SUPERNOVA - A SCALABLE PARALLEL AUDIO SYNTHESIS SERVER FOR SUPERCOLLIDER

Tim Blechmann
tim@klingt.org

ABSTRACT

SuperCollider [5] is a computer music system based on an object-oriented real-time scripting language and a separate audio synthesis server. The synthesis server is programmed using a sequential programming model and is only able to use one CPU core for audio synthesis, so it does not make full use of today's multi-core CPUs.

In order to overcome this limitation we have implemented Supernova, a drop-in replacement for the default synthesis server 'scsynth'. Supernova introduces extensions to the sequential programming mode, exposing parallelism explicitly to the SuperCollider language. The multi-threaded audio synthesis engine of Supernova is scalable and optimized for low-latency real-time applications.

1. INTRODUCTION

For many years the number of transistors per CPU has increased exponentially, roughly doubling every 18 months to 2 years. This behavior is usually referred to as 'Moore's Law'. Since the early 2000s, this does not necessarily increase the CPU performance any further, since the techniques that caused these performance gains have been maxed out [7]. Processors have been 'pipelined', executing instructions in a sequence of stages, which increases the throughput at the cost of instruction latency. Since the single stages require a reduced amount of logic, pipelined CPUs allow higher clock rates. These days, most processors use out-of-order execution engines, which can execute independent instructions in parallel. Using more transistors and further increasing the clock frequency for single CPU cores would cause a cubic growth in power consumption [2], imposing practical problems for cooling, particularly for mobile devices. While power consumption decreases with the die shrink, it has been suggested that this will reach its physical limits in the near future [3].

So instead of trying to increase the CPU speed, the computer industry started to increase the number of cores per CPU. These days, most mobile solutions use dual-core processors, while workstations are available with up to 8 cores. Some researchers expect the number of CPU cores to double every 18 months [1].

Parallel architectures are not necessarily new for computer music systems. In the early days of computer music, systems like the IRCAM Signal Processing Workstation (ISWP) [10], a NeXT-based computer with up to 24 Intel i860 coprocessors, was able to perform audio synthesis in real-time and was commonly used for the production of artistic works in the early 1990s. However the computer music systems that are in use these days mostly use a sequential programming model.

This paper is divided into the following sections. Section 2 gives an introduction to the levels of parallelism in computer music systems. Section 3 proposes two extensions to the SuperCollider node graph. Section 4 gives a rough overview on the architecture of Supernova, a replacement for the SuperCollider server scsynth with a concurrent audio synthesis engine. Section 5 presents and discusses benchmark results.

2. PARALLELIZING COMPUTER MUSIC SYSTEMS

There are different types of parallelism that applications can make use of. When discussing parallelism for multi-core processors, we will focus on **thread level parallelism** which describes the parallelization of an application into separate threads. However many audio synthesis engines can make use of **data level parallelism** using SIMD (single instruction, multiple data) instruction sets like SSE or Altivec for processing multiple samples in a single instruction. SIMD instructions are hardware-dependent, so they are usually generated by the compiler (or the low-level developer). Many recent CPUs use out-of-order execution engines, which means that they can execute multiple independent instructions at the same time. This type of parallelism is called **instruction level parallelism** and it is usually the responsibility of the instruction scheduler of the compiler to make optimal use of it. However there are some algorithms like using Estrin's scheme [6], which makes better use of instruction level parallelism for polynomial approximation than the commonly used Horner's scheme.

Both data level and instruction level parallelism can be used with a sequential programming model. In this section we discuss different approaches to the use of thread level parallelism for audio synthesis engines.

2.1. Pipelining

The basic approach for introducing parallelism into a sequential application is to use pipelining. The algorithm is split into sequential stages and each stage is computed in a separate thread. To achieve the optimal speedup, all pipeline stages should take the same CPU time and the number of pipeline stages should match the number of CPU cores. While pipelining is a simple technique to increase the throughput, it usually increases the latency. In terms of computer music systems this would increase the latency of a signal, which reduces its usability for real-time applications.

Nevertheless, pipelining is used in several parallel computer music systems. It was used in FTS [9], where subpatches could be assigned to certain CPUs, and crossing CPU boundaries introduced a delay of a signal vector. The same approach has recently been reimplemented in Pure Data using the `pd~` object [11] which creates a separate process for a subpatch. Jack2 [8], a multi-processor version of the Jack Audio Connection Kit, also implements pipelining to increase the throughput of its clients. To avoid the additional latency, Jack2 splits a single signal block into smaller blocks and runs its clients on these smaller block sizes.

This approach can only be used if the original block size is reasonably big, because the lowest reasonable size for the pipeline blocks is the size of a cache line. A second limiting factor is that the pipeline needs to be filled and emptied for processing each audio block.

2.2. Graph Parallelism

A fundamentally different approach is to traverse the signal graph in parallel. While this doesn't introduce any additional latency to the audio signal, it leads to some implementation problems. The signal graph may contain tens or thousands of nodes, depending on the granularity of the node graph; therefore it makes sense to distinguish between **coarse grained** and **fine grained** signal graph. I do not give a precise definition, but as a rule of thumb fine grained signal graphs can be seen as as graphs with a significant node scheduling overhead.

2.2.1. Parallelizing Fine Grained Graphs

Because of the scheduling overhead of fine grained signal graphs, it is not feasible to schedule each graph node separately, especially since sequential scheduling can be implemented very efficiently by iterating over a linearized data structure (the topologically sorted signal graph). Therefore one would need to spend some effort to combine multiple graph nodes to a single entity. While Ulrich Reiter and Andreas Partzsch have published an algorithm [12] to achieve this, it is rather impractical to use, since it does not take shared resources into account.

2.2.2. Parallelizing Coarse Grained Graphs

Coarse grained graphs do not need any graph clustering step, since by definition the scheduling overhead of its nodes can be neglected. A simple dataflow scheduling algorithm for coarse grained graphs has been implemented in Jack2 [4]. Each graph node is annotated with an activation count (initially, the number of predecessors it has) and a list of its successors. After a node has been evaluated, it decrements the activation count of its successors. If an activation count drops to zero, the corresponding node is ready for execution.

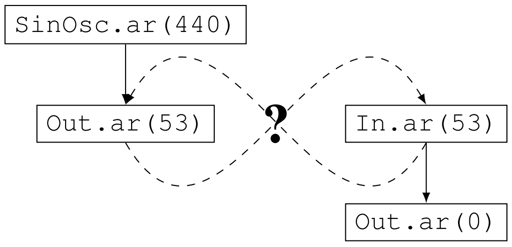

Figure 1: Signal graph with an implicit dependency

2.2.3. Graph Parallelism & Resource Handling

When introducing parallelism to an application, one needs to make sure that the semantics of the original program does not change. The signal graphs of many applications only contain information about **explicit dependencies**, which are caused by the signal flow. When graph nodes of an application access shared resources, the access order of the sequential program is usually determined by the topological sorting of the graph. To ensure the semantic correctness of the parallelized program, these **implicit dependencies** need to be added to the dependency graph.

Figure 1 shows a simple ugen graph, which is prone to implicit dependencies. The graph has two parts that could be evaluated in parallel if only explicit dependencies would be taken into account. It is not determined by the graph order whether `In.ar(53)` or `Out.ar(53)` should be evaluated first. If both parts of the graph were evaluated in parallel, it would be undefined whether the read or the write access to the bus would happen first. In fact, this is a race condition: during some DSP ticks the bus is read before, in other DSP ticks after it is written. This can corrupt the audio stream, since blocks may be missed or played back twice. In order to keep the semantics from the sequential version of the program, the implicit dependency from the sequential program needs to be added to the dependency graph.

3. EXTENDING THE SUPERCOLLIDER NODE GRAPH

The programming model of SuperCollider distinguishes between **unit generators** and **synths**. Unit generators are used to build larger entities (**synthdefs**), that can be instantiated on the server as synths. Using the terms of Section 2, unit generator graphs would qualify as fine grained graphs, while the synth graph could be seen as coarse grained.

However, SuperCollider does not have a proper notion of a dependency graph. Instead, its node graph models a tree hierarchy, with synths and **groups** as tree nodes and a group as its root. Groups are lists of nodes, which can be used to structure the audio synthesis and address multiple nodes as one entity. Listing 1 shows a small code example that would result in the node hierarchy shown in Figure 2.

Since the node graph is explicitly exposed to the language, the user needs to take care of the correct order of execution. While this imposes some responsibility on the user, it can be modified to explicitly specify parallelism. To achieve this, we propose two extensions to the SuperCollider node graph: parallel groups and satellite nodes.

3.1. Parallel Groups

The first approach to specify parallelism is the concept of **parallel groups**. Parallel groups would be available as `ParGroup` class in the SuperCollider language and have a semantics similar to groups. Like groups they can contain child nodes. However, instead of evaluating the child nodes sequentially, the order of execution is undefined, so all nodes can be evaluated in parallel. This provides the user with a simple facility to explicitly specify parallelism. Assuming that the generators of the earlier example can be evaluated in parallel, the code could be implemented using the proposed `ParGroup` class as shown in Listing 2. While the node hierarchy would be the same as shown in Figure 2, the dependency graph shown in Figure 3 is different.

Introducing parallel groups has the advantage of being compatible with scsynth. Scsynth can simply emulate parallel groups with sequential groups, since they provide all semantic properties as parallel groups.

3.2. Satellite Nodes

While parallel groups easily fit into the concept of the SuperCollider node graph, they impose some limitation to parallelism. Members of parallel groups are synchronized in two directions: they are evaluated after all predecessors of the parallel group have been evaluated and before all successors. For many applications one does not need to specify both directions of synchronization, but one synchronization constraint is sufficient. In the example above the generators need to be evaluated before the effect node, but they do not necessarily depend on a result of one of the predecessors of the parent (parallel) group. To express these single dependency relations, we introduce another concept that we call **satellite nodes**. Satellite predecessors have to be evaluated before their reference nodes, while satellite successors are evaluated after their reference node.

Listing 3 shows how satellite predecessors can be used: all generator synths are instantiated as satellite predecessors of the effect node, so they would be initially runnable. Typical use cases for satellite successors would be audio analysis synths like peak meters for GUI applications, or sound file recorders.

Since satellite nodes provide a facility to specify dependencies more accurately, the parallelism for many use cases can be increased. It is even be possible to dispatch satellite nodes at a lower priority in order to optimize graph throughput.

The combination of parallel group and satellite nodes should provide sufficient means to parallelize many use cases. They still do not model a dependency graph with arbitrary dependencies, so there are certain dependency graphs which may be cumbersome to formulate. But this limitation also avoids problems such as cyclic dependencies, and integrates well in the node hierarchy.

4. SUPERNOVA

Supernova is a parallel implementation of the SuperCollider server, that can be used as drop-in replacement for scsynth. It implements an extended OSC interface, which supports the necessary commands to instantiate parallel groups and satellite nodes. Supernova can dynamically load SuperCollider unit generators, although the source code needs to be slightly modified if the unit generator accesses resources like buffers or busses.

4.1. Resource Access

In scsynth, unit generators are known to be evaluated sequentially. Obviously, this is not the case for unit generators in Supernova. In order to ensure data consistency for concurrent execution, some care needs to be taken to achieve thread safety. The main data structures that are shared among unit generators are busses and buffers. To allow multiple readers for a resource, the unit generator

Listing 1: Using SuperCollider's Group class

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

Listing 2: Parallel Group Example

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

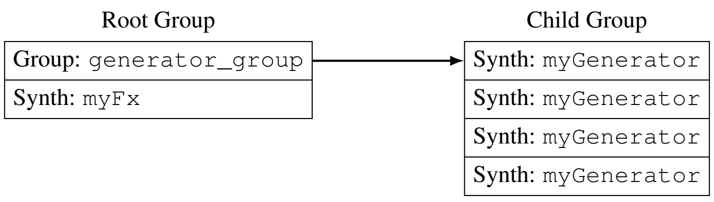

Figure 2: Node hierarchy for Listing 1

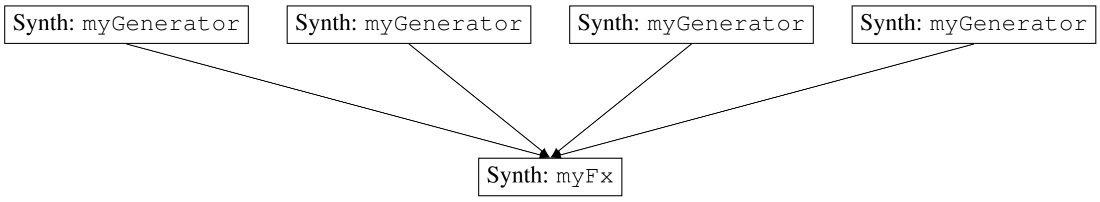

Figure 3: Dependency graph for Listing 2

API has been extended by adding reader-writer spin-locks for each bus or buffer. Before a unit generator can access a resource, it needs to acquire the corresponding spinlock. Since some unit generators require access to multiple resources, some care needs to be taken in order to prevent deadlocks. Therefore a simple locking policy is used: a total order of all resources is defined and locks need to be acquired in this order. If one lock cannot be acquired, all previously acquired locks need to be released before the locks will be acquired again.

While this ensures atomicity for write access, it does not take all the responsibility away from the user. For example, two synths may use the `Out.ar` ugen in parallel to write to the same bus without problems, since the actual order of the write access does not matter, but for `ReplaceOut.ar` the semantics would differ.

4.2. Dependency Graph

Internally, Supernova does not interpret the node graph directly as is done in scsynth, but the node graph is used to create a dependency graph data structure. This data structure does not have the notion of groups and synths any more, but its nodes contain sequences of synths. In this representation, sequential synths are combined into a single queue node to avoid the overhead of scheduling each synth as a single entity. While the construction of the dependency graph introduces some run-time overhead when the signal graph is changed, benchmarks suggest that it is reasonably low (tens of microseconds, depending on the size of the graph). So this does not significantly affect the real-time safety of the audio synthesis engine.

Listing 3: Satellite Node example

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

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

4.3. Limitations

For low-latency applications real-time computer music systems require a low worst-case scheduling latency. Therefore it is not feasible to use blocking primitives for synchronization of the main audio thread and the audio helper threads. Instead Supernova wakes all helper threads in the beginning of the main audio callback and all threads poll a lock-free stack, containing those queue nodes that are ready for execution. This greedy use of CPU resources is not friendly to other processes. Depending on the structure of the node graph a significant amount of CPU time could be spent in the busy waiting loops. Unless one uses highly tuned systems running the RT preemption patches for the Linux kernel, it seems to be the only way to dispatch threads quickly enough.

5. EXPERIMENTAL RESULTS

Supernova is designed for low-latency real-time operations. In order to evaluate the performance, we measured the execution times of the audio callback and stored them in a histogram with microsecond granularity. This approach has the advantage that it does not only show measure the thougput, but that it actually shows more detailed performance characteristics. For real-time audio synthesis, the speedup of the worst case is more interesting than the speedup of the average case, since a missed deadline would result in a possibly audible audio dropout.

The tests were carried out on an Intel Core i7 workstation, running Linux with RT preemption patches. Its worst-case scheduling latency was measured to be about 20 microseconds. Different test graphs were examined. Each graph layout was tested with up to 4 threads using parallel groups and is compared against sequential groups.

Figure 4 shows a typical histogram. One can observe

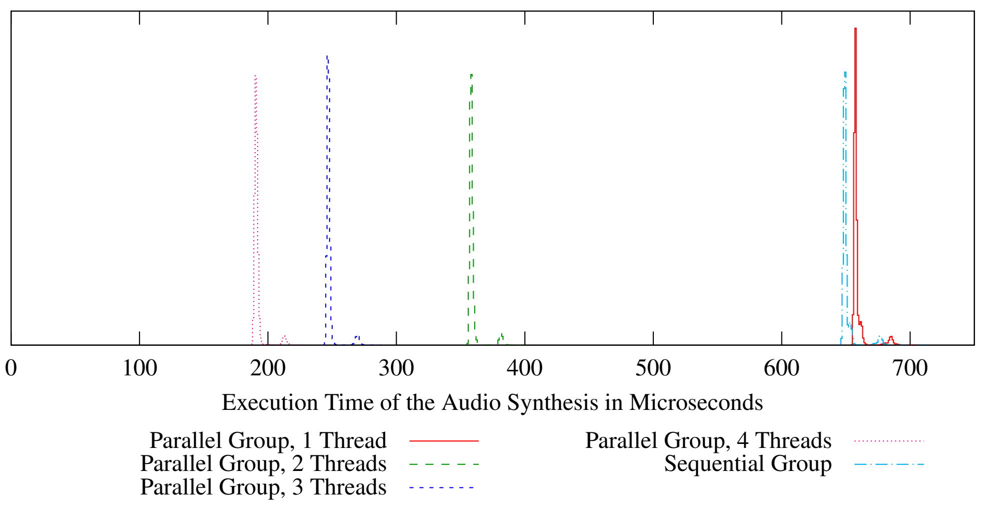

Figure 4: Execution Time Histogram, One Parallel Group with 256 Lightweight Synths

different aspects: the execution time for each test case shows little spread. Most of the histogram samples are found around one peak, which a second small peak roughly 20 microseconds after the first. The time difference between the first and second peak is in the order of magnitude of the worst-case scheduling latency that can be achieved by the workstation. So it is most likely a result of hardware effects. Since no samples can be found behind the second peak, the implementation can be considered as real-time safe.

Using the histograms, average-case and worst-case execution times can be determined and speedups can be computed. Figures 5 and 6 show the computed speedups for different use cases, both average-case and worst-case.

6. CONCLUSION

This paper introduces Supernova, a replacement for SuperCollider's default synthesis server scsynth. Supernova supports two extensions to the SuperCollider node graph, so that the user can explicitly express parallelism in the node hierarchy. Its multiprocessor aware synthesis engine is optimized for real-time audio synthesis, and scales well with the number of CPUs. The next release of SuperCollider will include Supernova as alternative to scsynth.

7. ACKNOWLEDGEMENTS

This paper is a revisited and extended version of the paper presented at the SuperCollider Symposium 2010. Many thanks to Dr. Dan Stowell and Dr. M. Anton Ertl for their valuable feedback and James McCartney for creating SuperCollider.

8. REFERENCES

[1] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov, "The impact of multicore on math software," in *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*. Springer-Verlag, 2006, pp. 1–10.

[2] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine, "The Intel Pentium M Processor: Microarchitecture and Performance," *Intel Technology Journal*, vol. 7, no. 02, pp. 21–36, 2003.

[3] L. B. Kish, "End of Moores law: thermal (noise) death of integration in micro and nano electronics," *Physics Letters A*, vol. 305, no. 3-4, pp. 144–149, 2002.

[4] S. Letz, Y. Orlarey, and D. Fober, "Jack audio server for multi-processor machines," in *Proceedings of the International Computer Music Conference*, 2005.

[5] J. McCartney, "SuperCollider, a new real time synthesis language," in *Proceedings of the International Computer Music Conference*, 1996.

[6] J.-M. Muller, *Elementary Functions: Algorithms and Implementation*. Birkhauser, 2006.

[7] K. Olukotun and L. Hammond, "The Future of Microprocessors," *Queue*, vol. 3, no. 7, pp. 26–29, 2005.

[8] Y. Orlarey, S. Letz, and D. Forber, "Multicore Technologies in Jack and Faust," in *Proceedings of the International Computer Music Conference*, 2008.

[9] M. Puckette, "Combining Event and Signal Processing in the MAX Graphical Programming Environment," *Computer Music Journal*, vol. 15, no. 3, pp. 68–77, 1991.

[10] ——, "FTS: A Real-time Monitor for Multiprocessor Music Synthesis," *Computer Music Journal*, vol. 15, no. 3, pp. 58–67, 1991.

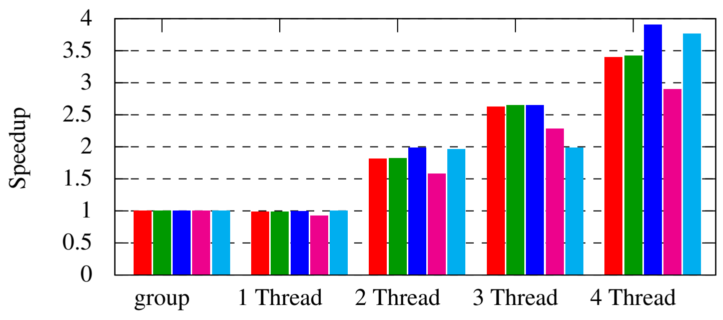

Figure 5: Average Case Speedup, for five different use cases

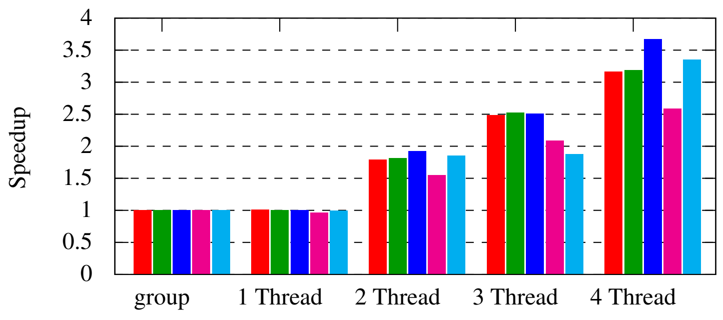

Figure 6: Worst Case Speedup, for five different use cases

[11] ——, "Thoughts on Parallel Computing for Music," in *Proceedings of the International Computer Music Conference*, 2008.

[12] U. Reiter and A. Partzsch, "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*, 2007.