Real-time Methods in Reversible Computation

Tommi Pesu and Iain Phillips

Imperial College London

Abstract. Bennett has shown how to simulate arbitrary forwards-only computations by fully reversible computation. In particular he has given a space-efficient linear time simulation. After describing a different linear-time reversible simulation with improved space efficiency, we initiate the study of real-time simulations. In addition to being linear-time, these must offer continuous progress, meaning that the delay between successive forward events must be bounded by a constant.

1 Introduction

Intel's co-founder Gordon E. Moore famously predicted in 1965 that the computational performance of modern computers would double every 18 months (Moore's Law). At the moment his law is being obeyed, due to continuous development in the area of minimising elements which make up the computer. As pointed out in [3], a linear increase in clock frequency is associated with a quadratic increase of elementary gates per unit area, leading to a cubic increase in heat dissipation if the energy expended per event remains constant. Thus the increase under Moore's Law has only been possible due to a vast increase in energy efficiency of elementary logical gates.

However there exists a physical limit of $kT \ln 2$ which is about 3×10^{-21} Joule at room temperature. This is the minimum amount of energy that a computer must waste to perform a calculation. With current advances following Moore's Law, this limit will be reached in about ten years [9]. Therefore in the near future something drastic will need to be done for computation power to be able to increase at the pace defined by Moore's Law.

A possible solution to this problem was suggested by Landauer [4], who argued that the thermodynamic limit of $kT \ln 2$ only applies to calculation performed in a irreversible way. Therefore if the calculation is performed in a reversible way then the cost of a calculation operation can be below the limit $kT \ln 2$ given by thermal noise.

Lecerf and Bennett continued this line of thought, proving independently that an irreversible Turing Machine can be simulated by a reversible Turing machine [6, 1].

The next development in reversible computation was more space efficient reversible simulation. Bennett [2] showed how to obtain a more space optimised version of reversible computation. Li and Vitányi [8] also looked at trade-offs between space and time, and trade-offs between space and irreversible erasure. Lange, McKenzie and Tapp [5] gave a method to perform reversible simulation

in linear space; however it comes at the cost of exponential time. Williams [10] generalised the results of [2, 5]. Buhrman, Tromp and Vitányi proved an upper bound on the trade-off between time and space, and showed that one can simultaneously achieve sub-exponential time and sub-quadratic space [3].

In this paper we are interested in real-time reversible simulations. Such simulations must in particular be linear-time, but we identify a further stronger property they should satisfy, which we call *continuous progress*. This means that the simulation of each forward step should not be indefinitely delayed. More precisely, there is a fixed finite bound p, independent of the time taken by the original forwards-only computation, such that when p steps of the simulation are performed, at least one step of advancement is made with respect to the original forwards-only computation. As far as we are aware, real-time reversible simulations have not been studied previously.

If interactive systems or systems that need to stream out data at constant intervals are ever implemented in practice with reversible computation, then it is critical that the algorithm that performs the simulation satisfies the definition of continuous progress. For example users of said computer would quite quickly get frustrated if the execution of their program occasionally stalled for an undefined amount of time.

An example of a program that needs to stream out data at a constant rate is an mp3 decoder/player. If the program stalls and fails to send frequency information to the physical speakers at a constant rate the listening experience will be poor. Therefore it is a necessary condition that all reversible computation simulations for playing music and displaying video use an algorithm that satisfies continuous progress.

Plenty of examples can also be found in computation systems in finance. For example in algorithmic trading the trading systems emit information about the market and if the sending of the information is delayed it might be too late to trade based on this information.

Given that details on how exactly reversible computation will work with programs where information is non-deterministically streamed in computation have not yet been fully studied, it is hard to say how exactly continuous progress will fit in. However, it can be said with high confidence that continuous progress is a crucial requirement once a physical reversible computer can be built and actual real world programs are run on these computers.

As far as linear-time simulations are concerned, the most efficient presently known is due to Bennett [2]. We offer an improvement on his linear-time algorithm, which we call the k-ratic algorithm. This operates at essentially the same rate as Bennett's algorithm, but uses roughly half as much space.

Neither algorithm satisfies the continuous progress property, since an unbounded amount of time is taken up in periodic releasing of memory. We show how to modify each algorithm to ensure continuous progress, using multiple threads.

The paper is organised as follows. In Section 2 we look at the relevant previous work. Then in Section 3 we introduce the k-ratic algorithm. In Section 4 we

describe changes to Bennett's linear-time algorithm and to the k-ratic algorithm to allow them to make continuous progress. We finish with some conclusions.

2 Previous Work on Linear-time Simulations

This section will review existing research related to linear-time reversible simulations of forwards-only computation. The section starts off with Bennett's original 1973 algorithm. We then discuss the pebble game, a tool commonly used in the study of reversible computation. Finally we look at Bennett's 1989 algorithm, which has improved space efficiency compared to the original 1973 version.

2.1 Bennett's 1973 Simulation

The basic idea of Bennett's simulation is to construct reversible versions of all the elementary operations of a Turing machine. Doing this for a universal Turing machine will mean that all possible computer programs can be reversibly simulated.

We simulate a 1-tape Turing machine with a 3-tape Turing machine. The three tapes of the simulating machine are the work tape, the history tape and the output tape. At the beginning, the work tape contains the input of the machine and the other two tapes are empty. In the first stage the original computation is performed and at the same time the history tape is filled with padding from each single computational step to make the computation reversible. In the second stage the output is copied from the work tape onto the output tape. Finally in the third stage the work tape is converted back to the initial input with the help of the history tape. We are left with the input on the work tape and the output on the output tape, with the history tape empty.

Let the original computation take space S and time T, and let the reversible version of this computation take space S' and time T'. The work tape will take up space S. The history tape will at worst take up space $O(T)$. Finally the output will in the worst case take as much space as the work tape. Therefore space usage will be in the worst case $S' = 2S + T = O(S + T)$. Performing the first stage will take time $O(T)$. In the worst case the output will be as big as the amount of computation done, and so the second stage takes at most $O(T)$. Finally the third stage also takes $O(T)$. This sums up to $T' = O(T)$.

Bennett's simulation is very memory hungry. Note that T can be as much as $O(2^S)$, so that $S' = O(2^S)$ in terms of S alone. Considering that most modern computers can perform more computations per second than they have bytes of RAM, the simulation is infeasible in practice. Therefore a better implementation for reversible simulation that uses less space is needed.

2.2 The Pebble Game

The pebble game was briefly introduced by Bennett in [2], and later taken up by other researchers [8]. The pebble game has a board with an unbounded number

of squares labelled with natural numbers from 1 upwards, and the player is given m pebbles. Each pebble represents δ steps of computation. The kth square on the board represents computation from the $(k-1)\delta$th step to the $k\delta$th step in the original forwards-only computation. If a pebble is placed on the kth square it means that enough information is stored so that the kth segment of δ steps can be performed reversibly using Bennett's 1973 method.

The pebble game has the following rules.

- Initially all the squares of the board are not pebbled.
- The player can place a pebble on the board either at square 1, or at square k if the $(k-1)$th square has been pebbled.
- The player can remove a pebble either at square 1 or at a square k if the $(k-1)$th square has been pebbled.
- The objective of the game is to place a pebble as far as possible in the list of squares and then clear the board to a situation where only the furthest pebble (the one with the greatest advancement) remains on the board.[1]
- The player can have a maximum of m pebbles on the board at the same time.

The number of pebbles allowed represents the space usage of the reversible simulation. Bennett's 1973 algorithm can be interpreted in the pebble game as follows. To advance n squares with n pebbles, first lay down n pebbles in order from square 1 to square n. Then remove the pebbles in reverse order by starting from $n-1$ and going down back to 1.

2.3 Bennett's 1989 Simulation

In 1989 Bennett presented an algorithm to reversibly simulate a machine running in time T and space S. Given a constant k, he shows how to place pebbles on up to k^n squares using $n(k-1)+1$ pebbles, with $(2k-1)^n$ moves in the pebble game.

Bennett proved that the simulation operates in time $T' = O(T^{1+\epsilon})$ (so non-linear) and space $O(S \log T)$. His analysis was later refined by Levine and Sherman [7], who demonstrated that there is a big constant factor in the memory bound that grows exponentially in terms of ϵ^{-1}. They state the time and space bounds as $T' = \Theta(T^{1+\epsilon}/S^{\epsilon})$ and $S' = \Theta(S(1+\ln(T/S))$ with a constant factor in the space bound of approximately $\epsilon 2^{1/\epsilon}$.

In Bennett's algorithm, k is constant while n varies. Bennett remarks briefly that a linear-time variant can be obtained by holding n fixed and varying k. We next look at this in more detail. Note that we swap over n and k to reflect their new statuses.

Let us denote Bennett's algorithm for parameters n and k by $B(n,k)$. It works as follows. Let the original forwards-only computation use space S. Each

[1] In the game as described in [8] all pebbles are removed from the board, but the present formulation, matching Bennett's original description, is more convenient for the algorithms considered here.

square in the pebble game represents $m \approx S$ steps in the original computation. Using the 1973 algorithm these m steps can be performed in time $O(S)$ and space $O(S)$.

In order to advance by n^k squares in the pebble game, we pebble n blocks of n^{k-1} by calling $B(n, k-1)$ successively on blocks 1 to n. This gives us single pebbles at the end of each of the n blocks. We refer to the computation so far as the *advancement phase.* We then use $B(n, k-1)$ in reverse on blocks $n-1$ down to 1. We are left with a single pebble at square n^k. We refer to this latter part of the computation as the *clearing phase.* For the base case $k = 0$ a single pebble is placed on the first square.

Remark 2.1. The case for $k = 1$ is effectively Bennett's original 1973 algorithm.

The recurrence relation for the number of steps of $B(n, k)$ is

$$\begin{aligned} R(n,0) &= 1 \\ R(n,k+1) &= (2n-1)R(n,k) \qquad (k \geq 0) \end{aligned}$$

with solution $R(n, k) = (2n - 1)^k$.

Remark 2.2. Note that we count the number of steps in the pebble game, even though in fact each pebble placed represents $m \approx S$ steps.

The number of pebbles used by $B(n, k)$ is given by

$$\begin{aligned} P(n,0) &= 1 \\ P(n,k+1) &= P(n,k) + n - 1 \qquad (k \geq 0) \end{aligned}$$

with solution $P(n, k) = k(n - 1) + 1$.

The time T taken by the original computation is mn^k. The time T' taken by $B(n, k)$ satisfies $T' = O(T)$, since

$$\frac{R(n,k)}{n^k} = \left(\frac{2n-1}{n}\right)^k \leq 2^k. \tag{1}$$

So the algorithm runs in linear time. The space usage is $S' = (k(n-1)+1)O(S) = O(ST^{1/k})$.

3 The k-ratic Algorithm

We present a new linear-time reversible simulation algorithm, which we call the *k-ratic* algorithm. Like Bennett's algorithm, it splits the computation into blocks. However they are no longer of equal size; each successive block is smaller than its predecessor. In a sense the algorithm is greedier in using pebbles; Bennett's algorithm leaves more pebbles unused when executing the earlier blocks.

Let the k-ratic algorithm with parameters n and k be denoted by $K(n, k)$. It will use n pebbles, and works as follows. We first call $K(n-1, k-1)$ on block 1. This will leave a single pebble at the end of the block. We then successively

call $K(n-2,k-1),\ldots,K(1,k-1)$ on blocks $2,\ldots,n-1$, respectively. We are now left with pebbles at the end of $n-1$ blocks. We place the final pebble after block $n-1$. As with Bennett's algorithm, we refer to the computation so far as the *advancement phase*. If we stop at this point we refer to this as the *advancement-only* k-ratic algorithm.

For the *full* k-ratic algorithm we must remove the first $n-1$ pebbles representing intermediate checkpoints. We do this by successively calling each of $K(1,k-1),\ldots,K(n-1,k-1)$ in reverse. We are left with a single pebble immediately after the end of the last block. As before, we refer to this latter part of the algorithm as the *clearing phase*.

In the base case for $k=0$ we simply place a pebble on the first square and terminate.

Remark 3.1. In the case for $k=1$, note that $K(n,1)$ is the same as $B(n,1)$, and is effectively Bennett's original algorithm of [1]. Just place n pebbles on successive squares, and then remove pebbles $n-1,\ldots,1$ to leave only the last pebble.

A graphical demonstration of the method for $k=2$ can be seen in Figure 1.

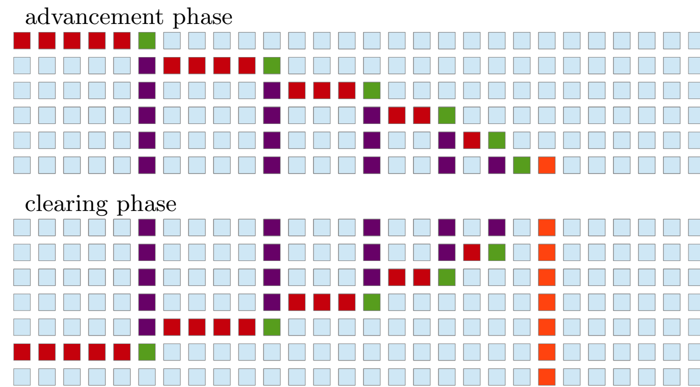

Fig. 1. Performing the k-ratic algorithm with $n=7$ and $k=2$.

Let us call the number of the square with the last pebble the *advancement* of $K(n,k)$; we denote it by $A(n,k)$. We get the following recurrence relation:

$$\begin{aligned} A(n,0) &= 1 \\ A(n,k+1) &= 1+\textstyle\sum_{i=1}^{n-1} A(i,k) \qquad (k\geq 0) \end{aligned}$$

Thus $A(n,1)=n$ and

$$A(n,2)=1+\sum_{i=1}^{n-1}A(i,1)=1+\sum_{i=1}^{n-1}i=\frac{n(n-1)}{2}+1\,.$$

The recurrence relation for the number of steps of $K(n,k)$ is

$$\begin{aligned} S(n,0) &= 1 \\ S(n,k+1) &= 1 + 2\textstyle\sum_{i=1}^{n-1} S(i,k) \qquad (k \geq 0) \end{aligned}$$

Thus $S(n,1) = 2n-1$ and

$$S(n,2) = 1 + 2\sum_{i=1}^{n-1} S(i,1) = 1 + 2\sum_{i=1}^{n-1}(2i-1) = 2(n-1)^2 + 1\,.$$

Let us denote the running time of the advancement-only k-ratic algorithm by $S_A(n,k)$. Then

$$S_A(n,k) = \frac{S(n,k)+1}{2}\,.$$

We now calculate an estimate of the advancement $A(n,k)$, using the following standard result.

Lemma 3.2. *Let $f : \mathbb{R} \to \mathbb{R}$ be continuous and non-decreasing on the range $[0,n]$. Then*

$$\int_0^{n-1} f(x)\,\mathrm{d}x \leq \sum_{i=1}^{n-1} f(i) \leq \int_1^n f(x)\,\mathrm{d}x\,.$$

Proposition 3.3. *For any $n \geq 1$, $k \geq 1$ we have $A(n,k) = (n^k/k!) + O(n^{k-1})$.*

Proof. We first show $A(n,k) \leq (n^k/k!) + O(n^{k-1})$ by induction on k. Clearly $A(n,1) \leq n^1 + O(1)$. Suppose that $A(n,k) \leq (n^k/k!) + O(n^{k-1})$. Then using Lemma 3.2

$$\begin{aligned} A(n,k+1) &= 1 + \textstyle\sum_{i=1}^{n-1} A(i,k) \\ &\leq 1 + \textstyle\sum_{i=1}^{n-1}(i^k/k!) + \sum_{i=1}^{n-1} O(i^{k-1}) \\ &\leq 1 + \textstyle\int_1^n x^k/k!\,\mathrm{d}x + \sum_{i=1}^{n-1} O(i^{k-1}) \\ &= 1 + (n^{k+1}-1)/(k+1)! + (n-1)O(n^{k-1}) \\ &= (n^{k+1}/(k+1)!) + O(n^k) \end{aligned}$$

We now show $A(n,k) \geq (n^k/k!) + O(n^{k-1})$. Clearly $A(n,1) \geq n^1 + O(1)$. Suppose that $A(n,k) \geq (n^k/k!) + O(n^{k-1})$. Then using Lemma 3.2

$$\begin{aligned} A(n,k+1) &= 1 + \textstyle\sum_{i=1}^{n-1} A(i,k) \\ &\geq 1 + \textstyle\sum_{i=1}^{n-1}(i^k/k!) + \sum_{i=1}^{n-1} O(i^{k-1}) \\ &\geq 1 + \textstyle\int_0^{n-1} x^k/k!\,\mathrm{d}x + \sum_{i=1}^{n-1} O(i^{k-1}) \\ &= 1 + (n-1)^{k+1}/(k+1)! + (n-1)O(n^{k-1}) \\ &= (n^{k+1}/(k+1)!) + O(n^k) \end{aligned}$$

We deduce that $A(n,k) = (n^k/k!) + O(n^{k-1})$ as required. □

The running time of $K(n,k)$ is no more than 2^k times the advancement:

Proposition 3.4. *For any $n \geq 1$, $k \geq 0$ we have $S(n,k) \leq 2^k A(n,k)$.*

Proof. By induction on k. It clearly holds for $k = 0$. Suppose $S(n,k) \leq 2^k A(n,k)$. Then

$$\begin{aligned} S(n,k+1) &= 1 + 2\sum_{i=1}^{n-1} S(i,k) \\ &\leq 1 + 2\sum_{i=1}^{n-1} 2^k A(i,k) \\ &= 2^{k+1}(1 + \sum_{i=1}^{n-1} A(i,k)) - (2^{k+1} - 1) \\ &= 2^{k+1} A(n,k+1) - (2^{k+1} - 1) \\ &\leq 2^{k+1} A(n,k+1) \end{aligned}$$

□

The k-ratic algorithm $K(n,k)$ achieves advancement of $A(n,k)$ squares in $S(n,k)$ steps. As before, each square in the pebble game corresponds to $m \approx S$ steps in the original computation. So we simulate $T = O(A(n,k))$ steps of the original computation in time $T' = O(S(n,k))$. It is clear from Proposition 3.4 that the k-ratic algorithm runs in linear time since $T' = O(T)$, just as for Bennett's algorithm. Indeed the ratio of 2^k of the number of steps of the simulating algorithm to the advancement achieved is the same in both cases, comparing Proposition 3.4 and Equation (1). The space usage is $S' = nO(S) = O(ST^{1/k})$ as in Bennett's algorithm.

However if we look in more detail at space, the k-ratic algorithm improves on Bennett's. Let us fix values for T/S, and for k. Bennett's algorithm uses $p_1 = k(n-1)+1$ pebbles with $n^k = T/S$. Thus $p_1 \approx (k^k T/S)^{1/k}$. The k-ratic algorithm uses $p_2 = n$ pebbles with $n^k/k! \approx T/S$. Thus $p_2 \approx (k! T/S)^{1/k}$. This gives us a ratio of $p_1/p_2 = (k^k/k!)^{1/k}$.

Proposition 3.5.

$$\lim_{k->\infty} (k^k/k!)^{1/k} = e$$

Proof. This is a consequence of Stirling's formula $n! \sim \sqrt{2\pi n}(n/e)^n$. □

By Proposition 3.5, p_1/p_2 tends to e as k increases; in fact it has a value ≥ 2 for $k \geq 6$. Thus the k-ratic algorithm uses roughly half as much space, a modest improvement.

The improvement is larger if we consider how much advancement can be made for a given amount of space (number of pebbles). Suppose we are given $k(n-1)+1$ pebbles as in Bennett's algorithm. Then $B(n,k)$ advances by n^k squares. The k-ratic algorithm can advance by

$$A(k(n-1)+1,k) = ((k(n-1)+1)^k/k!) + O(n^{k-1}) = (k^k/k!)n^k + O(n^{k-1})$$

The ratio $k^k/k!$ is of course simply a constant, but it is quite large for even small values of k:

Proposition 3.6. *For $k \geq 1$, $k^k/k! \geq 2^{k-1}$.*

Proof. By induction. It clearly holds for $k = 1$. We have

$$\frac{(k+1)^{k+1}}{(k+1)!} = \frac{(k+1)^k}{k!} \geq \frac{k^k + k.k^{k-1}}{k!} = 2\frac{k^k}{k!}$$

Hence result. □

In fact $10^{10}/10! = 2755.7$. Thus even for modest values of k we get an improvement on advancement for the same amount of space usage.

We now turn to the issue of how to choose a suitable value of the parameter k. Even though the reversible simulation of T steps is linear with respect to T, the constant factor 2^k grows exponentially. This makes it in practice necessary to choose a small value of k. This is further supported by the diminishing returns of lower memory usage as k increases.

$$T' = \Theta(2^k T) \tag{2}$$

$$S' = \Theta(S \sqrt[k]{(T/S)k!}) \tag{3}$$

In the formal definition of big-O notation the $k!$ and 2^k should not exist in the notations as they are constant. However in this case they are quite large and might in practice have a significant impact on the computation time and space usage.

We now consider what value of k we should choose for optimal results, given a particular value of T/S. By Proposition 3.5 we have $\sqrt[k]{k!} = \Theta(k)$, which using Equation (3) gives us

$$S' = \Theta(kS \sqrt[k]{(T/S)})\,.$$

To find the value of k for which S' reaches a minimum, we differentiate the function $y = x(T/S)^{1/x}$:

$$\mathrm{d}y/\mathrm{d}x = (T/S)^{1/x}(x - \ln(T/S))/x$$

meaning that $k \approx \ln(T/S)$ is the minimum.

As an example, if $T/S = 10^{20}$, space S' would improve up to roughly $\ln 10^{20} = 46.05$. However such a value of k would be far too large as far as time is concerned, given the 2^k slowdown in Equation (2). As far as space is concerned, after about $k = 7$ diminishing returns set in on the improvement in memory compared to the extra time.

Clearly there is a trade-off between space and time when choosing the value of k. How to resolve this will depend on the particular application. However we can estimate a suitable value of k based on the following heuristic. For a given value of T/S we decide on a value of k by finding the greatest k such that increasing from $k-1$ to k doubles the space usage. Increasing k further would mean that we were doubling the time taken but not halving the space. Call this value $k_M(T/S)$. See Figure 2 for a plot of k_M for values up to 10^{100}, yielding e.g. a value of 17 for $T/S = 10^{100}$. Note that the horizontal axis is log-scale.

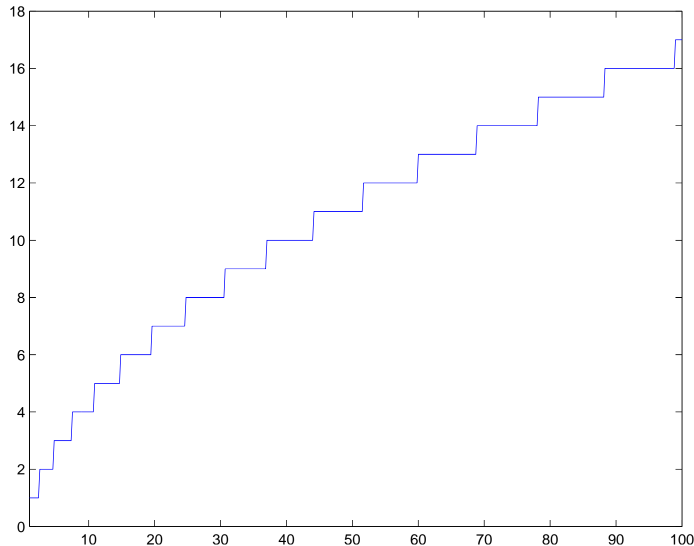

Fig. 2. Plot of $k_M(T/S)$ against $\log_{10}(T/S)$.

4 Continuous Progress

We are interested in real-time reversible simulations of forwards-only computations. As stated in the Introduction, we identify *continuous progress* as a requirement for a simulation to be real-time. It will be convenient to allow multiple threads in the simulating program. A single step of the simulation means that each of its threads makes a step (or idles). Simulating programs can take a variable parameter n which allows for the capacity to simulate an indefinitely increasing number of steps depending on n (with a corresponding increase in memory usage).

Definition 4.1. *A (multi-threaded) simulation program* $\mathsf{Sim}(n)$ *makes* continuous progress *if and only if there is some constant* $p \in \mathbb{N}$ *(not depending on* n*) such that for every* n*, the program that is being simulated advances at least one computational step for every* p *steps of computation performed by the simulating program* $\mathsf{Sim}(n)$*. If such a* p *exists, we call it the* progression factor *of the simulation.*

Remark 4.2. If the simulation is reversible, it may have to perform further computation after reaching maximum advancement, i.e. after the forward computation has been fully simulated. We will still allow such a simulation to satisfy the condition for continuous progress.

A progression factor greater than one does not prevent a simulation in real time, as long as the simulating computation is run on a faster processor than the original computation.

From Definition 4.1 it follows that the simulation operates in linear time compared to the program being simulated. If the latter makes T steps, then the former makes $\leq pT = O(T)$ steps. However continuous progress is a stricter requirement than just linear time simulation, as it could be the case that on average the simulation advances linearly, but occasionally progress hangs for arbitrarily long time.

Example 4.3. Consider a simulation that takes the following amount of steps per progression of the program being simulated; it is linear time but not continuously progressing:

$$F(n) = \begin{cases} n & \text{if } n \text{ is a power of 2} \\ 1 & \text{otherwise} \end{cases}$$

Letting $k = \lfloor \log n \rfloor$ we have

$$\sum_{i=0}^{n} F(i) = (n+1) - (k+1) + \sum_{j=0}^{k} 2^j = (2^{k+1} - 1) + n - k \leq 3n\,.$$

Therefore on average to progress n steps in the original computation approximately $3n$ steps need to be performed by the simulation. Hence the simulation is linear time. However it is not continuously progressing due to the increasing stalls in progress that happen at powers of two. Such stalls also occur in Bennett's algorithm $B(n,k)$ described in Section 2.3, and in the k-ratic method $K(n,k)$ described in Section 3; in both cases the interval between successive advancement steps can be as much as $O(n^{k-1})$ steps of simulation.

Lemma 4.4. *If a q-threaded algorithm has a progression factor of p then there is a corresponding single-threaded algorithm with progression factor qp.*

Proof. Simply schedule the q threads onto a single thread in a round-robin fashion. □

In particular cases where different threads carry out different amounts of work, we may be able to improve on the bound given by Lemma 4.4, of course.

Both Bennett's 1989 linear-time algorithm (Section 2.3) and the k-ratic algorithm (Section 3) fail to exhibit continuous progress, due to the interruptions to forward progress for the clearing phases. We now look at how to reprogram them onto multiple concurrent threads to ensure continuous progress. Multiple threads are not essential by Lemma 4.4, of course.

A natural point to utilise multi-threading in algorithms such as we have considered is at the point when the advancement phase for a block finishes. At this point there is a need to backtrack and erase previously laid pebbles before proceeding to the next block. It is possible to multi-thread this part by having one thread continue forward while another thread frees the pebbles laid down in the past.

We start by allowing a comparatively large number of threads, namely $O(2^k)$, where k is the constant parameter in the algorithms described earlier. In the case

of Bennett's algorithm by using 2^{k-1} threads we can get a progression factor of one.

Theorem 4.5. *For $k \geq 1$, we can program $B(n,k)$ onto 2^{k-1} threads with a progression factor of* 1*.*

Proof. By induction. The base case $k = 1$ with a single thread is clear. To perform $B(n, k+1)$ on n blocks, we divide the 2^k threads into 2^{k-1} used for advancement, and an equal number used for clearing. We start by using the advancement threads to perform the advancement phase of $B(n,k)$ on block 1. We know by induction that the progression factor is 1. Then for $i = 1, \ldots, n-1$, we simultaneously perform the clearing phase of $B(n,k)$ on block i and the advancement phase of $B(n,k)$ on block $i+1$. Again the progression factor is 1. Hence result. □

By using 2^{k-1} threads we have improved the parallel time for the advancement phase to be equal to the advancement n^k. This is a 2^{k-1} speed-up compared to the sequential version. The total number of pebbles used is given by

$$\begin{aligned} P'(n,1) &= n \\ P'(n,k+1) &= 2P'(n,k) + n - 2 \qquad (k \geq 1) \end{aligned}$$

with solution $P'(n,k) = (2^k - 1)(n-1) + 1$. This may be compared with $P(n,k) = k(n-1) + 1$ for the original algorithm.

Suppose now that we have fewer than 2^{k-1} threads available. Let the number of threads be 2^j where $0 \leq j < k$. Then we can run the algorithm of Theorem 4.5 on 2^j threads by time-sharing as in Lemma 4.4. We still get continuous progress, with a progression factor of 2^{k-1-j}. The pebble usage will still be $P'(n,k)$.

In particular, if we take the 2^{k-1}-thread version and schedule it onto a single processor, we get a progression factor of 2^{k-1}. The pebble usage is of course greater than the $P(n,k)$ of the original algorithm, though only by a constant factor.

If we wish to economise on memory, as an alternative to Theorem 4.5 we can use k threads instead of 2^{k-1}.

Theorem 4.6. *For $k \geq 2$, we can program $B(n,k)$ onto k threads with a progression factor of* 2^{k-2}*.*

Proof. By induction on k. For $k = 1$ we have one thread and we have a progression factor of $1 = 2^{k-1}$.

Suppose true for $k \geq 1$. The threaded algorithm for $k+1$ uses $k+1$ threads. The algorithm divides the work into n blocks $1, 2, \ldots, n$, each of size n^k. We perform each successive advancement of block i using threads 1 to k, but slowed down by a factor of 2 compared to the threaded algorithm for k, except if $k = 1$, when we proceed at the usual rate. Once block i is finished we clear it up using thread $k+1$ operating at the normal rate, while threads 1 to k are advancing through the next block $i+1$. The time taken by thread $k+1$ is $R(i,k)/2 \leq 2^{k-1}n^k$ using Equation (1). The time taken by threads 1 to k on block $i+1$ is

$2.2^{k-2}n^k = 2^{k-1}n^k$ if $k \geq 2$, and $1.2^{k-1}n^k = 2^{k-1}n^k$ if $k = 1$. Hence thread $k+1$ will finish no later than threads 1 to k. The progression factor is 2^{k-1}. Hence result. □

When scheduling onto a single processor we get a progression factor of $k2^{k-2}$, which is not quite as good as the 2^{k-1} offered by Theorem 4.5. However the space usage improves. The method of Theorem 4.6 uses $P''(n,k)$ pebbles where

$$\begin{aligned} P''(n,1) &= P(n,1) \\ P''(n,k+1) &= P''(n,k) + P(n,k) + n - 2 \qquad (k \geq 1) \end{aligned}$$

with solution $P''(n,k) = k(k+1)(n-1)/2 + 1$.

We can obtain a result similar to Theorem 4.6 for the k-ratic algorithm, but the progression factor increases by a multiple of two, due to the blocks being of different sizes, rather than all the same size. We first state a lemma concerning the advancement $A(n,k)$ of the k-ratic algorithm $K(n,k)$.

Lemma 4.7. *For $n \geq 2$ and $k \geq 0$ we have $A(n,k) \leq 2A(n-1,k)$.*

Proof. By induction on k. We easily check the case for $k = 0$. Suppose $A(n,k) \leq 2A(n-1,k)$ for all $n \geq 2$.

$$\begin{aligned} A(n,k+1) &= 1 + \textstyle\sum_{i=1}^{n-1} A(i,k) \\ &= 1 + A(1,k) + \textstyle\sum_{i=2}^{n-1} A(i,k) \\ &\leq 1 + A(1,k) + \textstyle\sum_{i=2}^{n-1} 2A(i-1,k) \\ &= 1 + A(1,k) + 2\textstyle\sum_{i=1}^{n-2} A(i,k) \\ &= 1 + 1 + 2(A(n-1,k+1) - 1) \\ &= 2A(n-1,k+1) \end{aligned}$$

□

The ratio of 2 in Lemma 4.7 is the best possible in general, since e.g. $A(1,2) = 1$ and $A(2,2) = 2$.

Remark 4.8. In fact for any $k \geq 0$, $\lim_{n\to\infty} A(n,k)/A(n-1,k) = 1$. To see this note that it is easy to show that $A(n,k) = A(n-1,k) + A(n-1,k-1)$ for any $k \geq 1$, $n \geq 2$. Also by Proposition 3.3 we have $\lim_{n\to\infty} A(n,k-1)/A(n,k) = 0$.

Theorem 4.9. *The k-ratic algorithm $K(n,k)$ can be programmed with k threads with a progression factor of 2^{k-1}.*

Proof. By induction on k. For $k = 1$ we have one thread. The progression factor is clearly $1 = 2^{k-1}$.

Suppose true for k. The threaded algorithm for $k+1$ uses $k+1$ threads. The algorithm divides the work into $n-1$ blocks $1, 2, \ldots, n-1$, where block i has size $A(n-i,k)$. We perform each successive advancement of block i using threads 1 to k, but slowed down by a factor of two compared to the threaded algorithm for k. Once block i is finished we clear it up using thread $k+1$ operating at the normal rate, while threads 1 to k are advancing through the next block $i+1$.

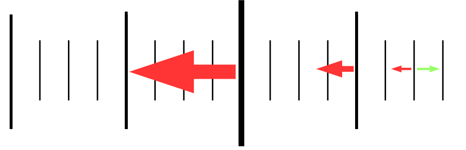

Fig. 3. Multi-threading with $k = 4$.

We illustrate this in Figure 3, which shows the way in which different erasure threads (in red) have to operate at increasing rates in order to keep up with forward progression (in green).

The time taken by thread $k+1$ is $S(i,k)/2 \leq 2^k A(i,k)/2 \leq 2^k A(i-1,k)$ using Proposition 3.4 and Lemma 4.7. The time taken by threads 1 to k on block $i+1$ is $2.2^{k-1}A(i-1,k) = 2^k A(i-1,k)$. Hence thread $k+1$ will finish no later than threads 1 to k. The progression factor is $2.2^{k-1} = 2^k$. Hence result. □

The number of pebbles used by the method of Theorem 4.9 is $k(n-1)-1$, compared to n for the original algorithm $K(n,k)$.

5 Conclusions

We have studied real-time reversible simulations of forwards-only computations. As far as we are aware, such simulations have not been studied previously.

The first part of this paper presented a new algorithm for reversible computation called the k-ratic method. The k-ratic method is a technique to reversibly simulate a forward-only computation. Letting T and S be the time and space used by the forwards-only computation, the k-ratic method uses $O(2^k T)$ time and $O(kS\sqrt[k]{T/S})$ space, where k is a constant. It also uses up to a factor of e less space than Bennett's linear-time algorithm. We considered how to pick a suitable value for k, taking into account the trade-off between time and space.

The latter part of the paper introduced the notion of continuous progress. For a program to satisfy the condition of continuous progress it is necessary that a upper bound must exist on the number of steps the simulating program can advance without the advancement of the program being simulated. The paper then explored how Bennett's technique and the k-ratic method can be modified with the help of multi-threading to satisfy the definition of continuous progress.

Two different ways to achieve continuous progress are discussed. The first method uses $O(2^k)$ threads and increases memory usage by a factor of $O(2^k)$. However it is able to achieve an upper bound of one for continuous progress. The second method increases memory usage by a factor of $O(k)$. However it is only able to provide a continuous progress upper bound of $O(2^k)$.

References

1. Bennett, C.: Logical reversibility of computation. IBM Journal of Research and Development 17, 525–532 (1973)

2. Bennett, C.: Time/space trade-offs for reversible computation. SIAM Journal on Computing 18(4), 766–776 (1989)
3. Buhrman, H., Tromp, J., Vitányi, P.: Time and space bounds for reversible simulation. In: Proceedings of 28th International Colloquium on Automata, Languages and Programming, ICALP 2001. LNCS, vol. 2076, pp. 1017–1027. Springer-Verlag (2001)
4. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development 5, 183–191 (1961)
5. Lange, K., McKenzie, P., Tapp, A.: Reversible space equals deterministic space. Journal of Computer and System Sciences 60(2), 354–367 (2000)
6. Lecerf, Y.: Machines de Turing réversibles. Récursive insolubilité en $n \in N$ de l'équation $u = \theta^n u$, où θ est un "isomorphisme de codes". Comptes Rendus 257, 2597–2600 (1963)
7. Levine, R., Sherman, A.: A note on Bennett's time-space tradeoff for reversible computation. SIAM Journal on Computing 19(4), 673–677 (1990)
8. Li, M., Vitányi, P.: Reversibility and adiabatic computation: Trading time and space for energy. Proc. Royal Society of London, Series A 452, 769–789 (1996)
9. Vitányi, P.: Time, space, and energy in reversible computing. In: Proceedings of the Second Conference on Computing Frontiers, 2005, Ischia, Italy, May 4-6, 2005. pp. 435–444. ACM (2005)
10. Williams, R.: Space-efficient reversible simulations (2000). `http://www.stanford.edu/~rrwill/spacesim9_22.pdf`