An extension of the Chandra-Stockmeyer parallelization algorithm

Tomislav Petrović

April 12, 2015

Introduction

In 1976 Dexter Kozen in [2] and Ashok K. Chandra and Larry J. Stockmeyer in [1] simultaniously and independently introduced the notion of alternating Turing Machine and later published a joint version of the paper [3]. In [1] Chandra and Stockmeyer set forth their *parallel computation thesis,* a hypothesis that states that for any reasonable parallel computation model, time and space are polynomially related and provide an algorithm that can be used to transform a sequential computation that uses s bits of space into a parallel one that computes in number of time steps polynomial in s. In [4] Borodin shows a similar result for transforming sequential computation into boolean circuits. On the other hand, in [5], Greenlaw, Hoover and Ruzzo explore the limits to parallel computation and the inherently sequential problems. These are the P-complete problems, problems that have feasible (polynomial time) sequential solutions but are believed to have no feasible highly parallel solutions (parallel polylogarithmic time and polynomial space).

More recently, A. Waterland, E. Angelino, R. P. Adams, J. Appavoo, and M. Seltzer in [6] and A. Waterland, S. Homer, J. Appavoo, and M. Seltzer in [7] show that for some computations it is in practice possible to determine a small set containing a subset of states that the machine regularly visits while computing on a given an input.

We'll show a simple extension to the algorithm of Chandra and Stockmeyer [1] which parallelizes sequential computations that don't necessarily use a small amount of space but for which, on a given input, we can determine a small set containing a subset of states that the machine visits while computing the output and for which we can determine the time bound between two visits. We follow the approach from L. Levin's lecture notes [8], however, we do not use any particular model of computation for the description of the algorithm - we just write it in pseudo-code. Inadvertently, this means that all the claims will be supported with pseudo-proofs.

Original algorithm

We describe a version of the original Chandra and Stockmeyer algorithm which we'll use to parallelize a sequential machine that uses s bits of space.

1. Starting with a single thread, for each of the 2^s possible configurations of the machine generate a thread by simultaneously forking s times. A binary string of length s is at the same time a full description of the sequential machine's state and an address/pointer to a thread assigned to that configuration.

2. Each thread simulates a single step of the sequential machine on it's configuration to obtain the subsequent configuration and stores it as a local pointer to the successor thread/configuration. If the configuration is a halting state (no step to simulate) store a pointer to self.

3. On each thread, for s iterations:
Replace the pointer to the successor with the successor's pointer to the successor.

Assuming that forking takes constant time, first phase takes $O(s)$ time steps on our parallel machine. The second phase takes $O(1)$ time steps if we can simulate one step of the sequential machine in constant time. The third phase takes $O(s^2)$ time steps if we need $O(s)$ time steps to follow a pointer and copy a pointer. If following and copying a pointer takes constant time then third phase takes $O(s)$ time steps as well. In both cases, however, the whole procedure takes time polynomial in s and space exponential in s.

In the third phase, after each iteration i, a thread/configuration either points to a state that is 2^i steps away on a sequential machine or it points to a halting configuration. As there are at most 2^s different configurations, after end of third phase a configuration either has a pointer to the halting configuration or the sequential machine will never reach a halting state from this configuration. Now, given an initial configuration of the sequential machine, all we need is to follow it's successor pointer in order to determine whether the sequential machine will eventually halt, and if so, what is it's output.

An extension

Suppose that for some program and its input we can obtain a small, efficiently computable set of configurations S which we know contains configurations that the sequential machine will visit while running the program on its input. We don't know which configurations in S will be visited or in which order, but we know that the maximum time between two visits is tv_{max}. We'll denote the function computing the elements of S with s and have $s(0)$ be the initial configuration of the machine, given the program and its input.

1a. Starting from a single thread, for each of the $|S|$ configurations in S generate a thread by simultaneously forking $\lceil log(|S|) \rceil$ times. Each thread

has an index x associated with it which at the same time serves as an address/pointer to the thread.

1b. Each thread computes $s(x)$, the xth configuration in the set S and stores it in a binary tree with node corresponding to $s(x)$ having a pointer to x. This way we can compute inverse of function s and find an index corresponding to configuration $s \in S$ in $\ell(s)$ steps.

2. Each thread x simulates sequential machine starting from configuration $s(x)$ for at most tv_{max} steps, after each step checking if the new configuration s' is in S or if the simulated machine is in a halting state. If s' is in S, thread stores a local successor pointer to thread $s^{-1}(s')$ and if the simulated machine is in a halting state it stores locally a pointer to the configuration s'.

3. On each thread, for at most $\lceil log(|S|) \rceil$ iterations:
If you don't have a pointer to the halting configuration stored, follow the pointer to the successor.
If the successor has a pointer to a halting configuration, copy it,
otherwise replace the pointer to the successor with successor's pointer to the successor.

Phase 1a takes $O(\lceil log(|S|) \rceil)$ time provided that forking threads takes constant amount of time. Denote with ts_{max} the maximum time needed to compute a configuration from S by function s. Denote with ℓ_{max} the maximum length of a configuration in S. The time needed to store a configuration in the binary tree is linear in its length. We have that phase 1b takes $O(ts_{max} + \ell_{max})$ time.

Second phase takes at most tv_{max} iterations, in each iteration we perform a check which takes at most ℓ_{max} steps. We can check if the simulated machine is in a halting state in constant time, and if configuration is longer than ℓ_{max} we can conclude that it is not in S. The time needed to check if the configuration is in the binary tree is linear in its length, and we have that second phase takes $O(tv_{max} \cdot \ell_{max})$ time.

The third phase takes $O(log(|S|))$ time, if following and copying a pointer takes constant time.

If we assume that the function computing the configurations in S takes time linear in the length of the configuration then phase 1b is finished in $O(\ell_{max})$ time and the whole procedure is done in $O(log(|S|) + tv_{max} \cdot \ell_{max})$ parallel time steps.

Furthermore, in [6, 7] it is shown that in practice, for some computations we can check in constant time if configuration is in S. We can also compute s as the difference from the initial configuration in $O(log(|S|))$ time steps. And we can compute the inverse of s in $O(log(|S|))$ time by checking only $O(log(|S|))$ bits of the whole configuration. In these cases phase 1b can be reduced to $O(log(|S|))$ time as we don't need to store configurations for computing the inverse of s. Phase 2 can be computed in $O(tv_{max} + log(|S|))$ as we are absolved from checking if the state s' is in S for every simulated step of the sequential

machine, and we only compute the inverse once we know s' in S, this taking $log(|S|)$ time. For such computations we can find the output of the sequential machine in $O(log(|S|) + tv_{max})$ parallel time.

References

[1] A.K.Chandra and L.J.Stockmeyer. Alternation. FOCS-1976.

[2] D. Kozen. On parallelism in Turing Machines. FOCS-1976.

[3] Ashok K. Chandra, Dexter C. Kozen, Larry J. Stockmeyer. Alternation. J. ACM, 28(1):114-133, 1981.

[4] A. Borodin. On relating time and space to size and depth. SIAM Journal on Computing, 6(4):733–744, 1977.

[5] R. Greenlaw, H. J. Hoover, W. L. Ruzzo. Limits to Parallel Computation: P-completeness Theory, Oxford University Press Inc (United States), 1995.

[6] A. Waterland, E. Angelino, R. P. Adams, J. Appavoo, and M. Seltzer. ASC: Automatically Scalable Computation. Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 575–590.

[7] A. Waterland, S. Homer, J. Appavoo, and M. Seltzer. Storing Computation. Submitted.

[8] http://www.cs.bu.edu/fac/lnd/toc/z/node15.html