DAW Design & Implementation #5

Parallel Algorithms for Realtime Audio

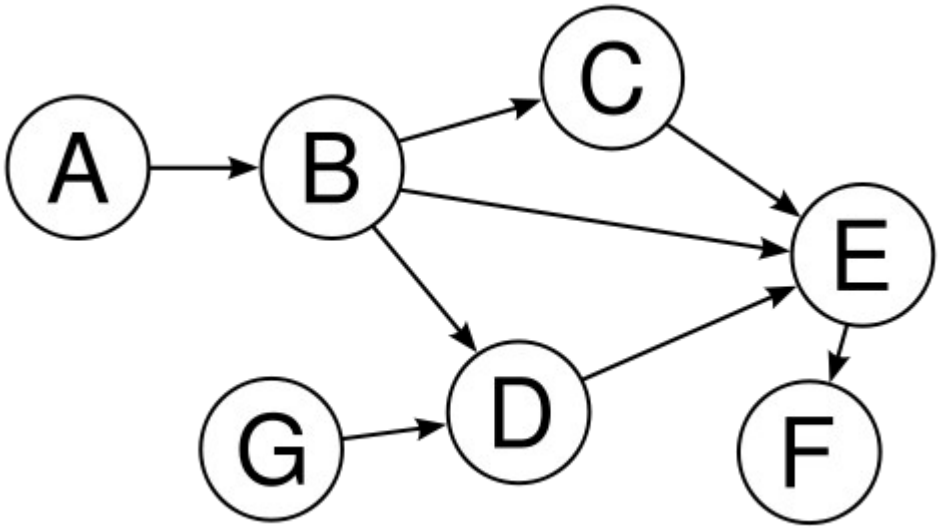
Plan of Attack

- Presentation on CraSynth by Lukas Kaser/Andreas Rothe
- ...waffle...
- High level task parallelism
- Low level data parallelism

High Level Task Parallelism

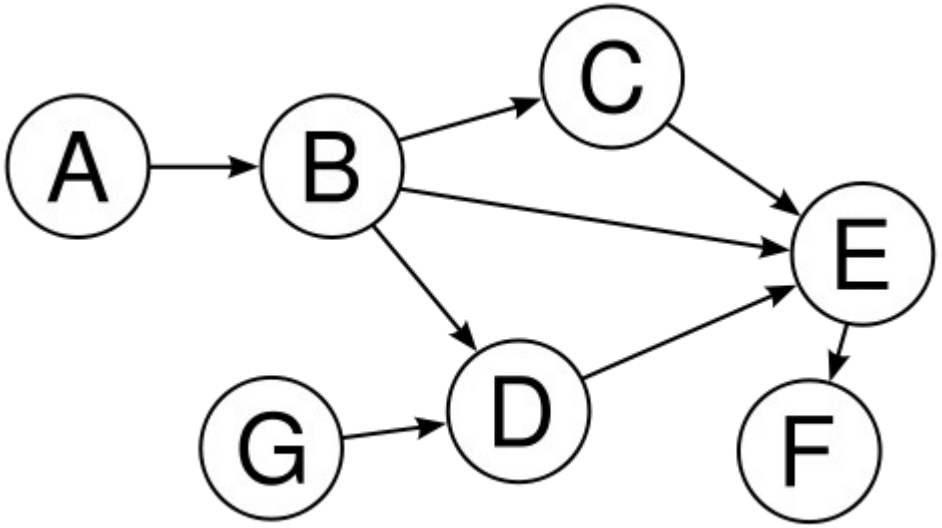
- Task parallelism: different operations performed on distinct data in parallel
- Data parallelism: same operation performed on “related” data in parallel

Directed Data graph



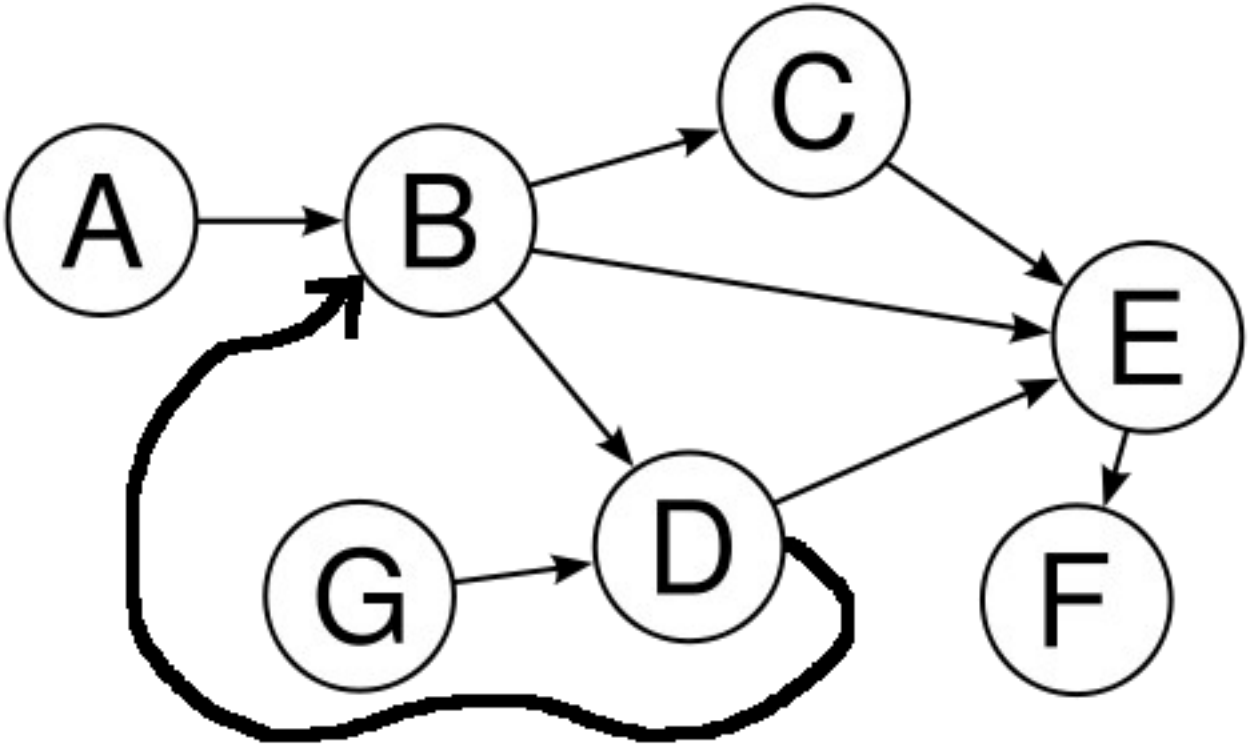
- Nodes or vertices connected together
- Directions have semantics
- CS terms: connections = “edges”, lines or arcs
- Sources: nodes with no incoming connections
- Sinks: nodes with no outgoing connections

Directed Acyclic Data Graph



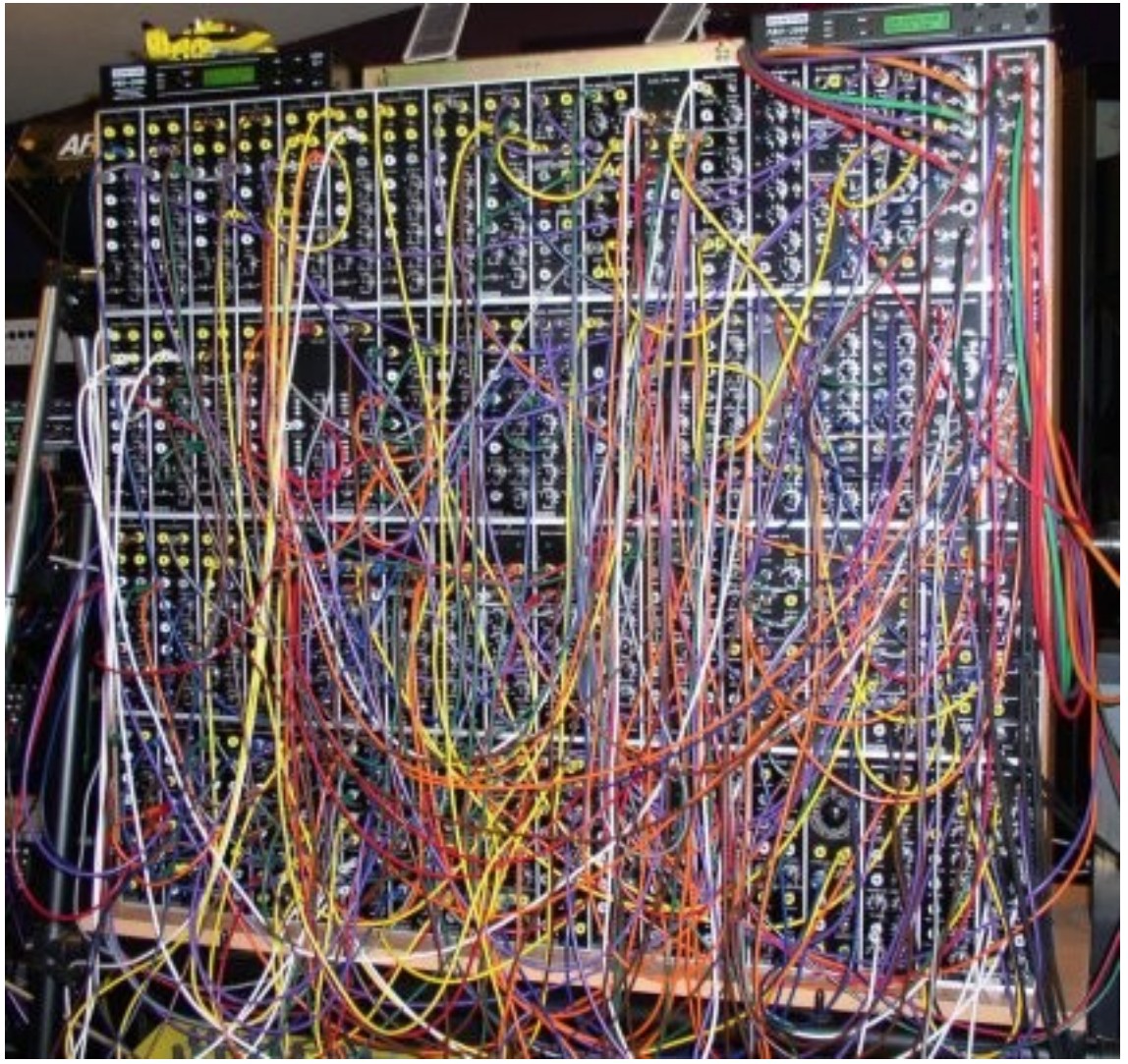
- No cycles
- no non-empty path that starts on one node and ends on another

Directed Cyclic Graph



- Cycles permitted
- CS and math theory doesn't have as much to say about these

What am I talking about?



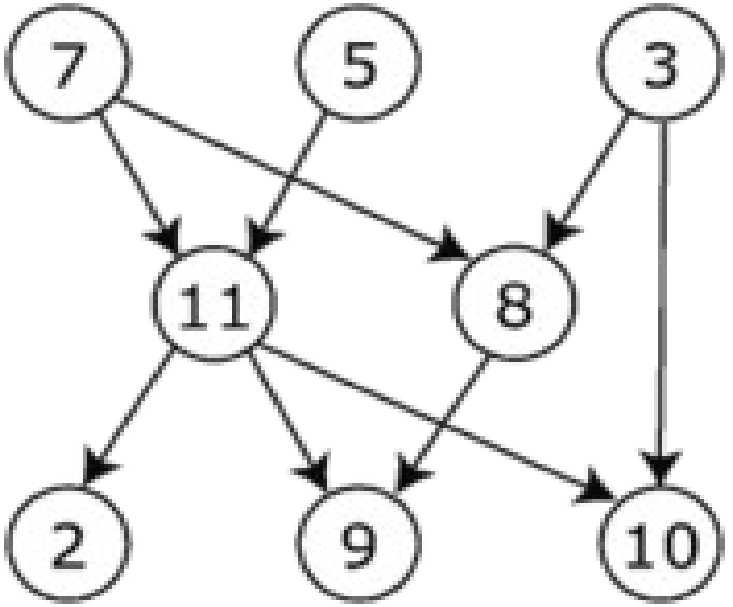
- Nodes are:
- tracks, busses,
- tone generators,
- FX plugins ....
- controls
- Connections are ...
- Cycles are ... ???

So ...

- Any sufficiently complex realtime audio software contains a series of nodes connected together to form a directed graph
- Execute the software by running one node after another.
- How to determine the order?

Graph Ordering/Sorting

- Any DAG has 1 or more “topological sorts”
- A list of the nodes in which each one comes before any other that it is connected to
- Many DAG's do not have unique sorting order



How to sort a DAG

```
void
jack_sort_graph (jack_engine_t *engine)
{
    /* called, obviously, must hold engine->client_lock */

    VERBOSE (engine, "++ jack_sort_graph");
    engine->clients = jack_slist_sort (engine->clients,
                      (JCompareFunc) jack_client_sort);
    jack_compute_all_port_total_latencies (engine);
    jack_rechain_graph (engine);
    VERBOSE (engine, "-- jack_sort_graph");
}
```

Comparing two nodes

```
static int
jack_client_sort (jack_client_internal_t *a, jack_client_internal_t *b)
{
        /* drivers are forced to the front, ie considered as sources
           rather than sinks for purposes of the sort */

        if (jack_client_feeds_transitive (a, b) ||
            (a->control->type == ClientDriver &&
             b->control->type != ClientDriver)) {
                return -1;
        } else if (jack_client_feeds_transitive (b, a) ||
                   (b->control->type == ClientDriver &&
                    a->control->type != ClientDriver)) {
                return 1;
        } else {
                return 0;
        }
}
```

Wait a minute!

What about cycles (feedback)?

Take a deep breath

- Each node has a special list of other nodes that it “feeds”
- List does not include connections to itself, or connections to sources or sinks
- If a connection is added that creates feedback between A and B (ie. There was already a path from A > B, and we add B > A) then instead of A being on B's list, A will be B's.
- These 3 conditions guarantee an acyclic graph
- This allows us to use regular sorting algorithms
- Thank you to CS professors everywhere and to Simon Jenkins

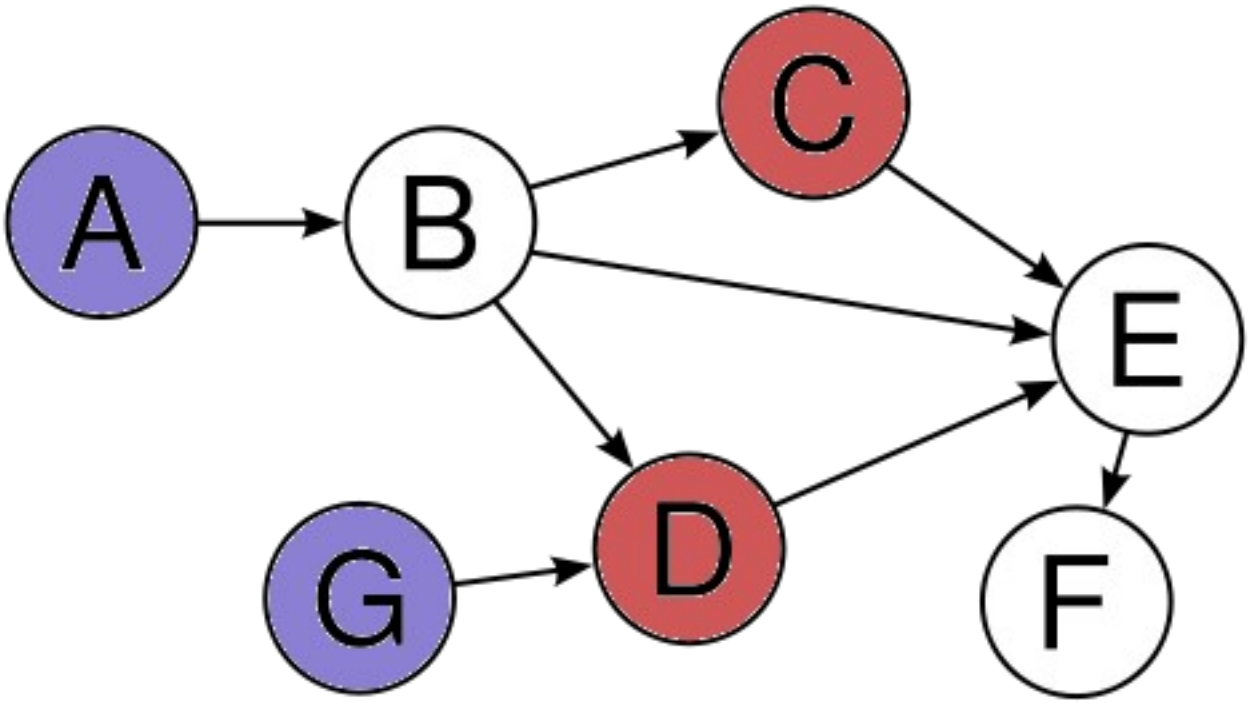
Problems with Graph Sorting

- Relatively expensive compared to adding/removing graph nodes
- May be replicated by node owners (e.g. ardour) that have internal ordering
- Feedback cycles need special handling

Audio “Paralellism”

- Many audio graphs are 100% serial
- Even the ones with some parallel aspects have serial aspects too

Parallel Graph Execution



Activation Flow

- Don't compute order before execution
  - Determine order “on the fly”

Activation Flow 2

- For each client, counter = number of input ports
- Find all graph node owners with no input ports or no connected input ports
- Execute these graph nodes
- Mark them as “executed”

Activation Flow 3

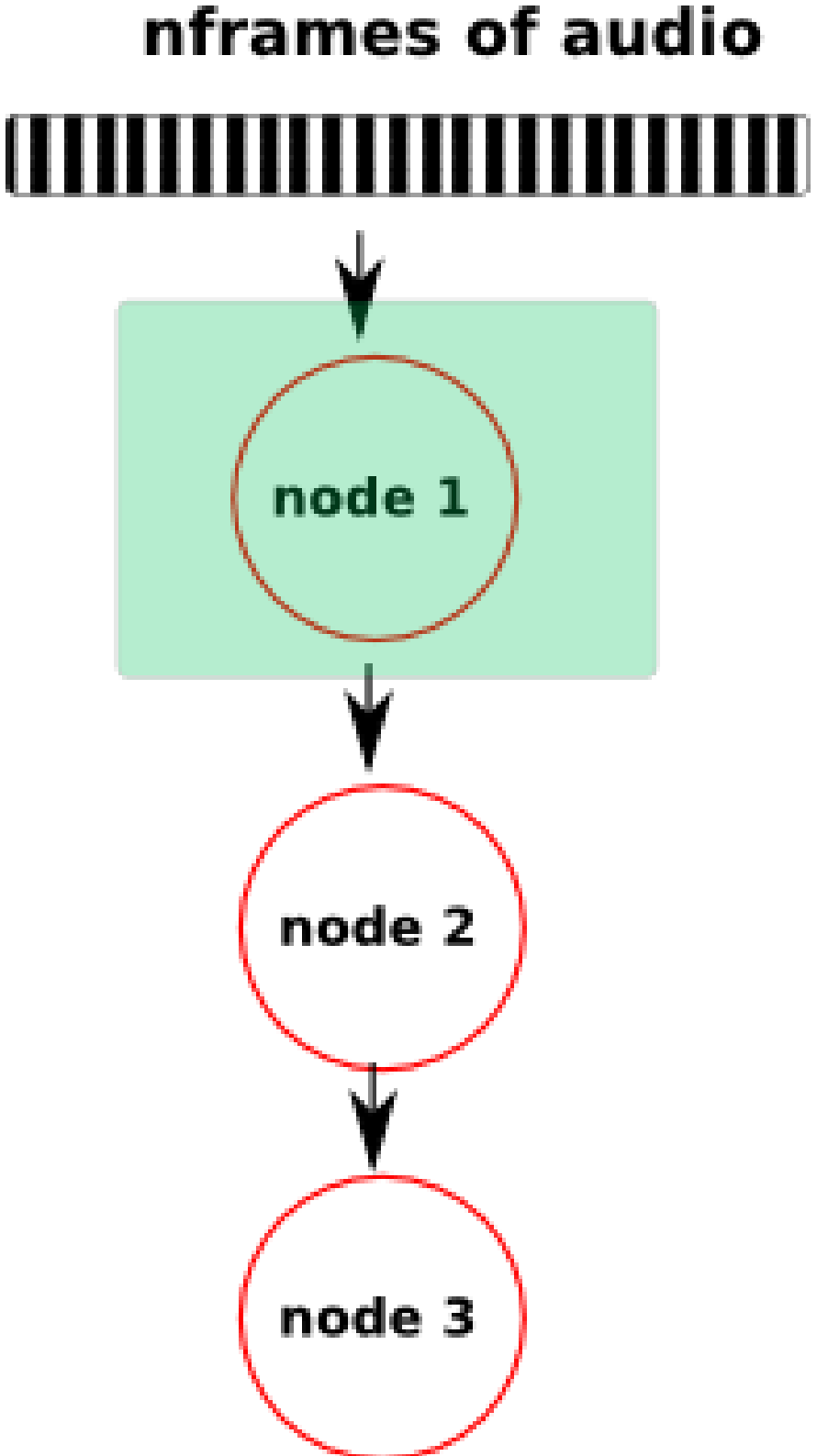
- Each just executed client decrements the counter on every other client connected to its outputs (1 per port connection)
- Find all clients where the counter is zero (and they have not executed already)
- Repeat cycle until there are no waiting clients left.

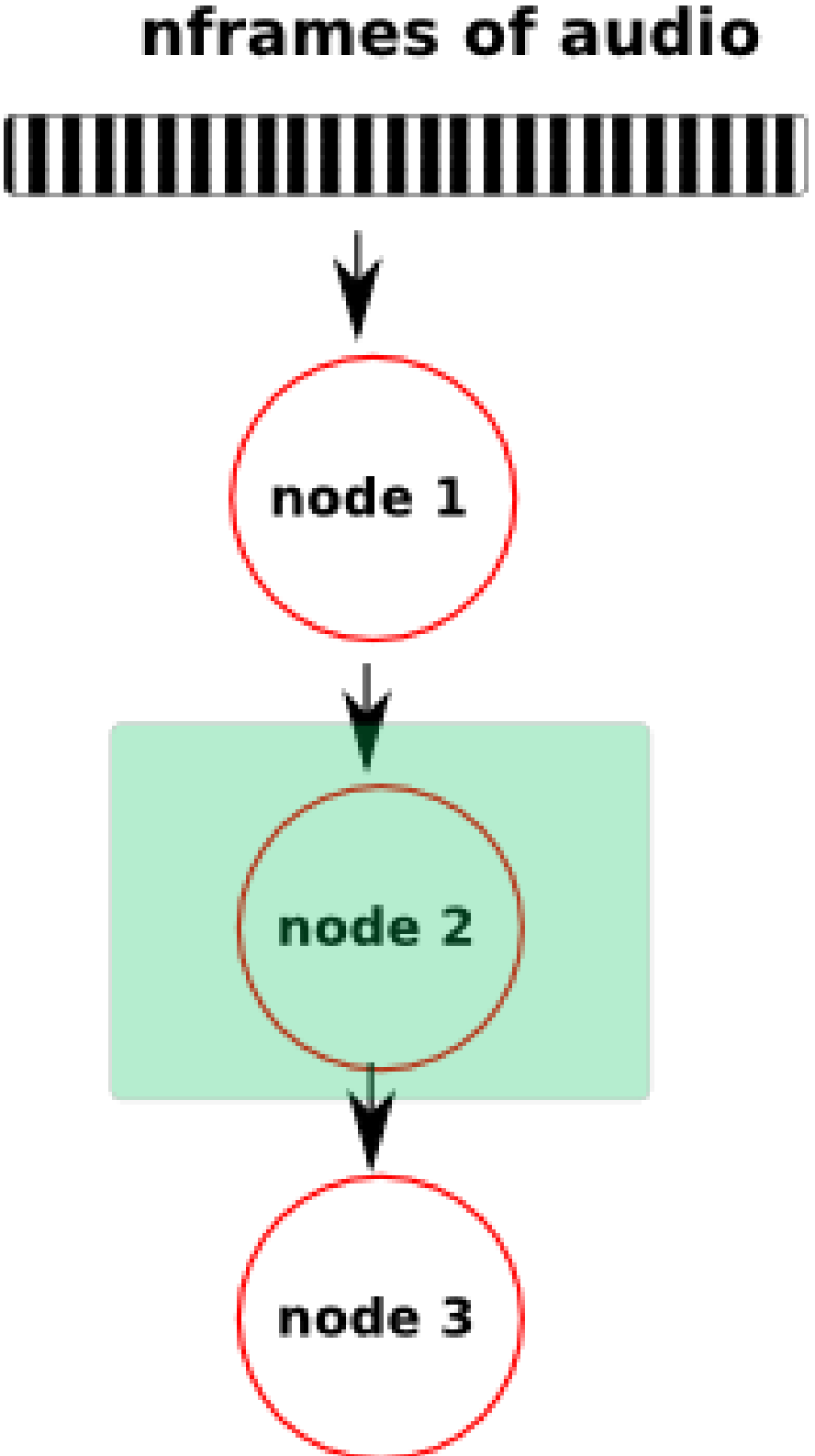
Benefits

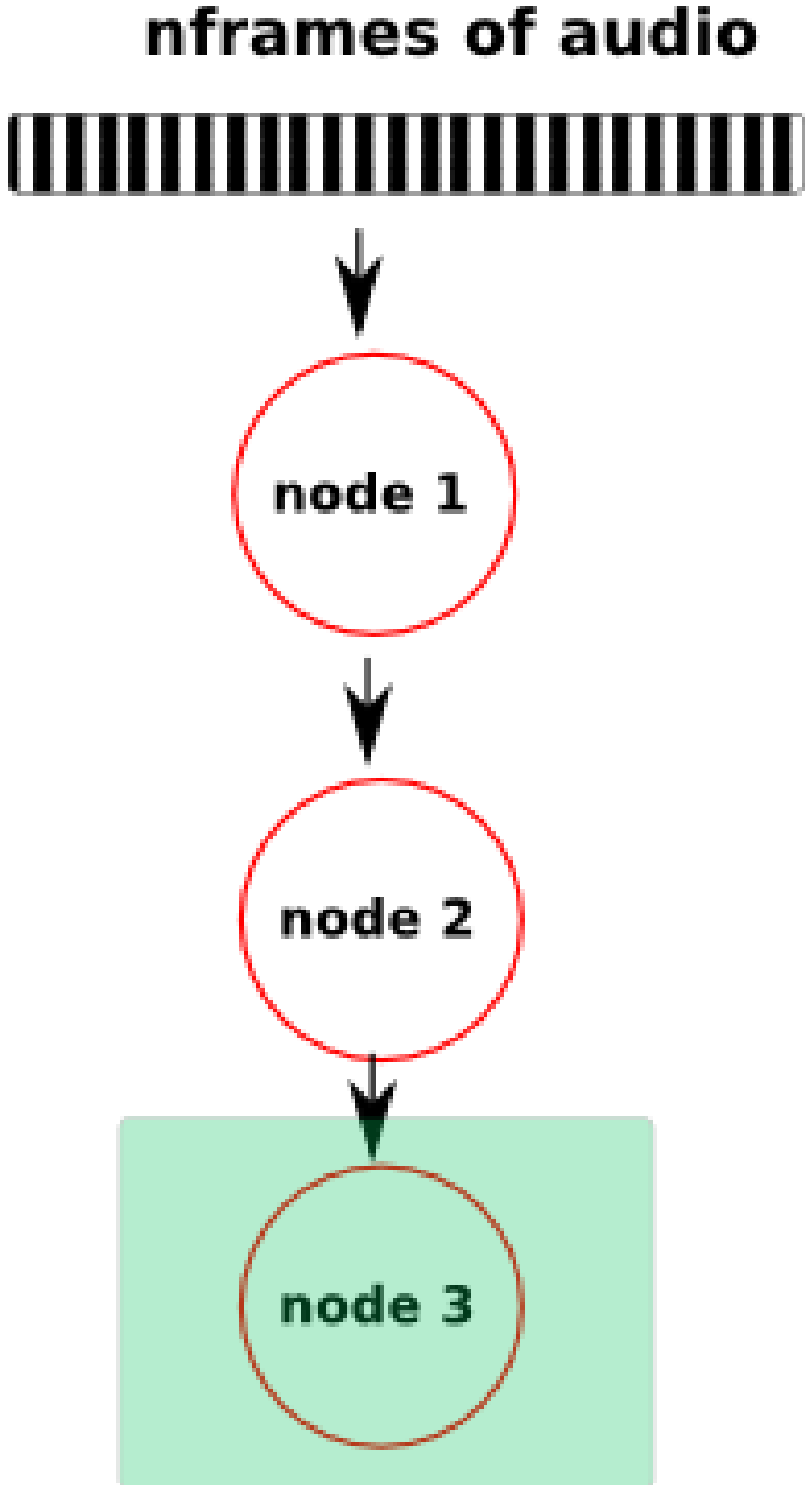
- Cheap when changing the graph, low cost at graph execution time
-

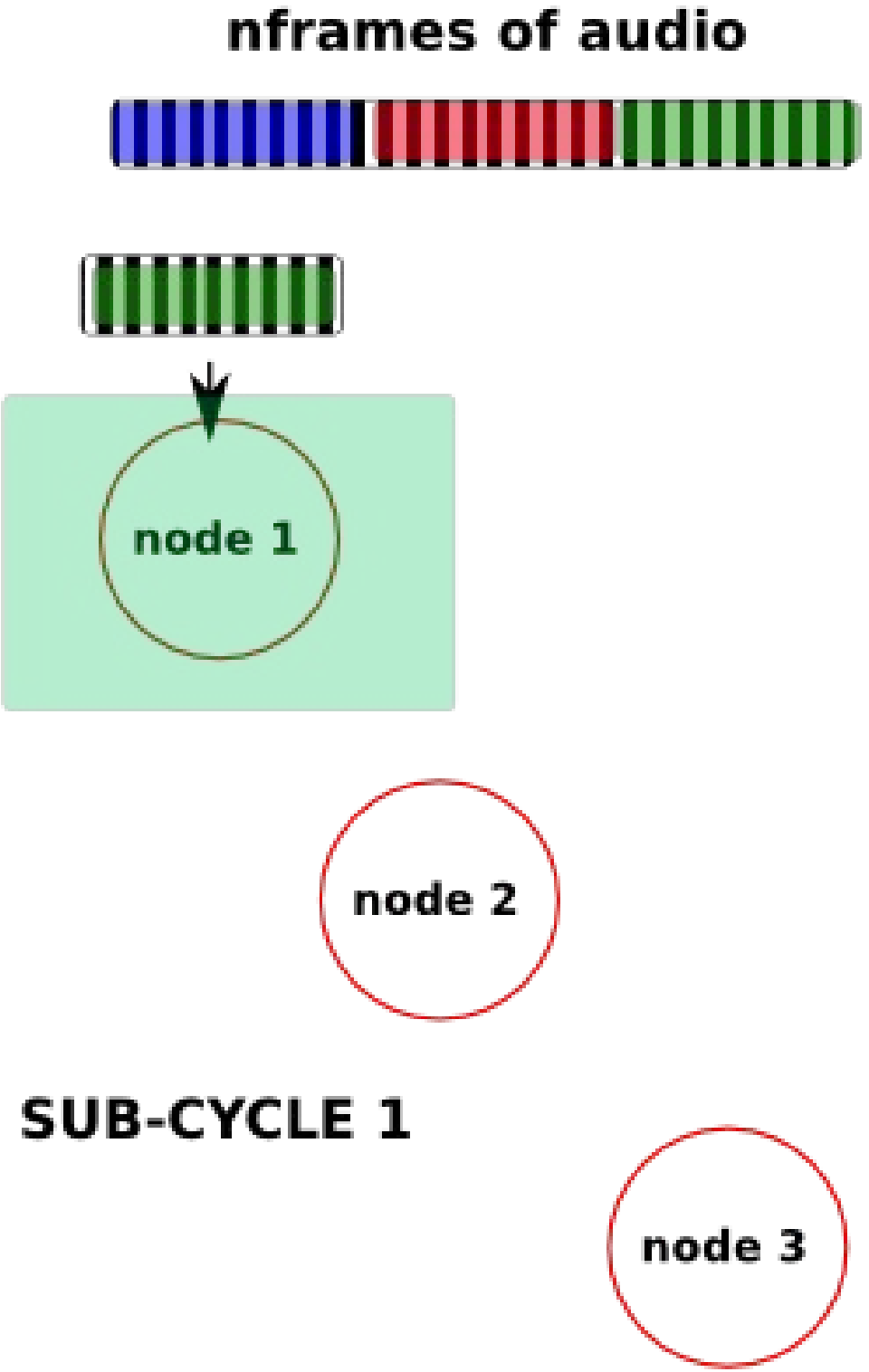
Artificial Parallelism

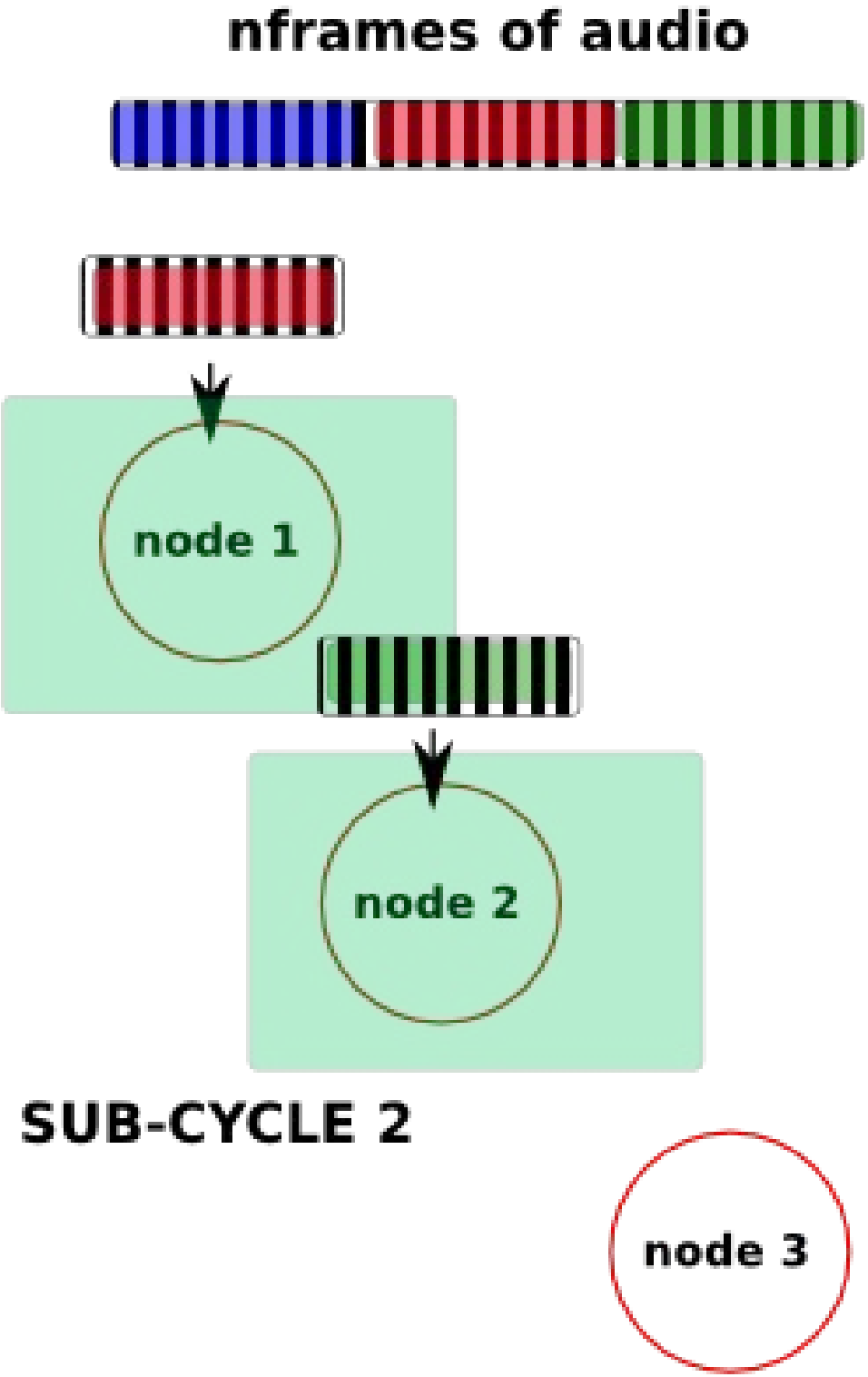
- Slicing up each block of audio

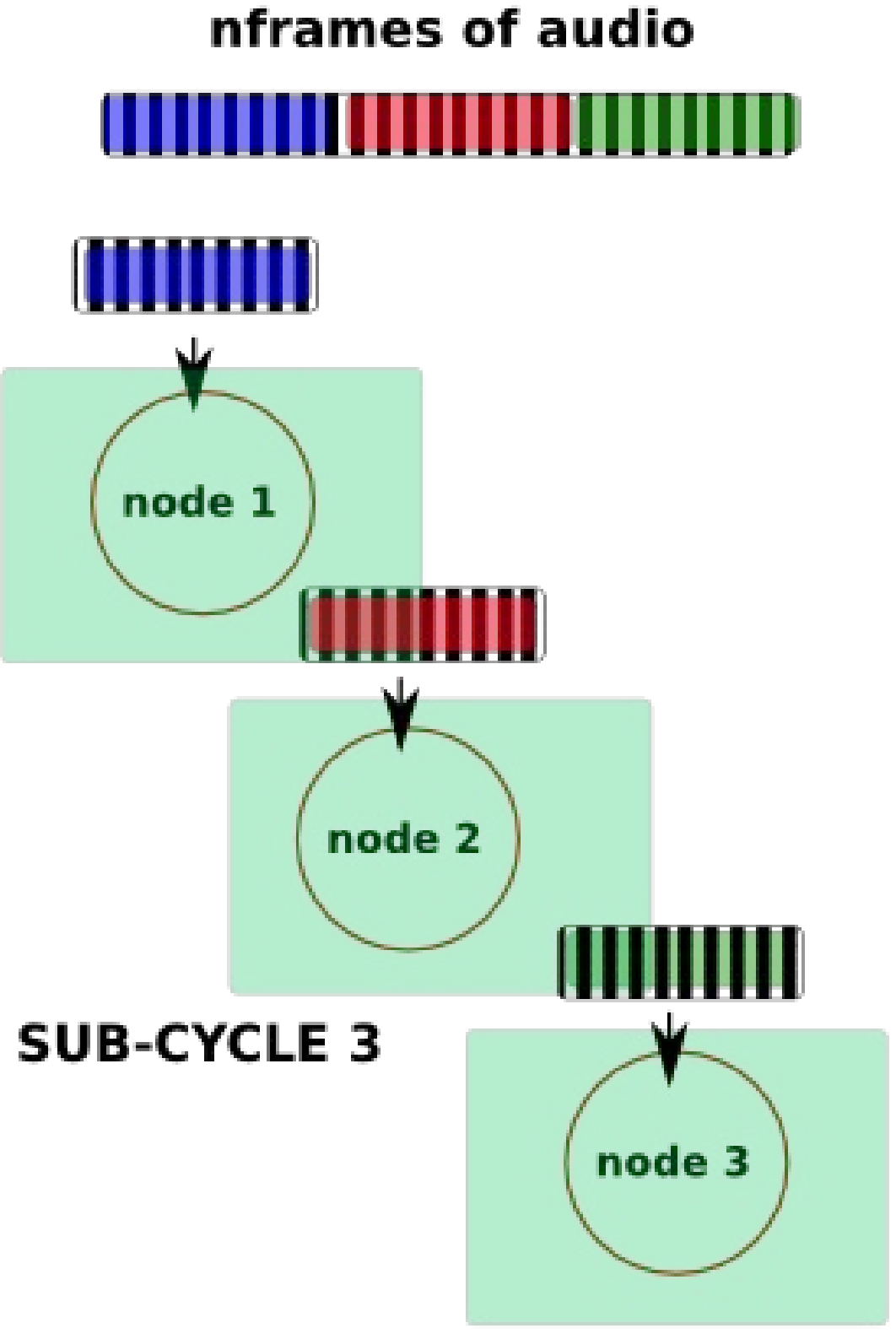

nframes of audio
node 1
node 2
node 3

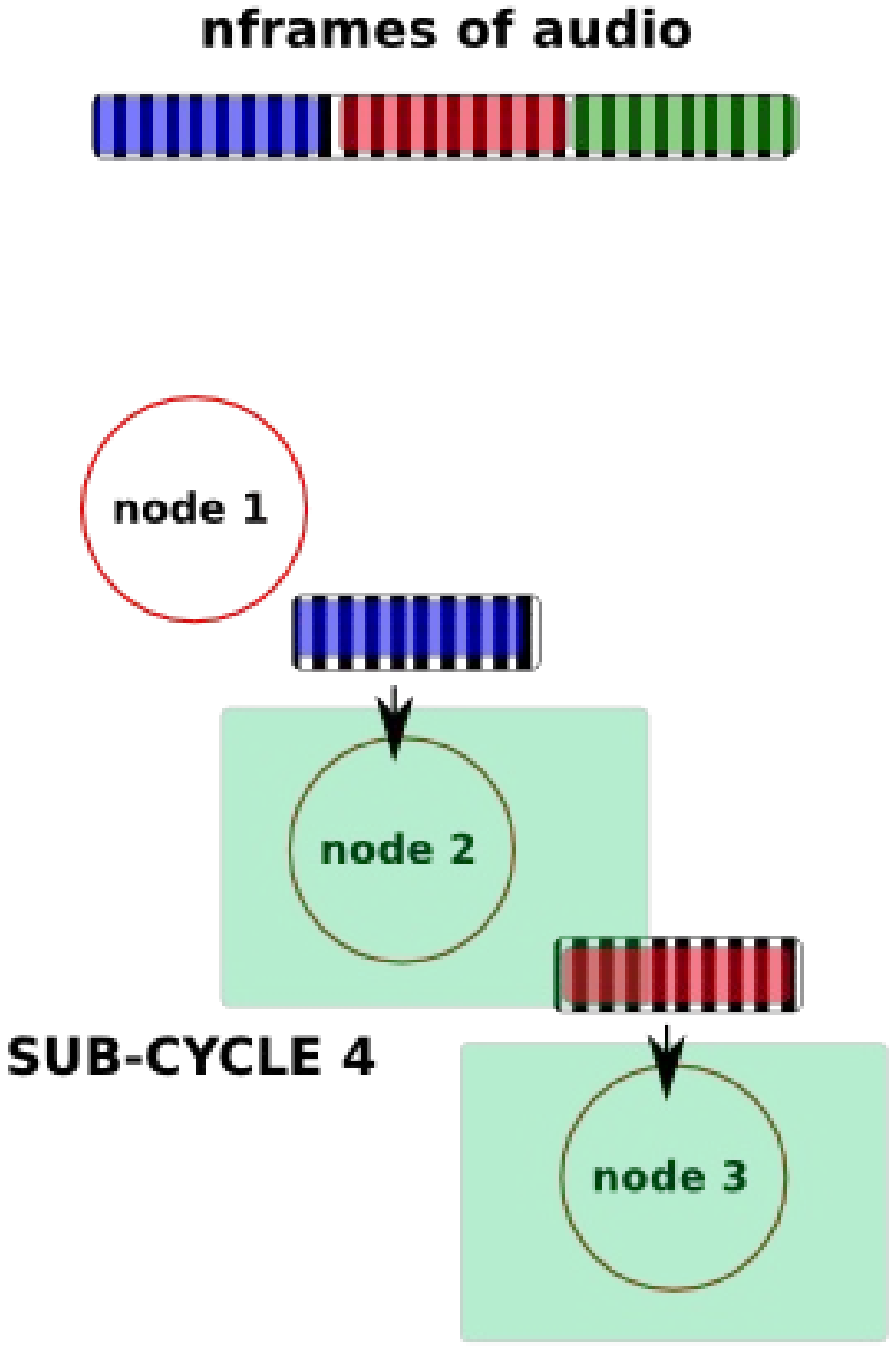

nframes of audio
node 1
node 2
node 3

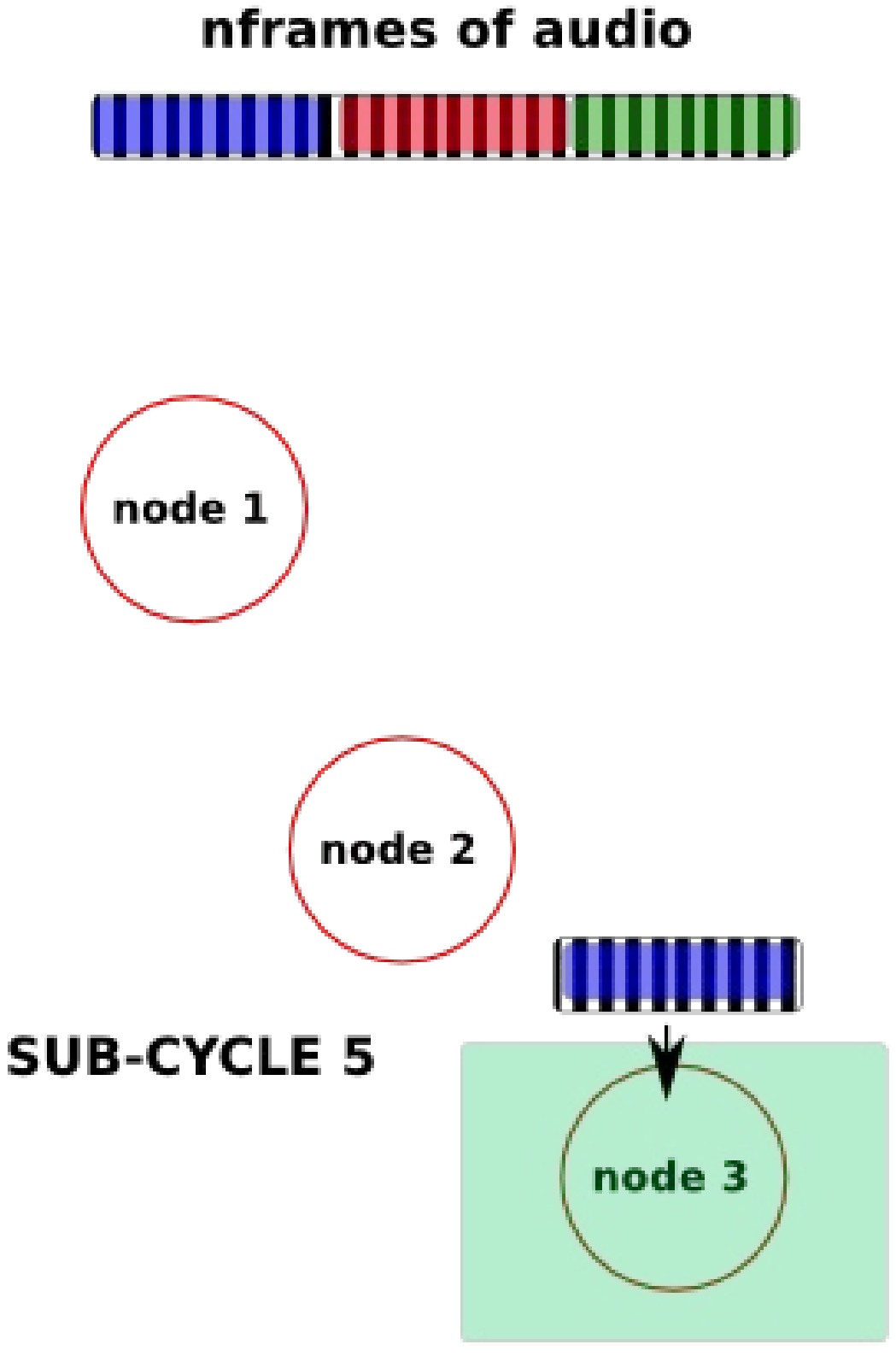

nframes of audio
node 1
node 2
node 3

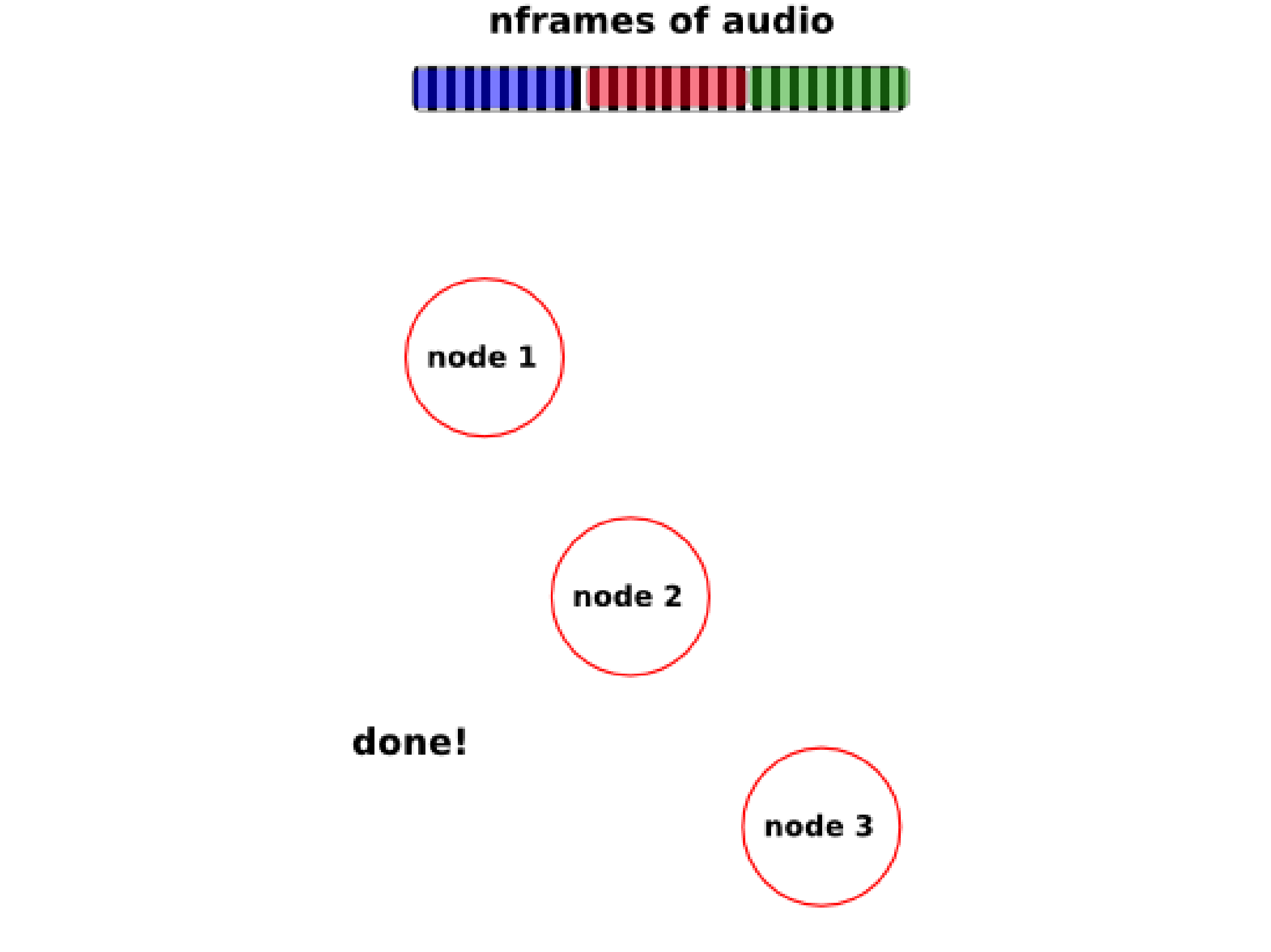

nframes of audio
node 1
node 2
SUB-CYCLE 1
node 3


nframes of audio
node 1
node 2
SUB-CYCLE 2
node 3


nframes of audio
node 1
node 2
node 3
SUB-CYCLE 3


nframes of audio
node 1
node 2
SUB-CYCLE 4
node 3


nframes of audio
node 1
node 2
SUB-CYCLE 5
node 3


nframes of audio
node 1
node 2
done!
node 3

Data Parallelism

- Same Instruction Multiple Data = SIMD
- Peak computation
- Gain & Pan

gain computation

```
for (n = 0; n < nframes; ++n) {
        audio_buffer[n] *= gain;
}
```

- 1 instruction cycle per audio sample (ignoring load/store to memory)
- Can we do better?

Peak Computation

```
for (n = 0; n < frames; ++n) {
      if (audio_buffer[n] > current_peak_max) {
            current_peak_max = audio_buffer[n];
      }
      if (audio_buffer[n] < current_peak_max) {
            current_peak_min = audio_buffer[n];
      }
}
```

- 2 conditionals per sample

Peak Computation 2

```
for (n = 0; n < nframes; ++n) {
     float abs_sample = fabs (audiobuffer[n]);
        if (abs_sample > current_peak) {
            current_peak = abs_sample;
        }
}
```

- Looks better – 1 conditional per sample
- How fast is fabs?

Peak Computation 3

```
current_peak = max_abs_of (audio_buffer, nframes);
```

- Would be nice!
- How about:

```
current_peak = max_abs_of (audio_buffer, 4);
```

- This can be done ...
- SSE/SSE2 processing unit on modern intel processors operates on 4 values at one time
- Old Altivec late-model PPC macs did the same
- Provides a variety of useful and **very** complicated operations

Peak Computation 4

```
for (n = 0; n < nframes; n += 4) {
        max_abs_of (&audio_buffer[n], current_peak);
}
```

- This is illustrative, not actual code
- Handling non-multiple of 4 can be an issue
- At least 4x faster
- Can be 10-30x faster in the real world

Gain Computation Revisited

```
for (n = 0; n < nframes; n += 4) {
        sse_multiply (&audio_buffer[n], gain);
}
```

- This is not threads and it doesn't even look parallel
- It **is** parallel and its very very powerful
- With some audio block sizes, this can save 30% of the execution time in a program like a DAW.
- SSE ops are often faster than main CPU equivalent

SIMD/SSE the bad news

- Very hard to understand documentation unless you have a lot of experience with low level processor architecture
- XMM “intrinsics” - compiler provided functions hide some of the complexity but not all
- When performance matters, its worth using this stuff

Next Week

- Time: synchronization, DLL's, “now”, latency
- 1 or 2 more project presentations