Non-Blocking Steal-Half Work Queues

Danny Hendler
Tel-Aviv University

Nir Shavit
Tel-Aviv University [0]

Abstract

The non-blocking work-stealing algorithm of Arora et al. has been gaining popularity as the multiprocessor load balancing technology of choice in both Industry and Academia. At its core is an ingenious scheme for stealing a single item in a non-blocking manner from an array based deque. In recent years, several researchers have argued that stealing more than a single item at a time allows for increased stability, greater overall balance, and improved performance.

This paper presents *StealHalf*, a new generalization of the Arora et al. algorithm, that allows processes, instead of stealing one, to steal up to half of the items in a given queue at a time. The new algorithm preserves the key properties of the Arora et al. algorithm: it is non-blocking, and it minimizes the number of CAS operations that the local process needs to perform. We provide analysis that proves that the new algorithm provides better load distribution: the expected load of any process throughout the execution is less than a constant away from the overall system average.

1 Introduction

The work-stealing algorithm of Arora et al. [2] has been gaining popularity as the multiprocessor load-balancing technology of choice in both Industry and Academia [1, 2, 5, 7]. The scheme allows each process to maintain a local work queue, and steal an item from others if its queue becomes empty. At its core is an ingenious scheme for stealing an individual item in a non-blocking manner from a bounded size queue, minimizing the need for costly CAS (Compare-and-Swap) synchronization operations when fetching items locally.

Though stealing one item has been shown sufficient to optimize computation along the "critical path" to within a constant factor [2, 4], several authors have argued that the scheme can be improved by allowing multiple items to be stolen at a time [3, 8, 9, 12]. Unfortunately, the only implementation algorithm for stealing multiple items at a time, due to Rudolph et al. [12], requires the local process owning the queue to use a strong synchronization operation (CAS or some other form of mutual exclusion) *for every push or pop*. Since the local process' operations are far more frequent, this makes the Rudolph et al. algorithm significantly less effective than that of Arora et al.

This state of affairs leaves open the question of designing an algorithm that, like Arora et al., does not use costly synchronization operations in every step, yet allows stealing multiple items at a time, thus achieving the stability, balance, and improved performance of algorithms like Rudolph et al.

[0]This work was supported in part by a grant from Sun Microsystems. Contact author: hendlerd@post.tau.ac.il

PODC 2002, July 21-24, 2002, Monterey, California, USA.
Copyright 2002 ACM 1-58113-485-1/02/0007...$5.00.

1.1 The New Algorithm

This paper presents a new work-stealing algorithm, *StealHalf*, a generalization of the Arora et al. algorithm, that allows processes to steal up to half of the items in a given queue at a time. Our StealHalf algorithm preserves the key properties of the Arora et al. algorithm: it is non-blocking and it minimizes the number of CAS operations that the local process needs to perform. We provide a performance analysis showing that our algorithm improves on Arora et al. by providing an overall system balance similar to the Rudolph et al. scheme: the expected load of any process throughout the execution is less than a constant away from the overall system average.

In our algorithm, as in the Arora et al. algorithm, each process has a local work queue. This queue is actually a deque, allowing the local process to push and pop items from the bottom, while remote processes steal items from the top. The Arora et al. algorithm is based on a scheme that allows the local process, as long as there is more than one item in the deque, to modify the bottom counter without a CAS operation. Only when the top and bottom are a distance of 1 or less apart, so that processes potentially overlap on the single item that remains in the deque, does the process need to use a CAS operation to perform consensus. This consensus is necessary because the inherent uncertainty [6] regarding a single read or write operation to the top and bottom counters can affect whether the last item to be stolen is still there or not. This yields an algorithm where for any monotonic sequence of k pushes or k pops,[1] the number of

[1]A monotonic sequence is a sequence of local operations that monotonically either increases or decreases the bottom counter. We define the synchronization complexity in terms of monotonic sequences since one cannot make claims in non-monotonic situations where the execution pattern of the underlying application causes thrashing back and forth on a single location.

CAS operations is $\Theta(1)$.

In order to steal more than one item at a time, say half of the number of items in a deque, one must overcome a much greater uncertainty. Since a stealing process might remove up to half of the items, there are now many values of the bottom counter, say, up to half of the difference between top and bottom, for which there is a potential overlap. Thus, eliminating the uncertainty regarding reading and writing the counter requires consensus to be performed for half the items in the deque, an unacceptable solution from a performance point of view, since it would yield an algorithm with $\Theta(k)$ synchronization complexity.

The key to our new algorithm is the observation that one can limit the uncertainty so that the local process needs to perform a consensus operation (use a CAS), only when the number of remaining items in the deque (as indicated by the difference between top and bottom), is a power of two! The scheme works so that as the distance between the bottom and top counters changes, it checks and possibly updates a special half-point counter. The uncertainty regarding a single read or write exists just as in the Arora et al. algorithm, only now it happens with respect to this counter. What we are able to show is that missing the counter update can only affect locations beyond the next power-of-two at any given point. For any monotonic sequence of k pushes or k pops, our algorithm uses only $\Theta(\log k)$ CAS operations, slightly worse than Arora et al., but exponentially better than the $\Theta(k)$ of the Rudolph et al. scheme.

Like the Arora et al. scheme, our StealHalf algorithm is non-blocking: the slowdown of any process cannot prevent the progress of any other, allowing the system as a whole to make progress, independently of process speeds. Like Arora et al., the non-blocking property in our algorithm is not intended to guarantee fault-tolerance: the failure of a process can cause loss of items. Arora et al. claim that their empirical testing on various benchmarks shows that the non-blocking property contributes to performance of work stealing, especially in multiprogrammed systems [2].

In our algorithm, the price for unsuccessful steal operations is a redundant copying of multiple item-pointers, whereas in the Arora algorithm only a single pointer is copied redundantly. However in our algorithm, successful steals transfer multiple items at the cost of a single CAS, whereas in the Arora et al. scheme a CAS is necessary for each stolen item.

1.2 Performance Analysis

Mitzenmacher [9] uses his differential equations approach [10] to analyze the behavior of work stealing algorithms in a dynamic setting, showing that in various situations stealing more than one item improves performance. Berenbrink et al. [3] have used a markov model to argue that a system that steals only one item at a time can slip into an instable state from which it cannot recover, allowing the number of items to grow indefinitely. This implies that no matter how much buffer space is allocated, at some point the system may overflow. Treating such overflow situations requires the use of costly overflow mechanisms [5]. Berenbrink et al. [3] further show that an Arora-like scheme that allows stealing, say, half of the items, will prevent that system from slipping into such an instable state. Rudolph et. al [12], and later Luling and Monien [8], prove that in a load balancing scheme that repeatedly balances the work evenly among random pairs of local work queues, the expected load of each process will vary only by a constant factor from the load of any other process and that the overall variance is small.

The algorithm we present is generic, in the sense that one can change the steal initiation policy to implement a variety of schemes including those of [2, 3, 12]. We choose to analyze its behavior in detail under a steal-attempt policy similar to [12]: ever so often, every process p performs the following operation, which we call *balancing initiation*: p flips a biased coin to decide if it should attempt to balance its work-load by stealing multiple items, where the probability of attempting is inversely proportional to its load. If the decision is to balance, then p randomly selects another process and attempts to balance load with it. Our main Theorem states that our algorithm maintains properties similar to those of Rudulph et al. We provide it here since the proof provided by Rudolph et al. turns out to be incomplete, and also because there are significant differences between the algorithms. As an example, the Rudolph et al. algorithm is symmetric: a process can both steal-items-from and insert-items-to the deque of another process. In our algorithm, however, a process can only steal items.

Assume Δ_u is a time period where no deque grows or shrinks by more than u items (through pushing or popping). Let $L_{p,t}$ denote the number of items in process p's deque at time t; also, let A_t denote the average-load at time t. Then if all processes perform balancing initiation every Δ_u time, there exists a constant α_u, not depending on the number of processes or the application, such that:

$$\forall p, t : E[L_{p,t}] < \alpha_u A_t$$

The existence of such Δ_u is a natural assumption in most systems, and can be easily maintained with slight protocol modifications in others. In contrast, it can easily be shown, that if one steals a single item at a time as in the Arora et al. scheme, even if steal-attempts are performed every Δ_1, there are scenarios in which the system becomes instable.

In summary, we provide the first algorithm for stealing multiple items using a single CAS operation, without requiring a CAS for every item pushed or popped locally.

An outline of the algorithm is presented in Section 2; the analysis is presented in Section 3; finally, correctness claims are presented at Section 4.

2 The StealHalf algorithm

As noted earlier, the key issue in designing an algorithm for stealing multiple items is the need to minimize the use of strong synchronization operations both when performing local operations and when stealing multiple items.

A first attempt at an algorithm might be to simply let a process steal a range of items by repeatedly performing the original code of stealing a single item for every item in the group. This would make the algorithm a steal-many algorithm[2] but at a very high cost: to steal k items, the thief would have to perform k synchronization operations. The algorithm presented here, utilizes an *extended deque* data structure, a variant of the deque of [2] depicted in Figure 1, to achieve synchronization at a low cost. The *extended deque* differs from the deque of [2] in two ways: (1) it is implemented as a cyclic array, and (2) it contains a member-structure, called *stealRange*, which defines the range of items that can be stolen atomically by a thief-process.

[2]Obviously with this technique the items-group is not stolen atomically.

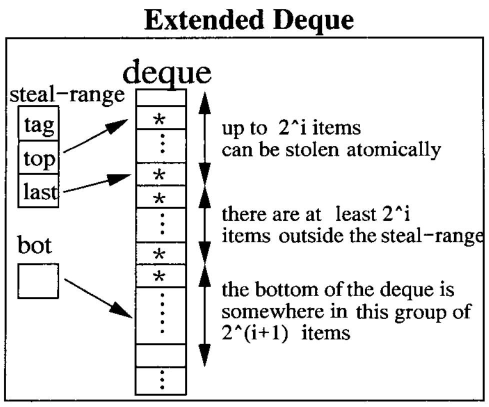

Figure 1: The extended deque

The algorithm allows stealing about half the items of the victim in a single synchronization operation. It does this by always maintaining the invariant that the number of items in every process p's *stealRange* is approximately half the number of total items in p's deque. Process p's *stealRange* is updated by any process that succeeds in stealing items from p's deque, and also by p itself. To keep the number of synchronization operation performed by a local process p as low as possible, p updates its *stealRange* when pushing or popping an item to/from its deque, only if at least one of the following events occur:

- The number of items in p's deque crosses a 2^i boundary, for some $i \in N$;
- A successful-steal operation from p's deque has occurred since the last time p modified its own *stealRange.*

2.1 Data structures

Each process owns an extended deque structure, which it shares with all other processes. In this structure: *deq* is an array that stores pointers or handles to items, which are generated and consumed dynamically. It has *DEQ_SIZE* entries. The range of occupied entries in *deq* includes all the entries in the range: $[top \cdots bot)_{DEQ\_SIZE}$[3]. As noted, it is implemented as a cyclic array.

The *stealRange* member-structure extends the *age* structure from [2]. It contains the following fields:

- *tag* - as in [2], a time stamp on the updating of the *stealRange* structure. A tag is required in order to overcome the *ABA* problem inherent to the *CAS* primitive. [4] As in [2], the bounded tag algorithm of [11] can be used.
- *top* - as in [2], the pointer to the top-most item in the deque. [5]

[3] $[a \cdots b)_m$ denotes the half-open entries-range, from entry a up-to entry b (including a but not b), modulus m.

[4] Assume a process p reads a value A from location m, and then performs a successful *CAS* operation, which modifies the contents of m. Process p's CAS should ideally succeed only if the contents of m was not modified since it read value A. The problem is, that if other processes modified m's value to value B and then back again to A in the interim - the CAS operation still succeeds. To overcome this problem, a tag field is added to the structure.

[5] Note, that because the extended deque is cyclic, the item pointed to by *top* is not necessarily the item stored at the top-most deque-entry.

The item pointed at by top is invariably the first item of the stealRange.

- *stealLast* - points to the last item to be stolen. The items that would be stolen next (unless the strealRange structure would be modified by the local process before the next steal) are in the range $[top \cdots stealLast]_{DEQ\_SIZE}$[6]. The *stealLast* variable can assume the special value *null*, which indicates there are no items to steal. The stealRange structure is modified via *CAS* operations.

The *bot* field points to the entry following the last entry containing an item. Since the extended deque is cyclic, *bot* may be smaller than *top*. If *bot* and *top* are equal, the extended deque is empty. In addition to the shared extended-deque structure, each process has a static and *local* structure, called *prevStealRange*, of the same type as *stealRange*. It is used by the local process to determine whether a steal has occurred since the last time the process modified the stealRange.

2.2 High-level extended-deque methods description

We specify the algorithm in a generic manner, that allows "plugging-in", in a modular way, components that allow flexibility in regard to the conditions/policy that control when a steal-attempt is initiated.

2.2.1 Balancing initiation code

The code that appears in figure 2 should be performed by every process periodically, throughout the computation.

```
IF (shouldBalance())
    {
    Process *victim=randomProcess();
    TryToSteal(victim);
    }
```

Figure 2: steal-initiation code

The *shouldBalance* method determines the policy regarding when to initiate a steal attempt[7]. Many policies are conceivable, a few of which are:

- Try to steal only when the local deque is empty: this is the scheme implemented by Arora et al. [2], and we call it: *steal-on-empty.*
- Try to steal probabilistically, with the probability decreasing as the number of items in the deque increases: this scheme was suggested by Rudolph et al. [12], and we call it *probabilistic balancing*[8]
- Try to steal whenever the number of items in the deque increases/decreases by a constant factor from the last time a steal-attempt was performed: this is the policy suggested in [3].

[6] $[a \cdots b]_m$ denotes the closed array-range, from entry a up-to entry b (including both a and b), modulus m.

[7] This can be implemented as inlined code rather than as a method, if this code should be performed very often.

[8] The scheme, as described, is always probabilistic in the sense that the victim process is selected at random. The *probabilistic balancing* scheme adds yet another probabilistic factor.

Steal-initiation policies can vary significantly with respect to the extent they make the system balanced, but obviously none can guarantee that the system is balanced, if balancing is not attempted frequently enough. Consequently, we have to make some reasonable assumptions regarding the frequency at which the steal-initiation code is performed.

Let Δ_u be a time-period small enough, such that it is guaranteed that no process p changes its load by more than u items during that period, by generating/consuming items (thefts notwithstanding). In the analysis we prove, that if the *probabilistic balancing* policy is employed, and the steal-initiation code is performed once every Δ_u period, then the expected number of deque-items of *any* process p at *any* time during the execution is no more than α_u times the total average, where α_u is a constant that does not depend on the number of processes or the dynamic pattern of item generation/consumption[9].

Not every steal-initiation policy can guarantee this property, though. It can easily be shown, that under the *steal-on-empty* policy, where only a single item is stolen at a time, a system can grow unbalanced beyond any bound, even if the steal-initiation code is performed every Δ_1 time period!

The balancing initiation code can be performed periodically by the system or by an application, or it can be implemented as a signal-handler.

2.2.2 pushBottom code

A high-level pseudo-code of *pushBottom* is shown in Figure 3.

```
RETURN_CODE pushBottom(Item *e) {
    IF deq is full
      return DEQ_FULL
    deq[bot] = e
    increment bot in a cyclic manner
    IF (deq length now equals 2^i for some i)
      Try to CAS-update the stealRange to
      contain max(1,2^(i-1)) items
    ELSE IF (stealRange != prevStealRange)
      Try to CAS-update the stealRange to contain max(1,2^i)
      items, where the current length of the deque is in
      the range [ 2^(i+1) ,2^(i+2) )

   IF CAS was performed successfully
      prevStealRange = stealRange
}
```

Figure 3: pushBottom pseudo code

The *pushBottom* operation is performed by the local process p, whenever it needs to insert a new item into its local deque. If the deque is not full, the new item is placed in the entry pointed at by *bot*, and *bot* is incremented in a cyclic manner. However, if following the insertion of the new item the length of p's deque becomes 2^i for some $i \geq 0$, then p tries to CAS-update its *stealRange* to contain the topmost 2^{i-1} items[10]. This is to make sure the length of the *stealRange* is not much less than half the total number of items in the deque[11]

Process p has to update the *stealRange* even if the deque-length is *not* a power of 2, if another process has succeeded in stealing items from p since the last time p updated its stealRange. Process p can identify this by comparing the current value of its *stealRange* with the last value it wrote to it - which is stored at its local *prevStealRange*. If the values differ, p tries to set the length of its *stealRange* (by a CAS) to be $max(1, 2^i)$, where $2^{i+1} \leq len(deq) < 2^{i+2}$.

If p performed a successful CAS, *prevStealRange* is updated with the value it wrote.

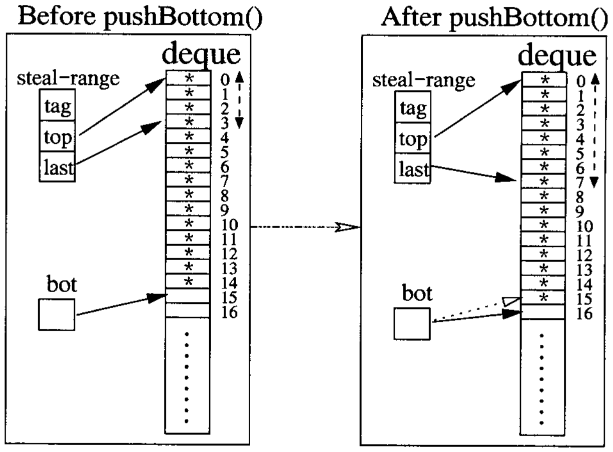

Figure 4: *The extended deque before and after a pushBottom.* Right after *pushBottom* inserts a new item into the deque, it checks whether the deque length reaches a power-of-2 boundary (in the above Figure, the 16'th item was added), in which case *stealRange* is expanded to contain the first half of the deque-elements.

Figure 4 depicts the state of an extended deque before and after the 16'th item is pushed into it. Since subsequent to the push the deque contains a power-of-2 number of items, *pushBottom* tries (and in this case succeeds) to expand *stealRange*. It is easily seen (and is proven in our analysis) that immediately after every successful CAS operation performed by *pushBottom*, assuming that the pushed item is not the first item in the deque[12], the following holds:

$$2 * len(stealRange) \leq len(deq) < 4 * len(stealRange)$$

2.2.3 popBottom code

A high-level pseudo-code of *popBottom* is presented in Figure 5. The *popBottom* operation is performed by the local process p whenever it needs to consume another deque-item.

- *Section 1*: In section 1 a process performs some pre-checks to make sure the method may proceed. It first checks whether the deque is empty, in which case the method returns *null*; it next checks whether the length of the *deque* is 2^i for some $i \geq 0$. If this is indeed the case, then the method tries to CAS-update its *stealRange* to contain the topmost 2^{i-2} items[13]. This is done in order to maintain the invariant that the length of *stealRange* never exceeds half the total number of

[9]Obviously, Δ_u itself *does* depend on this pattern.

[10]or 1, if i=0.

[11]We actually prove in the full paper that for any process p, at any time, the length of p's *stealRange* is at least $\frac{1}{8}$'th of the length of p's deque.

[12]If the pushed item is the first item of the deque, then the lengths of both the deque and the *stealRange* right after the CAS equal 1.

[13]or 1, if $i < 2$.

```
Item *popBottom()
{
Section 1
---------
      IF deq is empty
         return null
      IF (deq length now equals 2^i for some i)
        Try to CAS-update the stealRange to contain
        max(1,2^(i-2)) items
      ELSE IF (stealRange != prevStealRange)
        Try to CAS-update the stealRange to contain max(1,2^i)
        items, where the current length of the deque
        is in the range ( 2^(i+1) ... 2^(i+2) ]

      IF CAS was performed successfully
         prevStealRange = stealRange
      ELSE IF a CAS was attempted but failed
         return ABORT;

Section 2
---------
      Decrement bot in a cyclic manner
      Item *e = deq[bot]
      oldStealRange = this->stealRange

      IF oldStealRange does not contain bot
         return e (no need to synchronize)
      ELSE IF oldStealRange is empty
         {
         bot=0 (the last item - e - was already stolen
                by another process)
         return null
         }
      ELSE
         {
         Try to CAS-update the stealRange to be empty
         IF succeeded
            return e (e was not stolen so far)
         ELSE
            return null (the last item - e - was already
                         stolen by another process)
         }
  }
```

Figure 5: popBottom pseudo code

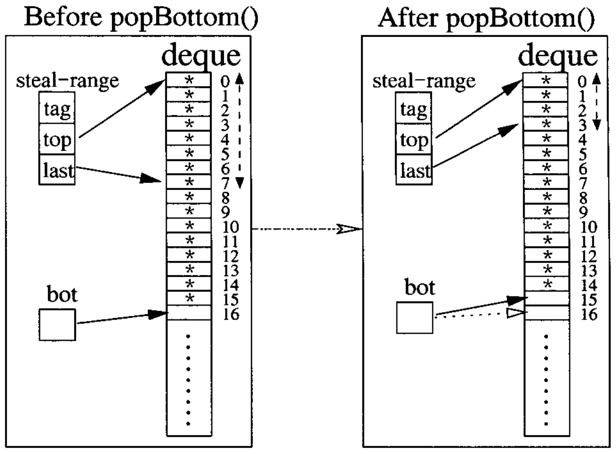

Figure 6: *The extended deque before and after a popBottom.* Just before *popBottom* pops the bottom-most item from the deque, it checks whether the deque length is exactly at a power-of-2 boundary (in the above Figure, the 16'th item is about to be popped), in which case *stealRange* is being shrank to contain the first quarter of the deque-elements

items in the deque, unless there's a single item in the deque.

As in *pushBottom*, p has to CAS-update the *stealRange* even if the deque-length is *not* a power of 2, if another process has succeeded in stealing items from p since the last time p updated its stealRange. Process p can identify this by comparing the current value of its *stealRange* with the last value it wrote to it - which is stored at *prevStealRange.* If the values differ, p tries to set the length of its *stealRange* to be $max(1, 2^i)$, where $2^{i+1} < len(deq) \leq 2^{i+2}$.

If *popBottom* performed a successful CAS, it updates *prevStealRange* with the value it wrote. If however a CAS was attempted and failed, *popBottom* returns a special value ABORT, indicating this situation. Note, that this can only happen if a successful steal has occurred concurrently with the method's execution.[14].

- *Section 2*: Section 2 is very similar to its steal-one [2] counterpart: it pops the bottom-item e off the deque, and then reads *stealRange.* If this read does *not* include e, then the method returns e and exits, as no other process may have stolen e; otherwise, there are 2 possibilities:
  - *stealRange* is empty - e was stolen by another process during the method's execution. The method returns *null.*
  - e is the one-and-only item in the deque, in which case the method tries to CAS-update stealRange with an empty range value. If it succeeds - it returns e as its result, otherwise it returns *null.*

Figure 6 depicts the state of an extended deque before and after the 16'th item is popped off it. Since prior to the pop the deque contains a power-of-2 number of items, *popBottom* tries to shrink *stealRange* (and in this case succeeds). It is easily seen (and is proven in our analysis) that immediately after every successful CAS operation performed by *popBottom* the following inequalities hold[15]

$$2 * len(stealRange) \leq len(deq) \leq 4 * len(stealRange)$$

2.2.4 tryToSteal code

The *tryToSteal* method is the method that actually attempts to perform a steal. It is called by a local process p, after the steal-initiation policy has determined that a steal-attempt should be made and a victim process has been selected. The pseudo-code of *tryToSteal* is shown in Figure 7.

The method receives 2 parameters: d - a pointer to the victim process' extended-deque, and *pLen* - the number of items in p's deque, and returns the number of items actually stolen. It first reads *d.stealRange* and computes its length,

[14]An alternative implementation is to retry again and again in a loop, until the method succeeds in updating *stealRange*.

[15]The below inequalities hold, unless the number of items in the deque after the pop is 0 or 1. If the pop empties the deque, then the lengths of both the deque and the *stealRange* after the operation are 0; if a single item is left in the deque after the pop, the lengths are both 1.

based on which the method can determine whether it is certain that the victim has more items than p has; if this is not the case - the method returns 0 without stealing any item.

Otherwise, about half of the guaranteed difference between the sizes of the 2 deques is copied to p's deque, and then p tries to CAS-update the victim's stealRange. The length of the new stealRange is set to be $max(1, 2^{i-2})$, where the length of the victim's deque before the theft was in the range: $[2^i \cdots 2^{i+1})$. As we prove in our analysis, this guarantees that the new stealRange has length that is at most half, and at least one eighth the remaining number of items.

If the CAS fails, the steal-attempt has failed also, and the method returns 0; otherwise - the steal-attempt has succeeded, and the method proceeds to update p's *bot* and *stealRange* to reflect the new number of items in the deque. Note the following:

- If another process q succeeds in stealing items from p's deque concurrently to p's successful steal-attempt, then p might fail in updating its own *stealRange*, but it would still manage to update its *bot* and complete the steal successfully.
- Although *tryToSteal* can be performed within a signal-handler, it is required that no local operations (namely *pushBottom* or *popBottom*) are performed concurrently with *tryToSteal*'s execution.

```
unsigned int *tryToSteal(ExDeque d, int pLen) {

     rangeLen = length of d.oldStealRange
     oldLen = length of d.deque
     IF pLen > 2*rangeLen -2
        return 0 (no need to steal)
     numToSteal = rangeLen - pLen/2
     Copy first numToSteal items to the bottom of local deq

     newSRLen = max(1, 2^(i-2)) [where 2^i <= oldLen < 2^(i+1)]
     CAS-update d.stealRange to contain newSRLen items

     IF CAS was performed successfully
        {
        update local bot to insert stolen items to the deque
        newDeqLen = new number of items at the local deque
        newSRLen=max(1,2^i) [where 2^(i+2)> newDeqLen >=2^(i+1)]
        CAS-update local stealRange to contain newSRLen items
        IF CAS was performed successfully
           prevStealRange = stealRange
        return numToSteal;
        }
      ELSE
        return 0;
  }
```

Figure 7: tryToSteal method pseudo-code

3 Analysis

In our analysis, we investigate the properties of the StealHalf algorithm under the probabilistic balancing policy, aiming to show that under reasonable assumptions, it keeps the system balanced. The probabilistic balancing policy we employ is very similar to the one described in [12], with the following main differences:

1. In [12], a process initiating load-balancing can either steal items from or insert items to a randomly selected process, whereas in our scheme the initiating process can only steal items. We therefore call the former scheme *symmetric* probabilistic balancing and the latter *asymmetric* probabilistic balancing.
2. The model presented in [12] is synchronous in the sense that it assumes that computation proceeds in time-steps and that all processes attempt load-balancing in the beginning of every time-step, whereas in the asynchronous model we investigate this cannot be assumed.

[12] supplies a proof that symmetric probabilistic balancing keeps the system well balanced. It turns out, however, that this proof is incomplete. In the analysis presented in this section, we therefore supply an alternative proof for the symmetric case and then extend it also for asymmetric probabilistic balancing and specifically for the StealHalf algorithm. For lack of space, some of the proofs are omitted.

3.1 Notation

Wherever possible, we follow the notation of [12].

To simplify the analysis, we assume that an execution is composed of a series of time-steps. In the beginning of each time-step, all processes flip biased coins to determine whether or not they should attempt balancing (in other words, they perform the *balancing initiation code*), and according to the result attempt or do not attempt balancing.

Let $L_{p,t}$ denote the number of items in p's work-pile in the beginning of step t; also, let A_t denote the average system workload in the beginning of step t, namely:

$$A_t = \frac{\sum_{p \in P} L_{p,t}}{|P|}.$$

Process p decides to perform a *balancing attempt* at time t with probability: $\frac{\mu}{L_{p,t}}$, for some constant $1 \geq \mu > 0$. When $L_{p,t}$ equals 0, we define $\frac{\mu}{L_{p,t}}$ to be 1. If p does decide to balance at time t, then it randomly selects another process q to balance with and tries to communicate with q to that effect. We denote this event by $\overrightarrow{select}(p, q, t)$ and therefore we have:

$$P(\overrightarrow{select}(p, x, t)) = \frac{\mu}{L_{p,t}} * \frac{1}{n-1}.$$

If at time t any one of the processes p or q selects the other in a *balancing attempt*, we denote this event by: *select*(p, q, t), namely:

$$select(p, q, t) \overset{def}{=} \overrightarrow{select}(p, q, t) \vee \overrightarrow{select}(q, p, t).$$

If at time t, there are a few processes that initiate a *balancing attempt* with process p, then only one of them gets selected randomly. If process q is the one, we say that *q approaches p at time t* and we denote this event by *approaches(q,p,t)*. If process q is the only process initiating a *balancing attempt* with p at time t, then *approaches(q,p,t)* also holds. If at time t, *approaches(q,p,t)* and p does not initiate a *balancing attempt* at time t, then p and q balance at time t. We denote a balancing event between p and q at time t by: *balance*(p, q, t). If at time t, *approaches(q,p,t)* and *approaches(p,q,t)*, then p and q balance at time t. If at time t, *approaches(p,q,t)* and *approaches(q,r,t)*, where $p \neq r$, then q decides between p and r with equal probability. We denote this event by *decide(q,x,t)*. Finally, if at time t both *decide(p,q,t)* and *decide(q,p,t)* hold, then *balance*(p, q, t) holds.

The changes in the work-pile size of any process p at time step t come from two sources: items which are added or consumed due to the code executed by p at that step, and items added as a result of balancing operations. Let *contrib*$[p,t]$ denote the change in the size of p's work-pile at time-step t which is a result of a balancing operation.

In Section 3.2 we analyze symmetric probabilistic balancing. In Section 3.3 we analyze asymmetric probabilistic balancing in general, and the StealHalf algorithm in particular.

3.2 Symmetric probabilistic balancing analysis

In the following we investigate the relationship between p's work-pile size in the beginning of time-step t, and the expectance of *contrib*$[p,t]$.

Lemma 3.1 *For all p and t we have:*

$$P(select(p,x,t)) = 1-(1-\frac{\mu}{L_{p,t}}\frac{1}{n-1})(1-\frac{\mu}{L_{x,t}}\frac{1}{n-1}).$$

Proof

$$select(p,q,t) \stackrel{def}{=} \overrightarrow{select}(p,q,t) \vee \overrightarrow{select}(q,p,t).$$

Consequently we have:

$$P(select(p,q,t)) = 1 - P(\neg select(p,q,t)) =$$

$$(1 - P(\neg\overrightarrow{select}(p,q,t))P(\neg\overrightarrow{select}(q,p,t))).$$

Finally note, that for all different pairs of processes x, y and for all t:

$$P(\neg\overrightarrow{select}(x,y,t)) = 1 - \frac{\mu}{L_{x,t}} * \frac{1}{n-1}.$$

Q.E.D.

Lemma 3.2 *Let $a_i, 1 \le i \le n$ be n positive real-numbers, then the following holds:*

$$\sum_{i=1}^{n}\frac{1}{a_i} \ge \frac{n}{(\sum_{i=1}^{n} a_i)/n}.$$

Proof We actually have to prove that[16]:

$$(\sum_{i=1}^{n} a_i)\sum_{j=1}^{n}\frac{1}{a_j} \ge n^2.$$

Note that for every two positive real-numbers a, b we have:

$$\frac{a}{b} + \frac{b}{a} \ge 2. \tag{1}$$

By using inequality 1 we get:

$$(\sum_{i=1}^{n} a_i)\sum_{j=1}^{n}\frac{1}{a_j} = n + \sum_{1\le i\ne j\le n}(\frac{a_i}{a_j}+\frac{a_j}{a_i}) \ge n + 2*\binom{n}{2} = n^2.$$

. Q.E.D.

The following lemma states that $P(select(p,q,t))$ and $P(balance(p,q,t)$ differ only by a constant factor.

[16]This lemma is actually a rephrasing of the arithmetic/harmonic mean inequality. Since the proof is very short, we provide it for the sake of presentation-completeness.

Lemma 3.3 *For every step t, and for every two processes p and q,*

$$\frac{1}{2e^2}P(select(p,q,t)) < P(balance(p,q,t)) \le P(select(p,q,t)).$$

Proof outline: p and q can balance at step t only if at least one of them selects the other at that step. Consequently

$$P(balance(p,q,t)) \le P(select(p,q,t)).$$

As for the other direction, note that if all the following conditions hold, it is guaranteed that $balance(p,q,t)$ holds:

C1: $select(p,q,t)$ holds;

C2: No other process r selects p or q at time t, namely:

$$\forall r \ne p,q : \neg\overrightarrow{select}(r,p,t) \wedge \neg\overrightarrow{select}(r,q,t).$$

C3: Neither of p,q select another process r at time t, or the following holds: $p[q]$ selects $q[p]$, $q[p]$ selects a different process r, but $q[p]$ decides to balance with $p[q]$ rather than with r. In other words:

$(\forall r \ne p,q : (\neg\overrightarrow{select}(p,r,t) \wedge \overrightarrow{select}(q,r,t))$ OR
$\exists r : [(\overrightarrow{select}(p,q,t) \wedge \overrightarrow{select}(q,r,t) \wedge decide(q,p,t))]$ OR
$\exists r : [\overrightarrow{select}(q,p,t) \wedge \overrightarrow{select}(p,r,t) \wedge decide(p,q,t)]$

Consequently we have:

$$P(balance(p,q,t)) \ge P(select(p,q,t)) * P(C2) * P(C3).$$

In the full proof we show that $P(C2) \approx \frac{1}{e^2}$ and $P(C3) > \frac{1}{2}$ and thus obtain the result.

Theorem 3.4 *Let $L_{p,t} = \alpha A_t$, $\alpha > 1$, then:*

$$E[contrib(p,t)] = -\Omega(\alpha).$$

Proof Clearly

$$E[contrib(p,t)] = \sum_{x\in P, x\ne p} P[balance(p,x,t)]\frac{L_{x,t}-L_{p,t}}{2} \tag{2}$$

Let P_α^+ denote the set of processes whose work-pile size is larger than αA_t at the beginning of time-step t, and let $P_\alpha^- = P - P_\alpha^+$. We get:

$$\begin{aligned} E[contrib(p,t)] &= \textstyle\sum_{x\in P_\alpha^+} P[balance(p,x,t)]\frac{L_{x,t}-L_{p,t}}{2} \\ &\quad + \textstyle\sum_{x\in P_\alpha^-} P[balance(p,x,t)]\frac{L_{x,t}-L_{p,t}}{2} \end{aligned} \tag{3}$$

Note, that the summation over P_α^+ contains only positive summands, whereas the summation over P_α^- contains only non-positive summands. Consequently, we can use Lemma 3.3 (for each summation separately) to get:

$$\begin{aligned} E[contrib(p,t)] &\le \textstyle\sum_{x\in P_\alpha^+} P[select(p,x,t)]\frac{L_{x,t}-L_{p,t}}{2} \\ &\quad + \frac{1}{2e^2}\textstyle\sum_{x\in P_\alpha^-} P[select(p,x,t)]\frac{L_{x,t}-L_{p,t}}{2} \end{aligned} \tag{4}$$

We now bound each sum separately from above. We start with the positive summands. By using Lemma 3.1 we get:

$$\textstyle\sum_{x\in P_\alpha^+} P[select(p,x,t)]\frac{L_{x,t}-L_{p,t}}{2} =$$

$\sum_{x\in P_\alpha^+}(1-(1-\frac{\mu}{L_{p,t}}\frac{1}{n-1})(1-\frac{\mu}{L_{x,t}}\frac{1}{n-1}))\frac{L_{x,t}-Lp,t}{2} =$

By multiplying and re-arranging we get:

$$= \underbrace{\sum_{x\in P_\alpha^+} \frac{\mu}{2(n-1)}\frac{L_{x,t}}{L_{p,t}}}_{A}$$

$$- \underbrace{\sum_{x\in P_\alpha^+} \frac{\mu}{2(n-1)}\frac{L_{p,t}}{L_{x,t}}}_{B}$$

$$+ \underbrace{\sum_{x\in P_\alpha^+} \frac{\mu^2}{2(n-1)^2}(\frac{1}{L_{x,t}}-\frac{1}{L_{p,t}})}_{C}.$$

We now bound A, $-B$ and C:

$$A = \frac{\mu}{2(n-1)L_{p,t}}\sum_{x\in P_\alpha^+} L_{x,t} \le \frac{\mu}{2(n-1)L_{p,t}}\sum_{x\in P} L_{x,t}$$
$$= \frac{\mu}{2(n-1)\alpha A_t}nA_t = \frac{\mu n}{2\alpha(n-1)} < 1.$$

B is positive and so $-B$ is bound by 0 from above. As for C:

$$C \le \frac{\mu^2}{2(n-1)^2}\underbrace{(1+1+\cdots+1)}_{|P_\alpha^+|} \le \frac{\mu^2}{n}.$$

Combining the upper bounds for A, $-B$ and C we get:

$$\sum_{x\in P_\alpha^+} P[balance(p,x,t)]\frac{L_{x,t}-L_{p,t}}{2} \le 2. \qquad (5)$$

Now, we bound the negative summands. First, note that $|P_\alpha^+| \le \frac{n}{\alpha}$ and so $|P_\alpha^-| \ge n(1-\frac{1}{\alpha})$.

Again, by multiplying and re-arranging summands we get the same A, $-B$ and C components, and we bound them from above. A and C can be bounded by 1 and $\frac{\mu^2}{n}$ respectively, in exactly the same way it was done for the positive summands; as for B:

$$B = \sum_{x\in P_\alpha^-} \frac{\mu}{2(n-1)}\frac{L_{p,t}}{L_{x,t}} = \frac{\mu\alpha A_t}{2(n-1)}\underbrace{\sum_{x\in P_\alpha^-}\frac{1}{L_{x,t}}}_{S}. \qquad (6)$$

Noting that the average work-pile length for processes in P_α^- is not more than A_t, and using Lemma 3.2 we get:

$$S \ge \frac{n(1-\frac{1}{\alpha})}{A_t}.$$

By substituting this upper bound for S in Equation 6 we get:

$$B \ge (\frac{\mu\alpha A_t}{2(n-1)})(\frac{n(1-\frac{1}{\alpha})}{A_t}) \ge \frac{\mu(1-\frac{1}{\alpha})}{2}\alpha = \Omega(\alpha).$$

Combining the upper bounds on A, B and C, we get:

$$\sum_{x\in P_\alpha^-} P[balance(p,x,t)]\frac{L_{x,t}-L_{p,t}}{2} = -\Omega(\alpha). \qquad (7)$$

Finally, substituting the upper bounds of Equations 5 and 7 in Equation 4 concludes the proof of the theorem. Q.E.D.

We now consider the effect of symmetric probabilistic balancing, when balancing attempts are performed at a certain frequency. We define the *balancing quantum* as the time duration of each time-step. If the *balancing quantum* is Δ, we say that the *balancing frequency* is $\frac{1}{\Delta}$.

We say that a load-balancing algorithm with balancing quantum Δ is *locally-bounding*, if there is a constant α, independent of the number of processes, such that for any execution the following is guaranteed:

$$\forall p,t: E[L_{p,t}] < \alpha A_t.$$

We also say that the work-queues system is *α-locally bounded* at time t, if the following inequalities hold during at time t:

$$\forall p: L_{p,t} < \alpha A_t.$$

As noted earlier, the difference between $L_{p,t}$ and $L_{p,t+\Delta}$ comes from two sources: from the effect of a balancing operation that may or may not take place at that time-step (and its contribution is denoted by *contrib*[p,t]), and from work-items that are generated
and/or consumed by p in its application execution during that time-step.

We denote by Δ_u a time-quantum small enough, such that the application execution (balancing operations notwithstanding) does not change the length of any work-pile by more than u items.

The following theorem proves that symmetric probabilistic balancing with frequency $\frac{1}{\Delta_u}$, for any integer u, is a *locally bounding* scheme.

Theorem 3.5 *Assume symmetric probabilistic balancing is employed with balancing-frequency $\frac{1}{\Delta_u}$, then there is a constant α_u, not depending on the number of processes or the application, such that if the system starts* α_u-locally bounded *- then the following inequalities hold:*

$$\forall p,t: E[L_{p,t}] < \alpha_u A_t.$$

Proof According to Theorem 3.4, there is a constant c, not depending on the number of processes, such that:

$$(\beta > 1) \wedge (Lp,t > \beta \cdot A_t) \Rightarrow E[contrib(p,t)] < -c\beta. \qquad (8)$$

We show that $\alpha_u = \frac{2*u}{c}$ is the constant we are seeking. Note, that if $L(p,t) > \alpha_u A_t$ at the beginning of an execution quantum, then $E[contrib(p,t)] < -2u$.

During a Δ_u time-quantum of execution, $L_{p,t}$ can grow by at most u items. During that period, the system average can decrease by at most u items, so during this period $(L_{p,t} - A_t)$ can grow by at most $2u$ items. Q.E.D.

Corollary 3.6 *Assume symmetric probabilistic balancing is employed with balancing-frequency $\frac{1}{\Delta_u}$, then there is a constant α_u, not depending on the number of processes or the application, such that if the system starts imbalanced and runs long enough - it eventually becomes* α_u-locally bounded

3.3 StealHalf probabilistic balancing analysis

The scheme described in [12] is symmetric in the sense that a balancing operation between two processes p,q can take place if either one of them initiates it. This is not the case for the StealHalf algorithm since it only allows *stealing* items and does *NOT* support insertion of items. In other words, if at time t, $L_{p,t} < L_{q,t}$, then only p can initiate a balancing operation at that time with q, and not vice-versa. In the following we prove that asymmetric probabilistic balancing also possesses the nice property of being *locally bounding.*

The following lemma states that $P(select(p,q,t))$ and $P(balance(p,q,t)$ differ only by a constant factor also for asymmetric probabilistic balancing.

Lemma 3.7 *For every time step t, and for every two processes p and q such that $L_{p,t} < Lq,t$ it holds that:*

$$\frac{1}{2e^2}P(\overrightarrow{select}(p,q,t)) < P(balance(p,q,t)) \leq P(\overrightarrow{select}(p,q,t)).$$

The proof is almost identical to the proof of Lemma 3.3.

Next, Theorem 3.8, corresponding to Theorem 3.4, is derived for the asymmetric case.

Theorem 3.8 *Let $L_{p,t} = \alpha A_t$, $\alpha > 1$, then it holds for asymmetric probabilistic balancing that:*

$$E[contrib(p,t)] = -\Omega(\alpha).$$

Proof outline: Note, that Equation 4 holds also for asymmetric balancing; however, unlike the symmetric case, where we have to consider balancing initiation by all possible process-pairs which include p - for asymmetric balancing, when we consider balancing operations at time t that affect p, we only have to consider the following events:

1. $\overrightarrow{select}(p,x,t)$, for $x \in P_\alpha^+$
2. $\overrightarrow{select}(x,p,t)$, for $x \in P_\alpha^-$

The proof's structure is similar to the proof of Theorem 3.5 (the corresponding theorem for the symmetric case).

Based on Theorem 3.8, the following theorem, corresponding to Theorem 3.5, is proven for the asymmetric case. The proof is almost identical.

Theorem 3.9 *Assume asymmetric probabilistic balancing is employed with balancing-frequency $\frac{1}{\Delta_u}$, then there is a constant α_u, not depending on the number of processes or the application, such that if the system starts α_u-*locally bounded - *then the following inequalities hold:*

$$\forall p,t : E[L_{p,t}] < \alpha_u A_t.$$

The following theorem states that the StealHalf algorithm maintains the following invariant for all processes p at all times: *at least one eighth the number of items in p's deque can be stolen atomically.*

Theorem 3.10 *Under StealHalf load balancing (for all policies) the following holds:*

$$\forall p,t : Len(stealRange(p,t)) \geq \frac{Len(deq(p,t))}{8}.$$

The proof proceeds by enumerating the statements which potentially modify either $Len(deq)$) or $Len(stealRange)$) and showing that the invariant is maintained after each one of them is executed. It is rather technical and for lack of space is not provided here; still, let us explain an interesting scenario that is encountered in the course of the proof, that of consecutive successful steals from the same process.

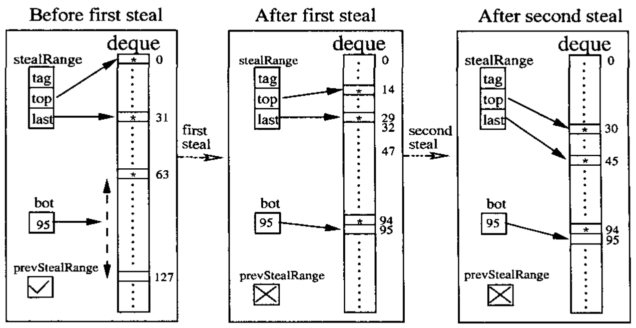

Figure 8: *A scenario of 2 consecutive steals.* In the first steal-operation, 14 items are stolen (out of the 32 items which can be stolen atomically). In the second steal, all of the 16 items in the stealRange are stolen. A $\surd$ sign in the prevStealRange box indicates that it's equal to the stealRange; an X sign indicates it is different from the stealRange

Figure 3.3 shows a scenario where two consecutive steal-operations are performed on process p's deque, while in the meantime p is not performing any operation on its deque. Initially, p's stealRange contains 32 items, but the first thief only steals 14 of them (which is what it needs to balance with p). When the thief looks at *bot*, it equals 95, but before the steal's CAS operation completes, it may change within the range [64..127]. To make sure that after the steal *stealRange* would not contain more than half the remaining items, the thief sets the new length of stealRange to be one eighth of the maximal possible value of *bot* plus 1, namely $\frac{128}{8} = 16$.

In the second steal shown at Figure Figure 3.3, the thief steals all of the 16 items in the stealRange, and again sets the new length of stealRange to contain $\frac{128}{8} = 16$ items. This is correct, since this time prevStealRange differs from stealRange, and so the local process p cannot change *bot* without performing a CAS.

Remember that Theorem 3.9 above assumes synchrony, in the sense that processing proceeds in time-steps, and all processes perform balancing-initiation in the beginning of every time-step. Additionally, it assumes that up to half the items of any process can be stolen atomically.

Contrary to this, the StealHalf algorithm's setting is entirely asynchronous, and though the balancing frequency of every process is guaranteed, processes perform their balancing-initiation operations in separate times; additionally, the StealHalf algorithm can only guarantee that at least one eighth of a process' items can be stolen atomically. The modifications required in the above proof to prove that StealHalf is a *locally bounding* scheme are straightforward. They are not brought here for lack of space, but are to appear in the full paper.

4 Correctness

In the full paper we prove the following theorem which shows that our algorithm has the same non-blocking property as that of Arora et al: the collective progress of processes in accessing the extended-deque structures is guaranteed.

Theorem 4.1 *The StealHalf algorithm on a collection of extended-deques, with operations pushBottom, popBottom, and tryToSteal, is non-blocking.*

We note that both algorithms are not fault tolerant in the sense that process failures, though non-blocking, can cause the loss of items.

In [13], Shavit and Touitou formally define the semantics of a pool data structure. A *pool* is an unordered queue, a concurrent data structure that allows each processor to perform sequences of push a pop operations with the usual semantics. In the full paper we provide the proofs of the key lemmata necessary to prove the following theorem:

Theorem 4.2 *The StealHalf algorithm on a collection of extended-deques, with operations pushBottom, popBottom, and tryToSteal, is a correct implementation of a pool data structure.*

We define the complexity of our algorithm in terms of the total number of synchronization operations necessary by analyzing it for monotonic sequences of pushBottom and popBottom operations. We do so since one cannot make claims in situations where the execution pattern of the underlying application causes thrashing back and forth on a single deque entry. For a given deque, a monotonic sequence is one in which all operations are either *pushBottom* or *popBottom* but not both, with a possible interleaving of *tryToSteal* operations. We prove the following:

Theorem 4.3 *For any monotonic sequence of length k by process p during which m successful steal attempts are performed on p's extended deque, p performs at most $\Theta(\log(k)+m)$ CAS operations on the extended deque.*

5 Acknowledgments

We would like to thank Dave Detlefs, Maurice Herlihy, Victor Luchangco, and Mark Moir for their helpful comments on earlier drafts of our algorithm. Though he denies it and has managed to conveniently lose all evidence, we believe Dave Detlefs came up with a steal-half algorithm very similar to ours in the context of his work on Sun's Parallel Garbage Collection [5].

References

[1] ACAR, U. A., BLELLOCH, G. E., AND BLUMOFE, R. D. The data locality of work stealing. In *ACM Symposium on Parallel Algorithms and Architectures* (2000), pp. 1–12.

[2] ARORA, N. S., BLUMOFE, R. D., AND PLAXTON, C. G. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems 34*, 2 (2001), 115–144.

[3] BERENBRINK, P., FRIEDETZKY, T., AND GOLDBERG, L. A. The natural work-stealing algorithm is stable. In *Proceedings of the 42th IEEE Symposium on Foundations of Computer Science (FOCS)* (2001), pp. 178–187.

[4] BLUMOFE, R., AND LEISERSON, C. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.* (November 1994), pp. 356–368.

[5] FLOOD, C., DETLEFS, D., SHAVIT, N., AND ZHANG, C. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)* (Monterey, CA, Apr. 2001).

[6] HERLIHY, M. Wait-free synchronization. *ACM Transactions On Programming Languages and Systems 13*, 1 (Jan. 1991), 123–149.

[7] LEISERSON, AND PLAAT. Programming parallel applications in cilk. *SINEWS: SIAM News 31* (1998).

[8] LULING, R., AND MONIEN, B. A dynamic distributed load balancing algorithm with provable good performance. In *ACM Symposium on Parallel Algorithms and Architectures* (1993), pp. 164–172.

[9] MITZENMACHER, M. Analysis of load stealing models based on differential equations. In *ACM Symposium on Parallel Algorithms and Architectures* (1998), pp. 212–221.

[10] MITZENMACHER, M. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems 12*, 10 (2001), 1094–1104.

[11] MOIR, M. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (August 1997)*, pp. 219–228.

[12] RUDOLPH, L., SLIVKIN-ALLALOUF, M., AND UPFAL, E. A simple load balancing scheme for task allocation in parallel machines. In *ACM Symposium on Parallel Algorithms and Architectures* (1991), pp. 237–245.

[13] SHAVIT, N., AND TOUITOU, D. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 30 (1997), 645–670.