Synthesizing Proteins

LPCT Project

Bernardo Toninho

26 October, 2021

1 Synthesizing Proteins Concurrently

Have you ever wondered how a protein is produced? Well, you are about to find out (skipping the complicated details)!

Proteins are essentially long chains of amino acids that are arranged in some specific, orderly fashion. Proteins perform many functions in biology, such as DNA replication, and effectively carry out all the functions that are necessary for life. Proteins differ from each other primarily in their sequence of amino acids, governed by the nucleotide sequence of their genes, which then undergoes so-called protein folding into a specific 3D structure that determines its function.

1.1 Transcription and Translation

In effect, to manifest a gene is to produce, or synthesize, its corresponding protein. Protein synthesis can be divided into two steps: **transcription** and **translation**. Transcription is the process by which the information in DNA is transferred to a messenger RNA (mRNA) molecule, which acts as a form of template. The resulting mRNA molecule is a single-stranded copy of the gene, which is then **translated** into a protein molecule.

During translation, the mRNA molecule is "read", relating the DNA sequence to the amino acid sequence in the protein. Each group of three bases in mRNA is called a **codon**. Each codon encodes a particular amino acid. The mRNA sequence is then used as a template to assemble the chain of amino acids that form a protein.

1.1.1 Transcribing DNA to mRNA

DNA is a complex molecule composed of two so-called polynucleotide chains (that form the famous double helix). The chains are composed of simpler units

called *nucleotides*. Each nucleotide is composed of one of four nucleobases: **C**ytosine, **G**uanine, **A**denine and **T**hymine. DNA strands are effectively described as a sequence of nucleobases, and so can be represented by a string of Cs, Gs, As and Ts.

The transcription process effectively scans the nucleobase sequence, one base at a time, and builds an RNA molecule out of so-called complementary nucleotides. The RNA transcript carries the same information as the DNA chain, but where the base Thymine (T) is replaced by the RNA-only base, Uracil (U).

While there is more to transcription than this, in our approximation, the following is a valid transcription of a DNA sequence:

DNA Sequence - `GGGCCGTCTTCTTCGTTAAA`

Transcribed mRNA - `GGGCCGUCUUCUUCGUUAAA`

1.1.2 Translating mRNA to Amino acid chains

The set of amino acids is as follows: Alanine (Ala), Arginine (Arg), Asparagine (Asn), Aspartic acid (Asp), Cysteine (Cys), Glutamine (Gln), Glutamic acid (Glu), Glycine (Gly), Histidine (His), Isoleucine (Ile), Leucine (Leu), Lysine (Lys), Methionine (Met), Phenylalanine (Phe), Proline (Pro), Serine (Ser), Threonine (Thr), Tryptophane (Trp), Tyrosine (Tyr) and Valine (Val). Don't worry, we will only use the shorthands to refer to these. Each amino acid is characterized by three mRNA nucleotide bases, according to the table in Figure 1.

Given a codon, we can determine its corresponding amino acid as follows: we use its first nucleotide to refer to the line on the left; its second nucleotide to refer to the column; and, its third nucleotide to refer to the line on the left. As you can see, there is a lot of redundancy in this process – many different codons translate to the same aminoacid (this is called genetic redundancy). For example, the codon `GUU` translates to Val, whereas the codon `AUG` translates to Met.

In order for the resulting amino acid chain to form a valid protein, it must adhere to the following (simplified) rules: its first aminoacid must be Met and it must be ended by one of the three termination codons (`UAA`, `UAG` or `UGA`).

Now that you know how proteins are produced from DNA sequences, your task is to implement a concurrent protein synthesis analyzer. Your analyzer must be able to answer two kinds of queries:

1. Given a DNA sequence, what is its transcription into mRNA.
2. Given a DNA sequence, what is its corresponding amino acid chain (and does that chain form a valid protein).

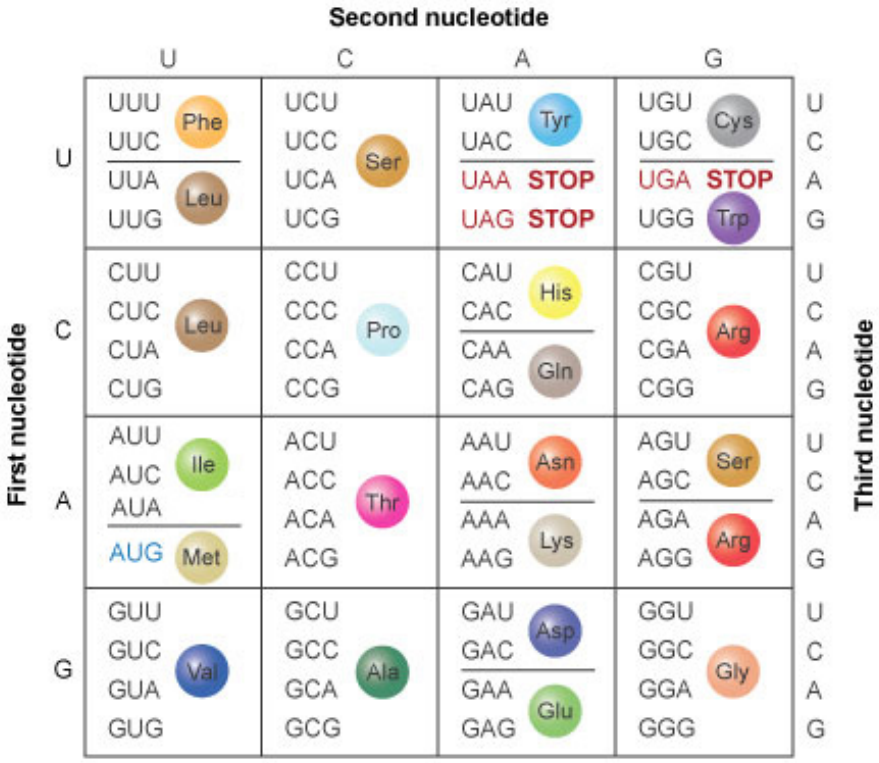

Figure 1: Amino Acid Table (© 2014 Nature Education)

The two queries pressupose the rules described above. You need not investigate further on how these processes actually happen in real cells.

While it may seem overkill to realize this as a concurrent program, lets do it anyway.

2 System Specification

Your system must be composed of two software components, a *client* and a *server*. Each component *must compile to a separate binary*.

2.1 Client

The client sends user-specified requests to the server, receives and prints the result to standard output.

The client and the server communicate using a standard *TCP socket*. The client binary should take as command line arguments the server address and port and a flag to determine whether it will perform a *transcription* or *translation* request. The client will then read from standard input the DNA sequence (as a string containing only occurrences of the letters C, G, A and T), whose length must be a multiple of three, send the corresponding request to the server, receive the response and print it to standard output. You may assume the DNA sequence only contains valid nucleotide bases.

To launch the client, the command should be:

```
./client -transcribe host:port
```

or

```
./client -translate host:port
```

respectively, where `host:port` is the server address and port, the flag `-transcribe` specifies that the request will be for transcription and the `-translate` flag signals a translation request. The client will then read the DNA string from standard input and send to the server a message of the form:

```
[Request <type> <data>]
```

The client will then receive from the server a message of the form:

```
[Result <type> <data> <valid>]
```

with the appropriate response. If a client loses contact with the server, it should print a `Disconnected from server` message to standard output and terminate.

For convenience, the provided repository contains an implementation of the messages (package `message`) you will need to use between the client and the

server, as well as their marshalling and unmarshalling to-and-from JSON byte slices. You may wish to reuse these messages inside your server, or use your own data types. If you want to reuse the message type but feel the need to add an extra field to the message, you will need to readjust the existing marshalling and unmarshalling code.

2.2 Server

The server receives requests from one or more clients, performs the corresponding transcription or translation (and validation) request and answers back with the reply.

To launch the server, the command should be:

```
./server port
```

The server **must** be able to handle client requests concurrently. This means that at any given time, the server can be performing translations and transcriptions for **different** clients (and so of various DNA sequences).

From the perspective of the server, a client request (using the message formats from the supplied code — see below):

```
[Request type DNA]
```

can (and should) be split into multiple smaller jobs, performed by multiple workers. The partition of the request into jobs and their aggregation into a final answer is relatively straightforward: in both translation and transcription, the order of the DNA and the transcribed mRNA matters, and so the aggregation must preserve the original order of the DNA sequence. For transcription, the operation is at the single nucleotide level. For translation, since the sequence is assumed to have a length that is a multiple of three, the operation ultimately can be performed at the codon level.

Note: A translation request requires performing transcription. The server can exploit this when splitting such a request into tasks that can be performed concurrently. For instance, you should think on how to divide the request such that there is the most potential for parallelism across the combined transcription-translation steps.

If a server loses contact with a client, it should stop working on any pending requests on behalf of the disconnected client (jobs that are already underway need not be forcibly terminated, you can just wait for them to complete and ignore the result).

2.2.1 Additional Server Requirements

1) The server must use multiple workers to perform the client jobs. However, despite the fact that goroutines are cheap, you probably want to limit the number of goroutines that the server creates. While the focus is on concurrency, you should aim to *maximize* the potential for parallelism.

2) Choosing a good way of organizing (and creating) workers, suited to the specific context of the problem is part of the challenge. There might be several equivalent options, but you should be able to argue why your choices are good. You will likely want to assign to each worker *a range* of the DNA / mRNA sequence, rather than a single base.

3) Choosing a good way of aggregating the work is also part of the challenge, considering the order of the sequence must be preserved. Note also that a translation request is effectively a two step process, so care is needed to ensure the absence of data races.

4) **Correctness is mandatory**. The server must only produce **correct** answers that **match** the client's request.

5) The server should **try** to minimize mean response time for client requests. If the server gets a very large request, followed by a small request, it is reasonable to expect that the small request finishes before the large request. If the server gets several requests of similar size, the requests should finish roughly (although not necessarily) in the order they arrived. Your code should document if/how these concerns are taken into account.

---

3 Supplied Code

- `message/message.go` defines the message types needed for client/server exchanges.
- `client/client.go` is where you will implement your client program.
- `server/server.go` is where you will implement your server program.

4 Grading

Projects are to be completed preferably in groups of two (at most three), which must match with the groups of the mini-project. The standard plagiarism rules apply and will be enforced.

The project deadline will be on the 8th of November at 23:59, enforced by Github Classroom. You must turn in your code and a **brief** report (add a PDF to the repository), documenting the various design choices in your work. The report should try to address the questions below.

Your code **must not** use locks or mutexes. All synchronization must be done using goroutines, channels, and Go's channel-based `select`. Waitgroups may be used to ensure a program does not terminate before it should. Any of the techniques seen in lecture may be used (but they need not all be used!).

You should try to avoid using fixed-size buffers and arrays to store things that can grow arbitrarily in size. For example, using fixed-size buffers for message queues might not be the best solution (which doesn't mean you should not use buffered channels entirely!). Its often better to use a linked list (see the `container/list` package, but note that it is **not** thread safe) or some other data structure that can expand to arbitrary size.

Your project will receive a better grade according to the following criteria:

- **Correctness:** Does the server answer correctly? (Critical!)
- **Worker creation:** When and how are workers created?
- **Worker management:** How are workers organized amongst themselves and in the overall system? How is work divided among and allocated to workers? How is work aggregated? Does the solution limit parallelism potential?
- **Data management:** How are data structure accesses managed in order to ensure the absence of data races?
- **Fairness and responsiveness:** How does the server aim to minimize mean response time?
- **Overall code quality**
- **Use of tests to validate your code**

The criteria are not listed in any particular order. The main focus will be on correctness and worker creation/management.

Note: The project may seem scary, but its bark is worse than its bite. Relax and try to have some fun!