JFP **22** (4–5): 382–438, 2012. © Cambridge University Press 2012
doi:10.1017/S0956796812000172

Lazy tree splitting

LARS BERGSTROM
Department of Computer Science, University of Chicago, Chicago, IL 60637, USA
(*e-mail:* larsberg@cs.uchicago.edu)

MATTHEW FLUET
Department of Computer Science, Rochester Institute of Technology, Rochester NY 14623-5603, USA
(*e-mail:* mtf@cs.rit.edu)

MIKE RAINEY*
Max Planck Institute for Software Systems, D-67663 Kaiserslautern, Rheinland-Phalz Germany
(*e-mail:* mrainey@mpi-sws.org)

JOHN REPPY
Department of Computer Science, University of Chicago, Chicago, IL 60637, USA
(*e-mail:* jhr@cs.uchicago.edu)

ADAM SHAW
Department of Computer Science, University of Chicago, Chicago, IL 60637, USA
(*e-mail:* ams@cs.uchicago.edu)

Abstract

Nested data-parallelism (NDP) is a language mechanism that supports programming irregular parallel applications in a declarative style. In this paper, we describe the implementation of NDP in Parallel ML (PML), which is a part of the Manticore system. One of the main challenges of implementing NDP is managing the parallel decomposition of work. If we have too many small chunks of work, the overhead will be too high, but if we do not have enough chunks of work, processors will be idle. Recently, the technique of Lazy Binary Splitting was proposed to address this problem for nested parallel loops over flat arrays. We have adapted this technique to our implementation of NDP, which uses binary trees to represent parallel arrays. This new technique, which we call *Lazy Tree Splitting* (LTS), has the key advantage of *performance robustness*, i.e., it does not require tuning to get the best performance for each program. We describe the implementation of the standard NDP operations using LTS and present experimental data that demonstrate the scalability of LTS across a range of benchmarks.

1 Introduction

Nested data-parallelism (NDP) (Blelloch *et al.*, 1994) is a declarative style for programming irregular parallel applications. NDP languages provide language features favoring the NDP style, efficient compilation of NDP programs, and various

* Portions of this work were completed while the author was affiliated with the University of Chicago.

common NDP operations like parallel maps, filters, and sum-like reductions. Irregular parallelism is achieved by the fact that nested arrays need not have *regular*, or rectangular, structure; i.e., sub-arrays may have different lengths. NDP programming is supported by a number of different parallel programming languages (Chakravarty *et al*., 2007; Ghuloum *et al*., 2007), including our own *Parallel ML* (PML) (Fluet *et al*., 2008a).

On its face, implementing NDP operations seems straightforward because individual array elements are natural units for creating *tasks*, which are small, independent threads of control.[1] Correspondingly, a simple strategy is to spawn off one task for each array element. This strategy is unacceptable in practice, as there is a scheduling cost associated with each task (e.g., the cost of placing the task on a scheduling queue) and individual tasks often perform only small amounts of work. As such, the scheduling cost of a given task might exceed the amount of computation it performs. If scheduling costs are too large, parallelism is not worthwhile.

One common way to avoid this pitfall is to group array elements into fixed-size chunks of elements and spawn a task for each chunk. *Eager Binary Splitting* (EBS), a variant of this strategy, is used by Intel's Thread Building Blocks (TBB) (Intel, 2008; Robison *et al*., 2008) and Cilk++ (Leiserson, 2009). Choosing the right chunk size is inherently difficult, as one must find the middle ground between undesirable positions on either side. If the chunks are too small, performance is degraded by the high costs of the associated scheduling and communicating. By contrast, if the chunks are too big, some processors go unutilized because there are too few tasks to keep them all busy.

One approach to picking the right chunk size is to use static analysis to predict task execution times and pick chunk sizes accordingly (Tick & Zhong, 1993). But this approach is limited by the fact that tasks can run for arbitrarily different amounts of time, and these times are difficult to predict in specific cases and impossible to predict in general. Dynamic techniques for picking the chunk size have the advantage that they can base chunk sizes on runtime estimates of system load. *Lazy Binary Splitting* (LBS) is one such chunking strategy for handling parallel `do-all` loops (Tzannes *et al*., 2010). Unlike the two aforementioned strategies, LBS determines chunks automatically and without programmer (or compiler) assistance and imposes only minor scheduling costs.

This paper presents an implementation of NDP that is based on our extension of LBS to binary trees, which we call *Lazy Tree Splitting* (LTS). LTS supports operations that produce and consume trees where tree nodes are represented as records allocated in the heap. We are interested in operations on trees because Manticore, the system that supports PML, uses *ropes* (Boehm *et al*., 1995), a balanced binary-tree representation of sequences, as the underlying representation of parallel arrays. Our implementation is purely functional as it works with immutable structures, although some imperative techniques are used under the hood for scheduling.

[1] We do not address *flattening* (or *vectorizing*) (Keller, 1999; Leshchinskiy, 2005) transformations here, since the techniques of this paper apply equally well to flattened or non-flattened programs.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

Lazy Tree Splitting exhibits *performance robustness*, i.e., it provides scalable parallel performance across a range of different applications and platforms without requiring any per-application tuning. Performance robustness is a highly desirable characteristic for a parallel programming language, for obvious reasons. Prior to our adoption of LTS, we used *Eager Tree Splitting* (ETS), a variation of EBS. Our experiments demonstrate that ETS lacks performance robustness: the tuning parameters that control the decomposition of work are very sensitive to the given application and platform. Furthermore, we demonstrate that the performance of LTS compares favorably to that of (ideally tuned) ETS across our benchmark suite.

This paper incorporates three substantial improvements to the material presented in the ICFP'10 paper of the same name. First, we identify a potential issue in our old approach where certain patterns of tree splitting can produce trees with arbitrary imbalance. We address this issue in Section 4 by presenting a new cursor-splitting technique and proving that the corresponding rope-processing codes are balance-preserving. Second, we present new benchmarking results from a larger, 48-core machine and demonstrate good scalability. Third, we present new experiments and examine the performance results in more depth.

2 Nested data-parallelism

In this section we give a high-level description of PML and discuss the runtime mechanisms we use to support NDP. More detail can be found in our previous papers (Fluet *et al.*, 2007a, 2007b, 2008a).

2.1 Programming model

Parallel ML is a programming language supported by the Manticore system.[2] Our programming model is based on a strict and mutation-free functional language (a subset of Standard ML (Milner *et al.*, 1997)), which is extended with support for multiple forms of parallelism. We provide fine-grain parallelism through several lightweight syntactic constructs that serve as hints to the compiler and runtime that the program may benefit from executing the computation in parallel. For this paper, we are primarily concerned with the NDP constructs, which are based on those found in NESL (Blelloch, 1990b, 1996).

Parallel ML provides a *parallel array*-type constructor (`parray`) and operations to map, filter, reduce, and scan these arrays in parallel. Like most languages that support NDP, PML includes comprehension syntax for maps and filters, but for this paper we omit the syntactic sugar and restrict ourselves the following interface:

[2] Manticore may support other parallel languages in the future.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

```
type 'a parray
val range   : int * int -> int parray
val mapP    : ('a -> 'b) -> 'a parray -> 'b parray
val filterP : ('a -> bool) -> 'a parray -> 'a parray
val reduceP : ('a * 'a -> 'a) -> 'a -> 'a parray -> 'a
val scanP   : ('a * 'a -> 'a) -> 'a -> 'a parray -> 'a parray
val map2P   : ('a * 'b -> 'c)
              -> ('a parray * 'b parray)
              -> 'c parray
```

The function `range` generates an array of the integers between its two arguments. The function `mapP` applies a function to all the elements of a `parray` in parallel. `filterP` applies a predicate in parallel over the input `parray` to produce a new `parray` containing only those elements corresponding to a true result from the predicate. The function `reduceP` takes a binary operator along with an identity value and applies the operator in parallel to the values in the `parray` until reaching a final result value. The function `scanP` produces a parallel prefix scan of the array. Both `reduceP` and `scanP` assume that the binary operation is associative. Finally, the function `map2P` applies a function to pairs of elements of two `parrays` in parallel; the output array has the length of the shorter input array. These parallel-array operations have been used to specify both SIMD parallelism that is mapped onto vector hardware (e.g., Intel's SSE instructions) and SPMD parallelism, where parallelism is mapped onto multiple cores; this paper focuses on exploiting the latter.

As a simple example, the main loop of a ray tracer generating an image of width `w` and height `h` can be written

```
fun raytrace (w, h) =
  mapP (fn y => mapP (fn x => trace (x, y))
                     (range (0,w-1)))
       (range (0,h-1))
```

This parallel map within a parallel map is an example of *nested data-parallelism.* Note that the time to compute one pixel depends on the layout of the scene, because the ray cast from position `(x,y)` might pass through a subspace that is crowded with reflective objects or it might pass through relatively empty space. Thus, the amount of computation performed by the `trace(x,y)` expression (and, therefore, performed by the inner `mapP` expression) might differ significantly depending on the layout of the scene. The main contribution of this paper is a technique for balancing the parallel execution of such irregular parallel programs in functional programming languages with ropes.

2.2 Runtime model

The Manticore runtime system consists of a small core written in C, which implements a processor abstraction layer, garbage collection, and a few basic scheduling primitives. The rest of our runtime system is written in BOM, a PML-like language. BOM supports several mechanisms, such as first-class continuations and mutable data structures, that are useful for programming schedulers but are not in

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

PML. Further details on our system may be found elsewhere (Rainey, 2007; Fluet *et al.*, 2008b; Rainey, 2009).

A task scheduling policy determines the order in which tasks execute and the mappings from tasks to processors. Our LTS is built on top of a particular task scheduling policy called *work stealing* (Burton & Sleep, 1981; Halstead Jr., 1984). In work stealing, we employ a group of workers, one per processor, that collaborate on a given computation. The idea is that idle workers which have no useful work to do bear most of the scheduling costs and busy workers which have useful work to do focus on finishing that work.

We use the following well-known implementation of work stealing (Frigo *et al.*, 1998; Blumofe & Leiserson, 1999). Each worker maintains a double-ended queue (deque) of tasks, represented as thunks. When a worker reaches a point of potential parallelism in the computation, it pushes a task for one independent branch onto the bottom of the deque and continues executing the other independent branch. Upon completion of the executed branch, it pops a task off the bottom of the deque and executes it. If the deque is not empty, then the task is necessarily the most recently pushed task; otherwise all of the local tasks have been stolen by other workers and the worker must steal a task from the top of some other worker's deque. Potential victims are chosen at random from a uniform distribution.

This work-stealing scheduler can be encapsulated in the following function, which is part of the runtime system core:

```
val par : (unit -> 'a) * (unit -> 'b) -> 'a * 'b
```

When a worker P executes `par (f,g)`, it pushes the task g onto the bottom of its deque[3] and then executes `f()`. When the computation of `f()` completes with result r_f, P attempts to pop g from its deque. If successful, then P will evaluate `g()` to a result r_g and return the pair (r_f, r_g). Otherwise, some other worker Q has stolen g, so P writes r_f into a shared variable and looks for other work to do. When Q finishes the evaluation of `g()`, then it will pass the pair of results to the return continuation of the `par` call. The scheduler also provides a generalization of `par` to a list of thunks.

```
val parN : (unit -> 'a) list -> 'a list
```

This function can be defined in terms of `par`, but we use a more efficient implementation that pushes all of the tasks in its tail onto the deque at once.

2.3 Ropes

In the Manticore system, we use ropes as the underlying representation of parallel arrays. Ropes, originally proposed as an alternative to strings, are persistent balanced binary trees with `seqs`, contiguous arrays of data, at their leaves (Boehm *et al.*, 1995). For the purposes of this paper, we define the rope type as follows:

[3] Strictly speaking, it pushes a continuation that will evaluate `g()`.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

```
datatype 'a crope
  = CLeaf of 'a * 'a seq
  | CCat of 'a * 'a crope * 'a crope
```

However, in our actual implementation there is extra information in the `Cat` nodes to support balancing. Read from left to right, the data elements at the leaves of a rope constitute the data of the parallel array it represents.

Since ropes can be physically dispersed in memory, they are well suited to being built in parallel, with different processors simultaneously working on different parts of the whole. Furthermore, the rope data structure is persistent, which provides, in addition to the usual advantages of persistence, two special advantages related to memory management. First, we can avoid the cost of store-list operations (Appel, 1989), which are sometimes necessary for maintaining an ephemeral data structure. Second, a parallel memory manager, such as the one used by Manticore (Fluet *et al.*, 2008b), can avoid making memory management a sequential bottleneck by letting processors allocate and reclaim sub-ropes independently.

As a parallel-array representation, ropes have several weaknesses when compared to contiguous arrays of, say, unboxed doubles. First, rope random access requires logarithmic time. Second, keeping ropes balanced requires extra computation. Third, mapping over multiple ropes is more complicated than mapping over multiple arrays, since the ropes can have different shapes. In our performance study in Section 5, we find that these weaknesses are not a limitation in practice and we know of no study in which NDP implementations based on ropes are compared side by side with implementations based on alternative representations, such as contiguous arrays.

The maximum length of the linear sequence at each leaf of a rope is controlled by a compile-time constant M. At runtime, a leaf contains a number of elements n such that $0 \leqslant n \leqslant M$. In general, rope operations try to keep the size of each leaf as close to M as possible, although some leaves will necessarily be smaller. We do not demand that a rope maximize the size of its leaves.

Requiring perfect balance of all ropes can lead to excessive rebalancing, because even a small change to a given rope can make the rope unbalanced. Thus, we use a different balancing policy that still maintains the asymptotic behavior of rope operations but where ropes are allowed to become slightly unbalanced. For a given rope r of depth d and length n, our relaxed balancing goal is $d \leqslant \lceil \log_2 n \rceil + 2$. This property is guaranteed by the function

```
val balance : 'a rope -> 'a rope
```

which takes a rope r and returns a balanced rope equivalent to r (returning r itself if it is already balanced). This function uses a simple parallel balancing algorithm that executes in time $O(n)$ on a single processor and $O(d^2)$ time on an unbounded number of processors. The idea is to repeatedly split the given rope into two halves of equal size, recursively balance each half in parallel, and to concatenate the two balanced sub-ropes. The base case occurs when the length of the given rope falls below M, in which case the algorithm serially flattens the rope to create a single leaf node.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

As noted above, rope operations try to keep the size of each leaf as close to M as possible. To build ropes, rather than using the `Cat` constructor directly, we define a specialized constructor:

```
val cat : 'a rope * 'a rope -> 'a rope
```

If `cat` is applied to two small leaves, it can coalesce them into a single larger leaf. Note that `cat` does not guarantee balance, although it will maintain balance if applied to two balanced ropes of equal size. We also define a similar function

```
val catN : 'a rope list -> 'a rope
```

which returns the smart concatenation of its argument ropes.

We sometimes need a fast, cheap operation for splitting a rope into multiple sub-ropes. For this reason, we provide

```
val split : 'a rope -> 'a rope * 'a rope
```

which splits its rope argument into two sub-ropes such that the sizes of these ropes differ by at most one. We also define

```
val splitN : 'a rope * int -> 'a rope list
```

which splits its parameter into n sub-ropes, where each sub-rope has the *same* size, except for one sub-rope that might be smaller than the others.

We sometimes use

```
val length : 'a rope -> int
```

which returns the number of elements stored in the leaves of a rope and

```
val size : 'a rope -> int
```

which returns the number of leaves of a rope.[4]

The various parallel-array operations described in Section 2.1 are implemented by analogous operations on ropes. Sections 3 and 4 describes the implementation of these rope-processing operations in detail.

3 The Goldilocks problem

In NDP programs, computations are divided into chunks, and chunks of work are spawned in parallel. These chunks might be defined by subsequences (of arrays, for example, or, in our case, ropes) or iteration spaces (say, k to some $k + n$). The choice of chunk size influences performance crucially. If the chunks are too small, there will be too much overhead in managing them; in extreme cases, the benefits of parallelism will be obliterated. On the other hand, if they are too large, there will not be enough parallelism, and some processors may run out of work. An ideal chunking policy apportions chunks that are neither too large nor too small, but are, like Goldilocks's third bowl of porridge, "just right." Some different chunking policies are considered in the sequel.

[4] In our actual implementation, these operations are constant time, as we cache lengths and sizes in `Cat` nodes.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

```
fun mapTary J f rp = let
  fun g chunk = fn () => mapSequential f chunk
  val n = length rp
  val chunks = splitN (rp, min (n, J * numProcs ()))
  in
    catN (parN (List.map g chunks))
  end
```

(a) T-ary decomposition

```
fun mapStructural f rp = (case rp
  of Leaf s => Leaf (mapSeq f s)
   | Cat (l, r) =>
       Cat (par (fn () => mapStructural f l,
                 fn () => mapStructural f r)))
```

(b) structural decomposition

Fig. 1. Two fragile implementations of the map operation.

3.1 Fragile chunking policies

A fragile chunking policy is prone either to creating an excessive number of tasks or to missing significant opportunities for parallelism. Let us consider two simple policies, T-ary decomposition and structural decomposition, and the reasons that they are fragile. In T-ary decomposition, we split the input rope into $T = \min(n, J \times P)$ chunks, where n is the size of the input rope, J is a fixed compile-time constant, and P is the number of processors, and spawn a task for each chunk. For example, in Figure 1(a), we show the T-ary decomposition version of the map operation.[5] In computations where all rope elements take the same time to process, such as those performed by regular affine (dense-matrix) scientific codes, the T-ary decomposition will balance the work load evenly across all processors because all chunks will take about the same amount of time. On the other hand, when rope elements correspond to varying amounts of work, performance will be fragile because some processors will get overloaded and others underutilized. Excessive splitting is also a problem. Observe that if a program creates i levels of `mapTary` applications and if each rope has length $n \geqslant J \times P$, then the T-ary decomposition creates $(J \times P)^i$ tasks at the leaves alone, which can be excessive when either i or P get large.

To remedy the imbalance problem, we might try structural decomposition, in which both children of a `Cat` node are processed in parallel and the elements of a `Leaf` node are processed sequentially. We show the structural version of the map operation in Figure 1(b). Recall that the maximum size of a leaf is determined by a fixed, compile-time constant called M and that rope-producing operations tend to keep the size of each leaf close to M. But by choosing an $M > 1$, some opportunities for parallelism will always be lost and by choosing $M = 1$, an excessive number of threads may be created, particularly in the case of nested loops.

[5] In this and subsequent examples, we use
val mapSequential : $('a -{>}' b) -> 'a$ rope $-{>}' b$ rope
which is the obvious sequential implementation of the map operation.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

```
fun mapETS SST f rp =
  if length rp <= SST then mapSequential f rp
  else let
    val (l, r) = split rp
    in
      cat (par (fn () => mapETS SST f l,
                fn () => mapETS SST f r))
    end
```

Fig. 2. The ETS implementation of the map operation.

3.2 Eager binary splitting

Eager Binary Splitting is a well-known approach that is used by many parallel libraries and languages, including Threading Building Blocks (Intel, 2008; Robison *et al.*, 2008)[6] and Cilk++ (Leiserson, 2009). In EBS (and, by extension, ETS), we group elements into fixed-size chunks and spawn a task for each chunk. This grouping is determined by the following recursive process. Initially, we group all elements into a single chunk. If the chunk size is less than the stop-splitting threshold (SST), evaluate the elements sequentially.[7] Otherwise, we create two chunks by dividing the elements in half and recursively apply the same process to two new chunks. In Figure 2, we show the ETS version of the map operation.

EBS has greater flexibility than the T-ary or structural decompositions because EBS enables chunk sizes to be picked manually. But this flexibility is not much of an improvement, because, as is well known (Intel, 2008; Robison *et al.*, 2008; Tzannes *et al.*, 2010), finding a satisfactory SST can be difficult. This difficulty is due, in part, to the fact that parallel speedup is very sensitive to SST. We ran an experiment that demonstrates some of the extent of this sensitivity. Figure 3 shows, for seven PML benchmarks (see Section 5 for benchmark descriptions), parallel speedup as a function of SST. The results demonstrate that there is no SST that is optimal for every program and furthermore that a poor SST is far from optimal.

The Raytracer benchmark demonstrates, in particular, how fragile ETS can be with respect to nesting and to relatively small ropes. Raytracer loses all of its speedup as SST is changed from 2^6 to 2^9. To understand why, first note that the two-dimensional output of the program is a $2^9 \times 2^9$ rope of ropes, representing the pixels of a square image. When, for instance, $SST = 2^7$, Raytracer has just 16 chunks that it can process in parallel: four for each row and four for each column, and when $SST \geqslant 2^9$, Raytracer has just one chunk it can process at a time (no parallelism). We could address this problem by transforming the two-dimensional representation into a single flat rope, but then the clarity of the code would be compromised, as we would have to use index arithmetic to extract any pixel. As a rule, our compiler should not encourage programmers to break with NDP style to achieve best performance.

[6] In the TBB manual, the option "simple partitioner" refers to EBS.
[7] In TBB, if SST is unspecified, the default is $SST = 1$, whereas Cilk++ only uses $SST = 1$.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

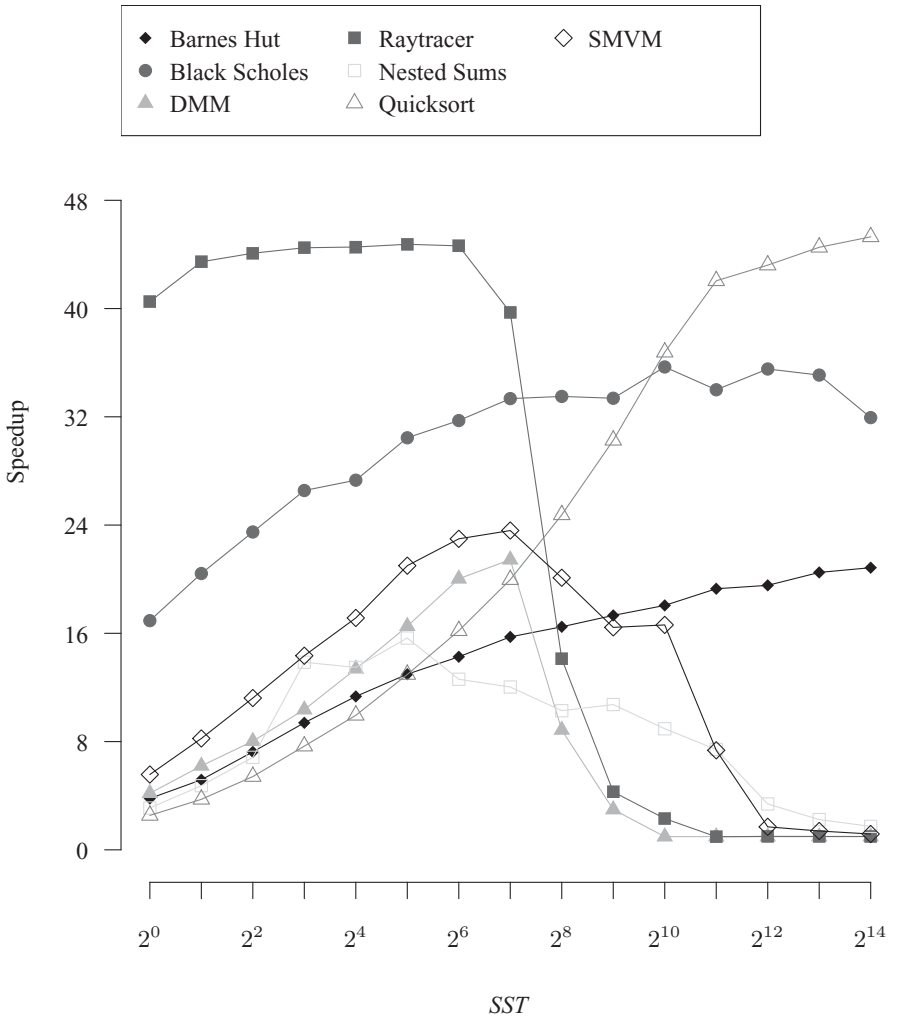


Fig. 3. Parallel speedup as a function of the Stop-Splitting Threshold (*SST*) (48 processors).

Recall that task execution times can vary unpredictably. Chunking policies that are based solely on fixed thresholds, such as EBS and ETS, are bound to be fragile because they rely on accurately predicting execution times. A superior chunking policy would be able to adapt dynamically to the current load across processors.

3.3 Lazy binary splitting

The LBS policy of Tzannes *et al.* (2010) is a promising alternative to the other policies because it dynamically balances load. Tzannes *et al.* show that LBS is capable of performing as well or better than each configuration of eager binary splitting, and does so without tuning.

Lazy Binary Splitting is similar to eager binary splitting but with one key difference. In LBS, we base each splitting decision entirely on a dynamic estimation

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

of load balance. Let us consider the main insight behind LBS. We call a processor hungry if it is idle and ready to take on new work, and busy otherwise. It is better for a given processor to delay splitting a chunk and to continue processing local iterations while remote processors remain busy. Splitting can only be profitable when a remote processor is hungry.

Although this insight is sound, it is still unclear whether it is useful. A naïve hungry-processor check would require inter-processor communication, and the cost of such a check would hardly be an improvement over the cost of spawning a thread. For now, let us assume that we have a good approximate hungry-processor check

```
val hungryProcs : unit -> bool
```

which returns `true` if there is probably a remote hungry processor and `false` otherwise. Later we explain how to implement such a check.

Lazy Binary Splitting works as follows. The scheduler maintains a current chunk c and a pointer i that points at the next iteration in the chunk to process. Initially, the chunk contains all iterations and $i = 0$. To process an iteration i, the scheduler first checks for a remote hungry processor. If the check returns false, then all of the other processors are likely to be busy, and the scheduler greedily executes the body of iteration i. If the check returns true, then some of the other processors are likely to be hungry, and the scheduler splits the chunk in half and spawns a recursive instance to process the second half.

Tzannes *et al.* (2010) show how to implement an efficient and accurate hungry-processor check. Their idea is to derive such a check from the work stealing policy. Recall that, in work stealing, each processor has a deque, which records the set of tasks created by that processor. The hungry-processor check bases its approximation on the size of the local deque. If the deque of a given processor contains some existing tasks, then these tasks have not yet been stolen, and therefore it is unlikely to be profitable to add to these tasks by splitting the current chunk. On the other hand, if the deque is empty, then it is a strong indication that there is a remote hungry processor, and it is probably worth splitting the current chunk. This heuristic works surprisingly well considering its simplicity. It is cheap because the check itself requires two local memory accesses and a compare instruction, and it provides an estimate that our experiments have shown to be accurate in practice.

Let us consider how LBS behaves with respect to loop nesting. Suppose our computation has the form of a doubly nested loop, one processor is executing an iteration of the inner loop, and all other processors are hungry. Consequently, the remainder of the inner loop will be split (possibly multiple times, as work is stolen from the busy processor and further split), generating relatively small chunks of work for other processors. Since the parallelism is fork-join, the only way for the computation to proceed to the next iteration of the outer loop is for all of the work from the inner loop to be completed. At this point, all processors are hungry, except for the one processor that completed the last bit of inner-loop work. This processor has an empty deque; hence, when it starts to execute the next iteration of the outer loop, it will split the remainder of the outer loop.

Since there is one hungry-processor check per loop iteration, and because loops are nested, most hungry-processor checks occur during the processing of the innermost

loops. Thus, the general pattern is clear: splits tend to start during inner loops and then move outward quickly.

4 Lazy tree splitting

Lazy tree splitting operations are not as easy to implement as ETS operations, because during the execution of any given LTS operation a split can occur while processing *any* rope element. This section presents implementations of five important LTS operations. The implementations we use are based on Huet's zipper technique (Huet, 1997) and a new technique we call *cursor splitting*. We first look in detail at the LTS version of map (`mapLTS`), because its implementation provides a simple survey of our techniques. We then summarize implementations of additional operations.

4.1 Implementing `mapLTS`

Structural recursion on its own gives no straightforward way to implement `mapLTS`. Consider the case in which `mapLTS` detects that another processor is hungry. How can `mapLTS` be ready to halve the as-yet-unprocessed part of the rope, keeping in mind that, at the halving moment, the focus might be on a mid-leaf element deeply nested within a number of `Cat` nodes? In a typical structurally recursive traversal (e.g., Figure 1(b)), the code has no explicit handle on either the processed portion of the rope or the unprocessed remainder of the rope; it can only see the current substructure. An implementation needs to be able to step through a traversal in such a way that it can, at any moment, pause the traversal, reconstruct both the processed results and the unprocessed remainder, divide the unprocessed remainder in half, and resume processing at the pause point.

An implementation of `mapLTS` should also be *balance-preserving*, meaning that a balanced input rope is mapped to a balanced output rope. Without balance preservation, chains of `mapLTS` applications can, under the right circumstances, yield ropes that are arbitrarily unbalanced. While it may at first appear that balanced ropes are unnecessary, since the structure of the rope is not used to guide the creation of parallel computations, balance is nonetheless important to guarantee an efficient algorithm for dividing the unprocessed remainder of a paused traversal. In fact, we will demonstrate a stronger property: that our implementation of `mapLTS` is *shape-preserving*, meaning that an input rope is mapped to an output rope with exactly the same shape. Hence, throughout the following and in Appendix A, equalities in properties, lemmas, theorems, and proofs denote structural equality of objects. Note that shape preservation implies balance preservation.

4.1.1 Cursor interface

A key component of our approach is a data structure called a *cursor*, which represents an intermediate step of a map computation,[8]

[8] We name the type constructor `map_cur` because other rope operations require a different type of cursor; see Section 4.2.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

```
type ('b, 'a) map_cur
```

The cursor records the sub-ropes that have been processed so far, the sub-ropes that remain to be processed, and enough information so that the exact tree structure of the corresponding rope can always be recovered. In the cursor, `'b` is the type of the elements of the processed sub-ropes and `'a` is the type of the elements of the unprocessed sub-ropes. Conceptually, a cursor describes a point in the rope with processed elements to the left and unprocessed elements to the right. In Section 4.1.3, we will see that cursors are implemented using techniques similar to Huet's zippers (Huet, 1997) and McBride's contexts (McBride, 2008).

Let us introduce a few simple operations over cursors so that we can describe the sequential part of `mapLTS`. The `root` operation returns the rope corresponding to the given cursor for the special case that the types of the unprocessed and processed elements are the same.

```
val root : ('b, 'b) map_cur -> 'b rope
```

Since ropes are homogenous with respect to their element type, it is not possible to obtain a rope from a cursor when the types of the unprocessed and processed elements are different.

The `lengthRight` operation returns the number of unprocessed data elements of the given cursor, which we consider to be to the right of the cursor's focus.

```
val lengthRight : ('b, 'a) map_cur -> int
```

Since a cursor represents an intermediate step of map computation with both processed and unprocessed elements, it must be possible to split a cursor into processed elements and two ropes of unprocessed elements and to later join two ropes together with the processed elements. The `split` and `join` operations provide this behavior,

```
val split : ('b, 'a) map_cur
            -> ('a rope * 'a rope * 'b map_cur_reb)

val join  : ('a rope * 'a rope * 'b map_cur_reb)
            -> ('b, 'a) map_cur
```

The call `split cur` returns `(rp1, rp2, reb)`, where `rp1` and `rp2` are ropes such that the rope `rp1` contains the first half of the unprocessed data elements of `cur` and rope `rp2` contains the remaining unprocessed data elements of `cur`, and `reb` is a special *rebuilder* value. For the time being, we use `map_cur_reb` as an abstract-type constructor without a specific implementation,

```
type 'b map_cur_reb
```

This rebuilder value records sufficient information so that the original cursor `cur` can be reconstructed by the `join` operation. The call `join (rp1, rp2, reb)` rebuilds the cursor `cur` that is uniquely determined by its three arguments.

To prove that our `mapLTS` implementation is shape-preserving, we will rely on the implementations of `split` and `join` to be well behaved, as expressed by the following property:

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

Property 1 (`split` and `join` are well-behaved). For any cursor `cur`, if `split cur` returns `(rp1, rp2, reb)`, then

```
join (rp1, rp2, reb)  =  cur
```

and

```
length rp1  =  (lengthRight cur) div 2
```

and

```
length rp2
   =  (lengthRight cur) - ((lengthRight cur) div 2)
```

4.1.2 mapLTS implementation

We factor the implementation of `mapLTS` into a coordination portion, which is responsible for introducing parallelism by splitting and joining cursors, and a computation portion, which is responsible for performing the mapping computation and stepping through intermediate cursors. This computational portion of `mapLTS` is provided by an auxiliary operation named `mapLTSUntil`, which is, in addition, capable of pausing its traversal based on the results of a runtime predicate,

```
val mapLTSUntil : (unit -> bool)
                  -> ('a -> 'b)
                  -> 'a rope
                  -> (('b, 'a) map_cur, 'b rope) progress
```

The first argument to `mapLTSUntil` is a polling function (e.g., `hungryProcs`); the second argument is a function to be applied to the individual data elements; and the third argument is a rope. The result of `mapLTSUntil` is a value of type `(('a, 'b) map_cur, 'b rope) progress`, where the `progress` type constructor[9] is defined as

```
datatype ('m, 'd) progress
  = More of 'm
  | Done of 'd
```

When `mapLTSUntil` returns a value `More cur'`, it represents the intermediate cursor when `mapLTSUntil` was paused, and when it returns a `Done rp'`, it represents the fully processed rope. The evaluation of `mapLTSUntil cond f rp` proceeds by applying `f` to the elements of `rp` from left to right until either `cond ()` returns `true` or the whole rope is processed.

To prove that our `mapLTS` implementation is shape-preserving, we will rely on the implementation of `mapLTSUntil` to be well behaved. Primarily, we require that `mapLTSUntil` preserves the shape of the input rope. We also require that `mapLTSUntil` only pauses and returns a new cursor when the number of unprocessed elements of the result cursor is less than or equal to that of the input rope and is greater than or equal to two. We require this behavior for two reasons. First, for termination, we require the number of unprocessed elements of the result cursor

[9] The `progress`-type constructor is used elsewhere in the implementation at different types, which motivates its polymorphic definition.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

```
fun mapLTS f rp =
  (case mapLTSUntil hungryProcs f rp
    of Done rp' => rp'
     | More cur' => let
         val (rp1, rp2, reb) = split cur'
         val (rp1', rp2') =
               par (fn () => mapLTS f rp1,
                    fn () => mapLTS f rp2)
         in
           root (join (rp1', rp2', reb))
         end)
```

Fig. 4. The LTS implementation of the map operation.

to be less than or equal to that of the input rope and to be greater than or equal to two so that splitting this cursor yields non-empty ropes that are strictly smaller than the input rope; this avoids the need for extraneous base cases. Second, for performance, we note that it is not worthwhile to pause execution if there are fewer than two unprocessed elements. In that case there is no opportunity for parallelism, and, as such, it is better to simply finish the map computation with a sequential execution. Although this second requirement seems unrelated to shape preservation, it is necessary to require this behavior to prove that the implementation that we give for `mapLTS` is shape-preserving. These requirements are expressed by the following property.

Property 2 (`mapLTSUntil` is well-behaved). For any rope `rp` and any predicate `cond`, if `mapLTSUntil cond (fn x => x) rp` returns `Done rp'`, then

$$\texttt{rp'} \;=\; \texttt{rp}$$

and if it returns `More cur'`, then

$$\texttt{root cur'} \;=\; \texttt{rp}$$

and

$$\texttt{length rp} \;\geqslant\; \texttt{lengthRight cur'}$$

and

$$\texttt{lengthRight cur'} \;\geqslant\; 2$$

Figure 4 gives our implementation of `mapLTS`. The `mapLTS` function attempts to complete its given map computation sequentially by calling `mapLTSUntil` on the rope `rp`. If the call to `mapLTSUntil` returns `Done rp'`, then the rmap computation is complete and `mapLTS` returns the result rope `rp'`. Otherwise, if `mapLTSUntil` returns `More cur'`, then `mapLTS` splits the remaining map computation in half (using `split`), recursively processes the two halves in parallel (using `par`), and joins the recursive results together (using `join`). The result of `mapLTS` is the rope obtained by applying `root` to the result cursor from the `join` operation.

Using our previously stated properties, we can prove that this implementation of `mapLTS` is *shape-preserving*.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

Theorem 1 (`mapLTS` is shape preserving). For any rope `rp`,

```
mapLTS (fn x => x) rp  =  rp
```

Proof
The proof is by strong induction on `length rp`, using Properties 1 and 2.
See Appendix A.4 for a detailed proof. □

It remains to implement the cursor type, the `mapLTSUntil` operation, and the `split` and `join` operations.

4.1.3 Cursor implementation

The crucial property of `mapLTSUntil` is that during the traversal of the input rope, it must maintain sufficient information to pause the traversal at any moment and reconstruct both the processed portion of the rope and the unprocessed remainder of the rope. Huet's zipper technique (Huet, 1997) provides the insight necessary to derive a persistent cursor data structure and functional operations over it, which enable this "pausable" traversal. A zipper is a representation of an aggregate data structure that factors the data structure into a distinguished substructure under focus and a one-hole context; plugging the substructure into the context yields the original structure. Zippers enable efficient navigation through and modification of a data structure. With a customized zipper representation and some basic navigation operations we arrive at an elegant implementation of `mapLTSUntil`.

To represent the cursor, we use a context representation similar to Huet's single-hole contexts (Huet, 1997), but with different types of elements on either side of the hole, as in McBride's contexts (McBride, 2008). Essentially, a context describes a path through a rope from the root to a particular sub-rope, while also recording the sub-ropes that branch off of this path; sub-ropes branching off to the left are processed, while sub-ropes branching off to the right are unprocessed. Thus, our context representation is defined as

```
datatype ('b, 'a) map_ctx
  = MCTop
  | MCLeft of ('b, 'a) map_ctx * 'a rope
  | MCRight of 'b rope * ('b, 'a) map_ctx
```

where `MCTop` represents an empty context, `MCLeft(ctx, rrp)` represents the context surrounding the left branch of a `Cat` node where `rrp` is the right branch and `ctx` is the context surrounding the `Cat` node, and `MCRight(lrp, ctx)` represents the context surrounding the right branch of a `Cat` node where `lrp` is the left branch and `ctx` is the context surrounding the `Cat` node. For a map computation, all sub-ropes to the left of the context's hole are processed (`'b rope`) and all sub-ropes to the right of the context's hole are unprocessed (`'a rope`). Given this context type, we define the cursor type as

```
type ('b, 'a) map_cur = ('b seq * 'a seq) * ('b, 'a) map_ctx
```

where the first element of the pair is the leaf located at the cursor, itself split into a sequence of processed and unprocessed elements, and the second element is the context surrounding the leaf.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

The implementations of `mapLTS` and `mapLTSUntil` require a number of operations to manipulate cursors and contexts. The `plug (rp, ctx)` operation plugs the rope `rp` into the context `ctx` for the special case that the types of the unprocessed and processed elements of the context are the same:

```
val plug : 'b rope * ('b, 'b) map_ctx -> 'b rope
fun plug (rp, ctx) = (case ctx
  of MCTop => rp
   | MCLeft (ctx', rrp) => plug (Cat (rp, rrp), ctx')
   | MCRight (lrp, ctx') => plug (Cat (lrp, rp), ctx'))
```

The `root ((pseq,useq), ctx)` operation, which returns the rope corresponding to a given cursor, simply reconstructs a leaf rope from the sequences `pseq` and `useq` and plugs the rope into the context `ctx`:

```
val root : ('b, 'b) map_cur -> 'b rope
fun root ((pseq, useq), ctx) =
  plug (Leaf (joinSeq (pseq, useq)), ctx)
```

The `leftmost (rp, ctx)` operation navigates to the leftmost leaf of `rp` and returns `(seq', ctx')`, the sequence `seq'` at that leaf and the context `ctx'` surrounding that leaf, as composed with the context `ctx`:

```
val leftmost : 'a rope * ('b, 'a) map_ctx
               -> 'a seq * ('b, 'a) map_ctx
fun leftmost (rp, ctx) = (case rp
  of Leaf seq => (seq, ctx)
   | Cat (lrp, rrp) => leftmost (lrp, MCLeft (ctx, rrp)))
```

We measure the lengths of context and cursor as the pair of the number of processed elements and the number of unprocessed elements:

```
infix 6 ++
fun (a1, b1) ++ (a2, b2) = (a1 + a2, b1 + b2)

val ctxLength : ('b, 'a) map_ctx -> int * int
fun ctxLength ctx = (case ctx
  of MCTop => (0, 0)
   | MCLeft (ctx', rrp) => (ctxLength ctx') ++ (0, length rrp)
   | MCRight (lrp, ctx') => (ctxLength ctx') ++ (length lrp, 0))

val curLength : ('b, 'a) map_cur -> int * int
fun curLength ((pseq,useq), ctx) =
  (ctxLength ctx) ++ (lengthSeq pseq, lengthSeq useq)
```

The `lengthRight` operation simply extracts the number of unprocessed elements from the length of a given cursor:

```
fun lengthRight cur = snd (curLength cur)
```

Similarly, we measure the size of context and cursor as the pair of the number of processed leaves and the number of unprocessed leaves:

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

```
val ctxSize : ('b, 'a) map_ctx -> int * int
fun ctxSize ctx = (case ctx
  of MCTop => (0, 0)
   | MCLeft (ctx', rrp) => (ctxSize ctx') ++ (0, size rrp)
   | MCRight (lrp, ctx') => (ctxSize ctx') ++ (size lrp, 0))

val curSize : ('b, 'a) map_cur -> int * int
fun curSize ((pseq,useq), ctx) =
  (ctxSize ctx) ++ (1, 1)
```

The `next (rp, ctx)` operation plugs the (processed) rope `rp` into the context `ctx`, then attempts to navigate to the next unprocessed leaf.

```
val next : 'b rope * ('b, 'a) map_ctx
           -> ('a seq * ('b, 'a) map_ctx, 'b rope) progress
fun next (rp, ctx) =
  (case ctx
    of MCTop => Done rp
     | MCLeft (ctx', rrp) => let
         val (seq'', ctx'') = leftmost (rrp, MCRight (rp, ctx'))
         in
           More (seq'', ctx'')
         end
     | MCRight (lrp, ctx') =>
         next (Cat (lrp, rp), ctx'))
```

This navigation can either succeed, in which case `next` returns `More (seq', ctx')` (see Figure 5(a)), where `seq'` is the sequence at the next leaf and `ctx'` is the context surrounding that leaf, or fail, in which case `next` returns `Done rp'` (see Figure 5(b)), where `rp'` is the whole processed rope.

4.1.4 *mapLTSUntil implementation*

With these context operations, we give the implementation of `mapLTSUntil` in Figure 6. The traversal of `mapLTSUntil` is performed by the auxiliary function `m`. The argument `seq` represents the sequence of the leftmost unprocessed leaf of the rope and the argument `ctx` represents the context surrounding that leaf.

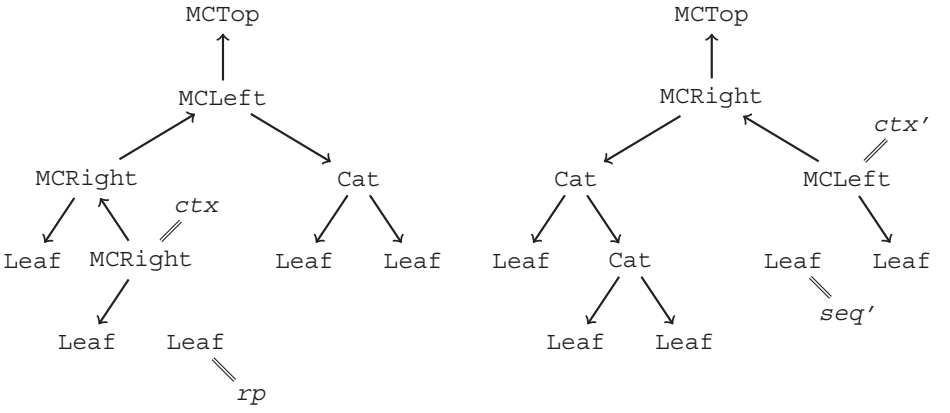
The processing of the sequence is performed by `mapUntilSeq`, a function with similar behavior to `mapLTSUntil`, but implemented over linear sequences

```
val mapUntilSeq : (unit -> bool)
                  -> ('a -> 'b)
                  -> 'a seq
                  -> ('a seq * 'b seq, 'b seq) progress
```

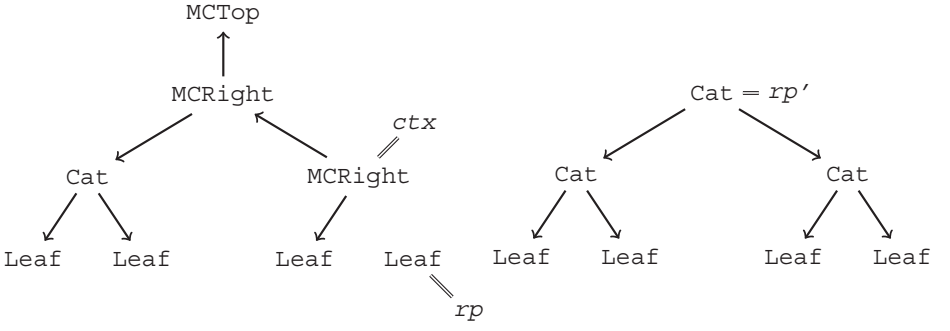
It is `mapUntilSeq` that actually calls the predicate `cond` and applies the function `f`. Note that `mapUntilSeq` must also maintain a context with processed elements to the left and unprocessed elements to the right, but doing so is trivial for a linear sequence. (Recall the standard accumulate-with-reverse implementation of `map` for lists.) Not surprisingly, we require that `mapUntilSeq` preserves the shape of the sequence as expressed by the following property.

Property 3 (`mapUntilSeq` is shape preserving). For any sequence `seq` and any predicate `cond`, if

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press



(a) next (*rp*, *ctx*) ⇒ More(*seq'*, *ctx'*)



(b) next (*rp*, *ctx*) ⇒ Done *rp'*

Fig. 5. Operations on contexts.

```
mapUntilSeq cond (fn x => x) seq
```

returns `Done seq'`, then

```
seq'  =  seq
```

and if it returns `More (useq',pseq')`, then

```
joinSeq (pseq', useq')  =  seq
```

Unlike `mapLTSUntil`, we do not require that `mapUntilSeq` only pauses and returns a pair of unprocessed and processed sequences when the number of unprocessed elements is greater than or equal to two. This difference exists because `mapUntilSeq` is called on behalf of `mapLTSUntil`; although a call to `mapUntilSeq` may return with one unprocessed element, the context maintained by `mapLTSUntil` may have additional unprocessed elements. Theorem 2 in Appendix A.1 proves that an implementation of `mapUntilSeq` satisfies Property 3, and therefore may be used in our shape-preserving implementation of `mapLTS`.

If `mapUntilSeq` returns a complete result (`Done pseq'`), then the traversal plugs the context with this completed leaf sequence and attempts to navigate to the next

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

```
fun mapLTSUntil cond f rp = let
  fun mSeq (seq, ctx) = case next (Leaf seq, ctx)
    of Done rp' => rp'
     | More (seq', ctx') => mSeq (mapSeq f seq', ctx')
  fun m (seq, ctx) = (case mapUntilSeq cond f seq
    of Done pseq' => (case next (Leaf pseq', ctx)
         of Done rp' => Done rp'
          | More (seq', ctx') => m (seq', ctx'))
     | More (useq', pseq') =>
         if snd (curLength ((pseq', useq'), ctx)) >= 2 then
           More ((pseq', useq'), ctx)
         else
           Done (mSeq (joinSeq (pseq', mapSeq f useq'), ctx)))
  val (seq, ctx) = leftmost (rp, MCTop)
  in
    m (seq, ctx)
  end
```

Fig. 6. The `mapLTSUntil` operation.

unprocessed leaf by calling `next (Leaf pseq', ctx)`. If `next` returns `Done rp'`, then the rope traversal is complete and the whole processed rope is returned. Otherwise, `next` returns `More (seq', ctx')` and the traversal loops to process the next leaf sequence (`seq'`) with the new context (`ctx'`).

If `mapUntilSeq` returns a partial result (`More (useq',pseq')`), then the traversal determines the number of unprocessed elements contained in both the unprocessed sequence `useq'` and the context `ctx`. If there are at least two unprocessed elements, then the traversal pauses and returns an intermediate cursor. (This pause and return gives `mapLTS` the opportunity to split the unprocessed elements and push the parallel mapping of these halves of the unprocessed elements onto the work-stealing deque.) If there are less than two elements, then the traversal sequentially processes the remaining unprocessed element to complete the rope traversal and return the whole processed rope. Theorem 3 in Appendix A.2 proves that this implementation of `mapLTSUntil` satisfies Property 2, and therefore may be used in our shape-preserving implementation of `mapLTS`.

4.1.5 *`split` and `join` implementation*

Finally, let us consider the implementation of `split` and `join`. The key idea behind the implementations of these operations is to introduce data structures that we call the *unzipped context* and the *unzipped cursor*, which enables us to temporarily break apart a (zipped) context or cursor and to later put the context or cursor back together

```
datatype dir = Left | Right
type ('b, 'a) unzip_map_ctx =
   'b rope list * 'a rope list * dir list
type ('b, 'a) unzip_map_cur = ('b, 'a) unzip_map_ctx
```

This representation divides a context into three lists: (1) a list of processed sub-ropes located above and left of the hole, (2) a list of unprocessed sub-ropes located above

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

and to the right of the hole, and (3) a list of branch directions. An unzipped cursor has the same type as that of an unzipped context, but has an additional invariant: the first elements of the `'b rope list` and the `'a rope list` components are `Leaf` ropes, corresponding to the `'b seq` and `'a seq` components of a (zipped) cursor. The zipped and unzipped contexts and cursors are just two different ways of representing the same context or cursor. For example, both the zipped context

```
MCRight (rp1, MCRight (rp2, MCLeft (MCRight (rp4, MCTop), rp3)))
```

and the unzipped context

```
([rp1, rp2, rp4],
 [rp3],
 [Right, Right, Left, Right])
```

represent a context of two right branches `rp1` and `rp2`, a left branch `rp3`, and a right branch `rp4`. It is easy to define operations to unzip a cursor

```
val ctxUnzip : ('b, 'a) map_ctx -> ('b, 'a) unzip_map_ctx
fun ctxUnzip c = (case c
  of MCTop =>
       (nil, nil, nil)
   | MCLeft (c, r) => let
       val (ls, rs, ds) = ctxUnzip c
       in
         (ls, r :: rs, Left :: ds)
       end
   | MCRight (l, c) => let
       val (ls, rs, ds) = ctxUnzip c
       in
         (l :: ls, rs, Right :: ds)
       end)

val curUnzip : ('b, 'a) map_cur -> ('b, 'a) unzip_map_cur
fun curUnzip ((pseq,useq), ctx) = let
  val (ls, rs, ds) = ctxUnzip ctx
  in
    ((Leaf pseq)::ls, (Leaf useq)::rs, ds)
  end
```

and *vice versa*

```
val ctxZip : ('b, 'a) unzip_map_ctx -> ('b, 'a) map_ctx
fun ctxZip (ls, rs, ds) = (case (ls, rs, ds)
  of (nil, nil, nil) =>
       MCTop
   | (ls, r :: rs, Left :: ds) =>
       MCLeft (ctxZip (ls, rs, ds), r)
   | (l :: ls, rs, Right :: ds) =>
       MCRight (l, ctxZip (ls, rs, ds)))

val curZip : ('b, 'a) unzip_map_cur -> ('b, 'a) map_cur
fun curZip ((Leaf pseq)::ls, (Leaf useq)::rs, ds) =
  ((pseq, useq), ctxZip (ls, rs, ds))
```

Although the zipped and unzipped contexts and cursors are different ways of representing the same context or cursor, they are each suited for different tasks.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

The zipped contexts and cursors are better suited (being both easier to code and faster to execute) for the step-by-step traversal of a rope used in the implementation of `mapLTSUntil`. On the other hand, the unzipped contexts and cursors are better suited for the splitting of a rope used in the implementation of `mapLTS`.

From the description of an unzipped context, it should be clear that our initial handle on the unprocessed elements of a context is through a list of ropes (of unprocessed elements). To split this list of ropes at the `n`th unprocessed element, we first divide this list of ropes into three parts based on `n`, using the `divideRopes` helper function

```
fun divideRopes (rp :: rps, n) =
  if n <= length rp then
    (nil, rp, n, rps)
  else let
    val (rps1, rp', n', rps2) = divideRopes (rps, n - length rp)
    in
      (rp :: rps1, rp', n', rps2)
    end
```

The application `divideRopes (rps, n)` returns `(rps1, rp, k, rps2)` such that `rps1 @ [rp] @ rps2` is equal to `rps` and `rps1` and `rp` contain at least the first `n` elements of the ropes of `rps`. The integer `k` is the index in `rp` at which the `n`th element of `rps` is found. As noted above, the inverse operation of `divideRopes` is simply the concatenation of `rps1`, `[rp]`, and `rps2`.

While `divideRopes` has roughly divided the unprocessed elements into those ropes that occur strictly before the split, the rope in which the split occurs, and those ropes that occur strictly after the split, our next task is to split the rope in which the split occurs. The application `splitAtAsCur (rp, n)` returns a cursor in which the "hole" occurs between the `n`th and `n+1`st elements of the rope `rp`

```
val splitAtAsCur : 'a rope * int -> ('a, 'a) map_cur
fun splitAtAsCur (rp, n) = let
  fun s (rp, ctx, n) = (case rp
    of Leaf seq => let
         val (lseq, rseq) = splitAtSeq (seq, n)
         in
           ((lseq, rseq), ctx)
         end
     | Cat (lrp, rrp) =>
         if n < length lrp then
           s (lrp, MCLeft (ctx, rrp), n)
         else
           s (rrp, MCRight (lrp, ctx), n - length lrp))
  in
    s (rp, MCTop, n)
  end
```

To recover the original rope, it suffices to use the `root` operation. We may also unzip the context returned by `splitAtAsCur` to obtain additional lists of ropes that occur before and after the split.

Our final pair of helper functions encode a list of ropes as a single rope and decode a single rope as a list of ropes. Encoding a list of ropes as a single rope

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

will be the last step of split, whereby the lists of unprocessed ropes returned by divideRopes and splitAtAsCur are turned into two single ropes for parallel processing in mapLTS. The application encodeRopes rps returns a rope rp and an integer l, where l is the length of the list rps. The length is used by decodeRope to reconstruct rps

```
val encodeRopes : 'a rope list -> 'a rope * int
fun encodeRopes rps = let
  fun e rs = (case rs
    of [rp] =>
         rp
     | rp :: rps =>
         Cat (rp, e rps))
  in
    (e rps, List.length rps)
  end
```

The application decodeRope (rp, l) returns a list of ropes rps

```
val decodeRope : 'a rope * int -> 'a rope list
fun decodeRope (rp, n) =
  if n = 1 then
    [rp]
  else (case rp
    of Cat (l, r) =>
       l :: decodeRope (r, n - 1))
```

We can now present the implementation of split, which, as specified above, takes a cursor cur and returns two ropes rp1 and rp2 and a rebuilder data structure, reb. The rope rp1 contains the first half of the unprocessed elements of cur and rp2 contains the remaining unprocessed elements. The rebuilder data structure reb provides sufficient information to reconstruct cur from rp1 and rp2. The complete code is shown in Figure 7. Let (ls, rs, ds) be the result of curUnzip cur. We divide the list of unprocessed sub-ropes rs into three parts: the sub-ropes rps1 that occur before position n, the sub-rope mrp containing the data element at position n, and the sub-ropes rps2 that occur after position n. Next, we let (mls, mrs, mds) be the unzipped cursor that splits the sub-rope mrp. We let n1 and n2 be the lengths of rps1 and mrs, respectively. These values enable us to later separate the rps1 sub-ropes from the mls sub-ropes, and the mrs sub-ropes from the rps2 sub-ropes. We let (rp1, l1) and (rp2, l2) be the rope encodings of rs1 @ mls and mrs @ rs2, respectively. The result of split is then

```
(rp1, rp2, (ls, ds, mds, n1, n2, l1, l2))
```

where the third component is the rebuilder, which therefore has the type

```
type 'b map_cur_reb =
  ('b rope list * dir list * dir list * int * int * int * int)
```

Recall that join takes encoded ropes rp1 and rp2 and rebuilder

```
(ls, ds, mds, n1, n2, l1, l2)
```

and returns the cursor that was originally split. The implementation of join follows straightforwardly by successively inverting each of the operations performed

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

```
fun split cur = let
  val n = snd (curLength cur) div 2
  val (ls, rs, ds) = curUnzip cur
  val (rps1, mrp, k, rps2) = divideRopes (rs, n)
  val (mls, mrs, mds) = curUnzip (splitAtAsCur (mrp, k))
  val (n1, n2) = (List.length rps1, List.length mrs)
  val (rp1, l1) = encodeRopes (rps1 @ mls)
  val (rp2, l2) = encodeRopes (mrs @ rps2)
  in
    (rp1, rp2, (ls, ds, mds, n1, n2, l1, l2))
  end

fun join (rp1, rp2, (ls, ds, mds, n1, n2, l1, l2)) = let
  val xs1 = decodeRope (rp1, l1)
  val (rps1, mls) = (List.take (xs1, n1), List.drop (xs1, n1))
  val xs2 = decodeRope (rp2, l2)
  val (mrs, rps2) = (List.take (xs2, n2), List.drop (xs2, n2))
  val mrp = root (curZip (mls, mrs, mds))
  val rs = rps1 @ [mrp] @ rps2
  in
    curZip (ls, rs, ds)
  end
```

Fig. 7. The implementation of `split` and `join`.

by `split`. Let `rps1` and `rps2` be the decodings of `(rp1, l1)` and `(rp2, l2)`, respectively, that are obtained by two calls to `decodeRope`. From `rps1` and `n1` we reconstruct the lists of sub-ropes `rs1` and `mls`, and from `rps2` and `n2` we reconstruct the lists of sub-ropes `mrs` and `rs2`. We then let `mrp` be

```
root (curZip (mls, mrs, mds))
```

Next, we let `rs` be `rs1 @ [m] @ rs2`. The original cursor is thus

```
curZip (ls, rs, ds)
```

which is the result returned by `join`. Theorem 4 in Appendix A.3 proves that these implementations of `split` and `join` satisfy Property 1, and therefore may be used in our shape-preserving implementation of `mapLTS`.

4.2 Implementing other operations

The implementation of `filterLTS` is very similar to that of `mapLTS`. Indeed, `filterLTS` uses the same context representation and operations as `mapLTS`, simply instantiated with unprocessed and processed elements having the same type:

```
val filterLTS : ('a -> bool) -> 'a rope -> 'a rope
type 'a filter_ctx = ('a, 'a) map_ctx
```

As with `mapLTS`, where the mapping operation was applied by the `mapUntilSeq` operation, the actual filtering of elements is performed by the `filterUntilSeq` operation. One complication of all rope-filter operations, including `filterLTS`, is

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

that filter operations are not balance-preserving because data elements are removed from the filter result rope based on the filter predicate. We make `filterLTS` balance-preserving by applying our parallel balancing function `balance` to the result rope.

The `reduceLTS` operation takes an associative operator, its zero, and a rope and returns the rope's reduction under the operator

```
val reduceLTS : ('a * 'a -> 'a) -> 'a  -> 'a rope -> 'a
```

Thus, the `reduceLTS` operation may be seen as a generalized sum operation. The implementation of `reduceLTS` is again similar to that of `mapLTS`, but uses a simpler context:

```
datatype 'a reduce_ctx
  = RCTop
  | RCLeft of 'a rope * 'a reduce_ctx
  | RCRight of 'a * 'a reduce_ctx
```

where `RCRight (z, c)` represents the context surrounding the right branch of a `Cat` node in which `z` is the *reduction* of the left branch and `c` is the context surrounding the reduction of the `Cat` node.

The `scanLTS` operation, also known as *prefix sums*, is used by many data-parallel algorithms. Like `reduceLTS`, the `scanLTS` operation takes an associative operator, its zero, and a rope, and returns a rope of the reductions of the prefixes of the input rope

```
val scanLTS : ('a * 'a -> 'a) -> 'a -> 'a rope -> 'a rope
```

For example,

```
scanLTS (op +) 0 (Cat (Leaf [1, 2], Leaf [3, 4]))
  ⇒ Cat (Leaf [1, 3], Leaf [6, 10])
```

In a survey on prefix sums, Blelloch (1990a) describes classes of important parallel algorithms that use this operation and gives an efficient parallel implementation of prefix sums, on which our implementation of `scanLTS` is based. The algorithm takes two passes over the rope. The first performs a parallel reduction over the input rope, constructing an intermediate rope in which partial reduction results are recorded at each internal node. The second pass builds the result rope in parallel by processing the intermediate rope. The efficiency of this second pass is derived from having constant-time access to the cached sums while it builds the result.

The result of this first pass is called a *monoid-cached tree* (Hinze & Paterson, 2006), specialized in the current case to *monoid-cached rope*. In a monoid-cached rope,

```
datatype 'a crope
  = CLeaf of 'a * 'a seq
  | CCat of 'a * 'a crope * 'a crope
```

each internal node caches the reduction of its children nodes. For example, supposing the scanning operator is integer addition, one such monoid-cached rope is

```
CCat (10, CLeaf (3, [1, 2]), CLeaf (7, [3, 4]))
```

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

Our implementation of Blelloch's algorithm is again similar to that of `mapLTS`, except that we use a context in which there are `ropes` to the right of the hole and `cached_ropes` to the left of the hole. Aside from some minor complexity involving the propagation of partial sums, the operations on this context are similar to those on the context used by `mapLTS`.

The `map2LTS` operation maps a binary function over a pair of ropes (of the same length)

```
val map2LTS : ('a * 'b -> 'c) -> 'a rope * 'b rope -> 'c rope
```

For example, the pointwise addition of the ropes `rp1` and `rp2` can be implemented as

```
map2LTS (op +) (rp1, rp2)
```

Note that `rp1` and `rp2` may have completely different branching structures, which would complicate any structural–recursive implementation. The zipper technique provides a clean alternative: we maintain a pair of contexts and advance them together in lock step during execution. The result rope is accumulated in one of these contexts.

Contexts and partial results nicely handle the processing of leaves of unequal length. When the `map2SeqUntil` function is applied to two leaves of unequal length, it simply returns a partial result that includes the remaining elements from the longer sequence. The `map2Until` function need only step the context of the shorter linear sequence to find the next leaf with which to resume the `map2SeqUntil` processing. We do need to distinguish `map2SeqUntil` returning with a partial result because of the polling function, in which case `map2Until` should also return a partial result (signaling that a task should be pushed to the work-stealing deque) from `map2SeqUntil` returning with a partial result do to exhausting one of the leaves, in which case `map2Until` should not return a partial result. The implementation straightforwardly extends to maps of arbitrary arity.

5 Evaluation

We have already demonstrated in Section 3 that with ETS manual tuning of the chunk size is essential to obtain acceptable parallel performance across all of our benchmarks. In this section, we present the results of additional experiments that demonstrate that LTS performance is always close to the best, hand-tuned ETS. Furthermore, these additional experiments demonstrate that no hand tuning was necessary to achieve good performance with LTS.

5.1 Experimental method

Our benchmark machine is a Dell PowerEdge R815 server, outfitted with 48 cores and 128 GB physical memory. This machine runs x86_64 Ubuntu Linux 10.04.2 LTS, kernel version 2.6.32-27. The 48 cores are provided by four 12 core AMD Opteron 6172 "Magny Cours" processors (Carver, 2010; Conway *et al.*, 2010). Each core operates at 2.1 GHz and has 64 KB each of instruction and data L1 cache and

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

512 KB of L2 cache. There are two 6 MB L3 caches per processor, each of which is shared by six cores, for a total of 48 MB of L3 cache.

We ran each experiment 10 times, and we report the average performance results in our graphs and tables. For most of these experiments the standard deviation was below 2%, thus we omit the error bars from our plots.

5.2 Benchmarks

For our empirical evaluation, we ran one synthetic benchmark and seven benchmark programs picked from our benchmark suite. Our maximum leaf size is 1,024, which is one setting that provided good performance on our test machine across all seven benchmarks.

The Barnes–Hut benchmark is an n-body simulation that calculates the gravitational forces between n particles as they move through two-dimensional space (Barnes & Hut, 1986). The Barnes–Hut computation consists of two phases. In the first, the simulation volume is divided into square cells via a quadtree so that only particles from nearby cells need to be handled individually, and particles from distant cells can be grouped together and treated as large particles. The second phase calculates gravitational forces using the quadtree to accelerate the computation. We represent the sequence of particles by a rope of mass-point and velocity pairs and the quadtree by an algebraic data type where every node is annotated with a mass point. Our benchmark runs for 20 iterations over 3,000,000 particles generated from the random Plummer distribution (Plummer, 1911). The program is adapted from a Data-Parallel Haskell program (Peyton Jones *et al.*, 2008).

The Raytracer benchmark renders a 2,000 × 2,000 image in parallel as a two-dimensional sequence, which is then written to a file. The original program was written in ID (Nikhil, 1991) and implements a simple ray tracer that does not use any acceleration data structures. The sequential version outputs each pixel to the image file as it is computed, whereas the parallel version first builds an intermediate rope of pixels and later flushes the rope to a file.

The Quicksort benchmark sorts a rope of 10 million integers in parallel. Our program is adapted from one that was originally written for NESL (Scandal Project, n.d.).

The SMVM benchmark is a sparse-matrix by dense-vector multiplication. The matrix contains 1,091,362 elements and the vector 16,614.

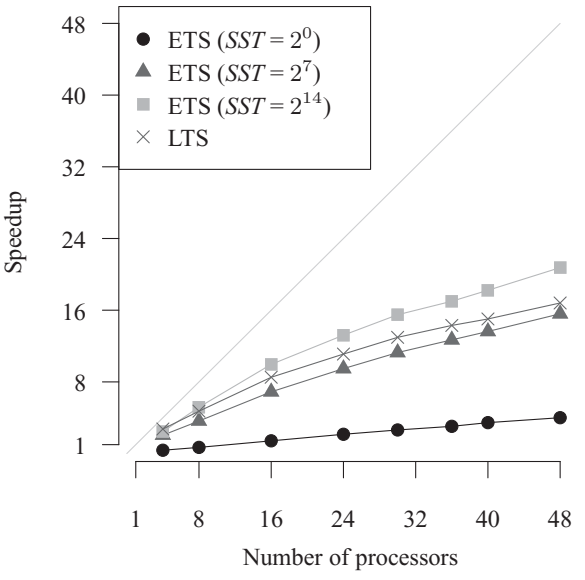
The DMM benchmark is a dense-matrix by dense-matrix multiplication in which each matrix is 600 × 600. We represent a matrix column as a rope of scalars and a matrix as a rope of columns.

The Black–Scholes benchmark computes the price of European options analytically using a partial differential equation. We store the options in a rope.

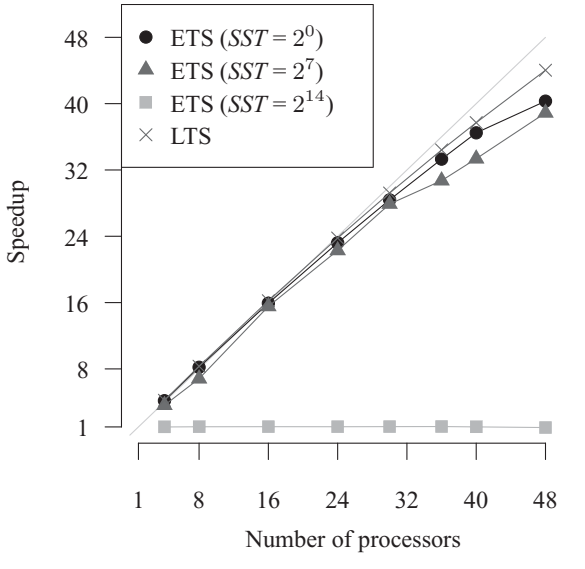
The Nested Sums benchmark is a synthetic benchmark that exhibits irregular parallelism. Its basic form is as follows:

```
let fun upTo i = range (0, i)
in mapP sumP (mapP upTo (range (0, 5999)))
end
```

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press



(a) Barnes–Hut



(b) Raytracer

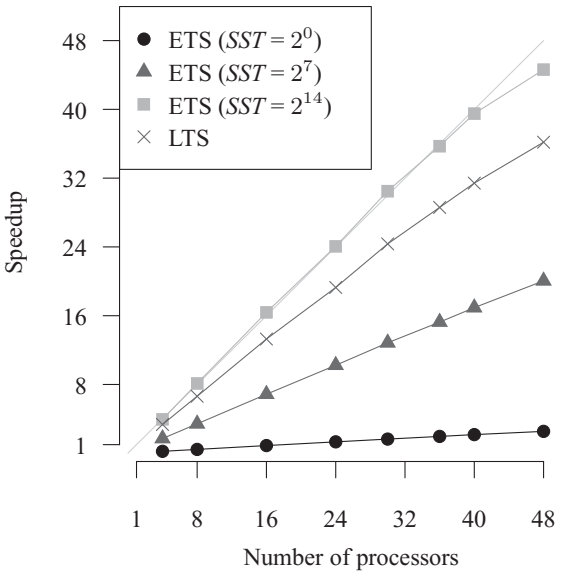
Fig. 8. Performance of LTS and ETS.

The program generates an 6,000-element array of 6,000-integer arrays and returns an array containing the sum of each sub-array.

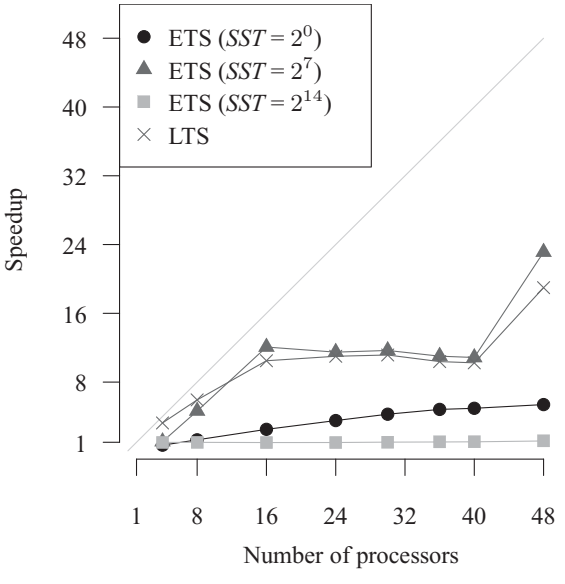
5.3 Lazy vs. *eager tree splitting*

Figures 8–11 show the performance of LTS and ETS side by side. Each graph contains four speedup curves for a single benchmark, with one curve for LTS and three curves for ETS with small, medium, and large settings of SST. We chose

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press



(a) Quicksort



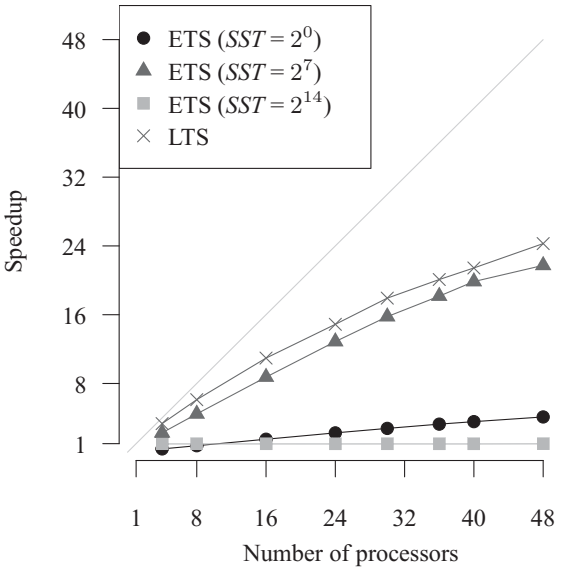
(b) SMVM

Fig. 9. Performance of LTS and ETS.

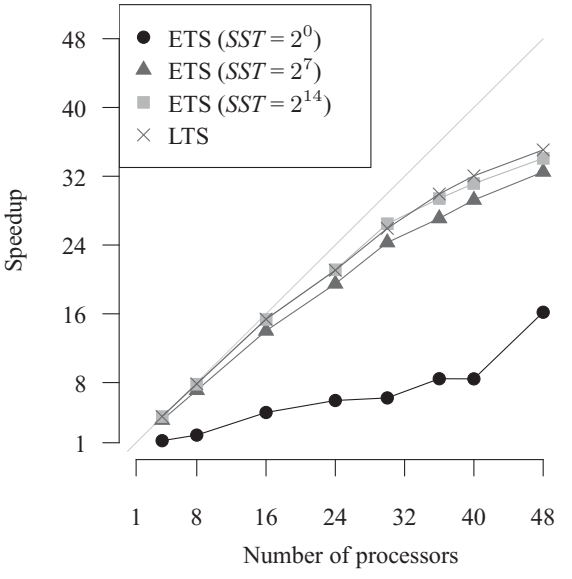
these particular SST values because they cover various extremes of performance, as shown in Figure 3. Observe that in each graph the LTS speedup is close to the greatest ETS configuration and that the performance curves of most of the ETS configurations are flat.

In Table 1, we present performance measurements for each of our benchmarks run in several different sequential configurations, as well as on 48 processors. The

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press



(a) DMM



(b) Black-Scholes

Fig. 10. Performance of LTS and ETS.

first column of data presents timing results for MLton. MLton is a sequential whole-program optimizing compiler for Standard ML (Weeks, 2006; MLton, n.d.), which is the "gold standard" for ML performance. The second data column gives the baseline performance of natural sequential PML versions of the benchmarks (i.e., parallel operations are replaced with their natural sequential equivalents). We are about a factor of 1.5–3.0x slower than MLton for all of the benchmarks except Nested Sums.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

Table 1. *Summary of performance. Execution time in seconds*

		PML				
Benchmark	MLton	Seq.	LTS	Best ETS 48	LTS 48	LTS 48 speedup
DMM	6.79	21.86	20.59	1.01	0.91	24.05
Raytracer	166.36	253.57	247.20	6.28	5.75	44.10
SMVM	5.21	15.31	13.52	0.64	0.81	18.86
Quicksort	28.41	59.39	65.26	1.33	1.64	36.21
Barnes Hut	165.84	502.17	521.63	24.27	29.57	16.98
Black Scholes	3.96	8.20	8.18	0.24	0.24	34.17
Nested Sums	7.19	25.93	25.86	2.30	1.02	25.42

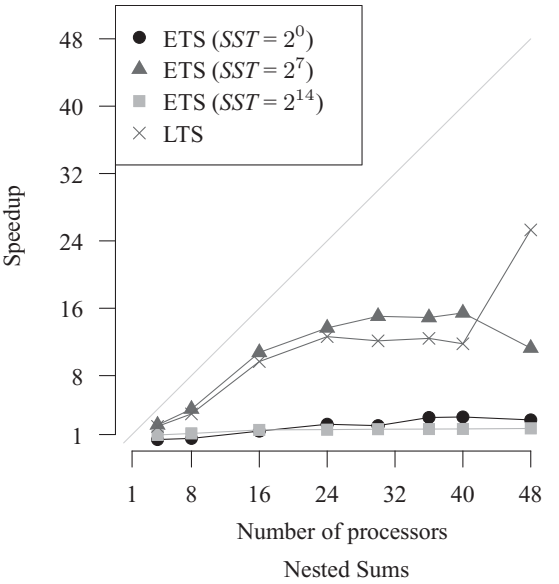


Fig. 11. Performance of LTS and ETS.

Considering MLton's suite of aggressive optimizations and maturity, the sequential performance of PML is encouraging. Our slower performance can be attributed to at least two factors. First, the MLton compiler monomorphizes the program and then aggressively flattens the resulting monomorphic data representations, whereas Manticore does no such monomorphization and the resulting code often involves boxed data representations. Second, our profiling shows higher GC overheads in our system. These issues can be addressed by improving the sequential performance of Manticore. The last two columns report the parallel execution time and speedup on 48 cores. Overall, the speedups are quite good. The Barnes–Hut benchmark, however, achieves a modest speedup, which we believe stems from a limit on the amount of parallelism in the program. This hypothesis is supported by the fact that increasing the input size improves the speedup results.

Observe that in many cases the 48-core LTS performance falls behind the best 48-core ETS performance. This gap may indicate that LTS involves some overhead

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

costs that are heavier than those of ETS. To break down the sources of these overheads, first recall that LTS requires the program to make one or more zipper traversals and that each zipper traversal requires heap allocations. To estimate the zipper overhead, we can compare the execution times in the columns labeled Seq. and LTS in Table 1. The LTS column contains the execution time of the benchmarks using the LTS runtime mechanisms (e.g., zippers), but without parallelism. We see that in the sequential case, the LTS version is about 24% slower, which is indeed a significant cost. By comparison, the ETS traversal uses a natural structural recursion in which the state is maintained via the runtime call stack. In many compilers, including Manticore, the natural recursion is often more efficient than a zipper because compiler optimizations are more effective at optimizing natural-recursive code and it can benefit from stack as opposed to heap allocation.

We also ran an experiment to measure LTS overheads in MLton, because MLton offers better sequential performance and uses a more-conventional C-style call stack, whereas Manticore uses heap-allocated continuations to represent the call stack (Appel, 1992; Fluet *et al.*, 2007b). In this experiment we ran SMVM sequentially using LTS and ETS versions and found that the LTS and ETS versions were completed in 8.49 and 4.99 seconds, respectively, indicating a 70% advantage for the ETS version. A likely contributor to this gap is the difference in heap allocation: LTS and ETS versions allocated 6.4 GB and 2.8 GB, respectively. In spite of these costs, the extra heap allocations in LTS do not necessarily harm its scalability because, in Manticore, the allocated zipper objects are almost always reclaimed by the same processor that performed the allocation.

Another possibility we considered is that LTS suffers because of communication costs from extra task migrations. Profiling data that we gathered suggest otherwise, however, because the data show no significant difference in the number of steals between LTS and the best ETS configuration. Furthermore, our profiling data show that the per-processor utilization for the best ETS configuration is never more than 3% than that of LTS, which is almost within our 2% error bar.

There is still a question of whether our technique trades one tuning parameter, SST, for another, the maximum leaf size. We address this concern in two ways. First, observe that even if performance is sensitive to the leaf size, this problem is specific to ropes, but neither ETS nor LTS. Second, we have measured the effect of the maximum leaf size on performance. Figure 12 shows the speedups for our benchmarks as a function of maximum leaf size on 48 processors. The results show that all of benchmarks perform well for maximum leaf sizes in the set $\{512, 1024, 2048\}$, so our choice of $1,024$ is justified. One concern is DMM, which is sensitive to M because it does many subscript operations on its two input ropes. One could reduce this sensitivity by using a flatter rope representation that provides a faster subscript operation.

6 Related work

Adaptive parallel loop scheduling. The original work on lazy binary splitting presents a dynamic scheduling approach for parallel `do-all` loops (Tzannes *et al.*, 2010).

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

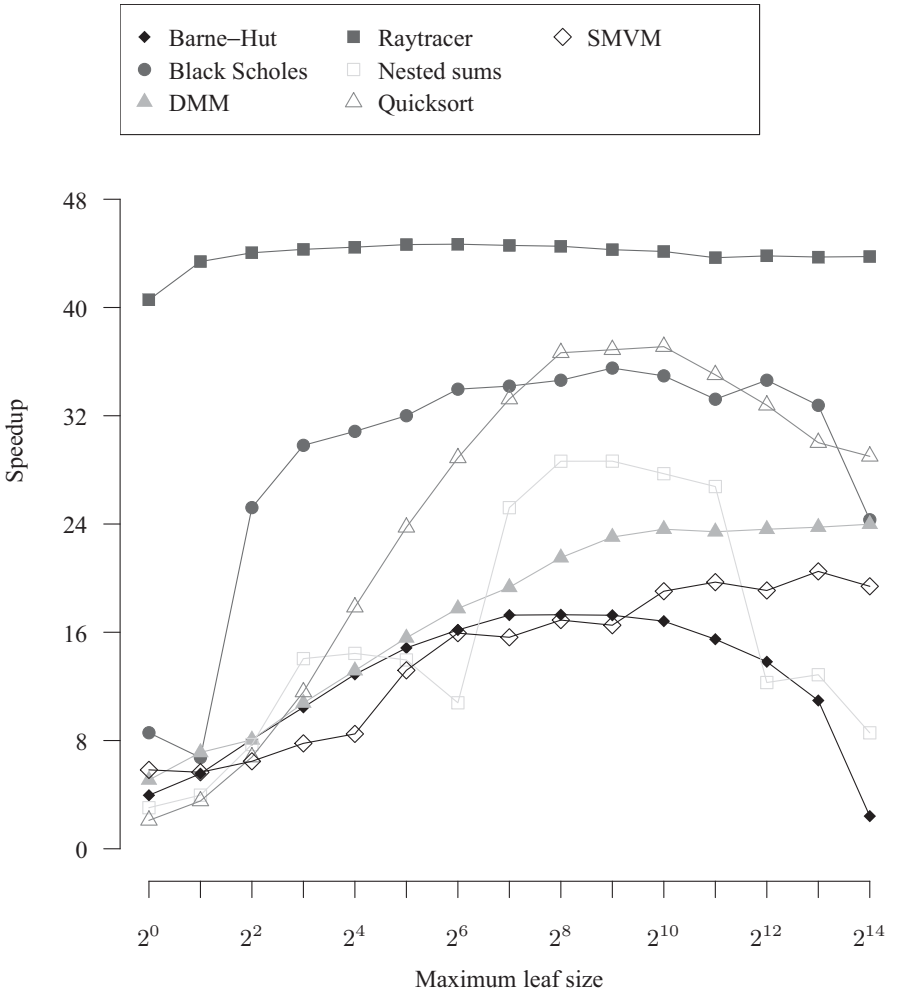


Fig. 12. The effect of varying the maximum leaf size on 48 processors.

Their work addresses splitting ranges of indices, whereas ours addresses splitting trees where tree nodes are represented as records allocated on the heap.

In the original LBS work, they use a *profitable parallelism threshold* (PPT) to reduce the number of hungry-processor checks. The PPT is an integer that determines how many iterations a given loop can process before doing a hungry-processor check. Our performance study has $PPT = 1$ (i.e., one hungry-processor check per iteration) because we have not implemented the necessary compiler mechanisms to do otherwise.

Robison *et al.* (2008) propose a variant of EBS called auto partitioning, which provides good performance for many programs and does not require tuning.[10] Auto partitioning derives some limited adaptivity by employing the heuristic that when

[10] Auto partitioning is currently the default chunking strategy of TBB (Intel, 2008).

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

a task detects it has been migrated it splits its chunk into at least some fixed number of sub-chunks. The assumption is that if a steal occurs, there are probably other processors that need work, and it is worthwhile to split a chunk further. As discussed by Tzannes, *et al.* (2010), auto partitioning has two limitations. First, for i levels of loop nesting, P processors, and a small, constant parameter K, it creates at least $(K \times P)^i$ chunks, which is excessive if the number of processors is large. Second, although it has some limited adaptivity, auto partitioning lacks performance portability with respect to the context of the loop, which limits its effectiveness for scheduling programs written in the liberal loop-nesting style of an NDP language.

Cutting off excess parallelism. One approach to the granularity problem is to try to limit the total number of tasks that get created so as to guarantee that the total cost of scheduling can be well amortized. Variations of the cutoff-based approach have been studied by Loidl and Hammond (1995) in the context of Haskell, and Lopez *et al.* (1996) and Tick and Zhong (1993) in the context of logic programming. Their key idea is that if a given task is small, the scheduler executes the task as a sequential computation, that is, completely free of scheduling costs. A limitation of the cutoff-based approaches is that they rely on there being a reasonably accurate way of predicting the task-execution time. Predicting task-execution time is difficult for many classes of programs, such as ray tracers, where execution time depends heavily on properties of the input data set, and is not feasible in general. In cases where prediction is not feasible, LTS can still be an effective approach, because LTS does not depend on prediction. LTS is concerned only with reducing the scheduling cost per task.

Flattening and fusion. NESL is a nested data-parallel dialect of ML (Blelloch *et al.*, 1994). The NESL compiler uses a program transformation called *flattening*, which transforms nested parallelism into a form of data parallelism that maps well onto SIMD architectures. Note that SIMD operations typically require array elements to have a contiguous layout in memory. Flattened code maps well onto SIMD architectures because the elements of flattened arrays are readily stored in adjacent memory locations. In contrast, LTS is a dynamic technique that has the goal of scheduling nested parallelism effectively on MIMD architectures. A flattened program may still use LBS (or LTS) to schedule the execution of array operations on MIMD architectures, so in that sense flattening and LTS are orthogonal.

There is, of yet, no direct comparison between an NDP implementation based on LTS and an implementation based on flattening. One major difference is that LTS uses a tree representation whereas flattening uses contiguous arrays. As such, the LTS representation has two disadvantages. First, tree random access is more expensive. For a rope it is $O(\log n)$ time, where n is the length of a given rope. Second, there is a large constant factor overhead imposed by maintaining tree nodes. One way to reduce these costs is to use a "bushy" representation that is similar to ropes but where the branching factor is greater than two and child pointers are stored in contiguous arrays.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

Data-parallel fusion is a program transformation that eliminates data-parallel operations under certain circumstances. It is implemented in the NESL (Chatterjee, 1993) and Data-Parallel Haskell (Chakravarty *et al.*, 2008) compilers, but not in Manticore currently. Fusion typically improves task granularity, thanks to increasing the work per task. To see why, consider the expression

```
mapP f (mapP g xs)
```

and its fused counterpart

```
mapP (f o g) xs
```

Combining the two `mapP`s yields a computation, which both generates fewer logical tasks – only one per array element instead of two – and sacrifices no parallelism. Although it improves granularity, fusion is limited as a granularity-control mechanism, because the transformation applies only when there are pairs of operations that can be fused. As such, additional mechanisms, such as LTS, are crucial for addressing granularity control in general.

Parallel depth-first scheduling. Work by Greiner and Blelloch (1996) proposes an implementation of NDP based on a scheduling policy, called Parallel Depth First (PDF), which is designed to minimize space usage. The practicality of PDF on modern machines is severely limited because the policy relies on a centralized task queue. Narlikar and Blelloch (1999) address this issue by proposing a scheduling policy called DFDeques, which is a hybrid of PDF and work stealing. Although DFDeques addresses the inefficiency of having a centralized queue, the scheduling costs involved in DFDeques are similar to those of plain work stealing, because the granularity-control mechanism of DFDeques involves switching from PDF to work stealing every time a fixed amount of memory has been allocated. LBS and LTS further improve on plain work stealing by optimizing for the special cases of loops and NDP operations.

Ct. Ct is an NDP extension to C++ (Ghuloum *et al.*, 2007). So *et al.* (2006) describe a fusion technique for Ct that is similar to the fusion technique of DPH. The fusion technique used by Ct is orthogonal to LTS for the same reasons as for the fusion technique of DPH. The work on Ct does not directly address the issue of building an automatic chunking strategy, which is the main contribution of LTS.

GpH. GpH introduced the notion of an *evaluation strategy* (Trinder *et al.*, 1998), which is a part of a program that is dedicated to controlling some aspects of parallel execution. Strategies have been used to implement eager-splitting-like chunking for parallel computations. We believe that a mechanism like an evaluation strategy could be used to build a clean implementation of lazy tree splitting in a lazy functional language.

Cilk. Cilk is a parallel dialect of C language extended with linguistic constructs for expressing fork-join parallelism (Frigo *et al.*, 1998). Cilk is designed for parallel function calls but not loops, whereas our approach addresses both.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

```
type 'a seq = 'a list
fun joinSeq (seq1, seq2) = List.append (seq1, seq2)
fun revSeq seq = List.rev seq
fun mapSeq f seq = List.map f seq
fun mapUntilSeq cond f seq = let
  fun lp (seq, acc) =
    (case seq
      of [] => Done (revSeq acc)
       | x::seq' =>
           if cond () then
              More (seq, revSeq acc)
           else
              lp (seq', (f x)::acc))
in
  lp (seq, [])
end
```

Fig. A 1. The `mapUntilSeq` operation for lists.

7 Conclusion

We have described the implementation of NDP features in the Manticore system. We have also presented a new technique for parallel decomposition, lazy tree splitting, inspired by the lazy binary splitting technique for parallel loops. We presented an efficient implementation of LTS over ropes, making novel use of the zipper technique to enable necessary traversals. Our techniques can be readily adapted to tree data structures other than ropes and are not limited to functional languages. A work-stealing thread scheduler is the only special requirement for our technique.

Lazy Tree Splitting compares favorably to ETS, requiring no application-specific or machine-specific tuning. For any of our benchmarks, LTS outperforms most or all configurations of ETS, and is, at worst, only 27% slower than the optimally tuned ETS configuration. As argued here by us and elsewhere by others (Tzannes *et al.*, 2010), the ETS approach is not feasible in general because, in order to achieve acceptable performance, the programmer has to tune each instance of a given parallel tree operation to the given context in which the operation appears and for each machine on which it is to be run. LTS achieves good performance without the need for tuning.

A Proofs

A.1 `mapSeqUntil` is shape preserving

Figure A 1 gives an implementation of `mapUntilSeq` for sequences implemented as lists. Although our actual implementation uses contiguous arrays for sequences, the implementation here demonstrates the essential behavior, in which the function maintains an implicit context with processed elements to the left and unprocessed elements to the right.

With the following lemma and theorem, we can conclude that our implementation of `mapSeqUntil` can be used safely by `mapLTS`.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

Lemma 1. For any sequence `seq`, any sequence `acc`, any predicate `cond`, with `f = fn x => x`, if `lp (seq, acc)` returns `Done pseq'`, then

```
pseq'  =  joinSeq (seq, revSeq acc)
```

and if it returns `More (useq', pseq')`, *then*

```
joinSeq (useq', pseq')  =  joinSeq (seq, revSeq acc)
```

Proof
By structural induction on `seq`. □

Theorem 2 (`mapUntilSeq` is shape preserving). Property 3 holds for the implementation of `mapSeqUntil`.

For any sequence `seq` and any predicate `cond`, if `mapUntilSeq cond (fn x => x) seq` returns `Done seq'`, then

```
seq'  =  seq
```

and if it returns `More (useq',pseq')`, then

```
joinSeq (pseq', useq')  =  seq
```

Proof
By Lemma 1. □

A.2 `mapLTSUntil` is well-behaved

The well-behavedness of our `mapLTSUntil` operation, namely, that it satisfies Property 2, will depend upon a number of properties about the context and cursor operations. For instance, the `leftmost` operation preserves the represented rope as well as its length and size.

Lemma 2. *For any rope* `rp` *and any context* `ctx`, *if* `leftmost (rp, ctx)` *returns* `(seq', ctx')`, *then*

```
plug (Leaf seq', ctx')  =  plug (rp, ctx)
```

and

```
(ctxLength ctx') ++ (0, lengthSeq seq')
   =  (ctxLength ctx) ++ (0, length rp)
```

and

```
(ctxSize ctx') ++ (0, 1)  =  (ctxSize ctx) ++ (0, size rp)
```

Proof
By assumption,

```
leftmost (rp, ctx)  =  (seq', ctx')        (1)
```

The proof is by structural induction on `rp`.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

- Suppose that

```
rp = Leaf seq                                    (2)
```

Therefore,

```
(seq', ctx')
   = leftmost (rp, ctx)              by (1)
   = leftmost (Leaf seq, ctx)        by (2)
   = (seq, ctx)                      by defn of leftmost
```

and

```
seq' = seq                                       (3)
```

and

```
ctx' = ctx                                       (4)
```

Hence,

```
plug (Leaf seq', ctx')
   = plug (Leaf seq, ctx)            by (3) and (4)
   = plug (rp, ctx)                  by (2)
```

and

```
(ctxLength ctx') ++ (0, lengthSeq seq')
 = (ctxLength ctx) ++ (0, lengthSeq seq)        by (3) and (4)
 = (ctxLength ctx) ++ (0, length (Leaf seq))    by defn of length
 = (ctxLength ctx) ++ (0, length rp)            by (2)
```

and

```
(ctxSize ctx') ++ (0, 1)
   = (ctxSize ctx) ++ (0, 1)                    by (4)
   = (ctxSize ctx) ++ (0, size (Leaf seq))      by defn of size
   = (ctxSize ctx) ++ (0, size rp)              by (2)
```

- Suppose that

```
rp = Cat (lrp, rrp)                              (5)
```

Therefore,

```
(seq', ctx')
   = leftmost (rp, ctx)                         by (1)
   = leftmost (Cat (lrp, rrp), ctx)             by (5)
   = leftmost (lrp, MCLeft (ctx, rrp))          by defn of leftmost
```

and

```
(seq', ctx') = leftmost (lrp, MCLeft (ctx, rrp))     (6)
```

By the induction hypothesis with `lrp`, `MCLeft (ctx, rrp)`, and (6),

```
plug (Leaf seq', ctx')
   = plug (lrp, MCLeft (ctx, rrp))                   (7)
```

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

and

```
(ctxLength ctx') ++ (0, lengthSeq seq')
   =   (ctxLength (MCLeft ctx, rpp)) ++ (0, length lrp)
```
(8)

and

```
(ctxSize ctx') ++ (0, 1)
   =   (ctxSize (MCLeft ctx, rpp)) ++ (0, size lrp)
```
(9)

Hence,

```
plug (Leaf seq', ctx')
   =   plug (lrp, MCLeft (ctx, rpp))      by (7)
   =   plug (Cat (lrp, rrp), ctx)         by defn of plug
   =   plug (rp, ctx)                     by (5)
```

and

```
(ctxLength ctx') ++ (0, lengthSeq seq')
   =   (ctxLength (MCLeft ctx, rpp)) ++ (0, length lrp)
                                          by (8)
   =   (ctxLength ctx) ++ (0, length rrp) ++ (0, length lrp)
                                          by defn of ctxLength
   =   (ctxLength ctx) ++ (0, (length rrp) + (length lrp))
                                          by defn of ++
   =   (ctxLength ctx) ++ (0, (length lrp) + (length rrp))
                                          by defn of +
   =   (ctxLength ctx) ++ (0, length (Cat (lrp, rrp)))
                                          by defn of length
   =   (ctxLength ctx) ++ (0, length rp)  by (5)
```

and

```
(ctxSize ctx') ++ (0, 1)
   =   (ctxSize (MCLeft ctx, rpp)) ++ (0, size lrp)
                                          by (9)
   =   (ctxSize ctx) ++ (0, size rrp) ++ (0, size lrp)
                                          by defn of ctxSize
   =   (ctxSize ctx) ++ (0, (size rrp) + (size lrp))
                                          by defn of ++
   =   (ctxSize ctx) ++ (0, (size lrp) + (size rrp))
                                          by defn of +
   =   (ctxSize ctx) ++ (0, size (Cat (lrp, rrp)))
                                          by defn of size
   =   (ctxSize ctx) ++ (0, size rp)      by (5)
```

□

Similarly, the `next` operation preserves the represented rope as well as its length and size.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

Lemma 3. For any rope `rp` *and any context* `ctx`, *if* `next (rp, ctx)` *returns* `Done rp'`, then

```
rp'  =  plug (rp, ctx)
```

and if it returns `More (seq', ctx')`, then

```
plug (Leaf seq', ctx')  =  plug (rp, ctx)
```

and

```
(ctxLength ctx') ++ (0, lengthSeq seq')
   =  (ctxLength ctx) ++ (length rp, 0)
```

and

```
(ctxSize ctx') ++ (0, 1)  =  (ctxSize ctx) ++ (size rp, 0)
```

Proof
The proof is by structural induction on `ctx`.

- Suppose that

  ```
  ctx  =  MCTop                                   (1)
  ```

  Hence,

  ```
  next (rp, ctx)
     =  next (rp, MCTop)      by (1)
     =  Done rp               by defn of next
  ```

  and, furthermore,

  ```
  rp
     =  plug (rp, MCTop)      by defn of plug
     =  plug (rp, ctx)        by (1)
  ```

  as required when `next (rp, ctx)` returns `Done rp`.
- Suppose that

  ```
  ctx  =  MCLeft (ctx', rrp)                      (2)
  ```

  Hence,

  ```
  next (rp, ctx)
     =  next (rp, MCLeft (ctx', rrp))      by (2)
     =  More (seq'', ctx'')                by defn of next
  ```

  where

  ```
  (seq'', ctx'')  =  leftmost (rrp, MCRight (rp, ctx'))    (3)
  ```

  By Lemma 2 with `rrp`, `MCRight (rp, ctx')`, and (3),

  ```
  plug (Leaf seq'', ctx'')
     =  plug (rrp, MCRight (rp, ctx'))                     (4)
  ```

  and

  ```
  (ctxLength ctx'') ++ (0, lengthSeq seq'')
     =  (ctxLength (MCRight (rp, ctx'))) ++ (0, length rrp)   (5)
  ```

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

and

```
(ctxSize ctx'') ++ (0, 1)
   =   (ctxSize (MCRight (rp, ctx''))) ++ (0, size rrp)
```
(6)

Furthermore,

```
plug (Leaf seq'', ctx'')
   =   plug (rrp, MCRight (rp, ctx''))        by (4)
   =   plug (Cat (rp, rrp), ctx')             by defn of plug
   =   plug (rp, MCLeft (ctx', rrp))          by defn of plug
```

and

```
(ctxLength ctx'') ++ (0, lengthSeq seq'')
   =   (ctxLength (MCRight (rp, ctx''))) ++ (0, length rrp)
                                              by (5)
   =   (ctxLength ctx') ++ (length rp, 0) ++ (0, length rrp)
                                              by defn of ctxLength
   =   (ctxLength ctx') ++ (0, length rrp) ++ (length rp, 0)
                                              by defn of ++
   =   (ctxLength (MCLeft (ctx', rrp))) ++ (length rp, 0)
                                              by defn of ctxLength
   =   (ctxLength ctx) ++ (length rp, 0)      by (2)
```

and

```
(ctxSize ctx'') ++ (0, 1)
   =   (ctxSize (MCRight (rp, ctx''))) ++ (0, size rrp)
                                              by (6)
   =   (ctxSize ctx') ++ (size rp, 0) ++ (0, size rrp)
                                              by defn of ctxSize
   =   (ctxSize ctx') ++ (0, size rrp) ++ (size rp, 0)
                                              by defn of ++
   =   (ctxsize (MCLeft (ctx', rrp))) ++ (size rp, 0)
                                              by defn of ctxSize
   =   (ctxSize ctx) ++ (size rp, 0)          by (2)
```

as required when `next (rp, ctx)` returns `More (seq'', ctx'')`.

- Suppose that

```
ctx  =  MCRight (lrp, ctx')
```
(7)

Hence,

```
next (rp, ctx)
   =   next (rp, MCRight (lrp, ctx''))        by (7)
   =   next (Cat (lrp, rp), ctx')             by defn of next
```

and

```
next (rp, ctx)  =  next (Cat (lrp, rp), ctx')
```
(8)

Proceed by cases on the result of `next (Cat (lrp, rp), ctx')`.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

— Suppose that the result is `Done rp''`.
Therefore,

```
next (Cat (lrp, rp), ctx')  =  Done rp''        (9)
```

By the induction hypothesis with `Cat (lrp, rp)`, `ctx'`, and (9),

```
rp''  =  plug (Cat (lrp, rp), ctx')             (10)
```

Hence,

```
next (rp, ctx)
   =  next (Cat (lrp, rp), ctx')      by (8)
   =  Done rp''                       by (9)
```

and, furthermore,

```
rp''
   =  plug (Cat (lrp, rp), ctx')          by (10)
   =  plug (rp, MCRight (lrp, ctx'))      by defn of plug
   =  plug (rp, ctx)                      by (7)
```

as required when `next (rp, ctx)` returns `Done rp''`.

— Suppose that the result is `More (seq'', ctx'')`.
Therefore,

```
next (Cat (lrp, rp), ctx')  =  More (seq'', ctx'')    (11)
```

By the induction hypothesis with `Cat (lrp, rp)`, `ctx'`, and (11),

```
plug (Leaf seq'', ctx'')  =  plug (Cat (lrp, rp), ctx')
                                                      (12)
```

and

```
(ctxLength ctx'') ++ (0, lengthSeq seq'')
   =  (ctxLength ctx') ++ (length (Cat (lrp, rp)), 0)      (13)
```

and

```
(ctxSize ctx'') ++ (0, 1)
   =  (ctxSize ctx') ++ (size (Cat (lrp, rp)), 0)          (14)
```

Hence,

```
next (rp, ctx)
   =  next (Cat (lrp, rp), ctx')      by (8)
   =  More (seq'', ctx'')             by (11)
```

and, furthermore,

```
plug (Leaf seq'', ctx'')
   =  plug (Cat (lrp, rp), ctx')          by (12)
   =  plug (rp, MCRight (lrp, ctx'))      by defn of plug
   =  plug (rp, ctx)                      by (7)
```

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

and

```
(ctxLength ctx'') ++ (0, lengthSeq seq'')
  =   (ctxLength ctx') ++ (length (Cat (lrp, rp)), 0)
                                                  by (13)
  =   (ctxLength ctx') ++ (length lrp + length rp, 0)
                                                  by defn of length
  =   (ctxLength ctx') ++ (length lrp, 0) ++ (length rp, 0)
                                                  by defn of ++
  =   (ctxLength (MCRight (lrp, ctx'))) ++ (length rp, 0)
                                                  by defn of ctxLength
  =   (ctxLength ctx) ++ (length rp, 0)     by (7)
```

and

```
(ctxSize ctx'') ++ (0, 1)
   =   (ctxSize ctx') ++ (size (Cat (lrp, rp)), 0)
                                                  by (14)
   =   (ctxSize ctx') ++ (size lrp + size rp, 0)
                                                  by defn of size
   =   (ctxSize ctx') ++ (size lrp, 0) ++ (size rp, 0)
                                                  by defn of ++
   =   (ctxsize (MCRight (lrp, ctx'))) ++ (size rp, 0)
                                                  by defn of ctxSize
   =   (ctxSize ctx) ++ (size rp, 0)      by (7)
```

as required when `next (rp, ctx)` returns `More (seq'', ctx'')`.

□

The following lemmas and theorem enable us to use this implementation of `mapLTSUntil` in our `mapLTS`.

Lemma 4. For any sequence `seq` and any context `ctx`, *with* `f = fn x => x`,

```
mSeq (seq, ctx)  =  plug (Leaf seq, ctx)
```

Proof

The proof is by strong induction on `snd (ctxSize ctx)`.

The induction hypothesis is

for any `seq'` and `ctx'`

such that `snd (ctxSize ctx') < snd (ctxSize ctx)`,

```
mSeq (seq', ctx')  =  plug (Leaf seq', ctx')
```

Proceed by cases on the result of `next (Leaf seq, ctx)`.

- Suppose that the result is `Done rp'`.
  Therefore,

```
next (Leaf seq, ctx)  =  Done rp'                (1)
```

  By Lemma 3 with `Leaf seq`, `ctx`, and (1)

```
rp'  =  plug (Leaf seq, ctx)                     (2)
```

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

Hence,

```
mSeq (seq, ctx)
   =  rp'                          by defn of mSeq and (1)
   =  plug (Leaf seq, ctx)         by (2)
```

- Suppose that the result is `More (seq', ctx')`.
  Therefore,

```
next (Leaf seq, ctx)  =  More (seq', ctx')          (3)
```

Note that

```
mapSeq (fn x => x) seq'  =  seq'                    (4)
```

is assumed to hold for an implementation of `mapSeq`.
By Lemma 3 with `Leaf seq`, `ctx`, and (3)

```
plug (Leaf seq', ctx')  =  plug (Leaf seq, ctx)     (5)
```

and

```
(ctxLength ctx') ++ (0, lengthSeq seq')
   =  (ctxLength ctx) ++ (length (Leaf seq), 0)     (6)
```

and

```
(ctxSize ctx') ++ (0, 1)
   =  (ctxSize ctx) ++ (size (Leaf seq), 0)         (7)
```

Note that

```
snd (ctxSize ctx)
   =  snd ((ctxSize ctx) ++ (size (Leaf seq), 0))
                                   by defn of snd and ++
   =  snd ((ctxSize ctx') ++ (0, 1))   by (7)
   =  snd (ctxSize ctx') + 1           by defn of snd and ++
```

Hence,

```
snd (ctxSize ctx')  <  snd (ctxSize ctx)            (8)
```

By the induction hypothesis with `seq'`, `ctx'`, and (8),

```
mSeq (seq', ctx')  =  plug (Leaf seq', ctx')        (9)
```

Hence,

```
mSeq (seq, ctx)
   =  mSeq (mapSeq (fn x => x) seq', ctx')
                                   by defn of mSeq and (3)
   =  mSeq (seq', ctx')            by (4)
   =  plug (Leaf seq', ctx')       by (9)
   =  plug (Leaf seq, ctx)         by (5)
```

□

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

Lemma 5. For any sequence `seq`, any context `ctx`, any predicate `cond`, with `f = fn x => x`, if `m (seq, ctx)` returns `Done rp'`, then

```
rp'  =  plug (Leaf seq, ctx)
```

and if it returns `More cur'`, then

```
root cur'  =  plug (Leaf seq, ctx)
```

and

```
snd (ctxLength ctx) + (lengthSeq seq)
   ⩾  snd (curLength cur')
```

and

```
snd (curLength cur')  ⩾  2
```

Proof
The proof is by strong induction on `snd (ctxSize ctx)`.
The induction hypothesis is:

for any `seq'` and `ctx'`
such that `snd (ctxSize ctx') < snd (ctxSize ctx)`,
if `m (seq', ctx')` returns `Done rp''`,
then

```
rp''  =  plug (Leaf seq', ctx')
```

and if it returns `More cur''`,
then

```
root cur''  =  plug (Leaf seq', ctx')
```

and

```
snd (ctxLength ctx') + (lengthSeq seq')
   ⩾  snd (curLength cur'')
```

and

```
snd (curLength cur'')  ⩾  2
```

Proceed by cases on the result of `mapUntilSeq cond (fn x => x) seq`.

- Suppose that the result is `Done pseq'`.
  Therefore,

  ```
  mapUntilSeq cond (fn x => x) seq  =  Done pseq'        (1)
  ```

  By Property 3 with `seq`, `cond` and (1),

  ```
  pseq'  =  seq                                          (2)
  ```

  Proceed by cases on the result of `next (Leaf pseq', ctx)`.

  — Suppose that the result is `Done rp'`.
  Therefore,

  ```
  next (Leaf pseq', ctx)  =  Done rp'                    (3)
  ```

  By Lemma 3 with `Leaf pseq'`, `ctx`, and (3),

  ```
  rp'  =  plug (Leaf pseq', ctx)                         (4)
  ```

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

Hence,

```
m (seq, ctx)
   =  Done rp'        by defn of m, (1), and (3)
```

and, furthermore,

```
rp'
   =  plug (Leaf pseq', ctx)      by (4)
   =  plug (Leaf seq, ctx)        by (2)
```

as required when `m (seq, ctx)` returns `Done rp'`.

— Suppose that the result is `More (seq', ctx')`.
Therefore,

```
next (Leaf pseq', ctx)  =  More (seq', ctx')        (5)
```

By Lemma 3 with `Leaf pseq'`, `ctx`, and (5),

```
plug (Leaf seq', ctx')  =  plug (Leaf pseq', ctx)   (6)
```

and

```
(ctxLength ctx') ++ (0, lengthSeq seq')
   =  (ctxLength ctx) ++ (length (Leaf pseq'), 0)   (7)
```

and

```
(ctxSize ctx') ++ (0, 1)
   =  (ctxSize ctx) ++ (size (Leaf pseq'), 0)       (8)
```

```
snd (ctxSize ctx)
   =  snd ((ctxSize ctx) ++ (size (Leaf pseq'), 0))
                                        by defn of snd and ++
   =  snd ((ctxSize ctx') ++ (0, 1))    by (8)
   =  snd (ctxSize ctx') + 1            by defn of snd and ++
```

Hence,

```
snd (ctxSize ctx')  <  snd (ctxSize ctx)            (9)
```

Proceed by cases on the result of `m (seq', ctx')`.

– Suppose that the result is `Done rp''`.
Therefore,

```
m (seq', ctx')  =  Done rp''                        (10)
```

By the induction hypothesis with `seq'`, `ctx'`, and (9),

```
rp''  =  plug (Leaf seq', ctx')                     (11)
```

Hence,

```
m (seq, ctx)
   =  m (seq', ctx')        by defn of m, (1), and (5)
   =  Done rp''             by (10)
```

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

and, furthermore,

```
rp''
   =  plug (Leaf seq', ctx')      by (11)
   =  plug (Leaf pseq', ctx)      by (6)
   =  plug (Leaf seq, ctx)        by (2)
```

as required when `m (seq, ctx)` returns `Done rp''`.

– Suppose that the result is `More cur''`.
Therefore,

$$\texttt{m (seq', ctx')} \;=\; \texttt{More cur''} \tag{12}$$

By the induction hypothesis with `seq'`, `ctx'`, and (9),

$$\texttt{root cur''} \;=\; \texttt{plug (Leaf seq', ctx')} \tag{13}$$

and

$$\texttt{snd (ctxLength ctx') + (lengthSeq seq')} \;\geqslant\; \texttt{snd (curLength cur'')} \tag{14}$$

and

$$\texttt{snd (curLength cur'')} \;\geqslant\; 2 \tag{15}$$

Hence,

```
m (seq, ctx)
   =  m (seq', ctx')      by defn of m, (1), and (5)
   =  More cur''          by (12)
```

and, furthermore,

```
root cur''
   =  plug (Leaf seq', ctx')      by (13)
   =  plug (Leaf pseq', ctx)      by (6)
   =  plug (Leaf seq, ctx)        by (2)
```

and

```
snd (ctxLength ctx) + (lengthSeq seq)
  ⩾  snd (ctxLength ctx)
  =  snd ((ctxLength ctx) ++ (length (Leaf pseq'), 0))
                                  by defn of snd and ++
  =  snd ((ctxLength ctx') ++ (0, lengthSeq'))
                                  by (7)
  =  snd (ctxLength ctx') + (lengthSeq seq')
                                  by defn of snd and ++
  ⩾  snd (curLength cur'')        by (14)
```

and

```
snd (curLength cur'')
   ⩾  2        by (15)
```

as required when `m (seq, ctx)` returns `More cur''`.

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

- Suppose that the result is `More (useq', pseq')`.
  Therefore,

  ```
  mapUntilSeq cond (fn x => x) seq  =  More (useq', pseq')   (16)
  ```

  By Property 3 with `seq`, `cond` and (16),

  ```
  joinSeq (pseq', useq')  =  seq                              (17)
  ```

  Proceed by cases on the result of `snd (curLength ((pseq',useq'), ctx)) >= 2`.

  — Suppose that the result is `true`.
  Therefore,

  ```
  snd (curLength ((pseq', useq'), ctx))  ⩾  2                 (18)
  ```

  Hence,

  ```
  m (seq, ctx)
     =  More ((pseq', useq'), ctx)
              by defn of m, (16), and (18)
  ```

  and

  ```
  root ((pseq', useq'), ctx)
   =  plug (Leaf (joinSeq (pseq', useq')), ctx)   by defn of root
   =  plug (Leaf seq, ctx)                        by (17)
  ```

  and

  ```
  snd (ctxLength ctx) + (lengthSeq seq)
  = snd (ctxLength ctx) + (lengthSeq (joinSeq (pseq', useq')))
                                                  by (17)
  = snd (ctxLength ctx) + (lengthSeq pseq') + (lengthSeq useq')
                                                  by defn of lengthSeq
                                                  and joinSeq
  ⩾ snd (ctxLength ctx) + (lengthSeq useq')
  = snd ((ctxLength ctx) ++ (lengthSeq pseq', lengthSeq useq'))
                                                  by defn of snd and ++
  = snd (curLength ((pseq', useq'), ctx))         by defn of curLength
  ```

  and

  ```
  snd (curLength ((pseq', useq'), ctx))
     ⩾  2        by (18)
  ```

  as required when `m (seq, ctx)` returns `More ((pseq', useq'), ctx)`.

  — Suppose that the result is `false`.
  Therefore,

  ```
  snd (curLength ((pseq', useq'), ctx))  <  2                 (19)
  ```

  Note that

  ```
  mapSeq (fn x => x) useq'  =  useq'                          (20)
  ```

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

is assumed to hold for an implementation of mapSeq.
By Lemma 4 with seq, ctx,

```
mSeq (seq, ctx)  =  plug (Leaf seq, ctx)        (21)
```

Hence,

```
m (seq, ctx)
   =  Done (mSeq (joinSeq (pseq', mapSeq f useq'), ctx))
        by defn of m, (16), and (19)
```

and

```
mSeq (joinSeq (pseq', mapSeq f useq'), ctx)
   =  mSeq (joinSeq (pseq', useq'), ctx)      by (20)
   =  mSeq (seq, ctx)                         by (17)
   =  plug (Leaf seq, ctx)                    by (21)
```

as required when m (seq, ctx) returns Done (mSeq (joinSeq (pseq', mapSeq f useq'), ctx)).

□

Theorem 3 (mapLTSUntil is well-behaved). Property 2 holds for the implementation of mapLTSUntil.

For any rope rp and any predicate cond, if mapLTSUntil cond (fn x => x) rp returns Done rp', then

```
rp'  =  rp
```

and if it returns More cur', then

```
root cur'  =  rp
```

and

```
length rp  ⩾  lengthRight cur'
```

and

```
lengthRight cur'  ⩾  2
```

Proof
Note that

```
mapLTSUntil cond (fn x => x) rp
   =  m (seq, ctx)      by defn of mapLTSUntil
```

where

```
(seq, ctx)  =  leftmost (rp, MCTop)        (1)
```

By Lemma 2 with rp, MCTop, and (1),

```
plug (Leaf seq, ctx)  =  plug (rp, MCTop)        (2)
```

and

```
(ctxLength ctx) ++ (0, lengthSeq seq)
   =  (ctxLength MCTop) ++ (0, length rp)        (3)
```

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

and

```
(ctxSize ctx) ++ (0, 1)  =  (ctxSize MCTop) ++ (0, size rp)    (4)
```

Proceed by cases on the result of `m (seq, ctx)`.

- Suppose that the result is `Done rp'`.
  Therefore,

  ```
  m (seq, ctx)  =  Done rp'                                    (5)
  ```

  By Lemma 5 with `ctx`, `cond`, and (5),

  ```
  rp'  =  plug (Leaf seq, ctx)                                 (6)
  ```

  Hence,

  ```
  mapLTSUntil cond (fn x => x) rp
    =  m (seq, ctx)        by defn of mapLTSUntil
    =  Done rp'            by (5)
  ```

  and

  ```
  rp'
    =  plug (Leaf seq, ctx)      by (6)
    =  plug (rp, MCTop)          by (2)
    =  rp                        by defn of plug
  ```

  as required when `mapLTSUntil cond (fn x => x) rp` returns `Done rp'`.
- Suppose that the result is `More cur'`.
  Therefore,

  ```
  m (seq, ctx)  =  More cur'                                   (7)
  ```

  By Lemma 5 with `ctx`, `cond`, and (7),

  ```
  root cur'  =  plug (Leaf seq, ctx)                           (8)
  ```

  and

  ```
  snd (ctxLength ctx) + (lengthSeq seq)  ⩾  snd (curLength cur')
                                                               (9)
  ```

  and

  ```
  snd (curLength cur')  ⩾  2                                   (10)
  ```

  Hence,

  ```
  mapLTSUntil cond (fn x => x) rp
    =  m (seq, ctx)        by defn of mapLTSUntil
    =  More cur'           by (7)
  ```

  and

  ```
  root cur'
    =  plug (Leaf seq, ctx)      by (8)
    =  plug (rp, MCTop)          by (2)
    =  rp                        by defn of plug
  ```

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

and

```
length rp
   =   snd((0, 0) ++ (0, length rp))
                                    by defn of snd and ++
   =   snd((ctxLength MCTop) ++ (0, length rp))
                                    by defn of ctxLength
   =   snd((ctxLength ctx) ++ (0, lengthSeq seq))
                                    by (3)
   =   snd(ctxLength ctx) + (lengthSeq seq)
                                    by defn of snd and ++
   ⩾   snd (curLength cur')         by (9)
```

and

```
snd (curLength cur')
   ⩾  2        by (10)
```

as required when `mapLTSUntil cond (fn x => x) rp` returns `More cur'`.

□

A.3 split and join are well-behaved

The well-behavedness of our `split` and `join` operations will depend on the property that the `zipCursor` operation is a left-inverse of the `unzipCursor` operation.

Lemma 6. For any (zipped) context `ctx`,

```
zipCtx (unzipCtx ctx)  =  ctx
```

Proof
By structural induction on `ctx`. □

Lemma 7. For any (zipped) cursor `cur`,

```
zipCursor (unzipCursor cur)  =  cur
```

Proof
By Lemma 6. □

We have that `divideRopes (rps, n)` returns `(rps1, rp, k, rps2)` such that `rps1 @ [rp] @ rps2` is equal to `rps` and `k` is the index in `rp` at which the `n`th element of `rps` is found.

Lemma 8. For any nonempty list of ropes `rps` and any integer `n`, such that $0 \leqslant$ `n` and `n` $\leqslant$ `sumLengths rps`, if `divideRopes (rps, n)` returns `(rps1, rp, k, rps2)`, then

```
rps1 @ [rp] @ rps2  =  rps
```

and

```
length rp  ⩽  k
```

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

and

```
k = n - (sumLengths rps1)
```

Proof
By structural induction on `rps`. □

We have that the `root` operation is a left-inverse of the `splitAtAsCur` operation and also that `splitAtAsCur` returns a cursor in which the "hole" occurs between the `n`th and `n`+1st elements of the rope `rp`.

Lemma 9. For any rope `rp` and any integer `n`, such that $0 \leqslant$ `n` and `n` $\leqslant$ `length rp`, if `splitAtAsCur (rp, n)` returns `cur`, then

```
root cur = rp
```

and

```
curLength cur = (n, (length rp) - n)
```

Proof
By structural induction on `rp`. □

Finally, we have that the `decodeRope` operation is a left-inverse of the `encodeRopes` operation.

Lemma 10. For any nonempty list of ropes `rps`,

```
decodeRope (encodeRope rps) = rps
```

Proof
By structural induction on `rps`. □

With the following theorem, we can conclude that our implementation of `split` and `join` can be used safely by `mapLTS`.

Theorem 4 (`split` and `join` are well-behaved). Property 1 holds for the implementations of `split` and `join`.

For any cursor `cur`, if `split cur` returns `(rp1, rp2, reb)`, then

```
join (rp1, rp2, reb) = cur
```

and

```
length rp1 = (lengthRight cur) div 2
```

and

```
length rp2 = (lengthRight cur) - ((lengthRight cur) div 2)
```

Proof
By Lemmas 7, 8, 9, and 10. □

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

A.4 mapLTS is shape preserving

Theorem 1 (mapLTS is shape preserving). For any rope rp,

```
mapLTS (fn x => x) rp  =  rp
```

Proof
The proof is by strong induction on `length rp`.
The induction hypothesis is:
for any `rp'`
such that `length rp' < length rp`,

```
mapLTS (fn x => x) rp'  =  rp'
```

Proceed by cases on the result of `mapLTSUntil hungryProcs (fn x => x) rp`.

- Suppose that the result is `Done rp'`.
  Therefore,

  ```
  mapLTSUntil hungryProcs (fn x => x) rp  =  Done rp'    (1)
  ```

  By Property 2 with `rp` and `hungryProcs` and (1),

  ```
  rp'  =  rp    (2)
  ```

  Therefore,

  ```
  mapLTS (fn x => x) rp
    =  rp'      by defn of mapLTS and (1)
    =  rp       by (2)
  ```

- Suppose that the result is `More cur'`.
  Therefore,

  ```
  mapLTSUntil hungryProcs (fn x => x) rp  =  More cur'    (3)
  ```

  By Property 2 with `rp` and `hungryProcs` and (3),

  ```
  root cur'  =  rp    (4)
  ```

  and

  ```
  length rp  ⩾  lengthRight cur'    (5)
  ```

  and

  ```
  lengthRight cur'  ⩾  2    (6)
  ```

  Note that `lengthRight cur' ⩾ 2` implies that

  ```
  lengthRight cur'  >  (lengthRight cur') div 2    (7)
  ```

  and

  ```
  (lengthRight cur') div 2  ⩾  1    (8)
  ```

  By Property 1,

  ```
  join (rp1, rp2, reb) = cur'    (9)
  ```

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

and

```
length rp1  =  (lengthRight cur') div 2
```
(10)

and

```
length rp2
   =  (lengthRight cur') - ((lengthRight cur') div 2)
```
(11)

Note that

```
length rp1
   =  (lengthRight cur') div 2      by (10)
   <  lengthRight cur'              by (7)
   ⩽  length rp                     by (5)
```

Hence,

```
length rp1  <  length rp
```
(12)

By the induction hypothesis with `rp1` and (12),

```
mapLTS (fn x => x) rp1  =  rp1
```
(13)

Note that

```
length rp2
   =  (lengthRight cur') - ((lengthRight cur') div 2)
                                    by (11)
   ⩽  (lengthRight cur') - 1        by (8)
   <  lengthRight cur'
   ⩽  length rp                     by (5)
```

Hence,

```
length rp2  <  length rp
```
(14)

By the induction hypothesis with `rp2` and (14),

```
mapLTS (fn x => x) rp2  =  rp2
```
(15)

Note that, by the definitions of `mapLTS` and `par2`,

```
rp1'  =  mapLTS (fn x => x) rp1
```
(16)

and

```
rp2'  =  mapLTS (fn x => x) r\p2
```
(17)

Therefore,

```
mapLTS (fn x => x) rp
   =  root (join (rp1', rp2', reb))
                                    by defn of mapLTS and (3)
   =  root (join (rp1, rp2', reb))  by (16) and (13)
   =  root (join (rp1, rp2, reb))   by (17) and (15)
   =  root cur'                     by (9)
   =  rp                            by (4)
```

□

https://doi.org/10.1017/S0956796812000172 Published online by Cambridge University Press

Acknowledgments

We would like to thank the anonymous referees and editor for their helpful comments and suggestions. This work was performed in part while John Reppy was serving at the National Science Foundation. It was also supported in part by National Science Foundation Grants CCF-0811389, CCF-0811419, and CCF-1010568. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation or the US Government.

References

Appel, Andrew W. (1989) Simple generational garbage collection and fast allocation. *Softw. Pract. Exp.* **19**(2), 171–183.

Appel, Andrew W. (1992) *Compiling with Continuations.* Cambridge, UK: Cambridge University Press.

Barnes, J. & Hut, P. (1986) A hierarchical $O(N \log N)$ force calculation algorithm. *Nature* **324**(Dec.), 446–449.

Blelloch, Guy E. (1990a, Nov.) *Prefix Sums and Their Applications.* Tech. rept. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Blelloch, Guy E. (1990b) *Vector Models for Data-Parallel Computing.* Cambridge, MA: MIT Press.

Blelloch, Guy E. (1996) Programming parallel algorithms. *Commun. ACM* **39**(3), 85–97.

Blelloch, Guy E. & Greiner, J. (1996) A provable time and space-efficient implementation of NESL. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, PA, USA. New York, NY: ACM, pp. 213–225.

Blelloch, Guy E., Chatterjee, S., Hardwick, Jonathan C., Sipelstein, J. & Zagha, M. (1994) Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.* **21**(1), 4–14.

Blumofe, Robert D. & Leiserson, Charles E. (1999) Scheduling multi-threaded computations by work stealing. *J. ACM* **46**(5), 720–748.

Boehm, Hans-J., Atkinson, R. & Plass, M. (1995) Ropes: An alternative to strings. *Softw. Pract. Exp.* **25**(12), 1315–1330.

Burton, F. Warren & Sleep, M. Ronan. (1981) Executing functional programs on a virtual tree of processors. In *Functional Programming Languages and Computer Architecture* (FPCA '81). New York, NY: ACM, pp. 187–194.

Carver, T. (2010, Mar) *Magny-Cours and Direct Connect Architecture 2.0.* Accessed January 2012. Available at: http://developer.amd.com/documentation/articles/pages/Magny-Cours-Direct-Connect-Architecture-2.0.aspx.

Chakravarty, Manuel M. T., Leshchinskiy, R., Peyton Jones, S. & Keller, G. (2008) Partial vectorisation of Haskell programs. *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, Nice, France. New York, NY: ACM.

Chakravarty, Manuel M. T., Leshchinskiy, R., Peyton Jones, S., Keller, G. & Marlow, S. (2007) Data parallel Haskell: A status report. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming,* San Francisco, CA, USA. New York, NY: ACM, pp. 10–18.

Chatterjee, S. (1993) Compiling nested data-parallel programs for shared-memory multiprocessors. *ACM Trans. Program. Lang. Syst.* **15**(3), 400–462.

Conway, P., Kalyanasundharam, N., Donley, G., Lepak, K. & Hughes, B. (2010) Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro* **30**, 16–29.

Fluet, M., Rainey, M., Reppy, J., Shaw, A. & Xiao, Y. (2007a) Manticore: A heterogeneous parallel language. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming,* San Francisco, CA, USA. New York, NY: ACM, pp. 37–44.

Fluet, M., Ford, N., Rainey, M., Reppy, J., Shaw, A. & Xiao, Y. (2007b) Status report: The manticore project. In *Proceedings of the 2007 ACM SIGPLAN Workshop on ML*, Victoria, BC, Canada. New York, NY: ACM, pp. 15–24.

Fluet, M., Rainey, M., Reppy, J. & Shaw, A. (2008a) Implicitly threaded parallelism in Manticore. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, Victoria, BC, Canada. New York, NY: ACM, pp. 119–130.

Fluet, M., Rainey, M. & Reppy, J. (2008b) A scheduling framework for general-purpose parallel languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, Victoria, BC, Canada. New York, NY: ACM, pp. 241–252.

Frigo, M., Leiserson, Charles E. & Randall, Keith H. (1998, Jun) The implementation of the Cilk-5 multithreaded language. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '98), Montreal, Canada. New York, NY: ACM, pp. 212–223.

Ghuloum, A., Sprangle, E., Fang, J., Wu, G. & Zhou, X. (2007, Oct) *Ct: A Flexible Parallel Programming Model for Tera-Scale Architectures*. Tech. rept. Intel. Accessed January 2012. Available at: http://techresearch.intel.com/UserFiles/en-us/File/terascale/Whitepaper-Ct.pdf.

Halstead Jr., Robert H. (1984) Implementation of multilisp: LISP on a multiprocessor. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*. New York, NY: ACM, pp. 9–17.

Hinze, R. & Paterson, R. (2006) Finger trees: A simple general-purpose data structure. *J. Funct. Program.* **16**(2), 197–217.

Huet, G. (1997) The zipper. *J. Funct. Program.* **7**(5), 549–554.

Intel (2008) *Intel Threading Building Blocks Reference Manual*. Santa Clara, CA: Intel Corporation. Available at: http://www.threadingbuildingblocks.org/.

Keller, G. (1999) *Transformation-Based Implementation of Nested Data Parallelism for Distributed Memory Machines*. PhD thesis, Technische Universität Berlin, Berlin, Germany.

Leiserson, Charles E. (2009) The Cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, San Francisco, CA, USA. New York, NY: ACM, pp. 522–527.

Leshchinskiy, R. (2005) *Higher-Order Nested Data Parallelism: Semantics and Implementation.* PhD thesis, Technische Universität Berlin, Berlin, Germany.

Loidl, H.-W. & Hammond, K. (1995) On the granularity of divide-and-conquer parallelism. In *Proceedings of the Glasgow Workshop on Functional Programming*, Ullapool, Scotland. New York: Springer-Verlag, pp. 8–10.

Lopez, P., Hermenegildo, M. & Debray, S. (1996) A methodology for granularity-based control of parallelism in logic programs. *J. Symb. Comput.* **21**(Jun), 715–734.

McBride, C. (2008) Clowns to the left of me, jokers to the right (pearl): Dissecting data structures. In *Conference Record of the 35th Annual ACM Symposium on Principles of Programming Languages* (POPL '08), San Francisco, CA, USA. New York, NY: ACM, pp. 287–295.

Milner, R., Tofte, M., Harper, R. & MacQueen, D. (1997) *The Definition of Standard ML (revised)*. Cambridge, MA: MIT Press.

MLton (n.d.) *The MLton Standard ML Compiler*. Accessed January 2011. Available at: http://mlton.org.

Narlikar, Girija J. & Blelloch, Guy E. (1999) Space-efficient scheduling of nested parallelism. *ACM Trans. Program. Lang. Syst.* **21**(1), 138–173.

Nikhil, Rishiyur S. (1991, Jul) *ID Language Reference Manual*. Cambridge, MA: Laboratory for Computer Science, MIT.

Peyton Jones, S., Leshchinskiy, R., Keller, G. & Chakravarty, Manuel M. T. (2008) Harnessing the multicores: Nested data parallelism in Haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*. New York, NY: Springer-Verlag, pp. 138–138.

Plummer, H. C. (1911) On the problem of distribution in globular star clusters. *Mon. Not. R. Astron. Soc.* **71**(Mar), 460–470.

Rainey, M. (2007, Jan) *The Manticore Runtime Model*. M.Phil. thesis, University of Chicago, Illinois, USA. Available at: http://manticore.cs.uchicago.edu.

Rainey, M. (2009) Prototyping nested schedulers. In *Semantics Engineering with Plt Redex*, Felleisen, M., Findler, R. & Flatt, M. (eds.). Cambridge, MA: MIT Press.

Robison, A., Voss, M. & Kukanov, A. (2008) Optimization via reflection on work stealing in TBB.*Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. Los Alamitos, CA: IEEE Computer Society Press.

Scandal Project (n.d.) *A Library of Parallel Algorithms Written NESL*. Accessed January 2012. Available at: http://www.cs.cmu.edu/ scandal/nesl/algorithms.html.

So, B., Ghuloum, A. & Wu, Y. (2006) Optimizing data parallel operations on many-core platforms. *First Workshop on Software Tools for Multi-Core Systems*, (STMCS), Manhattan, NY.

Tick, E. & Zhong, X. (1993) A compile-time granularity analysis algorithm and its performance evaluation. In *Selected Papers of the International Conference on Fifth Generation Computer Systems* (FGCS '92). New York, NY: Springer-Verlag, pp. 271–295.

Trinder, Philip W., Hammond, K., Loidl, H.-W. & Peyton Jones, S. L. (1998) Algorithm + strategy = parallelism. *J. Funct. Program.* **8**(1), 23–60.

Tzannes, A., Caragea, G. C., Barua, R. & Vishkin, U. (2010) Lazy binary-splitting: A run-time adaptive work-stealing scheduler. In *Proceedings of the 2010 ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*. New York, NY: ACM, pp. 179–190.

Weeks, S. (2006, Sep) *Whole Program Compilation in MLton*. Invited talk at ML'06 workshop. Accessed January 2011. Slides available at: http://mlton.org/pages/References/attachments/060916-mlton.pdf.