View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Infoscience - École polytechnique fédérale de Lausanne

On Lock-Free Work-stealing Iterators for Parallel Data Structures

Aleksandar Prokopec
EPFL
Email: aleksandar.prokopec@epfl.ch

Dmitry Petrashko
EPFL
Email: dmitry.petrashko@epfl.ch

Martin Odersky
EPFL
Email: martin.odersky@epfl.ch

***Abstract*—In modern programming high-level data-structures are an important foundation for most applications. With the rise of multicores, there is a trend of supporting data-parallel collection operations in general purpose programming languages. These operations are highly parametric, incurring abstraction performance penalties. Furthermore, data-parallel operations must scale when applied to irregular workloads. Work-stealing is a proven technique for load balancing irregular workloads, but general purpose work-stealing also suffers abstraction penalties.**

We present a generic data-parallel collections design based on work-stealing for shared-memory architectures that overcomes abstraction penalties through callsite specialization of data-parallel operation instances. Moreover, we introduce *work-stealing iterators* that allow fine-grained and efficient work-stealing for particular data-structures. By eliminating abstraction penalties and making work-stealing data-structure-aware we achieve up to $60\times$ better performance compared to JVM-based approaches and $3\times$ speedups compared to tools such as Intel TBB.

I. Introduction

While the declarative nature of data-parallel programming makes programs easier to understand and maintain, as well as to apply to a plethora of different problems, implementing an efficient data-parallel framework remains a challenging task. This task is only made harder by the fact that data-parallel frameworks offer genericity on several levels. First, parallel operations are generic both in the type of the data records and the way these records are processed. Orthogonally, records are organized into data sets in different ways depending on how they are accessed – as arrays, hash-tables or trees. Let us consider the example of a subroutine that computes the mean of a set of measurements to illustrate these concepts. We show both its imperative and data-parallel variant.

```
1 def mean(x: Array<Int>) = {
2   var sum = 0
3   while (i < x.length)
4     { sum += x(i); i += 1 }
5   return sum / x.length }
```

```
1 def mean(x: Array<Int>) = {
2   val sum = x.par.fold(0) {
3     (acc, v) => acc + v
4   }
5   return sum / x.length }
```

The data-parallel operation that the declarative-style `mean` subroutine relies on is `fold`, which aggregates multiple values into a single value. This operation is parametrized by the user-specified aggregation operator. The data set is an array and the data records are the array elements, in this case integers. A naive implementation of a parallel `fold` method [14] might be as follows:

```
6 def fold<T>(x: Iterable<T>, z: T, op: (T, T) => T) = {
7   val subsets = x.iterator.split
8   val results = subsets.inParallel { subset =>
9     var sum = z
10    while (subset.hasNext) sum = op(sum, subset.next())
11    sum }
12  return results.foldLeft(z)(op) }
```

We assume collections have a method that returns an `iterator` that can be efficiently `split` into subsets [14] [15]. These subsets are processed in parallel – from a high-level perspective, this is done by the `inParallel` call. Once all the workers complete, their `results` can be aggregated sequentially. We focus on the work done by separate workers, namely, lines 9 through 11. Note that the `while` loop in those lines resembles the imperative variant of the method `mean`, with several differences. The neutral element of the aggregation `z` is generic and specified as an argument. Then, instead of comparing a local variable `i` against the array length, method `hasNext` is called, which translates to a dynamic dispatch. The second dynamic dispatch updates the state of the iterator and returns the `next` element and another dynamic dispatch is required to apply the summation operator to the integer values.

These inefficiencies are referred to as the *abstraction penalties*. We can identify several abstraction penalties in the previous example. First of all, in typical object-oriented languages such as Java or C++ the dynamic dispatches amount to reading the address of the virtual method table and then the address of the appropriate method from that table. Second, and not immediately apparent, the iterator abstraction inherently relies on maintaining the traversal continuation. The method `next` must read an integer field, check the bounds and write the new value back to memory before returning the corresponding value in the array. The imperative implementation of `mean` merely reads the array value and updates `i` in the register. The third overhead has to do with representing method parameters in a generic way. In languages like Java, Scala and OCaml primitive values passed to generic methods are converted to heap objects and their references are used instead. This is known as *boxing* and can severely impact peformance. While in languages like C++ templating can specialize the `fold` for primitive types, generic type parameters are a problem for many runtimes.

To achieve parallel speedups proper load balancing is required. In the simplified `fold` implementation we used the hypothetical `inParallel` method that assigns subsets of work to different workers. This approach of statically partitioning the workload has been studied extensively, but it does not guarantee optimal speedup in all cases [9]. Consider

the following example of naively computing a list of prime numbers smaller than N.

```
13  (3 until N) filter { i =>
14    (2 to ⌈sqrt(i)⌉) forall { d => i % d != 0 } }
```

For each of the numbers i between 3 and N the filter predicate checks if any number up to the square root of i divides i. The amount of computation for each element depends on its value, making this data-parallel computation irregular. If the numbers are specified as part of the program input, then there is no way for static analysis to optimally partition the work at compile time.

This paper focuses on runtime *workload-driven* load balancing. So far, *work-stealing* has proven an efficient runtime load balancing technique for irregular problems [3] [5], and the collections design we propose adopts it as well. It was shown that making work-stealing data-parallelism aware allows better load balancing [1]. For this reason, our design integrates work-stealing with the shape of the data-structure, allowing the size of the *batches* to adapt to the workload. As we will show, existing approaches incur *scheduling penalties* by relying only on general-purpose work-stealing and not making work-stealing data-structure-aware [14].

The goal of this paper is to twofold. First, we show how the aforementioned abstraction penalties can be eliminated in a generic way for different data-structures and data-parallel operations, achieving near optimal performance. We rely on an abstraction called a *kernel* of a data-parallel operation, which consists of the specialized code for traversing and processing a batch of data for a specific data-parallel operation instance. Second, we show how to minimize the scheduling penalties by employing fine-grained work-stealing for different data-structures in a generic, efficient and lock-free manner. We will introduce the concept of *work-stealing iterators*, which abstract over how work is divided into batches and how it is stolen. Note that knowledge of neither kernels nor work-stealing iterators is required for the data-parallel framework user, but using them allows extending the framework with new operations and data collections.

The rest of the paper is organized as follows. Section II describes work-stealing iterators and kernel abstractions for different data-structures and data-parallel operations. In Section III we evaluate the performance of data-parallel collection operations on a range of microbenchmarks and on larger benchmark applications. Section IV presents the related work.

Finally, Section V concludes.

II. Design and implementation

It is common that tasks recursively spawn subtasks in task parallel programming, potentially generating additional work to be stolen. This fact drives the design of many runtimes based on work-stealing [5] [10] – only a single, oldest task is stolen at a time, the execution of which hopefully creates more subtasks. Conversely, parallelism units in data parallel programming are not tasks but individual collection elements that do not generate more work. Thus, stealing must proceed in batches to reduce the scheduling penalty.

The work-stealing tree scheduler [1] exploits this observation by dividing the remaining workload equally between

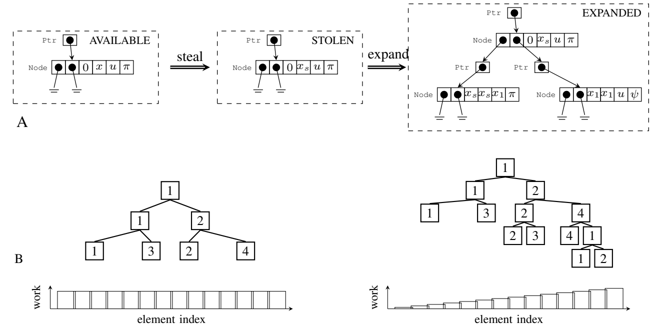

Fig. 1. (A) Stealing in the work-stealing tree scheduler; (B) Scheduling irregular and uniform workloads

the stealer and the victim when a steal occurs. This lock-free scheduling algorithm is based on the CAS instruction. CAS is a robust synchronization primitive preferable to mutexes, semaphores and critical sections due to being encoded as a single CPU instruction that does not lead to context switches. The advantages of CAS-based lock-free algorithms are well known [6], and lock-free algorithms and data-structures are still an active area of research today [12]. Lock-freedom is important for work-stealing data-parallel workloads – a stealer should not have to wait for the worker to allow stealing, as the worker may work on an unknown workload indefinitely long.

Workstealing-tree scheduler works by having each worker keep the loop iteration index and atomically increment it to inform potential stealers of its progress. The iteration index is kept in the work-stealing node structure belonging to a specific processor π. Each work-stealing node traverses a specific subset of the parallel loop. This is shown in Figure 1A in the AVAILABLE state – the 0 and the u denote the bounds of the parallel loop, and x denotes the current value of the iteration index. A stealing process ψ invalidates this index to prevent the victim from further increments and, importantly, at the same time captures the information about its progress. This is shown in the STOLEN state in Figure 1A. Subsequent updates to the iteration index are disallowed and the work-stealing node is split into two child nodes, each of which holds roughly half of the remaining elements of the original node.

This approach to scheduling data-parallel operations is particularly efficient in load-balancing irregular data-parallel operations, as well as uniform ones. Two different data-parallel workloads and the typical states of the work-stealing tree data-structure at the end of the data-parallel operation are shown in Figure 1B for illustration purposes. The uniform workload like the fold mentioned in the introduction yields a balanced work-stealing tree in which every worker processes roughly the same number of elements and works in isolation most of the time without communicating other workers. The irregular workload (Figure 1B) like the prime number computation mentioned earlier yields an unbalanced work-stealing tree in which the worker 1 processes the smaller numbers earlier than the worker 4 completes the computation on the bigger ones. Instead of remaining idle, worker 1 steals some of the expensive elements. In general, the unbalancing factor in a work-stealing subtree is proportional to the workload irregularity in the corresponding part of the parallel loop. When the irregularity is high, there is enough work per each element

to amortize the scheduling penalties of creating new work-stealing tree nodes. Conversely, when irregularity is low, work per element may be low too, but there are less nodes being created. The scheduling is thus fully adaptive and occurs at runtime – we say that it is *workload-driven*.

We omit the details of how the scheduler uses the work-stealing tree, i.e. expands it or assigns workers to specific nodes – this was already discussed in detail in related work [1]. We instead focus on the code that the workers and stealers execute. The pseudocode we show closely resembles Scala, but relies on language features available in modern general-purpose programming languages.

Lets start by showing the pseudocode for a worker executing a parallel loop. We assume that the worker is assigned a batch determined by the integers `start`≥ 0 and `until`$\geq$`start`. It also maintains a globally visible integer `progress` which it updates atomically with a `CAS`. This value denotes the first loop element within ⟨`start`, `until`⟩ that the worker is not obliged to process. We use pseudocode resembling Python, with the `def` keyword for method definitions, and the addition of `val` and `var` for single and multiple assignment variable declaration, respectively. We annotate values with types by prefixing them with a : sign. Indentation is used to delimit blocks for conciseness.

```
1 def work() =
2   var loop = true
3   var step = 0
4   while (loop)
5     step = update(step)
6     val p = READ(progress)
7     if (p ≥ until ∨ p < 0) loop = false else
8       if (CAS(progress, p, min(until, p + step)))
9         apply(p, min(until, p + step))
```

The algorithm uses a value `step` to decide how many loop elements to commit to in each iteration. Updating `step` in line 5 and its effect on scheduling was studied elsewhere [8] [13] [1], but it suffices to say that this value has to be varied to achieve the best speedup. In each loop iteration the worker reads the value of `progress` and tries to atomically increment it with a `CAS`. If it succeeds, it is committed to process all elements smaller than the last value written to `progress`. It does so by calling `apply` in line 9, which executes a user-specified operation on each element within the specified range. Section II-B shows how `apply` corresponds to a specific operation instance. The stealer invalidates the `progress` by executing the following.

```
10 def markStolen() =
11   val p = READ(progress)
12   if (p < until ∧ p ≥ 0)
13     if (¬CAS(progress, p, -p - 1)) markStolen()
```

This pushes the tree node into the STOLEN state from Figure 1A. Note that replacing the current value of `progress` with a negative value allows decoding the previous state uniquely. Also, neither the worker nor any of the stealers write to `progress` after it becomes negative. We do not show how the remaining work is split after `markStolen` completes – at this point there is sufficient information to reach a consensus on that in a lock-free way. Note that while this kind of execution of arbitrary parallel loops is not itself lock-free because a specific worker commits to processing specific elements, the work-stealing process is, as stealers proceed without the help of the victim as long as there are elements left in `progress`.

```
1 def work(it: StealIterator<T>) =
2   var step = 0
3   var res = zero
4   while (it.state() == 𝔸)
5     step = update(step)
6     val batch = it.nextBatch(step)
7     if (batch >= 0)
8       res = combine(res, apply(it, batch))
9   it.result = res
```

Fig. 2. The generalized loop scheduling algorithm

A. Work-stealing iterators

This section augments the *iterator* abstraction with the facilities that support work-stealing. The previously shown `progress` value served this purpose for parallel loops.

There are several parts of the presented work-stealing scheduler that we can generalize. We read the value of `progress` in line 6 to see if it is negative (indicating a steal) or greater than or equal to `until` (indicating that the loop is completed) in line 7. Here the value of `progress` indicates the `state` the iterator is in – either available ($\mathbb{A}$), stolen ($\mathbb{S}$) or completed ($\mathbb{C}$). In line 8 we atomically update `progress`, consequently deciding on the number of elements that can be processed. This can be abstracted away with a method `nextBatch` taking a desired number of elements to traverse and returning an estimated number of elements to be traversed, or -1 if there are none left. Figure 2 shows an updated version of the loop scheduling algorithm that relies on these methods. Iterators should also abstract the method `markStolen` shown earlier. We show the complete work-stealing iterator interface in Figure 3. The additional method `owner` returns the index of the worker owning the iterator. The method `next` can be called as long as the method `hasNext` returns `true`, just as with the ordinary iterators. Method `hasNext` returns `true` if `next` can be called before the next `nextBatch` call. Finally, the method `split` can only be called on $\mathbb{S}$ iterators and it returns a pair of iterators such that the disjoint union of their elements are the remaining elements of the original iterator. This implies that `markStolen` must internally encode the iterator state immediately when it gets stolen.

The contracts of these methods are formally expressed below. We implicitly assume termination and a specific iterator instance. Unless specified otherwise, we assume linearizability. When we say that a method M is owner-specific (π-specific), it means that every invocation by a worker π is preceeded by a call to `owner` returning π. For non-owner-specific M `owner` returns $\psi \neq \pi$.

Contract `owner`. If an invocation returns π at time t_0, then $\forall t_1 \geq t_0$ invocations return π.

Contract `state`. If an invocation returns $s \in \{\mathbb{S}, \mathbb{C}\}$ at time t_0, then all invocations at $t \geq t_0$ return s, where C and S denote completed and stolen states, respectively.

Contract `nextBatch`. If an invocation exists at some time t_0 then it is π-specific and the parameter `step` ≥ 0. If the return value c is -1 then a call to `state` at $\forall t_1 > t_0$ returns $s \in \{\mathbb{S}, \mathbb{C}\}$. Otherwise, a call to `state` at $\forall t_{-1} < t_0$ returns $s = A$, where A is the available state.

Contract `markStolen`. Any invocations at t_0 is non-owner-specific and every call to `state` at $t_1 > t_0$ returning $s \in \{\mathbb{S}, \mathbb{C}\}$.

```
10 StealIterator<T>
11   def owner(): Int
12   def state(): 𝔸 ∨ 𝕊 ∨ ℂ
13   def nextBatch(step: Int): Int
14   def markStolen(): Void
15   def hasNext: Boolean
16   def next(): T
17   def split(): (StealIterator<T>, StealIterator<T>)
```

Fig. 3. The `StealIterator` interface

Contract `next`. A non-linearizable π-specific invocation is linearized at t_1 if there is a `hasNext` invocation returning `true` at $t_0 < t_1$ and there are no `nextBatch` and `next` invocations in the interval $\langle t_0, t_1 \rangle$.

Contract `hasNext`. If a non-linearizable π-specific invocation returns `false` at t_0 then all `hasNext` invocations in $\langle t_0, t_1 \rangle$ return `false`, where there are no `nextBatch` calls in $\langle t_0, t_1 \rangle$.

Contract `split`. If an invocation returns a pair (n_1, n_2) at time t_0 then the call to `state` returned $\mathbb{S}$ at some time $t_{-1} < t_0$.

Traversal contract. Define $\overline{X} = x_1 x_2 \ldots x_m$ as the sequence of return values of `next` invocations at times $t'_1 < t'_2 < \ldots < t'_m$. If a call to `state` at $t > t'_m$ returns $\mathbb{C}$ then $e(i) = \overline{X}$. Otherwise, let an invocation of `split` on an iterator i return (i_1, i_2). Then $e(i) = \overline{X} \cdot e(i_1) \cdot e(i_2)$, where $\cdot$ is concatenation. There exists a fixed E such that $E = e(i)$ for all valid sequences of `nextBatch` and `next` invocations.

The last contract states that an iterator always traverses the same elements in the same order. Having formalized the work-stealing iterators, we show several concrete implementations.

`IndexIterator`. This is a simple iterator implementation following from refactorings in Figure 2. It is applicable to parallel ranges, arrays, vectors and data-structures where indexing is fast. The implementation for ranges in Figure 4 uses the `private` keyword for the fields `nextProgress` and `nextUntil` used in `next` and `hasNext`. Since their contracts ensure that only the owner calls them, their writes need not be globally visible. The field `progress` is marked with the keyword `atomic`, and is modified by the CAS in the line 34, ensuring that its modifications are globally visible through a memory barrier. All method contracts are straightforward to verify and follow from the linearizability of CAS. For example, if `state` returns $\mathbb{S}$ or $\mathbb{C}$ at time t_0, then the `progress` was either negative or equal to `until` at t_0. All the writes to `progress` are CAS instructions that check that `progress` is neither negative nor equal to `until`. Therefore, `progress` has the same value $\forall t > t_0$ and `state` returns the same value $\forall t > t_0$ (contract *state*).

`HashIterator`. Hash tables are a ubiquitous data structure in programming languages and in a variety of applications that rely on efficient set membership or key-based lookup operations. The implementation of work-stealing iterators for flat hash-tables we show in this section is similar to the iterators for data-structures with fast indexing. Thus, the iteration state can still be represented with a single integer field `progress`, and invalidated with `markStolen` in the same way as with `IndexIterator`. The `nextBatch` has to compute the expected number of elements between to array entries using

```
18 RangeIterator implements StealIterator<Int>
19   private var nextProgress = -1
20   private var nextUntil = -1
21   atomic var progress: Int
22   val owner: Int
23   val until: Int
24   def state() =
25     val p = READ(progress)
26     if (p ≥ until) return ℂ
27     else if (p < 0) return 𝕊
28     else return 𝔸
29   def nextBatch(s: Int): Int =
30     if (state() ≠ 𝔸) return -1
31     else
32       val p = READ(progress)
33       val np = math.min(p + s, until)
34       if (¬CAS(progress, p, np)) return nextBatch(s)
35       else
36         nextProgress = p
37         nextUntil = np
38         return np - p
39   def markStolen() =
40     val p = READ(progress)
41     if (p < until ∧ p ≥ 0)
42       if (¬CAS(progress, p, -p - 1)) markStolen()
43   def hasNext = return nextProgress < nextUntil
44   def next() =
45     nextProgress += 1
46     return nextProgress - 1
```

Fig. 4. The `IndexIterator` implementation

the load factor `lf` as follows:

```
47 def nextBatch(step: Int): Int =
48   val p = READ(progress)
49   val np = math.min(p + (step / lf).toInt, until)
50   if (¬CAS(progress, p, np)) return nextBatch(step)
51   else
52     nextProgress = p; nextUntil = np; return np - p
```

We change the `next` and `hasNext` implementations so that they traverse the range between `nextProgress` and `nextUntil` as a regular single-threaded hash-table iterator implementation. This implementation relies on the hashing function to achieve good load-balancing, which is common with hash-table operations.

`TreeIterator`. A considerable number of applications use the tree representation for their data. Text editing applications often represent text via ropes, and HTML document object model is based on an n-ary tree. Ordered sets are usually implemented as balanced search trees, and most priority queues are balanced trees under the hood. Persistent hash tables present in many functional languages are based on hash tries. Parallelizing operations on trees is thus a desirable goal.

Stealers for flat data structures were relatively simple, but efficient lock-free tree stealers are somewhat more involved. In this section we do not consider unbalanced binary trees since they cannot be efficiently parallelized – a completely unbalanced tree degrades to a linked list. We do not consider n-ary trees either, but note that n-ary tree stealers are a straightforward extension to the stealers presented in this section. We note that stealers for trees in which every node contains the size of its subtree can be implemented similarly to `IndexIterator`s presented earlier, since their iteration state can be encoded as a single integer. Finally, iteration state for trees with pointers to parent nodes can be encoded as a memory address of the current node, so their stealers are trivial.

Therefore, in this section we show a stealer for balanced binary search trees in which nodes do not have parent pointers and do not maintain size of their corresponding subtrees. In

```
53 TreeIterator<T> implements StealIterator<T>
54   private val localstack: Array<Tree>
55   private var depth: Int
56   atomic var stack: Bitset
57
58   def state() =
59     val s = READ(stack)
60     return s & 0x3
61
62   def markStolen() =
63     val s = READ(stack)
64     if (s & 0x3 = 𝔸)
65       val ns = (ns & ~0x3) | 𝕊
66       if (¬CAS(stack, s, ns)) markStolen()
67
68   def topBitset(s: Bitset) =
69     val d = 2 * depth
70     return (s & (0x3 << d)) >> d
71
72   def top() = localstack[depth - 1]
73
74   def pop(s: Bitset) =
75     depth = depth - 1
76     localstack(depth) = null
77     return s & ~(0x3 << (2 + 2 * depth))
78
79   def push(s: Bitset, v: Bitset, t: Tree)
80     localstack(depth) = t
81     depth = depth + 1
82     return s | (v << (2 * depth))
83
84   def switch(s: Bitset, v: Bitset) =
85     val d = 2 * depth
86     return (s & ~(0x3 << d)) | (v << d)
87
88   def nextBatch(step: Int): Int =
89     val s = READ(stack)
90     var ns = s
91     var batchSize = -1
92     if (s & 0x3 ≠ 𝔸) return -1 else
93       val tm = topBitset(s)
94       if (tm = 𝔹 ∨ (tm = 𝕋 ∧ top().right.isLeaf))
95         ns = pop(ns)
96         while (topBitset(ns) = ℝ ∧ depth > 0)
97           ns = pop(ns)
98         if (depth = 0) ns = (ns & ~0x3) | ℂ
99         else
100          ns = switch(ns, 𝕋)
101          batchSize = 1
102          setNextValue(top().value)
103      else if (tm = 𝕋)
104        ns = switch(ns, ℝ)
105        val n = top().right
106        while (¬r.left.isLeaf ∧ bound(depth) ≥ step)
107          ns = push(ns, 𝕃, n)
108          n = n.left
109        if (bound(depth) < step)
110          ns = push(ns, 𝔹, n)
111          batchSize = bound
112          setNextSubtree(n)
113        else
114          ns = push(ns, 𝕋, n)
115          batchSize = 1
116          setNextValue(n.value)
117    while (¬CAS(stack, s, ns))
118      val ss = READ(stack)
119      if (ss ∈ { 𝕊, ℂ }) return -1
120    return batchSize
```

Fig. 5. The `TreeIterator` data type and helper methods

this representation each node is either an inner node containing a single data `value` and pointers to the `left` and the `right` child, or it is a leaf node (`isLeaf`), in which case it does not contain data elements. AVL trees, red-black trees and binary hash tries fit precisely this description.

The trees we examine have two important properties. First, given a node T at the depth d and the total number of keys N in the entire tree, we can always compute a bound on the depth of the subtree rooted at T from d and N. Similarly, we can compute a bound on the number of keys in the subtree of T. These properties follow from the fact that the depth of a balanced tree is bounded by its number of elements. Red-black trees, for example, have the property that their depth d is less than or equal to $2 \log N$ where N is the total number of keys.

Given a tree root, the iteration state can be encoded as a `stack` of decisions $\mathbb{L}$ and $\mathbb{R}$ that denote turning left or right at a given node. The top of the stack contains a terminal symbol $\mathbb{T}$, meaning that the corresponding node is currently being traversed, or a symbol $\mathbb{B}$ indicating an entire subtree as a batch of elements that the stealer last committed to process. Examples of tree iteration states are shown in Figure 6, where a worker first traverses a node C, proceeds by traversing a single node B and then commits to the entire subtree D.

We represent `stack` with a bitset, using 2 bits to store single stack entry. The first 2 bits of this bitset have a special role – they encode one of the three stealer states $\mathbb{A}$, $\mathbb{S}$ and $\mathbb{C}$. The stealers and workers update the stealer state atomically by replacing the `stack` bitset with CAS instructions as shown in Figure 5. Stealers invoke `markStolen` that atomically changes the stealer state bits in line 66, making sure that any subsequent `nextBatch` calls fail by invalidating their next CAS in line 117, which in turn requires the state bits to be equal to $\mathbb{A}$. Workers invoke `nextBatch` that atomically changes the currently traversed node or a subtree. A worker owning a particular stealer additionally maintains the actual stack of tree nodes on the current traversal path in its local array `localstack`, whose size is bounded by the depth d of the corresponding tree. For convenience, it also maintains the current depth `depth`. Calling `nextBatch` starts by checking to see if the node is in the available state $\mathbb{A}$, and returning -1 if it is not. Part of the code between lines 93 and 116 is identical to that of a regular sequential tree iterator – it identifies the currently traversed node and replaces it, updating `localstack` and `depth` in the process. At each point it updates the tentative new state of the stack `ns` by adding and removing the symbols $\mathbb{L}$, $\mathbb{R}$, $\mathbb{B}$ and $\mathbb{T}$ using the helper `push`, `pop` and `switch` methods. Note that in line 106 the worker relies on the `bound` value at a given `depth` to estimate the number of elements in particular subtrees, and potentially decide on batching the elements. Calls to `setNextValue` and `setNextSubtree` set the next value or subtree to be traversed – they update the stealer state so that the subsequent `next` and `hasNext` calls work correctly.

Once all the updates of the worker-local state are done, the worker attempts to atomically change the state of the `stack` with the new value `ns` in line 117, failing only if a concurrent steal has occurred, in which case it returns -1.

We do not show the pseudocode for splitting the stealer after it has been marked stolen, but show the important classes of different states the stealer can be in, in Figure 7. We encode the state of the stack with a regular expression of stack symbols – for example, the expression $\mathbb{R}^*\mathbb{L}\mathbb{T}$ means that the stack contains a sequence of right turns followed by a single left turn and then the decision $\mathbb{T}$ to traverse a single node. A stealer in such a state should be split into two stealers with states $\mathbb{L}^*$ and $\mathbb{R}^*\mathbb{L}\mathbb{B}$, as shown in Figure 7. The remaining states $\mathbb{R}^*\mathbb{B}$ and $\mathbb{R}^*\mathbb{T}$ are omitted for brevity.

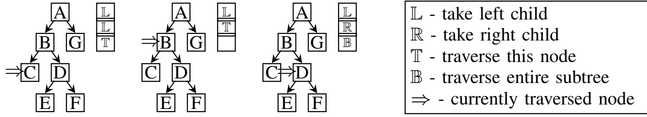

Fig. 6. `TreeIterator` state diagram

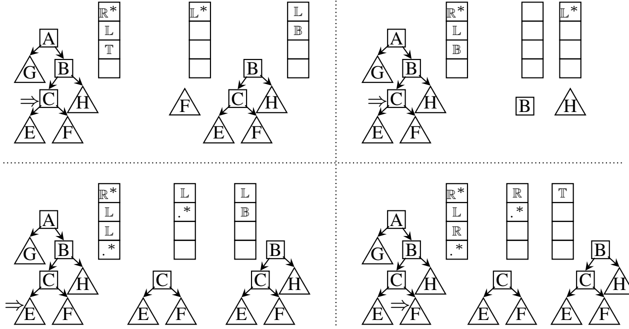

Fig. 7. `TreeIterator` splitting rules

B. Kernel callsite specialization

We saw in Figure 2 that the worker uses the work-stealing iterator to commit to processing batches of elements. The `apply` call in line 8 conceals the details of how elements are processed. In this section we show that the `apply` implementation depends on a specific data-parallel operation instance. We focus our attention on the previously mentioned *kernel* abstraction.

Each data-parallel operation invocation site creates a kernel object, which describes how a batch of elements is processed and what the resulting value is, how to combine values computed by different workers and what the neutral element for the result is. The kernel interface is shown in Figure 8. The method `apply` takes the iterator and the number of elements estimate returned by `nextBatch`. It uses the iterator to traverse those elements and compute the result of type `R`. The method `combine` is used to merge two different results and `zero` returns the neutral element.
How these methods work is best shown through an example of a concrete data-parallel operation. The `foreach` operation takes a user-specified function object `f` and applies it in parallel to every element of the collection. Assume we have a collection `xs` of integers and we want to assert that each integer is positive:

```
125 xs.foreach(x => assert(x > 0))
```

The generic `foreach` implementation is as follows:

```
126 def foreach(f: Int => Void) =
127   val k = new Kernel<Int, Void>
128     def zero =
129     def combine(a: Void, b: Void): Void =
130     def apply(it: StealIterator<T>, batch: Int) =
131       while (it.hasNext) f(it.next())
132   invokeParallel(k)
```

The `Void` type indicates no return value – the `foreach` function is executed merely for its side-effect, in this case a potential assertion. Methods `zero` and `combine` always return the `Void` value `()` for this reason. Most of the processing time is spent in the `apply` method, so its efficiency drives the running time of the operation. We use

```
121 Kernel<T, R>
122   def zero: R
123   def combine(a: R, b: R): R
124   def apply(it: StealIterator<T>, batch: Int): R
```

Fig. 8. The `Kernel` interface

the Scala Macro system [2] to inline the body of the function `f` into the `Kernel` at the callsite:

```
133 def apply(it: StealIterator<T>, batch: Int) =
134   while (it.hasNext) assert(it.next() > 0)
```

Another example is the `fold` operation from the introduction and computing the sum of a sequence of numbers `xs`:

```
135 xs.fold(0)((acc, x) => acc + x)
```

In general, whenever we have a function literal directly applied to its arguments, we inline it:

```
inline[(x => body)(v)] ⇒ inline[body[x := v]]
inline[x => body] ⇒ x => inline[body]
inline[v] ⇒ v
```

Operation `fold` computes a resulting value, which has the integer type in this case. Results computed by different workers have to be added together using `combine` before returning the final result. After inlining the code for the neutral element and the body of the folding operator, we obtain the following kernel:

```
136 new Kernel<Int, Int>
137   def zero = return 0
138   def combine(a: Int, b: Int) = return a + b
139   def apply(it: StealIterator<T>, batch: Int) =
140     var sum = 0
141     while (it.hasNext) sum = sum + it.next()
142     return sum
```

Where `fold` returns a scalar value, some operations like `map`, `flatMap` and `filter` return collections. They use data-structure-specific *combiners* [14] to build these resulting collections. Combiners define methods += for adding elements and `combine` to efficiently merge the elements of two combiners into a new one.
While the inlining shown in the previous examples avoids a dynamic dispatch to a function object, the `while` loop still contains two virtual calls to the work-stealing iterator. Maintaining the iterator requires writes to memory instead of registers. It also prevents optimizations like loop-invariant code motion, e.g. hoisting the array bounds check necessary when the iterator traverses an array. For these reasons, we would like to inline the iteration into the `apply` method itself. This, however, requires knowing the specifics of the data layout in the underlying data-structure. Within this paper we rely on the macro system to apply these transformations at compile-time – we will require that the collection type is known statically to eliminate the `next` and `hasNext` calls.

`IndexKernel`. Data-structures with fast indexing such as arrays and ranges can be traversed efficiently by using a local variable `p` as iteration index. Figure 9 shows range and array kernel implementations for the `fold` example discussed earlier. Array bounds checks inside a `while` loop are visible to the compiler or a runtime like the JVM and can be hoisted out. On platforms like the JVM potential boxing of primitive objects resulting from typical functional object abstractions is eliminated. Finally, the dynamic dispatch is eliminated from the loop. The loop thus obtained has optimal performance as shown in the evaluation in Section III.

`TreeKernel`. The work-stealing iterator for trees introduced in Section II-A assumed that any subtree can be

```
143 def apply(
144   i: RangeIterator,
145   batch: Int) =
146   var sum = 0
147   var p = i.nextProgress
148   val u = i.nextUntil
149   while (p < u)
150     sum = sum + p
151     p += 1
152   return sum
```

```
153 def apply(
154   i: ArrayIterator<T>,
155   batch: Int) =
156   var sum = 0
157   var p = i.nextProgress
158   val u = i.nextUntil
159   while (p < u)
160     sum = sum + array(p)
161     p += 1
162   return sum
```

Fig. 9. Specialized indexed kernel `apply` methods for the `fold` operation

```
163 def apply(i: TreeIterator<T>, batch: Int) =
164   def traverse(t: Tree): Int =
165     if (t.isLeaf) t.element
166     else traverse(t.left) + traverse(t.right)
167   val root = i.nextStack(0)
168   return traverse(root)
```

Fig. 10. Specialized `Tree` kernel `apply` method for the `fold` operation

traversed with the `next` and `hasNext` calls by using a private stack, much like the linearizable `nextBatch` that relies on an atomic stack. Pushing and popping on this private stack can be avoided by traversing the subtree directly. Figure 10 shows a kernel in which the `root` of the subtree is traversed with a nested recursive method `traverse`. In Section III we show that this kind of traversal improves running time several times when compared with the iterator approach.

HashKernel. The hash-table kernel is based on an efficient `while` loop like the array and range kernels, but must account for empty array entries. Assuming flat hash-tables with linear collision resolution, the `while` loop in the kernel implementation of the previously mentioned `fold` is as follows:

```
169 while (p < u)
170   val elem = array(p)
171   if (elem ≠ null) sum = sum + elem
172   p += 1
```

Work-stealing iterator implementations for hash-tables based on closed addressing are similar.

III. Performance evaluation

The goals of our design were to reduce abstraction and scheduling penalties to negligible levels. In this section we present a performance improvement breakdown that validates these goals by identifying each of the penalties separately and showing that they are overcome. We then introduce a range of different workloads to evaluate our load-balancing approach. We compare against imperative sequential programs written in Java, against existing Scala Parallel Collections, a corresponding imperative C version and the Intel TBB library wherever a comparison is feasible. In the first part we show microbenchmarks addressing specific abstraction penalties. We proceed by introducing a range of irregular workloads to assess scheduling efficiency, and conclude by showing larger data-parallel applications.

We perform the evaluation on the Intel i7-3930K hex-core 3.4 GHz processor with hyperthreading and an 8-core 1.2 GHz UltraSPARC T2 with 64 hardware threads. Aside from the different number of cores and processor clock, another important difference between them is in the memory throughput - i7 has a single dual-channel memory controller, while the UltraSPARC T2 has four dual-channel memory controllers.

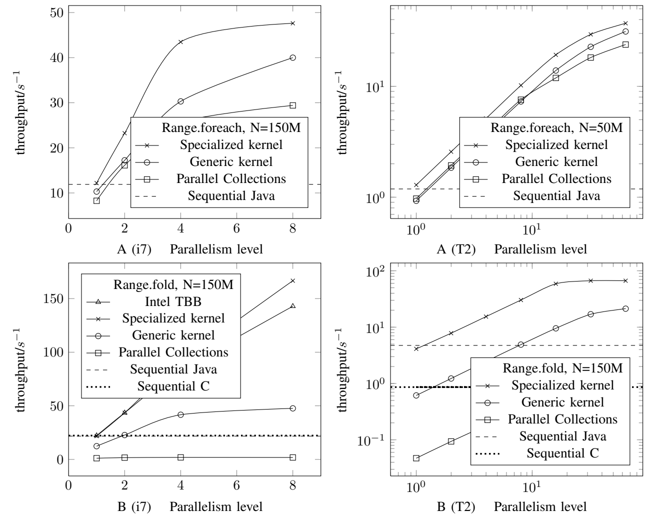

Fig. 11. Uniform workload microbenchmarks I on Intel i7 and UltraSPARC T2; A - `ParRange.foreach`, B - `ParRange.fold`

A. Abstraction penalties

The microbenchmarks in Figures 11 and 12 have a minimum cost uniform workload – the amount of computation per each element is fixed and the least possible. Those tests are targeted at detecting abstraction penalties discussed earlier. The microbenchmark in Figure 11A consists of a data-parallel `foreach` loop that occasionally sets a volatile flag (without a potential side-effect the compiler may optimize away the loop in the kernel).

```
173 par_for (i <- 0 until N)
174   if ((i * i) & 0xffffff == 0) flag = true
```

Figure 11A shows a comparison between Parallel Collections, a generic work-stealing kernel and a work-stealing kernel specialized for ranges from Figure 9. In this benchmark Parallel Collections do not instantiate primitive types and hence do not incur the costs of boxing, but still suffer from iterator and function object abstraction penalties. Inlining the function object into the `while` loop for the generic kernel shows a considerable performance gain. Furthermore, the range-specialized kernel outperforms the generic kernel by 25% on the i7 and 15% on the UltraSPARC (note the log scale).

Figure 11B shows the same comparison for parallel ranges and the `fold` operation shown in the introduction:

```
175 (0 until N).par.fold(_ + _)
```

Scala Parallel Collections abstract over the data type in this benchmark, which leads to boxing. The speed gain for a range-specialized work-stealing kernel is $20\times$ to $60\times$ compared to Parallel Collections and $2.5\times$ compared to the generic kernel. Figure 12C shows the same `fold` microbenchmark applied to parallel arrays. While Parallel Collections again incur the costs of boxing, the generic and specialized kernel have a much more comparable performance here. Furthermore, due to the low amount of computation per element, this microbenchmark spends a considerable percentage of time fetching the data from the main memory. This is particularly noticeable on the i7 – its dual-channel memory architecture becomes a bottleneck in this microbenchmark, limiting the potential speedup to $2\times$.

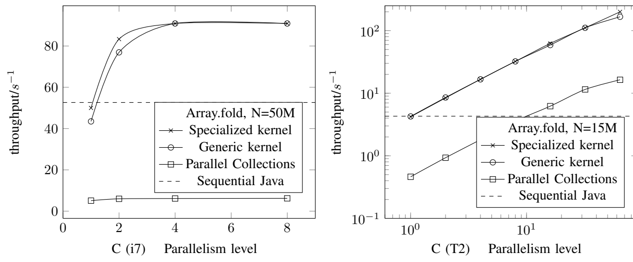

Fig. 12. Uniform workload microbenchmarks II on Intel i7 and UltraSPARC T2; C - `ParArray.fold`, D - `Tree.fold`

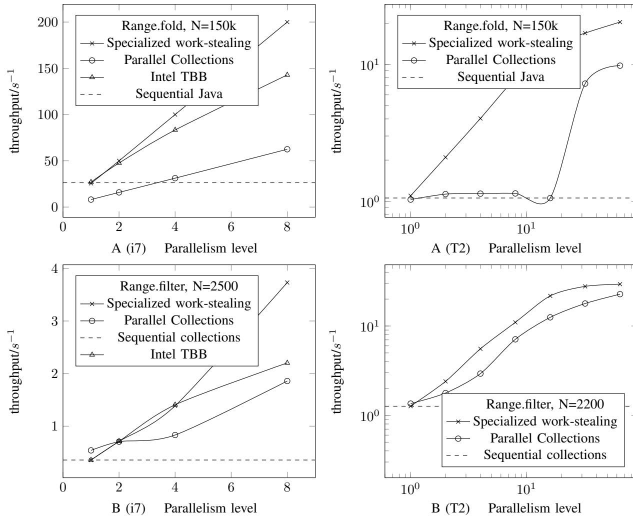

Fig. 13. Irregular workload microbenchmarks on Intel i7 and UltraSPARC T2; A - *step* workload, B - *exponential* workload

UltraSPARC, on the other hand, shows a much better scaling here due to its eight-channel memory architecture and a lower computational throughput.

The performance of the `fold` operation on balanced binary trees is shown in Figure 12D. Here we compare the generic and specialized `fold` kernels against a manually written recursive traversal subroutine. In the same benchmark we compare against the `fold` on functional lists from the Scala standard library, used in sequential functional programming. While the memory-bandwidth is the bottleneck on the i7, we witness linear scaling on the UltraSPARC. The performance difference between the generic and the specialized kernel is $2\times$ to $3\times$.

The linear scaling with respect to the sequential baseline of specialized kernels in Figures 11 and 12 indicates that our approach has negligible abstraction penalties. Having shown that abstraction penalties were eliminated, we turn to irregular workloads to compare our scheduling approach against the existing Scala Parallel Collections, as well as pure Java and C implementations that provide a baseline.

B. Scheduling penalties

Figure 13 shows improved performance compared to Parallel Collections not only due to overcoming abstraction penalties, but also due to improved scheduling. The Parallel Collections rely on a `Splitter` abstraction that divides an iterator into subsets *before* the parallel traversal begins. Their

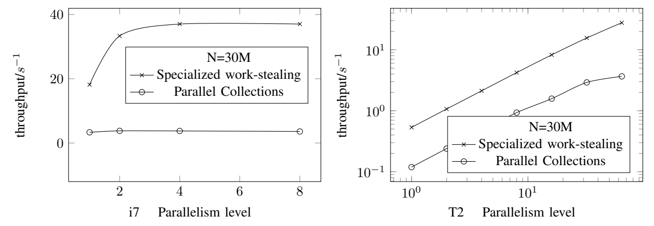

Fig. 14. Standard deviation on Intel i7 and UltraSPARC T2

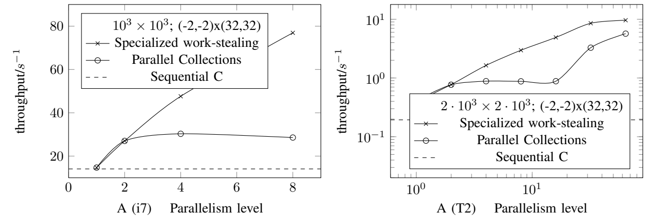

Fig. 15. Mandelbrot set computation on Intel i7 and UltraSPARC T2

scheduler chooses a batching schedule such that the batch sizes increase exponentially [14]. This scheduler is adaptive – when a worker steals a batch it divides it again. However, due to scheduling penalties of creating splitters and task objects, and then submitting them to a thread pool, this subdivision only proceeds up to a fixed threshold $\frac{N}{8P}$, where N is the number of elements and P is the number of processors. Concentrating the workload in a sequence of elements smaller than the threshold yields a suboptimal speedup. Work-stealing iterators allow smaller batches and potentially single element granularity.

In Figure 13A we run a parallel `fold` method on a *step* workload $\chi(0.97, \frac{n}{N})$ – the first 97% of elements have little associated with them, while the rest of the elements require a high amount of computation. Intel TBB exhibits a sublinear scaling in this microbenchmark, being about 25% slower compared to the work-stealing tree scheduling. Due to a predetermined work scheduling scheme where the minimum allowed batch size depends on the number of threads existing Parallel Collections scheduler [14] only yields a speedup on UltraSPARC with more than 16 threads.

As shown in Figure 13B, Intel TBB is up to $2\times$ slower compared to work-stealing tree scheduling for an *exponential* workload where the amount of work assigned to the n-th element grows with the function $2^{\frac{n}{100}}$, while the existing Scala Parallel Collections do not cope with it well. In Figure 14 we show performance results for an application computing a standard deviation of a set of measurements. The relevant part of it is as follows:

```
176 val mean = measurements.sum / measurements.size
177 val variance = measurements.aggregate(0.0)(_ + _) {
178   (acc, x) => acc + (x - mean) * (x - mean) }
```

As in the previous experiments, Parallel Collections scale but have a large constant penalty due to boxing. We show several larger benchmark applications as well. We start by showing an application that renders an image of the Mandelbrot set in parallel. The Mandelbrot set is irregular in the sense that points outside the circle $x^2 + y^2 = 4$ are not in the set, but all the points within it require some amount of computation to determine their set membership. An image containing the Mandelbrot set thus represents an irregular workload.

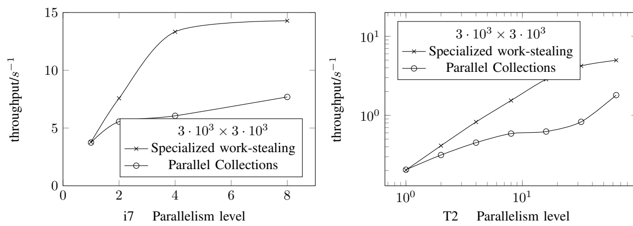

Fig. 16. Raytracing on Intel i7 and UltraSPARC T2

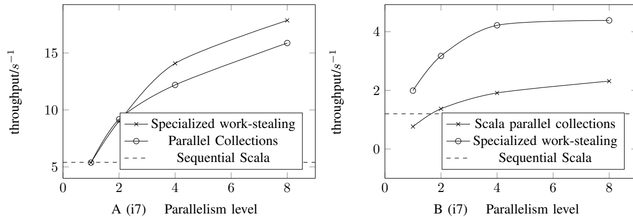

Fig. 17. (A) Barnes-Hut simulation; (B) PageRank on Intel i7

We show the running times of rendering a Mandelbrot set in Figure 15. The aforementioned computationally demanding circle is in the lower left part of the image. We can see a similar effect as in the Figure 13A – with a fixed threshold there is only a 50% to $2\times$ speedup until P exceeds 16. In Figure 16 we show the performance of a parallel raytracer, implemented using existing Parallel Collections and work-stealing tree scheduling. Raytracing renderers project a ray from each pixel of the image being rendered, and compute the intersection between the ray and the objects in the scene. The ray is then reflected several times up until a certain threshold. This application is inherently data-parallel – computation can proceed independently for different pixels. The workload characteristics depend on the placement of the objects in the scene. If the objects are distributed uniformly throughout the scene, the workload will be uniform. The particular scene we choose contains a large number of objects concentrated in one part of the image, making the workload highly irregular.

The fixed threshold on the batch sizes causes the region of the image containing most of the objects to end up in a single batch, thus eliminating most of the potential parallelism. On the i7 Parallel Collections barely manage to achieve the speedup of $2\times$, while the data structure aware work-stealing easily achieves up to $4\times$ speedups. For higher parallelism levels the batch size becomes small enough to divide the computationally expensive part of the image between processors, so the plateau ends at $P = 32$ on UltraSPARC. The speedup gap still exists at $P = 64$ – existing Parallel Collections scheduler is $3\times$ slower than the work-stealer tree scheduler. We have parallelized the Barnes-Hut n-body simulation algorithm. This simulation starts by finding the bounding box of all the particles, and then dividing them into a fixed number of rectangular sectors within that bounding box, both in parallel. Quadtrees are then constructed in parallel for the particles within each sector and merged into a singular quadtree. Finally, the positions and speeds of all the particles are updated in parallel using the quad tree to approximate the net force from the distant particles. We simulated the movement of two stellar bodies composed of $25k$ stars. We recorded the average simulation step length across a number of simulation iterations.

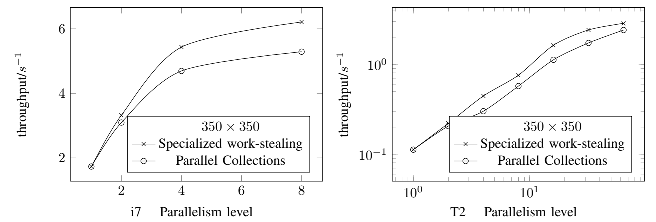

Fig. 18. Triangular matrix multiplication on Intel i7 and UltraSPARC T2

Although this turned out to be a relatively uniformly distributed workload with most of the stars situated in the sectors in the middle of the scene, we still observed a consistent 10% increase in speed with respect to preemptive scheduling in Scala Parallel Collections, as shown in Figure 17A.

The PageRank benchmark in Figure 17B shows how fusing the operations such as `map`, `reduce`, `groupBy` and `aggregate` leads to significant speedups. The single-threaded work-stealing tree version is already 35% faster compared to the standard sequential Scala collections.

The last application we choose is triangular matrix multiplication, in which a triangular $N \times N$ matrix is multiplied with a vector of size N. Both the matrix and the vector contain arbitrary precision values. This application has a less irregular workload shown in Figure 1B – the amount of work to compute the n-th element in the resulting vector is $w(n) = n$. We call this workload triangular. Figure 18 shows a comparison of the existing Parallel Collections scheduler and data structure aware work-stealing. The performance gap is smaller but still exists, work-stealing tree being 18% faster on the i7 and 20% faster on the UltraSPARC. The downsides of fixed size threshold and preemptive batching are thus noticeable even for less irregular workloads, although less pronounced.

IV. Related work

Data-parallelism is a well-established concept in parallel programming languages dating back to APL in 1962, subsequently adopted by many languages and frameworks.

The fixed-size batching [8] was an early technique that allowed a more fine-grained load-balancing for scheduling data-parallel loops. It divides the loop into batches and workers synchronize to obtain them from a central queue. This technique fails to load balance irregular workloads well. Other variable size batching approaches were proposed like *guided self-scheduling* [13], *factoring* [7] and *TSS* [17], but their static partitioning decisions have proven detrimental.

Work-stealing is a load balancing technique first used in the Cilk programming language [5] to support task parallelism. In work-stealing each worker maintains its own work queue and steals work from other workers when its queue is empty. Work-stealing is works well in problems with irregular workloads [3]. It was traditionally used as the load balancing technique for task parallelism [10] [11], but can be applied to data parallelism too [1] [16]. *Work-stealing tree scheduling* [1] is a load balancing technique in which work is kept in a tree rather than a work queue. Each node in the tree contains a subset of the data-parallel loop and is owned by a single worker. A stealer notifies the owner of the desired leaf node that the node

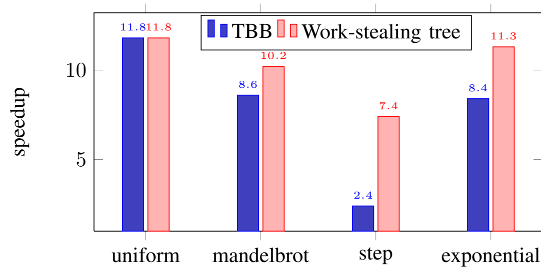

Fig. 19. Comparison of Work-stealing tree and Intel TBB

is invalidated and replaces it with two leaf nodes, dividing the remaining work. Due to a work-stealing mechanism closely tied to data-parallel loops and its tendency to keep the worker in isolation as long as possible this technique can efficiently schedule highly irregular workloads that traditional approaches [8] [13] [14] cannot cope with.

Intel TBB is a data-parallel programming C++ library based on work-stealing. TBB previously used a partitioner that required tuning by manually finding a split threshold optimal for a specific combination of a workload and a machine. Recently they have introduced the auto-partitioner that does not require this. In auto-partitioner mode every thread splits tasks in its own work queue when it detects stealing. The largest difference with respect to our approach is that the TBB auto-partitioner only allows the worker to split the work, whereas in our approach stealers are also allowed to split in a lock-free manner. Intel TBB relies on C++ templates to generate highly efficient code in a similar way that we use Scala Macros. Intel TBB also aims to be a general purpose data-parallel library and is currently an industry standard.

Figure 19 shows the comparison of our library with Intel TBB on a 6-core Intel Core i7-3930K with hyperthreading, focusing on irregular workloads such as Mandelbrot set computation, *step* and *exponential*, discussed in section III. These tests are particularly challenging to data-parallel schedulers.

In the context of the JVM compilation techniques were proposed to eliminate boxing selectively, like the *generic type specialization* transformation used in Scala [4]. While generic type specialization can be used to eliminate boxing, it does not eliminate other abstraction penalties. For this reason we rely on the Scala macro system [2], but note that our technique can be applied to languages with a templating mechanism like C++ or as a separate preprocessing step.

V. Conclusion

Whereas in traditional work-stealing basic units of parallelism are parallel function calls, this article proposes a rich set of specialized representations taking advantage of data-structure specifics to allow more efficient scheduling. The key idea is that, on one hand, such specialized parallel work representations can be processed serially with overheads that nearly match those of the sequential elision and, orthogonally, those representations allow more fine-grained work-stealing than traditional methods. This work-stealing proceeds in a completely lock-free manner allowing the idle worker threads to steal work from busy workers without waiting for them to participate in the stealing.

Comparing against already existing data-parallel frameworks shows that this approach eliminates abstraction penalties in that it parallelizes baseline workloads optimally, while overcoming scheduling penalties and achieving superior performance on highly irregular workloads. These results go a long way in showing that overcoming these penalties should and can be done automatically by the framework without the manual tuning by the data-parallel library user.

References

[1] (authors blinded). Near optimal work-stealing tree scheduler for highly irregular data-parallel workloads. (year blinded). LCPC 2013: https://dl.dropboxusercontent.com/u/4217817/submission/scheduler2.pdf.

[2] E. Burmako and M. Odersky. Scala Macros, a Technical Report. In *Third International Valentin Turchin Workshop on Metacomputation*, 2012.

[3] G. Cong, S. B. Kodali, S. Krishnamoorthy, D. Lea, V. A. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP*, pages 536–545, 2008.

[4] I. Dragos and M. Odersky. Compiling generics through user-directed type specialization. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 42–47, New York, NY, USA, 2009. ACM.

[5] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[6] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.

[7] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: a method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101, Aug. 1992.

[8] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Trans. Softw. Eng.*, 11(10):1001–1016, Oct. 1985.

[9] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 3–14, New York, NY, USA, 2009. ACM.

[10] D. Lea. A java fork/join framework. In *Java Grande*, pages 36–43, 2000.

[11] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. Steal tree: Low-overhead tracing of work stealing schedulers. In *Proceedings of the 2013 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, 2013.

[12] M. Moir and N. Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications, D. Metha and S. Sahni Editors*, pages 47–14 — 47–30, 2007. Chapman and Hall/CRC Press.

[13] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, Dec. 1987.

[14] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A generic parallel collection framework. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 136–147, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[16] O. Tardieu, H. Wang, and H. Lin. A work-stealing scheduler for X10's task parallelism with suspension. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 267–276, New York, NY, USA, 2012. ACM.

[17] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Trans. Parallel Distrib. Syst.*, 4(1):87–98, Jan. 1993.