Dynamic Memory ABP Work-Stealing

Danny Hendler[1], Yossi Lev[2], and Nir Shavit[2]

[1] Tel-Aviv University
[2] Sun Microsystems Laboratories & Tel-Aviv University

Abstract. The non-blocking work-stealing algorithm of Arora, Blumofe, and Plaxton (hencheforth *ABP work-stealing*) is on its way to becoming the multiprocessor load balancing technology of choice in both Industry and Academia. This highly efficient scheme is based on a collection of array-based deques with low cost synchronization among local and stealing processes. Unfortunately, the algorithm's synchronization protocol is strongly based on the use of fixed size arrays, which are prone to overflows, especially in the multiprogrammed environments which they are designed for. This is a significant drawback since, apart from memory inefficiency, it means users must tailor the deque size to accommodate the effects of the hard-to-predict level of multiprogramming, and add expensive blocking overflow-management mechanisms.
This paper presents the first *dynamic memory* work-stealing algorithm. It is based on a novel way of building non-blocking dynamic memory ABP deques by detecting synchronization conflicts based on "pointer-crossing" rather than "gaps between indexes" as in the original ABP algorithm. As we show, the new algorithm dramatically increases robustness and memory efficiency, while causing applications no observable performance penalty. We therefore believe it can replace array-based ABP work-queues, eliminating the need to add application specific overflow mechanisms.

1 Introduction

The ABP work-stealing algorithm of Arora, Blumofe, and Plaxton [1] has been gaining popularity as the multiprocessor load-balancing technology of choice in both Industry and Academia [2, 1, 3, 4]. The scheme implements a provably efficient work-stealing paradigm due to Blumofe and Leiserson [5] that allows each process to maintain a local work deque, and steal an item from others if its deque becomes empty. It has been extended in various ways such as stealing multiple items [6] and stealing in a locality-guided way [2]. At the core of the ABP algorithm is an efficient scheme for stealing an item in a non-blocking manner from an array-based deque, minimizing the need for costly Compare-and-Swap (CAS) synchronization operations when fetching items locally.

Unfortunately, the use of fixed size arrays[1] introduces an inefficient memory-size/robustness tradeoff: for n processes and total allocated memory size m, one

[1] One may use cyclic array indexing but this does not help in preventing overflows.

R. Guerraoui (Ed.): DISC 2004, LNCS 3274, pp. 188–200, 2004.

can tolerate at most $\frac{m}{n}$ items in a deque. Moreover, if overflow does occur, there is no simple way to malloc additional memory and continue. This has, for example, forced parallel garbage collectors using work-stealing to implement an application specific blocking overflow-management mechanism [3, 7]. In multiprogrammed systems, the main target of ABP work-stealing [1], even inefficient over-allocation based on an application's maximal execution-DAG depth [1, 5] may not always work. If a small subset of non-preempted processes end up queuing most of the work items, since the ABP algorithm sometimes starts pushing items from the middle of the array even when the deque is empty, this will lead to overflow.[2]

This state of affairs leaves open the question of designing a dynamic memory algorithm to overcome the above drawbacks, but to do so while maintaining the low-cost synchronization overhead of the ABP algorithm. This is not a straightforward task, since the the array-based ABP algorithm is quite unique: it is possibly the only real-world algorithm that allows one to transition in a lock-free manner from the common case of using loads and stores to using a costly CAS *only* when a potential conflict requires processes to reach consensus. This somewhat-magical transition rests on the ability to detect these boundary synchronization cases based on the relative gap among array indexes. There is no straightforward way of translating this algorithmic trick to the pointer based world of dynamic data structures.

1.1 The New Algorithm

This paper introduces the first lock-free[3] *dynamic-memory* version of the ABP work-stealing algorithm. It provides a near-optimal memory-size/robustness tradeoff: for n processes and total pre-allocated memory size m, it can potentially tolerate up to $O(m)$ items in a single deque. It also allows one to malloc additional memory beyond m when needed, and as our empirical data shows, it is far more robust than the array-based ABP algorithm in multiprogrammed environments.

An ABP style work-stealing algorithm consists of a collection of `deque` data structures with local processes performing pushes and pops on the "bottom" end of the deque and multiple thieves performing pops on the "top" end. The new algorithm implements the deque as a doubly-linked list of $\Theta(m)$ nodes, each of which is a short array that is allocated and freed dynamically from a shared pool. It can also use malloc to add nodes to the shared pool in case its node supply is exhausted.

The main technical difficulties in the design of the new algorithm arise from the need to provide performance comparable to that of ABP. This means the doubly linked list must be manipulated using only loads and stores in the common case, and transition in a lock-free manner to using a costly CAS *only* when a potential conflict requires consensus.

[2] The ABP algorithm's built-in "reset on empty" heuristic may help in some, but not all, of these cases.

[3] Our abstract Deque definition is such that the original ABP algorithm is also lock-free.

The potential conflict which requires CAS-based synchronization occurs when a pop by a local process and a pop by a thief might both be trying to remove the same item from the deque. The original ABP algorithm detects this scenario by examining the gap between the `Top` and `Bottom` array indexes, and uses a CAS operation only when they are "too close." Moreover, in the original algorithm, the empty deque scenario is checked simply by checking whether `Bottom` $\leq$ `Top`.

A key algorithmic feature of our new algorithm is the creation of an equivalent mechanism to allow detection of these boundary situations in our linked-list structures using the relations between the `Top` and `Bottom` pointers, even though these point to entries that may reside in different nodes. On a high level, our idea is to prove that one can restrict the number of possible ways the pointers interact, and therefore, given one pointer, it is possible to calculate the different possible positions for the other pointer which implies such a boundary scenario. The different empty deque scenarios are depicted in Figure 2.

The other key feature of our algorithm is that the dynamic insertion and deletion operations of nodes into the doubly linked-list (when needed in a push or pop) is performed so the local thread uses only loads and stores. This contrasts, at least intuitively, with the more general linked-list deque implementations [8, 9] which require a double-compare-and-swap synchronization operation [10] to insert and delete nodes.

1.2 Performance Analysis

We compared our new dynamic-memory work-stealing algorithm to the original ABP algorithm on a 16-node shared memory multiprocessor using the benchmarks of the style used by Blumofe and Papadopoulos [11]. We ran several standard *Splash2* [12] applications using the Hood scheduler [13] with the ABP and new work-stealing algorithms. Our results, presented in Section 3, show that the new algorithm performs as well as ABP, that is, the added dynamic-memory feature does not slow the applications down. Moreover, the new algorithm provides a better memory/robustness ratio: the same amount of memory provides far greater robustness in the new algorithm than the original array-based ABP work-stealing. For example, running Barnes-Hut using ABP work-stealing with an 8 fold level of multiprogramming causes a failure in 40% of the executions if one uses the deque size that works for stand-alone (non-multiprogrammed) runs. It causes *no failures* on using the new dynamic memory work-stealing algorithm.

2 The Algorithm

2.1 Basic Description

An ABP style work-stealing algorithm consists of a collection of `deque` data structures. Each deque is local to some process, which can perform LIFO `Push` and `Pop` operations on it (`PushBottom` and `PopBottom` on the "bottom" end of the deque), and is remote to multiple potential thieves, which can perform FIFO `Pop` operations on it (`PopTop` on the "top" end of the deque).

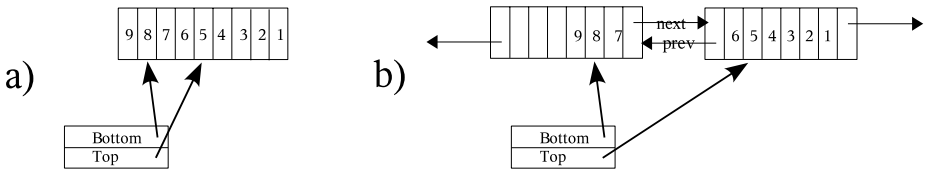

Fig. 1. The original ABP deque structure (a) vs. that of the new dynamic deque. (b) The structure after 9 `PushBottom` operations, 4 successful `PopTop` operations, and one `PopBottom` operation (In practice the original ABP deque uses cell indexes and not pointers as in our illustration).

The new algorithm implements the deque as a doubly-linked list of short arrays, as depicted in Figure 1 (the size of the arrays is a tuneable parameter). The nodes are allocated and freed from a shared pool, and the only case in which one may need to malloc additional storage is if the shared pool is exhausted.

The main technical difficulty in our design arises from the wish to maintain the same synchronization efficiency that characterizes the ABP algorithm; We use only loads and stores for `PushBottom` and `PopBottom` in the common case, and transition in a lock-free manner to using a costly CAS *only* when a potential conflict requires processes to reach consensus. This potential conflict occurs when the local `PopBottom` and a thieve's `PopTop` might concurrently try to remove the same item from the deque. The original ABP algorithm detects this scenario by examining the gap between the `Top` and `Bottom` array indexes, and uses a CAS operation only when they are "too close." Moreover, in the original algorithm, the empty deque scenario is checked simply by checking whether `Bottom` $\leq$ `Top`.

In the new linked-list structure, we need an equivalent mechanism to allow us to detect these situations even if the `Top` and `Bottom` pointers point to array entries that reside in different nodes. Our solution is to prove that one can restrict the number of possible scenarios among the pointers. Given some pointer, we show that the "virtual" distance of the other, ignoring which array it resides in, can be no more than 1. We can thus easily test for each of these scenarios. (Several such scenarios are depicted in parts (a) and (b) of Figure 2).

The next problem one faces is the maintenance of the deque's doubly-linked list structure. We wouldn't like to use CAS operations when updating the `next` and `previous` pointers, since this will cause a significant performance penalty. Our solution is to allow only the local process to update these fields, thus preventing `PopTop` operations from doing so when moving from one node to another. We would like to keep the deque dynamic, which means freeing old nodes when they're not needed anymore. This restriction immediately implies that an active list node may point to an already freed node, or even to a node which was freed and reallocated again, essentially ruining the list structure. As we prove, the algorithm can overcome this problem by having a `PopTop` operation that moves to a new node, free only the node preceding the old node and not the old node itself. This allows us to maintain the invariant that the doubly-linked list structure between the `Top` and `Bottom` pointers is preserved. This is true even in scenarios such as that in Figure 2 where the pointers cross over.

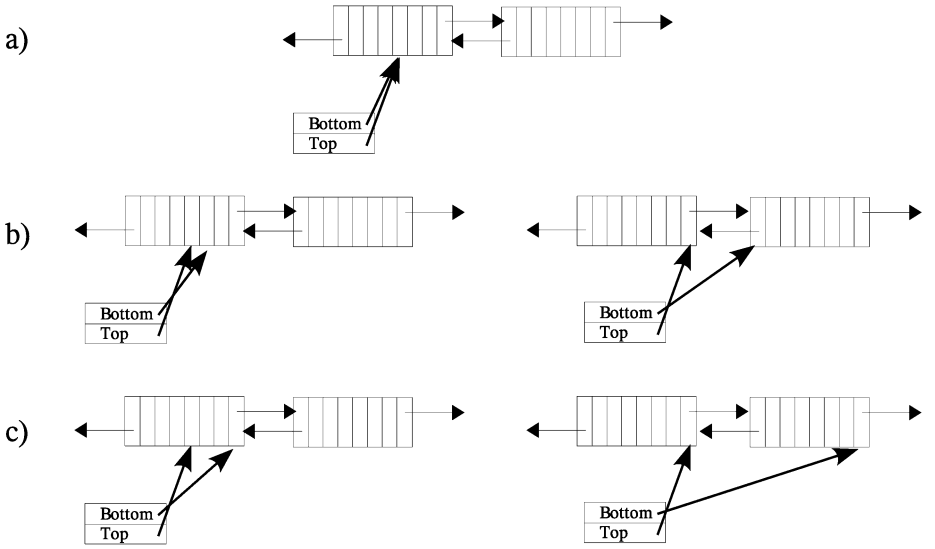

Fig. 2. The different types of empty deque scenarios. (a) Simple: `Bottom` and `Top` point to the same cell. (b) Simple Crossing: both the left and right scenarios are examples where `Bottom` passed over `Top` by one cell, but they still point to neighboring cells. (c) Non-Simple Crossing (with the reset-on-empty heuristic): both the left and right scenarios are examples of how pointers can cross given the reset-on-empty heuristic, between the reset of `Bottom` to the reset of `Top`.

Finally, given that the `PopTop` operation may be executed concurrently by many processes, the node that is pointed to by `Top` at the beginning of the method may be freed during the method execution. We must thus limit the `PopTop` method to read (in order to pop) array entries only within the `Top` node and not across nodes.

2.2 The Implementation

The `C++` like Pseudo Code for the different deque methods is given in Figures 3 and 4. The deque object saves the `Bottom` and `Top` pointers information in the `Bottom` and `Top` data members, and uses the *EncodeBottom*, *DecodeBottom*, *EncodeTop* and *DecodeTop* macros to encode/decode this information into a CAS-able size word. Underlined commands in the Pseudo Code stand for code blocks which will be described later. We now describe each of the methods.

PushBottom. The method begins by reading `Bottom` and storing the pushed value in the cell it is pointing to (Lines 1-2). Then it calculates the next value of `Bottom` linking a new node to the list if necessary (Lines 3-14). Finally the method updates `Bottom` to its new value (Line 15). As in the original ABP algorithm, this method is executed only by the owner process, and therefore regular writes suffice (both for the value and `Bottom` updates). Note that the new node is linked to the list before `Bottom` is updated, so the list structure is preserved for the nodes between `Bottom` and `Top`.

```
void DynamicDedeque::PushBottom(ThreadInfo theData)
{
1  <currNode, currIndex> = DecodeBottom(Bottom); // Read Bottom data
2  currNode->itsDataArr[currArrIndex] = theData; // Write data in current bottom cell
3  if (currIndex!=0)
4  {
5               newNode = currNode;
6               newIndex = currIndex-1;
7  }
8  else
9  {  // Allocate and link a new node:
10              newNode = AllocateNode();
11              newNode->next = currNode;
12              currNode->prev = newNode;
13              newIndex = DequeNode::ArraySize-1;
14 }
15 Bottom = EncodeBottom(newNode,newArrIndex);    // Update Bottom
}



ThreadInfo DynamicDedeque::PopTop()
{
16 currTop = Top;     // Read Top
17 <currTopTag, currTopNode, currTopIndex> = DecodeTop(currTop);
18 currBottom = Bottom; // Read Bottom
19 EmptinessTest(currBottom,currTop);

20 if (currTopIndex!=0)     // if deque isn't empty, calculate next top pointer:
21 {  // stay at current node:
22              newTopTag = currTopTag;
23              newTopNode = currTopNode;
24              newIndex = currIndex-1;
25 }
26 else
27 {  // move to next node and update tag:
28              newTopTag = currTopTag+1;
29              newTopNode = currTopNode->prev;
30              newTopIndex = DequeNode::ArraySize-1;
31 }
32 retVal = currTopNode->itsDataArr[currTopIndex]; // Read value
33 newTopVal = Encode(newTopTag,newTopNode,newTopIndex);
34 if (CAS(&Top, currTop, newTopVal))  //Try to update Top using CAS
35 {
36              FreeOldNodeIfNeeded();
37              return retVal;
38 }
39 else
40 {
41              return ABORT;
42 }
}
```

Fig. 3. Pseudo Code for the `PushBottom` and `PopTop` operations.

PopTop. The method begins by reading the `Top` and `Bottom` values, in that order (Lines 16-18). Then it checks whether these values indicate an `EMPTY` deque, and returns if they do (Line 19). Otherwise, it calculates the next position for `Top` (Lines 20-31). Before updating `Top` to its new value, the method must read the value which should be returned if the steal succeeds (Line 32) (this read

```
ThreadInfo DynamicDedeque::PopBottom()
{
43 <oldBotNode,oldBotIndex > = DecodeBottom(Bottom); // Read Bottom Data
44 if (oldBotIndex != DequeNode::ArraySize-1)
45 {
46              newBotNode = oldBotNode;
47              newBotIndex = oldBotIndex+1;
48 }
49 else
50 {
51              newBotNode = oldBotNode->next;
52              newBotIndex = 0;
53 }
54 retVal = newBotNode->itsDataArr[newBotIndex];      // Read data to be popped
55 Bottom = EncodeBottom(newBotNode,newBotIndex);     // Update Bottom
56 currTop = Top;                                     // Read Top
57 <currTopTag,currTopNode,currTopIndex> = DecodeTop(currTop);

58 if (oldBotNode == currTopNode &&          // Case 1: if Top has crossed Bottom
59     oldBotIndex == curTopIndex  )
60 {
      //Return bottom to its old possition:
61              Bottom = EncodeBottom(oldBotNode,oldBotIndex);
62              return EMPTY;
63 }
64 else if ( newBotNode == currTopNode &&    // Case 2: When popping the last entry
65           newBotIndex == currTopIndex )   //         in the deque (i.e. deque is
66 {                                         //         empty after the update of bottom).

      //Try to update Top's tag so no concurrent PopTop operation will also pop the same entry:
67              newTopVal = Encode(currTopTag+1, currTopNode, currTopIndex);
68              if (CAS(&Top, currTop, newTopVal))
69              {
70                 FreeOldNodeIfNeeded();
71                 return retVal;
72              }
73              else  // if CAS failed (i.e. a concurrent PopTop operation already popped the last entry):
74              {
                //Return bottom to its old possition:
75                 Bottom = EncodeBottom(oldBotNode,oldBotIndex);
76                 return EMPTY;
77              }
78 }
79 else // Case 3: Regular case (i.e. there was at least one entry in the deque after bottom's update):
80 {
81                 FreeOldNodeIfNeeded();
82                 return retVal;
83 }
}
```

Fig. 4. Pseudo Code for the `PopBottom` operation.

cannot be done after the update of `Top` since then the node may already be freed by some other concurrent `PopTop` execution). Finally the method tries to update `Top` to its new value using a CAS operation (Line 34), returning the popped value if it succeeds, or `ABORT` if it failed. In case of success, the method also checks if there is an old node that needs to be freed (Line 36). As explained earlier, a node is released only if `Top` moved to a new node, and the node released is not the old `Top`'s node, but its preceding one.

PopBottom. The method begins by reading `Bottom` and updating it to its new value (Lines 43-55) after reading the value to be popped (Line 54). Then it reads the value of `Top` (Line 56), to check for the special cases of popping the last entry of the deque, and popping from an empty deque. If the `Top` value read points to the old `Bottom` position (Lines 58-63), then the method rewrites `Bottom` to its old position, and returns `EMPTY` (since the deque was empty even without this `PopBottom` operation). Otherwise, if `Top` is pointing to the new `Bottom` position (Lines 64-78), then the popped entry was the last in the deque, and like in the original algorithm, the method updates the `Top` tag value using a CAS, to prevent a concurrent `PopTop` operation from popping out the same entry. If neither of the above is true, then there was at least one entry in the deque after the `Bottom` update (lines 79-83), in which case the popped entry is returned. Note that, as in the original algorithm, most executions of the method will be short, and will not involve any CAS-based synchronization operations.

Memory Management. We implement the shared node pool using a variation of Scott's shared pool [14]. It maintains a local group of g nodes per process, from which the thread may allocate nodes without the need to synchronize. When the nodes in this local group are exhausted, it allocates a new group of g nodes from a shared LIFO pool using a CAS operation. When a thread frees a node, it returns it to its local pool, and if the size of the local group exceeds $2g$, it returns g nodes to the shared pool. In our benchmarks we used a group size of 1, which means that in case of a fluctuation between pushing and popping, the first node is always local and CAS is not necessary.

Omitted Code Blocks. We describe here the code segments that were not included in the pseudo code given in Figures 3 and 4:

- *The Emptiness Test in the* `PopTop` *method (Line 19):* This code segment is a possible return point from the `PopTop` method, if the `Bottom` and `Top` pointers, that were read in lines 16 and 18, respectively, indicate an empty deque. The code segment does the following:
  1. Checks if the `Top` and `Bottom` values read indicate an empty deque. The different possible empty deque scenarios will be discussed in the correctness proof.
  2. If an empty deque is detected, the `Top` pointer is read again to see if it was changed from the first read value. If it was indeed changed, ABORT is returned, otherwise EMPTY is returned.
  3. If the deque was not empty, the `PopTop` method continues at Line 20.

- *The reclamation of old list nodes by the* `Pop` *methods:* Nodes may be reclaimed both by `PopTop` and `PopBottom` operations, as follows:
  - *Reclamation of a node by the* `PopTop` *method (Line 36):* The `PopTop` method reclaims a list node if and only if it changed the `Top` pointer and the new `Top` pointer points to a different node than the old one. In this case, the method reclaims the node pointed to by the next pointer of the old `Top` node.

- *Reclamation of a node by the* `PopBottom` *method (Line 70 or 81):* The `PopBottom` method reclaims a list node if and only if it changed the `Bottom` pointer and the new `Bottom` pointer points to a different node than the old one. In this case, the method reclaims the old `Bottom` node.

2.3 Enhancements

We briefly describe two enhancements to the above dynamic-memory deque algorithm.

Reset-on-Empty. In the original ABP algorithm the `PopBottom` operation uses a heuristic that resets `Top` and `Bottom` to point back to the beginning of the array every time it detects an empty deque (including the case of popping the last entry by `PopBottom`). This reset operation is necessary in ABP since it is the only "anti-overflow" mechanism at its disposal.

Our algorithm does not need this method to prevent overflows, since it works with the dynamic nodes. However, adding a version of this resetting feature gives the potential of improving our space complexity, especially when working with large nodes.

There are two issues to be noted when implementing the reset-on-empty heuristic in our dynamic deque. The first issue is that while performing the reset operation, we create another type of empty deque scenario, in which `Top` and `Bottom` do not point to the same cells nor to neighboring ones (see part *c* of Figure 2). This scenario requires a more complicated check for the empty deque scenario by the `PopTop` method (Line 19). The second issue is that we must be careful when choosing the array node to which `Top` and `Bottom` point after the reset. In case the pointers point to the same node before the reset, we simply reset to the beginning of that node. Otherwise, we reset to the beginning of the node pointed to by `Top`. Note, however, that `Top` may point to the same node as `Bottom` and then be updated by a concurrent `PopTop` operation, which may result in changing on-the-fly the node to which we direct `Top` and `Bottom`.

Using a Base Array. In the implementation described, all the deque's nodes are identical and allocated from the shared pool. This introduces a trade-off between the performance of the algorithm and its space complexity: small arrays save space but cost in allocation overhead, while large arrays cost space but reduce the allocation overhead.

Our heuristic improvement is to use a large array for the initial `base` node, allocated for each of the deques, and use the pool only when overflow space is needed. This base node is used only by the process/deque it was originally allocated to, and is never freed to the shared pool. Whenever a `Pop` operation frees this node, it raises a local boolean flag, indicating that the base node is now free. When a `PushBottom` operation needs to allocate and link a new node, it first checks this flag, and if true, links the base node to the deque (instead of a regular node allocated from the shared pool).

3 Performance

We evaluated the performance of the new dynamic memory work-stealing algorithm in comparison to the original fixed-array based ABP work-stealing algorithm in an environment similar to that used by Blumofe and Papadopoulos [11] in their evaluation of the ABP algorithm. Our preliminary results include tests running several standard Splash2 [12] applications using the *Hood Library* [13] on a 16 node Sun Enterprise™ 6500, an SMP machine formed from 8 boards of two 400MHz UltraSparc® processors, connected by a crossbar UPA switch, and running a Solaris™ 9 operating system.

Our benchmarks used the work-stealing algorithms as the load balancing mechanism in Hood. The Hood package uses the original ABP deques for the scheduling of threads over processes. We compiled two versions of the Hood library, one using an ABP implementation, and the other using the new implementation. In order for the comparison to be fair, we implemented both algorithms in `C++`, using the same tagging method.

We present our results running the *Barnes Hut* and *MergeSort* Splash2 [12] applications. Each application was compiled with the minimal ABP deque size needed for a stand-alone run with the biggest input tested. For the Dynamic deque version we've chosen a base-array size of about 75% of the ABP deque size, a node array size of 6 items, and a shared pool size such that the total memory used (by the deques and the shared pool together) is no more than the total memory used by all ABP deques. In all our benchmarks the number of processes equaled the number of processors on the machine.

Figure 5 shows the total execution time of both algorithms, running stand-alone, as we vary the input size. As can be seen, there is no real difference in performance between the two approaches. This is in spite of the fact that our tests show that the deque operations of the new algorithm take as much as 30% more time on average than those of ABP. The explanation is simple: work stealing accounts for only a small fraction of the execution time in these (and in fact in most) applications. In all cases both algorithms had a 100% completion rate in stand-alone mode, i.e. none of the deques overflowed.

Figure 6 shows the results of running the *Barnes Hut* [12] application (on the largest input) in a multiprogrammed fashion by running multiple instances of Hood in parallel. The graph shows the completion rate of both algorithms as a function of the multiprogramming level (i.e. the number of instances run in parallel). One can clearly see that while both versions perform perfectly at a multiprogramming level of 2, ABP work-stealing degrades rapidly as the level of multiprogramming raises, while the new algorithm maintains its 100% completion rate. By checking Hood's statistics regarding the amount of work performed by each process, we noticed that some processes complete 0 work, which means much higher work loads for the others. This, we believe, caused the deque size which worked for a stand-alone run (in which the work was more evenly distributed between the processes) to overflow in the multiprogrammed run. We also note that as the work load on individual processes increases, the chances of a "reset-on-empty" decrease, and the likelihood of overflow increases. In the

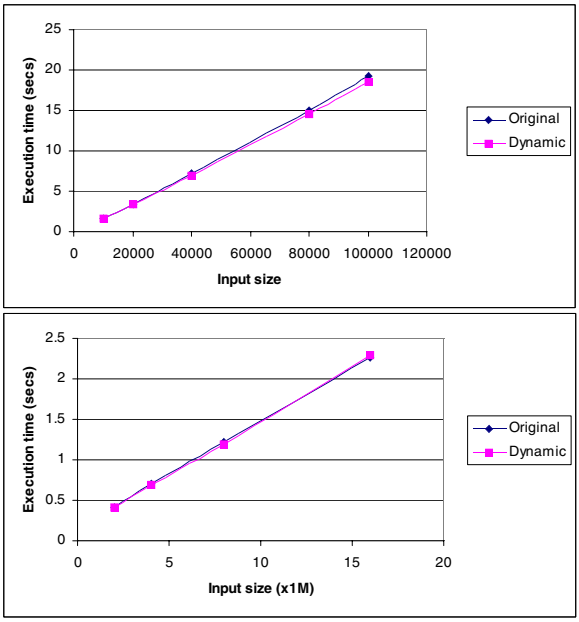

Fig. 5. *Barnes Hut* Benchmark on top and *Msort* on the bottom.

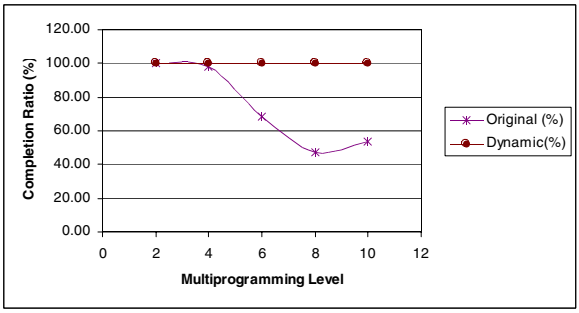

Fig. 6. *Barnes Hut* completion ratio vs. level of multiprogramming.

new dynamic version, because 25% of the memory is allocated in the common shared pool, there is much more flexibility in dealing with the work imbalance between the deques, and no overflow occurs.

Our preliminary benchmarks clearly show that for the same amount of memory, we get significantly more robustness with the new dynamic algorithm than with the original ABP algorithm, with a virtually unnoticeable effect on the application's overall performance. It also shows that the deque size depends on the maximal level of multiprogramming in the system, an unpredictable parameter which one may want to avoid reasoning about by simply using our new dynamic memory version of the ABP work stealing algorithm.

4 Proof

Our full paper will provide a proof that the algorithm is a linearizable implementation of an ABP style deque. Our specification differs slightly from that of ABP to allow one to use linearizability [15] as the consistency condition and not a weaker specialized form of serializability as used in the proof of the original ABP algorithm [1, 16]. Our revised specification is designed to allow an `ABORT` as a return value from `PopTop`, which does not affect the actual implementation algorithms, but serves to simplify their proofs.

There are two main claims that need to be proved about our algorithm: that it is lock-free, and that it is linearizable to the sequential specification of the ABP deque. The first claim is trivial, since the algorithm contains no loops, and the only case in which a `PopTop` operation returns `ABORT` is if some other concurrent operation changed `Top` and therefore made progress.

The linearizability proof, however, is much more complex. Here we provide only the linearization points of each of the deque's methods:

PushBottom. The linearization point of this method is always the `Bottom` update operation in the end of the method (Figure 3, Line 15).

PopBottom. The linearization point of this method depends on its returned value. In case the return value is:

- `EMPTY`: The linearization point here is the read of the `Top` pointer (Figure 4, Line 56).
- A deque entry: The linearization point here is the `Bottom` update (Figure 4, Line 55).

PopTop. The linearization point of this method depends on its returned value. In case the return value is:

- `EMPTY`: The linearization point here is the read of the `Bottom` pointer (Figure 3, Line 18).
- `ABORT`: The linearization point here is the operation that first observed the change of `Top`. This is either the CAS operation (Line 34), or a reread of `Top` done inside the emptiness test code block.
- A deque entry: If the deque was not empty right before the CAS operation at Line 34, the linearization point is that CAS operation. Otherwise, it is the first operation that changed the deque to be empty, in the interval after the execution of Line 18, and right before the execution of the CAS operation at Line 34.

5 Conclusions

We have shown how to create a dynamic memory version of the ABP work stealing algorithm. It may be interesting to see how our dynamic-memory technique is applied to other schemes that improve on ABP-work stealing such as the locality-guided work-stealing of Blelloch [2] or the steal-half algorithm of Hendler and Shavit [6].

References

1. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. Theory of Computing Systems **34** (2001) 115–144
2. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: ACM Symposium on Parallel Algorithms and Architectures. (2000) 1–12
3. Flood, C., Detlefs, D., Shavit, N., Zhang, C.: Parallel garbage collection for shared memory multiprocessors. In: Usenix Java Virtual Machine Research and Technology Symposium (JVM '01), Monterey, CA (2001)
4. Leiserson, Plaat: Programming parallel applications in cilk. SINEWS: SIAM News **31** (1998)
5. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. Journal of the ACM **46** (1999) 720–748
6. Hendler, D., Shavit, N.: Non-blocking steal-half work queues. In: Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing. (2002)
7. Detlefs, D., Flood, C., Heller, S., Printezis, T.: Garbage-first garbage collection. Technical report, Sun Microsystems – Sun Laboratories (2004) To appear.
8. Agesen, O., Detlefs, D., Flood, C., Garthwaite, A., Martin, P., Moir, M., Shavit, N., Steele, G.: DCAS-based concurrent deques. Theory of Computing Systems **35** (2002) 349–386
9. Martin, P., Moir, M., Steele, G.: Dcas-based concurrent deques supporting bulk allocation. Technical Report TR-2002-111, Sun Microsystems Laboratories (2002)
10. Greenwald, M.B., Cheriton, D.R.: The synergy between non-blocking synchronization and operating system structure. In: 2nd Symposium on Operating Systems Design and Implementation. (1996) 123–136 Seattle, WA.
11. Blumofe, R.D., Papadopoulos, D.: The performance of work stealing in multiprogrammed environments (extended abstract). In: Measurement and Modeling of Computer Systems. (1998) 266–267
12. Arnold, J.M., Buell, D.A., Davis, E.G.: Splash 2. In: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures, ACM Press (1992) 316–322
13. Papadopoulos, D.: Hood: A user-level thread library for multiprogrammed multiprocessors. In: Master's thesis, Department of Computer Sciences, University of Texas at Austin. (1998)
14. Scott, M.L.: Personal communication: Code for a lock-free memory management pool (2003)
15. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12** (1990) 463–492
16. Blumofe, R.D., Plaxton, C.G., Ray, S.: Verification of a concurrent deque implementation. Technical Report CS-TR-99-11, University of Texas at Austin (1999)