Performance Engineering for ∞SCITE

Dominik Ernst, Peter Georg, Katharina Jahn, Tilo Wettig

November 9, 2020

1 Introduction

The recent emergence of high-throughput single-cell sequencing technologies has led to an unprecedented data explosion in the field of computational oncology. This development in combination with the high noise profiles inherent to single-cell measurements led to a number of computationally challenging problems. One such problem is the inference of mutation histories of tumours from high-throughput single-cell DNA sequencing data. A tumour develops through uncontrolled cell divisions that are first enabled and later exacerbated by a continuous acquisition of genetic alterations. Since these mutations happen at the level of individual cells and are passed on to descendent cells, this process creates a complex pattern of genetically distinct yet related cell populations referred to as clones and subclones. This genetic diversity poses a major challenge to targeted cancer therapies which often fail to eradicate the tumour in its entirety leaving minute resistant subclones that regrow the tumour and eventually lead to patient death. Knowing the mutational history and clonal composition of a tumour allows one in principle to adapt the therapy according to the clonal composition and improves the chances of a successful outcome. Due to the high noise level found in single-cell mutation calls, most methods employ a probabilistic search scheme to characterise not only the most likely mutational history of a tumour but also the uncertainty in the inference. Unfortunately for larger datasets comprising the mutational information of 10,000s of cells, the runtimes of these methods can become prohibitive. Here, we study a specific algorithm named SCITE [1] and its successor ∞SCITE[2] to show how high-performance computing can be used to enhance performance in the presence of large datasets.

2 Overview of the algorithm

SCITE and ∞SCITE are probabilistic search schemes that infer a mutation history from a dataset of noisy single-cell mutations calls. Fig. 1 gives a high-level overview.

2.1 Mutation trees

A mutation history is represented as a *mutation tree* T, a rooted tree whose non-root nodes are labelled by the mutations observed in a tumour. This tree describes a partial temporal order of mutations in the sense that for any two mutations located on the same path from the root to a leaf, the mutation encountered first predates the second. Two mutations located in different branches have occurred in different cell populations, and their temporal order is unknown.

While each node in the mutation tree represents a mutation, it also represents a (potential) tumour subclone that has acquired all mutations on the path from the root to this node (inclusive). Cells sampled from a tumour have a mutational profile consistent with one of the subclones represented in the tumour's mutation tree. Therefore, a mutation tree can be augmented by attaching the cells to the nodes representing their respective subclone of origin.

2.2 Model

The input data D is a binary matrix with missing entries called the *mutation matrix* that encodes the single-cell mutation calls. Each column of the matrix provides the observed mutational state of a single

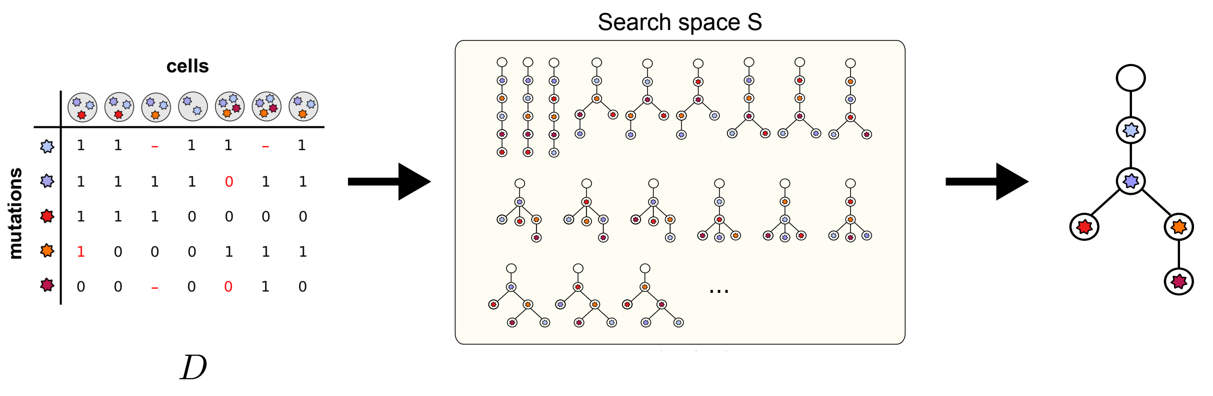

Figure 1: Overview of the SCITE search scheme. Left: The input data D is a binary matrix with missing entries ('-'). Red text colour highlights missing observations, as well as false positive and false negative observations. The locations of the latter two are unknown in real data. Centre: The search space consists of all mutation trees whose non-root nodes are labelled with the observed mutations. Right: Maximum Likelihood or maximum aposteriori tree obtained from the MCMC-based search using a probabilistic error model. Due to noise in the observed data, typically no mutation tree exists that fully represents the observed genotypes.

cell. A 1/0 entry represents the observation of a mutated/normal state at the respective mutation site (row). A '-' encodes a missing observation.

2.2.1 Sequencing errors

In practice, single-cell mutation calls can differ from a cell's true mutation state. There are two types of observational errors: If the true mutation value is 0, we may observe a 1 with some probability α (false positive), and if the true mutation value is 1, we may observe a 0 with probability β (false negative). Let D_{ij} be the observed state of cell j at site i and E_{ij} be the true state of cell j at site i. Then we have the following likelihoods:

$$P(D_{ij} = 0|E_{ij} = 0) = 1 - \alpha \qquad P(D_{ij} = 1|E_{ij} = 0) = \alpha$$
$$P(D_{ij} = 0|E_{ij} = 1) = \beta \qquad P(D_{ij} = 1|E_{ij} = 1) = 1 - \beta$$

In the following, we denote $\boldsymbol{\theta} = (\alpha, \beta)$.

2.2.2 Model likelihood and posterior

In [1] it has been shown that the model likelihood $P(D|T, \boldsymbol{\theta})$ can be computed efficiently under the assumption that sequencing errors are independent of each other:

$$P(D|T, \boldsymbol{\theta}) = \prod_{j=1}^{m} \sum_{v \in T} \prod_{i=1}^{n} P(D_{ij}|T, v, \boldsymbol{\theta}) P(v|T, \boldsymbol{\theta})$$

The outer product computes likelihoods separately for each sample. The sum over all attachment points v of the sample to the mutation tree T effectively marginalises out the sample placement. Finally, the inner product computes the likelihood of placing a sample m at a node v in the mutation tree. The posterior probability is then

$$P(T, \boldsymbol{\theta}|D) \propto P(D|T, \boldsymbol{\theta}) P(T, \boldsymbol{\theta})$$

2.2.3 The doublet model of ∞SCITE

Due to technical limitations in single-cell processing, a part of the samples will not be single cells but two (or even more) cells. The mutation profiles of such multi-cell samples will be a mixture of the mutation profiles of the involved cells with sequencing errors on top. In contrast to SCITE, the ∞SCITE algorithm uses a weighted mixture model to account for the possibility of doublets:

$$P(D|T,\boldsymbol{\theta}) = \prod_{j=1}^{m}((1-\delta)P_{sc}(D|T,\boldsymbol{\theta}) + \delta P_d(D|T,\boldsymbol{\theta}))$$

Here, $P_{sc}(D|T,\boldsymbol{\theta})$ represents the single-cell based likelihood and $P_d(D|T,\boldsymbol{\theta})$ represents the doublet based likelihood. The parameter δ stands for the doublet probability. It can be either fixed or learned from the data. The time complexity of calculating the doublet part of the likelihood is in $O(mn^2)$, as we have to account for all pairings of cell attachments to the tree nodes. More details are given in [2].

2.3 Algorithm overview

SCITE and its successor ∞SCITE comprise a Markov chain Monte Carlo (MCMC) sampling scheme to estimate the joint posterior probability distribution of the mutation tree space and the sequencing error rates $\boldsymbol{\theta} = (\alpha, \beta)$. A high-level overview of the MCMC scheme is given in Algorithm 1.

Algorithm 1 SCITE MCMC Outline

```
1:  procedure MCMC($D, reps, its$)                ▷ $D$ is an $m \times n$ matrix ($m$ samples, $n$ mutations)
2:      $S \leftarrow \emptyset$
3:      for $r \leftarrow 1, ..., reps$ do                                    ▷ Number of MCMC restarts
4:          $curr \leftarrow (T, \theta)$                                          ▷ Sample initial state
5:          for $i \leftarrow, ..., its$ do                                     ▷ Length of MCMC chain
6:              $prop \leftarrow (T', \theta')$          ▷ sample new state $(T', \theta') = (T', \theta)$ or $(T', \theta') = (T, \theta')$
7:              $propScore \leftarrow score(prop, D)$              ▷ computation of state score in $O(nm)$
8:              if accept $prop$ then                  ▷ Accept better score, or worse score with low prob.
9:                  $curr \leftarrow prop$
10:             if sample step then                     ▷ Sample in constant intervals after burn-in phase
11:                 $S \leftarrow S \cup curr$
12:     return $S$
```

Details on the MCMC moves, the scoring function and the acceptance probability can be found in [1]. The codes of SCITE [3] and ∞SCITE [4] are available on github.

2.4 Code parallelisability

Apart from trivial parallelisation by running multiple MCMC chains in parallel, the scoring function called in each MCMC step for the proposed state lends itself to parallelisation, as partial scores are computed separately for each sample. This holds independently of the chosen sample model, i.e., whether samples are expected to be single cells [1] or doublet samples are considered [2]. High-level representations of the scoring functions are given in Algorithm 2 and Algorithm 3). With the doublet scoring being more expensive than the singlet scoring (quadratic time with respect to the tree size vs linear), we can expect the parallelisation to be particularly useful in the setting of ∞SCITE.

3 Single-thread optimizations

3.1 2D matrix data structure

Profiling results (using emulating profiler `callgrind`) show a large runtime percentage spent in the memory management functions `free` (35%) and `new` (30%), stemming mostly from a matrix data structure in the function `scoreTreeAccurate` computation.

The matrix data structure consists of an `int**` pointer to an array of row pointers, which each point to the first element in a row, which are laid out in a contiguous, linearized buffer. This necessitates a separate allocation for the row pointers and an indirection on access.

Algorithm 2 Outline of scoring function (SCITE)

```
1: procedure SCORE($(T', \theta'), D$)          ▷ $D$ is an $m \times n$ matrix ($m$ samples, $n$ mutations)
2:     $propScore \leftarrow 0.0$               ▷ log score of proposed state $(T', \theta')$
3:     for $s \leftarrow 1, ..., m$ do          ▷ iterate over all samples
4:         $sScore \leftarrow 0.0$              ▷ log score per sample
5:         for $v \leftarrow 1, ..., n+1$ do    ▷ iterate over all nodes in $T'$
6:             $sScore \leftarrow sScore + aScore(s, v, D, T', \theta')$    ▷ aScore() computation in O(1)
7:         $propScore \leftarrow propScore + log(sScore)$
8:     return $propScore$
```

Algorithm 3 Outline of scoring function (∞SCITE)

```
1: procedure DOUBLETSCORE($(T', \theta'), D$)    ▷ $D$ is an $m \times n$ matrix ($m$ samples, $n$ mutations)
2:     $propScore \leftarrow 0.0$                ▷ log score of proposed state $(T', \theta')$
3:     for $s \leftarrow 1, ..., m$ do           ▷ iterate over all samples
4:         $sScore \leftarrow 0.0$               ▷ log score per sample
5:         $scScore \leftarrow singletScore(s, D, (T', \theta'))$    ▷ singletScore() computation in $O(n)$
6:         $doubScore \leftarrow doubletScore(s, D, (T', \theta'))$  ▷ doubletScore() computation in $O(n^2)$
7:         $sScore \leftarrow \log((1-\delta) \cdot scScore + \delta \cdot doubScore)$    ▷ score weighted by doublet rate $\delta$
8:         $propScore \leftarrow propScore + sScore$
9:     return $propScore$
```

Instead, this data structure can be realized with a nested `array`

```
std::array<std::array<T, M>, N> data;
```

The operator `matrix[n]` returns another `std::array`, which can again be indexed with square brackets, making the syntax `matrix[n][m]` possible. The square bracket operators will be inlined by the compiler and work with the constants M and N for efficient index computation without memory indirection. Most importantly, this data structure requires no dynamic allocation and therefore resides on the stack.

So far, this data structure is used for the integer attachment matrices in the `scoreTreeAccurate` function, but can be used in other places as well, for example the double precision `logScores` matrices. It has minimal impact on the written code, as the `matrix[i][n]` syntax still works. It also allows one to use a simple assignment instead of a deep copy function, and eliminates the need for deallocation and tracking of ownership, which is a big risk for memory leaks.

Consequently, using the new matrix data structure in the accurate-only scenario, the time taken up by the de/allocation function drops from 64% to 3%. The runtime in this particular scenario (measured at $l = 2000$) drops from 32.5s to 5.1s.

3.2 Hotspot: `getAttachmentScoresFast`

Consider the code

```
for(int i=1; i<=n; i++) {
        int node = bft[i];
        attachmentScore[node] = attachmentScore[parent[node]];
        attachmentScore[node] -= logScores[dataVector[node]][0];
        attachmentScore[node] += logScores[dataVector[node]][1];
}
```

Listing 1: `getAttachmentScoresFast`, *scoreTree.cpp*, original code

The proposed optimization precomputes the subtraction of the two elements in the 4×2 `logScores` vector to a 4 element array. This not only saves a subtraction, but also loads only one element instead of two, and reads from a simple array instead of a row-pointer matrix structure. By assigning the complete expression to `attachmentScore[node]` in one go, instead of three stores, which the compiler cannot eliminate due to aliasing, only one store is generated.

```
double summedLogScores[4];
for (int i = 0; i < 4; i++) {
        summedLogScores[i] = logScores[i][1] - logScores[i][0];
}

for (int i = 1; i <= n; i++) {
        int node = bft[i];
        attachmentScore[node]
            = attachmentScore[parent[node]]
              + summedLogScores[dataVector[node]];
}
```

Listing 2: `getAttachmentScoresFast`, *scoreTree.cpp*, optimized code

The optimized code speeds up this loop from 5.5 to 3.3 cycles per loop iteration.

3.3 Hotspot: `rootAttachmentScore` (28% of runtime)

The code in listing 3 calculates a sum over indexed values from the 4×2 matrix `logScores`. `dataVector` is an integer array containing values $\in [0, 3]$.

```
for (int gene=0; gene<n; gene++){
        score += logScores[dataVector[gene]][0];
}
```

Listing 3: `rootAttachmentScore`, *scoreTree.cpp*, original code

Unrolling by a factor $2\times$ and splitting the sums double the throughput from 4 to 2 cycles per loop iteration on the Skylake architecture.

3.4 Hotspot: `getMaxEntry` (18% of runtime)

This functions simply finds the maximum entry of a double array.

Since this is a function with a well-defined algorithm that is unlikely to be changed in the future, much larger changes to the code are possible. This particular function is also very suitable for vectorization with the `vmaxpd` instruction, which computes `max(a,b)` for four values at once. This instruction has a latency of 4 cycles and a throughput of 2 per cycle, which would require $8\times$ unrolling on top of the $4\times$ unrolling for vectorization. However, this is totally out of proportion to the loop trip count of around 100, and any speedup would be swallowed by the loop epilogue that combines the unrolled values. Instead, an unroll factor of only 2 on top of the vectorization is chosen. This results in a throughput of only a quarter of the possible maximum, but represents the fastest compromise for the short loop.

3.5 Vector allocations

The frequent creation/destruction and copying of standard vectors leads to a large number of C library function calls `malloc`, `free`, `memset` and `memmove`. This is due to the interaction of `getDoubletAttachmentScoreMatrixFast` and `getSampleDoubletScoreFast`. The former creates, fills and returns a new vector of vectors containing doublet scores, which is then passed by value to the latter. This involves first allocation and initialization of the vectors in the creating function, and then allocation and copy in the receiving function, followed by destruction for both copies.

Changing the signature of the receiving function from

```
double getSampleDoubletScoreFast(
    vector<vector<double>> doubletAttachmentScore...)
```

to

```
double getSampleDoubletScoreFast(
    vector<vector<double>> const & doubletAttachmentScore...)
```

by making the first parameter a const reference eliminates the copy, without changing the function signature at the call site.

The described function sequence happens inside a loop. Preallocating the vectors outside the loop and reusing the same vector amortizes the vector construction over the loop iterations. The creating function's signature is changed from

```
void getDoubletAttachmentScoreMatrixFast
        (...)
```

by adding a reference to the vectors it should use as output:

```
void getDoubletAttachmentScoreMatrixFast
        (vector<vector<double>>& doubletAttachmentScore, ...)
```

The runtime contribution of the mentioned functions is reduced to a negligible amount.

3.6 Exponential functions

When doublets are enabled, profiling with `perf` identifies the `exp` function from the math library as the largest contributor with 60-70%. The `exp` function appears in the function `getSampleDoubletScoreFast`, where all scores of a sample are summed up as

```
for score in doubletScores:
    sum += exp( score - maxScore );
doubletScoreSum = log(sum) + maxScore
```

GCC does not vectorize this function without the relaxation of floating point semantics with the compiler option `-Ofast`. This could, however, have effects elsewhere in the code base.

Since the maximum value of all scores is subtracted from all scores before taking the exponential, the maximum exponent is a zero, and all other values are negative. The whole sum is therefore guaranteed to be larger than 1, from which it follows that the resulting logarithm reduces the magnitude of errors made in the computation of the exponential. A less accurate but faster exponential function might be good enough.

A very simple approximation described by Schraudolph [5] to compute e^x exploits the properties of floating-point formats of the form $M \cdot 2^E$ with mantissa M and exponent E. The natural logarithm can be converted to a dual logarithm by multiplying with $log2(e)$. If the exponent x is split into an integer and a fractional part $i + f$, the desired computation is 2^{i+f}. By left shifting the higher word of i by 20 bits and reinterpreting the result as a double, the integer part is shifted into the mantissa, which results in 2^i. This crude approximation can be improved by replacing the integer shift with a floating point multiplication by 2^{53}, which acts as a sort of linear interpolation in the mantissa.

Including the bias in the exponent, this can be realized with a multiplication, an add and an FP to int conversion:

```
double fastExp_schraudolph(double x) {
   int i = (1048576 / M_LN2) * x + 1072693248;

   double result = 0;
   reinterpret_cast<int*>(&result)[1] = i;

   return result;
}
```

A better approximation, described on stackoverflow [6], also splits x into $i + f$, converts i to integer and shifts it into the mantissa. An polynomial $P, [0,1] \to [1,2[$ that approximates 2^x is used to compute the mantissa from the fractional part. The full calculation is therefore $2^x = 2^{i+f} = 2^f \cdot 2^i \approx p(f) \cdot 2^i$, which again corresponds to the floating point format $M \cdot 2^E$ with i as the exponent E and $p(f)$ as the mantissa M.

The order of the polynomial p can be chosen to trade off between precision and performance. The tool `lolremez` [7] uses the Remez algorithm to compute polynomial coefficients that minimize the maximum relative error of the approximation in the range $[0,1[$. The command

```
lolremez -d 5 -r 0:1 "2^x" "2^x"
```

computes coefficients for a polynomial of order 5 in the range $[0,1[$ that approximates 2^x with relative weighting.

The described method has been extended to double precision in AVX intrinsics for orders 1 to 5.

Table 1 shows the results for different exponential function computation methods and polynomial orders. The impact of errors made in the computation of a single exponential function is consistently about two orders of magnitude lower in the final result. The simple Schraudolph approximation has low and inconsistent accuracy, which also shows up in the comparably bad total error. Each additional polynomial degree decreases both errors by one or two orders of magnitude. Note that for all versions, the same optimal tree is computed for the test case.

Each increase in the order of the approximating polynomial adds another FMA operation. The measured results show an increase by 0.4-0.8 cycles per full AVX vector with each additional polynomial degree, which fits the 2 FMA per cycle processor capability. Due to the small benefit in the total runtime of the lower degree approximations (order 5 is about 10% slower than order 3), a polynomial degree of at least 5 is probably the best option. At degree 5, the total runtime decreases from $164s$ to $29.6s$.

asd	maximum relative error (empirical)	relative error tree score	performance (AVX) cycles/element
libm (GCC)	0	0	12
libm-vec (GCC)	0	0	4
Schraudolph	10^{-2}	10^{-3}	
Order 1	10^{-2}	10^{-4}	1.5
Order 2	10^{-3}	10^{-5}	1.6
Order 3	10^{-5}	10^{-7}	1.8
Order 4	10^{-6}	10^{-8}	1.9
Order 5	10^{-8}	10^{-10}	2.1

Table 1: Maximum error for each individual function call compared to libm version, relative error of the tree score, and performance in cycles per element for different exponential function calculation methods (GCC 8.3, Coffelake CPU).

The sampling profiler `perf` still shows a 60% runtime contribution of `getSampleDoubletScoreFast`, which computes the exponential sums, of which about 70% can still be attributed to the exponential functions. The exponential function throughput is therefore still a significant runtime contributor that would reemerge as a limiter if other parts were optimized further.

4 Parallelization

In the previous section we have optimized the single-threaded performance of the code. However, most modern computer systems have multiple cores, some even multiple CPUs per system. In case of high-perfomance computing one tends to use multiple computers (or nodes) in parallel to further decrease the total runtime. First, it is crucial to add parallelization running on one node. At a later stage we could also use multiple nodes to further reduce the runtime as required. In fact vectorization, a concept

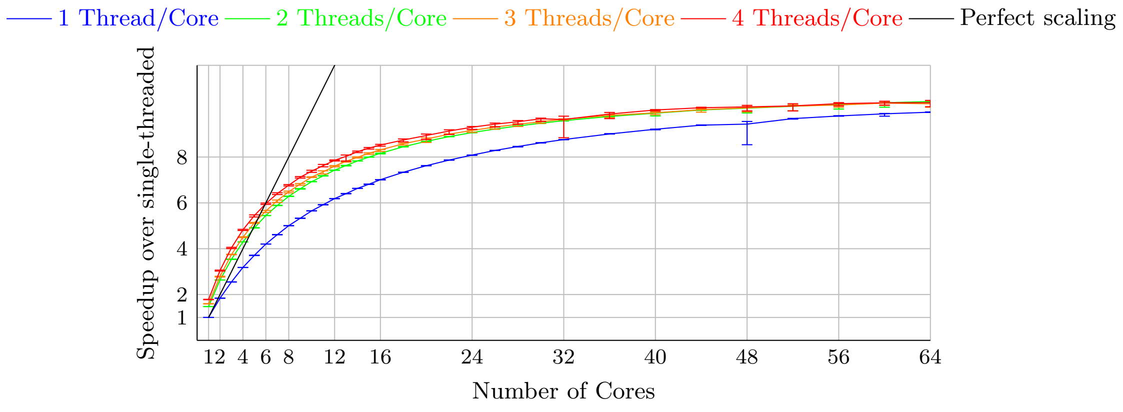

Figure 2: Strong-scaling benchmark running on an Intel Xeon Phi 7230 with up to 64 cores and 256 threads.

introduced in the single-thread performance section, is a form of parallelization as well, but within one core.

In general any form of parallelization adds an overhead, hence for the parallelization to be effective the overhead should be low compared to the total runtime of the parallelized code. In addition we note that most, or even any, code includes a point of serialization, i.e., not all parts can be parallelized.

4.1 On-node parallelization

For on-node parallelization, multi-threading using OpenMP is a popular choice. For OpenMP it is crucial to find a loop that can be parallelized, i.e., loop iterations can be worked on independently, and that contains most of the total runtime. This way the overhead introduced by spawning threads and joining them once work is done is low if each thread can work independently for a relatively long time.

In our case all functions described in the single-thread optimization are called in the score function. The score function consists of two parts: (1) a loop to calculate the score for each sample individually and (2) a loop to reduce the scores of all samples to one value. Hence this function is a very good candidate for multi-threading using OpenMP. The first loop can be easily parallelized using an OpenMP `parallel for` statement. The second loop can be parallelized using an OpenMP `parallel for reduction` clause. The second loop obviously does not allow for linear scaling with the number of threads since it contains reductions. In this case a lower bound for scaling is given by log_2(number of threads).

To check our multi-threading implementation we ran a representative data set ($m = 50000$ samples, $n = 100$ mutations, 5000 doublets, 0.1% false positives, 15% false negatives) on an Intel Xeon Phi 7230 and an AMD Epyc 7502P.

First, we discuss the results obtained running on Intel Xeon Phi 7230, see Fig. 2. The CPU has a total amount of 64 cores, with each core running up to 4 threads (SMT) in parallel. Note that the Intel Xeon Phi 7230 supports AVX-512, but our code is currently limited to AVX2. Hence the single-threaded performance on this machine is not optimal, but this has no crucial effect on the parallel efficiency. We can see that adding multi-threading only to run with more than one thread on a single core can already improve the performance by a factor of almost 1.8 running with 4 threads on one core. The scaling over cores looks good up to 4-6 cores. Running this particular data set on more than 8 cores is not beneficial. The total runtime is barely reduced adding further cores to the computation.

Running the same benchmark on an AMD Epyc 7502P with 32 cores and up to 2 threads (SMT) per core, see Fig. 3, we see very similar results. Again the scaling looks good up to 4 cores, but running on more than 8 cores is not beneficial. Compared to the results obtained on the Intel Xeon Phi, running with more than 1 thread per core has a lower impact on the AMD Epyc.

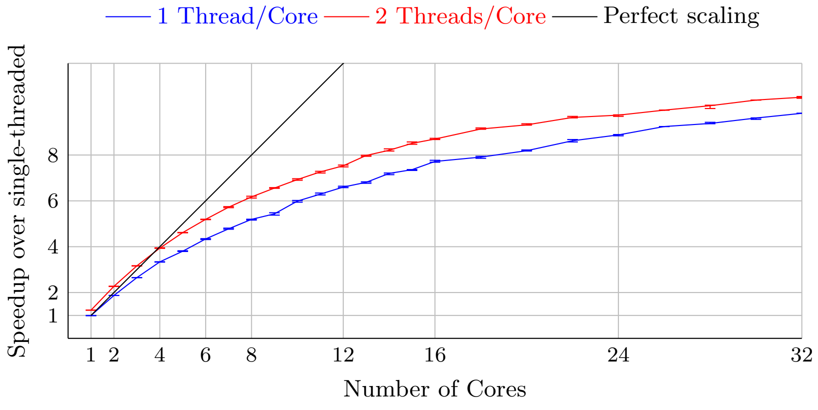

Figure 3: Strong-scaling benchmark running on an AMD Epyc 7501P with up to 32 cores and 64 threads.

4.2 Multi-node parallelization

Multi-node parallelization using multi-processing, e.g., by means of MPI, is a complex task compared to simple node-level parallelization using multi-threading. Compared to OpenMP multi-threading we need to manage data transfer explicitly, i.e., whenever one process changes data, the data have to be sent to all other processes before these can work on the data again. In addition this communication is slow compared to communication within a node using multi-threading, i.e., it adds a relevant communication overhead. For the multi-node parallelization to be efficient, this overhead needs to be low compared to the total runtime. This is possible when data only rarely needs to be shared with other nodes. In case of our application, the loop to calculate the independent scores for each sample would be a candidate for computation to be distributed over many nodes. However for the data we tested with, the total runtime of this loop is already low, hence the possible benefit of multi-processing is low as well. The added overhead will very likely lead to bad parallel efficiency.

The second loop reducing the scores of each sample to one score is also only efficient if the chunk of the reduction to be calculated by each process is large enough to hide the added communication costs.

Overall it currently is not required and very likely unprofitable to add multi-node parallelization. At a later stage, when the size of the data set is further increased, e.g., with an increased number of samples and mutations (in particular, single-cell high-throughput whole-genome sequencing), this will change and a combination of multi-node and on-node parallelization will be necessary to further reduce the total runtime.

4.3 Task parallelization

As already shown, both on-node and and multi-node parallelization impose an overhead. Hence we can never achieve perfect scaling. In practice, applications are often run several times with different input data or parameters. In this case we can simply run multiple instances, i.e., tasks, of the same application in parallel. In theory this should allow for perfect scaling. However, in case of running many tasks on a single node, many compute resources are shared. This may lead to a negative effect on the runtime of a single task. In addition, the number of tasks we can run in parallel might be limited. To minimize the total runtime we need to find the sweet spot for the combination of task parallelization and multi-threading.

Table 2 shows the speedup for various combinations of number of tasks per node and cores per node running on the Intel Xeon Phi. We can see that runnning as many tasks in parallel as possible is the most efficient. Up to 32 tasks there is no negative effect of running in parallel. Only when running 64 tasks in parallel we see a degration of efficiency. In practice running 64, 32, or 16 tasks (with 1, 2, or 4 cores each respectively) in parallel on a single Intel Xeon Phi seem to be the best choices.

Table 3 shows the speedup for various combinations of number of tasks per node and cores/threads per node running on the AMD Epyc 7502P. Up to 8 tasks there is no negative effect of running in parallel.

Cores: / Tasks:	1	2	4	8	16	32	64
1	1.0	1.7	2.7	3.8	4.7	5.4	5.8
2		2.0	3.4	5.4	7.6	9.4	10.8
4			4.0	6.8	10.7	15.1	18.8
8				8.0	13.5	21.3	29.9
16					16.0	26.9	42.4
32						32.0	53.4
64							61.3

Table 2: Speedup of running identical parallel tasks on the Intel Xeon Phi (with 4 threads per core) compared to running all tasks in serial on a single core.

Running with more than 8 tasks there is no benefit to be seen anymore. This is a very different result compared to what we obtained on the Intel Xeon Phi that can be explained by the different architectures: The AMD Epyc 7502P consists of 8 groups of 4 cores with a private L3 cache. Once we run with more than one task in such a group, the L3 cache is shared among all processes in this group. In practice running 8 tasks with 8 threads each in parallel on a single AMD Epyc 7502P seems to be the best choice.

Cores	1		2		4		8		16		32	
Threads / Tasks	1	2	2	4	4	8	8	16	16	32	32	64
1	1.0	1.2	1.9	2.3	3.3	3.9	5.2	6.2	7.7	8.7	9.8	10.4
2			2.0	2.5	3.7	4.5	6.6	7.8	10.3	12.3	14.8	16.4
4					4.0	4.9	7.4	9.0	13.3	15.7	19.8	22.8
8							8.0	9.7	15.0	18.1	24.9	28.2
16									8.0	9.9	15.0	18.2
32											8.0	9.9

Table 3: Speedup of running identical parallel tasks on the AMD Epyc 7502P compared to running all tasks in serial on a single core with a single thread.

5 Conclusions

Numerical studies of tumor progression models are faced with rapidly increasing amounts of data due to advances in single-cell sequencing technologies and hence unacceptable runtimes. We have considered a pertinent algorithm in the field, ∞SCITE, and applied performance-engineering techniques to speed up its performance for large datasets. We first identified hotspots that dominate the runtime and then applied a number of optimization techniques to these hotspots: data-structure optimization, elimination of unnecessary loads and stores, loop unrolling, vectorization, improved memory management, and approximation of the exponential function. The combination of these improvements led to an overall speedup of $4\times$ to $5\times$. We also parallelized the code over cores using OpenMP and investigated the scaling behavior. Given the nature of the problem, it is also possible to run independent tasks on different cores. The optimal choice of how many tasks should be run on how many cores turns out to depend on the hardware architecture. In the cases we benchmarked, the optimal choice yields nearly perfect speedup. We did not yet implement MPI parallelization over nodes since the nature of the problem allows for trivial parallelization and since, at the present time, individual tasks are small enough in terms of memory footprint and runtime.

References

[1] K. Jahn, J. Kuipers and N. Beerenwinkel, *Tree inference for single-cell data*, *Genome Biology* **17** (2016) 1.

[2] J. Kuipers, K. Jahn, B. J. Raphael and N. Beerenwinkel, *Single-cell sequencing data reveal widespread recurrence and loss of mutational hits in the life histories of tumors*, *Genome Research* **27** (2017) 1885.

[3] https://github.com/cbg-ethz/SCITE.

[4] https://github.com/cbg-ethz/infSCITE.

[5] N. N. Schraudolph, *A Fast, Compact Approximation of the Exponential Function*, *Neural Computation* **11** (1999) 853.

[6] https://stackoverflow.com/questions/47025373/fastest-implementation-of-exponential-function-using-sse.

[7] https://github.com/samhocevar/lolremez.