Lazy Tree Splitting

Lars Bergstrom Mike Rainey
John Reppy Adam Shaw
University of Chicago
{larsberg,mrainey,jhr,ams}@cs.uchicago.edu

Matthew Fluet
Rochester Institute of Technology
mtf@cs.rit.edu

Abstract

Nested data-parallelism (NDP) 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 common NDP operations like parallel maps, filters, and sum-like reductions. In this paper, we describe the implementation of NDP in Parallel ML (PML), part of the Manticore project. Managing the parallel decomposition of work is one of the main challenges of implementing NDP. If the decomposition creates too many small chunks of work, performance will be eroded by too much parallel overhead. If, on the other hand, there are too few large chunks of work, there will be too much sequential processing and processors will sit idle.

Recently the technique of Lazy Binary Splitting was proposed for dynamic parallel decomposition of work on flat arrays, with promising results. We adapt Lazy Binary Splitting to parallel processing of binary trees, which we use to represent parallel arrays in PML. We call our technique *Lazy Tree Splitting* (LTS). One of its main advantages is its performance robustness: per-program tuning is not required to achieve good performance across varying platforms. We describe LTS-based implementations of standard NDP operations, and we present experimental data demonstrating the scalability of LTS across a range of benchmarks.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages; D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Languages, Performance

Keywords nested-data-parallel languages, scheduling, compilers, and run-time systems

1. Introduction

Nested data-parallelism (NDP) [BCH+94] 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 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.*, subarrays may have different lengths. NDP programming is supported by a number of different parallel programming languages [CLP+07, GSF+07], including our own *Parallel ML* (PML) [FRRS08].

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 TBB [Int08, RVK08] and Cilk++ [Lei09]. 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 [TZ93]. 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 [TCBV10]. 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* [BAP95], a balanced binary-tree representation of sequences, as the underlying representation of parallel arrays. Our implementation is purely functional in that it works with immutable structures, although some imperative techniques are used under the hood for scheduling.

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

[1] We do not address *flattening* (or *vectorizing*) [Kel99, Les05] transformations here, since the techniques of this paper apply equally well to flattened or non-flattened programs.

LTS 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.

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 [FRR+07, FFR+07, FRRS08].

2.1 Programming model

PML is the programming language supported by the Manticore system.[2] Our programming model is based on a strict, but mutation-free, functional language (a subset of Standard ML [MTHM97]), 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 will 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 [Ble90b, Ble96].

PML 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 the following interface:

```
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
```

The function `range` generates an array of the integers between its two arguments, `mapP`, `filterP`, and `reduceP` have their usual meaning, except that they may be evaluated in parallel, and `scanP` produces a prefix scan of the 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 across the `trace(x,y)` expression (and, therefore, across the inner `mapP` expression) may differ significantly depending on the layout of the scene. A robust technique for balancing the parallel execution of this unbalanced computation is the primary contribution of this paper.

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 PML. Further details on our system may be found elsewhere [FRR08, Rai09, Rai07].

A task-scheduling policy determines the order in which tasks execute and the assignments from tasks to processors. Our LTS is built on top of on a particular task-scheduling policy called *work stealing* [BS81, Hal84]. 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 [BL99, FLR98]. Each worker maintains a deque (double-ended queue) 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 par2 : (unit -> 'a) * (unit -> 'b) -> 'a * 'b
```

When a worker P executes `par2 (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 `par2` call. The scheduler also provides a generalization of `par2` to a list of thunks.

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

This function can be defined in terms of `par2`, 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 our 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 `seq`s, contiguous arrays of data, at their leaves [BAP95]. For the purposes of this paper, we view the rope type as having the following definition:

```
datatype 'a rope
  = Leaf of 'a seq
  | Cat of 'a rope * 'a rope
```

although in our actual implementation there is extra information in the `Cat` nodes to support balancing. Read from left to right, the

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

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

data elements at the leaves of a rope constitute the data of a parallel array it represents.

Since ropes are 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 [App89], which would be necessary for maintaining an ephemeral data structure. Second, a parallel memory manager, such as the one used by Manticore [FRR08], can avoid making memory management a sequential bottleneck by letting processors allocate and reclaim ropes independently.

As a parallel-array representation, ropes have several weaknesses as opposed 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 may have different shape. In our performance study in Section 5, we find that these weaknesses are not crippling by themselves, yet 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 run-time, a leaf contains a number of elements n such that $0 \leq n \leq M$. In general, rope operations try to keep the size 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.

Relaxing the perfect balance requirement reduces excessive balancing, yet maintains the asymptotic behavior of rope operations. Our rope-balancing policy is a relaxed, parallel version of the sequential policy used by Boehm, *et al.* [BAP95]. The policy of Boehm, *et al.* is as follows. For a given rope r of depth d and length n, the balancing goal is $d \leq \lceil \log_2 n \rceil + 2$. This property is enforced 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).

In our rope-balancing policy, only those ropes that are built serially are balanced by `balance`, *i.e.*, the serial balancing process only ever takes place within a given chunk. There is no explicit guarantee on the balance of a rope containing subropes that are built by different processors. For such a rope, the amount of rope imbalance is proportional to the distribution of work across processors rather than the size of the rope itself. As we discuss in Section 5, across all our benchmarking results, balancing has minimal impact on performance.

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

```
val cat2 : 'a rope * 'a rope -> 'a rope
```

If `cat2` is applied to two small leaves, it may coalesce them into a single larger leaf. Note that `cat2` 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 subropes. For this reason, we provide

```
val split2 : 'a rope -> 'a rope * 'a rope
```

which splits its rope parameter into two subropes such that the size of these ropes differs by at most one. We also define

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

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

We sometimes use

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

which takes a rope r and returns the number of elements stored in the leaves of r. [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. Those 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 strategy apportions chunks that are neither too large nor too small, but are, like Goldilocks's third bowl of porridge, "just right." Some different chunking strategies are considered in the sequel.

3.1 Fragile chunking strategies

A fragile chunking strategy is prone either to creating an excessive number of tasks or to missing significant opportunities for parallelism. Let us consider a two simple strategies, 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 processors, and spawn a task for each chunk. For example, in Figure 1(a), we show the T-ary decomposition version of the rope-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 for i levels of nesting and sufficiently-large ropes, the T-ary decomposition creates $(J \times P)^i$ tasks overall, 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 rope-map operation in Figure 1(b). Recall that the maximum size of a leaf is determined by

[4] In our actual implementation, this operation takes constant time, as we cache lengths in `Cat` nodes.

[5] In this and subsequent examples, we use the function `mapSequential` with type

```
('a -> 'b) -> 'a rope -> 'b rope
```

which is the obvious sequential implementation of the rope-map operation.

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

(a) T-ary decomposition

```
fun mapStructural f rp = (case rp
  of Leaf s => mapSequential f rp
   | Cat (l, r) =>
       Cat (par2 (fn () => mapStructural f l,
                  fn () => mapStructural f r)))
```

(b) structural decomposition

Figure 1. Two fragile implementations of the rope-map operation.

```
fun mapETS SST f rp =
  if length rp <= SST then mapSequential f rp
  else let
    val (l, r) = split2 rp
    in
      cat2 (par2 (fn () => mapETS SST f l,
                  fn () => mapETS SST f r))
    end
```

Figure 2. The ETS implementation of the rope-map operation.

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.

3.2 Eager binary splitting

EBS is a well-known approach that is used by many parallel libraries and languages, including Thread Building Blocks [Int08, RVK08] and Cilk++ [Lei09]. In EBS (and, by extension, eager tree splitting (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.[6] Otherwise, we create two chunks by dividing the elements in half and recursively apply the same process to the two new chunks. For example, in Figure 2, we show the ETS version of the rope-map operation.

EBS has greater flexibility than the T-ary or structural decompositions because EBS allows chunk sizes to be picked manually. But this flexibility is not much of an improvement, because, as is well known [Int08, RVK08, TCBV10], 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 efficiency* as a function of SST. The parallel efficiency is the sixteen-processor speedup divided by sixteen times 100, where the baseline time for the speedup is taken from the sequential evaluation. For example, 100% parallel efficiency represents perfect linear speedup and 6.25% parallel efficiency represents almost no speedup. The results demonstrate that there is no SST that is optimal for every program and furthermore that a poor SST is far from optimal.

[6] In TBB, if SST is unspecified, the default is $SST = 1$, whereas Cilk++ only uses $SST = 1$.

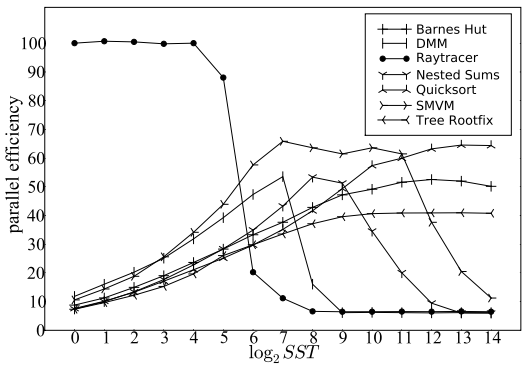


Figure 3. Parallel efficiency is sensitive to SST (16 processors).

The Raytracer benchmark demonstrates, in particular, how fragile ETS can be with respect to nesting and to relatively small ropes. Raytracer loses 80% of its speedup as SST is changed from 32 to 128. The two-dimensional output of the program is a 256×256 rope of ropes, representing the pixels of a square image. When $SST = 128$, Raytracer has just two chunks it can process in parallel: the first 128 rows and the second. 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. It is a break with the nested-data-parallel programming style.

Recall that task execution times can vary unpredictably. Chunking strategies 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 strategy would be able to adapt dynamically to the current state of load balance across processors.

3.3 Lazy binary splitting

The LBS strategy of Tzannes, *et al.* [TCBV10] is a promising alternative to the other strategies because it has good adaptivity to dynamic load balance. 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.

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

LBS 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.* [TCBV10] 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 accurate estimate of whether splitting is profitable.

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 the other processors. Because 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.

Because 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 for ropes

LTS operations are not as easy to implement as ETS operations, because, during the execution of a given LTS operation, a split can occur while processing *any* rope element. This section presents implementations of five important LTS operations. The technique we use is based on Huet's zipper technique [Hue97] and a new technique we call *splitting a context*. We first look in detail at the LTS version of map (`mapLTS`) because its implementation offers a simple survey of our techniques. Then we summarize implementations of the additional operations.

4.1 Implementing `mapLTS`

Structural recursion, on its own, offers 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 handle on either the processed portion of the rope or the unprocessed remainder of the rope; it can only see the current substructure. We need to be able to "step through" a traversal in such a way that we can, at any moment, pause the traversal, reconstruct processed

```
fun mapLTS f rp =
  if length rp <= 1 then
    mapSequential f rp
  else (case mapUntil hungryProcs f rp
    of More (u, p) => let
         val (u1, u2) = split2 u
         in
           catN (parN [fn () => balance p,
                       fn () => mapLTS f u1,
                       fn () => mapLTS f u2])
         end
     | Done p => balance p)
```

Figure 4. The LTS implementation of the rope-map operation.

results, divide the unprocessed remainder in half, and resume processing at the pause point.

A key piece of our implementation is an internal operation called `mapUntil`. The `mapUntil` operation is capable of pausing its traversal based on a runtime predicate:

```
val mapUntil :
  (unit -> bool) -> ('a -> 'b)
  -> 'a rope
  -> ('a rope * 'b rope, 'b rope) progress
```

The first argument to `mapUntil` is a polling function (*e.g.*, `hungryProcs`); the second argument is the function to be applied to the individual data elements; and the third argument is the input rope. Instead of returning a fully processed rope, `mapUntil` returns a value of type `('a rope * 'b rope, 'b rope) progress`, where the type constructor `progress` is defined as

```
datatype ('a, 'b) progress
  = More of 'a
  | Done of 'b
```

In the result of `mapUntil`, a value `More (u, p)` represents a partially processed rope where `u` is the unprocessed part and `p` is the processed part; a value `Done p` represents a fully processed rope. The evaluation of `mapUntil 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. Before we describe the implementation of `mapUntil`, we examine how `mapUntil` is used to implement `mapLTS`.

The `mapLTS` algorithm, shown in Figure 4, starts by checking the length of the input rope. When the rope length is greater than one (the interesting case), the algorithm calls `mapUntil` to start processing elements. If this call returns a partial result (`More (u, p)`), then `mapLTS` splits the unprocessed subrope `u` and schedules the parallel evaluation of the balancing (if necessary) of the processed subrope `p` and the recursive mapping of the halves of the unprocessed subrope `u`. At the join of the parallel computation, the three now processed subropes are concatenated and returned. Note that because this algorithm is recursive, splitting may continue until a single rope element is reached. If the call to `mapUntil` returns a complete result (`Done p`), then `p` is balanced (if necessary) and returned. Balancing `p` (in either the `More` or `Done` cases) may be profitable here because the ropes returned by `mapUntil` may be unbalanced.

It remains to implement the `mapUntil` operation. The crucial property of the `mapUntil` operation is that it during the traversal of the input rope, it must maintain sufficient information to, at any moment, pause the traversal and reconstruct both the processed portion of the rope and the unprocessed remainder of the rope. Huet's zipper technique [Hue97] provides the insight necessary to derive a persistent 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 allow efficient navigation through and modification of a data structure. With a customized zipper representation, some basic navigation operations, and our novel context-splitting technique, we arrive at an elegant implementation of `mapUntil`.

To represent the rope-map traversal, we use a context representation similar to Huet's single-hole contexts [Hue97], but with different types of elements on either side of the hole, as in McBride's contexts [McB08]. Thus, our context representation is defined as

```
datatype ('a, 'b) ctx
  = Top
  | CatL of 'a rope * ('a, 'b) ctx
  | CatR of 'b rope * ('a, 'b) ctx
```

where `Top` represents an empty context, `CatL(r, c)` represents the context surrounding the left branch of a `Cat` node where `r` is the right branch and `c` is the context surrounding the `Cat` node, and `CatR(l, c)` represents the context surrounding the right branch of a `Cat` node where `l` is the left branch and `c` is the context surrounding the `Cat` node. Note that, for a rope-map traversal, all subropes to the left of the context's hole are processed (`'b rope`) and all subropes to the right of the context's hole are unprocessed (`'a rope`).

The implementation of `mapUntil` will require a number of operations to manipulate a context. The `leftmost (rp, c) ⇒ (s', c')` operation plugs the (unprocessed) rope `rp` into the context `c`, then navigates to the leftmost leaf of `rp`, returning the sequence `s'` at that leaf and the context `c'` surrounding that leaf:

```
val leftmost : 'a rope * ('a, 'b) ctx
               -> 'a seq * ('a, 'b) ctx

fun leftmost (rp, c) = (case rp
  of Leaf s => (s, c)
   | Cat (l, r) => leftmost (l, CatL (r, c)))
```

The `start` operation simply specializes `leftmost` to the case of the whole unprocessed rope in the empty context (see Figure 5(a)):

```
val start : 'a rope -> 'a seq * ('a, 'b) ctx

fun start rp = leftmost (rp, Top)
```

It is used to initialize the `mapUntil` traversal. The `next (rp, c)` operation plugs the (processed) rope `rp` into the context `c`, then attempts to navigate to the next unprocessed leaf.

```
val next :
  'b rope * ('a, 'b) ctx
  -> ('a seq * ('a, 'b) ctx, 'b rope) progress

fun next (rp, c) = (case c
  of Top => Done rp
   | CatL (r, c') =>
       More (leftmost (r, CatR (rp, c')))
   | CatR (l, c') => next (cat2 (l, rp), c'))
```

This navigation can either succeed, in which case `next` returns `More (s', c')` (see Figure 5(c)), where `s'` is the sequence at the next leaf and `c'` 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.

The final operation on contexts is an operation to split a context into a pair of ropes — the unprocessed subrope that occurs to the right of the hole and the processed subrope that occurs to the left of the hole. It is convenient for the `splitCtx` operation to additionally take an unprocessed rope and a processed rope meant to fill the hole, which are incorporated into the result ropes (see Figure 5(d)):

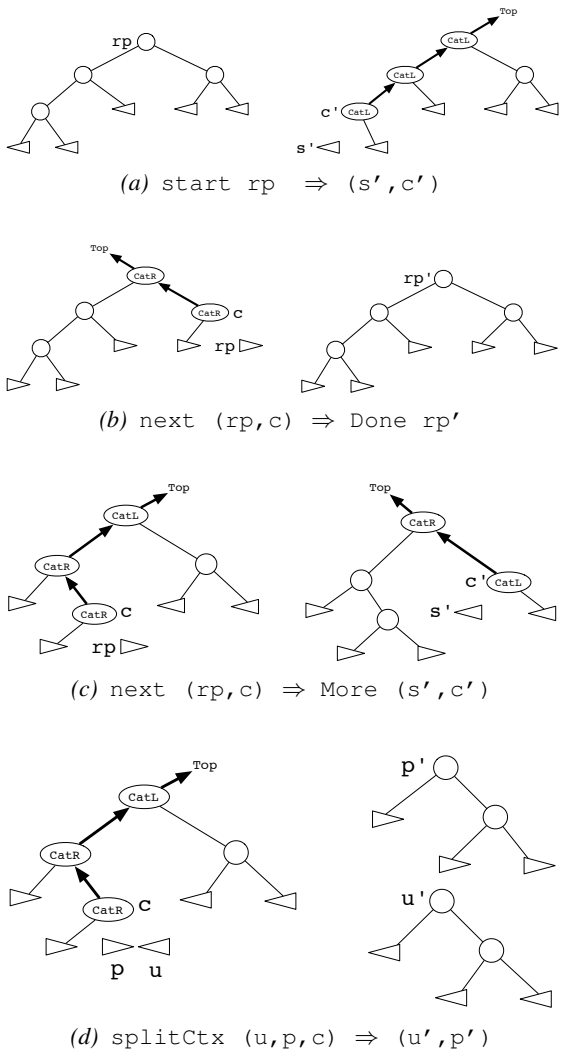


Figure 5. Operations on contexts. A right-facing leaf node denotes a processed node and facing the left an unprocessed node.

```
val splitCtx : 'a rope * 'b rope * ('a, 'b) ctx
               -> 'a rope * 'b rope

fun splitCtx (u, p, c) = (case c
  of Top => (u, p)
   | CatL (u', c') =>
       splitCtx (cat2 (u, u'), p, c')
   | CatR (p', c') =>
       splitCtx (u, cat2 (p, p'), c'))
```

With these context operations, we give the implementation of `mapUntil` in Figure 6. The traversal of `mapUntil` is performed by the function `lp`. The argument `s` represents the sequence of the leftmost unprocessed leaf of the rope and the argument `c` represents the context surrounding that leaf.

The processing of the sequence is performed by `mapSeqUntil`, a function with similar behavior to `mapUntil`, but implemented over linear sequences. It is `mapSeqUntil` that actually calls `cond` and applies the function `f`. Note that `mapSeqUntil` must also maintain a context with processed elements to the left and unprocessed elements to the right, but

```
fun mapUntil cond f rp = let
  fun lp (s, c) = (case mapSeqUntil cond f s
    of More (us, ps) =>
         More (splitCtx (Leaf us, Leaf ps, c))
     | Done ps => (case next (Leaf ps, c)
         of Done p' => Done p'
          | More (s', c') => lp (s', c')))
in
  lp (start rp)
end
```

Figure 6. The `mapUntil` operation.

doing so is trivial for a linear sequence. (Recall the standard accumulate-with-reverse implementation of `map` for lists.)

If `mapSeqUntil` returns a partial result (`More (us, ps)`), then the traversal pauses and returns its intermediate results by splitting its context. (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 `mapSeqUntil` returns a complete result (`Done ps`), then the traversal plugs the context with this completed leaf sequence and attempts to navigate to the next unprocessed leaf by calling `next (Leaf ps, c)`. If `next` returns `Done p'`, then the rope traversal is complete and the whole processed rope is returned. Otherwise, `next` returns `More (s', c')` and the traversal loops to process the next leaf sequence (`s'`) with the new context (`c'`).

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) ctx
```

As with `mapLTS`, where the mapping operation was applied by the `mapSeqUntil` operation, the actual filtering of elements is performed by the `filterSeqUntil` operation.

The `reduceLTS` operation takes an associative operator and 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
  = Top
  | CatL of 'a rope * 'a reduce_ctx
  | CatR of 'a * 'a reduce_ctx
```

where `CatR (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 an important building block of a data-parallel programming language. Like `reduceLTS`, the `scanLTS` operation takes an associative operator and 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
```

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 describes classes of important parallel algorithms that use this operation and gives an efficient parallel implementation of prefix sums [Ble90a], 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* [HP06], 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]))
```

Our implementation of Blelloch's algorithm is again similar to that of `mapLTS`, except that we use a context in which there are `rope`s to the right of the hole and `cached_rope`s 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 -> 'b 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. Note that we do need to distinguish `map2SeqUntil` returning with a partial result due to 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.

4.3 Rebalancing

In our implementation, there are two circumstances in which we need to do balancing. The first is in `filterLTS`, because the filtering predicate may drop elements at arbitrary positions inside the rope. The second is in operations like `mapLTS`, because such operations may split at an arbitrary rope leaf.

5. Evaluation

We have already presented data that shows the performance of ETS is sensitive to the SST parameter. In this section, we present the results of additional experiments that demonstrate that LTS performs as well or better than ETS over a range of benchmarks. Furthermore, it demonstrates scalable performance without any application-specific tuning.

5.1 Experimental method

Our test machine has four quad-core AMD Opteron 8380 processors running at 2.5GHz. Each core has a 512Kb L2 cache and shares a 6Mb L3 cache with the other cores of the processor. The system has 32Gb of RAM and runs Debian Linux (kernel version 2.6.31.6-amd64). 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 use six benchmark programs from our benchmark suite and one synthetic benchmark. Each benchmark is written in a pure, functional style and was originally written by other researchers and ported to PML. All benchmarks use the same max leaf size ($M = 256$), which provides the best average performance over the programs in our benchmark suite.

The Barnes-Hut benchmark [BH86] is a classic N-body problem solver. Each iteration has two phases. In the first phase, a quadtree is constructed from a sequence of mass points. The second phase then uses this tree to accelerate the computation of the gravitational force on the bodies in the system. Our benchmark runs 20 iterations over 200,000 particles generated in a random Plummer distribution. Our version is a translation of a Haskell program [GHC].

The Raytracer benchmark renders a 256×256 image in parallel as two-dimensional sequence, which is then written to a file. The original program was written in ID [Nik91] and is a simple ray tracer that does not use any acceleration data structures. The sequential version differs from the parallel code in that it outputs each pixel to the image file as it is computed, instead of building an intermediate data structure.

The Quicksort benchmark sorts a sequence of 1,000,000 integers in parallel. This code is based on the NESL version of the algorithm [Sca].

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 100×100.

The Tree Rootfix benchmark takes as input a tree structure in which each node is annotated with a value and returns, for each node, the sum of the values on the path from the root of the tree down to that node. This code is based on the NESL version of the algorithm [Sca] and we use it to measure the performance of the `scanP` operation.

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
```

5.3 Lazy *vs.* eager tree splitting

Our most important experimental results come from a comparing LTS to ETS side by side. Figure 7 shows speedup curves for all seven of our benchmarks. For each graph, we plot the speedup curve (over sequential PML performance) of ETS with SST values of 1, 128, and 16384 and of LTS.

		PML			
Benchmark	MLton	Seq.	LTS	Par. 16	Speedup
Barnes Hut	7.71s	14.63s	20.62s	2.20s	6.64
Raytracer	2.29s	3.58s	3.54s	0.22s	16.15
Quicksort	1.36s	3.93s	5.61s	0.51s	7.77
SMVM	0.07s	0.15s	0.19s	0.02s	8.94
DMM	0.84s	3.49s	4.12s	0.30s	11.65
Tree Rootfix	3.79s	8.43s	10.44s	1.32s	6.38
Nested Sums	0.21s	1.46s	1.80s	0.14s	10.17

Table 1. The performance of LTS for seven benchmarks.

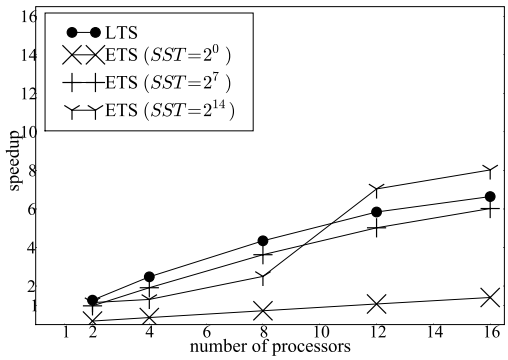
We have argued that one of the main advantages of LTS over ETS is that LTS does not require tuning for each benchmark. These graphs show that LTS is better than most configurations of ETS, and that the downside of picking a poor SST value for ETS can be quite severe (*e.g.*, Figure 7*(b)* with an SST of 128). They also show that not only is the best choice of SST for ETS dependent on the particular benchmark, but in some cases it is also dependent on the number of processors (*e.g.*, Figure 7*(a)* and *(f)*).

With an optimal pick of SST value, ETS can outperform LTS, because of lower overhead. In our experiments, we collected data for every $SST \in \{2^i \mid 0 \leq i \leq 14\}$ and compared the best ETS performance against LTS for each benchmark on 16 processors. We found that even with always choosing the best SST value for the given benchmark and number of processors, ETS was never more than 20% faster than LTS. In practice, it is impossible to make such precise and specialized tuning decisions *a priori*, since workloads and compute resources are unpredictable. Therefore, we believe that LTS provides a much better solution to the Goldilocks problem.

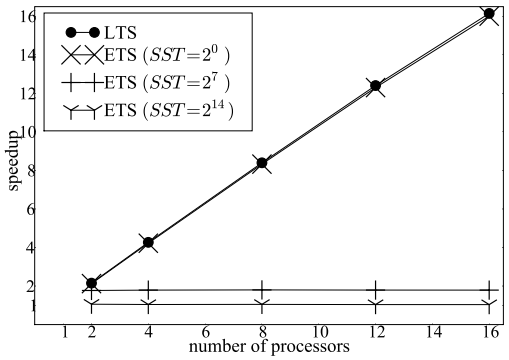
To address the question of why optimal ETS is faster than LTS, we collected profiling data for our benchmarks. This data shows that the per-processor utilization for ETS is never more than 3% greater than that of LTS, which is almost within our 2% error bar. Thus, we believe that the performance gap has to do with increased overhead, rather than poorer scheduling. We also considered the possibility that rebalancing was the source of the performance gap, but our profiling data showed that the total time spent rebalancing is an insignificant fraction of the the total program's run time. Thus, we believe that the main source of this performance gap is the overhead of using a zipper to implement LTS (this point is discussed in further detail below).

In Table 1, we present performance measurements for our seven benchmarks run in several different sequential configurations, as well as on 16 processors.

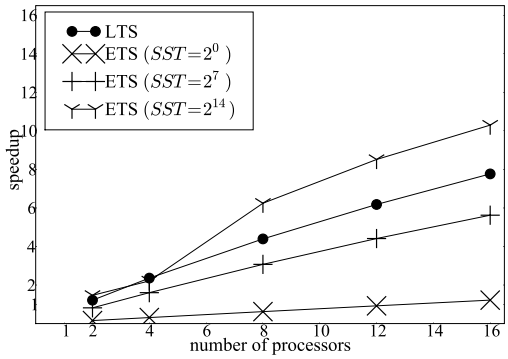
The first column of data presents timing results for MLton. MLton is a sequential whole-program optimizing compiler for Standard ML [MLt, Wee06], which is the "gold standard" for ML performance. The second data column gives the baseline performance of the natural sequential PML versions of the benchmarks (*i.e.*, parallel operations are replaced with their natural sequential equivalents). We are about a factor of two slower than MLton for all of the benchmarks except DMM and Nested Sums. 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. Since our ropes are polymorphic, we use a boxed representation for the array elements, instead of an unboxed representation. Second, our profiling shows higher GC overheads in our system. We expect to address these issues as we improve our sequential performance.



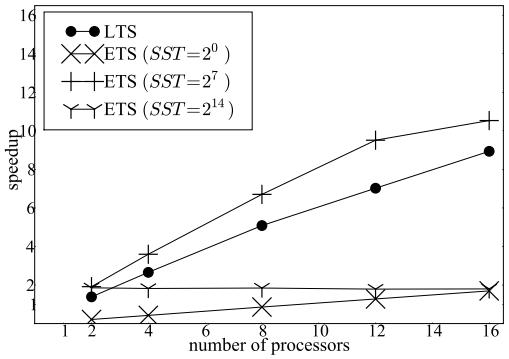
(a) Barnes-Hut



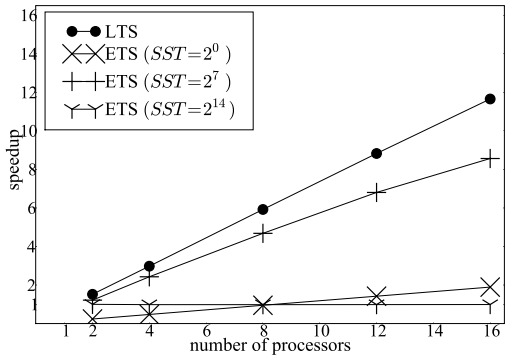
(b) Raytracer



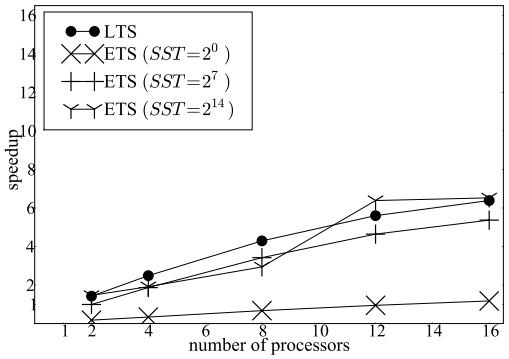
(c) Quicksort



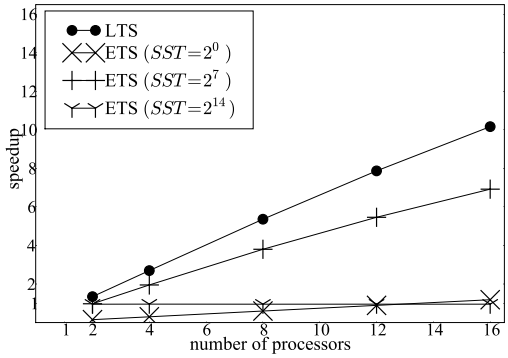
(d) SMVM



(e) DMM



(f) Tree Rootfix



(g) Nested Sums

Figure 7. Comparison of lazy tree splitting (LTS) to eager tree splitting with ETS.

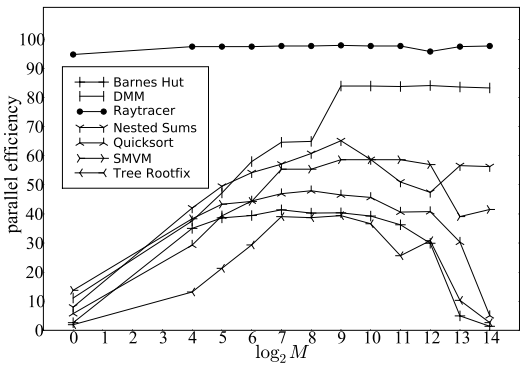


Figure 8. The effect of varying max leaf size M (16 processors)

The third data column reports the execution time of the benchmarks using the LTS runtime mechanisms (*e.g.*, zippers), but without parallelism. By comparing these numbers with the natural sequential measurements, we get a measure of the overhead of the LTS mechanisms. On average, the LTS version is about 24% slower. We have determined through profiling that the main source of this overhead is *not* from calls to `hungryProcs` or rebalancing. Instead, the primary source of the overhead comes from maintaining the traversal state via the zipper context. Such a strategy is less efficient than implicitly maintaining the state via the run-time call stack in a natural structural recursion.[7]

The last two columns report the parallel execution time and speedup on sixteen processors. Overall, the speedups are quite good. The super-linear speedup of the Raytracer is explained by a reduction in GC load per processor. This reduction happened because each processor has its own local heap, so the total size of the available heap increases with the number of processors. Our GC architecture is described in more detail elsewhere [FRR08].

The Barnes-Hut benchmark 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 problem size to 400,000 particles improves the speedup results. The DMM benchmark is 25-27% slower than a perfect speedup, which is also modest considering the large amount of parallelism available in the program. We attribute the slower performance on DMM to an increase in overheads incurred by the LTS Zipper traversal. Observe that the sequential version of DMM that uses a LTS is 20% slower than a similar version that does not.

There is still a question of whether our technique trades one tuning parameter (SST) for another, the max leaf size (M). We address this concern in two ways. First, observe that even if performance is sensitive to M, this problem is specific to ropes, but neither ETS nor LTS. Second, consider Figure 8 which shows, for each of our benchmark programs, the parallel efficiency as a function of M (the parallel efficiency has the same meaning as it does in Figure 3). The results show all benchmarks performing well for $M \in \{512, 1024, 2048\}$. One concern is DMM, which is sensitive to M because it does many random access operations on its two input ropes. One can reduce this sensitivity by using an alternative rope representation that provides more efficient random access.

[7] Our implementation uses heap-allocated continuations to represent the call stack [App92, FFR+07].

6. Related work

Adaptive parallel loop scheduling The original work on lazy binary splitting presents a dynamic scheduling approach for parallel `do-all` loops [TCBV10]. 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 which determines how many iterations a given loop can process before a doing 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.* propose a variant of EBS called auto partitioning [RVK08], which offers good performance for many programs and does not require tuning.[8] Auto partitioning derives some limited adaptivity by employing the heuristic that when a task detects it has been migrated it splits its chunk into at least some fixed number of subchunks. 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.* [TCBV10], auto partitioning has two limitations. First, for i levels of loop nesting, P processors, and a small, constant parameter K, it creates $(K \times P)^i$ 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.

Granularity control Early work by Loidl and Hammond in the context of Haskell compared three strategies for deciding whether to create a thread for parallel work or continue in sequence [LH95]. In simulation, they found that using a simple cut-off generated more speedup than more complicated strategies that dynamically determine whether to create a thread and which thread to run based on a priority associated with the function to run. This cut-off is a value based on a granularity estimation function provided to the parallel primitives. They found, as we did, that speedup was highly dependent upon the cut-off value. Their approach differs from ours in that the cut-off value is statically provided to the runtime; they require a function that can report a granularity metric of the work to perform based on the function being called and the data computed upon. Notably, their work handles any divide-and-conquer algorithm, whereas our solution specifically addresses parallel map operations.

Tick and Zhong presented an approach using compile-time granularity analysis in concurrent logic programs [TZ93]. Their compiler creates a call graph,[9] collapses all strongly-connected components (mutually-recursive functions), and then walks up the collapsed graph creating recurrence equations representing cost estimates. These recurrence questions are solved at compile time and used at run time for cost estimation of functions based on their dynamic inputs. This work does not discuss how these cost metrics are integrated into their scheduler, but does provide an 85–91% accurate estimator of runtime costs for arbitrary functions across their suite of benchmarks. Their static analysis takes advantage of logic programming language features, but demonstrates a potentially more effective approach to determining a satisfactory PPT.

Data parallelism NESL is a nested data-parallel dialect of ML [BCH+94]. The NESL compiler uses a program transformation

[8] Auto partitioning is currently the default chunking strategy of TBB [Int08].

[9] This language is not higher-order, which greatly simplifies the construction of the call graph.

called *flattening*, which transforms nested parallelism into a form of data parallelism that maps well onto SIMD architectures. Note that SIMD operations typically require arrays 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 major disadvantages. First, tree random access is costlier, 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.

The NESL backend written by Chatterjee [Cha93] and Data Parallel Haskell [CLPK08] performs fusion of parallel operations in order to increase granularity. We do not currently implement such transformations. While fusion reduces overall work for data-parallel operations, it reduces the work per element but does not affect the coarsening of the iterations within a data-parallel operation. Such fusion techniques are orthogonal to LTS.

Narlikar and Blelloch present a parallel depth-first (PDF) scheduler that is designed to minimize space usage [NB99]. Later work by Greiner and Blelloch on proposes an implementation of NDP based on the PDF scheduler [BG96]. The PDF schedule is a greedy schedule that is based on the depth-first traversal of the parallel execution graph. The PDF schedule is as close to the sequential schedule as possible in the sense that the scheduler only ever goes ahead of the sequential schedule when the scheduler is limited by data dependencies. In contrast, the work stealing approach used by LTS has each processor doing an independent depth-first traversal of that processor's own portion of the parallel execution graph.

The work on space efficient scheduling does not address the issue of building an automatic chunking strategy, which is the main contribution of LTS. Narlikar and Blelloch coarsen loops manually in order to obtain scalable parallel performance in their performance study. LTS finds good chunk sizes automatically, without programmer assistance.

Ct is an NDP extension to C++ [GSF+07]. So *et al.* describe a fusion technique for Ct that is similar to the fusion technique of DPH [SGW06]. 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," [THLP98] 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 the C language extended with linguistic constructs for expressing fork-join parallelism [FLR98]. Cilk is designed for parallel function calls but not loops, whereas our approach addresses both.

7. Discussion

The main idea of lazy splitting is to maintain some extra information so it is always possible to spawn off half of the remaining work. This paper presents an instantiation of this idea for operations that produce and consume ropes. Although the main idea has potential to be adapted to a larger class of divide-and-conquer programs, we believe at least three substantial challenges must be met before this goal can be achieved. The first challenge is to support other tree representations, such as, for example, red-black trees. Specifically, one must derive efficient traversal patterns that preserve the invariants of such structures. Second, LTS programs involve zippers, which are an implementation detail. Are there general techniques to derive LTS specifications automatically from more natural specifications? For example, is there a mechanical process for deriving LTS programs (*e.g.*, `mapLTS`) from structural-recursive programs (*e.g.*, `mapStructural`)? One possible approach is to use a static analysis to identify divide-and-conquer recursive functions, then apply a program transformation to generate analogous lazy-splitting versions. Third, there is a need for general techniques to aggregate work for small problem sizes (rope leaves effectively provide this mechanism in the system described here). Failure to provide such techniques will result in excessive overhead and limited scalability.

The splitting strategy used by LBS and our LTS can cause unnecessary splitting. To understand why, observe that splitting is prone to start at the innermost loops and then work its way to the outer loops, as discussed at the end of Section 3. Having the thief worker split the *outermost* loops is more efficient because the outer iterations usually contain the most work.

Our current implementation uses innermost splitting for two reasons. First, to support outermost splitting would involve special support from the language implementation, as splitting the outermost loop would involve modifying a part of the whole continuation, not just a part of the continuation of the current loop. Second, in our empirical study, for each benchmark, we observed that total number of splits stayed under the low hundreds. Since, steals are extremely fast in our test machine, having a few extra steals made little difference. We expect that an implementation based on outermost stealing would be superior for larger machines.

8. 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 the necessary traversals. Our techniques can be readily adapted to tree data structures other than ropes and is not limited to functional languages. A work-stealing thread scheduler is the only special requirement of our technique.

LTS compares favorably to ETS, requiring no application-specific or machine-specific tuning. For any given benchmark, LTS outperforms most or all configurations of ETS, and is, at worst, only 20% slower than the optimally tuned ETS configuration. Since, in general, optimal tuning of ETS for arbitrary programs and computational resources is not possible, we believe that LTS is a superior implementation technique. The ability of LTS to enable good parallel performance without requiring application-specific tuning is very promising.

Acknowledgments

We would like to thank the anonymous referees for their helpful suggestions and the National Science Foundation for their support under 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 these organizations or the U.S. Government.

References

[App89] Appel, A. W. Simple generational garbage collection and fast allocation. *SP&E*, **19**(2), 1989, pp. 171–183.

[App92] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[BAP95] Boehm, H.-J., R. Atkinson, and M. Plass. Ropes: an alternative to strings. *SP&E*, **25**(12), December 1995, pp. 1315–1330.

[BCH+94] Blelloch, G. E., S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *JPDC*, **21**(1), 1994, pp. 4–14.

[BG96] Blelloch, G. E. and J. Greiner. A provable time and space efficient implementation of NESL. In *ICFP '96*. ACM, May 1996, pp. 213–225.

[BH86] Barnes, J. and P. Hut. A hierarchical $o(n \log n)$ force calculation algorithm. *Nature*, **324**, December 1986, pp. 446–449.

[BL99] Blumofe, R. D. and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, **46**(5), 1999, pp. 720–748.

[Ble90a] Blelloch, G. E. Prefix sums and their applications. *Technical Report CMU-CS-90-190*, School of Computer Science, Carnegie Mellon University, November 1990.

[Ble90b] Blelloch, G. E. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

[Ble96] Blelloch, G. E. Programming parallel algorithms. *CACM*, **39**(3), March 1996, pp. 85–97.

[BS81] Burton, F. W. and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81*. ACM, October 1981, pp. 187–194.

[Cha93] Chatterjee, S. Compiling nested data-parallel programs for shared-memory multiprocessors. *ACM TOPLAS*, **15**(3), July 1993, pp. 400–462.

[CLP+07] Chakravarty, M. M. T., R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *DAMP '07*. ACM, January 2007, pp. 10–18.

[CLPK08] Chakravarty, M. M. T., R. Leshchinskiy, S. Peyton Jones, and G. Keller. Partial Vectorisation of Haskell Programs. In *DAMP '08*. ACM, January 2008.

[FFR+07] Fluet, M., N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status Report: The Manticore Project. In *ML '07*. ACM, October 2007, pp. 15–24.

[FLR98] Frigo, M., C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98*, June 1998, pp. 212–223.

[FRR+07] Fluet, M., M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *DAMP '07*. ACM, January 2007, pp. 37–44.

[FRR08] Fluet, M., M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. In *ICFP '08*, Victoria, BC, Candada, September 2008. ACM, pp. 241–252.

[FRRS08] Fluet, M., M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. In *ICFP '08*, Victoria, BC, Candada, September 2008. ACM, pp. 119–130.

[GHC] GHC. Barnes Hut benchmark written in Haskell. Available from `http://darcs.haskell.org/packages/ndp/examples/barnesHut/`.

[GSF+07] Ghuloum, A., E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A flexible parallel programming model for tera-scale architectures. *Technical report*, Intel, October 2007. Available at `http://techresearch.intel.com/UserFiles/en-us/File/terascale/Whitepaper-Ct.pdf`.

[Hal84] Halstead Jr., R. H. Implementation of multilisp: Lisp on a multiprocessor. In *LFP '84*. ACM, August 1984, pp. 9–17.

[HP06] Hinze, R. and R. Paterson. Finger trees: a simple general-purpose data structure. *JFP*, **16**(2), 2006, pp. 197–217.

[Hue97] Huet, G. The zipper. *JFP*, **7**(5), 1997, pp. 549–554.

[Int08] Intel. *Intel Threading Building Blocks Reference Manual*, 2008.

[Kel99] Keller, G. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. Ph.D. dissertation, Technische Universität Berlin, Berlin, Germany, 1999.

[Lei09] Leiserson, C. E. The Cilk++ concurrency platform. In *DAC '09*, San Francisco, California, 2009. ACM, pp. 522–527.

[Les05] Leshchinskiy, R. *Higher-Order Nested Data Parallelism: Semantics and Implementation*. Ph.D. dissertation, Technische Universität Berlin, Berlin, Germany, 2005.

[LH95] Loidl, H. W. and K. Hammond. On the Granularity of Divide-and-Conquer Parallelism. In *GWFP '95*. Springer-Verlag, 1995, pp. 8–10.

[McB08] McBride, C. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *POPL '08*. ACM, January 2008, pp. 287–295.

[MLt] MLton. The MLton Standard ML compiler. Available at `http://mlton.org`.

[MTHM97] Milner, R., M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[NB99] Narlikar, G. J. and G. E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM TOPLAS*, **21**(1), 1999, pp. 138–173.

[Nik91] Nikhil, R. S. *ID Language Reference Manual*. Laboratory for Computer Science, MIT, Cambridge, MA, July 1991.

[Rai07] Rainey, M. The Manticore runtime model. Master's dissertation, University of Chicago, January 2007. Available from `http://manticore.cs.uchicago.edu`.

[Rai09] Rainey, M. Prototyping nested schedulers. In M. Felleisen, R. Findler, and M. Flatt (eds.), *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[RVK08] Robison, A., M. Voss, and A. Kukanov. Optimization via Reflection on Work Stealing in TBB. In *IPDPS '08*. IEEE Computer Society Press, 2008.

[Sca] Scandal Project. A library of parallel algorithms written NESL. Available from `http://www.cs.cmu.edu/~scandal/nesl/algorithms.html`.

[SGW06] So, B., A. Ghuloum, and Y. Wu. Optimizing data parallel operations on many-core platforms. In *STMCS '06*, 2006.

[TCBV10] Tzannes, A., G. C. Caragea, R. Barua, and U. Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP '10*, Bangalore, India, January 2010. ACM, pp. 179–190.

[THLP98] Trinder, P. W., K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *JFP*, **8**(1), January 1998, pp. 23–60.

[TZ93] Tick, E. and X. Zhong. A compile-time granularity analysis algorithm and its performance evaluation. In *FGCS '92*, Tokyo, Japan, 1993. Springer-Verlag, pp. 271–295.

[Wee06] Weeks, S. Whole program compilation in MLton. Invited talk at ML '06 Workshop, September 2006. Invited talk; slides available at `http://mlton.org/pages/References/attachments/060916-mlton.pdf`.