HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

HSR – University of Applied Sciences Rapperswil

Institute for Software

Parallator
Transformation towards C++ parallelism using TBB

Master Thesis: Fall Term 2012

Martin Kempf
martin.kempf@gmail.com

Supervised by Prof. Peter Sommerlad

Version: February 21, 2013

Abstract

Intels TBB (Threading Building Blocks) library provides a high level abstraction to write parallel programs in C++. It addresses the fact that parallel hardware with increasing number of cores in processors is likely to be the main source of performance gains. With this change in common computer architecture, an application must provide enough parallel work to exploit the computational power of a multi-core CPU. Parallel programs using multiple threads with independent operations are needed. Operations in a loop are often, although not explicitly defined, a source of independent operations and provide therefore potential to introduce parallelism. With algorithm calls of TBB, loop operations can be explicitly parallelized while making use of the efficient task scheduling mechanism that uses threads for parallelism.
A manual transformation of loops to the corresponding TBB algorithm is time consuming and includes pitfalls like carelessly introduced data races. Parallator, an Eclipse plug-in is therefore developed using the C++ Development Tooling (CDT) project to replace loops with TBB calls and provide hints for possible parallelization issues. The plug-in lets the user decide a transformation to apply. This approach differs from automatic parallelization introduced by compilers and improves the situation of a rather conservative handling towards parallelism by a compiler. Parallator is based on a former project [Kes10] and uses its tree pattern matching algorithm on the program code's abstract syntax tree to find potential code for parallelism and possible parallelization issues.

Management Summary

Motivation

The introduction of parallelism into existing applications gets increasingly important. This is based on the fact that desktop computers are equipped with multi-core processors and that an increasing number of cores in processors is likely to be the main source of improved performance in computer hardware. Introducing parallelism into an existing application involves finding possible sources for parallelism and the implementation towards parallelism. This process is time consuming and error prone.

Goal

In this master thesis, this situation is addressed with the Parallator project. The project aims to assist the C++ developer using Eclipse C++ Development Tooling (CDT) in the process of introducing parallelism in an existing application. To simplify the transformation to parallelism, Intel's TBB (Threading Building Blocks) library is used that provides a high level abstraction to write parallel programs in C++. Beside other features it provides algorithms to parallelize loops. Loops often show similar independent computation over iterations and are therefore a source for parallelism.
The chosen approach enables the user to choose whether a transformation should be applied or not and overcome the issue of too conservatively applied optimizations by a compiler. Possible issues, like a potential data race that let to undefined program behavior should be addressed with a warning.

Results

The resulting CDT plug-in detects iterator- and index-based loops as well as STL *for_each* calls that can be transformed to the equivalent *parallel_for* / *parallel_for_each* counterpart in TBB. A warning is provided if the transformation is not safe according to potential conflicting access of a shared variable, but the transformation can be proceeded anyway if the users allows to do so. The implementation also features a natural support for the conversion of the loop's body to C++11 lambdas with capture evaluation, STL or Boost bind functors.
A performance measurement of an implemented transformation has shown an improvement of about 50% compared to its serial equivalent. This shows the potential of Parallator.

Outlook

The application of Parallator in a real life project must be further evaluated to show its benefit. The project evaluated also transformations that have not been implemented. An

interesting project would be the transformation of loops and STL algorithm calls to its parallelized version implemented in Thrust[1].

[1]http://thrust.github.com/

Contents

1. Introduction

"The free lunch is over" [Sut05]. The free lunch of having the performance of software increased, just by the increased clock rate of the processor. Without any changes in the source code, the application runs faster and the user can profit of faster response times. The end of this era in desktop applications is introduced by the shift in computer architecture from uniprocessor systems to multiprocessing systems. This change started in the last decade and it is expected that the number of processing units even in desktop computers will increase in the future [MRR12]. While in uniprocessor system all tasks are executed by one CPU in sequential order, the tasks in a multiprocessing system can be executed in parallel by several CPUs. But this also requires the application to provide work that can be executed in parallel.
The programming languages mostly used in desktop applications are designed for serial execution. They do not provide simple constructs to express parallelism. Threads are a possibility but not a simple one and can be seen as a mean to parallelism. An example for the serial intention in programming languages is a *for* loop. It enforces an execution ordering that might not be needed as the semantics remain the same in a possible parallel execution of the loop iterations [MRR12]. This lack of expressing parallelism forced the development of *parallel programming model* that provide simpler possibilities to express parallel executions.
Intels TBB (Threading Building Blocks) [Cor12b] library is a parallel programming model that provides a high level abstraction to write parallel programs in C++. The library features an efficient implementation of several *algorithm strategy patterns*. *Algorithm strategy patterns* can be seen as a guide to structure an algorithm, in our case a parallel algorithm. These *parallel patterns* mostly have a serial equivalent [MRR12].
The change to a new era of computer architecture demands also tooling support [SL05]. This project aims to support a programmer in introducing parallelism in an existing application by *finding* serial equivalents of parallel patterns and the *transformation* to them. As implementation of the parallel patterns, TBB is used. This results in an application that is scalable: The performance of the application increases with the number of processing units available by the system.

1.1. What is Parallelism

Parallelism is something natural for humans. It appears daily. An example is the parallel checkout in a grocery store. Each customer buys goods and does not share them with other customers. This enables the cashiers do work independently of other customers. The parallel checkout in a grocery store can be done if multiple cashiers are available and the goods are not shared between the customers. With this real life example in mind this section explains parallelism in computer hardware and software and also clarifies the usage of terms in this work. The terms explained are related to parallel programs using *shared memory*, as this project is focused on TBB which follows this parallel programming paradigm. Other parallel programming paradigms such as *message passing* or *data flow programming* are not covered here.

1.1.1. Parallelism vs. Concurrency

Concurrency is when at least two tasks can run in overlapping time periods. It does not necessarily mean that tasks are executed simultaneously. Concurrency just expresses the condition of having two tasks that do not need to end before the next one can start. For *parallelism*, it is required to have the task run genuinely parallel. Which is only possible if at least two processing units are available. This gets clear by looking at the multitasking capability of operating systems to run multiple applications at the same time. On a computer with only one processor, with a single processing unit or core, the execution of the multiple tasks can not be genuinely parallel. Through task switching the tasks are scheduled on the processor in an alternating manner. Because the tasks can be switched several times per second we have the illusion of real parallelism. If we consider a multiprocessor system or a multi-core processor where the processing units are on the same chip, the tasks can be executed real parallel. We refer to this as hardware concurrency. The execution of multiple independent tasks on a system providing hardware concurrency is therefore faster. Not only because the tasks can run in parallel, also the time used for context switching is saved. The switch of a tasks needs a backup of the stack and also processor registers of the currently running task, to be restored when the task is resumed for execution.
Possible parallelism relates to the number of hardware threads a processor provides. It is the measure of how many tasks can be genuinely run in parallel. With SMT (Simultaneous Multi-threading) it is also possible to have more than one hardware thread on a single core. For Intel processors this technique is better known as hyper-threading.

1.1.2. Explicit vs. Implicit Parallelism

The term *implicit parallelism* describes the condition if parts of the program are automatically executed in parallel. Automatic parallelization is done at the instruction level. This is called *instruction level parallelism* (ILP). In that, a processor evaluates available parallelism, e.g. nearby instructions that do not depend on each other, and executes these instructions at the same time. In *explicit parallelism* the parts that can run in parallel are expressed explicitly. The means of expressing parallelism explicitly is further explained in Section 1.2.3 on page 4.

1.1.3. Performance in Parallelism

The performance of a serial program can be measured by the amount of computational work. Strive for less computational work improves performance. This is not entirely valid for parallel programs. It has two reasons. First, the bottleneck might not be in computation. A possible bottleneck can also be memory access or communication between processors. And second, the algorithm is not scalable: the execution is not faster although more cores are available. The reason might be in the algorithm's *span*. This is the time it takes to perform the longest chain of tasks that must be performed sequentially. It is also known as the critical path [MRR12]. The span and communication issue can be seen in Figure 1.1 on the next page. The sequential execution (a) of tasks A, B, C, D might be executed in parallel (b) and (c). As soon as the task are dependent (b) or they have not equal execution time (c) the span is increased and limits scalability.

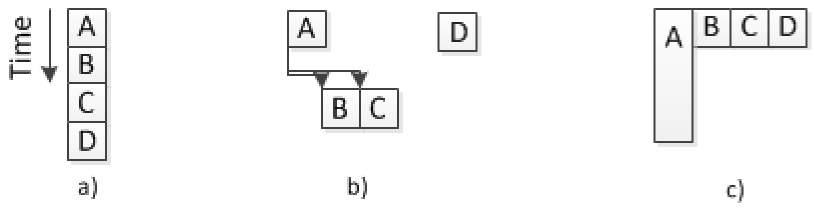


Figure 1.1.: Seriall (a) and parallel execution of tasks with dependency (b) and unequal computation time between tasks (c). Both increase the span and limit scalability.

1.2. Parallel Programming

Parallel programming is needed to develop scalable applications. This involves to express parallelism explicitly as explained in this section. The result is an application runnable on new hardware with increased hardware concurrency also with an increase in performance.

1.2.1. The Need For Explicit Parallelism

Implicit parallelism as explained in Section 1.1.2 on the preceding page can be introduced automatically. The compiler or the processor must detect the implicit parallelism in order to exploit it. There are many cases where the compiler or the processor can not detect the implicit parallelism [MRR12]. A loop is a serial control construct and may hide parallelism because the ordering does not need to be maintained. Loops over-specify the ordering. Listing 1.1 shows an example. It sums the elements of an array. Each loop iteration depends on the prior iteration, thus the iteration cannot be done in parallel. At least in some compilers dependency analysis. But as it is shown later that this kind of dependency in loops can be explicitly parallelized using the reduce pattern (Section 1.4.5 on page 15).

```
double summe(int n, double a[n]) {
  double mysum = 0;
  int i;
  for (i = 0; i < n< ++i)
    mysum += a[i];
  return mysum
}
```

Listing 1.1: An ordered sum with dependency in loop [MRR12].

1.2.2. Strategies for Parallelism

For a scalable application, the strategy is to find *scalable parallelism*. The best strategy is *data parallelism*. Data parallelism means parallelism where the computational work increases with the problem size. Often the problem is divided in smaller subproblems and the number of subproblems increases with the problem size. Solving a problem by the division of a problem into a number of uniform parts represent regular parallelism.

Sometimes also called *embarrassing parallelism*. In irregular parallelism, the parallelism is not obvious because the parallelization enforces different operations to be applied in parallel. Both, regular and irregular parallelism are a kind of data parallelism and therefore scalable. Two example show that. An example for fine-grained data parallelism is shown in Listing 1.2 where a matrix is multiplied by 2.

```
double A[100][100]
for (int i = 0; i < 100; i++) {
  for (int j = 0; j < 100; j++) {
    A[i][j] = 2 * A[i][j];
  }
}
```

Listing 1.2: Fine-grained data parallelism with independent loop operations.

100×100 operations are executed independently in parallel. The separation into subproblems requires in this example more attention as there is much more potential of concurrency than hardware can provide. It is therefore also more practical to partition the matrix in blocks of n×m and execute the operation on these blocks concurrently. An example for irregular parallelism and more coarse-grained example is the parallel compilation of different files. The tasks are more complex and the operations in compiling are different as the content of the files differs. But it is still scalable as the work increases with additional files.
A contrast to scalable parallelism is *functional decomposition*. In functional decomposition, functions that can run independently are executed in parallel. If the functions take the same amount of time and the overhead in parallelization is rare, it increases the performance. But it does not scale [MRR12]. Functional decomposition is also used to maintain the responsiveness of the user interface when a computation intensive action is started. Each functionality runs in a different thread, the user interface handling and the computational intensive action.
Sometimes the term *task parallelism* is used as a parallelism strategy. But its meaning often conflicts with functional decomposition that does not scale or with scalable parallelism where the tasks differ in control flow. The distinguished parallelism in this work are *regular parallelism* and *irregular parallelism* as explained in this section.
For the best performance, the different parallelism strategies and also serial strategies are combined. An example can be separated in different parallel phases. The parallel phases are executed serially. This makes clear that serial performance of code can not be neglected for overall performance [MRR12].

1.2.3. Mechanism for Parallelism

Most modern processors provide two possibilities to express explicit parallelism in software for parallel execution [MRR12]:

- **Thread parallelism:** Hardware threads are used for the execution of a separate flow of instructions. The flow of operations in software threads can be different and hardware threads can be used to execute regular as well as irregular parallelism.

- **Vectorization:** This is the mechanism to execute the same flow of instruction on multiple data elements. It can therefore naturally be used for regular parallelism but also with limitation for irregular parallelism. To support vectorization a processor provides a set of *vector instructions* that operate on *vector registers*. The specific vector instruction is then applied in parallel to each element that is stored in one vector register.

Potential for vectorization can be detected by a compiler and transformed appropriately [MRR12]. This is called *auto vectorization*. But vectorization is not detected in every case. Additionally the vectorization capabilities regarding to register sizes and available instructions are different. It is therefore also required to express vectorization explicitly.

1.2.4. Key Features For Performance

While we focused in Section 1.1.3 on page 2 on the performance that parallelism can provide and also limitations, this section explains the two key features to reach performance in modern computer hardware. The two key features are *data locality* and *availability of parallel operations*. The available parallelism can be also defined as *parallel slack* and tasked-based programming is a mean for an efficient execution of available parallelism.

Data Locality

Data locality describes the reuse of data from nearby locations in time or space. It is a crucial issue for a good use of memory bandwidth and efficient use of cache. For an algorithm it is therefore important to have a good data locality. This influences how the problem is subdivided and in which order the different tasks are executed. Section 1.4.2 on page 10 will show an implementation that is aware of data locality.

Parallel Slack and Task-Based Programming

We have already mentioned the problem of having more concurrency than a hardware can provide. The *potential parallelism* is in this cases higher than the *actual parallelism*. The amount of "extra" parallelism available is called *parallel slack*. This is nothing bad and enables the scheduler more flexibility to exploit machine resources. But it also needs another abstraction above threads.
If all the possible independent operations are executed in a different thread, the overhead in scheduling the threads swamps the performance gain of parallel execution. On the other side, the performance is also influenced if more threads could have been used. The best performance can be achieved if the number of running threads is equal to the available hardware threads. Task-based programming prevents time consuming thread switching. The term task defines a light weight object that performs its computation without interruption and returns when his computation is finished. A task scheduler maintains a pool of threads and is responsible for spawning the tasks across the available threads in a way that the computational work is balanced equally on the threads. Thinking in tasks, we can focus on the dependency of the task and leave the efficient scheduling to the scheduler.

1.2.5. Pitfalls

The shared memory communication brings also the problem of concurrent access to the same memory location which can result in a race condition. Race conditions occur whenever two or more different threads access the same memory location and one access is a write access. Consider a variable that holds the number of currently available pieces of a product. Multiple people / threads can buy the product concurrently. The activity is as shown in Listing 1.3.

```
 1 bool buyProduct(productId) {
 2    int amount = stock.get(productId);
 3    if (amount > 0) {
 4       amount = amount - 1;
 5       stock.put(productId, amount);
 6       return true;
 7    } else {
 8       return false;
 9    }
10 }
```

Listing 1.3: Pseudo code of a race condition. The variable `amount` as well as `stock` are shared between the threads that execute the method `buyProduct()` concurrently.

Assuming two persons concurrently want to buy the same product and there is only one piece available. If the function `buyProduct()` is executed in parallel, a product can be bought although it is not available anymore. This is the case if line 2 is executed simultaneously with the result that both threads read that there is one piece left. If the shared variables are only accessed in a read only manner, there is no potential for a race condition. To prevent from race conditions the access on the shared data must be synchronized.
Another problem that arises in shared memory parallel programming and are introduced with synchronization of shared data are *deadlocks*. They occur if two threads wait on each other because each thread restricts the other in accessing shared data. *Load balancing* is another issue and occurs if the work of the threads are not of equal size. A solution on this is task-based programming as introduced in Section 1.2.4 on the previous page with an efficient implementation as explained in Section 1.4.2 on page 10.

1.3. Selected C++11 Features

In this project we make use of some features that are introduced in the new C++ standard, C++11 [Bec12]. The used features are explained in this section. We will also have a look on the memory model introduced with C++11 regarding the implication of this to the usage of containers.

1.3.1. Lambda Expression

A *lambda expression* is just a simpler way to create a *functor* (function object) that could otherwise be written manually. We show this in a comparison of a functor that is equal to a lambda expression. An algorithm that takes a functor as argument is `std::equal(first1, last1, first2, binaryPred)`. It compares values of two sequences and returns true if for each value in the sequence the binary predicate `binaryPred` holds. The binary predicate as explicit functor `CompareIgnoreCaseAndChar` in Listing 1.4 on the next page evaluates to `true` if the characters provided to the overloaded function call operator equals-ignore-case or if the character is equal to the value set in the constructor call (line 13) and therefore ignored in the comparison.

Essentially the functor has three parts:

- A field for storing the character that is ignored in the comparison.
- A constructor that captures the character to ignore.
- A definition of what to execute if the functor is used.

```cpp
class CompareIgnoreCaseAndChar {
  const char ignore;
public:
  CompareIgnoreCaseAndChar(const char ignore) : ignore(ignore) {}

  bool operator()(const char c1, const char c2) {
    return (c1==ignore) || (tolower(c1) == tolower(c2));
  }
};

bool equals_ignore_case_and_char(std::string s1, std::string s2, const char c1)
    {
  return std::equal(s1.begin(), s1.end(),
          s2.begin(), CompareIgnoreCaseAndChar(c1));
}
```

Listing 1.4: Using a manually written functor `CompareIgnoreCaseAndChar`.

Writing the same functor as lambda expression saves a lot of work as it can be seen in Listing 1.5. It shows the equivalent function of Listing 1.4. The lambda expression can be seen from line 3 to line 5.

```cpp
1 bool equals_ignore_case_and_char_lambda(std::string s1, std::string s2, const
      char ignore) {
2   return std::equal(s1.begin(), s1.end(), s2.begin(),
3       [=] (const char c1, const char c2) {
4         return (c1==ignore) || (tolower(c1) == tolower(c2));
5       }
6   );
7 }
```

Listing 1.5: Using lambda expression the code from Listing 1.4 can be written more concisely.

The syntax of a lambda expression is shown in Figure 1.2 on the following page with the numbered sections explained as follows:

1. This part defines how to capture local variables and make them available in section 5. Global variables do not need to be captured as they are available on default. Possibilities in capturing are: [=]-capture by value, [&]-capture by reference, [ ]-nothing to capture. They influence how the fields in the "generated" functor are declared and passed in the constructor call. It is also possible to define a specific capture mode for each variable to capture. E.g., a capture definition of [=, &x, &y] defines a capture by value for all local variables except for `x` and `y` that are captured by reference. Additionally, a lambda expression inside a member function captures all the member fields by reference independent of the defined capture mode.

2. The parameter list defines the parameter that can be passed in a call of the lambda expression. These parameters become the parameters to the overloaded function call operator.

3. The modifier mutable can be used to declare a non constant function call operator. If omitted, the function call operator is constant in every case. A variable captured by value can not be modified in the lambda body (5).

4. Defines the type of the return value. It is optional as it can be deduced from a possible return statement int the lambda body (5). If no return statement exists in section 5 the return type is evaluated to void.

5. The body of the lambda expression. It becomes the body of the function call operator.

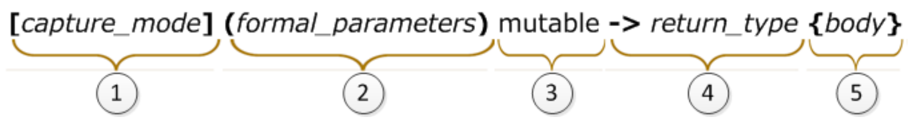

[capture_mode] (formal_parameters) mutable -> return_type {body}
1
2
3
4
5

Figure 1.2.: Syntax of a lambda expression.

1.3.2. Defined Multi-Threaded Behavior with STL Container

Prior to C++11, the language specification did not state anything about thread safety of containers. Further, threading was not a part of the language specification. Additional libraries needed to be used for threading support in C++. This has changed with the new standard. A memory model is now part of the specification. It exactly defines a data race and assures defined behavior in case the program is free of a data race. Additionally, implementation requirements for containers are given. The standard defines the following in section 23.2.2 Container data races [Bec12]:

> ...implementations are required to avoid data races when the contents of the contained object in different elements in the same sequence, excepting `vector<bool>`, are modified concurrently.

Further the standard defines the following data structures from the STL (Standard Template Library) to be sequencing container satisfying the statement above: `vector`, `forward_list`, `list`, and `deque`. Additionally also `array` but with limited sequence operations because it has a fixed number of elements. Referring back to the quote, these containers avoid data races in concurrent modification of different elements. For example a `vector<int> x` with a size greater than one, `x[1] = 5` and `x[2] = 10` can be executed concurrently without a race. The exception states a `vector<bool> y` with a potential data race in case of concurrent execution of `y[0] = true` and `y[1] = true`.
Operations that change the container object itself, e.g. insertion of an element, or a conflicting access to the container by different threads are a source of a data race and need separate synchronization mechanism. A conflicting access is defined as two operations on the same container element while at least one is a write.

1.4. Threading Building Blocks

Intel's TBB (Threading Building Blocks) library provides a set of "threading building block" for a high-level, task-based parallelism to abstract from platform details and threading mechanisms for scalability and performance [Cor12b]. A developer does not need to care about thread management. This reduces the complexity and the number of code lines otherwise present in multi-threaded applications. It provides ability to express data parallelism and also an efficient implementation of the parallelization. It makes use of the hardware supported threading but does not provide ability to enforce vectorization. For vectorization TBB relies on auto vectorization introduced in the compiler [MRR12]. Starting with an introduction example followed by a detailed description of the different components involved in the example, this section gives an overview on the possibilities TBB provides and are used in this work.

1.4.1. Introducing Example

In Listing 1.6 we apply an arbitrary function `Foo` to every array element by iteration over all array elements. This is done in serial and defined order.

```
 1 void SerialApplyFoo(float a[], size_t n ) {
 2   for(size_t i = 0; i != n; ++i) {
 3     Foo(a[i]);
 4   }
 5 }
```

Listing 1.6: Serial application of function `Foo` on every array element [AK12].

The method `SerialApplyFoo()` provides typical potential for data parallelism. The operation `Foo` in the loop are independent and can be parallelized. This is only valuable if the operation is time consuming or the array is big enough to not swamp parallelism with the overhead. If so, complicated considerations as how many threads execute how many iterations, arises too. Listing 1.7 shows the parallel version of Listing 1.6.

```
 1 ApplyFoo { //Loop body as function object
 2   float *const my_a;
 3 public:
 4   ApplyFoo( float *a ) : my_a(a) {}
 5   void operator()( const blocked_range<size_t>& range ) const {
 6     float *a = my_a;
 7     for( size_t i=range.begin(); i != range.end(); ++i )
 8       Foo(a[i]); //loop body
 9   }
10 };
11
12 void ParallelApplyFoo( float a[], size_t n ) {
13   parallel_for(blocked_range<size_t>( 0, n ),
14         ApplyFoo(a),
15         auto_partitioner() ); //partitioning hint
16 }
```

Listing 1.7: Serial application of function `Foo` on every array element [AK12].

The loop body is extracted into a function object (line 1-10). Using the `parallel_for` parallel algorithm (line 13-15) the functor object is applied on subranges. The subranges are the result of a splitting mechanism of the whole iteration space defined as `blocked_range` object (line 13). With a partitioning hint (line 15) the size of the subrange is influenced and therefore also the number of parallel tasks spawned, since each subrange is executed by one task. The scheduling of the tasks is managed by the task scheduler (Section 1.4.2) of TBB. The following sections explain the different parameters in more detail.

1.4.2. Task Scheduling

The task scheduler is the component that underlies the different parallel algorithms. It is the key component for load balanced execution of the tasks and makes TBB useful for nested parallelism.

Task Graph and Work Stealing

The task scheduling algorithm in TBB is a work stealing algorithm. In that, the scheduler maintains a pool of worker threads where each thread has a ready pool for tasks that can be executed (Figure 1.3). The ready pool is modeled as a double-ended queue, a deque. Whenever a new task is spawned, it is added at the front of the deque. If the thread is ready, the next task is popped from the front of its deque, hence running the *youngest* task that is created before by this thread. If the thread is ready and its deque is empty (see worker 1 in Figure 1.3), it steals the task from the back of a randomly chosen other deque maintained by another worker thread (victim). This steals the *oldest* task created by another thread.

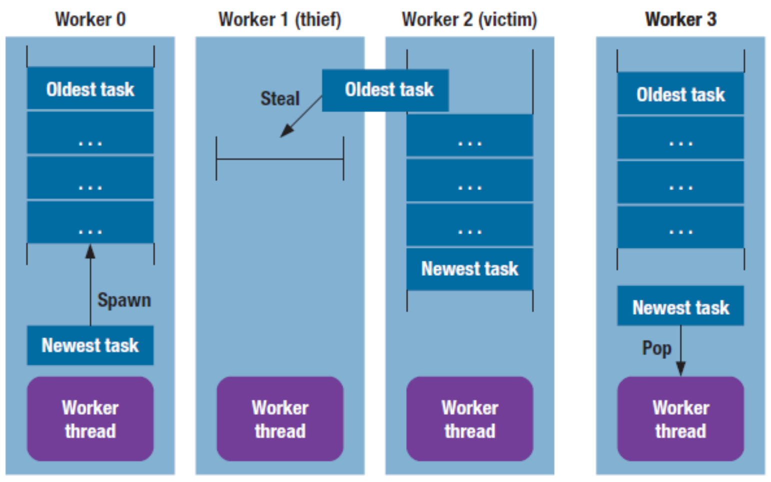


Figure 1.3.: Obtaining tasks from the local double-ended queues (deques) maintained by the worker threads. This strategy unfolds recursively generated task trees in a depth-first manner, minimizing memory use. [KV11]

This algorithm works best if the task graph is a tree. The task scheduler evaluates this task graph in a preferably most efficient way using the work stealing algorithm. A task tree is created by tasks (executed by threads) that split its work to other tasks and wait until the child tasks finished there work and join back[1]. A possible task tree is shown

[1]This is the default behavior but also more complex dependencies than the simple parent-child relationship can be specified.

in Figure 1.4 resulted from a recursively split range of data. Applying the work stealing algorithm on this task tree results in depth-first execution order on the same thread to minimize memory use whereas stolen work is executed in breadth-first order to occupy the worker threads and exploit parallelism. As the stolen oldest task is high in the task tree it represents large chunk of work and is further again executed in depth-first manner until it needs to steal more work.

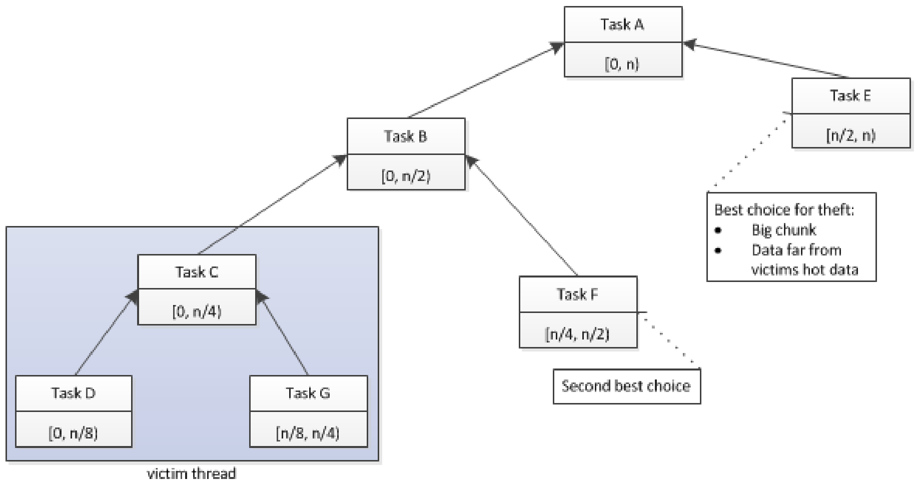


Figure 1.4.: Recursive subdivision of a range to generate tasks. Each box represents a task that will perform computation on a subrange. The leaves represent tasks that haven't yet executed and the internal nodes represent tasks that have executed and chose to subdivide their range.

Parallelism is exploited by stolen work. This is only possible if a worker threads has no work. Therefore there is no guarantee that potentially parallel tasks actually execute in parallel.

Instantiation of Task Scheduler

The task scheduler is initialized using the `task_scheduler_init` class. This initialization is optional. A task scheduler is created automatically the first time a thread uses task scheduling services. The task scheduler is destroyed again when the last thread that uses the scheduler exits. The number of threads used by the task scheduler and the stack size of the worker threads can be set in a manual initialization. But it is best to leave the decision of how many threads to use to the task scheduler, as the hardware threads is a global shared resource. Also other threads running on the system want to use the available hardware threads. For the programmer there is no way to know how many threads would be optimal considering also other threads running on the system.

1.4.3. Iteration Space / Range Concept

To divide the work into chunks that are equaly spread on the tasks, a range concept is used. A range object provides functions to recursively split the iteration space into smaller chunks. The size of the chunk is evaluated by the scheduling system. It uses the range object to split. Splitting is stopped if the serial execution is faster than to split further. The chunks can then be executed by tasks concurrently. Range objects can also be seen as a wrapper to indices. While a loop iterates serially over the indices and accesses the data container, ranges are executed in parallel by tasks whereas each task iterates serially over the wrapped indices of a range. Custom range objects can be declared with specific criteria on how and when to split. TBB also provides common used range objects that implement the range concept. They are explained in the following sections. Figure 1.4 on the preceding page shows a recursively split range that generates tasks. It also outlines the relationship of the range concept with the task scheduling regarding to data locality.

blocked_range<Value>

A `blocked_range<Value>` represents a half-open range [i,j) that can be recursively split. It is constructed with the start range value `i` and the end range value `j`. An optional third parameter defines the grain size which specifies the degree of splitting. In `blocked_range`, the grain size specifies the biggest range considered indivisible. On default, grain size is set to 1. The requirements of the `Value` template parameter are shown in Table 1.1. The template argument type `D` references a type that is implicitly convertible to `std::size_t`.

Pseudo signature	Semantics
`Value::Value( const Value& )`	Copy constructor.
`Value:: Value()`	Destructor.
`void operator=( const Value& )`	Assignment.
`bool operator<( const Value& i, const Value& j )`	Value i precedes value j.
`D operator-( const Value& i, const Value& j )`	Number of values in range [i,j).
`Value operator+( const Value& i, D k )`	kth value after i.

Table 1.1.: Value requirement for `blocked_range` [Cor12c]

.

Examples that model the `Value` concept are integral types, pointers, and STL random-access iterators whose difference can be implicitly converted to `std::size_t` [Cor12c]. The application of the `blocked_range<Value>` can be seen in the introducing parallel example of Listing 1.7 on page 9 where the chunk is serially processed with access to subsumed indices of the range through `begin()` for the start index and `end()` for the end index. The returned values by these functions meet the `Value` requirement.

blocked_range2d<RowValue,ColValue>

A `blocked_range2d<RowValue,ColValue>` represents a half-open two dimensional range [i0,j0)×[i1,j1). The grain size can be specified for each axis of the range. For `RowValue` and `ColValue` the same requirements must be satisfied as stated in table 1.1. A `blocked_range2d` perfectly fits for a parallel matrix multiplication where each input matrices can be split along one dimension to perfectly split the data needed for calculating an entry in the resulting matrix as Figure 1.5 on the next page shows.

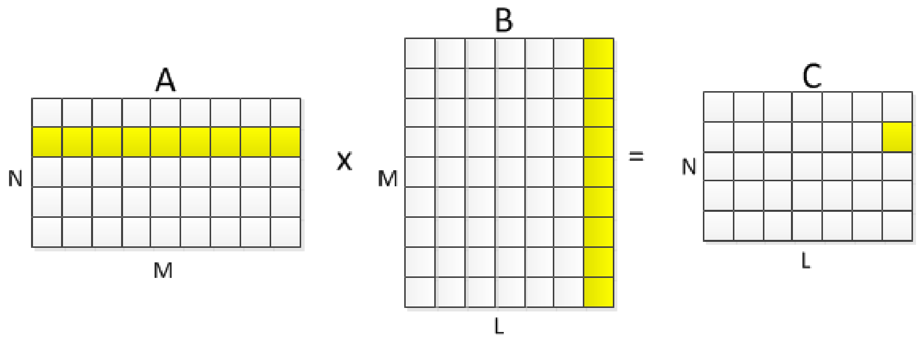


Figure 1.5.: Data depencency in matrix mutliplication.

By mapping a number of horizontal strips of matrix A and a number of vertical strips of matrix B to one task, the calculation can be done in parallel where each task calculates the resulting values in matrix C. Listing 1.8 implements the algorithm and shows the application of `blocked_range2d` with the corresponding access to the indices using the rows dimension for matrix A and the column dimension for matrix B.

```
 1 const size_t L = 150;
 2 const size_t M = 225;
 3 const size_t N = 300;
 4
 5 class MatrixMultiplyBody2D {
 6   float (*my_a)[L];
 7   float (*my_b)[N];
 8   float (*my_c)[N];
 9 public:
10   void operator()(const tbb::blocked_range2d<size_t>& r) const {
11     float (*a)[L] = my_a;
12     float (*b)[N] = my_b;
13     float (*c)[N] = my_c;
14     for (size_t i = r.rows().begin(); i != r.rows().end(); ++i) {
15       for (size_t j = r.cols().begin(); j != r.cols().end(); ++j) {
16         float sum = 0;
17         for (size_t k = 0; k < L; ++k)
18           sum += a[i][k] * b[k][j];
19         c[i][j] = sum;
20       }
21     }
22   }
23   MatrixMultiplyBody2D(float c[M][N], float a[M][L], float b[L][N]) :
24       my_a(a), my_b(b), my_c(c) {
25   }
26 };
27
28 void ParallelMatrixMultiply(float c[M][N], float a[M][L], float b[L][N]) {
29   parallel_for(tbb::blocked_range2d<size_t>(0, M, 16, 0, N, 32),
30       MatrixMultiplyBody2D(c, a, b), tbb::simple_partitioner());
31 }
```

Listing 1.8: Parallel matrix multiplication using `blocked_range2d` [Cor12c].

The `parallel_for` on line 29 recursively splits the `blocked_range2d` until the pieces are no larger than 16×32. With reference to Figure 1.5 a task calculates the results from maximum 16 rows of matrix A and 32 columns of matrix B.

blocked_range3d<PageValue,RowValue,ColValue>

A `blocked_range3d<PageValue,RowValue,ColValue>` is the three-dimensional extension of `blocked_range2d` . A typical usage is the parallelization of operation with three dimensional data structures.

1.4.4. Controlling Chunking

The grain size is not the only parameter that affects the chunk size. The other parameter is the partitioner as seen in the introducing example in Listing 1.7 on page 9 where an `auto_partitioner` is used. Table 1.2 shows the available partitioner and its affect on the chunk size together with the grain size.

Partitioner	**Description**	**When used with blocked_range(i,j,g)**
simple_partitioner	Chunksize bounded by grain size.	$\lceil g/2 \rceil \leq$ chunksize $\leq$ g
auto_partitioner	Automatic chunk size.	$\lceil g/2 \rceil \leq$ chunksize
affinity_partitioner	Automatic chunk size and cache affinity.	

Table 1.2.: Overview and description of the different partitioner.

Automatic Chunking

Using parallel algorithms without specifying a partitioner, the `auto_partitioner` is used. It determines the chunk size automatically. Initially the number of subranges is proportional to the number of processing threads, and further additional subranges are created only when necessary to balance load, hence if a thread gets idle and is ready to work on another chunk.

Upper Bound of Chunksize

With a `simple_partitioner` the range is from the very beginning divided into subranges until the subrange is not further dividable. Since this limit is grain size dependent, the `simple_partitioner` provides an upper bound for the chunk size.

Optimize for Cache Affinity

An optimization on cache usage provides the `affinity_partitioner`. It uses the automatic chunking algorithm as `auto_partitioner` but advices the parallel algorithm to assign iterations to the same processors as in an earlier execution of the same loop (or another loop) with the same `affinity_partitioner`.

Ideal Chunking

Finding the ideal parameters for loop template using ranges and partitioners is a difficult task. A too small grain size may cause scheduling overhead in execution and swamps the speedup gained from parallelism. A too large grain size may reduce parallelism unnecessarily. For example, a chosen grain size for a range that is the half of the entire iteration space splits only once and the potential parallelism is two. Generally, to relay on `auto_partitioner` provides a good solution.

1.4.5. Parallel Algorithms

TBB provides several parallel algorithms that help to parallelize common operations. In the following, some algorithms are described in detail whereby the `parallel_for`, `parallel_reduce` and `parallel_scan` are loop templates that make use of the range concept and partitioner for chunking, the `parallel_for_each` is an algorithm for streams.

parallel_for Template

The `parallel_for` function template performs parallel iteration over a range of values. There are two versions for `parallel_for`, a compact notation and a more general version that uses a range explicitly. The syntax for the compact notation is shown in Listing 1.9.

```
template<typename Index, typename Func>
Func parallel_for( Index first, Index last, Index step, const Func& f [,
    task_group_context& group] );
```

Listing 1.9: Syntax of `parallel_for` for compact notation.

An example usage of this compact notation is shown in Listing 1.10 which represents another parallel version of the introduction example. The step is explicitly specified, if omitted it is set to one. The partitioning strategy is always `auto_partitioner`. It supports only unidimensional iteration spaces and the `Index` template parameter requires a type convertible to integers. Iterators can therefore not be used in this compact notation.

```
void ParallelApplyFoo(float* a, size_t n) {
  tbb::parallel_for(size_t(0), n, size_t(1), [=](size_t i) {
    Foo(a[i]);
  });
}
```

Listing 1.10: Example of using `parallel_for` with compact notation.

The syntax of the more general version of `parallel_for` is shown in Listing 1.11

```
template<typename Range, typename Body>
void parallel_for( const Range& range, const Body& body, [, partitioner[,
    task_group_context& group]] );
```

Listing 1.11: Syntax of `parallel_for` for general usage.

The requirements of the `Body` concept is a unary functor with an overloaded constant call operator that takes a range object as argument. For every constructed subrange the body object is copied and applied on the range. For an example on this general version we refer to the introducing example in Listing 1.7 on page 9.
Both versions of `parallel_for` and all upcoming described algorithms are featuring an optional parameter for a specified `task_group_context`. A `task_group_context` represents a group of tasks that can be canceled in order to abort the computation. The `task_group_context` objects form a forest of trees where the root of each tree is a `task_group_context` constructed as isolated. Isolated is one kind of a possible relation to the parent, in fact isolated means the group has no parents. Canceling a `task_group_context` causes the entire subtree rooted at it to be canceled. Therefore also a cancellation of a `task_group_context` does not affect other `task_group_context` trees created as isolated. The default parent relation behavior is bound. A root task that is associated with a bound `task_group_context` becomes, in contrast to isolated, the child of the innermost running task's group. A child task automatically

joins its parent task's group. The parameter to the parallel algorithms specifies to which group context the created tasks belong and affects therefore the behavior in cancellation.

parallel_reduce Template

With `parallel_reduce` a reduction can be calculated over a range. Listing 1.12 shows two possible syntax for the reduction.

```
template<typename Range, typename Body>
void parallel_reduce(const Range& range, const Body& body [, partitioner[,
    task_group_context& group]] );

template<typename Range, typename Value, typename Func, typename Reduction>
Value parallel_reduce(const Range& range, const Value& identity, const Func&
    func, const Reduction& reduction, [, partitioner[, task_group_context& group
    ]] );
```

Listing 1.12: Syntax of `parallel_reduce`.

Referring to the first syntax of a `parallel_reduce` call, the body argument needs the overloaded `operator()` as in `parallel_for`. Additionally, a function `join(const Body& b)` for the reduction of the values calculated in concurrent tasks and a splitting constructor is required. Listing 1.13 shows an implementation of a parallelized `std::count`. It counts the values equal to 42 in an array.

```
 1 class Count {
 2 public:
 3   int my_count;
 4   void operator()(const tbb::blocked_range<const int*>& r) {
 5     my_count += std::count(r.begin(), r.end(), 42);
 6   }
 7   Count(Count& x, tbb::split) : my_count(0) {
 8   }
 9   void join(const Count& y) {
10     my_count += y.my_count;
11   }
12   Count() : my_count(0) {
13   }
14 };
15
16 int parallelCount(const int data[], size_t n) {
17   Count counter;
18   tbb::parallel_reduce(tbb::blocked_range <const int*> (data, data + n), counter
       );
19   return counter.my_count;
20 }
```

Listing 1.13: Parallelized `std::count` using `tbb::parallel_reduce`. Counts the occurrence of 42 in the `data` array.

Consider the special splitting constructor on line 7. This constructor is used to split the body to apply on the recursively split ranges as described in Section 1.4.3 on page 11. A splitting of the reduction body is only done if a worker thread is ready to steal a subrange. Otherwise, a subrange or several subranges are processed by the same reduction object. Hence a body is not split every time a range is split, but the converse is necessarily true. This must especially be regarded on line 5 to not discard earlier accumulated counts. The already mentioned `join()` function on line 9 is called to join the split reduction object back to its origin and accumulate the calculated values.

The second syntax outlined in Listing 1.12 on the preceding page makes the use of lambda expressions in conjunction with `parallel_reduce` possible. The previously described reduction body is separated into a functor applied on the range and another functor that does the reduction. The additional `identity` parameter represents the identity of the reduction operator. Listing 1.14 shows the usage of the lambda friendly syntax also simplified by the usage of numeric operation provided by the STL.

```cpp
int parallelCount(const int data[], size_t n) {
  return tbb::parallel_reduce(tbb::blocked_range<const int*>(data, data + n),
      0, [=] (tbb::blocked_range <const int*> r, int i) -> int {
        return i += std::count(r.begin(), r.end(), 42);
      }, std::plus<int >());
}
```

Listing 1.14: Parallelized `std::count` using `tbb::parallel_reduce` with lambda friendly syntax. Counts the occurrence of 42 in the `data` array.

This version of `parallel_reduce` directly returns the result of the reduction.

parallel_scan Template

The `parallel_scan` calculates a parallel prefix also known as parallel scan. It allows to parallelize calculations over a sequence of data that appear to have inherently serial dependencies. A serial implementation of a parallel prefix, the calculation of a the running sum, can be seen in Listing 1.15

```cpp
1 int runningSum(int y[], const int x[], int n) {
2   int temp = 0;
3   for( int i = 1; i <= n; ++i ) {
4     temp = temp + x[i];
5     y[i] = temp;
6   }
7   return temp;
8 }
```

Listing 1.15: Serial implementation of parallel prefix that calculates the running sum [Cor12c].

From line 4 we realize the serial dependency as the current sum is dependent on the previous calculated sum. But for faster calculation, here also, the range must be split in subranges to do the calculation in parallel. To make a parallel calculation possible, the serial dependency can be resolved, by calculating a parallel prefix in two phases. In a pre-scan phase, only the summary of a subrange is calculated without setting the resulting value of each element. In a second phase, a final-scan is done to calculate also the resulting value for each element using summary results from pre-scanned ranges. As the calculation happens in two phase and the parallel prefix operation is applied at most twice as often as in the serial implementation, it can outperform the serial version only with right grain size and if sufficient hardware threads are available.

The syntax of `parallel_scan` can be seen in Listing 1.16. Additionally, a partitioner can be explicitly specified as third parameter, but an `affinity_partitioner` can not be used.

```cpp
template<typename Range, typename Body>
void parallel_scan(const Range& range, Body& body);
```

Listing 1.16: Syntax of `parallel_scan`.

The exact working of `parallel_scan` can best be explained by an example body (Listing 1.17) in conjunction with a possible execution flow (Figure 1.6 on the next page) where each color denotes a separate body object. The example shows the calculation of the running sum over 16 integers. In step 1 and 2, the body object is split using the splitting constructor (line 23). In step 3 a final-scan as well as two pre-scan can be executed in parallel using the overloaded `operator()` on line 14 with the scan variation distinction on line 18. In step 5 and 6 the `reverse_join` function on line 26 is executed where the summary of scans are provided to other bodies to enable parallel final-scans in step 7. The parameter in `reverse_join` is a body object from which the calling body was split before. In the final step the origin body object is updated with the summary calculated in the last range.

```
 1 template<typename T>
 2 class Body {
 3   T sum;
 4   T* const y;
 5   const int* const x;
 6 public:
 7   Body(int y_[], const int x_[]) :
 8       sum(0), x(x_), y(y_) {
 9   }
10   T get_sum() const {
11     return sum;
12   }
13   template<typename Tag>
14   void operator()(const tbb::blocked_range<int>& r, Tag) {
15     T temp = sum;
16     for (int i = r.begin(); i < r.end(); ++i) {
17       temp = temp + x[i];
18       if (Tag::is_final_scan())
19         y[i] = temp;
20     }
21     sum = temp;
22   }
23   Body(Body& b, tbb::split) :
24       x(b.x), y(b.y), sum(0) {
25   }
26   void reverse_join(Body& a) {
27     sum = a.sum + sum;
28   }
29   void assign(Body& b) {
30     sum = b.sum;
31   }
32 };
```

Listing 1.17: Body to calculate the running sum [Cor12c].

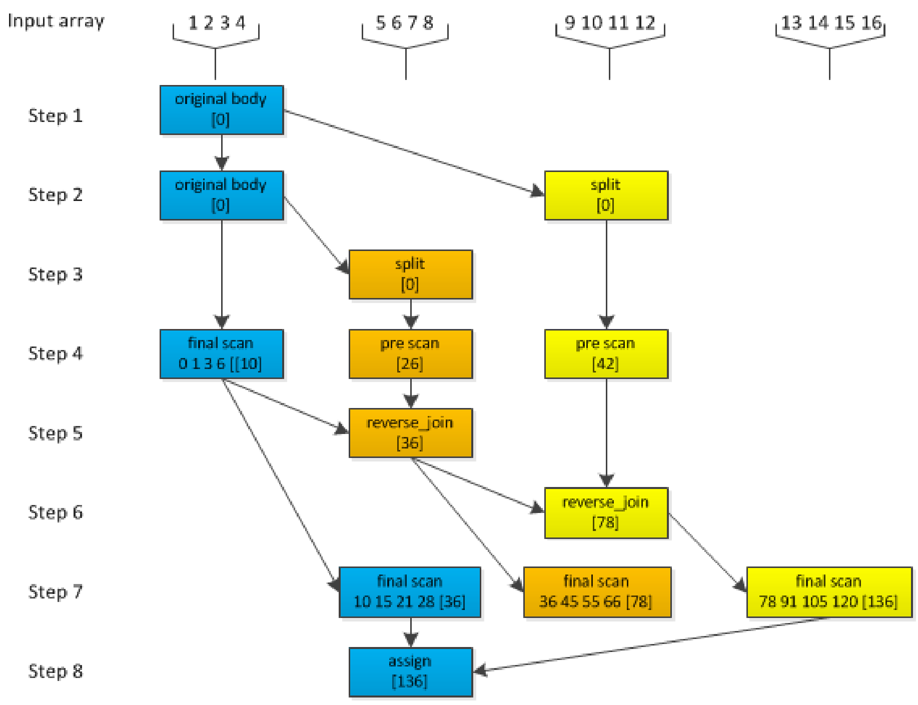


Figure 1.6.: Possible execution flow in calculating the running sum with `parallel_scan`. Each color denotes a separate body object and the values in the brackets represent the summary of the corresponding range [Cor12c].

parallel_for_each Template

The `parallel_for_each` is the parallel equivalent to `std::for_each`. It can operate directly with iterators as it can be seen form the syntax in Listing 1.18.

```
template<typename InputIterator, typename Func>
void parallel_for_each (InputIterator first, InputIterator last, const Func& f
    [, task_group_context& group]);
```

Listing 1.18: Syntax of `parallel_for_each`.

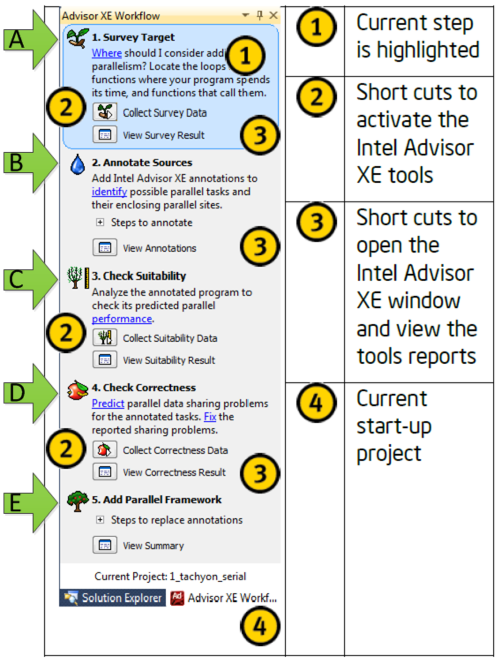
The provided functor or function pointer as last argument is applied to the result of dereferencing iterator in the range [first,last). Internally, the algorithm can not operate with a `blocked_range` and recursive splitting because this is not applicable on *InputIterators* as they do not provide random access. Every element is therefore executed in its own task. Additionally, if the user does not provide random-access iterators, the algorithm ensures that no more than one task will ever act on the iterators concurrently, but this restriction is relaxed with random-access iterators [Cor12d] [KV11]. But still, as every element is processed by a separate task also in case of a provided random-access iterators, the algorithm can only beat the serial equivalent if the computation time on the element exceeds the splitting and task scheduling overhead.

1.5. Existing Tools

There are some tools that already support the user in parallelize serial code. Some of them are quite mature and used in practice. Therefore they are analyzed and can show which advices are useful for a user in the process of parallelization. Two existing tools are introduced in this section.

1.5.1. Intel Parallel Advisor

The Intel Parallel Studio equips the developer with tools to design, build, debug, verify and tune serial and parallel applications. The whole suite consists of different tools integrated with Microsoft Visual Studio. The one that is most related to this work is *Intel Parallel Advisor*. It guides the user with a workflow (Figure 1.7). The steps in the workflow (A...E) are explained here.


Advisor XE Workflow

1. Survey Target
Where should I consider add[ing] parallelism? Locate the loops [and] functions where your program spends its time, and functions that call them.
Collect Survey Data
View Survey Result

2. Annotate Sources
Add Intel Advisor XE annotations to identify possible parallel tasks and their enclosing parallel sites.
Steps to annotate
View Annotations

3. Check Suitability
Analyze the annotated program to check its predicted parallel performance.
Collect Suitability Data
View Suitability Result

4. Check Correctness
Predict parallel data sharing problems for the annotated tasks. Fix the reported sharing problems.
Collect Correctness Data
View Correctness Result

5. Add Parallel Framework
Steps to replace annotations
View Summary

Current Project: 1_tachyon_serial

1	Current step is highlighted
2	Short cuts to activate the Intel Advisor XE tools
3	Short cuts to open the Intel Advisor XE window and view the tools reports
4	Current start-up project

Figure 1.7.: Intel Advisor XE Workflow window [Cor12a].

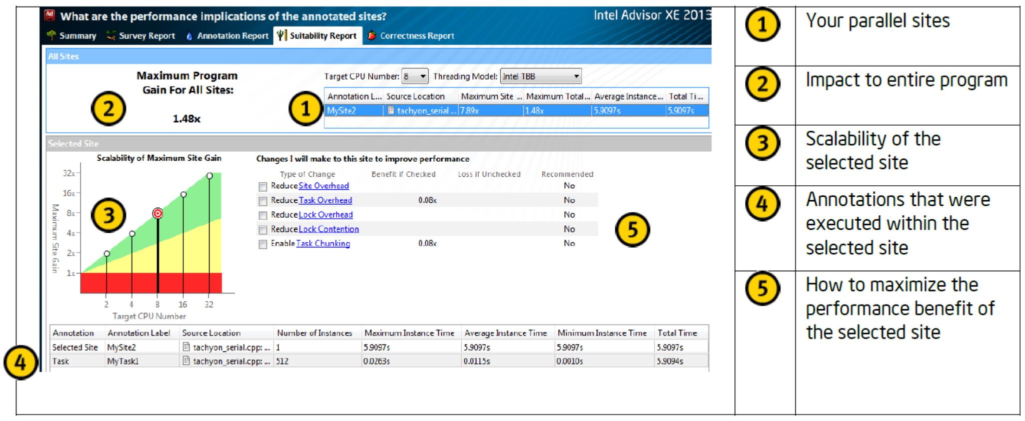
A: Survey Target: This step helps in finding call-sites and loops that consume most of the time in execution and are therefore hot spots to experiment with parallelism.

B: Annotate Sources: The source can be annotated with different annotations to model a parallel execution. A task annotation may surround the loop body to mark each loop iteration as a task. The task then is surrounded with a parallel site annotation, e.g. around the whole loop statement. The tasks inside the parallel site are then executed in parallel. With an annotation wizard the user is guided through the annotation step.

C: Check Suitability: In this step the performance of the parallel experiment is evaluated. The performance of the parallel site as well as of the whole program can be analyzed. The information provided by the suitability check can be seen in Figure 1.8 on the following page.

D: Check Correctness: The correctness tool will provide help in identifying data issues like data races in the parallel experiment. A potential data race can then experimentally be made safe by letting the Advisor annotate the racing statement with lock annotations. For verification, the suitability check and the correctness check need to be performed again.

E: Add Parallel Framework: The last step is to replace the annotations with a parallel framework. In this step the Advisor provides documentation to assist with the transformation. There is no automatic transformation.



Maximum Program Gain For All Sites:

1.48x

Target CPU Number: 8 Threading Model: Intel TBB

Annotation L...	Source Location	Maximum Site ...	Maximum Total...	Average Instance...	Total Ti...
MySite2	tachyon_serial ...	7.89x	1.48x	5.9097s	5.9097s

Changes I will make to this site to improve performance

Type of Change	Benefit if Checked	Loss if Unchecked	Recommended
☐ Reduce Site Overhead			No
☐ Reduce Task Overhead	0.08x		No
☐ Reduce Lock Overhead			No
☐ Reduce Lock Contention			No
☐ Enable Task Chunking	0.08x		No

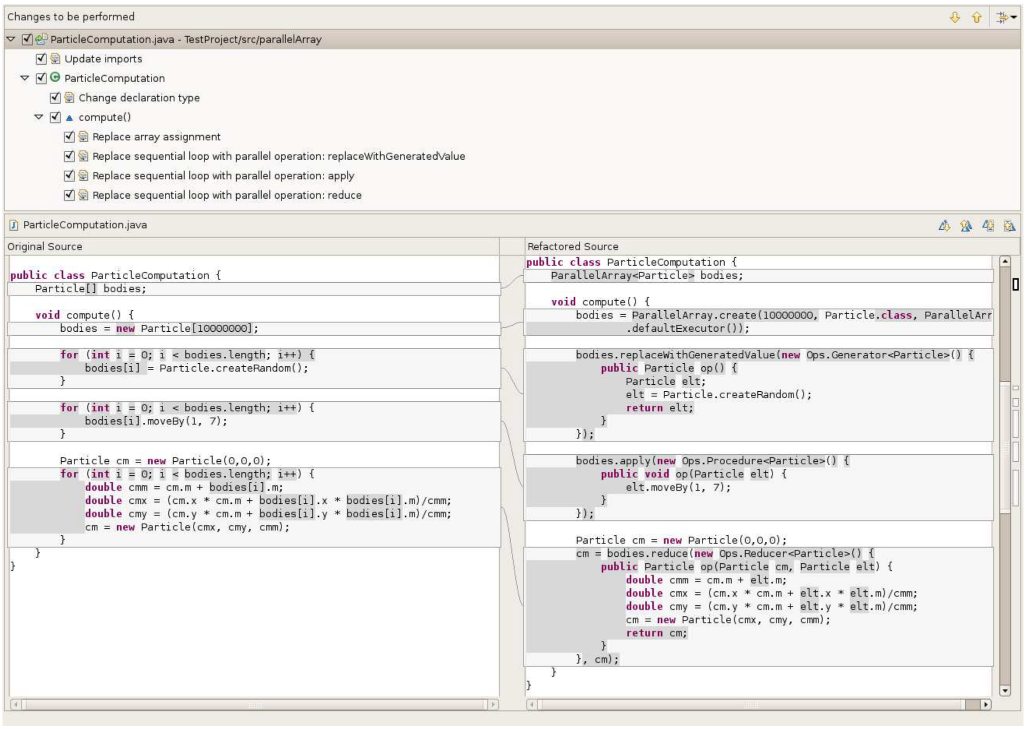
Annotation	Annotation Label	Source Location	Number of Instances	Maximum Instance Time	Average Instance Time	Minimum Instance Time	Total Time
Selected Site	MySite2	tachyon_serial.cpp: ...	1	5.9097s	5.9097s	5.9097s	5.9097s
Task	MyTask1	tachyon_serial.cpp: ...	512	0.0263s	0.0115s	0.0010s	5.9094s

1	Your parallel sites
2	Impact to entire program
3	Scalability of the selected site
4	Annotations that were executed within the selected site
5	How to maximize the performance benefit of the selected site

Figure 1.8.: Suitability data collected by Advisors suitability tool [Cor12a].

1.5.2. ReLooper

ReLooper [DD12] is an Eclipse plug-in to refactor loop parallelism in Java. It is semi-automatic and guides the user in the refactoring process. It supports the refactoring of Java arrays or vectors into `ParallelArray` [Lea12] class. A `ParallelArray` allows to apply operations on the elements of arrays in parallel. The available operations are similar to the parallel algorithms that TBB provides. As an example it also provides the operation to reduce all elements into a single value. The internals of `ParallelArray` are very similar to TBB. A pool of worker threads processes the operations on the array elements in parallel whereby the runtime library is responsible for balancing the work among the processors in the system. Figure 1.9 shows sequential loops that perform different computations over an array of `Particle` objects on the left side and the refactored version that uses `ParallelArray` on the right side.



Changes to be performed

- ✓ ParticleComputation.java - TestProject/src/parallelArray
  - ✓ Update imports
  - ✓ ParticleComputation
    - ✓ Change declaration type
    - ✓ compute()
      - ✓ Replace array assignment
      - ✓ Replace sequential loop with parallel operation: replaceWithGeneratedValue
      - ✓ Replace sequential loop with parallel operation: apply
      - ✓ Replace sequential loop with parallel operation: reduce

ParticleComputation.java

Original Source

```java
public class ParticleComputation {
    Particle[] bodies;

    void compute() {
        bodies = new Particle[10000000];

        for (int i = 0; i < bodies.length; i++) {
            bodies[i] = Particle.createRandom();
        }

        for (int i = 0; i < bodies.length; i++) {
            bodies[i].moveBy(1, 7);
        }

        Particle cm = new Particle(0,0,0);
        for (int i = 0; i < bodies.length; i++) {
            double cmm = cm.m + bodies[i].m;
            double cmx = (cm.x * cm.m + bodies[i].x * bodies[i].m)/cmm;
            double cmy = (cm.y * cm.m + bodies[i].y * bodies[i].m)/cmm;
            cm = new Particle(cmx, cmy, cmm);
        }
    }
}
```

Refactored Source

```java
public class ParticleComputation {
    ParallelArray<Particle> bodies;

    void compute() {
        bodies = ParallelArray.create(10000000, Particle.class, ParallelArr
                .defaultExecutor());

        bodies.replaceWithGeneratedValue(new Ops.Generator<Particle>() {
            public Particle op() {
                Particle elt;
                elt = Particle.createRandom();
                return elt;
            }
        });

        bodies.apply(new Ops.Procedure<Particle>() {
            public void op(Particle elt) {
                elt.moveBy(1, 7);
            }
        });

        Particle cm = new Particle(0,0,0);
        cm = bodies.reduce(new Ops.Reducer<Particle>() {
            public Particle op(Particle cm, Particle elt) {
                double cmm = cm.m + elt.m;
                double cmx = (cm.x * cm.m + elt.x * elt.m)/cmm;
                double cmy = (cm.y * cm.m + elt.y * elt.m)/cmm;
                cm = new Particle(cmx, cmy, cmm);
                return cm;
            }
        }, cm);
    }
}
```

Figure 1.9.: The programmer selects the `bodies` array and *Convert To Parallel Array* refactoring. *ReLooper* converts the sequential loops (left-hand side) into parallel loops (right-hand side) using the `ParallelArray` framework [DD12].

ReLooper uses a data-flow analysis that determines objects that are shared among loop iterations, and detects writes to the shared objects. This static analysis is based on the SSA (single static assignment) intermediate representation of a program and it analyzes both, the program in source code and in byte code. Additionally the data-flow analysis is conservative, it could give false warnings. Therefore it is crucial that the refactoring is interactive. The user as expert can decide whether to move on even though warnings exist. The decision to parallelize and ignore the warnings can be done for every loop in the example of figure 1.9.

1.6. Project Goals

TBB provides a rich set of threading building blocks that help in parallelize sequential algorithms. They provide an abstraction to parallelize operations on loop elements with different kind of dependencies. A manual conversion of the loops into parallelized TBB calls is time consuming, error prone and enforces advanced knowledge as of the following reasons:

- Finding loops that provide data or task parallelism is difficult. It requires a conscious analysis of the code.
- The manual transformation to parallelism can introduce errors. Especially if knowledge of lambda expressions and functors is not well-founded. Whether to pass a

variable as reference or value is essential in parallelism as references allow to share the same object.

- Introducing parallelism may also introduce data races that are overseen in a manual conversion.

Most STL algorithms operate on a sequence of data with different kind of dependencies. These algorithms, either explicitly programmed using a loop or the call of the specific algorithm, can be parallelized as shown in Listing 1.13 on page 16. Former work in this area [Kes10] have shown a working recognition of loops representing an STL algorithm and the conversion to the corresponding STL algorithm call.

The project goal can be stated as semi-automatic transformation of source code into semantically equal, parallelized code using TBB, eventually after user acknowledgment. As mentioned, the loops and STL call show a potential for parallelism. Besides the parallelization of already recognized STL algorithms by [Kes10], further conversions of loops into its equivalent STL calls remain another goal of the project.

The limitation a **semi**-automatic transformation with user acknowledgment is founded by the *program equivalence* problem. Determining whether two programs are equivalent is in general undecidable. This can be proven using the halting problem. The possibility to find a transformation into the semantic equal parallelized program from any program would contradict with program equivalence. For restricted cases, a guarantee for a semantic preserving transformation is possible. For many other cases, a transformation might be valid but can not be automatically guaranteed so user involvement is required to apply it. Nevertheless, the mechanics of such transformations should be automatic, once the user acknowledges them.

The focus in this project is on the recognition of parallelism and the transformation to expose them. Profiling analysis as shown by Intel Advisor (Section 1.5.1 on page 20) to find hot spots for parallelism, is not the focus.

The implemented transformation shall be applied in existing projects to show the usability and performance gain. Furthermore, the implementation will be served as an Eclipse CDT [Fou12] plug-in.

1.7. About this Report

In chapter 2 potential source code constructs for parallelism are analyzed in detail also evaluating the criteria for a loop to be safe for parallelism. The chapter describes also the tree pattern matching approach of the DS-8 project and how it can be used in this project. Chapter 3 describes the implementation details of the Parallator Eclipse plug-in. The final chapter 4 presents the project results, including a measurement to outline the performance impact of the implemented transformations. Further, open problems and enhancements are described and the project concludes with a personal reflection and acknowledgment.

2. Analysis

Loops with implied sequential execution provide a great potential for parallelism. This analysis is focused on the detection of loops to parallelize and the possible transformation to a TBB specific algorithm. The last section (Section 2.6 on page 44) then states some possible other code, where parallelism could be introduced using TBB.
The chapter starts with an introduction to the DeepSpace-8 (DS-8) project where loops are recognized and transformed to a specific STL algorithm. It gives an overview on the semantics of different loops and the limitation in the abstraction with an algorithm call. The potential usage of TBB to parallelize concrete loop examples is shown in Section 2.2 on page 26. Section 2.3 on page 33 deepens the semantic analysis of the already implemented transformation towards STL algorithm because similar considerations must also be taken as in the analysis of whether the stated potential loops are safe for parallelization (Section 2.4 on page 35). The technique used in the DS-8 project to find and transform loops is tree pattern matching. Section 2.5 on page 40 starts with an introduction into the implemented approach and is followed by an analysis on its usage and limitation to detect loops transformable towards TBB algorithms.

2.1. Loop Analysis and Transformation in DS-8 Project

The findings in loop analysis and transformation from the DS-8 project are summarized in this section. It shows the problems and limitations in loop analysis and transformation towards STL algorithms. These findings are also valid for this project.

2.1.1. Focus on Specific Loop Variants

An approach to detect **all** loops that correspond to a specific STL algorithms is not possible. A loop analysis towards a *for_each* algorithm for example, constitutes an analysis of the range of a loop. In this range, every element of the iterating sequence must be processed somehow. Extracting the range of a loop is equal to the halting problem which is not decidable [Sip06]. Whether the loop constitutes a valid *for_each* code pattern or not is therefore undecidable too. Additionally the body of a loop can be arbitrary complex and represents itself a complete program ("turing-complete systems") [Sip06]. A statement about the logical algorithm it implements, such as "Implements a *for_each* algorithm", can not be made according the Rice's theorem [Sip06]. But for specific cases the recognition of loop candidates for an STL algorithm is possible as shown in [Kes10].

Detectable Loop Variants

Listing 2.1 on the following page shows concrete *for*- and *while*-loops where the iteration start value and end value is detected. In all cases exactly one increment on the iteration variable must be in place.

```
int main() {
  char s[] = { 'a', 'b', 'c', 'd', 'e' };

  //start value found in for loop init section, end value in for loop condition
  for (char *it = s, *xy = s + 1; it != s + 5; ++it) {
    process(*it);
  }

  //start value found before the for loop, end value in for loop condition
  char* it = s + 1;
  it = s;
  for (; it != s + 5; ++it) {
    process(*it);
  }

  //start value found before the while loop, end value in while loop condition
  it = s;
  while (it != s + 5) {
    process(*it);
    it++;
  }
}
```

Listing 2.1: Possible range evaluation in loop constructs with different iteration variable initializers.

Undetectable or Wrongly Detected Loop Variants

Listing 2.2 shows cases where the loop is not detectable or the range is wrongly detected. Wrongly detected can be expressed as loops that match an assumed code pattern but additional code constructs let the range deduction fail.

```
void increment(char* &it) {
  it++;
}

int main () {
  char s[] = { 'a', 'b', 'c', 'd', 'e' };
  char *it;

  //increment hidden by a function call
  for (it = s; it != s + 5; increment(it)) {
    process(*it);
  }

  //wrongly detected loop range because of complex loop condition
  bool run = true;
  for (it = s; (it != s + 5) && run; ++it) {
    process(*it);
    if (*it == 'b') {
      run = false;
    }
  }
}
```

Listing 2.2: Examples of undetected loops and wrong range evaluation.

Both example from Listing 2.2 show the possible complexity that can be faced in parts of a loop. The incrementation part can be an arbitrary expression and the increment therefore be hidden by a function call. The range is wrongly detected in the second example as the condition can be an arbitrary expression and it is not possible to cover all cases.

Conclusion

Many potential loops are detected with the situations covered from Listing 2.1 on the previous page and it is suggested to focus on index based loops to reach more potential loops for a transformation towards STL [Kes10]. The loop analysis is focused on iterator based loops mainly because of the simpler transformation to STL algorithms which are based on iterators.

2.1.2. Loop Body Transformation

In the transformation of a loop to an STL algorithm, the loop body must be transformed into a functor or lambda. Additionally, the iteration variable can not be accessed anymore in the functors body. This leads to a transformation of iterator-based accesses to value type-based accesses. Iterator embrace a pointer akin syntax of dereferencing (∗) and member access (−>). Listing 2.3 and Listing 2.4 show this transformation.

```
vector<string> vec = getData();
for (vector<string>::iterator it = vec.begin(); it != vec.end(); ++it) {
  std::cout << *it << std::endl ;
  it->clear();
}
```

Listing 2.3: Iterator access syntax [Kes10].

```
vector<string> vec = getData();
std::for_each(vec.begin(), vec.end() [](string &it) {
  std::cout << it << std::endl ;
  it.clear();
});
```

Listing 2.4: Transformed iterator access [Kes10].

Recognize also that there is a semantic loss when moving from iterators to value types. This can be seen in Listing 2.5 where the loop semantic can not be maintained in a *for_each* algorithm. The analysis of *for_each* loop candidates does not check this and leaves this detection to the user.

```
vector<string> vec = getData();
for (vector<string>::iterator it = vec.begin() + 1; it != vec.end(); ++it) {
  std::cout << (*it + *(it - 1)) << std::endl;
}
```

Listing 2.5: Loop showing th semantic loss in transformation to *for_each* [Kes10].

2.2. Potential Transformations Using TBB

As already seen, TBB provides different parallel algorithms to parallelize loops with different dependencies between loop iterations. This section describes the possible loop variants that resemble a specific TBB algorithm and the possible transformations. Whether a transformation is safe regarding to a possible conflicting variable modification in a parallelized execution is analyzed in Section 2.4 on page 35. As STL algorithm do also imply a certain dependency in loop iterations, they are analyzed too for a possible parallelization.
During this analysis, the Thrust library [JH13], which is a parallel algorithms library

that resembles the C++ standard template library, has shown interesting for this project. Thrust uses TBB as one possible back end for parallelization. The potential usage of Thrust for this project is analyzed in Section 2.2.3 on page 32.

2.2.1. Loops to TBB Parallel Algorithms

The available analysis mechanism for loops works best and sufficiently reliable for specific loop variants [Kes10]. This section is focused mainly on elaborating specific loop variants that can be transformed to a TBB equivalent. For some algorithms it is simpler to support the transformation from index-based **and** iterator-based loops. This distinction affects also the transformation of the loop body. The variants and its specific handling are introduced in this section.

2.2.1.1. Loops to tbb::parallel_for_each

The `tbb::parallel_for_each` algorithm operates on *InputIterator* which means that this algorithm can directly be used with iterators and that index-based loops are not that simple to transform to this algorithm [Kes10].

Iterator-Based Loop

Listing 2.6 shows a possible iteration-based loop candidate and Listing 2.7 the parallelized equivalent using a lambda expression.

```
std::vector<WebSite> sites = getWebSites();
for (vector<WebSite>::iterator site = sites.begin(); site != sites.end(); ++site
    ) {
  site->search(searchString);
}
```

Listing 2.6: Iterator-based loop as `tbb::parallel_for_each` candidate.

```
std::vector<WebSite> sites = getWebSites();
tbb::parallel_for_each(sites.begin(), sites.end(), [](WebSite& site) {
  site.search(searchString);
});
```

Listing 2.7: Resulting `tbb::parallel_for_each` call using lambda expression.

As the loop body consists of only one function call that includes the iteration variable, this function call can be transformed to a unary functor that is called with the bound iteration variable using `std::bind` (Listing 2.8).

```
std::vector<WebSite> sites = getWebSites();
tbb::parallel_for_each(sites.begin(), sites.end(), std::bind(&WebSite::search,_1
    , searchString));
```

Listing 2.8: Resulting `tbb::parallel_for_each` call using argument binding.

Index-Based Loop

Listing 2.9 on the following page shows an index-based loop with the same functionality. The transformation of such a loop to `tbb::parallel_for_each` would require to be able to link the index value of 0 to sites.begin(), sites.size() to sites.end() and recognize their usage in terms of a vector access using `vector::at` or `vector::operator[]` [Kes10]. This can be seen if we consider Listing 2.7 on the previous page as the result of the index-based loop example.

```
std::vector<WebSite> sites = getWebSites();
for (vector<WebSite>::size_type i = 0; i != sites.size(); ++i) {
  sites[i].search(searchString); //alternative: sites.at(i).search(searchString)
}
```

Listing 2.9: Index-based loop as `tbb::parallel_for_each` candidate.

It is shown in [Kes10] that there exist cases in which the transformation is not as straight forward as described above.

2.2.1.2. Loops to tbb::parallel_for

The `tbb::parallel_for` algorithm provides the possibility to parallelize index-based loops without great effort. This comes from the fact that the parameters for the iteration range in `tbb::parallel_for` are not restricted to *Iterators*. Remember the *Index* concept (Section 1.4.5 on page 15) or the *Value* concept (Section 1.4.3 on page 12). Regarding to the body section, there is no semantic loss (expect the sequential execution) compared to a *for* statement. The loop range is simply separated into subranges executed by different tasks but the loop body can remain the same (apart from parallelism issues). The following examples show possible loops with a parallelized equivalent using the compact version and the more generic version of `tbb::parallel_for`.

Index-Based Loop

The index-based example in Listing 2.9 can be parallelized using `tbb::parallel_for` as shown in Listing 2.10.

```
std::vector<WebSite> sites = getWebSites();
tbb::parallel_for(size_t(0), sites.size(), [sites](const int i) {
  sites[i].search(searchString);
});
```

Listing 2.10: Resulting `tbb::parallel_for` call (compact version) from index-based loop using lambda expression.

A transformation of the index-based loop body from Listing 2.9 to a bind variation is often not possible. In most cases it would require the introduction of a new function to which the arguments (`sites` and `i`) are bound.

The compact version allows also iteration steps different to an increment by one. Therefore it is also possible to map the loop in Listing 2.11 to the `tbb::parallel_for` equivalent in Listing 2.12 on the following page

```
void addValueOnEverySecondElement(int data[], int size, int const value) {
  for (int i = 0; i != size; i += 2) {
      data[i] = data[i + 1] + value;
  }
}
```

Listing 2.11: Index-based loop with an increment by two as `tbb::parallel_for` candidate.

```cpp
void addValueOnEverySecondElement(int data[], int size, int const value) {
  tbb::parallel_for(0, size, 2, [data, value](const int i) {
    data[i] = data[i + 1] + value;
  });
}
```

Listing 2.12: Resulting `tbb::parallel_for` call with step size.

Iterator-Based Loop

In iterator based loops, the iteration variable does not satisfy the *Index* interface. The compact notation can therefore not be used. But using a `blocked_range` explicitly enables the possibility to parallelize also iterator-based loops with `tbb::parallel_for`. Not all iterators can be used. But all iterators that satisfy the *Value* interface (Section 1.4.3 on page 12) are supported. Listing 2.13 shows the parallelized version of Listing 2.6 on page 27 using `tbb::parallel_for`.

```cpp
std::vector<WebSite> sites = getWebSites();
tbb::parallel_for(
  tbb::blocked_range<std::vector<WebSite>::iterator>(sites.begin(), sites.end())
      ,
      [=] (const tbb::blocked_range<std::vector<WebSite>::iterator>& site) {
        const std::vector<WebSite>::iterator it_end = site.end();
        for (std::vector<WebSite>::iterator it = site.begin(); it != it_end; it
            ++) {
          it->search(searchString);
        }
      });
```

Listing 2.13: Resulting `tbb::parallel_for` call from an iterator-based loop.

A transformation of the loop body from iterator-based to value type-based access is not needed as the iteration is explicitly done using the *Range* object. It is therefore also possible to have access on the iteration variable which enables to parallelize loops that have operations on the iteration variable as shown in Listing 2.14.

```cpp
void multiplyValueOnEverySecondElements(int* data, int size, int const value) {
  for (int* i = data; i != data + size; i += 2) {
      *i = *(i + 1) * value;
  }
}
```

Listing 2.14: `tbb::parallel_for` candidate with operations on the iteration variable in the loop body.

A step size different to one is also possible in an explicit usage of a *Range*. But it must be assured, that the iteration of the sub ranges are aligned correctly. The range must first be scaled down by the step size. In the iteration over the range the index used to access the sequence is then upscaled again to get correct indexes. An example of this process is shown in Listing 2.15 on the next page that shows the parallelized version of Listing 2.14.

```
 1 void multiplyValueOnEverySecondElements(int* data, int size, int const value) {
 2   int end = ((data + size) - data - 1) / 2 + 1; //scale down the range (last -
       first - 1) / step_size + 1
 3   tbb::parallel_for(tbb::blocked_range<int>(0, end), [data, value](const tbb::
       blocked_range<int>& r) {
 4     //scale up to access the correct offsets: k = first + range_iteration *
         step_size
 5     int *k = data + r.begin() * 2;
 6     for(int i = r.begin(); i != r.end(); i++, k = k + 2) {
 7       *k = *(k + 1) * value;
 8     }
 9   });
10 }
```

Listing 2.15: Using `tbb::parallel_for` and `blocked_range` explicitly with a step size.

The variable `k` is correctly aligned on line 5. It is achieved with pointer arithmetic. Pointer arithmetic does not allow to use multiplication and therefore the *Range* is instantiated using integer type. This allows to do the calculation on line 5 that uses the *Range* object. Figure 2.1 shows the mapping of the ranges to the data index. The shown range objects can result from recursively splitting a sequence of size 20 with a grain size of 2 and using the `auto_partitioner`.

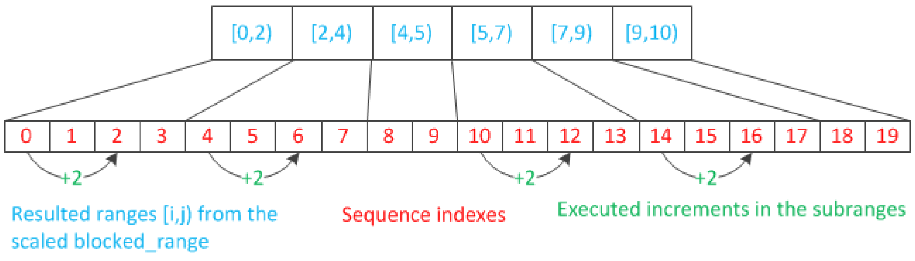


Figure 2.1.: Mapping the scaled and split *Range* objects to the indexes of the sequence.

2.2.2. STL Algorithms to TBB Parallel Algorithms

STL algorithms imply a loop with a specific data dependency between the iterations. This section is focused on which and how STL algorithms can be parallelized using TBB.

Example - Mapping STL to TBB

The `std::count` algorithm increments a counter in case the iteration element equals a certain value. The counter is shared over loop iterations. This algorithm shows a dependency between loop iterations that can be parallelized using `tbb::parallel_reduce`. Listing 2.16 shows the serial `std::count` version from STL that counts the number of elements in `vec` that are equal to 42. This example can be parallelized as shown in Listing 2.17 on the next page using the reduction body as introduced in Section 1.4.5 on page 15.

```
int count (std:vector<int> vec) {
  return std::count(vec.begin(), vec.end(), 42);
}
```

Listing 2.16: Usage of `std::count`.

```cpp
class Count {
public:
  int my_count;
  void operator()(const tbb::blocked_range<std::vector<int >::iterator>& r) {
    my_count += std::count(r.begin(),r.end(),42);
  }
  Count(Count& x, tbb::split) :
      my_count(0) {
  }
  void join(const Count& y) {
    my_count += y.my_count;
  }
  Count() : my_count(0) {
  }
};

int count (std::vector<int> vec) {
  Count counter;
  tbb::parallel_reduce(tbb::blocked_range<std::vector<int >::iterator>(vec.begin
      (), vec.end()), counter);
  return counter.my_count;
}
```

Listing 2.17: Parallelized implementation of `std::count`.

Such a transformation implies code generation of the reduction body. This code could be more generic to be used for other STL algorithm that show a reduction like data dependency. These reasons induce a parallelized library with STL equivalent calls.

Parallelization of STL Algorithms

A library that provides parallelized equivalents to STL algorithms is Thrust. Thrust parallelizes the STL algorithm using TBB and shows therefore, that nearly all STL algorithm can be parallelized. The library is built up using basic constructs that are reused in the provided parallelized algorithms [ea13]. An analysis of Thrust has also shown that the reduction body and scan body are template functors. For their reuse, these functors are parametrized with another functor that is used to apply the desired operation in the iteration on the subranges. A rough overview on how the provided parallel algorithms of Thrust are implemented can be found in Table A.1 on page 82.

Conclusion

A transformation as needed from Listing 2.16 on the preceding page to Listing 2.17 is too specific and would induce duplicated code. With the usage of thrust, the transformation is simplified as shown in Listing 2.18 that shows the parallelized equivalent of Listing 2.16 on the previous page using Thrust.

```cpp
int count (std:vector<int> vec) {
  return thrust::count(vec.begin(), vec.end(), 42);
}
```

Listing 2.18: Parallelization of `std::count` using Thrust.

More on Thrust and a possible integration to this project follows in the next section.

2.2.3. Thrust

Thrust parallelizes algorithms of the STL. The implementation of Thrust allows the programmer to use different back ends for parallelization. Besides TBB, CUDA and OpenMP are possible back ends. This might imply additional information that must be specified in a transformation of loops or STL calls to equivalent algorithms in Thrust. This section shows how the additional information can be handled for a straight forward transformation as shown in Listing 2.18 on the preceding page. Further, the limitations of Thrust compared to a direct usage of TBB are outlined.

Minimize Overhead in Transformation

To determine which implementation of the called thrust algorithm to take, the type of the provided iterator is inspected. This process is known as static dispatching since the dispatch is resolved at compile time and no runtime overhead occurs in dispatching [JH13]. There are three different possibilities to influence the evaluated type of an iterator:

- **TBB specific vector:**
  A vector declared of type `thrust::tbb::vector` works the same as a `std::vector` but the returned iterator type, e.g. in calls of `begin()` or `end()`, is tagged with `thrust::tag::tbb`. This indicates the execution of the algorithm using TBB. In a transformation, this would imply to change the declaration of the used vector variable.

- **Retagging to TBB iterator:**
  Instead of changing the type of the vector variable it is also possible to retag the returned iterator. Retagging as seen in Listing 2.19 with `thrust::tbb::tag` enables the parallel execution with TBB.

```
int count (std::vector<int> vec) {
  return thrust::count(thrust::retag<thrust::tbb::tag>(vec.begin()), thrust
      ::retag<thrust::tbb::tag>(vec.end()), 42);
}
```

  Listing 2.19: Change the inspected type of an iterator with retagging.

- **TBB specific Compiler Option:**
  Setting the compiler option `-DTHRUST_HOST_SYSTEM=THRUST_HOST_SYSTEM_TBB`, every iterator returned by a `std::vector` indicates the execution using TBB. This option enables a straight forward transformation as shown in Listing 2.18 on the preceding page. Also pointers to array elements are interpreted as TBB iterators.

Limitations of Thrust using TBB

Thrust wraps TBB to parallelize the STL algorithms. The user is not aware of any TBB specific properties. This limits thrust to use some features of TBB:

- the grain size can not be specified. The default value is used.
- the partitioner used can not be specified. `auto_partitioner` is used as default.

Additionally the interface of the algorithms in Thrust makes use of iterators. This complicates the transformation from index-based loops in some cases as outlined in Section 2.2.1.1 on page 27.

2.3. Semantic Analysis Towards STL Algorithms

The existing transformations of loops to STL Algorithms using a lambda functor implement a default behavior. The lambda parameter is always passed by reference and a default reference capture is used. This solution is universal and works in every case. But it makes the understanding of the code more difficult, especially the default capture behavior, as the origin of a variable in the lambda body can not be seen from the lambda capture. This section analysis which variable need to be captured by value and which by reference to keep equal semantic. Whether a lambda parameter can be passed by reference or when a constant reference is sufficient is analyzed too. The constant reference variant is preferred to a pass by value variant as the later might induce an expensive copy operation.

2.3.1. Pass Lambda Parameter by Reference / Constant Reference

In some implementations the loop body only reads the iteration element, where in other implementations the iteration element is changed e.g. for in-place manipulation. This influences the declaration of the lambda parameters. Considering the example in Listing 2.20 the sequence element is manipulated in-place. In order to allow in-place manipulation and therefore write access on the iteration element using `std::for_each` with a lambda functor, the sequence element must be passed by reference to the lambda functor as it can be seen in Listing 2.21.

```
void addValueOnElements(int data[], int size, int const value) {
  for (int *it = data; it != data + size; ++it) {
    *it += value;
  }
}
```

Listing 2.20: In-place change of a sequence using a *for* loop.

```
void addValueOnElements(int data[], int size, int value) {
  for_each(data, data + size, [=] (int &it) {
    it += value;
  });
}
```

Listing 2.21: In-place change of a sequence using `std::for_each`.

If the loop body contains only read access on the iteration element, it is safe to pass the element by a constant reference. But considering the example in Listing 2.22 on the following page it is not sufficient to only analyze the loop body to determine read only access. As the iteration element is passed by reference to the function `manipulate` where the element is changed. A loop body only analysis would misclassify the iteration variable as passed by constant reference candidate.

```
void manipulate(int& i, int const value) {
  i = j;
}

void addValueOnElements(int data[], int size, int const value) {
  for(int *it = data; it != data + size; ++it) {
    manipulate(*it, value);
  }
}
```

Listing 2.22: Requires enhanced analysis method to recognize the iteration variable as passed by constant reference candidate for lambda parameter.

2.3.2. Capture by Reference or by Value

All the variables used in a loop body must be available in the lambda body. As described in Section 1.3.1 on page 6 this is done with lambda captures. The code sample in Listing 2.23 shows various cases to describe the evaluation of lambda captures from a loop body.

```
 1 class Util {
 2   private:
 3     void doSomething() {};
 4   public:
 5     int initValue;
 6 
 7     Util(int value) : initValue(value) {}
 8 
 9     void addValueOnElements(int data[], int size, int const value) {
10       Util t(0);
11       int count = 0;
12       for (int *it = data; it != data + size; ++it) {
13         *it += (value + initValue);
14         t.initValue++;
15         this->initValue++;
16         doSomething();
17         count++;
18       }
19       std::cout << "Manipulated items: " << count;
20     }
21 };
```

Listing 2.23: Different cases for lambda capture evaluation from loop body.

The variables to capture are evaluated as follows:

- **Variable `it`** - Line 13: Not needed to capture as `it` will be available from the lambda parameter.
- **Variable `value`** - Line 13: Must be captured as the variable is declared before the loop body. The variable is not changed in the body and can therefore be captured by value.
- **Variable `t`** - Line 14: Must be captured as the variable is declared before the loop body. The state of the object `t` is changed but not accessed later on. It is therefore also possible to capture by value. A write on a variable that is captured but not accessed later does not make sense mostly and is considered an uncommon case.

- **Local field `initValue`** - Line 15: The access on a local field requires to capture the *this* pointer by value. The *this* pointer references the object whose `addValueOnElements` function runs the loop.
- **Local member function `doSomething`** - Line 16: The call of a local member function enforces to capture the *this* pointer by value.
- **Variable `count`** - Line 17: Must be captured as the variable is declared before the loop body. The variable is changed and accessed after the loop. This enforces to capture the variable by reference to maintain equal semantic.

The resulting equivalent code with the transformed lambda in a `std::for_each` call is shown in Listing 2.24.

```cpp
 1 class Util {
 2   private:
 3     void doSomething() {};
 4   public:
 5     int initValue;
 6
 7     Util(int value) : initValue(value) {}
 8
 9     void addValueOnElements(int data[], int size, int const value) {
10       Util t(0);
11       int count = 0;
12       std::for_each(data, data + size, [this, value, t, &count](int& it) {
13         it += value;
14         t.initValue++;
15         this->initValue++;
16         doSomething();
17         count++;
18       });
19       std::cout << "Manipulated items: " << count;
20     }
21 };
```

Listing 2.24: Lambda expression with evaluated captures.

All the considerations can be applied to transformations using a functor, where the capture variant influences the constructor declaration of the functor.

2.4. Semantic Analysis Towards Parallelism

In STL calls, functor, lambda functors and functions constitute a body that is applied in iterations. In loops, the body is applied in iterations. For parallelization the body is copied and applied to chunks in parallel. If variables are shared between the bodies there is potential for a conflicting access. This analysis defines when variables are shared in various cases and when the access is conflicting. Also the data dependency in loop iterations is analyzed. The over restricted definition at the beginning of this section of whether a loop is parallelizable or not is relaxed later on for a reduction or a scan.

2.4.1. Conflicting Memory Accesses

Before we consider an example to elaborate a conflicting memory accesses, some definitions are made:

- *Definition 1.* A **parallel section** contains program instructions that will be executed in parallel in a particular loop candidate for parallelization [DD12].
- *Definition 2.* Each variable represents an **object** that is located at one or several memory location. Scalar objects, as integer variables or also a pointer occupy one memory location. An instance of a class is itself an object and constitutes of further memory locations for the fields [Bec12].
- *Definition 3.* A **shared object** is an object that is shared between parallel loop iterations. Fields of a shared object are shared themselves [DD12]. A **pointer variable** occupies a memory location and points to another address. If two pointer point to the same location, the object at the shared location is a shared object. A **reference variable** is just an alias to another variable. The aliased variable is a shared object.
- *Definition 4.* A write on a shared object is a **race**. This comes from the definition, that a data race occurs if two operations of different threads access the same object (memory location) and one of them is a write [Bec12].

The definitions are illustrated in Listing 2.25.

```
 1 class Util {
 2
 3 private:
 4   void doSomething() { //arbitrary operations. Field upate?}
 5 public:
 6   int initValue;
 7
 8   Util(int value) : initValue(value) {}
 9
10   void executeOnElements(SomeClass* data[], int size, int const value) {
11     Util a(0);
12     Util* pA = new Util(0);
13     int count = 0;
14     for (SomeClass **it = data; it != data + size; ++it) {
15       (*it)->execute(value + initValue); //possible write on shared object
16       this->initValue++; // initValue is shared
17       doSomething();      // possible write on shared object
18       count++;            // count is shared
19       a.initValue++;      // initValue is shared
20       Util b = a;         // new object b with values of a
21       b.initValue = 3;    // b is not shared
22       Util* pB = &b;      // new pointer pB to object b
23       pB->initValue++;    // object pB points to is not shared
24       Util* pC = pA;      // new pointer pC points to same object as pA points to
25       pC->initValue++;    // object pC points to is shared
26       pC = new Util(0);   // pC points to new object
27       pC->initValue++;    // object pC points to is is not shared
28     }
29   }
30 };
```

Listing 2.25: Examples for evaluating writes on shared objects.

The parallel section enfolds from line 15 to line 27. A variable can reference a shared object during the whole parallel section or just at a specific program point [DD12]. Variable `a` is shared at any time in the parallel section. This is not the case for the object the pointer variable `pC` points to. At line 25 `pC` points to a shared object whereas on line 27 the object `pC` points to is not shared. According to the comments in Listing 2.25 the following lines constitute a race: 16, 18, 19, 25. The operations on line 15 and 17 may have potential for a data race as in the called function a shared object might be updated. For line 15 this is possible if the `data` array consists of pointers pointing to the same object and the function `execute` updates fields of this object. We can conclude that a parallel section without a race as defined is safe for parallelization and does not produce a data race.

2.4.2. Lambda Functors Towards Parallelism

In a possible parallelization of a `std::for_each` call that uses a lambda functor, the lambda functor must be analyzed for a *race* as explained in the prior section. Most cases are similar to those shown in Listing 2.25 on the preceding page. But for lambda functors it is crucial to analyze how the variable are captured for the parallel section. This is shown in Listing 2.26.

```
 1 void addValueOnElements(int data[], int size, int value) {
 2   int count = 0;
 3   std::for_each(data, data + size, [value,&count] (int& it) mutable {
 4     if (it != 5) {
 5       it += value;
 6       count++;
 7       value++;
 8     }
 9   });
10 }
```

Listing 2.26: Reference captured `count` produces a *race*.

The variable `count` is a shared object in the parallel section (line 4-8) because it is captured by reference for the lambda body. The write on line 6 constitutes therefore a *race* and the parallelization is not safe. Apart from this issue and considering the example without the variable `count`: The variable `value` is captured by value and can only be modified because the lambda is mutable. As the lambda might be copied and executed in parallel, `value` is not a shared object over the iterations. There is no *race*. But the semantic can also not be kept the same as in a serial execution. To conclude, a lambda is safe for parallelization:

- if no write accesses on reference captured variables exist,
- and is not mutable.

2.4.3. Independent Iterations

The recognition of a *race* is one aspect in transformation towards parallelization. Another similar aspect is the dependence of the iteration. It defines whether it is possible to execute the iteration in a different order than in sequential order. This is especially interesting in parallel algorithms where access on the iteration variable is possible. It enables to manipulate which sequence element is accessed in an iteration. Index- and iterator-based loops are equal in this aspect.

Loop Carries a Dependency

A loop carries a dependency if two different iterations access the same sequence element (memory location) and one of them is a write operation. If this can happen, the result of the computation depends on the iteration order. In a parallel execution this order can not be influenced. The parallelization of such a loop in C++ results in a data race [Bec12]. Listing 2.27 shows an example with operations on the iteration variable and the step size is 3.

```cpp
void multiplyValueOnEveryThirdElement(int data[], int size, int const value) {
  for (int i = 2; i < size; i += 3) {
      data[i] = data[i - 1] * data[i - 2] * value;
  }
}
```

Listing 2.27: Loop with operations on the iteration variable.

It needs to be analyzed whether `data[i]` references the same element as in `data[i - 1]` or `data[i - 2]` for any other iteration. One need to extract the index operations. Normalizing the loop (increment by 1, start at 0) provides the indexes `[3 * i + 2]` , `[3 * i - 1 + 2]` and `[3 * i - 2 + 2]`. With the GCD test [ZC91] it is verified whether these three indexes can ever be the same for any iteration. If yes, there is a dependency. The loop in the example is ready for parallelization. Of course this test can only guarantee correctness if the elements in the sequence do not refer to the same memory location. This is especially a problem in a sequence of pointers.

Loop Independence and Criteria in a Reduction

Listing 2.28 shows a classical example of a reduction. It calculates the sum of all elements in a sequence.

```cpp
int arraySum(int data[], int size) {
  int sum = 0;
  for (int i = 0; i != size; ++i) {
      sum += data[i];
  }
  return sum;
}
```

Listing 2.28: Serial implementation to compute the sum of elements in an array.

According to the analysis so far, this loop constitute a *race* for `sum`. But because the operations in the loop body accumulate to one variable, the loop can be parallelized using reduction. Each reduction body of a subrange then calculates the sum independently to a variable not shared over different tasks.
The operator used in the reduction must be associative. Associative and commutative are defined as follows for a binary operator $\otimes$:

- **Associative:** $(a \otimes b) \otimes c = a \otimes (b \otimes c)$
- **Commutative:** $a \otimes b = b \otimes a$

It shows that for associative operation the order of application is independent. The need for associativity in reduction can be seen in the fact that the sum of element 2 and 3 is not guaranteed to be computed before the sum of elements 4 and 5. The operation in reduction does not need to be commutative because the operands to the binary operator are never reversed in a reduction.

Loop Independence and Criteria in Scan

Listing 2.29 shows a classical example of a scan. It calculates the running sum of all elements in a sequence.

```
int runningSum(int y[], const int x[], int n) {
  int temp = 0;
  for( int i = 1; i <= n; ++i ) {
    temp = temp + x[i];
    y[i] = temp;
  }
  return temp;
}
```

Listing 2.29: Serial implementation of parallel prefix that calculates the running sum [Cor12c].

Here again, the analysis so far constitutes a *race* for variable `temp`. The loop also carries a dependency. The dependency gets obvious when writing the dependency explicitly as `y[i] = x[i − 1] + x[i]`. Nevertheless this loop can be parallelized using a scan as explained in Section 1.4.5 on page 15. The scan produces all partial reductions on an input sequence and stores the results in a new output sequence [MRR12]. It is therefore also crucial that the operator used for the reductions is associative.

2.4.4. Additional Criteria for Parallelization

Despite conflicting memory access and iteration independence, there are other criteria that hinder in parallelization of loops. Namely these are statements that abort the loop traversal: `return`, `break` or `throw` [DD12]. This general statement applies not to all loops. A *find_if* loop returns the iterator position when the predicate matches. Although this loop aborts, a parallelization is possible. The *find_if* algorithm could be applied in parallel on the chunks. With a minimum reduction, the first iteration position can be evaluated. The *find_if* implementation in Thrust works like this.

In general in a sequential program, the execution is continued after the loop in case of a loop abortion. For the parallel execution this implies that all parallel tasks need to be aborted too if one iteration stops. In TBB this is possible through an explicit canceling request using the *task_group_context* (see Section 1.4.5 on page 15). A thrown exception in an iteration is an implicit canceling request. Canceling a *task_group_context* causes the entire subtree rooted at it to be canceled [Cor12c]. In the default behavior, this will cancel all the tasks created for a called TBB algorithm.

2.5. Tree Pattern Matching

In the DS-8 project, a tree pattern matching algorithm is used to identify loops equivalent to the STL algorithm. The algorithm is applied on the program code's abstract syntax tree (AST). In that way, loops are identified and specific parts of a loop, as the iteration variable, can be extracted to be further used in the transformation. In contrast to *regular expressions* that define patterns to match in a lexical representation, tree patterns additionally specify hierarchical information to match: How nodes in a tree must relate to each other. This section explains the tree pattern matching algorithms and important concepts further. More detailed descriptions can be found in [Kes10] [Kem12]. The algorithm as implemented in DS-8 is analyzed towards its usage and limitations in the recognition of further STL algorithms and parallelization issues as outlined in Section 2.4 on page 35.

2.5.1. Available Implementation

The algorithm is best imagined as if the root of a tree pattern is placed at every node in the AST. Each time the tree pattern can fully cover the subtree, rooted at the node covered by the patterns root, the node matches the tree pattern. Figure 2.2 illustrates different matches. The root of the pattern is marked with an “R”. The curvy line in the tree pattern defines that the nodes matching the connected patterns must not be exactly aligned. The tree pattern is built up of different pattern objects. The different available patterns are introduced in the following sections and an example shows the dynamics with concrete patterns.

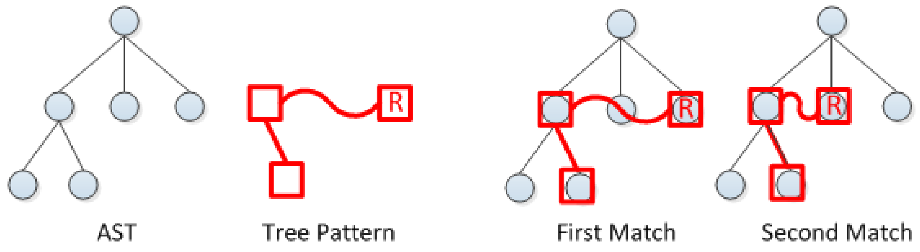


Figure 2.2.: Tree pattern matching in the AST. The node covered by the tree patterns root is a match.

Structural Patterns

Structural patterns are used to define how matching nodes must be related to each other. An example of a structural pattern is the *Dominates* pattern. It matches a node that satisfies the first pattern and the node matching the second pattern must be an ancestor of the parent. Figure 2.3 on the next page shows this pattern. The green marked node satisfies the first pattern. The red nodes are the nodes that are traversed to look for a node that satisfies the second pattern. Another structural relation is the sibship. An example is the *AnyLeftSiblingSatisfies* pattern. This pattern takes another pattern as parameter that specifies how any of a node's left sibling(s) must match to get found. This pattern is also shown in Figure 2.3 on the following page.

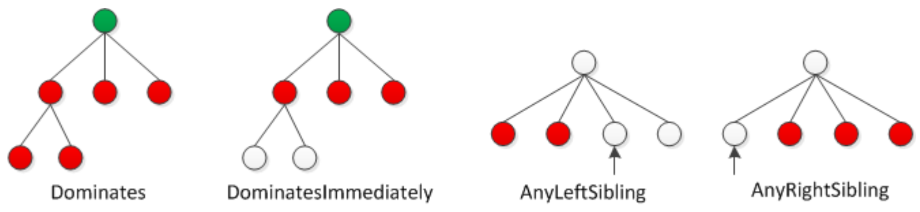


Figure 2.3.: Overview of some structural patterns implemented in the DS-8 project [Kem12].

Logical Patterns

A subtree can be required to satisfy multiple patterns at the same time or to satisfy any one of a set of arbitrary patterns [Kes10]. Figure 2.4 shows an example of subtree that requires to satisfy multiple patterns.

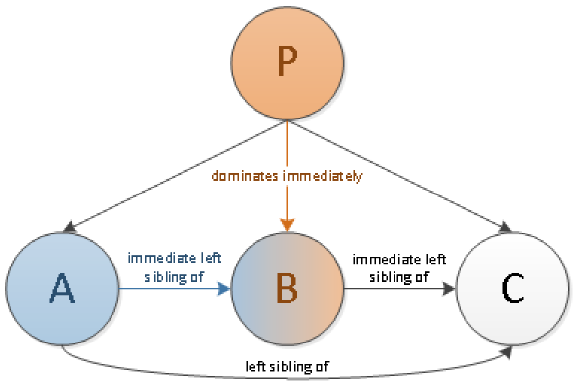


Figure 2.4.: B is immediately dominated by P AND is the immediate right sibling of A [Kes10]

Semantic Patterns

The semantic patterns are often parameters to logical or structural patterns. The implementation of these patterns analyze internals of an AST node to define whether the node matches the pattern. An example pattern is the *IsOperatorIncrement* pattern. It satisfies if the node in question is an increment on a node that matches another pattern. For the analysis of an increment, the AST node must be a *UnaryExpression* with the increment operator.

Capture and Reference

An important concept in the implemented tree pattern matching algorithm is capture and reference. A capture pattern can wrap any other pattern and stores the node matching the wrapped pattern. This allows to extract specific nodes. This can be good illustrated as in Figure 2.5 on the following page where the pattern matches at the node covered by the root of the tree pattern. In this moment the node matching the wrapped pattern of the capture is stored.

A reference pattern allows to wrap a capture pattern. It references not the capture pattern but the node stored by the capture pattern. This allows do define that two nodes in a tree pattern must be equal. Figure 2.6 shows a pattern that defines that the node dominating another must be equal. The equality of the nodes can be specified with different comparators. E.g. source equivalent, if the nodes don't need to be exactly the same node but represent the same source code. Capture and reference is used to map an increment on the iteration variable in the body of a loop to a captured iteration variable in the initializer of a *for* loop.

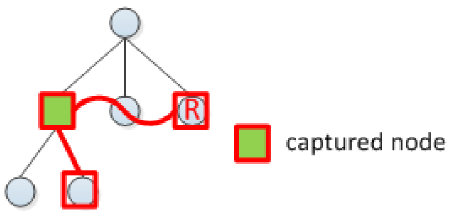


Figure 2.5.: Node behind "R" matches and the node behind the capture pattern is stored.

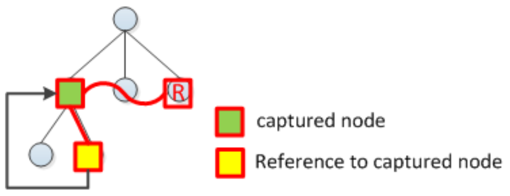


Figure 2.6.: The captured node must dominate an equal node.

Pattern Matcher Dynamics

The pattern matcher is addressed with the visitor pattern [GHJ95]. While traversing the nodes of the AST, each visited node is checked against the tree pattern, starting at the root of the tree pattern. The check whether a node satisfies the pattern or not, is the responsibility of the pattern itself. An example will show this behavior. The tree pattern used in this example is shown in Figure 2.7. It represents the tree pattern with identical hierarchical constraints as shown in Figure 2.2 on page 40 but using the pattern objects as described above. The round objects are semantic patterns and can be read as a pattern that satisfies for any node of kind C or D respectively.

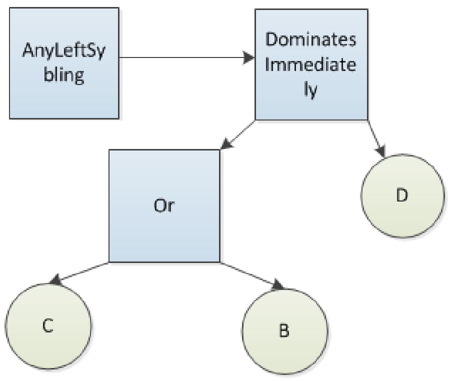


Figure 2.7.: Combined tree pattern composed of different pattern objects.

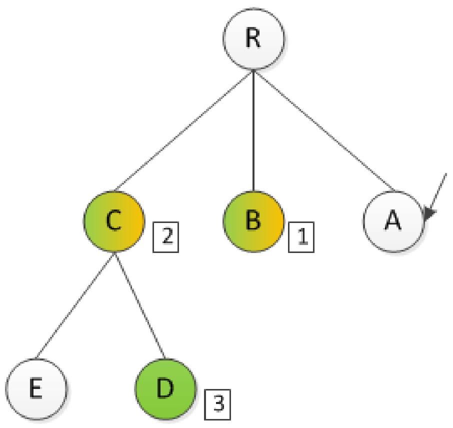


Figure 2.8.: Navigation of the pattern shown in Figure 2.7 [Kem12].

Figure 2.8 on the preceding page shows the synthetic AST in which we search for a node matching the pattern in Figure 2.7 on the previous page. The situation shows the pattern matcher at node A with the numbered navigation to find node A satisfying the pattern. The navigation done by the *AnyLeftSibling* pattern is marked in orange and the navigation of the *DominatesImmediately* is marked in green. The check whether a pattern matches a node is executed as long as the pattern as a whole does not match. Node B does not dominate a node D whereon the next left sibling is checked. Recognize also that the pattern matcher also visits node B that matches the pattern from Figure 2.7 on the preceding page too. To assemble a tree pattern the AST of the code to found must be known.

Textual Notation of Patterns

For the simplified explanation of patterns and to avoid the need of a picture for every combined pattern, a textual notation is used in this report. The notation is shown in an example. The following textual pattern describes the patten from Figure 2.7 on the previous page.

- ***AnyLeftSibling(DominatesImmediately(Or(C,B), A))***

2.5.2. Usage and Limits in Tree Pattern Matching

The algorithm as implemented proved to be satisfiable for the transformation of loops to *find_if* and *for_each* calls. The usage and limitation in further transformation to STL calls and parallelization issues is analyzed here.

Recognition of STL Algorithms

The tree pattern matching algorithm works reliable to find **specific** loops that show a behavior equivalent to an STL algorithm. The more complex the implementation of an STL algorithm is, the more variants exist to implement this algorithm and the more difficult it gets to cover the variants with a tree pattern. Algorithms that show a unique characteristic are easier to detect. This is one point and comes naturally with the theorem of Rice.
Another point that complicates the detection of an algorithm are function calls. Some STL algorithms can be implemented using another algorithm. This is also possible in a custom implementation. A pattern in this case might be hidden by the function call and is out of scope for the visits of the pattern matcher. Table A.1 on page 82 shows the evaluation according to recognizability of STL algorithms.

Check for Parallelism

The parallelization issues discussed in Section 2.4 on page 35 with the evaluation for their possible implementation with tree pattern matching is given in the following list:

- **Shared object detection:** This involves the detection of variables in a parallel section that are declared before the loop. This is a similar problem as looking for iteration variable access in a loop, which is possible. The detection of variables that are captured by reference in a lambda is a similar problem too. The detection gets hard or infeasible if a pointer variable is declared in the parallel section and initialized with a pointer variable declared before the loop. This requires an advanced analysis of how a variable is initialized or changed in the parallel section. To detect all accesses on shared object, also the potential shared objects behind a function call must be analyzed. This is not possible with the current tree pattern matching implementation.

- **Write access on shared object:** This is possible for write access not hidden by function calls. An advanced analysis is needed for overloaded operators, to detect write access operations.
- **Different elements in a sequence:** This is not possible with pattern matching.
- **GCD test:** The extraction of the values needed for the GCD test is possible and might be a task for capture patterns. To test for overlapping writes and reads of the same sequence element it is also required to extract the value of the last index in the iteration. Each possible read index must be checked against possible write indexes. The evaluation of the last index in the iteration can be difficult or even unfeasible if the last index is e.g. a user input.
- **Reduce dependency detection:** The detection of a reduction to one variable in a loop is possible to specify with a tree pattern. The used reduction operator can be extracted.
- **Scan dependency detection:** The detection of a scan like dependency in a loop is possible too. The used reduction operator for the partial reductions can be extracted.

To conclude, the most problems can be tackled with tree pattern matching as implemented. And if it is not feasible to safely state "safe for parallelization" tree pattern matching allows to state at least, that there is a potential problem. The user is then in charge to do the transformation or not. Of course, complex code still imply more complicated tree patterns for different variations.

2.6. Potential further Refactorings

During the analysis of this project also other possible refactoring came up. Also influenced by other work done in this area. This ideas are collected here with a statement of its feasibility.

2.6.1. Divide-and-Conquer Algorithms

Computational tasks are perfectly parallelizable if they operate on different parts of the data or they solve different subproblems. Divide and conquer algorithms fall in this category. A static analysis can determine dependencies between the recursive tasks, e.g. whether the tasks write within the same ranges of an array [RR99]. Figure 2.9 on the following page shows the pseudo code of a serial and a parallel divide-and-conquer algorithm. An important factor in the parallel version is the grain size, here expressed as `SEQ_THRESHOLD`. It specifies the bound of when the problem is better solved using the serial algorithm than to split further. If the problem size is not yet below the threshold, the problem is split and executed in parallel for further split or serial execution.

```
// Sequential version

solve (Problem problem) {
  if (problem.size <= BASE_CASE )
    solve problem directly
  else {
    split problem into independent tasks

    solve each task


    compose result from subresults
  }
}
```

```
// Parallel version

solve (Problem problem) {
  if (problem.size <= SEQ_THRESHOLD )
    solve problem SEQUENTIALLY
  else {
    split problem into independent tasks
    IN_PARALLEL{ //fork
      solve each task
    }
    wait for all tasks to complete //join
    compose result from subresults
  }
}
```

Figure 2.9.: Sequential and parallel pseudo code for a divide-and-conquer algorithm[DME09]

A concrete implementation of a parallel version using `tbb::parallel_invoke` is shown in Listing 2.30. It uses lambda expressions with reference capture to overcome the issue of accepting only nullary functors as parameter to `parallel_invoke` and its ignorance of return values for a provided nullary function. On line 4 the threshold to use the serial version can be seen. With the `parallel_invoke` on line 8 two tasks are spawned to recursively calculate the Fibonacci number, the results are caputred in `x` and `y`. The composition of the subresults can be seen on line 12.

```
 1 void parallelFib(int N, long &sum) {
 2   if (N < 2)
 3     sum = N;
 4   else if (N < 1000)
 5     sum = serialFib(N);
 6   else {
 7     long x, y;
 8     tbb::parallel_invoke(
 9       [&] () { parallelFib(N-1, x);},
10       [&] () { parallelFib(N-2, y);}
11     );
12     sum = x + y;
13   }
14 }
```

Listing 2.30: Parallel Fibonacci number calculation using `tbb::parallel_invoke` [Gue12].

Another parallel implementation of parallel computation of a Fibonacci number can also use the low level tasking interface of TBB [Cor12c]. Both implementations make use of the task scheduler that turns this potential parallelism of the tasks into real parallelism in a very efficient way as described in Section 1.4.2 on page 10.

In Java it has shown that a refactoring of sequential divide-and-conquer algorithms into a parallel version that makes use of Java's `ForkJoinTask` is feasible and results in a speedup [DME09].

2.6.2. Replace Threading Architecture

Threads are used far before the time TBB is introduced. There might be useful transformation to make use of TBBs rich and scalable task scheduler for algorithms that do not match to one of the high-level templates. While the result of a refactoring like this would increase the maintainability, productivity and maybe also the performance, the recognition of TBB feature equivalent parts in custom made threading infrastructure and the transformation towards the TBB feature would not be feasible.

3. Implementation

Parallator extends the DS-8 project. It introduces the transformation using TBB into the CDT plug-in of the DS-8 project. This chapter describes the integration of the transformation towards parallelism as well as the changes to make the existing transformation to STL algorithms applicable in practice. Section 3.1.2 on the next page describes the implemented parallel library that simplified the transformation towards parallelism. Section 3.2 on page 49 covers the details of the integration to the existing plug-in and the final architecture. A detailed description of the patterns used to cover the semantic analysis is given in Section 3.3 on page 62.

3.1. Library for Parallelization

A header-only library to parallelize iterator-based loops is implemented. The header file can simply be copied to the project for usage. The motivation for such a library, the implementation details and design decision are reported here.

3.1.1. Motivation

Listing 3.1 is a good example that shows several aspects that motivated a separate library. The iteration on the *Range* object is boilerplate code that needs to be inserted for every iterator-based loop. Code generation in the IDE has always the issue of introducing names for variables. Arbitrary chosen variable names make the understanding of the generated code more difficult. The boilerplate code can be parametrized with template parameters and extracted to a *Body* functor that can be reused in other transformation. In that way, the code to generate is minimized to the call of the library function. The boilerplate code is hidden. The library is inspired by thrust, but additional TBB specific features like the selection of the grain size as well as the partitioner shall be possible to the user.

```
std::vector<WebSite> sites = getWebSites();
tbb::parallel_for(
  tbb::blocked_range<std::vector<WebSite>::iterator>(sites.begin(), sites.end())
    ,
      [=] (const tbb::blocked_range<std::vector<WebSite>::iterator>& site) {
        const std::vector<WebSite>::iterator it_end = site.end();
        for (std::vector<WebSite>::iterator it = site.begin(); it != it_end; it
            ++) {
          it->search(searchString);
        }
      });
```

Listing 3.1: Motivation for an own library – `tbb::parallel_for` usage with iterator.

3.1.2. The Library

The library features its own namespace `::ptor`. When applying a transformation the header file is copied to the working project. It is not possible to change the implementation of the algorithm provided but the library can also be used while writing new code and not only in a transformation. This section demonstrates the usage of the parallel algorithms provided by the library as well as the implementation issues considered.

Example Usage

The `ptor::for_each` algorithm is the parallel `std::for_each` equivalent and allows the user to define the grain size and the partitioner to use. Listing 3.3 shows the usage of this algorithm. It is the equivalent parallel implementation of Listing 2.7 on page 27 and also from Listing 3.2. A transformation from iterator-based loops or from a `std::for_each` call is therefore possible with minimal code generation.

```
std::vector<WebSite> sites = getWebSites();
std::for_each(sites.begin(), sites.end(), [](WebSite& site) {
  site.search(searchString);
});
```

Listing 3.2: Serial equivalent of `ptor::for_each`.

```
std::vector<WebSite> sites = getWebSites();

//without chunk control
ptor::for_each(sites.begin(), sites.end(), [](WebSite& site) {
  site.search(searchString);
})();

//with chunk control
ptor::for_each(sites.begin(), sites.end(), [](WebSite& site) {
  site.search(searchString);
})(1, tbb::simple_partitioner());
```

Listing 3.3: Example usage of `ptor::for_each` with and without chunk control.

Separation of Chunk Control

As seen in the examples of Listing 3.3 the chunk control parameters are separated from the algorithm call. This way a `ptor::for_each` call serves the same parameters as a `std::for_each` call but with possible additional parallelization specific parameters. The execution of the examples is shown in Figure 3.1 on the next page.

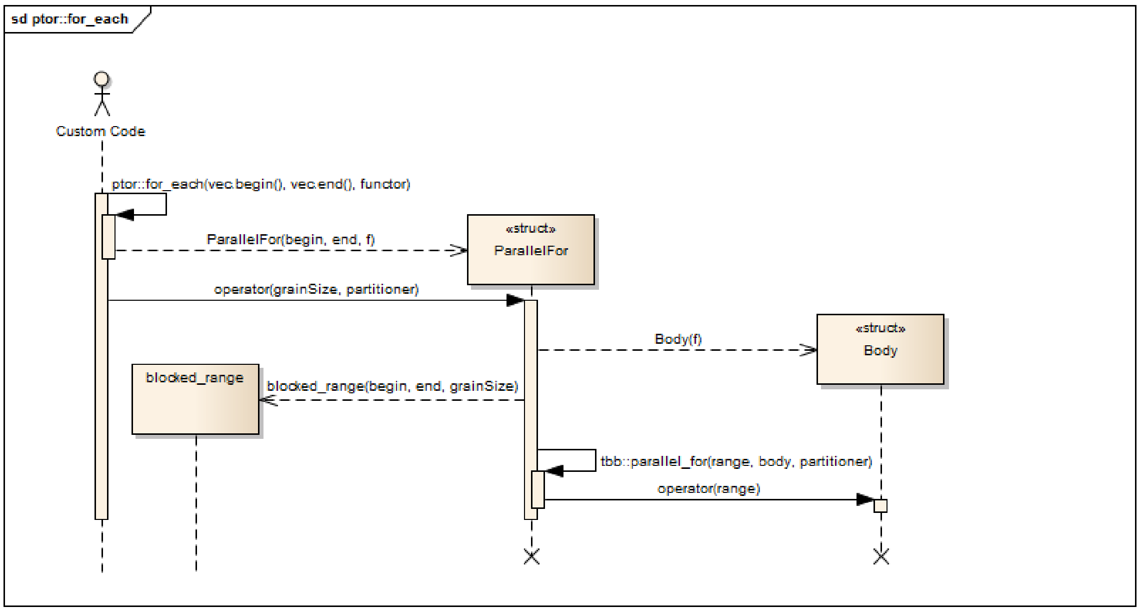

sd ptor::for_each
Custom Code
ptor::for_each(vec.begin(), vec.end(), functor)
ParallelFor(begin, end, f)
«struct» ParallelFor
operator(grainSize, partitioner)
Body(f)
«struct» Body
blocked_range
blocked_range(begin, end, grainSize)
tbb::parallel_for(range, body, partitioner)
operator(range)

Figure 3.1.: Interactions in the execution of a `ptor::for_each` call with chunk control.

The separation of the chunk control parameter is possible through the usage of the `ParallelFor` functor. In a first call the functor is created and stores copies of the iterators and the callable object provided. In a second call the returned functors overloaded call operator is called with the chunk control parameters. The `ParallelFor` functor has now all parameters available to setup the `tbb::parallel_for` call. For this the *Range* object is created with the iterators and the grain size provided. The *Body* functor is parametrized with the provided functor. In this process, `ptor::for_each` is a factory function template to deduce the template arguments for the `ParallelFor` functor. Listing 3.4 shows a simplified definition of the `ParallelFor` functor.

```
/*
 * Functor that encapsulates parameters always needed in calls to
 * tbb::parallel_for. It provides function call operator overloads
 * to parametrize the call.
 */
template <typename RandomAccessIterator, typename UnaryFunction>
struct ParallelFor {

  RandomAccessIterator begin;
  RandomAccessIterator end;
  UnaryFunction f;

  ParallelFor(RandomAccessIterator begin, RandomAccessIterator end,
      UnaryFunction f) : begin(begin), end(end), f(f) {};

  /*
   * Parallel for_each implementation using parallel_for with grain_size
   * and partitioner to specify.
   */
  template <typename Partitioner>
  void operator()(std::size_t grainSize, Partitioner partitioner) {
    tbb::parallel_for(tbb::blocked_range<RandomAccessIterator>(begin, end,
        grainSize),
        Body<tbb::blocked_range<RandomAccessIterator>, UnaryFunction>(f),
            partitioner);
  }
};
```

Listing 3.4: Implementation of the `ParallelFor` functor.

The `Body` object implements the *Body* concept for the `tbb::parallel_for` call. It is the object that is copied whenever a task runs on an available worker thread. The implementation of the `Body` object uses the `std::for_each` algorithm to apply the provided callable object on the chunk.

3.2. The Plug-In

The already implemented CDT plug-in from the DS-8 project provided a good structure to integrate transformations towards parallelism. The function calls of `ptor::for_each` (Section 3.1.2 on page 47) and `tbb::parallel_for_each` (Section 1.4.5 on page 15) show a very similar interface compared to `std::for_each`. Also the loops to detect for the transformation into parallelized calls show very similar code structure. The given separation into mainly analysis components and transformation components could be reused. Three important issues needed to be integrated into the given structure: (1) Transformation from function calls, (2) transformation from index-based / blocked range iterator-based loops and (3) analysis for parallelization issues. Another important aspect was the reuse of given transformation components, e.g. the converter from a loop body into a *bind* functor, in transformations from loops to parallelized calls. The resulting architecture that features these additional requirements is explained in this section. Section 3.2.1 shows the overview of the different components involved in the plug-in implementation as well as an outlook of how the functionality is exposed to the plug-in user. Section 3.2.2 on the following page and Section 3.2.3 on page 53 show how the analysis and the transformation components respectively are enriched, Section 3.2.4 on page 59 shows the integration details of these two modules in Eclipse.

3.2.1. Overview

Figure 3.2 on the next page shows all the plug-in components involved to transform serial code into parallelized code. The CDT Codan (Code Analysis) framework [Fou13] is used as the driver for starting the code analysis. It uses the *semantic analyzers* (Section 3.2.2 on the following page) and results in Eclipse *resource marker* for loop and call candidates for parallelization.

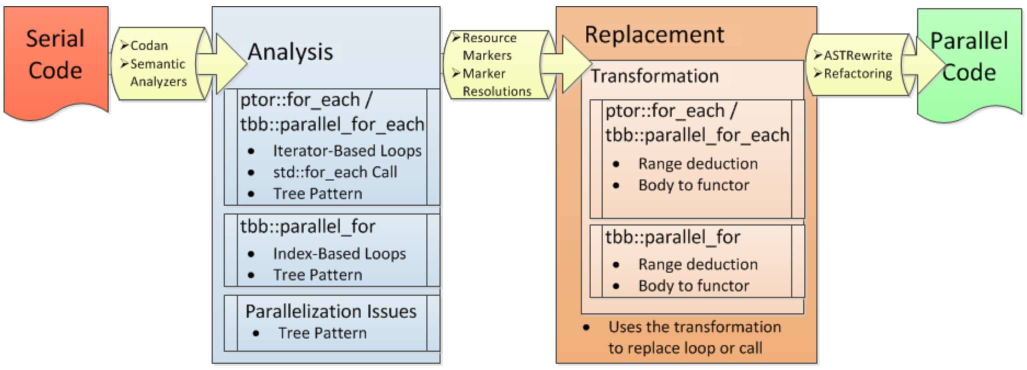
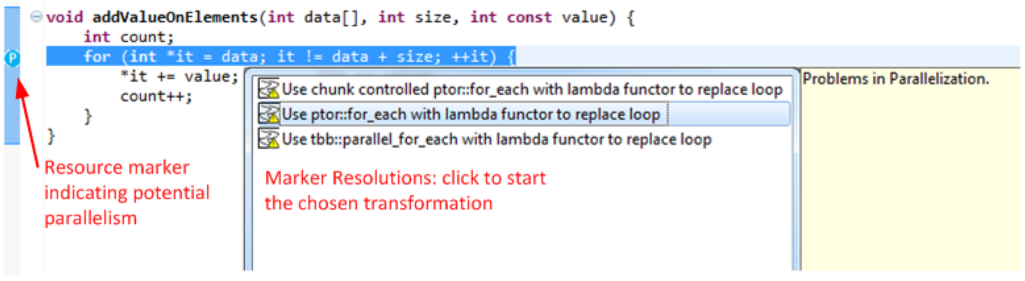


Figure 3.2.: Functionality overview.

If the user chooses to parallelize a candidate, by a click on a *resource marker*, the related *marker resolutions* are provided and show the options of how the candidate can be parallelized. Figure 3.3 shows this situation with possible warnings for parallelization. For every possible *resolution* there is a *replacement*. A *replacement* assembles components from the transformation module. If the user chooses a *resolution* the *replacement* uses the transformation components to generate the resulting call. For the call, range parameters are evaluated and the loop body is transformed into a lambda functor or *bind* functor. In this conversion of the loop body to a functor, the lambda capture evaluation, the lambda parameter modifier evaluation and a possible iterator to value type conversion take place. Eventually, if the new function call is generated successfully, the Eclipse refactoring framework is used to rewrite the source code and replace the respective loop or call.



```
void addValueOnElements(int data[], int size, int const value) {
    int count;
    for (int *it = data; it != data + size; ++it) {
        *it += value;
        count++;
    }
}
```

Use chunk controlled ptor::for_each with lambda functor to replace loop
Use ptor::for_each with lambda functor to replace loop
Use tbb::parallel_for_each with lambda functor to replace loop

Problems in Parallelization.

Resource marker indicating potential parallelism
Marker Resolutions: click to start the chosen transformation

Figure 3.3.: *Marker resolutions* to parallelize the loop.

3.2.2. Analysis

The analysis module covers all the components involved to detect loop and call candidates for parallelism and to introduce a warning in case a transformation to parallelism is not safe. This section explains the integration of the new components into the existing plug-in implementation.

Iteration Variable Requirements for Parallelism

The iteration variable type in a transformation to a `tbb::parallel_for` call using the compact notation must conform to the *Index* concept. For a transformation to `tbb::parallel_for_each` or `ptor::for_each` the iteration variable type must be an iterator that follows the constraints given by the *Value* concept. More precisely, for `tbb::parallel_for_each` an *InputIterator* is allowed but does not show performance benefits as explained in Section 1.4.5 on page 15. The difference between index-based loops compared to iterator-based loops is the type of the iteration variable and how an element in the loop body is accessed. While in iterator-based loops the sequence element is received by dereferencing the iterator, in index-based loops the sequence element is received using the subscript operator of the sequence object. The detection of the possible loops as shown in Listing 2.1 on page 25 remain the same. The already existing `IsLoopCandidate` pattern is therefore changed to be parameterizable with an `IterationPatternProvider`. The specific implementation provides the pattern to identify the iteration variable and the pattern to detect the access on the sequence element using the iteration variable. Figure 3.4 shows the classes involved to detect a *for_each* loop. The `IsForEachCandidate` pattern uses the `IsLoopCandidate` pattern to detect loops.

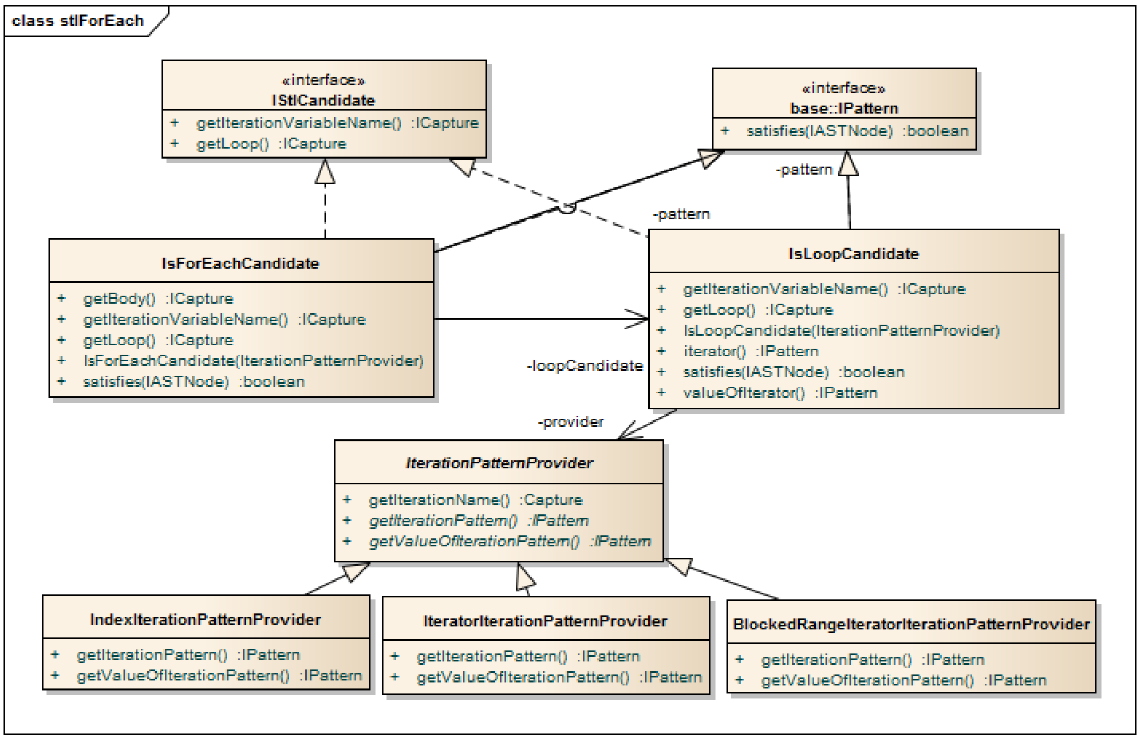

class stlForEach
«interface»
IStlCandidate
+ getIterationVariableName() :ICapture
+ getLoop() :ICapture
«interface»
base::IPattern
+ satisfies(IASTNode) :boolean
-pattern
-pattern
IsForEachCandidate
+ getBody() :ICapture
+ getIterationVariableName() :ICapture
+ getLoop() :ICapture
+ IsForEachCandidate(IterationPatternProvider)
+ satisfies(IASTNode) :boolean
IsLoopCandidate
+ getIterationVariableName() :ICapture
+ getLoop() :ICapture
+ IsLoopCandidate(IterationPatternProvider)
+ iterator() :IPattern
+ satisfies(IASTNode) :boolean
+ valueOfIterator() :IPattern
-loopCandidate
-provider
IterationPatternProvider
+ getIterationName() :Capture
+ getIterationPattern() :IPattern
+ getValueOfIterationPattern() :IPattern
IndexIterationPatternProvider
+ getIterationPattern() :IPattern
+ getValueOfIterationPattern() :IPattern
IteratorIterationPatternProvider
+ getIterationPattern() :IPattern
+ getValueOfIterationPattern() :IPattern
BlockedRangeIteratorIterationPatternProvider
+ getIterationPattern() :IPattern
+ getValueOfIterationPattern() :IPattern

Figure 3.4.: *IterationPatternProvider* are used to parametrize the *for_each* candidate pattern.

The patterns returned by the different `IterationPatternProvider` are further described in Section 3.3.4 on page 68. The criteria from Section 2.4 on page 35 that define whether a loop is safe for parallelism or not are not integrated in the shown *for_each* candidate pattern. They are incorporated during the creation of the *replacement* as described in Section 3.2.4 on page 59.

Detecting std::for_each Calls

An extensive pattern implementation covers the detection of `std::for_each` calls that can be parallelized. The integration of the pattern in the analysis module is shown in Figure 3.5. Further implementation details are reported in Section 3.3.3 on page 66.

Semantic Analyzer

The whole analysis module is abstracted behind the `ISemanticAnalyzer` interface. Figure 3.5 shows the different implementations for the different STL or TBB *for_each* algorithm candidates. The function `getOffendingNodes()` provides the AST nodes that match the specific pattern used in a *SemanticAnalyzer*. For the integration of TBB algorithms, the `ForEachSemanticAnalyzer` is abstracted and specific implementations for index-based and iteration-based loop candidates are implemented. All these implementations use `IsForEachCandidate` to detect the loop but provide the respective `IterationPatternProvider` to distinguish index based and iterator based loops as explained above.

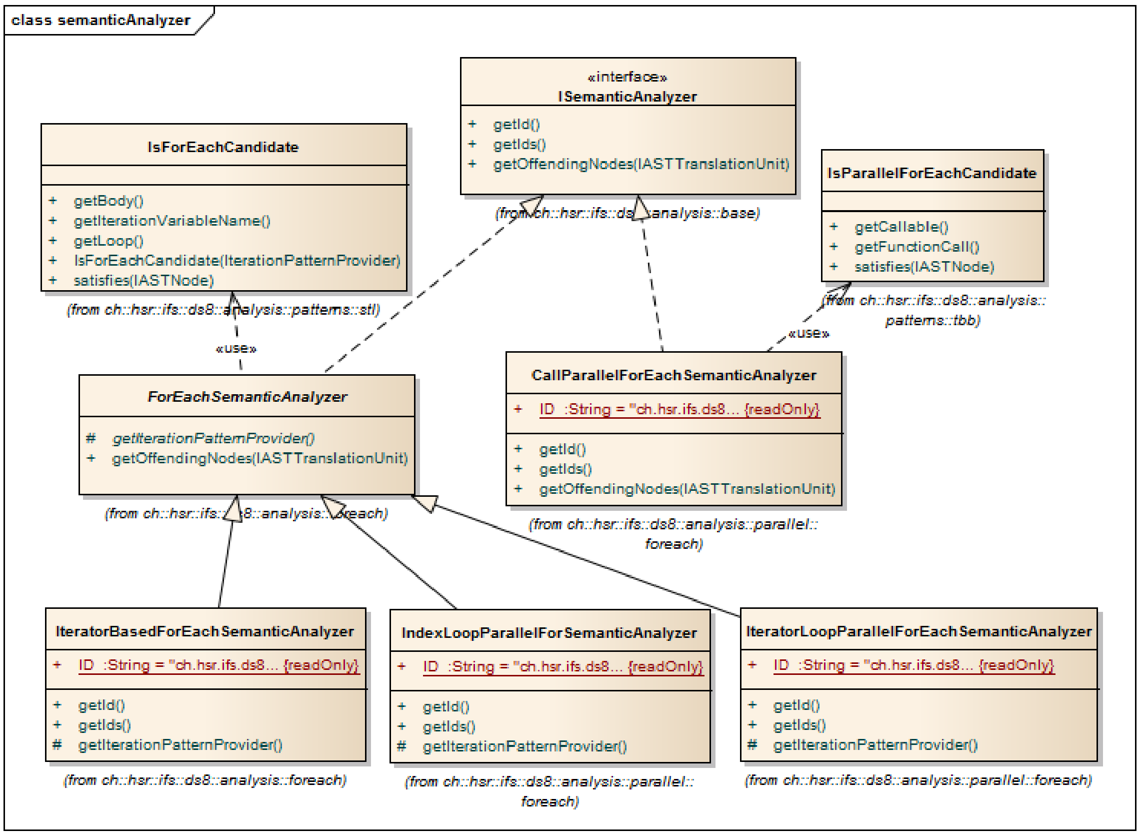

class semanticAnalyzer
«interface»
ISemanticAnalyzer
+ getId()
+ getIds()
+ getOffendingNodes(IASTTranslationUnit)
(from ch::hsr::ifs::ds8::analysis::base)
IsForEachCandidate
+ getBody()
+ getIterationVariableName()
+ getLoop()
+ IsForEachCandidate(IterationPatternProvider)
+ satisfies(IASTNode)
(from ch::hsr::ifs::ds8::analysis::patterns::stl)
IsParallelForEachCandidate
+ getCallable()
+ getFunctionCall()
+ satisfies(IASTNode)
(from ch::hsr::ifs::ds8::analysis::patterns::tbb)
«use»
«use»
ForEachSemanticAnalyzer
# getIterationPatternProvider()
+ getOffendingNodes(IASTTranslationUnit)
(from ch::hsr::ifs::ds8::analysis::foreach)
CallParallelForEachSemanticAnalyzer
+ ID :String = "ch.hsr.ifs.ds8... {readOnly}
+ getId()
+ getIds()
+ getOffendingNodes(IASTTranslationUnit)
(from ch::hsr::ifs::ds8::analysis::parallel::foreach)
IteratorBasedForEachSemanticAnalyzer
+ ID :String = "ch.hsr.ifs.ds8... {readOnly}
+ getId()
+ getIds()
# getIterationPatternProvider()
(from ch::hsr::ifs::ds8::analysis::foreach)
IndexLoopParallelForSemanticAnalyzer
+ ID :String = "ch.hsr.ifs.ds8... {readOnly}
+ getId()
+ getIds()
# getIterationPatternProvider()
(from ch::hsr::ifs::ds8::analysis::parallel::foreach)
IteratorLoopParallelForEachSemanticAnalyzer
+ ID :String = "ch.hsr.ifs.ds8... {readOnly}
+ getId()
+ getIds()
# getIterationPatternProvider()
(from ch::hsr::ifs::ds8::analysis::parallel::foreach)

Figure 3.5.: Different implementations of `ISemanticAnalyzer` to analyze the code for possible *for_each* candidates.

The `ParallelForEachSemanticAnalyzer` is introduced for the recognition of `std::for_each` calls. It uses the `IsParallelForEachCandidate` pattern as introduced above to find the calls. Each *SemanticAnalyzer* provides its own indication string (the ID). This ID is set to the *resource* *marker* and later used to distinguish the different steps executed in generating the resolutions. This is explained further in Section 3.2.4 on page 59.

Generate Warnings for Parallelization

The detection of parallelism issues are implemented using the tree pattern matching algorithm. To generate hints for parallelism issues a pattern can be wrapped by so called message patterns. The message patterns define which hint and when the hint is collected. The hints are collected in a message context provided to a message pattern. If a `RelaxedMessagePattern` is used the hint is collected in case the wrapped pattern does not match, additionally the message pattern relaxes the pattern match and defines the wrapped pattern to match for every node. If a `SatisfyingMessagePattern` is used, the verification result of the wrapped pattern is not changed and the specific hint is collected in case the wrapped pattern matches. Because the message patterns are patterns themselves they can be used in combination with any other pattern and collect hints during verification. Listing 3.5 show the sample usage of the `SatisfyingMessagePattern`.

```
MessageContext context = new MessageContext();
IPattern pattern = getPattern();
SatisfyingMessagePattern messagePattern = new SatisfyingMessagePattern(pattern,
    context) {

  @Override
  protected String getPatternMessage() { //override to specify message
    return "specific message";
  }
};
messagePattern.satisfies(getBody()); //generate messages
message = context.getMessages(); //get messages
```

Listing 3.5: Sample code to generate messages in the plug-in implementation.

The shown process decouples a pattern from the message that should be generated in case of a match or when the wrapped pattern does not match. To see when the messages for parallelism are generated consider Section 3.2.4 on page 59, the patterns to detect parallelism issues are described in Section 3.3.2 on page 64.

3.2.3. Replacement and Transformation

The analysis chapter has shown different source code constructs that resemble a candidate for parallelism. Although different in source, they can result in an equivalent call. An iterator-based loop or an `std::for_each` call can both result in a `ptor::for_each` or `tbb::parallel_for_each` call. Another situation that bring up the feeling for extensive reuse of different components is the conversion of the loop body that is used in transformation towards STL algorithms and TBB algorithms. In the following sections the components that assemble a *replacement* are explained as well as their collaboration to generate the parallelized call including namespace handling, header file includes and the integration of the parallelization library.

Components Overview

Figure 3.6 on the next page show the packages involved in a *replacement* and their dependencies.

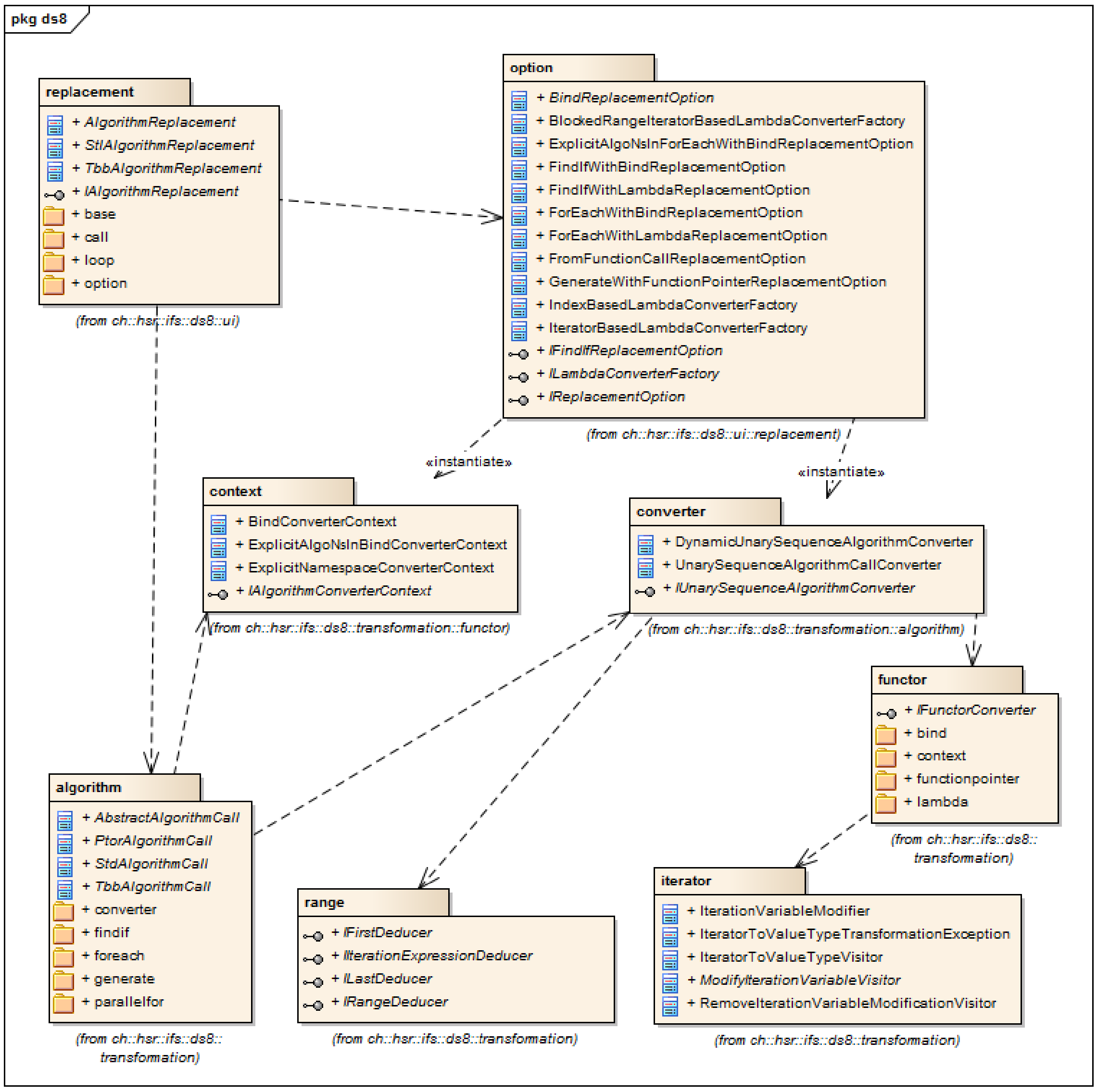

pkg ds8
replacement
+ AlgorithmReplacement
+ StlAlgorithmReplacement
+ TbbAlgorithmReplacement
+ IAlgorithmReplacement
+ base
+ call
+ loop
+ option
(from ch::hsr::ifs::ds8::ui)
option
+ BindReplacementOption
+ BlockedRangeIteratorBasedLambdaConverterFactory
+ ExplicitAlgoNsInForEachWithBindReplacementOption
+ FindIfWithBindReplacementOption
+ FindIfWithLambdaReplacementOption
+ ForEachWithBindReplacementOption
+ ForEachWithLambdaReplacementOption
+ FromFunctionCallReplacementOption
+ GenerateWithFunctionPointerReplacementOption
+ IndexBasedLambdaConverterFactory
+ IteratorBasedLambdaConverterFactory
+ IFindIfReplacementOption
+ ILambdaConverterFactory
+ IReplacementOption
(from ch::hsr::ifs::ds8::ui::replacement)
«instantiate»
«instantiate»
context
+ BindConverterContext
+ ExplicitAlgoNsInBindConverterContext
+ ExplicitNamespaceConverterContext
+ IAlgorithmConverterContext
(from ch::hsr::ifs::ds8::transformation::functor)
converter
+ DynamicUnarySequenceAlgorithmConverter
+ UnarySequenceAlgorithmCallConverter
+ IUnarySequenceAlgorithmConverter
(from ch::hsr::ifs::ds8::transformation::algorithm)
functor
+ IFunctorConverter
+ bind
+ context
+ functionpointer
+ lambda
(from ch::hsr::ifs::ds8::transformation)
algorithm
+ AbstractAlgorithmCall
+ PtorAlgorithmCall
+ StdAlgorithmCall
+ TbbAlgorithmCall
+ converter
+ findif
+ foreach
+ generate
+ parallelfor
(from ch::hsr::ifs::ds8::transformation)
range
+ IFirstDeducer
+ IIterationExpressionDeducer
+ ILastDeducer
+ IRangeDeducer
(from ch::hsr::ifs::ds8::transformation)
iterator
+ IterationVariableModifier
+ IteratorToValueTypeTransformationException
+ IteratorToValueTypeVisitor
+ ModifyIterationVariableVisitor
+ RemoveIterationVariableModificationVisitor
(from ch::hsr::ifs::ds8::transformation)

Figure 3.6.: Different components and its dependencies that assemble a *replacement*.

The different packages and their responsibility are explained in the following sections.

ch.hsr.ifs.ds8.transformation.iterator

This package contains the functionality to transform iterator-based to value-type based access in the conversion of a loop body to a bind functor or lambda functor. This functionality was already available. The transformation is not needed for index-based loops. But still, a modification in the loop body transformed to a lambda body is needed. The iteration increment in the body of a *while* loop must be removed. An `IterationVariableModifier` provides a generic `modifyIterationVariableInNode` function that accepts any `IASTNode`, performs the modification and returns the node again. The modification is implemented by visitors [GHJ95]. While the `IteratorToValueTypeVisitor` traverses the given node and replaces any child nodes identified as iterator accesses by their value type equivalent and also any modification of the iteration variable, the `RemoveIterationVariableModificationVisitor` performs only the remove of the iteration variable modification while traversing.

ch.hsr.ifs.ds8.transformation.functor

This package contains classes responsible for creating lambda or bind functors out of the loop body. The transformation uses the iterator-transformation package for the modification around the iteration variable as described above. For lambda functors the modified loop body statement is then analyzed to evaluate the lambda captures and the declaration modifier of the lambda parameter. See Section 3.3.1 on page 62 for the implementation of this evaluation. With the evaluated information, a lambda expression taking an argument of the value type of the iteration expression (iterator-based loop) is created that wraps the body statement. Every implementation of a loop body to a callable object implements the `IFunctorConverter` interface. The only function to implement is `getFunctor` that is expected to return the resulting callable object. Figure 3.7 shows the extension to convert a index-based loop body to the lambda functor.

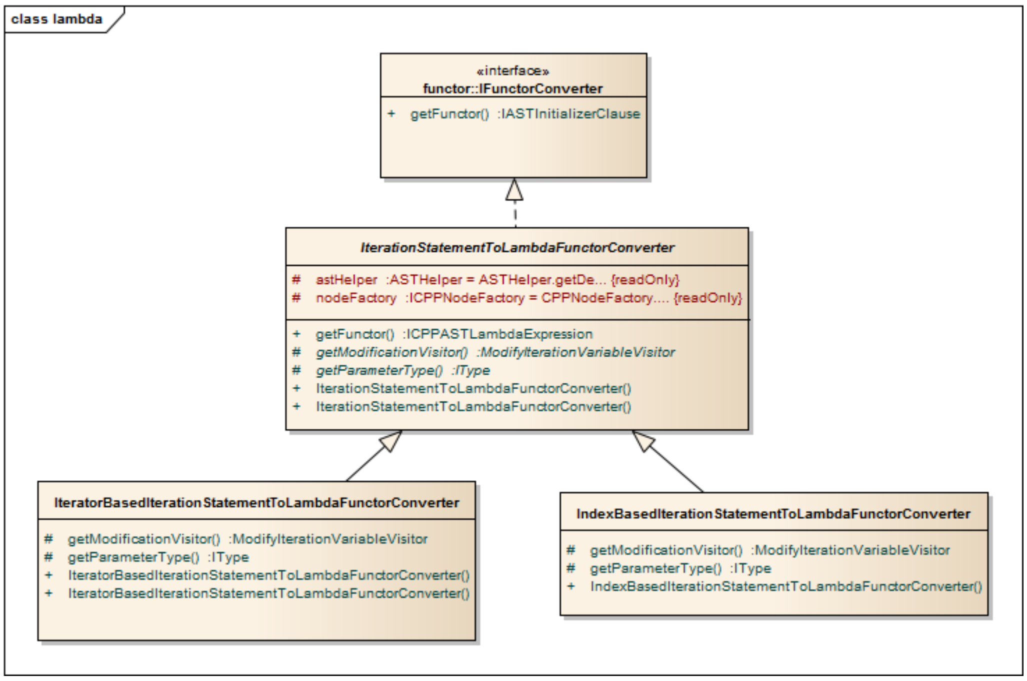

class lambda
«interface»
functor::IFunctorConverter
+ getFunctor() :IASTInitializerClause
IterationStatementToLambdaFunctorConverter
# astHelper :ASTHelper = ASTHelper.getDe... {readOnly}
# nodeFactory :ICPPNodeFactory = CPPNodeFactory.... {readOnly}
+ getFunctor() :ICPPASTLambdaExpression
# getModificationVisitor() :ModifyIterationVariableVisitor
# getParameterType() :IType
+ IterationStatementToLambdaFunctorConverter()
+ IterationStatementToLambdaFunctorConverter()
IteratorBasedIterationStatementToLambdaFunctorConverter
# getModificationVisitor() :ModifyIterationVariableVisitor
# getParameterType() :IType
+ IteratorBasedIterationStatementToLambdaFunctorConverter()
+ IteratorBasedIterationStatementToLambdaFunctorConverter()
IndexBasedIterationStatementToLambdaFunctorConverter
# getModificationVisitor() :ModifyIterationVariableVisitor
# getParameterType() :IType
+ IndexBasedIterationStatementToLambdaFunctorConverter()

Figure 3.7.: Different loop body to lambda functor converters.

They differ in the `ModifyIterationVariableVisitor` used and in the lambda parameter type evaluation. As an example the transformation of a *while* loop body to a lambda is shown in Listing 3.6 on the following page and Listing 3.7 on the next page. Matching code segments in the original and the transformed block are highlighted with the same color.

```
int i = 0;
while (i != size) {
  cout << data[i] << endl;
  i++;
}
```

Listing 3.6: Original compound statement to transform to a lambda.

```
[data](const int& i) {
  cout << data[i] << endl;
}
```

Listing 3.7: Resulting lambda expression.

During this project the former implemented transformation towards bind functors using `bind1st` and `bind2nd` are removed because these bind implementation is deprecated since C++11 (Meeting decision: 13.11.2012). The also available transformation towards the formerly called *TR1* bind implementation is enriched with namespace handling and header file include handling as explained below.

ch.hsr.ifs.ds8.transformation.range

Loop range deduction is an important task in the transformation towards STL and TBB algorithm. It handles the extraction of AST Nodes representing the iteration start and end value as well as the iteration variable. These nodes are later used to create the algorithm call. No changes in this package were needed as the analysis is the same for both loop variants, iterator-based and index-based loops. The `IRangeDeducer` interface extends the interfaces `IFirstDeducer`, `ILastDeducer` and `IIterationExpressionDeducer` from Figure 3.6 on page 54 that are responsible for deducing the iteration start, iteration end node and iteration variable respectively. There is no direct implementation for a *RangeDeducer* but with the various implementations of *FirstDeducers* and *LastDeducers* a *RangeDeducer* can be implemented.

ch.hsr.ifs.ds8.transformation.algorithm.converter

An *AlgorithmConverter* is able to provide all arguments necessary to create a call of an STL algorithm or a TBB algorithm. All implemented transformations operate on one sequence and have the same declaration as the formal parameters. Therefore the general interface and its name: `IUnarySequenceAlgorithmConverter`. It is an extension of the interfaces `IRangeDeduction` and `IFunctorConverter`. The generic implementation of this interface, the `DynamicUnarySequenceAlgorithmConverter` class, can be configured with a *FirstDeducer*, a *LastDeducer*, a *IterationExpressionDeducer* from the range-transformation package and a *FunctorConverter* from the functor-transformation package. This class can therefore serve as *AlgorithmConverter* for a loop transformation to any unary sequence algorithm. In the transformation from a `std::for_each` call no real range deduction must be done. This fact is considered with the implementation of `UnarySequenceAlgorithmCallConverter` that simply extracts the range and functor nodes from the function call expression representing the `std::for_each` call.

ch.hsr.ifs.ds8.transformation.functor.context

A *FunctorContext* is introduced to collect and provide information during a transformation. It is responsible for ADL (argument dependent lookup) respecting namespace handling as well as for collecting the header includes for a generated algorithm call. The context is extensively used in *replacements* with argument binding as the context abstracts the handling for `std::bind` and `boost::bind` [Boo13]. This can be shown in an example. Listing 3.8 represents a loop convertible to a *find_if* call. The resulting call with argument binding using boost is shown in Listing 3.9 and using `std::bind` in Listing 3.10.

```
void findIfCallConditionInBody(unsigned char* line) {
  const unsigned char *it;
  for (it = line; it != line + 4; ++it) {
    if (*it < 's') {
      break;
    }
  }
}
```

Listing 3.8: Original compound statement to transform to a lambda.

```
#include "boost/bind.hpp"
#include <functional>
#include <algorithm>

void findIfCallConditionInBody(unsigned char* line) {
  const unsigned char *it;
  it = find_if(line, line + 4,
      bind(std::less<const unsigned char>(), _1, cref('s')));
}
```

Listing 3.9: Resulting `boost::bind` expression.

```
#include <functional>
#include <algorithm>

void findIfCallConditionInBody(unsigned char* line) {
  const unsigned char *it;
  using std::placeholders::_1;
  it = find_if(line, line + 4,
      bind(less<const unsigned char>(), _1, cref('s')));
}
```

Listing 3.10: Resulting `std::bind` expression.

It is common in the usage with `std::bind` to declare a placeholder before the application of the argument binding. Because of ADL, the namespace prefix for the *find_if* call and for the predefined binary functor `less` can be omitted. This is not the case in argument binding with boost. The namespace prefix must be specified for the binary functor `less` but as it is an argument to the *find_if* call the namespace prefix can also be omitted for the algorithm call. Recognize also the different header includes needed and that the placeholders and the `bind` function in boost are not in a separate namespace. To handle these variations, the context is provided to the *FunctorConverters* that create calls to `bind`. If these converters are used to create a `bind` call, the specific information like header file to include or whether the call should be created with a namespace prefix, is set to the context.

ch.hsr.ifs.ds8.transformation.algorithm

An *AlgorithmCall* class assembles a specific algorithm call. It needs a *FunctorContext* to know whether a namespace prefix for the algorithm call is needed and to add the algorithm specific header include. An *AlgorithmConverter* provides the parameter nodes for the call. Remember, the parameter nodes can be gathered either by *Deducers* in case of a transformation from a loop, or by the `UnarySequenceAlgorithmCallConverter` in case of a transformation from a function call.

ch.hsr.ifs.ds8.ui.replacement.option

While the class hierarchy for a *replacement* is chosen towards a unification of STL replacements and TBB replacements, the *replacement options* show a class hierarchy for the unification of the different options to transform a loop body. Figure 3.8 shows some replacement options implemented. A replacement option is responsible for providing the replacement specific *AlgorithmConverter* and *FunctorContext*. An implementation of a `ILambdaConverterFactory` is used to handle the different iteration variable type criteria and provides methods to create the correct *FunctorConverter*.

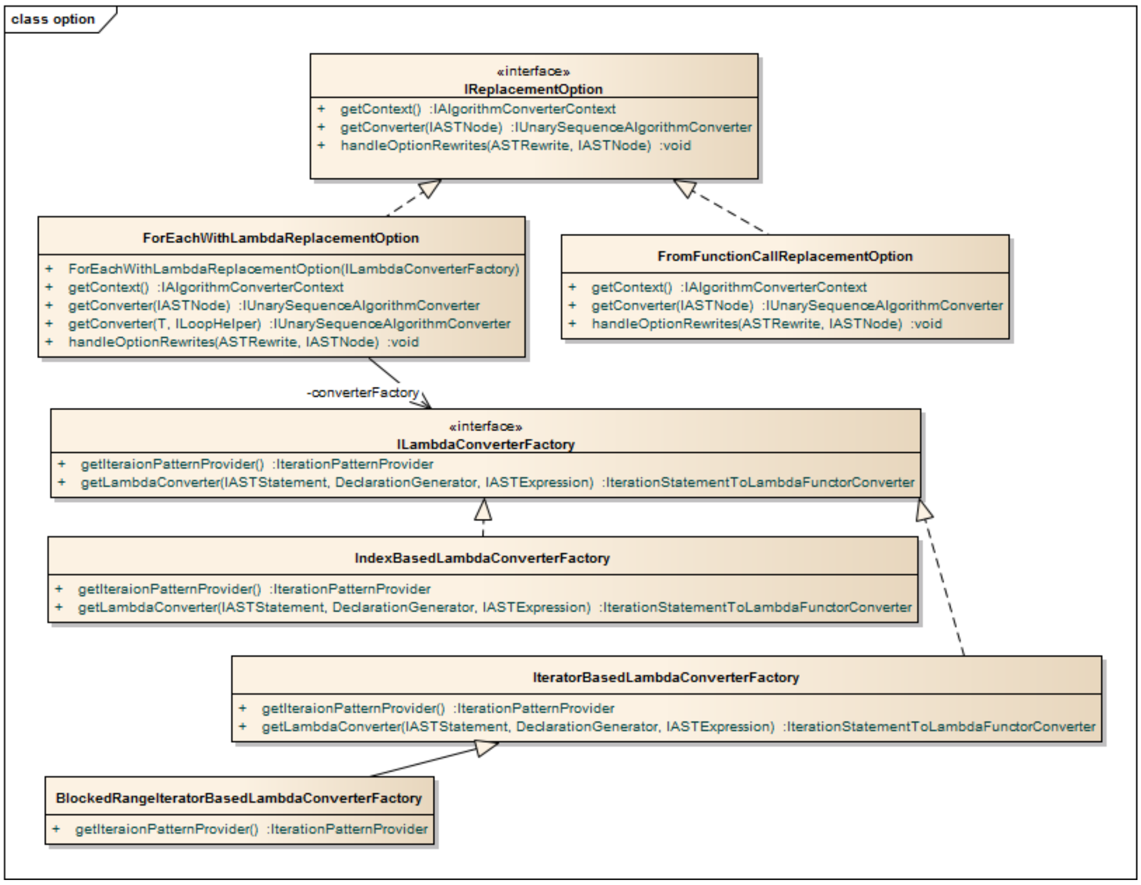

class option
«interface»
IReplacementOption
+ getContext() :IAlgorithmConverterContext
+ getConverter(IASTNode) :IUnarySequenceAlgorithmConverter
+ handleOptionRewrites(ASTRewrite, IASTNode) :void
ForEachWithLambdaReplacementOption
+ ForEachWithLambdaReplacementOption(ILambdaConverterFactory)
+ getContext() :IAlgorithmConverterContext
+ getConverter(IASTNode) :IUnarySequenceAlgorithmConverter
+ getConverter(T, ILoopHelper) :IUnarySequenceAlgorithmConverter
+ handleOptionRewrites(ASTRewrite, IASTNode) :void
FromFunctionCallReplacementOption
+ getContext() :IAlgorithmConverterContext
+ getConverter(IASTNode) :IUnarySequenceAlgorithmConverter
+ handleOptionRewrites(ASTRewrite, IASTNode) :void
-converterFactory
«interface»
ILambdaConverterFactory
+ getIteraionPatternProvider() :IterationPatternProvider
+ getLambdaConverter(IASTStatement, DeclarationGenerator, IASTExpression) :IterationStatementToLambdaFunctorConverter
IndexBasedLambdaConverterFactory
+ getIteraionPatternProvider() :IterationPatternProvider
+ getLambdaConverter(IASTStatement, DeclarationGenerator, IASTExpression) :IterationStatementToLambdaFunctorConverter
IteratorBasedLambdaConverterFactory
+ getIteraionPatternProvider() :IterationPatternProvider
+ getLambdaConverter(IASTStatement, DeclarationGenerator, IASTExpression) :IterationStatementToLambdaFunctorConverter
BlockedRangeIteratorBasedLambdaConverterFactory
+ getIteraionPatternProvider() :IterationPatternProvider

Figure 3.8.: Different *replacement options* implemented.

ch.hsr.ifs.ds8.ui.replacement

Every *resolution* provided to the user has its own *algorithm replacement* implementation. An algorithm replacement is responsible for building up the Eclipse *Change* object with all changes needed to apply to the file. The changes compose of the algorithm call, the header includes and possible placeholder declarations in case of a loop body transformation towards argument binding. In collaboration with a *replacement option* and an *AlgorithmCall* the AST nodes are created. The *ASTRewriter* is then used to modify the AST and create the *Change* object. After applying the change, the Eclipse linked mode is entered in case of a transformation towards a `ptor::for_each` call. This enables to set the grain size and the partitioner.

3.2.4. Codan Integration

In Codan, *checkers* can be implemented to create markers indicating a *problem* at a specific source code position. An checker can be registered and the different problems reportable by a checker are configurable to run at different times, e.g. when a file is saved. Problems can also have generic preferences and belong to a category. The resolutions can also be registered and defined to which problem they belong. An alternative to register the resolutions is a resolution generator. Generators can be registered and are then used for generating the resolutions of different problems.

Checker Implementation

The plug-in implementation features three different checkers that are responsible for reporting the possible transformations. One checker reports loops convertible to STL algorithm, another reports loops convertible to TBB algorithms and the third reports *for_each* calls transformable to TBB algorithms. This differentiation is needed as checkers can only report problems of the same category and loop transformation are configured to have generic preferences which is not needed for transformation from function calls. The preferences in loop transformation (see Figure B.9 on page 87) provide the configuration to choose between boost argument binding or the standard argument binding. Figure 3.9 on the following page shows the sequence executed to report a problem. The checkers use the *SemanticAnalyzers* described in Section 3.2.2 on page 50.

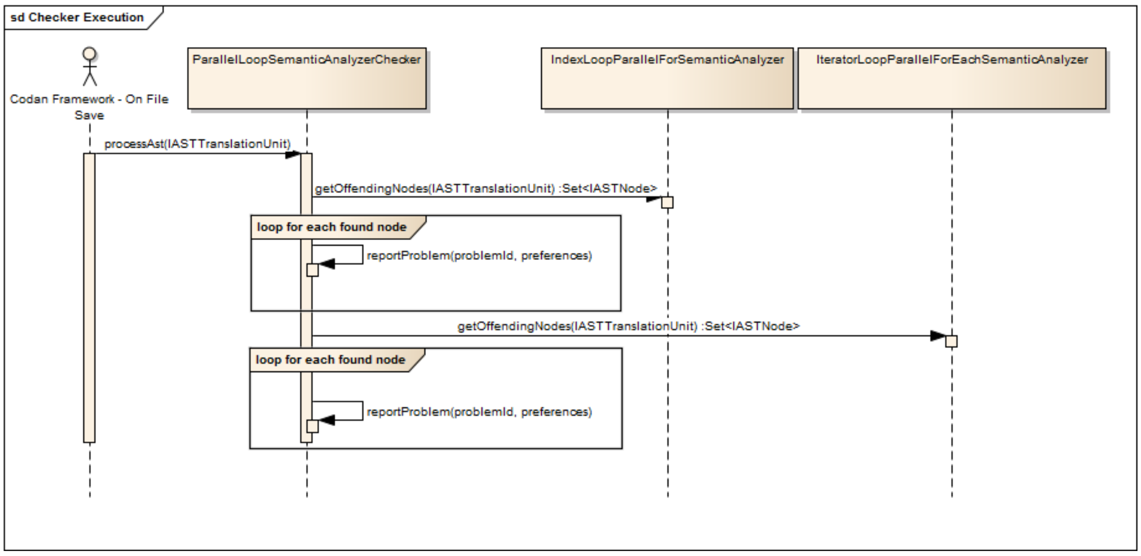

sd Checker Execution
Codan Framework - On File Save
ParallelLoopSemanticAnalyzerChecker
IndexLoopParallelForSemanticAnalyzer
IteratorLoopParallelForEachSemanticAnalyzer
processAst(IASTTranslationUnit)
getOffendingNodes(IASTTranslationUnit) :Set<IASTNode>
loop for each found node
reportProblem(problemId, preferences)
getOffendingNodes(IASTTranslationUnit) :Set<IASTNode>
loop for each found node
reportProblem(problemId, preferences)

Figure 3.9.: Execution of a checker to report a marker representing parallelizable loops.

Resolution Generator Implementation

A specific marker can have more than one resolution. But also not every possible resolution for a loop to parallelize is applicable. The transformation using argument binding is only applicable if the loop body satisfies certain criteria. The already available implementation of the DS-8 project handles this issue by a resolution generator to create only the applicable resolutions for the found loop. The resolution generator uses a specific aggregator to create the resolutions. Using the id of the reported marker a specific aggregator is determined which in turn is responsible for creating all applicable resolutions. An aggregator is also responsible for generating warning hints for the resolution to apply. The created resolutions are adapters [GHJ95] to a specific *replacement* that is executed if the user chooses to do so. Figure 3.10 on the following page shows the process to create the resolutions to transform loops to parallelism.

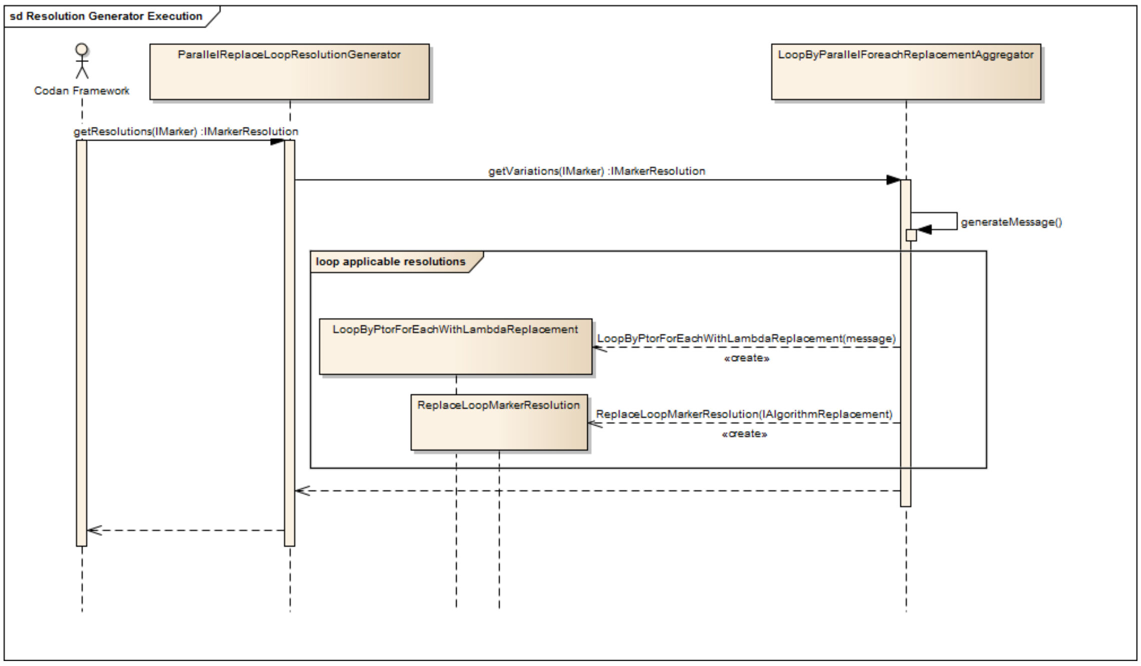

sd Resolution Generator Execution
Codan Framework
ParallelReplaceLoopResolutionGenerator
LoopByParallelForeachReplacementAggregator
getResolutions(IMarker) :IMarkerResolution
getVariations(IMarker) :IMarkerResolution
generateMessage()
loop applicable resolutions
LoopByPtorForEachWithLambdaReplacement
LoopByPtorForEachWithLambdaReplacement(message)
«create»
ReplaceLoopMarkerResolution
ReplaceLoopMarkerResolution(IAlgorithmReplacement)
«create»

Figure 3.10.: Execution of a resolution generator to create applicable resolutions.

For some reasons, the `getResolutions(marker)` method is triggered many times by the Codan framework. If the whole process is executed several times, the performance is bad. This issue is solved by storing the resolutions for a specific marker in the generator and return a stored resolution if one exists. If a marker is removed, the corresponding resolutions are removed too.

An alternative implementation would have been to register all resolutions in the `plugin.xml` and state to which *problem* they belong. A resolution derived from `AbstractCodanCMarkerResolution` could then override the `isApplicable(marker)` method to evaluate e.g. whether an argument binding resolution is possible. In such an implementation, every possible resolution is instantiated only once. The same resolution object is used to execute the actions for the same problem id at different positions in the code. The warnings in parallelization are specific to the source code. To have a specific warning in the resolution, the warning could then be passed in the open argument list of a marker. The already existing design is kept as it worked and provides a simple possibility for context specific information in resolutions.

3.2.5. Used Extension Points

The extension points used to integrate the plug-in are listed here.

org.eclipse.cdt.codan.core.checkers

Registration of the CodeAn checkers with the problem they report and the problem category assignment.

org.eclipse.ui.ide.markerResolution

Registration of the marker generator classes used to generate the resolutions.

org.eclipse.core.resources.markers

Registration of own marker types. They are needed to change the appearance of reported transformation candidates. Two marker types are registered. One for STL algorithms and one for transformations towards parallelism. Both are derived from `org.eclipse.cdt.codan.core.codanProblem`.

org.eclipse.ui.editors.markerAnnotationSpecification

The appearance definition of the different marker types. Different properties as the icon of the reported *problem* with a specific marker type are specified. A marker annotation specifies also that a marker type used for parallelism is layered on top of the marker type for STL algorithms.

org.eclipse.ui.editors.annotationTypes

This is the mapping from marker annotations to marker types to define which marker type has which appearance.

3.3. Pattern for Semantic Analysis

The semantic analysis described in Section 2.3 on page 33 and Section 2.4 on page 35 is implemented using the available tree pattern matching algorithm. The new patterns introduced are explained in this section. Consider [Kes10] for further explanation of other extensive patterns, e.g. the `IsForEachCandidate` pattern.

3.3.1. Semantic Analysis Pattern

The implementation of the pattern for the semantic analysis related to parallelism makes extensive use of code analysis functions provided by the CDT plug-in. These patterns are explained here.

IsImplicitMemberFunctionCall

An implicit member function call must be recognized in order to evaluate the *this*-pointer as value capture. The pattern matches a function call, extracts the function name and creates a binding for this name using the `CPPVisitor` from CDT. A binding represents the semantics of a name in the AST. In case of a `ICPPMember` binding the function is identified as a member function call.

IsInOuterScope

This pattern verifies whether a variable is already known in an outer scope. An outer scope is relative to an *inner* scope. The node representing the inner scope must be specified for this pattern. The other parameter is a capture pattern containing the variable that is verified for its existence in the outer scope. The pattern searches the AST node representing the outer scope. It must be a compound statement. Each compound statement node has a reference to a *Scope* object. Scopes can be used to look-up names and contain the variables introduced. If the variable is found using the *Scope* object, the variable is already defined. Note that the scope objects are hierarchical. In a look-up, also the hierarchically higher ordered scopes are considered. Listing 3.11 marks inner and outer scope with the nodes contained in the scopes.

```cpp
void addValueOnElements(int data[], int size, int value) {
  //function body is outer CompoundStatement with Scope object: count, data, value, size
  int count = 0;
  for (int i = 0; i != size; ++i) {
    //loop body is the inner CompoundStatement with Scope object: <empty>
    data[i] += value;
    count++;
  }
}
```

Listing 3.11: Example showing variables introduced in inner and outer scope.

IsWriteAccessOnVariable

The occurrence of a write to a variable is recognized if the variable is an immediate left child of a binary expression with an assignment operator, or the variable is involved in an increment or decrement operation either prefix or postfix. In the pattern implementation the analysis of a write access is done upwards in the AST. The verification is more like whether the variable to test is dominated immediately by a binary expression whose left child is the variable to test for a write access. This approach enables a simple application for a write access verification on a given node representing a variable. Line 1 and 2 of Listing 3.12 show possible situations of write access to variable `var`.

```cpp
1 var++; var = i = 5; //writes on var
2 var.field++; var.field = 9; //writes on var in field modification
3 var.functionCall(); functionCall(var); //potential write on var in function call
```

Listing 3.12: Recognized writes and potential writes on variable.

The implementation does also identify a variable in a write access if the variable is either a parameter in a function call or a member function is called on the variable. See line 3 in Listing 3.12. This test is made to cover a potential write on the variable, as the tree pattern matching algorithm does not analyze the called function. Still, the implementation is upwards and verifies whether the variable to test is dominated by a function call expression. This misclassifies `value` in Listing 3.13 on the following page as potential write because the variable is dominated by the function call `std::for_each`. This problem is solved by a *scopeStopper* pattern. The function call that dominates the variable to test must be dominated by the scope stopper. For the example, the scope stopper must match a node that is dominated by the `std::for_each` call.

```
std::vector<int> vec = getData();
int value = 10;
std::for_each(vec.begin(), vec.end(), [value] (int it) {
  it += value;
});
```

Listing 3.13: Need for a scope stopper in `IsWriteOnAccessVariable`.

The implementation as described does not observe operator overloading to identify operators with write access.

IsField

A field must be recognized in order to evaluate the *this*-pointer as value capture. The pattern uses the `CPPVisitor` from CDT to create a binding. In case of a `IField` binding, the named AST node is evaluated as a field.

3.3.2. Pattern for Parallelism Analysis

The described patterns above are combined to define patterns that can be used to verify whether a lambda body or a loop body is safe for parallelism. In this process, shared objects must be identified. The detection of a non field shared object for a lambda can be done by an analysis of the lambda capture, for a loop the outer scope of the body must be analyzed. Common is the detection of a field (always shared) in either the lambda and the loop body. Also the iteration variable must be detected as it, or the element accessed in the sequence respectively, is in most cases not a shared object. At the time the loop is verified for safety in parallelism the iteration variable is already deduced. The deduction of the iteration variable in a lambda expression is easier and is simply the lambda parameter. These different and common aspects in analyzing a loop body or a lambda body for parallelization led to the patterns shown in Figure 3.11 on the next page.

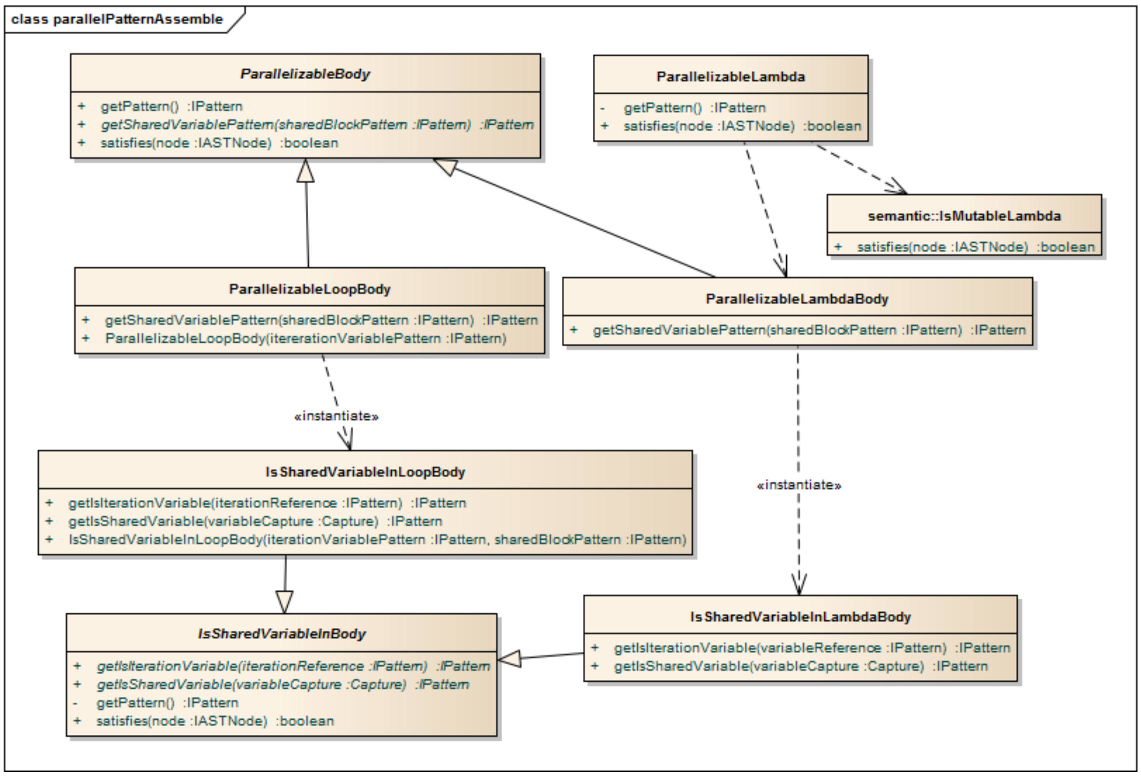

class parallelPatternAssemble
ParallelizableBody
+ getPattern() :IPattern
+ getSharedVariablePattern(sharedBlockPattern :IPattern) :IPattern
+ satisfies(node :IASTNode) :boolean
ParallelizableLambda
- getPattern() :IPattern
+ satisfies(node :IASTNode) :boolean
semantic::IsMutableLambda
+ satisfies(node :IASTNode) :boolean
ParallelizableLoopBody
+ getSharedVariablePattern(sharedBlockPattern :IPattern) :IPattern
+ ParallelizableLoopBody(itererationVariablePattern :IPattern)
ParallelizableLambdaBody
+ getSharedVariablePattern(sharedBlockPattern :IPattern) :IPattern
«instantiate»
IsSharedVariableInLoopBody
+ getIsIterationVariable(iterationReference :IPattern) :IPattern
+ getIsSharedVariable(variableCapture :Capture) :IPattern
+ IsSharedVariableInLoopBody(iterationVariablePattern :IPattern, sharedBlockPattern :IPattern)
«instantiate»
IsSharedVariableInBody
+ getIsIterationVariable(iterationReference :IPattern) :IPattern
+ getIsSharedVariable(variableCapture :Capture) :IPattern
- getPattern() :IPattern
+ satisfies(node :IASTNode) :boolean
IsSharedVariableInLambdaBody
+ getIsIterationVariable(variableReference :IPattern) :IPattern
+ getIsSharedVariable(variableCapture :Capture) :IPattern

Figure 3.11.: Different patterns involved in verification of a loop or lambda to be safe for parallelism.

The `ParallelizableBody` assembles a pattern that is responsible for traversing the body and identify shared variables with write access. Each variable is potentially shared. Whether the variable is actually shared is identified using the other shown patterns. An identified shared variable is then verified for write access. The `IsSharedVariableInLoopBody` delivers the `IsInOuterScope` pattern for shared variable identification, the `IsSharedVariableInLambdaBody` delivers a pattern assembly to search for a variable in the lambda capture equally named to a potential shared variable. The `ParallelizableLoopBody` needs an iteration variable identification pattern in the constructor, the `ParallelizableLambdaBody` receives the iteration variable identification pattern from the `IsSharedVariableInLambdaBody`. It is a pattern that searches a variable in the lambda parameter equally named to a potential shared variable. Listing 3.14 shows the usage of the `ParallelizableLoopBody` to verify a compound statement to be safe for parallelism.

```
ParallelizableBody parallelBody = new ParallelizableLoopBody(new SourceEquals("
    iterVarName"));
if (parallelBody.satisfies(loopBodyNode) {
  //parallelizable
} else {
  //not parallelizable
}
```

Listing 3.14: Usage of `ParallelizableLambdaBody` in the plug-in code.

The following verifications are not executed compared to the evaluated ones:

- Shared objects hidden by function calls are not reliably detected. Instead a function call in parallel section is recognized as a potential problem in parallelization. Shared pointer variables as described in Section 2.4.1 on page 35 are only detected if the pointer is not modified in the parallel section. To fully test for a shared pointer variable at a point in the parallel section, the variable occurrence in assignments must be analyzed and its r-value must then also be analyzed for a shared object in the same manner.
- A GCD test [ZC91] is not performed. But an iteration variable involved in an operation are recognized as a potential problem in parallelization and a warning is provided.
- A verification for different elements in a sequence can not be verified. No warning is provided.
- A functor or a function pointer used in a `std::for_each` call is not analyzed for writes on shared variables. Instead, a warning is provided that the used functor or function pointer is assumed to be free of shared variable modification.

3.3.3. For_Each call Pattern for Parallelism

An iterator usable for parallelism must implement the *Value* concept. This is one criteria for the first two parameters of a `std::for_each` call. Another criteria is the callable third parameter that can be a function pointer, lambda expression or a functor. A simplified visualization of the extensive pattern implementation can be seen in Figure 3.12 on the following page with the description of which part cover the different criteria.

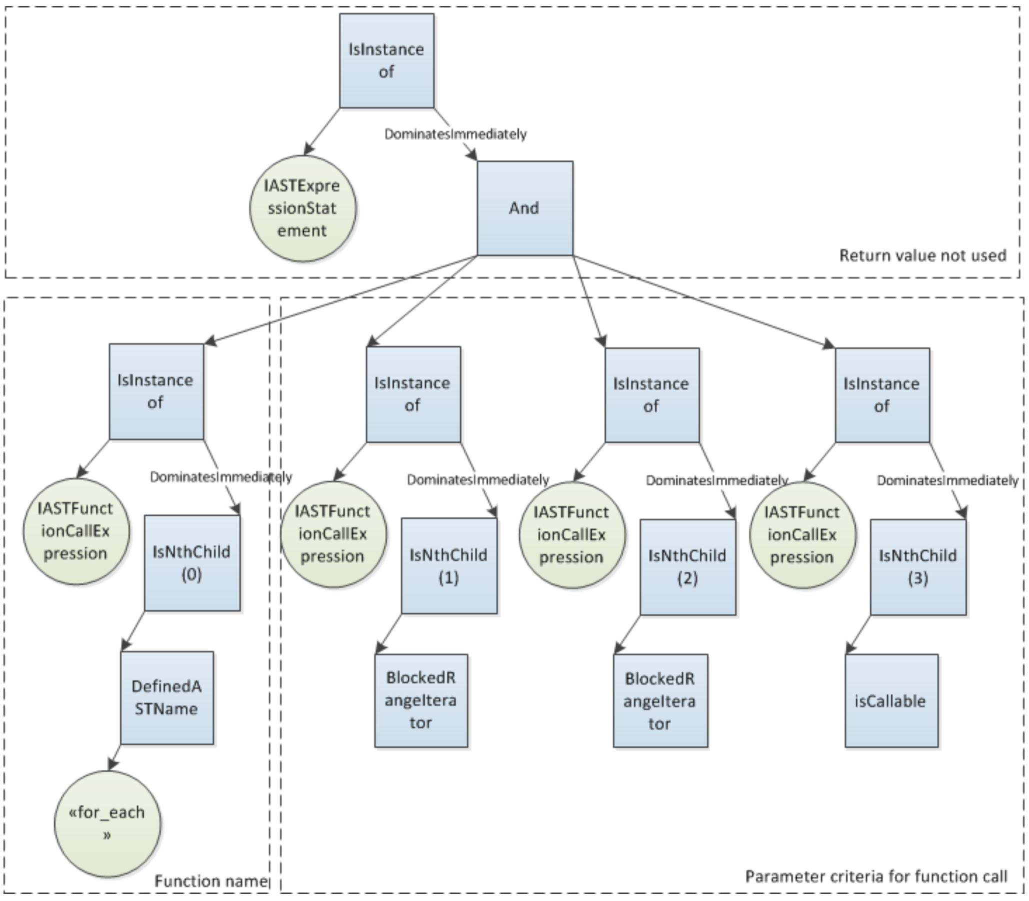


Figure 3.12.: Extensive pattern to recognize `std::for_each` calls.

The pattern basically defines a *for_each* call whose return value is not used and the criteria for each child of the function call. They represent the function name (1st child) and the parameters to the function call (other children). Any function call matching this pattern is considered a candidate for `tbb::parallel_for_each` or `ptor::for_each`. One criteria is not shown in the visualization: The limitation to only three parameters. This is covered by testing the absence of a right sibling for the third parameter. The *BlockedRangeIterator* pattern is the *ImplementsBlockedRangeValueConcept* pattern and the *ImplementsIteratorInterface* pattern connected with the logical *AND* pattern. The *ImplementsBlockedRangeValueConcept* pattern does not fully test all *Value* concept criteria. The tests for an available copy constructor, destructor and assignment operator overload is not checked as they are available on default and assumed to be not overridden privately. The requirements for a type *T* implementing the *Iterator* concept are [Kes10]:

- operator != (T)
- operator ++
- operator *

3.3.4. Iteration Variable Criteria for Parallelism

The iteration variable type in loops for parallelism must implement the *Value* concept. This criteria holds for index-based and iterator-based loops. In transformation to a `tbb::parallel_for` call the iteration type is further restricted to an integral type (*Index* concept). The patterns described here implement this verification and are returned by the specific `IndexIterationPatternProvider` as shown in Section 3.2.2 on page 50.

Index-Based Loops

The verification for an integral type is not possible with the AST provided by the CDT plug-in. Instead, a check is made for the iteration variable type to implement the *Value* concept as internally `tbb::parallel_for` uses the *Range* object. This is sufficiently safe because additionally the occurrence of the iterator variable in a subscript access is verified too and only integral types are allowed in the subscript operator of a sequence. The pattern as returned by the `IndexIterationPatternProvider` are described as follows, where *variableName* denotes the captured iteration variable name:

- **Iteration Variable Type Pattern:**
  DominatesImmediately(ImplementsBlockedRangeValueConcept, variableName)) – The expression dominating the variable name must satisfy the *Value* concept.
- **Value Of Iteration Variable Pattern:**
  IsImmediatelyDominatedBy(variableName, IsInstanceof(IASTArraySubscriptExpression.class)) – the variable name is dominated by an array subscript expression.

The access using the `at()` function is not considered for a possible access to a sequence element.

Iterator-Based Loops

The verification for iterators that satisfy the *Value* concept is straight forward. The pattern as returned by the `BlockedRangeIteratorIterationPatternProvider` are described as follows, where *variableName* denotes the captured iteration variable name:

- **Iteration Variable Type Pattern:**
  DominatesImmediately(And(ImplementsIteratorInterface, ImplementsBlockedRangeValueConcept), variableName)) – The expression dominating the variable name must satisfy the *Iterator* concept and the *Value* concept.
- **Value Of Iteration Variable Pattern:**
  IsDereference(variableName) – the iterator is dereferenced.

4. Summary

In this chapter the achieved project results are summarized and the performance impact of implemented transformations is evaluated. Known issues are shown and possible improvements are listed. The project concludes with the personal impressions made during the last couple of months.

4.1. Results

The transformation supported by the plug-in are outlined and analyzed according to the performance benefits. Due to time reasons the performance is analyzed in a rather synthetic environment. Interesting would also be to see how many loops in a project are detected for parallelism and how many loops are recognized correctly with parallelization issues. This analysis could not be done for time reasons.

4.1.1. Measurement

The measurement to evaluate the implemented transformation is explained here. The test setup as well as the results are described.

Setup

For the measurement a simple *for* loop iterating over an STL vector is used. The vector is randomly initialized with floating point numbers. Each iteration multiplies the element in the vector with 1.2 and stores it back at the same position. Listing 4.1 shows this operation.

```cpp
for (auto it = vec_data.begin(); it != vec_data.end(); it++) {
  *it *= 1.2;
}
```

Listing 4.1: Simple loop operation for measurement.

Using the plug-in, the loop is transformed to an equivalent `ptro::for_each` call as shown in Listing 4.2. Because no grain size and partitioner is specified, the default values are taken. These are a grain size of one and the auto partitioner.

```cpp
ptor::for_each(vec_data.begin(), vec_data.end(), [](float& it) {
  it *= 1.2;
})();
```

Listing 4.2: Loop from Listing 4.1 parallelized for measurement.

The measurement also contains the serial version using an `std::for_each` call. The task scheduler is initialized with default values at the beginning to prevent from including the time for the task scheduler initialization in the measurement. The parallelization of such a loop with a vector containing only ten elements brings no improvement. The overhead hides the parallelization benefit. Instead, the measurements are executed with increasing number of elements, starting with one thousand. For the measurement, a configuration as shown in Table 4.1 is used.

Operating System	Ubuntu 12.04, 64 Bit
GCC Version	4.7.2
TBB Version	4.1
CPU	Intel Xeon(R) CPU E5520 2.27GHz x 16 (16 cores)
Compiler Optimization Option	-O3 most optimized

Table 4.1.: Configuration of measurement.

Results

The results of the measurement are shown in Table 4.2.

Problem Size [number of elements]	Serial Loop [ms]	std::for_each [ms]	ptor::for_each [ms]	Improvement
1000	0.001	0.001	0.182	-18100.00%
10000	0.013	0.011	0.177	-1261.54%
100000	0.129	0.129	0.333	-158.14%
1000000	0.805	0.788	0.363	54.91%
10000000	8.287	8.479	4.332	47.73%
100000000	82.91	83.518	43.804	47.17%

Table 4.2.: Measurement results.

The parallelization start to provide an improvement somewhere between 100000 and 1000000 entries in the vector. The improvement is calculated by the formula `1 − (parallelTime / serialTime)`. A resulting improvement of 50% would therefore represent a parallel execution time half of the serial execution time.

4.1.2. Resulting Plug-In

The implemented plug-in shows several transformation that help the programmer to introduce parallelism into an existing C++ project. The plug-in is able to detect iterator-based and index-based loops as well as `std::for_each` calls transformable to an equivalent TBB algorithm. Possible problems in parallelism are evaluated and provided to the user. Not every problem can be recognized. But the plug-in gives also warnings in case there is a potential problem. This approach enables the user to explicitly choose which code shall be parallelized. The available tree pattern matching algorithm from the DS-8 project could be used to detect code for parallelism as well as for the evaluation of possible parallelism warnings. The measurement from Section 4.1.1 on the preceding page has shown performance impact of the transformation. While this is a rather synthetic test, it shows the potential of the plug-in. But its usage in a real life project must further be evaluated.

The following sections provide an overview of the transformation available in the plug-in. The transformation to STL algorithms were already available in the DS-8 project but enhanced with namespace prefix and header include handling.

Loop to `std::for_each`

An iterator-based *for* or *while* loop can be transformed to a `std::for_each` call. The following resolution variants are provided:

- Lambda expression
- Argument binding with `std::bind`
- Argument binding with `boost::bind`

Loop to `std::find`/`std::find_if`

An iterator-based *for* or *while* loop can be transformed to a `std::find` or a `std::find_if` call. The decision for `std::find` or `std::find_if` is handled by the plug-in. The following resolution variants are provided:

- Lambda expression
- Argument binding with `std::bind`
- Argument binding with `boost::bind`

Loop to `std::generate`

An iterator-based *for* or *while* loop can be transformed to a `std::generate` call. The following resolution variants are provided:

- Function pointer

Loop to `tbb::parallel_for_each`

An iterator-based *for* or *while* loop can be transformed to a `tbb::parallel_for_each` call. Problems in parallelization are reported as warnings. The following resolution variants are provided:

- Lambda expression
- Argument binding with `std::bind`
- Argument binding with `boost::bind`

`std::for_each` call to `tbb::parallel_for_each`

A `std::for_each` call can be transformed to a `tbb::parallel_for_each` call. Problems in parallelization are reported as warnings. The callable object to apply on the sequence element can either be one of the following:

- Lambda expression
- Functor
- Function pointer

`std::for_each` call to `ptor::for_each`

A `std::for_each` call can be transformed transformed to a `ptor::for_each`. Problems in parallelization are reported as warnings. The callable object to apply on the sequence element can either be one of the following:

- Lambda expression
- Functor
- Function pointer

This transformation does also support the manual chunk control. The grain size and the partitioner to use can be chosen.

Loop to `ptor::for_each`

An iterator-based *for* or *while* loop can be transformed to a `ptor::for_each` call. Problems in parallelization are reported as warnings. The following resolution variants are provided:

- Lambda expression
- Argument binding with `std::bind`
- Argument binding with `boost::bind`

This transformation does also support the manual chunk control. The grain size and the partitioner to use can be chosen.

Loop to `tbb::parallel_for`

An index-based *for* or *while* loop can be transformed to a `tbb::parallel_for` call. Problems in parallelization are reported as warnings. The following resolution variants are provided:

- Lambda expression

4.2. Known Issues

This section presents some unresolved problems or bugs which could not be fixed during the project.

4.2.1. STL Implementation Dependent Behavior

The result of the transformation from a *for* loop into a call of *for_each* using vector iterators is dependent on the implementation of the vector class. The example in listing 4.3 is transformed to code in listing 4.4 using the STL implementation shipped with the Microsoft Visual C++ compiler, whereas listing 4.5 shows the resulting transformation using GCC and its STL implementation. These examples stay for the different behavior whenever a the type of the vector element must be deduced.

```
void testForEachConversion(vector<int> numbers) {
  for(auto it = numbers.begin(); it != numbers.end(); ++it) {
    *it += 10;
  }
}
```

Listing 4.3: Transformation result from listing 4.3 using Microsoft Visual C++ compiler.

```
void testForEachConversion(vector<int> numbers) {
  std::for_each(numbers.begin(), numbers.end(),
      [](std::allocator<int >::value_type& it) {
        it += 10;
  });
}
```

Listing 4.4: Transformation result from Listing 4.3 using Microsoft Visual C++ compiler.

```
void testForEachConversion(vector<int> numbers) {
  std::for_each(numbers.begin(), numbers.end(),
      [](int& it) {
        it += 10;
  });
}
```

Listing 4.5: Transformation result from Listing 4.3 using GCC.

4.2.2. Index-based Loops and size_t

The type `size_t` is an alias for an integral type and is often used in the standard library to represent sizes and counts [plu13]. Index-based loops with a comparison of the loop running condition against a variable of type `size_t` are recognized but the applied transformation to an equivalent `tbb::parallel_for` call results in code that is not compilable. Listing 4.6 shows the source code that is transformed to Listing 4.7. This code is not compilable as the function template `tbb::parallel_for` can not be instantiated with different template arguments for the first two parameters. A manual adaption as shown in Listing 4.8 on the next page is needed.

```
void addValueOnElementsIndex(int data[], size_t size, int const value) {
  for (int i = 0; i != size; ++i) {
    data[i] += value;
  }
}
```

Listing 4.6: Example of index-base loop with a comparison to variable of type `size_t`.

```
void addValueOnElementsIndex(int data[], size_t size, int const value) {
  tbb::parallel_for(0, size, [data, value](const int& i) { //does not compile
    data[i] += value;
  });
}
```

Listing 4.7: Not compilable resulting `tbb::parallel_for` after transformation.

```
void addValueOnElementsIndex(int data[], size_t size, int const value) {
  tbb::parallel_for(size_t(0), size, [data, value](const int& i) {
    data[i] += value;
  });
}
```

Listing 4.8: Manually adapted `tbb::parallel_for` call.

4.2.3. Error in Nested Loop Detection

An outer loop is misclassified as transformable the same way like an inner loop that is correctly recognized as a transformable loop. This situation is faced if an outer loop is used to execute the inner loop for a constant number of times. Listing 4.9 shows an example of such a situation. The loop on line 2 is wrongly marked as transformable loop because of the found transformable loop on line 3.

```
1 void testNestedLoop(std::vector<int> numbers) {
2   for (int i = 0; i < 5; i++) {
3     for (auto it = numbers.begin(); it != numbers.end(); ++it) {
4       *it *= 1.2;
5     }
6   }
7 }
```

Listing 4.9: Situation showing misclassified outer loop.

The reason for this behavior is the loop recognition pattern that is defined with too loosely borders. An iteration variable initialization search and the search for a loop running condition is just addressed with a dominates pattern. The outer loop dominates the iteration variable of the inner loop and is therefore wrongly recognized as a transformable loop. Additionally, the tree pattern matching algorithm takes more time for the misclassification which results in a long running code analysis. It is expected to solve this issue by a definition that the iteration variable must additionally be a left child of the loop body. Since this error came up fairly late in the project, too less time was at hand to fix this bug. This behavior shows also that it is better to formulate a pattern as exact as possible to prevent from misclassification.

4.3. Outlook

The project could not cover all loop variants transformable to an equivalent call in TBB. During the project also other ideas came up that might help to increase the performance of an existing application. Further transformation ideas and performance increasing features are therefore reported here.

4.3.1. Improvements to the Existing Implementation

Due to time reasons, the current implemented transformations could not be implemented to provide the best usability. The possible improvements are listed in the following sections.

Specific Messages for Parallelization Issues

The current implementation states a general message in case of any parallelization issue. A specific message would help the user to retrace the problem. A specific message could include variables that are recognized to be a shared object with modification in the parallel section. Since the tree pattern matching algorithm is not able to identify every problem with 100% certainty, a differentiation in problems that are safely recognized and does that have potential problems would also be helpful.

Warnings in Parallelized Calls

The implemented patterns to identify parallelization issues in a possible transformation towards parallelism could also be used to analyze calls that already use TBB. The patterns are then used to identify data races or potential data races.

Preview of Algorithm

While choosing the resolution to apply, the description of the resolution could include a preview of the code that would be inserted.

Link Warning to Code

A possible warning could be selectable and the code responsible for the warning is marked in the code.

4.3.2. Further Loop Variants

The following transformation of loops could also be implemented:

- Enhance the implemented transformation to STL algorithms for index based loops as already suggested in [Kes10].
- Transformation of loops with different step size to `tbb::parallel_for` as suggested in Section 2.2.1 on page 27.
- Transformation of nested loops to TBB calls using the `blocked_range2d` object. This implementation would enable the parallelization of matrix multiplication as shown in Listing 1.8 on page 13.

4.3.3. Thrust Integration

The integration of thrust into the plug-in would enable the parallelization of several STL calls. The plug-in implementation could then enable the following features:

- The recognition of further loops that resemble a specific STL algorithm and the transformation to them or to the parallelized version of Thrust.
- The recognition of STL calls and the transformation to the parallelized version of Thrust.

The plug-in could also help in setting up a project using Thrust with TBB. This would include the integration of the Thrust header files in the include path and setting the compiler options to use the TBB back end (see Section 2.2.3 on page 32). Note that not all algorithms are parallelized with TBB in Thrust. Some implementation just provide the serial implementation.

4.4. Personal Reflection

Multi-core processors are the most important source of future performance gain. I knew this already. But the impact of this fact to programmers, language designer and also to the development environments was not that clear to me. During the project I realized that the new multi-core era forces to change many things in software development. Multi-threading is not only used for the separation of concerns and responsiveness of the user interface but also for increasing the performance. As in parallel programs data races are introduced quickly I think it is a very important task for an IDE to help the programmer to write correct code.

The project has shown that a semi automatic transformation of loops towards parallelism is possible and led to an increase in performance. I am satisfied with the outcome of the project although I expected to implement more transformations. I had good preliminary work I could base on. On one side the already implemented loop to STL algorithm transformations from Pascal Kesseli on the other side my former work with a try to use a DSL for the pattern creation. The later helped me a lot in creating the patterns for the analysis of the parallelization issues. The hardest part was to get a good understanding of parallelism. When can a loop be parallelized and when does it show situations resulting in a data race.

I thank my supervising professor Peter Sommerlad for all the support during the project. It was helpful in many situations to ask for his competent opinion. I also thank Mirko Stocker for co-assisting me during the project and the continuous helpful feedback in writing the report. I further thank the whole team from the IFS Rapperswil for their standby whenever CDT related questions came up.

A. Parallelize STL

The following table lists if and how the STL algorithms can be parallelized. Some of the results are gained in an analysis of the implemented algorithms in Thrust. The column "DS-8 Ready" states whether the specific STL algorithm is detectable with tree pattern matching.

STL Alogrithm	**header**	**DS-8 Ready**	**Parallelizable**	**TBB paradigm for parallelization**	**Remarks**
std::adjacent_find	algorithm	yes	no		the elements at the borders would also be needed for a comparison, not only comparison in the ranges provided to the different task.
std::all_of, std::any_of, std::none_of	algorithm	yes	yes	parallel_reduce	predicate extraction. Implementable using find_if. Any_of == !none_of.
std::binary_search	algorithm	no	no		To detect this algorithm, an analysis whether the elements are sorted would be needed in advance. Binary search is too complicated to detect because of too many implementation variants.
std::copy	algorithm	yes	yes	parallel_for	only possible if random access iterator for OutputIterator. Uses transform. If copy is used for I/O, it does not make much sense.

STL Alogrithm	**header**	**DS-8 Ready**	**Parallelizable**	**TBB paradigm for paralleliza-tion**	**Remarks**
std::copy_backward	algorithm	yes	yes	parallel_for	only possible if random access iterator for OutputIterator. If copy is used for I/O, it does not make much sense.
std::count	algorithm	yes	yes	parallel_reduce	
std::count_if	algorithm	yes	yes	parallel_reduce	The transformation must also include a an extraction of the comparison into a pred-icate function or the usage of an existing predicate (e.g. std::less¡int¿).
std::equal	algorithm	yes	yes	parallel_reduce	implemented with find_if_not. Find_if_not. reduction with minimum index of found element.
std::equal_range	algorithm	no	no		To detect this algorithm, an analysis whether the elements are sorted would be needed in advance. Uses too complicated lower_bound.
std::fill	algorithm	yes	yes	parallel_for	
std::fill_n	algorithm	yes	yes	parallel_for	The size parameter might be difficult to deduce if not a constant in for-loop.
std::find	algorithm	yes	yes	parallel_reduce	reduction with minimum index of found element. Might be slower as iteration is not stopped when found.
std::find_first_of	algorithm	yes	yes	parallel_reduce	maybe blocked_range2d with paral-lel_reduce
std::find_if	algorithm	yes	yes	parallel_reduce	reduction with minimum index of found element. Might be slower as iteration is not stopped when found.
std::for_each	algorithm	yes	yes	parallel_for	

STL Alogrithm	**header**	**DS-8 Ready**	**Parallelizable**	**TBB paradigm for paralleliza-tion**	**Remarks**
std::generate	algorithm	yes	yes	parallel_for	
std::generate_n	algorithm	yes	yes	parallel_for	The size parameter might be difficult to deduce if not a constant in for-loop.
std::includes	algorithm	yes	yes	parallel_reduce	The algorithm is only working on sorted ranges. For parallel_reduce logical AND is used
std::inplace_merge	algorithm	yes	no		
std::iter_swap	algorithm	yes	no		Switch of two values. Parallelization does not make sense.
std::lexicographical_compare	algorithm	yes	yes	parallel_reduce	Might be slower.
std::lower_bound	algorithm	no	no		
std::make_heap	algorithm	no	no		
std::max_element	algorithm	yes	yes	parallel_reduce	
std::merge	algorithm	yes	yes	parallel_invoke	Parallel merge with Divide and conquer implementation: `http://www.drdobbs.com/parallel/parallel-merge/229204454?pgno=3`
std::min_element	algorithm	yes	yes	parallel_reduce	
std::mismatch	algorithm	yes	yes	parallel_reduce	use find_if implementation. Find_if uses parallel_reduce
std::next_permutation	algorithm	no	no		Inherent serial.
std::nth_element	algorithm	no	no		Too complex to detect.
std::partial_sort	algorithm	no	no		Too complex to detect.
std::partial_sort_copy	algorithm	no	no		Too complex to detect.
std::partition	algorithm	yes	yes	parallel_for	usage of count_if, remove_copy_if, scatter_if

STL Alogrithm	header	DS-8 Ready	Parallelizable	TBB paradigm for paralleliza-tion	Remarks
std::pop_heap	algorithm	no	no		Too complex to detect.
std::prev_permutation	algorithm	no	no		Too complex to detect. Inherent serial.
std::push_heap	algorithm	no	no		Too complex to detect.
std::random_shuffle	algorithm	yes	no		Does not make sense.
std::remove	algorithm	yes	yes	parallel_for	remove_if, scatter_if, transform, for_each, Needs RandomAccessIterator
std::remove_copy	algorithm	yes	yes	parallel_for	remove_copy_if, scatter_if, transform, for_each, Needs RandomAccessIterator
std::remove_copy_if	algorithm	yes	yes	parallel_for	remove_copy_if, scatter_if, transform, for_each, Needs RandomAccessIterator
std::remove_if	algorithm	yes	yes	parallel_for	remove_if, scatter_if, transform, for_each, Needs RandomAccessIterator
std::replace	algorithm	yes	yes	parallel_for	using transform_if
std::replace_copy	algorithm	yes	yes	parallel_for	using replace_copy_if, transform, for_each,
std::replace_copy_if	algorithm	yes	yes	parallel_for	using transform, for_each,
std::replace_if	algorithm	yes	yes	parallel_for	using transform_if
std::reverse	algorithm	yes	yes	parallel_for	swap_ranges with reverse_iterator, for_each
std::reverse_copy	algorithm	yes	yes	parallel_for	same as copy with reverse_iterator
std::rotate	algorithm	yes	no		maybe with scatter.
std::rotate_copy	algorithm	yes	yes	parallel_for	with copy.
std::search	algorithm	yes	yes	parallel_for	not in thrust. Parallel search of subranges. Might be slower.
std::search_n	algorithm	yes	yes	parallel_for	not in thrust. Parallel search of subranges. Might be slower.

STL Alogrithm	**header**	**DS-8 Ready**	**Parallelizable**	**TBB paradigm for parallelization**	**Remarks**
std::set_difference	algorithm	yes	yes	parallel_for	Not parallelized with TBB in thrust. Algorithm expects sorted ranges to test difference.
std::set_intersection	algorithm	yes	yes	parallel_for	Not parallelized with TBB in thrust. Algorithm expects sorted ranges to test intersection.
std::set_symmetric_ difference	algorithm	yes	yes	parallel_for	Not parallelized with TBB in thrust. Algorithm expects sorted ranges to test symmetric difference.
std::set_union	algorithm	yes	yes	parallel_for	Not parallelized with TBB in thrust. Algorithm expects sorted ranges to create union. Algorithm uses std::copy algorithm
std::sort	algorithm	no	yes	parallel_invoke	stable_sort, merge_sort with parallelized divide and conquer. Sort difficult to detect.
std::sort_heap	algorithm	no	yes	parallel_invoke	before sort, need to be arranged as heap. Sort is parallelizable
std::stable_partition	algorithm	yes	yes		stable_partition_copy, remove_copy_if.
std::stable_sort	algorithm	no	yes	parallel_invoke	merge_sort with parallelized divide and conquer.
std::swap	algorithm	yes	no		to few calculation to vanish overhead of parallelism.
std::swap_ranges	algorithm	yes	yes	parallel_for	uses for_each
std::transform	algorithm	yes	yes	parallel_for	OutputIterator parameter needs to be a RandomAccessIterator for parallelization. Analysis, return value needed or not.
std::unique	algorithm	yes	yes	parallel_for	uses unique_copy, copy_if

STL Alogrithm	**header**	**DS-8 Ready**	**Parallelizable**	**TBB paradigm for paralleliza-tion**	**Remarks**
std::unique_copy	algorithm	yes	yes	parallel_for	copy_if (transform, exclusive_scan, scat-ter_if)
std::upper_bound	algorithm	no	no		too complicated to detect.
std::partial_sum	numeric	yes	yes	parallel_scan	Different variants to write this algorithm (e.g += variant). Also recognition of func-tional Functor (e.g. std::multiplies¡¿ pos-sible)
std::accumulate	numeric	yes	yes	parallel_reduce	operation to reduce must be associative and commutative.
std::adjacent_difference	numeric	yes	yes	parallel_for	using transform
std::inner_product	numeric	yes	yes	parallel_reduce	both operation must be associative and commutative. The one to reduce as well as the one for “multiplicatio”
std::is_sorted	algorithm	yes	yes	parallel_reduce	abort if not sorted → serial might be much faster
std::is_sorted_until	algorithm	yes	yes	parallel_reduce	algorithm explanation: returns pointer to one behind last object that is not sorted. abort if not sorted → serial might be much faster

Table A.1.: Summary of STL algorithms analysis towards recognizability and parallelism.

B. User Guide

B.1. Installation of Plug-in

Parallator needs Eclipse Juno (4.2) with CDT 8.1.1 installed. This can be done either by installing a bundled "Eclipse IDE for C/C++ developers CDT distribution" from the Eclipse download page[1] or by installing CDT from an existing Eclipse installation using its update site[2]. Now, the Parallator plug-in can be installed. Choose the menu "Help", "Install New Software..."and add the URL of the Parallator update site[3]. Proceed with the steps through the installation and accept the license.

B.2. Use of Plug-in

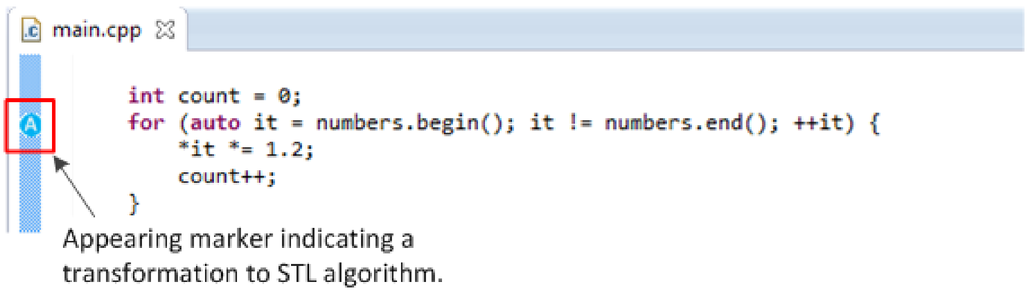
This user manual covers the application and configuration of the implemented transformation. After the installation, the Parallator plug-in is ready to use.

B.2.1. Recognition of Possible Transformations

There is different visual support for the indication of possible transformations available in the current working project. As soon as the Parallator plug-in recognizes possible transformations the following features can be used to navigate to reported code fragments.

Marker in the Vertical Ruler

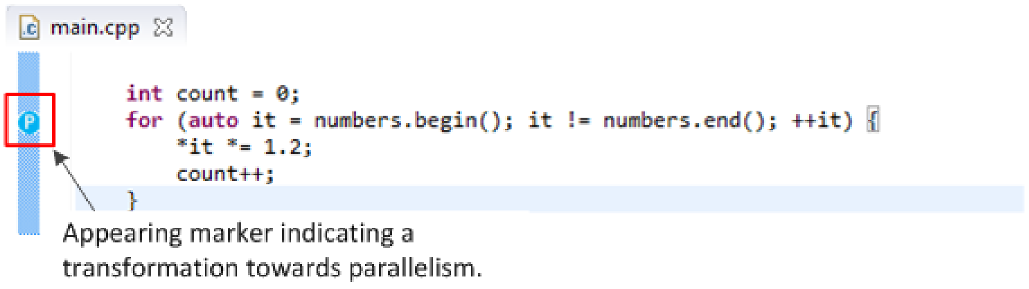
Any source code recognized from Parallator is indicated by a marker on that line in the vertical ruler. Figure B.1 shows the indication for a transformation to an STL algorithm. Figure B.2 on the next page shows the indication for a transformation towards parallelism.


main.cpp
Appearing marker indicating a transformation to STL algorithm.

```
int count = 0;
for (auto it = numbers.begin(); it != numbers.end(); ++it) {
    *it *= 1.2;
    count++;
}
```

Figure B.1.: Marker for a possible transformation to an STL algorithm.

[1]http://www.eclipse.org/downloads/
[2]http://download.eclipse.org/tools/cdt/releases/juno
[3]https://sinv-56017.edu.hsr.ch/tools/ds8/development/juno/


main.cpp
Appearing marker indicating a transformation towards parallelism.

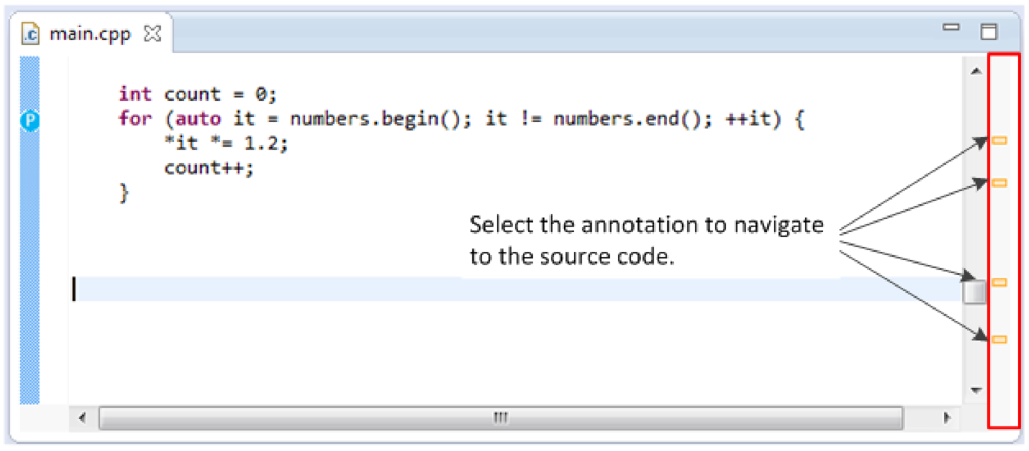
```
int count = 0;
for (auto it = numbers.begin(); it != numbers.end(); ++it) {
    *it *= 1.2;
    count++;
}
```

Figure B.2.: Marker for a possible transformation towards parallelism.

If the code on a specific line shows a code construct transformable by both categories. The icon for transformations towards parallelism is shown.

Navigate with the Overview Ruler

The overview ruler provides ability to see all possible transformations of a file. Clicking on an entry let you navigate in a fast manner to a next transformation candidate. Figure B.3 shows the overview ruler with the annotations usable to navigate.

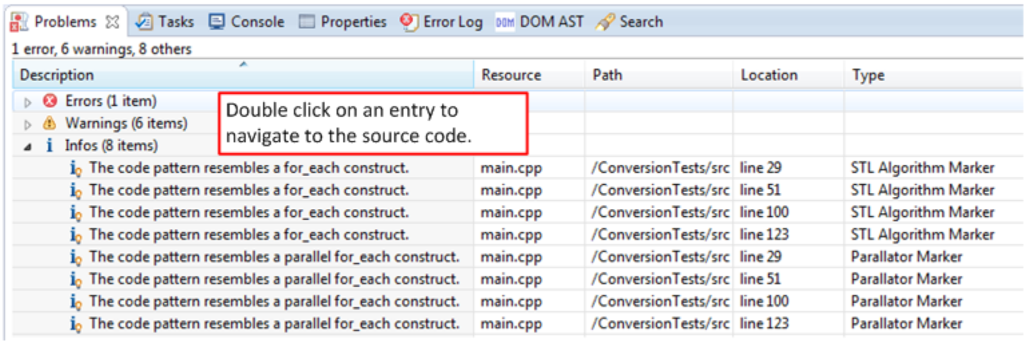


```
int count = 0;
for (auto it = numbers.begin(); it != numbers.end(); ++it) {
    *it *= 1.2;
    count++;
}
```

Figure B.3.: Potential transformation indication in the overview ruler.

Navigate with the Problems View

All possible transformation applicable in a project are listed in the *Problems* view. Figure B.4 on the next page shows the problem view with STL algorithm markers and parallelization markers.



1 error, 6 warnings, 8 others

Description	Resource	Path	Location	Type
Errors (1 item)				
Warnings (6 items)				
Infos (8 items)				
The code pattern resembles a for_each construct.	main.cpp	/ConversionTests/src	line 29	STL Algorithm Marker
The code pattern resembles a for_each construct.	main.cpp	/ConversionTests/src	line 51	STL Algorithm Marker
The code pattern resembles a for_each construct.	main.cpp	/ConversionTests/src	line 100	STL Algorithm Marker
The code pattern resembles a for_each construct.	main.cpp	/ConversionTests/src	line 123	STL Algorithm Marker
The code pattern resembles a parallel for_each construct.	main.cpp	/ConversionTests/src	line 29	Parallator Marker
The code pattern resembles a parallel for_each construct.	main.cpp	/ConversionTests/src	line 51	Parallator Marker
The code pattern resembles a parallel for_each construct.	main.cpp	/ConversionTests/src	line 100	Parallator Marker
The code pattern resembles a parallel for_each construct.	main.cpp	/ConversionTests/src	line 123	Parallator Marker

Double click on an entry to navigate to the source code.

Figure B.4.: The *Problems* view lists all possible transformation candidates in a project.

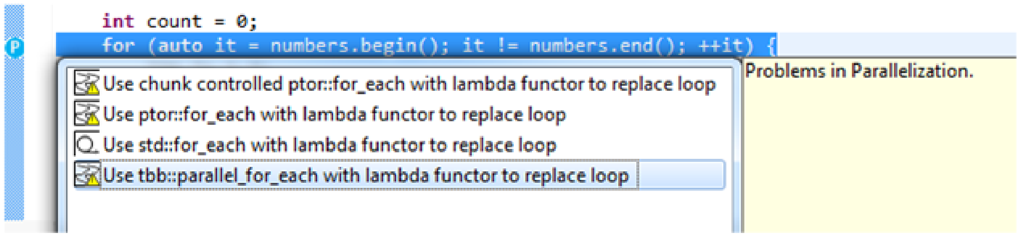
A double click on an entry let you navigate to the specific source code position.

B.2.2. Apply a Transformation

A transformation can be applied in two different ways.

Selection in the Vertical Ruler

Click on the marker in the vertical ruler to see the transformations applicable for the code at the specified line. Double click to apply the transformation. Figure B.5 shows the possible transformations appeared after a click on the marker icon. Note that if several markers are placed on one line, the transformations for all markers are shown when clicked on the marker. A warning for a parallelization issue appears when a transformation is selected.



```
int count = 0;
for (auto it = numbers.begin(); it != numbers.end(); ++it) {
```

Use chunk controlled ptor::for_each with lambda functor to replace loop
Use ptor::for_each with lambda functor to replace loop
Use std::for_each with lambda functor to replace loop
Use tbb::parallel_for_each with lambda functor to replace loop

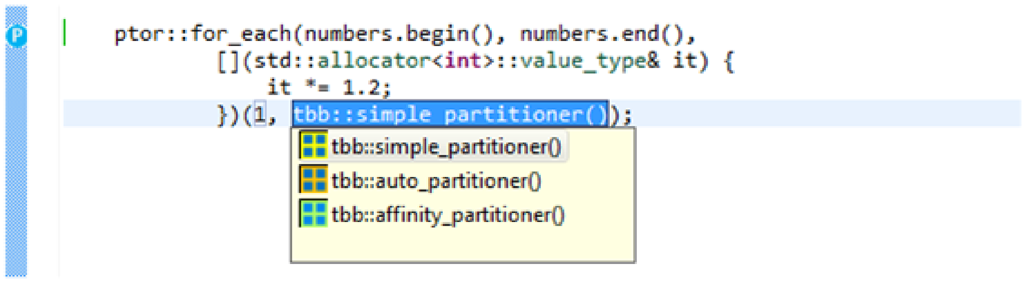
Problems in Parallelization.

Figure B.5.: A Click on the marker shows applicable transformation. Double click on a transformation to apply it.

Table B.1 on the next page shows the transformation icons with the described meaning. If a transformation with manual chunk control is chosen, the plug-in marks the two configurable parameters. Navigate between the positions using the tabulator, use the keyboard or the mouse to select a partitioner and complete by pressing enter. Figure B.6 on the following page shows the configurable parameters marked for navigation.

○→	Transformation to STL algorithm.
○⚠	Transformation to STL algorithm with warning.
≋	Transformation for parallelism.
≋⚠	Transformation for parallelism with warning.

Table B.1.: Icons for transformation and its explanation.



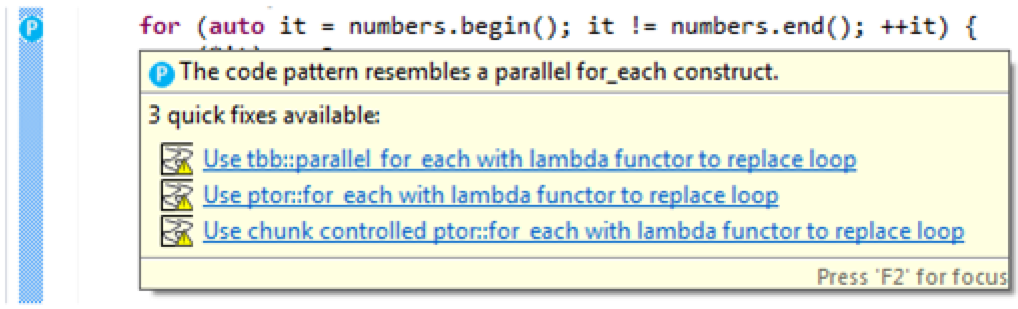
```
ptor::for_each(numbers.begin(), numbers.end(),
        [](std::allocator<int>::value_type& it) {
            it *= 1.2;
        })(1, tbb::simple_partitioner());
```

tbb::simple_partitioner()
tbb::auto_partitioner()
tbb::affinity_partitioner()

Figure B.6.: Use the tabulator to switch between the configurable parameters. Finish editing by enter.

Hover over a Marked Line

The transformation to apply can also be chosen from a pop-up that appears when hovering in the source code over a marked line. Figure B.7 shows the appearing pop-up.



```
for (auto it = numbers.begin(); it != numbers.end(); ++it) {
```

The code pattern resembles a parallel for_each construct.

3 quick fixes available:

- Use tbb::parallel_for_each with lambda functor to replace loop
- Use ptor::for_each with lambda functor to replace loop
- Use chunk controlled ptor::for_each with lambda functor to replace loop

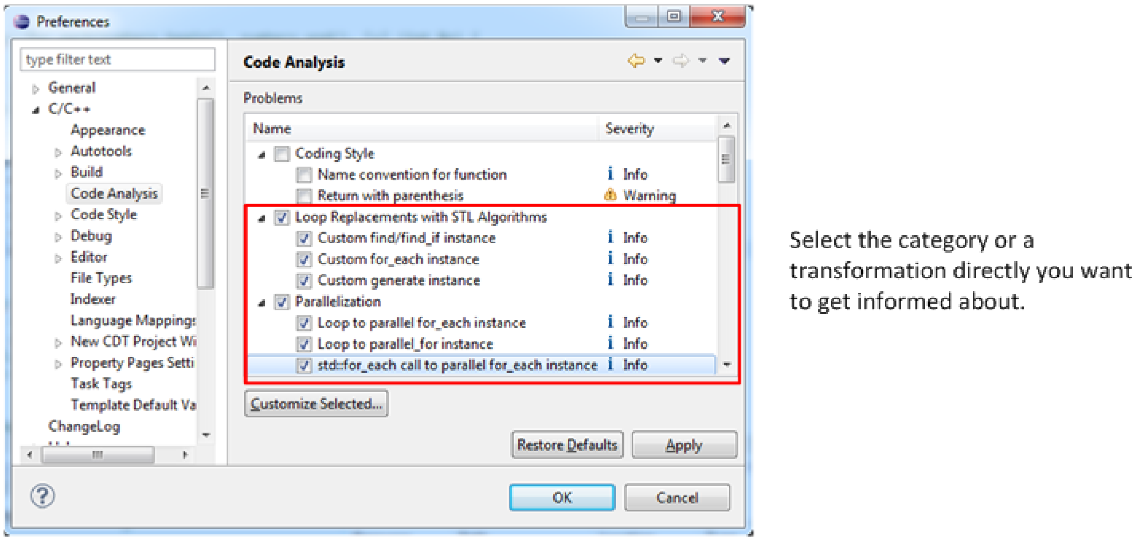
Press 'F2' for focus

Figure B.7.: Quick-fix pop-up appeared when hovering over a marked line.

Select a transformation to apply. If the code on a specific line shows a code construct transformable to parallelism *and* to an STL algorithm, the pop-up shows only the transformation towards parallelism.

B.2.3. Choose Transformations to Recognize

Open the preferences located in "Window - Preferences" and select the property "C/C++ - Code Analysis". On the appearing page the recognition of possible transformations can be enabled or disabled. Figure B.8 on the following page shows the code analysis configuration page.



Code Analysis

Problems

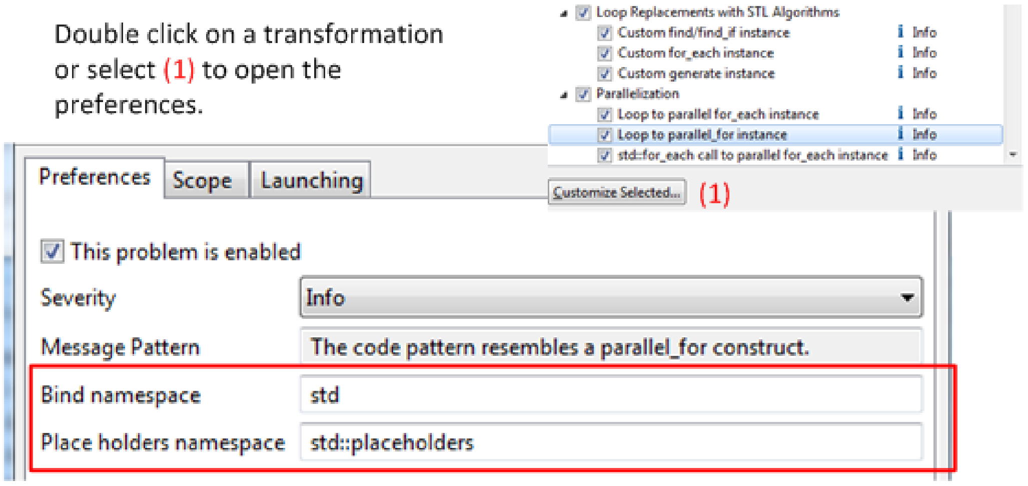
Name	Severity
☐ Coding Style	
☐ Name convention for function	Info
☐ Return with parenthesis	Warning
☑ Loop Replacements with STL Algorithms	
☑ Custom find/find_if instance	Info
☑ Custom for_each instance	Info
☑ Custom generate instance	Info
☑ Parallelization	
☑ Loop to parallel for_each instance	Info
☑ Loop to parallel_for instance	Info
☑ std::for_each call to parallel for_each instance	Info

Select the category or a transformation directly you want to get informed about.

Figure B.8.: Preferences to configure the possible transformation to detect.

B.2.4. Configure Argument Binding

Parallator supports the transformation of loops to STL or TBB algorithms. A possible usage of argument binding is possible in such a transformation. In the preferences page of the transformation, the argument binding can be configured. Figure B.9 shows the pre-entered configuration to use the argument binding form the C++11 standard.



Double click on a transformation or select (1) to open the preferences.

Name	Severity
☑ Loop Replacements with STL Algorithms	
☑ Custom find/find_if instance	Info
☑ Custom for_each instance	Info
☑ Custom generate instance	Info
☑ Parallelization	
☑ Loop to parallel for_each instance	Info
☑ Loop to parallel_for instance	Info
☑ std::for_each call to parallel for_each instance	Info

☑ This problem is enabled

Severity	Info
Message Pattern	The code pattern resembles a parallel_for construct.
Bind namespace	std
Place holders namespace	std::placeholders

Figure B.9.: Preferences to configure the argument binding.

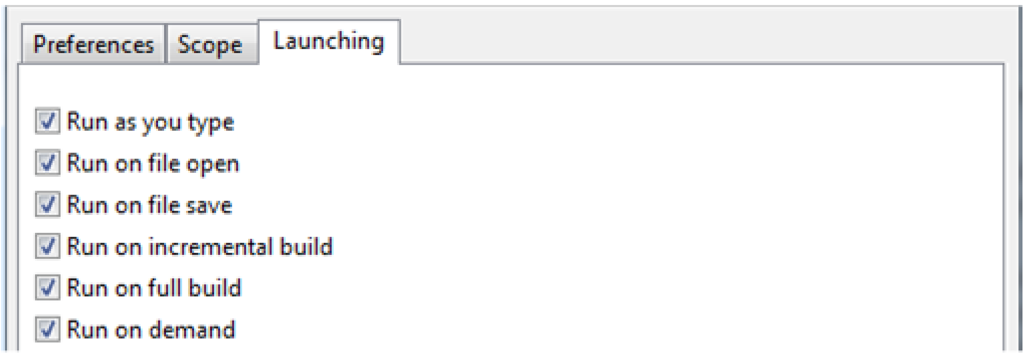
Table B.2 on the following page shows the configuration entries needed to choose boost binding.

	Bind namespace	**Place holders namespace**
Std Bind	std	std::placeholders
Boost Bind	boost	<empty>

Table B.2.: Configuration entries for choosing the argument binding.

B.2.5. Configure Launch Mode

Choose the launch mode in the preference page of a transformation. You can decide at which times a transformation is recognized. Figure B.10 shows the possible entries. Parallator runs a quite extensive analysis and "Run as you type" might be too time consuming in larger projects.



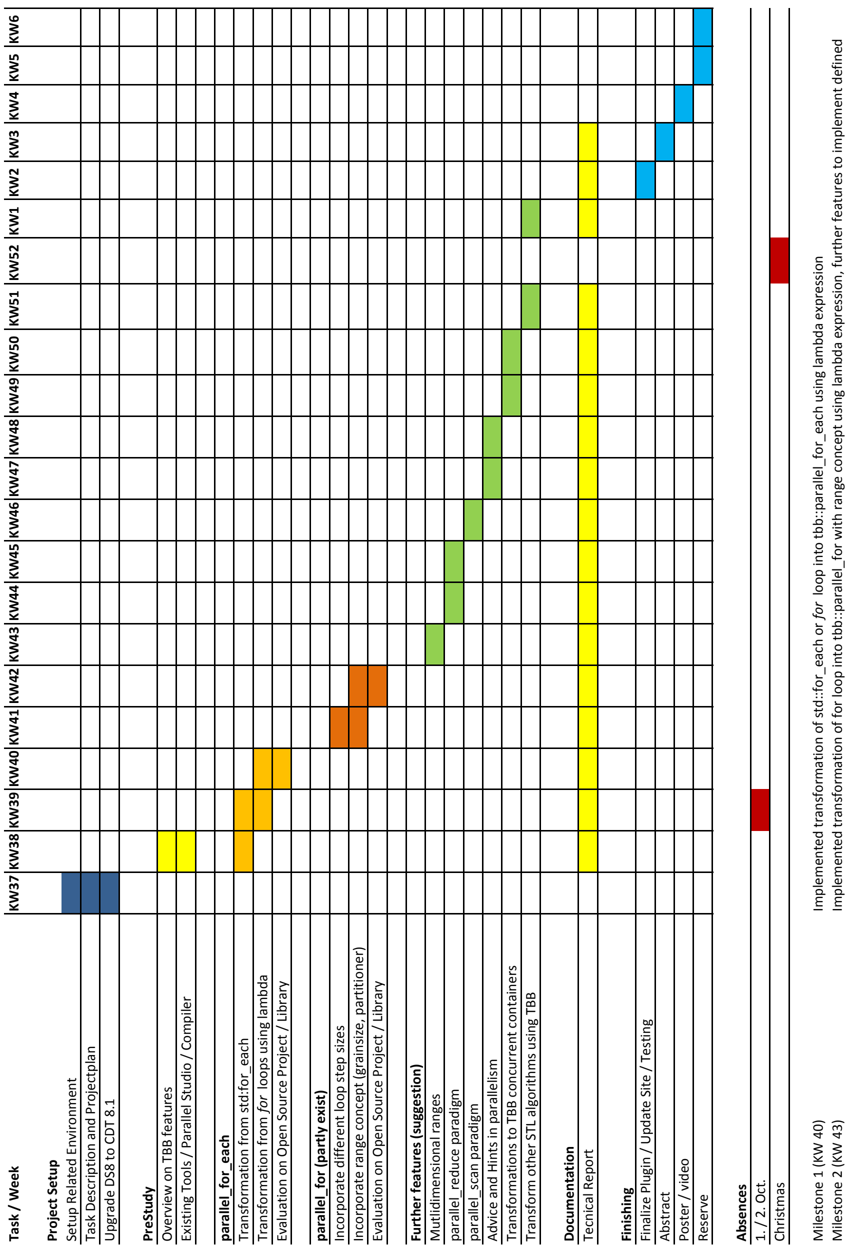
- [✓] Run as you type
- [✓] Run on file open
- [✓] Run on file save
- [✓] Run on incremental build
- [✓] Run on full build
- [✓] Run on demand

Figure B.10.: Preferences to configure the launch mode.

If only "Run on demand" is selected, start the analysis by selecting a project and choose "Run C/C++ Code Analysis" in the context menu of the project.

C. Project Management

The project plan created at the beginning of the project is shown in Figure C.1.



Task / Week	KW37	KW38	KW39	KW40	KW41	KW42	KW43	KW44	KW45	KW46	KW47	KW48	KW49	KW50	KW51	KW52	KW1	KW2	KW3	KW4	KW5	KW6
Project Setup																						
Setup Related Environment	🟦																					
Task Description and Projectplan	🟦																					
Upgrade DS8 to CDT 8.1	🟦																					
PreStudy																						
Overview on TBB features		🟨																				
Existing Tools / Parallel Studio / Compiler		🟨																				
parallel_for_each																						
Transformation from std::for_each		🟧	🟧																			
Transformation from *for* loops using lambda			🟧	🟧																		
Evaluation on Open Source Project / Library				🟧																		
parallel_for (partly exist)																						
Incorporate different loop step sizes					🟧																	
Incorporate range concept (grainsize, partitioner)					🟧	🟧																
Evaluation on Open Source Project / Library						🟧																
Further features (suggestion)																						
Mutlidimensional ranges							🟩															
parallel_reduce paradigm								🟩	🟩													
parallel_scan paradigm										🟩												
Advice and Hints in parallelism											🟩	🟩										
Transformations to TBB concurrent containers													🟩	🟩								
Transform other STL algorithms using TBB															🟩		🟩					
Documentation																						
Tecnical Report		🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨		🟨	🟨	🟨			
Finishing																						
Finalize Plugin / Update Site / Testing																		🟦				
Abstract																			🟦			
Poster / video																				🟦		
Reserve																					🟦	🟦
Absences																						
1. / 2. Oct.			🟥																			
Christmas																🟥						

Milestone 1 (KW 40)	Implemented transformation of std::for_each or *for* loop into tbb::parallel_for_each using lambda expression
Milestone 2 (KW 43)	Implemented transformation of for loop into tbb::parallel_for with range concept using lambda expression, further features to implement defined

Figure C.1.: Project plan and milestones of the project.

The tasks starting from week 43 were considered only ideas of what could be possible in this project. But the tasks until week 43 could not be finished as planned. Figure C.2 reflects the project and shows the scheduling of the work done.

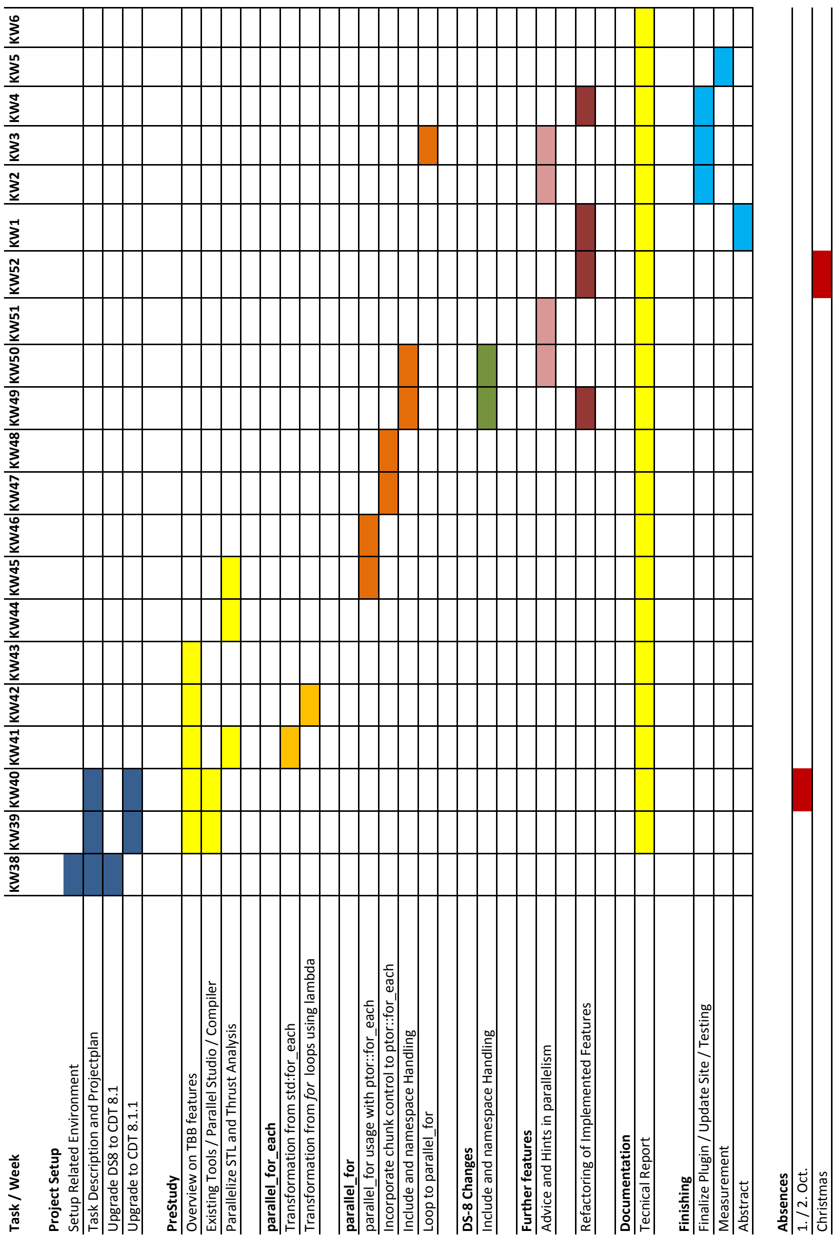


Figure C.2.: Effective project plan of the project.

The important points that led to this quite different project schedule compared to the expected one are:

- Understanding the features of TBB took longer. I constantly learned how TBB works but it was harder than expected. This resulted also in a greater effort to write the introduction and dive into the topic of parallelism. But it was also needed to get the clear understanding in the application of TBB with iterators on STL containers.

- TBB's `tbb::parallel_for_each` makes it easy to introduce parallelism with iterator-based loops but has shown also worse performance in many cases compared to `tbb::parallel_for`. This fact was then faced with `ptor::for_each`. This enabled also the incorporation of chunk control with the linked mode feature but this whole task was not expected.
- In general the refactoring of the features was underestimated but resulted in an unified work flow for the transformations and a clean plug-in architecture.
- Measurements have been done to analyze the features of TBB. But finding an open source project to apply the transformation has shown to be difficult and the implementation of the transformation were just not finished. The measurement of the performance impact of an implemented transformation is made at the end.

C.1. Time Schedule

The expected duration of a master thesis is 810 hours of work (27 ECTS * 30 hours). I worked 847.5 hours over the past 21 weeks. The distribution over the project weeks can be seen from Figure C.3.

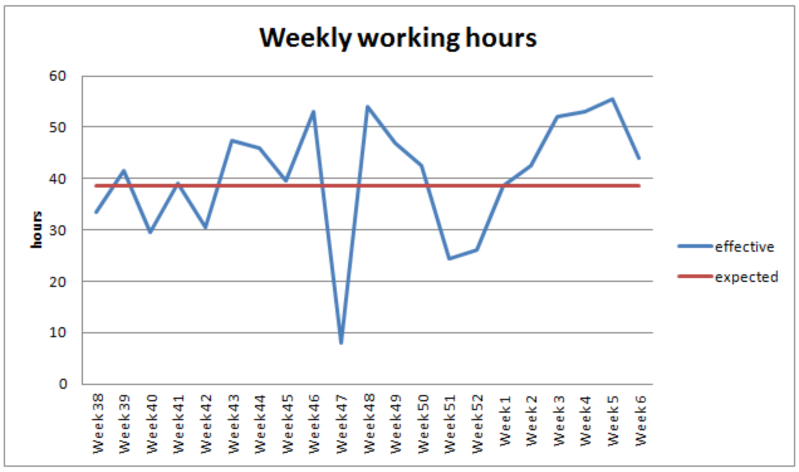


Figure C.3.: Weekly working hours.

Some comments on the working hours are listed here:

- In general, the variation in the weekly working hours are sourced by the time needed to work on the MSE seminar module.
- In week 47 an effort was needed to reach the milestone 2 of the MSE seminar module.
- The lower working time in week 51 is related to the additional time needed for the finish of the seminar paper.

D. Project Setup

Important installation instructions to develop and use the plug-in are explained in this chapter. Also the changes to the continuous integration environment compared to the DS-8 project and the DS-8 DSL project [Kem12] are reported.

D.1. Development environment

This section reports important issues in setting up the environment to develop on the Parallator plug-in.

D.1.1. Used Versions

The software and the versions used for the development are listed in table D.1

Software	Version
Java	1.6
Eclipse Juno	4.2
Eclipse CDT Plug-In	8.1.1
TBB	4.1
Tycho	0.16.0
Maven	3.0.4
GCC used by CDT	4.7
MSVC used by CDT	VC 10

Table D.1.: Used tools in development.

D.1.2. Installation and Configuration of TBB

The plug-in supports the Microsoft Visual C++ (MSVC) and the Gnu Common Compiler (GCC). This section covers the needed build settings for a C++ project setup in Eclipse CDT for both compiler. TBB is only supported with the MSVC compiler on windows. The installation of TBB is therefore explained for Windows (usage with MSVC) and for the Ubuntu distribution (usage with GCC).

On Windows

The following pre-conditions must be fulfilled for the setup of a project in CDT using MSVC and TBB:

- Installed MSVC, e.g. the compiler shipped with the Microsoft Visual Studio

- Downloaded TBB version for Windows from the download page[1] and unzipped into a directory. In the following steps referenced with `<TBB_ROOT>`.

For the configuration of the C++ project in CDT follow these steps:

1. Create a new project: Choose "New - C++ Project".
2. Enter a project name, select "Hello World C++ Project" as project type and select the "Microsoft Visual C++" toolchain.
3. Step through the wizard with "Next" and "Finish" the wizard.
4. Open the properties page by selecting "Properties" in the context menu of the created project.
5. Open "C/C++ Build - Settings" and select the "Preprocessor" settings of the "C++ Compiler (cl)".
6. Add the directory to the installed MSVC compiler e.g. "C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\include" to the include path.
7. Add `<TBB_ROOT>\include` to the include path.
8. Switch to the "Libraries" settings of the "Linker (link)".
9. Add `<TBB_ROOT>\lib\ia32\vc10` to the additional libpath.
10. Repeat step 5 to step 9 for the setup of a "Release" configuration.
11. Apply the settings and close the configuration with "Close".

For the correct environment of the MSVC compiler, start Eclipse from the "Visual Studio Command Prompt" that sets up the environment for using Microsoft Visual Studio tools.

On Ubuntu

The following pre-conditions must be fulfilled for the setup of a project in CDT using MSVC and TBB:

- Installed GCC compiler with C++11 feature support. At least 4.7 is therefore needed. Good installation descriptions can be found in the web.[2]
- Downloaded TBB version for Linux from the download page[3] and unzipped into a directory. In the following steps referenced with `<TBB_ROOT>`.

For the installation of TBB, do the following step:

1. Make the library files of TBB available in the system. One way to do this is copying the library files to the library directory of linux. In a concrete case: copy the files from `<TBB_ROOT>/lib/intel64/cc4.1.0_libc2.4_kernel2.6.16.21/` to `/lib/`.

[1]http://threadingbuildingblocks.org/download#stable-releases
[2]http://charette.no-ip.com:81/programming/2011-12-24_GCCv47/
[3]http://threadingbuildingblocks.org/download#stable-releases

For the configuration of a C++ project in CDT follow these steps:

1. Create a new project: Choose "New - C++ Project".
2. Enter a project name, select "Hello World C++ Project" as project type and select the "Linux GCC" toolchain.
3. Step through the wizard with "Next" and "Finish" the wizard.
4. Open the properties page by selecting "Properties" in the context menu of the created project.
5. Open "C/C++ Build - Settings" and select the "Includes" settings of the "GCC C++ Compiler".
6. Add `<TBB_ROOT>\include` to the "include paths (-l)".
7. Switch to "Miscellaneous" settings of the "GCC C++ Compiler".
8. Add `-std=c++11` to the "other flags" field. This enables C++11 features.
9. Switch to "Libraries" settings of the "GCC C++ Linker".
10. Add `tbb` to "Libraries (-l)".
11. For timing functions, also add `rt` to "Libraries (-l)"
12. Repeat step 5 to step 11 for the setup of a "Release" configuration.
13. Apply the settings and close the configuration with "Close".

D.2. Continuous Integration

The continuous integration environment was already setup by the DS-8 project and the DS-8 DSL. As build server, Hudson is used. The plug-in is built using maven and tycho. For the version control management, SVN is used. The changes in the setup compared to the former projects are reported here.

D.2.1. Build Environment

The DS-8 DSL project has introduced a new plug-in on which the DS-8 plug-in was related. Since the DS-8 DSL project did not provide the simplification expected, the dependency upon its plug-in is removed. The changes involved to remove the dependency can be seen in the SVN revision: 584.

D.2.2. Testing

The DS-8 DSL project had already integrated some features of the CDT Testing plug-in[4]. But at this time, the testing plug-in did not exist as a separate plug-in and the features were just copied. During this project a switch is made to use CDT Testing and remove the former copied functionality. The CDT Testing plug-in is needed to add external resources, e.g. header files of the STL, to the parser.

[4] https://github.com/IFS-HSR/ch.hsr.ifs.cdttesting used version: 3.0.0

E. Upgrade to CDT 8.1.1

In the upgrade of the DS-8 project to use CDT Version 8.1.1 two changes were needed that are reported here.

Type of Dependent Expression

Since the resolution of Bug 2999111 [Sch12] the expression type of the iteration variable `it` of Listing E.1 resolves to `TypeOfDependentExpression` instead of a `CPPUnknownBinding`.

```
template<class T>
void printContainer(T &t) {
  for (typename T::iterator it = t.begin(); it != t.end(); ++it) {
        cout << *it << endl;
    }
}
```

Listing E.1: The type of variable `it` resolves to `TypeOfDependentExpression`.

The type of the variable `it` must be deduced for a declaration of the iteration variable in a lambda expression. Beside other components the `DecltypeFallBackDeclarationGenerator` class is responsible for the type evaluation. In the case of Listing E.1 the explicit type is difficult to deduce. This is addressed with the C++11 construct `decltype`. The condition to insert `decltype` in the declaration was based on the `CPPUnknownBinding` and had to be changed to test for a `TypeOfDependentExpression`.

Typedef Specialization

Since the resolution of Bug 2999111 [Sch12] the expression type of a template class declared variable like `it` in Listing E.2, resolves to a `CPPTypedefSpecialization` binding that wraps the new type introduced with a `decltype` declaration. The iterator type in STL vector implementation is often declared with a typedef declaration. For the evaluation of whether the type follows the restrictions of the *Iterator* concept or not, the wrapped type must be used. To receive this type, a binding inherited from `ITypedef` is iteratively unpacked as long as the type is still a `ITypedef` binding. On the resulting binding that represents a type, the *Iterator* concept is evaluated.

```
vector<int> v(10);
for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
  *it = rand();
}
```

Listing E.2: The expression type of variable `it` resolves to `CPPTypedefSpecialization`.

List of Figures

Bibliography

[AK12] Intel Corporation Alexey Kukanov. Parallel programming with intel threading building blocks. Website access October 23, 2012. https://www.sics.se/files/projects/multicore/days2008/IntelTBB_MCDays2008_Stockholm.pdf.

[Bec12] Pete Becker. C++ standard. Website access November 04, 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.

[Boo13] Boost.org. Boost c++ libraries. Website access February 1, 2013. http://www.boost.org/.

[Cor12a] Intel Corporation. Design parallel performance with less risk and more impact. Website access November 27, 2012. http://software.intel.com/zh-cn/sites/default/files/design-parallel-performance_studioxe-evalguide.pdf.

[Cor12b] Intel Corporation. Intel threading building blocks for open source. Website access October 23, 2012. http://threadingbuildingblocks.org/.

[Cor12c] Intel Corporation. Intel threading building blocks, reference manaual, document number 315415-016us. Website access October 23, 2012. http://threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference.pdf.

[Cor12d] Intel Corporation. parallel_for and parallel_for_each usage with stl vector. Website access October 26, 2012. http://software.intel.com/en-us/forums/topic/328652.

[DD12] Ralph Johnson Danny Dig. Relooper: Refactoring for loop parallelism. Website access November 1, 2012. http://hdl.handle.net/2142/14536.

[DME09] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 397–407, Washington, DC, USA, 2009. IEEE Computer Society.

[ea13] Jared Hoberock et al. Parallelizing the standard algorithms library. Website access January 20, 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3408.pdf.

[Fou12] The Eclipse Foundation. Eclipse cdt (c/c++ development tooling). Website access November 27, 2012. http://www.eclipse.org/cdt/.

[Fou13] The Eclipse Foundation. Cdt - static analysis. Website access January 30, 2013. http://wiki.eclipse.org/CDT/designs/StaticAnalysis.

[GHJ95] Erich Gamma, Richard Helm, and Ralph and Johnson. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, 1 edition, 1995.

[Gue12] Paul Guermonprez. Parallel programming course threading building blocks (tbb). Website access November 1, 2012. http://intel-software-academic-program.com/courses/manycore/Intel_Academic_-_Parallel_Programming_Course_-_InDepth/IntelAcademic_Parallel_08_TBB.pdf.

[JH13] Nathan Bell Jared Hoberock. Thrust - parallel algorithms library. Website access January 17, 2013. http://thrust.github.com/.

[Kem12] Martin Kempf. Dsl for specifying ds-8 tree patterns. Technical report, Institute for Software, HSR, Switzerland, 2012.

[Kes10] Pascal Kesseli. Loop analysis and transformation towars stl algorithms. Technical report, Institute for Software, HSR, Switzerland, 2010.

[KV11] Wooyoung Kim and Michael Voss. Multicore desktop programming with intel threading building blocks. *IEEE Softw.*, 28(1):23–31, January 2011.

[Lea12] Doug Lea. D. lea. parallelarray package extra166y. Website access November 1, 2012. http://gee.cs.oswego.edu/dl/concurrency-interest/index.html.

[MRR12] Michael D. McCool, Arch D. Robison, and James Reinders. *Structured parallel programming patterns for efficient computation.* Elsevier/Morgan Kaufmann, Waltham, MA, 2012.

[plu13] plusplus.com. size_t - c++ reference. Website access February 6, 2013. http://www.cplusplus.com/reference/cstring/size_t/.

[RR99] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '99, pages 72–83, New York, NY, USA, 1999. ACM.

[Sch12] Markus Schorn. Cdt - bugzilla bug 299911. Website access October 8, 2012. https://bugs.eclipse.org/bugs/show_bug.cgi?id=299911.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation.* Thomson Course Technology, 2nd edition, 2006.

[SL05] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[Sut05] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for parallel and vector computers.* ACM, New York, NY, USA, 1991.