Semantic Aspects of Parallelism for SuperCollider

Tim BLECHMANN
Vienna, Austria
tim@klingt.org

Abstract

Supernova is a new implementation of the SuperCollider server scsynth, with a multi-threaded audio synthesis engine. To make use of this thread-level parallelism, two extensions have been introduced to the concept of the SuperCollider node graph, exposing parallelism explicitly to the user. This paper discusses the semantic inplications of these extensions.

Keywords

SuperCollider, Supernova, parallelism, multi-core

1 Introduction

These days, the development of audio synthesis applications is mainly focussed on off-the-shelf hardware and software. While some embedded, low-power or mobile systems use single-core CPUs, most computer systems which are actually used in musical production use multi-core hardware. Except for some netbooks, most laptops use dual-core CPUs, single-core workstations are getting rare.

Traditionally, audio synthesis engines are designed to use a single thread for audio computation. In order to use multiple CPU cores for audio computation, this design has to be adapted by parallelizing the signal processing work.

This paper is divided into the following sections: Section 2 describes the SuperCollider node graph which is the base for the parallelization of Supernova. Sections 3 introduces the Supernova extensions to SuperCollider with a focus on their semantic aspects. Section 4 discusses different approaches of other parallel audio synthesis systems.

2 SuperCollider Node Graph

SuperCollider has a distinction between instrument definitions, called **SynthDefs**, and their instantiations, called **Synths**. Synths are organized in **groups**, which are linked lists of **nodes** (synths or nested groups). The groups therefore form a hierarchical tree data structure, the **node graph** with a group as root of the tree.

Groups are used for two purposes. First, they define the order of execution of their child nodes, which are evalutated sequentially from head to tail using a depth-first traversal algorithm. The node graph therefore defines a total order, in which synths are evaluated. The second use case for groups is to structure the audio synthesis and to be able to address multiple synths as one entity. When sending a node command to a group it is applied to all its child nodes. Groups can be moved inside the node graph like a single node.

2.1 Semantic Constraints for Parallelization

The node graph is designed as data structure for structuring synths in a hierarchical manner. Traversing the tree structure is used to determine the order of execution, but it does not contain any notion of parallelism. While synths may be able to run in parallel, it is impossible for the synthesis engine to know this in advance. Synths do not communicate with each other directly, but instead they use global busses to exchange audio data. So any automatic parallelization would have to create a dependency graph depending on the access pattern of synths to global resources. The current implementation lacks a possibility to determine, which global resources are accessed by a synth. But even if it would be possible, the resources which are accessed by a synth are not constant, but can change at control rate or even at audio rate. Introducing automatic parallelization would therefore introduce a constant overhead and the parallelism would be limited by the granularity in which resource access could be predicted by the runtime system.

Using pipelining techniques to increase the throughput would only be of limited use, ei-

ther. The synthesis engine dispatches commands at control rate and during the execution of each command, it needs to have a synchronized view of the node graph. In order to implement pipelining across the boundaries of control rate blocks, a speculative pipelining with a rollback mechanism would have to be used. This approach would only be interesting for non-realtime synthesis. Introducing pipelining inside control-rate blocks would only be of limited use, since control rate blocks are typically small (usually 64 samples). Also the whole unit generator API would need to be restructured, imposing considerable rewrite effort.

Since neither automatic graph parallelization nor pipelining a feasible, we introduced new concepts to the node graph in order to expose parallelism explicitly to the user.

3 Extending the SuperCollider Node Graph

To make use of thread-level parallelism, Supernova introduces two extensions to the SuperCollider node graph. This enables the user to formulate parallelism explicitly when defining the synthesis graph.

3.1 Parallel Groups

The first extension to the node graph are **parallel groups**. As described in Section 2, groups are linked lists of nodes which are evaluated in sequential order. Parallel groups have the same semantics as groups, but with the exception, that their child nodes are not ordered. This implies that they can be executed in in separate threads. This concept is similar to the **SEQ** and **PAR** statements, which specify blocks of sequential and parallel statements in the concurrent programming language [Hyde, 1995].

Parallel groups are very easy to use in existing code. Especially for additive synthesis or granular synthesis with many voices, it is quite convenient to instantiate synths inside a parallel groups, especially since many users already use groups for these use cases in order to structure the synthesis graph. For other use cases like polyphonic phrases, all independent phrases could be computed inside groups, which are themselves part of a parallel group.

Listing 1 shows a simple example, how parallel groups can be used to write a simple polyphonic synthesizer of 4 synths, which are evaluated before a effect synth.

3.2 Satellite Nodes

Parallel groups have one disadvantage. Each member of a parallel group is still synchronized with two other nodes, it is executed after the parallel group's predecessor and before its successor. For many use cases, only one relation is actually required. Many generating synths can be started without waiting for any predecessor, while synths for disk recording or peak followers for GUI applications can start running after their predecessor has been executed, but no successor has to wait for its result.

These use cases can be formulated using **satellite nodes**. These satellite nodes, are nodes which are in dependency relation with only one reference node. The resulting dependency graph has a more fine-grained structure, compared to a dependency graph, which is only using parallel groups.

Listing 2 shows, how the example of Listing 1 can be formulated with satellite nodes under the assumption, that none of the generator synths depends on the result of any earlier synth. Instead of packing the generators into a parallel group, they are simply defined as satellite predecessors of the effect synth.

It is even possible to prioritize dependency graph nodes to optimize graph progress. In order to achive the best throughput, we need to ensure, that there are always at least as many parallel jobs available as audio threads. To ensure this, a simple heuristic can be used, which always tries to increase the number of jobs, that are actually runnable.

- Nodes with successors have a higher priority than nodes without.
- Nodes with successors early in the dependency graph have a high priority.

These rules can be realized with a heuristic that splits the nodes into three categories

Listing 1: Parallel Group Example

```
var generator_group, fx;
generator_group = ParGroup.new;
4.do {
    Synth.head(generator_group,
                \myGenerator)
};
fx = Synth.after(generator_group,
                  \myFx);
```

with different priorities: 'regular' nodes having the highest priority, satellite predecessors with medium priority and satellite successors with low priority. While it is far from optimal, this heuristic can easily be implemented with three lock-free queues, so it is easy to use it in a real-time context.

3.3 Common Use Cases & Library Integration

The SuperCollider language contains a huge class library. Some parts of the library are designed to help with the organization of the audio synthesis like the pattern sequencer library or the Just-In-Time programming environment JITLIB.

The pattern sequencer library is a powerful library, that can be used to create sequences of Events. Events are dictionaries, which can be interpeted as musical events, with specific keys having predefined semantics as musical parameters [McCartney,]. Events may contain are the keys `group` and `addAction`, which if present are used to specify the position of a node on the server. With these keys, both parallel groups and satellite nodes can be used from a pattern environment. In many cases, the pattern sequencer library is used in a way that the created synths are mutually independent and do not require data from other synths. In these cases both parallel groups and satellite predecessors can safely be used.

The situation is a bit different with JITLIB. When using JITLIB, the handling of the synthesis graph is completely hidden from the user, since the library wraps every syntesis node inside a proxy object. JITLIB nodes communicate with each other using global busses. This approach makes it easy to take the output of one node as input of another and to quickly reconfigure the synthesis graph. JITLIB therefore requires a deterministic order for the read/write access to busses, which cannot be guaranteed when instantiating nodes in parallel groups, unless additional functionality is implemented to read always those data, which are written during the previous cycle. Satellite nodes cannot be used to model the data flow between JITLIB nodes, since they cannot be used to formulate cycles.

Listing 2: Satellite Node example

```
var fx = Synth.new(\myFx);

4.do {
    Synth.preceeding(fx,
        \myGenerator)
};
```

4 Related Work

During the last few years, support for multicore audio synthesis has been introduced into different systems, that impose different semantic constraints.

4.1 Max/FTS, Pure Data & Max/MSP

One of the earliest computer-music systems, the Ircam Signal Processing Workstation (ISPW) [Puckette, 1991], used the Max dataflow language to control the signal processing engine, running on a multi-processor extension board of a NeXT computer. FTS was implementing explicit pipelining, so patches could be defined to run on a specific CPU. When audio data was sent from one CPU to another, it was delayed by one audio block size.

Recently a similar approach has been implemented for Pure Data [Puckette, 2008]. The `pd~` object can be used to load a subpatch as a separate process. Moving audio data between parent and child process adds one block of latency, similar to FTS. Therefore it is not easily possible to modify existing patches without changing the semantics, unless a latency compensation is taken into account.

The latest versions of **Max/MSP** contains a `poly~` object, which can run several instances of the same abstraction in multiple threads. However, it is not documented, if the signal is delayed by a certain ammount of samples or not. And since only the same abstraction can be distributed to multiple processors, it is not a general purpose solution.

An automatic parallelization of max-like systems is rather difficult to achieve, because max-graphs have both explicit dependencies (the signal flow) and implicit ones (resource access). In order to keep the semantics of the sequential program, one would have to introduce implicit dependencies between all objects, which access a specific resource.

4.2 CSound

Recent versions of CSound implement automatic parallelization in order to make use of multicore hardware [ffitch, 2009]. This is feasible, because the CSound parser has a lot of

knowledge about resource access patterns and the instrument graph is more constrained compared to SuperCollider. Therefore the CSound compiler can infer many dependencies automatically, but if this is not the case, the sequential implementation needs to be emulated.

The automatic parallelization has the advantage, that existing code can make use of multicore hardware without any modifications.

4.3 Faust

Faust supports backends for parallelization, an open-mp based code generator and a custom work-stealing scheduler [Letz et al., 2010]. Since Faust is only a signal processing language, with little notion of control structures. Since Faust is a compiled language, it cannot be used to dynamically modify the signal graph.

5 Conclusions

The proposed extensions to the SuperCollider node graph enable the user to formulate signal graph parallelism explicitly. They integrate well into the concepts of SuperCollider and can be used to parallelize many use cases, which regularly appear in computer music applications.

References

John ffitch. 2009. Parallel Execution of Csound. In *Proceedings of the International Computer Music Conference.*

Daniel C. Hyde, 1995. *Introduction to the programming language Occam.*

Stphane Letz, Yann Orlarey, and Dominique Fober. 2010. Work Stealing Scheduler for Automatic Parallelization in Faust. In *Proceedings of the Linux Audio Conference.*

James McCartney. *SuperCollider Manual.*

Miller Puckette. 1991. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, 15(3):68–77.

Miller Puckette. 2008. Thoughts on Parallel Computing for Music. In *Proceedings of the International Computer Music Conference.*