Simultaneous Bounds on Time and Space

Jonathan F. Buss* and Kenneth W. Regan

[1] Cheriton School of Computer Science
University of Waterloo
[2] Department of Computer Science and Engineering
University at Buffalo

Abstract. We consider complexity classes defined by imposing simultaneous bounds on time and space. We show that general time-space tradeoffs can be recursively extended, thereby achievng better improvements in time than obtainable from simple padding. For example, a constant-factor speedup at linear time implies a quadratic improvement in the minimal time required to accept the hardest languages accepted in linear space;, that is, any language acceptable using cn bits of space becomes acceptable in $2^{(cn/2)+o(n)}$ time.

1 Introduction

Time and space have long been considered the most important computational resources. Each has received extensive investigation, both on its own and in relation to the other, as exemplified by the standard time and space complexity classes $\mathsf{Time}(t)$ and $\mathsf{Space}(s)$. Comparatively little is known regarding computations in which both resources are simultaneously and non-trivially bounded in single computations. Let $\mathsf{TiSp}(t, s)$ denote the class of problems decidable by some multitape Turing machine that uses time $O(t)$ and also space $O(s)$. Likewise let $\mathsf{FTiSp}(t, s)$ denote the class of functions computable by some multitape Turing machine that uses time $O(t)$ and also space $O(s)$.

Consider the following well-known results.

Proposition 1. *Le $t_1(n) > n$ and $t_2(n)$ be time-constructible functions such that $t_1 \log t_1 \in o(t_2)$, and let s_1 and s_2 be space-constructible functions such that $s_1 \in o(s_2)$.*

- $\mathsf{Time}(t_1) \neq \mathsf{Time}(t_2)$ *(Hartmanis and Stearns [4]).*
- $\mathsf{Space}(s_1) \neq \mathsf{Space}(s_2)$ *(Stearns, Hartmanis and Lewis [9]).*
- *The proofs of the above can be combined, yielding* $\mathsf{TiSp}(t_1, s_1) \neq \mathsf{TiSp}(t_2, s_2)$ *(with slightly stronger constructibility requirements).*

* Supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada and by SUNY Buffalo.

Proposition 2 (Savitch [8]).

$$\mathsf{NTime}(t) \subseteq \mathsf{TiSp}\big(2^{O(t^2)}, t^2\big) \ .$$

Proposition 3 (Hopcroft, Paul, Valiant [6]; also Adelman, Loui [1]).

$$\mathsf{TiSp}(t,t) \subseteq \mathsf{TiSp}(2^{O(t/\log t)}, t/\log t) \ .$$

In the succeeding thirty years, no significant improvements to the above results have been found. For example, the following questions remain open.

- Can the time bound in either Proposition 3 or 2 be improved, using the same space?
- Under what conditions is $\mathsf{TiSp}(t, s_1) \neq \mathsf{TiSp}(t, s_2)$?
- Under what conditions is $\mathsf{TiSp}(t_1, s) \neq \mathsf{TiSp}(t_2, s)$?
- What is the relationship of $\mathsf{TiSp}(t, r)$ to $\mathsf{TiSp}(u, s)$ for $t = o(u)$ and $s = o(r)$, allowing more time but less space?

We do not have absolute answers to these questions—such would be a major breakthrough. Instead, we consider possible new relationships among these questions, based on principles of recursion and self-reference. The main idea, which is specific to function classes and which we have not seen given clear focus in the literature, is as follows.

Suppose, for example, that one finds a clever algorithm to show that[3]

$$\mathsf{FTiSp}(2^{n/2\lg n}, n) \subseteq \mathsf{FTiSp}(2^{n/3\lg n}, n\lg n) \ ,$$

which is a speedup by a factor of $2^{n/6\lg n}$. By simply repeating the algorithm an appropriate number of times, the factor $2^{n/6\lg n}$ of speedup can be obtained for any initial time bound above $2^{n/2\lg n}$—e.g., $\mathsf{FTiSp}(2^{(n/2)+(n/6\lg n)}, n)$ is contained in $\mathsf{FTiSp}(2^{n/2}, n\lg n)$. We show, however, that a much greater speedup follows; Theorem 10 below implies that $\mathsf{FTiSp}(2^{(n/2)+(n/6\lg n)}, n)$ is contained in $\mathsf{FTiSp}(2^{n/3}, n\lg^2 n)$.

We obtain our speedups by applying the assumed speedup recursively. Consider a (reasonably efficient) universal machine U. Under our supposition, the following linear-space computable function is in

[3] Note that the first class contains many natural PSpace-complete problems: a formula, graph or similar structure encoded with n bits generally has only $n/c\log n$ distinct variables, nodes or the like. Also note that the second time bound is the 2/3 power of the first, achieved at only a logarithmic-factor cost in space—the supposition is by no means trivial.

$\mathsf{FTiSp}(2^{n/3\log n}, n\log n)$, where γ stands for an instantaneous description of an arbitrary Turing machine, as U would encode it.

$$I(\gamma, r, j) = \begin{cases} U^j(\gamma), & \text{if } j \leq 2^{|\gamma|/2\log|\gamma|} \text{ and for all } i,\ 0 \leq i \leq j, \\ & \qquad |U^i(\gamma)| \leq r \\ \perp & \text{otherwise.} \end{cases}$$

Our assumption therefore implies a machine S that can speed up $2^{n/2\log n}$ steps of U into $2^{n/3\log n}$ steps of its own, at the cost of using quasilinear space. The new idea is this: *apply S recursively to its own computations.* Although a recursive simulation may cost time and/or space, the major issue is that the original algorithm changed the "shape" of the computation. We began with a machine S_0 computing f with an extremely "long and narrow" time-space profile. Since the machine S has a less-extreme profile, we cannot simply re-apply the tradeoff assumption. We show, however, that the initial tradeoff assumption can be combined with carefully controlled recursion to give larger tradeoffs.

The effect of our improvements depends on the available depth of recursion. The example above has the minimal non-trivial recursion depth of 2, where it "tops out" at time 2^n. If one assumes a constant-factor speedup of linear time, the resulting recursion depth can be nearly n, and yields a quadratic speedup at exponential time: any language acceptable using cn bits of space becomes acceptable in $2^{(cn/2)+o(n)}$ time.

The paper is organized as follows. We give our conventions and definitions in the next section. In Section 3, we discuss standard padding techniques to translate hypothetical inclusions up to higher complexities. We show that, for simultaneous bounds, padding can be interleaved with repetition to produce greater speedups than padding alone. The main part, Section 4, adds the further element of recursion and applies speedups to sped-up computations. Section 5 gives some instances of the main theorem, and applications for upper time bounds on linear space problems that result from the initially assumed base tradeoffs.

Strictly speaking, the results of Section 3 are not needed for the main results in Section 4. Understanding the former, however, should assist understanding the latter. We end in the final section with some example special cases of the main theorem.

2 Conventions and Definitions

All of our Turing machines will use the same fixed input and work alphabet Σ, with at least two symbols in addition to the blank, and have a

read-only input tape plus some number of work tapes. If we us a function f in a time or space bound, we assume that $f(n)$ is monotonically non-decreasing and its binary representation $bin(f(n))$ is computable from $bin(n)$ in time polynomial in $\lg \max\{n, f(n)\}$ (the size of input plus output). We say that such a function is *strongly tisp-constructible*.[4]

Definition 4. *For functions t and s, $\mathsf{TiSp}^\dagger(t, s)$ is the set of languages decided by some multitape Turing machine that uses $t(n)(1 + o(1))$ steps and $s(n)(1 + o(1))$ work tape cells on each input of length n. If the machine has a read-only input tape, then the space excludes the input itself.*

$\mathsf{FTiSp}^\dagger(t, s)$ is the set of functions computable in the same time and space bounds, where the space bound does apply *to the output tape.*

Note that a machine using time $2t(n)$ may not be convertible to use time $t(n)(1 + o(1))$ with the same fixed alphabet, even when $t(n) \in \omega(n)$. However, $\mathsf{TiSp}^\dagger(O(t), O(s))$ is the same as $\mathsf{TiSp}(O(t), O(s)) = \mathsf{TiSp}(t, s)$.

We apply space bounds to the output for two reasons: first, so that two or more consecutive space-limited computations require only the sum of their individual times and the maximum of their space, and second, our recursive method requires space-limited output. As a pleasant effect of the convention, we need not distinguish between the output of a machine and its final configuration.

We assume that there is a universal machine U, such that for every machine M and string x, U on input $\langle M, x\rangle$ simulates the computation of M on input x. Further, there are a function $v : \mathbf{N} \to \mathbf{N}$ (independent of M and x) and constants c_M and d_M depending only on M such that to simulate t steps of M, which use space s, requires at most $c_M t \cdot v(s)$ steps of U and uses space at most $d_M s$, for all sufficiently long inputs.

We leave v incompletely specified to allow for easy translation to alternative machine models. For general multitape Turing machines, we can take $v(s) = O(\log s)$. Single-tape Turing machines require $v(s) = \Omega(s)$; however, Turing machines with a larger constant number of tapes can take $v(s) = O(1)$ [3]. The programming-language model of Jones [7, 2] also admits a universal constant in place of c_M and d_M.

The following technical lemma is immediate.

Lemma 5. *Suppose that some machine M accepts a language of pairs $(x, y) \in \Sigma^* \times \Sigma^*$ in time $t(|x, y|)$ and space $s(|x, y|)$. Then there is another machine*

[4] A weaker constructibility hypothesis would suffice. We do not explore the issue here.

M', with one extra work tape, such that (1) M' accepts from input x and initial extra-work-tape contents y if and only if M accepts the pair x, y, and (2) M' uses time $t(|x, y|)$ and space $s(|x, y|)$.

For a function f, we shall use the notation $f^{(k)}$ to denote the k-fold iteration of f. For a machine M, $M^{(k)}$ denotes the result of running M for k steps from the given configuration (that is, the k-fold iteration of the transition function applied to configurations).

Finally, we reserve the variable n to denote the length of an input or for the resource bound $\lambda n.n$. When no ambiguity arises, we shall abbreviate "$f(n)$" to "f", for any function f.

3 Basic Translations of Speedups

First we show that simple padding and repetition apply to time-space bounds.

Suppose that for some functions t and t', with $t < t'$, any computation in time t' can be replaced by a computation of time t, at the cost of increasing the space usage. Consequences we derive from such a speedup will involve the ratio t'/t; thus we let $h = t'/t$ and use ht in place of t'.

Lemma 6. *Suppose that for some functions h, t, s and $\hat{s}$,*

$$\mathsf{FTiSp}^{\dagger}(ht, \hat{s}) \subseteq \mathsf{FTiSp}^{\dagger}(t, s) \ .$$

Then for all strongly tisp-constructible functions $p(n) > n$,

$$\mathsf{FTiSp}^{\dagger}\big(h(p)t(p), \hat{s}(p)\big) \subseteq \mathsf{FTiSp}^{\dagger}\big(t(p), s(p)\big) \ .$$

Proof. If $s\big(p(n)\big) > n$, use standard padding, replacing input x by $x01^{p(|x|)-|x|-1}$. Otherwise, write the padding string $01^{p(|x|)-|x|-1}$, with end-markers, on a separate work tape, and apply the machine M' of Lemma 5.

If the original space is linear, a computation using more time than ht may be sped up by a factor h by repeating the sped-up version.

Lemma 7. *Suppose that for some functions t, h and s, where $s(n) \in \Omega(n)$,*

$$\mathsf{FTiSp}^{\dagger}(h(n)t(n), n) \subseteq \mathsf{FTiSp}^{\dagger}\big(t(n), s(n)\big) \ .$$

Then for all strongly tisp-constructible functions $p(n) \geq 1$ and $z(n) \geq 1$,

$$\mathsf{FTiSp}^{\dagger}\Big(z(n)h\big(p(n)\big)t\big(p(n)\big), p(n)\Big) \subseteq \mathsf{FTiSp}^{\dagger}\Big(z(n)t\big(p(n)\big), s\big(p(n)\big)\Big) \ .$$

Proof. For any Turing machine M, define

$$I_M(\gamma) = \begin{cases} M^{\left(h(|\gamma|)t(|\gamma|)\right)}(\gamma) & \text{if the computation uses space } |\gamma| \\ \perp & \text{otherwise.} \end{cases}$$

Clearly, I_M is in $\mathsf{FTiSp}^{\dagger}\big(h(n)t(n), n\big)$ and thus *ex hypothesi* in the class $\mathsf{FTiSp}^{\dagger}\big(t(n), s(n)\big)$.

Now suppose that machine M computes a function g in time zht and space s. The following algorithm also computes g.

On input x of length n:
  $\gamma \leftarrow x01^{p(|x|)-|x|-1}$
  `Repeat` $z(n)$ times:
    $\gamma \leftarrow I_M(\gamma)$
  `Return` γ

Using the assumed algorithm to compute I_M, the above uses time $p(n) + z(n) \cdot (t(p(n)) + O(1)) = z(n)t(p(n)(1 + o(1))$ and space $s(p(n))(1 + o(1))$, as required.

If t has polynomial growth rate (whence s does also), taking $p(n)$ to be a linear function and repeating the computation of the previous lemma an arbitrary constant number of times yields the following corollary, which connects our no-constant-factors classes with the usual constant-factors-don't-matter classes.

Corollary 8. *Let t have polynomial growth rate. If*

$$\mathsf{FTiSp}^{\dagger}(h(n)t(n), n) \subseteq \mathsf{FTiSp}\big(t(n), s(n)\big) \ ,$$

then

$$\mathsf{FTiSp}\big(h(n)t(n), n\big) \subseteq \mathsf{FTiSp}\big(t(n), s(n)\big) \ .$$

One can increase the speedup of Lemma 7 (at additional cost in space) by repeatedly applying it, with changing values of p and z. For example, first using $p = n$ and $z = h(s)t(s)/t(n)$ and then using $p = s$ and $z = 1$ yields that the containment $\mathsf{FTiSp}^{\dagger}(h(n)t(n), n) \subseteq \mathsf{FTiSp}^{\dagger}(t(n), s(n))$ implies the containments

$$\begin{aligned} \mathsf{FTiSp}^{\dagger}\left(h(n)h\big(s(n)\big)t\big(s(n)\big), n\right) &\subseteq \mathsf{FTiSp}^{\dagger}\left(h\big(s(n)\big)t\big(s(n)\big), s(n)\right) \\ &\subseteq \mathsf{FTiSp}^{\dagger}\left(t\big(s(n)\big), s^{(2)}(n)\right) \ . \end{aligned}$$

This process can continue for any constant number i of iterations. For the sake of brevity, let

$$P^i_{h,s}(n) =_{\text{def}} \prod_{j=0}^{i-1} h\left(s^{(j)}(n)\right)$$

and (for use later)

$$Q^i_{h,v,s}(n) =_{\text{def}} \prod_{j=0}^{i-1} \frac{h\big(s^{(j)}(n)\big)}{v\big(s^{(j+1)}(n)\big)} \ .$$

These quantities denote factors of time savings achievable by recursion and iteration from our ground assumption of a factor-of-$h(n)$ time savings. They show how the time-savings factor scales with the upward allowance in space granted by iterating $s(n)$.

Corollary 9. *If*

$$\mathsf{FTiSp}^\dagger\big(h(n)t(n), n\big) \subseteq \mathsf{FTiSp}^\dagger\big(t(n), s(n)\big) \ ,$$

then for each fixed $i \geq 0$,

$$\mathsf{FTiSp}^\dagger\Big(P^i_{h,s}(n)\, t\big(s^{(i)}(n)\big), n\Big) \subseteq \mathsf{FTiSp}^\dagger\Big(t\big(s^{(i)}(n)\big), s^{(i+1)}(n)\Big) \ .$$

Proof. For each j in $\{0, 1, 2, \ldots, i\}$ in succession, apply Lemma 7 with $p = s^{(j)}(n)$ and $z(n) = t\big(s^{(j)}(n)\big)P^j_{h,s}(n)/t(n)$.

The essence here is to progress from a time savings of a factor of $h(n)$ to one of $P^i_{h,s}(n)$, which not only raises the speedup factor h to the power i, but also converts the allowance of using more space via $s^{(j)}(n)$ at each stage $j < i$ into a savings of even more time. The main cost is that the minimal time in which this speedup is enjoyed is shifted upward from $t(n)$ to $t(s^{(i)}(n))$.

4 Increased Speedup Via Implicit Recursion

To extend the results of the previous section to a non-constant number of iterations, we need some form of uniformity. The hypothesis thus far, that for each M computing a function in $\mathsf{FTiSp}^\dagger(ht, n)$ there is an S_M computing the same function in $\mathsf{FTiSp}^\dagger(t, s)$, does not immediately imply that the mapping $M \mapsto S_M$ is computable. As an alternative to hypothesizing efficient computability of this mapping, we shall accept

some loss of time by utilizing speed-ups of a fixed machine that incorporates a universal machine.

Theorem 10. *Let $t(n)$ be a time function, $s(n)$ a space function, c a constant, and $i(n)$ a function such that $t\big(s^{(2)}(r)\big) < c\, t\big(s(n)\big)$ for some constant c and all sufficiently large n, and $s^{(i(n))}(n) = \omega(n)$. If*

$$\mathsf{FTiSp}^{\dagger}\left(h(n)t(n), n\right) \subseteq \mathsf{FTiSp}^{\dagger}\left(t(n), s(n)\right)) \quad ,$$

then

$$\mathsf{FTiSp}^{\dagger}\left(t\big(s^{(i(n)-1)}(n)\big)Q^{i(n)}_{h,v,s}(n)\,,\, n\right) \subseteq \mathsf{FTiSp}^{\dagger}\left(t\big(s^{(i(n)-1)}(n)\big),\, s^{(i(n))}(n)\right) \quad . \tag{1}$$

Proof. For the universal machine U, let

$$I_{ht}(\gamma, r, m) = \begin{cases} U^{(m)}(\gamma) & \text{if } m < h(r)t(r) \text{ and the computation} \\ & \quad \text{uses space at most } r, \\ U^{(h(r)t(r))}(\gamma) & \text{if } m \geq h(r)t(r) \text{ and the computation} \\ & \quad \text{uses space at most } r, \\ \bot & \text{otherwise.} \end{cases}$$

Since $I_{ht} \in \mathsf{FTiSp}^{\dagger}(h(n)t(n), n)$, by the hypothesis it is also in the class $\mathsf{FTiSp}^{\dagger}\big(t(n), s(n)\big)$, via some machine S_{ht}.

Consider the following algorithm B. Via implicit use of the Recursion Theorem, we will call it initially with $T = B$ itself.

B (γ, r, m, $\langle T \rangle$):
  `If` $r < |m|$ or $r < |\gamma|$
    `Return` $\bot$

  `While` $m > h(r)\, t\big(s(r)\big)$ [*If the next level of speedup is available, go to it.*]
    $\gamma \leftarrow \langle T, \gamma, r, m \rangle$
    $m \leftarrow mv\big(s(r)\big)/h(r)$
    $r \leftarrow s(r)$
  `EndWhile`

  `While` $m \geq h(r)t(r)$ [*Apply the best available speedup, as often as needed*]
    $m \leftarrow m - h(r)t(r)$
    $\gamma \leftarrow S_{ht}\big(\gamma, r, h(r)t(r)\big)$
  `EndWhile`

  `If` $m > t(r)$ [*Finish the computation*]

```
  Return S_ht(γ, r, m)
Else
  Return I_ht(γ, r, m)   [using U directly]
```

Lemma 10.1. *Let machine T_B implement algorithm B. Then for all γ, r and m, an invocation of $T_B(\gamma, r, m, \langle T_B \rangle)$ computes $I(\gamma, r, m)$.*

Proof (Proof of lemma). Consider an execution of the algorithm that computes the body of the first `While`-loop i times. We use induction on i. If $i = 0$, the claim follows immediately from the definitions of I and S_{ht}. Now suppose that the body of the first `While`-loop is executed. The condition that $t\big(s^{(2)}(r)\big) \leq c\, t\big(s(n)\big)$ implies that the ratio $h(r)t(s(r))/m$ decreases by at least a constant factor in each iteration; thus the loop eventually terminates. At each iteration, the value of $I(\gamma, r, m, \langle T \rangle)$ remains unchanged; thus the correct output results.

Let $\mu(n)$ be the time to multiply and divide numbers of n bits each.

Lemma 10.2. *Suppose that on inputs $\gamma, r, m, \langle T \rangle$, the first `While`-loop executes i times. The value of variable m at termination of the loop is $m_i = m \cdot \prod_{j=0}^{i-1} \big(v(s^{(j+1)})/h(s^{(j)})\big)$. The total run time is at most $m_i/h\left(s^{(i)}(r)\right) + O\left(\sum_{j=0}^{i-1} s^{(j)}(r)\right) + O(i\mu(n))$. If $m_i \leq t\left(s^{(i)}(r)\right)$, the space is $s^{(i)}(r)(1 + o(1)) + O(n)$; otherwise the space is $s^{(i+1)}(r)(1 + o(1)) + O(n)$.*

Proof (Proof of lemma). The value of m after i iterations is immediate.

Since all of the values used in the algorithm have at most n bits, the time exclusive of the calls to S_{ht} and to I is at most $O\left(\mu(n)\right) + O\left(\sum_{j=0}^{i-1} s^{(j)}(r)\right)$. The second `While`-loop and the final invocation of S_{ht} or U together use at most

$$\left\lceil \frac{m_i}{h\left(s^{(i)}(r)\right) t\left(s^{(i)}(r)\right)} \right\rceil \left(t\left(s^{(i)}(r)\right) + O(n) \right)$$

steps. These yield the required bound on time.

If $m_i \leq t\left(s^{(i)}(r)\right)$, the space used is $s^{(i)}(r)(1+o(1))+O(n)$; otherwise at least one call to S_{ht} occurs and the space is $s^{(i+1)}(r)(1 + o(1)) + O(n)$.

Thus the required bounds hold.

These two lemmas immediately yield the required containment of the theorem.

5 Applications to Linear Space

Our time-space tradeoffs become assertions about the relationship between space and time individually when the left-hand side of equation (1) equals the upper bound on time needed for a given initial usage in space. When this usage is n, the original time is $t_0(n) = 2^{cn}$ where the constant c depends on (the alphabet of) the initial machine. Different values of both the initial time-factor savings $h(n)$ *and* the base time $t(n)$ translate into different ultimate upper bounds on $t'(n)$ for which $\mathsf{Space}[n] \subseteq \mathsf{Time}[t'(n)]$ is obtained.

Alas, the dependence is difficult enough that we have no closed formula to express the exact value of this gain under general conditions, so we give some examples in the table of Figure 1. The first columns give the assumption: the base time $t(n)$, the speedup factor adjusted to reflect any value of $v(n)$, and the space used in the sped-up computation. The last two columns give $i(n)$, the maximum usable depth of recursive speedup, and the resulting amount of time required to accept all of $\mathsf{Space}(n)$, under the assumptions.

Base time $t(n)$	Initial speedup factor $h(n)/v(s(n))$	Initial space cost $s(n)$	Maximum iterations $i(n)$	Resulting time to accept all of $\mathsf{Space}(n)$
n	b	an	$\log_{ab}(2^n/n)$	$2^{n/2}$
n^2	n	n^2	$\lg(n/\lg n) - 1$	$2^{n/2}$
n^c	n^b	n^a	$\log_a\left(\frac{a(a-1)}{(ba+c(a-1))}\frac{n}{\lg n}\right)$	$2^{\frac{a(a-1)}{ba+c(a-1)}n}$
$2^{n/3\lg n}$	$2^{n/6\lg n}$	an	2	$2^{2n/3}$

Fig. 1. Examples of recursive speed-up

To illustrate, we state the first line of the table as a theorem. It shows the consequences of a constant-factor speedup (with no increase in the size of the tape alphabet or in the number of tapes) combined with a constant-factor increase of space. Recursively applying the speedup, as provided by Theorem 10, yields that all of $\mathsf{Space}(n)$ then can enjoy a square-root improvement of its time, if the same linear space is allowed.

Theorem 11. *If* $\mathsf{FTiSp}^\dagger(n\log n), n) \subseteq \mathsf{FTiSp}^\dagger(O(n), O(n))$, *then* $\mathsf{Space}[n] \subseteq \mathsf{Time}[2^{n/2}]$.

Recursively applying the speedup, as provided by Theorem 10, yields that all of $\mathsf{Space}(n)$ then can enjoy a square-root improvement of its time, if the same space is allowed.

6 Conclusions and Speculations

We have introduced a novel technique for improving given time-space tradeoffs starting from linear space. It often gives a greater factor of time improvement as compared to standard padding, at a cost of more space. Our applications in the last section are really about the nature of "long and skinny" computations of functions $f(x)$, typified by using n work-tape cells and taking 2^n steps, on inputs x of length n. It follows by counting that at some (indeed, many) timesteps j, the string y_j on the work-tape at step j has relative Kolmogorov complexity $KC(y_j|x) \geq n$. (For this and other notions of Kolmogorov complexity, see [**?**].) However, if $f(x)$ is computable in time $2^{n-g(n)}$ for non-constant $g(n)$, regardless of space and even allowing synchronous parallel computation such as by cellular automata, then *every* y_j stored has $KC(y_j|x) \leq n - g(n) + C$ (for some contant C depending only on the machine M involved) and so is non-random relative to x, since j itself plus M describe y_j from x. Thus all time-space tradeoffs involve reducing the potential randomness of storage.

We take this as reason to believe that "Tower-of-Hanoi"-type computations that snake through a large number of space-n IDs are necessary for some space-n computable functions $f(x)$. We have not been able to create a more-substantial theorem around this observation, however. A related observation is that if $KC(y_j|x) = O(\log n)$ for all j (where $n = |x|$), then $f(x)$ is computable in polynomial time, because there are only polynomially many such strings y_j (together with states of M and tape head positions), and a longer computation must be looping. It is curious that this observation doesn't care whether the "KC" is the standard uncomputable definition, or some highly resource-bounded restriction of Kolmogorov complexity able to unpack y_j from j. Overall, we hope that our results will bring more attention to means of recursively extracting structure from long space-bounded computations.

References

1. ADELMAN, L., AND LOUI, M. Space-bounded simulations of multitape Turing machines. *mst 14* (1981), 215–222.

2. BEN-AMRAM, A., AND JONES, N. Computational complexity via programming languages: Constant factors do matter. *Acta Informatica 37* (2000).
3. FÜRER, M. Data structures for distributed counting. *J. Comput. System Sci. 29* (1984), 231–243.
4. HARTMANIS, J., AND STEARNS, R. On the computational complexity of algorithms. *Trans. Amer. Math. Soc. 117* (1965), 285–306.
5. HENNIE, F., AND STEARNS, R. Two–way simulation of multitape Turing machines. *J. Assoc. Comp. Mach. 13* (1966), 533–546.
6. HOPCROFT, J., PAUL, W., AND VALIANT, L. On time versus space. *J. Assoc. Comp. Mach. 24* (1977), 332–337.
7. JONES, N. Constant factors do matter. In *Twenty-fifth Annual ACM Symposium on the Theory of Computing* (1993), pp. 602–611.
8. SAVITCH, W. Relationship between nondeterministic and deterministic tape complexities. *J. Comput. System Sci. 4* (1970), 177–192.
9. STEARNS, R., HARTMANIS, J., AND LEWIS II, P. Hierarchies of memory-limited computations. In *Proceedings, 6th Ann. IEEE Symp. Switching Circuit Theory and Logical Design* (1965), pp. 179–190.