7805 Class project, 2012: Group 1

Tony Garnock-Jones, Zahra Jafargholi, Ravishankar Rajagopal,
Bochao Shen, Saber Shokat Fadaee, Triet Vo Huu

April 28, 2012

1 Overview

We show that any function computable in time $T(n)$ on a non-deterministic random-access computer (as defined below) is computable in time $T(n)(\log T(n))^{O(1)}$ on a non-deterministic k-tape sequential-access Turing machine.[1] We provide a chain of simulating machines, each of which efficiently simulates the preceding machine, leading from our initial random-access machine model to our final sequential-access model.

Definition 1. (Efficient simulation, after [1].) A machine of some kind that is $T(n)$-time-bounded is said to be *efficiently simulated* by some machine of another kind when the simulating machine is time-bounded by $T(n)$ multiplied by a polynomial of $\log T(n)$; that is, when the overhead of simulation is bounded by a polylog function of $T(n)$.

Although we start from a specific machine model, that of a random-access computer, it is shown in [1] that a number of variants, including random-access Turing machines, Kolmogorov machines and Schönhage machines, all efficiently simulate each other and therefore share the same notion of n^k computability, $k \in \mathbb{N}$.

Outline

We first define the machine models we will be using: random-access computers (RACs), random-access Turing machines (RTMs), and k-tape sequential-access Turing machines. We then show that every specific instance of the RAC model can be efficiently simulated by an RTM, and furthermore that the simulating RTM's behavior is bounded in certain ways that will be important to the proof of our main claim. Finally, we move to a non-deterministic setting and show that any computation performed by a non-deterministic RTM can be efficiently simulated on a non-deterministic k-tape Turing machine, where $k > 1$. In order to accomplish this, we prove and make use of the fact that k-tape Turing machines can sort in time $n \log n$, where n is the length of the input.

[1] We restrict ourselves to considering situations where $T(n) \geq n$.

2 Definitions: RAC, ℓ-bit RTM, k-tape TM

Definition 2. (Random-access Computer, RAC.) A random-access computer is, quoting [1], "an abstract machine with a sequence of memory locations", where the size of each memory cell and the number of such cells varies with the size of the input to the machine. In particular, each memory cell is ℓ bits wide, where $\ell = c \log n$, c is a constant specific to the particular machine being considered, and n is the size of the machine's input. Furthermore, there are exactly 2^ℓ memory cells available, meaning that the size of each cell is exactly large enough to represent any address in the machine's memory space.

RACs are similar in spirit to the CPUs we use in our day-to-day work with desktop computers. Time in an RAC is measured in terms of the number of instructions executed. In order to be precise about the simulations we will be defining, we choose to pin down a few details omitted in [1]:

- Our RAC will have a fixed number of ℓ-bit registers.
- The instructions in our RAC will be LOAD from memory to register, STORE from register to memory, the usual register-to-register arithmetic operations, and control transfers. Our primary concern will be with memory accesses, so the exact instruction set other than loads and stores is not important.

Definition 3. (ℓ-bit Random-access Turing Machine, ℓ-bit RTM.) We start from the verbatim[2] definition of an RTM from [1]:

> An RTM is a Turing machine with three linear tapes called the *main tape*, the *address tape* and the *auxiliary tape*, such that the head of the main tape (the *main head*) is always on the cell whose number is the contents of the address tape. An instruction for an RTM has the form
>
> $$(p, \alpha_1, \alpha_2, \alpha_3) \to (q, \beta_1, \beta_2, \beta_3, \gamma_1, \gamma_2)$$
>
> and means the following: If the control state is p and the symbols in the observed cells on the three tapes are binary digits $\alpha_1, \alpha_2, \alpha_3$ respectively, then print binary digits $\beta_1, \beta_2, \beta_3$ in the respective cells, move the address head to the left (resp. to the right) if $\gamma_1 = -1$ (resp. $\gamma_1 = 1$), move the auxiliary-tape head with respect to γ_2 and go to control state q.

We extend Gurevich and Shelah's definition with the constraint that both the address and the auxiliary tapes are bounded so as to be no longer than $O(\ell)$ bits each.[3]

Definition 4. (k-tape sequential-access Turing machines, k-tape TMs.) A k-tape TM is a Turing machine with k linear tapes, each of which has a corresponding head, which is constrained to move one cell at a time along its tape, just like a normal single-tape Turing machine. Instructions for k-tape TMs have the form

$$(p, \alpha_1, \alpha_2, ...\alpha_k) \to (q, \beta_1, \beta_2, ...\beta_k, \gamma_1, \gamma_2, ...\gamma_k)$$

[2] There are a couple of typos in Gurevich's paper; we have corrected them in the presentation here.

[3] Note that this is different to the class of *frugal* RTMs, defined as part of the chain of machines in [1]. The lengths of the address and auxiliary tapes in a frugal RTM are bounded by $O(\log t)$ at every moment in the operation of the machine; we do not rely on such a strict bound here, so do not further consider frugal RTMs as defined by Gurevich and Shelah.

where p is the control state, the αs are the contents of the cells under the heads of the k tapes, q is the resulting control state, the βs are to be written into the cells under the heads, and the γs control each head's movements in the same way as they do for RTMs as defined above.

3 Efficient simulation of RACs using ℓ-bit RTMs

Lemma 5. *RACs with ℓ-bit memory cells can be efficiently simulated by ℓ-bit RTMs.*

Proof. Since RACs with ℓ-bit memory cells also have exactly 2^ℓ such cells, we simulate the action of the RAC's memory with the main tape of our RTM. The main tape is logically divided into 2^ℓ ℓ-bit-wide sections, each corresponding to one memory cell in the simulated RAC. The auxiliary tape of the RTM is used to hold the registers of the RAC and also to perform arithmetic involved in the execution of RAC instructions.

The RAC's instructions are then encoded into the state transition function of the RTM. Control transfer instructions can be encoded directly in the state transition function, and register-to-register arithmetic and logic instructions can be performed in $O(\ell)$ space and $O(\ell^2)$ time on the auxiliary tape. The remaining instructions are LOAD and STORE.

Each LOAD instruction takes $O(\ell)$ time to copy from a register held on the auxiliary tape to the address tape. Once the copy is completed, ℓ adjacent bits must be copied from the main tape to the auxiliary tape, which involves ℓ transfers of a single bit and ℓ incrementations of the value held on the address tape. Transferring one bit takes worst-case $O(\ell)$ time for the head to move back and forth to the right location on the auxiliary tape, since the tape itself has length bounded by $O(\ell)$, and incrementing the address tape's value takes worst-time $O(\ell)$ steps, so overall the process of copying the ℓ bits from the main to the auxiliary tape takes a total of $O(\ell^2)$ time. In summary, then the LOAD instruction as a whole takes time bounded by $O(\ell^2)$. The time bound for STORE instructions is the same, since the copying of the bits simply takes place in reverse.

Since every class of instruction can be completed in $O(\ell^2)$ time, we see that the simulation completes in $O(T(n)\ell^2)$. Recalling that $\ell = c \log n$, and $T(n) \geq n$ by assumption, we see that this is equivalent to saying the simulation completes in $O(T(n)(c \log T(n))^2)$, which is itself equivalent to $T(n)(\log T(n))^{O(1)}$, which is what is required for the definition of efficient simulation to apply. □

4 Efficient simulation of non-deterministic ℓ-bit RTMs using non-deterministic k-tape TMs

Simulating a computation on a non-deterministic ℓ-bit RTM using a non-deterministic multi-tape TM is carried out by using the non-deterministic input to the latter to *guess* the sequence of steps involved in the former's computation and then checking the correctness of the guess. However, in order to do the latter, we need to show that multi-tape TMs can sort efficiently.

We proceed in this section by first getting the question of efficient sorting out of the way, and then being more precise about the kind of non-determinism we are using. We then formally define the way the simulating machine represents the steps taken by the simulated RTM, and finally we bring all the pieces together to show our main result.

4.1 Efficient sorting

This section is based on a lemma given in [2] as part of a proof that the graph isomorphism problem is in NQL.

Lemma 6. *Let L be a list containing $|L|$ natural numbers, and let $C(L)$ be an encoding of L, of length $|C(L)|$ symbols. Then, L can be sorted by a k-tape TM in time $O(|C(L)| \log |L|)$.*

Proof. In the following let $[n] = \{1, 2, \ldots n\}$. Let $L = (l_i \in N : i \in [m])$ be a list (sequence) of natural numbers $(l_1, l_2, \ldots l_m)$. Then the encoding $C(L) \in \Sigma^*$ is defined as

$$C(L) = c(l_1)\#c(l_2)\# \ldots c(l_m)\#$$

where Σ, the input alphabet to our sorting machine, is the set including 0 and 1 as well as the pound symbol (#), and $c(v)$ is the binary representation of $v \in \mathbb{N}$. For example, $C((1, 2, 13)) =$ 1#10#1101#.

We define Γ, our tape alphabet, to be Σ with the addition of the symbol comma (,). We proceed using an algorithm based on merge sort, where we repeatedly merge adjacent sorted sublists until just one list remains. At the ith iteration over the input, the sublists we consider have length 2^{i-1}.

We choose to use three tapes: the first, main, tape holds the input and the partial outputs at each iteration, and the second and third tapes are used as working space to hold the sublists to be merged.

The operation of the machine in each iteration is as follows:

1. Return the heads to the beginnings of the tapes.
2. If we are at the end of the main tape, then
   - if we haven't yet processed any sublists during this iteration, the original input was empty. Halt, since an empty list is always sorted.
   - otherwise, go to step 1.
3. Copy symbols from the main tape to the second tape until a pound symbol is seen.
4. If the end of the tape follows the pound symbol we just saw, then
   - if this was the first sublist we processed during this iteration, then we are done: the input has been sorted. Halt.
   - otherwise, we have an odd number of sublists. The final iteration of the algorithm will take care of this case. Return to step 1.
5. Otherwise, copy symbols from the main tape to the third tape until a pound symbol is seen.
6. Return the heads of the second and third tapes to the beginning, and move the head of the main tape back over the two sublists just copied to the first symbol of the first sublist of the two.
7. If the symbol on the second tape is the pound symbol, place a comma on the main tape and then copy symbols from the third tape onto the main tape until we see a pound symbol on the third tape also. Place a pound symbol on the main tape and go to step 2.

8. Otherwise, if the symbol on the third tape is the pound symbol, place a comma on the main tape and then copy symbols from the second tape onto the main tape until we see a pound symbol on the second tape also. Place a pound symbol on the main tape and go to step 2.

9. Otherwise, compare the encodings of the numbers under the heads of the second and third tapes. If the smaller is the element from the second tape, copy it to the main tape; otherwise, copy the element from the third tape. Place a comma on the main tape. Rewind whichever of the tapes did not have an element copied from it so that its element will be compared again next time round. Go to step 7.

If there are an even number of pound-symbol-delimited sublists in the input, then they are of equal length at the start of each iteration; and if there are an odd number in the input, then all but one will have equal length. To see this, consider that as the algorithm begins, all sublists are of length 1, and as it proceeds, each pair of length x sublists is replaced by a single length $2x$ sublist, except for any odd-man-out at the end of the input, which will not be processed until the final iteration.

It remains to be shown that the machine takes time $O(|C(L)| \log |L|)$. We can see that there will be $log_2|L|$ iterations, since each iteration halves the number of remaining sublists until a single sublist remains. Each iteration will take $O(|C(L)|)$ steps, since every element of the input is examined on each iteration, and $O(1)$ steps are needed to perform the necessary copyings and comparisons for each element of the input. Combining these two observations gives us the required bound. □

4.2 Non-determinism

Definition 7. We define a *non-deterministic* Turing machine M to be the equivalent of some other deterministic machine M whenever M accepts some input x in $T(n)$ steps *if and only if* there exists some additional input y of size bounded by $T(n)(\log T(n))^{O(1)}$ such that M' accepts (x, y).

4.3 Traces of the operation of ℓ-bit RTMs

Informally, the *trace* of an ℓ-bit RTM computation is the sequence of states that the machine passes through during a computation, including the relevant portions of the machine's tapes.

Definition 8. (Traces and trace tuples.) The trace of an ℓ-bit RTM computation is the sequence of trace tuples

$$(t, q_t, a_t, I_t, b_t, J_t, c_t) : t < T(n)$$

such that at time t, the TM is in state q_t, the contents of the address tape are a_t, the address tape's head is positioned at I_t, the contents of the auxiliary tape are b_t, the auxiliary head's position is J_t, and the character under the head of the main tape is c_t.

Note that a_t and b_t are each $O(\ell)$ bits long, by the definition of an ℓ-bit RTM.

Definition 9. (Consistency of trace tuples.) A tuple $(t, q', a', I', b', J', c')$ is *consistent* with the tuple $(t+1, q, a, I, b, J, c)$ if, when the machine is in the state described by the first tuple at time t, each of q, a, I, b and J *could* result from the operation of the machine's transition function given the inputs q', a', I', b' and J'.

Note that c' and c are not checked as part of this definition: they are dealt with separately below.

4.4 Main result

Theorem 10. *k-tape nondeterministic TMs can efficiently simulate non-deterministic ℓ-bit RTMs.*

Proof. Let M be an ℓ-bit RTM and M' be the non-deterministic k-tape TM intended to simulate the action of M. The input (x, y), incorporating both the original input x and the non-deterministic contribution y to M, is placed on one of the tapes of M', along with a further non-deterministic contribution y'. M' uses y' to construct a guess of a possible trace for the computation of M on input (x, y), guessing tuples in order from $t = 0$ through $t = T(n)$. Since M's address and auxiliary tapes are bounded in size by $O(\ell)$, and ℓ is bounded by a polylog function of $T(n)$, each tuple to be guessed consumes at most polylog $T(n)$ bits of information from y'. In other words,

- t is a number between 0 and $T(n)$ and therefore requires $\log T(n)$ bits
- q_t has constant size and depends only on the definition of M
- a_t has size bounded by $O(\log T(n))$ by definition
- b_t is also bounded by $O(\log T(n))$ by definition
- I_t is a number representing a location on the address tape, at most $O(\log T(n))$, and so is of size $O(\log \log T(n))$
- J_t is a number representing a location on the auxiliary tape, therefore is also of size $O(\log \log T(n))$
- c_t has constant size and depends only on the size of the alphabet handled by M

so the overall size of each tuple is bounded by $O(\log T(n))$.

M' guesses the trace tuples one by one, performing an initial correctness-check as it goes by making sure that the first tuple is correct, that each subsequent tuple is consistent with its predecessor, and finally that the last tuple in the trace has $q_{T(n)}$ an accepting state.

Recall that when we defined trace tuple consistency above, we explicitly neglected to check the c_t in each tuple. The reason for this is that the main tape's head accesses the main tape in a random-access fashion rather than sequentially. Once the initial pass over the generated trace is complete, however, we go back and check the c_t for consistency. To do so,

1. Sort the trace first by the contents of the address tape, a_t, and then by time, t. After this process, groups with equal a_t will exist, with each tuple's t increasing within each group. By lemma 6, we know that this sorting process can be carried out efficiently.
2. For each block with the same value of a_t and increasing t, we first check that the c_t component of the first tuple is the same as the character M' observes on its copy of the original input to the machine at address a_t. If this is not the case, our trace is inconsistent with the operation of M.
3. For two adjacent tuples within the same block, let c be the c_t component of the first tuple and c' be the corresponding component on the second tuple. Then, for the pair of tuples to be consistent with each other, one of the following conditions must hold:
   (a) $c = c'$ and the computation following from application of M's transition function to the first tuple would not have altered the main tape at a_t; or

(b) $c \neq c'$, and the transition function applied to the first tuple would have written exactly c' at a_t.

If each a_t-group of tuples is consistent, then the entire trace represents a legitimate accepting execution of M, and so M' accepts.

Since we generate the trace in time $T(n)$, perform the first consistency-check pass in $O(T(n))$, sort the trace in $O(T(n) \log T(n))$, and perform the second consistency-check pass in $O(T(n))$, we see that the overall process has time bounded by $O(T(n) \log T(n))$, which is a polylog overhead compared to $T(n)$, and so the whole process is an efficient simulation. □

References

[1] Yuri Gurevich and Saharon Shelah. Nearly linear time. In Albert R. Meyer and Michael A. Taitslin, editors, *Logic at Botik*, volume 363 of *Lecture Notes in Computer Science*, pages 108–118. Springer, 1989.

[2] C. P. Schnorr. Satisfiability is quasilinear complete in NQL. *J. ACM*, 25(1):136–145, January 1978.