PASTHA – Parallelizing Stencil Calculations in Haskell

Michael Lesniak

Research Group Programming Languages / Methodologies
Universität Kassel
Kassel, Germany
mlesniak@uni-kassel.de

Abstract

Stencils are typical building blocks for many numerical scientific applications. Different parallelization methods exist, the choice of a method depends on the given stencil, parallel programming system etc. Implementing stencils in a library simplifies application programming, allows to experiment with different parallelization methods, and supports their automatic adaptation to a given stencil. This paper introduces *PASTHA*, a prototype for a Haskell library that allows to declaratively describe stencil-based problems and calculate them in parallel. The description is flexible enough to cover all 2D stencils we are aware of. Implementation is based on task queues and strict evaluation. We report on experiments with a Gauß-Seidel stencil, where we achieved speedups of up to 4 on six cores, and with global and local sequence scoring from the Haskell bioinformatics library `bio`. For local scoring, the running time was reduced by a factor of 55, which is partially due to PASTHA.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Languages, Performance

1. Introduction

Stencils are building-blocks for numerical scientific applications. A stencil is a pattern to describe the calculation of matrix elements by defining the relative positions of elements required for the calculation. The values at these positions are used to update the value of the current position. Stencils are used for example in the approximation of partial differential equation systems by applying iterative numerical approximation methods such as the Jacobi or Gauß-Seidel relaxation method.

Despite the progress in both hardware and software development the demand for increasing processing power of stencil codes is still prevailing. Greater processing power leads to more detailed simulations, increases the quality of the results or reduces the time for finishing them. Therefore improving the calculation speed of stencil-based applications leads to a general improvement of numerical applications.

The current solution to this problem is no longer limited to increasing clock rates: instead parallelization of applications executed on modern shared-memory multicore architectures is favoured. Unfortunately the increase of computing power comes with a price: not only need the developers handle the single-threaded correctness of their programs but have to handle for example thread synchronization and race conditions as well at least in the traditional explicit thread-parallelism model.

One possibility to manage the complexity while still enabling parallelism is to hide it from the developer. Libraries are well known to achieve this goal: Firstly, they can handle the parallel aspects of the execution. Secondly, they can provide (semi-)automatic techniques to choose the most performant implementation depending on the specific problem. Thirdly, by allowing a declarative description of the problem calculation, details are abstracted. Developing a library to handle the parallel calculation for a class of problems like stencil calculation should fulfill some constraints. These constraints should make the library useful for real-world applications:

- The provided functions of the library should be flexible enough to be applied on different classes of problems while at the same time the learning curve for simple examples should be small.
- The library should use the resources of modern multicore hardware and provide a good speedup.
- The efficiency of the provided functions should not be significantly worse than the efficiency of existing hand-coded versions. On the other hand small decreases in computation speed for having a more convenient development environment are a common trade-off.
- Depending on the problem type it should assist – or even automatize – the choice for the most performant implementation.

Haskell provides a solid basis for parallel programming due to its active research on parallel programming models and the resulting developer-friendly parallel programming techniques. Since Haskell supports parallelization using explicit, semi-implicit and data parallel techniques, it allows to explore and compare the advantages and disadvantages of the different approaches to parallel stencil calculation while using a common code basis and common abstractions. As a functional language, its support for higher level abstraction and thus declarative stencil descriptions and parallelization strategies is better than in most imperative programming languages. Our contributions are

- The definition of types in Haskell that allow to declaratively express 2D stencil-based problems. Addressing both multiple data matrices and arbitrary referencing of values from past iterations is possible.
- *PASTHA*, the prototype of a library for parallel stencil calculations. It allows us to evaluate parallel strategies for stencil-based calculations and to calculate stencil-based problems on multicore architectures. Our implementation is based on task

DAMP'10, January 19, 2010, Madrid, Spain.

queues and strict evaluation by which we achieve speedups for Gauß-Seidel-based stencils of 3 on four and 4 on six cores for instance.

- The parallelization of the affine sequence score algorithm in the Haskell bioinformatics library[1]. Thereby we explore the advantages and disadvantages of the library's current parallelization strategies and provide some real-world benchmarks and experiences. It was of special importance to us that biolib users should be able to use the parallelization without having to change their code too much. We were able to achieve speedups of up to 55 on four cores. This speedup is due to strict evaluation, computing in the IO monad, and partially the use of PASTHA.

Despite being a prototype implementation, PASTHA can already be used to calculate some commonly used and declaratively described stencils in parallel. For other stencils the user has to define two simple functions to describe the dependency relation between stencil elements. Due to the prototype nature of PASTHA, our design decision was aimed at perspicuity: When having the choice between a simple, clean and possibly slower implementation and a more complex one (for example, involving customized shared data structures, STM etc.) we chose the simpler one.

The remainder of the paper is structured as follows. Section 2 introduces a voltage diffusion simulation. This voltage diffusion simulation will serve as a guiding example throughout the paper. We formalize the different components of a stencil-based problem in a declarative form and present types for these components in Haskell. Section 3 points out how to define the voltage simulation in Haskell and initiate its parallel calculation. Section 4 explains the general parallelization scheme for this class of problems, describes our implementation in Haskell and provides benchmarks. In Section 5 we give a short introduction to sequence scoring, demonstrate how to transform programs for using PASTHA and provide benchmarks comparing the original implementation of the `biolib`-algorithms with the PASTHA-version. Section 6 compares this paper to related work and Section 7 concludes and describes future research ideas.

2. A declarative model of stencil problems in Haskell

In this section we give reasons for the usage of stencils by explaining how to model the diffusion of voltage on a metal sheet and calculate the voltage's distribution. We generalize the introduced terms of the example in such a way that arbitrary stencil-based problems can be expressed. Using the defined terms we show how to transform the definitions into Haskell types.

2.1 A voltage diffusion simulation

Our aim is to determine the resulting pattern voltages of a two-dimensional conductive metal sheet with constant voltage applied only at the boundaries (Figure 1). The resulting pattern is modeled by the linear partial differential Laplace Equation

$$\frac{\partial^2 v}{\partial^2 x} + \frac{\partial^2 v}{\partial^2 y} = 0 \tag{1}$$

Since this equation can only be solved numerically we apply the Jacobi relaxation method on a two-dimensional grid of points on the surface of the metal sheet. Let $V_{i,j}$ denote the voltage at the grid point (i, j). Applying (1) to the grid results in the equation

$$V_{i,j}^{(k)} = \frac{V_{i-1,j}^{(k-1)} + V_{i+1,j}^{(k-1)} + V_{i,j-1}^{(k-1)} + V_{i,j+1}^{(k-1)}}{4} \tag{2}$$

The approximation algorithm iterates successively over all internal

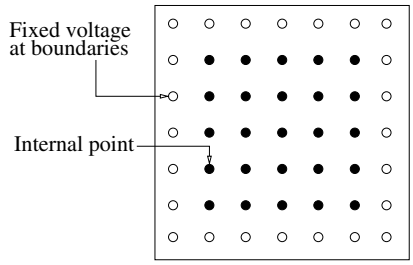

Figure 1. A two dimensional conducting metal sheet.

```
// Alternative:
// while (!converged) ...
while (steps++ < maxSteps) {
  for (i=0; i < N; ++i)
    for (j=0; j < N; ++j)
      VNew[i,j] = (V[i-1,j] + V[i+1,j] +
                   V[i,j-1] + V[i,j+1]) / 4.0;
  Copy(VNew->V);
}
```

Figure 2. Code to calculate values for the voltage diffusion simulation using a Jacobi-stencil.

points (i, j) evaluating $V_{i,j}$, as shown in Figure 2. This is repeated until either a fixed number of iterations has been calculated or the minimal difference of the same matrix element for the current and previous iterations is smaller than a threshold value ε. The voltage simulation already contains all typical elements of a stencil-based problem: a data matrix with initial values, a stencil described by references to other elements of V, a function computing the average of the neighbour elements defined by V for each grid point, and a convergence condition.

While the Jacobi stencil only refers to values of the previous iteration, there are also commonly used stencils with more advanced dependencies (see Figure 3): The Gauß-Seidel stencil is similar to

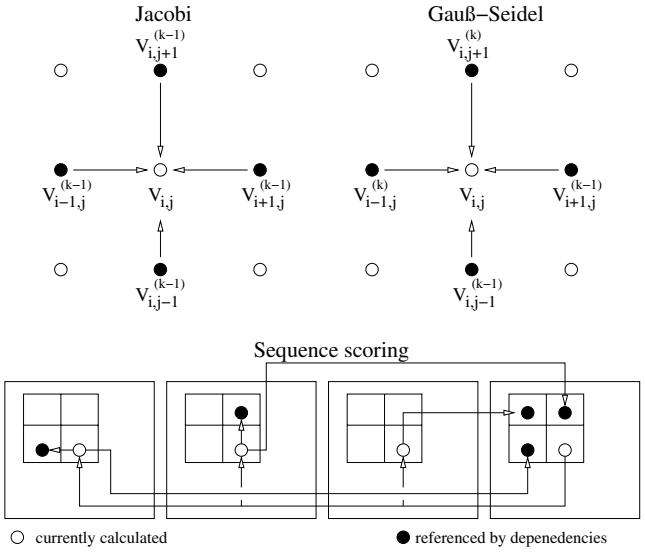

Figure 3. Jacobi, Gauß-Seidel and sequence scoring stencils

a Jacobi stencil but its upper and left element are from the current iteration not the previous one. The stencil we use in Section 5 to parallelize sequence scoring is based on the wavefront stencil: an

element depends on the values from the upper, left, and upper-left elements that have to be calculated in the previous iteration. In this particular case the calculation is more complicated because a total of four data matrices is needed, all matrices having interdependencies of their own.

The next section shows our approach to make PASTHA applicable for many stencil-based problems by introducing a general declarative stencil description.

2.2 A generalized declarative stencil definition

A general definition for stencil-based problems should allow the description of common 2D stencils, and moreover needs mechanisms to describe stencil problems with an arbitrary number of data matrices and references to past iterations. The following definition accounts for this.

A *stencil-based problem* is described by the dimensions and number n of the data matrices, the initial values in the matrices, a tuple of n (stencil, function) pairs, and a convergence condition. This can be formalized in a 5-tuple where $w, h, n \in \mathbb{N}$ are the dimension and number of the data matrices:

$$((w,h), n, init, (stencil \times function)^n, conv)$$

Note, that all data matrices need to have the same dimensions, which is the case in all stencil problems we are aware of. Each initial value is defined through a tuple from $((W,H,N),\mathbb{R})$, where $W = [1 \dots w], H = [1 \dots h], N = [1 \dots n]$, that is for each given coordinate a value is specified. To cover stencil-based problems with more than one data matrix, we need to define tuples of $(stencil, function)$-pairs. Each pair defines the dependencies in one data matrix, as well as the corresponding update function.

While it would have been more convenient for the user to describe both the dependencies and how to update elements in a single definition, splitting the dependency definitions and calculation functions simplifies the analysis of dependencies and thus allows a performant parallelization.

A *stencil* is defined by a k-tuple of four-dimensional tuples where each element of the list defines the dependency on another element:

$$stencil = (\mathbb{N}^-, \mathbb{Z}, \mathbb{Z}, N)^k$$

for an arbitrary $k \in \mathbb{N}$. The first tuple-element describes the referenced iteration: the current one is numbered 0, the previous -1, and so on. The second and third elements define the horizontal and vertical coordinates of a referenced element in the *data matrix*, relative to the current position. The last element defines which of the data matrices is referenced. The Gauß-Seidel stencil can thus be defined by

$$\{(0,-1,0,0), (-1,1,0,0), (-1,0,-1,0), (0,0,1,0)\} \tag{3}$$

The updated values for each element in a data matrix are calculated by a *function*. It receives the values of the elements referenced by a stencil and can perform arbitrary complex operations. Since some functions need the coordinates of the currently calculated element, they are a parameter:

$$function = (W,H) \times \mathbb{R}^k \to \mathbb{R}$$

The *convergence condition* is checked after each iteration (the data for the check can be calculated while elements are processed); if it is fulfilled, the calculation terminates. The first common condition is ε-difference: if

$$\min_{i,j} |V_{i,j} - V^{-1}_{i,j}| \leq \varepsilon \tag{4}$$

holds for a defined threshold value ε and V^{-1} as the data matrix of the previous iteration, the calculation stops. The second one is to terminate after a fixed number of iteration steps. Formally we introduce the set

$$conv = \{(\text{Epsilon}, \mathbb{R}), (\text{Steps}, \mathbb{N})\}$$

to distinguish between the two possibilities.

The voltage diffusion simulation can now be expressed as a stencil-based problem by

$$((w,h), 1, init, (s,f), (Epsilon, \varepsilon))$$

with

$$f((x,y), l, r, b, t) = \frac{l+r+b+t}{4}$$

and s defined by equation (3). The particular values for w, h and $init$ depend on the requested precision and the voltages at the boundaries.

2.3 Describing stencils in Haskell

A generalized stencil-based problem is defined by a type `StencilProblem`

```
data StencilProblem a = StencilProblem {
       stencils     :: [Stencil]
     , functions    :: [Function a]
     , matrix       :: Matrix a
     , convergence  :: Convergence
}
```

which precisely describes the components of the abstract definition. To support more than one data matrix, both stencils and functions are lists. In the calculation of the nth data matrix the nth stencil and nth function of these lists are chosen for the value update. Therein

```
type Stencil = [(Int,Int,Int,Int)]
```

and

```
type Function a = (Int, Int) -> [a] -> a
```

A function receives the coordinate of the currently calculated element and a list of values obtained by reading the data matrix values referenced by the appropriate stencil.

To store the data matrices and all necessary past iterations, a four-dimensional unboxed IO array is used:

```
type Matrix a = IOUArray (Int,Int,Int,Int) a
```

While staying pure is naturally preferable, pure arrays do (currently!) not offer the performance achievable by IO based ones. Since we provide functions to access arbitrary data matrices of specific iterations (see Section 3) a user can conveniently work with a `Matrix` without having to have its internal representation in mind.

The convergence condition is specified by the type `Convergence`

```
data Convergence = Epsilon Double | Steps Int
```

and defines either a ε-threshold or a maximal number of steps.

In this section we have shown how to generalize a stencil-based problem and define the components in terms of Haskell types. The next section exemplifies these types by applying them to the voltage diffusion simulation and demonstrates how a parallel calculation is initiated.

3. Using PASTHA

In this section we use the declarative definitions for stencils to describe and calculate the voltage diffusion simulation in Haskell. Additionally we introduce some of the helper functions.

By using the types of Section 2.3, we define and calculate the voltage diffusion problem for a grid size of 2000×2000 and 10 iterations by:

```
import Pastha.Types
import Pastha.Calc.TaskQueue

voltageProblem :: Int -> Int -> Int ->
                  IO (StencilProblem Double)
voltageProblem width height steps = do
    let stencil  = gaussSeidelStencil
        function = \_ vals -> sum vs / 4.0
        converge = Steps steps

    sheet <- newMatrix 1 0 width height >>=
             fillBoundaries 200.0 400.0
                            500.0 100.0

    return $ StencilProblem [stencil]
                [function] sheet
                convergence

main = do
    problem <- voltageProblem 2000 2000 10
    run 4 50 50 problem

    let dm = dataMatrix problem
    return ()
```

PASTHA already defines common stencils, e.g. for Jacobi, Gauß-Seidel, and Wavefront-based stencils. To hide the internally used `Matrix`-type, convenience functions (for example generating matrices with the right dimension for common stencils) are already provided: The call to `newMatrix` receives the needed iteration depth, number of data matrices, and their width and height; `fillBoundaries` fills the boundaries of the data matrices with the corresponding values.

To calculate the data matrices for the defined stencil, we import one of the calculation modules of `Pastha.Calc` and call the respective `run`-function. In addition to the problem `run` receives parameters which depend on the chosen module. For the `TaskQueue`-based strategy they are the number of threads to use and the horizontal and vertical block sizes (see Section 4.4). The function `dataMatrix` is a convenience function to provide transparent access to the values of the first data matrix. Currently a user needs to choose both the number of threads and the block sizes. We plan to develop mechanisms to automatically determine these parameters in future research (see Section 7).

4. Parallel stencil calculation using PASTHA

After giving an overview of the different existing parallelization techniques currently supported by Haskell we describe our implementation for the parallelization of common stencil calculations, explain our design decisions, and provide benchmarks.

4.1 Parallelization in Haskell

Haskell supports three types of parallelization for multicore architectures: semi-implicit, data-parallelism, and explicit. Semi-implicit parallelization is characterized by the `par`- and `pseq`-functions. These functions offer the possibility to annotate those parts of an expression that are a potential source for parallelization – the user does not need to handle threads, communication, and synchronization at all. Data-parallel Haskell allows you to compute independent data elements in parallel. The elements are expressed in a vectorized form and thus provide a natural source for parallelization known as data parallelism. By hiding parallelism, both techniques complicate reasoning and in-detail measuring of performance.

The explicit type of parallelization supported by Haskell is characterised by parallel threads and manual handling of their creation, communication, and synchronization. An arbitrary number of threads is started using `forkIO` and the communication between them is accomplished over `MVars`: these are thread-safe containers for values that block a thread either when it tries to read and no value has been stored or it tries to write and the previously stored value has not been read yet. For many parallel scenarios this simple synchronization scheme is not sufficient: threads do not want to send or receive single values but need to continuously exchange data for synchronization. *Channels*, built on top of `MVars` implement a synchronized concurrent unbounded FIFO-stream for an arbitrary number of values of the same type: reading of a channel blocks if the channel is empty but writing is always possible.

For our prototype implementation of PASTHA we chose explicit threading and channels for the communication between threads: this parallel programming model offers clear semantics, performant synchronization and direct control over the parallel execution at the cost of extra complexity. It also offers manual profiling, for example by using logfiles, in the IO monad. Manual profiling was needed because compilers have no support for parallel profiling. This drawback will change by the rise of a parallel profiler in the next version of GHC [13].

4.2 How to calculate stencils in parallel

It is difficult to parallelize stencil calculations in such a way that it still has significantly better performance than a sequential solution. In a stencil-based problem elements depend on other elements' values, hence dependencies of the order of calculation are introduced. For our explicit parallel programming model this implies that threads have to be synchronized to calculate elements in the correct order and we will have to use data structures allowing concurrent access.

To utilize all available threads and take advantage of modern caches, we use the well-known block-based approach for parallel stencil calculation, as for example described in [16]: the matrix is divided into blocks, which are calculated in an order that satisfies the data dependencies: Every block contains border elements which in order to be calculated require elements of the adjacent blocks in addition to the elements of the same block. Just like a stencil element the block can only be calculated if all adjacent blocks it depends on have already been processed (see Figure 4). To

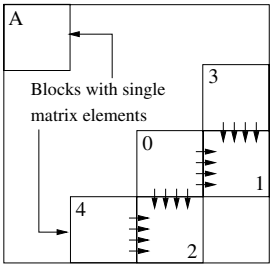

Figure 4. Block-dependency scheme of a Gauß-Seidel based stencil, arrows represent dependencies. Block A can be calculated directly because it has no unfulfilled dependencies. Block 2 can be calculated only if both Blocks 0 and 4 have been processed.

calculate the whole matrix of one iteration, the initially independent block is calculated. By finishing the calculation, dependencies of other blocks are fulfilled and can now be processed. This is repeated until all blocks of the matrix have been calculated.

The convergence condition plays the second important role in the parallelization. In a block-based parallelization the minimal ε-difference is calculated as follows:

- For each block a local block-dependent minimal ε is calculated.

- When a block's calculation is finished the current (global) ε is updated if the local one is smaller.

At the end of an iteration the global ε contains the minimum over all local ε.

There are different ways to implement the needed synchronization, varying in their general applicability, performance etc. For a Gauß-Seidel based stencil one possibility is a *Wavefront-synchronization* with blocks that can be realised as follows (see Figure 5): The matrix is divided into diagonals, whereby the blocks in

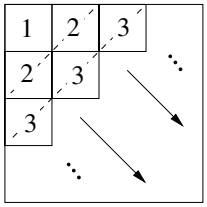

Figure 5. Diagonalization in a block-based wavefront parallelization.

each diagonal can be calculated independently by available threads. Finished threads wait until all blocks in the current diagonal have been calculated before starting with the next one. The iteration is completed when the last block in the lower-right corner has been calculated. The synchronization differs depending on the underlying parallelization runtime system: Haskell uses channels and `MVars`, OpenMP[6] uses barriers after each diagonal, and MPI[17] sends messages both for synchronization and updating of border values (cp. [16] for more information about OpenMP and MPI Wavefront implementations).

On a more global level we can picture other, more advanced parallelization strategies, for example strategies which are more adapted to specific classes of stencils: a strategy *only* to be applied to Jacobi-based stencils needs fewer synchronizations because blocks are independent. This particular strategy typically performs better but is not applicable on a broader basis. For the prototype implementation of PASTHA we wanted to cover the parallelization of all classes of 2D-stencils, hence we developed and implemented a much more general scheme using taskqueues. This scheme is introduced in the next section.

4.3 Parallel stencil calculation with Taskqueues

Our implementation for a block-based parallel processing of arbitrary 2D-stencils is based on the scheme in Figure 6. A main thread supervises the calculation and coordinates the worker threads that only do the number crunching. We decided to include a main-thread since other parallelization approaches complicate the communication about (partially) finished blocks between worker-threads and require more advanced synchronization of data structures.

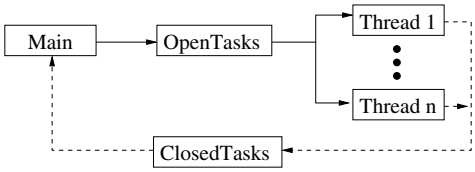

Figure 6. The communication scheme between the main- and worker threads.

For communication between the main thread and all worker threads (task)queues are used: in general a *taskqueue* is a concurrent data structure used in parallel programming to distribute tasks and other messages among threads. Each thread polls the queue for new messages and can write messages back as well.

To distribute blocks among the worker threads, a channel of type `OpenTasks` (called *open-channel*) is used:

```
type OpenTasks = Chan Task
```

To describe messages for this channel, we define a `Task` by

```
data Task =
       Task Block
     | Stop
```

Thus this `Task` can be either a block that can be calculated or a message stopping the receiving thread. Each block is described by a type `Block`

```
type Block = (Int, Int, Int, Int)
```

which specifies the horizontal and vertical coordinates and with this the block's dimension as part of the matrix.

Results are sent to the *back-channel*, defined by the type `ClosedTasks`

```
type ClosedTasks = Chan Result
type Result = (Block, Successors, Double)
type Successors = [(Block, Dependencies)]
type Dependencies = [Block]
```

The *back-channel*'s messages describe which block has been finished, what other blocks can be calculated now if the dependencies are fulfilled (called *successors*) and describes the minimal difference between iterations for the finished block to check for ε-convergence. Note that not blocks themselves are sent back but simply their coordinates and dimensions.

So far we have just defined what types of messages can be sent and what types they consist of. Still missing is the order in which unprocessed blocks can be inserted into the queue. For each stencil two additional functions are needed. These functions describe which successive blocks can be calculated and what dependencies they still have: The initialization function `init :: [Task]` defines all initial blocks that can be calculated without further dependencies, that is those blocks the parallelization can start with. The successor function `next :: (Int,Int,Int,Int) -> Block -> Successors` receives the dimensions of the matrix and the block size as the first parameter. For a finished block it returns the list of successors as a list of tuples; each tuple-element contains a block that can now be calculated and a list of the remaining dependencies. PASTHA currently provides those functions for common stencils, e.g. Jacobi, Gauß-Seidel, and Wavefront-based ones. The underlying scheme for the `init`- and `next`-function for a Gauß-Seidel stencil is shown in Figure 4: Block A is returned by the `init`-function; it does not have any dependencies and calculation can start immediately. When `next` is called with block 0 it returns `[(1,[3]), (2,[4])]`: block 1 depends on block 0 and to be calculated correctly also needs block 3 to already be computed.

The parallel calculation of a stencil is now done as follows: the main thread first forks the threads. For each iteration, i.e. until all blocks have been calculated it then

- Fills the open-channel with the blocks returned by `init`.
- Waits for a block on the finished-channel.
- With a finished block:
  - Compares and stores the minimal ε for this iteration.
  - Checks whether the dependencies of the successor blocks have already been fulfilled; if so, puts them in the open-channel, if not discards them.

  We use `Data.Set` with a lookup-complexity of $O(\log n)$ to both store and check for finished blocks.

After all blocks have been calculated the minimal ε is checked. If it is lower than the value defined by the user the calculation is terminated by filling the open-channel with one `Stop`-message for each thread. Otherwise, the next iteration begins.

Each worker-thread tries to pull messages of the open-channel, blocking until a message is received. With a message a thread

- Checks whether `Stop` is received: if so it terminates. Otherwise it has received a block.
- Calculates both the values of the matrix for the given block and the minimal ε.
- Wraps the result into a `Result`, sends it back over the closed-channel and polls the queue for the next message.

The next section provides benchmarks that show the achieved speedups for different stencils and analyzes potential optimizations for this approach.

4.4 Benchmarks

We tested the taskqueue-based implementation on a 2.2 GHz 8-core AMD Opteron 875, a Linux-kernel 2.6.18, and GHC 6.10.4 to measure the speedup for the voltage simulation using the Gauß-Seidel and for comparison a Jacobi-stencil. Additionally we describe the effect of varying block sizes during the performance. When utilizing more than four cores we sometimes had huge discrepancies in the run time (According to [15] performance problems are encountered when using eight cores with GHC under Linux). Therefore we ran each test twenty times and always report the minimum, the maximum, and the average speedup. For our tests we used a non-threaded sequential version (an implementation of the algorithm of Figure 2) and a parallel version which used two up to seven cores. Preliminary tests with development versions of GHC (to avoid the performance problems mentioned above) sometimes resulted in more speedup but were not reliable and significant and therefore are not presented here.

The first benchmark measured the speedup for the voltage simulation on a 2000×2000 grid with a uniform blocksize of 50 for ten iterations. Experiments with ε-difference came to similar results. The blocksize 50 delivered the best average speedup. The speedups are shown in Figure 7. We receive in the best as well as in

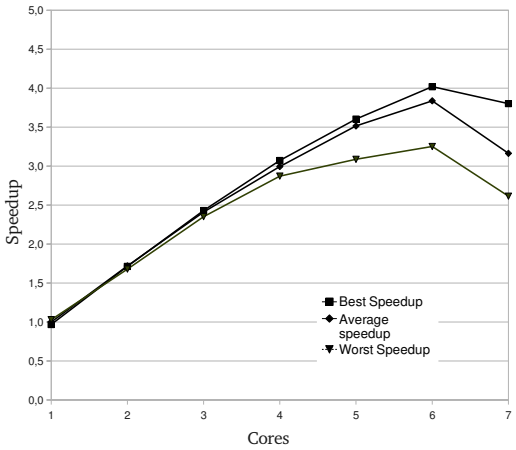

Figure 7. Speedup of the voltage diffusion simulation for a 2000×2000 grid, a uniform blocksize of 50 and 10 iterations

the average case speedups of up to 4 on six cores. Nevertheless the achievable speedup is restricted by the parallel runtime overhead for instance by the cost for synchronization and communication.

Thereupon we simulated the voltage diffusion using a Jacobi stencil. This stencil only needs values of the previous iteration, hence the `init` and `next`-functions are much easier to implement and less communication about finished blocks is needed. Figure 8 shows the according speedup graph. As can be seen, the decrease

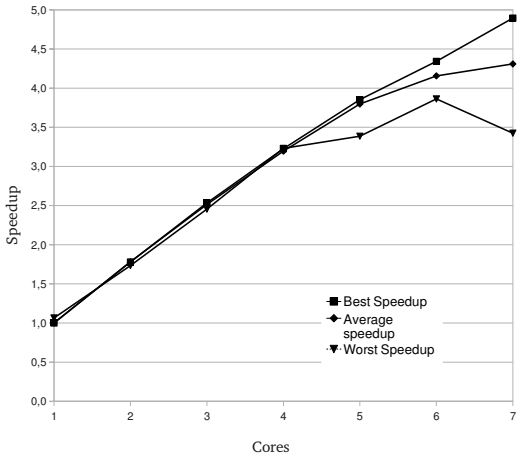

Figure 8. Speedup of the voltage diffusion simulation using a Jacobi stencil

in the communication pays off: since all tasks can be calculated independently we receive better speedup. This indicates that the implementation of the taskqueue does not provide a huge overhead; rather it costs little when it is used only to pull out values and discard the sent-back results.

The performance of the calculation depends on different factors, e.g. on the specific calculation, the number of threads, the parallel runtime system, and the block size. Figure 9 shows the relation between uniform blocksizes of various sizes and the achieved speedup for the voltage problem using four cores. Even though the tolerance

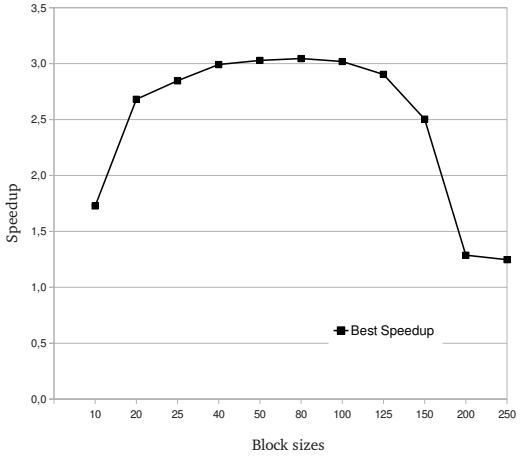

Figure 9. Dependency between block sizes and speedup.

for good speedups is huge (blocksizes from 40 up to 100 provide nearly the same speedup) the user still has to run a few tests with different block sizes to identify the block size for the best speedup. We plan to automatize this step (see Section 7).

5. Parallelizing sequence scoring and alignment

We transformed a class of sequence analysis algorithms into our stencil description and compared the parallel performance with the

original sequential implementation. The sequential implementation was taken from the Haskell bioinformatics library `bio`, which is a collection of bioinformatics-related data structures, algorithms and supporting utilities.

After a short introduction into sequence scoring algorithms (where we leave out the biological background) we show the transformation of the algorithms into a stencil-based problem and how an existing call to a `bio`-library function has to be rewritten to allow parallel calculation. We conclude with a benchmark comparing the sequential with our parallel implementation.

5.1 Background: Sequence scoring

One of the most common and at the same time computational expensive problems in bioinformatics is the comparison of two genomic sequences. A genomic sequence consists of strings of the alphabet $\{A, C, G, T\}$ of nucleotide acid bases.

Given two sequences S_1 and S_2 we want to determine the similarity between the entire sequences which is called *global scoring*. With an exemplary score-function $c(x, y)$

$$c(x,y) = \begin{cases} 1 & x = y \\ 0 & x \neq y \\ -1 & x = - \; or \; y = - \end{cases}$$

covering the cases of a match and mismatch of two sequence elements and the case when the position is empty (denoted by the symbol $-$) we are able to calculate a global score: apply c to every two elements of S_1 and S_2 and sum them up. For the sequences $S_1 = AAGGCA$ and $S_2 = GACGA$ the maximal score is 2: if the empty symbol was at another position the score would either be equal or less:

```
S1     A  A  G  G  C  A
S2     G  A  C  G  -  A
Score  0  1  0  1 -1  1  = 2
```

In the evolutionary process subsequences of acid bases were moved, inserted or deleted. The current model only differentiates between a *match*, *mismatch* or *space*. For better accordance to the underlying biological model the influence of consecutive spaces (called *gaps*) should be larger. This desired property can be modelled by functions that give the first occurring gap a very low (typically negative) score and the immediately following gaps a score linear in their length. This class of functions is called *affine gap functions*: Let g_o be the initial gap penalty for the first gap and g_e the penalty for the n successive ones; the class of affine gap functions is then described by

$$g(n) = g_o + ng_e$$

With $g_o = -10$ and $g_e = -1$ the maximal possible score for the sequences S_1 and S_2 under an affine gap function is -7:

```
S1     A  A  G  G   C A
S2     G  A  C  G   - A
Score  0  1  0  1 -10 1  = -7
```

The global score is determined by examining both sequences in their entirety. However we are often interested in the comparison of local subsequences. Local subsequences often have more similarity and thus a much higher score: gaps which lower the overall score when whole sequences are considered can be excluded from the subsequence and therefore have no influence on the score. Since only local sections of the sequences are considered, this scoring is called *local scoring*. It is determined by scoring all subsequences of S_1 against all subsequences of S_2 and finding the maximal value. The local score for our example is maximal for the two subsequences `AGG` and `ACG` with a score of 2:

```
S1        |A  A  G|  G  C  A
S2     G  |A  C  G|  A
Score     |1  0  1|        = 2.
```

locally important subsequences

The next section demonstrates how the search for a global or local score, respectively, is expressed in terms of matrices, recursive dependencies between matrix elements, and simple functions.

5.2 Scoring as a stencil problem

To calculate the score of two sequences S_1 and S_2 by an affine gap function, we define a set of recurrence relations that work on matrices. Since the difference between global and local scoring is only in the initialization of border elements and the determination of the result, the calculation and therefore its parallelization is the same; we only present the formulas for global scoring.

Let V, G, E and F be matrices with the dimension $(length(S_1)+1) \times (length(S_2) + 1)$. The initial values for the matrices V, E and F are defined by

$$V(i,0) = E(i,0) = g_o + ig_e \quad (5)$$
$$V(0,j) = F(0,j) = g_o + jg_e \quad (6)$$

The matrix G does not need initialization values. The other values of the matrices are given by the recurrence relations

$$V(i,j) = max\{E(i,j), V(i,j), G(i,j)\} \quad (7)$$
$$G(i,j) = V(i-1,j-1) + \sigma(S_1(i), S_2(j)) \quad (8)$$
$$E(i,j) = max\{E(i,j-1), V(i,j-1) - g_o\} - g_e \quad (9)$$
$$F(i,j) = max\{F(i-1,j), V(i-1,j) - g_o\} - g_e \quad (10)$$

with σ as the score function between two bases and $S_k(n)$ as the element at position n for string k. The data dependencies are more complex than in typical stencil-based problems with only one stencil: G depends on values of V, both E and F on values of V and previously calculated ones respectively and V depends on elements from all other matrices.

An iterative implementation iterates over each point of (i, j), typically rowwise and calculates the values of the matrices such that implicit dependency relations are fulfilled. For example in the order $E(i,j), F(i,j), G(i,j)$ and finally $V(i,j)$. After each of the elements has been calculated the global score is the value in $V(length(S_2) + 1, length(S_1) + 1)$. To obtain the local score, V's maximal value has to be found.

To use the developed parallelization strategies for stencil-based problems, we need to transform the matrices initializations and the recurrence relations into a stencil-based problem: stencils are implicitly defined by the references to other elements in the recurrences, functions are defined directly through equations (7) to (10). The initial values are filled according to equations (5) and (6). Since one iteration is sufficient to calculate all values in V the convergence condition states that after one iteration the calculation should terminate.

5.3 Using PASTHA to score

We implemented a module `Pastha.Bio` which uses PASTHA to implement parallel versions of the scoring algorithms of biolib. We took great care to hide internal PASTHA details from the biolib-user by providing a similar interface as the existing functions do.

We now illustrate exemplary how to transform a call to perform local scoring from the biolib into a call to a function provided by `Pastha.Bio`. To sequentially calculate the local score in the biolib for two sequences `s1` and `s2` with a predefined scoring matrix `score` and gap penalty of -10 for the opening gap and -1 for each successive gap, we call `Bio.Alignment.AAlign.local_score`:

```
import Bio.Alignment.AAlign

calc =
  let score = local_score score (-10,-1) s1 s2
  in score
```

To gain the performance achievement of unboxed IO arrays and use explicit threading, we needed to lift the computations into the IO monad. As in the example above the local score of two sequences can be obtained by

```
import Pastha.Bio.AAlign

calc = do
  score <- local_score <bw> <bh>
                       sco (-10,-1) s1 s2
  return score
```

and is calculated in parallel. While PASTHA can determine the number of available cores using `numCapabilites`, the user needs to provide values for the block width and height (called `<bw>` and `<bh>` in the example; see Section 4.4).

Since all functions from `Bio.Alignment.AAlign` are implemented in `Pastha.Bio.AAlign` the adaption for parallel performance is reduced to lifting the necessary calculations into the IO monad.

5.4 Performance

To measure the speedup, we used the same machine and test setup as for the benchmarks described in Section 4.4. We tested both the sequential and parallel version with a randomly generated sequence over the alphabet $\{A, C, G, T\}$ for global and local scoring. Each test was ran for different lengths of sequences. We provide only the speedup for sequences of length 2000 since results did not differ a lot for other lengths. Shorter sequences were calculated too fast using PASTHA's implementation (leading to imprecision in the measurement), longer ones took too much time for the original sequential version, especially for local scoring.

Figure 10 shows the speedup for calculating the global score of two sequences. The original sequential implementation makes heavy use of lazy evaluation and lists to simulate arrays to save memory but both techniques lead to poor performance for bigger sequences because the garbage collector has to traverse through the arrays on each call. The superlinear speedup happens because we did not simply parallelize the sequential implementation but also developed a new, strict, and IO-based implementation that performs and scales a lot better than a non-parallelized pure version. Surprisingly we observe a drop in the speedup when PASTHA used more than four cores; when using six and seven cores the speedup raised again. We are currently investigating this unusual behaviour.

Figure 11 shows the speedup for local scoring. While global scoring only uses one specific element in the lower right corner of the data matrix and thus profits from lazy evaluation, local scoring needs to find the maximal value in the whole matrix, thus every element needs to be evaluated; this is computationally very expensive with pure arrays. PASTHA's implementation for both global and local scoring is nearly equivalent. Finding the maximum value in the data matrix is therefor negligible.

Summarizing the achieved results, we think that the speedup justifies the sacrifice of pureness in this case. It remains interesting to see if future work (see Section 7), especially pure parallelizations, compares to this explicit parallelization.

6. Related work

Since stencil-based calculations are one of the key patterns in scientific calculation[3] a lot of work has been done both to describe

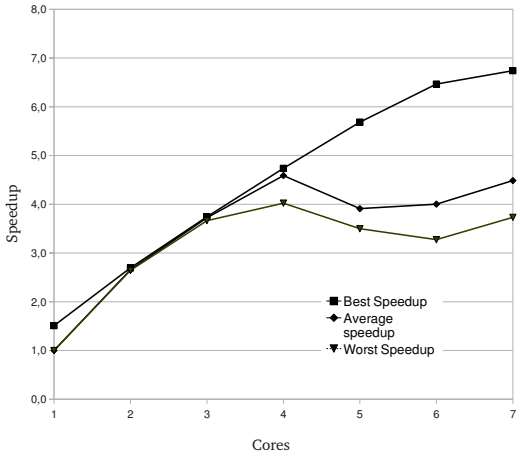

Figure 10. Maximal, average and minimal speedup for global scoring.

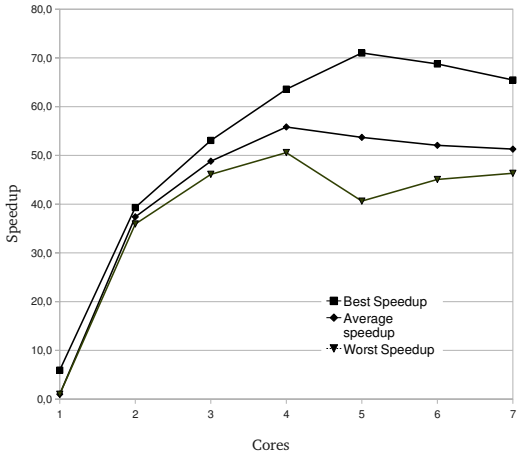

Figure 11. Maximal, average and minimal speedup for local scoring.

them conveniently and to improve their performance using parallelization. Traditional performance-demanding systems are written in C/C++[18] and Java[14] and parallelized using OpenMP for shared- and MPI for distributed-memory architectures. Distributed-memory systems are especially useful for large problem instances where thousands of cores can be utilized while multicore shared-memory systems are nowadays found in commodity desktop systems. For example, the C++-framework Janus [10, 11] provides a template-based system to express (among others mesh-like structures) stencil-based calculations and calculate them in parallel using MPI. It was later extended to support shared-memory parallelization using OpenMP[12].

The general idea of providing a high-level framework for parallel stencil calculations is an active research topic: [9] describes a parallel framework that uses different levels of parallelization which depend on the given stencil to provide a performant parallel calculation; [7] shows how a performant optimization strategy for a stencil can be determined automatically. Both ideas are very promising and interesting, especially with a purely declarative sten-

cil description where (semi)automatic analysis can guide the optimizations.

Another approach to parallel stencil calculation are very high-level languages, for example Chapel[5] and Fortress[2], which provide (quasi) implicit parallelization; [4] describes how to implement stencil calculations using seemingly sequential code. Since stencil calculations are easy to express in such high-level languages, this is an interesting alternative approach to a purely declarative description, although it limits the possibilities of stencil-dependent optimizations.

Parallelization of sequence scoring and alignment on shared-memory architectures using a block-based waveform pattern has already been described in [8], where a nearly linear speedup has been achieved. Unfortunately the authors do not make any statements to the question whether their approach could be generalized to allow the parallel computation of general stencil-based problems.

As far as we know, the idea of providing a general framework for parallel stencil calculation has not yet been transformed to modern functional languages.

7. Conclusion and future work

We have shown how to define general stencil-based problems in Haskell using a declarative description and developed an example using voltage diffusion as a basis. The definition of sequence scoring algorithms from the biolib as stencil-based problems indicated that our approach is sound. Not only can it be used to solve real-world problems but also it delivers a superlinear speedup of up to 4.8 for four cores in the biolib example. This speedup was partially achieved by lifting the calculations into the IO monad and using unboxed arrays.

We developed the prototype for a library to test, implement, and evaluate parallelization strategies for stencil-based problems. By implementing a taskqueue-based parallelization strategy, we demonstrated that the approach of describing stencils on a more general basis and calculating them in parallel is working in a functional context and leads to speedups of up to 4 for six cores.

The prototype implementation can already be used to parallelize common 2D-based stencils. Nevertheless we still see room for improvement, for example in the aspects of advanced automatisms, adaption to specific stencils, and evaluation of other parallelization techniques from Haskell, to mention some. We plan to

- Evaluate both semi-implicit parallelism, data parallel Haskell and other alternatives, e.g. GPUs, in terms of performance and usability for stencil problems.
- Generate the dependency functions for the Taskqueue automatically by analyzing the stencils.
- Implement specialized parallelization strategies with limited general applicability but better parallel performance for specific classes of stencils and apply them automatically.
- Extend PASTHA to heuristically determine a block size that achieves good speedup.

Moreover it remains interesting to investigate whether our definition and implementation can be applied to other existing and new problems. For example for already implemented algorithms in packages on Hackage which have not yet been parallelized.

Acknowledgments

I am grateful to Claudia Fohry for comments, insightful discussions and hints, Katja Bianchy for proof reading and Jens Breitbart for initial comments. I thank the anonymous reviewers for their helpful feedback.

References

[1] bio: a bioinformatics library for haskell. `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/bio`.

[2] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification. 2007. URL `http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf`.

[3] Krste Asanovic, Ras Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John D. Kubiatowicz, Edward A. Lee, Nelson Morgan, George Necula, David A. Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine A. Yelick. The parallel computing laboratory at u.c. berkeley: A research agenda based on the berkeley view. Technical Report UCB/EECS-2008-23, EECS Department, University of California, Berkeley, Mar 2008. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-23.html`.

[4] R.F. Barret, P. C. Roth, and S. W. Poole. Finite difference stencils implemented using chapel. Technical Report TM-2007/119, 2007.

[5] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21 (3):291–312, 2007. ISSN 1094-3420. doi: http://dx.doi.org/10.1177/1094342007078442.

[6] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. ISSN 1070-9924. doi: 10.1109/99.660313. URL `http://dx.doi.org/10.1109/99.660313`.

[7] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. pages 1–12, 2008.

[8] Zhihua Du, Zhen Ji, and Feng Lin. Parallel computing for optimal genomic sequence alignment. pages 532–535, 2006.

[9] Hikmet Dursun, Ken-Ichi Nomura, Liu Peng, Richard Seymour, Weiqiang Wang, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. A multilevel parallelization framework for high-order stencil computations. pages 642–653, 2009. doi: http://dx.doi.org/10.1007/978-3-642-03869-3_61.

[10] Jens Gerlach and Mitsuhisa Sato. Generic programming for parallel mesh problems. In *ISCOPE*, pages 108–119, 1999.

[11] Jens Gerlach, Peter Gottschling, and Uwe Der. A generic c++ framework for parallel mesh-based scientific applications. In *HIPS*, pages 45–54, 2001.

[12] Jens Gerlach, Zheng-Yu Jiang, and Hans Werner Pohl. Integrating openmp into janus. pages 101–114, 2001.

[13] Don Jones, Simon Marlow, and Satnam Singh. Parallel performance tuning for haskell. In *Haskell '09: Proceedings of the second ACM SIGPLAN symposium on Haskell*. ACM, 2009. URL `http://www.haskell.org/~simonmar/papers/threadscope.pdf`.

[14] John F. Karpovich, Matthew Judd, W. Timothy Strayer, and Andrew S. Grimshaw. A parallel object-oriented framework for stencil algorithms. In *HPDC*, pages 34–41, 1993.

[15] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. pages 65–78, 2009. doi: http://doi.acm.org/10.1145/1596550.1596563.

[16] Gregory F. Pfister. *In search of clusters: the coming battle in lowly parallel computing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998. ISBN 0-13-437625-0.

[17] Marc Snir and Steve Otto. Mpi-the complete reference: The mpi core, 1998.

[18] Gregory V. Wilson. *Parallel Programming Using C++*. MIT Press, Cambridge, MA, USA, 1996. ISBN 0262731185. Foreword By-Stroustrup, Bjane.