

Exploiting Multi-Core Architectures for Fast Modular Synthesis

Jürgen Reuter

Karlsruhe,

Germany,

www.soundpaint.org

Abstract

Recently, CPU speed increases only slowly, while the number of transistors per chip keeps growing exponentially. Consequently, processors with multi-core architectures are pervading the market. Unfortunately, most existing software still can not exploit the parallelism. Since modular software synthesis implementations typically simulate parallel hardware, they are designated to run on parallel hardware. We examine different approaches for parallelization of a modular software synthesizer and discuss their advantages and disadvantages with respect to both the performance gain and the impact on the software architecture.

Keywords

modular synthesis, multi-core architectures, parallelization, scheduling, performance analysis

1 Introduction

Most modern operating system (OS) kernels handle multi-core architectures already well. The expertise originates from support for symmetrical multiprocessor (SMP) architectures, that became affordable in the mid 90s. For example, SMP support for Linux was introduced with the 2.0.x kernel series, and much of it was subsequently improved over and over again [1] to optimize e.g. heuristics for the mapping of threads onto CPUs. Still, SMP systems were too expensive to pervade the mass market, while the speed of single processor systems kept increasing exponentially. Only recently, multi-core architectures entered the mass market due to stagnating CPU frequency. Kernels typically handle multi-core architectures effectively the same way like SMPs, such that current kernels can be considered to have support for multi-core processing as mature as for SMPs. The only significant difference between SMP and multi-core is the reduced communication latency between the processor cores and their caches, since the electrical signals between cores on a multi-core processor do not leave the chip die. Lea [2]

and Magee and Kramer [3] present a comprehensive introduction to multithreaded programming tailored for Java.

Since most applications spawn only a rather limited number of threads, we may want to have a closer look at massively multithreaded applications. For a modular software synthesizer this means we should take into account performance issues that may arise from the overhead of managing a large number of modules. One may think that for real-world modular software synthesis applications, the typical number of modules is rather low, such that there is no need for the synthesis to scale with the number of modules, such that one should solely care for not running any processor into idle state. We truly may assume a small number of modules for a typical manually built synthesizer. However, think of polyphonic synthesizers, where only a single instance of a voice is manually wired, but the system actually runs multiple instances of this voice, thus multiplying the number of modules.

Another typical situation with a potentially large number of modules is when applying modular synthesis invisibly for the user as an automatically assembled back-end engine. For example, the `SoundPaint` [4] software converts images to sound by scanning images horizontally pixel by pixel and using the pixels' RGB values to control an array of oscillators with one oscillator per pixel line. `SoundPaint` could be easily parallelized by implementing each of the complex oscillators as a module of its own and summing up their outputs. The overall number of modules would then be roughly the number of pixel lines multiplied with the number of modules per oscillator.

With modular synthesis as an invisible back-end engine, note that the scope of this paper reaches beyond pure modular software synthesis to virtually any software synthesis that may be split into modules.

1.1 Paper Outline

We start examining if and how existing popular modular software synthesizers exploit multi-core parallelism (Sect. 2). Next, we develop an informal model of modular synthesis (Sect. 3) and discuss possible sequential and parallel implementations of this model (Sect. 4). Then, we deploy our thread parallel, barrier-synchronized implementation with parameterizable number of threads and evaluate the effect of changing the number of threads on the overall performance (Sect. 5). We discuss further possible steps for performance tuning based on a more fine-grained barrier synchronization (Sect. 6). Finally, we summarize our results (Sect. 7).

2 State of the Art

There are many modular software synthesizers, but parallelism is almost never considered in the corresponding documentation of these systems. Only a few of them support composing truly hierarchical modules – a property that will be helpful for parallelization, as we shall see later. We discuss a few popular representatives of different species of software systems for modular synthesis.

Pure Data (Pd) [5] supports a hierarchy of modules with its *sub-patches* and *abstractions* feature. A sub-patch represents just an individual patch that is referred to by a single symbol as a shortcut. A Pd abstraction defines a module that can be instantiated in multiple places in the hierarchy of patches. Events in Pd are propagated strictly sequentially in order to guarantee deterministic behavior. Audio computation is performed in blocks of 64 samples each, resulting in an audio network run every 1.45 milliseconds for a sample rate of 44100 Hz. For this purpose, Pd sorts all audio objects into a linear order for running down this linear list in a sequential run.

The Chuck system [6] supports built-in language constructs for parallelism. Chuck’s compiler compiles Chuck programs into *shreds* and passes them to a virtual machine that *shred-ules* them. This architecture promises potential for exploitation of parallel hardware, since the virtual machine could schedule the shreds on multiple cores. However, the actual audio engine of Chuck traverses the *global unit generator graph* sequentially in depth-first order for computing each sample.

A natural alternative to developing *multi-threaded applications* is to use the underlying

OS’s multitasking capabilities by running *multiple processes* in parallel. Such a system is by design parallel and exploits multi-core systems well. The Jack Audio Connection Kit [7] is a typical representative of this species of software. It connects multiple audio applications that are each running in processes of their own. Still, Jack’s chain of FIFOs for data exchange between the clients is run strictly sequentially, thus making no benefit of multiple cores. Therefore, *jackdmp* [8] activates multiple clients in parallel, if they depend on input from the same preceding client. This approach still does not yield full pipelining parallelism. Pipelining with Jack however, can be achieved if clients perform the actual processing of audio data in a separate thread in order to return from their `process()` function as quickly as possible. In any case, Jack’s overhead for inter-process communication and synchronization can become a bottleneck in the system. Especially for a synthesis with a large number of primitive modules, the OS is flooded with a large number of heavily interacting processes that run for only very short cycles, making this approach inefficient. Jack itself does not support a hierarchical topology of modules, but there are extensions like *Ingen* [9] that add such higher-level functionality to Jack. In summary, Jack works well for connecting audio applications in a modular way, but it has not been designed to realize fine-grained, sample synchronous modular synthesis.

3 Modular Synthesis Model

We first give an informal description of our model for modular synthesis with respect to the module topology and timing. Then, we present a sequential and various thread parallel implementations.

3.1 Module Topology

We assume that the building blocks of our modular synthesizer are instances of *modules*. Each module has optional input and output terminals. An output terminal delivers a stream of data samples that represents a mono audio channel. Similarly, an input terminal accepts a stream of such data samples. There are two types of modules: a predefined set of *primitive modules*, also called *cells*, and *composed modules*. A cell represents an acoustic entity that is atomic with respect to our model. Like all modules, a cell may optionally have input or output terminals. Typical cells are for example oscillators (audio output terminal only), a

simple low-pass filter (input and output) or a cell that represents an audio speaker device (audio input terminal only). A composed module is a set of optionally interconnected child modules (also called *sub-modules*), that themselves may be composed modules. A composed module’s output terminal is defined in terms of one of its sub-modules’ output terminals. Similarly, a sub-module’s input terminal is defined in terms of one of its parent module’s input terminals. Alternatively, the input terminal of a sub-module may be *connected* to at most one output terminal of another sub-module, provided that both sub-modules have the same parent module.

Fig. 1 depicts an example composition of module instances with an illegal interconnect between `Delay1` and `Delay2`, since they have different parents `Filter1` and `Filter2`. In this figure, each rectangle represents a module, and the bullets on a rectangle’s border indicate input and output terminals of the corresponding module.

Note that the overall structure of the modules as just defined is by construction always that of a tree, with the composed modules appearing as inner tree nodes, and the primitive modules as tree leaves. This inconspicuous observation will become essential when optimizing locality in the parallel implementation of the synthesis, as we will see later (Sect. 4.3).

Fig. 2 illustrates the tree that corresponds to the example in Fig. 1. The tree structure is indicated by the dashed lines. The top level node of the tree represents the uppermost module and can thus be considered to represent the complete synthesizer.

3.2 Module Timing

We assume that all modules are clocked by a central clock that runs at sample rate and thus defines a sequence of time steps t_0, t_1, t_2, \dots . For each time step t_i of the clock, each module provides a new sample at each of its output terminals. According to the module’s connections, this sample is computed in terms of output terminals from other modules, that is either from sub-modules or from module siblings via the module’s input terminals. To keep the computation well-defined, the resulting sample values should be independent of the order by which the modules update their outputs. Similarly, in case of connections that cause signal loops, care must be taken to avoid a dangling circular defi-

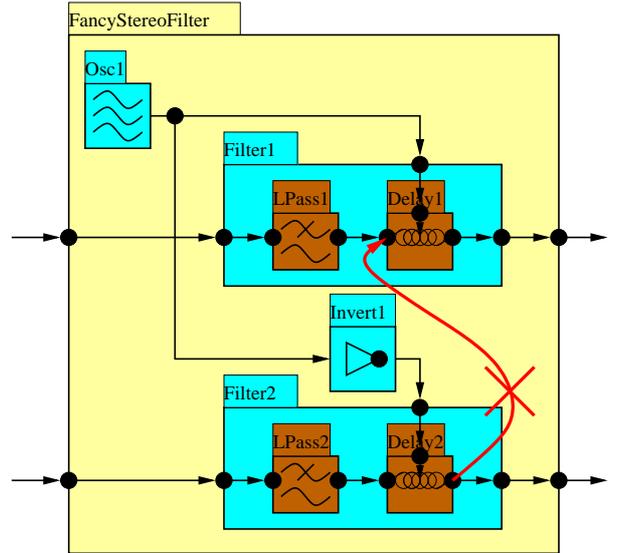


Figure 1: Example module structure, including an illegal connection between two sub-modules of different parents

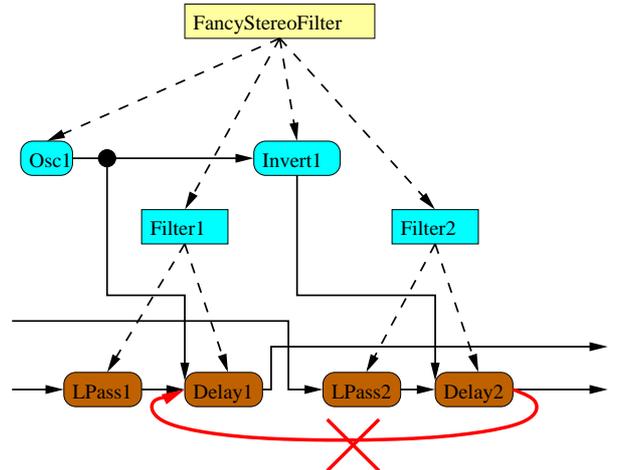


Figure 2: Tree representation that corresponds to the module hierarchy in the previous figure

nition of a module’s sample output. Therefore, we specify that each sample of a module’s output terminal for time step t_i is defined in terms of samples from the output terminals of other modules at time step t_{i-1} . We call this behavior *sample synchronous operation* of the modules.

4 Modular Synthesis Implementation

We shortly describe a simple, straight-forward sequential implementation based on a two-phases compute/update protocol. Next, we develop a parallel implementation based on the idea of the two-phases protocol and optimize it

step by step.

4.1 Sequential Implementation

For sample synchronous operation of modules, each computation of a module’s output samples at time step t_i depends on other modules’ output samples at time step t_{i-1} . The probably simplest implementation that fulfills this condition is based on a two-phases compute/update protocol, as it is used in many cellular automata implementations. In the first phase, for each module the output samples for the next time step are computed, but the results are kept private to the module. This way, other modules still may access this module’s output samples from the previous time step, in order to compute their new output samples. In the second phase, each module publishes its new output samples. Then, the next cycle continues with the first phase for the next time step, and so on. Fig. 3 illustrates the two-phases scheme.

```

while (true) do {
  // Phase 1: Compute
  for all modules do {
    compute outputs for next time step
    in terms of other module’s outputs,
    but keep results private to this
    module
  }
  // Phase 2: Update
  for all modules do {
    publish outputs to other modules
  }
}

```

Figure 3: Pseudo code for two-phases compute/update protocol

4.2 Naïve Parallel Implementation

A straight-forward parallel implementation of modular synthesis can be obtained directly from the above sequential implementation (Fig. 3) by performing the for loop in parallel for each of the two phases. This parallelization does not change the program semantics, as for both for loops, the order of iteration on the modules does not matter with respect to program semantics.

Since both for loops iterate over the same set of modules (assuming that the module structure does not change in-between), a straight-forward idea for parallelization is to assign a thread of its own to each module in order to perform the

work on the modules in parallel. Each thread will then perform the two phases just on its corresponding module. Still, the parallel implementation must ensure that the two phases keep strictly separated in time. That is, the second phase may be entered only after all threads have completed the first phase, and vice versa. The standard solution for such a condition is to implement a *barrier synchronization*: only after all participating threads have reached the barrier, they may continue. The resulting layout for the straight-forward parallel implementation is depicted in Fig. 4. The dashed lines indicate when a thread runs idle, waiting for all other threads to reach the barrier.

To utilize a multi-core system best, one of the main goals is to keep all cores actively running. Note that it is fine if a thread runs idle, as long as there are enough other threads left to keep all cores running.

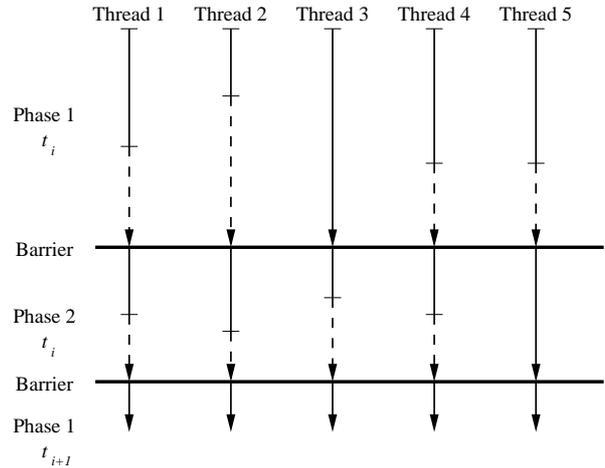


Figure 4: Idealized parallel two phases protocol implementation with barrier synchronization

4.3 Advanced Parallel Implementation

Actually, the illustration in Fig. 4 implicitly assumes that each of the threads is executed on a CPU core of its own. However, typically there will be more threads than cores, such that the OS scheduler will on the fly assign the threads to cores and execute them for example in a round-robin fashion. The result for a system with only two cores may be looking as depicted in Fig. 5.

In this figure, we observe that `cpu2` runs idle while `cpu1` is finishing Thread 3. Thread 3 has to perform the most work, as can be seen in the previous figure. This situation could be avoided by telling the scheduler to either assign longer time intervals for Thread 3 or reschedule

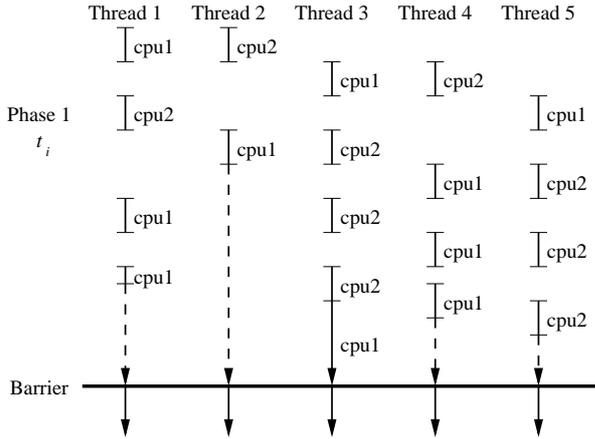


Figure 5: Real implementation on a system with two cores and round robin scheduling (Phase 2 omitted here for clarity)

it more often than the other threads. In general, the more work a thread has to perform before the next barrier, the more time should be scheduled to it when using a round robin scheduler. However, this effect is mitigated for a growing number of threads if the variance of work among the threads keeps constant.

On the other hand, with a number of threads that is large compared to the number of available cores (as is typical when creating a thread for each module), the scheduler has to switch between threads a large number of times. Consequently, an obvious approach to overcome this situation is to reduce the number of threads: we make a thread responsible for handling multiple modules.

We suspect that there is a trade-off in the number of threads that we are going to spawn. Too less threads increase the chance that cores runs idle before the barrier. One thread per module may be fine if the number of threads keeps small as compared to the number of CPU cores. However, too many threads may increase the OS overhead and thus slow down the overall performance. We now take a closer look at two corner cases in the number of threads: one thread per CPU core and one thread per module. To make our algorithm complete, we finally have to decide how to map modules to threads.

4.3.1 One Thread per CPU Core

On the one extreme, we may spawn only just as many threads as there are CPU cores in the system, and we distribute all modules among them. An even smaller number of threads is not sensible, since we want to keep all CPU cores

working on the synthesis.

Pros With only one thread per CPU core, we tremendously reduce the number of context switches in the underlying OS. Moreover, there are only as many participants involved in the barrier as there are CPUs.

Cons On the other hand, our software architecture gets more complicated, as we must decide which modules to assign to which thread and generally handle multiple modules in a single thread.

4.3.2 One Thread per Module

On the other extreme, we may spawn as many threads as there are modules. An even larger number of threads is not sensible unless there are primitive modules that themselves may be parallelized. The actual mapping of the threads to the CPU cores would be still automatically decided by the OS.

Pros This approach leaves us with a clean software architecture that is tailored to the problem rather than to the underlying hardware. Still, many OSs support a user application giving hints or requests as on which core to schedule a specific thread (which we do not further pursue here). Having many threads also leverages the problem of some threads reaching the barrier much quicker than others: if there are much more threads than CPUs, the OS has left many threads for scheduling such that CPUs do not quickly run idle.

Cons The disadvantage of having one thread per module is that we get additional overhead by numerous context switches in the OS kernel when switching between threads, and barrier synchronizations with as many participants as there are modules in the synthesizer. Context switches cause additional overhead on the OS and hardware level and thus may slow down the overall performance, especially, if much of the data in the CPU caches is displaced or invalidated and must be reloaded from main memory when the thread is rescheduled (*cache thrashing*).

4.3.3 Modules-to-Thread Mapping

Once we have spawned n threads, another problem that remains is which modules to assign to what thread. One possibility is to distribute the modules among the threads such that the variance of the estimated work among all threads is minimized, such that all threads reach the barrier almost at the same time. This approach re-

quires good estimates of the average work each module has to perform during a compute phase.

In this work, we pursue a different approach. Assuming that inter-thread communication is expensive (especially with respect to the cache coherence), we strive to put those modules into a thread that are tightly connected. Given the hierarchical composition of modules as explained in Sect. 3.1, we already have a natural neighbourhood structure of modules with respect to connections between the modules. Thus, to map modules to threads, we just perform a depth-first walk along the tree and assign them to a given number of threads in this order.

5 Evaluation

What remains to examine is the ideal number of threads to spawn for our advanced parallel algorithm. Note that, by construction, our parallel implementations may be even run with just a single thread. However, there is no point in having a parallel implementation over the sequential one, if the parallel implementation is running with just a single thread. Still, we want to run the parallel implementation also with only a single thread to measure the overhead of the parallel implementation over the sequential one.

5.1 Evaluation Environment

We examined the performance behavior of the algorithm with a simple implementation of a modular synthesizer written in Java 1.7.0 (IcedTea Runtime Environment, build 1.7.0-b21 and IcedTea Server VM, build 1.7.0-b21, mixed mode). The measurements were performed on a Linux kernel 2.6.23 (2.6.23.1-42.fc8) running on an Intel Core2 Quad CPU Q6600 @ 2.40 GHz with 2 GByte of system main memory. The Java virtual machine by default maps Java threads to native linux threads.

Since a full SoundPaint implementation on top of our modular synthesis software is not yet available, we chose a configuration that comes at least close to SoundPaint with respect to the numbers of modules and their structure. More precisely, we chose an array of 2000 sine wave oscillators with attached (constant) frequency controls. Each two of the oscillators represent the stereo output of a horizontal pixel line of an image. Finally, two mixers mix the output of all oscillators down to a stereo signal, and a tape recorder module writes the stereo samples to the hard disk drive.

With this environment, the sequential algorithm as described in Sect. 4.1 takes roughly

38s to synthesize 2s of stereo sound.

5.2 Trimming the Synchronization Overhead

We expect much overhead by the barrier synchronization for each sample and of as much as some thousand modules. Therefore, the number of threads of our parallel implementation can be adjusted, rather than naïvely spawning a thread for each module.

The graph with the square dots in Fig. 6 shows how the total running time of the parallel algorithm varies with the number of threads. For convenience of comparison, the running costs of the sequential algorithm also appear in this figure in the leftmost column (#threads=0). We observe that the parallel algorithm with only one thread takes roughly 39s and thus is insignificantly slower than the sequential version with 38s. For any number of threads below 4, there is always some core of our QuadCore CPU running idle. On the other hand, if we adjust the number of threads to some value above 4, we just add more overhead. We achieve the best performance, if we set the number of threads exactly to the number of available cores (assuming that there are no other processes or threads running concurrently that consume a substantial amount of processing power).

5.3 Optimizing the Modules-to-Thread Mapping

For the above benchmark, we just distributed modules among threads in a round-robin manner. As discussed in Sect. 4.3.3, we suppose to gain some more speed by acknowledging topological neighbourhood between modules when distributing them among the threads. Consequently, we put those modules into the same thread that are close in the hierarchy of modules. The graph with the cross dots in Fig. 6 shows the result of this small change in the initialization of our parallel algorithm. One would expect only a slightly better cache hit rate in the CPU, thus gaining a very small performance enhancement, if at all. Surprisingly, the performance gain is much higher: when using 4 threads, the optimized parallel algorithm takes with less than 17s about 25% less time as compared to the round-robin assignment of modules to threads. It is unlikely that this effect is primarily caused by some threads reaching the barrier much earlier than others (as discussed in Sect. 4.2), since this effect is observ-

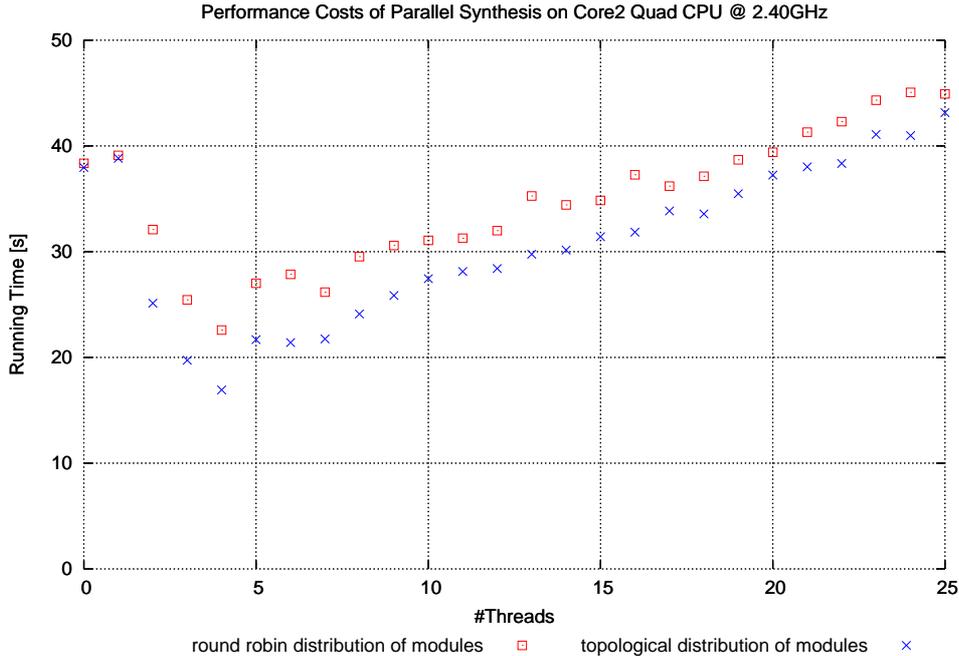


Figure 6: Benchmark for massively parallel synthesis similar to SoundPaint

able throughout all numbers of threads (at least some of which definitely would result in a fairly good distribution of modules). The precise reason for this amazing enhancement is currently not clear and should be further examined. In any case, it is important to acknowledge that the assignment policy of modules to the threads plays an important role.

5.4 Summary

Fig. 7 shows the speed-up of all parallel runs as compared to the sequential algorithm. In the optimal case of 4 parallel threads and with topologically close modules assigned to the same thread, we gain a speed-up factor of 2.24 over the sequential algorithm, which is still far away from the theoretical optimum speed-up factor of 4.0, but not too bad, if we consider the overhead for keeping thousands of modules synchronized at sample rate.

6 Future Work

Due to time and space restrictions, this work omits an in-depth study of cache data locality effects. In particular, we should more carefully examine the unexpectedly large impact of our proposed modules-to-thread mapping that follows the tree structure of the modules.

We almost completely ignored threads running idle at barriers. If this issue should turn out to have a major negative impact on the over-

all performance, we may pursue the following approach: Not all modules' terminal outputs need to be protected by the barrier, since there are typically many input/output terminals that are used only within in a thread. Therefore, we buffer all current modules' output values that may be needed by other threads, but internally to this thread continue processing with the next cycle. This way, we compensated for a temporary imbalance in the workload between the threads. If the buffers keep repeatedly running full or no more next cycle is possible due to lacking local terminals (such that we still end up waiting at the barrier), this symptom may indicate major imbalance, and we may consider migrating a module from one thread to another one. The buffering is helpful for monitoring such imbalance and reacting proactively.

7 Conclusion

Given the clear results of our evaluation, we want to state the following rules of thumb for parallelization on multi-core architectures:

- Do use multiple threads to exploit multiple cores.
- Think about the number of threads you can and/or you want to spawn.
- Ideally, keep the number of threads that your application spawns adjustable.

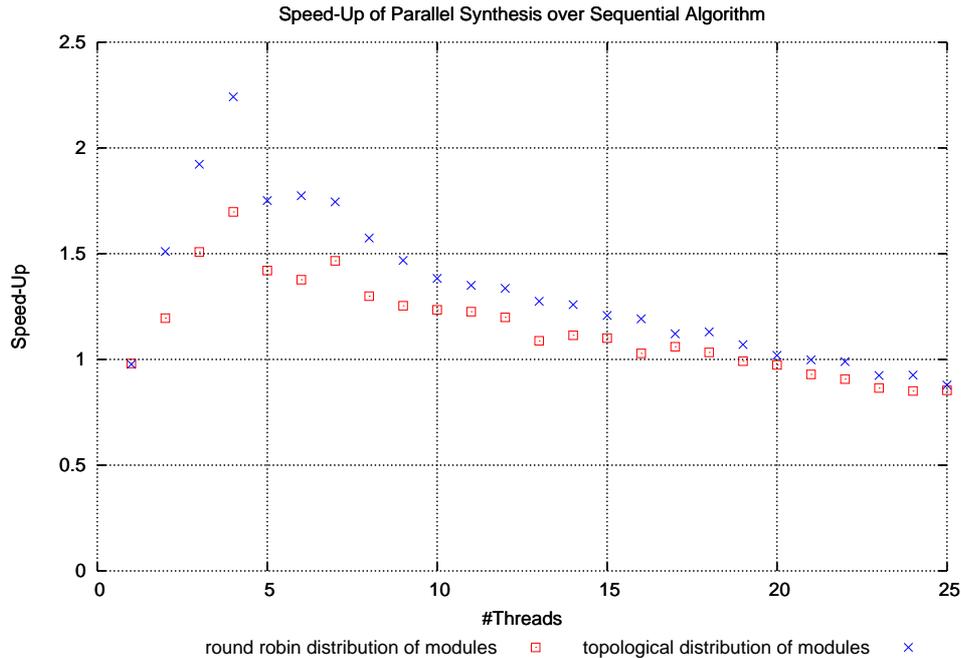


Figure 7: Speed-up of above benchmark over sequential algorithm

- Think about how to distribute the work to be done among your threads.
- Be aware of what the OS scheduler may or may not do when scheduling threads, like thrashing effects, communication latency between the cores, cost of synchronization primitives, and locality of cached data.

Following these rules, we were able on a QuadCore CPU to gain a speed-up factor of about 2.24 over a sequential implementation of a modular software synthesizer. This result is still far away from the theoretically possible speed-up factor of approximately 4.0, but there are some issues left for further research, such that we are confident to get even closer towards the optimal speed-up in the future.

The modular synthesizer that was used for evaluation in this work is available at <http://www.soundpaint.org/modsynth/>.

References

- [1] Enkh Tumenbayar. Linux SMP HOWTO. <http://tldp.org/HOWTO/SMP-HOWTO.html>, July 2002.
- [2] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison Wesley Professional, second edition, November 1999.
- [3] Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, Inc., New York, NY, USA, second edition, 2006.
- [4] Jürgen Reuter. Soundpaint – painting music. In *Proceedings of the 3rd International Linux Audio Conference (LAC2005)*, pages 79–85, Karlsruhe, Germany, April 2005. ZKM, Karlsruhe.
- [5] Miller Puckette. Pure data (Pd). <http://crca.ucsd.edu/~msp/software.html>, 2007.
- [6] Ge Wang, Perry R. Cook, and Ananya Misra. Designing and implementing the chuck programming language. In *Proceedings of the International Computer Music Conference (ICMC 2005)*, 2005.
- [7] Paul Davis. Jack audio connection kit. <http://jackaudio.org/>, 2007.
- [8] Stéphane Letz, Yann Orlarey, and Dominique Fofer. Jack server for multi-processor machines. In *Proceedings of the 3rd International Linux Audio Conference (LAC2005)*, pages 29–36, Karlsruhe, Germany, April 2005. ZKM, Karlsruhe.
- [9] Dave Robillard. Ingen. <http://wiki.drobilla.net/Ingen/>, 2007.