Information Processing Letters 116 (2016) 100–106

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

On the efficiency of localized work stealing

Warut Suksompong [a,*], Charles E. Leiserson [b], Tao B. Schardl [b]

[a] *Department of Computer Science, Stanford University, 353 Serra Mall, Stanford, CA 94305, USA*
[b] *MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St, Cambridge, MA 02139, USA*

ARTICLE INFO

Article history:
Received 12 May 2015
Received in revised form 1 October 2015
Accepted 2 October 2015
Available online 8 October 2015
Communicated by M. Yamashita

Keywords:
Parallel algorithms
Multithreaded computation
Work stealing
Localization

ABSTRACT

This paper investigates a variant of the work-stealing algorithm that we call the *localized work-stealing algorithm*. The intuition behind this variant is that because of locality, processors can benefit from working on their own work. Consequently, when a processor is free, it makes a steal attempt to get back its own work. We call this type of steal a *steal-back*. We show that the expected running time of the algorithm is $T_1/P + O(T_\infty P)$, and that under the "even distribution of free agents assumption", the expected running time of the algorithm is $T_1/P + O(T_\infty \lg P)$. In addition, we obtain another running-time bound based on ratios between the sizes of serial tasks in the computation. If M denotes the maximum ratio between the largest and the smallest serial tasks of a processor after removing a total of $O(P)$ serial tasks across all processors from consideration, then the expected running time of the algorithm is $T_1/P + O(T_\infty M)$.

1. Introduction

Work stealing is an efficient and popular paradigm for scheduling multithreaded computations. While its practical benefits have been known for decades [4,8] and several researchers have found applications of the paradigm [2,5,9,10], Blumofe and Leiserson [3] were the first to give a theoretical analysis of work stealing. Their scheduler executes a fully strict (i.e., well-structured) multithreaded computations on P processors within an expected time of $T_1/P + O(T_\infty)$, where T_1 is the minimum serial execution time of the multithreaded computation (the *work* of the computation) and T_∞ is the minimum execution time with an infinite number of processors (the *span* of the computation.)

In multithreaded computations, it sometimes occurs that a processor performs some computations and stores the results in its cache. Therefore, a work-stealing algorithm could potentially benefit from exploiting locality, i.e., having processors work on their own work as much as possible. Indeed, an experiment by Acar et al. [1] demonstrates that exploiting locality can improve the performance of the work-stealing algorithm by up to 80%. Similarly, Guo et al. [6] found that locality-aware scheduling can achieve up to 2.6× speedup over locality-oblivious scheduling. In addition, work-stealing strategies that exploit locality have been proposed. Hierarchical work stealing, considered by Min et al. [11] and Quintin and Wagner [12], contains mechanisms that find the nearest victim thread to preserve locality and determine the amount of work to steal based on the locality of the victim thread. More recently, Paudel et al. [13] explored a selection of tasks based on the application-level task locality rather than hardware memory topology.

In this paper, we investigate a variant of the work-stealing algorithm that we call the *localized work-stealing algorithm*. In the localized work-stealing algorithm, when a processor is free, it makes a steal attempt to get back its own work. We call this type of steal a *steal-back*. We show that the expected running time of the algorithm is $T_1/P + O(T_\infty P)$, and that under the "even distribution

* Corresponding author.
E-mail addresses: warut@cs.stanford.edu (W. Suksompong), cel@mit.edu (C.E. Leiserson), neboat@mit.edu (T.B. Schardl).

http://dx.doi.org/10.1016/j.ipl.2015.10.002
0020-0190/© 2015 Elsevier B.V. All rights reserved.

of free agents assumption", the expected running time of the algorithm is $T_1/P + O(T_\infty \lg P)$. In addition, we obtain another running-time bound based on ratios between the sizes of serial tasks in the computation. If M denotes the maximum ratio between the largest and the smallest serial tasks of a processor after removing a total of $O(P)$ serial tasks across all processors from consideration, then the expected running time of the algorithm is $T_1/P + O(T_\infty M)$.

This paper is organized as follows. Section 2 introduces the setting that we consider throughout the paper. Section 3 analyzes the localized work-stealing algorithm using the delay-sequence argument. Section 4 analyzes the algorithm using amortization arguments. Section 5 considers variants of the localized work-stealing algorithm. Finally, Section 6 concludes and suggests directions for future work.

2. Localized work-stealing algorithm

Consider a setting with P processors. Each processor owns some pieces of work, which we call *serial tasks*. Each serial task takes a positive integer amount of time to complete, which we define as the *size* of the serial task. We assume that different serial tasks can be done in parallel and model the work of each processor as a binary tree whose leaves are the serial tasks of that processor. The trees are balanced in terms of the number of serial tasks on each branch, but the order in which the tasks occur in the binary tree is assumed to be given to us. We then connect the P roots as a binary tree of height $\lg P$, so that we obtain a larger binary tree whose leaves are the serial tasks of all processors.

As usual, we define T_1 as the work of the computation, and T_∞ as the span of the computation. The span T_∞ corresponds to the height of the aforementioned larger binary tree plus the size of the largest serial task. In addition, we define T'_∞ as the height of the tree not including the part connecting the P processors of height $\lg P$ at the top or the serial tasks at the bottom. Since T'_∞ corresponds to a smaller part of the tree than T_∞, we have $T'_\infty < T_\infty$.

The randomized work-stealing algorithm [3] suggests that whenever a processor is free, it should "steal" randomly from a processor that still has work left to do. In our model, stealing means taking away one of the two main branches of the tree corresponding to a particular processor, in particular, the branch that the processor is not working on. The randomized work-stealing algorithm performs $O(P(T_\infty + \lg(1/\epsilon)))$ steal attempts with probability at least $1 - \epsilon$, and the execution time is $T_1/P + O(T_\infty + \lg P + \lg(1/\epsilon))$ with probability at least $1 - \epsilon$.

This paper investigates a localized variant of the work-stealing algorithm. In this variant, whenever a processor is free, it first checks whether some other processors are working on its work. If so, it "steals back" randomly only from these processors. Otherwise, it steals randomly as usual. We call the two types of steal a *general steal* and a *steal-back*. The intuition behind this variant is that sometimes a processor performs some computations and stores the results in its cache. Therefore, a work-stealing algorithm could potentially benefit from exploiting locality, i.e., having processors work on their own work as much as possible.

We make a simplifying assumption that each processor maintains a list of the other processors that are working on its work. When a general steal occurs, the stealer adds its name to the list of the owner of the serial task that it has just stolen (not necessarily the same as the processor from which it has just stolen.) For example, if processor P_1 steals a serial task owned by processor P_2 from processor P_3, then P_1 adds its name to the P_2's list (and not P_3's list.) When a steal-back is unsuccessful, the owner removes the name of the target processor from its list, since the target processor has finished the owner's work.

An example of an execution of localized work-stealing algorithm can be found in [14]. We assume that the overhead for maintaining the list and dealing with contention for steal-backs is constant. This assumption is reasonable because adding (and later removing) the name of a processor to a list is done when a general steal occurs, and hence can be amortized with general steals. Randomizing a processor from the list to steal back from takes constant time. When multiple processors attempt to steal back from the same processor simultaneously, we allow an arbitrary processor to succeed and the remaining processors to fail, and hence do not require extra processing time.

3. Delay-sequence argument

In this section, we apply the delay-sequence argument to establish an upper bound on the running time of the localized work-stealing algorithm. The delay-sequence argument is used in [3] to show that the randomized work-stealing algorithm performs $O(P(T_\infty + \lg(1/\epsilon)))$ steal attempts with probability at least $1 - \epsilon$. We show that under the "even distribution of free agents assumption", the expected running time of the algorithm is $T_1/P + O(T_\infty \lg P)$. We also show a weaker bound that without the assumption, the expected running time of the algorithm is $T_1/P + O(T_\infty P)$.

Since the amount of work done in a computation is always given by T_1, independent of the sequence of steals, we focus on estimating the number of steals. We start with the following definition.

Definition 1. The *even distribution of free agents assumption* is the assumption that when there are k *owners* left (and thus $P - k$ *free agents*), the $P - k$ free agents are evenly distributed working on the work of the k owners. That is, each owner has P/k processors working on its work.

While this assumption might not hold in the localized work-stealing algorithm as presented here, it is intuitively more likely to hold under the hashing modification presented in Section 5. When the assumption does not hold, we obtain a weaker bound as given in Theorem 4.

Before we begin the proof of our theorem, we briefly summarize the delay-sequence argument as used by Blumofe and Leiserson [3]. The intuition behind the delay-sequence argument is that in a random process in which multiple paths of the process occur simultaneously, such as work stealing, there exists some path that finishes last.