# Parallel Pipelines for Streaming Data

Magnus Erik Hvass Pedersen

September 13, 2022

## Abstract

This paper is a demonstration of a little-known method for parallelizing the computation of serially dependent functions on streaming data, using a particular kind of Parallel Pipeline. This can be used to convert a serial computation into a parallel computation whenever you have chained / nested functions working on streaming data. For example, this can be used in audio processing to make audio effects that are connected in series instead run in parallel on multiple CPU cores, without any modifications to the audio effects themselves. This can be used to greatly improve the efficiency on multi-core CPU systems.

## 1 Introduction

Parallel computation will become ever-more important in the future, because the single-core CPU speed is stagnating, so future performance gains will mostly come from using multiple CPU cores.

The parallelization method that is demonstrated here can be used whenever you have a chained or nested series of functions that are processing streaming data. The method is counter-intuitive and little-known; I personally don't recall having seen this method before, and it took an extensive internet-search to find out that it is perhaps a new kind of Parallel Pipeline.

We use audio processing as an example here, because that was the application area that inspired this work. In a Digital Audio Workstation (DAW) there are a number of audio tracks, and a number of auxiliary tracks for sending audio to be processed by effects, and a master track for mixing all the outputs of the individual tracks. The obvious or "naive" way of processing this in parallel, is to compute each individual track in its own CPU thread, but this creates a problem when some of the processing is to be done in series, e.g. when multiple tracks send their outputs to an aux track for combined processing with an effect such as a reverb, or when there are effects on the master track such as emulations of analog audio compressors and equalizers. That is why DAWs typically only parallelize the individual tracks, and when there are effects on the aux or master tracks, the entire computation becomes serially dependent and is therefore run on a single CPU core.

A Parallel Pipeline solves this problem quite easily by using a buffering-system to make multiple CPU cores work in parallel on sub-streams of data that are shifted slightly in time. The only drawback is that it adds some delay / latency to finish processing of the entire stream of input data. If low latency is important and the input data can be split into smaller sub-streams, as can be done in audio processing, then the latency can be reduced by processing smaller sub-streams instead.

# 2 Related Work

After getting the idea for this parallelization method, I did an extensive internet search for similar ideas, but could only find vague descriptions that resembled this method. There are many academic research papers on parallel computing but those are often quite obscure. For example, one paper on parallel pipelines written by 5 people from MIT and Intel was nearly incomprehensible, so it was unclear if their method was able to solve the same problem as the method presented here.

There is a popular C++ library called oneTBB developed by Intel for multi-core CPU programming, which also supports parallel pipelines. But from its docs and tutorial it appears that it also cannot convert a chain of serially dependent functions working on streaming data into a fully parallel pipeline. So perhaps the method in this paper really is a new invention. Either way, this paper is hopefully easy to understand so it can help many more people become familiar with the method.

# 3 Overview of Examples

This paper gives four examples for converting different combinations of functions from serial to parallel processing using buffered pipelines. The functions are denoted with the capital letters *F*, *G* and *H*, and in audio processing these could be effects such as an equalizer, compressor and reverb. For iteration *i* the input data is denoted $x_i$ and the output data is denoted $y_i$. The four examples are:

1. How to calculate $y_i = G(F(x_i))$ using 2 parallel threads. In audio processing this would correspond to two effects *F* and *G* in series e.g. on the same audio track – which may seem very counter-intuitive that it is even possible to process this in parallel.
2. How to calculate $y_i = H(G(F(x_i)))$ using 3 parallel threads. In audio processing this would correspond to 3 effects *F*, *G* and *H* in series e.g. on the same audio track.
3. How to calculate $y_i = F(x_i) + G(F(x_i))$ using 2 parallel threads. In audio processing this might correspond to one effect *F* calculated on an audio track $x_i$, and this being sent to an aux track with another effect *G*, and then the results being mixed to the master output track.
4. How to calculate $y_i = H(F(x_i) + G(z_i))$ using 3 parallel threads. In audio processing this might correspond to two audio tracks $x_i$ and $z_i$ being processed with effects *F* and *G*, respectively, and then their outputs are grouped and further processed with effect *H*.

These four examples should give you a good understanding of the underlying idea of buffering the output of one function to be used as the input in another function that is being run in parallel, so you can make a parallel pipeline for your particular problem.

# 4 Computer Code

Computer code in C++ for these four examples has been made available for download on GitHub[1]. The code uses dummy functions for *F*, *G* and *H* that just "sleep" the execution threads to simulate heavy computations, and the data being processed is just simple string symbols, so you can see how the data is being processed by the functions that are being run in parallel in different CPU threads. The pseudo-code and the actual output of the C++ code is also shown in the following sections.

1 https://github.com/Hvass-Labs/Parallel-Pipelines

# 5 Example 1

The first example has two nested or chained functions $F$ and $G$ that map the input $x_i$ to the output $y_i$:

$$y_i = G(F(x_i)) \tag{1}$$

To compute this in parallel, the idea is to compute $F(x_i)$ in one CPU thread and assign the result to the buffer variable $b_i$, while simultaneously computing in another CPU thread $G(b_{i-1})$ using the buffered output from the previous iteration $b_{i-1}$, so as to produce the final output for the previous iteration $y_{i-1}$. This introduces an extra iteration of latency / delay, because the result $y_i$ is first available after processing the input data for iteration $i+1$.

The flow-charts for both the serial and parallel computations are shown in Figure 1, where the functions $F$ and $G$ are shown as blue squares, the input data $x_i$ is a lime-colored circle, the output data $y_i$ is a green circle, and the parallel pipeline buffers $b_i$ and $b_{i-1}$ are shown as red circles.

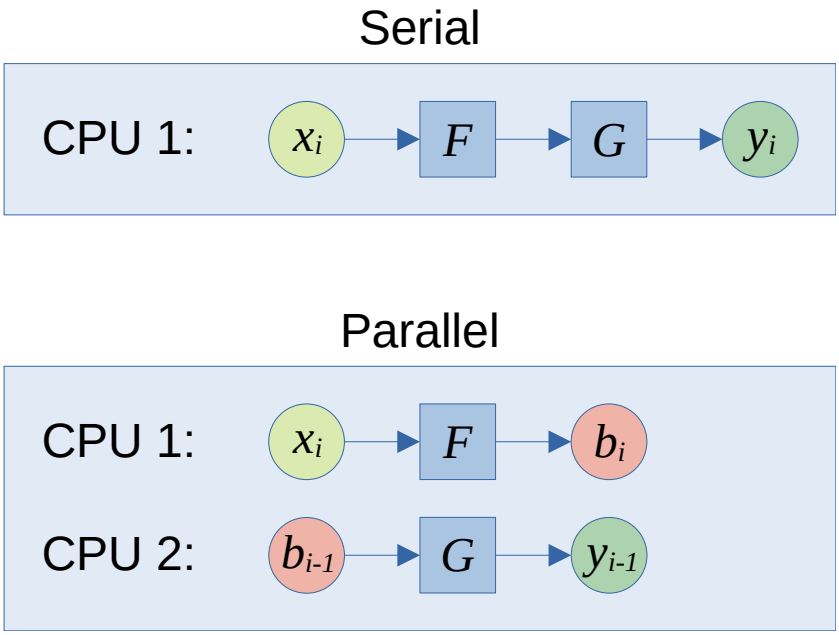


*Figure 1: Computation flow-chart for **Example 1** to compute $y_i = G(F(x_i))$ in a Parallel Pipeline using 2 CPU threads.*

## 5.1 Buffer Management

There are different ways of managing the buffers for the parallel pipeline, which were denoted $b_i$ and $b_{i-1}$ and shown as red circles in Figure 1. The easiest is to handle the buffering in the main CPU thread. Then we only need a single buffer to save the output of $F(x_i)$ that was computed in one CPU thread, and which is to be used as input in the other CPU thread. We denote this buffer as $F_{buffer}$ in math notation, and as `F_buffer` in the C++ code and the pseudo-code below.

If for some reason you don't want to manage the buffering in the master-thread, then you must have two buffers, one for reading and another for writing, and then e.g. have the writing thread use an atomic or synchronized switch between the two buffers to avoid parallel race-conditions.

## 5.2 Pseudo-Code

The following pseudo-code shows how to compute $y_i = G(F(x_i))$ in a parallel pipeline using 2 CPU threads, with the parallel buffering being handled by the main CPU thread. As previously discussed, we need to compute an extra iteration `n+1` instead of just `n`, because of the buffering in the parallel pipeline. Also note that variable initializations are omitted in the pseudo-code.

```
for (int i=0; i<n+1; i++)
{
    Use thread 1 to calculate F(x[i]);
    Use thread 2 to calculate G(F_buffer);
    Wait for both threads to finish;
    Get final result from thread 2: y[i-1] = G(F_buffer);
    Update buffer with result from thread 1: F_buffer = F(x[i]);
}
```

## 5.3 Result

The following shows the actual output of the C++ program for computing $y_i = G(F(x_i))$ with both serial and parallel executions. The input is a list of 10 strings to symbolize arbitrary input-data $x_i$.

```
Serial:
Step 0:  Thread 1: G(F(x_0))
Step 1:  Thread 1: G(F(x_1))
Step 2:  Thread 1: G(F(x_2))
Step 3:  Thread 1: G(F(x_3))
Step 4:  Thread 1: G(F(x_4))
Step 5:  Thread 1: G(F(x_5))
Step 6:  Thread 1: G(F(x_6))
Step 7:  Thread 1: G(F(x_7))
Step 8:  Thread 1: G(F(x_8))
Step 9:  Thread 1: G(F(x_9))
Elapsed time: 2004.334316ms

Parallel:
Step 0:  Thread 1: F(x_0)  Thread 2: G(--)
Step 1:  Thread 1: F(x_1)  Thread 2: G(F(x_0))
Step 2:  Thread 1: F(x_2)  Thread 2: G(F(x_1))
Step 3:  Thread 1: F(x_3)  Thread 2: G(F(x_2))
Step 4:  Thread 1: F(x_4)  Thread 2: G(F(x_3))
Step 5:  Thread 1: F(x_5)  Thread 2: G(F(x_4))
Step 6:  Thread 1: F(x_6)  Thread 2: G(F(x_5))
Step 7:  Thread 1: F(x_7)  Thread 2: G(F(x_6))
Step 8:  Thread 1: F(x_8)  Thread 2: G(F(x_7))
Step 9:  Thread 1: F(x_9)  Thread 2: G(F(x_8))
Step 10: Thread 1: F(--)   Thread 2: G(F(x_9))
Elapsed time: 1106.959222ms
```

The total time-usage is roughly 2000 msec in the serial execution, because there are 10 iterations of two function calls for *F* and *G*, and each of these functions “sleep” the thread for 100 msec to simulate heavy computation. In parallel execution the total time-usage is roughly 1100 msec, which is nearly half the time of the serial execution, except for approximately 100 msec, which is the extra latency / delay of one iteration, due to the buffering in the parallel pipeline. This can also be seen by the `G(--)` in the first iteration and `F(--)` in the last iteration, which means that those functions are being called with empty input-data at the beginning and end of the stream of input-data.

# 6 Example 2

The second example has 3 nested functions *F*, *G* and *H* that map the input $x_i$ to the output $y_i$:

$$y_i = H(G(F(x_i))) \tag{2}$$

To compute this in parallel, the idea is to compute $F(x_i)$ in one CPU thread and assign the result to the buffer variable $b_i$, while simultaneously computing in another CPU thread $G(b_{i-1})$ using the buffered output from the previous iteration $b_{i-1}$, and assign the result to another buffer $c_{i-1}$, and simultaneously computing in a third CPU thread $H(c_{i-2})$ using the buffered output from the previous iteration $c_{i-2}$, so as to produce the final output for the previous iteration $y_{i-2}$. This introduces an extra two iterations of latency / delay, because the result $y_i$ is first available after processing iteration $i+2$. The serial and parallel computation flow-charts are shown in Figure 2, using the same color-codings as in Figure 1.

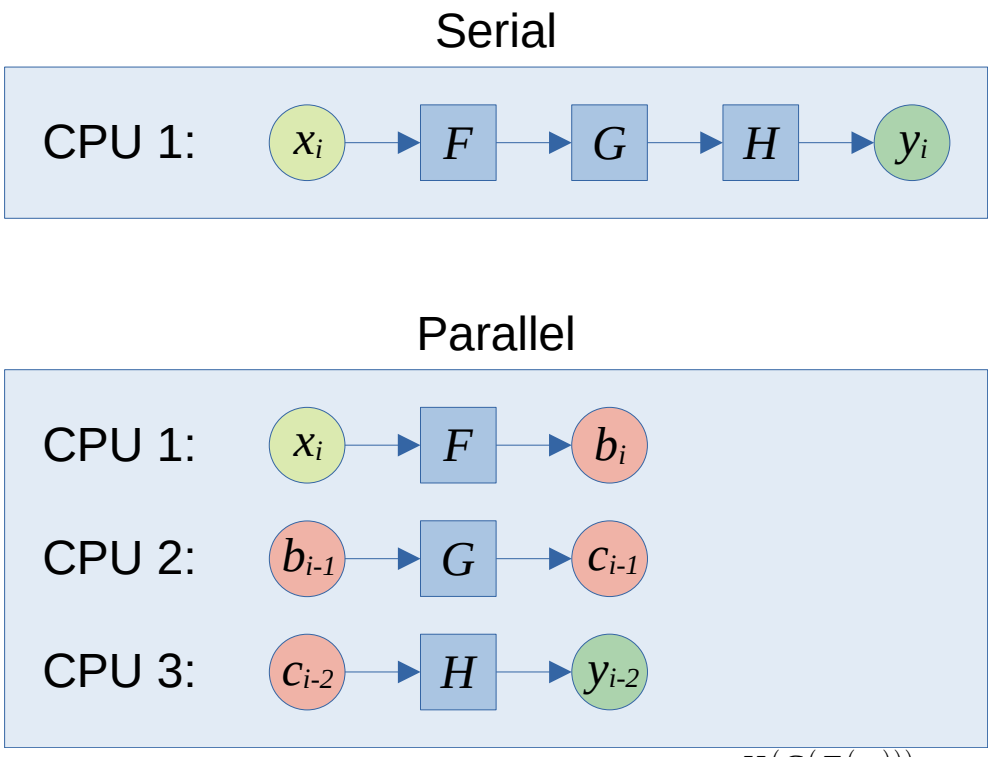


*Figure 2: Computation flow-chart for **Example 2** to compute $y_i = H(G(F(x_i)))$ in a Parallel Pipeline using 3 CPU threads.*

## 6.1 Pseudo-Code

The following pseudo-code shows how to compute $y_i = H(G(F(x_i)))$ in a parallel pipeline using 3 CPU threads, with the parallel buffering being handled by the main CPU thread. We denote the buffers as `F_buffer` and `G_buffer` instead of $b_{i-1}$ and $c_{i-2}$ respectively. And because of this buffering in the parallel pipeline, we need to compute two extra iterations `n+2` instead of just `n`.

```
for (int i=0; i<n+2; i++)
{
    Use thread 1 to calculate F(x[i]);
    Use thread 2 to calculate G(F_buffer);
    Use thread 3 to calculate H(G_buffer);
    Wait for all 3 threads to finish;
    Get final result from thread 3: y[i-2] = H(G_buffer);
    Update the buffers with the results from threads 1 and 2:
        G_buffer = G(F_buffer);
        F_buffer = F(x[i]);
}
```

## 6.2 Result

The following shows the actual output of the C++ program for computing $y_i = H(G(F(x_i)))$ with both serial and parallel executions. The input is a list of 10 string symbols for the input-data $x_i$.

```
Serial:
Step 0:  Thread 1: H(G(F(x_0)))
Step 1:  Thread 1: H(G(F(x_1)))
Step 2:  Thread 1: H(G(F(x_2)))
Step 3:  Thread 1: H(G(F(x_3)))
Step 4:  Thread 1: H(G(F(x_4)))
Step 5:  Thread 1: H(G(F(x_5)))
Step 6:  Thread 1: H(G(F(x_6)))
Step 7:  Thread 1: H(G(F(x_7)))
Step 8:  Thread 1: H(G(F(x_8)))
Step 9:  Thread 1: H(G(F(x_9)))
Elapsed time: 3005.789506ms

Parallel:
Step 0:  Thread 1: F(x_0)  Thread 2: G(--)      Thread 3: H(--)
Step 1:  Thread 1: F(x_1)  Thread 2: G(F(x_0))  Thread 3: H(G(--))
Step 2:  Thread 1: F(x_2)  Thread 2: G(F(x_1))  Thread 3: H(G(F(x_0)))
Step 3:  Thread 1: F(x_3)  Thread 2: G(F(x_2))  Thread 3: H(G(F(x_1)))
Step 4:  Thread 1: F(x_4)  Thread 2: G(F(x_3))  Thread 3: H(G(F(x_2)))
Step 5:  Thread 1: F(x_5)  Thread 2: G(F(x_4))  Thread 3: H(G(F(x_3)))
Step 6:  Thread 1: F(x_6)  Thread 2: G(F(x_5))  Thread 3: H(G(F(x_4)))
Step 7:  Thread 1: F(x_7)  Thread 2: G(F(x_6))  Thread 3: H(G(F(x_5)))
Step 8:  Thread 1: F(x_8)  Thread 2: G(F(x_7))  Thread 3: H(G(F(x_6)))
Step 9:  Thread 1: F(x_9)  Thread 2: G(F(x_8))  Thread 3: H(G(F(x_7)))
Step 10: Thread 1: F(--)   Thread 2: G(F(x_9))  Thread 3: H(G(F(x_8)))
Step 11: Thread 1: F(--)   Thread 2: G(F(--))   Thread 3: H(G(F(x_9)))
Elapsed time: 1207.153470ms
```

The total time-usage is roughly 3000 msec in the serial execution, as there are 10 iterations of 3 function calls for *F*, *G* and *H*, and each of these "sleep" the thread for 100 msec. In parallel mode the total time-usage is roughly 1200 msec, which is nearly 1/3 the time of the serial execution, except for roughly 200ms, which is the extra latency of two iterations due to the pipeline buffering.

# 7 Example 3

The third example has two nested functions $F$ and $G$ that map the input $x_i$ to the output $y_i$:

$$y_i = F(x_i) + G(F(x_i)) \tag{3}$$

Figure 3 shows the serial execution as well as two versions of parallel pipelines. In the first parallel version, one CPU thread computes $F(x_i)$ and assigns the result to the buffer variable $b_i$, while the second CPU thread computes $G(b_{i-1})$ using the buffered output from the previous iteration $b_{i-1} = F(x_{i-1})$ and then sums the result $G(b_{i-1})$ with $b_{i-1}$ to get the final output for the previous iteration $y_{i-1}$. Note that it is important to sum with $b_{i-1}$ from the previous iteration to match the indices. The second parallel version is similar, but the summing is done on the main thread.

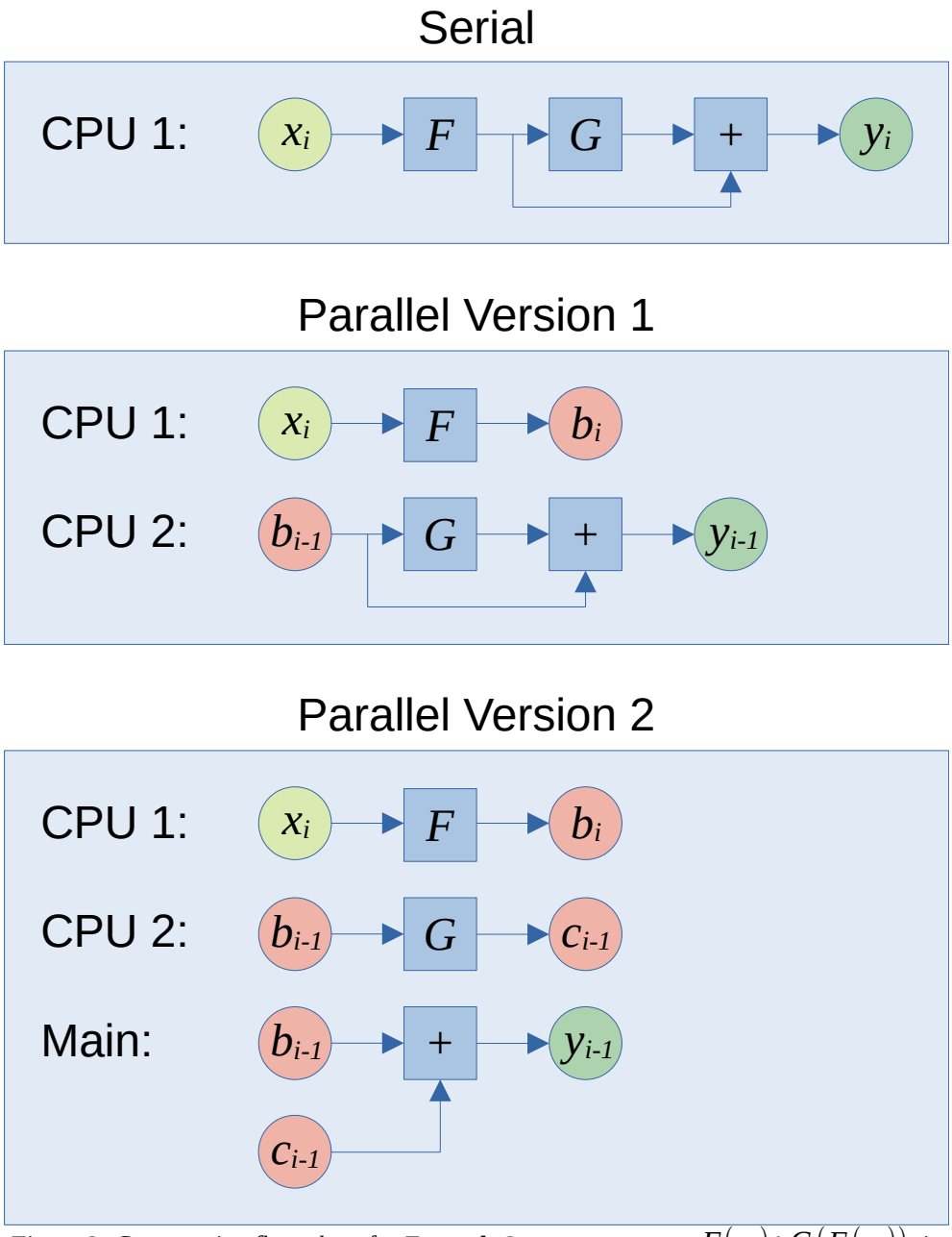


*Figure 3: Computation flow-chart for **Example 3** to compute $y_i = F(x_i) + G(F(x_i))$ in two versions of Parallel Pipelines using 2 CPU threads.*

## 7.1 Pseudo-Code

The following pseudo-code shows version 1 of the parallel pipeline from Figure 3, where the summation is being handled in the second thread, but the parallel buffering is still being handled by the main CPU thread. We denote the buffer as `F_buffer` instead of $b_i$. And because of this buffering in the parallel pipeline, we need to compute one extra iteration `n+1` instead of just `n`.

```
for (int i=0; i<n+1; i++)
{
    Use thread 1 to calculate F(x[i]);
    Use thread 2 to calculate G(F_buffer) + F_buffer;
    Wait for all 2 threads to finish;
    Get final result from thread 2: y[i-1] = G(F_buffer) + F_buffer;
    Update buffer with the result from thread 1: F_buffer = F(x[i]);
}
```

The second parallel version is almost identical, it just computes the summation on the main thread:

```
for (int i=0; i<n+1; i++)
{
    Use thread 1 to calculate F(x[i]);
    Use thread 2 to calculate G(F_buffer);
    Wait for all 2 threads to finish;
    Compute final result on main thread: y[i-1] = G(F_buffer) + F_buffer;
    Update buffer with the result from thread 1: F_buffer = F(x[i]);
}
```

## 7.2 Result

The following shows the output of the C++ program for computing $y_i = F(x_i) + G(F(x_i))$ with both serial and parallel executions using the second parallel version. The input is a list of 10 symbols $x_i$.

```
Serial:
Step 0:  Thread 1: F(x_0) + G(F(x_0))
Step 1:  Thread 1: F(x_1) + G(F(x_1))
Step 2:  Thread 1: F(x_2) + G(F(x_2))
...
Step 7:  Thread 1: F(x_7) + G(F(x_7))
Step 8:  Thread 1: F(x_8) + G(F(x_8))
Step 9:  Thread 1: F(x_9) + G(F(x_9))
Elapsed time: 2004.156225ms

Parallel:
Step 0:  Thread 1: F(x_0)  Thread 2: G(--)      Thread Main: -- + G(--)
Step 1:  Thread 1: F(x_1)  Thread 2: G(F(x_0))  Thread Main: F(x_0) + G(F(x_0))
Step 2:  Thread 1: F(x_2)  Thread 2: G(F(x_1))  Thread Main: F(x_1) + G(F(x_1))
...
Step 7:  Thread 1: F(x_7)  Thread 2: G(F(x_6))  Thread Main: F(x_6) + G(F(x_6))
Step 8:  Thread 1: F(x_8)  Thread 2: G(F(x_7))  Thread Main: F(x_7) + G(F(x_7))
Step 9:  Thread 1: F(x_9)  Thread 2: G(F(x_8))  Thread Main: F(x_8) + G(F(x_8))
Step 10: Thread 1: F(--)   Thread 2: G(F(x_9))  Thread Main: F(x_9) + G(F(x_9))
Elapsed time: 1106.460851ms
```

The total time-usage is roughly 2000 msec in the serial execution, as there are 10 iterations of 2 function calls for *F* and *G*, and each of these "sleep" the thread for 100 msec. In parallel mode the total time-usage is roughly 1100 msec, which is nearly half the time of the serial execution, except for roughly 100ms, which is the extra latency of one iteration due to the pipeline buffering.

# 8 Example 4

The 4th and final example has 3 nested functions *F*, *G* and *H* that map the input $x_i$ to the output $y_i$:

$$y_i = H(F(x_i) + G(z_i)) \tag{4}$$

Figure 4 shows the serial execution as well as two versions of parallel pipelines. The first parallel version computes $b_i = F(x_i)$ in one thread and $c_i = G(z_i)$ in another thread, and these buffers are then used to compute the final output $y_{i-1} = H(b_{i-1} + c_{i-1})$ in a third thread. Because of the buffers there is one extra iteration of latency / delay before the final output $y_i$ is ready. The second parallel version is very similar, but it computes the sum of the two pipeline buffers in the main thread.

## 8.1 Pseudo-Code

The following pseudo-code shows version 1 of the parallel pipeline from Figure 4, where the summation is being handled in the third thread, but the parallel buffering is still being handled by the main CPU thread. We denote the buffers as `F_buffer` and `G_buffer` instead of $b_i$ and $c_i$. And because of this buffering in the parallel pipeline, we need to compute one extra iteration `n+1` instead of just `n`. Note that even though we have 3 threads, we only have 1 extra iteration of latency, because the extra latency only depends on the number of buffer-layers, as discussed in Section 10.

```
for (int i=0; i<n+1; i++)
{
    Use thread 1 to calculate F(x[i]);
    Use thread 2 to calculate G(z[i]);
    Use thread 3 to calculate H(F_buffer + G_buffer);
    Wait for all 3 threads to finish;
    Get final result from thread 3: y[i-1] = H(F_buffer + G_buffer);
    Update buffers with the results from threads 1 and 2:
        F_buffer = F(x[i]);
        G_buffer = G(z[i]);
}
```

The following pseudo-code shows version 2 of the parallel pipeline from Figure 4, which is very similar to the above, except that the summation is done on the main thread, so we only need a single buffer which is denoted `F_G_sum_buffer`. We still just need to compute one extra iteration `n+1`.

```
for (int i=0; i<n+1; i++)
{
    Use thread 1 to calculate F(x[i]);
    Use thread 2 to calculate G(z[i]);
    Use thread 3 to calculate H(F_G_sum_buffer);
    Wait for all 3 threads to finish;
    Get final result from thread 3: y[i-1] = H(F_G_sum_buffer);
    Update buffer with the sum of results from threads 1 and 2:
        F_G_sum_buffer = F(x[i]) + G(z[i]);
}
```

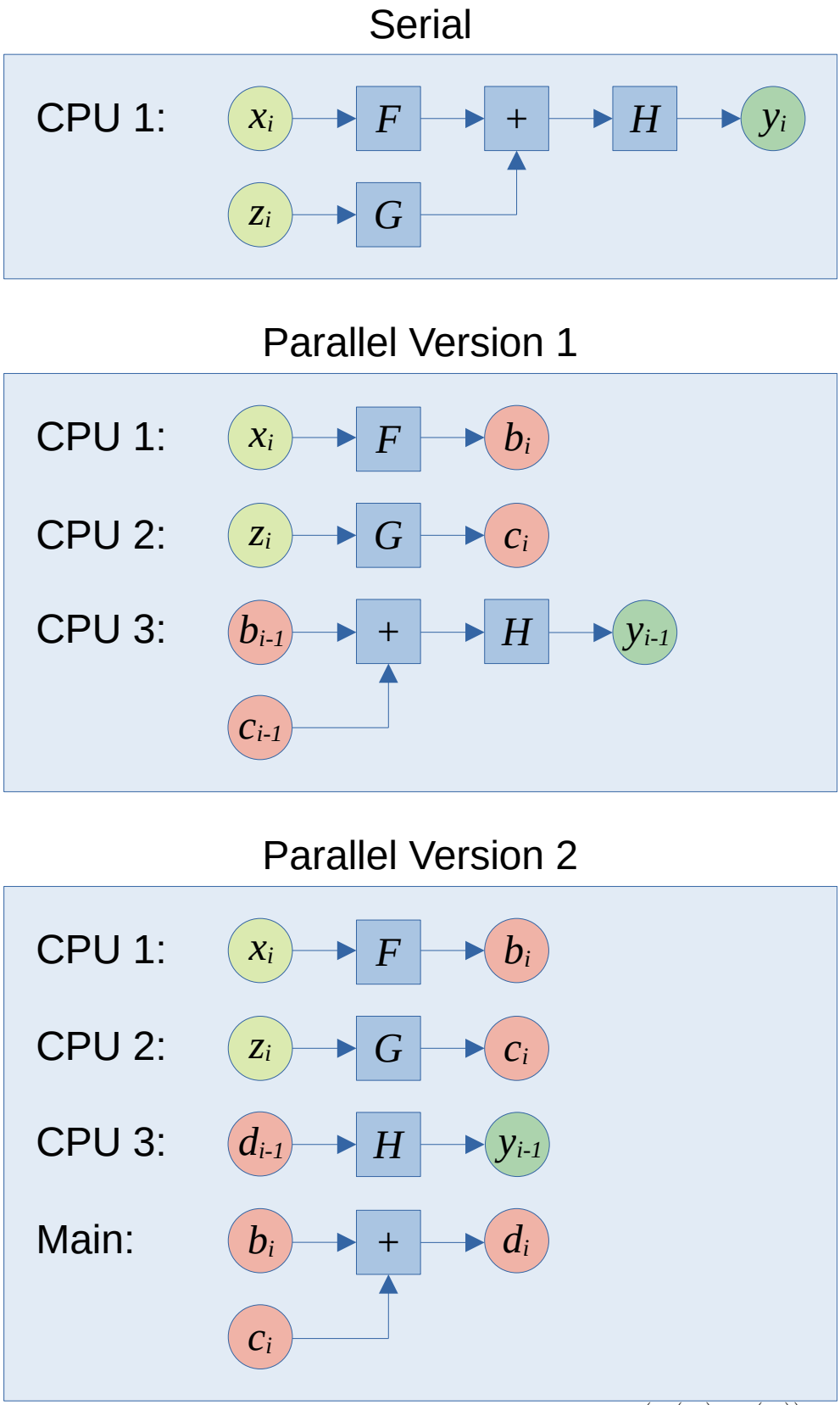


*Figure 4: Computation flow-chart for **Example 4** to compute* $y_i = H(F(x_i) + G(z_i))$ *in two versions of Parallel Pipelines using 3 CPU threads.*

## 8.2 Result

The following shows the output of the C++ program for computing $y_i = H(F(x_i) + G(z_i))$ with both serial and parallel executions using the second parallel version from Figure 4. The input is two lists $x_i$ and $z_i$ each having 10 string symbols.

```
Serial:
Step 0:  Thread 1: H(F(x_0) + G(z_0))
Step 1:  Thread 1: H(F(x_1) + G(z_1))
Step 2:  Thread 1: H(F(x_2) + G(z_2))
Step 3:  Thread 1: H(F(x_3) + G(z_3))
Step 4:  Thread 1: H(F(x_4) + G(z_4))
Step 5:  Thread 1: H(F(x_5) + G(z_5))
Step 6:  Thread 1: H(F(x_6) + G(z_6))
Step 7:  Thread 1: H(F(x_7) + G(z_7))
Step 8:  Thread 1: H(F(x_8) + G(z_8))
Step 9:  Thread 1: H(F(x_9) + G(z_9))
Elapsed time: 3004.522127ms

Parallel:
Step 0:  Thread 1: F(x_0)  Thread 2: G(z_0)  Thread 3: H(--)
Step 1:  Thread 1: F(x_1)  Thread 2: G(z_1)  Thread 3: H(F(x_0) + G(z_0))
Step 2:  Thread 1: F(x_2)  Thread 2: G(z_2)  Thread 3: H(F(x_1) + G(z_1))
Step 3:  Thread 1: F(x_3)  Thread 2: G(z_3)  Thread 3: H(F(x_2) + G(z_2))
Step 4:  Thread 1: F(x_4)  Thread 2: G(z_4)  Thread 3: H(F(x_3) + G(z_3))
Step 5:  Thread 1: F(x_5)  Thread 2: G(z_5)  Thread 3: H(F(x_4) + G(z_4))
Step 6:  Thread 1: F(x_6)  Thread 2: G(z_6)  Thread 3: H(F(x_5) + G(z_5))
Step 7:  Thread 1: F(x_7)  Thread 2: G(z_7)  Thread 3: H(F(x_6) + G(z_6))
Step 8:  Thread 1: F(x_8)  Thread 2: G(z_8)  Thread 3: H(F(x_7) + G(z_7))
Step 9:  Thread 1: F(x_9)  Thread 2: G(z_9)  Thread 3: H(F(x_8) + G(z_8))
Step 10: Thread 1: F(--)   Thread 2: G(--)   Thread 3: H(F(x_9) + G(z_9))
Elapsed time: 1106.835034ms
```

The total time-usage is roughly 3000 msec in the serial execution, as there are 10 iterations of 3 function calls for *F*, *G* and *H*, and each of these "sleep" the thread for 100 msec. In parallel mode the total time-usage is roughly 1100 msec, which is nearly 1/3 the time of the serial execution, except for roughly 100 msec, which is one iteration of extra latency due to the pipeline buffering.

# 9 Functions With States / Memory

If the functions *F*, *G* and *H* have internal states or memory that get updated on each call, they may not support being called with "empty" data. In this case the functions should not get called with "empty" data, which happens in the parallel pipeline at the start and end of the stream of input-data, as shown by the calls to `F(--)`, `G(--)` and `H(--)` in the results above. This can easily be avoided when programming the parallel pipeline.

Other functions may support "empty" data. For example, it is typical in audio processing that effects have an internal state that gets updated on each function call, but we can just input silence at the start and end of the stream of input-data, and the audio effects should be able to handle this.

# 10 Latency Reduction

As shown in the examples above, the parallel pipelines increase the latency / delay so one or more extra iterations are required before the entire stream of input-data has been fully processed. The number of extra iterations does not depend on the number of CPU threads, but instead depends on the number of “buffering layers” needed in the parallel pipeline. Example 1 (Section 5) has 1 buffering layer in the parallel pipeline so it needs 1 extra iteration to fully compute the entire stream of input-data, while Example 2 (Section 6) has 2 buffering layers in the parallel pipeline so it needs 2 extra iterations to fully compute the entire stream of input-data.

The extra latency is not a problem if you just want to process an entire stream of input-data from beginning to end, because the overall time-usage of the parallel pipeline is still just a fraction of the serial execution, even with the extra iterations of latency. But in real-time systems such as audio processing in DAWs, the extra latency can be a big problem, especially when there are several layers of buffering in the parallel pipeline, because each layer creates one more iteration of latency.

For example, say we are running a DAW at a sample-rate of 48 kHz. The sound-card asks the DAW to compute one block of audio-data at regular intervals with e.g. 480 samples corresponding to 10 msec. The audio is being processed in blocks so as to lower the overhead of function calls etc. as it would be too costly to only compute one sample at a time. In the notation of this paper, the input-data $x_i$ would actually be an entire block or array of audio-data with 480 samples (each sample is just a numerical value), and the DAW needs to compute some functions $F$, $G$ and $H$ on $x_i$ in order to produce the output block of audio-data with 480 samples as well. If we just have one layer of buffering as in Example 1 (Section 5) then we get an extra latency of 480 samples or 10 msec, and for two layers of buffering as in Example 2 (Section 6) we get 960 samples or 20 msec of extra latency. A latency of 20-30 msec starts to be perceptible to a human, so this would limit the parallel pipelines to just one or two layers of parallelization, which may be sufficient for many audio use-cases, but the extra latency would still be undesirable for real-time interaction with the system.

Fortunately there is a simple solution, which is to use mini-blocks of data. For example, instead of using the entire blocks of audio-data with 480 samples, we could split these into 10 mini-blocks of 48 samples each, and run the parallel pipeline with these mini-blocks instead. With one buffering layer we would then get an extra latency of only 48 samples or 1 msec, for a total latency of 11 msec instead of 10 msec without using the parallel pipeline. And for two buffering layers we would get an extra latency of only 96 samples or 2 msec, for a total latency of 12 msec instead of 10 msec when not using the parallel pipeline. These costs would be negligible compared to the great benefit of being able to use more CPU cores in the computation.

As the mini-blocks are much smaller, the processing overhead is going to be bigger as well, because the DAW has to call the processing functions e.g. 10 times per block instead of just once. Exactly how much this costs would require experimentation, because computers are so advanced with CPU caches etc. that the result can be surprising. But even if we cannot have mini-blocks of only 48 samples but need e.g. 120 samples, it would still make the extra latency from the parallel pipeline so small, that it would be well worth it to obtain much greater usage of a multi-core CPU system. We just want to limit the overhead of using mini-blocks. Also, using mini-blocks should not affect the stability of the system because the sound-card still uses its original block-size of e.g. 480 samples.

# 11 Graph-Conversion Algorithm

If your system has a fairly simple data-flow that does not change over time, then you can just implement the parallel pipeline manually. But other applications have more complicated data-flows that can change over time, e.g. when the user changes the audio-routing in a DAW. In this case you need an algorithm for converting the serial computational graph into a parallel pipeline.

It would probably be possible to get nearly 100% utilization of a multi-core CPU using parallel pipelines. But this is a very hard problem to solve optimally for every use-case, because deciding which processing functions to combine into different CPU threads is similar to a “bin-packing” problem which is known to have NP-hard complexity. But sub-optimal solutions are probably going to be acceptable for most applications, as they will certainly run much faster than serial execution.

For audio processing in a DAW, a few simple heuristic rules are probably sufficient to greatly improve the multi-core CPU usage. For example, effects on audio-tracks, grouped-tracks, aux-tracks, and the master-bus can be run in separate threads for each track and using parallel pipelines and buffering only when necessary. Within a single track, if the audio effects exceed 100% CPU usage, then the effects can be split into two or more CPU threads using parallel pipelines. These few simple rules are probably sufficient to get very high multi-core CPU usage in most cases.

# 12 Future Research

There are few research papers on multi-core CPU usage for audio processing. A paper by Kiefer et al.[2] modified a commercial system for live DJ performance, which is simpler than a full DAW, because the DJ system only has 4 audio tracks with minor flexibility in the routing of effects within those tracks. Their method used a queue for the different functions to be computed, and they made different scheduling algorithms for selecting which functions to run on which CPU threads, while ensuring they were executed in the correct order. The authors were apparently unfamiliar with our pipelining method here, because they were unable to parallelize the serially dependent functions. So it could be interesting to see a combination of some of their ideas for scheduling algorithms with the parallelization method presented in this paper. It could probably reach very high CPU efficiency.

This is an interesting research topic that is unfortunately closed off to the general public as most computer code for commercial audio software is “Top Secret”. Hopefully some audio company will release an open-source and simplified “dummy” version of their audio-engine in the future, that could be used by everyone to try and improve various aspects including the parallel CPU efficiency.

# 13 Conclusion

This paper demonstrated how to convert serially connected functions that are working on streaming data into a fully parallel pipeline. The only penalty is that it introduces some extra latency, which may be undesirable in real-time systems. But in case the streaming data can be further split into sub-streams, as is the case with audio data, the latency can be greatly reduced by having the parallel pipeline instead work on these smaller sub-streams of data.

2 M.A. Kiefer, K. Molitorisz, J. Bieler, W.F. Tichy, *“Parallelizing a Real-time Audio Application - A Case Study in Multithreaded Software Engineering”*, 2015.