

Near Optimal Work-Stealing Tree Scheduler for Highly Irregular Data-Parallel Workloads

Aleksandar Prokopec^(✉) and Martin Odersky

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
aleksandar.prokopec@gmail.com, martin.odersky@epfl.ch

Abstract. We present a work-stealing algorithm for runtime scheduling of data-parallel operations in the context of shared-memory architectures on data sets with highly-irregular workloads that are not known a priori to the scheduler. This scheduler can parallelize loops and operations expressible with a parallel reduce or a parallel scan. The scheduler is based on the work-stealing tree data structure, which allows workers to decide on the work division in a lock-free, workload-driven manner and attempts to minimize the amount of communication between them. A significant effort is given to showing that the algorithm has the least possible amount of overhead.

We provide an extensive experimental evaluation, comparing the advantages and shortcomings of different data-parallel schedulers in order to combine their strengths. We show specific workload distribution patterns appearing in practice for which different schedulers yield suboptimal speedup, explaining their drawbacks and demonstrating how the work-stealing tree scheduler overcomes them. We thus justify our design decisions experimentally, but also provide a theoretical background for our claims.

1 Introduction

In data-parallel programming models parallelism is not expressed as a set process interactions but as a sequence of parallel operations on data sets. Programs are typically composed from high-level data-parallel operations, and are declarative rather than imperative in nature, which is of particular interest when it comes to programming the ever more present multicore systems. Solutions to many computational problems contain elements which can be expressed in terms of data-parallel operations [12].

We show several examples of data-parallel programs in Fig. 1. These programs rely heavily on higher-order data-parallel operations such as `map`, `reduce` and `filter`, which take a function argument – they are parametrized by a mapping function, a reduction operator or a filtering predicate, respectively. The first example in Fig. 1 computes the variance of a set of measurements `ms`. It starts by computing the mean value using the higher-order operation `sum`, and then `maps` each element of `ms` into a set of squared distances from the mean value, the `sum` of which divided by the number of elements is the variance `v`. The amount of work executed

for each measurement value is equal, so we call this workload *uniform*. This need not be always so. The second program computes all the prime numbers from 3 until N by calling a data-parallel `filter` on the corresponding range. The `filter` uses a predicate that checks that no number from 2 to \sqrt{i} divides i . The workload is not uniform nor independent of i and the processors working on the end of the range need to do more work. This example also demonstrates that data-parallelism can be *nested* – the `forall` can be done in parallel as each element may require a lot of work. On the other hand, the `reduce` in the third program that computes a sum of numbers from 0 to N requires a minimum amount of work for each element. A good data-parallel scheduler must be efficient for all the workloads – when executed with a single processor the `reduce` in the third program must have the same running time as the `while` loop in the fourth program, the data-parallelism of which is not immediately obvious due to its imperative style.

```

1 val sz = ms.size      1 val r = 3 until N      1 val r = 0 until N      1 var sum = 0
2 val a = ms.sum / sz  2 val ps = r filter { 2 val sum = r reduce { 2 var i = 0
3 val ds = ms map {    3 i =>                      3 (acc, i) =>           3 while (i < N) {
4 x => (x - avg)^2     4 2 to [sqrt] forall { 4 acc + i              4 sum += i
5 }                   5 d => i % d != 0 }    5 }                   5 i += 1
6 val v = ds.sum / sz 6 }                               6 }
7 }

```

Fig. 1. Data parallel program examples

It has been a trend in many languages to provide data-parallel bulk operations on collections [3–5, 17, 18]. Data-parallel operations are generic as shown in Fig. 1 – for example, `reduce` takes a user-provided operator, such as number addition, string concatenation or matrix multiplication. The computational costs of these generic parts, and hence the workload distribution, cannot always be determined statically, so efficient assignment of work to processors often relies on the runtime scheduling. Scheduling in this case entails dividing the elements into batches on which the processors work in isolation. Work-stealing [1, 7, 8, 15, 20] is one solution to this problem. In this technique different processors occasionally steal batches from each other to load balance the work – the goal is that no processor stays idle for too long.

In this paper we propose and describe a runtime scheduler for data-parallel operations on shared-memory architectures that uses a variant of work-stealing to ensure proper load-balancing. The scheduler relies on a novel data structure with lock-free synchronization operations called the **work-stealing tree**. To show that the work-stealing tree scheduler is optimal we focus on evaluating scheduler performance on uniform workloads with a minimum amount of computation per element, irregular workloads for which this amount varies and workloads with a very coarse granularity.

Our algorithm is based on the following assumptions. There are no fast, accurate means to measure elapsed time with sub-microsecond precision, i.e. there is no way to measure the running time. There is no static or runtime information about the cost of an operation – when invoking a data-parallel operation we do

not know how much computation each element requires. There are no hardware-level interrupt handling mechanisms at our disposal – the only way to interrupt a computation is to have the processor check a condition. We assume OS threads as parallelism primitives, with no control over the scheduler. We assume that the available synchronization primitives are monitors and the CAS instruction. We assume the presence of automatic memory management.

The rest of the paper is organized as follows. Section 2 describes related work and alternative schedulers we compare against. Section 3 describes the work-stealing tree scheduler. In Sect. 4 we evaluate the scheduler for different workloads as well as tune several of its parameters, and in Sect. 5 we conclude.

2 Related Work

Per processor (henceforth, worker) work assignment done statically during compile time or linking, to which we will refer to as *static batching*, was studied extensively [13, 19]. Static batching cannot correctly predict workload distributions for any problem, as shown by the second program in Fig. 1. Without knowing the numbers in the set exactly, batches cannot be statically assigned to workers in an optimal way – some workers may end up with more work than the others. Still, although cost analysis is not the focus here, we advocate combining static analysis with runtime techniques.

To address the need for load balancing at runtime, work can be divided into a lot of small batches. Only once each worker processes its batch, it requests a new batch from a centralized queue. We will refer to this as *fixed-size batching* [14]. In fixed-size batching the workload itself dictates the way how work is assigned to workers. This is a major difference with respect to static batching. In general, in the absence of information about the workload distribution, scheduling should be *workload-driven*. A natural question arises – what is the ideal size for a batch? Ideally, a batch should consist of a single element, but the cost of requesting work from a centralized queue is prohibitively large for that. For example, replacing the increment `i += 1` with an atomic CAS can increase the running time of a `while` loop by nearly a magnitude on modern architectures. The batch size has to be the least number of elements for which the cost of accessing the queue is amortized by the actual work. There are two issues with this technique. First, it is not scalable – as the number of workers increases, so does contention on the work queue (Fig. 6). This requires increasing batch sizes further. Second, as the granularity approaches the batch size, the work division is not fine-grained and the speedup is suboptimal (Fig. 8, where size is less than 1024).

Guided self-scheduling [16] solves some granularity issues by dynamically choosing the batch size based on the number of remaining elements. At any point, the batch size is R_i/P , where R_i is the number of remaining elements and P is the number of workers – the granularity becomes finer as there is less and less work. Note that the first-arriving worker is assigned the largest batch of work. If this batch contains more work than the rest of the loop due to irregularity, the speedup will not be linear. This is shown in Figs. 8-20 and 9-35.

Factoring [10] and *trapezoidal self-scheduling* [21] improve on guided-self scheduling, but have the same issue with those workload distributions.

One way to overcome the contention issues inherent to the techniques above is to use several work queues rather than a centralized queue. In this approach each processor starts with some initial work on its queue and commonly steals from other queues when it runs out of work – this is known as *work-stealing*, a technique applicable to both task- and data-parallelism. One of the first uses of work-stealing dates to the Cilk language [2, 8], in which processors relied on the fast and slow version of the code to steal stack frames from each other. Recent developments in the X10 language are based on similar techniques [20]. Work-stealing typically relies on the use of work-stealing queues [1, 7, 8, 15] and dequeues [6], implementations ranging from blocking to lock-free. While in the past data-parallel collections frameworks relied on using task-parallel schedulers under the hood [11, 17, 18], to the best of our knowledge, the tree data structure was not used for synchronization in work-stealing prior to this work, nor for data-parallel operation scheduling.

3 Work-Stealing Tree Scheduler

In this section we describe the work-stealing tree data structure and the scheduling algorithm that the workers run. We first briefly discuss the aforementioned fixed-size batching. We have mentioned that the contention on the centralized queue is one of its drawbacks. We could replace the centralized queue with a queue for each worker and use work-stealing. However, this seems overly eager – we do not want to create as many work queues as there are workers for each parallel operation, as doing so may outweigh the actually useful work. We should start with a single queue and create additional ones on-demand. Furthermore, fixed-size batching seems appropriate for scheduling parallel loops, but what about the reduce operation? If each worker stores its own intermediate results separately, then the `reduce` may not be applicable to non-commutative operators (e.g. string concatenation). It seems reasonable to have the work-stealing data-structure store the intermediate results, since it has the division order information.

With this in mind, we note that a tree seems particularly applicable. When created it consists merely of a single node – a root representing the operation and all the elements of the range. The worker invoking the parallel operation can work on the elements and update its progress by writing to the node it owns. If it completes before any other worker requests work, then the overhead of the operation is merely creating the root. Conversely, if another worker arrives, it can steal some of the work by creating two child nodes, splitting the elements and continuing work on one of them. This proceeds recursively. Scheduling is thus workload-driven – nodes are created only when some worker runs out of work meaning that another worker had too much work. Such a tree can also store intermediate results in the nodes, serving as a reduction tree.

How can such a tree be used for synchronization and load-balancing? We assumed that the parallelism primitives are OS threads. We can keep a pool of

threads [15] that are notified when a parallel operations is invoked – we call these workers. We first describe the worker algorithm from a high-level perspective. Each worker starts by calling the tail-recursive `run` method in Fig. 2. It looks for a node in the tree that is either not already owned or steals a node which some other worker works on by calling `findWork` in line 3. This node is initially a leaf, but we call it a subtree. The worker works on the subtree by calling `descend` in line 5, which calls `workOn` on the root of the subtree to work on it until it is either completed or stolen. In the case of a steal, the worker continues work on one of the children if it can own it in line 11. This is repeated until `findWork` returns \perp (`null`), indicating that all the work is completed.

```

struct Ptr           1 def run(): Unit =           7 def descend(leaf: Ptr): Unit =
  child: Node        2   val leaf =                 8   val nosteals = workOn(leaf)
struct Node         3   findWork(root)             9   if ( $\neg$ nosteals)
  left, right: Ptr   4   if (leaf  $\neq$   $\perp$ )         10  val sub = READ(leaf.child).left
  start, until: Int  5     descend(leaf)       11  if (tryOwn(READ(sub.child)))
  progress: Int      6     run()                 12  descend(subnode)
  owner: Owner

```

Fig. 2. Work-stealing tree data-types and the scheduling algorithm

In Fig. 2 we also present the *work-stealing tree* and its basic data-types. We use the keyword `struct` to refer to a compound data-type – this can be a Java class or a C structure. We define two compound data-types. `Ptr` is a reference to the tree – it has only a single member `child` of type `Node`. Write access to `child` has to be atomic and globally visible (in Java, this is ensured with the `volatile` keyword). `Node` contains immutable references to the `left` and `right` subtree, initialized upon instantiation. If these are set to \perp we consider the node a leaf. We initially focus on parallelizing loops over ranges, so we encode the current state of iteration with three integers. Members `start` and `until` are immutable and denote the initial range – for the root of the tree this is the entire loop range. Member `progress` has atomic, globally visible write access. It is initially set to `start` and is updated as elements are processed. Finally, the `owner` field denotes the worker that is working on the node. It is initially \perp and also has atomic write access. Example trees are shown in Fig. 3.

Before we describe the operations and the motivation behind these data-types we will define the states work-stealing tree can be in (see Fig. 3), namely its invariants. This is of particular importance for concurrent data structures which have non-blocking operations. Work-stealing tree operations are lock-free, a well-known advantage [9], which comes at the cost of little extra complexity in this case.

INV1. Whenever a new node reference `Ptr p` becomes reachable in the tree, it initially points to a *leaf* `Node n`, such that `n.owner = \perp` . Field `n.progress` is set to `n.start` and `n.until \geq n.start`. The subtree is in the AVAILABLE state and its range is `(n.start, n.until)`.

INV2. The set of transitions of `n.owner` is $\perp \rightarrow \pi \neq \perp$. No other field of `n` can be written until `n.owner` $\neq \perp$. After this happens, the subtree is in the OWNED state.

INV3. The set of transitions of `n.progress` in the OWNED state is $p_0 \rightarrow p_1 \rightarrow \dots \rightarrow p_k$ such that `n.start` = $p_0 < p_1 < \dots < p_k < \text{n.until}$. If a worker π writes a value from this set of transitions to `n.progress`, then `n.owner` = π .

INV4. If the worker `n.owner` writes the value `n.until` to `n.progress`, then that is the last transition of `n.progress`. The subtree goes into the COMPLETED state.

INV5. If a worker ψ overwrites p_i , such that `n.start` $\leq p_i < \text{n.until}$, with $p_s = -p_i - 1$, then $\psi \neq \text{n.owner}$. This is the last transition of `n.progress` and the subtree goes into the STOLEN state.

INV6. The field `p.child` can be overwritten only in the STOLEN state, in which case its transition is `n` \rightarrow `m`, where `m` is a copy of `n` with `m.left` and `m.right` being fresh leaves in the AVAILABLE state with ranges $r_l = \langle x_0, x_1 \rangle$ and $r_r = \langle x_1, x_2 \rangle$ such that $r_l \cup r_r = \langle p_i, \text{n.until} \rangle$. The subtree goes into the EXPANDED state.

This seemingly complicated set of invariants can be summarized in a straightforward way. Upon owning a leaf, that worker processes elements from that leaf’s range by incrementing the `progress` field until either it processes all elements or another worker requests some work by invalidating `progress`, in which case the leaf is replaced by a subtree such that the remaining work is divided between the new leaves.

Now that we have formally defined a valid work-stealing tree, we provide an implementation of the basic operations (Fig. 4). These operations will be the building blocks for the scheduling algorithm that balances the workload. A worker must attempt to acquire ownership of a node before processing its elements by calling the method `tryOwn`, which returns `true` if the claim is successful. After reading the `owner` field in line 14 and establishing the AVAILABLE state, the worker attempts to atomically push the node into the OWNED state with the CAS in line 15. This CAS can fail either due to a faster worker claiming ownership or spuriously – a retry follows in both cases.

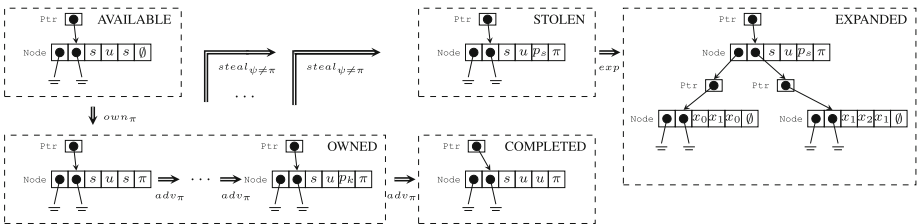


Fig. 3. Work-stealing subtree state diagram

```

13 def tryOwn(n: Node): Boolean =
14   if (READ(n.owner) ≠ ⊥) false
15   else if (CAS(n.owner, ⊥, π)) true
16   else tryOwn(n)
17
18 def tryAdvance(n: Node, p: Int): Int =
19   val q = min(p + STEP, n.until)
20   if (¬CAS(n.progress, p, q)) -1
21   else q - p
22
23 def isLeaf(n: Node): Boolean =
24   n.left == ⊥
25
26 def isEligible(n: Node): Boolean =
27   n.until - READ(n.progress) > 1
28
29 def trySteal(ptr: Ptr): Boolean =
30   val c_t0 = READ(ptr.child)
31   if (¬isLeaf(c_t0)) true else
32     val p_t1 = READ(c_t0.progress)
33     if (p_t1 == c_t0.until) false
34     else if (p_t1 ≥ 0)
35       val negp = -p_t1 - 1
36       CAS(c_t0.progress, p_t1, negp)
37       trySteal(ptr)
38     else
39       val c_exp = newExpanded(c_t0)
40       if (CAS(ptr.child, c_t0, c_exp))
41         true
42       else trySteal(ptr)

```

Fig. 4. Basic work-stealing tree operations

A worker that claimed ownership of a node repetitively calls `tryAdvance`, which attempts to reserve a batch of size `STEP` by atomically incrementing the `progress` field, eventually bringing the node into the `COMPLETED` state. If `tryAdvance` returns a nonnegative number, the owner is obliged to process that many elements, whereas a negative number is an indication that the node was stolen.

A worker searching for work must call `trySteal` if it finds a node in the `OWNED` state. This method returns `true` if the node was successfully brought into the `EXPANDED` state by any worker, or `false` if the node ends up in the `COMPLETED` state. Method `trySteal` consists of two steps. First, it attempts to push the node into the `STOLEN` state with the `CAS` in line 35 after determining that the node read in line 29 is a leaf. This `CAS` can fail either due to a different steal, a successful `tryAdvance` call or spuriously. Successful `CAS` in line 35 brings the node into the `STOLEN` state. Irregardless of success or failure, `trySteal` is then called recursively. In the second step, the expanded version of the node from Fig. 3 is created by the `newExpanded` method, the pseudocode of which is not shown here since it consists of isolated singlethreaded code. The `child` field in `Ptr` is replaced with the expanded version atomically with the `CAS` in line 39, bringing the node into the `EXPANDED` state.

We now describe the scheduling algorithm that the workers execute by invoking the `run` method. There are two basic modes of operation a worker alternates between. First, it calls `findWork`, which returns a node in the `AVAILABLE` state (line 3). Then, it calls `descend` to work on that node until it is stolen or completed, which calls `workOn` to process the elements. If `workOn` returns `false`, then the node was stolen and the worker tries to descend one of the subtrees rather than searching the entire tree for work. This decreases the total number of `findWork` invocations. The method `workOn` checks if the node is in the `OWNED` state (line 47), and then attempts to atomically increase `progress` by calling `tryAdvance`. The worker is obliged to process the elements after a successful advance, and does so by calling the `kernel` method, which is nothing more than the `while` loop like the one in Fig. 1. Generally, `kernel` can be any kind of a workload. Finally, method `findWork` traverses the tree left to right and whenever

it finds a leaf node it tries to claim ownership. Otherwise, it attempts to steal it until it finds that it is either COMPLETED or EXPANDED, returning \perp or descending deeper, respectively. Nodes with 1 or less elements left are skipped.

We explore alternative `findWork` implementations in Sect. 4. For now, we state but do not prove the following claim. If the method `findWork` does return \perp , then all the work in the tree was obtained by different workers that had called `tryAdvance` except $M < P$ loop elements distributed across M leaf nodes where P is the number of workers. This follows from the fact that the tree grows monotonically.

```

42 def workOn(ptr: Ptr): Boolean =
43   val node = READ(ptr.child)
44   var batch = -1
45   do
46     val p = READ(node.progress)
47     if (p >= 0 & p < node.until)
48       batch = tryAdvance(node, p)
49       if (batch ≠ -1)
50         kernel(p, p + batch)
51     else batch = -1
52   while (batch ≠ -1)
53   if (READ(node.progress) ≥ 0)
54     true
55   else
56     trySteal(ptr)
57     false
58 def findWork(ptr: Ptr): Node =
59   val node = READ(ptr.child)
60   if (isLeaf(node))
61     if (tryOwn(node)) node
62     else if (¬isEligible(node)) ⊥
63     else if (¬trySteal(ptr))
64       findWork(ptr)
65     else
66       val right = node.right
67       if (tryOwn(READ(right.child)))
68         READ(right.child)
69       else findWork(ptr)
70   else
71     val leftsub = findWork(node.left)
72     if (leftsub ≠ ⊥) leftsub
73     else findWork(node.right)

```

Fig. 5. Scheduling algorithm

Note that `workOn` is similar to fixed-size batching – the only difference is that an arrival of a worker invalidates the node here, whereas multiple workers simultaneously call `tryAdvance` in fixed-size batching, synchronizing repetitively. The next section starts by evaluating the impact this has on performance.

4 Evaluation

As hinted in the introduction, we want to evaluate how good our scheduler is for uniform workloads with a low amount of work per element. The reasons for this are twofold – first, we want to compare speedups against an optimal sequential program. Second, such problems appear in practical applications. We thus ensure that the third and fourth program from Fig. 1 really have the same performance for a single processor. We will call the `while` loop from Fig. 1 the *sequential baseline*.

Parallelizing the baseline seems trivial. Assuming the workers start at roughly the same time and have roughly the same speed, we can divide the range in equal parts between them. However, an assumption from the introduction was that the workload distribution is not known and the goal is to parallelize irregular workloads as well. In fact, the workload may have a coarse granularity, consisting only of several elements.

For the reasons above, we verify that the scheduler abides the following criteria:

C1 There is no noticeable overhead when executing the baseline with a single worker.

C2 Speedup is optimal for both the baseline and typical irregular workloads.

C3 Speedup is optimal when the work granularity equals the parallelism level.

Workloads we choose correspond to those found in practice. Uniform workloads are particularly common and correspond to numeric computations, text manipulation, Monte Carlo methods and applications that involve basic linear algebra operations like vector addition or matrix multiplication. In Fig. 8 we denote this workload as UNIFORM. Triangular workloads are present in primality testing, multiplication with triangular matrices and computing an adjoint convolution (TRIANGLE). In higher dimensions computing a convolution consists of several nested loops and can have a polynomial workload distribution (PARABOLA). Depending on how the problem is formulated, the workload may be increasing or decreasing (INVTRIANGLE, HILL, VALLEY). In combinatorial problems such as word segmentation, bin packing or computing anagrams the problem subdivision can be such that the subproblems corresponding to different elements differ exponentially – we model this with an exponentially increasing workload EXP. In raytracing, PageRank or sparse matrix multiplication the workload corresponds to some probability distribution, modelled with workloads GAUSSIAN and RANDIF. Finally, in problems like Mandelbrot set computation or Barnes-Hut simulation we have large conglomeration of elements which require a lot of computation while the rest require almost no work. We call this workload distribution STEP.

All the tests were performed on an Intel i7 3.4 GHz quad-core processor with hyperthreading and Oracle JDK 1.7, using the server VM. Our implementation is written in the Scala programming language, which uses the JVM as its backend. JVM programs are commonly regarded as less efficient than programs written in C. To show that the evaluation is comparative to a C implementation, we must evaluate the performance of corresponding sequential C programs. The running time of the `while` loop from Fig. 1 is roughly 45 ms for 150 million elements in both C (GNU C++ 4.2) and on the JVM – if we get linear speedups then we can conclude that the scheduler is indeed optimal. We can thus turn our attention to criteria C1.

We stated already that the STEP value should ideally be 1 for load-balancing purposes, but has to be more coarse-grained due to communication costs that could overwhelm the baseline. In Fig. 6A we plot the running time against the STEP size, obtained by executing the baseline loop with a single worker. By finding the minimum STEP value with no observable overhead, we seek to satisfy criteria C1. The minimum STEP with no noticeable synchronization costs is around 50 elements – decreasing STEP to 16 doubles the execution time and for value 1 the execution time is 36 times larger (not shown for readability).

Having shown that the work-stealing tree is as good as fixed-size batching, we evaluate its effectiveness with multiple workers. Figure 6B shows that the mini-

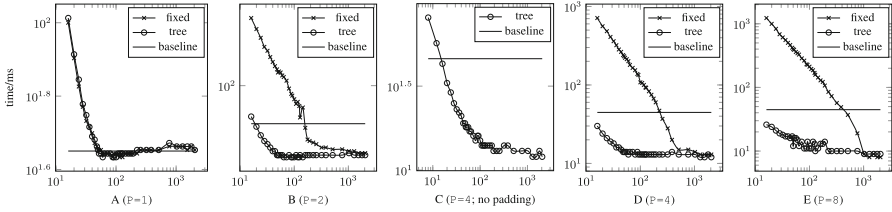


Fig. 6. Baseline running time (ms) vs. STEP size

mum STEP for fixed-size batching increases for 2 workers, as we postulated earlier. Increasing STEP decreases the frequency of synchronization and the communication costs with it. In this case the 3x slowdown is caused by processors having to exchange ownership of the `progress` field cache-line. The work-stealing tree does not suffer from this problem, since it strives to keep processors isolated – the speedup is linear with 2 workers. However, with 4 processors the performance of the naive work-stealing tree implementation is degraded (Fig. 6C). While the reason is not immediately apparent, note that for greater STEP values the speedup is once again linear. Inspecting the number of elements processed in each node reveals that the uniform workload is not evenly distributed among the topmost nodes – communication costs in those nodes are higher due to *false sharing*. Even though the two processors work on different nodes, they modify the same cache line, slowing down the CAS in line 20. Why this exactly happens in the implementation that follows directly from the pseudocode is beyond the scope of this paper, but it suffices to say that padding the node object with dummy fields to adjust its size to the cache line solves this problem, as shown in Fig. 6D, E.

The speedup is still not completely linear as the number of workers grows. Our baseline does not access main memory and only touches cache lines in exclusive mode, so this may be due to worker wakeup delay or scheduling costs in the work-stealing tree. After checking that increasing the total amount of work does not change performance, we focus on the latter. Inspecting the number of tree nodes created at different parallelism levels in Fig. 7B reveals that as the number of workers grows, the number of nodes grows at a superlinear rate. Each node incurs a synchronization cost, so could we decrease their total number?

Examining a particular work-stealing tree instance at the end of the operation reveals that different workers are battling for work in the left subtree until all the elements are depleted, whereas the right subtree remains unowned during this time. As a result, the workers in any subtree steal from each other more often, hence creating more nodes. The cause is the left-to-right tree traversal in `findWork` as defined in Fig. 5, a particularly bad stealing strategy we will call **Predefined**. As shown in Fig. 7B, the average tree size for 8 workers nears 2500 nodes. So, let's try to change the preference of a worker by changing the tree-traversal order in line 70 based on the worker index i and the level l in the tree. The worker should go left-to-right if and only if $(i \gg (l \bmod \lceil \log_2 P \rceil)) \bmod 2 = 1$ where P is the total number of workers. This way, the first path from the root

to a leaf up to depth $\log_2 P$ is unique for each worker. The choice of the subtree after a steal in lines 10 and 66 is also changed like this – the detailed implementation of `findWork` for this and other strategies is shown in the appendix. This strategy, which we call **Assign**, decreases the average tree size at $P = 8$ to 134. Interestingly, we can do even better by doing this assignment only if the node depth is below $\log_2 P$ and randomizing the traversal order otherwise. We call this strategy **AssignTop** – it decreases the average tree size at $P = 8$ to 77. Building on the randomization idea, we introduce an additional strategy called **RandomWalk** where the traversal order in `findWork` is completely randomized. However, this results in a lower throughput and bigger tree sizes. Additionally randomizing the choice in lines 10 and 66 (**RandomAll**) is even less helpful, since the stealer and the victim clash more often.

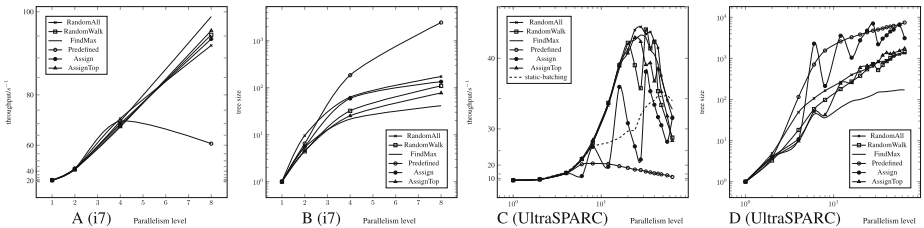


Fig. 7. Comparison of `findWork` implementations

The results of the five different strategies mentioned thus far lead to the following observation. If a randomized strategy like **RandomWalk** or **AssignTop** works better than a suboptimal strategy like **Predefined** then some of its random choices are beneficial to the overall execution time and some are disadvantageous. So, there must exist an even better strategy which only makes the choices that lead to a better execution time. Rather than providing a theoretical background for such a strategy, we propose a particular one which seems intuitive. Let workers traverse the entire tree and pick a node with most work, only then attempting to own or steal it. We call this strategy **FindMax**. Note that this cannot be easily implemented atomically, but a quiescently consistent implementation may still serve as a decent heuristic. This strategy yields an average tree size of 42 at $P = 8$, as well as a slightly better throughput – we conclude by choosing it as our default strategy. Also, the diagrams in Fig. 7 reveal the postulated inverse correlation between the tree size and total execution time, both for the Intel i7-2600 and the Sun UltraSPARC T2 processor (where `STEP` is set to 600), which is particularly noticeable for **Assign** when the total number of workers is not a power of two. For some P **RandomAll** works slightly better than **FindMax** on UltraSPARC, but both are much more efficient than static batching, which deteriorates heavily once P exceeds the number of cores.

The results so far go a long way in justifying that C1 is fulfilled. We focus on the C2 and C3 next by changing the workloads, namely the `kernel` function. Figures 8 and 9 show a comparison of the work-stealing tree and the other

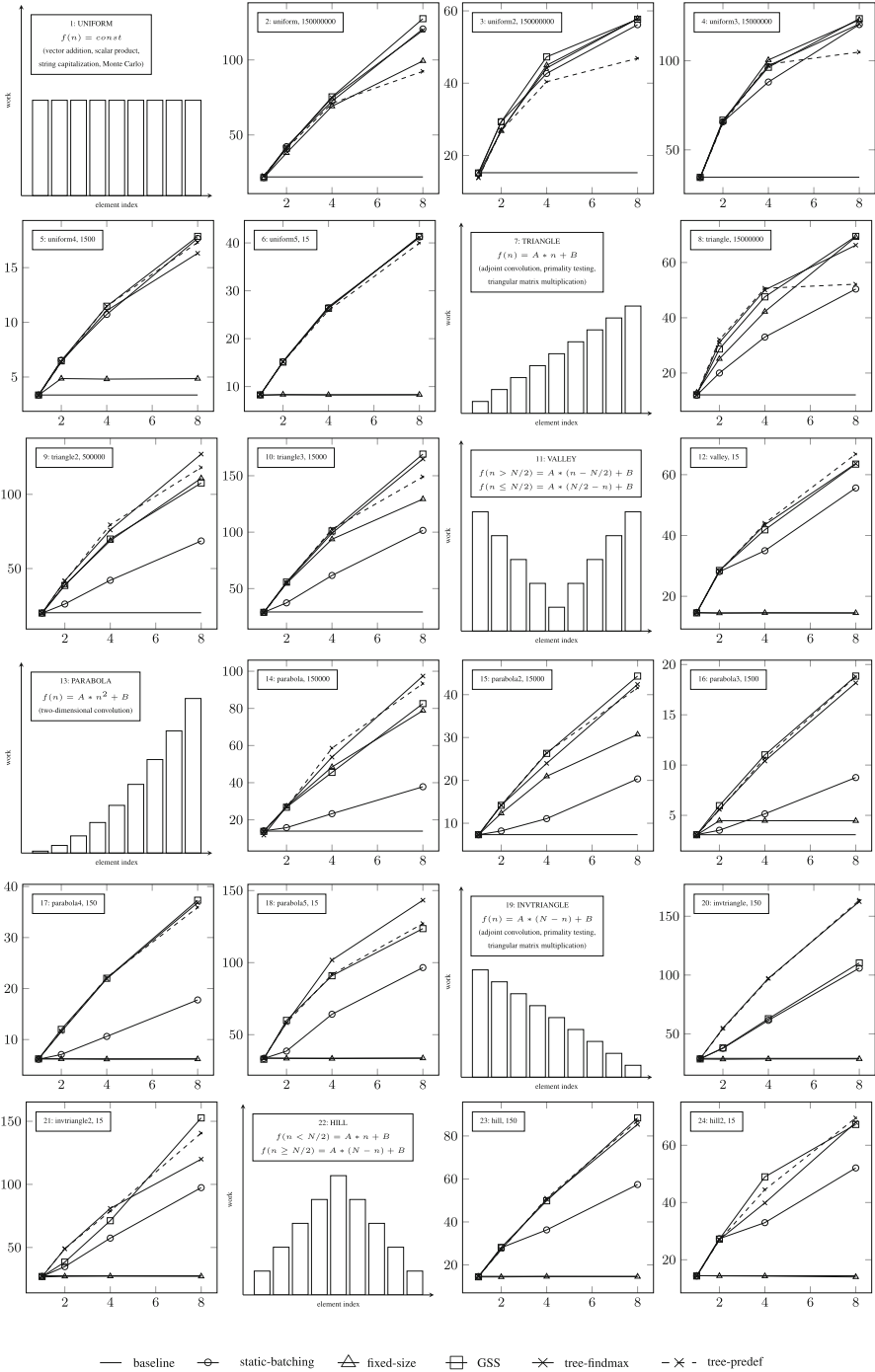


Fig. 8. Comparison of different kernel functions I (throughput/ s^{-1} vs. #workers)

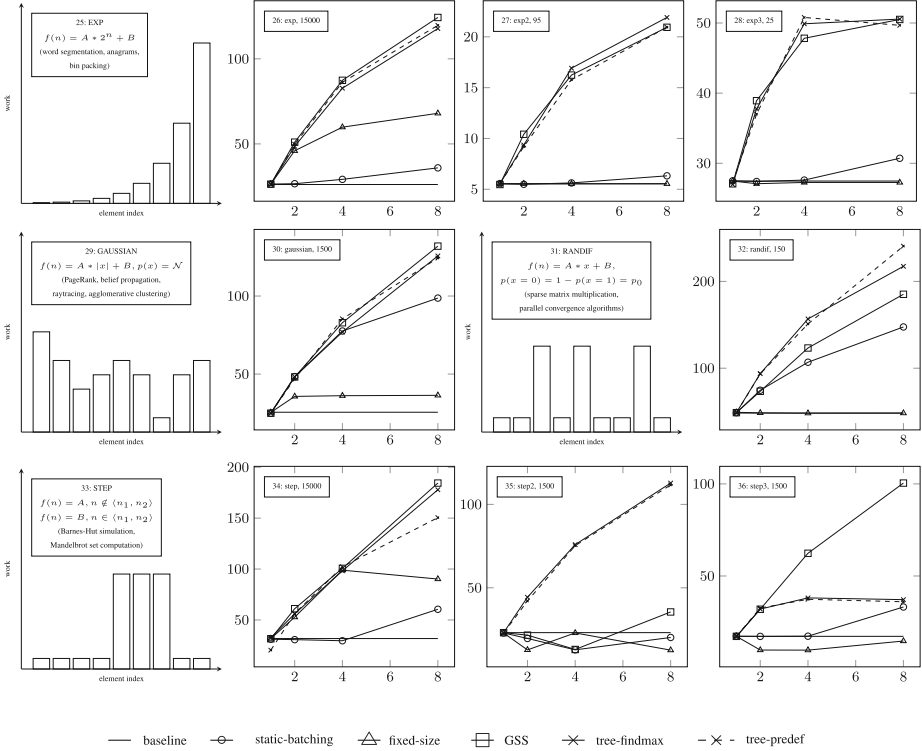


Fig. 9. Comparison of different kernel functions II (throughput/ s^{-1} vs. #workers)

schedulers on a range of different workloads. Each workload pattern is illustrated prior to its respective diagrams, along with corresponding real-world examples. To avoid memory access effects and additional layers of abstraction each workload is minimal and synthetic, but corresponds to a practical use-case. To test C3, in Fig. 8-5, 6 we decrease the number of elements to 16 and increase the workload heavily. Fixed-size batching fails utterly for these workloads – the total number of elements is on the order of or well below the estimated STEP. These workloads obviously require smaller STEP sizes to allow stealing, but that would annul the baseline performance, and we cannot distinguish the two. We address these seemingly incompatible requirements by modifying the work-stealing tree in the following way. A mutable `step` field is added to `Node`, which is initially 1 and does not require atomic access. At the end of the `while` loop in the `workOn` method the `step` is doubled unless greater than some value `MAXSTEP`. As a result, workers start processing each node by cautiously checking if they can complete a bit of work without being stolen from and then increase the `step` exponentially. This naturally slows down the overall baseline execution, so we expect the `MAXSTEP` value to be greater than the previously established `STEP`. Indeed, on the i7-2600, we had to set `MAXSTEP` to 256 to maintain the baseline performance and

at $P = 8$ even 1024. With these modifications work-stealing tree yields linear speedup for all uniform workloads.

Triangular workloads such as those shown in Fig. 8-8, 9, 10 show that static batching can yield suboptimal speedup due to the uniform workload assumption. Figure 8-20 shows the inverse triangular workload and its negative effect on guided self-scheduling – the first-arriving processor takes the largest batch of work, which incidentally contains most work. We do not inverse the other increasing workloads, but stress that it is neither helpful nor necessary to have batches above a certain size.

Figure 9-28 shows an exponentially increasing workload, where the work associated with the last element equals the rest of the work – the best possible speedup is 2. Figure 9-30, 32 shows two examples where a probability distribution dictates the workload, which occurs often in practice. Guided self-scheduling works well when the distribution is relatively uniform, but fails to achieve optimal speedup when only a few elements require more computation, for reasons mentioned earlier.

In the STEP distributions all elements except those in some range $\langle n_1, n_2 \rangle$ are associated with a very low amount of work. The range is set to 25% of the total number of elements. When its absolute size is above MAXSTEP, as in Fig. 9-34, most schedulers do equally well. However, not all schedulers achieve optimal speedup as we decrease the total number of elements N and the range size goes below MAXSTEP. In Fig. 9-35 we set $n_1 = 0$ and $n_2 = 0.25N$. Schedulers other than the work-stealing tree achieve almost no speedup, each for the same reasons as before. However, in Fig. 9-36, we set $n_1 = 0.75N$ and $n_2 = N$ and discover that the work-stealing tree achieves a suboptimal speedup. The reason is the exponential batch increase – the first worker acquires a root node and quickly processes the cheap elements, having increased the batch size to MAXSTEP by the time it reaches the expensive ones. The real work is thus claimed by the first worker and the others are unable to acquire it. Assuming some batches are smaller and some larger as already explained, this problem cannot be worked around by a different batching order – there always exists a workload distribution such that the expensive elements are in the largest batch. In this adversarial setting the existence of a suboptimal work distribution for every batching order can only be overcome by randomization. We omit the details due to reasons of space, but briefly explain how to randomize batching in the appendix, showing how to improve the expected speedup.

Finally, we conclude this section by comparing the new scheduler with an existing scheduler implementation used in the Scala Parallel Collections [17] in Fig. 10. The Scala Parallel Collections scheduler is an example of an adaptive data-parallel scheduler relying a task-parallel scheduler under the hood [15]. The batching order is chosen so that the sizes increase exponentially. At any point, the largest batch (task) is eligible for stealing – after a steal, the batch is divided in the same batching order. Due to the overheads of preemptively creating batch tasks and scheduling them, Scala Parallel Collections use a bound on the minimum batch size.

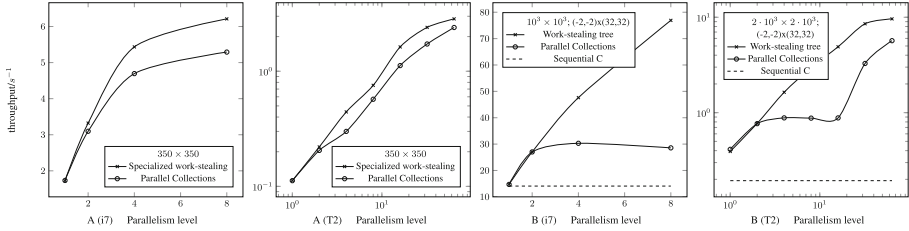


Fig. 10. (A) Matrix multiplication and (B) Mandelbrot sets on i7 and UltraSPARC T2

In Fig. 10 we evaluate the performance of Scala Parallel Collections against the new scheduler against two benchmark applications – triangular matrix multiplication and Mandelbrot set computation. Triangular matrix multiplication has a linearly increasing workload. Scala Parallel Collections scale as the number of processors increases on both the i7 and the UltraSPARC machine, although they are slower by a constant factor. However, in the Mandelbrot set benchmark where we render set in the part of the plane ranging from $(-2, -2)$ to $(32, 32)$, they do not scale beyond $P = 2$ on the i7, and only start scaling after $P = 16$ on the UltraSPARC. The reason is that the computationally expensive elements around the coordinates $(0, 0)$ end up in a single batch and work on them cannot be parallelized. The work-stealing tree offers a more lightweight form of work-stealing with smaller batches and better load balancing.

5 Conclusion

We presented a scheduling algorithm for data-parallel operations that fulfills the specified criteria. Based on the experiments, we draw the following conclusions:

1. Minimum batch size on modern architectures needed to efficiently parallelize the sequential baseline typically ranges from a few dozen to several hundred elements.
2. There is no need to make batches larger than some architecture-specific size `MAXSTEP`, which is independent of the problem size – in fact, the approach employed by guided self-scheduling and factoring can be detrimental.
3. Batching can and should occur in isolation – by having workers communicate only when they run out of work batching can be more fine-grained (Fig. 6).
4. Certain workloads require single element batches, in which case batch size has to be modified dynamically. Exponentially increasing batch size from 1 up to `MAXSTEP` works for different workloads (Fig. 9).
5. When the dominant part of the workload is distributed across a range of elements smaller than `MAXSTEP`, the worst-case speedup can be 1. Randomizing the batching order can improve the average speedup.

We hinted that the work-stealing tree serves as a reduction tree, and we show the details in the appendix. We give some theoretical background to the conclusions from the experiments in the appendix as well. In the paper, we focused

on parallel loops, but arrays, hash tables and trees are also eligible for parallel traversal [3, 17, 18]. The range iterator state was encoded with a single integer, but the state of other data structure iterators, as well as batching and stealing, may be more complex. While the CAS-based implementation of `tryAdvance` and `trySteal` ensures lock-freedom, CAS instructions in those methods can be replaced with short critical sections for more complicated iterators – the work-stealing tree algorithm is potentially applicable to other data structures in a straightforward way.

A Appendix

We provide the appendix section to further explain some of the concepts mentioned in the main paper which did not fit there. The information here is provided for convenience and it should not be necessary to read this section, but doing so may give useful insight.

A.1 Work-Stealing Reduction Tree

As mentioned, the work-stealing tree is a particularly effective data-structure for a reduce operation. Parallel reduce is useful in the context of many other operations, such as finding the first element with a given property, finding the greatest element with respect to some ordering, filtering elements with a given property or computing an aggregate of all the elements (e.g. a sum).

There are two reasons why the work-stealing tree is amenable to implementing reductions. First, it preserves the order in which the work is split between processors, which allows using non-commutative operators for the reduce (e.g. computing the resulting transformation from a series of affine transformations can be parallelized by multiplying a sequence of matrices – the order is in this case important). Second, the reduce can largely be performed in parallel, due to the structure of the tree.

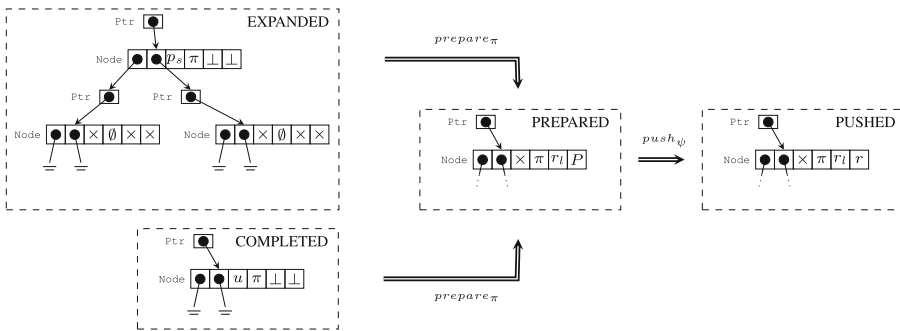


Fig. 11. Reduction state diagram

The work-stealing tree reduce works similar to a software combining tree [9], but it can proceed in a lock-free manner after all the node owners have completed their work, as we describe next. The general idea is to save the aggregated result in each node and then push the result further up the tree. Note that we did not save the return value of the `kernel` method in line 50 in Fig. 5, making the scheduler applicable only to parallelizing `for` loops. Thus, we add a local variable `sum` and update it each time after calling `kernel`. Once the node ends up in a COMPLETED or EXPANDED state, we assign it the value of `sum`. Note that updating an invocation-specific shared variable instead would not only break the commutativity, but also lead to the same bottleneck as we saw before with fixed-size batching. We therefore add two new fields with atomic access to `Node`, namely `lresult` and `result`. We also add a new field `parent` to `Ptr`. We expand the set of abstract node states with two additional ones, namely PREPARED and PUSHED. The expanded state diagram is shown in Fig. 11.

The `parent` field in `Ptr` is not shown in the diagram in Fig. 11. The first two boxes in `Node` denote the left and the right child, respectively, as before. We represent the iteration state (`progress`) with a single box in `Node`. The iterator may either be stolen (p_s) or completed (u), but this is not important for the new states – we denote all such entries with \times . The fourth box represents the owner, the fifth and the sixth fields `lresult` and `result`. Once the work on the node is effectively completed, either due to a steal or a normal completion, the node owner π has to write the value of the `sum` variable to `lresult`. After doing so, the owner announces its completion by atomically writing a special value P to `result`, and by doing so pushes the node into the PREPARED state – we say that the owner prepares the node. At this point the node contains all the information necessary to participate in the reduction. The sufficient condition for the reduction to start is that the node is a leaf or that the node is an inner node and both its children are in the PUSHED state. The value `lresult` can then be combined with the `result` values of both its children and written to the `result` field of the node. Upon writing

```

74 def complete(sum: T, tree: Ptr) =
75   node = READ(tree.child)
76   stolen = READ(node.progress) < 0
77   if (stolen)
78     trySteal(tree)
79     node = READ(tree.child)
80     node.lresult = sum
81   else
82     node.lresult = sum
83   while (READ(node.result) ==  $\perp$ )
84     CAS(node.result,  $\perp$ ,  $\mathbb{P}$ )
85   pushUp(tree)
86    $\neg$ stolen
87
88 def pushUp(tree: Ptr): Unit =
89   node = READ(tree.child)
90   res0 = READ(node.result)
91   if (res0 ==  $\perp$ )
92     // not yet in PREPARED state
93   else if (res0  $\neq$   $\mathbb{P}$ )
94     // already in PUSHED state
95   else
96     res =  $\perp$ 
97     if (isLeaf(node)) res = lresult
98     else
99       left = READ(node.left.child)
100      right = READ(node.right.child)
101      rl = READ(left.result)
102      rr = READ(right.result)
103      if (rl  $\neq$   $\perp$   $\wedge$  rr  $\neq$   $\perp$ )
104        res = lresult + rl + rr
105      if (res  $\neq$   $\perp$ )
106        if (CAS(node.result, res0, res))
107          if (tree.parent  $\neq$   $\perp$ )
108            pushUp(tree.parent)
109          else tree.synchronized
110            { tree.notifyAll() }
111        else pushUp(tree)

```

Fig. 12. Reduction pseudocode

to the `result` field, the node goes into the PUSHED state. This push step can be done by any worker ψ and assuming all the owners have prepared their nodes, the reduction is lock-free. Importantly, the worker that succeeds in pushing the result must attempt to repeat the push step in the parent node. This way the reduction proceeds upwards in the tree until reaching the root. Once some worker pushes the result to the root of the tree, it notifies that the operation was completed, so that the thread that invoked the operation can proceed, in case that the parallel operation is synchronous. Otherwise, a future variable can be completed or a user callback invoked.

Before presenting the pseudocode, we formalize the notion of the states we described. In addition to the ones mentioned earlier, we identify the following new invariants.

INV6. Field `n.lresult` is set to \perp when created. If a worker π overwrites the value \perp of the field `n.lresult` then `n.owner` = π and the node `n` is either in the EXPANDED state or the COMPLETED state. That is the last write to `n.lresult`.

INV7. Field `n.result` is set to \perp when created. If a worker π overwrites the value \perp of the field `n.result` with \mathbb{P} then `n.owner` = π , the node `n` was either in the EXPANDED state or the COMPLETED state and the value of the field `n.lresult` is different than \perp . We say that the node goes into the PREPARED state.

INV8. If a worker ψ overwrites the value \mathbb{P} of the field `n.result` then the node `n` was in the PREPARED state and was either a leaf or its children were in the PUSHED state. We say that the node goes into the PUSHED state.

We modify `workOn` so that instead of lines 53 through 57, it calls the method `complete` passing it the `sum` argument and the reference to the subtree. The pseudocodes for `complete` and an additional method `pushUp` are shown in Fig. 12.

Upon completing the work, the owner checks whether the subtree was stolen. If so, it helps expand the subtree (line 78), reads the new node and writes the `sum` into `lresult`. After that, the owner pushes the node into the PREPARED state in line 84, retrying in the case of spurious failures, and calls `pushUp`.

The method `pushUp` may be invoked by the owner of the node attempting to write to the `result` field, or by another worker attempting to push the result up after having completed the work on one of the child nodes. The `lresult` field may not be yet assigned (line 92) if the owner has not completed the work – in this case the worker ceases to participate in the reduction and relies on the owner or another worker to continue pushing the result up. The same applies if the node is already in the PUSHED state (line 94). Otherwise, the `lresult` field can only be combined with the `result` values from the children if both children are in the PUSHED state. If the worker invoking `pushUp` notices that the children are not yet assigned the `result`, it will cease to participate in the reduction. Otherwise, it will compute the tentative result (line 104) and attempt to write it to `result` atomically with the CAS in line 106. A failed CAS triggers a retry, otherwise `pushUp` is called recursively on the parent node. If the current

node is the root, the worker notifies any listeners that the final result is ready and the operations ends.

A.2 Work-Stealing Tree Traversal Strategies

We showed experimentally that changing the traversal order when searching for work can have a considerable effect on the performance of the work-stealing tree scheduler. We described these *strategies* briefly how, but did not present a precise, detailed pseudocode. In this section we show different implementations of the `findWork` and `descend` methods that lead to different tree traversal orders when stealing.

Assign. In this strategy a worker with index i invoking `findWork` picks a left-to-right traversal order at some node at level l if and only if its bit at position $l \bmod \lceil \log_2 P \rceil$ is 1, that is:

$$(i \gg (l \bmod \lceil \log_2 P \rceil)) \bmod 2 = 1 \quad (1)$$

The consequence of this is that when the workers descend in the tree the first time, they will pick different paths, leading to fewer steals assuming that the workload distribution is relatively uniform. If it is not uniform, then the workload itself should amortize the creation of extra nodes. We give the pseudocode in Fig. 13.

```

112 def assign.left(p: Ptr, i: Int) =
113   val bit = i >> (p.level % log2(P))
114   bit % 2 == 1
115
116 def assign.choose(p: Ptr, i: Int) =
117   if (left(p, i)) READ(p.child).left
118   else READ(p.child).right
119
120 def assign.descend(leaf: Ptr): Unit =
121   val nosteals = workOn(leaf)
122   if (¬nosteals)
123     val sub =
124       choose(leaf, thisWorker.index)
125     if (tryOwn(READ(sub.child)))
126       descend(subnode)
127
127 def assign.findWork(ptr: Ptr): Node =
128   val node = READ(ptr.child)
129   if (isLeaf(node))
130     if (tryOwn(node)) node
131     else if (¬isEligible(node)) ⊥
132     else if (¬trySteal(ptr)) findWork(ptr)
133   else
134     val ptr =
135       choose(ptr, thisWorker.index)
136     if (tryOwn(READ(ptr.child)))
137       READ(ptr.child)
138     else findWork(ptr)
139   else if (left(ptr, thisWorker.index))
140     val leftsub = findWork(node.left)
141     if (leftsub ≠ ⊥) leftsub
142     else findWork(node.right)
143   else
144     val rightsub = findWork(node.right)
145     if (rightsub ≠ ⊥) rightsub
146     else findWork(node.left)

```

Fig. 13. Assign strategy

AssignTop. This strategy is similar to the previous one with the difference that the assignment only works as before if the level of the tree is less than or equal to $\lceil \log_2 P \rceil$. Otherwise, a random choice is applied in deciding whether traversal should be left-to-right or right-to-left. We show it in Fig. 14 where we only redefine the method `left`, and reuse the same `choose`, `descend` and `findWork`.

```

147 def top.left(p: Ptr, idx: Int) =          153 def randomall.left(p: Ptr, idx: Int) =
148   if (p.level ≤ log2(P))                 154   coinToss()
149   val bit = i >> (p.level % log2(P)) 155
150   bit % 2 == 1
151   else
152   coinToss()

```

Fig. 14. AssignTop and RandomAll strategies

RandomAll. This strategy randomizes all the choices that the stealer and the victim make. Both the tree traversal and the node chosen after the steal are thus changed in `findWork`. We show it in Fig. 14.

RandomWalk. Here we only change the tree traversal order that the stealer does when searching for work and leave the rest of the choices fixed to victim picking the left node after expansion and the stealer picking the right node. The code is shown in Fig. 15.

```

156 def randomwalk.left(p: Ptr, i: Int) =    170 def rwalk.findWork(ptr: Ptr) =
157   val bit = i >> (p.level % log2(P))    171   val node = READ(ptr.child)
158   bit % 2 == 1                          172   if (isLeaf(node))
159                                           173     if if (tryOwn(node)) node
160 def rwalk.choose(p: Ptr, i: Int) =      174     else if (~isEligible(node)) ⊥
161   if (left(p, i)) READ(p.child).left    175     else if (~trySteal(ptr))
162   else READ(p.child).right              176     findWork(ptr)
163                                           177   else
164 def rwalk.descend(leaf: Ptr) =          178   val r = node.right
165   val nosteals = workOn(leaf)           179   if (tryOwn(READ(r.child)))
166   if (~nosteals)                         180     READ(r.child)
167   val sub = READ(leaf.child).left        181     else findWork(ptr)
168   if (tryOwn(READ(sub.child)))           182   else if (left(ptr, thisWorker.index))
169     descend(subnode)                    183     val leftsub = findWork(node.left)
                                           184     if (leftsub ≠ ⊥) leftsub
                                           185     else findWork(node.right)
                                           186   else
                                           187     val rightsub = findWork(node.right)
                                           188     if (rightsub ≠ ⊥) rightsub
                                           189     else findWork(node.left)

```

Fig. 15. RandomWalk strategy

FindMax. This strategy, unlike the previous ones, does not break tree traversal early as soon as a viable node is found. Instead, it traverses the entire work-stealing tree in left-to-right order and returns a reference to a node with the most work. Only then it attempts to own or steal that node. As noted before, this kind of search is not atomic, since some nodes may be stolen and expanded in the meantime and processors advance through the nodes they own. However, we expect steals to be rare events so in most cases this search should give an exact or a nearly exact estimate. The decisions about which node the victim and the stealer take after expansion remain the same as in the basic algorithm from Fig. 5. We show the pseudocode for **FindMax** in Fig. 16.

```

190 def findmax.search(p: Ptr): Ptr =
191   if (isLeaf(p.child)) p
192   else
193     val lp = search(p.child.left)
194     val rp = search(p.child.right)
195     val l = READ(lp.child)
196     val r = READ(rp.child)
197     if (remains(l) > remains(r)) l
198     else r
199
200 def findmax.remains(n: Node) =
201   n.until - READ(n.progress)
202
202 def findmax.findWork(ptr: Ptr): Node =
203   val maxp = search(tree)
204   val max = READ(maxp.child)
205   if (remains(max) > 0)
206     if (tryOwn(max)) max
207     else if (!isEligible(max)) ⊥
208     else if (trySteal(maxp))
209       val subnode = READ(maxp.right.child)
210       if (tryOwn(subnode)) subnode
211       else findWork(ptr)
212     else findWork(ptr)
213   else ⊥

```

Fig. 16. FindMax strategy

A.3 Speedup and Optimality Analysis

In Fig. 9-36 we identified a workload distribution for which the work-stealing reduction tree had a particularly bad performance. This coarse workload consisted of a major prefix of elements which required a very small amount of computation followed by a minority of elements which required a large amount of computation. We call it coarse because the number of elements was on the order of magnitude of a certain value we called `MAXSTEP`.

To recap, the speedup was suboptimal due to the following. First, to achieve an optimal speedup for at least the baseline, not all batches can have fewer elements than a certain number. We have established this number for a particular architecture and environment, calling it `STEP`. Second, to achieve an optimal speedup for ranges the size of which is below `STEP·P`, some of the batches have to be smaller than the others. The technique we apply starts with a batch consisting of a single element and increases the batch size exponentially up to `MAXSTEP`. Third, there is no hardware interrupt mechanism available to interrupt a worker which is processing a large batch, and software emulations which consist of checking a volatile variable within a loop are too slow when executing the baseline. Fourth, the worker does not know the workload distribution and cannot measure time. All this caused a single worker obtain the largest batch before the other workers had a chance to steal some work for a particular workload distribution. Justifying these claims requires a set of more formal definitions. We start by defining the context in which the scheduler executes.

Definition 1 (Oblivious conditions). *If a data-parallel scheduler is unable to obtain information about the workload distribution, nor information about the amount of work it had previously executed, we say that the data-parallel scheduler works in oblivious conditions.*

Assume that a worker decides on some batching schedule c_1, c_2, \dots, c_k where c_j is the size of the j -th batch and $\sum_{j=1}^k c_j = N$, where N is the size of the range. No batch is empty, i.e. $c_j \neq 0$ for all j . In oblivious conditions the worker does not know if the workload resembles the baseline mentioned earlier, so it must assume that it does and minimize the scheduling overhead. The baseline is not only important from a theoretical perspective being one of the potentially worst-case workload distribution, but also from a practical one – in many problems

parallel loops have a uniform workload. We now define what this baseline means more formally.

Definition 2 (The baseline constraint). *Let the workload distribution be a function $w(i)$ which gives the amount of computation needed for range element i . We say that a data-parallel scheduler respects the baseline constraint if and only if the speedup s_p with respect to a sequential loop is arbitrarily close to linear when executing the workload distribution $w(i) = w_0$, where w_0 is the minimum amount of work needed to execute a loop iteration.*

Arbitrarily close here means that ϵ in $s_p = \frac{P}{1+\epsilon}$ can be made arbitrarily small.

The baseline constraint tells us that it may be necessary to divide the elements of the loop into batches, depending on the scheduling (that is, communication) costs. As we have seen in the experiments, while we should be able to make the ϵ value arbitrarily small, in practice it is small enough when the scheduling overhead is no longer observable in the measurement. Also, we have shown experimentally that the average batch size should be bigger than some value in oblivious conditions, but we have used particular scheduler instances. Does this hold in general, for every data-parallel scheduler? The answer is yes, as we show in the following lemma.

Lemma 1. *If a data-parallel scheduler that works in oblivious conditions respects the baseline constraint then the batching schedule c_1, c_2, \dots, c_k is such that:*

$$\frac{\sum_{j=1}^k c_j}{k} \geq S(\epsilon) \quad (2)$$

Proof. The lemma claims that in oblivious conditions the average batch size must be above some value which depends on the previously defined ϵ , otherwise the scheduler will not respect the baseline constraint.

The baseline constraint states that $s_p = \frac{P}{1+\epsilon}$, where the speedup s_p is defined as T_0/T_p , where T_0 is the running time of a sequential loop and T_p is the running time of the scheduler using P processors. Furthermore, $T_0 = T \cdot P$ where T is the optimal parallel running time for P processors, so it follows that $\epsilon \cdot T = T_p - T$. We can also write this as $\epsilon \cdot W = W_p - W$. This is due to the running time being proportionate to the total amount of executed work, whether scheduling or useful work. The difference $W_p - W$ is exactly the scheduling work W_s , so the baseline constraint translates into the following inequality:

$$W_s \leq \epsilon \cdot W \quad (3)$$

In other words, the scheduling work has to be some fraction of the useful work. Assuming that there is a constant amount of scheduling work W_c per every batch, we have $W_s = k \cdot W_c$. Lets denote the average work per element with \bar{w} . We then have $W = N \cdot \bar{w}$. Combining these relations we get $N \geq k \cdot \frac{W_c}{\epsilon \cdot \bar{w}}$, or shorter $N \geq k \cdot S(\epsilon)$. Since N is equal to the sum of all batch sizes, we derive the following constraint:

$$\frac{\sum_{j=1}^k c_j}{k} \geq \frac{W_c}{\epsilon \cdot \bar{w}} \quad (4)$$

In other words, the average batch size must be greater than some value $S(\epsilon)$ which depends on how close we want to get to the optimal speedup. Note that this value is inversely proportionate to the average amount of work per element \bar{w} – the scheduler could decide more about the batch sizes if it knew something about the average workload, and grows with the scheduling cost per batch W_c – this is why it is especially important to make the `workOn` method efficient. We already saw the inverse proportionality with ϵ in Fig. 6. In part, this is why we had to make `MAXSTEP` larger than the chosen `STEP` (we also had to increase it due to increasing the scheduling work in `workOn`, namely, W_c). This is an additional constraint when choosing the batching schedule.

With this additional constraint there always exists a workload distribution for a given batching schedule such that the speedup is suboptimal, as we show next.

Lemma 2. *Assume that $S(\epsilon) > 1$, for the desired ϵ . For any fixed batching schedule c_1, c_2, \dots, c_k there exists a workload distribution such that the scheduler executing it in oblivious conditions yields a suboptimal schedule.*

Proof. First, assume that the scheduler does not respect the baseline constraint. The baseline workload then yields a suboptimal speedup and the statement is trivially true because $S(\epsilon) > 1$.

Otherwise, assume without the loss of generality that at some point in time a particular worker ω is processing some batch c_m the size of which is greater or equal to the size of the other batches. This means the size of c_m is greater than 1, from the assumption. Then we can choose a workload distribution such that the work $W_m = \sum_{i=N_m}^{N_m+c_m} w(i)$ needed to complete batch c_m is arbitrarily large, where $N_m = \sum_{j=1}^{m-1} c_j$ is the number of elements in the batching schedule coming before the batch c_m . For all the other elements we set $w(i)$ to be some minimum value w_0 . We claim that the obtained speedup is suboptimal. There is at least one different batching schedule with a better speedup, and that is the schedule in which instead of batch c_m there are two batches c_{m_1} and c_{m_2} such that c_{m_1} consists of all the elements of c_m except the last one and c_{m_2} contains the last element. In this batching schedule some other worker can work on c_{m_2} while ω works on c_{m_1} . Hence, there exists a different batching schedule which leads to a better speedup, so the initial batching schedule is not optimal.

We can ask ourselves what is the necessary condition for the speedup to be suboptimal. We mentioned that the range size has to be on the same order of magnitude as S above, but can we make this more precise? We could simplify this question by asking what is the necessary condition for the worst-case speedup of 1 or less. Alas, we cannot find necessary conditions for all schedulers because they do not exist – there are schedulers which do not need any preconditions in order to consistently produce such a speedup (think of a sequential loop or, worse, a “scheduler” that executes an infinite loop). Also, we already saw that a suboptimal speedup may be due to a particularly bad workload distribution, so maybe we should consider only particular distributions, or have some conditions

on them. What we will be able to express are the necessary conditions on the range size for the existence of a scheduler which achieves a speedup greater than 1 on any workload. Since the range size is the only information known to the scheduler in advance, it can be used to affect its decisions in a particular implementation.

The worst-case speedups we saw occurred in scenarios where one worker (usually the invoker) started to work before all the other workers. To be able to express the desired conditions, we model this delay with a value T_d .

Lemma 3. *Assume a data-parallel scheduler that respects the baseline constraint in oblivious conditions. There exists some minimum range size N_1 for which the scheduler can yield a speedup greater than 1 for any workload distribution.*

Proof. We first note that there is always a scheduler that can achieve the speedup 1, which is merely a sequential loop. We then consider the case when the scheduler is parallelizing the baseline workload. Assume now that there is no minimum range size N_1 for which the claim is true. Then for any range size N we must be able to find a range size $N + K$ such that the scheduler still cannot yield speedup 1 or less, for a chosen K . We choose $N = \frac{f \cdot T_d}{w_0}$, where w_0 is the amount of work associated with each element in the baseline distribution and f is an architecture-specific constant describing the computation speed. The chosen N is the number of elements that can be processed during the worker wakeup delay T_d . The workers that wake up after the first worker ω processes N elements have no more work to do, so the speedup is 1. However, for range size $N + K$ there are K elements left that have not been processed. These K elements could have been in the last batch of ω . The last batch in the batching schedule chosen by the scheduler may include the N th element. Note that the only constraint on the batch size is the lower bound value $S(\epsilon)$ from Lemma 1. So, if we choose $K = 2S(\epsilon)$ then either the last batch is smaller than K or is greater than K . If it is smaller, then a worker different than ω will obtain and process the last batch, hence the speedup will be greater than 1. If it is greater, then the worker ω will process the last batch – the other workers that wake up will not be able to obtain the elements from that batch. In that case there exists a better batching order which still respects the baseline constraint and that is to divide the last batch into two equal parts, allowing the other workers to obtain some work and yielding a speedup greater than 1. This contradicts the assumption that there is no minimum range size N_1 – we know that N_1 is such that:

$$\frac{f \cdot T_d}{w_0} \leq N_1 \leq \frac{f \cdot T_d}{w_0} + 2 \cdot S(\epsilon) \quad (5)$$

Now, assume that the workload $w(i)$ is not the baseline workload w_0 . For any workload we know that $w(i) \geq w_0$ for every i . The batching order for a single worker has to be exactly the same as before due to oblivious conditions. As a result the running time for the first worker ω until it reaches the N th element can only be larger than that of the baseline. This means that the other workers will wake up by the time ω reaches the N th element, and obtain work. Thus, the speedup can be greater than 1, as before.

We have so far shown that we can decide on the average batch size if we know something about the workload, namely, the average computational cost of an element. We have also shown when we can expect the worst case speedup, potentially allowing us to take prevention measures. Finally, we have shown that any data-parallel scheduler deciding on a fixed schedule in oblivious conditions can yield a suboptimal speedup. Note the wording “fixed” here. It means that the scheduler must make a definite decision about the batching order without any knowledge about the workload, and must make the same decision every time – it must be deterministic. As hinted before, the way to overcome an adversary that is repetitively picking the worst case workload is to use randomization when producing the batching schedule. This is the topic of the next section.

A.4 Overcoming the Worst-Case Speedup Using Randomization

Recall that the workload distribution that led to a bad speedup in our evaluation consisted of a sequence of very cheap elements followed by a minority of elements which were computationally very expensive. On the other hand, when we inverted the order of elements, the speedup became linear. The exponential backoff approach is designed to start with smaller batches first in hopes of hitting the part of the workload which contains most work as early as possible. This allow other workers to steal larger pieces of the remaining work, hence allowing a more fine grained batch subdivision. In this way the scheduling algorithm is workload-driven – it gives itself its own feedback. In the absence of other information about the workload, the knowledge that some worker is processing some part of the workload long enough that it can be stolen from is the best sign that the workload is different than the baseline, and that the batch subdivision can circumvent the baseline constraint. This heuristic worked in the example from Fig. 9-36 when the expensive elements were reached first, but failed when they were reached in the last, largest batch, and we know that there has to be a largest batch by Lemma 1 – a single worker must divide the range into batches the mean size of which has a lower bound. In fact, no other deterministic scheduler can yield an optimal speedup for all schedules, as shown by Lemma 2. For this reason we look into randomized schedulers.

In particular, in the example from the evaluation we would like the scheduler to put the smallest batches at the end of the range, but we have no way of knowing if the most expensive elements are positioned somewhere else. With this in mind we randomize the batching order. The baseline constraint still applies in oblivious conditions, so we have to pick different batch sizes with respect to the constraints from Lemma 1. Lets pick exactly the same set of exponentially increasing batches, but place consequent elements into different batches randomly. In other words, we permute the elements of the range and then apply the previous scheme. We expect some of the more expensive elements to be assigned to the smaller batches, giving other workers a higher opportunity to steal a part of the work.

In evaluating the effectiveness of this randomized approach we will assume a particular distribution we found troublesome. We define it more formally.

Definition 3 (Step workload distribution). A step workload distribution is a function which assigns a computational cost $w(i)$ to each element i of the range of size N as follows:

$$w(i) = \begin{cases} w_e, & i \in [i_1, i_2] \\ w_0, & i \notin [i_1, i_2] \end{cases} \quad (6)$$

where $[i_1, i_2]$ is a subsequence of the range, w_0 is the minimum cost of computation per element and $w_e \gg w_0$. If $w_e \geq f \cdot T_d$, where f is the computation speed and T_d is the worker delay, then we additionally call the workload highly irregular. We call $D = 2^d = i_2 - i_1$ the span of the step distribution. If $(N - D) \cdot \frac{w_0}{f} \leq T_d$ we also call the workload short.

We can now state the following lemma. We will refer to the randomized batching schedule we have described before as the **randomized permutation with an exponential backoff**. Note that we implicitly assume that the worker delay T_d is significantly greater than the time T_c spent scheduling a single batch (this was certainly true in our experimental evaluation).

Lemma 4. *When parallelizing a workload with a highly irregular short step workload distribution the expected speedup inverse of a scheduler using randomized permutations with an exponential backoff is:*

$$\langle s_p^{-1} \rangle = \frac{1}{P} + \left(1 - \frac{1}{P}\right) \cdot \frac{(2^k - 2^d - 1)!}{(2^k - 1)!} \cdot \sum_{i=0}^{k-1} 2^i \frac{(2^k - 2^i - 1)!}{(2^k - 2^i - 2^d)!} \quad (7)$$

where $D = 2^d \gg P$ is the span of the step workload distribution.

Proof. The speedup s_p is defined as $s_p = \frac{T_0}{T_p}$ where T_0 is the running time of the optimal sequential execution and T_p is the running time of the parallelized execution. We implicitly assume that all processors have the same the same computation speed f . Since $w_e \gg w_0$, the total amount of work that a sequential loop executes is arbitrarily close to $D \cdot w_e$, so $T_0 = \frac{D}{f}$. When we analyze the parallel execution, we will also ignore the work w_0 . We will call the elements with cost w_e *expensive*.

We assumed that the workload distribution is highly irregular. This means that if the first worker ω starts the work on an element from $[i_1, i_2]$ at some time t_0 then at the time $t_1 = t_0 + \frac{w_e}{f}$ some other worker must have already started working as well, because $t_1 - t_0 \geq T_d$. Also, we have assumed that the workload distribution is short. This means that the first worker ω can complete work on all the elements outside the interval $[i_1, i_2]$ before another worker arrives. Combining these observations, as soon as the first worker arrives at an expensive element, it is possible for the other workers to parallelize the rest of the work.

We assume that after the other workers arrive there are enough elements left to efficiently parallelize work on them. In fact, at this point the scheduler will typically change the initially decided batching schedule – additionally arriving

workers will steal and induce a more fine-grained subdivision. Note, however, that the other workers cannot subdivide the batch on which the current worker is currently working on – that one is no longer available to them. The only batches with elements of cost w_e that they can still subdivide are the ones coming after the first batch in which the first worker ω found an expensive element. We denote this batch with c_ω . The batch c_ω may, however, contain additional expensive elements and the bigger the batch the more probable this is. We will say that the total number of expensive elements in c_ω is X . Finally, note that we assumed that $D \gg P$, so our expression will only be an approximation if D is very close to P .

We thus arrive at the following expression for speedup:

$$s_p = \frac{D}{X + \frac{D-X}{P}} \quad (8)$$

Speedup depends on the value X . But since the initial batching schedule is random, the speedup depends on the random variable and is itself random. For this reason we will look for its expected value. We start by finding the expectation of the random variable X .

We will now solve a more general problem of placing balls to an ordered set of bins and apply the solution to finding the expectation of X . There are k bins, numbered from 0 to $k - 1$. Let c_i denote the number of balls that fit into the i th bin. We randomly assign D balls to bins, so that the number of balls in each bin i is less than or equal to c_i . In other words, we randomly select D slots from all the $N = \sum_{i=0}^{k-1} c_i$ slots in all the bins together. We then define the random variable X to be the number of balls in the non-empty bin with the smallest index i . The formulated problem corresponds to the previous one – the balls are the expensive elements and the bins are the batches.

An alternative way to define X is as follows:

$$X = \sum_{i=0}^{k-1} \begin{cases} \text{number of balls in bin } i & \text{if all the bins } j < i \text{ are empty} \\ 0 & \text{otherwise} \end{cases} \quad (9)$$

Applying the linearity property, the expectation $\langle X \rangle$ is then:

$$\langle X \rangle = \sum_{i=0}^{k-1} \langle \text{number of balls in bin } i \text{ given that all the bins } j < i \text{ are empty, and } 0 \text{ otherwise} \rangle \quad (10)$$

The expectation in the sum is conditional on the event that all the bins coming before i are empty. We call the probability of this event p_i . We define b_i as the number of balls in any bin i . From the properties of conditional expectation we then have:

$$\langle X \rangle = \sum_{i=0}^{k-1} p_i \cdot \langle b_i \rangle \quad (11)$$

The number of balls in any bin is the sum of the balls in all the slots of that bin which spans slots n_{i-1} through $n_{i-1} + c_i$. The expected number of balls in a bin i is thus:

$$\langle b_i \rangle = \sum_{i=n_{i-1}}^{n_{i-1}+c_i} \langle \text{expected number of balls in a single slot} \rangle \quad (12)$$

We denote the total capacity of all the bins $j \geq i$ as q_i (so that $q_0 = N$ and $q_{k-1} = 2^{k-1}$). We assign balls to slots randomly with a uniform distribution – each slot has a probability $\frac{D}{q_i}$ of being selected. Note that the denominator is not N – we are calculating a conditional probability for which all the slots before the i th bin are empty. The expected number of balls in a single slot is thus $\frac{D}{q_i}$. It follows that:

$$\langle b_i \rangle = c_i \cdot \frac{D}{q_i} \quad (13)$$

Next, we compute the probability p_i that all the bins before the bin i are empty. We do this by counting the events in which this is true, namely, the number of ways to assign balls in bins $j \geq i$. We will pick combinations of D slots, one for each ball, from a set of q_i slots. We do the same to enumerate all the assignments of balls to bins, but with $N = q_0$ slots, and obtain:

$$p_i = \frac{\binom{q_i}{D}}{\binom{q_0}{D}} \quad (14)$$

We assumed here that $q_i \geq D$, otherwise we cannot fill all D balls into bins. We could create a constraint that the last batch is always larger than the number of balls. Instead, we simply define $\binom{q_i}{D} = 0$ if $q_i < D$ – there is no chance we can fit more than q_i balls to q_i slots. Combining these relations, we get the following expression for $\langle X \rangle$:

$$\langle X \rangle = D \cdot \frac{(q_0 - D)!}{q_0!} \sum_{i=0}^{k-1} c_i \cdot \frac{(q_i - 1)!}{(q_i - D)!} \quad (15)$$

We use this expression to compute the expected speedup inverse. By the linearity of expectation:

$$\langle s_p^{-1} \rangle = \frac{1}{P} + \left(1 - \frac{1}{P}\right) \cdot \frac{(q_0 - D)!}{q_0!} \sum_{i=0}^{k-1} c_i \cdot \frac{(q_i - 1)!}{(q_i - D)!} \quad (16)$$

This is a more general expression than the one in the claim. When we plug in the exponential backoff batching schedule, i.e. $c_i = 2^i$ and $q_i = 2^k - 2^i$, the lemma follows.

The expression derived for the inverse speedup does not have a neat analytical form, but we can evaluate it for different values of d to obtain a diagram. As a sanity check, the worst expected speedup comes with $d = 0$. If there is only

a single expensive element in the range, then there is no way to parallelize execution – the expression gives us the speedup 1. We expect a better speedup as d grows – when there are more expensive elements, it is easier for the scheduler to stumble upon some of them. In fact, for $d = k$, with the conventions established in the proof, we get that the speedup is $\frac{1}{P} + (1 - \frac{1}{P}) \cdot \frac{c_0}{D}$. This means that when all the elements are expensive the proximity to the optimal speedup depends on the size c_0 of the first batch – the less elements in it, the better. Together with the fact that many applications have uniform workloads, this is also the reason why we advocate exponential backoff for which the size of the first batch is 1.

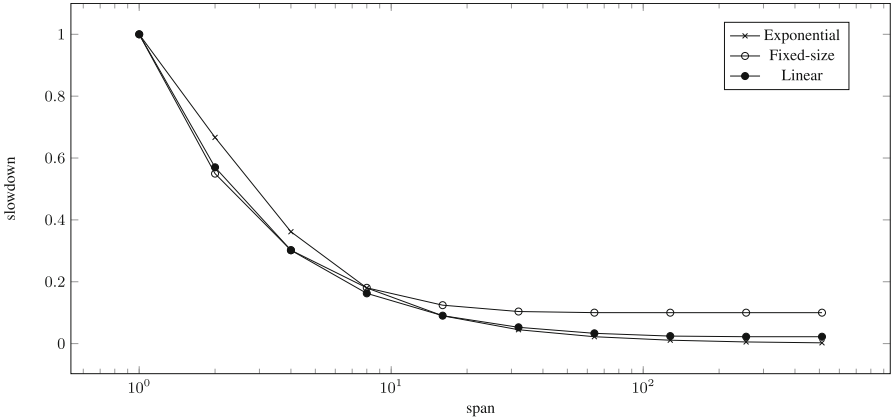


Fig. 17. Randomized scheduler executing step workload – speedup vs. span

We call the term $\frac{(q_0 - D)!}{q_0!} \sum_{i=0}^{k-1} c_i \cdot \frac{(q_i - 1)!}{(q_i - D)!}$ the *slowdown* and plot it with respect to span D on the diagram in Fig. 17. In this diagram we choose $k = 10$, and the number of elements $N = 2^{10} = 1024$. As the term nears 1, the speedup nears 1. As the term approaches 0, the speedup approaches the optimal speedup P . The quicker the term approaches 0 as we increase d , the better the scheduler. We can see that fixed-size batching should work better than the exponential backoff if the span D is below 10 elements, but is much worse than the exponential backoff otherwise. Linearly increasing the batch size from 0 in some step $a = \frac{2 \cdot (2^k - 1)}{k \cdot (k - 1)}$ seems to work well even for span $D < 10$. However, the mean batch size $\bar{c}_i = \frac{S}{k}$ means that this approach may easily violate the baseline constraint, and for $P \approx D$ the formula is an approximation anyway.

The conclusion is that selecting a random permutation of the elements should work very well in theory. For example, the average speedup becomes very close to optimal if less than $D = 10$ elements out of $N = 1024$ are expensive. However, randomly permuting elements would in practice either require a preparatory pass in which the elements are randomly copied or would require the workers to randomly jump through the array, leading to cache miss issues. In both cases the baseline performance would be violated. Even permuting the order of the

batches seems problematic, as it would require storing information about where each batch started and left off, as well as its intermediate result – for something like that we need a data structure like a work-stealing tree and we saw that we have to minimize the number of nodes there as much as possible.

There are many approaches we could study, many of which could have viable implementations, but we focus on a particular one which seems easy to implement for ranges and other data structures. Recall that in the example in Fig. 9-36 the interval with expensive elements was positioned at the end of the range. What if the worker alternated the batch in each step by tossing the coin to decide if the next batch should be from the left (start) or from the right (end)? Then the worker could arrive at the expensive interval on the end while the batch size is still small with a relatively high probability. The changes to the work-stealing tree algorithm are minimal – in addition to another field called `rresult` (the name of which should shed some light on the previous choice of name for `lresult`), we have to modify the `workOn`, `complete` and `pushUp` methods. While the latter two are straightforward, the lines 47 through 51 of `workOn` are modified. The new `workOn` method is shown in Fig. 18.

```
def workOn(ptr: Ptr): Boolean =
  val node = READ(ptr.child)
  var batch = -1
  var sum = 0
  do
    val p = READ(node.progress)
    if (notCompleted(p) && notStolen(p))
      if (coinToss())
        batchs = tryAdvanceLeft(node, p)
        if (notStolen(batch)) sum += kernel(p, p + decodeStep(batch))
      else
        batch = tryAdvanceRight(node, p)
        if (notStolen(batch)) sum += kernel(p, p + decodeStep(batch))
    else batch = -1
  while (batch ≠ -1)
  complete(sum, ptr)
```

Fig. 18. Randomized loop method

The main issue here is to encode and atomically update the iteration state, since it consists of two pieces of information – the left and the right position in the subrange. We can encode these two positions by using a long integer field and a long CAS operation to update it. The initial 32 bits can contain the position on the left side of the subrange and the subsequent 32 on the right side. With this in mind, the methods `tryAdvanceLeft`, `tryAdvanceRight`, `notStolen`, `notCompleted` and `decodeStep` should be straightforward.

We evaluate the new scheduler on the distribution from Fig. 9-36 and show the results in Fig. 19. The first two diagrams (STEP2 and STEP3) show that with the expensive interval at the beginning and the end of the range the work-stealing tree achieves a close to optimal speedup. However, there is still a worst case scenario that we have to consider, and that is to have a step workload with the expensive interval exactly in the middle of the range. Intuition tells us that

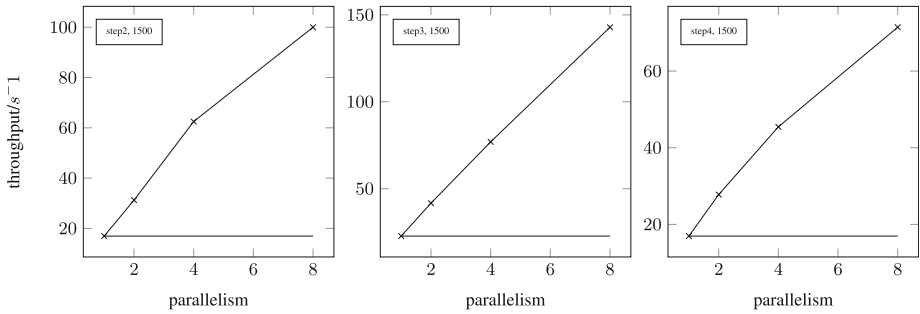


Fig. 19. The randomized work-stealing tree and the STEP3 workload

the probability to hit this interval early on is smaller, since a worker has to progress through more batches to arrive at it. The workload STEP4 in the third diagram of Fig. 19 contains around 25% expensive elements positioned in the middle of the range. The speedup is decent, but not linear for STEP4, since the bigger batches seem to on average hit the middle of the range more often.

Having shown that randomization does help scheduling both in theory and in practice, we conclude that the problem of overcoming particularly bad workload distributions is an algorithmic problem of finding a batching schedule which can be computed and maintained relatively quickly, leaving this task as future work.

References

1. Arora, N.S, Blumofe, R.D., Plaxton, C.G: Thread scheduling for multiprogrammed multiprocessors. In: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '98, pp. 119–129. ACM, New York (1998)
2. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput.* **37**(1), 55–69 (1996)
3. Buss, A., Harshvardhan, Papadopoulos, I., Pearce, O., Smith, T., Tanase, G., Thomas, N., Xu, X., Bianco, M., Amato, N.M., Rauchwerger, L.: STAPL: standard template adaptive parallel library. In: Proceedings of the 3rd Annual Haifa Experimental Systems Conference, SYSTOR '10, pp. 14:1–14:10. ACM, New York (2010)
4. Chakravarty, M.M.T., Leshchinskiy, R., Peyton Jones, S., Keller, G., Marlow, S.: Data parallel Haskell: a status report. In: Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP '07, pp. 10–18. ACM, New York (2007)
5. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R.R., Bradshaw, R., Weizenbaum, N.: FlumeJava: easy, efficient data-parallel pipelines. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, pp. 363–375. ACM, New York (2010)
6. Chase, D., Lev, Y.: Dynamic circular work-stealing deque. In: Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '05, pp. 21–28. ACM, New York (2005)

7. Cong, G., Kodali, S.B., Krishnamoorthy, S., Lea, D., Saraswat, V.A., Wen, T.: Solving large, irregular graph problems using adaptive work-stealing. In: ICPP, pp. 536–545 (2008)
8. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI '98, pp. 212–223. ACM, New York (1998)
9. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers, San Francisco (2008)
10. Hummel, S.F., Schonberg, E., Flynn, L.E.: Factoring: a method for scheduling parallel loops. *Commun. ACM* **35**(8), 90–101 (1992)
11. Intel Software Network. Intel Cilk Plus. <http://cilkplus.org/>
12. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley, Reading (1992)
13. Koelbel, C., Mehrotra, P.: Compiling global name-space parallel loops for distributed execution. *IEEE Trans. Parallel Distrib. Syst.* **2**(4), 440–451 (1991)
14. Kruskal, C.P., Weiss, A.: Allocating independent subtasks on parallel processors. *IEEE Trans. Softw. Eng.* **11**(10), 1001–1016 (1985)
15. Lea, D.: A java fork/join framework. In: Java Grande, pp. 36–43 (2000)
16. Polychronopoulos, C.D., Kuck, D.J.: Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.* **36**(12), 1425–1439 (1987)
17. Prokopec, A., Bagwell, P., Rompf, T., Odersky, M.: A generic parallel collection framework. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 136–147. Springer, Heidelberg (2011)
18. Reinders, J.: Intel Threading Building Blocks, 1st edn. O'Reilly & Associates, Sebastopol (2007)
19. Sarkar, V.: Optimized unrolling of nested loops. In: Proceedings of the 14th International Conference on Supercomputing, ICS '00, pp. 153–166. ACM, New York (2000)
20. Tardieu, O., Wang, H., Lin, H.: 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, pp. 267–276. ACM, New York (2012)
21. Tzen, T.H., Ni, L.M.: Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *IEEE Trans. Parallel Distrib. Syst.* **4**(1), 87–98 (1993)