COMPSCI 267 Final Project Report (Group 40)

Aadil Manazir, Tony Hong, Sharan Sahu

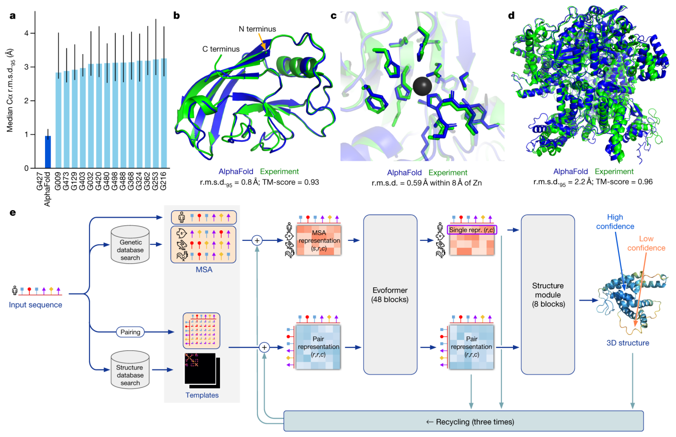
May 2023

1 Introduction

Training and serving large-scale neural networks require complicated distributed system techniques. Alpa is a tool that aims to automate large-scale distributed training and serve with just a few lines of code with data, operator, and pipeline parallelism. However, Alpa only targets popular deep-learning models such as Transformers and ResNet. It would be ideal for Alpa to support other novel neural network architectures. In particular, our project is focused on targeting AlphaFold, an ML model that predicts a protein's 3D structure from its amino acid sequence. In particular, we will discuss the implementation and design choices we made for speeding up AlphaFold using parallelization of MSA construction and Alpa for evaluation of AlphaFold.

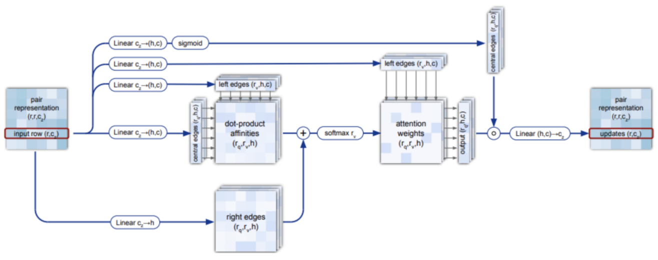
2 AlphaFold Prediction Pipeline

Alphafold is a ML model that predicts a protein's 3D structure from its amino acid sequence. There are 3 stages that we must go through to produce a protein's 3D structure as can be seen in the graphic below:

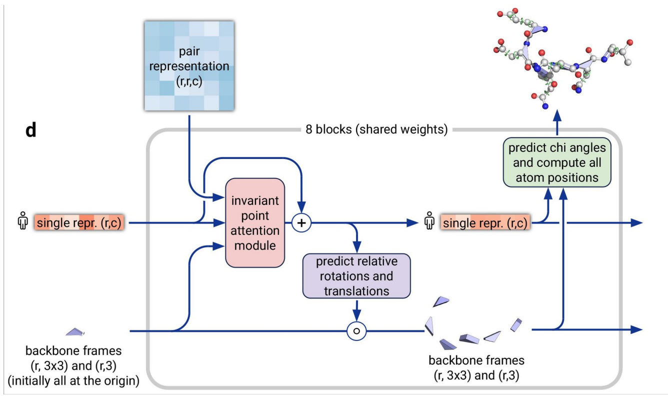

a
Median Cα r.m.s.d.95 (Å)
G427
AlphaFold
b
N terminus
C terminus
AlphaFold Experiment
r.m.s.d.95 = 0.8 Å; TM-score = 0.93
c
AlphaFold Experiment
r.m.s.d. = 0.59 Å within 8 Å of Zn
d
AlphaFold Experiment
r.m.s.d.95 = 2.2 Å; TM-score = 0.96
e
Input sequence
Genetic database search
Pairing
Structure database search
MSA
Templates
MSA representation (s,r,c)
Pair representation (r,r,c)
Evoformer (48 blocks)
Single repr. (r,c)
Pair representation (r,r,c)
Structure module (8 blocks)
High confidence
Low confidence
3D structure
← Recycling (three times)

The first stage which we call MSA construction includes an exhaustive database search against the input amino acid to construct a multiple sequence alignment (MSA) which identifies similar, but not identical, sequences that have been identified in living organisms. This enables the determination of the parts of the sequence that are more likely to mutate, and allows us to detect correlations between them. Additionally, we also try to identify proteins that may have a similar structure to the input (“templates”), and constructs an initial representation of the structure, which we call the “pair representation”. This is, in essence, a model of which amino acids are likely to be in contact with each other.

The next stage is called model inference which combines MSA and pair representations refinement (called predictions in the graphic below) and protein construction (called relaxation and output in the graphic). The prediction model is essentially a neural network made of evoformer blocks. These blocks are similar to blocks seen in transformers using the attention mechanism. We won’t go into the details exhaustively, but for now you can understand it as an “oracle” that can quickly identify which pieces of information are more informative. The objective of this part is to refine the representations for both the MSA and the pair interactions, but also to iteratively exchange information between them. A better model of the MSA will improve the network’s characterization of the geometry, which simultaneously will help refine the model of the MSA.



Lastly, this information is taken to the last stage: the structure module. This sophisticated piece of the pipeline takes the refined “MSA representation” and “pair representation”, and leverages them to construct a three-dimensional model of the structure. The end result is a long list of Cartesian coordinates representing the position of each atom of the protein, including side chains.


d
pair representation (r,r,c)
8 blocks (shared weights)
single repr. (r,c)
invariant point attention module
predict relative rotations and translations
single repr. (r,c)
predict chi angles and compute all atom positions
backbone frames (r, 3x3) and (r,3) (initially all at the origin)
backbone frames (r, 3x3) and (r,3)

AlphaFold currently uses two types of workloads for end-to-end predictions: MSA (Multiple Sequence Alignment) construction based on CPUs and model inference on GPUs. The first CPU stage is the overall bottleneck in computation as it can take hours for a single protein due to the large database sizes and I/O bottlenecks. In contrast, the GPUs in the CPU stage are relatively idle. Thus, this results in low GPU utilization and restricting the capacity of large-scale structure predictions.

3 Implementation and Design Choices

3.1 MSA Construction Runtime Analysis

The MSA construction stage is run by the CPU. During this stage, AlphaFold searches through three separate databases to generate the MSA using `jackhmmer`. Sequentially searching each database leads to a bottleneck [2]. The way that MSA construction works is that it looks through other well-studied and known amino acid sequences and determines how similar a given sequence is to other sequences. That is, what is the minimum number of changes needed to align two sequences together? Using this, AlphaFold can extrapolate different metrics about how the protein structure should turn out.

Methods like `jackhmmer` and `hhblits` use modern homology detection methods like Pairwise Alignment Hidden Markov Models for MSA construction. However, more classical techniques like Dynamic Programming can be used for MSA construction, and both these techniques yield similar asymptotic runtimes up to some constants. Let α be the length of the longest supported sequence in the AlphaFold pipeline, and let n be the number of sequences in the databases. If we have a sequence of length l, we would expect a runtime of $\mathcal{O}(n \times \alpha \times l)$. The runtime analysis proof of this has been provided below.

Runtime Analysis Proof: Consider two strings, denoted $x[1,...,n]$ and $y[1,...,m]$. Let us define a sub-problem, denoted $E[i,j]$ = Edit distance between $x[1,...,i]$ and $y[1,...,j]$. Then, notice that we can write the following recurrence relation:

$$E[i,j] = \min \begin{cases} E[i, j-1] + 1 & \text{Insertion} \\ E[i-1, j] + 1 & \text{Deletion} \\ E[i-1, j-1] + \mathbb{1}\left(x[i] \neq y[j]\right) & \text{Substitution} \end{cases}$$

Using this, we can solve the problem using the following:

Code Block 1: Minimum Edit Distance Solution

```
for i = 1 to n:
    for j = 1 to m:
        E[i, j] = min(E[i, j-1] + 1, E[i-1, j] + 1, E[i - 1, j-1] + (x[i] !=
            y[j])
return E[n, m]
```

This naturally leads to a runtime of $\mathcal{O}(nm)$. Applying this to our use case, we have n total comparisons, each of which take maximum $\mathcal{O}(\alpha \times l)$. Hence, this yields a total runtime of $\mathcal{O}(n \times \alpha \times l)$.

Clearly, we can't change α and l. However, if we parallelize across the queried databases, we can reduce n to the maximum number of sequences in a single database.

3.2 Parallelization of MSA Construction

Initially, we attempted to divide each database query into chunks, and then divide the chunks amongst the available processors. However, after a significant amount of effort, we weren't able to make this work with the python `jackhmmer` library.

Since chunking the queries wasn't possible, we instead decided to parallelize the MSA construction by dividing the databases amongst the processors. Depending on the number of available CPUs, we assigned the databases accordingly, allowing each CPU to sequentially search through its assigned database. We also created a new function to support this multi-processing, `run_jackhammer_search`. The function first initializes a `jackhmmer` runner with the given binary path, database path, and other parameters, and then conducts a query on the FASTA file. The function returns a tuple containing the sequence, the database name, and the result of the query. The code for `run_jackhammer_search` is seen below.

Code Block 2: `run_jackhammer_search`

```
def run_jackhmmer_search(sequence, fasta_path, db_config,
jackhmmer_binary_path):
    db_name = db_config['db_name']
    jackhmmer_runner = jackhmmer.Jackhmmer(
        binary_path=jackhmmer_binary_path,
        database_path=db_config['db_path'],
        get_tblout=True,
        num_streamed_chunks=db_config['num_streamed_chunks'],
        z_value=db_config['z_value'])
    result_for_sequence = jackhmmer_runner.query(fasta_path)
    return (sequence, db_name, result_for_sequence)
```

The AlphaFold `get_msa` function performs multiple sequence alignment for a set of sequences across different databases. Using the the python `cuncurrent.futures` module, we modified the `get_msa` function to divide the databases amongst the processors, each of which ran `run_jackhammer_search` on their assigned database(s). The new `get_msa` function creates a mapping of each unique sequence to a FASTA file, initializes a dictionary for results, and then runs `run_jackhammer_search` concurrently on multiple processors for each sequence against each database. The function returns a dictionary containing the alignment results for each sequence from each database. The code for the `get_msa` function is seen below.

Code Block 3: `get_msa`

```
import concurrent.futures
from functools import partial

def run_jackhmmer_search(sequence, fasta_path, db_config,
    jackhmmer_binary_path):
```

```
    db_name = db_config['db_name']
    jackhmmer_runner = jackhmmer.Jackhmmer(
        binary_path=jackhmmer_binary_path,
        database_path=db_config['db_path'],
        get_tblout=True,
        num_streamed_chunks=db_config['num_streamed_chunks'],
        z_value=db_config['z_value'])
    result_for_sequence = jackhmmer_runner.query(fasta_path)
    return (sequence, db_name, result_for_sequence)


def get_msa(sequences, databases):
    sequence_to_fasta_path = {}
    for sequence_index, sequence in enumerate(sorted(set(sequences)), 1):
        fasta_path = f'target_{sequence_index:02d}.fasta'
        with open(fasta_path, 'wt') as f:
            f.write(f'>query\n{sequence}')
        sequence_to_fasta_path[sequence] = fasta_path

    raw_msa_results = {sequence: {} for sequence in sequence_to_fasta_path.
       keys()}

    with concurrent.futures.ProcessPoolExecutor() as executor:
        tasks = []
        for sequence, fasta_path in sequence_to_fasta_path.items():
            for db_config in databases:
                task = executor.submit(run_jackhmmer_search, sequence,
                   fasta_path, db_config, JACKHMMER_BINARY_PATH)
                tasks.append(task)

        with tqdm.notebook.tqdm(total=len(tasks), bar_format=TQDM_BAR_FORMAT
           ) as pbar:
            for future in concurrent.futures.as_completed(tasks):
                sequence, db_name, result_for_sequence = future.result()
                raw_msa_results[sequence][db_name] = result_for_sequence
                pbar.update(n=1)

    return raw_msa_results
```

We also wanted to note that we also tried to parallelize MSA construction using distributed computing via Ray. The structure holds similar to what is seen above except we allocate different workers to different databases to do multiprocessing for MSA construction. The code for this can be seen below:

Code Block 4: `get_msa_ray`

```
import ray
import jackhmmer
from tqdm.notebook import tqdm

ray.init()

@ray.remote
def run_jackhmmer_search(sequence, fasta_path, db_config,
    jackhmmer_binary_path):
    db_name = db_config['db_name']
    jackhmmer_runner = jackhmmer.Jackhmmer(
        binary_path=jackhmmer_binary_path,
        database_path=db_config['db_path'],
        get_tblout=True,
        num_streamed_chunks=db_config['num_streamed_chunks'],
        z_value=db_config['z_value'])
    result_for_sequence = jackhmmer_runner.query(fasta_path)
    return (sequence, db_name, result_for_sequence)

@ray.remote
def create_fasta_file(sequence, sequence_index):
    fasta_path = f'target_{sequence_index:02d}.fasta'
    with open(fasta_path, 'wt') as f:
        f.write(f'>query\n{sequence}')
    return (sequence, fasta_path)

def get_msa(sequences, databases, JACKHMMER_BINARY_PATH):
    sequence_to_fasta_path = {}
    fasta_tasks = []
    for sequence_index, sequence in enumerate(sorted(set(sequences)), 1):
        fasta_tasks.append(create_fasta_file.remote(sequence, sequence_index
            ))

    for result in ray.get(fasta_tasks):
        sequence_to_fasta_path[result[0]] = result[1]

    raw_msa_results = {sequence: {} for sequence in sequence_to_fasta_path.
        keys()}
    tasks = []
    for sequence, fasta_path in sequence_to_fasta_path.items():
        for db_config in databases:
            task = run_jackhmmer_search.remote(sequence, fasta_path,
                db_config, JACKHMMER_BINARY_PATH)
            tasks.append(task)

    with tqdm(total=len(tasks)) as pbar:
        for future in ray.get(tasks):
            sequence, db_name, result_for_sequence = future
            raw_msa_results[sequence][db_name] = result_for_sequence
            pbar.update(n=1)

    return raw_msa_results
```

we first initialize Ray using `ray.init()`. We then define two remote functions: `run_jackhmmer_search` and `create_fasta_file`. `run_jackhmmer_search` performs a jackhmmer search, while `create_fasta_file` creates a fasta file for a given sequence. We use the `@ray.remote` decorator to mark these functions as remote. In the `get_msa` function, we first create a list of `create_fasta_file` tasks using `remote()` and submit them to Ray using `ray.get()`. We then iterate over the results to get the paths to the fasta files. We create a dictionary `sequence_to_fasta_path` to store the fasta file paths for each sequence. We then create a list of `run_jackhmmer_search` tasks for each combination of sequence and database, and submit them to Ray. We iterate over the results using `ray.get()` and update the `raw_msa_results` dictionary with the results of each search. Finally, we return `raw_msa_results`. We were having a lot of problems with distributed computing because of dependency issues with the workers and consequently could not get this method to work efficiently.

We will discuss the performance of the parallelized MSA construction in the Performance section.

3.3 Using Alpa to Parallelize AlphaFold Inference

The model inference stage is run by the GPU. To speed this up, we used the Alpa library, which provides auto-parallelization capabilities to parallelize `jax` functions across a distributed device cluster. It automatically finds the best parallelization strategy and does the code transformations that yield that strategy. [1]

To use Alpa, we needed to modify the AlphaFold repository to make use of Alpa in its prediction step. First, we forked the AlphaFold repository, and added these two necessary dependencies for Alpa in the requirement.txt file:

Code Block 5: Alpa dependencies

```
alpa
jaxlib==0.3.22+cuda111.cudnn805
```

We then determined where the model inference occurred in the AlphaFold repository. Namely, prediction occurs in the below code:

Code Block 6: AlphaFold Prediction

```
    model_runner = model.RunModel(cfg, params)
    processed_feature_dict = model_runner.process_features(np_example,
       random_seed=0)
    prediction = model_runner.predict(processed_feature_dict, random_seed=
       random.randrange(sys.maxsize))
```

Hence, in our forked AlphaFold repository, we modified the `predict` function to have the decorator `@alpa.parallelize` decorator, which automatically parallelizes the `jax` function. The automatic parallelization feature of Alpa can potentially lower the runtime of the `predict` method by distributing the computations across multiple devices in a cluster, leading to faster computation times. The code is seen below.

Code Block 7: Predict Function Parallelization

```
@alpa.parallelize
  def predict(self,
              feat: features.FeatureDict,
              random_seed: int,
              ) -> Mapping[str, Any]:
    """Makes a prediction by inferencing the model on the provided features.

    Args:
      feat: A dictionary of NumPy feature arrays as output by
        RunModel.process_features.
      random_seed: The random seed to use when running the model. In the
        multimer model this controls the MSA sampling.

    Returns:
      A dictionary of model outputs.
    """
    self.init_params(feat)
    logging.info('Running␣predict␣with␣shape(feat)␣=␣%s',
                 tree.map_structure(lambda x: x.shape, feat))
    result = self.apply(self.params, jax.random.PRNGKey(random_seed), feat)

    # This block is to ensure benchmark timings are accurate. Some blocking
        is
    # already happening when computing get_confidence_metrics, and this
        ensures
    # all outputs are blocked on.
    jax.tree_map(lambda x: x.block_until_ready(), result)
    result.update(
        get_confidence_metrics(result, multimer_mode=self.multimer_mode))
    logging.info('Output␣shape␣was␣%s',
                 tree.map_structure(lambda x: x.shape, result))
    return result
```

Although we were able to finish an initial implementation of AlphaFold with Alpa, we were not able to successfully run it because of several dependency issues. We were able to resolve a few, but we were not able to resolve all of them in time. We discuss the details of these issues in the Obstacles and Challenges section.

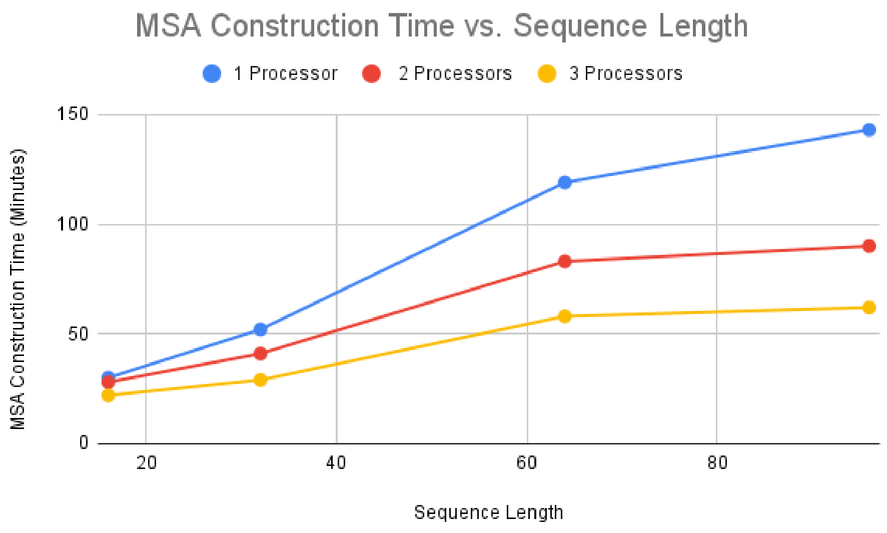
4 Performance

4.1 MSA Construction Performance

For one, two, and three processors, we compare the MSA construction runtime for four proteins of varying sequence lengths. Note that for each single protein, more than three processors would not be beneficial, as there are only three databases to query. However, using many processors on a job of several proteins will likely decrease average runtime [2]. Below is a table and a graph of the MSA construction runtime vs. protein sequence length.

Table 1: Runtime (minutes) for MSA Construction on different numbers of processors

Sequence Length	**1 Processor**	**2 Processors**	**3 Processors**
16	30	28	22
32	52	41	29
64	119	83	58
96	143	90	62



We see that running serial MSA construction with 1 processor scales roughly as we predicted in our runtime analysis above. In this case, since we have a fixed number of databases, the serial MSA implementation runtime is roughly proportional to $O(l)$, where l is the sequence length. As we increase the number of processors, we see that our parallelized MSA construction is roughly $\mathcal{O}(l)$ with better scaling. We can see that the parallelized MSA construction led to a significant runtime speed-up. However, using three processors, we weren't able to achieve a 3x speed up. This is likely because the runtime isn't always uniformly distributed across each database. Namely, the database that has the longest runtime is always the bottleneck. Additionally, there are overheads to consider, as MSA construction tends to be I/O intensive. This might be why the degree of runtime improvement is greater for longer sequences- for shorter sequences, the overhead cost may be more substantial in relation to the query cost.

5 Obstacles and Challenges

A significant amount of our time was spent integrating the AlphaFold codebase into Perlmutter. The setup script in AlphaFold initially modified many files/directories in the root directory, and Perlmutter did not allow for this. It took a significant amount of effort and time to devise workarounds for all of the commands that we didn't have permission for. In addition to the changes in the setup script, we also had to debug the actual AlphaFold implementation as well. This was challenging because our workarounds would sometimes lead to additional dependency issues.

Another challenge was integrating Alpa's dependencies with AlphaFold's dependencies. Some initial issues that we faced were having incompatible jax and jaxlib versions and missing a dependency in our path. Another issue was that Alpa only supports up to Python 3.9 while AlphaFold was running on Python 3.10. This caused some conflicts within the dependencies. There was also an issue with a dependency (openmm) in AlphaFold itself that we spent a few days trying to resolve. Eventually, we saw that someone else had raised an issue on the AlphaFold repository about this and it was fixed by the AlphaFold team after a few days. The most time-consuming issue was that Alpa is incompatible with Jax 0.4.0 and above, and AlphaFold was running on a Jax version >= 0.4.0. Although we explicitly installed Jax 0.3.5 via pip, AlphaFold was permanently stuck on running with the latest Jax version (0.4.8). We tried several workarounds and spent days trying to solve these issues, but unfortunately, we weren't able to solve this challenge. This completely halted our progress in applying Alpa to the prediction stage of AlphaFold. In the future, when Alpa is compatible with newer dependency versions, we hope to test our Alpa code and its performance.

6 Conclusion

In this report, we detail our progress toward parallelizing the AlphaFold prediction pipeline. We provide a runtime analysis of the MSA construction step. We detail our processor-level parallelization of the MSA construction and analyze its performance in relation to our runtime analysis. We describe our progress, challenges, and efforts toward parallelizing the inference step using the Alpa Python library.

In the future, we think that it would be interesting to see the performance of MSA construction in a distributed setting. For example, we could also use Ray to distribute computations to multiple machines in order to increase performance.

References

[1] Alpa Contributors. Alpa: A system for training and serving large-scale neural networks. `https://github.com/alpa-projects/alpa`, 2021. Accessed: 2022-05-13.

[2] Bozitao Zhong, Xiaoming Su, Minhua Wen, Sicheng Zuo, Liang Hong, and James Lin. Parafold: Paralleling alphafold for large-scale predictions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '21. IEEE, 2021.