Task Region | N3832

Pablo Halpern   Arch Robison   {pablo.g.halpern, arch.robison}@intel.com
Hong Hong   Artur Laksberg   Gor Nishanov   Herb Sutter
{honghong, arturl, gorn, hsutter}@microsoft.com

2014-01-17

1 Abstract

This paper introduces C++ library functions `task_region`, `task_run` and `task_wait` that enable developers to write expressive and portable parallel code.

This proposal subsumes and improves the previous proposal, N3711.

2 Motivation and Related Proposals

The Parallel STL proposal N3724 augments the STL algorithms with the inclusion of parallel execution policies. Programmers use these as a basis to write additional high-level algorithms that can be implemented in terms of the provided parallel algorithms. However, the scope of N3724 does not include lower-level mechanisms to express arbitrary fork-join parallelism.

Over the last several years, Microsoft and Intel have collaborated to produce a set of common libraries known as the Parallel Patterns Library (PPL) by Microsoft and the Threading Building Blocks (TBB) by Intel. The two libraries have been a part of the commercial products shipped by Microsoft and Intel. Additionally, the paper is informed by Intel's experience with Cilk Plus, an extension to C++ included in the Intel C++ compiler in the Intel Composer XE product.

The `task_region`, `task_run` and the `task_wait` functions proposed in this document are based on the `task_group` concept that is a part of the common subset of the PPL and the TBB libraries. The paper also enables approaches to fork-join parallelism that are not limited to library solutions, by proposing a small addition to the C++ language.

A previous proposal, N3711, was presented to the Committee at the Chicago meeting in 2013. N3711 closely follows the design of the PPL/TBB with slight modifications to improve exception safety.

This proposal adopts a simpler syntax than N3711 – one that eschews a named object in favor of three namespace-scoped functions. It improves N3711 in the following ways:

- The exception handling model is simplified and more consistent with normal C++ exceptions.
- Strict fork-join parallelism is now enforced at compile time.
- The syntax allows scheduling approaches other than child stealing.

We aim to converge with the language-based proposal for low-level parallelism described in N3409 and related documents.

3 Introduction

Consider an example of a parallel traversal of a tree, where a user-provided function `compute` is applied to each node of the tree, returning the sum of the results:

```
template<typename Func>
int traverse(node *n, Func&& compute)
{
    int left = 0, right = 0;

    task_region([&] {
        if (n->left)
            task_run([&] { left = traverse(n->left, compute); });
        if (n->right)
            task_run([&] { right = traverse(n->right, compute); });
    });

    return compute(n) + left + right;
}
```

The example above demonstrates the use of two of functions proposed in this paper, `task_region` and `task_run`.

The `task_region` function delineates a region in a program code potentially containing invocations of tasks spawned by the `task_run` function.

The `task_run` function spawns a *task*, a unit of work that is allowed to execute in parallel with respect to the caller. Any parallel tasks spawned by `task_run` within the `task_region` are joined back to a single thread of execution at the end of the `task_region`.

`task_run` takes a user-provided function object `f` and starts it asynchronously – i.e. it may return before the execution of `f` completes. The implementation's scheduler may choose to run `f` immediately or delay running `f` until compute resources become available.

`task_run` can be invoked (directly or indirectly) only from a user-provided function passed to `task_region`, otherwise the behavior is undefined:

```
void g();

void f()
{
    task_run(g);    // OK, invoked from within task_region in h
}

void h()
{
    task_region(f);
}

int main()
{
    task_run(g);    // No active task_region. Undefined behavior.
    return 0;
}
```

4 Task Parallelism Model

4.1 Strict Fork-join Task Parallelism

The model of parallelism supported by the constructs in this paper is called *strict fork-join task parallelism*, which has decades of research behind it and is the form of structured parallelism supported by all of the prominent parallel languages, including Cilk, X10, Habanero, and OpenMP. These languages can be subdivided into two groups: those with *fully-strict* semantics and those with *terminally-strict* semantics.

In a fully-strict computation (Cilk and Cilk Plus), a task cannot complete until it has joined with all of its immediate child tasks. This form of fork-join parallelism provides strong guarantees and make a program easy to reason about. A child task can be written with the assumption that its parent will still be running, just as if it were called synchronously. Fully-strict semantics diverge less from serial computation than probably any other parallel semantics while admitting powerful parallelism. The `traverse` example above is a fully-strict computation.

In a terminally-strict computation (X10, Habanero, and OpenMP), parallelism is expressed as a set of nested regions (`finish` blocks in X10) which delimit the scope of parallel work. A task will join with an ancestor task at the end the nearest enclosing parallel region (i.e., the `finish` block that has been most recently entered and not yet exited). Unlike the case of fully-strict semantics, a task can exit without joining with its children. Terminally-strict semantics are sometimes more convenient for programmers because they allow arbitrary refactoring of code without concern for parent-child task boundaries. On the other hand, the relaxed nesting compared to fully-strict languages makes the code harder to reason about, since a function can effectively return before it is finished (i.e., while sub-tasks are still running). The following variation of the `traverse` example is a terminally-strict computation. Note that, since none of the sub-computations is known to complete until the top-level `task_region` completes, none of the intermediate computations can depend on the results of child-computations:

```
template<typename Func>
void traverse(node *n, Func&& compute)
{
    if (n->left)
        task_run([&] { traverse(n->left, compute); });
    if (n->right)
        task_run([&] { traverse(n->right, compute); });

    n->value = compute(n->value);

    // left and right traverse() calls are NOT joined on return.
}

template<typename Func>
void do_traverse(node *n, Func&& compute)
{
    task_region([&]{ traverse(n, compute); });
}
```

The constructs in this paper have terminally-strict semantics for two reasons: 1) It is easier than fully-strict semantics to express using a library syntax and 2) some of the authors felt that it added important convenience for the programmer. In the Issues section, however, we describe some ways in which we might mitigate the disadvantages of terminally-strict semantics, possibly getting the best of both worlds.

It is important to note that strict semantics is not a given in parallelism proposals. Coming from a background in concurrency, many people look to unstructured constructs such as `std::async`, citing their flexibility vs. the comparative rigidity of strict parallelism. While these constructs have their place, the research has shown that fine-grain, large-scale parallelism benefits from a highly-structured approach. Just as a compiler can implement much more efficient memory allocation for local variables, which are highly structured, than for (unstructured) heap-allocated variables, so, too, can a compiler and scheduler take advantage of the structure of strict fork-join parallelism to implement efficient queuing and scheduling of parallel tasks. The algorithms described in N3724 and successor proposals neither require nor benefit from unstructured parallelism.

4.2 Non-mandatory Parallelism

Whereas concurrency constructs such as threads, producer-consumer queues, and the like are primarily about program *structure*, parallelism constructs of the kind presented in this paper are primarily about maximum exploitation of available hardware resources to achieve *performance*. Critical to this distinction is that, while separate threads are independently expected to make forward progress, parallel tasks are not.

The most common scheduling technique for fork-join parallelism is called work-stealing. In a work-stealing scheduler, each hardware resource (usually a CPU core) maintains a queue (which may or may not be FIFO) of tasks that are ready to run. If a CPU's queue becomes empty, it "steals" a task from the queue of some other CPU. In this way, the CPUs stay busy and process their work as quickly as possible. Conversely, if all of the CPUs are busy working on their own tasks, then those tasks will be executed serially until the queues are empty. In fact, if the operating system allocates only one CPU to a process, then the entire parallel computation would be completed on a single core. This quality allows a program to scale efficiently from one core to many cores without recompilation.

The constructs in this paper allow a programmer to indicate tasks that are *permitted* to run in parallel, but does not *mandate* that they actually *run* concurrently. An important consequence of this approach, known as "serial semantics," is that a task in the queue will not make any forward progress until another task (on the same or different core) completes, whether or not there is a dependency relationship between them. Thus, using concurrency constructs such as producer-consumer queues between parallel tasks (including between parent and child tasks or between sibling tasks) is a sure way to achieve deadlock. If the program is not valid as a serial program, then it is not valid as a parallel program, either.

5 Interface

The proposed interface is as follows. With the exception of `task_region_final`, the implementation of each of the functions defined herein is permitted to return on a different native thread than that from which it was invoked. See Thread Switching in the issues section for an explanation of when this matters and how surprises can be mitigated.

5.1 Header `<experimental/parallel_task>` synopsis

```
namespace std {
namespace experimental {
namespace parallel {

    class task_cancelled_exception;

    template<typename F>
      void task_region(F&& f);
    template<typename F>
      void task_region_final(F&& f);

    template<typename F>
      void task_run(F&& f);
    void task_wait();

}
}
}
```

5.2 Class `task_cancelled_exception`

```
class task_cancelled_exception : public exception {
public:
    task_cancelled_exception() noexcept;
    task_cancelled_exception(const task_cancelled_exception&) noexcept;
    task_cancelled_exception& operator=(const task_cancelled_exception&) noexcept;
    virtual const char* what() const noexcept;
};
```

The class `task_cancelled_exception` defines the type of objects thrown by `task_run` or `task_wait` if they detect that an exception is pending within the current parallel region. See Exception Handling, below.

5.3 Function template `task_region`

```
template<typename F>
  void task_region(F&& f);
template<typename F>
  void task_region_final(F&& f);
```

Requires: `F` shall be `MoveConstructible`. The expression, `(void) f()`, shall be well-formed.

Effects: Invokes the expression `f()` on the user-provided object, `f`.

Throws: `exception_list`, as specified in Exception Handling.

Postcondition: All tasks spawned from `f` have finished execution. `task_region_final` always returns on the same native thread as that on which it was invoked. (See Thread Switching in the Issues section.)

Notes: It is expected (but not mandated) that `f` will (directly or indirectly) call `task_run`.

5.4 Function template `task_run`

```
template<typename F>
  void task_run(F&& f);
```

Requires: The invocation of `task_run` shall be directly or indirectly nested within an invocation of `task_region`. `F` shall be `MoveConstructible`. The expression, `(void) f()`, shall be well-formed.

Effects: Causes the expression `f()` to be invoked asynchronously at an unspecified time prior to the next invocation of `task_wait` or completion of the nearest enclosing `task_region`, where "nearest enclosing" means the `task_region` that was most recently invoked and which has not yet returned. [*Note:* the relationship between a `task_run` and its nearest enclosing `task_region` is similar to the relationship between a `throw` statement and its nearest `try` block. – *end note*]

Throws: `task_cancelled_exception`, as defined in Exception Handling.

Remarks: The invocation of the user-supplied invocable, `f`, may be immediate or may be delayed until compute resources are available. `task_run` might or might not return before invocation of `f` completes.

`task_run` may be called directly or indirectly from a function object passed to `task_run`. The nested task started in such manner is guaranteed to finish with rest of the tasks started in the nearest enclosing `task_region` [*Note:* An implementation is allowed to join tasks at any point, for example at the end of the enclosing `task_run`. – *end note*]

[*Example*:

```
task_region([] {
    task_run([] {
        task_run([] {
            f();
        });
    });
});
```

– *end example*]

5.5 Function `task_wait`

```
void task_wait();
```

Requires: The invocation of `task_wait` shall be directly or indirectly nested within an invocation of `task_region`.

Effects: Blocks until the tasks spawned by the nearest enclosing `task_region` have finished.

Throws: `task_cancelled_exception`, as defined in Exception Handling.

Postcondition: All tasks spawned by the nearest enclosing `task_region` have finished.

[*Example:*

```
task_region([&]{
    task_run([&]{ process(a, w, x); }); // Process a[w] through a[x]
    if (y < x) task_wait();             // Wait if overlap between [w,x) and [y,z)
    process(a, y, z);                   // Process a[y] through a[z]
});
```

– *end example*]

6 Exception Handling

Every `task_region` has an associated exception list. When the `task_region` starts, its associated exception list is empty.

When an exception is thrown from the user-provided function object passed to `task_region`, it is added to the exception list for that `task_region`. Similarly, when an exception is thrown from the user-provided function object passed into `task_run`, the exception object is added to the exception list associated with the nearest enclosing `task_region`. In both cases, an implementation may discard any pending tasks that have not yet been invoked. Tasks that are already in progress are not interrupted except at a call to task_run or task_wait, as described above.

If the implementation is able to detect that an exception has been thrown by another task within the same nearest enclosing `task_region`, then `task_run` or `task_wait` may throw `task_cancelled_exception`.

When `task_region` finishes with a non-empty exception list, the exceptions are aggregated into an `exception_list` object (defined below), which is then thrown from the `task_region`.

Instances of the `task_cancelled_exception` exception thrown from `task_run` or `task_wait` are not added to the exception list of the corresponding `task_group`.

The order of the exceptions in the `exception_list` object is unspecified.

The `exception_list` class was first introduced in N3724 and is defined as follows:

```
class exception_list : public exception
{
public:
    typedef exception_ptr value_type;
    typedef const value_type& reference;
    typedef const value_type& const_reference;
    typedef size_t size_type;
    typedef vector<exception_ptr>::iterator iterator;
    typedef vector<exception_ptr>::const_iterator const_iterator;

    exception_list(vector<exception_ptr> exceptions);

    size_t size() const;
    const_iterator begin();
    const_iterator end();
private:
    // ...
};
```

7 Scheduling Strategies

A possible implementation of the `task_run` is to spawn individual tasks and immediately return to the caller. These *child* tasks are then executed (or *stolen*) by a scheduler based on the availability of hardware resources and other factors. The parent thread may participate in the execution of the tasks when it reaches the join point (i.e. at the end of the execution of the function object passed to the `task_region`). This approach to scheduling is known as *child stealing*.

Other approaches to scheduling exist. In the approach pioneered by Cilk, the parent thread immediately executes the spawned task at the spawn point. The execution of the rest of the function – i.e., the *continuation* – is stolen by the scheduler if there are hardware resources available. Otherwise, the parent thread returns from the spawned task and continues as if it had been a normal function call instead of a spawn. This approach to scheduling is known as *continuation stealing* (or *parent stealing*).

Both approaches have advantages and disadvantages. It has been shown that the continuation stealing approach provides lower asymptotic space guarantees and prevents threads from stalling at a join point. Child stealing is generally easier to implement without compiler involvement. N3872 provides a worthwhile primer that addresses the differences between, and respective benefits of, these scheduling approaches.

It is the intent of this proposal to enable both scheduling approaches and, in general, to be as open as possible to additional scheduling approaches.

8 Issues

The constructs proposed in this paper have a strong theoretical foundation from previous work on language-based parallelism such as Cilk, X10, and Habanero. However, there are some practical issues that arise from trying to harmonize these constructs with the existing C++ threading model. For example, the terminal strictness of X10 and Habanero is more difficult to achieve in C++ because the strict scoping of C++ function variables is less forgiving than the garbage collected variables in the other two languages.

This section describes a couple if important issues along with some discussion of how they might be addressed.

8.1 Thread Switching

One of the properties of continuation stealing and greedy scheduling is that a `task_region` or `task_run` call might return on a different thread than that from which it was invoked, assuming scheduler threads are mapped 1:1 to standard threads. This phenomenon, which is new to C++, can be surprising to programmers and break programs that rely on the OS thread remaining the same throughout the serial portions of the function (for example, in programs accessing GUI objects, mutexes, thread-local storage and thread ID).

There are a number of possible approaches to mitigate the problems caused by thread switching. In considering mitigation proposals, it is important to avoid overly-constraining future implementations in order to support today's limited view of threads. For example, this proposal does not require that parallelism be implemented using OS threads at all – it could be implemented using specialized hardware such as GPUs or using other OS facilities such as special light-weight threads.

Additionally, the solution to problems caused by thread switching may be different for mutexes than for thread-local storage (TLS) and thread ID. For example, a prototype mutex exists that works well with thread switching and has nice theoretical properties that allow it to be used for both parallelism and for traditional concurrency. The desired behavior for thread-local storage varies depending on its intended use, even in the absence of thread switching. For example, the handle to a GUI object might need to be shared among all of the tasks executing on behalf of a single original thread. Conversely, thread-local caches should not be shared between concurrently-executing tasks. With or without thread switching, we will certainly need new TLS-like facilities.

In this proposal, the `task_region_final` function template provides a minimal but powerful approach for addressing thread switching. Using this facility, a user can be sure that both thread-local variables and mutexes are in a consistant state before and after the execution of a parallel computation. This is expected to solve most of the issues that a user may run into in well-structured parallel code. Because `task_region_final` requires stalling at a join point, it can potentially reduce parallel speed-up. For this reason, our advice to users would be to use `task_region` except in circumstances where thread identity is important. In implementations that do not support greedy scheduling, the behavior of `task_region_final` would probably be identical to `task_region`.

We have also discussed the possibility of language or library constructs to mark a function as potentially returning on a different thread than that on which it was called. A straw-man proposal involves a `thread_switching` keyword that would be applied as a suffix in the declarator for such functions:

```
void f() thread_switching;
```

If function decorated with the `thread_switching` modifier were called from a function that did not have the modifier, the compiler would inject code at the call site that would, on return from the decorated function, stall the caller until the original thread became available to resume execution:

```
void f() thread_switching;

int main() {
    auto thread_id_begin = std::this_thread::get_id();
    f();
    auto thread_id_end = std::this_thread::get_id();
    assert(thread_id_end == thread_id_begin);
    return 0;
}
```

Alternatively, calling a decorated function from an undecorated function could simply be ill formed, requiring the programmer to call `task_region_final` explicitly to avoid an error:

```
void f() thread_switching;

void g() {
    f();                // ill-formed
}

void h() {
    task_region_final([&]{
        f();            // OK
    });
}
```

Other approaches are also being considered, including making a theoretical distinction between a “thread” as defined in C++11 and a “worker” as the agent that executes tasks. Making this distinction would solve certain problems with parallelism and thread identity, including issues of object and thread lifetimes that the `thread_switching` keyword does not address.

Final resolution of issues related to thread-switching may need to wait until a thorough discussion of “execution agents.”

8.2 Returning with Unjoined Children

The *terminally strict* semantics of the constructs proposed in this paper allow a function to return to the caller while some of its child tasks are still running. As in the case of thread switching, this behavior can be surprising to programmers and break programs that rely on functions finishing their work before they return. Although unstructured concurrency constructs such fire-and-forget threads already violate these assumptions, we are attempting, in this paper, to define much more structured constructs that operate at a finer granularity of work. Programmers writing *structured* parallel code need to be put on notice when a function invoked in their program might spawn parallel tasks and return without joining them first. The compiler may need to generate stack-allocated stack frames for such functions and/or its optimizer might be impaired in doing its job if it needs to defensively assume that any function might return with child tasks still running.

One approach to address this issue is to add a keyword similar to the `thread_switching` keyword described above. As in the case of the `noexcept` keyword, both the progammer and the compiler know what to expect of the function and can take appropriate action. If an undecorated function were to call a decorated function, the compiler could insert a join point on return from the decorated function. Alternatively, the program could be ill-formed unless the call to the decorated function is nested in a `task_region`.

As a special case, a lambda expression that directly calls a decorated function could be considered to inherit the decoration. This special rule would make it possible to avoid the explicit decoration for a lambda expression passed into `task_region` while still invoking a decorated function:

```
void f() unjoined_children;
task_region([] {
    f(); // OK, surrounding lambda implicitly has unjoined_children
});
```

This special case would apply to the `thread_switching` keyword, as well.

It has also been suggested that a single keyword like `thread_switching` could serve *both* purposes: to indicate that a function might return on a different thread *and* that it might have outstanding children. Unifying these keywords minimizes language changes but also limits the expressiveness of the interface. For example, while many functions in a parallel call hierarchy may want to return on a different thread (to avoid stalling overhead), most would probably not need to return with children still running. If both concepts were expressed with a single keyword, neither the user nor the compiler would be able to take advantage of this distinction. It has also been suggested that these concepts could somehow be combined with the `resumable` keyword being proposed for resumable functions, though it is not yet clear how this combination would work.

8.3 Moving Forward with Unresolved Issues

The straw-man proposals described briefly above are not the only possible ways to address the issues of thread switching and unjoined children. The authors continue to research a number of alternatives, both with and without language support, and we seek committee guidance in the form of feedback on the ideas we've presented as well as additional ideas. We believe, however, that working through these issues will have only a modest impact on the library interfaces described in this paper and that these interfaces should therefore be used as the basis for adding strict fork-join library constructs to the parallelism TS.

9 References

TBB Threading Building Blocks (TBB)

PPL Parallel Patterns Library (PPL)

Cilk The Cilk Project

Cilk Plus cilkplus.org home page

X10 X10 Home Page

Habanero Habanero Extreme Scale Software Research Project

OpenMP openmp.org home page

N3724 *A Parallel Algorithms Library*, J. Hoberock, O. Giroux, V. Grover, H. Sutter, et al., 2013-08-30

N3711 *Task Groups As a Lower Level C++ Library Solution To Fork-Join Parallelism*, A. Laksberg, H. Sutter, 2013-08-15

N3409 *Strict Fork-Join Parallelism*, Pablo Halpern, 2012-09-24

N3872 *A Primer on Scheduling Fork-Join Parallelism with Work Stealing*, Arch Robison, 2014-01