BJYZ Team Design Document

Part 1: Problem Formulation

Read mapping is a fundamental task in computational biology, relevant to reference-guided assembly, structural variant calling, single nucleotide variant calling, and gene expression analysis. The reference genome contains millions to billions of nucleotides, while individual reads are typically between 75 and 300 nucleotides in length. The goal of read mapping is to determine the start and end positions of each read within the reference genome. Our team will develop an ultrafast, memory-memory efficient, and accurate genome-scale short read mapper from the ground up. Specifically, we integrated hash tables, seed-and-extend and banded Smith-waterman algorithms to match reads to the reference genome efficiently.

Input:

- File 1: A fasta (https://en.wikipedia.org/wiki/FASTA_format) file containing a reference genome with millions to billions of nucleotides
- File 2: A fastq (https://en.wikipedia.org/wiki/FASTQ_format) file containing millions to billions of 75 nucleotides to 300 nucleotides reads

Output:

- SAM (https://en.wikipedia.org/wiki/SAM_(file_format)) formatted file containing the coordinates of the mapping location on File 1 of each of read in File 2

Metrics:

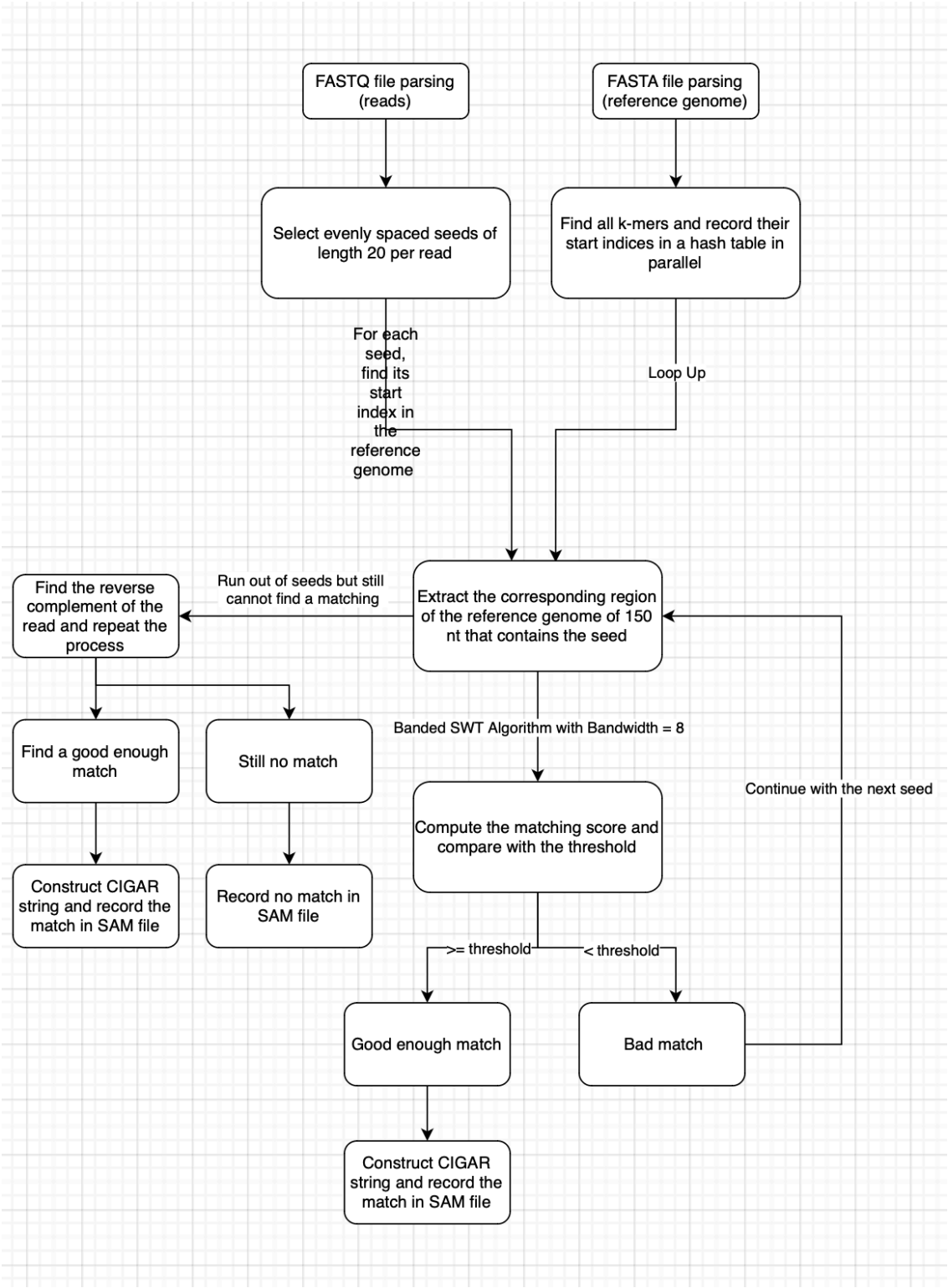
- False Positive (FP): mismapped read (read start and end coordinates are off by more than 5 nucleotides)
  False Negative (FN): unmapped read that has a valid mapping
  True Positive (TP): correctly mapped read (read start and end coordinates are within 5 nt of the truth interval)
  True Negative (TN): unmapped read that does not have a valid mapping
  Recall: TP/(TP+FN)
  Precision: TP/(TP+FP)

Runtime constraints (while maintaining performance metrics: 90% precision and 90% recall):

- Midterm: 1,000 reads per minute (128GB RAM)
- Bronze: 10,000 reads per minute (64GB RAM)
- Silver: 100,000 reads per minute (32GB RAM)
- Gold: 1,000,000 reads per minute (16GB RAM)
- Platinum: 10,000,000 reads per minute (8GB RAM)

Part 2: SWD Levels (UML)

Architectural Design



Modules

```
BJYZ/
├── src/ # the short read mapper algorithm
| ├── fasta_reader.py # Reader class for reference genome
| ├── fastq_reader.py # Reader class for reads
| ├── sam_writer.py # Writer class for SAM Files
| ├── substring_index.py # Indexing + Banded SWT algorithm
| ├── read_mapper.py # Project main entry point
├── tests/ # the short read mapper algorithm
| ├── test_fasta_reader.py # Unit tests for fasta_reader
| ├── test_fastq_reader.py # Unit tests for fastq_reader
| ├── test_sam_writer.py # Unit tests for sam_writer
| ├── test_substring_index.py # Unit tests for substring_index algorithm
| ├── test_index_parallel.py # Unit tests for parallel version, tells the number of
                              reads per minute, precision, recall, memory, F1 etc.
| └── file/ # Short examples of input files
| ├── example.fasta # example fasta file
| └── example.fastq # example fastq file
├── data/ # Folder containing input file
| ├── fasta files
| └── fastq files
├── doc/
| ├── BJYZ Design Document.pdf # Design document
| ├── BJYZ Final Presentation.pdf # Final presentation slides
| ├── BJYZ Midterm Presentation.pdf # Midterm Presentation slides
| ├── final_result.png # Summary of final metrics
| ├── midterm_result.png # Summary of midterm metrics
| ├── roles.png # Summary of role assignment
| └── workflow.png # UML file
├── test.slurm # slurm for testing
├── test-result.out # slurm output for testing that contains the number of reads
├── generate-sam.slurm # slurm for generating sam file for challenging dataset 1
├── requirements.txt # Python dependencies
└── README.md # Description of project
```

Libraries

- Biopython: parses input fasta and fastq files
- Psutils: retrieves information on system utilization (CPU, memory, etc)

Part 3: Role Assignment

We have roughly divided the semester into four phases. In each phase, every team member will have a specific role to fulfill. Roles will rotate with each new phase, giving everyone the opportunity to work on all aspects of the project.

	Phase 1: **Sept. 17th** **→ Oct. 8th**	**Phase 2:** **Oct 8th** **→ Oct. 29th**	**Phase 3:** **Oct. 29th** **→ Nov. 19th**	**Phase 4:** **Nov. 19th** **→ Nov. 26th**
Project Manager	Bowen Yao	Zikang Chen	Yuqi Chen	Jingwu Wang
Documentation lead	Jingwu Wang	Bowen Yao	Zikang Chen	Yuqi Chen
Testing lead	Yuqi Chen	Jingwu Wang	Bowen Yao	Zikang Chen
I/O & Orchestration lead	Zikang Chen	Yuqi Chen	Jingwu Wang	Bowen Yao

Each role has specific responsibilities listed below:

- **Project Manager**: Oversees the overall project, keeps track of deadlines, sets specific weekly goals, and monitors progress; coordinates discussions on short read mapper algorithms.
- **Documentation Lead**: Writes the design document and README file in a timely manner; comments on important parts of the code to improve readability.
- **Testing Lead**: Develops automated test cases; thoroughly tests the code and reports any bugs.
- **I/O & Orchestration Lead**: Imports FASTA and FASTQ files and parses them; exports results in BAM format; devises a modular architecture to ensure individual team members can work independently; ensures that runtime constraints are met.

Here are the gantt charts that track our progress made before the midterm and before the final.

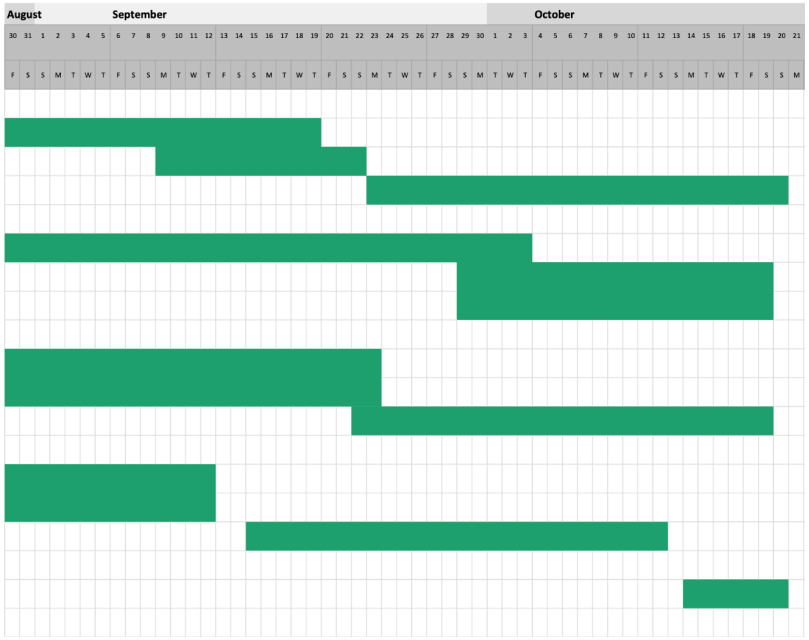
Midterm

Legend: On track | Low risk | Med risk | High risk | Unassigned

Project start date: 8/25/2024

Scrolling increment: 5

Milestone description	Category	Assigned to	Progress	Start	Days
Algorithm					
Hashing	On Track	Bowen Yao	100%	8/30/2024	21
Suffix array	On Track	Bowen Yao	100%	9/9/2024	14
SWT	On Track	Zikang Chen	100%	9/23/2024	28
Documentation					
Code comments	On Track	Jingwu Wang	100%	8/30/2024	35
Design Doc	On Track	Bowen Yao	100%	9/29/2024	21
Readme	On Track	Bowen Yao	100%	9/29/2024	21
Testing					
Testing fasta, fastq readers	On Track	Yuqi Chen	100%	8/30/2024	25
Testing output sam files	On Track	Yuqi Chen	100%	8/30/2024	25
Testing performance	On Track	Jingwu Wang	100%	9/22/2024	28
I/O					
Writing fasta readers	On Track	Zikang Chen	100%	8/30/2024	14
Writing fastq readers	On Track	Zikang Chen	100%	8/30/2024	14
Writing sam output	On Track	Yuqi Chen	100%	9/15/2024	28
Presentation Preparation					
Preparing slides	On Track	All	100%	10/14/2024	7



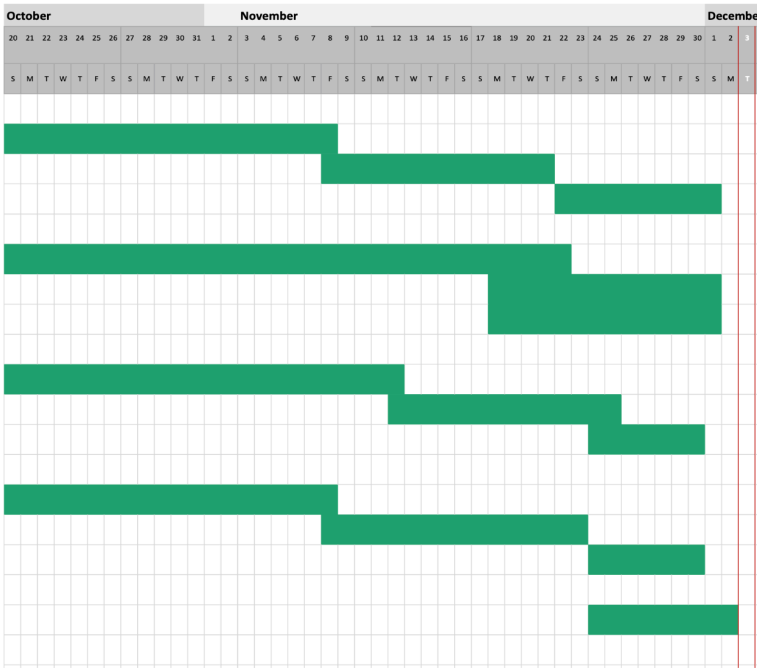
Final Project

Legend: On track | Low risk | Med risk | High risk | Unassigned

Project start date: 10/19/2024

Scrolling increment: 1

Milestone description	Category	Assigned to	Progress	Start	Days
Algorithm Optimization					
Implementing the parallel design	On Track	Zikang Chen	100%	10/19/2024	21
banned SWT	On Track	Yuqi Chen	100%	11/8/2024	14
fine-tune parameters	On Track	Jingwu Wang	100%	11/22/2024	10
Documentation					
Code comments	On Track	Bowen Yao	100%	10/19/2024	35
Design Doc	On Track	Zikang Chen	100%	11/18/2024	14
Readme	On Track	Yuqi Chen	100%	11/18/2024	14
Testing					
Testing performance	On Track	Jingwu Wang	100%	10/19/2024	25
Testing format of sam files	On Track	Bowen Yao	100%	11/12/2024	14
Testing on NOTs	On Track	Zikang Chen	100%	11/24/2024	7
I/O					
Writing CIGAR string	On Track	Yuqi Chen	100%	10/19/2024	21
Writing other fields	On Track	Jingwu Wang	100%	11/8/2024	16
SAM file formats	On Track	Bowen Yao	100%	11/24/2024	7
Presentation Preparation					
Preparing slides	On Track	All	100%	11/24/2024	9



Part 4: Planning

Midterm Goal:

Develop a functional version of the short-read mapper as soon as possible.

Algorithm Development:

- Scoring Table Research: Investigate how to define a good alignment.
- Determine the Best Alignment Method: Implement the exact matching algorithm, which is a midterm requirement.
    - Explore different approaches such as substring matching and dynamic programming (DP).
- Research Other Matching Algorithms:
    - Hashing techniques
    - Suffix Array/Tree
    - Burrows-Wheeler Transform (BWT)
    - Smith-Waterman (SW)
    - (Additional algorithms can be explored)

I/O Implementation:

- File Formats: Understand key I/O formats such as FASTA, FASTQ, SAM, and BAT.
- Libraries: Identify useful libraries to parse input files and generate output files.
- I/O Integration: Implement I/O operations and integrate them with the algorithm.

Testing:

- Ground Truth: Familiarize yourself with the expected results from the ground truth table.
- Implementation Testing: Verify the correctness of the implementation.
- Performance Metrics:
    - Understand and track Precision, Recall, and Run Time.
    - Build and evaluate performance metrics.
- Performance Testing: Test the overall performance of the implementation.

Documentation:

- Design Document: Update the design document on a weekly basis.
- README: Revise the README file whenever new progress is made.
- Code Comments: Ensure the code is well-commented throughout development.

Study Material:

- Bowtie2 Intro
- Bowtie2 Manual
- Watch YouTube tutorials to better understand FASTA, FASTQ, SAM, and BAT file formats.

Questions:

- Will we be provided the API for metrics?

- Phred Score
- Example of how to measure a good alignment
    - Some threshold for good alignment, How do metrics come into play here?

We have explored two algorithms:

- **Suffix Array:** The time complexity of this algorithm is O(m log n) and takes several minutes to match 1,000 reads. Additionally, it is memory intensive and takes O(n) space.
- **Smith-Waterman Algorithm:** This algorithm generates the most accurate matching, but the time complexity is O(nm) and takes more than an hour to match 1,000 reads.

Midterm algorithm:
Our current algorithm integrates a hash table, the seed-and-extend approach, and the Smith-Waterman algorithm. It achieves reasonable efficiency, taking approximately four seconds to match 1,000 reads.

For the reference genome, we determine the value of k—set to 30 in this case for optimal performance. We then iterate through all k-mers in the reference genome, storing their starting indices in a hash table.

For each read, we fine-tune the parameters by setting the seed length to k and selecting 30 evenly spaced seeds per read. Each seed is mapped to the reference genome using the pre-built hash table. Next, we extract a corresponding region of 150 base pairs from the reference genome that contains the seed. The Smith-Waterman algorithm is then applied to this region and the read to compute an alignment score. If the score exceeds the early exit threshold (set to 0.3 * the read length), we terminate the loop and assign this start index as the alignment for the read. Otherwise, we move on to the next seed. If no match is found after examining all seeds, we repeat the process for the reverse complement of the read.

We have found that tuning the algorithm's parameters greatly impacts its performance in terms of speed and match quality:

- **k value:** A larger k results in faster execution but reduces the likelihood of finding matches.
- **Number of seeds:** Increasing the number of seeds per read slows the algorithm due to more comparisons but improves the chances of finding matches.
- **Score threshold:** Raising the threshold improves the accuracy of the match between the read and the reference genome section.

Final Goal: Achieve the silver goal (100,000 reads per minute).

Algorithm Development:

- Implemented banded Smith-Waterman algorithm.
- Added parallelism for the construction of the hash tables.
- Fine-tuned the parameters used in the algorithm to improve performance.
- Tried to consider the reverse complements of the reads as normal reads but that decreased the accuracy of matches
- Used profiling tools to identify the functions of the code that consumed most time and tried to reduce the number of calls.

I/O Implementation:

- Add the construction of CIGAR strings to our code.
- Complete all the required fields of the sam format and export to an output file.
- Explore the possibility of parallelizing I/O.

Testing:

- Understand how to use slurm files to submit jobs to NOTS.
- Understand the various compute nodes on NOTS and which one would be more suitable for our job.
- Test the final code on NOTS and record its metrics like number of reads, precision, recall, f1 score, memory usage etc and compare with the midterm metrics.
- Compare the sam output file with the reference sam file to make sure that our implementation is correct.

Documentation:

- Design Document: Update the design document based on the midterm feedback. Like adding limitations/further development/challenges and a conclusion. Update the gantt chart to make it more readable.
- README: Revise the README file based on the final code.
- Code Comments: Double check the code is well-commented.

Study Material:

- Algorithm Ideas
- I/O helper

Questions:

- Why is performance on our local machine better than on NOTS?
- Can we use dummy values for some of the fields in the SAM output file?
- How will our final implementation be evaluated?

Final algorithm:

1. Preprocessing the Reference Genome

Goal: Speed up lookups for aligning incoming reads by indexing all k-length substrings (k-mers) of the reference genome.

We chose k = 20. The exact size depends on desired specificity: larger k reduces false matches but might miss variants, while smaller k might yield too many candidates. Then, for the given reference genome G, extract every k-length substring and record its starting index. Store these mappings in a hash table, where keys are k-mer strings and values are lists of all start indices in G where that k-mer appears.

2. Processing the Read and Seed Generation
Goal: Use seeds of the read to quickly locate potential alignment regions in the reference genome.

Given a short read R of length |R|, define a seed length k (the same k used for indexing the reference). The read is divided into a series of seeds, S_1, S_2, …, S_l. We choose l = |R| / 2 when using k=20.

3. Retrieving Potential Candidate Locations
Goal: Identify where in the reference genome these seeds appear to narrow down potential alignment targets.

For each seed S_i, use the precomputed hash table from Step 1 to find all reference start indices H(S_i) = {l_{i1}, l_{i2}, …} where that seed occurs. Next, we extract a corresponding region of 150 base pairs from the reference genome that contains the seed.

4. Applying the Banded Smith-Waterman Alignment
Goal: Efficiently compute the optimal local alignment between the read and each candidate reference substring.

A full Smith-Waterman alignment compares every cell of a 2D dynamic programming matrix, which can be computationally expensive. To optimize, we restricted the alignment computation to a diagonal band around the main diagonal of the DP matrix. This reduces computational complexity significantly. For each candidate region G_c, run the banded SW algorithm with the read R. The algorithm yields an alignment score A(l_{ij}), which measures how well R aligns to that portion of G.

5. Scoring and Early Exit Strategy
Goal: Quickly decide when a good enough alignment is found and avoid unnecessary computation.

Define a threshold score: $\tau = 0.1 \times |R|$ This means if the alignment score A(l_{ij}) for a candidate region exceeds 10% of the read length (or another chosen proportion), we consider this candidate sufficiently good to stop searching further. This threshold helps save time: as soon as a reasonably good alignment is discovered, the mapper exits early and reports this alignment.

If the current candidate region does not surpass τ, move on to the next candidate reference start index. Keep checking until either: A suitable candidate is found (score > τ), or all candidates have been exhausted without finding a good alignment.

6. Reverse Complement Case
Goal: Handle reads that map to the opposite strand of the reference.

If no candidate region yields a score above τ, compute the reverse complement of the read R and repeat steps starting from the seeding process (Step 2) with the reversed read. If this still yields no good alignments, the read may be considered unmapped.

7. Constructing the Final Alignment Output
Goal: Once a good alignment is found, finalize the reporting.

After identifying the best scoring alignment (either in forward or reverse-complement orientation), trace back through the Smith-Waterman DP matrix to reconstruct the alignment operations. These operations are often summarized in a CIGAR string. If multiple candidate regions produce good scores, the one with the highest score is chosen.

Besides the three parameters we had in the midterm, we now have a new parameter for our final implementation: the bandwidth of the banded SWT algorithm. We observe that when we have a smaller number for bandwidth, the number of reads will increase significantly, but the accuracy will be compromised as well. Finally, we chose a bandwidth of 8.

Final Testing:

We have developed comprehensive testing for our implementation.

- ReadFasta and ReadFastq:
    - Test on empty file
    - Test incorrect input path
    - Test on malformed file
    - Test on a simple correct file
- SAMWriter:
    - Validate the SAM file output
    - Check correct header
    - Check correct mapping
- Comparison of Expected Start Position
    - For each read in the dataset, compare the matched indices returned by the mapping algorithm with the corresponding expected start and end positions from the ground truth table.
- Performance Metrics:
    - Wall Clock Time
    - CPU Time
    - Memory Usage
    - Number of reads per minute
    - Precision
    - Recall
    - F1 Score

Part 5: Results

Midterm Metrics For Test Dataset 1:

Wall Clock Time (s)	CPU Time (s)	Reads Per Minute	Memory Usage (MB)
4.30	4.27	14,019	41.91

Break-down:

True Positive	False Positive	True Negative	False Negative
811	0	188	1

Precision	Recall	F1 Score
1.00	1.00	1

Midterm Metrics For Test Dataset 2:

Wall Clock Time (s)	CPU Time (s)	Reads Per Minute	Memory Usage (MB)
4.27	4.24	14,151	40.58

Break-down:

True Positive	False Positive	True Negative	False Negative
799	0	188	13

Precision	Recall	F1 Score
1.00	0.98	0.99

Final Metrics For Challenging Test Dataset 1:

Wall Clock Time (s)	CPU Time (s)	Reads Per Minute	Memory Usage (MB)
37.60	33.34	531,822	614.60

Break-down:

True Positive	False Positive	True Negative	False Negative
259,851	7,322	66,132	0

Correctness	Precision	Recall	F1 Score
97.8%	0.97	1.00	0.99

Final Metrics For Challenging Test Dataset 2:

Wall Clock Time (s)	CPU Time (s)	Reads Per Minute	Memory Usage (MB)
39.37	35.48	507,888	632.62

Break-down:

True Positive	False Positive	True Negative	False Negative
258,145	8,856	66,199	105

Correctness	Precision	Recall	F1 Score
97.31%	0.97	1.00	0.98

Part 6: Reflections

Challenges:

- One of the biggest challenges was using NOTS. Initially, we found out that running the same program on NOTS would actually take more time than running on our local machines. We were very confused until we scheduled a meeting with a specialist at the center of computing. It turned out that there were many compute notes on NOTS and some of them were more powerful than the others. In order to maximize performance, we need to specify we wanted to use the most powerful node.
- Another challenge was to output the SAM files. Since we did not decide to use any external packages to write to SAM files, we needed to make sure that our program write everything in the correct format, which took us some time to test and debug.

Milestones:

- We were able to achieve ~14,000 number of reads per minute at the midterm.
- We were able to achieve ~500,000 number of reads per minute at the final.

Limitations:

- The parameters in the algorithm are fine-tuned towards the challenging datasets that were provided. If another dataset is used for testing, those parameters might not be the best ones in terms of accuracy and performance.

Further Development

- Inspired by other groups, we will try to use Cython to compile the python code into C code to reduce the run time.
- We will also look into how to utilize the GPUs on NOTS for further acceleration.

Part 7: Conclusion

In summary, we developed an ultrafast and memory efficient short read mapper over the semester. It integrated hash tables, seed-and-extend and banded Smith-waterman algorithms to match reads to the reference genome efficiently. Ultimately, we were able to achieve ~500,000 number of reads per minute while maintaining precision and recall around 0.98. We are proud of our achievements and want to express our gratitude to Dr. Todd J Treangen, Dr. Fritz Sedlazeck, and Natalie Kokroko.

Part 8: References

- Alser, M., Rotman, J., Deshpande, D., Taraszka, K., Shi, H., Baykal, P. I., Yang, H. T., Xue, V., Knyazev, S., Singer, B. D., Balliu, B., Koslicki, D., Skums, P., Zelikovsky, A., Alkan, C., Mutlu, O., & Mangul, S. (2021). Technology dictates algorithms: Recent

developments in read alignment. *Genome Biology*, *22*(1). https://doi.org/10.1186/s13059-021-02443-7

- Y. -L. Liao, Y. -C. Li, N. -C. Chen and Y. -C. Lu, "Adaptively Banded Smith-Waterman Algorithm for Long Reads and Its Hardware Accelerator," 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP), Milan, Italy, 2018, pp. 1-9, doi: 10.1109/ASAP.2018.8445105.

- Treangen, T. J., & Salzberg, S. L. (2011). Repetitive DNA and next-generation sequencing: computational challenges and solutions. *Nature reviews. Genetics*, *13*(1), 36–46. https://doi.org/10.1038/nrg3117