Would You Like Some Syntactic Sugar With Your TBB?

Evan Driscoll

December 20, 2007

Abstract

We have begun implementation on a source-to-source transformer that greatly eases the burden of writing code that uses Intel Thread Building Blocks (TBB). TBB is a C++ library that provides a parallel programming interface that is at a higher level than raw threads. However, because it works within the bounds of standard C++, there is a lot of syntactic overhead involved with writing a TBB application. We take a step back and show that the interface could be made far nicer if one is willing to forgo this need, and allow an extra preprocessing step.

1 Introduction

It is a well known fact that single-threaded performance of microprocessors has ceased the rapid increase of the past. Chip manufacturers are spending the additional transistors afforded by today's technologies to build multicore chips. Programmers who wish to continue to benefit from Moore's Law must create programs which can operate in parallel on these separate cores.

However, parallel programming is a hard programming. Many programming models and systems have been created in an attempt to ease the burden on the programmer, allowing them to think at a higher level. One such system is Intel's Thread Building Blocks (TBB) [3].

TBB is a C++ library that provides a fairly-high level model of concurrency. It comprises a set of C++ classes that allow parallel code to be expressed in several different ways, as well as some supporting libraries such as concurrent data structures and atomic types that have even broader applicability.

However, TBB has one substantial problem, which is that the syntactic overhead of using TBB is rather high. This limitation is imposed by the fact that TBB is simply a C++ library, and doesn't extend the language. This has the great benefit that it can theoretically be used with any C++ compiler (Intel most directly supports the Intel compiler, GNU's GCC, and Microsoft's C++ compiler), but it also makes other solutions look better in comparison, especially OpenMP [2] and Cilk [9].

The objections to TBB syntax take one of two forms. First, TBB uses some fairly advanced parts of C++ and exposes the programmer to them. To use TBB forces the programmer to use templates, operator overloading, and for task-based programming, even placement new. We argue this is not a particularly strong objection however. Experience with these bits of C++ will make the programmer comfortable enough that these syntactic warts cause little more than an eye roll.

The second objection to the syntax is much more important and much more difficult to deal with. When parallelizing a loop, it is necessary to take the loop body and implement it as a method inside a class. This has two consequences: the programmer has to write a ton of boilerplate code (since the loop body no longer has access to the local variables of the original function, these have to be duplicated in — then copied into — the new class) and the loop body is now lexically separated from the function it was previously in[1].

In this paper, we extend C++ with new language features for expressing parallelism. The new language features mostly eliminate the syntactic overhead of TBB, which should make it much easier to program and much easier to use with existing programs. We implement a prototype tool, called *tbbetter* for translating the extended language to pure C++ for compilation by any back end compiler.

[1]In TBB's defense, it should be noted that programming with traditional threads libraries, such as pthreads, produces more or less the same objections. The loop body must still be split into a function, and, if it needs more than one argument, the programmer must write a struct to pass all of the data in. In the typical case, TBB may be ahead of pthreads with respect to amount of boilerplate, since the programmer doesn't have to explicitly create threads. These objections take weight when TBB is compared to other models, such as OpenMP.

The organization of the paper proceeds as follows. Section 2 will provide an overview of the TBB library, and contrast it with other systems. Section 3 covers our language extensions, describing the syntax and semantics of the additions. Section 4 provides an example program excerpt demonstrating the benefit of our extensions. Section 5 discusses the prototype implementation. Section 6 discusses some attributes of our solution. Section 8 concludes.

2 TBB Overview

At a high level, the highlights of the TBB library are:

1. Loop-based components
2. Task-based components
3. Stream programming components
4. Concurrent containers
5. Cache-aware concurrent memory allocators
6. Communication primitives (locks and atomic types)

The latter three components can be viewed as supporting libraries for inter-thread communication, and would actually likely be useful even if a different mechanism were used to express parallelism. For instance, even when using threads directly it would often be useful to have a concurrent hash table. These components are not of particular interest for this project; the syntax for them is not particularly lacking.

It is the first three components where TBB stands to improve. It is also where TBB most stands above other offerings. Even a traditional thread library, such as pthreads, wouldn't be much if there wasn't synchronization primitives for instance. The first three components however provide a substantially different interface for expressing what can be executed in parallel. Furthermore, behind what the programmer enters is a task stealing thread scheduler that provides high potential for runtime performance gains that would require a lot of extra work if the programmer used a traditional threads library instead.

We have considered the portions of these components which we have used during class, and tried to produce improved syntax for those portions. We haven't programmed using the stream components, but the following sections cover the first two components in more detail.

2.1 Loop-based components

The loop-based components abstract tasks that are traditionally performed in loops. There are four classes which fall into this category: `parallel_for`, `parallel_while`, `parallel_reduce`, and `parallel_scan`.

This paper mostly focuses on `parallel_while`; for reasons that are explained later this section, this is probably the most egregious syntactic wart in all of TBB, so it is the component that stands to improve the most. We also describe syntax for an improved interface for `parallel_for`.

The author hasn't used either `parallel_reduce` or `parallel_scan`, so these were not addressed in this work. However, there is no reason why a more serious project would not be able to develop similar extensions for these loops. In particular, `parallel_reduce` implements a parallel reduction (for instance, finding the sum of an array, or the extreme elements of an array). Throughout this paper OpenMP will be mentioned as a source for some inspiration, and this is a prime example of where lessons could be learned; OpenMP also provides a mechanism for parallel reductions. `parallel_scan` seems fairly limited to applicability, and in fact the TBB tutorial doesn't even cover it.

The remaining two constructs, `parallel_for` and `parallel_while`, are discussed in the following sections.

2.1.1 `parallel_for`

To transform a `for` loop into one that uses `parallel_for`, the programmer follows a series of steps. At a high level, he implements a new class that contains a method which performs some number of iterations of the loop, the bounds of which are passed as a parameter to the method. One invocation of the method usually performs a couple hundred to some thousands of iterations of the original loop. If it performs too few iterations, the overhead of scheduling and task management becomes overwhelming. If it performs too many, there could be load imbalance issues as the tasks contain differing amounts of work, or there may even be too little parallelism.

Using `parallel_for` is appropriate when there are no loop carried dependencies and the number of iterations performed can be determined before the loop begins. (Unfortunately, functions such as `strlen` can't really be implemented. Consider a very simple version that loops over the array until the terminating zero is found. While there are no loop-carried data dependencies, there is a control dependence from iteration i to $i+1$ — if iteration i discovers the zero, iteration $i+1$ cannot safely run or it risks a buffer overflow.) In some cases where `parallel_for` cannot be used, `parallel_while` can.

The use of `parallel_for` bears a lot of relation to the `#pragma parallel for` construct in OpenMP [2]. Both are used for programming loops that are usually most naturally expressed as `for` loops, and both have somewhat similar restrictions on what loops they can parallelize. Some options that are present in OpenMP — for instance statically allocating iterations to the threads — are not available in TBB, and others — such as which variables are local and which are shared — is expressed through the way the body class is coded instead of the declarative style of OpenMP. On the other hand, TBB is in other senses more flexible. The iteration bounds needn't be simple integers, and TBB provides a class that will allow easy iteration over a 2-D array without nested `parallel_for` loops, something that in OpenMP would require extra code.

2.1.2 `parallel_while`

The model of `parallel_while` is a bit different from that of `parallel_for`. It is appropriate to use `parallel_while` when the work that a loop carries out can be though of as having two parts: one part runs sequentially and produces work to be done, and the second part actually performs that work. Intel refers to the two parts as the "stream" and the "body"; this paper uses "generator" to refer to the first part instead. Later we will see in detail an example from an Othello AI that can be parallelized nicely with `parallel_while`. When parallelized, the generator pulls a move that should be evaluated off of the beginning of a list of possible moves; the body applies that move to the current board position and evaluates the new position, either by continuing the walk of the decision tree or by direct evaluation.

The reason that `parallel_while` is the worst offender syntactically is because the generator and body portions of the loop *both* have to be implemented in separate methods. In practice, they are often in different classes as well; the TBB tutorial uses this technique. Thus the problems mentioned in the introduction are doubled in severity. Since there are two classes, there is twice as much boilerplate to write. Furthermore, not only is the loop body lexically separate from the function it used to be in, but what used to be one body has now been separated into *two* functions across *two* classes!

Section 4 and the appendix provide an example from an Othello AI that uses `parallel_while`, which should make these objections concrete.

2.2 Task-based components

TBB provides a `task` class for performing task-based parallelism. Conceptually this is very similar to the language Cilk [9], which is an extension to C. Task-based programming seemingly maps most comfortably to tasks which would be written recursively if they were in a pure sequential language. The canonical example, used both in the TBB tutorial and Cilk manual as a demonstration of task-based parallelism, is a recursive computation of the Fibonacci series. More practically, we wrote a task-based parallel version of the Othello AI, which is a recursive walk over the tree of possible moves from a given board position. TBB tasks support both explicit continuation passing style (CPS) programs as well as blocking tasks; this is much the same as Cilk.

If `parallel_while` is the most egregious example of poor TBB syntax, their task-based programming must be the least. Because the tasks in task-based parallelism are usually invocations of a recursive function anyway, the problem where the programmer has to pull code from where it is and put it in its own function doesn't exist. There is still boilerplate to write: because the `task`'s `execute` method doesn't take parameters, what would normally be parameters must again be stored as member variables. However, there is typically less than in the case of a parallel loop since there are likely fewer parameters than local variables accessed from within the loop. While `parallel_while` could be called a coding horror, TBB tasks are not much more than coding annoyances.

However, there is still room for improvement. Tasks are where the low-level syntax of TBB actually becomes an issue. There is a fair mess of easy-to-get-wrong details when programming with this part of the library; we will give two examples. First, `task` objects contain a reference count that is used by the scheduler to know when to either reactivate a blocked task or run its continuation. Under the current system, the programmer must explicitly set this reference count. However, from the examples that we've seen, it seems that this usually could be fairly easily be determined automatically. (In fact, the author is surprised that the library doesn't track it already.) Assuming there is no subtlety here and this is actually the case, this is perfect room for an automatic transformation. (The one complex bit is if tasks are created after others have already started running. The reference count needs to be set before the first is spawned, so the program would have to know what tasks are coming up in the future. This could cause a problem if tasks are created inside a loop where the number of iterations can't be determined statically.)

Second, there are other minor details that make a lot of difference. The main example of this comes from the author's experience converting an initial task-based Othello AI to one that used CPS. When converting a recursive function to a task, the recursive invocations turn into a two step process: first the programmer creates a new task object and fills in the member variables corresponding to the arguments, and then he spawns the task to begin execution. When using blocking tasks, the creation of the child task looks as follows[2] (taken from [5]):

```
FibTask& a =
   *new( allocate_child() )
         FibTask(n-1,&x);
```

When using continuation passing, the first thing that the programmer has to do is allocate a continuation:

```
FibContinuation& c =
   *new( allocate_continuation() )
         FibContinuation(sum);
```

However, now any child tasks that are created must be created as children of the *continuation* rather than children of the currently executing task. (This is so the scheduler wakes the continuation when the children are done executing.) Thus the creation of the children must be changed to:

```
FibTask& a =
   *new( c.allocate_child() )
         FibTask(n-1,&c.x);
```

There are two changes (both of them adding "`c.`" before something), but only the second will produce a compiler error if it is forgotten. TBB's library could track this (barring subtleties the author is overlooking), but it doesn't. The debugging version of the library it will fail on most violations of this, but even then it gets the problem wrong, failing an assertion with the message "attempt to spawn task whose parent has a ref_count==0 (forgot to set_ref_count?)." (The reason it fails is because the reference count on the continuation was set, not the reference count on the parent task.)

Because of these low-level details, an improvement to the preset TBB task syntax would still be welcome. A lot of benefit could be gained even from changes to the TBB library (for instance, to be either more forgiving or at least give a better diagnostic in the second example above) and possibly some preprocessor macros, but making the syntax as nice as Cilk would require a new compilation stage.

3 Language Extension

This section describes our proposed language extensions. We begin with a new variable qualifier for types, `tbb_shared`, then cover our extensions for `parallel_for`, `parallel_while`, and tasks. We then mention that there is a C++ elision of programs using our syntax that can be created using only `#define` (and so requires no preprocessing other than what C and C++ perform anyway).

Our parallel `for` and `while` loops are spelled `concurrent_for` and `concurrent_while` instead of `parallel_for` and `parallel_while` so as not to conflict with the TBB names.

3.1 `tbb_shared`

We first add a new qualifier, `tbb_shared`. This marks a variable as one that is shared between threads. In OpenMP, when using a parallel loop such as `#pragma parallel for`, the programmer specifies which variables should be thread-local and which are shared between threads[3]. We choose a different approach for dealing with the same question. By default, all variables are considered thread-private. Variables that should be shared are declared with the `tbb_shared` qualifier, as in the following definition:

```
tbb_shared int nBest;
```

Note that this makes the fact that it is shared a property of the variable rather than specific to a loop.

The decision of whether to take this approach or one more like OpenMP (which would change the syntax of `concurrent_for` and `concurrent_while` and obviate the need for `tbb_shared`) was not clear. The

[2]Note also the use of placement new here as something that a language extension would obviate the need for.

[3]It is also possible to specify a default, then those which deviate from the default.

OpenMP approach requires the repetition of variable names at the loop but has the benefit of explicitness. In addition, it has the property that one variable can appear as thread-local in one loop but shared in another; whether this is a benefit or not is debatable. Our approach has conciseness going for it, as well as the ability to determine whether a variable is shared or not directly from the declaration. From the standpoint of a new language, the arguments for neither side really stands out. However, we have an additional benefit: our approach makes the C++ elision of code that uses our loops a little more direct. Nevertheless, it would be possible to change the syntax of loops to something like the following:

```
concurrent_while(x>0) with shared(x, y, z)
```

In addition, presently the `tbb_shared` qualifier automatically implies `volatile`. The intention behind this was that variables that are shared between threads should almost always be marked volatile anyway. (This avoids compiler optimizations that can easily compromise correctness.) Thus the declaration above is exactly the same as

```
tbb_shared volatile int nBest;
```

In our example later (section 4), the `lock` object is shared between threads, so is marked `tbb_shared`. However, it is passed to a constructor as an argument which only takes non-`volatile` objects. This necessitates a cast to remove the `volatile` qualifier. Also, a large variable (for instance an array) might be shared for efficiency reasons, but treated read-only and hence doesn't need `volatile`. Because of these reasons, we now feel that this was not the correct approach. (It would be a one-line change in the source to fix this decision.)

Finally, we could have used `volatile` itself as the indicator for shared variables, but this seems like a poor idea.

One further property of our transformation that deserves mention is that globals variables also need to be decorated with `tbb_shared` to be shared. If a global variable without that qualifier is used within a loop, it will be thread-local. However, as discussed in section 6.1, thread-local variables not declared within the loop body itself are read-only, so a situation cannot arise where one thread writes to a global variable `G` and another works incorrectly because it does not see that modification.

3.2 `concurrent_for`

The proposed syntax for `concurrent_for` is as follows:

```
concurrent_for(var, start, end [,grainsize])
```

- `var` is the name of the iteration variable, or perhaps a declaration of it (e.g. `int x`) with the qualification that it breaks the C++ elision (see section 3.5)
- `start` is the initial value of the index
- `end` is the ending value of the index
- `grainsize` optionally specifies the size of the range done by each subtask (otherwise it uses TBB's auto partitioner)

Currently, `concurrent_for` only parses in our tool; no transformation is done.

There are two concessions to make the C++ elision work well. The first is to use commas as separators between the clauses instead of semicolons. This allows each clause to be a macro parameter (with `grainsize` taken care of by a variable-length macro parameter list). The second concession is to break what is traditionally the first clause (containing something like `i=0`) into two parts, the variable and starting index. One problem with it is that if `var` declares a variable, the C++ elision won't work. It would have been possible to add yet another clause (the type of `var`), but we felt that was too unwieldy.

It would have been possible to make the syntax mostly the same as `for` with an additional optional parameter:

```
concurrent_for(init; cond; incr [,grainsize])
```

This also would work well enough with the C++ elision. (The first macro parameter would hold the first three clauses, then `__VA_ARGS__` can pick up the optional grainsize. In fact, this would have fixed the problem mentioned in the previous paragraph.) However, we feel that the proposed syntax above has a couple important benefits. The loops that can be expressed by `parallel_for` through a reasonable automatic process are somewhat limited. In particular, the loop should look something like the following:

```
  for(int i=start ; i<end ; ++i)
```

In particular, the loop should initialize the loop variable to some starting value, increment the counter each iteration, and break when it reaches an end value. Anything else would require at least some degree of special handling. For instance, counting down would be difficult to do, and incrementing by more than one would require special handling[4]. The syntax as presented above forces the programmer to confront this issue and make sure that it fits into this form.

TBB provides mechanisms for extending the abilities above (it is possible to implement a new class to divide the original range into subranges), and if this were a much larger project, it might be worthwhile to provide support for this. However, the present form probably handles the majority of real-world cases, and effort would be better spent elsewhere for a while.

3.3 `concurrent_while`

Our adaption of `parallel_while` introduces three new keywords, `concurrent_while`, `cwhile_generator`, and `cwhile_iterator`. A stylized use looks as follows:

```
cwhile_iterator WorkItem nextItem;
concurrent_while(workItemQueue.size() > 0)
{   cwhile_generator {
        nextItem = workItemQueue.pop();
    }
    doWork(nextItem);
}
```

Recall that `parallel_while` uses a generator object to produce work. For each piece of work produced, the following sequence of events occurs:

1. The generator object checks to see if there is more work to be done (`workItemQueue.size() > 0`)

2. The generator object retrieves the next piece of work, and puts it into a *`cwhile_iterator`* object (`nextItem = workItemQueue.pop()`)

3. The body object gets the next piece of work from the iterator object, then performs the work on it (`doWork(nextItem)`)

[4]The problem there is that the logic that splits up the overall range into smaller ranges makes no guarantees about how it does it. For instance, say that the loop was originally `i+=2`, and went from 0 through 4. It would be reasonable to split this into two subranges of 0 through 2 and 3 through 4. But then the task that got the second subrange would start at 3, which wouldn't have been run originally. It's possible to work around this, but for this project not worth it.

Code in the condition of the `concurrent_while` as well as that which appears within the block annotated `cwhile_generator` will be transformed into the generator object; the remainder of the body of the `concurrent_while` will be transformed into the body object. (Just as a side note, the braces around line 4 above could be removed in the same way that braces can be omitted from single-statement bodies in conditionals and loops.)

The new qualifier `cwhile_iterator` marks the variable which is used to transfer work from the generator to the body object. It corresponds to an out parameter in the generator's `pop_if_present` method[5] and an in parameter in the body's `operator()`, which actually performs the work.

In each `concurrent_while` there must be exactly one `cwhile_generator`. Syntactically this can appear anywhere, but logically it should always be first. (It wouldn't be unreasonable to require that it does, though we don't.) We also require that there is use of exactly one variable qualified by `cwhile_iterator`.

Again, there are a number of interesting design choices here. First is the method of indicating what the iterator object is. We seriously considered a syntax such as `concurrent_while(workItem ; workItemQueue.size() > 0)`, as well as a couple other alternatives, but decided to keep consistency with the (equally questionable) decision to make `tbb_shared` a qualifier.

A second choice was the method of indicating what portion of the `concurrent_while`'s body is meant for the generator object and which portion is meant for the body object. The first idea was to use two statement blocks immediately following the `concurrent_while`, perhaps marked with annotations:

```
concurrent_while(workItemQueue.size() > 0)
cwhile_generator {
    nextItem = workItemQueue.pop();
}
cwhile_body {
    doWork(nextItem);
}
```

However, this idea quickly went out the window when we made C++ elision an explicit goal, as there is no way to use macros to "fix up" the above code. The

[5]This method returns a boolean indication of whether there is work left — which in our transformation comes from the condition of the `concurrent_while` — and, if there is work available, stores it in the out parameter.

next idea was to just put both of those inside the body of the concurrent_while:

```
  concurrent_while(workItemQueue.size() > 0)
  {   cwhile_generator {
          nextItem = workItemQueue.pop();

      }
      cwhile_body {
          doWork(nextItem);
      }
  }
```

This is an equally viable solution as the one we finally choose, and has the benefit of being slightly more explicit about what is going on.

Finally there is just a question of naming; cwhile_generator and cwhile_iterator are pretty clunky. We didn't want to leave off the prefix though, because just like tbb_shared, both generator and especially iterator are names that can and do show up in real code.

All of these alternate syntaxes would be very easy to change to should the desire arise once one knows how. The harder part was figuring out all of what needed changed to make the parsing occur, and the the transformation and pretty printing code.

3.4 Tasks

For tasks, we basically steal from Cilk [9]. We can use pretty much the same syntax that they do, up to what TBB provides support for. A Cilk Fibonacci number generator looks as follows:

```
cilk int fib (int n)
{
  if (n<2) return n;
  else {
      int x, y;
      x = spawn fib (n-1);
      y = spawn fib (n-2);
      sync;
      return (x+y);
  }
}
```

A transformation for tasks (unimplemented) would proceed as follows. First, we can key off of the cilk keyword to know that we should translate this function into a task object. It would wrap the function in a class, turning parameters into member variables. For each spawn occurrence, the translated code would need to allocate an object, set the member variables, and then spawn. It would need to determine the number of such objects that will be created, and set the reference count automatically. On sync, it would simply call the sync call within TBB's task object. Finally, any call sites to the function that appear elsewhere the transformation would need to replace with code to create a task object and run it. It should be possible to translate any Cilk program that only uses basic features into a TBB program using transformations along this line.

Cilk also includes some more advanced features that wouldn't currently translate to TBB. The main one here is inlets. In Cilk, when a child task completes execution, it can be told to execute an "inlet," which is essentially a closure that executes in the context of the parent. Inlets are guaranteed to execute atomically with respect to other inlets of the same invocation, so provide a useful method for updating the state of the parent procedure. Perhaps a more important thing that it does is allow the parent to abort other children. This was inspired by a problem Blumofe et. al. had in their original Cilk work ([1]) where a chess AI experienced far worse results than the other benchmarks. The reason for this was that the AI was pruning the search space during the sequential version, but in the parallel version it would spawn off those tasks before they would get pruned. Thus the parallel versions were evaluating far more board positions than the sequential version. The abort mechanism in inlets was added to compensate for this.

Unfortunately, TBB currently does not have mechanisms either for inlets or aborting subtasks. Because of this there isn't any transformation we could make from these Cilk constructs to TBB. However, if support for aborts was added to TBB in the future, we are confidant that it would not be hard to find a suitable transformation from Cilk to TBB's hypothetical mechanisms.

3.5 C++ Elision

One goal we had through most of the development was to allow for a *C++ elision* of a program using our syntax, performable with just the preprocessor. (This was inspired by the fact that Cilk has the same property.) By adding #define macros to "define away" tbb_shared and cwhile_iterator, transform concurrent_while to just plain while, and perform a similar transformation for concurrent_for, the result is a valid C++ program.

This elision will perform the same in terms of correctness as the parallel version if the parallel version doesn't have race conditions.

4 Example

This section will show an example program that uses our syntax. It is an excerpt from an Othello AI assigned as a project earlier in the semester.

The original code which we will parallelize is as follows:

```
 1  while( moves.size() ) {
 2      b = board;
 3      b.applyMove( moves.front() );
 4      moves.pop();
 5
 6      quality = Lookahead(
              b, other_color, newdepth );
 7
 8      if( quality > nBest ) {
 9        nBest = quality;
10      }
 1  }
```

The first thing that we need to do is separate the generator portion from the body portion. The queue `moves` contains a list of moves which we want to consider. On line 3, `moves.front()` retrieves the next piece of work, and line 4 updates the `moves` array, which needs to be done in the sequential portion. We need to store the move we want to consider in an iterator variable, so we need to tease apart line 3 to do this. Line 2 doesn't have anything to do with the creation of work, so it can be moved later. After these transformations we have the following:

```
 1  OthelloMove move;
 2  while( moves.size() ) {
 3      move = moves.front();
 4      moves.pop();
 5
 6      b = board;
 7      b.applyMove(  );
 8
 9      quality = Lookahead(
              b, other_color, newdepth );
10
 1      if( quality > nBest ) {
 2          nBest = quality;
 3      }
 4  }
```

The variables `b` (line 6) and `quality` (line 9) are defined outside of the loop. For reasons discussed in section 6.1, we need to declare them within the loop. For stylistic reasons, the author would have written it this way in the first place. Also, `nBest` is intended to be shared between all threads, so it needs to be protected by a lock (or atomic update, as in the version in the appendix). Finally, we throw in our new keywords, and we have the final version of the loop:

```
 1  cwhile_iterator OthelloMove move;
 2  concurrent_while( moves.size() ) {
 3      cwhile_generator {
 4          move = moves.front();
 5          moves.pop();
 6      }
 7
 8      OthelloBoard b = board;
 9      b.applyMove(  );
10
 1      int quality = Lookahead(
             b, other_color, newdepth);
 2
 3      if( quality > nBest ) {
 4          tbb::spin_mutex::scoped_lock
                l((tbb::spin_mutex&)lock);
 5          if( quality > nBest ) {
 6              nBest = quality;
 7          }
 8      }
 9  }
```

We also need to make a couple changes elsewhere. First, we can remove the declarations of `b` and `quality` since they are now declared on the inside of the loop. (We do need to add another declaration of `b` in another branch however.) The variable `nBest` is intended to be shared, so should be marked `tbb_shared`. We need to declare `lock` (and mark it `tbb_shared`). We need to create a `task_scheduler_init` object in `main`. Finally, we need to add a prototype for the `Lookahead` function. After that we simply tell the Makefile to use our transformer (as explained later, this is as simple as redefining `CC` and `LD`), to link against the TBB libraries, and give it switches to tell it where the libraries and TBB headers are.

The original size of the loop was 11 lines, and it is now 19. Three of those lines are either entirely blank or consist of just a closing brace, one additional line could be removed if we didn't use the double-checked locking pattern, and we removed two lines from outside the loop. The above code is really not any more difficult to understand than originally.

By contrast, the appendix includes the version that the author wrote for our earlier assignment. There are about 59 lines there, an increase of over 5x in code size. The loop was split up and moved out of the old location. And finally, tests between the generated version and the hand-written version show that the generated version performs only about 1% slower on average, within the deviations experienced during the tests[6].

For this test we ran several versions of Othello on an 8-core Clovertown machine, running with a lookahead of 7 moves. Performance results are the mean of three runs. This is using our implementation described in the next section.

- Serial: this is the original, unmodified version. Average execution time was 206 seconds.
- Elision: this is our parallel version with the new keywords `#define`d away. Average execution time: 197 seconds.
- GEN-1: our parallel version restricted to 1 thread. Average execution time: 215 seconds.
- Hand-8: hand-written TBB version, set to 8 threads. Average execution time: 37.6 seconds.
- GEN-8: generated TBB version, set to 8 threads. Average execution time: 37.8 seconds.
- Hand-auto: hand-written TBB version, let the number of threads up to TBB. Average execution time: 36.7 seconds.
- GEN-auto: hand-written TBB version, let the number of threads up to TBB. Average execution time: 37.2 seconds.

In the comparisons of generated code vs. hand-written code, the hand-written code wins, but only by a tiny margin. In addition, the C++ elision actually performed slightly better than the serial version; we are not sure why.

[6]Bear in mind, however, that the transformations were designed with this assignment fresh in memory, so the generated code is very similar to the hand-written code. It's fairly dangerous to generalize this to conclude that it would do well for *any* TBB usage. However, we *do* think that it would do well over a wide variety of applications.

5 Implementation

We have a prototype implementation of the transformation of `parallel_while` only. Currently our C++ front end will parse `concurrent_for`, but it performs no transformations on it.

Our project builds off of a C++ front end called Elsa, written by Scott McPeak. Elsa began as a proof-of-concept of the utility of a generalized LR parser-generator called Elkhound [7], but has since matured into a full-fledged project. It is now being maintained as part of a set of projects called Oink ([10]), maintained by Daniel Wilkerson and Karl Chen. Elsa allows us to do C++ source-to-source transformations with reasonable ease, though the pretty-printing feature required a good bit of work to bring up to a usable state.

The author used Elsa in a previous project done with Gregory Cooksey, during which we patched around two dozen bugs in the pretty-printing ability of Elsa. For a number of reasons, we never integrated most of these back into the original project, which meant that the first step of this project was to reapply the patches we developed last year. Unfortunately we (mostly I) weren't very disciplined about our source control management for the early part of the project, and often committed several fixes at once, bug fixes at the same time as code for our project proper, and fixes for one bug that accidentally reverted earlier fixes requiring us to reapply patches[7]. Thus the first part of the project was teasing apart the old patches and applying them to the updated version of Elsa. The alternative, applying the changes to Elsa made in the last year, would also have worked, a secondary goal of this project was to provide an excuse to extract all those patches so we can send them upstream, which is something that has been in the back of my mind for a long time. The good news along this front is that we only found about five new bugs during this project, compared to over thirty last year.

The other part of the implementation is a bit of glue. There is a wrapper script for GCC that makes it possible to simply replace the `g++` on the command line. For instance, instead of saying `g++ othello.cc -O2 -g -o othello`, it is possible to say `tbbetter othello.cc -O2 -g -o othello` and the code will be preprocessed by our transformer. This is also adapted from our previous project, which in turn adapted it from Ben Liblit's CBI project where it was

[7]This mess, and the inability to send "here's a patchfile for your ticket number 132" is part of the reason these didn't move upstream.

developed for Liblit's own source-to-source transformation [6].

6 Discussion

There are two interesting points not previously discussed. The first is a sequence of steps leading to the fact that thread-local variables cannot be changed. The second is discussion of how the next version of the C++ standard will impact this work.

6.1 Thread-locals are const

There is an interesting property that comes out of the transformation along with TBB's own requirements regarding the use of thread-local variables.

Consider a loop that the programmer wants to run in parallel. If it is a `for` loop, it must have no loop-carried dependences. If it is a `while` loop, it can have some, but they must all be within the generator portion. If this is not true, what happens during execution of the parallel version? Iteration i depends on $i-1$, but there is no guarantee that i will execute after $i-1$. We suspect because of this reason, TBB requires that the functions that implement the loop body are `const`.

However, when our transformer operates, it turns unshared variables that are used in the body but declared outside the body into class members. This means that the body object cannot modify some local variables! This seems counterintuitive, but this is actually a desirable property for the reasons given above.

In section 4, when we moved the declarations of `b` and `quality` into the loop, this is why. Had we left them as-is, it would have caused a compiler error when it tried to write to them from within a const method.

Finally, note that `tbb_shared` variables are not affected by this because they are modified through a reference, nor is the generator object for `concurrent_while` because `pop_if_present` is not declared `const`.

7 C++0x

The next version of the C++ standard is due out in the next year or three, and there are a couple of aspects of it that will impact this work.

First, there is a possibility that TBB will grow in importance. There is a proposal that would add to the standard library what is essentially a substantial portion of TBB [8]. However, there is not an accompanying proposal to modify the language of C++, which will leave developers in the same position as TBB developers are now.

Second, there is a good chance that C++0x will incorporate a proposal to add anonymous functions (lambdas) and closures to the language [11]. This is actually very exciting, because it means that a tool like this work describes will cease to be very important, which would be very helpful for the adoption of TBB or a similar library. Closures would virtually eliminate the high-level syntactic issues for loops, which we said were very important, at the expense of some additional low-level syntax, which we said is largely unimportant. Closures would allow the code that comprises a loop to stay where the loop logically belongs, and it would avoid the boilerplate code that is saving locals in members.

The following is a guess of what TBB could look like using the proposed closure syntax, parallelizing the same loop as before:

```
 1  auto gen = <&>(OthelloMove& move) -> bool
 2  {   if(moves.size() > 0) {
 3          move = moves.front();
 4          moves.pop();
 5          return true;
 6      }
 7      else {
 8          return false;
 9      }
10  }
 1  auto body = <&>(OthelloMove move)
 2  {   OthelloBoard b = board;
 3      b.applyMove(move);
 4
 5      quality = Lookahead(
             b, other_color, newdepth );
 6
 7      if( quality > nBest ) {
 8        scoped_lock l(lock);
 9        nBest = quality;
20      }
 1  }
 2  parallel_while(gen, body);
```

The new syntax appears on lines 1 and 11. First, the `auto` keyword, traditionally used to indicate that a variable should have automatic storage duration but long-ignored since it's done automatically, has been overloaded. (This is actually a different proposal to the C++ committee, also very likely to be accepted.) It now declares a variable whose type is whatever the type of the initializer is. For instance, `auto i = 3;` declares an integer, and `auto f = 4 + 2.0;` declares a floating point number. Using `auto` in this way avoids the need to explicitly write the types of `gen` and `auto`.

The other new syntax is `<&>`, which specifies the start of the definition of a closure. The `(OthelloMove& move)` on line 1 and similar bit of line 11 defines the arguments to the closure. The closure on line 1 has the same type as `pop_if_present` does now, and the closure on line 11 is the same as the body's `operator()`. Furthermore, the closure beginning on line 1 returns a `bool`.

Both closures have access to the enclosing function's local variables.

Closures will make dealing with a TBB-like library far easier than the present situation. However, there are a couple reasons why our tool is still interesting. First, it's likely going to be a long time before this is actually usable. The standard has to be ratified then compiler vendors have to actually implement it. And for a long time after that, people will still want to use older versions. For instance, despite GCC 4 being out for a long time, the default on the CSL machines is still 3.4. Despite being several years out of date, Visual C++ 6.0 is still not terribly uncommon. The second reason is that even though closures represent a vast improvement over the present situation, we think our syntax is still notably better than closures, though a question of whether it is better *enough* to justify the loss of standards compliance is a different matter.

8 Conclusion

In this paper, we presented arguments in favor of the use of TBB, but also our objections to its syntax. To rectify this problem, we proposed a new syntax that extends C++ with parallel programming constructs, and explained how to do a relatively simple source-to-source transformation to reduce them to straight C++. We briefly mentioned our present prototype which performs the transformation for `concurrent_while`, and showed that for our simple benchmark, it produced equivalent results to a hand-written version with far less work.

If this project were to be continued in the future, there are a number of things to do. From more of a research standpoint, `parallel_reduce` and `parallel_scan` were almost completely unconsidered by us, and developing a syntax and transformation for them is left as future work. There is also a better evaluation that should be done, with a broader base of programs. One useful source of comparison material is a contest Intel had for TBB programs [4]. It would be useful to compare these hand-optimized programs to ones where the transformations were done automatically. Finally, modifications to TBB itself to allow something like inlets and aborts would be useful even without these transformations.

From an engineering standpoint, there's a lot that can be done:

- Finish the transformation for `concurrent_for`. Currently it just parses.
- Implement transformations for Cilk code.
- Improve edge cases. For instance, `concurrent_while`s can currently not be nested. This is largely an implementation artifact (though the implicit nature of determining the `cwhile_iterator` a loop uses is equally responsible); there just hasn't been reason to do this yet.
- Bring Elsa up to production quality. Fix the rest of the bugs mentioned earlier.
- Implement wrappers for other compilers. One of the big benefits of doing this as a source-to-source transformation is that the transformer can theoretically be put in front of any compiler. However, currently there is only a driver script for GCC. Also, working on other compilers in Elsa's current state would almost certainly expose many new bugs.

References

[1] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 1995), ACM, pp. 207–216.

[2] Dagum, L., and Menon, R. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng. 5*, 1 (1998), 46–55.

[3] Intel Corporation. Intel Thread Building Blocks 2.0 for open source. http://threadingbuildingblocks.org/. Accessed December 20, 2007.

[4] Intel Corporation. Coding with Intel TBB sweepstakes. http://softwarecommunity.intel.com/articles/eng/1515.htm, September 2007. Accessed December 20, 2007.

[5] Intel Corporation. *Intel Thread Building Blocks Tutorial*, 1.6 ed. Intel Corporation, 2007.

[6] Liblit, B. The cooperative bug isolation project. http://www.cs.wisc.edu/cbi/. Accessed December 20, 2007.

[7] McPeak, S. Elkhound: A fast, practical GLR parser generator. Tech. Rep. UCB/CSD-2-1214, Computer Science Division, University of California, Berkeley, December 2002. Also available at http://www.cs.berkeley.edu/~smcpeak/papers/elkhound_cc04.ps.

[8] Robison, A. D. A proposal to add parallel iteration to the standard library. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2104.pdf, September 2006. Accessed December 20, 2007.

[9] Supercomputing Technologies Group. *Cilk 5.4.6 Reference Manual*. MIT Laboratory for Computer Science, 1998–. Available online at http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf.

[10] Wilkerson, D., Chen, K., and McPeak, S. Oink: a collaboration of C++ static analysis tools. http://www.cubewano.org/oink/. Accessed December 20, 2007.

[11] Willcock, J., Järvi, J., Gregor, D., Stroustrup, B., and Lumsdaine, A. Lambda expressions and closures for c++. http://www.research.att.com/~bs/N1968-lambda-expressions.pdf, February 2006.

A Hand-written TBB code

The following is hand-written `parallel_while` code for the Othello example discussed in section 4. The first bit is the code that actually executes the loop, and takes the place of the loop body:

```
parallel_while<LookaheadMoveEvaluator> w;
QueueGenerator g(moves);
LookaheadMoveEvaluator e(nBest, board, other_color, newdepth);
w.run(g, e);
```

Then is the definition of the generator class:

```
class QueueGenerator
{
    queue<OthelloMove> & _moves;

  public:
    QueueGenerator(queue<OthelloMove> & moves)
      : _moves(moves)  { }

    bool pop_if_present(OthelloMove & out_move)
    {
       if(_moves.empty())
          return false;
       else {
          out_move = _moves.front();
          _moves.pop();
          return true;
       }
    }
};
```

And finally the body class:

```
class LookaheadMoveEvaluator
{
    atomic<int> & _nBest;
    OthelloBoard const & _board;
    char _otherColor;
    int _newDepth;

  public:

    LookaheadMoveEvaluator(atomic<int> & nBest,
                           OthelloBoard const & board,
                           char otherColor,
                           int newDepth)
      : _nBest(nBest)
      , _board(board)
      , _otherColor(otherColor)
      , _newDepth(newDepth)
    {}

    void operator() (OthelloMove & move) const
    {
        OthelloBoard b = _board;
        b.applyMove(move);

        int quality = Lookahead( b, _otherColor, _newDepth );
        int curNBest = _nBest;

        while(quality > curNBest &&
              _nBest.compare_and_swap(quality, curNBest) != curNBest)
        {
          curNBest = _nBest;
        }
    }

    typedef OthelloMove argument_type;

};
```