# Multiple Sequence Alignment with Speculative Parallelism

Leon Xie[1], Taekseung Kim[1]

[1] School of Computer Science, Carnegie Mellon University

**Abstract** :

We implemented a speculative parallel version of the randomized iterative Berger-Munson algorithm for multiple sequence alignment, using C++ and OpenMPI. We achieved near-linear speedup on some test cases, and encountered inherent speedup limitations on others. We determined two main limitations: (1) a theoretical upper bound on speedup based on input and randomness, and (2) divergence between processors during parallel steps.

**Index Terms:** multiple sequence alignment, speculative computation, computational biology, parallelism.

## 1 Background

Multiple sequence alignment (MSA) involves arranging biological sequences — DNA, RNA, or proteins — to identify regions of similarity that may indicate functional, structural, or evolutionary relationships. The computational challenge emerges from the need to optimally insert gaps within sequences to maximize alignment scores based on substitution matrices and gap penalties Wikipedia contributors, n.d. For pairwise sequence alignment, dynamic programming algorithms like Needleman-Wunsch and Smith-Waterman construct a scoring matrix where each cell $M(i, j)$ represents the optimal alignment score between prefixes of length $i$ and $j$ from the respective sequences. The recurrence relation typically considers three possible moves from cell to cell — match/mismatch, vertical gap, and horizontal gap —, and takes the best possible move for each cell.

$$M(i,j) = \max \begin{cases} M(i-1, j-1) + s(x_i, y_j), \\ M(i-1, j) + g, \\ M(i, j-1) + g \end{cases}$$

Here, $s(x_i, y_j)$ represents the substitution score and $g$ the gap penalty. The optimal alignment is reconstructed by backtracking from $M(i, j)$ to $M(0, 0)$. Diagonal moves represent matches and mismatches, while vertical and horizontal moves represent gaps.

While this solution works for two sequences, we focus on the case where we have $n > 3$ sequences. Multiple sequence alignment is much more complex compared to two sequence alignment. If we apply a similar dynamic programming solution multiple, we end up with time and space complexity of $O(L^n)$, where $L$ is the sequence length.

Berger et al. (1991) address this limitation with a randomized, iterative algorithm. Their approach begins with an initial alignment and proceeds through repeated random partitioning of the sequences into two groups. These groups are then realigned against each other while maintaining internal gap positions, creating a new candidate alignment. If this candidate improves the overall score, it replaces the current alignment; otherwise, it is rejected. This process iterates until we reach a specified stopping criteria, effectively sampling the vast solution space without exhaustive exploration.

In this algorithm, they key data structure is the current best sequence alignment, represented by a vector of strings. After an iteration, a gap-positions data structure represents the changes made to the starting alignment.

In each iteration, the key operations include randomly partitioning the alignment, aligning between the partitions, checking if the score is better, and updating the alignment with new gaps if the score is better.

The input to this algorithm is a list of sequences, and the output is a list of sequences with gaps inserted, representing an alignment. The opportunity for parallelism is twofold: we could parallelize each iteration of the algorithm, and we could parallelize between iterations. We focus on the latter.

Initially, parallelization between iterations seems hopeless, since there is a sequential dependency pattern between iterations. When an alignment is accepted, the next iteration uses this alignment as a starting point. However, the opportunity for parallelism between iterations comes from one key observation — each iteration only depends on the previously accepted iteration, not necessarily the iteration right before it.

For instance, imagine that iteration 1 accepts, iterations 2 and 3 reject, and iteration 4 accepts. Iteration 4 only depended on the data from iteration 1. So, we could conceivably compute iterations 2, 3, and 4 in parallel.

Note that the amount of parallelism is limited by the length of rejection-chains, where many iterations all reject in a row.

Yap et al. (1996) use this idea (called speculative computation) to parallelize Berger-Munson. In their algorithm, multiple processors simultaneously explore different potential alignments, each representing an potential iteration the sequential algorithm. Let $p$ be the number of processors. In one parallel step, the processors compute possible iterations $i$ through $i + p - 1$ in parallel. This is visually represented in figure 1. Numbers in the boxes represent iterative algorithm step numbers. There are 28 iterative steps, completed in 13 parallel steps. Lines between parallel steps represent data being broadcast.

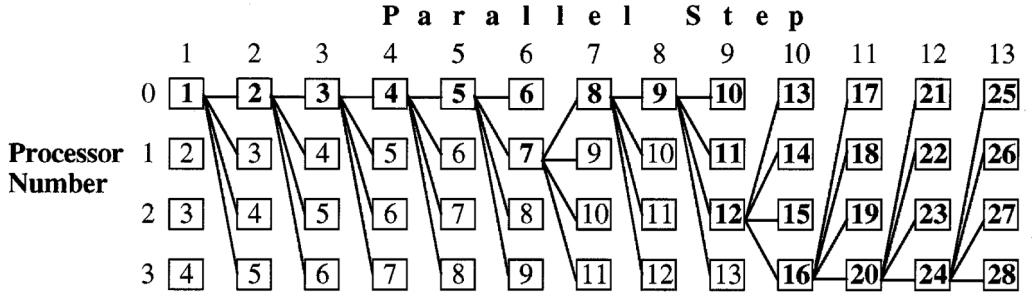


**Figure 1.** Mapping of algorithm components to parallel hardware

Each processor $P_0 ... P_{p-1}$ speculates that all previous processors will reject. If $P_0 ... P_{k-1}$ all reject, and $P_k$ accepts, we have effectively computed iterative steps $i$ through $i+k-1$ in parallel, and can move onto iteration $i+k$. $P_k$'s data is to initialize the next round. However, $P_{k+1} ... P_{p-1}$ will have wasted work, with their results thrown out.

This approach overcomes the inherently sequential nature of the original algorithm by speculatively computing multiple alignment possibilities in parallel, significantly accelerating runtime and while preserving solution quality.

## 2 Approach

We implemented the algorithm described above from scratch, using C++ and OpenMPI to support a message passing model. We targeted GHC machines equipped with Intel i7-9700 CPUs, each featuring 8 cores. This hardware choice provided a balanced environment for our parallel algorithm implementation, but also has limits because it is a shared resource, and runtime varied depending on how much resource is left on the machine.

### *2.1 Problem Mapping to Parallel Architecture*

Although Berger-Munson is an iterative algorithm, we were able to use the speculative approach described above to map the algorithm to multiple cores. Each core is responsible for computing a complete iterative step of the algorithm. This one-to-one mapping between iterations and cores allows for efficient parallel execution.

### *2.2 Data Structure and Operation Mapping*

Each core maintains a copy of the alignment and computes an independent iterative step on it.

Communication between cores occurs primarily between parallel steps, where some processor has discovered a better alignment. When this happens, the processor needs to broadcast the gap positions of the new alignment to the other processors.

Data sharing is done through MPI's communication primitives

This mapping utilizes the full processing capability of each core while there are continuous rejects, making the calculation step parallel. For example, in Figure 1, the sequential algorithm would take 3 iterative steps to compute iteration 10 (reject), iteration 11 (reject), and iteration 12 (accept). The parallel algorithm is able to do these in one parallel step (parallel step 9).

### *2.3 Algorithm Modifications*

We did not need to modify the sequential algorithm. One of the key advantages of our approach is that we maintained algorithm correctness without modifying the underlying serial algorithm. This is one of the primary benefits of the speculative Berger-Munson technique - it naturally lends itself to parallelization without requiring fundamental changes to the algorithm's logic or output. The parallel implementation produces results identical to the serial version, ensuring correctness while gaining performance benefits.

### *2.4 Optimization Process*

After initially running our code on some inputs, we discovered that for some inputs, we fell short of optimal (linear) speedup. To address this, we engaged in performance debugging. We discovered that our main limitations were twofold: an inherent theoretical limit on speedup, and divergence of processors during parallel steps.

**2.4.1 Theoretical Limit on Speedup** First, we discovered that most of the performance limitations encountered appear to be inherent to the algorithm itself. As described earlier, when we have long chains of rejects, we have good parallelism. But, if a certain run happens to produce many accepts (relative to the total length), our parallelism is limited. We found experimentally that this limitation may severely decrease possible speedup, to a theoretical maximum of as low as 2.5 for 8 processors.

For example, in figure 1, we see 28 sequential steps completed in 13 parallel steps. In this case, the theoretical maximum on speedup is $\frac{28}{13} = 2.15$, which is less than the linear ideal of 4. This limitation is implementation-independent. We define the theoretical limit on speedup as $\frac{s_I}{s_P}$, where $s_P$ is the number of iterations, and $s_P$ is the number of parallel steps.

More data on this is presented in our results section.

When testing several inputs, we found that generally inputs with less sequences had a worse limitation on potential speedup. Our termination criteria is dependent on the number of sequences in the input. Smaller sequences will terminate much quicker. Additionally,

in the beginning of a run, we tend to see more accepts, since we start with a naiive alignment. As we proceeds, accepts become more sparse. If we terminate early, we have proportionally more accepts interspersed with our reject-chains, reducing our parallelism.

**2.4.2 Practical Speedup Limitations (Divergence)** We engaged in additional performance to identify and understand the gap between theoretical possible speedup and actual speedup observed. In the end, we determined **divergence** to the primary bottleneck.

This involved:

1. Profiling the execution to identify bottlenecks. This included broadcast, reduction, extra steps in the parallel algorithm, and divergence.
2. Analyzing the tendency based on the input size (number of sequences, length of sequences)
3. Measuring load balance across processing units, by printing out the time for each processors for one parallel step. This helped us identify divergence as a primary limitation.

One simple limitation is the initialization time of OpenMPI. For inputs with minimal wall-clock time, this has a significant impact on speedup.

We began with identifying several possible bottlenecks. These were:

- MPI collective communication routines (broadcast, reduction)
- Extra steps for parallel algorithm (data copying and processing from accepted processor to other processors)
- Divergence between processors during each parallel step

We manually instrumented our code with timing code to get the runtime of each step in order to identify bottlenecks.

Our initial hypothesis was that broadcast would be the most expensive. Since we are broadcasting data equal to the sequence length in each parallel step, we thought the movement of this data would be expensive. However after timing the code, we found that the broadcast time was negligible (less than 0.04% of total execution time).

Next, we noticed that for some iterations, all-reduce was seemingly taking a long time. We thought this was strange, since in our use case, the amount of data being reduced is small and constant (we use all-reduce to check if any processor has accepted — this only involves a flag and processor ID). Additionally, the time for all-reduce was varying strangely relative to the input size — the only tendency we were able to discern is that all-reduce time increases with total runtime.

It turns out that although we put timer code right before and after the call to all-reduce, we were not observing the effect of all-reduce itself, but rather observing the effect of **divergence**. We'll return to this shortly.

Next, we considered the overhead of parallelization. Initialization of MPI takes about 0.24 seconds. There are also some extra steps in the parallel algorithm. After the accepted processor broadcasts its partition and gap-positions, some processing by the other processors is done to construct an alignment from this. We measured the time taken for this, which turned out to be negligible, similar to the time of broadcast.

Finally, we considered the effect of divergence between the processors. During any parallel step, each processor needs to perform an alignment between two groups of sequences — this is an expensive computation. However, due to both randomness, and problem size differences, this computation may take differing amounts of time for each processor. The runtime of alignment is determined by the lengths of the two groups being aligned. As a pre-processing step, "global" (gaps that exist across a group of the partition) are removed, changing

the groups length. The partition for each processor differs, which means the group lengths may differ, so the runtime of alignment may differ. Additionally, there is always some natural random variation in computational runtime.

We figured this may be causing a slowdown, so we profiled the time taken to align sequences for one parallel step of a run. This data is presented below, in the results section. In summary, we found that the fastest processor (of 8) took 0.182 seconds to do alignment, while the slowest took 0.224 seconds. This represents a significant (about 20%) divergence.

We then realized that our earlier measurements for all-reduce were misrepresentative. In our code, we have no explicit barriers. So, all-reduce also acts as an implicit barrier. So processors that finish alignment, and proceed to all-reduce, need to wait for all other processors to finish their alignments before reduction can be done. In short, we were not only measuring the time for all-reduce, but also the divergence.

We then re-measured the all-reduce step, isolating it from the divergence measurement. We did this by adding a explicit MPI barrier right before the reduce, and only began our timer after the barrier. This way, the timer only begins after all processors finish their alignment, and is truly capturing the time of the reduction step only. The results from this were that all-reduce took negligble time — much less than even broadcast, measured above. This led us to the conlusion that divergence was the dominant bottleneck.

We did not see an easy way to reduce divergence between processors. The methods studied in this class involved better load balancing, but in this case each processor only has one task, with varying runtimes that are somewhat random.

### *2.5 Code Base*

The implementation was built from scratch, without relying on existing codebases. This approach allowed us to keep the implementation specific to the characteristics of speculative Berger-Munson and our target hardware, avoiding any constraints or overhead that might have been present in pre-existing implementations.

## 3 Results

We used the BAliBASE (Thompson et al. (2005)), a MSA benchmarking dataset, to run our experiments and collect data.

To maintain consistency of results, we used pseudorandomness in our benchmarks instead of true randomness. This way, we can be sure that any effects observed are due to changes in processor count, and not due to randomness during partitioning. Additionally, this allows us to guarantee correctness compared to the sequential algorithm.

### *3.1 Performance Measurement*

To measure performance, we first recorded the wall-clock time. Then, we calculated the speedup relative to the sequential algorithm.

Finally, we define **speedup effectiveness** as the ratio of the observed speedup to the theoretical speedup, as discussed earlier. Essentially, given the theoretical speedup constraint, how close did our speedup get. A perfect speedup effectiveness would be 1, or 100%.

Table 1 displays theoretical speedup against the number of processors, and Table 2 is wall-clock time based on the number of processor and dataset.

| Number of Processors | Number of Iterations | Number of Parallel Steps | Theoretical Speedup |
|---|---|---|---|
| 1 | 2652 | N/A | 1 |
| 2 | 2653 | 1342 | 1.977 |
| 4 | 2655 | 689 | 3.853 |
| 8 | 2655 | 359 | 7.395 |

**Table 1**
Theoretical speedup of BB12007 (few very-long sequences).

| Test ID | Number of Processors | Wall-Clock Time |
|---|---|---|
| BBS30009 | 1 | 45.50 |
| BBS30009 | 8 | 6.82 |
| BB12007 | 1 | 54.72 |
| BB12007 | 8 | 24.53 |

**Table 2**
Wall-clock time

| Number of Processors | Speedup | Theoretical Speedup | Speedup Effectivness |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 2 | 1.528 | 1.615 | 0.945 |
| 4 | 2.034 | 2.274 | 0.895 |
| 8 | 2.231 | 2.907 | 0.767 |

**Table 3**
Speedup Effectiveness for input BB12007 (few very long sequences). Notice that although the absolute speedup seems poor, the speedup effectiveness is good.

### *3.2 Size of the inputs*

Table 4 shows the the problem size, where BBS30009 indicates a medium number of short sequences. BB12007 indicates a small number of very-long sequences. We chose these inputs because one has a good speedup with good theoretical speedup, and other one has poor absolute speedup, but good speedup considering theoretical speedup, where theoretical speedup is not linearly increasing with the number of processors.

### *3.3 Speedup Graphs and Configuration*

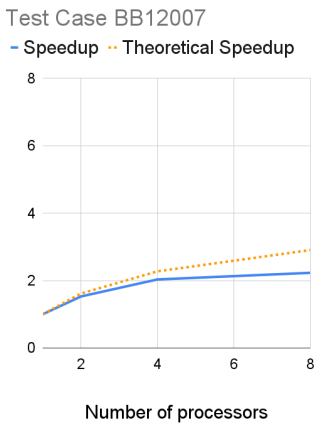


**Figure 2.** Speedup Graph for BB12007, Long very, few

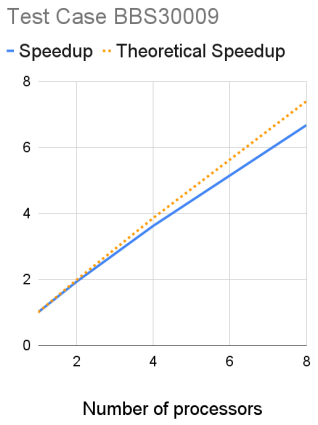


**Figure 3.** Speedup Graph for BBS30009, Med ium, many

We did our benchmark based on a single-threaded CPU, where we implement the sequential version of our algorithm. Figure 2 shows that speedup is not linear, but nearly close to theoretical speedup which is the optimal value. Figure 3 shows a dataset where speedup is

| Input ID | Number of Seqences | Average Sequence Length |
|---|---|---|
| BBS30009 | 38 | 74 |
| BB12007 | 8 | 1263 |

**Table 4**
Problem size

near linear, also close to theoretical speedup.

### *3.4 Importance of Problem Size*

The problem size (and shape) impacts runtime and speedup. Our input size has two dimensions: number of sequences, and average sequence length.

More sequences results in longer convergence times. Longer convergence time results in more long reject chains (since as we get near optimal we are more likely to reject). This would lead to better theoretical speedup.

Additionally, longer sequences lead to longer time per iteration, since the (non-parllel) alignment step would take longer. This results in a longer wall-clock runtime.

(As a side note, an interesting extension of this project would be to explore parallelizing within each iteration. That is, making the alignment step itself parallel. We briefly discussed this, and it turns out you can fill out the DP table in parallel by following a specific execution order based on the dependencies in the table. The dependencies are vertical and diagonal from top-left to bottom-right, but there are no dependencies across the other diagonal, from top-right to botom-left. Implementing this would yield a faster wall-clock time, as each iteration would be faster. However, it wouldn't impact speedup.)

### *3.5 Parallelism Analysis*

**3.5.1 Theoretical Limitations** As discussed earlier, there is a theoretical limit to our parallelism.

**Impact of Accepts** When there is an accept, we do not use the calculations that are done by processors with ID greater than the one that accepted. This introduces significant loss of parallelism, which is bad for our theoretical speedups, and caps our observed speedup.

**Impact of Problem Size** As we discussed before, after observing the wall-clock time and the speedup data, we learned that more sequences generally lead to better theoretical speedup. This can be seen in figures 2 and 3

**3.5.2 Implementation Limitations** Here we attempt to explain the gap between observed speedup and theoretical speedup (blue and orange lines in figures 2 and 3). Our speedup effectiveness is observed to decrease as processor count increases (figure 3).

**OpenMPI Collective Communication (Broadcast and Reduce)** We initially assumed that the broadcast and reduction steps would be the bottleneck, since this would be the communication cost. Since the data we broadcast is dependent on the length of sequences, we thought this would be the bottleneck. However, when we measured runtimes of differnet steps, the broadcast was negligible.

We discovered that there is significant divergence between processors, due to the difference between problem size. Empirically, we observed up to 20% divergence for a single parallel step (figure 4).

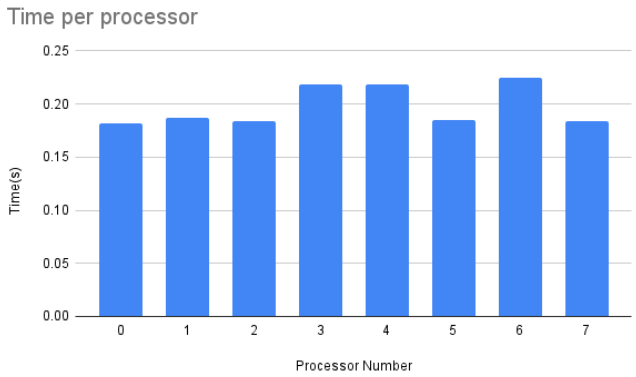


**Figure 4.** Runtime of alignment step between processors during one parallel step, on input BB12007. The fastest processor finished in 0.182 seconds, while the slowest took 0.224 seconds. This is a .042 second difference, which is about 20% of the runtime for this step. Processors 0, 1, 2,5, and 7 are waiting idly for processors 3, 4, and 6 to finish.

After adding a barrier before the all-reduce step, we observed that all-reduce only took 0.0004 seconds over the entire course of execution (of a total runtime of 24.53 seconds).

Due to this, we believe divergence is the main cause of inefficiency.

**Overhead of Copying Data** We copy the data when we are calculating the alignment, but it turned out to be insignificant (around as same as broadcast and reduction).

**Divergence** As we discussed in the previous section, divergence was taking a lot of time compared to other possible bottlenecks. Based on figure 4, we could see that time could differ to 20%.

## 4 Deeper Analysis (Percentage Profiling for BB12007)

For test case BB12007 (few very-long sequences), using 8 cores, we found that more than 90% of runtime was spent doing useful work (alignment). About 5% was lost due to divergence. Less than 1% was spent on broadcast and reduction. Less than 1% was spent on data copying and processing for the parallel implementation. About 1% was spent on MPI initialization. About 4% is unknown.

## 5 Choice of Target Machine

We chose OpenMPI as it maps to our algorithm most naturally. Each processor has its own data, and communicates with other processors.

We believe GHC was a good initial target, as it provided us with 8 cores. Multi-threading doesn't realy help us, since we would still need to do the same amount of work (two threads on one core will just take twice as long). So, it would have been interesting to see our performance on more cores (PSC) for instance.

This presents some interesting tradeoffs — accepts will be even worse for parallelism, since many more cores will have wasted work if an early core (say, $P_1$) accepts. On the other hand, we would parallelize long reject chains much better. Essentially, the effect of problem size on theoretical speedup is magnified.

## 6 Distribution of Work

Our split of the work was roughly 60% by Leon, and 40% by Taekseung.

Leon:

- Set up Git repository, along with Makefile and project structure.
- Wrote header files and function signatures.
- Integrated MSA code (Taekseung) into sequential version
- Implemented speculative parallel version
- Added timer code for profiling
- Ran experiments
- Analyzed parallel limitations

Taekseung:

- Research on MSA libraries
- Group-to-group MSA alignment implementation (sequential)
- Dataset management (determined length and number of sequences for each input, helped choose inputs to use)
- Wrote shell file for experiments
- Ran experiments
- Data analysis

Collective:

- Research into computational biology algorithms
- Pair programming
- Analyzed experimental results, both made important insights
- Make poster, practice presentation
- Write up report

## References

Berger, M. S. and D. L. Munson (1991). “A novel approach to multiple sequence alignment”. In: *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*. Dallas, TX, USA: IEEE, pp. 562–565.

Lawrence Livermore National Laboratory (n.d.). *MPI Tutorials*. `https://hpc-tutorials.llnl.gov/mpi/`. Accessed: 2025-04-30.

Pearson, W. R. and D. J. Lipman (1988). “FASTA: A fast algorithm for database search of protein and nucleic acid sequences”. In: *Proceedings of the National Academy of Sciences* 85.8, pp. 2444–2448.

Thompson, Julie D., Frédéric Plewniak, and Olivier Poch (2005). “BAliBASE 3.0: Latest developments of the multiple sequence alignment benchmark”. In: *Proteins: Structure, Function, and Bioinformatics* 61.1, pp. 127–136.

Wikipedia contributors (n.d.). *Multiple sequence alignment*. `https://en.wikipedia.org/wiki/Multiple_sequence_alignment`. Accessed: 2025-04-30.

Yap, Tieng K., Ophir Frieder, and Robert L. Martino (1996). “Parallel Computation in Biological Sequence Analysis”. In: *IEEE Transactions on Knowledge and Data Engineering* 8.4, pp. 533–539.