University of Auckland Research Repository, ResearchSpace

Version

This is the Accepted Manuscript version. This version is defined in the NISO recommended practice RP-8-2008 http://www.niso.org/publications/rp/

Suggested Reference


Copyright

Items in ResearchSpace are protected by copyright, with all rights reserved, unless otherwise indicated. Previously published items are made available in accordance with the copyright policy of the publisher.

The final publication is available at Springer via http://dx.doi.org/10.1007/s10766-010-0150-5

For more information, see General copyright, Publisher copyright, SHERPA/RoMEO.
Parallel Iterator for Parallelizing Object-Oriented Applications

Nasser Giacaman and Oliver Sinnen

Received: 18 August 2009 / Accepted: 8 March 2010

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. This paper presents a powerful Parallel Iterator concept to be used in object-oriented programs for the parallel traversal of a collection of elements. The Parallel Iterator may be used with any collection type (even those inherently sequential) and it supports several scheduling schemes which may even be decided dynamically at run-time. Some additional features are provided to allow early termination of parallel loops, exception handling and a solution for performing reductions. With a slight contract modification, the Parallel Iterator interface imitates that of the Java-style sequential iterator. All these features combine together to promote minimal, if any, code restructuring. Along with the ease of use, the results reveal negligible overhead and the expected inherent speedup.

Keywords: object-oriented, desktop applications, Parallel Iterator, loop scheduling

1 Introduction

Any software programmer developing a parallel application will quickly face the notorious challenges of parallel computing. First there are theoretical challenges such as task decomposition, dependence analysis and task scheduling. Then there are practical challenges such as portability, synchronization and debugging. Unlike traditional embarrassingly parallel applications, desktop applications contribute a new spectrum of challenges since they are generally irregular with short run-times. They also run on non-dedicated systems, let alone knowing what the system specifications are in the first place (e.g. number of processors, amount of memory and so on). Even though parallel computing is decades old, desktop parallelization is fairly new and

N. Giacaman · O. Sinnen
Department of Electrical and Computer Engineering, University of Auckland, New Zealand
E-mail: ngia003@aucklanduni.ac.nz, o.sinnen@auckland.ac.nz
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 and address these challenges [1,2].

The objective of parallelizing an application is to improve its performance. Due to this objective, parallelizing code has traditionally been paired with general code optimizations for performance, especially in the scientific and engineering area. It is therefore no surprise that applications for these domains have been and are still written in low-level, but speed efficient languages like Fortran or C [3]. When it comes to desktop applications and object-oriented languages, however, one wants to improve the performance through parallelization, but without sacrificing the benefits of high-level languages and the software engineering approach to programming. Hence the challenge is to parallelize object-oriented code without resorting to low-level approaches, defying the purpose of code abstraction, encapsulation and so on. This paper addresses this challenge. Our primary motivation is to relieve programmers from many of the effortful and tedious aspects demanded from parallel computing (section 2), but of course without neglecting the end performance. Hence, we focus on easing the parallelization of object-oriented programs.

Iterative computations usually carry the lion’s share of computational load, and in object-oriented languages this is often implemented with 

\textbf{iterators}. This paper introduces the concept of a \textit{Parallel Iterator} (section 3). It provides a solution for many object-oriented programming situations where parallel traversal of a collection of elements is required. For programs that involve task parallelism as opposed to data parallelism, the task concept [4,5] is more appropriate.

The Parallel Iterator is a new powerful concept for object-oriented programming that serves to iterate any collection type in a thread-safe manner, even those inherently sequential (e.g. linked-lists and trees). The typical iteration code of the program remains the same, meaning that the structure of the program is not changed. The Parallel Iterator has also been implemented with the exact interface of the familiar Java-style sequential iterator. The Parallel Iterator may be used with any threading environment and very elegantly with OpenMP [6] (which is supported by many compilers, including Visual C++, Intel and GCC).

Several scheduling schemes are supported: static, dynamic and guided (covering all major loop-scheduling schemes). In addition, the scheduling policy and chunk size may be decided dynamically during run-time for each loop. The concept of \textit{reductions} is integrated with the Parallel Iterator concept, providing a true object-oriented solution for reductions generalized to allow user-defined reductions on any data type. Global semantics for the \textbf{break} statement are also presented to achieve early loop termination, as well as providing helpful means to handle object-oriented exceptions in a parallel traversal. Performance of the Parallel Iterator (section 5) reveals that the overhead is small and justifiable for object-oriented code using iterators in the first place.

\section{Background}

In object-oriented programming, \textit{collections} (or containers) are used to store objects (i.e. \textit{elements}). They come in many forms, including lists, linked-lists, trees, sets, maps, stacks and so on. Some of these collections are random-access (amortized
constant-time access to random elements), e.g. array-lists, while others are inherently sequential, e.g. linked-lists. Regardless of the collection type, *iterators* provide a consistent means to access the elements. Iterators are not only used for basic collections, but also for complex data types, for example traversing an XML document.

2.1 The Java-style sequential iterator

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 invoked to retrieve that element. To illustrate the concept of the Parallel Iterator, consider the following very simple, but typical, sequential code segment making use of a sequential iterator to resize a list of images:

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

2.2 Traditional parallelism approaches

The inherent parallelism of the above example is to resize the images in parallel. Unfortunately, it is not thread-safe in a parallel environment to simply share the sequential iterator since it leads to a classical race condition. The sequential Java-style iterator requires two separate method calls in order to retrieve an element: the first (`hasNext()`) to check if any element exists, while the second (`next()`) actually retrieves it. Multiple calls to `hasNext()` may return `true` to more than one thread, when in fact only one element remains in the iterator.

Consequently, what are the possible tools to be used by a desktop application developer? Thread libraries are available for most object-oriented languages, the easiest solutions a developer could implement are discussed below (section 5.1 evaluates the performance of these approaches):

- **Locking on each iteration:**
  A sequential iterator is shared amongst all the threads. Any thread that attempts to get an element must gain exclusive access to the iterator using a lock: this allows the thread to atomically call `hasNext()` and `next()`. The programmer is responsible to ensure thread-safe access to the iterator, as well as ensuring all threads finish the loop at the same time.

- **Concurrent collection:**
  Using a thread-safe `queue` (as in Java’s `java.util.concurrent` package) will result in a small-grain dynamic load distribution. By sharing this collection with all the threads, iterations are distributed one at a time as a thread requests work.

- **Synchronized method:**
  Another common approach is to create a monitor by using `synchronized` methods (or `synchronized` statements) as supported by Java. This causes threads to acquire the object’s *intrinsic lock* [7] before processing the `synchronized` code.
– **Static decomposition of the collection:**

This involves manually creating a static distribution of the images. The collection is distributed into multiple sub-collections where each sub-collection is assigned to a thread. This corresponds to a static scheduling policy where iterations are assigned to threads before any work begins.

Unfortunately, the above solutions are unfavorable for several reasons:

– There is a considerable amount of manual programming effort required by the programmer. This includes the responsibility to ensure thread-safety depending on the respective approach taken (for example, a barrier synchronization at the end of each loop to ensure no thread progresses past the loop while other threads are still traversing their share),

– Each approach resembles a single scheduling policy, reducing flexibility:
  – The various load distribution (due to the scheduling policy) might be inadequate to achieve good performance, and
  – If a different scheduling policy is required, not only will this involve considerably more work on the implementation side, but the user’s iteration code might need to be modified also.

– Code concerning the original program model (the business logic, here image resizing) becomes tangled with code dealing with parallelism [8] as it requires restructuring the original code.

### 3 Parallel Iterator

The solution proposed here is the concept of a Parallel Iterator. It may conceptually be considered a thread-safe wrapper around a sequential iterator. Although the Parallel Iterator does not manage thread creation, it does possess awareness of the threads accessing it. For illustration, simple examples will be presented throughout this section; however, the strength of the Parallel Iterator is demonstrated with more complex examples (for example traversing complex data collections in section 3.7).

#### 3.1 Interface and usage

The Parallel Iterator concept has been implemented for two object-oriented languages: Java and C++(Qt). Regardless of the language, the underlying concepts remain the same. This paper focuses on the Java version, while the C++(Qt) version is presented in [9] by using OpenMP to create the threading environment. 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.

However, there is a slight modification to the usage contract of these methods. If thread A invokes `hasNext()` and true is returned, the Parallel Iterator reserves at least one element for thread A. Consequently, only thread A can access that element by invoking `next()`. Even if there was only one element remaining in the iterator, all other threads will receive `false` since that element has been allocated to thread A. The implication of this is that any thread that receives a `true` from `hasNext()` must
eventually follow up with a next() invocation, otherwise that element will not be traversed by any other thread. If hasNext() returns true, any subsequent call to hasNext() has no effect and will continue to return true until next() is eventually called (i.e. multiple calls to hasNext() will not accumulate more elements).

Each thread must not only have pairing invocations of hasNext() and next(), but every thread is required to continue invoking the two until hasNext() returns false. The first reason is because hasNext() serves as a synchronization barrier (explained below). The second reason is because hasNext() in general reserves more than one iteration. Similarly, a thread must not invoke next() without first receiving true from hasNext() since an element needs to be allocated (a run-time exception is thrown in such a case). Below is the parallel version of the image resizing application using the Parallel Iterator:

```java
List list = getFiles();
ParIterator<Image> it = ParIterator.createParIterator(list);

// each thread does this (using OpenMP or threads)
while (it.hasNext()) {
    Image image = it.next();
    resize(image);
} // Parallel Iterator implicit barrier
```

All logic in regards to scheduling policy 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. As shown, the parallel iteration code remains the same as the sequential version; in fact, the interface of the Java Parallel Iterator extends the standard Java Iterator interface.

**Implicit barrier synchronization**

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. When all the iterations have been completed and the last thread calls hasNext(), the barrier synchronization is released and all threads receive false. Consequently, all iterations have been completed before any thread proceeds past the loop (therefore preserving sequential semantics). If the programmer insists on not having the default barrier, they may explicitly turn it off when the Parallel Iterator is created (section 3.2).

**3.2 Construction**

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, as well as the number of threads accessing the Parallel Iterator (defaults to the number of processors). The programmer also has the option to turn off the synchronization barrier. In order to have a single programming paradigm, the Parallel Iterator may also be used to traverse arrays and integer ranges:
3.3 Semantics

The Parallel Iterator may conceptually be considered a thread-safe wrapper around a sequential iterator. As a consequence, it can virtually support all collections. One important assumption is that iterations can be processed in any order, because the Parallel Iterator will not process them in sequential order. If a certain order is necessary for all the loop iterations, then such a loop cannot be parallelized. However, if only a partial order is necessary, a Parallel Iterator may be created that enforces the (partial) order (demonstrated in section 3.3.1 and section 3.7). Hence, again the implementation details are encapsulated in the Parallel Iterator.

Note that access to shared variables, synchronization in the loop body are not addressed by the Parallel Iterator. These can be handled in the usual way (such as locks from thread libraries or the corresponding primitives and clauses in OpenMP). From the Parallel Iterator perspective, any thread may safely modify elements it receives from the Parallel Iterator, but elements should not be inserted into or deleted from the collection being traversed (just like Java’s fail-fast iterators).

3.3.1 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, dynamic and guided scheduling policies [6]. 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 [10]. Figure 1 shows examples of major scheduling policies when 3 threads iterate over a collection. Let $n$ be the number of iterations to be distributed amongst $p$ threads.

- **Static:** all iterations are assigned to threads before the execution of the loop. This may either be block or cyclic:
  - **Block** (figure 1(a)): each thread is assigned one large chunk. If $\lceil n/p \rceil = \lfloor n/p \rfloor$, each thread gets $n/p$ iterations. Otherwise, the first $p - q$ threads will get a...
chunk of ⌈n/p⌉ iterations, while the other q threads get a chunk of ⌊n/p⌋ iterations, where q = p × ⌈n/p⌉ − n. The Parallel Iterator parameters are:
createParIterator(mycollection, PI.Schedule.STATIC);

- Cyclic (figure 1(b)): iterations are grouped into smaller chunks of chunksize iterations and threads are assigned chunks in a round-robin fashion. The Parallel Iterator parameters are:
createParIterator(mycollection, PI.Schedule.STATIC, 1);

- Dynamic [11] (figure 1(c)): each thread requests a chunk of iterations to process. When all iterations of a chunk have completed, another chunk is requested until all chunks have been assigned. The Parallel Iterator parameters are:
createParIterator(mycollection, PI.Schedule.DYNAMIC);

- Guided [12] (figure 1(d)): similar to dynamic, except the size of each chunk decreases. A thread requesting a new chunk is assigned q = ⌈r/p⌉ iterations, where r is the remaining number of unassigned iterations. If q < chunksize, then chunksize iterations are assigned (except on the last chunk where fewer than chunksize iterations might remain). The Parallel Iterator parameters are:
createParIterator(mycollection, PI.Schedule.GUIDED);

In some cases, an additional purpose that chunk sizes serve is to enforce partial ordering for iterations that have simple ordering constraints. As well as enforcing this partial ordering, the Parallel Iterator hides the details from the programmer. For example, assume a partial ordering on a collection such that every i\textsuperscript{th} element is associated with the (i + 1)\textsuperscript{th} element, where 1 ≤ i < collectionSize and i is an odd integer. In such a situation, a Parallel Iterator may be produced (specifying either dynamic or static scheduling) with a chunk size of 2 (or any positive even integer).

3.3.2 Supported collections

The major advantage of the Parallel Iterator is that its usage will be familiar to object-oriented programmers who have used iterators in general and the Java-style sequential iterator in particular. Just as the sequential iterator supports traversal of virtually any collection type, the Parallel Iterator supports this too. For example, the user may traverse a Set in parallel using a dynamic scheduling policy, where each thread is assigned 10 iterations at a time:

ParIterator<MyObj> pi = ParIterator.createParIterator(myset, PI.Schedule.DYNAMIC, 10);

The user may just as easily traverse a LinkedList in parallel using a static scheduling policy with chunk size of 5:

ParIterator<MyObj> pi = ParIterator.createParIterator(mylist, PI.Schedule.STATIC, 5);

In these examples, the actual iteration code does not change even though the scheduling policies and collection types have changed. This provides a real object-oriented approach to parallel traversal of any collection.

3.3.3 Parallel remove semantics

As a thread accesses elements assigned to it through the Parallel Iterator, it may safely modify those elements; this follows the same policy of the sequential iterator.
Elements of the collection being traversed may be modified in-place (this is not the same as modifying the collection by adding or removing elements directly to or from the collection). In addition to the hasNext() and next() methods, some sequential iterators define a remove() method. This is the only way a collection may be modified while traversing it with a Java-style fail-fast iterator (the change is applied through the iterator rather than directly to the collection). Extending this concept to the Parallel Iterator is possible, although it requires some compromise between the expected semantics and performance. The ideal semantic for supporting remove() with the Parallel Iterator would be: although elements are removed from the underlying collection, the original scheduling policy stays in place. Namely, the Parallel Iterator continues to traverse the original underlying collection, and elements removed by arbitrary threads do not affect the original scheduling policy.

**Implementation (and performance) challenges for a parallel remove()**

A “true” remove (where the elements are really removed from the underlying collection during traversal) would require a clone of the original collection being traversed. This allows arbitrary elements from the underlying collection to be safely removed without affecting the original scheduling policy (i.e. changes in the underlying collection are not reflected in the Parallel Iterator). Naturally, this cloning process is essential for collections such as arrays (in order to avoid the shifting down of elements). The efficiency may of course be improved, for example using a copy-on-write approach, or performing a partial clone (e.g. if half the collection has already been traversed, then no need to clone the completed half).

**Proposed semantics for a performant parallel remove()**

As is expected when parallelizing a loop, it is the case that the iterations are independent of each other. Therefore, the timing of an element’s removal from the underlying collection (through the Parallel Iterator) should not affect the correctness of other iterations (since they should be independent). For this reason, the following definition of a parallel remove() integrates nicely with the Parallel Iterator:

Marks for removal from the underlying collection the last element returned by this Parallel Iterator to the current thread. Although the actual removal might take place during traversal, the element is only guaranteed to be removed from the underlying collection only when all threads have finished traversal. This method can be called only once per call to next(). The behavior of a Parallel Iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

By only marking the element for removal, this solves the above implementation difficulties (while still maintaining acceptable semantics). With this definition, an implementation may decide to delay all deletions until the end of traversal when all threads complete. This means that the underlying collection appears unchanged during traversal (even if elements are removed via the Parallel Iterator); from a semantics point of view, this is acceptable since iterations are supposed to be independent of each other anyway (as example is given in section 3.7).
3.4 Object-oriented reductions

A short background is presented on the underlying concepts that the proposed object-oriented reduction concept has been developed upon: reductions and thread-local storage. The interface for the Java implementation is then presented (which is applicable to most object-oriented languages).

Reductions

For programs that share variables, programmers must provide mutual exclusion to ensure correct results. In a threading library, this is typically solved using a mutex. Unfortunately, programs with fine-grained parallelism would suffer heavily in performance. A reduction is a standard parallelization problem [13] where each thread maintains its own copy of the variable to avoid excessive locking, rather than sharing a variable between all threads. However, this means each thread will have a partial result that needs to be reduced (i.e. added) into a final result. OpenMP provides a restricted solution with the reduction clause where only reductions such as $+$, $\times$, $-$, $\&$, $\|$, $\&\&$, $\mid\mid$, minimum and maximum are supported. As well as these common reductions, the primary advantage of the solution proposed here is that it is object-oriented allowing any kind of reduction and using any data type.

Thread-local storage

Thread-local storage is a common method used in parallel computing: although threads refer to the same global variable, they actually refer to different memory locations (hence the variable is local to each thread) [14]. Several languages have been extended to support thread-local storage, for example Java’s `ThreadLocal` class [15]. By creating a thread-local variable, each thread may read and write to it using `get()` and `set()` respectively with no need for locking. However, there is no way a thread may access all the thread-local values within the thread-local variable. This means that if a programmer wishes to reduce all the local values within the thread-local variable, then each thread must copy its local value to another (non thread-local) global variable (since only that thread is able to read the local value).

3.4.1 Reductions with the `Reducible` object

By combining the two concepts of reductions and thread-local storage, an object-oriented solution to reductions is proposed. This is presented in the form of the `Reducible` object: it is especially attractive when used in conjunction with the Parallel Iterator. The `Reducible` object is used just like a typical thread-local variable, where threads access the local storage using `get()` and `set()`. 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.

The example below illustrates these components when trying to find the maximum of a list of unsigned integers. A Parallel Iterator is created in order to manage the parallel traversal of the list. A `Reducible` object of type `Integer` is created to
act as the thread-local maximum variable. In this example, each thread updates its thread-local maximum on every iteration. After all threads have finished, the reduce() operation is invoked on the Reducible object specifying the maximum reduction. To implement this example in OpenMP would require the list to be copied to an array first. Furthermore, this simple example is only used to convey the concept (which may be applied to more complex situations not handled by OpenMP).

```java
// initialize Parallel Iterator and Reducible
List list = ...; // get list of numbers
ParIterator<Integer> pi = ParIterator.createParIterator(list);
Reducible<Integer> localMax = new Reducible<Integer>(0);

// each thread does this
while (pi.hasNext()) {
    int v = pi.next();
    if (v > localMax.get())
        localMax.set(v);
}

// final code, executed by any thread
int finalMax = localMax.reduce(Reduction.IntegerMAX);
```

Using the Reducible object is just like a ThreadLocal object, with the addition that the local values may be reduced by any thread. The current implementation provides a number of type-specific reductions, the example above using Reduction.IntegerMAX shows the MAX reduction on the Integer class. Reductions can generally be performed sequentially or in parallel (using a tree-network [13]). From the user’s point of view, invoking a reduction in parallel is straightforward: the reduce() method has an optional boolean parameter to denote if the reduction is to be performed in parallel:

```java
// reduction executed in parallel
int finalMax = localMax.reduce(Reduction.IntegerMAX, true);
```

A reduction is only calculated once, therefore subsequent calls to reduce() result in the pre-calculated value being returned. Finally, when the user instantiates a Reducible object, they have the option of specifying a default value (which is assigned to each of the thread’s private copy, just like OpenMP’s firstprivate). If no default value has been specified, then a runtime exception is thrown when the user attempts to get() the local value for a thread that has not set() a value or call reduce() when no threads have set() any value.

### 3.4.2 User-defined reductions

Providing only a few common reductions may be insufficient since there may be specific reductions that a programmer requires. The user is therefore allowed to define any custom reduction, which is simply used in place of the supplied reductions. This is achieved by the user providing an object that implements the Reduction interface. Only one method needs to be implemented, defining the reduction of two elements into one. For example, the following defines a possible reduction on Color objects:

```java
Reduction<Color> colorReduction = new Reduction<Color>() {
    public Color reduction(Color first, Color second) {
        // User code defining reduction logic
    }
};
```
A reduction may then be performed as usual, only this time specifying the user-defined reduction:

    Color finalColor = localColor.reduce(colorReduction);

This approach allows the programmer to easily define complex reductions that could even involve entire data structures, for example concatenating lists or maps. The reduction must be associative (the order of performing the reduction makes no difference) and commutative (the order of the thread-local values makes no difference) since the interface does not specify order.

### 3.5 Parallel break semantics

An important concept in iterative computation is the **break** statement. In a sequential loop, this statement stops the execution of any more iterations in the loop. But in a loop being executed by multiple threads, does the **break** statement mean to cancel only in the local thread or across all threads? The decision should be left to the developer, since there are legitimate cases for both approaches as discussed below. The programmer needs to be aware of an important aspect before breaking from a parallel environment [16]. In the sequential version of a loop, the **break** occurs at a particular iteration \( x \) after completing a subset \( S \) of the entire iteration space. In the parallel traversal of the loop, iterations are partitioned between the different processors according to a particular scheduling policy. If the parallel loop terminates also at iteration \( x \), the completed iterations are not necessarily the same as those completed during the sequential version (subset \( S \)). However, for a large and common class of problems this does not matter (this condition is in most cases the same as the condition that the iterations are independent). A prominent example is searching in a data structure until a certain object has been found.

#### 3.5.1 Global **break**

The most likely type of break to be performed in a parallel loop is that of a global break: this is when the programmer wishes to cancel loop iteration across all threads. For example, an item has been found in a parallel search or the user pressed the cancel button. In such situations, all threads should stop their iterations. This is achieved by one thread invoking **globalBreak()** on the Parallel Iterator. All threads will then receive a **false** the next time they call **hasNext()** and they all return synchronized from this last **hasNext()** call. The advantage of this approach is that each thread breaks out of its loop in a controlled manner at an iteration boundary.

```java
boolean itemFound = false;
while ( pi.hasNext() ) {
    itemFound = searchDocument( pi.next(), searchQuery );
    if ( itemFound )
        pi.globalBreak();
}
```
3.5.2 Local break

An alternative type of parallel loop breaking is that of a local break: only the current thread wishes to cancel, while the other iterations should still be processed. An example includes a program checking whether too many threads are being used and decides the current thread should stop iterating (to reduce disk contention, for instance). In this case the thread should execute a `break` with local semantics so that iterations for the other threads are not affected. This is achieved by invoking a `localBreak()` to allow the Parallel Iterator to clean up, such as releasing any elements previously allocated to the breaking thread. Even though a thread successfully calls `localBreak()`, the `hasNext()` remains a synchronization barrier to ensure that no thread (including the breaking thread) prematurely progresses past the loop:

```java
while ( pi.hasNext() ) {
    resize( pi.next() );
    if ( tooManyThreads ) {
        boolean broke = pi.localBreak();
        ...
    }
}
```

An important aspect of the local break is that it guarantees unprocessed elements (originally assigned to the breaking thread) to eventually be processed; this is what distinguishes a global break from a local break. Even if all threads call `localBreak()`, the Parallel Iterator will guarantee that at least one thread remains to traverse any unprocessed elements. Therefore, a boolean is returned from the `localBreak()` to denote whether the current thread was successfully excused from traversing the loop. If a thread calls `localBreak()`, and receives `true`, then all its previously allocated elements are released. The other threads traversing the Parallel Iterator will then share these elements (after those threads complete their normal iterations) using a dynamic schedule with chunk size 1 (regardless of the original schedule).

3.6 Parallel semantics for exceptions

An exception is an event that diverts a program from its normal execution flow [17]. Many programming languages support exceptions to separate error-code from the actual user code and generalize error handling. Exceptions are important in object-oriented languages. The duties of the Parallel Iterator are solely to dispense elements amongst threads. As such, the Parallel Iterator must be aware of what the threads are doing. Consequently, the Parallel Iterator itself cannot “catch” exceptions. However, when used in combination with standard exception handling mechanisms, the Parallel Iterator provides a convenient interface to manage exceptions in a meaningful way in a parallel environment. Consider the following loop being executed by multiple threads using a Parallel Iterator:

```java
while ( pi.hasNext() ) {
    ...
    // Exception thrown
    ...
}
```
In the Java threading model, for example, an exception encountered within this loop will be propagated up the call stack. If the exception is unhandled, the thread will terminate: all the other threads continue executing and are completely oblivious to the thread’s termination. Even if the exception is eventually handled (but not handled within scope of the Parallel Iterator), then that thread has still abruptly exited traversal of the parallel loop without the Parallel Iterator’s awareness. All the other threads will forever wait at the barrier since the (potentially terminated) thread still has not come back to call `hasNext`.

To overcome this, exceptions must be caught (either using a try/catch or try/finally) within the Parallel Iterator’s scope so that the Parallel Iterator may be acted upon; otherwise, the Parallel Iterator remains unaware of the thread’s state and continues forcing other threads to wait (this idiom of catching potential exceptions is equivalent to that of Java Locks [15], where any potential exceptions must be caught in order to release the Lock). When a thread encounters an exception, the following behaviors are possible:

- **Do nothing:**
  The thread catches an exception but decides to ignore it, so it continues to call `hasNext()` and `next()` as usual

- **Stop locally:**
  The thread catches an exception and decides only it should stop iterating, so it calls `localBreak()`

- **Stop globally:**
  The thread catches an exception and decides all iterations should stop, so it calls `globalBreak()`

This shows that the `localBreak()` and `globalBreak()` of section 3.5 are crucial in meaningful exception handling.

**The register helper method**

It is difficult to define what should happen when an exception is encountered in a parallel loop [18]. In a sequential loop, an exception may be encountered by any iteration: the exception is propagated up to the nearest handler and it is assumed this was the only exception to have occurred. However, in a parallel loop it is possible that multiple concurrent iterations encounter an exception (for example, `FileNotFoundException`). In such a case, the programmer may want to know about all the exceptions that occurred.

The `hasNext()` method acts as a boundary between the iterations: a new iteration $i$ is said to begin when `hasNext()` is invoked, regardless of whether the element has been accessed yet or not. If the element has not been accessed yet (using `next()`), then we are in the pre-`next()` region (this implies that the thread is intending to invoke `next()` to access the element). If the element has been accessed, but still not finished the iteration, then we are in the post-`next()` region. The iteration of $i$ is considered complete only when the thread invokes `hasNext()` again, where the next element becomes the current element being traversed by this thread.

In order to handle exceptions in parallel, the programmer would need to manually implement logic to record the exception, the iteration in which it occurred, the thread that encountered it and then notify the other threads. Managing all this is
difficult for the programmer, especially in determining the current iteration if the exception occurs in the pre-`next()` region. The Parallel Iterator provides a helper method to conveniently record this information. The programmer is only required to catch exceptions (standard procedure as if using a sequential iterator). When an exception is caught, `register(Exception)` is invoked which will record within a `ParIteratorException` the Exception encountered, the Thread that encountered the exception (for potential debugging purposes), and the iteration in which the exception occurred (determined using `hasNext()` as iteration boundary). Consider the following example of processing a collection of files, where a number of exceptions could arise:

```java
while ( pi.hasNext() ) {
    try {
        // Exception thrown
    ...
    } catch(FileNotFoundException fe) {
        // no thread will stop, just record exception
        pi.register(fe);
    } catch (TooManyThreadsException te) {
        pi.register(te); // disk contention, reduce thread count
        pi.localBreak(); // only the current thread will stop
    } catch (DiskFullException de) {
        pi.register(de); // full disk, might need to clean up
        pi.globalBreak(); // all other iterations should stop
    }
}
```

In some cases, the programmer may only wish to record the exception and then continue processing the other elements; in other cases, the programmer may wish to either cancel the current thread or all threads. Even if the programmer does not wish to use the `register()` method provided, they must at least catch the exceptions so that the thread does not terminate before it has a chance to inform the Parallel Iterator (otherwise the other threads will continue waiting for it to complete). Finally, when the Parallel Iterator has completed traversing the collection, all exceptions that occurred during the parallel traversal may be accessed using `getExceptions()` (the programmer may then use Java standard methods such as `printStackTrace()` on the exception object to determine the exact location at which the exception occurred).

3.7 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 (for example all nodes on the same level are independent of each other, and in particular all leafs).
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. This paper proposes a generic and flexible Tree Parallel Iterator that does exactly that. It implements the same simple interface as the regular Parallel Iterator and provides an encapsulated, tested and optimized form of parallel processing. Apart from its direct contribution, it illustrates the power of the Parallel Iterator concept in conjunction with sophisticated forms of parallel processing. Many other parallel processing patterns can be elegantly implemented using this concept.

In order to traverse a tree, one needs a tree collection. Since Java does not have a tree collection as part of the Collections framework, a Node interface is defined to represent the nodes in a tree. All nodes in a tree, except for the root node, have a parent node. A node may have children nodes, otherwise the node is a leaf node. By storing data within this interface, users may take advantage of the Tree Parallel Iterator. In addition to this generic Tree Parallel Iterator, a Document Object Model (DOM) [15] Tree Parallel Iterator is defined to traverse a DOM Document. Such a Document represents an entire HTML or XML document, and will be used in the following example. From the user’s point of view, the Tree Parallel Iterator is a generic extension of the Parallel Iterator.

**Partial ordering in the Tree Parallel Iterator**

The Tree Parallel Iterator implements a well-established work-stealing schedule [19]. The schedule in terms of the partial order semantics from the user’s point of view is discussed here, while the implementation is discussed in section 4.6. As will be shown, the user code to traverse a more complex collection (such as a tree) is no more complicated than the code to traverse a simple collection (such as a list); this is essentially the objective behind iterators! This shows how the Parallel Iterator concept may be extended to incorporate other schedules to traverse other collections.

The partial ordering requires that a node (i.e. an iteration in the Parallel Iterator) only be executed when its parent has completed. Therefore, only the root node may initially be processed. When the root node has completed, its direct children nodes may be scheduled; the children’s children nodes remain unscheduled until the respective parent node has completed. This partial ordering retains the structure of the tree; without such a policy, the nodes in the tree are no different than elements in a list (in which case the standard Parallel Iterator may be used). In addition to maintaining the tree structure, this policy allows us to achieve the following:

- A node $n$ (and subsequently the sub-tree rooted at $n$) may safely be removed during parallel traversal (i.e. extending the remove() method from the iterator interface),
- Similarly to remove(), a replace() method may also be defined in parallel semantics (illustrated below).

**SVG shape recognition example**

The following example application involves traversing a Scalable Vector Graphics (SVG) file [20]. The SVG file is essentially an XML document that defines vector-based graphics. The application is a parser that reads an input SVG file containing
various shapes defined as generic path elements\(^1\). A path is essentially a finite series of \(x,y\) points, meaning that a circle might be defined by multiple points around its circumference (rather than defining it using the center point and radius). Not only does this reduce the graphics resolution (i.e. it is no longer truly scalable as SVG should be), but it also increases the SVG file size. Therefore, the program transforms the input SVG file into a true SVG file, by replacing paths with the correct shapes and their parameters. DOM Document represents the tree being traversed:

```java
domParser = new DOMParser();
domParser.parse(new File("shapes.svg"));
Document doc = parser.getDocument();
```

Once the Document has been constructed, it is passed to the Tree Parallel Iterator:

```java
DomParIterator pi = TreeParIterator.createParIterator(doc, numThreads);
```

and a number of Reducible objects (section 3.4.1) are created to calculate the total number of each shape and the biggest shape:

```java
Reducible<Integer> numCircles = new Reducible<Integer>(0);
Reducible<Integer> numRectangles = new Reducible<Integer>(0);
Reducible<Shape> biggestShape = new Reducible<Shape>(null);
```

By specifying the Document (i.e. the tree) and the number of threads, a Parallel Iterator has just been constructed: it may be used as seen throughout this chapter. The application is presented in full below, and then the various parts are explained:

```java
// each thread does this
...
// private variables for each thread
1: int myCircles = 0;
2: int myRectangles = 0;
3: Shape myBiggestShape = null;
...
// start loop
6: while (pi.hasNext()) {
7:  Node n = pi.next();  // DOM Node
8:  ...
9:  Element e = (Element) n;
10: if (e.getNodeName().equals("g")) {
11:    String visibility = e.getAttribute("visibility");
12:    if (visibility.equals("invisible")
13:      pi.remove();
14: } else if (e.getNodeName().equals("path")) {
15:    Shape shape = ShapeGuessor.guess(e);
16:    if (myBiggestShape == null) {
17:      myBiggestShape = shape;
18:    } else if (shape.getArea() > myBiggestShape.getArea()) {
19:      myBiggestShape = shape;
20:    } else if (shape.getArea() > myBiggestShape.getArea()) {
21:      switch (shape.getType()) {
```

---

\(^1\) This application is motivated by the SVG export feature of the OpenOffice.org application suite, which defines all shapes as generic path elements.
// switch on type of shape
22:    case CIRCLE:
23:        // shape is a circle, record the circle's attributes
24:        numCircles++;
25:        Circle circle = (Circle) shape;
26:        int radius = circle.getRadius();
27:        int centerX = circle.getCenterX();
28:        int centerY = circle.getCenterY();
29:        // create a new element to represent the circle
30:        Element newElement = createElement("circle");
31:        // set the attributes to the new circle element
32:        newElement.setAttribute(r, String.valueOf(radius));
33:        ...  
34:        // replace the old path element with the new circle element
35:        pi.replace(newElement);
36:    case RECTANGLE:
37:        ...
38:    default:
39:        // unknown shape, keep as path
40:    }
41: // end loop
42:    // update Reducible values with the private values
43:    numCircles.set(myCircles);
44:    numRectangles.set(myRectangles);
45:    biggestShape.set(myBiggestShape);

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 of the other collections. This is the whole idea behind iterators: to hide the collection implementation. The Tree Parallel Iterator, just like the Parallel Iterator, remains faithful to this idea.

The first point of interest in the above example is that of the remove() method on line 13. This behaves consistently with the idea discussed in section 3.3.3: the last node returned by the Parallel Iterator is to be removed from the underlying collection. However, the semantics of a remove() on a tree varies slightly compared to a flat collection (such as a list). In a tree, removing a node implies that the children nodes are also to be removed. Therefore, calling remove() ensures that the children nodes will not be scheduled for traversal (since they were removed when their parent node was removed). Achieving this is possible, even in a parallel traversal, since the children nodes have not been scheduled. The other point of interest is that of line 31. Similar to the semantics of remove(), the Tree Parallel Iterator also supports a replace() method that allows a new element to be put in place of the last element returned by the Tree Parallel Iterator. In this example, path elements are replaced with the respective shape element.

When each thread completes the loop, it updates the Reducible object to denote the respective sub-results it has computed (lines 40-42). Finally, the main thread will execute the reduction across the respective Reducible values (notice the use of a customized reduction for the Shape object):

```java
int totalNumCircles = numCircles.reduce(Reduction.IntegerSUM);
int totalNumRectangles = numRectangles.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;
}
}

This example did not make use of features such as parallel break semantics (section 3.5) or exception handling (section 3.6) in order to maintain the example size; however, these features may still be used with the Tree Parallel Iterator. In section 5.3, performance benchmarks of the Tree Parallel Iterator are presented (both for this particular SVG example, as well as another computationally intensive application).

4 Implementation

The Parallel Iterator uses the sequential iterator’s standard interface: `hasNext()` only returns a boolean while the `next()` method returns a reserved element, if any, for the calling thread. Although the calls of `hasNext()` and `next()` are interleaved, they need to appear atomic to the threads accessing the Parallel Iterator. The primary purpose of the Parallel Iterator’s `hasNext()` method is to reserve elements. If `hasNext()` has reserved at least one element for the thread, then `true` is returned. Otherwise the thread waits at a barrier until all threads complete, i.e. all iterations have been executed, and then returns `false` to denote the traversal has completed.

4.1 Scheduling policies

Before discussing schedule-specific implementation details, an overview of the main concepts is presented. This is largely based around two categories of collections:

- **Random access collections** (section 4.1.1) provide index-based constant-time access to elements in the collection, such as array-lists or vectors.
- **Inherently sequential collections** (section 4.1.2) do not provide index-based constant-time access to the collection, such as linked-lists or sets. The Parallel Iterator therefore uses a sequential iterator to access elements.

Note that the copying, locking and other management discussed below are all implemented within the Parallel Iterator and hidden from the user.

4.1.1 Random access collections

If the collection to traverse supports index-based access in (amortized) constant-time, then index computations are performed within the Parallel Iterator to keep track of the next iteration for each thread. Consequently, elements are accessed directly using the integer index. Each thread stores the following integers:

- `nextIndex` specifies the next index to iterate
- `chunkStop` specifies the index just outside the current chunk

Using these two indices, and \( n \) to denote the collection size, the index boundary test (used below) is defined as follows:
if (nextIndex > n)
   // no more elements for this thread
else
   chunkStop = min(chunkStop, n) // “left-overs” for last chunk

Whenever nextIndex is less than chunkStop, this means at least one element (i.e. the element at nextIndex) remains for the thread. If so, this element is returned. Otherwise, depending on the scheduling policy, one of the following is performed to determine the next chunk of elements:

- Static scheduling:
  1. Increment nextIndex by chunksize \times (numThreads - 1). This moves nextIndex to the start of the next chunk
  2. Increment chunkStop by chunksize \times numThreads. This moves chunkStop to the end of the next chunk
  3. Perform index boundary test

- Dynamic or guided scheduling:
  For dynamic and guided scheduling, nextFree is a shared integer index that represents the start of the next chunk to be allocated.
  1. Obtain exclusive access to nextFree by acquiring a lock
  2. Update nextIndex to nextFree
  3. (For guided scheduling) calculate new chunksize
  4. Increment nextFree by chunksize
  5. Perform index boundary test
  6. Update chunkStop to nextFree
  7. Release lock

Figure 2 shows an example where a list of 9 elements are being traversed using a chunk size of 4. Thread A has iterations 0, 1, 2, and 3 while thread B has iterations 4, 5, 6 and 7. If this is static scheduling, then thread A also gets iteration 8. If this is dynamic scheduling, then the first thread to finish their chunk will get iteration 8. The elements shaded in the collection are those that have been accessed by their respective thread. In the next iteration, thread A will execute 3 while thread B executes 5.
4.1.2 Inherently sequential collections: on-demand copying

In section 4.1.1, elements from the underlying collection were accessed using integer indexing. Unfortunately, this is inefficient for collections that do not support constant-time access to random elements. The working principle now is that the Parallel Iterator essentially acts as a thread-safe wrapper to the sequential iterator. Since the sequential iterator is traversed only once, this means the sequential iterator will not usually be pointing to the next correct element for the current thread. In order to handle this, elements need to be copied (accessed using the sequential iterator) and reserved for the correct thread to access them. During this process, the sequential iterator is locked to avoid other threads interrupting.

The semantics of **copying** as used here actually refers to retaining a pointer to the original object and storing this pointer in a buffer (essentially an array). It does not refer to making a deep clone or duplicate of the original object. In this way, minimal amounts of memory and time are used. The second concept used by the Parallel Iterator that extends copying is **on-demand copying**. Rather than copying all the elements of a collection, elements are copied into the buffer only as required. The advantages of on-demand copying is that copying is performed in parallel as other threads are executing iterations. Also, in the case the Parallel Iterator breaks (as in section 3.5), less time would have been wasted on unnecessary copying. The approach differs depending on the scheduling policy of the Parallel Iterator (the implementation for guided scheduling is essentially identical to dynamic scheduling):

**Dynamic scheduling** When a thread first calls hasNext, it copies the next chunksize iterations: this requires exclusive access (by acquiring a lock) to the sequential iterator as the next chunksize elements are copied to the thread’s private subarray. Locking is not needed for every call to hasNext, but is performed only once every chunksize calls to hasNext. Each thread keeps up to chunksize unprocessed iterations in its private subarray at any one time. The first thread to complete its private subarray will copy the next chunksize iterations. Subsequent hasNext calls will not require locking if there are unprocessed elements remaining in the private subarray.

Figure 3 shows the internals of a Parallel Iterator used to traverse a linked-list (an inherently sequential collection) using a dynamic scheduling policy with chunk size of 3. In figure 3(a), thread B calls hasNext() for the first time. Since none of the iterations have been requested yet, 3 elements are copied over from the sequential iterator to the private buffer for thread B, and true is returned to thread B as shown in figure 3(b). Thread B now has 3 iterations reserved for it. Further calls to hasNext() by thread B do not affect the state of the iterator since there are still 3 elements to be processed, and these are only accessible when thread B calls next().

Figure 3(c) shows the state of the Parallel Iterator when all elements have been assigned to threads (due to hasNext()), but not necessarily processed yet (since some are yet to be accessed using next()). Thread A has completed iterations 3 to 5, while thread B is working on iteration 6 but has already reserved iterations 7 and 8 as well (due to the scheduling chunk size). Therefore thread A invokes hasNext() to see if any more elements remain, but false will be returned.

**Static scheduling** By definition, static scheduling requires that the iterations are allocated first before any iteration is executed. But rather than naively copying all the
elements at the beginning, the elements are copied on-demand. This is illustrated using the collection of figure 4, where thread A is to be allocated iterations 0, 1, 4, 5 and 8 while thread B gets 2, 3, 6 and 7 (i.e. static cyclic scheduling with chunk size 2). The Parallel Iterator for this implementation is visualized using figure 4. In figure 4(a), it is seen that only iterations 0 to 3 have been copied over into the subarray section; thread A has also completed iteration 0 while thread B has completed iterations 2 and 3. The sequential iterator is currently pointing at iteration 4 as the next element to copy across.

Assume now that thread B is ready for its next iteration: iteration 6. Since the sequential iterator is still pointing at 4, thread B copies across all iterations up to iteration 7 as in figure 4(b); it then processes iteration 6 and leaves iterations 4 and 5 for thread A. This is on-demand copying, where iterations are only copied when required. When thread A finishes iteration 1, it will find that its next iteration chunk (iterations 4 and 5) has already been copied over. The advantage of on-demand copying is that the copying is not a startup cost, since the copying and processing (by other threads) occurs at the same time and overlaps with computation.

4.2 Reductions

The Reducible class may essentially be viewed as an extended thread-local variable with thread-local set() and get() methods (just like Java’s ThreadLocal class [15]). In addition to behaving like a thread-local, the class defines a reduce() method that accepts a Reduction object:

```java
public class Reducible<E> {
    private Map<Thread,E> locals = ...;
    public E get() { ... }
    public void set(E e) { ... }
    public E reduce(Reduction<E> red) {
```
Fig. 4 Static scheduling with on-demand copy. Situation shown when thread \( B \) needs its next iteration (element 6), but the sequential iterator still points at element 4 (due for thread \( A \)) (a). Therefore, thread \( B \) also copies elements on behalf of thread \( A \) (b).

```java
if (has not been reduced yet)
    reduce local values and store result
else
    return result
```

Unlike a typical thread-local method which behaves locally to the calling thread, the `reduce()` method may be invoked by any thread. All the internal thread-local values within the `Reducible` object are accessed and reduced to a single result using the `Reduction` object (which defines an arbitrary reduction); the result is then stored in the case `reduce()` is called again (i.e. the reduction is performed only once).

### 4.3 Parallel remove

The `remove()` acts on the last element given to the respective thread through `next()`. Consequently, the Parallel Iterator stores this element until the thread calls `next()` again (in which case the new element is stored). In the case of most Parallel Iterator implementations, this element is stored into a collection containing elements to be removed; at the end when all threads reach the synchronization barrier, one thread commits all the deletions from the underlying collection. Java’s `removeAll()` method defined in the `Collection` interface is used to ensure efficiency.

### 4.4 Parallel break

The local and global breaks are implemented as conditions inside the Parallel Iterator. An invocation of `localBreak()` will set the condition for that thread. The `globalBreak()` sets the condition for all threads. This way, all subsequent calls to `hasNext()` made by any thread return `false`, regardless of the number of elements yet to be processed. If a thread successfully breaks from the Parallel Iterator when it calls
localBreak(), all reserved elements are released for the other threads to traverse. The breaking thread then waits at the barrier in the hasNext() method (assuming the default barrier was not disabled). When globalBreak() is called, the current iterations being executed are allowed to complete. For example, assume thread A is working on an iteration and then thread B calls globalBreak(). If thread C calls hasNext(), it will be blocked (to synchronize loop termination between all threads) and false will be returned as soon as currently executing iterations are completed. Note that OpenMP does not allow a break statement within a work sharing construct such as parallel for.

4.5 Exceptions

When a thread invokes register() on the Parallel Iterator, it stores the exception to a java.util.concurrent.ConcurrentHashMap. The Parallel Iterator creates a wrapper exception (ParIteratorException) to associate other information with the exception. Along with the exception that has occurred, this also includes the current thread registering the exception (determined using Thread.currentThread()) and the element the thread is currently working on (according to the hasNext boundary as explained in section 3.6).

4.6 Tree Parallel Iterator

The Tree Parallel Iterator must enforce a partial order: a node may only be executed when its parent node has completed. To enforce this partial order, the Tree Parallel Iterator only enqueues the children of a node at the time it completes. A potential scheduling scheme to achieve this partial ordering is work-stealing [21]. The variant that the Tree Parallel Iterator implements is based on the randomized work-stealing [19], which will be summarized below in the context of the Tree Parallel Iterator. Each thread has a private deque (a double ended queue) to store nodes that are ready to execute. When a thread operates on nodes on its own deque, a last in first out (LIFO) policy is used (therefore operating on the latest local node). When a thread’s private deque is empty, it becomes a thief and selects a victim thread at random. The thief attempts to steal the oldest node on the victim’s deque, therefore using a first in first out (FIFO) policy when stealing.

The benefit of work-stealing has been extensively developed and evaluated [22]. The reason that nodes are executed using a LIFO policy on the local deque is to take the initiative to execute a process towards the depth of the tree [21]; such behavior models that of a sequential depth-first traversal (therefore reducing parallelism overhead when sufficient work exists). Threads encourage parallelism when they steal with a FIFO policy, as this expands the breadth of the tree (the thief takes ownership of a new sub-branch). This naturally encourages good data locality and cache reuse [23] since thieves favor the victim’s oldest node (i.e. the coldest node in the victim’s cache), while the hottest nodes are left for the local thread.

Figure 5 shows an example of 3 stages of the Tree Parallel Iterator, where 2 threads are traversing a tree consisting of 10 nodes. As discussed in section 3.7, a node is only scheduled to execute when its parent node has been completed. Therefore,
Initially only the root node is ready for execution. Thread B steals the oldest node from thread A. Both threads take from their local deque.

Fig. 5 Example run of the work-stealing implementation using 2 threads. Each thread maintains a local deque to store nodes that are ready to execute. Threads favor nodes on their local deque using a LIFO policy, while threads steal from other deques using a FIFO policy. Nodes are only added to the deque when the respective parent node has completed.

initially only the root node is ready to execute, and this is placed on one of the thread’s deque (figure 5(a)). In the context of the Parallel Iterator, the thread to process this node is the one that calls hasNext() first: in this 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. At this stage, the Tree Parallel Iterator will enqueue the children of node 0 to thread A’s private deque (figure 5(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 (node 1). In the meantime, thread A takes it’s latest node (node 3) and executes it. Thread B has encouraged parallelism by stealing work in a breadth manner. On the other hand, thread A has sufficient amount of work and continues to execute in a sequential depth-first manner (therefore minimizing unnecessary parallelism overhead).

When thread B completes its computation (figure 5(c)), it enqueues the children (nodes 4 and 5) of the last node it completed. Consequently, thread A does not need
to perform another steal since it has unprocessed nodes on it’s deque; thread A now executes its most recent local node. Similarly, node 9 becomes thread A’s most recent local node when node 3 is completed. The parallel traversal of the tree is considered complete when all threads are attempting to steal. The remove() for the Tree Parallel Iterator is implemented similarly as discussed for the other collections in section 4.3, with the addition that children nodes are not enqueued if a remove() was called.

The major advantage of the Parallel Iterator is that implementation details are hidden from the user. First, by encapsulating the parallelization logic within the Parallel Iterator, this separates it from the business logic of the loop body (this is especially valuable when traversing non-trivial collections in parallel, such as trees). As a result, the business logic is very similar to the sequential version. Second, the scheduling scheme details are hidden: this allows for other 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, therefore processing the tree in a bottom-up manner). Since the implementation details are hidden from the user, the user code (to iterate nodes) does not require modification when a different scheduling scheme is used.

5 Performance

In this section, the Parallel Iterator is extensively tested using a number of benchmark applications for both the C++ (Qt) and Java implementation. The objective is to evaluate the overhead it introduces and its ability to exploit the inherent parallelism of an iterative computation. Hence, we compare the scalability across a number of processors and the overhead in comparison to the sequential execution using sequential iterators (i.e. the baseline). The benchmarks were executed 4 times, taking the median of each. The times were measured using the respective timing functions provided within the Java or C++ (Qt) libraries.

The performance of the Parallel Iterator is compared to the traditional Java parallelism approaches in section 5.1. Section 5.2 focuses on more disk-intensive and potential desktop applications, and compares the performance of reductions to a commercial implementation of Qt. Finally, the Tree Parallel Iterator is evaluated in section 5.3. All the C++ (Qt) benchmarks use OpenMP to create threads, while the Java benchmarks use Java threads. The benchmarks ran on shared memory systems which may be considered typical future desktop platforms running Linux. The first has two Quad-Core Intel Xeon processors (total of 8 cores) running at 1.86GHz with 8GB of RAM. The second system has four Quad-Core Intel Xeon processors (total of 16 cores) running at 2.4GHz with 64GB of RAM.

5.1 Comparing to traditional Java parallelism approaches

We first present results to emphasize the competitiveness of the Parallel Iterator, even in terms of speed. Performance of the Parallel Iterator is compared to some of the traditional approaches (discussed in section 2.2) that programmers would typically take to parallelize traversal of a collection of elements (namely in Java). We further break down the locking approach into two categories, depending on the underlying
The Parallel Iterator is compared to some of the traditional Java parallelism approaches using different workloads.

Implementation of the Lock: **fair locking** (when the lock is competed for, access to the lock is favored for the thread that has been waiting the longest) versus **unfair locking** (when competed for, no order is guaranteed for access to the lock) [15].

The two graphs in figure 6 represent different workloads (each workload contains 1 million iterations). Both benchmarks compute a synthetic load (here the Newton-Raphson method) in each iteration. Figure 6(a) shows the speedup for a balanced, but fine-grained, workload (each iteration takes on average 3.5µs). 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 6(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 6(a)), while...
it performed worst in the second workload (figure 6(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.

5.2 Disk-intensive applications

We now investigate the scalability of the Parallel Iterator using more realistic desktop applications, in particular those requiring high amounts of disk access. These results use the C++(Qt) implementation of the Parallel Iterator.

5.2.1 Image resizing

The next experiment involves image resizing. This is an example of a typical desktop application with inherent parallelism. Three different image sizes were used: small (400x300), medium (800x600) and large (1600x1200). Testing was performed on three sets of images, each containing a total of 96 images as shown in table 1:

<table>
<thead>
<tr>
<th>Set</th>
<th>#Small</th>
<th>#Medium</th>
<th>#Large</th>
<th>Image Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>96</td>
<td>uniform</td>
</tr>
<tr>
<td>B</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>by size (L,M,S)</td>
</tr>
<tr>
<td>C</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>random</td>
</tr>
</tbody>
</table>

Table 1: Three sets of images were used in the image resizing benchmark.

Figures 7(a), (b) and (c) show the speedup over the number of employed processors for the different scheduling policies and chunk sizes. Figure 7(a) is for set A, figure 7(b) is for set B, and figure 7(c) is for set C. Regardless of the scheduling policy or set of images, all three figures show the parallel code executed with imperceptible overhead when running on a single processor.

Dynamic scheduling with low chunk sizes performed most consistently across all sets of images, achieving speedups between 90-99% of the ideal speedup. Figure 7(b) shows static block and guided scheduling performed poorest when using set B, producing speedups of roughly 47% of the ideal speedup. This is expected as the first thread gets the biggest portion of the large images. However, guided scheduling performed well on sets A and C, commonly producing speedups between 90-99% of the ideal speedup. Static scheduling really only performed well for set A, with speedups of over 95% of the ideal speedup when the iteration space is divided evenly amongst the threads.
These results once again confirm the importance of having a flexible way to modify the scheduling scheme of an iteration space. Static scheduling with a block chunk size produced poor results for computations with varying loads, yet this is possibly one of the easiest policies for programmers to manually achieve (section 2.2). Dynamic scheduling with a low chunk size would be most effective for image resizing, yet it was inappropriate for the Mandelbrot application. Consequently, the Parallel Iterator provides this flexibility since the chunk size and scheduling policy are parameters that may be chosen dynamically during run-time.

5.2.2 Word count and permutation (reductions)

The word count benchmark reads text files in a folder (and subfolders recursively) and counts the occurrence of each word. Such an application is typical of a search functionality on a desktop environment. The word permutation benchmark also recursively reads text files in a folder, but also performs more complex permutations on each word. Such behavior would be typical of a spell checker functionality on a desktop environment. In both benchmarks, since files are searched in parallel, a reduction is required to collate individual file results into one final result set.

Figure 8 shows the speedup of the word count benchmark, comparing QtConcur- rent (QTC) (section 6) with the Parallel Iterator (PI). Two folders with C++ source
files were used, one contained 650 files while the other contained 1755 files; the average sequential times were 0.5 and 2.1 seconds respectively. The first observation is the performance degradation due to the disk access bottleneck, but remember this is a realistic desktop application. The Parallel Iterator scaled better than QtConcurrent reaching a speedup of 2.9 with 6 processors for 1755 files. QtConcurrent peaked at 4 processors with a speedup of 2.2, but this required disabling processors since there was no support to limit the thread count\(^2\). The Parallel Iterator is more flexible since it allows the thread count to be adjusted as required per loop.

The speedup for the computationally intense word permutation program is shown in figure 9. Two folders containing C++ source files were used; sequential times took 18 seconds for 85 files and 186 seconds for 650 files. Disk access became less of an issue because more computation occurs for each file. On average, the Parallel Iterator performed slightly better than QtConcurrent on both folders. However, with only 85 files used, the efficiency slightly degrades as more threads came into contention.

In figure 8, the Parallel Iterator’s performance is slightly better than QtConcurrent. This may be attributed to the high number of reductions that QtConcurrent makes. Assume a parallel program where N elements and P threads are involved. For QtConcurrent, N-1 reductions (therefore 1754 reductions in the case of the 1755-file word count) are required regardless of the thread count. The Parallel Iterator however only requires P-1 sequential reductions (therefore 7 when 8 threads are used). In addition to this, QtConcurrent requires that the code for each iteration be restructured into a separate method.

5.3 Tree Parallel Iterator

In this section we evaluate the Java implementation of the Tree Parallel Iterator as discussed in section 3.7. We use two benchmarks: the first is the SVG shape

\(^2\) In the meantime Qt 4.4 introduced the QThreadPool class to allow control over thread count. However, this thread count will consequently be applied to the entire ThreadPool, not only to one loop.
application presented in section 3.7, while the second one is more computationally intensive. The important observation here is that good speedups can be achieved with the Parallel Iterator for very high level programs, employing object-oriented code, XML and SVG. In both benchmarks, the baseline is the sequential implementation that makes use of the standard Java sequential iterator.

We are currently unaware of any existing tools that enable the parallel traversal of tree collections in Java, in particular with the requirement that iterations are only executed when the parent node (i.e. iteration) has been completed. To parallelise such a collection, programmers would be required to manually manage the scheduling of nodes (iterations). Therefore no results are available comparing these benchmarks to other tools. Instead, these benchmarks illustrate the performance one would expect from using the Tree Parallel Iterator, depending on the characteristics of their application.

Figure 10 shows the speedup for the processing of an SVG file where the shapes are recognized. The 4 workloads shown denote the shape types in the SVG file. Notice that triangles and rectangles are a lot easier to recognize than ellipses. The fourth workload is a mixture of triangles, rectangles and ellipses. Just as we have explored in the previous benchmarks, speedup improves with more computationally intensive applications.

The next set of benchmarks involve computing a synthetic load (here the Newton-Raphson method to produce a Newton fractal) at each node of the tree. Our first benchmark of figure 11(a) 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 11(b) repeats the same experiment, only this time performing more computation at each node: in this case, each execution of the Newton-Raphson method 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 11(b) 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 3.7, but 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 all implementation details from the user, in particular the scheduling and synchronization of nodes amongst multiple threads.

6 Related work

The importance of loop parallelization and loop scheduling have been extensively studied before [10]. The work presented here is distinct since it promotes preserving the qualities of object-oriented sequential code while still providing flexibility. It applies standard parallel concepts (such as scheduling policies and reductions) in a way object-oriented programmers are familiar with, namely using iterators and
without code restructuring. The semantics for local and global breaks have also been integrated with the Parallel Iterator. In other tools, if the programmer wants the behavior of the global break then they must manually implement this.

Over 100 proposals for concurrency in object-oriented languages were surveyed in [24]. Not all of the proposals contained solutions for data parallelism; the most influential and relevant ones are discussed below in addition to others. At first glance, it may seem that many previous attempts have proposed a solution to the problem of parallel iteration in object-oriented languages. However, the Parallel Iterator presented here differentiates in important aspects.

Some approaches such as DatTel [25], Parallel Standard Template Library (PSTL) [26] and Standard Template Adaptive Parallel Library (STAPL) [27] aim to provide parallel extensions to the Standard Template Library (STL) [28] in C++. DatTel is a data-parallel template library that overloads certain STL functions. Although the parallelism is hidden from the user, DatTel is not suited for mainstream object-oriented parallel computing. It is restricted to containers composed of simple data types and therefore does not support collections with user-defined types which is an essential aspect in object-oriented programming.

PSTL provides a \texttt{par_apply} algorithm; STAPL provides a \texttt{pforall} parallel region manager. Both of these are used similarly to the \texttt{parallel_for} in Threading Building Blocks (TBB) [29] which was recently released by Intel for parallel programs in C++. Unfortunately, STAPL, PSTL and TBB do not allow the user to directly traverse collections (examples of valid ranges to traverse include integers, STL random iterators and pointers). The main problem is the restructuring of the code and the creation of a new object. First, the programmer must create a function object to specify work to be done for a range of elements. Secondly, the context of the code changes since now the logic is in another class (the function object).

CC++ [30] provides a \texttt{parfor} statement (parallel semantics of the sequential \texttt{for} statement) to iterate over a collection where each iteration is executed in parallel. However, efficiency is lost when iterations contain low computation since the initialization, update and test parts of the \texttt{parfor} loop remain sequential. In [31], a \texttt{for_each_par} template method was proposed based on the idea of range partition adaptors which converts a range into a collection of sub-ranges. However, only collections with random access iterators are supported and the user must create objects to partition the collection.

Other attempts exist but have been superseded by OpenMP’s \texttt{parallel for} [6]. Despite OpenMP’s success, it is insufficient for many object-oriented collections as it can only be applied to an integer range and the collection needs to be accessible using the loop index. It therefore cannot be used directly to traverse many collections, such as linked-lists or sets. Furthermore, the Parallel Iterator allows for several features not available in OpenMP. To name just a few: user-defined reductions, exception handling mechanism, global and local break semantics. OpenMP provides a \texttt{reduction} clause but is limited to only a few predefined reductions; furthermore, aggregate types (such as arrays), pointer types and reference types may not appear in the reduction clause.

The style and purpose of iterators in scientific-targeted languages such as Chapel [32] is substantially different from that of Java or C++. The purpose of iterators here is to traverse a collection; Chapel iterators on the other hand are essentially functions that return a sequence of values (as opposed to a function that only returns
one value). Therefore, the iterator concept in Chapel is tightly coupled with the loop; however, in Java and C++, iterators are associated with a collection of elements [32]. The semantics and syntax of the Chapel iterator is very similar to that of the CLU iterator [33].

Modern parallel languages (such as Chapel, Fortress [34] and X10 [35]) tend to target large-scale scientific applications; therefore they focus on a distributed memory model. Since this paper addresses desktop applications in light of mainstream multi-core processors, the focus is on a shared memory programming model. The different target applications is further highlighted by the code syntax of Fortress: it mimics that of mathematical notation as it is aimed for scientists.

The Microsoft Parallel Extensions [36] to the .NET Framework support parallel semantics of the foreach statement with the `Parallel.ForEach` static method. Since the parallelism is controlled by the Parallel Extensions, the programmer does not have any control over threads, scheduling policies and parallel break semantics. By catching an `AggregateException`, programmers may handle exceptions thrown from the body of the `Parallel.ForEach` method. Although programmers can analyze thrown exceptions, it is unknown which iteration or thread threw the exception.

Some approaches support aggregate operations, such as PLINQ [37] and ParallelArray [38], thereby removing the loop altogether. Consider for example ParallelArray. The collection to traverse in parallel is stored in a ParallelArray class:

```csharp
ParallelArray<Image> images = ...;
```

This allows various operations to be applied:

```csharp
images.apply(resizeProcedure);
```

By removing the loop, this is essentially a change in the programming style; it provides a higher level and black-box style of programming. Whether this is an advantage or disadvantage is a matter of preference: those with database programming or functional programming experience may prefer this style [38]. The Parallel Iterator is not a new programming style: it is an extension to the sequential iterator using the same object-oriented principles. For example, sometimes the programmer may want more explicit control: the Parallel Iterator allows this with minimal difficulty, such as fine-tuning the number of threads, scheduling policy, or early termination.

More recently, QtConcurrent [39] provides a `mappedReduced` method that is based on MapReduce [40], where the programmer specifies two methods. The first method (executing in parallel) defines the computation to be performed on individual elements from the collection (the map). The second method (executing sequentially) combines the intermediate results into one final result (the reduce). Just like TBB’s reduction, this requires restructuring the loop code into a new method defining work for one element; every loop iteration now results in a separate method call. QtConcurrent’s reduction method is executed as many times as there are elements. With the Parallel Iterator, reductions are only executed as many times as there are threads. This potentially improves performance since less time is spent in the reduction stage, as observed in the experimental results of section 5.2.2.

The implementation of the work-stealing [21] schedule is based on the randomized work-stealing variant [19]. Processing an XML document in parallel has been explored using both static [41] and dynamic [23] partitioning schemes. However, the implementation of the Tree Parallel Iterator was merely to demonstrate how the
simplicity of the Parallel Iterator concept is applied to hide the details of traversing complex data collections in parallel.
7 Conclusions

This paper presented the Parallel Iterator for object-oriented programs on shared memory systems. The Parallel Iterator concept allows parallel traversal of a collection of elements while the structure of the program remains unchanged. This may be performed virtually with any collection, even those inherently sequential, therefore being a faithful extension to the sequential iterator. The interface of the Parallel Iterator even imitates the standard Java-style sequential iterator. The Parallel Iterator promotes encapsulation and separation of concerns by hiding parallelization and collection details from the programmer. When used in combination with OpenMP, the structure of the sequential program remains virtually unchanged.

The core concept of the Parallel Iterator is sufficient for many, but not all iterative computations. As such, the scope of the Parallel Iterator was expanded to solve more parallel computing situations. The first includes user-defined reductions in an object-oriented approach, essentially allowing programmers to realize arbitrary reductions for any data type. The second included support for exception handling in a parallel loop. This is especially important for object-oriented languages. Another feature included parallel semantics for the loop break, and also parallel semantics for removing elements during traversal. These features complement each other and can even be used elegantly together. The Parallel Iterator concept was further extended to allow parallel traversal of tree structures, such as XML documents, in the form of the Tree Parallel Iterator.

In addition to the ease of use compared to other common approaches, the results show negligible overhead with effective load scheduling which produce the expected inherent speedups. It was also confirmed that flexible scheduling policies are important and this is easy with the Parallel Iterator, even decidable dynamically at run-time. Such fine-tuning available to the programmer includes scheduling scheme, chunk size and the number of threads involved. In particular for interactive desktop applications involving heavy disk usage, controlling the thread count is important. The performance of the Parallel Iterator is in many cases superior to that of traditional parallelism approaches, including QtConcurrent.

Future work

Overall, the major components of the Parallel Iterator concept have been deeply studied. Consequently, both the semantics and the implementation of the Parallel Iterator are stable. In fact, the Java implementation of the Parallel Iterator has been released and is available for public use. Parallel Task will follow shortly. However, there are some interesting aspects that could be explored further:

- Implementing and evaluating further scheduling schemes for the Tree Parallel Iterator (such as bottom-up dependences, or other variants of the work-stealing).
- Study and optimize the Parallel Iterator for data locality. For example, the distribution of elements amongst threads and how performance is affected by cache effects (for example, to avoid false sharing) and NUMA (Non-Uniform Memory Access) systems.

3 www.ece.auckland.ac.nz/~sinnen/ParallelIterator
References


