Object-Oriented Parallelisation: Improved and Extended Parallel Iterator

Nasser Giacaman, Oliver Sinnen and Lama Akeila
Department of Electrical and Computer Engineering, The University of Auckland, New Zealand
ngia003@aucklanduni.ac.nz, o.sinnen@auckland.ac.nz, lake003@aucklanduni.ac.nz

Abstract

The need to parallelise desktop applications is becoming increasingly essential with the mainstream adoption of multi-cores. In object-oriented languages, sequential iterators handle iterative computations of a sequential program; similarly, the parallel iterator was developed to handle the iterative computations of a parallel program. This paper presents the progress of the parallel iterator concept. New features, such as support for reductions and global break semantics, allow the parallel iterator to undertake more situations. With a slight contract modification, the parallel iterator interface now mimics that of the sequential iterator. All these features combine together to promote minimal, if any, code restructuring. The reduction frequently outperforms related work and the importance of providing simple and flexible fine-tuning capability is affirmed.

keywords: parallel computing, object-oriented programming, parallel iterator, loop scheduling, reductions

1. Introduction

Developing parallel programs has traditionally been, and still remains, a challenging task. Parallelising a desktop application, in comparison to traditional embarrassingly parallel applications, further complicates this task; such applications tend to be user-interactive, run on non-dedicated systems, have short runtimes and irregular structures. However, it is clear that these desktop applications must be parallelised if they are to benefit from multi-core processors that now dominate mainstream computing [11, 3].

The focus needs to be placed on object-oriented languages due to their popularity [12] in desktop application development. For example, common languages for Windows programming include C++, C# and Java. The K Desktop Environment (KDE) for Linux is developed in C++ using Trolltech’s Qt toolkit [13] and Mac OS X is developed using Objective-C. Hence, it is needed to ease the parallelisation of object-oriented programs.

In many applications, the lion’s share of computational load rests within iterative computations. In object-oriented languages, this is typically handled using iterators. Since traditional sequential iterators are inappropriate for parallel programming, the parallel iterator concept [6] was developed for Qt/C++. Since then, the parallel iterator has been advanced to include a better usage contract (as demonstrated using the new Java implementation), support for reductions and support for global break semantics.

Contributions

Further extensions to the parallel iterator concept are presented. This includes a true object-oriented solution for reductions, generalised to allow user-defined reductions on any data type. With the proposal of a minor contract, the interface of the parallel iterator is reconsidered: this allows the exact interface as the familiar Java-style sequential iterator. To achieve early loop termination, global semantics for the break statement are presented. When the parallel iterator is used in combination with OpenMP [9] (which is supported by many compilers, including Visual C++, Intel and GCC 4.2), the loop code does not require restructuring, even when reductions are necessary.

The structure of the paper is as follows: Section 2 provides some background to the sequential iterator and reductions, while section 3 reviews the parallel iterator concept. Recent extensions to the parallel iterator are presented in section 4 while their implementation is discussed in section 5. Section 6 analyses the performance results before concluding in section 7.

2. Background

In object-oriented programming, collections (or containers) are used to store objects. They come in many forms, including lists, linked-lists, sets, maps and stacks. Some of these collections are random-access (constant-time access to random elements), while others are inherently sequential. Regardless of the collection type, iterators provide a consistent means to access the elements.
The Java-style sequential iterator

The Java-style sequential iterator provides two primary functions. The first function, `hasNext()`, inquires to see if any elements remain to be traversed. If this returns `true`, then `next()` will retrieve that element. Below is typical sequential code making use of the sequential iterator to resize a list of images.

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

Different approaches to parallelise this were evaluated in [6]. Unfortunately, most require substantial developer effort, involve code restructuring and have inadequate scheduling policies. Section 3 uses this example to illustrate the parallel iterator concept.

Reductions

A reduction is a standard parallelisation problem [5] and is explained by the simple example in figure 1. The 3 threads calculate the sum of a list containing 9 elements, but each thread calculates only a partial sum for 3 elements. Rather than sharing a variable (the total sum) between all threads, it is better that each thread maintains its own copy of the variable to avoid excessive locking. However, this means each thread will have a partial result (`sumA`, `sumB` and `sumC` respectively) that need to be reduced (i.e. added) into a final result.

3. Parallel iterator

The parallel iterator concept presented in [6] involved a C++ implementation to iterate the collections provided by Qt; this section summarises the parallel iterator concept while section 4 focuses on the new extensions. The parallel iterator may conceptually be considered a thread-safe version of the sequential iterator; as a consequence, it supports virtually all collection types. The parallel iterator does not manage thread creation, but is totally aware of threads accessing it. It has the flexibility to be used with either OpenMP threads or a threading library; OpenMP is recommended since this largely keeps the structure and context of the program unchanged.

Access to shared variables can be handled in the same way, using data-sharing attribute clauses for OpenMP or locks for thread libraries. Any thread may safely modify elements it receives from the parallel iterator but elements should not be inserted or deleted into the collection being traversed (just like Java’s `fail-fast iterators`).

Old interface and usage

The original parallel iterator interface consisted of only one member function: `bool next(E& e)`. This function combined the `hasNext()` and `next()` functions into one indivisible function. A thread invoking `next()` tests to see if any elements remain. If so, the reference variable (denoted by `E& e`) is set to that element and `true` is returned all in one operation. Section 4.2 presents the latest interface contract, which now imitates the sequential iterator’s interface.

In the example below, OpenMP’s `parallel` construct causes multiple threads to execute the `while` loop. The only change compared to the sequential version is the use of the parallel iterator and condensing the `hasNext()` and `next()` functions into one atomic `next()` function. Essentially, the structure of the program remains the same.

```cpp
List list = getFiles();
ParIterator<Image> *pi = Factory::createParIterator<List,Image>(list);
#pragma omp parallel {
    Image image;
    while (pi->next(image)) {
        resize(image);
    } // par iterator implicit barrier
} // OpenMP implicit barrier
```

In order to preserve semantics of the sequential iterator, the last call to `next()` contains an implicit barrier synchronisation. This way, all threads finish iterations before proceeding past the loop. A parallel iterator is easily constructed using the factory pattern below; the user only needs to supply the collection of elements to traverse.

```cpp
template <typename T, typename E>
static ParIterator<E> * createParIterator
    (T& collection,
     int schedulePol = DYNAMIC,
     int chunksize = DEFAULT);
```

For flexibility and performance, an optimal scheduling policy and chunk size may be specified to override the defaults. In order to have a single programming paradigm, the parallel iterator may also be used to traverse arrays and integer ranges using similar constructors.
Flexible scheduling policies and chunk sizes

The major scheduling policies [9] supported by the parallel iterator include:

- **Static**: threads are assigned an equal number of iterations in a round-robin fashion before any iteration begins. This can be either *block* where a single large chunk is given for each thread, or *cyclic* where iterations are grouped into smaller chunks.

- **Dynamic** (default scheduling policy): iterations are assigned only as threads request work.

- **Guided**: similar to dynamic, except the chunk size decreases as iterations are distributed.

The iteration code does not change even though the scheduling policy and collection type have changed. This provides a real object-oriented approach to parallel traversal of any collection. An important assumption is that iterations may be processed in any order, since the parallel iterator will not process them sequentially. However, the parallel iterator currently supports partial ordering by specifying a chunk size and future work will introduce constructs to help programmers specify more complex loop carried dependencies.

4. Extensions to the parallel iterator

The parallel iterator as described in section 3 is sufficient for many, but not all, iterative computations. The scope of the parallel iterator is extended to handle even more situations, without programmers having to manually implement them. For example, it may be necessary to perform reductions across threads or terminate parallel traversal for all threads. Both these features are now integrated with the parallel iterator concept. Furthermore, the parallel iterator concept has successfully been implemented with an alternative interface contract making it even simpler to use.

4.1. Reductions

For programs that share variables, programmers must provide mutual exclusion to produce correct results. In a threading library, this is typically solved using a mutex. Unfortunately, programs with fine-grained parallelism would suffer heavily in performance [2]. As a consequence, the concept of reductions [5] has been integrated with the Qt/C++ version of the parallel iterator. OpenMP provides a restricted solution with the reduction clause where only the reductions +, *, -, |, &, |, ^, &&, and || are supported for C/C++. As well as these common reductions, the proposed concept here also supports others such as *minimum* and *maximum*. But the primary advantage of the solution proposed here is that it is object-oriented in that the user may perform any kind of reduction, using any data type.

There are three components in making such a reduction. First, a Reducible object is created to wrap the variable by providing the initial value. Secondly, the Reducible class will initialise and manage copies of the variable for each thread, and each thread may retrieve a pointer to its private copy through the getMyCopy() method. Finally, when all threads have finished their work, the join() method is invoked to perform the specified reduction on all private copies and return the final result.

The example below illustrates these components when trying to find the minimum in a list of numbers. First, the variable min is declared and initialised. A parallel iterator is created in order to manage the parallel traversal of the list. A Reducible object of type int is then created for the min variable. Prior to starting execution of the loop, each thread requests a pointer to its own private copy of the variable through the getMyCopy() call. After all threads have finished, the join() operation is invoked on the Reducible object specifying the minimum reduction (although this particular simple example could be implemented in OpenMP, it is used to convey the concept which may be applied to more complex situations not handled by OpenMP).

```c++
List list = ...; // get list of numbers
int min = MAX;
ParIterator<int> *pi = Factory::createParIterator<List,int>(list);
Reducible<int> globalMin(min);
#pragma omp parallel
{
  int v;
  int* myMin = globalMin.getMyCopy();
  while(pi->next(v)) {
    if (v < *myMin)
      *myMin = v;
  }
}
min = globalMin.join(Reduction::MIN);
```

The advantage of this approach compared to related work (discussed below) is that only minor code modifications are required. In fact, the structure of the code remains unchanged when used in combination with OpenMP. Section 4.2 even introduces the new parallel iterator interface that further reduces modifications. Even in the case that OpenMP is not used, this concept becomes especially more useful since the programmer does not have to manually implement this.

Figure 2 shows a visualisation of the above code example. The parallel iterator (ParIterator *pi) is used on a list (List list) of 8 numbers and is traversed by
Reducible objects correspond to iterations for thread B, while the white numbers correspond to iterations for thread A. The `Reducible` object stores private copies of the minimum for each thread and is updated as each thread progresses.

Figure 2 shows the state of the reducible object after 2 elements have been processed by each thread (so far thread A got 25 and 9, while thread B got 19 and 21). The most recent element assigned to each thread is stored in variable `v` (9 and 21 respectively for threads A and B). The minimum element traversed by each thread is stored in `myMin` (9 and 19 respectively). When all iterations have been completed, the values stored in the reducible object will be reduced into one final value (achieved through the `join()` function).

Note that the supplied reductions work for any data type, as long as the data type supports the respective operation. As an example, if `MIN` is to be used to reduce a list of `MyClass` objects, then the `MyClass` class needs to have the `<` operator defined (operator overloading). Also, user-defined types need to have a copy constructor defined so that the values inside `Reducible` may be initialised correctly for each thread.

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 a function defining the reduction of two elements into one.

Consider, for example, an application that needs to perform a reduction on `Color` objects. How would multiple `Color` objects be reduced into a single `Color` value? The programmer simply defines a function as below:

```java
Color mixRGB( Color a, Color b ) {
    // User code defining reduction logic
}
```

A reduction may then be performed as usual, only this time specifying the user-defined function:

```java
Color finalCol = globalColor.join(mixRGB);
```

Reductions can generally be performed sequentially or in parallel, but the current parallel iterator implementation is performed sequentially by a single thread. This is acceptable since the number of values to reduce is limited to the thread count and the target is desktop applications. The reduction operation must be associative (an operation in which the order of evaluating the reduction makes no difference, such as addition). Also, since the reduction is executed sequentially (as shown in figure 1), the user-defined reductions do not need to be coded thread-safe.

4.2. Improved contract

Some languages, such as Java, do not support reference variables. Therefore, the previous parallel iterator interface cannot be used for such languages. Furthermore, slight code modification is required for the loop since the interface was different to the sequential iterator. To develop a Java version of the parallel iterator, the alternative approach involved using 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. It is thread-safe just like the C++ parallel iterator, but better since the programmer uses the same approach as the sequential version. Just like the C++ version, a parallel iterator is created using a convenient factory class and has the same flexible scheduling policies.

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 an 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 approach is that any thread that receives a `true` from `hasNext()` must follow up with a `next()` invocation; otherwise, that element will not be traversed by any other thread. Similarly, a thread must not invoke `next()` without first receiving `true` from a `hasNext()` invocation since an element needs to be allocated first.

A single invocation of `hasNext()` may in fact reserve more than one element for a thread, solely depending on the scheduling policy and chunk size. For example, consider a dynamic scheduling policy with chunk size 5. When thread A first gets a `true` from `hasNext()`, the first 5 elements are reserved for thread A. Multiple calls to `hasNext()` will always return `true` so long as at least one of these 5 elements are yet to be claimed by thread A (only accessible
via subsequent calls to `next()`. During this time, any call to `hasNext()` by thread B will reserve 5 other elements in a similar manner.

This interface can even be adapted to the C++ implementation. The biggest advantage of this approach is that the parallel iteration code now truly remains the same as the sequential version; in fact, the interface of the Java parallel iterator extends the standard Java `Iterator` interface. Note that although the Java `Iterator` interface specifies a `remove()` function, not all implementations support this (such implementations throw an `UnsupportedOperationException`). The parallel iterator takes this approach since defining parallel semantics for remove is at least difficult.

4.3. Parallel semantics for `break`

An important concept in iterative computation is the `break` statement. In a sequential loop, this statement means to stop executing 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 up to the developer, since there are legitimate cases for both approaches as illustrated below. Note that OpenMP does not even allow a `break` statement to be used within a work sharing construct such as `parallel for`. The simple, but powerful, concept below has been implemented for both the C++ and Java versions.

The programmer, however, needs to be aware of an important aspect before breaking from a parallel environment [10]. In the sequential version of a loop, the `break` occurs at a particular point B 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 point B after completing a subset S of the entire iteration space, all threads should stop their iterations. This is achieved by one thread invoking `globalBreak()` on the parallel iterator, which informs all threads (when they call `hasNext()` that no more iterations exist. The advantage of this approach is that each thread breaks out of its loop in a controlled manner at an iteration boundary.

```
while ( pi.hasNext() ) {
    if ( itemFound )
        pi.globalBreak();
    else
        itemFound = searchDocument( pi.next() );
}
```

Global break

It may also be the case that the programmer wishes to cancel 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, which informs all threads (when they call `hasNext()`) that no more iterations exist. The advantage of this approach is that each thread breaks out of its loop in a controlled manner at an iteration boundary.

```
while ( pi.hasNext() ) {
    if ( tooManyThreads )
        pi.globalBreak();
    else
        resize( pi.next() );
    pi.localBreak();
}
```

Without the break semantics the iterator could not be used for search, or a naive implementation would have to be developed. The benefit can be analysed statistically: if the ‘solution’ is at a random position in the iteration space, one processor on average needs to do N/2 iterations. For P processors this becomes N/2P iterations, as each processor gets N/P iterations (balanced load). The global break avoids such needless computation, which is especially important in an interactive desktop environment.

Related work

The importance of loop parallelisation and loop scheduling have been extensively studied before [1, 4, 7]. However, the here presented research is distinguished 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.

OpenMP[9] provides a 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. Intel Threading Building Blocks (TBB) [8] and QtConcurrent [14] support user-defined reductions. But since TBB requires reductions to be defined within a function object, this causes code restructuring and context change. Furthermore, multiple reductions cannot be defined within the same class definition. More recently, QtConcurrent provides a `mappedReduced` function. Again, this requires restructuring the loop code into a new function defining
work for one element; every loop iteration now results in a separate function call. QtConcurrent’s reduction function 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, which is executed sequentially. In all these tools, if the programmer wants the behaviour of our global break then they must manually implement this.

5. Implementation

5.1. Reductions

The Reducible class may essentially be viewed as a modified thread-local variable. In addition to behaving like a thread-local, it defines a join function accepting a function pointer defining an arbitrary reduction. Unlike a typical thread-local function which behaves locally to the calling thread, the join function is invoked by any thread. All internal thread-local values within the Reducible object are accessed and reduced to a single result.

The current implementation performs the reduction using only one thread (the one invoking join) as shown in figure 1. This means the reduction function does not need to be coded thread safe. For simple reductions, especially only over a few scalar variables, this is sufficient. Future work may allow parallel reductions and will be beneficial for larger reductions.

5.2. New interface contract

To explain the implementation for the new interface, an example on an inherently sequential collection (such as a linked-list) is used. Figure 3 shows the internals of a parallel iterator used to traverse a linked-list 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 3 elements remain to be processed, and these are only accessible when thread B calls next().

Note that copying is only necessary when accessing inherently sequential collections, such as a linked-list since accessing elements from the collection takes time proportional to the collection size. Also note that deep copying is not performed, but rather a pointer is retained to the elements. Only during such copying (inside a hasNext() invocation), will locking of the sequential iterator occur. No locking occurs when a thread calling hasNext() already has unprocessed elements in the private buffer. No locking occurs inside next() also, since this function accesses reserved elements from the private buffer.

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 policy chunk size). Therefore thread A invokes hasNext() to see if any more elements remain, but false will be returned.

5.3. Local and global 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, as well as release any allocated elements (to allow another thread to traverse them). The globalBreak() sets the condition for all threads. This way, all subsequent calls to hasNext() made by any thread returns false, regardless of the number of elements yet to be processed. When globalBreak() is invoked, any current iterations being executed are allowed to complete in order to safely terminate. 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 synchronise loop termination between all threads) and false will be returned as soon as currently executing iterations are completed.

6. Performance

The basic working of the parallel iterator has already been captured [6], demonstrating low overhead and scalability. Now, we focus on the performance of reductions compared to a commercial implementation of Qt
We also reinforce the notion of scheduling policies applied to the Java implementation of the parallel iterator (where programmers use the same interface as the standard sequential iterator). The benchmarks ran on a shared memory system which may be considered a typical future desktop platform running Linux. It has two quad-core Intel Xeon processors (total of eight cores) running at 1.86GHz with 8GB of RAM.

6.1. Reductions

The first benchmark, word count, recursively reads text files in a folder and counts the occurrence of each word. Such an application is typical of a search functionality on a desktop environment. The second benchmark, word permutation, also recursively reads text files in a folder. However, rather than just counting occurrences of each word, more complex permutations are performed on each word. Such behaviour 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. Both QtConcurrent’s and the parallel iterator’s reduction are performed sequentially by one thread.

Figure 4(a) shows the speedup of the word count benchmark, comparing QtConcurrent (QTC) 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 slow disk access, 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 is no support to limit thread count.

The speedup for the computationally intense word permutation program is shown in figure 4(b). 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 speedup began plateauing as more threads came into contention.

A likely contributor to QtConcurrent’s slight performance woe, especially as evidenced in figure 4(a), is the high number of sequential reductions. 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.

QtConcurrent uses all the available processors, but in some cases the programmer might want to reduce or increase the thread count. This is especially important in a desktop environment when other applications are executing, especially with disk-intensive applications such as searching. By default the parallel iterator uses all available processors, but this may be modified dynamically at runtime with a parameter.

6.2. Scheduling policies

Here we show the Java implementation and the importance of scheduling policies. The first benchmark, the Mandelbrot set, is a fractal composed of a set of points in the complex plane [15]. The application has an unbalanced workload since the computation varies from point to point (the computation of each point is independent). The sequential version computes the fractal for 300x500 points in about 47 seconds. The second benchmark produces a balanced workload by repeating a simple computation for each point in a 300x500 plane. The sequential time for this benchmark is 5 seconds for all points.

Figure 5(a) shows the speedup for the Mandelbrot benchmark using 5 different scheduling policies. The overhead witnessed is negligible in this benchmark since the paral-
parallel code executed with the same time on a single processor, regardless of the scheduling policy used. Due to the unbalanced nature of the Mandelbrot points, static block scheduling (explained in section 3) performed worst, sometimes barely achieving half the ideal speedup. However, static scheduling performed well with a small chunk size (e.g. chunk size 1 as shown), achieving over 90% the ideal speedup. Dynamic scheduling achieved around 85-90% of the ideal speedup, with a chunk size of 100 performing slightly better than chunk size of 1. However, guided scheduling performed the best, achieving an average speedup of 95% the ideal speedup.

While figure 5(a) demonstrates the important influence of different scheduling types, figure 5(b) demonstrates the importance of different chunk sizes when a dynamic scheduling scheme is used. Most notable is chunk size 1 only scaling up to 3 processors, after which it starts to lose speedup (this is attributed to thread contention while locking the sequential iterator as discussed in section 5.2). The benchmark scales to 6 processors simply by doubling the chunk size, but still the speedup is only half the ideal speedup. Due to the large number of iterations and the low computation per iteration, such an application requires a high chunk size in order to scale with increasing processor count. Using 8 processors, a speedup of 6.3 is achieved with a chunk size of 100 (reducing a 5 second application to 0.8 seconds).

These two benchmarks demonstrate the importance of having flexible means to fine-tune iteration code. With the parallel iterator, the programmer can easily change the schedule type, the chunk size and the number of threads involved. In fact, all these parameters may even be specified at runtime as simple arguments when creating the parallel iterator. Providing the programmer with this flexibility is important, since different applications have different characteristics. The programmer is given the possibility to performance tune depending on the granularity, imbalance and disk access of iterations. Most important is that iteration code remains the same from the programmer’s point of view, especially since the parallel iterator now uses the same interface as the standard sequential iterator.

7. Conclusions

This paper proposed an extended and improved parallel iterator concept, making it more applicable to parallelising object-oriented iterative computations. The addition of user-defined reductions and global break semantics allow more complex iterative situations to be handled. The interface of the parallel iterator has been modified to imitate the standard Java-style sequential iterator. Therefore, when used in combination with OpenMP, the structure of the program remains virtually unchanged. The performance of the parallel iterator’s reduction is in many cases superior, or at least comparable, to that of QtConcurrent. The results demonstrate the power of flexible scheduling policies, and the parallel iterator makes this easy for the programmer. Even controlling the thread count is important, especially for an interactive desktop application involving heavy disk usage.

References