# Parallel Chunking

Yucheng Low

May 6, 2025

## 1 Content Defined Chunking

The basic principle of Content Defined Chunking (CDC) is to split a file into smaller chunks, produce variable sized-chunks, that are mostly stable to insertions and deletions. This is as opposed to fixed-size chunking where a file is split into constant sized chunks where insertions/deletions which are not exactly chunk aligned will cause all remaining chunks to be modified. Content defined chunking is excellent at data duplication as files with similar contents (for instance two neural network models stored as different file formats) can deduplicate against each other effectively.

We will not go too deep into implementation and usage of content defined chunking here. We recommend reading the GearHash paper, or our prior paper Git is for Data for a more detailed description. A challenge with CDC is that due to the variable block sizes, it is generally necessary to start chunking from the start of the file. However, this can be a performance issue with very large files in which the CDC procedure can be a bottleneck and that is what we will like to address in this paper.

Instead of assuming any particular chunking procedure, we will just rely on a simplified formal description. Given a determinstic rolling hash function $H(x) \to \{0, 1\}$ which accepts a short string of length $k$ and returns an boolean. Given a string $S$. The basic CDC procedure works as follows:

```
ChunkStart = 0
for $i$ in [0, $|S| - k$):
    if H(file[$i \ldots i + k$]) == 1:
        AddChunk(file[ChunkStart...$i$])
        ChunkStart = $i$
AddChunk(file[ChunkStart...])
```

A minimum ($m$) and maximum ($M$) chunk size is normally used to avoid too tiny and overly large chunks:

```
ChunkStart = 0
for $i$ in [0, $|S| - k$)):
    if ($i$ - ChunkStart $\geq M$) or (H(file[$i$...$i+k$]) == 1 and $i$ - ChunkStart $\geq m$):
        AddChunk(file[ChunkStart...$i$])
        ChunkStart = $i$
AddChunk(file[ChunkStart...])
```

(Note that this simplified formulation does not cover more advanced procedures like the the low variance chunking method in the Git Is for Data Paper, nor the adaptive chunking method in FastCDC).

The goal is to find conditions in which we can "seek" into an arbitrary location $|S|$ and find a set of chunks that are guaranteed to align with chunks produced by starting from the beginning of the file with the basic CDC procedure above.

## 2 Parallel Chunking

Consider two threads, first thread beginning chunking from $S[0]$ and second thread beginning chunking from an arbitrary location.

Under what condition will the chunks produced by thread 2 be gauranteed to line up with the chunks produced by thread 1?

Assume that second thread finds 3 consecutive chunk boundaries $c_0, c_1$ and $c_2$ with the following conditions:

1. The chunk sizes are not "near" the chunk size limits: $m < c_1 - c_0 \leq M - m$ and $m < c_2 - c_1 \leq M - m$.

2. There is no other chunk boundary within between $S[c_0...c_2]$. i.e. there is no substring $s$ of length $k$ in the range $S[c_0...c_2 + k]$ such that $H(s) = 1$. (Equivalently, the basic chunking procedure will find $c_1, c_2$ a if I start chunking from $c_1$ and set $m = 0$)

Then, we claim that first thread will align with the second thread on either $c_1$ or $c_2$ and so produce the same chunks after that.

### Proof

Let $b$ the first thread's final chunk boundary such that $b \leq c_1$.

Note that $c_1 - M \leq b$ as since the maximum chunk size is $M$.

There are 4 cases.

**Case 1** If $c_1 - M < b \leq c_0 - m$ ,

As $b$ is the last chunk boundary found by first thread where $b \leq c_1$ then there must not be another chunk between $c_1 - M$ and $c_0$ as that will be contradiction.

So since $b + m \leq c_0$ , then $c_0$ must be next chunk boundary found. After which we have aligned with the second thread.

**Case 2** If $c_0 - m < b \leq c_0$ .

The next chunk boundary found by first thread must be between $b+m$ and $b+M$.

From the case condition and condition 1, $b + m \leq c_0 + m < c_1$

And from the case condition $b + M > c_1$

So combining $b + m < c_1 < b + M$

Hence $c_1$ will be found by first thread.

**Case 3** If $c_0 < b \leq c_1 - m$.

As the next chunk boundary found by Thread 1 must be between $b+m$ and $b+M$,

We have $b + m \leq c_1$ from the case condition.

and $b + M > c_0 + M > c_1$ combining the case condition and that $c_1 - c_0 \leq M - m$ from condition 1.

So combining $b + m < c_1 \leq b + M$

As there are no other chunk boundaries from condition 2, $c_1$ must be found by the first thread.

**Case 4** If $c_1 - m < b \leq c_1$.

As the next chunk boundary found by Thread 1 must be between $b+m$ and $b+M$,

We must have $b + m < c_2$ from the case condition and condition 1.

Also, $b + M > c_1 + M - m \geq c_2$ combining the case condition and that $c_2 - c_1 \leq M - m$ from condition 1.

So combining $b + m < c_2 \leq b + M$

As there are no other chunk boundaries from condition 2, $c_2$ must be found by the first thread.

## 3 Implementation Errors

Now this procedure assumes that the hashing procedure *always* operates on a consistent window of $k$ bytes. However, as it turns in our implementation this is not the case. Specifically the Rust Gear Hash implementation is streamed and combining with a common performance optimization of skipping $m$ bytes, we really have the following implementation:

```
ChunkStart = 0
HashStreamStart = 0
while $i < |S|$ :
    if ($i$ - ChunkStart $\geq$ $M$) or (H(file[HashStreamStart...$i$]) == 1):
        AddChunk(file[ChunkStart, i])
        ChunkStart = $i$
        HashStreamStart = $i + m$
        $i = i + m$
        $i = i + 1$
    AddChunk(file[ChunkStart...])
```

Which has the odd side effect that for bytes $m$ to $m + k$ within a chunk we are hashing less than $k$ bytes. This means that in general, the hash output is **a function of the starting point of a chunk**, and can disagree on hash values for positions $m$ to $m + k$ of a chunk.

We can describe this using the following alternative notation for the hash function $H$.

$H(a, b) \rightarrow \{0, 1\} | b > a + m$ which is the output of the rolling hash chunking procedure on bytes $S[a...b]$. Note that this is still a rolling hash and its output in the usual case will depend only on the right-most $k$ bytes of the string $S[a...b]$. The implementation error above means that $H(a_1, b_1) = H(a_2, b_1)$ IFF $m + k \leq b_1 - a_1$ and $m + k \leq b_2 - a_2$.

Now, with this implementation, can we still perform parallel chunking?

I do not believe so. Due to the hash disagreement, under adverserial settings it is possible to construct two chunk sequences which will *never* align.