Work Stealing Scheduler for Automatic Parallelization in Faust

Stéphane Letz, Yann Orlarey, Dominique Fober

Work Stealing Scheduler for Automatic Parallelization in Faust

Stephane Letz and **Yann Orlarey** and **Dominique Fober**
GRAME
9 rue du Garet, BP 1185
69202 Lyon Cedex 01,
France,
{letz, orlarey, fober}@grame.fr

Abstract

FAUST 0.9.10 [1] introduces an alternative to OpenMP based parallel code generation using a Work Stealing Scheduler and explicit management of worker threads. This paper explains the new option and presents some benchmarks.

Keywords

FAUST, Work Stealing Scheduler, Parallelism

1 Introduction

Multi/many cores machines are becoming common. There is a challenge for the software community to develop adequate tools to take profit of the new hardware platforms [3] [8]. Various libraries like Intel Thread Building Blocks [2] or "extended C like" Cilk [5] can possibly help but still require the programmer to precisely define sequential and parallel sub part of the computation and use the appropriate library call or building blocks to implement the solution.

In the audio domain, work has been done to parallelize well known algorithms [4], but very few systems aim to help in *automatic* parallelization. The problem is usually tricky with stateful systems and imperative approaches where data has to be shared between several threads, and concurrent access have to be accurately controlled.

On the contrary, the functional approach used in high level specification languages generally helps in this area. It basically allows the programmer to define the problem in an abstract way completely unconnected of implementations details, and let the compiler and various backends do the hard job.

By exactly controlling when and how state is managed during the computation, the compiler can decide what multi-threading or parallel generation techniques can be used. More sophisticated methods to reorganize the code to fit specific architectures can then be tried and analyzed.

1.1 The Faust approach

FAUST [2] is a programming language for real-time signal processing and synthesis designed from scratch to be a compiled language. Being efficiently compiled allows FAUST to provide a viable high-level alternative to C/C++ to develop high-performance signal processing applications, libraries or audio plug-ins.

The FAUST compiler is built as a stack of code generators. The scalar code generator produces a single computation loop. The vector generator rearrange the C++ code in a way that facilitates the autovectorization job of the C++ compiler. Instead of generating a single sample computation loop, it splits the computation into several simpler loops that communicates by vectors. The result is a Direct Acyclic Graph (DAG) in which each node is a computation loop.

Starting from the DAG of computation loops (called "tasks"), FAUST was already able to generate parallel code using OpenMP directives [1]. This model has been completed with a new dynamic scheduler based on a *Work Stealing* model.

2 Scheduling strategies for parallel code

The role of scheduling is to decide how to organize the computation, in particular how to assign tasks to processors.

2.1 Static versus dynamic scheduling

The scheduling problem exists in two forms: static and dynamic. In static scheduling usually done at compile time, the characteristics of a parallel program (such as task processing times, communication, data dependencies, and synchronization requirements) are known before

[1] this work was partially supported by the ANR project ASTREE (ANR-08-CORD-003)

[2] http://www.threadingbuildingblocks.org/

program execution. Temporal and spatial assignment of tasks to processors are done by the scheduler to optimize the execution time. In dynamic scheduling on the contrary, only a few assumptions about the parallel program can be made before execution, and thus, scheduling decisions have to be realized on-the-fly, and assignment of tasks to processors are made at run time.

2.2 DSP specific context

We can list several specific elements of the problem:

- the graph describes a DSP processor which is going to read n input buffers containing p frames to produce m output buffers containing the same p number of frames

- the graph is completely known in advance, but precise cost of each task and communication times (memory access, cache effects...) is not known

- the graph can be computed in a single step: each task is going to process p frames in a single step, or buffers can be cut in slices and the graph can be executed several times on sub slices to fill output buffers. (see "pipelining" section)

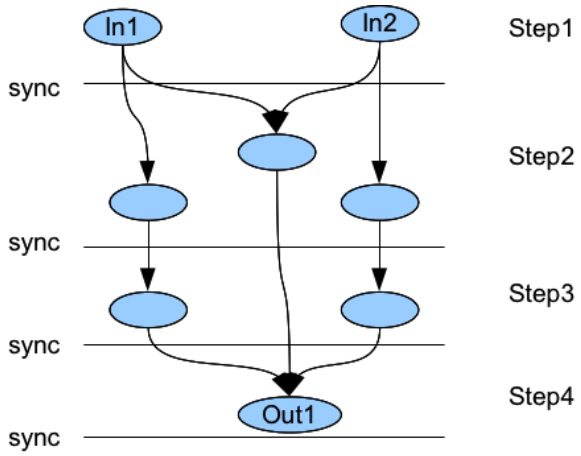

Figure 1: *Tasks graph with forward activations and explicit synchronization points*

2.3 OpenMP mode

The OpenMP based parallel code generation is activated by passing the `---openMP` (or `--omp`) option to the FAUST compiler. It implies the `--vec` option as the parallel code generation is built on top of the vector code generation.

In OpenMP mode, a topological sort of the graph is done. Starting from the inputs, tasks are organized as a sequence of groups of parallel tasks. Then appropriate OpenMP directives are added to describe a *fork/join* model. Each group of task is executed in parallel and synchronization points are placed between the groups (Figure 1). This model gives good results with the Intel icc compiler. Unfortunately until recently, OpenMP implementation in g++ was quite weak and inefficient, and even unusable in an real-time context on OSX.

Moreover the overall approach of organizing tasks as *a sequence of groups of parallel tasks* is not optimum in the sense that synchronization points are required for all threads when part of the graph could continue to run. In some sense the synchronization strategy is too strict and a data-flow model is more appropriate.

2.3.1 Activating Work Stealing Scheduler mode

The scheduler based parallel code generation is activated by passing the `--scheduler` (or `--sch`) option to the FAUST compiler. It implies the `--vec` option as the parallel code generation is built on top of the vector code generation.

With the `--scheduler` option, the FAUST compiler uses a very different approach. A data-flow model for graph execution is used, to be executed by a dynamic scheduler. Parallel C++ code embedding a Work Stealing Scheduler and using a pool of worker threads is generated.

Threads are created when the application starts and all participate in the computation of the graph of tasks. Since the graph topology is known at compilation time, the generated C++ code can precisely describe the execution flow (which tasks have to be executed when a given task is finished...). Ready tasks are activated at the beginning of the compute method and are executed by available threads in the pool. The control flow then circulate in the graph from inputs task to output tasks in the form of *activations* (ready task index called *tasknum*) until all output tasks have been executed.

2.4 Work Stealing Scheduler

In a Work Stealing Scheduler [7], idle threads take the initiative: they attempt to *steal* tasks from other threads. This is possible by having each thread owns a *Work Stealing Queue*, a special double-ended queue with a Push operation, a *private* LIFO Pop operation [3] and a *public*

[3] which does not need to be multi-thread aware

FIFO Pop operation [4]. The basic idea of work stealing is for each processor to place work when it is discovered in its local WSQ, greedily perform that work from its local WSQ, and steal work from the WSQ of other threads when the local WSQ is empty.

Starting from a ready task, each thread executes it and follows the data dependencies, possibly pushing ready output tasks into its own local WSQ. When no more tasks can be executed on a given computation path, the thread pops a task from its local WSQ using its private LIFO Pop operation. If the WSQ is empty, the thread is allowed to steal tasks from other threads WSQ using their public FIFO Pop operation.

The local LIFO Pop operation allows better cache locality and the FIFO steal Pop larger chuck of work to be done. The reason for this is that many work stealing workloads are divide-and-conquer in nature, stealing one of the oldest task implicitly also steals a (potentially) large subtree of computations that will unfold once that piece of work is stolen and run.

For a given cycle, the whole number of frames is used in one graph execution cycle. So when finished, a given task will (possibly) activate its output task only, and activation goes forward.

2.4.1 Code generation

The compiler produces a `computeThread` method called by all threads:

- tasks are numbered and compiled as a big *switch/case* block

- a *work stealing* task which aim to find out the next ready task is created

- an additional *last task* is created

2.5 Compilation of different type of nodes

For a given task in the graph, the compiled code will depend of the topology of the graph at this stage.

2.5.1 One output and direct link

If the task has one output only and this output has one input only (so basically there is a single link between the two tasks), then a *direct activation* is compiled, that is the tasknum of the next task is the tasknum of the output task, and there its no additional step required to find out the next task to run.

[4] which has to be multi-thread aware using lock-free techniques and is thus more costly

2.5.2 Several outputs

If the task has several outputs, the code has to:

- init tasknum with the `WORK_STEALING` value

- if there is a direct link between the given task and one of the output task, then this output task will be the next to execute. All other tasks with a direct link are pushed on current thread WSQ [5]

- otherwise for output tasks with more than one input, the activation counter is atomically decremented (possibly returning the tasknum of the next task to execute)

- after execution of the activation code, tasknum will either contains the actual value of the next task to run or `WORK_STEALING`, so that the next ready task if found by running the work stealing task.

2.5.3 Work Stealing task

The special *work stealing task* is executed when the current thread has no more next task to run in its computation path and its WSQ is empty. The `GetNextTask` function aims to find out a ready task by possibly stealing a task to run from any of the other threads except the current one. If no task is ready then `GetNextTask` returns `WORK_STEALING` value and the thread loops until it finally finds a task or the whole computation ends.

2.5.4 Last task

Output tasks of the DAG are connected and activate the special *last task* which in turn quits the thread.

```
void computeThread(int thread)
 {
     TaskQueue taskqueue;
     int tasknum = -1;
     int count = fFullCount;

     // Init input and output
     FAUSTFLOAT* input0
        = &input[0][fIndex];
     FAUSTFLOAT* input1
        = &input[1][fIndex];
     FAUSTFLOAT* output0
        = &output[0][fIndex];

     // Init graph
     int task_list_size = 2;
     int task_list[2] = {2,3};
     taskqueue.InitTaskList(
        task_list_size,
        task_list,
        fDynamicNumThreads,
```

[5] The chosen task here is the first in the task output list, more sophisticated choice heuristics could be tested at this stage.

```
        cur_thread,
        tasknum);

    // Graph execution code
    while (!fIsFinished) {
      switch (tasknum) {
       case WORK_STEALING: {
         tasknum
           = GetNextTask(thread);
         break;
       }
       case LAST_TASK: {
         fIsFinished = true;
         break;
       }
       case 2: {
         // DSP code
         PushHead(4);
         tasknum = 3;
         break;
       }
       case 3: {
         // DSP code
         PushHead(5);
         tasknum = 4;
         break;
       }
       case 4: {
         // DSP code
         tasknum
           = ActivateOutput(LAST_TASK)
         break;
       }
       case 5: {
         // DSP code
         tasknum
           = ActivateOutput(LAST_TASK)
         break;
       }
      }
    }
  }
```

Listing 1: example of `computeThread` method

2.6 Activation at each cycle

The following steps are done at each cycle:

- n-1 threads are resumed and start the work
- the calling thread also participates
- ready tasks (tasks that depends of the input audio buffers or tasks with no inputs) are dispatched between all threads, that is each thread pushes sub-tasks in its private WSQ and takes one of them to be directly executed
- after having done its part of the work, the main thread waits for all worker threads to finish

This is done in the following `compute` method called by the master thread:

```
void compute(int fullcount,
             FAUSTFLOAT** input,
             FAUSTFLOAT** output) {
  this->input = input;
  this->output = output;
  for (fIndex = 0;
    fIndex < fullcount;
    fIndex += 1024) {
    fFullCount
        = min(1024, fullcount-fIndex);
    TaskQueue::Init();
    // Initialize end task
    fGraph.InitTask(1,1);
    // Only initialize tasks with inputs
    fGraph.InitTask(3,1);
    // Activate worker threads
    fIsFinished = false;
    fThreadPool.
        SignalAll(fDynamicNumThreads-1);
    // Master thread participates
    computeThread(0);
    // Wait for end
    while (!fThreadPool.IsFinished()) {}
  }
}
```

Listing 2: master thread compute method

2.7 Start time

At start time, n worker threads are created and put in sleep state, thread scheduling properties and priorities are set according to calling thread parameters.

3 Pipelining

Some graphs are sequential by nature. Pipelining techniques aim to create and exploit parallelism in those kind of graph to better use multicores machines. The general idea is that for a sequence of two tasks A and B, instead of running A on the entire buffer size *then* run B, we want to split the computation in n sub-buffers and run A on the first sub-buffer, then B can be run on A first sub-buffer output while A runs on second sub-buffer and so on.

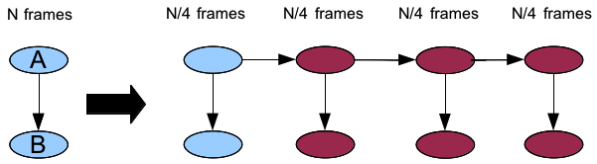

Figure 2: *Very simple graph rewritten to express pipelining, A is a recursive task, B is a non recursive one*

Different code generation techniques can be used to obtain the desired result. We choose to rewrite the initial graph of tasks, by *duplicating n times* each task to be run on a sub part of the buffer. This way the previously described code

generator can be directly used without major changes.

3.1 Code generation

The pipelining parallel code generation is activated by passing `-sch` option as well as the `--pipelining` option (or `-pipe`) with a value, the factor the initial buffer will be divided in.

Previously described code generation strategy has to be completed. The initial tasks graph is rewritten and each task is split in several sub-tasks:

- *recursive tasks* [6] are rewritten as a list of n connected sub-tasks (that is activation has to go from the first sub-task to the second one and so on). Each sub-task is then going to run on buffer-size/n number of frames. There is no real gain for the recursive task itself since it will still be computed in a sequential manner. But output sub-tasks can be activated more rapidly and possibly executed immediately is they are part of a non recursive task (Figure 2).

- *non recursive tasks* are rewritten as a list of n non connected sub-tasks, so that all sub-tasks can possibly be activated in parallel and thus run on several cores at the same time. Each sub-task is then going to run on buffer-size/n number of frames.

This strategy is used for all tasks in the DAG, the rewritten graph enters the previously described code generator and the complete code is generated.

4 Benchmarks

To compare the performances of these various compilation schemes in a realistic situation, we have modified an existing architecture file (Alsa-GTK on Linux and CoreAudio-GTK on OSX) to continuously measure the duration of the `compute` method (600 measures per run). We give here the results for three real-life applications: *Karplus32*, a 32 strings simulator based on the Karplus-Strong algorithm (figure 3), *Sonik Cube* (figure 4), the audio part software of an audio-visual installation and *Mixer* (figure 5), a multi-voices mixer with pan and gain.

Instead of time, the results of the tests are expressed in MB/s of processed samples because memory bandwidth is a strong limiting factor for today's processors (an audio application can never go faster than the memory bandwidth).

[6] those where the computation of the frame n depends of the computation of previous frames

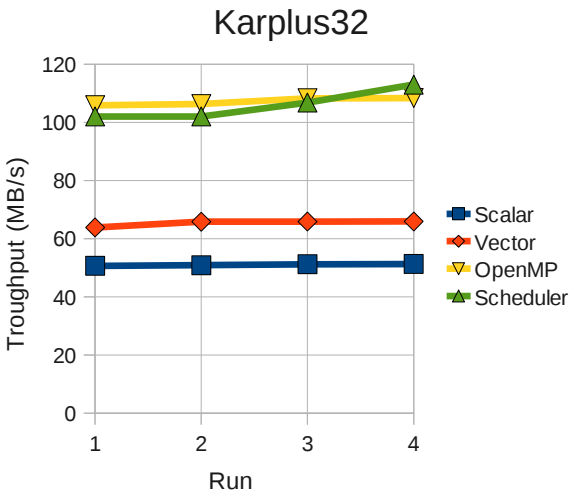

Figure 3: *Compared performances of the 4 compilation schemes on karplus32.dsp*

As we can see, in both cases the parallelization introduces a real gain of performances. The speedup for *Karplus32* was x2.1 for OpenMP and x2.08 for the WS scheduler. For *Sonik Cube* the speedup with 8 threads was of x4.17 for OpenMP and x5.29 for the WS scheduler. It is obviously not always the case. Simple applications, with limited demands in terms of computing power, tend to perform usually better in scalar mode. More demanding applications usually benefit from parallelization.

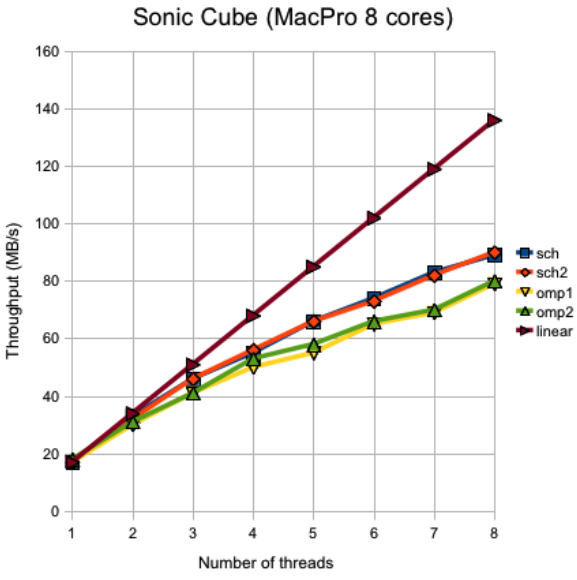

Figure 4: *Compared performances of different generation mode from 1 to 8 threads*

The efficiency of OpenMP and WS scheduler are quite comparable, with an advantage to WS scheduler with complex applications and more CPUs. Please note that not all implementations of OpenMP are equivalent. Unfortunately the GCC 4.4.1 version is still unusable for real time audio application. In this case the WS scheduler is the only choice. The efficiency is also dependent of the vector size used. Vector sizes of 512 or 1024 samples usually give the best results.

The pipelining mode does not show a clear speedup in most cases, but *Mixer* is an example when this new mode helps.

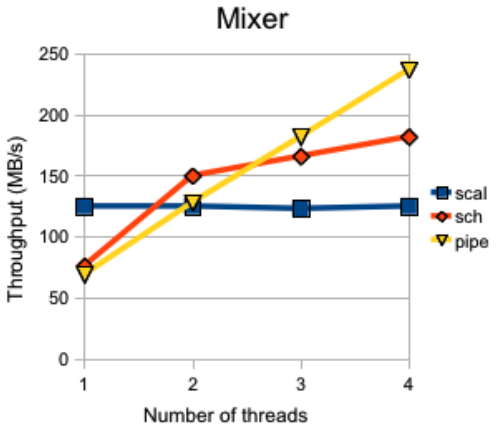

Figure 5: *Compared performances of different generation mode from 1 to 4 threads on OSX with a buffer of 4096 frames*

5 Open questions and conclusion

In general, the result can greatly depends on the DSP code going to be parallelized, the chosen compiler (icc or g++) and the number of threads to be activated when running. Simple DSP effects are still faster in scalar of vectorized mode and parallelization is of no use. But with some more complex code like *Sonik Cube* the speedup is quite impressive as more threads are added. But they are still a lot of opened questions that will need more investigation:

- improving dynamic scheduling, since right now there is no special strategy to choose the task to wake up in case a given task has several output to activate. Some ideas of static scheduling algorithms could be used in this context.

- the current compilation technique for pipelining actually duplicates each task code n times. This simple strategy will probably show its limit for more complex effects with a lot of tasks.

- there is no special strategy to deal with thread affinity, thus the performances can degrade if tasks are switched between cores.

- the effect of memory bandwidth limits and cache effects have to be better understood

- composing several pieces of parallel code without sacrificing performance can be quite tricky. Performance can severally degrade as soon as too much threads are competing for the limited number of physical cores. Using abstraction layers like Lithe [6]or OSX libdispatch [7] could be of interest.

References

[1] Y. Orlarey, D. Fober, and S. Letz. *Adding Automatic Parallelization to Faust.* Linux Audio Conference 2009.

[2] Yann Orlarey, Dominique Fober, and Stephane Letz. *Syntactical and semantical aspects of faust.* Soft Computing, 8(9):623632, 2004.

[3] J.Ffitch, R.Dobson, and R.Bradford. *The Imperative for High-Performance Audio Computing.* Linux Audio Conference 2009.

[4] Fons Adriaensen. *Design of a Convolution Engine optimised for Reverb.* Linux Audio Conference 2006.

[5] Robert D.Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. *Cilk: an efficient multithreaded runtime system.* In PPOPP 95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 207216, New York, NY, USA, 1995. ACM.

[6] Heidi Pan, Benjamin Hindman, Krste Asanovic. *Lithe: Enabling Efficient Composition of Parallel Libraries.* USENIX Workshop on Hot Topics in Parallelism (HotPar'09), Berkeley, CA. March 2009.

[7] Blumofe, Robert D. and Leiserson, Charles E. *Scheduling multithreaded computations by work stealing.* Journal of ACM, volume 46, number 5, 1999, New York NY USA.

[8] David Wessel et al. *Reinventing Audio and Music Computation for Many-Core Processors.* International Computer Music Conference 2008.

[7] http://developer.apple.com/