Bounds on Multithreaded Computations by Work Stealing

by

Warut Suksompong

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology
June 2014

Signature redacted

Author .....

Department of Electrical Engineering and Computer Science
May 23, 2014

Certified by ..... Signature redacted

Charles E. Leiserson, Professor of Computer Science and Engineering
Thesis Supervisor
May 23, 2014

Signature redacted

Accepted by .....

Prof. Albert R. Meyer, Chairman, Masters of Engineering Thesis Committee

Bounds on Multithreaded Computations by Work Stealing

by

Warut Suksompong

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2014, in partial fulfillment of the requirements
for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Blumofe and Leiserson [6] gave the first provably good work-stealing work scheduler for multithreaded computations with dependencies. Their scheduler executes a fully strict (i.e., well-structured) computation on P processors in expected time $T_1/P + O(T_\infty)$, where T_1 denotes the minimum serial execution time of the multithreaded computation, and T_∞ denotes the minimum execution time with an infinite number of processors.

This thesis extends the existing literature in two directions. Firstly, we analyze the number of successful steals in multithreaded computations. The existing literature has dealt with the number of steal attempts without distinguishing between successful and unsuccessful steals. While that approach leads to a fruitful probabilistic analysis, it does not yield an interesting result for a worst-case analysis. We obtain tight upper bounds on the number of successful steals when the computation can be modeled by a computation tree. In particular, if the computation starts with a complete k-ary tree of height h, the maximum number of successful steals is $\sum_{i=1}^{n}(k-1)^i\binom{h}{i}$.

Secondly, we investigate 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 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 $T_1/P+O(T_\infty M)$.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Acknowledgements

First and foremost, I would like to thank my thesis supervisor, Prof. Charles E. Leiserson. Charles is a constant source of great advice, and his guidance has been instrumental in shaping this thesis. It has truly been a pleasure working with him.

I would also like to thank Tao B. Schardl. Tao provided helpful comments for most of the work in this thesis, and his positive attitude to things has been a source of inspiration for me. I never found it boring to work on research problems with Tao.

In addition, I would like to thank MIT for offering a variety of resources and serving as a world in which I can learn best.

Thanks also to my friends, near and far, for helping me with countless things throughout my life.

Last but not least, I would like to thank my family for their unconditional love and support.

Contents

Chapter 1

Introduction

Scheduling multithreaded computations is a complicated task, with a number of heuristics that the scheduling algorithm should follow. Firstly, it should keep enough threads active at any given time so that processor resources are not unnecessarily wasted. At the same time, it should maintain the number of active threads at a given time within reasonable limits so as to fit memory requirements. In addition, the algorithm should attempt to maintain related threads on the same processor so that communication across processors is kept to a minimum. Of course, these heuristics can be hard to satisfy simultaneously.

The two main scheduling paradigms that are commonly used for scheduling multithreaded computations are *work sharing* and *work stealing*. The two paradigms differ in how the threads are distributed. In work sharing, the scheduler migrates new threads to other processors so that underutilized processors have more work to do. In work stealing, on the other hand, underutilized processors attempt to "steal" threads from other processors. Intuitively, the migration of threads occurs more often in work sharing than in work stealing, since no thread migration occurs in work stealing when all processors have work to do, whereas a work-sharing scheduler always migrates threads regardless of the current utilization of the processors.

The idea of work stealing has been around at least as far back as the 1980s. Burton and Sleep's work [8] on functional programs on a virtual tree of processors and Halstead's work [16] on the implementation of Multilisp are among the first to outline the idea of work stealing. These

authors point out the heuristic benefits of the work-stealing paradigm with regard to space and communication.

Since then, many variants of the work-stealing strategy has been implemented. Finkel and Manber [13] outlined a general-purpose package that allows a wide range of applications to be implemented on a multicomputer. Their underlying distributed algorithm divides the problem into subproblems and dynamically allocates them to machines. Vandevoorde and Roberts [22] described a paradigm in which much of the work related to task division can be postponed until a new worker actually undertakes the subtask and avoided altogether if the original worker ends up executing the subtask serially. Mohr et al. [19] focused on combining parallel tasks dynamically at runtime to ensure that the implementation can exploit the granularity efficiently. Feldmann et al. [12] and Kuszmaul [18] applied the work-stealing strategy to a chess program, and Halbherr et al. [15] presented a multithreaded parallel programming package based on threads in explicit continuation-passing style. Several other researchers [2, 3, 4, 5, 7, 9, 11, 17, 20, 23] have found applications of the work-stealing paradigm or analyzed the paradigm in new ways.

An important contribution to the literature of work stealing was made by Blumofe and Leiserson [6], who gave the first provably good work-stealing scheduler for multithreaded computations with dependencies. 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.) Furthermore, the scheduler has provably good bounds on total space and total communication in any execution.

This thesis extends the existing literature in two directions. Firstly, we analyze the number of successful steals in multithreaded computations. The existing literature has dealt with the number of steal attempts without distinguishing between successful and unsuccessful steals. While that approach leads to a fruitful probabilistic analysis, it does not yield an interesting result for a worst-case analysis. We obtain tight upper bounds on the number of successful steals when the computation can be modeled by a computation tree. In particular, if the computation starts with

a complete k-ary tree of height h, the maximum number of successful steals is $\sum_{i=1}^{n}(k-1)^i\binom{h}{i}$.

Secondly, we investigate 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 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 $T_1/P+O(T_\infty M)$.

The remainder of this thesis is organized as follows. Chapter 2 introduces some definitions and notations that we will use later in the thesis. Chapter 3 explores upper bounds on the number of successful steals in multithreaded computations. Chapter 4 provides an analysis of the localized work-stealing algorithm. Finally, Chapter 5 concludes and suggests directions for future work.

Chapter 2

Definitions and Notations

This chapter introduces some definitions and notations that we will use later in the thesis.

Binomial Coefficients

For positive integers n and k, the binomial coefficient $\binom{n}{k}$ is defined as $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ if $n \geq k$, and $\binom{n}{k} = 0$ if $n < k$. A usual interpretation is that $\binom{n}{k}$ gives the number of ways to choose k objects from n distinct objects. This interpretation is consistent when $n < k$, since there is indeed no way to choose k object from n distinct objects in that case.

The binomial coefficients satisfy Pascal's identity [10]

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \tag{2.1}$$

for all integers $n, k \geq 1$. One can verify the identity directly using the definition.

Trees

Let *EMPT* denote the empty tree with no nodes, and let *TRIVT* denote the trivial tree with only one node.

For $h \geq 0$, let $CBT(h)$ denote the complete binary tree with height h. For instance, the tree

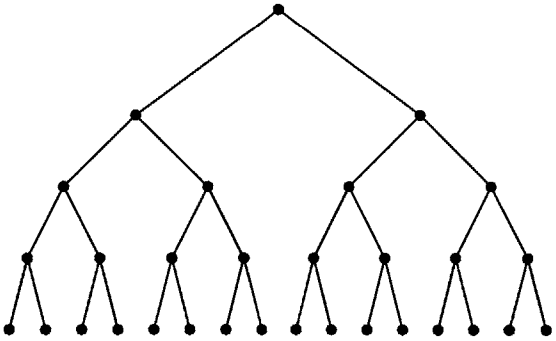


Figure 2.1: The complete binary tree $CBT(4)$ of height 4

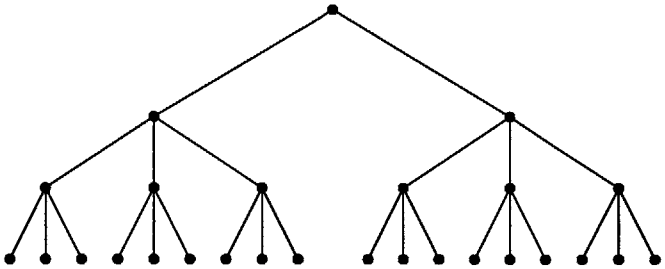


Figure 2.2: The almost complete ternary tree $ACT(2,3,3)$ of height 3 and root branching factor 2

$CBT(4)$ is shown in Figure 2.1.

For $k \geq 2, h \geq 0$, and $1 \leq b \leq k-1$, let $ACT(b,k,h)$ denote the almost complete (or complete) k-ary tree with $b \cdot k^h$ leaves, where the root has b children, and every other node has 0 or k children. For instance, the tree $ACT(2,3,2)$ is shown in Figure 2.2. By definition, we have $CBT(h) = ACT(1,2,h)$. Also, the tree $ACT(b,k,h)$ is complete exactly when $b = 1$.

For any tree T, let $|T|$ denote the size of the tree T, i.e., the number of nodes in T.

Computation Trees

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 model the work of each processor as a binary tree whose leaves are the serial tasks of that processor. 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.

We define T_1 as the minimum serial execution time of the multithreaded computation (the *work* of the computation), and T_∞ as the minimum execution time with an infinite number of processors (the *span* of the computation.) 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 task at the bottom. In particular, we have $T'_\infty < T_\infty$.

Chapter 3

Worst-Case Bound on Successful Steals

The existing literature on work stealing has dealt extensively with the number of steal attempts, but so far it has not distinguished between successful and unsuccessful steals. While that approach leads to a fruitful probabilistic analysis, it does not yield an interesting result for a worst-case analysis. In this chapter, we establish tight upper bounds on the number of successful steals when the computation can be modeled by a computation tree. In particular, if the computation starts with a complete k-ary tree of height h, the maximum number of successful steals is $\sum_{i=1}^{n}(k-1)^i\binom{h}{i}$.

The chapter is organized as follows. Section 3.1 introduces the setting that we consider throughout the chapter. Section 3.2 analyzes the case where at the beginning all the work is owned by one processor, and the computation tree is a binary tree. Section 3.3 generalizes Section 3.2 to the case where the computation tree is an arbitrary rooted tree. Section 3.4 generalizes Section 3.3 one step further and analyzes the case where at the beginning all processors could possibly own work. Finally, Section 3.5 elaborates on a few properties of the potential functions that we define to help with our analysis.

3.1 Setting

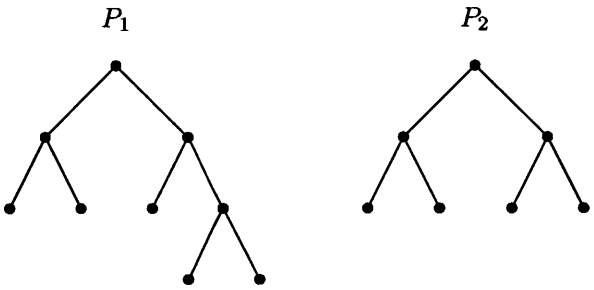
This section introduces the setting that we will consider throughout the chapter.

Consider a setting with P processors. Each processor owns some pieces of work, which we define as *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 model the work of each processor as a tree whose leaves are the serial tasks of that processor.

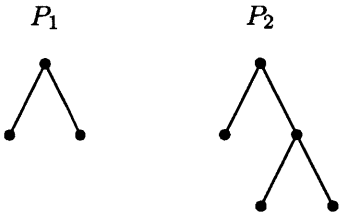
We are concerned with the worst-case number of successful steals that can occur throughout the execution of all serial tasks, as a function of the structure of the trees of the P processors. That is, given the structure of the trees of the processors, suppose that an adversary is allowed to choose the sizes of the serial tasks in the trees. Moreover, whenever a processor finishes its current work, the adversary is allowed to choose which other processor this processor should steal from.

In this setting, it is not an interesting question to ask about the worst-case number of steals overall, whether successful or not. Indeed, the adversary can choose to make almost all steals unsuccessful, and thus the worst-case number of steals is governed by the maximum amount of work among the P processors. But as we are concerned only with successful steals here, the adversary cannot benefit from making steals unsuccessful.

Before we begin our analysis of the number of successful steals, we consider an example of an execution of work stealing in Figure 3.1. There are two processors in this example, P_1 and P_2, and the initial structure of the trees corresponding to the two processors are shown in Figure 3.1(a). Three successful steals are performed, yielding the tree structures in 3.1(b), (c), and (d) respectively. In this example, the maximum number of successful steals that can be performed in the execution is also 3. Observe that we have not chosen the sizes of the serial tasks in the example. Nevertheless, we will show in the next section that for any sequence of steals, such that the one in Figure 3.1, the adversary can choose the sizes of the serial tasks so that the execution of work stealing follows the given sequence of steals.



(a) Initial configuration



(b) Processor P_2 steals from processor P_1

(c) Processor P_1 steals from processor P_2

(d) Processor P_1 steals from processor P_2

Figure 3.1: Example of an execution of work stealing

3.2 Binary Tree and One Starting Processor

In this section, we establish the upper bound on the number of successful steals in the case where at the beginning, all the work is owned by one processor, and the computation tree is a binary tree. We first prove a lemma showing that we only need to be concerned with the structure of the trees, and not the size of the serial tasks. Then, we define a potential function to help with establishing the upper bound, and we derive a recurrence relation for the potential function. The recurrence directly yields an algorithm that computes the upper bound in time $O(|T|n)$. Finally, we show that the maximum number of steals that can occur if our configuration starts with the tree $CBT(h)$ is $\sum_{i=1}^{n} \binom{h}{i}$.

Recurrence

Recall that in our setting, the adversary is allowed to choose the sizes of the serial tasks in the trees. Intuitively, the adversary wants to make sure that when a steal occurs, the trees not involved in the steal are working on serial tasks that are large enough so that they cannot finish the tasks yet. We formalize this intuition in the following lemma.

Lemma 1. *Fix the structure of the binary trees belonging to the P processors. Given a sequence of steals performed on the trees, the adversary can choose the sizes of the serial tasks in such a way that when the execution proceeds, the steals that occur coincide with the given sequence of steals.*

Proof. First, the adversary chooses every serial task to be of size 1. Consider the steals in the given sequence, one by one, and imagine an ongoing execution. Suppose that the next steal dictates that processor P_i steal from processor P_j. The adversary waits until P_i finishes its remaining serial tasks, and meanwhile increases the sizes of the serial tasks belonging to all trees except P_i, if necessary, to ensure that those trees do not finish before P_i. Then the adversary dictates that P_i steal (successfully) from P_j. In this way, the adversary can implement the given sequence of steals. □

Lemma 1 tells us that in order to establish the upper bound on the number of successful steals, it suffices to consider only the structure of the given trees and determine a maximum sequence of

steals.

Suppose that we are given a configuration in which all processors but one start with an empty tree (or one serial task—it makes no difference to possibilities of sequences of steals), while the exceptional processor starts with a tree T. How might a sequence of steals proceed? The first steal is fixed—it must split the tree T into its left and right subtrees, T_l and T_r. From there, one way to proceed is, in some sense, to be greedy. We obtain as many steals out of T_l as we can, while keeping T_r intact. As such, we have $P-1$ processors that we can use to perform steals on T_l, since the last processor must maintain T_r. Then, after we are done with T_l, we can perform steals on T_r using all of our P processors. This motivates the following definition.

Definition 2. *Let $n \geq 0$ be an integer and T a binary tree. The nth potential of T is defined as the maximum number of steals that can be obtained from a configuration of $n+1$ processors, one of which has the tree T and the remaining n of which have empty trees. The nth potential of T is denoted by $\Phi(T, n)$.*

If we only have one processor to work with, we cannot perform any steals, hence $\Phi(T, 0) = 0$ for any tree T. Moreover, the empty tree $EMPT$ and the trivial tree with a single node $TRIVT$ cannot generate any steals, hence $\Phi(EMPT, n) = \Phi(TRIVT, n) = 0$ for all $n \geq 0$.

In addition, the discussion leading up to the definition shows that if a binary tree T has left subtree T_l and right subtree T_r, then for any $n \geq 1$, we have

$$\Phi(T, n) \geq 1 + \Phi(T_l, n-1) + \Phi(T_r, n).$$

By symmetry, we also have

$$\Phi(T, n) \geq 1 + \Phi(T_r, n-1) + \Phi(T_l, n).$$

Combining the two inequalities yields

$$\Phi(T, n) \geq 1 + \max\{\Phi(T_l, n-1) + \Phi(T_r, n), \Phi(T_r, n-1) + \Phi(T_l, n)\}.$$

In the next theorem, we show that this inequality is in fact always an equality.

Theorem 3. *Let T be a binary tree with more than 1 node, and let T_l and T_r be its left and right subtrees. Then for any $n \geq 1$, we have*

$$\Phi(T, n) = 1 + \max\{\Phi(T_l, n-1) + \Phi(T_r, n), \Phi(T_r, n-1) + \Phi(T_l, n)\}.$$

Proof. Since we have already shown that the left-hand side is no less than the right-hand side, it only remains to show the reverse inequality.

Suppose that we are given any sequence of steals performed on T using n processors. As we have noted before, the first steal is fixed—it must split the tree T into its two subtrees, T_l and T_r. Each of the subsequent steals is performed either on a subtree of T_l or a subtree of T_r (not necessarily *the* left or right subtrees of T_l or T_r.) Assume for now that the last steal is performed on a subtree of T_r. That means that at any particular point in the stealing sequence, subtrees of T_l occupy at most $n-1$ processors. (Subtrees of T_l may have occupied all n processors at different points in the stealing sequence, but that does not matter.) We can canonicalize the sequence of steals in such a way that the steals on subtrees of T_l are performed first using $n-1$ processors, and then the steals on subtrees of T_r are performed using n processors. Therefore, in this case the total number of steals is at most $1 + \Phi(T_l, n-1) + \Phi(T_r, n)$.

Similarly, if the last steal is performed on a subtree of T_l, then the total number of steals is at most $1 + \Phi(T_r, n-1) + \Phi(T_l, n)$. Combining the two cases, we have

$$\Phi(T, n) \leq 1 + \max\{\Phi(T_l, n-1) + \Phi(T_r, n), \Phi(T_r, n-1) + \Phi(T_l, n)\},$$

which gives us the desired equality.

□

Algorithm

When combined with the base cases previously discussed, Theorem 3 gives us a recurrence that we can use to compute $\Phi(T, n)$ for any binary tree T and any value of n. But how fast can we compute

the potential? The next corollary addresses that question. Recall from Chapter 2 that $|T|$ denotes the size of the tree T.

Corollary 4. *There exists an algorithm that computes the potential $\Phi(T, n)$ in time $O(|T|n)$.*

Proof. The algorithm uses dynamic programming to compute $\Phi(T, n)$. For each subtree T' of T and each value $0 \leq i \leq n$, it computes $\Phi(T', i)$ using the recurrence given in Theorem 3. There are $O(|T|n)$ subproblems to solve, and each subproblem takes $O(1)$ time to compute. Hence the running time is $O(|T|n)$. □

Complete Binary Trees

An interesting special case is when the initial tree T is a complete binary tree, i.e., a full binary tree in which all leaves have the same depth and every parent has two children. Recall from Chapter 2 that $CBT(h)$ denotes the complete binary tree with height h. The next corollary establishes the potential of $CBT(h)$. Recall also from Chapter 2 that $\binom{a}{b} = 0$ if $a < b$.

Corollary 5. *We have*

$$\begin{aligned}\Phi(CBT(h), n) &= \binom{h}{1} + \binom{h}{2} + \ldots + \binom{h}{n} \\ &= \sum_{i=1}^{n} \binom{h}{i}.\end{aligned}$$

Proof. The case $h = 0$ holds, since

$$\Phi(CBT(0), n) = \sum_{i=1}^{n} \binom{0}{i} = 0.$$

The case $n = 0$ holds similarly. Now suppose $h, n > 0$. Since the two subtrees of $CBT(h)$ are symmetric, the recurrence in Theorem 3 yields

$$\Phi(CBT(h), n) = 1 + \Phi(CBT(h-1), n-1) + \Phi(CBT(h-1), n).$$

We proceed by induction. Suppose that the formula for the potential values hold for $CBT(h-1)$.

Using Pascal's identity (Equation 2.1), we have

$$\begin{aligned}\Phi(CBT(h),n) &= 1 + \Phi(CBT(h-1),n-1) + \Phi(CBT(h-1),n) \\ &= \binom{h-1}{0} + \sum_{i=1}^{n-1}\binom{h-1}{i} + \sum_{i=1}^{n}\binom{h-1}{i} \\ &= \sum_{i=1}^{n}\left(\binom{h-1}{i-1} + \binom{h-1}{i}\right) \\ &= \sum_{i=1}^{n}\binom{h}{i},\end{aligned}$$

as desired.

□

For fixed n, $\Phi(T_{bin,perf,h},n)$ grows as $O(h^n)$. Indeed, one can obtain the (loose) bound

$$\sum_{i=1}^{n}\binom{h}{i} \leq h^n$$

for $h,n \geq 2$, for example using the simple bound

$$\binom{h}{i} = \frac{h(h-1)\cdots(h-i+1)}{i!} \leq h(h-1)^{i-1}$$

and then computing a geometric sum.

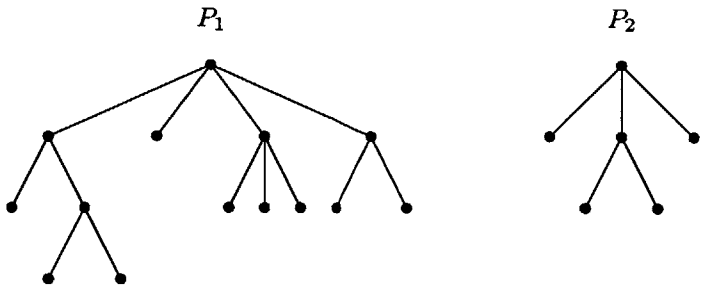
We have established the upper bound on the number of successful steals in the case where at the beginning, all the work is owned by one processor, and the computation tree is a binary tree. In fact, this configuration is used in usual implementations of the work-stealing algorithm, and as such the analysis presented in this section suffices. Nevertheless, in the next sections we will generalize to configurations where the work can be spread out at the beginning and take the form of arbitrary rooted trees as well.

3.3 Rooted Tree and One Starting Processor

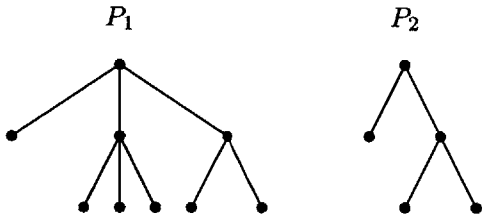
In this section, we consider a generalization of Section 3.2 to the configuration in which the starting tree is an arbitrary rooted tree. The key observation is that we can transform an arbitrary rooted tree into a "left-child right-sibling" binary tree that is equivalent with respect to steals. With this transformation, an algorithm that computes the upper bound in time $O(|T|n)$ follows. Finally, we show that the maximum number of steals that can occur if our configuration starts with the tree $ACT(b,k,h)$ is $\sum_{i=1}^{n}(k-1)^i\binom{h}{i} + (b-1)\sum_{i=0}^{n-1}(k-1)^i\binom{h}{i}$.

We assume that nodes in our tree can have any number of children other than 1. The reason is that if a node has one child, it is not clear how a steal should proceed. We still maintain the assumption that at the beginning all work is owned by one processor. So far, we have not specified whether when processor P_i steals from processor P_j it should steal the left subtree or the right subtree. In fact, it must steal the subtree that P_j is not working on. Up to now, this distinction has not mattered in our analysis, because the processors are symmetric: when one tree breaks into its two subtree, it does not matter which processor has which subtree. In this section, however, we will use the convention that a processor works on serial tasks in its right subtree, and when another processor steals from it, that processor steals the left subtree.

With this convention, we can define how a steal is performed on arbitrary rooted trees. Suppose that processor P_i steals from processor P_j, and assume that the root node of P_j has m children. If $m > 2$, then P_i steals the leftmost subtree, leaving the root node and the other $m-1$ subtrees of P_j intact. On the other hand, if $m = 2$, then the usual steal on binary trees is applied. That is, P_i steals the left subtree, leaving P_j with the right subtree, while the root node disappears. An example of an execution of work stealing on rooted trees is shown in Figure 3.2. In this example, the root node has four children, and therefore the steal takes away the leftmost subtree and leaves the remaining three subtrees intact. This definition of stealing in rooted trees is not arbitrary. It is, in fact, the one commonly used in parallel computing.



(a) Initial configuration



(b) Processor P_2 steals from processor P_1

Figure 3.2: Example of an execution of work stealing on rooted trees

Algorithm

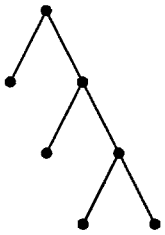
The key to establishing the upper bound in the case of arbitrary rooted trees is the observation that the problem can in fact be reduced to the case of binary trees, which we have already settled in Section 3.2. Indeed, one can check that a node with k children is equivalent (with respect to stealing) to a left-child right-sibling binary tree [10] of height $k - 1$. For instance, the complete quarternary tree of height 1 is equivalent to a left-child right-sibling binary tree of height 3, as shown in Figure 3.3.

With this equivalence, one can transform any rooted tree into a binary tree by transforming each node with more than two children into a left-child right-sibling binary tree on its own. The transformation is shown in Figure 3.4. Therefore, we can use the same algorithm as in the case of binary trees to compute the maximum number of successful steals, as the next theorem shows. Recall from Chapter 2 that $|T|$ denotes the size of the tree T.

Definition 6. *A node in a rooted tree is called a singleton node if it has exactly one child.*

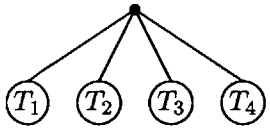
Theorem 7. *Let T be a rooted tree with no singleton nodes. There exists an algorithm that*

(a) The complete quarternary tree of height 1

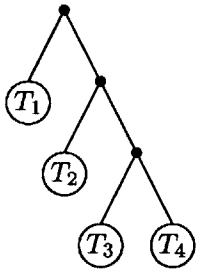


(b) Left-child right-sibling binary tree representation of the tree in (a)

Figure 3.3: Rooted tree and left-child right-sibling binary tree



(a) Tree whose root node has 4 children



(b) Binary tree resulting from transformation

Figure 3.4: Transformation from rooted tree into binary tree

computes the potential $\Phi(T, n)$ *in time* $O(|T|n)$.

Proof. We transform T into a binary tree according to the discussion leading up to this theorem, and apply the algorithm described in Corollary 4. The transformation takes time of order $|T|$, and the algorithm takes time of order $|T|n$. □

Complete k-ary Trees

As in our analysis of binary trees, an interesting special case is the case of a complete k-ary tree, i.e., a full k-ary tree in which all leaves have the same depth and every parent has k children. Moreover, we also determine the answer for almost complete k-ary trees. Recall from Chapter 2 that for $k \geq 2, h \geq 0$, and $1 \leq b \leq k-1$, $ACT(b, k, h)$ denotes the almost complete (or complete) k-ary tree with $b \cdot k^h$ leaves.

Theorem 8. *For* $k \geq 2, h \geq 0$, *and* $1 \leq b \leq k-1$, *we have*

$$\begin{aligned}\Phi(ACT(b,k,h),n) &= \left((k-1)\binom{h}{1} + (k-1)^2\binom{h}{2} + \cdots + (k-1)^n\binom{h}{n}\right) \\ &\quad + (b-1)\left(\binom{h}{0} + (k-1)\binom{h}{1} + \cdots + (k-1)^{n-1}\binom{h}{n-1}\right) \\ &= \sum_{i=1}^{n}(k-1)^i\binom{h}{i} + (b-1)\sum_{i=0}^{n-1}(k-1)^i\binom{h}{i}.\end{aligned}$$

Proof. We first show that the formula is consistent even if we allow the tree $ACT(1, k, h+1)$ to be

written as $ACT(k,k,h)$ for $h \geq 0$. Indeed, we have

$$\begin{aligned}
\Phi(ACT(1,k,h+1),n) &= (k-1)\binom{h+1}{1} + (k-1)^2\binom{h+1}{2} + \cdots + (k-1)^n\binom{h+1}{n} \\
&= (k-1)\left(\binom{h}{0} + \binom{h}{1}\right) + (k-1)^2\left(\binom{h}{1} + \binom{h}{2}\right) \\
&\quad + \ldots + (k-1)^n\left(\binom{h}{n-1} + \binom{h}{n}\right) \\
&= \left((k-1)\binom{h}{1} + (k-1)^2\binom{h}{2} + \cdots + (k-1)^n\binom{h}{n}\right) \\
&\quad + (k-1)\left(\binom{h}{0} + (k-1)\binom{h}{1} + \cdots + (k-1)^{n-1}\binom{h}{n-1}\right) \\
&= \Phi(ACT(k,k,h),n),
\end{aligned}$$

where we used Pascal's identity (Equation 2.1).

We proceed to prove the formula. The case $b \cdot k^h = 1$ holds, since both the left-hand side and the right-hand side are zero. The case $n = 0$ holds similarly. Consider the tree $ACT(b,k,h)$, where $b \cdot k^h > 1$ and $2 \leq b \leq k$. Using the consistency of the formula that we proved above, it is safe to represent any nontrivial tree in such form. Now, the recurrence in Theorem 3 yields

$$\begin{aligned}
\Phi(ACT(b,k,h),n) = 1 + \max\big\{&\Phi(ACT(b-1,k,h),n) + \Phi(ACT(1,k,h),n-1), \\
&\Phi(ACT(b-1,k,h),n-1) + \Phi(ACT(1,k,h),n)\big\}.
\end{aligned}$$

Letting $A = \Phi(ACT(b-1,k,h),n) + \Phi(ACT(1,k,h),n-1)$ and $B = \Phi(ACT(b-1,k,h),n-1) + \Phi(ACT(1,k,h),n)$, we have

$$\begin{aligned}
A &= \left(\sum_{i=1}^{n}(k-1)^i\binom{h}{i} + (b-2)\sum_{i=0}^{n-1}(k-1)^i\binom{h}{i}\right) + \sum_{i=1}^{n-1}(k-1)^i\binom{h}{i} \\
&= \left(\sum_{i=1}^{n}(k-1)^i\binom{h}{i} + (b-1)\sum_{i=0}^{n-1}(k-1)^i\binom{h}{i}\right) - 1
\end{aligned}$$

and

$$\begin{aligned} B &= \left(\sum_{i=1}^{n-1}(k-1)^i\binom{h}{i} + (b-2)\sum_{i=0}^{n-2}(k-1)^i\binom{h}{i}\right) + \sum_{i=1}^{n}(k-1)^i\binom{h}{i} \\ &= A - (b-2)(k-1)^{n-1}\binom{h}{n-1} \\ &\leq A. \end{aligned}$$

Therefore, we have

$$\begin{aligned} \Phi(ACT(b,k,h),n) &= 1 + A \\ &= \sum_{i=1}^{n}(k-1)^i\binom{h}{i} + (b-1)\sum_{i=0}^{n-1}(k-1)^i\binom{h}{i}, \end{aligned}$$

as desired. □

We have established the upper bound on the number of successful steals in the configuration with one processor having an arbitrary rooted tree at the beginning. In the next section, we generalize one step further by allowing any number of processors to own work at the beginning.

3.4 Rooted Tree and Many Processors

In this section, we consider a generalization of Section 3.3 where the work is not limited to one processor at the beginning, but rather can be spread out as well. We derive a formula for computing the potential function of a configuration based on the potential function of the individual trees. This leads to an algorithm that computes the upper bound for the configuration with trees $T_1, T_2, \ldots, T_P$ in time $O(P^3 + P(|T_1| + |T_2| + \ldots + |T_P|))$. We then show that for complete k-ary trees, we only need to sort the trees in order to compute the maximum number of steals. Since we can convert any computation tree into a binary tree, it suffices throughout this section to analyze the case in which all trees are binary trees.

Formula

Suppose that in our configuration, the P processors start with trees $T_1, T_2, \ldots, T_P$. How might a sequence of steals proceed? As in our previous analysis of the case with one starting processor, we have an option of being greedy. We pick one tree—say T_1—and obtain as many steals as possible out of it using one processor. After we are done with T_1, we pick another tree—say T_2—and obtain as many steals as possible out of it using two processors. We proceed in this way until we pick the last tree—say T_P—and obtain as many steals out of it using all P processors.

We make the following definition.

Definition 9. *Let $T_1, T_2, \ldots, T_n$ be binary trees. Then $\Phi(T_1, T_2, \ldots, T_n)$ is the maximum number of steals that we can get from a configuration of n processors that start with the trees $T_1, T_2, \ldots, T_n$.*

Note that we are overloading the potential function operator Φ. Unlike the previous definition of Φ, this definition does not explicitly include the number of processors, because it is simply the number of trees included in the argument of Φ. It follows from the definition that $\Phi(T_1, T_2, \ldots, T_n) = \Phi(T_{\sigma(1)}, T_{\sigma(2)}, \ldots, T_{\sigma(n)})$ for all permutations σ of $1, 2, \ldots, n$.

From the discussion leading up to the definition, we have

$$\Phi(T_1, T_2, \ldots, T_P) \geq \Phi(T_{\sigma(1)}, 0) + \Phi(T_{\sigma(2)}, 1) + \ldots + \Phi(T_{\sigma(P)}, P-1)$$

for any permutation σ of $1, 2, \ldots, P$. It immediately follows that

$$\Phi(T_1, T_2, \ldots, T_P) \geq \max_{\sigma \in S_P} \left(\Phi(T_{\sigma(1)}, 0) + \Phi(T_{\sigma(2)}, 1) + \ldots + \Phi(T_{\sigma(P)}, P-1) \right),$$

where S_P denotes the symmetric group of order P, i.e., the set of all permutations of $1, 2, \ldots, P$.

The next theorem shows that this inequality is in fact an equality.

Theorem 10. *Let $T_1, T_2, \ldots, T_P$ be binary trees. We have*

$$\Phi(T_1, T_2, \ldots, T_P) = \max_{\sigma \in S_P} \left(\Phi(T_{\sigma(1)}, 0) + \Phi(T_{\sigma(2)}, 1) + \ldots + \Phi(T_{\sigma(P)}, P-1) \right).$$

Proof. We have already shown that the left-hand side is no less than the right-hand side, hence it only remains to show the reverse inequality.

Suppose that we are given any sequence of steals performed on $T_1, T_2, \ldots, T_P$ using the P processors. Each steal is performed on a subtree of one of the trees $T_1, T_2, \ldots, T_P$.

Assume without loss of generality that the last steal performed on a subtree of T_1 occurs before the last steal performed on a subtree of T_2, which occurs before the last steal performed on a subtree of T_3, and so on. That means that at any particular point in the stealing sequence, subtrees of T_i occupy at most i processors, for all $1 \leq i \leq P$. (Subtrees of T_i may have occupied a total of more than i processors at different points in the stealing sequence, but that does not matter.) We can canonicalize the sequence of steals in such a way that all steals on subtrees of T_1 are performed first using one processor, then all steals on subtrees of T_2 are performed using two processors, and so on, until all steals on subtrees of T_P are performed using P processors. Therefore, in this case the total number of steals is no more than $\Phi(T_1, 0) + \Phi(T_2, 1) + \ldots + \Phi(T_P, P-1)$.

In general, let σ be the permutation of $1, 2, \ldots, P$ such that the last steal performed on a subtree of $T_{\sigma(1)}$ occurs before the last steal performed on a subtree of $T_{\sigma(2)}$, which occurs before the last steal performed on a subtree of $T_{\sigma(3)}$, and so on. Then we have

$$\Phi(T_1, T_2, \ldots, T_P) \leq \Phi(T_{\sigma(1)}, 0) + \Phi(T_{\sigma(2)}, 1) + \ldots + \Phi(T_{\sigma(P)}, P-1).$$

Therefore,

$$\Phi(T_1, T_2, \ldots, T_P) \leq \max_{\sigma \in S_P} \left(\Phi(T_{\sigma(1)}, 0) + \Phi(T_{\sigma(2)}, 1) + \ldots + \Phi(T_{\sigma(P)}, P-1) \right),$$

which gives us the desired equality. □

Algorithm

Now that we have a formula to compute $\Phi(T_1, T_2, \ldots, T_P)$, we again ask how fast we can compute it. Recall from Chapter 2 that $|T|$ denotes the size of the tree T.

Corollary 11. *There exists an algorithm that computes the potential* $\Phi(T_1, T_2, \ldots, T_P)$ *in time* $O(P^3 + P(|T_1| + |T_2| + \ldots + |T_P|))$.

Proof. The potentials $\Phi(T_i, j)$ can be precomputed by dynamic programming in time $O(P(|T_1| + |T_2| + \ldots + |T_P|))$ using the algorithm in Corollary 4. It then remains to determine the maximum value of $\Phi(T_{\sigma(1)}, 0) + \Phi(T_{\sigma(2)}, 1) + \ldots + \Phi(T_{\sigma(P)}, P-1)$ over all permutations σ of $1, 2, \ldots, P$. A brute-force solution that tries all possible permutations σ of $1, 2, \ldots, n$ takes time $O(P!)$. However, our maximization problem is an instance of the assignment problem, which can be solved using the classical "Hungarian method". The algorithm by Tomizawa [21] solves the assignment problem in time $O(P^3)$. Hence, the total running time is $O(P^3 + P(|T_1| + |T_2| + \ldots + |T_P|))$. □

Complete Trees

It is interesting to ask whether we can do better than the algorithm in Corollary 11 in certain special cases. Again, we consider the case of complete trees. In this subsection we assume that the P processors start with almost complete (or complete) k-ary trees (defined in Chapter 2) for the same value of k. The case where the values of k are different across different processors is harder to deal with, as will be explained in the next section.

Suppose the processors start with the trees $ACT(b_1, k, h_1), ACT(b_2, k, h_2), \ldots, ACT(b_P, k, h_P)$, where $1 \leq b_1, b_2, \ldots, b_P \leq k-1$. We may assume without loss of generality that $b_1 \cdot k^{h_1} \leq b_2 \cdot k^{h_2} \leq \ldots \leq b_P \cdot k^{h_P}$. Intuitively, in order to generate the maximum number of successful steals, one might want to allow larger trees more processors to work with, because larger trees can generate more steals than smaller trees. It turns out that in the case of complete trees, this intuition always works, as is shown in the following theorem.

Theorem 12. *Let* $b_1 \cdot k^{h_1} \leq b_2 \cdot k^{h_2} \leq \ldots \leq b_P \cdot k^{h_P}$. *Consider almost complete (or complete)* k*-ary trees* $ACT(b_1, k, h_1)$,$ACT(b_2, k, h_2)$,...,$ACT(b_P, k, h_P)$. *We have*

$$\Phi(ACT(b_1, k, h_1), ACT(b_2, k, h_2), \ldots, ACT(b_P, k, h_P)) = \sum_{i=1}^{P} \Phi(ACT(b_i, k, h_i), i-1).$$

Proof. We use the exchange argument: if the trees are not already ordered in increasing size, then

there exist two consecutive positions j and $j+1$ such that $b_j \cdot k^{h_j} > b_{j+1} \cdot k^{h_{j+1}}$. We show that we may exchange the positions of the two trees and increase the total potential in the process. Since we can always perform a finite number of exchanges to obtain the increasing order, and we know that the total potential increases with each exchange, we conclude that the maximum potential is obtained exactly when the trees are ordered in increasing size.

It only remains to show that any exchange of two trees $b_j \cdot k^{h_j}$ and $b_{j+1} \cdot k^{h_{j+1}}$ such that $b_j \cdot k^{h_j} > b_{j+1} \cdot k^{h_{j+1}}$ increases the potential. Denote the new potential after the exchange by N and the old potential before the exchange by O. We would like to show that $N > O$. We have

$$\begin{aligned}
N - O = &\left(\sum_{i=1}^{j}(k-1)^i\binom{h_j}{i} + (b_j - 1)\sum_{i=0}^{j-1}(k-1)^i\binom{h_j}{i}\right) \\
&+ \left(\sum_{i=1}^{j-1}(k-1)^i\binom{h_{j+1}}{i} + (b_{j+1} - 1)\sum_{i=0}^{j-2}(k-1)^i\binom{h_{j+1}}{i}\right) \\
&- \left(\sum_{i=1}^{j-1}(k-1)^i\binom{h_j}{i} + (b_j - 1)\sum_{i=0}^{j-2}(k-1)^i\binom{h_j}{i}\right) \\
&- \left(\sum_{i=1}^{j}(k-1)^i\binom{h_{j+1}}{i} + (b_{j+1} - 1)\sum_{i=0}^{j-1}(k-1)^i\binom{h_{j+1}}{i}\right) \\
= &\,(k-1)^j\binom{h_j}{j} + (b_j - 1)(k-1)^{j-1}\binom{h_j}{j-1} \\
&- (k-1)^j\binom{h_{j+1}}{j} - (b_{j+1} - 1)(k-1)^{j-1}\binom{h_{j+1}}{j-1}.
\end{aligned}$$

We consider two cases.

Case 1: $h_j = h_{j+1}$ and $b_j > b_{j+1}$.

We have

$$\begin{aligned}
N - O &= (k-1)^j \binom{h_j}{j} + (b_j - 1)(k-1)^{j-1} \binom{h_j}{j-1} \\
&\quad - (k-1)^j \binom{h_j}{j} - (b_{j+1} - 1)(k-1)^{j-1} \binom{h_j}{j-1} \\
&= (b_j - 1)(k-1)^{j-1} \binom{h_j}{j-1} - (b_{j+1} - 1)(k-1)^{j-1} \binom{h_j}{j-1} \\
&= ((b_j - 1) - (b_{j+1} - 1))(k-1)^{j-1} \binom{h_j}{j-1} \\
&= (b_j - b_{j+1})(k-1)^{j-1} \binom{h_j}{j-1} \\
&> 0,
\end{aligned}$$

since $b_j > b_{j+1}$.

Case 2: $h_j > h_{j+1}$.

We have

$$\begin{aligned}
N - O &= (k-1)^j \binom{h_j}{j} + (b_j - 1)(k-1)^{j-1} \binom{h_j}{j-1} \\
&\quad - (k-1)^j \binom{h_{j+1}}{j} - (b_{j+1} - 1)(k-1)^{j-1} \binom{h_{j+1}}{j-1} \\
&\geq (k-1)^j \binom{h_j}{j} - (k-1)^j \binom{h_{j+1}}{j} - (b_{j+1} - 1)(k-1)^{j-1} \binom{h_{j+1}}{j-1} \\
&\geq (k-1)^j \binom{h_j}{j} - (k-1)^j \binom{h_{j+1}}{j} - (k-1)(k-1)^{j-1} \binom{h_{j+1}}{j-1} \\
&= (k-1)^j \binom{h_j}{j} - (k-1)^j \left(\binom{h_{j+1}}{j} + \binom{h_{j+1}}{j-1} \right) \\
&= (k-1)^j \binom{h_j}{j} - (k-1)^j \binom{h_{j+1}+1}{j} \\
&= (k-1)^j \left(\binom{h_j}{j} - \binom{h_{j+1}+1}{j} \right) \\
&> 0,
\end{aligned}$$

since $h_j \geq h_{j+1} + 1$.

In both cases, we have $N - O > 0$, and hence any exchange increases the potential, as desired. □

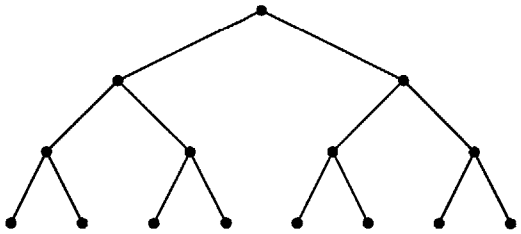


Figure 3.5: The complete binary tree $CBT(3)$ of height 3

It follows from Theorem 12 and Corollary 11 that in the case of complete k-ary trees, one only needs to sort the trees with respect to their size in order to compute the potential. It follows that the running time of the algorithm is bounded by the running time of sorting, which is $O(P \lg P)$.

We have established the upper bound on the number of successful steals in a configuration with all processors having an arbitrary rooted tree at the beginning by defining a potential function. The next section takes a closer look at some properties of our potential function.

3.5 Properties of Potential Function

In the previous sections, we have shown recurrence formulae for computing the potential function, as well as closed-form formulae for certain special cases, such as complete trees. In this section, we shed more light on properties and computations of the potential function. We provide counterexamples to two conjectures of "nice" properties of the potential function, and we give alternative ways of computing the potential function.

Tree Ordering

Theorem 12 raises the natural question of whether we can do better than the algorithm in Corollary 11 in general. In particular, one might expect that we can always order the trees according to their size as in Theorem 12 and compute the potential of the configuration. This is not necessarily true, however, as the following example shows.

Consider the complete binary tree $CBT(3)$ of height 3 (Figure 3.5) and the complete ternary tree $ACT(1, 3, 2)$ of height 2 (Figure 3.6). Suppose that we have a configuration of 5 processors

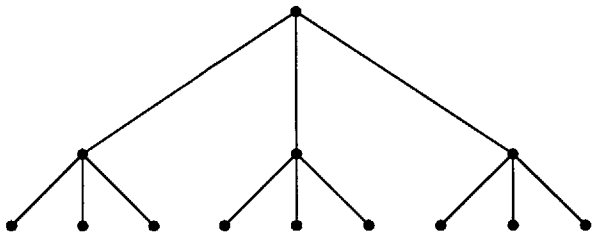


Figure 3.6: The complete ternary tree $ACT(1,3,2)$ of height 2

i	$\Phi(CBT(3),i)$	$\Phi(ACT(1,3,2),i)$
0	0	0
1	3	4
2	6	8
3	7	8
4	7	8

Table 3.1: Potentials of the trees $\Phi(CBT(3),i)$ and $\Phi(ACT(1,3,2),i)$

($P = 5$), three of which start with the tree $CBT(3)$, and the remaining two of which start with the tree $ACT(1,3,2)$. Thus the maximum number of successful steals in this configuration is given by the potential $\Phi(CBT(3), CBT(3), CBT(3), ACT(1,3,2), ACT(1,3,2))$.

To calculate this potential, we first calculate the potentials $\Phi(CBT(3),i)$ and $\Phi(ACT(1,3,2),i)$ for $0 \leq i \leq 4$. Since both trees are complete k-ary trees, the formula given in Theorem 8 applies. The results are shown in Table 3.1.

We are now ready to compute $\Phi(CBT(3), CBT(3), CBT(3), ACT(1,3,2), ACT(1,3,2))$. We do that by considering all possible ordering of the trees. The results are shown in Table 3.2.

The orderings that yield the maximum potential have the trees $CBT(3)$ in positions $\{0,1,3\}$, $\{0,1,4\}$, and $\{0,3,4\}$. None of these orderings provide an order between the trees $CBT(3)$ and $ACT(1,3,2)$. Indeed, such orderings would require $CBT(3)$ to be in either positions $\{0,1,2\}$ or positions $\{2,3,4\}$. This example shows that an inherent order between trees does not exist even when the trees are complete k-ary trees, if the trees have different values of k.

Positions of $CBT(3)$	Positions of $ACT(1,3,2)$	Total potential
0,1,2	3,4	25
0,1,3	**2,4**	**26**
0,1,4	**2,3**	**26**
0,2,3	1,4	25
0,2,4	1,3	25
0,3,4	**1,2**	**26**
1,2,3	0,4	24
1,2,4	0,3	24
1,3,4	0,2	25
2,3,4	0,1	24

Table 3.2: Calculation of the potential $\Phi(CBT(3), CBT(3), CBT(3), ACT(1,3,2), ACT(1,3,2))$

i	$\Phi(CBT(3),i) + \Phi(ACT(1,3,2),i+1)$	$\Phi(ACT(1,3,2),i) + \Phi(CBT(3),i+1)$
0	**4**	3
1	**11**	10
2	14	**15**
3	**15**	**15**

Table 3.3: Placement of two trees in consecutive positions

Monotonicity

Another interesting question to ask is the following: Given two trees, T_1 and T_2, and two positions, is it always optimal to place one tree in the "higher" position and the other tree in the "lower" position? If the answer is affirmative, it could lead to a way of simplifying the calculation of the potential in Theorem 12.

Unfortunately, the answer is negative, and it remains negative even when we limit our attention to two *consecutive* positions. The same two trees $CBT(3)$ and $ACT(1,3,2)$ once again provide a counterexample.

As shown in Table 3.3, we cannot determine which tree to place in the "higher" position without considering the positions themselves. There exists a position in which it is optimal to place $ACT(1,3,2)$ in the higher position ($i = 0, 1$), a position in which it is optimal to place $CBT(3)$ in the higher position ($i = 2$), and a position in which both options are equally optimal ($i = 3$).

A follow-up question is whether this "turning point" at which the optimality changes from one tree being in the higher position to the other tree being in the higher position happens at most once. In this example, there is only one turning point from $i = 1$ to $i = 2$. If this statement is true in general, it could yet lead to a way to simplify the potential computation. We do not know the answer to this question.

Alternative Computation Methods

In this subsection, we provide alternative ways to compute the potential $\Phi(T, n)$ that are equivalent to the method given in Theorem 3.

As an example, consider the binary tree T_1 given in Figure 3.7, and suppose that we wish to compute $\Phi(T_1, 2)$. In general, when we compute the potential $\Phi(T, n)$, the recurrence formula given in Theorem 3 chooses one of the two branches and decreases the second argument in the potential function to $n - 1$ for that branch, while maintaining the second argument in the potential function as n for the other branch. In the tree in Figure 3.7, we choose one of the two branches for each non-leaf node, and the chosen branches are bolded. The number next to each node indicates the value of n corresponding to that node.

For example, the bottom-right node N_{15} has value 0, because the way down from the root node to it contains two bolded branches. On the other hand, node N_{11} has value 2, because the way down from the root node to it contains no bolded branches. It is easy to see that the label of any node is no more than n, and it can also be negative (as in the case of node N_5 here.)

We make the following definition.

Definition 13. *For a binary tree T, let $\Gamma(T)$ denote the set of all the ways to choose one of the two branches for each internal (i.e., non-leaf) node of a binary tree T.*

If a tree T contains k internal nodes, then $|\Gamma(T)| = 2^k$. Indeed, each internal node allows a choice between its two branches, independent of other choices.

Recall that the base cases of the recurrence in Theorem 3 include the case where $n = 0$ and the case where the tree is the trivial tree, both of which have zero potential. A potential of one arises every time we invoke the recurrence. This suggests the following way of computing the potential.

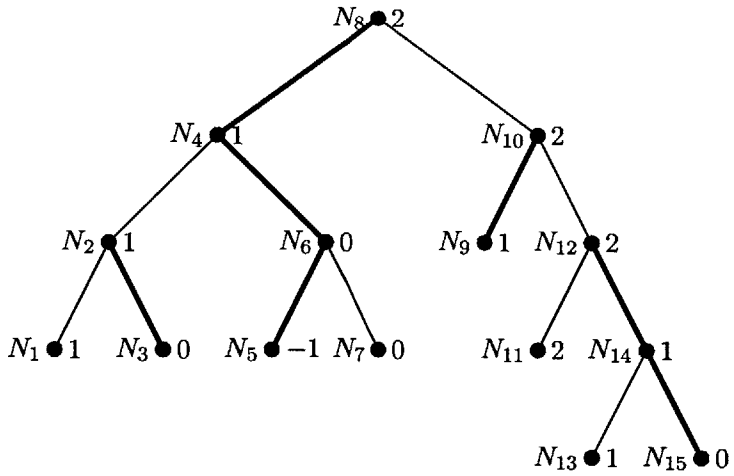


Figure 3.7: Breakdown of potential calculation in an example binary tree

Theorem 14. *Let $n \geq 0$. For each fixed $\gamma \in \Gamma(T)$, label each node with n subtracted by the number of chosen branches on the path from the root node to it, and let $f_1(\gamma)$ denote the number of internal nodes with positive label. Then $\Phi(T, n) = \max_{\gamma \in \Gamma(T)} f_1(\gamma)$.*

Proof. In the recurrence in Theorem 3, the only place where a potential arises is every time we invoke the recurrence on an internal node, and a potential of exactly 1 arises. Therefore, for fixed γ, a potential of 1 arises exactly when an internal node has a positive label (since we can invoke the recurrence on it). Since $\Phi(T, n)$ is simply the maximum obtained over all possible ways to choose branches, we have the desired result. □

For the binary tree in Figure 3.7 and the particular way of choosing its branches, one can check that six internal node have positive labels. Indeed, the six internal nodes with positive labels are circled in Figure 3.8.

Two equivalent methods of calculating the potential immediately follows from Theorem 14, where we consider a node that results from using the recurrence instead of the node that is used for the recurrence. In one case, we consider the child node that is connected via a chosen branch, while in the other case we consider the child node that is connected via an unchosen branch.

Corollary 15. *Let $n \geq 0$. For each fixed $\gamma \in \Gamma(T)$, label each node with n subtracted by the number of chosen branches on the path from the root node to it, and let $f_2(\gamma)$ denote the number of nodes*

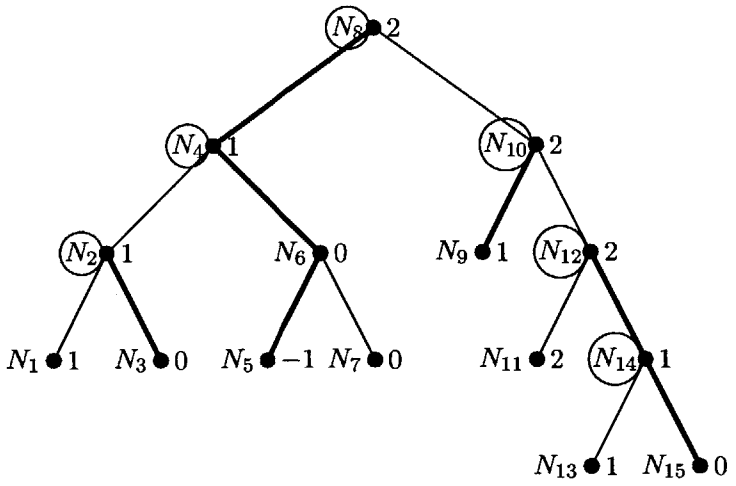


Figure 3.8: Potential calculation according to Theorem 14

with positive label such that the node has a parent and the edge joining it with its parent is **not** *chosen. Then* $\Phi(T, n) = \max_{\gamma \in \Gamma(T)} f_2(\gamma)$.

Proof. For each internal node with positive label, the child that it connects to via an unchosen edge has the same label, in particular positive. Hence the number of internal nodes with positive label is the same as the number of nodes with positive label such that the node has a parent and the edge joining it with its parent is not chosen. Combined with Theorem 14, this concludes the proof. □

The nodes that satisfy the condition in Corollary 15 are circled in Figure 3.9. There are six such nodes, which matches our previous computation.

Corollary 16. *Let* $n \geq 0$. *For each fixed* $\gamma \in \Gamma(T)$, *label each node with* n *subtracted by the number of chosen branches on the path from the root node to it, and let* $f_3(\gamma)$ *denote the number of nodes with nonnegative label such that the node has a parent and the edge joining it with its parent is chosen. Then* $\Phi(T, n) = \max_{\gamma \in \Gamma(T)} f_3(\gamma)$.

Proof. For each internal node with positive label, the child that it connects to via a chosen edge has a label that is 1 less, in particular nonnegative. Hence the number of internal nodes with positive label is the same as the number of nodes with nonnegative label such that the node has a parent

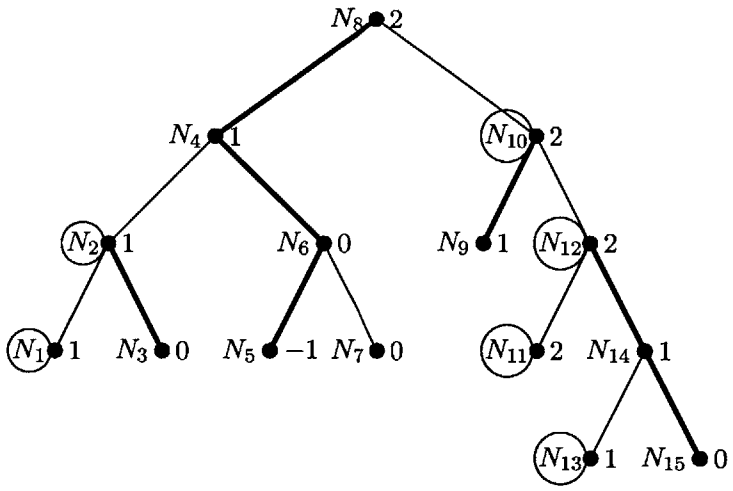


Figure 3.9: Potential calculation according to Corollary 15

and the edge joining it with its parent is chosen. Combined with Theorem 14, this concludes the proof. □

The nodes that satisfy the condition in Corollary 16 are circled in Figure 3.10. Again, there are six such nodes.

Theorem 14 gives a way of computing the potential based on the number of internal nodes with a certain property of its label. Is there a corresponding way based on the number of leaf nodes? The next theorem shows such a way.

Theorem 17. *Let $n \geq 0$. For each fixed $\gamma \in \Gamma(T)$, label each node with n subtracted by the number of chosen branches on the path from the root node to it, and let $f_4(\gamma)$ denote the number of leaf nodes with nonnegative label not equal to n. Then $\Phi(T, n) = \max_{\gamma \in \Gamma(T)} f_4(\gamma)$.*

Proof. For each node with nonnegative label such that the node has a parent and the edge joining it with its parent is chosen, the node can be associated to a leaf node (which may happen to be itself) as follows: Follow the path with unchosen edges down to a leaf. Since the edges on the path are not chosen, the label of the leaf is the same as the label of the node, in particular nonnegative.

It remains to check that the correspondence is in fact bijective. Since every node we consider has a chosen edge joining it to its parent, it cannot be on a path from another node down to a leaf that contains only unchosen edges. Hence the correspondence is injective. In addition, from

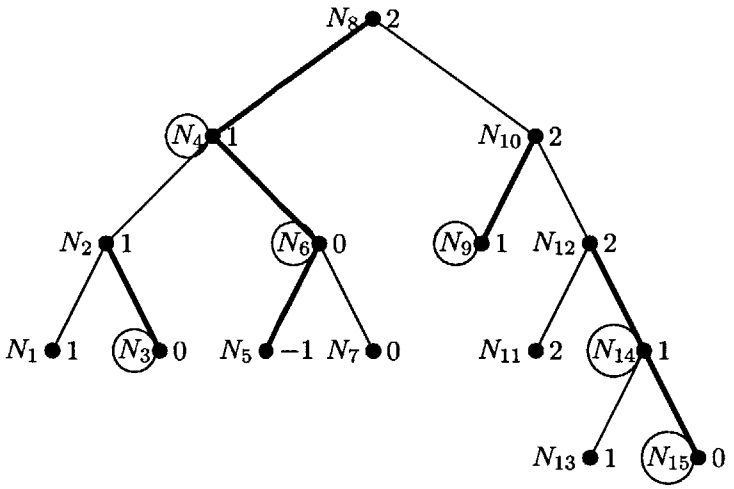


Figure 3.10: Potential calculation according to Corollary 16

every leaf node with nonnegative label not equal to n, we can trace back up the tree until we encounter a chosen edge. Since the label is strictly less than n, this chosen edge must exist. Hence the correspondence is also surjective. Combined with Corollary 16, this concludes the proof. □

The nodes that satisfy the condition in Theorem 17 are circled in Figure 3.11. Once again, there are six such nodes.

While the different methods of calculating the potential function do not directly translate into a faster algorithm, they present alternative viewpoints of the potential function calculation. After all, the potential function and the associated recurrence are central to establishing the upper bounds on the number of successful steals throughout the chapter. We hope that a deeper understanding of the potential function will lead to faster or more general algorithms to calculate the number of successful steals.

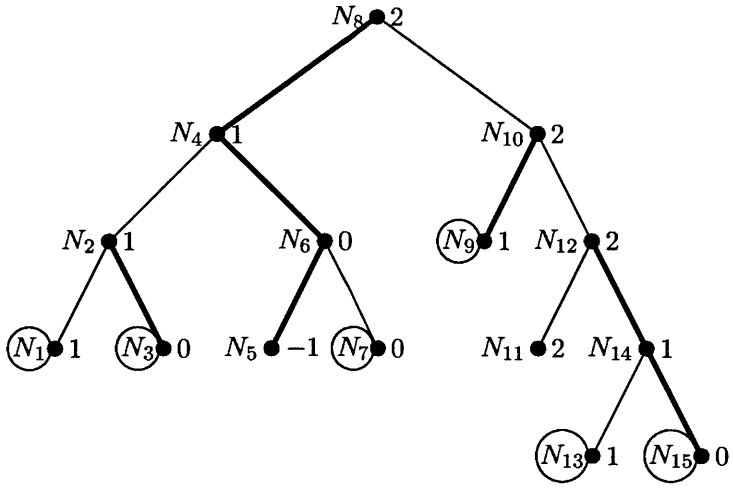


Figure 3.11: Potential calculation according to Theorem 17

Chapter 4

Localized Work Stealing

In multithreaded computations, it is conceivable 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. In this chapter, we consider a localized variant of the randomized work-stealing algorithm, henceforth called the *localized work-stealing algorithm*. An experiment by Acar et al. [1] shows that exploiting locality can improve the performance of the work-stealing algorithm by up to 80%.

This chapter focuses on theoretical results on the localized work-stealing algorithm. We show that under the "even distribution of free agents assumption", the expected running time of the algorithm is $T_1/P + O(T_\infty \lg P)$. 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 $T_1/P + O(T_\infty M)$.

The chapter is organized as follows. Section 4.1 introduces the setting that we consider throughout the chapter. Section 4.2 analyzes the localized work-stealing algorithm using the delay-sequence argument. Section 4.3 analyzes the algorithm using amortization arguments. Finally, Section 4.4 considers variants of the localized work-stealing algorithm.

4.1 Setting

This section introduces the setting that we consider throughout the chapter.

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 model the work of each processor as a binary tree whose leaves are the serial tasks of that processor. 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.

Recall from Chapter 2 that we define T_1 as the work of the computation, and T_∞ as the span of the computation. 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 task at the bottom. In particular, $T'_\infty < T_\infty$.

The randomized work-stealing algorithm [6] 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 $1 - \epsilon$.

Here we investigate 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

P_1's work	P_2's work	P_3's work	P_1's list	P_2's list	P_3's list	Action
P_1	P_2	P_3	–	–	–	–
P_1	P_2	–	–	–	–	P_3 finishes
P_1	P_2	P_2	–	P_3	–	P_3 steals randomly, finds P_2's work, P_2 adds P_3 to its list
P_1	–	P_2	–	P_3	–	P_2 finishes
P_1	P_2	P_2	–	P_3	–	P_2 checks its list, steals back, find its work from P_3
–	P_2	P_2	–	P_3	–	P_1 finishes
P_2	P_2	P_2	–	P_1, P_3	–	P_1 steals randomly, find P_2's work from P_3, P_2 adds P_1 to its list
P_2	–	P_2	–	P_1, P_3	–	P_2 finishes
P_2	P_2	P_2	–	P_1, P_3	–	P_2 checks its list, steals back randomly, find its work from P_1
–	–	P_2	–	P_1, P_3	–	P_1 and P_2 finish
–	–	P_2	–	P_1	–	P_2 checks its list, attempts to steal back from P_3, but (presumably) P_3 is down to one serial task, so P_2's steal is unsuccessful. P_2 removes P_3 from its list. Also, P_1 attempts to steal randomly from P_2 but is unsuccessful.
–	–	P_2	–	–	–	P_2 checks its list, attempts to steal back from P_1 but is unsuccessful. P_2 removes P_1 from its list. P_1 attempts to steal randomly from P_3 but is unsuccessful, again because P_3 is down to one serial task.
–	–	–	–	–	–	P_3 finishes

Table 4.1: Example of an execution of the localized work-stealing algorithm

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.

Table 4.1 shows an example of an execution of localized work-stealing algorithm, where the columns correspond to the owner of the work being done by P_i, P_i's list, and the action. We assume that the overhead for maintaining the list and dealing with contention for steal-backs is constant. Furthermore, we assume that all serial tasks cannot spawn within themselves.

4.2 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 [6] 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 number of steal attempts is $O(P^2(T_\infty + \lg(P/\epsilon)))$ with probability at least $1 - \epsilon$.

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.

Definition 18. *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 4.4. When the assumption does not hold, we obtain a weaker bound as given in Theorem 21.

Theorem 19. *With the even distribution of free agents assumption, the number of steal attempts is $O(P \lg P(T_\infty + \lg(P/\epsilon)))$ with probability at least $1 - \epsilon$, and the expected number of steal attempts is $O(P \lg PT_\infty)$.*

Proof. Consider processor 1. At timestep t, let S^t denote the number of general steals occuring at that timestep, and let X^t be the random variable

$$X^t = \begin{cases} 1/P^t, & \text{if processor 1 steals back from } P^t \text{ other processors;} \\ 0, & \text{if processor 1 is working.} \end{cases}$$

We define a *round* to be a consecutive number of timesteps t such that

$$\sum_t \left(S^t + PX^t\right) \geq P,$$

and such that this inequality is not satisfied if we remove the last timestep from the round. Note that this condition is analogous to the condition of a round in [6], where the number of steals is between $3P$ and $4P$. Here we have the term S^t corresponding to general steals and the term PX^t corresponding to steal-backs.

We define the *critical path* of a processor to be the path from the top of its binary tree to the serial task of the processor whose execution finishes last. We show that any round has a probability of at least $1 - 1/e$ of reducing the length of the critical path.

We compute the probability that a round does not reduce the length of the critical path. Each general steal has a probability of at least $1/P$ of stealing off the critical path and thus reducing its length. Each steal-back by the processor has a probability of $1/P^t$ of reducing the length of the critical path. At timestep t, the probability of not reducing the length of the critical path is therefore

$$\left(1 - \frac{1}{P}\right)^{S^t} \left(1 - X^t\right) \leq e^{-\frac{S^t}{P} - X^t},$$

where we used the inequality $1 + x \leq e^x$ for all real numbers x. Therefore, the probability of not reducing the length of the critical path during the whole round is at most

$$\prod_t e^{-\frac{S^t}{P} - X^t} = e^{-\sum_t \left(\frac{S^t}{P} - X^t\right)} \leq e^{-1}.$$

With this definition of a round, we can now apply the delay-sequence argument as in [6]. Note

that in a single timestep t, we have $S^t \leq P$ and $PX_t \leq P$. Consequently, in every round, we have $P \leq \sum_t \left(S^t + PX^t\right) \leq 3P$.

Suppose that over the course of the whole execution, we have $\sum_t \left(S^t + PX^t\right) \geq 3PR$, where $R = cT_\infty + \lg(1/\epsilon)$ for some sufficiently large constant c. Then there must be at least R rounds. Since each round has a probability of at most e^{-1} of not reducing the length of the critical path, the delay-sequence argument yields that the probability that $\sum_t \left(S^t + PX^t\right) \geq 3PR = \Theta(P(T_\infty + \lg(1/\epsilon)))$ is at most ϵ.

We apply the same argument to every processor. Suppose without loss of generality that processor 1's work is completed first, then processor 2's work, and so on, up to processor P's work. Let S_i denote the number of general steals up to the timestep when processor i's work is completed, and let X_i^t denote the value of the random variable X^t corresponding to processor i. In particular, S_P is the total number of general steals during the execution, which we also denote by S. We have

$$\Pr\left[S_i + \sum_t PX_i^t \geq \Theta(P(T_\infty + \lg(1/\epsilon)))\right] \leq \epsilon.$$

Now we use our even distribution of free agents assumption. This means that when processor i steals back, there are at most $(i-1)/(P-i+1)$ processors working on its work. Hence $X_i^t \geq (P-i+1)/(i-1)$ whenever $X_i^t \neq 0$. Letting W_i be the number of steal-backs performed by processor i, we have

$$\Pr\left[S_i + \frac{P(P-i+1)}{i-1} W_i \geq \Theta(P(T_\infty + \lg(1/\epsilon)))\right] \leq \epsilon.$$

For processor $2 \leq i \leq P-1$, this says

$$\Pr\left[\frac{i-1}{P(P-i+1)} S_i + W_i \geq \Theta\left(\frac{i-1}{P-i+1}(T_\infty + \lg(1/\epsilon))\right)\right] \leq \epsilon.$$

In particular, we have

$$\Pr\left[W_i \geq \Theta\left(\frac{i-1}{P-i+1}(T_\infty + \lg(1/\epsilon))\right)\right] \leq \epsilon.$$

For processor P, we have

$$\Pr\left[S + \frac{P}{P-1}W_P \geq \Theta(P(T_\infty + \lg(1/\epsilon)))\right] \leq \epsilon.$$

Since $P \geq P - 1$, we have

$$\Pr\left[S + W_P \geq \Theta(P(T_\infty + \lg(1/\epsilon)))\right] \leq \epsilon.$$

Since $\sum_{i=2}^{P-1} \frac{i-1}{P-i+1}$ grows as $P \lg P$, adding up the estimates for each of the P processors and using the union bound, we have

$$\Pr\left[S + \sum_{i=1}^{P} W_i \geq \Theta(P \lg P(T_\infty + \lg(1/\epsilon)))\right] \leq P\epsilon.$$

Substituting ϵ with ϵ/P yields the desired bound.

Since the tail of the distribution decreases exponentially, the expectation bound follows.

□

The bound on the execution time follows quickly from Theorem 19.

Theorem 20. *With the even distribution of free agents assumption, the expected running time, including scheduling overhead, is* $T_1/P + O(T_\infty \lg P)$. *Moreover, for any* $\epsilon > 0$, *with probability at least* $1 - \epsilon$, *the execution time on* P *processors is* $T_1/P + O(\lg P(T_\infty + \lg(P/\epsilon))$.

Proof. The amount of work is T_1, and Theorem 19 gives a bound on the number of steal attempts. We add up the two quantities and divide by P to complete the proof. □

Without the even distribution of free agents assumption, we obtain a weaker bound, as the following theorem shows.

Theorem 21. *The number of steal attempts is* $O(P^2(T_\infty + \lg(P/\epsilon)))$ *with probability at least* $1 - \epsilon$.

Proof. We apply a similar analysis using the delay-sequence argument as in Theorem 19. The

difference is that here we have $X_i^t \geq 1/P$ instead of $X_i^t \geq (P-i+1)/(i-1)$. Hence, instead of

$$\Pr\left[S_i + \frac{P(P-i+1)}{i-1} W_i \geq \Theta(P(T_\infty + \lg(1/\epsilon)))\right] \leq \epsilon,$$

we have

$$\Pr\left[S_i + W_i \geq \Theta(P(T_\infty + \lg(1/\epsilon)))\right] \leq \epsilon.$$

The rest of the analysis proceeds using the union bound as in Theorem 19. □

Remark 22. *In the delay-sequence argument, it is not sufficient to consider the critical path of only one processor (e.g. the processor that finishes last).*

For example, suppose that there are 3 processors, P_1, P_2, and P_3. P_1 owns 50 serial tasks of size 1 and 1 serial task of size 100, P_2 owns 1 serial task of size 1 and 1 serial task of size 1000, and P_3 owns no serial task. At the beginning of the execution, P_3 has a probability of 1/2 of stealing from P_1. If it steals from P_1 and gets stuck with the serial task of size 100, P_1 will perform several steal-backs from P_3, while the critical path is on P_2's subtree.

Hence, the steal-backs by P_1 do not contribute toward reducing the length of the critical path.

4.3 Amortization Analysis

In this section, we apply amortization arguments to obtain bounds on the running time of the localized work-stealing algorithm. We show that 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 $T_1/P + O(T_\infty M)$.

We begin with a simple bound on the number of steal-backs.

Theorem 23. *The number of steal-backs is at most $T_1 + O(PT_\infty)$ with high probability.*

Proof. Every successful steal-back can be amortized by the work done by the stealer in the timestep following the steal-back. Every unsuccessful steal-back can be amortized by a general steal. Indeed, recall our assumption that after each unsuccessful steal-back, the target processor is removed from

the owner's list. Hence each general steal can generate at most one unsuccessful steal-back. Since there are at most $O(PT_\infty)$ general steals with high probability, we obtain the desired bound. □

The next theorem amortizes each steal-back against general steals, using the height of the tree to estimate the number of general steals.

Theorem 24. *Let N denote the number of general steals in the computation. Recall from Chapter 2 that T'_∞ denotes the height of the tree not including the part connecting the P processors of height $\lg P$ at the top or the serial task on the bottom. (In particular, $T'_\infty < T_\infty$.) Then there are at most $T'_\infty N$ steal-back attempts.*

Proof. Suppose that a processor P_i steals back from another processor P_j. This means that earlier, P_j performed a general steal on P_i which resulted in this steal-back. We amortize the steal-back against the general steal. Each general steal generates at most T'_∞ steal-backs (or $T'_\infty + 1$, to be more precise, since there can be an unsuccessful steal-back after P_j completed all of P_i's work and P_i erased P_j's name from its list.) Since there are N general steals in our computation, there are at most $T'_\infty N$ steal-back attempts.

After P_j performed the general steal on P_i, it is possible some other processor P_k makes a general steal on P_j. This does not hurt our analysis. When P_i steals back from P_k, we amortize the steal-back against the general steal that P_k makes on P_j, not the general steal that P_j makes on P_i. □

Since there are at most $O(PT_\infty)$ general steals with high probability, Theorem 24 shows that there are at most $O(T'_\infty PT_\infty)$ steals in total with high probability.

The next theorem again amortizes each steal-back against general steals, but this time also using the size of the serial tasks to estimate the number of general steals.

Theorem 25. *Define N and T'_∞ as in Theorem 24, and let X be any positive integer. Remove a total of at most X serial tasks from consideration (For example, it is a good idea to exclude the largest or the smallest serial tasks.) For each processor i, let M_i denote the ratio between its largest and the smallest serial tasks after the removal. Let $M = \max_i M_i$. Then the total number of steal-back attempts is $O(N \min(M, T'_\infty)) + T'_\infty X$.*

Proof. There can be at most $T'_\infty X$ steal-backs performed on subtrees that include one of the X serial tasks, since each subtree has height at most T'_∞.

Consider any other steal-back that processor P_i performs on processor P_j. It is performed against a subtree that does not include one of the X serial tasks. Therefore, it obtains at least $1/(M+1)$ of the total work in that subtree, leaving at most $M/(M+1)$ of the total work in P_j's subtree. We amortize the steal-back against the general steal that P_j performed on P_i earlier.

How many steal-backs can that general steal generate? P_i can only steal back at most half of P_j's work (since P_j is working all the time, and thus will finish half of its work by the time P_i steals half of its work). To obtain the estimate, we solve for K such that

$$\left(\frac{M}{M+1}\right)^K = \frac{1}{2},$$

and we obtain

$$K = \frac{\lg 2}{\lg(M+1) - \lg(M)}.$$

By integration, we have

$$\int_M^{M+1} \frac{1}{M+1} dx < \int_M^{M+1} \frac{1}{x} dx < \int_M^{M+1} \frac{1}{M} dx,$$

so that

$$\frac{1}{M+1} < \ln(M+1) - \ln(M) < \frac{1}{M},$$

or

$$M < \frac{1}{\ln(M+1) - \ln(M)} < M+1.$$

Since lg and ln are off each other by only a constant factor, K grows as $O(M)$. This means that one random steal will be amortized against at most $O(M)$ steal-backs. Combined with the estimate involving T_∞ from Theorem 24, we have the desired bound, assuming that there are no general steals performed on P_i or P_j during these steal-backs.

Now we show that this last assumption is in fact unnecessary. That is, if there are general steals performed on P_i or P_j during these steal-backs, our estimate still holds. If a general steal is

performed on P_i after P_i steals back from P_j, we amortize this steal-back "in a special way" against this general steal instead of against the general steal that P_j made on P_i. Since each general steal can be amortized against "in a special way" by at most one steal-back, our estimate holds.

On the other hand, if a general steal is performed on P_j, then the steal-backs that P_i has performed on P_j become an even higher proportion of P_j's work, and the remaining steal-backs proceed as usual. So our estimate also holds in this case. □

Applying Theorem 25, we may choose $O(P)$ serial tasks to exclude from the computation of M without paying any extra "penalty", since the penalty $O(PT'_\infty)$ is the same as the number of general steals. After we have excluded these serial tasks, if M turns out to be constant, we obtain the desired $O(PT_\infty)$ bound on the number of steal-backs. The next theorem formalizes this fact.

Theorem 26. *Define N and T'_∞ as in Theorem 24, and remove any $O(P)$ serial tasks from consideration. For each processor i, let M_i denote the ratio between its largest and the smallest serial tasks after the removal. Let $M = \max_i M_i$. Then the expected execution time on P processors is $T_1/P + O(T_\infty \min(M, T'_\infty))$.*

Proof. The amount of work is T_1, and Theorem 25 gives a bound on the number of steal-back attempts in terms of the number of steal attempts. Since we know that the expected number of steal attempts is $O(PT_\infty)$, the expected number of steal-back attempts is $O(PT_\infty \min(M, T'_\infty))$. We add this to the amoung of work and divide by P to complete the proof. □

Remark 27. *In the general case, it is not sufficient to amortize the steal-backs against the general steals. That is, there can be (asymptotically) more steal-backs than general steals, as is shown by the following example.*

Suppose that the adversary has control over the general steals. When there are k owners left, the adversary picks one of them, say P_i. The other $k-1$ owners are stuck on a large serial task while P_i's task is being completed. The $P-k$ free agents perform general steals so that P_i's tree is split evenly (in terms of the number of serial tasks, not the actual amount of work) among the $P-k+1$ processors. Then P_i finishes its work, while the other $P-k$ processors are stuck on a large serial task. P_i performs repeated steal-backs on the $P-k$ processors until each of them is

only down to its large serial task. Then they finish, and we are down to $k-1$ owners. In this case, $O(P^2T_\infty)$ steal-backs are performed, but only $O(P^2)$ general steals.

In particular, it is not sufficient to use the bound on the number of general steals as a "black box" to bound the number of steal-backs. We still need to use the fact that the general steals are random.

Neither the delay-sequence argument in Section 4.2 nor the amortization arguments in Section 4.3 settles the question of whether the high-probability bound on the number of steals in the original work-stealing algorithm also holds in the localized work-stealing algorithm, i.e., whether the number of steal attempts is $O(P(T_\infty + \lg(P/\epsilon)))$ with probability at least $1-\epsilon$. In fact, it is not even clear intuitively whether the statement might hold or not. The question remains open to the best of our knowledge.

4.4 Variants

In this section, we consider two variants of the localized work-stealing algorithm. The first variant, hashing, is designed to alleviate the problem of pile-up in the localized work-stealing algorithm. It assigns an equal probability in a steal-back to each owner that has work left. In the second variant, mugging, a steal-back takes all or almost all of the work of the processor being stolen from. A simple amortization argument yields an expected number of steals of $O(PT_\infty)$.

Hashing

Intuitively, the way in which the general steals are set up in the localized work-stealing algorithm supports pile-up on certain processors' work. Indeed, if there are several processors working on processor P_1's work, the next general steal is more likely to get P_1's work, in turn further increasing the number of processors working on P_1's work.

A possible modification of the general steal, which we call *hashing*, is as follows: first choose an owner uniformly at random among the owners who still has work left, then choose a processor that is working on that owner's work uniformly at random.

Loosely speaking, this modification helps in the critical path analysis both with regard to the general steals and to the steal-backs. Previously, if there are k owners left, a general steal has a $\frac{k}{P}$ probability of hitting one of the k remaining critical paths. Now, suppose there are $P_1, P_2, \ldots, P_k$ processors working on the k owners' work, where $P_1 + \ldots + P_k = P$. The probability of hitting one of the critical paths is

$$\frac{1}{k}\left(\frac{1}{P_1} + \ldots + \frac{1}{P_k}\right) \geq \frac{k}{P}$$

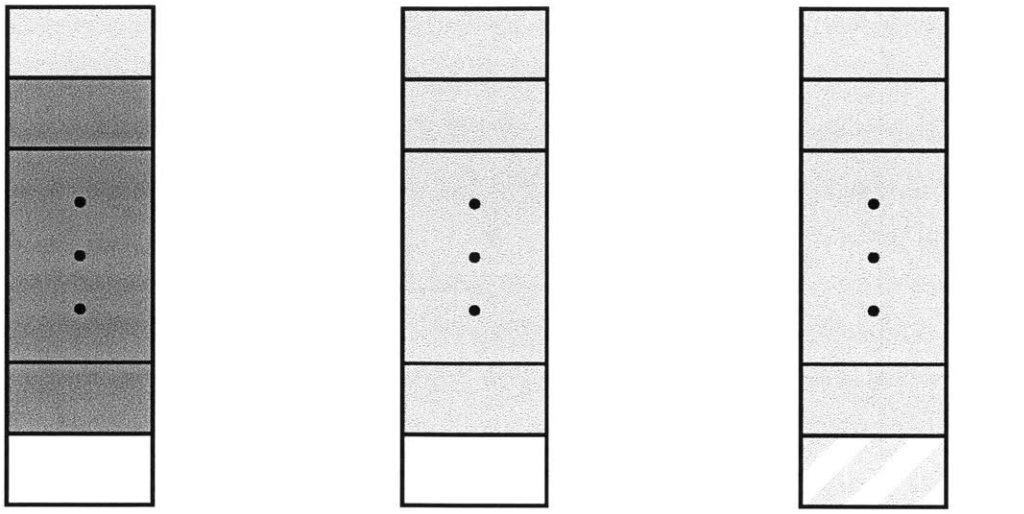
by the arithmetic-harmonic mean inequality [14]. Also, the modified algorithm chooses the owner randomly, giving each owner an equal probability of being stolen from.

Mugging

A possible modification of the steal-back, which we call *mugging*, is as follows: instead of P_i taking only the top thread from P_j's deque during a steal-back (i.e. half the tree), P_i takes either (1) the whole deque, except for the thread that P_j is working on; or (2) the whole deque, including the thread that P_j is working on (in effect preempting P_j.) Figure 4.1 shows the processor of P_j in each of the cases.

Figure 4.1(a) corresponds to the unmodified case, Figure 4.1(b) to case (1), and Figure 4.1(c) to case (2). The yellow threads are the ones that P_i steals from P_j, while the white threads are the ones that P_j is working on. In Figure 4.1(c), the bottom thread is preempted by P_i's steal.

In both modifications here, if we still assume as before that that all serial tasks cannot spawn within themselves, then each general steal can generate at most one steal-back. Therefore, the expected number of steal-backs is $O(PT_\infty)$, and the expected number of total steals is also $O(PT_\infty)$.



(a) Work stealing (b) Variant (1) of mugging (c) Variant (2) of mugging

Figure 4.1: Deque of processor in variants of mugging

Chapter 5

Conclusion and Future Work

Work stealing is a fascinating paradigm from which many interesting questions arise. Research in this area has exploded since Blumofe and Leiserson's important contribution [6], but there are still further directions to explore. This thesis has led the way on two of those directions: the worst-case analysis of successful steals, and the analysis of the localized work-stealing algorithm.

As for the analysis of successful steals, it is interesting to consider the extent to which we can generalize the setting. This thesis has restricted the attention to computation trees, but in general, computations need not be trees. Can we prove similar results about the more general case of computation DAGs? What sort of upper bounds can we show?

Another natural direction is to analyze the expected or high-probability bound on the number of successful steals. We know from [6] that the expected number of total steal attempts, whether successful or not, is $O(PT_\infty)$. How close is the upper bound on the number of successful steals to that bound? If the two bounds differ, can we determine the particular cases in which they differ?

The main question on the localized work stealing algorithm is whether we can prove or disprove the expected running time bound of $T_1/P+O(T_\infty)$. We have found this question to be difficult, and we have not seen theoretical evidence that strongly suggests the answer one way or the other. It would also be interesting to rigorously analyze the performance of the hashing variant and compare it to our vanilla algorithm.

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–12, July 2000.

[2] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. An empirical evaluation of work stealing with parallelism feedback. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006)*, July 2006.

[3] Kunal Agrawal, Charles E. Leiserson, Yuxiong He, and Wen Jing Shu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems (TOCS)*, 26(3), September 2008.

[4] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Executing task graphs using work-stealing. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS 2010)*, 2010.

[5] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119–129, June 1998.

[6] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[7] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, January 1997.

[8] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 187–194, 1981.

[9] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 21–28, July 2005.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, July 2009.

[11] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, November 2009.

[12] R. Feldmann, P. Mysliwietz, and B. Monien. Game tree search on a massively parallel system. *Advances in Computer Chess*, pages 203–219, 1993.

[13] Raphael Finkel and Udi Manber. DIB—a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[14] Philip Wagala Gwanyama. The HM-GM-AM-QM inequalities. *The College Mathematics Journal*, 35(1):47–50, January 2004.

[15] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software, and Applications*, October 1994.

[16] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 9–17, 1984.

[17] Richard M. Karp and Yanjun Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, July 1993.

[18] Bradley C. Kuszmaul. *Synchronized MIMD Computing.* PhD thesis, Massachusetts Institute of Technology, May 1994.

[19] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[20] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 237–245, July 1991.

[21] N. Tomizawa. On some techniques useful for solution of transportation network problems. *Networks*, 1(2), 1972.

[22] Mark T. Vandevoorde and Eric S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

[23] Y. Zhang and A. Ortynski. The efficiency of randomized parallel backtrack search. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, October 1994.