Work Dealing

[Extended Abstract]

Danny Hendler*
Tel-Aviv University
hendlerd@post.tau.ac.il

Nir Shavit†
Tel-Aviv University
shanir@cs.tau.ac.il

ABSTRACT

This paper introduces *work-dealing*, a new algorithm for "locality oriented" load distribution on small scale shared memory multi-processors. Its key feature is an unprecedented low overhead mechanism (only a couple of loads and stores per operation, and no costly compare-and-swaps) for dealing-out work to processors in a globally balanced way. We believe that for applications in which work-items have process affinity, especially applications running in dedicated mode ("stand alone"), work-dealing could prove a worthy alternative to the popular work-stealing paradigm.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous; D.2 [**Software**]: Miscellaneous

General Terms

Algorithms, Theory

Keywords

distributed, wait-free, load-balancing

1. INTRODUCTION

The need to improve the locality of thread execution in shared memory multiprocessors has long been a goal of algorithm designers. Recently, attempts have been made to analyze and improve the locality of work-stealing based algorithms [1, 12]. This paper presents *work-dealing*, a new algorithm for "locality oriented" load distribution. It is based on a novel, non-blocking way of dealing-out work items to processes as the items are created, rather than rely on processes to steal work from others when they become idle.

*The first author's work was supported in part by a grant from Sun Microsystems.

†Part of this work was performed while the second author was at Sun Microsystems Laboratories.

SPAA'02, August 10-13, 2002, Winnipeg, Manitoba, Canada.

1.1 Background

Work-stealing, and specifically the work-stealing algorithm of Arora et al. [2] has been shown in [2, 3, 6, 11] to be an effective alternative to load-sharing [9, 13] for balancing work load on shared memory multiprocessors. Work stealing allows each process to maintain a local work queue, and steal an item from others if its queue becomes empty. The philosophy behind work-stealing is that the high cost of synchronizing (using a Compare-and-Swap (CAS) operation) to move items from one work queue to another, will fall on the processes without work, thus minimizing the delay of the processes with work, the ones most likely on the computation's critical path. The beauty of the Arora et al. algorithm [2] rests in a scheme for stealing an individual item from a bounded size queue, while minimizing the need for costly CAS synchronization operations when fetching items locally.

In recent years, it has become clear that in addition to good load balancing, good data locality is essential in obtaining high performance from modern parallel systems [1]. A recent paper by Acar et al. [1] shows how one can improve the "data locality" of thread distribution in the Arora et al. algorithm by adding "process affinity" information to threads. The algorithm has, in addition to the work-stealing queue, a special "mailbox" queue to each process, and works by adding, per thread, a pointer in the mailbox of the "process of highest affinity" so that it will be more likely to steal that thread.

Though it improves data locality, the Acar et al. algorithm introduces a significant overhead over the Arora et al. algorithm: several costly synchronization operations are necessary in handling each and every thread. Unlike in the Arora et al. algorithm, this costly synchronization is performed whenever a new thread is generated. In other words, it is not restricted to the stealing processes that have no work, but is performed by all processes, including ones that are potentially on the execution's critical path.

But there is an even bigger issue here. Even with an efficient locality-oriented stealing mechanism, the very nature of work stealing implies that threads will preferably be executed by the generating local process, even if their affinity is for another remote process. Such affinity for remote processes can be found in popular applications such as parallel garbage collection [5], where for some subtasks work is equally split among all processes in terms of locality, but is generated by processes with little locality.

This paper argues that given the above limitations, there is room to consider a work-distribution methodology in which

work, rather than being added to a local work-pile and only rarely stolen to another higher-locality process, is dealt-out a-priory to processes based on locality considerations. Following such a heuristic, while preserving a global overall work balance, would potentially allow great locality in work items execution. However, to enable such a scheme, one would have to figure out a way to place work created by one process into the work-pile of another with little synchronization overhead.

1.2 Work Dealing

We introduce *work dealing*, a new load sharing algorithm that has minimal overhead for distributing items among processes. In fact, the algorithm requires only a couple of loads and stores, and no costly synchronization operations, to push or pop an item onto a work-pile, be it local or remote.

The algorithm is intended for small scale shared memory machines running applications in dedicated mode (as opposed to multiprogrammed mode), but has a variant that will work in multiprogrammed mode as well. In addition to its low synchronization overhead, the algorithm is dynamic in its memory use, that is, work-piles do not need to be fixed size arrays as in [1, 2], thus avoiding the need to build specialized "overflow" mechanisms [5].

In work-dealing, each process, as it generates work items, will distribute them to other processes based on a pre-agreed distribution policy. This can be a simple policy such as Round Robin, or a more sophisticated "locality guided" approach.

The key algorithmic idea behind the scheme is very simple. We begin by noting that the design of a fixed size non-blocking (in fact, wait-free) producer-consumer buffer (PCB) for one consumer and one producer is a matter of folklore [8]. For our algorithm, we designed a *dynamically sized* non-blocking PCB with similar properties for one producer and one consumer. Our construction is such that each produce or consume operation requires at most a couple of loads and stores, with the added advantage that producers need only access a head pointer while consumers need only access a separate tail pointer. Now, imagine that each process i's work-pile consists of n such PCBs with i as the consumer and each of the n processes $j \in \{1 \dots n\}$ as a producer. For n processes, we have a total of n^2 such PCBs in the system, each initially consisting of a head and tail pointer and a dummy item-record. The dynamic memory used by PCBs can be provided by the system if lock-free memory management is provided [4, 7]. Alternately, in Section 2.2 we provide a simple dynamic memory recycling scheme with very low synchronization overhead: in practice it will amount to an uncontended CAS operation per several thousands of recycle operations.

How are the PCBs used? Figure 1.2 shows the dealing-out of work-items according to a *simple* work dealing policy. According to this policy, each process creating items distributes them in Round Robin fashion, one per work-pile, into the appropriate PCB for which it is a producer. To consume items, a process goes through the PCBs where it is a consumer, draining one and moving on to the next in a Round Robin fashion. Thus, in a loaded system, no matter how imbalanced the actual work generation is, the amortized cost of working on an item, that is, pushing and popping it, is a couple of loads and stores, and no matter how imbalanced the actual work generation is, there are no costly synchronization operations. In an unloaded system, a process might have to go through several of its PCBs before finding an item to work on. For small scale machines, given that the traversal involves no synchronization operations, only the reading of some locally cached variables, this overhead is negligible.[1]

A more sophisticated *locality guided* policy, has the same consumption policy as *simple*, but deals out work based on process-affinity information available for each work item. In a nutshell, an item generated by a given process i is placed in the appropriate PCB in the work-pile of the highest-affinity process, unless by doing so the overall balance among the number of items placed by i in all work piles is invalidated. A typical choice for an overall balance is that any local work-pile did not receive more than L times the average distributed by i to all others, for some $L > 2$. If the balance is invalidated, the item is placed in Round Robin fashion in the next work-pile PCB that has received from i less than twice the average it distributed to all others. The locality-guided policy is described in detail in Section 3.

We also present an *adaptive* policy intended for multiprogrammed environments. One can combine the adaptive and locality guided policies in a natural way. For lack of space we leave the detailed description and analysis of this policy to the full paper.

1.3 Performance Analysis

In Section 4 we analyze the system balance, memory overhead, and execution time of work-dealing. It is shown that work-dealing with *simple* and *locality-guided* policy asymptotically distributes work optimally (up to a constant) - and consequently its execution-time can be shown to also be asymptotically optimal-up-to-a-constant - with probability that increases as the number of produced items increases.

Let A be an application, let $Optimal(A)$ denote the execution time of A with optimal speed-up, and let $\mathrm{WD}(A)$ denote execution-time using work-dealing under the *simple policy*. We show that for every $0 \leq p < 1$ and $\alpha > 0$ there is a number $K(p, \alpha)$, such that if A produces more than $K(p, \alpha)$ items, and the item-generation pattern allows a high degree of parallelism, the following inequality holds:

$$P\left(\frac{\mathrm{WD}(A)}{Optimal(A)} \leq 3 + \alpha\right) \geq p$$

A similar result is obtained for the *locality-guided* policy under the assumption that work-items have affinity to all processes with equal probability.

The *locality guided* policy tries to increase the number of items which are executed by the process to which they have affinity, while preventing the system from getting too imbalanced. An important question is: What is the probability for any item i to be executed by the process to which it has affinity? We call this probability, the *affinity hit-ratio* of the scheme. In the analysis we prove, that under reasonable assumptions, the *locality guided* policy with parameter L has *affinity hit-ratio* of at least $1 - \frac{1}{L}$.

Assume a bound M is known a-priori on the maximal number of items which may exist simultaneously in the system. We show that work-dealing using our RS recycling-

[1]Note that after traversing and finding all of its n work-pile PCBs empty, a process becomes idle and initiates a termination detection protocol. For lack of space we do not describe this protocol in the manuscript.

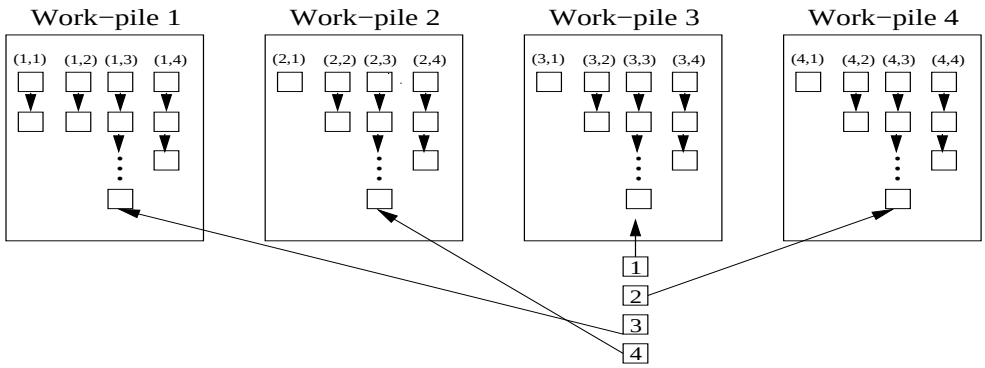

Figure 1: Work dealing with 4 processes: simple policy. Process i'th work-pile includes its 4 consume-PCBs. The ordered pair above a PCB contains its indices in the `pcbs` array. The bottom of the figure shows the distribution of the next 4 items process 3 produces. They are distributed to process 3 produce-PCBs: 3, 4, 1 and 2, respectively.

scheme can completely avoid memory overflow, with memory overhead of at most $(2n+1)G$ work items, where G is the granularity [2] with which items are recycled by the RS scheme and n is the number of processes. This is an improvement over the work-stealing algorithms of [1, 2] which by nature are designed for a fixed size array and thus incur a much larger memory-overhead if memory-overflows are to be completely avoided. For extremely imbalanced applications, work stealing may require $\Omega(nM)$ memory, hence the use of specialized overflow mechanisms [5].

2. THE WORK DEALING ALGORITHM

Figure 2 shows the data structures used by the algorithm. The main data-structure used by the algorithm is the `pcbs` shared bi-dimensional array of Producer-Consumer-Buffers (PCBs). The first dimension designates the consumer-process and the second dimension designates the producer-process; e.g., `pcbs`[2][5] is the PCB where items produced by process 5 are added, so that they can be consumed by process 2. A PCB is represented by a head and tail ITEM pointers. The PCB's producer links new items to the tail of the PCB, while the PCB's consumer removes items from the head of the PCB. An item is composed of an opaque VALUE and a `next` pointer, that is either *null* or points to the next item.

2.1 High-level Methods Description

2.1.1 PCB data-structures, variables and methods

The methods with which PCB structures can be manipulated are shown in Figure 2. The `createPCB` method creates a new PCB and initializes its `head` and `tail` pointers to a dummy item whose value is never used. The `PCBEmpty` method receives a PCB pointer as its parameter and returns a boolean value indicating whether or not that PCB is empty. A PCB is empty if and only if the `next` pointer of the item pointed at by the PCB's `head` pointer is null. The `produce` method receives a PCB-pointer and an item-pointer as its parameters and chains the item to the tail of the PCB. It then modifies the PCB's tail to point at the new item. Finally, the `consume` method receives a pointer to a non-empty PCB as its parameter. The method unlinks the item pointed at by the PCB's head-pointer from the PCB and recycles its memory; it then modifies the head-pointer so that it points to the next item, and returns that item's value as its return-code.

[2]See section 2.2

2.1.2 `putItem`

The `putItem` method is shown in Figure 3. A new item is created by calling the `newItem` method, which allocates space for a new item and returns a pointer to it.

The new item's value is set to the value received as the method-parameter, and then a `putOracle` (whose implementation depends on the policy) is called, to determine to which consume-PCB the item should be added. Finally, the item is added to the chosen PCB by calling the `produce` method.

2.1.3 `getItem`

The `getItem` method is shown in Figure 3. It tries to retrieve an item from one of the PCBs of the calling consumer. It consists of a loop, each iteration of which checks a single PCB. A `getOracle` is called in each iteration to determine which PCB should be checked. If a non-empty PCB is found, the `consume` method is called to retrieve the value of the first item in the queue. The loop continues, until either a non-empty PCB is found, or the `getOracle` detects the termination of the algorithm.

2.2 Recycling Item Records

The linked-list implementation of PCBs used by the work-dealing algorithm calls the `newItem` and `recycleItem` methods for allocating a new item and for recycling it, respectively. This requires the availability of an effective distributed mechanism, preferably non-blocking, for recycling item records. If the work-dealing algorithm is implemented on top of a multiprocessing platform that has an adequate dynamic-memory implementation available [4], then that implementation can be used; the availability of an adequate dynamic-memory implementation is not guaranteed, however.

In this section we present a simple and efficient item-record recycling scheme, which we call the Recycling Scheme (RS, for short). The RS is simple, non-blocking, and allows

```
//
// types and variables
//
typedef structure
{
   ITEM *next;
   VALUE val;
} ITEM;

typedef structure
{
   ITEM *head;
   ITEM *tail;
} PCB;

shared PCB pcbs[P_NUM][P_NUM];
unsigned int localId;

//
// methods
//
PCB* createPCB()
{
1   PCB *PCB = (PCB *) malloc (sizeof PCB);
2   ITEM *i=newItem();
3   i.next=null;
4   PCB.head=PCB.tail=i;
5   return PCB;
}

void produce(PCB *PCB, ITEM *i)
{
1   PCB.tail->next=i;
2   PCB.tail=i;
}

boolean PCBEmpty(PCB *PCB)
{
1   if (PCB.head->next==null)
2      return true;
3   else
4      return false;
}


VALUE consume(PCB *PCB)
{
1   ITEM *tmp=PCB.head;
2   PCB.head=PCB.head->next;
3   recycleItem(tmp);
4   return PCB.head.val;
}
```

Figure 2: PCB data-structures, variables and methods

to control the tradeoff between memory-overhead and contention.

The RS maintains a pool of linked-item-lists, which are allocated to processes dynamically. We call these lists *RS item-list*s, or just *item list*s. All allocated linked-lists initially contain the same number of items - *ALLOC-SIZE*, which is a parameter of the algorithm. We call the *ALLOC-SIZE* parameter the allocation's *granularity*. Another parameter of the algorithm, is the number of *item list*s that are generated during the RS initialization - *LISTS-NUM*.

The *item lists* are managed by the *RS-pool*, which is a non-blocking concurrent pool. We do not specify the implementation of the pool, which could be implemented using a variety of methods, one of which is the non-blocking FIFO queue of Michael and Scott [10]. This queue requires a CAS operation per insert or delete. Each process p maintains two item-lists throughout execution:

```
VALUE getItem()
{
1    do
2       {
3       unsigned int nextPCB=getOracle();
4       if (nextPCB==TERMINATE)
5          exit();

6       PCB *PCB=pcbs[localId][nextPCB];
7       if (PCBEmpty(PCB))
8             continue;

9       return consume(PCB);
10      } forever;
}

void putItem(VALUE val)
{
1   ITEM *i = newItem();
2   i->val=val;
3   i->next=null;
4   unsigned int nextToPut=putOracle(val);
5   PCB *pcb=&pcbs[nextToPut][localId];
6   produce(pcb, i);
7   return;
}
```

Figure 3: putItem and getItem methods

- `produceList` - This list is used by the process to produce new items. Whenever p needs to produce a new item, it unlinks the `head` item from the `produceList`, stores the application value into its *val* field and inserts it to the selected consuming-process' PCB. When p needs a new item and its `produceList` is empty, a new *item list* is retrieved from the RS-pool.

- `consumeList` - This list is being built by a process p as it consumes items. When the list length becomes ALLOC-SIZE, it is enqueued into RS-pool and p starts to accumulate a new list.

Figure 4 shows how the various methods of the RS allow item lists to be added from a process' `consumeList` to the shared pool once the process recycled enough items, and similarly how a new list of unused item records is dequeued from the shared pool and added to a process' `produceList` when it becomes empty. There is obviously a tradeoff between the number of items moved into the shared RS pool at a time and the frequency of such operations. In general though, based on the results of [10], on small scale machines the granularity of the operations can be tuned such that the frequency of allocations implies that concurrent CAS operations happen infrequently, eliminating any real synchronization overhead. We do not elaborate on these methods which are straightforward in their implementation. [3]

2.3 Work-dealing Policies

The generic work-dealing algorithm that we presented uses the `putOracle` and `getOracle` methods. A process p uses these methods to determine which produce-PCB should receive p's newly produced item, and from which consume-PCB p should obtain the next item to consume, respectively.

[3]Note, that for the sake of presentation simplicity, the code assumes that enough items are allocated initially to accomodate maximal load. The code can be extended in a strightforward manner so that this assumption would not be required.

```
void initialize() {
1    unsigned int i;
2    for (i=0;i<LISTS_NUM; i++)
3       enqueue(RS_pool,newItemList(ALLOC-SIZE)); }

void newItem() {

1    if (produceList->head == null)
        {
2       recycle(produceList);
3       produceList = dequeue(RS_pool);
        }
4    ITEM *i = produceList->head;
5    produceList->head = i->next;
6    produceList->itemsNum--;
7    return i; }

void recycleItem(ITEM *i) {
1    if (consumeList == null)
        {
2       consumeList = new ITEM_LIST;
3       consumeList.itemsNum = 1;
4       consumeList.head = i;
5       i->next = null;
        }
6    else
        {
7       ITEM *head = consumeList.head;
8       i->next = head;
9       consumeList.head = i;
10      consumeList.itemsNum++;
11      if (consumeList.itemsNum == ALLOC_SIZE)
12         enqueue(RS_pool, consumeList);
        } }
```

Figure 4: RS methods

We call a specific implementation of these oracle-methods a *work-dealing policy*, or just *policy* for short.

Many different policies are conceivable, and they differ with respect to their properties and to the architectures and applications for which they are appropriate. In the following, we present a high-level description of few possible policies by describing the `putOracle` and `getOracle` functions of these policies.

- *Simple policy*: The *simple* policy is intended as an example to the reader of the simplicity of the dealing approach. Its put-oracle distributes process j's items one by one, in a Round Robin fashion, to each of j's production PCBs: the k'th item is added to `pcbs`$[(j + k) \bmod n][j]$. In the simple policy's get-oracle, which is the same for the more sophisticated locality-based policy below, a process p scans p's consume-PCBs in a Round Robin manner. Whenever a non-empty PCB is found, the items in it are returned one by one, until it is made empty. In the analysis we prove that it makes sure that the work-distribution is optimal up-to-a-constant. Moreover, if process-speeds are equal (up to a constant) throughout execution, we prove that asymptotic execution time is optimal up to a constant, with high probability, provided enough items are produced.

- *Locality-guided policy*: The key advantage of work dealing, the advantage that justifies the sacrifice of reduced overhead of local execution by the generating process, is the ability to have many work items executed by the process for which they have affinity [1]. We follow the process-affinity approach introduced by Acar et al. [1]. According to this approach, it is desirable to allow some way of specifying an item's *process-affinity* and to increase the probability that items are performed by the "preferred" process with the locality advantage. In Section 3 we describe this policy in detail.

- *Adaptive policy*: A drawback of the above policies is that if the system becomes imbalanced, there is no mechanism to re-balance it. Consequently, it is not appropriate for multi-programmed systems where process execution quota's may vary. For such systems, an *adaptive policy* can provide more assurance that system balance can be maintained over time. For this policy, each *PCB* is extended with 2 counters: *Produced* counter - which stores the number of items inserted into the PCB by its producer, and *Consumed* counter - which stores the number of items retrieved from this PCB by its consumer. The *adaptive policy*'s *getOracle* scans each process p's consume-PCBs in a Round Robin fashion, until it finds a non-empty PCB from which the next item to be consumed by p is retrieved. The *adaptive policy*'s *putOracle*, by using the *Produced* and *Consumed* counters associated with every PCB, balances the load dynamically over p's produced-PCBs based on the number of items counted in each PCB at the given time, as opposed to the total number ever placed in it. For lack of space we do not elaborate on this policy.

2.4 Correctness

In [14], 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 and pop operations with the usual semantics. In the full paper we prove that:

THEOREM 1. *The work-dealing scheme with RS recycling, with operations getItem and putItem, is a non-blocking implementation of a pool data structure.*

We note that our work-dealing algorithms, like the work-stealing algorithms of Arora et al. [2] are not fault tolerant, in the sense that process failures, though non-blocking, can cause the loss of items.

3. THE LOCALITY-GUIDED POLICY

In this section we describe work-dealing under the *locality-guided policy*. The policy fits applications where the work represented by any work-item i can be executed more efficiently on a specific process *affil(i)* than on other processes.

The policy uses an algorithm-parameter $L, L > 2$, that represents the extent of distribution-imbalance allowed. Let us denote by *dist(p,q,t)* the number of items distributed by process p to process q by time t; and let $distAvg(p,t)$ denote the average number of items distributed by p to its produce-PCBs (including p's) by time t, then the *locality-guided* policy distributes all items according to their affinity, as long as the following invariant is not violated:

- *L-balance invariance*: At any time t, and for all p, q it holds that:

$$dist(p, q, t) \leq L \cdot distAvg(p)$$

In the special case where all items are distributed according to their affinity, $L = \infty$.

3.1 Implementation

The `getOracle` for the *locality-guided* policy is the same as the *simple policy*'s `getOracle`, as described in Section 2.3. Figure 5 shows the additional local variables required to implement the locality-guided policy, and the code implementing the `putOracle`.

```
// Local static memory
unsigned long dist[n];
unsigned long distAvg=1;
unsigned int nCount=0;
unsigned int nextPCB;

// The put Oracle
unsigned int putOracle(VALUE val)
{
1   unsigned int affin = val.affin;
2   if (++nCount == n)
       {
3      nCount=0;
4      distAvg++;
       }
5   if (dist[affin] < L*distAvg)
       {
6      dist[affin]++;
7      return affin;
       }
8   else
9      for (nextPCB = ++nextPCB % n;;nextPCB = ++nextPCB % n)
10        if (dist[nextPCB] < 2*distAvg)
             {
11           dist[nextPCB]++;
12           return nextPCB;
             }
}
```

Figure 5: Locality-guided policy variables and code

All values produced by the algorithm contain an `affin` field, storing the index of the process to which they have affinity. Each process p has an array `dist`; entry i stores the number of items distributed by p to process i. The `distAvg` long variable stores an approximation of the average number of items distributed by p. (It can easily be seen, that this approximation always misses the average by at most 1.) Finally, `nextPCB` is a Round Robin index.

The `putOracle` receives the produced-value `val` as its input and returns the index of the process to whose PCB the new value should be added. `putOracle` extracts from `val` the id of the process to which it has affinity - `affin`. If the number of items distributed by p to `affin` is less than L times `distAvg`, then `affin` is returned, indicating that the new value should be added to `affin`'s PCB; otherwise, so as to balance the system, the `dist` array is scanned in a Round Robin manner, until a process-id q is found to which p distributed so far less than twice the average, and the function returns q. The `nCount` counter is incremented on every activation and is used to increment `distAvg` every n items produced, so as to eliminate the need for a divide-operation for every item distributed.

4. PERFORMANCE ANALYSIS

In this section we analyze the performance of the work-dealing algorithm. After providing some definitions and notation, we analyze the memory overhead of the *RS* item recycling scheme, which is utilized by all policies, and then proceed with analysis of the *simple policy* followed by analysis of the *locality guided* policy.

4.1 Definitions and Notation

The computation is performed by a group of processes, Q, of size n. We assume each process has a unique index in the range [0..n-1] and we regard Q interchangeably as a group of indexes and as a group of processes.

Denote by *dist(t)* the number of items distributed by the computation up-to (and including) time t. We identify *dist(t)* with the number of activations of *putItem* that finished execution during the computation, up to time t. We denote by *dist(p,t)* the number of items which have been put into process p's consume-PCBs (either by p itself or by other processes) up to time t. Let i be an item. Initially we assume that the execution of the work represented by i takes the same time, no matter on which process it is executed, and we denote this time by $w(i)$. We relax this assumption later. let $W(p,t)$ denote the work represented by all the items that have been distributed to p up-to (and including) time t; also let W(t) denote the work represented by all the items that have been distributed to all processes up-to (and including) time t, and let $W_A(t) = \frac{W(t)}{n}$ be average-work distributed to a process. Finally, we say that the work-dealing by time t is $\alpha - balanced$, if the following holds:

$$\forall q \in Q : |\frac{W(q,t) - W_A(t)}{W_A(t)}| \leq \alpha$$

4.2 RS item recycling scheme

The following Theorem proves that by using the *RS* recycling scheme, work dealing can completely avoid memory-overflow while incurring only a small memory-overhead.

THEOREM 2. *Assume the maximal number of items which would exist simultaneously in the system, M, is know a priori; then work-dealing under any policy, using RS with granularity G, and fixed lists number of $\lceil \frac{M}{G} \rceil + 2n$, completely avoids memory-overflow, with memory overhead of at most $(2n+1)G$ items.*

PROOF. Any process p, at any time t, has a single *produceList* and a single *consumeList*, that can contain at most $2G$ unused items and so the maximal memory-overhead at any time due to these lists is $2nG$ items. The round-up of $\lceil \frac{M}{G} \rceil$ may contribute an additional overhead of less than G items. All other items are either used or are available for use of any process at the RS-pool, and thus no overflow would occur. □

Note that contrary to work-dealing, array-based work-stealing must generally incur a much larger memory-overhead if memory-overflows are to be completely avoided. For extremely unbalanced applications, $\Omega(nM)$ memory may be required.

4.3 Simple-Policy Analysis

In the analysis of the simple policy, we assume that the work represented by items generated by any process during an execution can be modeled by a random variable R with expectance μ and variance σ^2. R's coefficient-of-variance $(\frac{\mu}{\sigma})$ is denoted by CV.

The following theorem states, that under the *simple policy*, generated-items are distributed equally between processes.

THEOREM 3. *At any time t during the computation under the* simple policy*, the numbers of items distributed to different processes differ by at most n:*

$$\forall p, q, t : |dist(p,t) - dist(q,t)| \leq n$$

PROOF. With the *basic policy*, every process p distributes the items produced by it in a round-robin manner. Consequently, p's contribution to a distribution-imbalance between any pair of processes at all times is at most 1. □

We consider computations where the number of items generated throughout execution is much higher than n. We analyze the system-load at time t, after a large number of items has been produced, and we assume that at that time the following holds:

$$\forall p, q, t : dist(p,t) = dist(q,t). \tag{1}$$

Theorem 3 guarantees that this is a good approximation.

The following theorem proves that the simple-policy makes the work-distribution optimal up-to-a-constant with high-probability, provided enough items are produced.

THEOREM 4. *For every p, $0 < p < 1$ and α, $\alpha > 0$, there exists a number $K(p,\alpha)$ such that if more than $K(p,\alpha)$ items have been distributed up to time t according to the* simple policy*, the work-dealing by time t is $\alpha-$balanced in probability at least p. Moreover:*

$$k(p,\alpha) \leq (CV)^2 n \frac{1}{(1 - \sqrt[n]{p})\alpha^2} \tag{2}$$

PROOF. Observe, that $W(q,t)$ is the sum of a random-sample of size $dist(q,t)$ from a distribution with mean μ and variance σ^2. Consequently, at any time t and for any process q, the following holds:

$$E[W(q,t)] = dist(q,t)\mu, \qquad Var[W(q,t)] = dist(q,t)\sigma^2 \tag{3}$$

By using Chebyshev's inequality, we get the following:

$$P\left(|W(q,t) - dist(q,t)\mu| \leq k\sqrt{dist(q,t)}\sigma\right) \geq p \tag{4}$$

Consequently:

$$P\left(\forall q \in Q : |W(q,t) - dist(q,t)\mu| \leq k\sqrt{dist(q,t)}\sigma\right) \geq (1-\frac{1}{k^2})^n \tag{5}$$

Note, that $E[W(t)] = dist(t)\mu$, and so the optimal expected work-distribution is for each process to have received items representing $\frac{dist(t)\mu}{n}$ work.

Fix $k_p = \sqrt{\frac{1}{(1-\sqrt[n]{p})}}$ in Equation 5 to get a specific probability p. We now wish to make the relative deviation of $W(q,t)$ from the optimum smaller than α, for all processes, with probability p.We get:

$$\begin{aligned} \frac{k_p\sqrt{dist(q,t)}\sigma}{dist(q,t)\mu} \leq \alpha \quad &\Longrightarrow \quad \frac{\sigma}{\mu}\frac{k_p}{\sqrt{dist(q,t)}} \leq \alpha \\ &\Longrightarrow \quad dist(q,t) \geq (CV)^2(\tfrac{k_p}{\alpha})^2 \end{aligned} \tag{6}$$

Noting that $dist(q,t) = \frac{dist(t)}{n}$ completes the proof.

□

If a work-distribution scheme has the property proven in Theorem 4, we say the scheme is $K(p,\alpha)$ *eventually fair*.

Following the proof of Theorem 4, it is clear that we can somewhat relax the assumption that all work-items take exactly the same time on all processes and still prove *eventual fairness*. The following theorem states just that.

THEOREM 5. *Let $w(i,p)$ denote the time it takes process p to execute the work represented by item i, and assume there is a constant C such that the following holds for all items i:*

$$\forall p, q \in Q : \frac{w(i,p)}{w(i,q)} \leq C$$

Then for every p, $0 < p < 1$ and α, $\alpha > 0$, there exists a number $K(p,\alpha)$ such that if more than $K(p,\alpha)$ items have been distributed up to time t according to the simple policy*, the work-dealing by time t is $C(1+\alpha)-$balanced in probability at least p, with the same $k(p,\alpha)$ as in Theorem 4*

The proof is almost identical to that of Theorem 4, except that the random-variable R now represents the *average − time* it takes to execute a work-item over all the processes.

Using Theorem 5, we can now prove that execution-time under the *simple policy* approaches the optimum with high probability, provided enough items are produced. We derive this result for a category of applications defined as follows:

Let A be an application. We say that A has an *(n,C)-phased generation pattern* if for any execution of A, either serial or parallel, the items it generates can be inductively partitioned as follows:

- A generates a total of K items, where $nC \leq K < (n+1)C$.
- We denote by G_0 the group of first C items generated by A. Let W represent the total time it takes to process all the items A generates (if the execution is parallel, execution times for all processes are summed), and assume execution starts at time 0, then all of the items in G_0 are generated by time $\frac{W}{n}$.
- Let $G_i, 0 \leq i \leq (n-2)$ be the i'th group of items. We denote by G_{i+1} the group of items generated throughout the execution of the items in G_i. It holds that: $|G_{i+1}| \geq C$.
- We denote by G_n all the work-items that do not belong to any of the groups $G_i, 0 \leq i \leq (n-1)$. Clearly, $|G_n| < C$. Let $W(G_n)$ denote the time it takes to execute all of the items in G_n. It holds that $W(G_n) < \frac{W}{n}$.

The above definition captures the idea that many highly parallel applications can be viewed as if work is generated in a number of phases. In each phase, while some number C of items are being processed, another amount C of items is generated as a result of this processing. This kind of execution pattern implies that there is enough concurrency in the sense that there are no prolonged "idle periods": while items are worked on, sufficiently many new items are generated to fill the processsing pipeline.

The following theorem proves that for applications that have such an item-generation pattern, *eventual fairness* ensures asymptotic optimal performance.

THEOREM 6. *Assume a work-distribution scheme DS is* $K(p,\alpha)$ eventually-fair. *Let A be an* $(n, K(q,\alpha))$*-phased application, where* $q > \sqrt[n]{p}$. *Let* $Optimal(A)$ *and* $DS(A)$ *denote the optimal speed-up execution-time of A and the execution-time under the DS scheme, respectively, in an n-process system, then the following holds:*

$$P\left(\frac{DS(A)}{Optimal(A)} \leq 3+\alpha\right) \geq p$$

PROOF. Assume execution starts at time 0. Since A is $(n, K(q,\alpha))$-phased, the items in G_0 are generated by the DS scheme by time $\frac{W}{n}$ at the latest. After that, execution of all the items in any group $G_i, 0 \leq i \leq (n-1)$ takes no more than $(1+\alpha)\frac{W(G_i)}{n}$ time with probability at least q. Consequently, all the items in groups $W_i, 0 \leq i \leq (n-1)$ are processed by DS in time less than $\frac{(1+\alpha)}{n}\sum_{0\leq i\leq n-1} W(G_i)$, with probability at least $q^n \geq p$. Finally, all the items in G_n are processed by DS in time no more than $\frac{W}{n}$. It follows, that the total execution time of A under DS is less than $(3+\alpha)\frac{W}{n}$, with probability at least p. Noting that $Optimal(A) \geq \frac{W}{n}$ finishes the proof. □

We get as a corollary, that work-distribution under the *simple policy* has asymptotic optimal performance:

THEOREM 7. *Let A be an* $(n, K(q,\alpha))$*-phased application, where* $q > \sqrt[n]{p}$. *Let* $Optimal(A)$ *and* $WD_S(A)$ *denote the optimal speed-up execution-time of A and the execution-time under work-dealing with the simple policy, respectively, in an n-process system, then the following holds:*

$$P\left(\frac{WD_S(A)}{Optimal(A)} \leq 3+\alpha\right) \geq p$$

4.4 Locality-Guided-Policy Analysis

For the *locality-guided*-policy analysis, we assume that every generated item i has a single process, *affil(i)*, to which it is affiliated. We analyze the policy with $L = \infty$, namely all items are distributed to the process to which they are affiliated, and we make the following simplifying assumptions:

- For every item i generated during the computation, i has affinity to all processes with equal probability, namely:

$$\forall i, \forall q \in Q : P\left(\text{affil}(i) = q\right) = \frac{1}{n}$$

- As we did for the *simple*-policy, we assume that the work represented by any item i can be modeled by a random variable R with expectance μ and variance σ^2 and we denote R's coefficient-of-variance $(\frac{\mu}{\sigma})$ by CV.

We denote by $dist_A(t) = \frac{dist(t)}{n}$ the average number of items distributed to a process by time t; we say that the *item-dealing* by time t is $\alpha-balanced$, if the following holds:

$$\forall q \in Q : |\frac{dist(q,t) - dist_A(t)}{dist_A(t)}| \leq \alpha$$

The following theorem proves, that the item-dealing is asymptotically balanced.

THEOREM 8. *For every probability* $p, 0 < p < 1$ *and* $\alpha, \alpha > 0$*, there exists a number* $K(p,\alpha)$ *such that if more than* $K(p,\alpha)$ *items have been distributed up to time t according to their affinity, the* item-dealing *by time t is* α *- balanced in probability at least p. Moreover:*

$$K(p,\alpha) \leq \frac{n}{\alpha(1-\sqrt[n]{p})} \tag{7}$$

PROOF. $dist(q,t)$ is exactly the number of generated items whose affinity process is q. Consequently, $dist(q,t)$ is a binomial random variable with parameters $(dist(t), \frac{1}{n})$. It follows that:

$$\begin{array}{lcl} E[dist(q,t)] & = & dist_A(t) \\ Var[dist(q,t)] & = & dist_A(t)(1-\frac{1}{n}) = \Theta(dist_A(t)) \end{array} \tag{8}$$

By applying Chebyshev's inequality to Equations 8 we get:

$$P\left(|dist(q,t) - dist_A(t)| \leq k\sqrt{dist_A(t)}\right) \geq 1 - \frac{1}{k^2} \tag{9}$$

It follows that:

$$P\left(\forall q \in Q : |dist(q,t) - dist_A(t)| \leq k\sqrt{dist_A(t)}\right) \geq (1-\frac{1}{k^2})^n \tag{10}$$

Fix $K_p = \sqrt{\frac{1}{1-\sqrt[n]{p}}}$ in Equation 10 to get a specific probability p. We now wish to make the relative deviation of $dist(q,t)$ from the average smaller than α, for all processes, with probability p. We get:

$$\begin{array}{rcl} \frac{k_p\sqrt{dist_A(t)}}{dist_A(t)} \leq \alpha & \Longrightarrow & dist_A(t) \geq (\frac{K_p}{\alpha})^2 \\ & \Longrightarrow & dist(t) \geq n(\frac{K_p}{\alpha})^2 \\ & \Longrightarrow & dist(t) \geq \frac{n}{\alpha(1-\sqrt[n]{p})} \end{array}$$

□

Based on Theorem 8 we get as corollary the following theorem, corresponding to Theorem 4 for the *simple policy*:

THEOREM 9. *For every p,* $0 < p < 1$ *and* α, $\alpha > 0$*, there exists a number* $K(p,\alpha)$ *such that if more than* $K(p,\alpha)$ *items have been distributed up to time t according to the* locality guided policy *with* $L = \infty$*, the work-dealing by time t is* $\alpha - balanced$ *in probability at least p.*

The proof is almost identical to that of Theorem 4. Finally, based on Theorems 9 and 6, we get the following as corollary:

THEOREM 10. *Let A be an* $(n, K(q,\alpha))$*-phased application, where* $q > \sqrt[n]{p}$. *Let* $Optimal(A)$ *and* $WD_{LG\infty}(A)$ *denote the optimal speed-up execution-time of A and the execution-time under work-dealing with the locality-guided policy with* $L = \infty$*, respectively, in an n-process system, then the following holds:*

$$P\left(\frac{WD_{LG\infty}(A)}{Optimal(A)} \leq 3+\alpha\right) \geq p$$

An important question regarding a locality-guided distribution scheme is: What is the probability for any item e to be executed by the process to which it has affinity? We call this probability, the *affinity hit-ratio* of the scheme. The following theorem bounds the *affinity hit ratio* from below.

THEOREM 11. *The work-dealing algorithm, under the* locality guided *policy with parameter L, has expected* affinity hit-ratio *of at least* $1 - \frac{1}{L}$

PROOF. An item i produced by process p at time t is distributed to *affil(i)*, unless:

$$dist(p, \text{affil}(i), t) > L \cdot distAvg(p)$$

Consequently there are at most $\frac{n}{L}$ processes to which i cannot be distributed at time t. Since we assume that *affil(p)* has a uniform distribution, the theorem follows. □

5. ACKNOWLEDGMENTS

We wish to thank Yossi Azar, Dave Detlefs, Maurice Herlihy, Victor Luchangco, Mark Moir and Sivan Toledo for their valuable comments. The key idea of applying our work-dealing algorithm to improve the locality of work distribution was suggested to us by Dave Detlefs.

6. 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] DETLEFS, D. L., MARTIN, P. A., MOIR, M., AND JR., G. L. S. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing* (2001).

[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] HENDLER, D., AND SHAVIT, N. Non-blocking steal-half work queues. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing* (2002).

[7] HERLIHY, M., LUCHANGCO, V., MARTIN, P., AND MOIR, M. Dynamic-sized lockfree data structures, 2002. Technical Report TR-2002-110, Sun Microrsystems Laboratories.

[8] LAMPORT, L. Specifying concurrent program modules, 1993.

[9] 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.

[10] MICHAEL, M. M., AND SCOTT, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Symposium on Principles of Distributed Computing* (1996), pp. 267–275.

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

[12] NARLIKAR, G. J. Scheduling threads for low space requirement and good locality. In *ACM Symposium on Parallel Algorithms and Architectures* (1999), pp. 83–95.

[13] 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.

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