An Effective Load Balancing Policy for Geometric Decaying Algorithms *

JOSEPH GIL †
Dept. of Computer Science
The Technion, Israel
Technion City, Haifa 32000
ISRAEL

YOSSI MATIAS ‡
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
USA

Abstract

Parallel algorithms are often first designed as a sequence of rounds, where each round includes any number of independent constant time operations. This so-called work-time presentation is then followed by a processor scheduling implementation on a more concrete computational model. Many parallel algorithms are geometric-decaying in the sense that the sequence of work loads is upper bounded by a decreasing geometric series. A standard scheduling implementation of such algorithms consists of a repeated application of load balancing. We present a more effective, yet as simple, policy for the utilization of load balancing in geometric decaying algorithms. By making a more careful choice of when and how often load balancing should be employed, and by using a simple amortization argument, we show that the number of required applications of load balancing should be nearly-constant. The policy is not restricted to any particular model of parallel computation, and, up to a constant factor, it is the best possible.

Keywords: Design of algorithms, analysis of algorithms, parallel algorithms, load balancing.

*To appear in *Journal of Parallel and Distributed Computing*. A preliminary version of this paper was part of a paper presented in the *Second Annual ACM-SIAM Symposium on Discrete Algorithms*, 1991 [7].

†Part of research was done while author was at the Hebrew University and at the University of British Columbia. E-mail address: `yogi@cs.technion.ac.il`.

‡Part of research was done while author was at Tel Aviv University and at the University of Maryland Institute for Advanced Computer Studies, and was partially supported by NSF grants CCR-9111348 and CCR-8906949. E-mail address: `matias@research.att.com`.

1 Introduction

Many parallel algorithms are designed using the well known work-time methodology. (See [13, Chapter 1.3] for a detailed discussion and [19] for an early use in a PRAM algorithm.) Guided by Brent's theorem [3], the methodology suggests to first describe a meta-algorithm in terms of a sequence of rounds; each round may include any number of independent constant time operations. Second, the meta-algorithm is implemented on a p-processor parallel computer, e.g., a PRAM, using a scheduling principle: if the number of operations in round r is w_r, then each of the p processors should execute a set of $O(w_r/p)$ operations. Let $w = \sum_r w_r$ be the total number of operations, or *work*, of the meta-algorithm. Let t be the total number of rounds, or *time*, of the meta-algorithm. Then, using Brent's scheduling principle, the running time on a p-processor parallel computer should be $O(\sum_{r=1}^{t} \lceil w_r/p \rceil)$ which is $O(w/p + t)$.

The work-time presentation is often a simple and natural way to express a parallel algorithm, and forms the basis of many data parallel languages. Unfortunately, Brent's scheduling principle for adapting an algorithm given in the work-time presentation into a concrete parallel algorithm is only a guideline, rather than a constructive technique. Traditionally, its implementation is done in an ad-hoc manner, resulting with additional effort in the design and analysis of algorithms, as well as occasional degradation in performance. It is therefore desirable to have general scheduling techniques that can be easily and effectively used for adapting algorithms that are given in the work-time presentation into concrete algorithms.

In this note we consider a somewhat restricted, yet quite common, family of algorithms, the so-called *geometric-decaying* algorithms, for which simple and efficient scheduling techniques apply; these techniques are based on repeatedly employing a load balancing algorithm as a key procedure. More specifically, the scheduling in geometric-decaying algorithms is traditionally done by a straightforward application of a load balancing procedure after each round. Quite a few parallel algorithms are of this type; we note that many logarithmic-time linear work algorithms are geometric-decaying. See, e.g., algorithms described in [14, 13, 18].

The time devoted to the applications of the load balancing algorithms contributes to the total running time of the adapted algorithm, and should therefore be minimized. We show that the standard policy of employing load balancing for a geometric-decaying algorithm is somewhat wasteful, and provide an alternative, more effective, policy. An appealing ingredient of the reported technique is that no extra overhead is introduced into the algorithm; we only focus on the question of how often load balancing should be employed to get the most effective performance. The implementation of this policy does not require more than keeping track of the number of rounds and computing simple functions to determine the rounds in which load balancing should be employed. For the purpose of concreteness, the reader may wish to regard the ensuing discussion in the context of the PRAM model (see [13]), although it is applicable to other models of parallel computation as well.

Load balancing Let m independent tasks be distributed among n processors of a PRAM. The input to a processor P_i consists of m_i, the number of tasks allocated to this processor (its "load"), together with a pointer to an array of task representations: no other information about the global load distribution is available. The *load balancing* problem is to redistribute the tasks among the processors such that each processor has at most $O(1 + m/n)$ tasks. In other words, the loads are distributed asymptotically in an even manner, except of course for the case $m \ll n$.

The load balancing problem can clearly be solved by using an algorithm for the more difficult prefix-sum problem. Therefore, load balancing can be solved on circuits and on the EREW PRAM in $O(\lg n)$ time [15], and on the CRCW PRAM in $O(\lg n / \lg\lg n)$ time [4]. Specialized algorithms for load balancing run on the CRCW PRAM in $O((\lg\lg n)^3)$ time [11], and in $O(\lg^* n)$ time[1] with high probability [9] (also on the COMMON CRCW [1]). On the ROBUST CRCW PRAM [2, 12] load balancing can be solved in $O(\lg\lg n)$ time with high probability [6]. All the above algorithms require linear work.

[1]Let $\lg^{(i)} x \equiv \lg(\lg^{(i-1)} x)$ for $i > 1$, and $\lg^{(1)} x \equiv \lg x$; $\lg^* x \equiv \min\{i : \lg^{(i)} x \leq 2\}$. The function $\lg^*(\cdot)$ is extremely slow increasing and for instance $\lg^* 2^{65536} = 5$.

2 Geometric Decaying Algorithms

Assume that an algorithm is restricted as follows: a task which executes in round r can spawn at most one task and only for round $r+1$. (This assumption makes the algorithm susceptible to load balancing.) In particular, this guarantees that the number of tasks will not increase as the algorithm proceeds. Further assume that the number of tasks in round r is upper bounded by a geometrically decreasing sequence, i.e.,

$$\forall r \geq 0 \qquad w_r \leq n \cdot 2^{-\alpha r} \tag{1}$$

for some constant $\alpha > 0$, and where n is initial number of tasks. We call these algorithms *geometric-decaying* algorithms.

For randomized algorithms, we say that the algorithm is geometric-decaying if (1) holds with positive constant probability, that is to say, with a non zero probability which is not dependent on the problem size.

There is a weaker characterization for geometric-decaying algorithms. Suppose that at each round r, each task has a positive constant probability to terminate, provided that the number of tasks is at most $2^{-\alpha(r-1)}$ (which would be the case if in the first $r-1$ rounds the algorithm was geometric-decaying). Then, the algorithm can be shown to be geometric-decaying [8].

Consider a geometric decaying algorithm $\mathcal{A}$ whose work is $O(n)$ and running time is τ. Brent's scheduling principle suggests that it may be possible to implement $\mathcal{A}$ on a p-processor PRAM in time

$$O(n/p + \tau) \ . \tag{2}$$

If a certain implementation runs in time $O(n/p + \tau + T)$ we say that it has an *additive overhead* of T. We will typically be interested in the case that the additive overhead dominates the running time. When appropriate, we will therefore tacitly assume that $T \geq n/p + \tau$.

Devising general implementation policies of applying Brent's principle has become lucrative with the advent of very fast load balancing algorithms. Let ℓ be an upper bound on the running time of a given load balancing algorithm. (For randomized algorithms ℓ may be a random variable.) We investigate policies for the common case in which load balancing is no slower than the algorithm itself, and therefore assume that

$$\ell = O(\tau) \ . \tag{3}$$

Usually, the additive overhead is predominantly caused by the applications of a load balancing algorithm. An important measure of a general scheduling scheme is therefore the ratio

$$T/\ell \ . \tag{4}$$

We call this ratio the *multiplicative overhead* of the implementation, or the *overhead* in short. It is important to note that with the assumption (3) the overhead is also an upper bound for T/τ, the slowdown incurred by a concrete implementation.

A standard wasteful implementation

In the standard implementation, the load balancing algorithm is employed after each round, until the number of tasks becomes $O(p)$. At this point, the remainder of Algorithm $\mathcal{A}$ can be easily implemented in $O(\tau)$ time since no further scheduling is required.

Theorem 1 *The overhead in the standard implementation is* $O(\lg(\ell))$.

Proof. Since Algorithm $\mathcal{A}$ is geometric-decaying, the number of rounds required to reduce the number of tasks to $O(p)$ is $O(\lg(n/p))$. At round r each processor has $O(n \cdot 2^{-\alpha r})$ tasks; every round is therefore implemented in $O(n/p)$ time. Following from the assumption (3), we have that in the standard implementation the additive overhead is dominated by the applications of the load balancing algorithm. Since there are $O(\lg(n/p))$ such applications, we have

$$T = O(\ell \lg(n/p)) \ . \tag{5}$$

We note, however, that if the additive overhead dominates the term n/p, then we can substitute the multiplicative overhead $\lg(n/p)$ with $\lg \ell$. ∎

3 A New Effective Policy

Our objective is to reduce the overhead in the implementation. This would be obtained if we could reduce the number of times that a load balancing algorithm is employed while taking care not to incur too much of a slowdown as a result of the imbalance in the number of tasks.

To motivate our improved policy we note that after employing a $\Theta(\ell)$ time load balancing algorithm, we may have a phase in which the processors execute for $O(\ell)$ time without load balancing. Their potential wasteful execution can be amortized against the load balancing application that preceded this phase. In other words, it is "justified" to employ a new ℓ time load balancing algorithm only after each processor has run for $\Theta(\ell)$ time, following the previous load balancing.

Assume that after employing a load balancing procedure, each processor has at most y tasks. Then, by the above argument, the next load balancing procedure should be employed only after $\lceil \ell/y \rceil$ rounds since each round of this phase is implemented in $O(y)$ time. Accordingly, as long as $y \geq \ell$, load balancing is employed after each round, as in the standard policy. In this initialization phase, the load balancing application at each round is dominated by the execution time of the round. Since this is a geometric decaying algorithm, the total execution time in this initialization phase is $O(n/p)$, and therefore it does not contribute to the additive overhead. At the end of this phase each processor has at most ℓ tasks.

After the initialization phase, the algorithm runs in phases; phase i starts with an ℓ time load balancing, followed by t_i rounds. Let x_i be an upper bound on the number of tasks that each processor is responsible for in phase i ($x_0 = \ell$). The execution of each round in phase i takes $O(x_i)$ time, and the total execution time of the phase is thus $O(x_i t_i)$. We set t_i to the largest possible value so that the total execution time of phase i is $O(\ell)$:

$$t_i = \ell / x_i \ . \tag{6}$$

The upper bound on the number of tasks per processor in phase $i+1$ is

$$x_{i+1} = x_i / 2^{\alpha t_i} \tag{7}$$

and hence, by (6),

$$t_{i+1} = t_i 2^{\alpha t_i} \ . \tag{8}$$

For some $i = \lg^* \ell + O(1)$, $t_i = \ell$ and $x_i = O(1)$. From this point no further load balancing is required to complete the execution of the algorithm in time τ. The number of phases is thus $O(\lg^* \ell)$. Recall that the execution time of each processor in each phase is $O(\ell)$. This time can be amortized against the load balancing application at the beginning of the phase. The additive overhead is only determined by the applications of load balancing. The number of such applications is as the number of phases, implying an overhead of $O(\lg^* \ell)$,

We can therefore conclude:

Theorem 2 *The overhead of the improved load balancing policy is $O(\lg^* \ell)$. The additive overhead is $O(\ell \lg^* \ell)$.*

It follows that in most interesting settings, the improved policy leads to an implementation that is slower by a nearly constant factor than the idealistic prediction of Brent's principle.

On the optimality of the improved policy

The additive overhead in the policy presented here is $\omega(\ell)$. A natural question is therefore if this policy is the best possible (although an overhead of $O(\lg^* \ell)$ does not leave much room for improvement). Following the motivation given above, we may want to consider setting t_i to satisfy

$$\ell \ll t_i x_i \ll \ell \lg^* \ell \tag{9}$$

in an attempt to achieve an $o(\lg^* \ell)$ overhead. Suppose that t_i is set to satisfy $t_i x_i = \ell \lg^* \ell$, and let us ignore the extra execution time imposed on the processors due to imbalance. Then, (6) is replaced by

$$t_i = \ell \lg^* \ell / x_i$$

but (7) and (8) do not change. It is easy to verify that the overhead is still $\Theta(\lg^* \ell)$.

Moreover, it is easy to see that the performance of the above policy (when ignoring the extra execution due to imbalance as above) can match the performance of any other policy of load balancing utilization, up to a constant factor. The policy presented in this paper is therefore optimal.

4 Conclusions

This paper introduces a simple effective policy for employing load balancing algorithms for processor scheduling in geometric-decaying algorithms. The resulting implementation is as simple as the standard implementation with the only difference being a more careful account of when and how often load balancing should be used. The load balancing algorithm is used as a black box, and the suggested policy is thus applicable to many models of parallel computation.

We defined the overhead of policies of invoking load balancing to implement Brent's scheduling principle. The overhead can be used to evaluate and compare such policies. It was shown that the overhead for our policy is $O(\lg^* \ell)$, where ℓ is the run time of the load balancing algorithm in use. For most models this would imply an overhead of $O(\lg^* n)$ time. For the randomized CRCW PRAM, this would imply an overhead of $O(\lg^*(\lg^* n))$ time.

As a result, the overwhole additive overhead for processor scheduling in geometric-decaying algorithm, implied by the suggested policy, is $O(\lg n \lg^* n)$ time on circuits and the EREW PRAM, $O((\lg \lg n)^3 \lg^* n)$ time on the CRCW PRAM, and $O(\lg^* n \lg^*(\lg^* n))$ time on the randomized CRCW PRAM.

Many parallel algorithms in their work-time presentation are geometric-decaying. Our technique enables a simple cost-effective implementation with little effort. It was used for the first time to implement a fast optimal parallel hashing algorithm [7]. The hashing algorithm in [7] comprises two parts: the first part is a randomized geometric decaying algorithm which runs for $O(\lg\lg n)$ steps. By using the technique of this paper and the $O(\lg\lg n)$ time load balancing of [6], this part was implemented optimally with expected additive overhead of $O(\lg\lg n \lg^* n)$ time, which implies an work-optimal implementation that takes $O(\lg\lg n \lg^* n)$ time, i.e., using $p = n/\lg\lg n \lg^* n$ processors. The second part runs in $O(\lg\lg n)$ time using p processors, implying a work-optimal implementation for the entire algorithm that takes $O(\lg\lg n \lg^* n)$ expected time.

Subsequent to a preliminary version of this paper (presented in [7]), other policies of effective load balancing for other classes of algorithms were introduced, often using amortization arguments similar to the one used here [17, 9, 10, 5] (see also [16]).

References

[1] O. Berkman, P.B. Gibbons, and Y. Matias. On the power of randomization for the Common PRAM. In *Proc. 4th Israel Symp. on Theory of Comp. and Sys.*, pages 229–240, January 1995.

[2] A. Borodin, J.E. Hopcroft, M.S. Paterson, W.L. Ruzzo, and M. Tompa. Observations concerning synchronous parallel models of computation. Manuscript, 1980.

[3] R.P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21:201–208, 1974.

[4] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81:334–352, 1989.

[5] P.B. Gibbons, Y. Matias, and V. Ramachandran. Efficient low-contention parallel algorithms. In *6th ACM Symp. on Parallel Algorithms and Architectures*, pages 236–247, June 1994.

[6] J. Gil. Renaming and dispersing: Techniques for fast load balancing. *Journal of Parallel and Distributed Computing*, 23(2):149–157, November 1994.

[7] J. Gil and Y. Matias. Fast hashing on a PRAM—designing by expectation. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 271–280, January 1991.

[8] J. Gil and Y. Matias. Designing algorithms by expectations. *Information Processing Letters*, 51(1):31–34, 1994.

[9] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 698–710, October 1991.

[10] M.T. Goodrich. Using approximation algorithms to design parallel algorithms that may ignore processor allocation. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 711–722, 1991.

[11] T. Hagerup. Fast deterministic processor allocation. In *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, pages 1–10, 1993.

[12] T. Hagerup and T. Radzik. Every robust CRCW PRAM can efficiently simulate a Priority PRAM. In *2nd ACM Symp. on Parallel Algorithms and Architectures*, pages 117–124, 1990.

[13] J. JáJá. *Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, Inc., 1992.

[14] R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 869–941. North-Holland, Amsterdam, 1990.

[15] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, 1980.

[16] Y. Matias. *Highly Parallel Randomized Algorithmics*. PhD thesis, Tel Aviv University, Tel Aviv 69978, Israel, December 1992.

[17] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time—with applications to parallel hashing. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 307–316, May 1991.

[18] J.H. Reif, editor. *A Synthesis of Parallel Algorithms*. Morgan-Kaufmann, San Mateo, CA, 1993.

[19] Y. Shiloach and U. Vishkin. An $O(n^2 \lg n)$ parallel Max-Flow algorithm. *Journal of Algorithms*, 3:128–146, 1982.