Refining a Parallel Algorithm For Calculating Bowings

Cordelia Hall[1], Hans-Wolfgang Loidl[1], Phil Trinder[2], Kevin Hammond[3], John O'Donnell[1]

[1] Department of Computing Science, University of Glasgow, Scotland, U.K.
E-mail: {cvh,hwloidl,jtod}@dcs.glasgow.ac.uk
[2] The Computing Department, The Open University, Milton Keynes, England, U.K.
E-mail: P.W.Trinder@open.ac.uk
[3] Division of Computer Science, University of St. Andrews, Scotland, U.K.
E-mail: kh@dcs.st-and.ac.uk

Abstract. String players know that bowing properly is the hardest skill they have to learn. In other work [4], we develop an algorithm that calculates bowings for BowTech, a project that supports string performers. This algorithm takes a significant amount of time to execute (in our second test case, over an hour for the implementation, written in Ada). We have implemented the algorithm in GPH, a parallel superset of Haskell, and measured the quality of output on six pieces of music. The parallel program has been refined using the GranSim simulator and measured on two parallel architectures: a shared memory multiprocessor and a network of workstations.

1 Introduction

String players have long been aware that it is necessary to learn a new piece by trying out a variety of *bowings*. A *bowing* is denoted by a sequence of symbols indicating movement away from the tip (an upbow) or away from the frog (a downbow). This experimentation process is hard because there is nothing that helps the player remember the current state of the bow, i.e. the part of the bow from which the next stroke starts. In fact, it's often possible to see an orchestral musician creating a new bowing by moving the right arm up and down; this is evidence that they cannot always select an arbitrary note of the music and know the direction the bow will take when playing it, let alone the point at which the bow will touch the strings.

Many musicians are content to record only the direction of the bow, using the traditional notation. Somehow they think that the work they do in practicing the strokes used to produce a beautiful, clean sound will allow them to recreate their most successful efforts under the pressure of performance. This is remarkable for at least three reasons:

1. Playing at speed is essential in order to practice the smaller strokes that speed requires and get the balance and coordination right. However this is hard to do if the left hand can't yet manage it. The musician must have at least 2 versions of each bowing worked out; one to practice slowly with the left hand as it comes up to speed and a fast version. This requires a detailed memory of all the decisions made.
2. It's not easy to produce clean, good sound on demand. It takes lots of practice, and a careful study of the successful strokes so that they can be repeated. This means that it is essential to be able to recall the state of the bow when starting to work on a difficult spot (it's a bad idea to practice the spot starting at the *wrong* part of the bow).
3. Some music editions are unreliable, and musicians sightreading from them must learn to look ahead and protect themselves. For example, the edition we used for the Bach double violin concerto in D minor had a bowing for the solo first violin part that simply ran out of bow on the second line. These days, musicians have something like one rehearsal before performing, and need as much help as they can get from the editors of the music they read. So string players must be in the habit of analysing where they are in the bow.

There are some very successful schools of string teaching that bring as much thought and planning to bear upon this problem as they do on understanding the components of a piece of music. The goal of the BowTech project is to support this work, so that musicians who aren't able to analyse their technique will receive some help. Another goal is to process MIDI files so that a programmable synthesizer can produce sounds that are more like those of a real string instrument.

Currently, the algorithm calculates the position of the bow and tells the performer whether to be at the frog, middle or tip of the bow. Future work will focus on patterns of strokes as a basic unit, so that the musician can map out strategies for getting through a particular passage.

2 The background of the algorithm

The algorithm described by this paper is based in part on previous work [4], which used information continually provided by the musician to calculate whether a bowing resulted in a disaster (running out of bow at either the frog or the tip). When guided in detail, it maintained the state information required, but was tedious to use because it couldn't automatically deal with problems such as calculating the bow length for a mixture of long and short notes. The program would flag these as disastrous because the long bows eventually consume the bow available, when in contrast the player would perform these passages by allocating less bow to the long notes and more to the short ones.

Another significant problem was that the program would get stuck at either the frog or the tip after negotiating a complex passage, causing a long sequence of sixteenth or eighth notes that followed to be played at one end of the bow. The player would cope with this by gradually moving to the middle of the bow, where the sequence would sound better and be more under the player's control.

The new algorithm propagates bowing information and dynamics, just as the old one did. This tells subsequent passes whether the current note is an upbow or downbow, and whether it is soft, medium or loud. Each note is given the length required by its duration, multiplied by a factor describing the dynamic and another describing the metronome marking of the music. This value is then added to the current bow position if the bow is an upbow, and subtracted if it is a downbow. The tip of the bow is represented by zero, and the frog by one, so all bow positions are a fraction between those two boundaries.

The purpose of the new algorithm is to process an entire piece without requiring any help from the musician. It was clear that the algorithm needed to be able to select one of several possible lengths for a stroke in order to deal with difficult passages. The algorithm could not use backtracking over these alternatives, because the worst case complexity proved to be a little too common for comfort, and so initially, modified backtracking was adopted.

A piece was divided into segments or *blocks*. The first kind, a *simple* block, is composed of notes with the same duration; the second kind, a *complex* block, contains only notes of mixed durations. This segmentation was vitally important because the simple blocks could be handled without backtracking, greatly restricting the problem size.

For simple blocks, such as long passages of sixteenth notes, the algorithm takes advantage of the fact that they allow the player to make adjustments in the location of the bow. It makes the upbows longer, if the bow position is too close to the tip, and the downbows longer, if the bow position is too close to the frog. This means that the algorithm can work its way out of awkward positions and move back to the middle of the bow, which is what a player would do.

Complex blocks require some care because they are difficult to bow properly. The effect of each stroke must be worked out, ensuring that no stroke goes past either end of the bow. It was hard to find a good algorithm for complex blocks, and the current algorithm is the result of some experimentation. An early algorithm used backtracking, which was sensitive to problems that may occur at the end of a complex block. This algorithm was fast, but unfortunately it was impractical to write a backtracking algorithm that also kept track of all the information needed to assess the bowing of a complex block, so its results tended to be rather stupid.

The current algorithm, which we will call `A`, is slow, but its results are much better. It is roughly modelled on Knuth's algorithm for typesetting, TeX. Algorithm `A` generates all possible paths through a complex block, then compares them using three different scoring functions, `Average`, `Successive` and `Extremes`. Each scoring algorithm produces a value from 1 to 100, where 100 is the worst and 1 the best, and the composite score assigned a path is the average scores of the three.

Any path on which `A` runs out of bow is given 100 without further assessment. The first function, `Average`, takes the average bow position, returning a high score if the average is close to the middle of the bow, and a lower score if it is closer to the tip or frog. While a useful indicator of the value of a path, the average may be misleading if there is a sequence of notes at one end followed by another sequence at the other end. For this reason, the second algorithm checks for successive strokes close to either the frog or the tip. If there are more than 4 strokes at the tip, or more than 2 at the frog, then a lower score is returned. Finally, paths that require the bow to be at an extreme end of the bow make it more likely that the algorithm will run out of bow somewhere along the line, so the third algorithm assigns such paths lower scores.

These are three of a number of possible algorithms, and we plan to investigate more. However, the results were interesting.

3 What happened when the algorithm was applied

We applied the algorithm to six pieces: the three movements of the Bach double violin concerto (solo violin 1), and three movements from Handel's sonatas for violin and figured bass (the first movement of the sonata in A major, the third movement of the sonata in D major, and the fourth movement of Handel's sonata in E major). Each of these movements had some interesting problems for our algorithm to address. The algorithm assigned bowing annotations without stopping for direction by the musician (unlike our previous work). The musician provided music that was marked with dynamics and wrote in any slurs required for known technical reasons (e.g. the player would run out of bow here otherwise) or musical reasons. If the algorithm runs out of bow anyway, the unfortunate musician then has to figure out why.

All of the movements were assigned acceptable bowings (none ran out of bow). In many cases, the algorithm found a reasonable solution, one that might be found by a musician sight-reading the music. It marked each note with the words T(IP), M(ID) or F(ROG), depending upon the bow position after that note was played. These markings appear in the examples page at the end of the paper (excluding M markings, to reduce clutter).

Here are some of our conclusions after applying the algorithm to our test pieces:

1. The first movement of the Bach was relatively straightforward. It had lots of blocks of sixteenth or eighth notes which allowed the algorithm to maintain the desired average bow position (the middle of the bow). In our example, a note appeared which was so long that it had to be handled in one of two ways. Either the bow had to be at the frog when it started, or the stroke had to be shortened. The algorithm shortened the duration of that stroke. The notes between the two half notes were few enough that the second half note was reached before the algorithm moved back to the middle of the bow.
2. The second movement of the Bach concerto was interesting because it contained a passage which could not be played without shortening the long notes. This was duly discovered by the algorithm.
3. The first movement of the first Handel sonata in A major contained lots of tricky sections where sequences of four or five strokes were twice separated by longer notes that moved in the same direction. This was handled well by the algorithm, but might have been more of a problem if there had been a longer sequence like this.

4. The third movement of the Handel sonata in D major was challenging because the tempo was so slow that long notes tended to require a whole bow, leaving little margin for adaptation. The algorithm shortened these.
5. The fourth movement of the Handel sonata in E major was straightforward. Interestingly, the algorithm marked a different passage to be played near the frog, then managed to move away to the middle of the bow. This occurred just after a long note left the player near the frog, and was the same as Hall's solution while performing the piece.
6. The third movement of the Bach had one tricky type of passage, in which two sets of four sixteenth notes are played, the last three in each slurred together, and then two slurred triplets. The algorithm got into dangerous waters at the end of a long passage of these and marked most strokes in the last bar with an `F` because they occurred in a mixture of simple and complex blocks which forced the algorithm to have a local and not a global view of the situation.

4 Sequential Results

Our goal was to find an algorithm that generated good bowings in tough spots. We first implemented our algorithms in Ada, hoping for a fast implementation. We then measured the sequential times on the same Sun workstation that we ran the Ada version. The results are much worse probably because we used lists rather than arrays. Table 1 summarises the runtimes of both versions of the algorithm shown in a hours:minutes:seconds format.

Input	Runtime (Ada)	Runtime (Haskell)
1	1.3	37.0
2	56:49.6	14:45:49.8
3	0.4	4.9
4	49.6	9:18.9
5	0.4	8.8
6	`n/a`	59.4

Table 1. Sequential runtimes of `A` in Ada and in Haskell

5 GpH Programming Environment

The essence of the problem facing the parallel programmer is that, in addition to specifying *what* value the program should compute, explicitly parallel programs must also specify *how* the machine should organise the computation. There are many aspects to the parallel execution of a program: threads are created, execute on a processor, transfer data to and from remote processors, and synchronise with other threads. Managing all of these aspects on top of constructing a correct and efficient algorithm is what makes parallel programming so hard. One extreme is to rely on the compiler and runtime system to manage the parallel execution without any programmer input. Unfortunately, this purely implicit approach is not yet fruitful for the large-scale functional programs we are interested in.

The approach used in GpH is less radical: the runtime system manages most of the parallel execution, only requiring the programmer to indicate those values that might usefully be evaluated by parallel threads and, since our basic execution model is a lazy one, perhaps also the extent to which those values should be evaluated. We term these programmer-specified aspects the program's *dynamic behaviour*.

Parallelism is introduced in GpH by the `par` combinator, which takes two arguments that are to be evaluated in parallel. The expression `p ‘par‘ e` (here we use Haskell's infix operator notation) has the same value as `e`, and is not strict in its first argument, i.e. $\perp$ `'par' e` has the value of `e`. Its dynamic behaviour is to indicate that `p` could be evaluated by a new parallel thread, with the parent thread continuing evaluation of `e`. We say that `p` has been *sparked*, and a thread may subsequently be created to evaluate it if a processor becomes idle. Since the thread is not necessarily created, `p` is similar to a *lazy future* [8].

Since control of sequencing can be important in a parallel language [11], we introduce a sequential composition operator, `seq`. If `e1` is not $\perp$, the expression `e1 'seq' e2` has the value of `e2`; otherwise it is $\perp$. The corresponding dynamic behaviour is to evaluate `e1` to weak head normal form (WHNF) before returning `e2`.

5.1 Evaluation Strategies

Even with the simple parallel programming model provided by `par` and `seq` we find that more and more code is inserted in order to obtain better parallel performance. In realistic programs the algorithm can become entirely obscured by the dynamic-behaviour code.

Evaluation strategies use lazy higher-order functions to separate the two concerns of specifying the algorithm and specifying the program's dynamic behaviour. A function definition is split into two parts, the *algorithm* and the *strategy*, with values defined in the former being manipulated in the latter. The algorithmic code is consequently uncluttered by details relating only to the parallel behaviour. In fact the driving philosophy behind evaluation strategies is that *it should be possible to understand the semantics of a function without considering its dynamic behaviour.*

Because evaluation strategies are written using the same language as the algorithm, they have several other desirable properties. Strategies are powerful: simpler strategies can be composed, or passed as arguments to form more elaborate strategies. Strategies are extensible: the user can define new application-specific strategies. Strategies can be defined over all types in the language. Strategies are type safe: the normal type system applies to strategic code. Strategies have a clear semantics, which is precisely that used by the algorithmic language.

Evaluation strategies are used to specify the dynamic behaviour in of the bowing program described in this paper; a complete description and discussion of strategies can be found in [14]. A strategy is a function that specifies the dynamic behaviour required when computing a value of a given type. A strategy makes no contribution towards the value being computed by the algorithmic component of the function: it is evaluated purely for effect, and hence it returns just the nullary tuple `()`.

```
type Strategy a = a -> ()
```

Strategies Controlling Evaluation Degree The simplest strategies introduce no parallelism: they specify only the evaluation degree. The simplest strategy is termed `r0` and performs no reduction at all. Perhaps surprisingly, this strategy proves very useful, e.g. when evaluating a pair we may want to evaluate only the first element but not the second.

```
r0 :: Strategy a
r0 _ = ()
```

Because reduction to WHNF is the default evaluation degree in GPH, a strategy to reduce a value of any type to WHNF is easily defined:

```
rwhnf :: Strategy a
rwhnf x = x `seq` ()
```

Many expressions can also be reduced to *normal form* (NF), i.e. a form that contains *no* redexes, by the `rnf` strategy. The `rnf` strategy can be defined over built-in or datatypes, but not over function types or any type incorporating a function type as few reduction engines support the reduction of inner redexes within functions. Rather than defining a new `rnfX` strategy for each data type X, it is better to have a single overloaded rnf strategy that works on any data type. The obvious solution is to use a Haskell type class, NFData, to overload the rnf operation. Because NF and WHNF coincide for built-in types such as integers and booleans, the default method for rnf is rwhnf.

```
class NFData a where
  rnf :: Strategy a
  rnf = rwhnf
```

For each data type an instance of NFData must be declared that specifies how to reduce a value of that type to normal form. Such an instance relies on its element types, if any, being in class NFData. Consider lists and pairs for example.

```
instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs

instance (NFData a, NFData b) => NFData (a,b) where
  rnf (x,y) = rnf x `seq` rnf y
```

Data-oriented Parallelism A strategy can specify parallelism and sequencing as well as evaluation degree. Strategies specifying data-oriented parallelism describe the dynamic behaviour in terms of some data structure. For example parList is similar to seqList, except that it applies the strategy to every element of a list in parallel.

```
parList :: Strategy a -> Strategy [a]
parList strat []     = ()
parList strat (x:xs) = strat x `par` (parList strat xs)
```

Data-oriented strategies are applied by the using function which applies the strategy to the data structure x before returning it.

```
using :: a -> Strategy a -> a
using x s = s x `seq` x
```

A parallel map is a useful example of data-oriented parallelism; for example the parMap function defined below applies its function argument to every element of a list in parallel. Note how the algorithmic code map f xs is cleanly separated from the strategy. The strat parameter determines the dynamic behaviour of each element of the result list, and hence parMap is parametric in some of its dynamic behaviour.

```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f xs = map f xs `using` parList strat
```

5.2 GpH Workbench

GpH programs are developed with a workbench, i.e. an integrated suite of software tools, based on the Glasgow Haskell Compiler [10]. Guidelines for the use of these tools are given in the following subsection. The workbench includes both execution environment and analysis tools, as outlined below.

- *Hugs interpreter*, for fast development, experimentation and debugging of sequential code.
- *GHC Compiler and sequential runtime system* for fast execution of sequential code. The parallel program has the same semantics as it's sequential counterpart.
- *GHC Compiler and GUM parallel runtime system* for parallel execution on multiprocessors. GUM is efficient, robust and portable: being available on both shared- and distributed-memory architectures, including the Sun SPARCServer shared-memory multiprocessor and both a

CM5 [2] and networks of Sun and Alpha workstations. An IBM SP2 port is nearing completion. It is freely available and has users and developers worldwide [13].

The workbench also has a number of analysis tools, most of them dynamic analysers, or profilers. We plan to construct some static analysers, and more parallel profilers are already under development [5].

- *Sequential time and space profilers* supplied with GHC [12].
- *GranSim parameterisable parallel simulator* [6] is closely integrated with the GUM runtime system giving accurate results. It is parameterisable to emulate different target architectures, including an idealised machine, and provides a suite of visualisation tools to view aspects of the parallel execution of the program. runtime system produces a subset of the GranSim profile data and so can produce some of the profiles.

5.3 Parallelisation Guidelines

From our experiences engineering GPH programs we have developed some guidelines for constructing large non-strict functional programs. The guidelines are discussed in detail in [7,14].

1. **Sequential implementation.** Start with a correct implementation of an inherently-parallel algorithm.
2. **Parallelise and tune**.
   - *Seek top-level parallelism.* Often a program will operate over independent data items, or the program may have a pipeline structure.
   - *Time Profile* the sequential application to discover the "big eaters", i.e. the computationally intensive pipeline stages.
   - *Parallelise Big Eaters* using evaluation strategies.
   - *Idealised Simulation.* Simulate the parallel execution of the program on an idealised execution model, i.e. with an infinite number of processors, no communication latency, no thread-creation costs etc. This is a "proving" step: if the program isn't parallel on an idealised machine it won't be on a real machine.
   - *Realistic Simulation.* GranSim can be parameterised to closely resemble the GUM runtime system for a particular machine, forming a bridge between the idealised and real machines.
3. **Real Machine.** The GUM runtime system supports some of the GranSim performance visualisation tools. This seamless integration helps understand real parallel performance.

6 Parallel Algorithm

6.1 Coarse-grained Parallelism

In making `A` parallel, the goal was to assess some of the blocks separately from the others. As assigning a bowing is generally a sequential process, the music had to be broken into segments that started with a complex block and terminated either in a rest or in a series of simple blocks which together contained at least 16 notes. After each of these end points, the initial bow position could be the middle of the bow without disturbing the algorithm's correctness. The following is a fairly crude specification of the contents of a sequence:

```
music_seqs ::= complex_block
                {simple_block + complex_block} Rest music_seqs
             | complex_block
                {simple_block + complex_block} {note}16 music_seqs
             | e
```

Based on this decomposition of the input data the analysis can be performed for each of the segments in parallel. Therefore, we can use the generic `parMap` strategy on a list of segments (see Section 5). Since all parts of the output are known to be required for the overall output the `rnf` strategy is applied to every segment. This requires only the following change in the top level function of the original code:

```
result = concat (parMap rnf (write_music_list .
                             process_music metro) segments
```

The only difference to the sequential code is the use of `parMap rnf` instead of `map`.

Obviously the resulting algorithm is extremely coarse-grained. The amount of parallelism is limited by the number of segments in the input. Table 2 shows the number of segments together with the obtained average parallelism under the GranSim simulator in an idealised setup and a standard setup. The coarse granularity in this program makes it a good program to execute on a high latency workstation network because it requires hardly any communication. However, it is very sensitive towards scheduling decisions made by the runtime-system. We will discuss this in more detail in Section 8.

6.2 Using Granularity Information

In order to minimise the parallel runtime of the program we want to start the largest thread first because it represents the critical path in the computation. To this end we make use of granularity information in the program and generate those threads, which are likely to produce a lot of work, first. One general approach to this general problem is to use a static granularity analysis to estimate the costs of the generated threads and to pass this information to the parallel runtime-system in order to make better scheduling decisions. This is currently being studied in [9].

An alternative to this approach is to develop a strategy that uses a granularity estimate and creates the parallelism in the right order. In this approach the order is under the control of the programmer where the code for describing this structure is localised in one new strategy called `parGranList`. The definition of this strategy is given below:

```
parGranList :: Strategy a -> (a -> Int) -> [a] -> Strategy [a]
parGranList s gran_estim l_in = \ l_out ->
  parListByIdx s l_out $
  sortedIdx gran_list (sortLe ( \ (i,_) (j,_) -> i>j) gran_list)
  where -- spark list elems of l in the order specified by  (i:idxs)
        parListByIdx s l [] = ()
        parListByIdx s l (i:idxs) = parListByIdx s l idxs `sparking` s (l!!i)
        -- get the index of y in the list
        idx y [] = error "idx: x not in l"
        idx y ((x,_):xs) | y==x      = 0
                         | otherwise = (idx y xs)+1
        -- the `schedule' for sparking: list of indices of sorted input list
        sortedIdx l idxs = [ idx x l | (x,_) <- idxs ]
        -- add granularity info to elems of the input list
        gran_list = map (\ l -> (gran_estim l, l)) l_in
```

The purpose of the `parGranList` strategy is to spark all elements in the list `l_out` in an order of decreasing granularity. The function `gran_estim` provides an estimate of the granularity, i.e. of the amount of work to be performed in a parallel thread. Note that this estimate has to be applied to the input list `l_in` determining the order of the sparks in the output list. Thus, this strategy abstracts over the concrete definition of how to compute the results in the output list. The strategy proceeds in four steps:

1. First granularity estimates are added to each list element yielding `gran_list`.

2. Then the resulting list is sorted by these estimates using `sortLe`.
3. In order to obtain a "schedule" for the order in which the list elements should be sparked, a list of indices of the sorted list is computed using `sortedIdx`.
4. Finally, the index-list is used as the schedule for the `parListByIdx` strategy.

For clarity, the current version separates the sorting of the list from obtaining the list of indices, yielding a quadratic algorithm. Clearly, this could be improved by merging both steps. However, in our case, with only up to 39 threads this is unlikely to cause an efficiency problem.

With this strategy we can improve the parallelism in `A` by changing the main part of the code as follows:

```
result = concat (map (write_music_list .
                      process_music metro) segments
                `using` parGranList rnf length segments)
```

The per-thread activity profiles in the following section will show that this strategy indeed sparks the largest threads first. Because of using a very crude estimate for the granularity, the `length` function, the ordering is not perfect. However, it is good enough to avoid a significant delay in generating the dominating thread in the computation (see Figure 2).

7 Results from the GranSim Simulator

Input	Number of Segments	Average Parallelism (GranSim Light)	Average Parallelism (GranSim Standard)
1	23	2.3	2.6
2	17	n/a	n/a
3	5	1.7	1.5
4	5	3.2	n/a
5	5	2.1	1.9
6	39	n/a	n/a

Table 2. Results from parallel simulations of `A`

Table 2 shows the results of simulating the parallel execution of `A` under GranSim. Initially, we profiled the parallel program using GranSim-Light (with the flags `-bP -b: -F2s -H12M`), which simulates the execution on an idealised machine with an infinite number of processors and zero communication costs. The resulting average parallelism is primarily bounded by the number of segments that are processed by the parallel map operation. However, in many cases the generated threads are of largely different sizes, which reduces the achievable parallelism even further. Some of the inputs presented problems to the simulation and no parallelism figures are available for those. Because of the size of the total computation it was impractical to profile the behaviour of the algorithm on the second input.

After the idealised simulation we used the standard setup of GranSim to specify a machine similar to the shared memory parallel machine for which results appear in Section 8.1. This setup uses only four processors with a very low latency of only 40 machine cycles (flags `-bP -bp4 -b-M -bl40 -bG -bQ0 -by3 -bm200 -br200`). The resulting parallelism is very close to that on the idealised machine because of the small amount of communication contained in this program.

Table 1 shows the activity profile of GranSim-Light with Input 5. This profile measures the number of running and of blocked threads over the whole computation of the program. The typical step structure towards the end of the computation results from parallel threads that start at the same time but are of different length. Since no parallelism is generated after the initial `parMap` there is no work that could be picked up by idle processors after having finished the initial thread it has been assigned to.

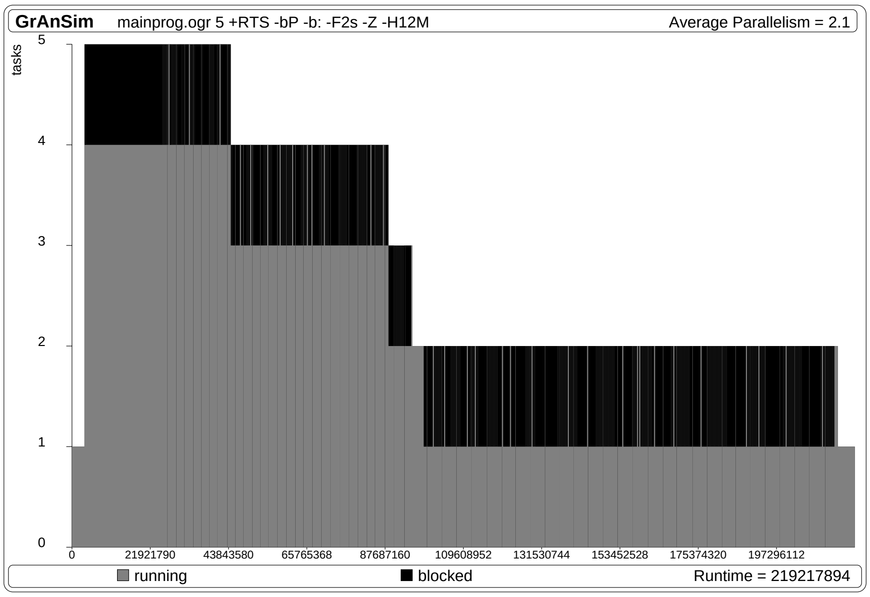

Fig. 1. Activity profile of running `A` under GranSim-Light

8 Multi-processor Results

This section presents results we obtained from running `A` on two different parallel architectures: a shared-memory machine and a workstation network. These two architectures differ sharply in their basic characteristics like latency, i.e. the time needed for communication between processors, and scalability, i.e. the number of processors that can be used.

8.1 Shared-memory Machine

The performance of coarse-grained parallel programs crucially depends on the schedule chosen for the parallel threads. The measurements of the algorithm under GranSim in the previous section has shown that in most cases only a small number of threads are created. Furthermore, these threads have largely varying runtimes with the longest thread determining the overall computation time. This was the main motivation for introducing the `parGranList` strategy in Section 6. However, this strategy only determines the order of generating parallelism, it does not guarantee an even load distribution. When running the algorithm under GUM on a four processor shared memory machine this turned out to be a problem. This is mainly due to the random distribution of work in the GUM runtime-system, which does not try to maintain information about the load of the processors. For programs that generate a large amount of parallelism this is satisfactory because it is sufficient to provide some work for every processor rather than maintaining a perfectly even load distribution. However, in the case of `A` the scheduling of two large threads on the same processor will drastically increase the runtime because there is not enough parallelism in the program to keep the remaining processors busy.

This problem is demonstrated in the per-thread profiles in Figure 2. These profiles show the activity of each of the five generated threads as horizontal lines. Four different states are shown by a varying thickness of the line: a running thread is shown as a thick line, a suspended thread is shown as medium line, a fetching thread is shown as a thin line, and a blocked thread is shown as a gap. Since the computations are independent no thread is blocked during the execution. However, in the left hand graph Thread 2 is suspended between circa 2,800 and 4,500 cycles (medium line). At the same time Thread 5 is running (thick line). This indicates that both threads are executed on the same processor. Thread 2 is descheduled in order to fetch remote data and Thread 5 is resumed during the data transfer in order to overlap computation with communication. However, since Thread 5 can execute for circa 1,700 cycles without interrupt, the critical Thread 2 is suspended, i.e. it could execute but the processor is busy running Thread 5. Similar situations can be observed for Threads 1, 3 and 5. In particular this is a problem for Thread 1, which is represents the critical path in the program and could be finished sooner if it had not been suspended for some time before.

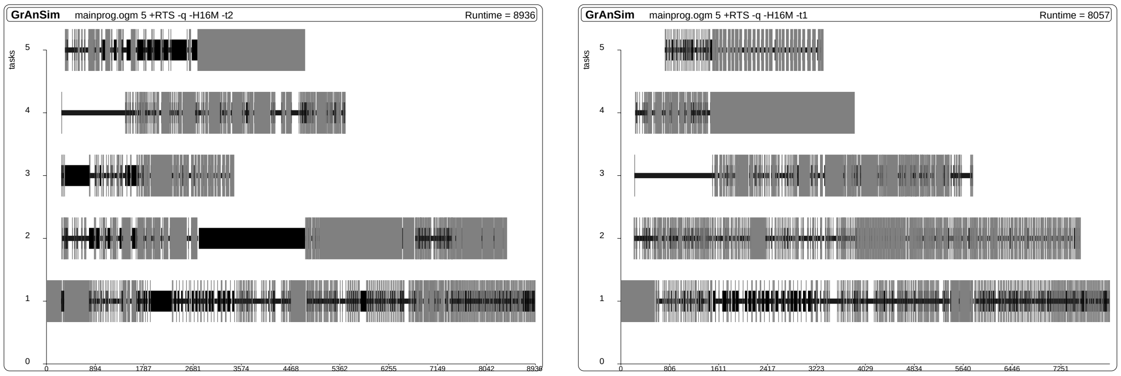

Fig. 2. Per thread activity profiles of `A` with thread pool sizes of 2 and 1 (shared memory machine)

One possibility to avoid such an uneven load balance would be to implement thread migration, which is currently not supported by GUM. With this feature an idle processor would obtain a suspended thread from a busy processor if no other work is available. This, however, is a very expensive operation and is only rarely necessary. As an alternative we use a runtime-system option, which specifies the thread pool size on each processor. In the graph at the left hand side of Figure 2 the thread pool size was 2, which causes two threads to compete for the same processor. In the graph at the right hand side of Figure 2 a thread pool size of 1 has been used. Because no other threads are generated as long as a thread is active on the processor the suspension of large threads disappears and the total runtime drops by about 10%. [1]

Figure 3 summarises the overall activity of the program using a granularity control strategy as well as a minimal thread pool size. During the first half of the computation, when sufficient parallelism is available, the utilisation is satisfactory. In the second half the lack of available threads causes rather poor performance. To improve this behaviour it would be necessary to expose further parallelism in the algorithm. In total we obtain an average parallelism of 1.8 on a four processor shared-memory machine.

[1] In the current version of GUM the main thread is treated specially, which causes very short periods of suspension for Thread 1. This is, however, an artifact of the current implementation rather than a principal problem.

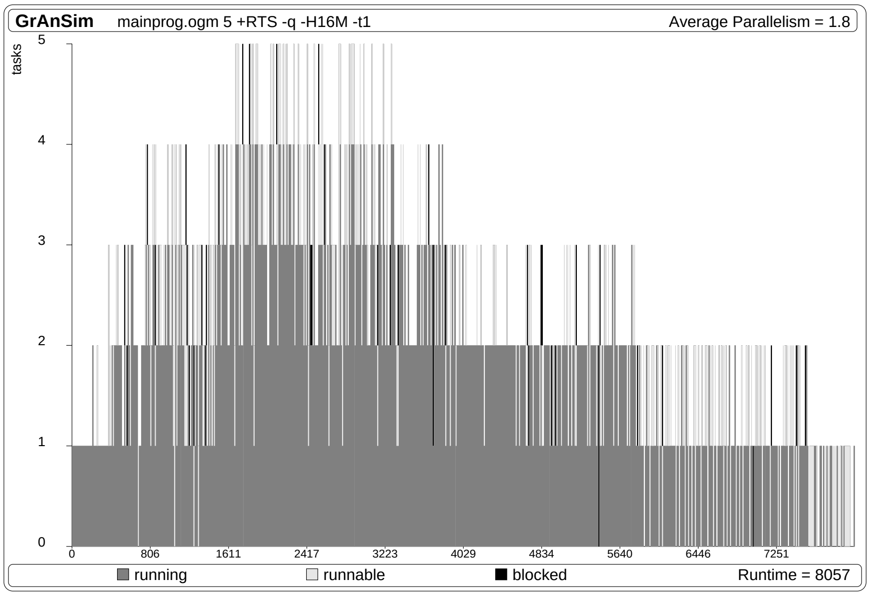

Fig. 3. Overall activity profile of `A` with thread pool size of 1

8.2 Workstation Network

The low amount of communication in `A` makes the program suitable for execution on a workstation network. We have run the program on 8 Suns 4/25 connected via ethernet. This configuration shows a latency of circa 4ms for a minimal packet size. Unfortunately, general set-up problems unrelated to our parallel system have prevented us from obtaining complete results so far.

For Input 2 the generated threads are long enough to generate real parallelism. However, in this case the computation requires more heap than is available on the machines used for these measurements. Therefore, we cannot present a full parallel execution of the program. However, the activity profile of a partial execution in Figure 4 shows a very high degree of parallelism with only a small amount of communication. Of course, this might be dominated by a sequential tail not shown here. However, these preliminary results indicate that the amount of communication will not be a main problem on the workstation network. Therefore, we hope to achieve real speed-ups on this architecture when using input data that generates roughly balanced parallelism without excessive heap usage.

9 Discussion

9.1 Summary

We have discussed an algorithm that calculates bowings for string players and focused on parallelising a Haskell version of this algorithm. Comparisons of the sequential runtimes revealed that a speedup of 8 to 15 would be required for the parallel Haskell version to match the sequential runtime of the existing Ada version. We believe that the use of lists rather than arrays in the Haskell version is the main source of inefficiency. In order to improve sequential performance of the Haskell code we plan to profile the code and replace the lists with tree data structures.

The initial parallelisation of `A` was extremely simple: we only had to replace one `map` function by a strategic `parMap rnf` construct. This exploits coarse-grained parallelism with a very small

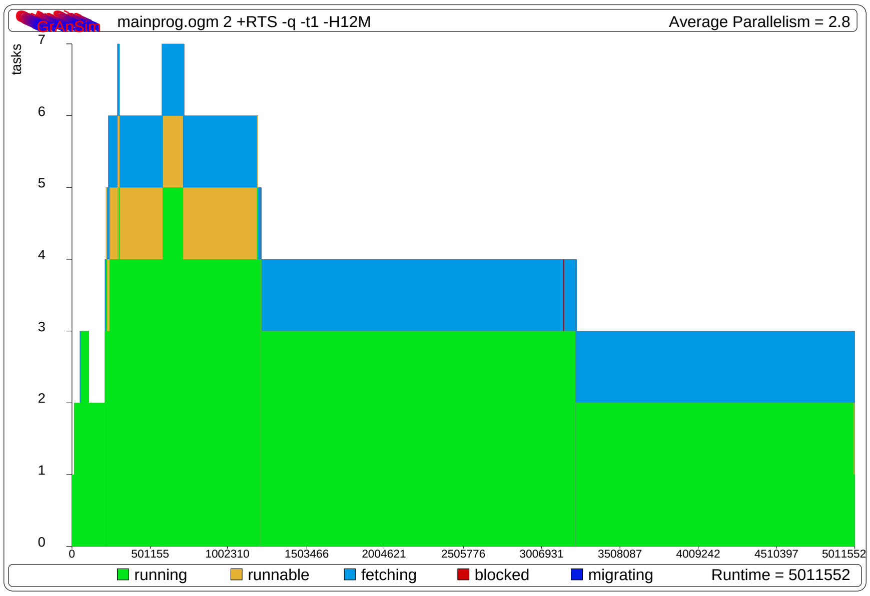

Fig. 4. Partial activity profile of running `A` on a workstation network (8 PEs)

amount of communication. However, if the generated threads differ largely in their relative sizes the resulting speedup is very small because the total runtime is determined by the runtime of the longest parallel thread. Therefore it is important to influence the scheduling of the parallel threads to some degree. We have done this on two different levels:

- On the program level we used a granularity control strategy `parGranList` to create the largest threads first; and
- on the runtime-system level we use a parameter for tuning the thread pool size in order to avoid an imbalance in the workload.

Both improvements together led to small but measurable runtime improvements of the parallel algorithm.

9.2 Future work

As future work we plan to perform more systematic measurements on the workstation network. Due to the low amount of communication we hope to achieve good speedups on this architecture. We also would like to increase the amount of parallelism in the program. One possibility would be to add pipeline parallelism in the top level function. Currently, the parallelism is generated in one stage of a series of function applications. Using the strategic function application operator we have developed as part of the strategies module [14] it would be easy to turn the series of function applications into a parallel pipeline. Because of the imbalance of the thread sizes it would also be advantageous to decompose each thread into smaller pieces of potentially parallel code.

Based on the experiences we obtained in parallelising and tuning the Haskell algorithm we plan to develop a parallel Ada version, too. In this step we can take advantage of the easy prototyping of different variants of parallelism in the Haskell code using evaluation strategies and pick the most efficient version for the Ada code. We expect this process to be far more cost effective than experimenting directly with different versions of a parallel Ada code.

There is also a wide variety of potential improvements of the quality of the result produced by the algorithm:

- We currently do not calculate the bow's distance from the bridge (although notes that are marked as short should be played closer to the bridge).
- Bow speed and bow weight (pressure) are also issues that should be addressed.
- More needs to be done on propagating as much context information as possible, so that a simple algorithm can look ahead and take advantage of it.
- We also would like to help the musician find out which notes must be slurred in order to escape bowing problems, and distinguish these from musical slurs. At present, musicians mix all of these together; we may someday help purists who would like to stick with the original bowing as much as possible.
- We may also develop an interface in which the musician can hear the results of this work, by driving a sample playback synthesiser. While this would tend to make the string sound more realistic, the individual bowings would not usually be audible. However, if we can calculate bow articulation, then we can produce music played in the baroque style, which would be audibly different from the modern style.
- Lots of rhythmic patterns and their usual solution could be incorporated into the algorithm.
- There is an interesting problem in specialising this work to the cello as opposed to the violin.

References

1. Askenfelt, A., *Measurement of the bowing parameters in violin playing. II:Bow-bridge distance, dynamic range, and limits of bow force*, J. Acoust. Soc. Am. 86(2), 503-516.
2. Davis K. "MPP Parallel Haskell" *Proc. IFL '96*, Bonn, Germany, (September 1996) Springer Verlag (in press).
3. Flesch, C., *The Art of Violin Playing*, Bk. 1, Carl Fischer, Inc. 1939.
4. Findlay, B. and Hall, C. *Simulation of Bowing Decisions on a String Instrument*, Quatriemes Journees d'Informatique Musicale (JIM'97), June 6-7, Lyon, France, 89-97.
5. K. Hammond, H-W. Loidl, and P.W. Trinder. Parallel Cost Centre Profiling. In *IFL'97 — Intl. Workshop on the Implementation of Functional Languages*, September 10–12, St. Andrews, Scotland, 1997.
6. Hammond K., Loidl H-W., and Partridge A.S. "Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell", *Proc. HPFC'95 — High Performance Functional Computing*, Denver, Colorado, (April 1995), pp. 208–221.
7. Loidl H.-W. and Trinder P.W. "Engineering Parallel Functional Programs" *Proc. IFL '97*, St Andrews, Scotland, (September 1997).
8. Mohr E., Kranz D.A., Halstead R.H., "Lazy Task Creation – a Technique for Increasing the Granularity of Parallel Programs", *IEEE Transactions on Parallel and Distributed Systems* **2(3)** (July 1991), pp. 264–280.
9. Loidl H.-W. *Granularity in Large-scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow. In preparation.
10. SL Peyton Jones, "Compilation by transformation: a report from the trenches", in *European Symposium on Programming (ESOP'96), Linköping, Sweden*, Springer Verlag LNCS 1058, pp 18-44, Jan 1996.
11. P Roe, "Parallel programming with functional languages", PhD thesis, CSC 91/R3, Department of Computing Science, University of Glasgow, April 1991.
12. Sansom, P.M., and Peyton Jones, S.L., "Time and Space Profiling for Non-Strict, Higher-Order Functional Languages", *Proc. POPL'95*, (1995), pp. 355–366.
13. Trinder P., Hammond K., Mattson J., Partridge A., and Peyton Jones S.L. "GUM: a Portable Parallel implementation of Haskell". Proceedings of Programming Languages Design and Implementation, Philadelphia, USA, (May 1996).
14. Trinder P.W. Hammond K. Loidl H-W. Peyton Jones S.L. "Algorithm + Strategy = Parallelism". *To appear in* The Journal of Functional Programming, **8(1)** (January 1998).