nodes (children and parents) would solve this problem. Also in the real world, maintenance blocks production to some extent – some of the products would be produced in a wrong way or delayed. In this project, we would just send a log, update the information, and continue. Lastly, in a real implementation, directly connecting each output to another input would not be a good idea, for there should be some quality testing and such in between. Having a node malfunction would then cause many of them to perform a poor job up the tree, which should be solved immediately.