Exercises on Implementation and Complexity of Distributed List Operations

Jan Plaza
Computer Science Department
SUNY Plattsburgh
101 Broad St., Plattsburgh, NY 12901, USA
jan.plaza@plattsburgh.edu
https://github.com/plazajan

February 11, 2024

Abstract

Our program simulates a chain (a case of line network) of asynchronous processing elements (CPEs), each communicating only with its neighbors. The chain can store a list of integers, one integer per CPE. This provides an environment for 40 exercises on design and implementation of distributed algorithms for list operations, from deque operations to sorting, and on evaluating their complexity. To aid students in the complexity analysis we introduce visualizations called Dominos and Activity Diagrams. The algorithms require a different design and may have complexity different from their counterparts on arrays or linked lists. They can be run on an any computer with Python 3 and offer students the familiar syntax of that language.

Keywords: distributed list, distributed computing, line network, simulation, Python 3, complexity, domino, activity diagram, educational.

1 Distributed Computing Environment, Ready to Use

We offer a programming environment that simulates a chain of processing elements (electronic circuits), each with its own simple processor, an independent clock and no shared memory. We refer to a single element as **Chain Processing Element (CPE)**. Each CPE is simulated by a separate thread in a Python 3 program. The CPEs communicate by message passing, using the

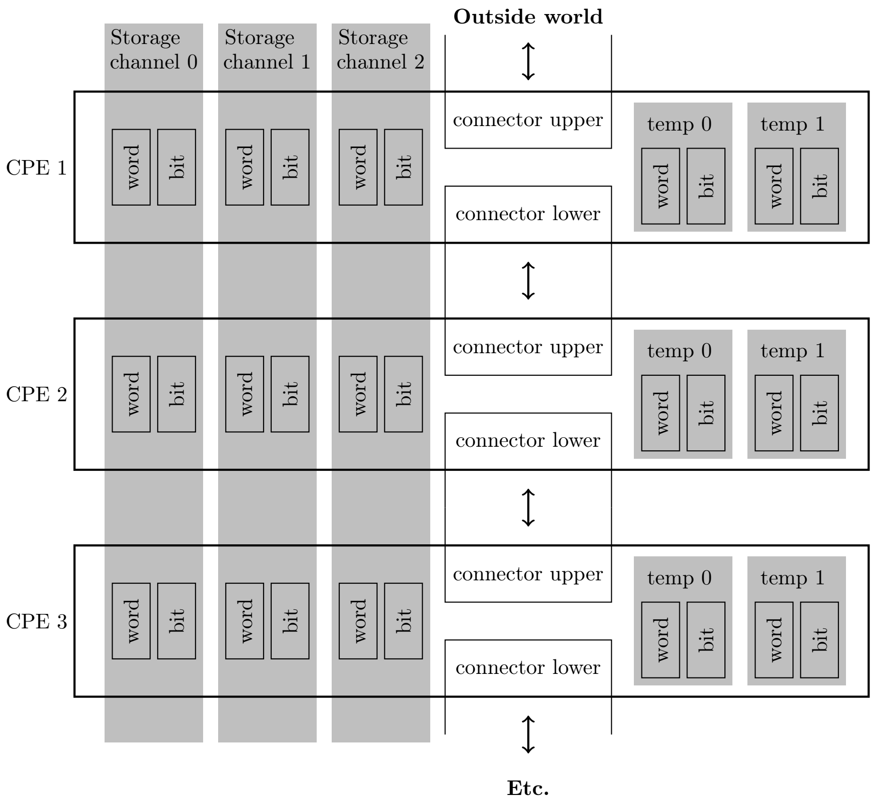

Figure 1: Chain Processing Elements (CPEs) - storage and communication.

4-phase bundled data handshaking protocol. All CPEs are alike and each has a very limited memory.

CPE 1 is the top element and the point where data can be provided to the chain, or results of calculation can be received by the outside world. Think of CPE 2 as located below CPE 1, and CPE 3 further down, etc. The chain is potentially infinite - a new CPE is added when needed.

We define chain as a case of a line network, as specified below. Distributed computing on a line network has been studied especially for sorting algorithms in [2], [3], [4], [5]. Our choice of characteristics of communication channels allows us to introduce visual tools of dominos and activity diagrams, and making it easier for beginning students to reason about distributed computation. Our objective is providing practice in design and complexity analysis, not achieving new complexity results.

The storage in a CPE and communication between CPEs are depicted in Figure 1 and described in the subsections below. As seen in Figure 1, the memory in a CPE is very limited, but the students will see that it is sufficient for the implementation of 40 common list operations.

1.1 Storage

Each CPE has persistent storage for three words and three bits. We think of word 0 and bit 0 as belonging to **storage channel 0**, word 1 and bit 1 as belonging to **storage channel 1**, and word 2 and bit 2 as belonging to **storage channel 2**. We are talking here about storage channels, not communication channels; one should not think that data is moving through channel 0, etc. When an algorithm that operates on the chain needs to store a list of numbers, it will store one item per CPE, all the items in the same storage channel. For instance, a merge algorithm will store one sorted list in channel 0, another in channel 1, and will put the computed merged list in channel 2.

There is a way of thinking among students that a memory location is non-empty if we store a value there, and otherwise the location is empty. While this is a useful abstraction, no place in the memory is ever truly empty. Places we consider empty store some accidental values (perhaps values from an earlier computation). So, while discussing hardware we need to think in terms of a memory location storing either a useful value or a non-useful/accidental value. The standard use of the bit in a storage channel is to provide that information about the corresponding word: if in channel 0 the `bit==True` then the word is non-useful/accidental; if the `bit==False` then the word is useful, stored by the current computation. Using the earlier intuition, think of the bit as a predicate `empty`.

Each CPE has also a temporary storage for two words and two bits; one word and one bit belonging to temp 0 storage and another word and bit to temp 1 storage.

A **word** is just an integer from 0 to 4095. A **bit** is just a Python Boolean object, `False` or `True`. While using commands to send a word or a bit, it is always to possible to send a **special value** `Any` - we do that when the actual value of word or bit is irrelevant (for instance when we just want to synchronize two CPEs); the option to send `Any` would not be available with the hardware CPEs, but in this simulation the object `Any` is specially designed to make debugging easier, for instance comparing `Any` to another object always raises an exception, even while evaluating `Any==Any`.

The list [1,2,3] can be pre-loaded into storage channel 0 by the statement `chain = ChainController([1,2,3])`. Then, the statement `chain.report(detailed=True)` shows values of word 0 and bit 0, etc. (The first column lists CPEs 1-4 of a chain CH3, `empt0` is the bit in channel 0.)

```
                word0  empt0
CH3.1               1  False
CH3.2               2  False
CH3.3               3  False
CH3.4             Any  True
```

According to the **list storage convention**, in a particular storage channel, a list starts in the top CPE and extends down to the first empty CPE word, called the **list terminator**; any words below the list terminator do not contribute to the list contents even if they are non empty. To clear channel 0 (make it empty) it is enough to set bit 0 in CH3.1 to `True`, this is exactly what the statement `chain.clear(0)` will do, where the `0` is the channel number. In the cleared channel, the values in the CPEs below CH3.1 do not matter:

```
                word0  empt0
CH3.1               1  True
CH3.2               2  False
CH3.3               3  False
CH3.4             Any  True
```

This last report can be presented in a more readable version by the statement `chain.report(detailed=False)`, which uses the dash symbol for “empty”:

```
                word0
CH3.1               -
CH3.2               2
CH3.3               3
CH3.4               -
```

1.2 Computation

In the proposed exercises, students will be designing distributed algorithms and writing programs/methods in the CPE class. The same set of programs is stored in every CPE in the chain.

A CPE can activate the CPE immediately below it by sending an **operation execution request**:

```
self.connectorLower.send_o(<operation name>, <channelA>, ...)
```

Such a request must precede any communications with the lower CPE. Operation request can never be sent to the CPE above, i.e. can never be sent through `connectorUpper`. When the operation request is sent, it is automatically received (without the lower CPE executing any explicit "receive" command in the method defining the operation in the CPE class), and it starts the execution of the program/method for the specified operation. Once the operation execution request has been received, the receiving CPE is said to be **active**. Once the execution of the program for the operation is complete, the CPE becomes **inactive**, just listening for a new operation request.

The storage channels are **persistent** (non-volatile) - not affected by the inactivity of the CPE. So, for instance, when a stack contents is stored in a channel, we can execute two push operations and the inactivity of the CPEs between these operations does not ruin the stack. In contrast to that, the temp 0 and temp 1 storage is **temporary** (volatile) - it can be relied upon only during the same period of activity of the CPE.

We are simulating a specific hardware design. While coding methods for chain operations in the CPE class, by **limited storage convention** the programmers can use only the following storage:

```
self.word[<channel_number>]
self.bit[<channel_number>]
self.temp_w[<index_0_or_1>]
self.temp_b[<index_0_or_1>]
```

and cannot use any variables, except the throw-away variable `_`. The code will use conditional statements, and in the case of reverse operation, also a while-loop. For the implementation of 40 common list operations, other programing constructs are unnecessary. In particular, recursion (which relies on an additional storage in a stack) is not allowed as it would violate the limited storage convention.

1.3 Communication

Each CPE can communicate directly only with the CPE immediately above and the CPE immediately below. If a CPE initiates "send" to one of these neighbors, it will block (pause execution) until the communication is complete - i.e. until it receives a confirmation that the neighbor has received the data. If a CPE initiates "receive" expecting some data from a particular neighbor, it will block until the data is actually received. The communication channel is guaranteed - messages cannot get lost. You cannot send a new message until the previous one is received, so the communication channel is ordered and unbuffered - the messages arrive in the order in which they were sent, one message is transferred at a time, and the channel does not store messages trying to deliver one while others are waiting or progressing behind.

As explained in the subsection above, channel 0, channel 1, channel 2 are abstract concepts related to the arrangement of storage in CPEs, but they are not communication channels. There is only one bidirectional communication channel between two neighboring CPEs. Any data received by a CPE is not labeled as destined for a particular storage channel, and it is up to the receiving CPE to store such data in the appropriate storage channel or in the temporary storage.

The communication commands (other than operation requests) are:

```
self.connectorLower.send_w(<word>)
self.connectorLower.send_b(<bit>)
self.connectorLower.send_W(<word>, <bit>)
self.connectorLower.send_B(<bit>, <bit>)

<store> = self.connectorLower.receive_w()
<store> = self.connectorLower.receive_b()
<store1>, <store2> = self.connectorLower.receive_W()
<store1>, <store2> = self.connectorLower.receive_B()
```

and they can be used with `connectorUpper` as well. (The observant reader have noticed that the `..._W` and `..._B` commands are redundant.)

1.4 Dominos, Activity Diagrams, and Complexity

While active, a CPE executes a sequence of communication commands with its lower neighbor (interspersed with communitacitons with the upper neighbor). The lower neighbor needs to execute a matching sequence of commands, otherwise errors or a deadlock would occur. The lower and upper sequence of communication commands executed by a CPE can be visualized as a “domino”.

For instance, the `member` operation that tests for membership in the list stored in `channelA` of a number received through `connectorUpper`, is coded as the following method in the CPE class.

```
def member(self, channelA):
    if not self.bit[channelA]: # if CPE's channelA is non-empty
        self.temp_w[0] = self.connectorUpper.receive_w()
        if self.word[channelA] == self.temp_w[0]: # found here
            self.connectorUpper.send_b(True)
        else: # keep looking below
            self.connectorLower.send_o("member", channelA)
            self.connectorLower.send_w(self.temp_w[0])
            self.temp_w[0] = self.connectorLower.receive_b()
```

```
                self.connectorUpper.send_b(self.temp_w[0])
        else: # if current CPE terminates the list
            _ = self.connectorUpper.receive_w()
            self.connectorUpper.send_b(False) # not found anywhere
```

Each CPE involved in the computation will execute a sequence of communication commands of types represented by the dominos in Figure 2.

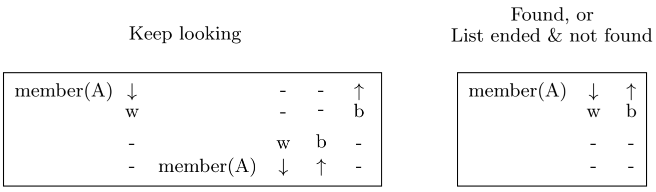

Figure 2: Dominos for the `member` operation

In either **domino**, the upper tag "member(A)" means that the CPE is executing the operation `member` on `channelA`. In the first domino, "member(A)" in the bottom row represents the operation activation request sent to the lower neighbor. The second domino covers two cases: when the item is found, and when the search has reached the end of the list without finding the item. The code for the cases is not the same: in the former case the bit sent up is `True`, while in the latter one - `False`. However at an abstract level concerned with a sequence of types of communications, they are the same, so they have the same domino.

For a chain computation to have a normal termination, the dominos of CPEs must match, as shown in Figure 3, where we test if 3 is a member of a list [1,2,3,4]. In this example, the word transmitted down is 3 and the bit transmitted up is `True`. CPE 4 and CPE 5 are essential for the representation of the list, but they are not activated. While comparing Figures 2 and 3, think of each domino as made of rubber that can stretch horizontally. A diagram showing matching dominos representing a computation is called **activity diagram**.

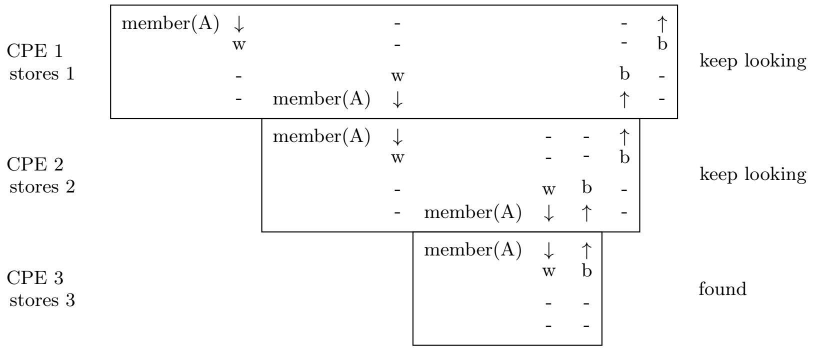

Figure 3: Activity diagram - looking for 3 in a list [1,2,3,4]

In this distributed environment, we discuss complexity under the **Communication Dominance Assumption**: the time spent on communications dominates the processing time; while estimating complexity we consider only communications.

We want to treat the chain of CPEs as a single computing device, so we concentrate on complexity measures which inform the user about practical aspects of using the chain to perform a sequence of operations. For a chain that is executing an operation, we ask students about the time when the CPEs would be available to perform another one.

The (worst case, time-) **top complexity** of an operation, is a measure of how long CPE 1 is active on a list that constitutes the worst case. (Once CPE 1 completes the execution of the operation and becomes inactive, it is available for another operation; there can be operations which are completed by CPE 1 before the rest of the chain completes the work. Another measure is the standard one used in distributed computing: the (worst case) **time complexity** is the maximum of the measures over all CPEs, how long it takes for CPE k, from the start of the operation by CPE 1 to the moment that CPE k is available for another operation, on a list which constitutes the worst case. So, in the case of our chain, time complexity is the time till the availability of all CPEs.

Because of Communication Dominance Assumption, we can read the complexity from activity diagrams. The top complexity is the width of the top domino in the worst case (as a function of the length of the list). The overall time complexity is the width of the activity diagram, in the worst case (as a function of the length of the list).

For the `member` operation, on a list of length n, in the worst case (when the item is not in the list), the activity diagram such as that in Figure 3 has width $O(n)$ being actually the width of the top domino, leading to the conclusion that overall time complexity and the top complexity of `member` are both linear.

A simple introductory exercise for students could be to modify `member` to produce `(<index>, True)`, where `<index>` is the list index of the first occurrence of the item, or `(Any, False)` if the item is not in the list.

2 A Sample Problem and Solution: Selection Sort

2.1 The Problem and Hints

Problem. Design and implement Selection Sort. Executing `sSort(channelA)` operation should sort the list in `channelA` in non-decreasing order, with the smallest item (if the list is not empty) in CPE1. Design the dominos, show an activity diagram and state the time complexity and top complexity of the algorithm.

Hints. Besides the operation `sSort(channelA)` you will need `minToTop(channelA)`. On a non-empty list, `sSort` executed by CPE n, starts `minToTop` in its lower neighbor and provides it with initial values. The execution of `minToTop` brings to CPE n the minimum value from the list that starts at CPE n and extends dawnwards. Once the minimum value is placed in `channelA`, CPE n starts `sSort` in its lower neighbor.

2.2 Designing Communication Among CPEs via Dominos

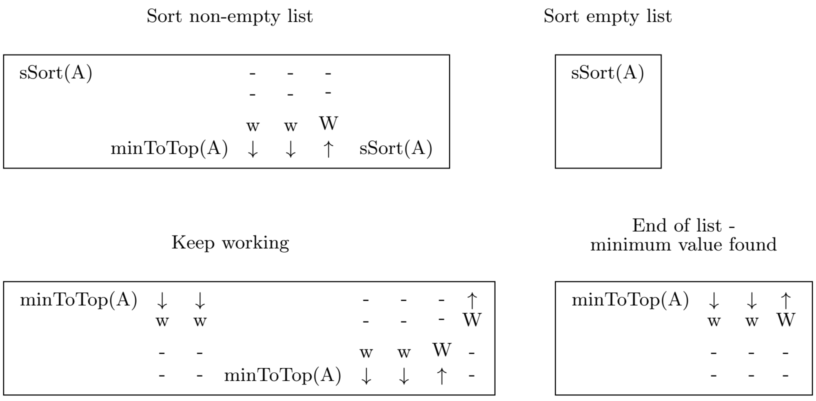

Figure 4: Dominos for selection sort

The CPEs executing `minToTop` pass downwards the minimumSoFar. Once these communications reach the list terminator, we know the actual minimum value for the entire list, and it will be passed upwards until it reaches the CPE that initiated this `minToTop` activity. Also, every CPE sends down the value from its `channelA`, in preparation for shifting the values in a portion of this channel. Starting from the bottom, the CPEs also send up a bit, which is `False` until the the location of the minimum value in the list is found. Once (the lowest) location is found, the shift-down starts for values in `channelA`, and the bit `True` is sent up.

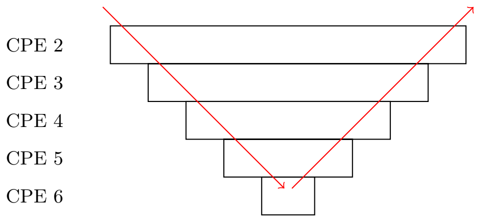

Figure 5: minToTop - activity diagram (simplified)

2.3 Activity of CPEs and Complexity

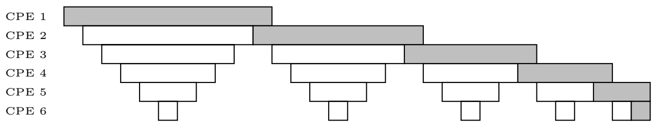

Figure 6: Selection sort - activity diagram on a 5-element list
(gray – sSort, white – minToTop)

The first inverted triangle on the left, brings the smallest value to CPE 1. The second triangle brings the second smallest value to CPE 2, ..., the n-th triangle brings the n-th smallest value to CPE n.

The top complexity of the algorithm is $O(n)$, where n is the length of the list. The overall time complexity is $O(n^2)$ and it is realized by the list terminator CPE.

2.4 An Implementation

```
def sSort(self, channelA):
    if not self.bit[channelA]: # if CPE's channel A is non-empty
        self.connectorLower.send_o("minToTop", channelA)
        # Send channel A word:
        self.connectorLower.send_w(self.word[channelA])
        # Send minSoFar:
        self.connectorLower.send_w(self.word[channelA])
        # Receive min and put in chA, discard the bit:
        self.word[channelA],_ = self.connectorLower.receive_W()
        self.connectorLower.send_o("sSort", channelA)
    else: # if current CPE terminates the list
        pass
```

```
def minToTop(self, channelA):
    if not self.bit[channelA]: # if CPE's channelA is non-empty
        # Preparation for shifting items down:
        # Receive channelA word from top neighbor:
        self.temp_w[0] = self.connectorUpper.receive_w()
        # Receive minSoFar from top neighbor:
        self.temp_w[1] = self.connectorUpper.receive_w()
        self.connectorLower.send_o("minToTop", channelA)
        # Preparation for shifting items down:
        # Send channelA word down:
        self.connectorLower.send_w(self.word[channelA])
        # Update minSoFar and send it down:
        if self.word[channelA] < self.temp_w[1]:
            self.connectorLower.send_w(self.word[channelA])
        else:
            self.connectorLower.send_w(self.temp_w[1])
        # Receive (min, found) pair:
        #     min - minimum value in the list under consideration.
        #     found - True, if the location of min has been found.
        self.temp_w[1],self.temp_b[1] = \
            self.connectorLower.receive_W()
        #print(self.name, self.temp_w[1], self.temp_b[1]) # debug.
        # Possibly start shifting, and send up (min, found):
        if not self.temp_b[1] \
               and self.word[channelA]!=self.temp_w[1]:
            self.connectorUpper.send_W(self.temp_w[1],False)
        else:
            self.word[channelA] = self.temp_w[0] # shift down chA
            self.connectorUpper.send_W(self.temp_w[1],True)
    else: # if current CPE terminates the list
        self.connectorUpper.receive_w() # from chA, discard
        # Receive minSoFar:
        self.temp_w[1] = self.connectorUpper.receive_w()
        # Send up (min, found) where found=False:
        self.connectorUpper.send_W(self.temp_w[1],False)
```

In the code for `minToTop`, there is a commented-out line which was used for debugging: `print(self.name, self.temp_w[1], self.temp_b[1])`.

3 Conclusion

There are 40 exercises on operations on distributed lists stored in a chain. The Python code and specifications of all exercises are available on GitHub, although they still undergo improvements. Some exercises are labeled as "harder" and have hints. For every exercise there are unit tests ready to be executed and a `ChainController` interface that facilitates customized testing by the student.

There are well known, high quality, general simulators of distributed computing, such as SimGrid and WRENCH, described in [1], which provide great many configuration options. Also, Hardware Description Languages, including SystemC and Chisel, can be used. Our exercises can be executed on any computer and offer students the familiar syntax of Python. Using a single distributed structure of a chain keeps the exercises simple and the students will not be overwhelmed.

The chain distributed system has been designed for exercises, and does not occur in practical applications of distributed computing, where the focus is most often on computers/processors forming an arbitrary, non-linear graph. Because of our chosen architecture, executions of distributed algorithms can be visualized using dominoes stacked in activity diagrams, and the complexity can be deduced from these diagrams.

The exercises can be of use to courses involving basic data structures, design of algorithms, models of computation, and to courses concerned with distributed computing or hardware design. Of course, they are not meant to provide material for the entire course, but selected exercises can be used at an appropriate stage. The exercises will be tried in the classroom by the author in the spring 2024 semester.

As far as we know, this specific version of line network has not been used before and algorithms/solutions are different from any currently available on the internet, possibly giving instructors a temporary advantage over solution-regurgitating Generative AI systems.

References

[1] Casanova, H., Tanaka, R., Koch, W., Ferreira da Silva, R., Teaching parallel and distributed computing concepts in simulation with WRENCH, *Journal of Parallel and Distributed Computing 156*, October 2021, pp. 53-63.

[2] Hofstee, H. P., Martin, A. J., Van De Snepscheljt, J. L. A., Distributed Sorting, *Science of Computer Programming 15*, 1990, pp. 119-133.

[3] Loui, M. C., The Complexity of Sorting on Distributed Systems *Information and Control 60*, 1984, pp. 70–85.

[4] Prasath, R. R., An alternative time — optimal distributed sorting algorithm on a line network, *NC2010: 6th International Conference on Networked Computing, Gyeongju, Korea (South)*, IEEE, 2010, pp. 1-6.

[5] Sasaki, A., A time-optimal distributed sorting algorithm on a line network, *Inform. Proc. Lett. 83*, 2002, pp. 21-26.