Supporting Partial Ordering with the Parallel Iterator

Nasser Giacaman and Oliver Sinnen
Department of Electrical and Computer Engineering, The University of Auckland, New Zealand
ngia003@aucklanduni.ac.nz, o.sinnen@auckland.ac.nz

Abstract

With the advent of multi-core processors, desktop application developers must finally face parallel computing and its challenges. A large portion of the computational load in a program rests within iterative computations. In object-oriented languages these are commonly handled using iterators which are inadequate for parallel programming. Consequently, the powerful Parallel Iterator concept was developed. This paper presents various developments of the Parallel Iterator, such as parallel traversal of complex collections with partial ordering (such as a tree). Other features include reductions, parallel remove semantics and exception handling. Along with the ease of use, the results reveal great speedup in comparison to traditional Java parallelism approaches.

Keywords: parallel programming, object-oriented, Parallel Iterator, loop scheduling, reductions

1. Introduction

Any software programmer developing a parallel application will quickly face the notorious challenges of parallel computing. Even though parallel computing is decades old, desktop parallelization is fairly new and was initiated by the introduction of desktop dual-core processors in 2005. If we desire performance improvements in our daily desktop experience, it is inevitable to parallelize the applications [1].

Iterative computations usually carry the lion’s share of computational load, and in object-oriented languages this is often implemented with iterators. Unfortunately, sequential iterators and other traditional parallelism approaches have their pitfalls (discussed in section 2). For this reason, the Parallel Iterator concept [2, 3] has been proposed (reviewed in section 3).

This paper presents recent advancements (section 4) of the Parallel Iterator: the major ones being support for parallel traversal of more complex collections with partial ordering (here, we concentrate on tree collections), support for exception handling and parallel remove semantics. We also present revised semantics of object-oriented reductions. The implementation (section 5) and performance (section 6) of the Tree Parallel Iterator is presented. We also benchmark the Parallel Iterator against traditional Java parallelism approaches.

2. Background

In object-oriented programming, collections are used to store objects (i.e. elements). They come in many forms, including linked-lists, trees, sets, maps, and so on. Regardless of the collection type, iterators provide a consistent means to access the elements. The Java-style sequential iterator provides two primary methods. The first method, hasNext(), inquires to see if at least one element remains to be traversed. If this returns true, then next() can be called to retrieve that element:

```java
List<File> list = getImages();
Iterator<File> it = list.iterator();
while (it.hasNext()) {
    File image = it.next();
    resize(image);
}
```

Traditional parallelism approaches: The inherent parallelism of the above example is to resize the images in parallel. What are the possible tools to be used by a programmer? Thread libraries are available for most object-oriented languages, the easiest solutions (evaluated in section 6.1) a developer could implement include:

- **Locking on each iteration:**
  A thread attempts to get an element by gaining exclusive access to the iterator using a lock: this allows the thread to atomically call hasNext() and next().

- **Concurrent collection:**
  Elements are stored in a thread-safe queue that is shared by all the threads.

- **Synchronized method:**
  A monitor is created by producing synchronized code: threads acquire the object’s intrinsic lock [4].

- **Static decomposition of the collection:**
  Iterators are assigned to threads before any work begins by decomposing into smaller ranges.

Unfortunately, the above solutions are unfavorable. First, a large amount of programming effort is required of users (e.g. to ensure thread-safety). Second, each of the above approaches resembles a single scheduling policy. This reduces flexibility since the respective load distribution is inadequate to achieve good performance (section 6.1). If a different scheduling policy is required, this will involve considerably more work on the implementation side and the user’s iteration code might also need to be modified.
3. Parallel Iterator

In this section we review the Parallel Iterator: a powerful concept for object-oriented programming that serves to iterate collections in a thread-safe manner. Although the Parallel Iterator does not manage thread creation, it does possess awareness of the threads accessing it. The Parallel Iterator uses the same standard interface of the sequential iterator: hasNext() returns a boolean denoting whether there are any elements remaining, while next() returns the next element. Below is the parallel version of the image resizing application using the Parallel Iterator:

```java
List<File> list = getImages();
ParIterator<File> it = ParIterator.createParIterator(list);

// each thread does this
while (it.hasNext()) {
    File image = it.next();
    resize(image);
}
```

All logic in regards to scheduling and synchronization are contained within the Parallel Iterator. From the user’s point of view, the Parallel Iterator is an ordinary iterator providing a uniform and thread-safe means to traverse elements in a collection. The Java Parallel Iterator extends the standard Java Iterator interface. In order to preserve semantics of the sequential iterator, the last call to hasNext() contains an implicit barrier synchronization: all threads that have completed their iterations will block at the loop boundary waiting for all other threads to complete. There is also a semantic requirement that a thread receiving true from hasNext() must call next() (since an element has been reserved for that thread).

A factory class is provided to simplify the creation of a Parallel Iterator. The user is required to at least supply the collection containing the elements to traverse. For added flexibility, the user may also specify a scheduling policy and chunk size in order to override the default policies. The Parallel Iterator also supports parallel break semantics to allow early loop termination. In such a case, threads wait at the implicit barrier before receiving false from hasNext() (therefore allowed to safely complete their current iteration).

Scheduling policies and chunk size: A scheduling policy determines how the iteration space is divided amongst threads into smaller chunks. The Parallel Iterator supports static (all iterations are assigned to threads before the execution of the loop), dynamic (each thread requests a chunk of iterations to process) and guided (similar to dynamic, except the size of each chunk decreases as iterations are distributed) scheduling policies [5]. A chunk size may also be specified, where the next chunksize iterations are reserved for the same thread. The purpose of the chunk size is to find the best trade-off between good load balancing and low overhead [6]. As the experimental results will show in section 6, these scheduling policies are very important.

4. Extensions to the Parallel Iterator

We have extended the Parallel Iterator with new features. We present these features by integrating them into a single example to best illustrate how they can work together. The first development we present is our extension of the Parallel Iterator concept to allow traversal of collections that involve partial ordering: the Tree Parallel Iterator. The other features include parallel semantics for remove(), support for exception handling, as well as revised semantics for object-oriented reductions.

The Tree Parallel Iterator: The Parallel Iterator is a powerful way to parallelize code. One of the conditions of using it efficiently and elegantly is that there are no dependences between the iterations (otherwise manual dependence control must be implemented by the user). There are some collections, however, whose processing requires a certain partial order. A very important and widely used class of such collections are tree structures. When traversed, it might for example be important that an element (node) is processed before its child node (i.e. top-down traversal), or the other way round (i.e. bottom-up). Even with such a restriction, a lot of iterations can be processed in parallel.

When processing such a collection in parallel, one would like to separate the business logic from the code responsible for the correct parallel execution, including scheduling and precedence order enforcement. In this paper we propose a generic and flexible Tree Parallel Iterator that does exactly that. It implements the same simple interface as the regular Parallel Iterator, only the tree collection to be processed must adhere to a tiny and simple tree interface (since Java does not have a tree collection as part of the Collections framework, we define a Node interface to represent nodes in a tree).

Partial ordering in the Tree Parallel Iterator: Here, we discuss the schedule in terms of the partial order semantics from the user’s point of view; the implementation is discussed in more depth in section 5. The partial ordering requires that a node (i.e. an iteration in the Tree Parallel Iterator) only be executed when its parent has completed. Therefore, initially only the root node may be processed. When the root node has completed, then the next level nodes may be scheduled: any other node remains unscheduled until its respective parent node has completed. This partial ordering retains the structure of the tree; without such an order constraint, the tree can be processed with the standard Parallel Iterator. This partial ordering (i.e. top-down) is only one possibility: others (e.g. bottom-up) are also possible.

SVG shape recognition example: In order to traverse a tree, we need a tree collection. One possible tree collection in Java is the Document Object Model (DOM) Document [7]: it represents an entire HTML or XML document. Our concrete example application involves traversing a Scalable Vector Graphics (SVG) file (which is essentially an XML document that defines vector-based graphics). Our example application recognizes shapes from a generic

470
path pattern. The input SVG file (containing generic definitions of shapes) is translated into another SVG file (containing specific definitions of the determined shapes). By specifying the Document (i.e. the tree), a Tree Parallel Iterator is constructed and may be used like the standard Parallel Iterator:

```java
1: Document doc = getDocument("shapes.svg");
2: TreeParIterator pi = ParIterator.create(doc);
3: Reducible<Integer> numCircles =
   new Reducible<Integer>(0);
4: Reducible<Shape> biggestShape =
   new Reducible<Shape>(null);
```

The Reducible objects introduce another feature: object-oriented reductions (discussed later in this section). The application is presented in full below, and then the various parts explained:

```java
// each thread does this
5: while (pi.hasNext()) {
6:   try {
7:     Node n = pi.next();
8:
9:     Element e = (Element)n;
10:    if (e.getNodeName().equals("group")) {
11:      String vis = e.getAttribute("visibility");
12:     } else if (vis.equals("invisible")
13:      pi.remove();
14:    } else if (e.getNodeName().equals("path")) {
15:      Shape shape = ShapeGuessor.guess(e);
16:     if (biggestShape.get() == null) {
17:       biggestShape.set(shape);
18:     } else if (shape.getArea() > biggestShape.get().getArea()){
19:       biggestShape.set().getArea());
20:    }
21:  } catch (Exception e) {
22:    // each thread does this
23:  }
24: }
```

1. **Reducible objects**: The thread-local methods are defined inside the Reduceable objects. These methods allow a new element to be copied into local storage using `ThreadLocal` (e.g. lines 8 and 9). The `reduce()` method is defined on the reduceable objects.

2. **Object-oriented reduction**: The `reduce()` method reduces the object in a specific way. For example, it compares the area of a shape to find the biggest shape, as shown in lines 16-18. The `reduce()` method can be defined by the programmer to perform different reductions.

3. **Parallel remove() semantics**: The `remove()` method removes the node from the collection. In a parallel context, this method is called by each thread and the final result is obtained when all threads have completed. The `replace()` method is called on the final result to replace the node in the collection.

The highlight of the Tree Parallel Iterator is that the user code to traverse the nodes essentially looks identical to the standard code to traverse any other collection. This is the whole idea behind iterators: to hide the underlying collection’s implementation. The Tree Parallel Iterator, just like the Parallel Iterator, remains faithful to this idea. Consequently, even when a partial ordering is necessary, the user code to iterate a tree collection in parallel remains elegant and independent of the underlying scheduling scheme.

**Parallel remove() semantics**: The first point of interest in the above example is that of the `remove()` method on line 13. Just like the standard sequential iterator, we have extended the Parallel Iterator to support `remove();` it removes from the underlying collection the last element returned by the Parallel Iterator to the current thread. In the context of the Tree Parallel Iterator, removing a node also removes the children nodes. This is safe while using the Tree Parallel Iterator since the children nodes have not been scheduled yet (e.g. line 13 of the above example).

The other point of interest is that of line 31. Similar to the semantics of `remove()`, the Tree Parallel Iterator supports a `replace()` method that allows a new element to be put in place of the last element returned by the Tree Parallel Iterator (this functionality is essentially a thread-safe wrapper around the replaceChild() method provided by the DOM Node interface). In this example, path elements are replaced with the respective shape element. Again, this is permitted in a parallel traversal since the parent has already completed and the children not started yet.

**Object-oriented reductions**: A reduction is a standard parallelization problem where each thread maintains its own copy of a variable to avoid excessive locking. Since each thread calculates only a partial result, these sub-results must be reduced into a single result. The previous solution [3] proposed reductions only for languages that support function pointers and operator overloading (such as C++, but not Java). Here we propose an alternative solution to support reductions for other languages such as Java.

The `Reducible` object is used just like a typical thread-local variable (such as Java’s `ThreadLocal` class [7]), where threads access the local storage using `get()` and `set()` (e.g. lines 16 and 17). In addition to these thread-local methods, there is a `reduce()` method that will perform a specified reduction across the thread-local values when threads have finished their work and return the final result:

```java
int totalNumCircles =
  numCircles.reduce(Reduction.IntegerSUM);
Shape finalBiggestShape =
  biggestShape.reduce(new Reduction<Shape>(){
    public Shape reduce(Shape first, Shape second){
      if (first.getArea() > second.getArea())
        return first;
      else
        return second;
    }});
```

The second reduction of `finalBiggestShape` is an example of a customized reduction. Programmers may define their own reduction to be performed on any object type. This truly object-oriented solution allows for more complex reductions, for example aggregate object merging.

**Parallel semantics for exceptions**: The Parallel Iterator has been extended to provide a helper method to conveniently record information when an exception occurs. The programmer is only required to catch exceptions (standard procedure as if using a sequential iterator). When an exception is caught, `register(Exception)` may be called (e.g. line 36) to record information such as the exception encountered, the thread that encountered the exception and the iteration in which the exception occurred (determined using
The thief at random. The thief has assigned node 0 (when it called hasNext() first: in our example, this happens to be thread A. In the meantime, thread B is trying to steal from another random thread (in this case it only has one other thread to steal from).

Since thread A has been assigned node 0 when it called hasNext(), it follows up with a call to next() to retrieve this node. The node is considered complete when thread A calls hasNext() again, implying that it has completed node 0 and wishes to be assigned another node. In this situation, the Tree Parallel Iterator will enqueue the children of node 0 to thread A’s private deque (figure 1(b)). Consequently, nodes 1, 2, and 3 are ready to be executed.

Now that 3 nodes have been enqueued to thread A’s deque, thread B has found its victim: it steals the oldest node from thread A, which happens to be node 1. In the meantime, thread A takes it’s latest node (node 3) and executes it. When thread B completes its computation (figure 1(c)), it enqueues the children (nodes 4 and 5) of the last node it completed. Consequently, thread B does not need to perform another steal since it has unprocessed nodes on its deque: thread B now executes its most recent local node. The parallel traversal of the tree is considered complete when all threads are attempting to steal.

The advantage of the Tree Parallel Iterator is that all implementation details of the scheduling and dependence control are hidden from the user. This allows for further scheduling implementations of the Tree Parallel Iterator. For example, another potential scheduling scheme might involve distributing nodes only when the children nodes are complete (i.e., processing leaf nodes first, in a bottom-up traversal). Since the implementation details would be hidden from the user, the user code (to iterate nodes) does not require modification when a different scheduling scheme is used: only a parameter is needed to select the right scheduling scheme based on the inherent dependence structure.

6. Performance

In this section we evaluate the performance of the Parallel Iterator compared to the traditional parallelism approaches, as well as the performance of the Tree Parallel Iterator. In each of the benchmarks, the baseline in determining the speedup is the sequential code that uses the standard sequential iterator. The benchmarks ran on a shared memory system which may be considered a typical future desktop platform running Linux. It has four quad-core Intel Xeon processors (total of 16 cores) running at 2.4GHz with 64GB of RAM.

6.1. Comparing to traditional approaches

We first present previously unpublished results to emphasize the competitiveness of the Parallel Iterator, even in
terms of speed. The performance of the Parallel Iterator is compared to some of the traditional Java parallelism approaches (as discussed in section 2) to traversal a collection of elements. We further break down the locking approach into two categories depending on the underlying lock implementation [7]: fair locking (when competed for, access to the lock is granted to the longest waiting thread) versus unfair locking (no order is guaranteed for access to the lock).

Each of the graphs in figure 2 represents a different workload (each workload contains 1 million iterations). Figure 2(a) shows the speedup for a balanced, but fine-grained, workload (each iteration takes on average 3.5 µs). Out of the traditional approaches, the best was static decomposition since this approach minimizes runtime overhead and eliminates lock contention. The Parallel Iterator (here using guided scheduling with chunk size of 5) executes with similar performance. As expected, the locking approaches (especially fair locking) perform very poorly. Rather surprising is the concurrent collection’s poor performance (using a concurrent queue from java.util.concurrent).

Figure 2(b) shows the speedup for an unbalanced workload. The Parallel Iterator (in this case using a dynamic scheduling policy with a chunk size of 10,000) is again the leading solution. Notice the inconsistency of the other traditional approaches: the static decomposition performed best for our first workload (figure 2(a)), while it performed worst in the second workload (figure 2(b)). The synchronized code and unfair locking were the only other approaches with respectable speedup.

These benchmarks show that the Parallel Iterator (with policy and chunksize tuning) is the only consistent solution across the different workloads. Most importantly is that the user’s iteration code remained unchanged across all workloads, even when a different scheduling scheme was used for the Parallel Iterator. This re-usability, combined with the performance across a range of workloads, is a very valuable contribution to object-oriented parallel programming.

6.2. Tree Parallel Iterator

In this section we evaluate the implementation of the Tree Parallel Iterator. Figure 3(a) shows the speedup for the first benchmark (the SVG shape application presented in section 4). The 4 workloads shown denote the shape types in the SVG file. The speedup improves for the more computationally intensive workloads (triangles and rectangles are a lot easier to recognize than ellipses). The important observation is that good speedups can be achieved with the Parallel Iterator for very high level programs, employing object-oriented code, XML and SVG.

The next set of benchmarks involve computing a synthetic load (here the Newton-Raphson method) at each node of the tree. The benchmark of figure 3(b) contains fine grained computations, where each node involves approximately 0.4ms of computation. We see that the benefit of parallelization is greater for larger trees. For example, a tree with only 2000 nodes scales up to 11 threads to a speedup of almost 5. However, a tree with 200,000 nodes scaled close to ideal speedup. This is quite encouraging considering the low amounts of computation at each node and the complex structure of the collection.

Figure 3(c) repeats the same experiment, only this time performing more computation at each node: in this case, each node computation takes 6ms. In such a workload, the speedup for each tree size is significantly better. The largest tree scaled to over 98% the ideal speedup, while the smallest tree produced speedups of over 75% the ideal speedup. This is encouraging, since for example, the runtime of the 2000 node tree in figure 3(c) is reduced from 12 seconds to 1 second; in terms of traditional parallel computing this is a very small workload.

In conclusion, the Tree Parallel Iterator is not only easy to use as discussed in section 4, it also yields good speedup. By traversing the nodes of a tree, threads are assigned nodes that are ready to execute. The user code is simple and resembles standard iterator logic: users need not concern with children nodes, and so on. The Tree Parallel Iterator hides implementation details from the user, in particular the scheduling and synchronization of nodes amongst multiple threads.

7. Related work

Over 100 proposals for concurrency in object-oriented languages were surveyed in [10]; the most influential ones relevant to the Parallel Iterator concept as a whole have previously been discussed [2, 3]. In this paper, we focus our
attention on the related work that are particularly relevant to the new features presented in this paper.

Modern parallel languages (such as Chapel [11], Fortress [12] and X10 [13]) target large-scale scientific applications, therefore focus on a distributed memory model. Since we address desktop applications in light of mainstream multi-core processors, we focus on a shared memory programming model. The Microsoft Parallel Extensions [14] to the .NET Framework supports a parallel semantics of the foreach statement with the Parallel.ForEach static method. Unlike the Parallel Iterator, the programmer does not have any control over scheduling policies. That approach is similar to that of Intel’s Threading Building Blocks [15].

Some approaches support aggregate operations, such as PLINQ [16] and ParallelArray [17], thereby removing the loop altogether. This is essentially a change in the programming style: it provides a higher level and black-box style of programming. The Parallel Iterator is not a new programming style: it is an extension to the sequential iterator using the same object-oriented principles.

Processing an XML document in parallel has been explored using both static [18] and dynamic [19] partitioning. Our implementation is more generalized for trees (not only XML documents) and uses the work-stealing [8] schedule based on the randomized work-stealing variant [9].

8. Conclusions

This paper proposed new advancements of the Parallel Iterator concept. It includes support for the parallel traversal of more complex collections that require partial ordering, in the form of the Tree Parallel Iterator. Other new features include parallel semantics for remove, support for exception handling, as well as improved support for object-oriented reductions. In addition to the ease of use, we demonstrate that the Parallel Iterator excels in performance compared to traditional Java parallelism approaches. Similarly, the Tree Parallel Iterator provides very good performance for complex collections with ordering constraints.

References

[2] N. Giacaman and O. Sinnen, “Parallel iterator for parallelising object oriented applications,” in The 7th WSEAS Inter-


Figure 3. Speedup for the Tree Parallel Iterator using 3 different workloads.