Efficient Lock-Free Work-stealing Iterators for Data-Parallel Collections

Aleksandar Prokopec, Dmitry Petrashko, Martin Odersky
LAMP, École Polytechnique Fédérale de Lausanne, Switzerland
aleksandar.prokopec@gmail.com, dmitry.petrashko@epfl.ch, martin.odersky@epfl.ch

***Abstract*—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. However, these operations often incur high-level abstraction and scheduling 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 more fine-grained and efficient work-stealing. By eliminating abstraction penalties and making work-stealing data-structure-aware we achieve several dozen times better performance compared to existing JVM-based approaches.

***Keywords*-data parallelism; work-stealing collections; callsite specialization; domain-specific work-stealing**

I. Introduction

While the declarative nature of data-parallel programming makes programs easier to understand and maintain, implementing an efficient data-parallel framework remains challenging. This task is made hard 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 in the way they 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 an example of a subroutine that computes the mean of a set of measurements. 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   sum / x.length }
```

```
6 def mean(x: Array[Int]) = {
7   val sum = x.par.fold(0) {
8     (acc, v) => acc + v
9   }
10  sum / x.length }
```

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

```
11 def fold[T](xs: Iterable[T], z: T,
12     op: (T, T) => T) = {
13   var it = xs.iterator, sum = z
14   while (it.hasNext) sum = op(sum, it.next())
15   sum } }
```

We focus on the lines 13 through 15. 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. 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 summation to the integer values. In languages like Java or C++ the dynamic dispatch amounts to reading the address of the virtual method table and then the address of the appropriate method from that table. Moreover, 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, where the imperative implementation just reads the array value and updates `i` in the register. Another overhead is related to generic method parameters. In languages like Java, Scala and OCaml primitive values passed to generic methods are *boxed* into heap objects. We refer to all inefficiencies above as the *abstraction penalties*.

Irrespective of eliminating abstraction penalties, to achieve parallel speedups proper load balancing is required. So far, *work-stealing* has proven an efficient runtime load balancing technique for irregular problems, and the collections design we propose adopts it. 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 [7].

The goal of this paper is to twofold. First, we show how the aforementioned abstraction penalties can be eliminated 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. Neither kernels nor work-stealing iterators are required knowledge for the data-parallel framework user, but they allow extending the framework with new operations and collections. Finally, these goals are not orthogonal, as eliminating abstraction penalties is necessary to realistically assess the scheduling quality – high abstraction penalties

could entirely mask scheduling penalties.

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 benchmarks. Section IV presents the related work. Finally, Section V concludes.

II. Design and Implementation

Tasks often recursively spawn subtasks in task parallel programming, potentially generating additional work to be stolen. This fact drives the design of many work-stealing based runtimes [4] – only a single task is stolen, 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, so stealing must proceed in batches to reduce the scheduling penalty. The *work-stealing tree scheduler* [8] exploits this observation by dividing the workload between the stealer and the victim when a steal occurs. This lock-free scheduling algorithm is based on the CAS (compare-and-swap) instructions. Advantages of CAS-based lock-free algorithms are well known and they are still an active area of research. Absence of locks is important for work-stealing data-parallel workloads as well – a stealer should not wait for the worker to allow stealing, as the worker could work on an unknown workload indefinitely long.

In workstealing-tree scheduling, each worker keeps the loop iteration index and updates it to inform potential stealers of its progress. The iteration index is kept in the work-stealing node owned by a specific processor. A stealer invalidates this index atomically, in a lock-free manner, to prevent the victim from further increments. Subsequent updates to the index are disallowed and the work-stealing node is split into two child nodes, each of which holds half of the yet non-traversed elements.

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 [8]. We examine a worker executing a parallel loop. The worker is assigned a batch determined by `start`≥ 0 and `until`$\geq$`start`. It also maintains a globally visible `progress` field 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. The code we show is in Scala, but relies on language features available in modern general-purpose programming languages.

```
1 def work() = {
2   var loop = true
3   while (loop) {
4     val p = READ(progress)
5     if (p >= until || p < 0) loop = false else
6       if (CAS(progress, p, min(until, p + step)))
7         apply(p, min(until, p + step)) } }
```

The algorithm uses a value `step` to decide how many loop elements to commit to in each iteration. Choosing the `step` value and its effect on scheduling was studied elsewhere [8] [3] [5], 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 7, 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.

```
8 def markStolen() = {
9   val p = READ(progress)
10  if (p < until && p >= 0)
11    if (!CAS(progress, p, -p - 1)) markStolen() }
```

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.

The method above is limited to parallel integer ranges. We therefore introduce work-stealing iterators that allow work-stealing tree scheduling on arbitrary data-structures.

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 ranges.

There are several parts of the presented work-stealing scheduler that we can generalize. We read the value of `progress` in line 4 to see if it is negative (indicating a steal) or greater than or equal to `until` (indicating that the loop is completed) in line 5. 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 6 we atomically update `progress`, consequently deciding on the number of elements that can be processed. This can be abstracted away with a method `nextBatch` that takes the desired batch size and returns an estimated batch size, or -1 if there are no elements left. We show an updated version of the loop scheduling algorithm that relies on these methods:

```
1 def work(it: StealIterator[T]) = {
2   var step = 0; var res = zero
3   while (it.state() == A) {
4     val batch = it.nextBatch(step)
5     if (batch >= 0)
6       res = combine(res, apply(it))  }
7   res }
```

The complete work-stealing iterator interface is shown below. 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 having to call `nextBatch` again. Finally, the method `split` can only be called on stolen iterators and returns a pair of iterators that traverse the remaining elements of the original.

```
8 trait StealIterator[T] {
9   def owner(): Int
10  def state(): 𝔸 ∨ 𝕊 ∨ ℂ
11  def markStolen(): Unit
12  def split(): (StealIterator[T], StealIterator[T])
13  def nextBatch(step: Int): Int
14  def hasNext: Boolean
15  def next(): T }
```

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

Contract `owner`. There exists a time t_0, such that all invocations at $\forall t_1 \geq t_0$ 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 $\mathbb{S}$ and $\mathbb{C}$ denote stolen and completed states, respectively.

Contract `nextBatch`. A call at some time t_0 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 $s = A$.

Contract `markStolen`. A call at t_0 is non-owner-specific and calls to `state` at $t_1 > t_0$ returns $s \in \{\mathbb{S}, \mathbb{C}\}$.

Contract `next`. A non-linearizable π-specific invocation exists at time 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 call returns `false` at time t_0 then $\forall t_1 > t_0$ `hasNext` returns `false`, and there are no `nextBatch` calls in $\langle t_0, t_1 \rangle$.

Contract `split`. If a call 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` calls at times $t'_1 < t'_2 < \ldots < t'_m$. If a call to `state` at $t > t'_m$ returns $\mathbb{C}$ then the sequence $e(i)$ traversed by an iterator i is $e(i) = \overline{X}$. Otherwise, if a `split` call on an iterator i returns (i_1, i_2), the $e(i) = \overline{X} \cdot e(i_1) \cdot e(i_2)$, where $\cdot$ is concatenation. The value $e(i)$ is unique for valid sequences of `nextBatch` and `next` calls. Less formally, for any combination of `split` calls, i traverses the same sequence of elements $e(i)$.

`IndexIterator`. This iterator is applicable to parallel ranges, arrays, vectors and data-structures where indexing is fast. The range implementation uses the fields `nextProgress` and `nextUntil`. Since their contracts ensure that only the owner calls `next` and `hasNext`, their writes need not be globally visible.

```
16 trait RangeIterator extends StealIterator[Int] {
17   val owner: Int; val until: Int
18   var nextProgress; var nextUntil = -1
19   @volatile var progress: Int
20   def state() = {
21     val p = READ(progress)
22     if (p ≥ until) ℂ else if (p < 0) 𝕊 else 𝔸 }
23   def nextBatch(s: Int): Int =
24     if (state() ≠ 𝔸) -1 else {
25       val p = READ(progress)
26       val np = math.min(p + s, until)
27       if (¬CAS(progress, p, np))
28         nextBatch(s)
29       } else {
30         nextProgress = p
31         nextUntil = np
32         np - p } }
33   def markStolen() = {
34     val p = READ(progress)
35     if (p < until ∧ p ≥ 0)
36       if (!CAS(progress, p, -p - 1))
37         markStolen() }
38   def hasNext = nextProgress < nextUntil
39   def next() = {
40     nextProgress += 1
41     nextProgress - 1 } }
```

`HashIterator`. The implementation of work-stealing iterators for hash-tables is similar to indexed iterator – state can be represented with a single integer field `progress`, and invalidated with `markStolen`. The `nextBatch` has to compute the expected number of elements between using the load factor `lf` as follows:

```
42 def nextBatch(step: Int): Int = {
43   val p = READ(progress)
44   val np = math.min(p + (step / lf).toInt, until)
45   if (¬CAS(progress, p, np)) nextBatch(step)
46   else {
47     nextProgress = p
48     nextUntil = np
49     np - p } }
```

`TreeIterator`. For the presentation of tree work-stealing iterators, we refer the readers to related work [6].

B. Kernel Callsite Generation

The worker uses the work-stealing iterator to commit to processing batches of elements. The `apply` call in line 6 conceals the details of how elements are processed. In this section we describe how the `apply` implementation is generated. We use Scala Macros [1] to manipulate ASTs at the parallel operation callsites. This allows us to inline generic components of the data-parallel operation and choose a more efficient operation implementation based on the collection type. Each data-parallel operation callsite defines a kernel object that describes how a batch is processed and what the resulting value is, how to combine values from different workers and what the neutral element is. The kernel interface is as follows:

```
50 trait Kernel[T, R] {
51   def zero: R
52   def combine(a: R, b: R): R
53   def apply(it: StealIterator[T]): R }
```

The method `apply` uses the iterator to traverse the elements and compute the result of type `R`. The method `combine` is used to merge two different results and `zero` returns the neutral element. The `fold` operation from

```
72 def apply(i: ArrayIterator[T]) = {
73   var sum = 0
74   var p = i.nextProgress
75   val u = i.nextUntil
76   while (p < u) {
77     sum = sum + i.array(p)
78     p += 1 }
79   return sum }
80 def apply(i: TreeIterator[T]) = {
81   def trav(t: Tree): Int = {
82     if (t.isLeaf) t.elem
83     else trav(t.left) + t.elem + trav(t.right) }
84   val root = i.getRoot
85   trav(root) }
```

Figure 1. Specialized kernel `apply` methods for the `fold` operation

the introduction that computes the sum of a sequence of numbers `xs`:

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

has the following generic implementation (`invokeParallel` creates a work-stealing tree and notifies the worker threads):

```
55 def fold(i: StealIterator[Int],
56   z: Int, op: (Int, Int) => Int) = {
57   val k = new Kernel[Int, Int] {
58     def zero = z
59     def combine(a: Int, b: Int): Int = op(a, b)
60     def apply(i: StealIterator[T]) = {
61       var sum = zero
62       while (i.hasNext) sum = op(sum, i.next())
63       sum } }
64   invokeParallel(i, k) }
```

The macro inlines the body of the folding operator, obtaining the following kernel:

```
65 new Kernel[Int, Int] {
66   def zero = z
67   def combine(a: Int, b: Int) = a + b
68   def apply(i: StealIterator[Int]) = {
69     var sum = 0
70     while (i.hasNext) sum = sum + i.next()
71     sum } }
```

While the inlining in this example avoids the function object, the `while` loop still contains the work-stealing iterator. Using the iterator prevents optimizations like loop-invariant code motion. Inlining the iterator requires statically knowing the underlying data structure and cannot be performed by existing specialization techniques [2].

`IndexKernel`. Figure 1 shows the array kernel implementation 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 resulting loop has optimal performance as shown in the evaluation in Section III.

`TreeKernel`. The tree work-stealing iterator [6] assumes that any subtree can be traversed with the `next` and `hasNext` calls by using a private stack, just like the linearizable `nextBatch` relies on an atomic stack. Batching can be achieved by traversing the subtree directly. Figure 1 shows a kernel in which the `root` of the subtree is traversed with a nested recursive method `traverse`.

III. Performance Evaluation

The goals of our design were to reduce abstraction and scheduling penalties to negligible levels. This section presents a performance improvement breakdown that validates these goals by identifying each of the penalties separately. We compare against programs written in Java, existing Scala Parallel Collections, a corresponding C version, OpenMP and the Intel TBB library wherever a comparison is feasible. We perform the evaluation on the Intel i7-3930K hex core 3.4 GHz processor with hyperthreading, 4x Xeon E5-4640 8-core 2.4 GHz with disabled hyperthreading and a 8-core 1.2 GHz UltraSPARC T2 with 64 hardware threads. An important difference between them is the memory throughput - i7 has a single dual-channel, while the UltraSPARC T2 has four dual-channel memory controllers.

Abstraction penalties. The microbenchmarks in Figure 2 have a minimum cost uniform workload. These tests are targeted at detecting abstraction penalties discussed earlier. The microbenchmark in Figure 2A consists of a data-parallel `foreach` loop. It shows a comparison between Parallel Collections, a generic kernel and a kernel specialized for ranges from Figure 1. In this benchmark, Parallel Collections [7] do not incur boxing costs, but suffer from iterator and function object abstraction penalties. Furthermore, the range-specialized kernel outperforms the generic kernel by 25% on the Xeon and 15% on the UltraSPARC (note the log scale).

Figure 2B evaluates parallel ranges and the `fold` operation from the introduction. Scala Parallel Collections suffer from integer boxing in this benchmark. The speed gain for a range-specialized kernel is $20\times$ to $60\times$ compared to Parallel Collections and $2.5\times$ compared to the generic kernel. Figure 2C 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 most of the time fetching the data from the main memory. This is particularly noticeable on the i7 – its dual-channel memory architecture becomes a bottleneck, limiting the speedup to $2\times$. UltraSPARC shows better scaling here due to its eight-channel memory architecture.

The `fold` operation on binary trees is shown in Figure 2D. Here we compare the generic and specialized `fold` kernels against a manually written recursive traversal subroutine. The performance difference between the generic and the specialized kernel is $2 - 3\times$.

Scheduling penalties. In Figure 3A we run a parallel `fold` method on a *step* workload – the first 97% of elements have no associated work, while the remaining 3% require a high amount of computation. Intel TBB is about 25% slower compared to the work-stealing tree scheduling. As shown in Figure 3B, Intel TBB is up to $2\times$ slower compared to work-stealing tree scheduling for an *exponential* workload where the work of the n-th element grows with the function $2^{\frac{n}{100}}$.

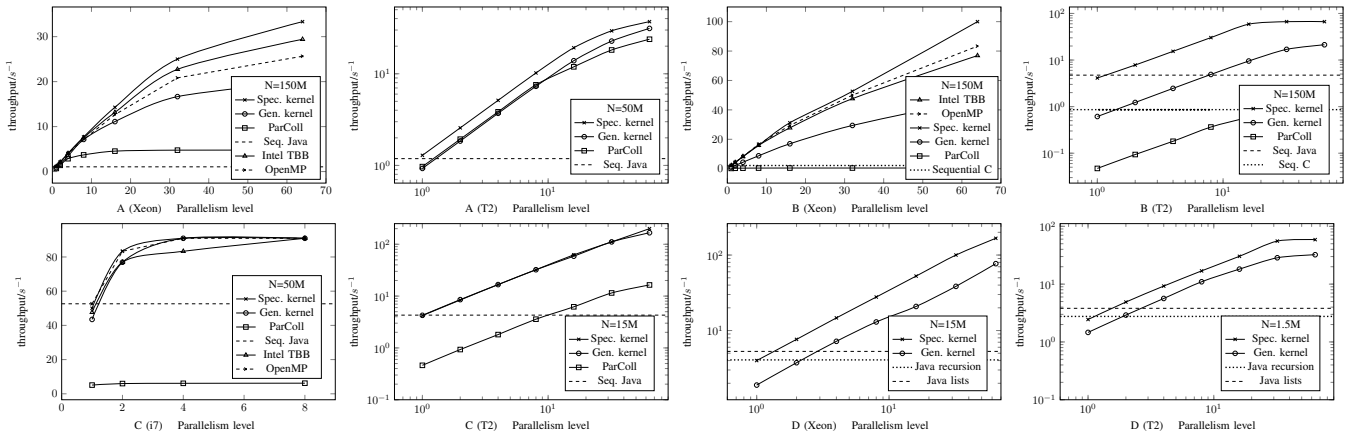

Figure 2. Benchmarks on Intel i7, 4x Xeon E5-4640 and UltraSPARC T2; A - `ParRange.foreach`, B - `ParRange.fold`, C - `ParArray.fold`, D - `Tree.fold`

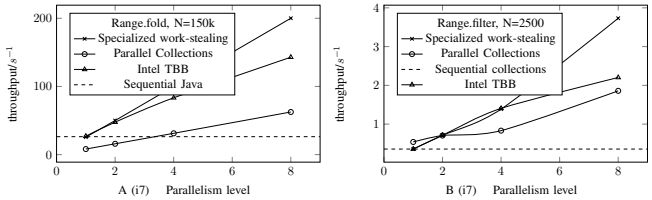

Figure 3. Irregular workloads on Intel i7; A - *step*, B - *exponential*

IV. Related Work

The fixed-size batching [3] was an early technique that allowed a more fine-grained load-balancing for scheduling data-parallel loops. This technique fails to load balance irregular workloads well. Other variable size batching approaches were proposed like *guided self-scheduling* [5], and *TSS* [9], but their static partitioning decisions have proven detrimental. In *work-stealing* [4] each worker maintains its own work queue and steals work from other workers when its queue is empty. *Work-stealing tree scheduling* [8] is a load balancing technique in which work is kept in a tree rather than a work queue. 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 [3] [5] [7] cannot cope with. Intel TBB is a C++ data-parallel programming library based on work-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.

V. Conclusion

Whereas in traditional work-stealing basic units of parallelism are parallel function calls, we proposed a set of specialized representations taking advantage of data-structure specifics to allow more efficient scheduling. The key idea is that, on one hand, these specialized representations can be processed serially with near-optimal overheads, and, orthogonally, these representations allow more fine-grained work-stealing. This work-stealing proceeds in a lock-free manner, allowing the idle worker threads to steal work from busy workers without waiting for their cooperation.

References

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

[2] I. Dragos and M. Odersky. Compiling generics through user-directed type specialization. ICOOOLPS '09, pages 42–47, New York, NY, USA, 2009. ACM.

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

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

[5] 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.

[6] A. Prokopec. *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime*. Phd thesis, École polytechnique fédérale de Lausanne, 2014.

[7] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A generic parallel collection framework. Euro-Par'11, pages 136–147, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] A. Prokopec and M. Odersky. Near optimal work-stealing tree scheduler for highly irregular data-parallel workloads. 2013.

[9] 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.