supernova, a multiprocessor-aware synthesis server for SuperCollider

Tim BLECHMANN
Vienna, Austria
tim@klingt.org

Abstract

SuperCollider [McCartney, 1996] is a modular computer music system, based on an object-oriented real-time scripting language and a standalone synthesis server. supernova is a new implementation of the SuperCollider synthesis server, providing an extension for multi-threaded signal processing. With adding one class to the SuperCollider class library, the parallel signal processing capabilities are exposed to the user.

Keywords

SuperCollider, multi-processor, real-time

1 Introduction

In the last few years, multi- and many-core computer architectures nearly replaced single-core CPUs. Increasing the single-core performance requires a higher power consumption, which would lower the performance per watt. The computer industry therefore concentrated on increasing the number of CPU cores instead of single-core performance. These days, most mobile computers use dual-core processors, while workstations use up to quad-core processors with support for Simultaneous Multithreading.

Most computer-music applications still use a single-threaded programming model for the signal processing. Although multi-processor aware signal processing was pioneered in the late 1980s, most notably with IRCAM's ISPW [Puckette, 1991], most systems, that are commonly used these days, have limited support for parallel signal processing. Recently, some systems introduced limited multi-processor capabilities, e.g. PureData, using a static pipelining technique similar to the ISPW [Puckette, 2008], adding a delay of one block size (usually 64 samples) to transfer data between processors. Jack2 [Letz et al., 2005] can execute different clients in parallel, so one can manually distribute the signal processing load to different applications. It can also be used to control multiple SuperCollider server instances from the same language process.

This paper is divided into the following sections. Section 2 gives an introduction to the difficulties when parallelizing computer music applications, Section 3 describes the general architecture of SuperCollider and the programming model of a SuperCollider signal graph. Section 4 introduces the concept of 'parallel groups' to provide multi-processor support for SuperCollider signal graphs. Section 5 gives an overview of the supernova architecture, Section 6 describes the current limitations.

2 Parallelizing Computer Music Systems

Parallelizing computer music systems is not a trivial task, because of several constraints.

2.1 Signal Graphs

In general, signal graphs are data-flow graphs, a form of directed acyclic graphs (DAGs), where each node does some signal processing, based on its predecessors. When trying to parallelize a signal graph, two aspects have to be considered. Signal graphs may have a huge number of nodes, easily reaching several thousand nodes. Traversing a huge graph in parallel is possible, but since the nodes are usually small building blocks ('unit generators'), processing only a few samples, the synchronization overhead would outweigh the speedup of parallelizing the traversal. To cope with the scheduling overhead, Ulrich Reiter and Andreas Partzsch developed an algorithm for clustering a huge node graph for a certain number of threads [Reiter and Partzsch, 2007].

While in theory the only ordering constraint between graph nodes is the graph order (**explicit order**), many computer music systems introduce an **implicit order**, which is defined by the order, in which nodes access **shared resources**. This implicit order changes

with the algorithm, that is used for traversing the graph and is undefined for a parallel graph traversal. Implicit ordering constraints make it especially difficult to parallelize signal graphs of max-like [Puckette, 2002] computer music systems.

2.2 Realtime Constraints

Computer Music Systems are realtime systems with low-latency constraints. The most commonly used audio driver APIs use a 'Pull Model'[1], that means, the driver calls a certain callback function at a monotonic rate, when the audio device provides and needs new data. The audio callback is a realtime function. If it doesn't meet the deadline and the audio data is delivered too late, a buffer under-run occurs, resulting in an audio dropout. The deadline itself depends on the driver settings, especially on the block size, which is typically between 64 and 2048 samples (roughly between 1 and 80 milliseconds). For low-latency applications like computer-based instruments playback latency below 20 ms are desired [Magnusson, 2006], for processing percussion instruments round-trip latencies for below 10 ms are required to ensure that the sound is perceived as single event. Additional latencies may be introduced by buffering in hardware and digital/analog conversion.

In order to match the low-latency real-time constraints, a number of issues has to be dealt with. Most importantly, it is not allowed to block the realtime thread, which could happen when using blocking data structures for synchronization or allocating memory from the operating system.

3 SuperCollider

supernova is designed to be integrated in the SuperCollider system. This Section gives an overview about the parts of SuperCollider's architecture, that are relevant for parallel signal processing.

3.1 SuperCollider Architecture

SuperCollider in its current version 3 [McCartney, 2002] is designed as a very modular system, consisting of the following parts (compare Figure 3.1):

Language (sclang) SuperCollider is based on an object-oriented scripting language with a real-time garbage collector, which is inspired by Smalltalk. The language comes with a huge class library, specifically designed for computer music applications. It includes classes to control the synthesis server from the language.

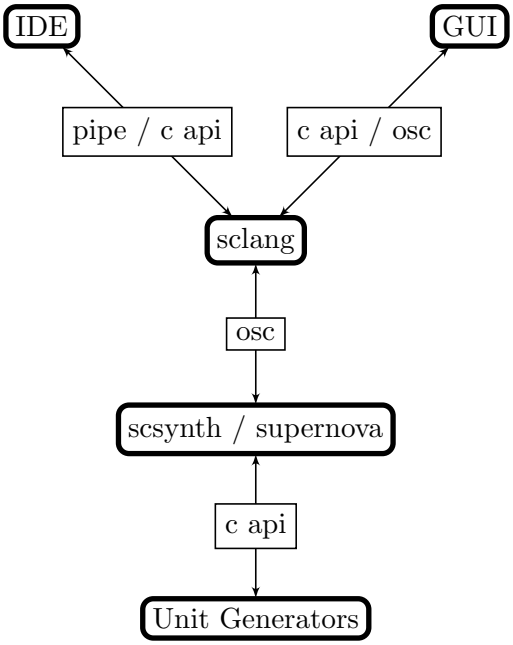

Figure 1: SuperCollider architecture

Synthesis Server (scsynth) The SuperCollider server is the signal processing engine. It is controlled from the Language using a simple OSC-based network interface.

Unit Generators Unit Generators (building blocks for the signal processing like oscillators, filters etc.) are provided as plugins. These plugins are shared libraries with a C-based API, that are loaded into the server at boot time.

IDE There are a number of integrated developer environments for SuperCollider source files. The original IDE is an OSX-only application, solely built as IDE for SuperCollider. Since it is OSX only, several alternatives exist for other operating systems, including editor modes for Emacs, Vim, gedit. Beside that, there is a plugin for Eclipse and a windows client called Psycollider.

GUI To build a graphical user interface for SuperCollider, two solutions are widely used. On OSX, the application provides Cocoa-based widgets. For other operating sys-

[1] e.g. Jack, CoreAudio, ASIO, Portaudio use a pull model

tems, one can use SwingOSC, which is based on Java's Swing widget toolkit.

This modularity makes it easy to change some parts of the system, while keeping the functionality of other parts. It is possible to use the server from other systems or languages, or control multiple server instances from one language process. Users can extend the language by adding new classes or class methods or write new unit generators (ugens).

3.2 SuperCollider Node Graph

The synthesis graph of the SuperCollider server is directly exposed to the language as a node graph. A **node** can be either a **synth**, i.e. a synthesis entity, performing the signal processing, or a **group**, representing a list of nodes. So, the node graph is internally represented as a tree, with groups at its root and inner leaves. Inside a group, nodes are executed sequentially, the node ordering is exposed to the user. Nodes can be instantiated at a certain position inside a group or moved to a different location.

4 Multi-core support with supernova

supernova is designed as replacement for the SuperCollider server scsynth. It implements the OSC interface of scsynth and can dynamically load unit generators. Unit Generators for scsynth and supernova are not completely source compatible, though the API changes are very limited, so porting SuperCollider ugens to supernova is trivial. The API difference is described in detail in Section 5.3.

4.1 Parallel Groups

To make use of supernova's parallel execution engine, the concept of **parallel groups** has been introduced. The idea is, that all nodes inside such a parallel group can be executed concurrently. Parallel groups can be nested, elements can be other nodes (synths, groups and parallel groups).

The concept of parallel groups has been implemented in the SuperCollider class library with a new `PGroup` class. Its interface is close to the interface of the `Group` class, but uses parallel groups on the server. Parallel groups can be safely emulated with traditional groups, making it easy to run code using the `PGroup` extension on the SuperCollider server.

4.2 Shared Resources

When writing SuperCollider code with the `PGroup` extension, the access to shared resources (i.e. buses and buffers) needs some attention to avoid data races. While with nodes in sequential groups the access to shared resources is ordered, this ordering constraint does not exist for parallel groups. E.g. if a sequential group G contains the synths S1 and S2, which both write to the bus B, S1 writes to B before S2, if and only if S1 is located before S2. If they are placed in a parallel group P, it is undetermined, whether S1 writes to B before or after S2. This may be an issue, if the order of execution affects the result signal. Summing buses would be immune to this issue, though.

Shared resources are synchronized at ugen level in order to assure data consistency. It is possible to read the same resource from different unit generators concurrently, but only one unit generator can write to a resource at the same time. If the order of resource access matters, the user is responsible to take care of the correct execution order.

5 supernova Architecture

The architecture of supernova is not too different from scsynth, except for the node graph interpreter. It is reimplemented from scratch in the c++ programming language, making heavy use of the stl, the boost libraries and template programming techniques. This Section covers the design aspects, that differ between the SuperCollider server and supernova.

5.1 DSP Threads

The multi-threaded dsp engine of supernova consists of the main audio callback thread and a number of worker threads, running with a realtime scheduling policy similar to the main audio thread. To synchronize these worker threads, the synchronization constraints, that are usually imposed (see Section 2.2), have to be relaxed. Instead of not using any blocking synchronization primitives at all, the dsp threads are allowed to use spin locks, that are blocking, but don't suspend the calling thread for synchronizing with other realtime threads. Since this reduces the parts of the program, that can run in parallel, this should be avoided whenever it is possible to use lock-free data structures for synchronization.

5.2 Node Scheduling

At startup supernova creates a configurable number of worker threads. When the main audio callback is triggered, it wakes the worker threads. All audio threads processes jobs from

a queue of runnable items. With each queue item, an activation count is associated, denoting the number of its predecessors. After an item has been executed, the activation counts of its successors are decremented and if it drops to zero, they are placed in the ready queue. The concept of using an activation count is similar to the implementation of Jack2 [Letz et al., 2005].

Items do not directly map to dsp graph nodes, since sequential graph nodes are combined to reduce the overhead for scheduling item. E.g. groups that only contain synths would be scheduled as one item.

5.3 Unit Generator Interface

The plugin interface of SuperCollider is based on a simple c-style API. The API doesn't provide any support for resource synchronization, which has to be added to ensure the consistency of buffers and buses. supernova is therefore shipped with a patched version of the SuperCollider source tree, providing an API extension to cope with this limitation. With this extension, reader-writer locks for buses and buffers are introduced, that need to be acquired exclusively for write access. The impact for the ugen code is limited, as the extension is implemented with C preprocessor macros, that can easily be disabled. So the adapted unit generator code for supernova can easily be compiled for scsynth.

In order to port a SuperCollider unit generator to supernova, the following rules have to be followed:

- ugens, that do not access any shared resources, are binary compatible.
- ugens, that are accessing buses have to guard write access with a pair of `ACQUIRE_BUS_AUDIO` and `RELEASE_BUS_AUDIO` statements and read access with `ACQUIRE_BUS_AUDIO_SHARED` and `RELEASE_BUS_AUDIO_SHARED`. Ugens like `In`, `Out` and their variations (e.g. `ReplaceOut`) fall in this category.
- ugens, that are accessing buffers (e.g. sampling, delay ugens) have to use pairs of `ACQUIRE_SNDBUF` or `ACQUIRE_SNDBUF_SHARED` and `RELEASE_SNDBUF` or `RELEASE_SNDBUF_SHARED`. In addition to that, the `LOCK_SNDBUF` and `LOCK_SNDBUF_SHARED` can be used to create a RAII-style lock, that automatically unlocks the buffer, when running out of scope.
- To prevent deadlocks, one should not lock more than one resource type at the same time. When locking multiple resources of the same type, only resources with adjacent indices are allowed to be locked. The locking order has to be from low indices to high indices.

6 Limitations

At the time of writing, the implementation of supernova still imposed a few limitations.

6.1 Missing Features

The SuperCollider server provides some features, that haven't been implemented in supernova, yet. The most important feature, that is still missing is non-realtime synthesis, that is implemented by scsynth. In contrast to the realtime mode, OSC commands are not read from a network socket, but from a binary file, that is usually generated from the language. Audio IO is done via soundfiles instead of physical audio devices. Other missing features are ugen commands for specific commands to specific unit generators and plugin commands to add user-defined OSC handlers. While these features exist, they are neither used in the standard distribution nor in the sc3-plugins repository[2].

6.2 Synchronization and Timing

As explained in Section 5.2 the audio callback wakes up the worker threads before working on the job queue. The worker threads are not put to sleep until all jobs for the current signal block have been run, in order to avoid some scheduling overhead. While this is reasonable, if there are enough parallel jobs for the worker threads, it leads to busy-waiting for sequential parts of the signal graph. If no parallel groups are used, the worker threads wouldn't do any useful work, or even worse, they would prevent the operating system to run threads with a lower priority on this specific CPU. This is neither very friendly to other tasks nor to the environment (power consumption). The user can make sure to parallelize the signal processing as much as possible and adapt the number of worker threads to

[2]The sc3-plugin repository (http://sc3-plugins.sourceforge.net/) is an svn repository of third-party unit generators, that is maintained independently from SuperCollider

match the demands of the application and the number of available CPU cores.

On systems with a very low worst-case scheduling latency, it would be possible to put the worker threads to sleep, instead of busy-waiting for the remaining jobs to finish. On a highly tweaked linux system with the RT Preemption patches enabled, one should be able to get worst-case scheduling latencies below 20 microseconds [Rostedt and Hart, 2007], which would be acceptable, especially if a block size of more than 128 samples can be used. On my personal reference machines, worst-case scheduling latencies of 100 µs (Intel Core2 Laptop) and 12 µs (Intel i7 workstation, power management, frequency scaling and SMT disabled) can be achieved.

7 Conclusions

supernova is a scalable solution for real-time computer music. It is tightly integrated with the SuperCollider computer music system, since it just replaces one of its components and adds the simple but powerful concept of parallel groups, exposing the parallelism explicitly to the user. While existing code doesn't make use of multiprocessor machines, it can easily be adapted. The advantage over other multiprocessor-aware computer music systems is its scalability. An application doesn't need to be optimized for a certain number of CPU cores, but can use as many cores as desired. It does not rely on pipelining techniques, so no latency is added to the signal. Resource access is not ordered implicitly, but has to be dealt with explicitly by the user.

8 Acknowledgements

I would like to thank James McCartney for developing SuperCollider and publishing it as open source software, Anton Ertl for the valuable feedback about this paper, and all my friends, who supported me during the development of supernova.

References

Stéphane Letz, Yann Orlarey, and Dominique Fober. 2005. Jack audio server for multiprocessor machines. In *Proceedings of the International Computer Music Conference*.

Thor Magnusson. 2006. Affordances and constraints in screen-based musical instruments. In *NordiCHI '06: Proceedings of the 4th Nordic conference on Human-computer interaction*, pages 441–444, New York, NY, USA. ACM.

James McCartney. 1996. SuperCollider, a new real time synthesis language. In *Proceedings of the International Computer Music Conference*.

James McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4):61–68.

Miller Puckette. 1991. FTS: A Real-time Monitor for Multiprocessor Music Synthesis. *Computer Music Journal*, 15(3):58–67.

Miller Puckette. 2002. Max at Seventeen. *Computer Music Journal*, 26(4):31–43.

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

Ulrich Reiter and Andreas Partzsch. 2007. Multi Core / Multi Thread Processing in Object Based Real Time Audio Rendering: Approaches and Solutions for an Optimization Problem. In *Audio Engineering Society 122th Convention*.

Steven Rostedt and Darren V. Hart. 2007. Internals of the RT Patch. In *Proceedings of the Linux Symposium*, pages 161–172.