VISUAL STUDIO 2010

PARALLEL PATTERNS LIBRARY, ASYNCHRONOUS AGENTS LIBRARY, & CONCURRENCY RUNTIME

PATTERNS AND PRACTICES

Bill Messmer
Parallel Computing Platform
Microsoft Corporation

TABLE OF CONTENTS

INTRODUCTION

In the last several years, computers with multiple processing cores have moved from the realm of servers and high end workstations into the mainstream market. Today, even inexpensive laptops include dual core processors. It is not uncommon to see desktop machines with four cores and eight hardware threads. No longer is it the case that each successive generation of hardware pushes further ahead with ever-increasing clock speeds. Instead, the number of hardware threads on successive generations of hardware has been increasing and will continue to do so. The way in which software is written is fundamentally changing in order to accommodate this shift in processor architecture. Software written in serial fashion simply will not continue to experience performance gains as generations of hardware advance. Instead, parallelism is being exploited.

Traditionally, developers have written to a programming model of explicit threading to take advantage of multi-core hardware. Visual Studio 2010, however, includes a number of new libraries designed to raise the level of abstraction and make the development of parallel software easier for the mainstream. For native code written in C++, the Parallel Patterns Library, the Asynchronous Agents Library, and the Concurrency Runtime are these libraries.

While these libraries may themselves present powerful abstractions to take advantage of parallelism, there are times it may not be entirely obvious how to apply these abstractions or what their caveats may be. This document details a series of patterns around the application of the constructs presented in the new libraries in Visual Studio 2010 as well as ways in which their use may be less than optimal.

PARALLEL LOOPS

One of the most common patterns in parallel programming is that of the so called “embarrassingly parallel” problem. Characterized by a set of largely independent work, such problems are generally very easy to parallelize. Often times, such problems present as loops with no loop carry dependencies between iterations. Consider a very simple serial example:

```
for(int y = 0; y < ySize; y++)
{
    for(int x = 0; x < xSize; x++)
    {
        Complex c(minReal + deltaReal * x,
                  minImaginary + deltaImaginary * y);

        Color pixColor = ComputeMandelbrotColor(c);

        …
    }
}
```

Figure 1

This is a classic example of computation of the Mandelbrot Set fractal. The iterations of the loops are entirely independent from any other iteration. Loops of this form are said to have no loop carry dependencies. As such, this problem falls into the “embarrassingly parallel” class. It is arguably the easiest of many patterns that can be effectively parallelized using Visual Studio 2010’s Parallel Patterns Library.

The PPL provides a series of APIs to allow for easy parallelization of loops such as those shown above. When utilizing these APIs to parallelize loops, it is important to note that the parallelization changes the sequential semantics of the loop. For loops such as the above which have no carry dependencies and no side-effects, this may not be significant. For less “embarrassingly parallel” loops, however, there are several important things to note:

- Because the loop is parallelized, the ordering of iterations of the loop is no longer guaranteed. Any side-effects contained within the loop may execute in arbitrary order.

- Since multiple iterations of the loop may execute simultaneously, exception flow of such loops is subtly different. Multiple exceptions may occur simultaneously causing the runtime to choose one to propagate. Likewise, a single iteration throwing an exception to a catch handler outside the loop does not immediately stop other concurrency executing iterations. It cancels the loop; however, the effect of that is no longer immediate.

- Sequential methods of impacting loop control flow (e.g.: the C++ break and continue statements) no longer work on parallelized loops. Different mechanisms of controlling the loop need to be considered.

PARALLEL FOR

The first such loop parallelization API provided by the Parallel Patterns Library is the parallel for loop. There are several variants of this API as shown below:

```
template<typename _Index_type, typename _Function>
void parallel_for(_Index_type _First,
                  _Index_type _Last
                  _Index_type _Step,
                  const _Function& _Func);

template<typename _Index_type, typename _Function>
void parallel_for(_Index_type _First,
                  _Index_type _Last,
                  const _Function& _Func);
```

Figure 2

Both of these versions allow a for loop executing over a predetermined range [_First, _Last) to be parallelized using any object (a C++ functor) supporting a function call operator with the signatures shown below:

```
void operator()(_Index_type);
void operator()(const _Index_type&);
```

Typically, the functor supplied to the *parallel_for* API is a C++ lambda as shown below in a parallelization of the Mandelbrot example of Figure 1:

```
parallel_for(0, ySize, [=](int y)
{
    for(int x = 0; x < xSize; x++)
    {
        Complex c(minReal + deltaReal * x,
                  minImaginary + deltaImaginary * y);

        Color pixColor = ComputeMandelbrotColor(c);

        …
    }
});
```

Figure 3

A very simple syntax change to the original serial C++ implementation has allowed the implementation to be parallelized by the Concurrency Runtime. Likewise, the loop could be further decomposed into the following:

```
parallel_for(0, ySize, [=](int y)
{
    parallel_for(0, xSize, [=](int x)
    {
        Complex c(minReal + deltaReal * x,
            minImaginary + deltaImaginary * y);

        Color pixColor = ComputeMandelbrotColor(c);

        …
    });
});
```

Figure 4

The PPL's parallel for implementation is designed with several things in mind:

- *Load balancing* -- a processor which is done with its assigned range of the parallel loop can help another processor by grabbing a portion of its assigned range on the next loop iteration.
- *Nested parallelism* – if you utilize a *parallel_for* within another *parallel_for*, they coordinate with each other to share scheduling resources.
- *Cancellation* – loops support early exit in a manner whereby iterations of the loop which have not yet started will not start once the decision to exit has been made

- *Exception handling* – if an iteration of the loop throws an exception, that exception will propagate to the thread calling the *parallel_for* API. In addition to moving the exception back to the original thread, any iteration which throws an exception causes the loop to be canceled. No other loop iterations will execute.

- *Cooperative blocking* – if a loop iteration blocks cooperatively, the range of iterations assigned to that processor can be acquired by other threads or processors.

- *Arbitrary types* – the type of the loop iteration variable is templatized to support any arbitrary user defined type which behaves according to the arithmetic concepts of an ordinal type.

PARALLEL FOR EACH

While the *parallel_for* API allows clients to parallelize simple iterative loops with predefined bounds, it does not address one of the most common ways "embarrassingly parallel" problems may show up in C++ code. The C++ standard template library provides a series of generic container types which are extensively used in production C++ code. Likewise, these containers provide a generic iteration pattern which allows for an operation to be applied to every element within the container.

Consider the STL algorithm *for_each*:

```
template<class _InIt, class _Fn1>
_Fn1 for_each(_InIt _First, _InIt _Last, _Fn1 _Func);
```

Figure 5

This might be used, for instance, to perform some computation on every element on a container. Consider the below:

```
std::vector<Object> someVector;

…

for_each(someVector.begin(), someVector.end(), [](Object& o){
    performComputation(o);
    });
```

Figure 6

If the *for_each* in Figure 6 has no loop-carry dependencies – that is that the **performComputation** call does not rely on computations on other objects in the container – then it, like the loop example in Figure 1, falls into the category of an “embarrassingly parallel” problem. The PPL provides a parallelized *for_each* algorithm, *parallel_for_each* which looks as follows:

```
template<typename _Iterator, typename _Function>
void parallel_for_each(_Iterator _First,
                       _Iterator _Last,
                       Const Function& _Func);
```

Figure 7

This can be used to parallelize the example of Figure 6 with a simple syntax change:

```
std::vector<Object> someVector;

…

parallel_for_each(someVector.begin(), someVector.end(), [](Object& o){
    performComputation(o);
    });
```

Figure 8

One important thing to note is that the container shown in Figure 8 is a **vector**. The iterators acquired from a **vector** are random access iterators and allow a *parallel_for_each* to parallelize iteration over the container very efficiently. The STL’s *for_each* algorithm supports iterators from random access iterators all the way to input iterators. The restrictions placed upon input iterators, however, make it impossible to effectively parallelize a fine-grained loop utilizing them. As such, the PPL’s *parallel_for_each* only supports iterators from random access down to forward iterators. While forward iterators have more restrictions than random access ones, the *parallel_for_each* algorithm is still able to parallelize loops across them albeit less efficiently than a parallelization across random access iterators.

The *parallel_for_each* algorithm is designed with the same goals in mind as those of the *parallel_for* algorithm. Loops using *parallel_for_each* support effective load balancing, nested parallelism, cancellation, exception handling and cooperative blocking.

PATTERNS

BREAKING OUT OF LOOPS

A common pattern in the execution of sequential loops is exiting the loop prior to the specified termination condition. A search of a data structure may, for example, utilize a loop to iterate through all nodes of the data structure breaking out of the loop when some predicate is matched. In sequential C++ code, the *break* statement fills this need. While the *break* statement does not work within a PPL parallel loop, the need for one does not disappear simply because the loop is parallelized. The same search code executed in sequential may be parallelized through use of the PPL's *parallel_for* or *parallel_for_each* construct.

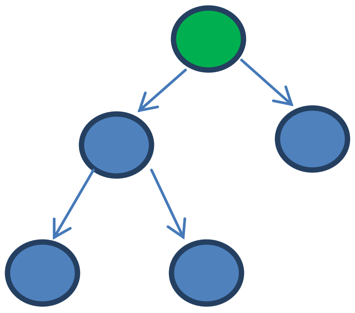
There are two main mechanisms for early termination of a parallel loop. The first is *cancellation* and the second is *exception handling*.

Loops within the PPL do not directly support the notion of cancellation. Only the *structured_task_group* and *task_group* constructs directly support the notion of canceling work. These two constructs (and any other fork-join construct based upon them) consider work to be part of a tree of tasks and groups. When new work is forked and joined, the group to which it belongs is conceptually considered to be a child of the group where the fork-join occurred. Consider:

```
parallel_for(0, 1, [](int i){
    if (i == 0)
    {
        parallel_for(0, 1, [](int j){
        });
    }
});
```

Figure 9

The tree constructed via the *parallel_for* loops above might look as follows to the runtime:



The root of the tree, colored in green above represents the outermost *parallel_for*. The left side of the tree again forks two tasks (the inner *parallel_for*). When a given group of work is canceled, the entire sub-tree of work rooted at that particular group is canceled.

As mentioned, the loop constructs within the PPL do not directly support cancellation semantics themselves. They do, however, respond to and participate in cancellations that occur higher in the work tree. If the intent of a loop is to be broken out of early, the loop itself can be wrapped in a *run_and_wait* call on a *structured_task_group* object to enable cancellation of the loop.

```
structured_task_group stg;
stg.run_and_wait([&d, &stg](){
    structured_task_group &ostg = stg;
    parallel_for_each(d.begin(), d.end(), [&ostg](T obj){
        if (matches_predicate(obj))
        {
            ostg.cancel();
        }
    });
```

Figure 10

One important note about utilizing the *cancel* method in this manner is that threads that are actively executing iterations of the loop will not stop until they reach an *interruption point*. The PPL's implementation of *parallel_for* places such a point between iterations of the loop. If the work within the body of each iterate is coarse enough, the loop body may need to have *interruption points* of its own or utilize some other mechanism such as the *is_current_task_group_canceling* method.

Exception handling presents a second possibility for early exit from a loop. Exceptions that flow through parallel constructs in the PPL cancel the work scheduled to those constructs implicitly. This means that if an exception flows out of a parallel loop and is caught immediately after the join point, it can be utilized to terminate the loop early.

```
try
{
    parallel_for_each(d.begin(), d.end(), [](T obj){
        if (matches_predicate(obj))
        {
            throw obj;
        }
    });
}
catch(const T&)
{
}
```

Figure 11

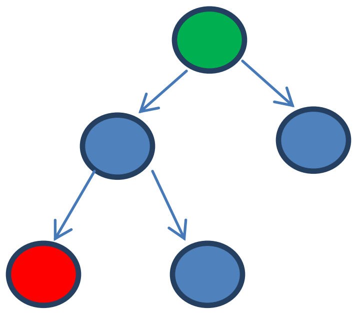
While the method of utilizing exceptions to “throw the answer” may seem simpler, it comes with some performance implications, especially if there are multiple levels of nested parallelism between the throw site and the catch site. Consider a parallel search of some graph structure:

```
void ParallelSearchHelper(Node **result,
                          structured_task_group *rootGroup,
                          Node *curNode,
                          Predicate pfn)
{
    if (pfn(curNode) &&
        InterlockedCompareExchangePointer(result,
                                          curNode,
                                          NULL) == NULL)
    {
        rootGroup->cancel();
    }
    else
    {
        parallel_for_each(curNode->successors.begin(),
                          curNode->successors.end(),
                          [=](Node *travNode){
            ParallelSearchHelper(result, rootGroup, travNode, pfn);
        });
    }
}

Node* ParallelSearch(Node* startNode, Predicate pfn)
{
    structured_task_group stg;
    Node *result = NULL;
    stg.run_and_wait([&stg, &result](){
        result = ParallelSearchHelper(&result, &stg, startNode, pfn);
        });
    return result;
}
```

Figure 12

Each level of the search executes a parallel_for_each to parallelize the graph search. A call to the *cancel* method as illustrated above in Figure 12 cancels nested work in a top down manner. Consider an unfolding of calls to *ParallelSearchHelper* as shown below. The original root call at the top of the tree is in green and the leaf node where the predicate matches is in red.



The call to the *cancel* method on the *structured_task_group* encapsulating the outermost *parallel_for_each* will traverse the work tree once pushing a cancellation message down the tree. If instead, the predicate match threw the result similarly to what is shown in Figure 11, the exception would propagate up the tree one level at a time. At each level of the tree, a cancellation would occur, traversing downward and sending cancellation messages. Depending on the depth of nested parallelism, this can result in significantly more traversals of the work tree and significantly more cancellation messages being sent.

Break out of a parallel loop by utilizing cancellation on an encapsulating task group. structured_task_group's run_and_wait and cancel methods provide an easy mechanism by which to do this.

ANTI-PATTERNS

SMALL LOOP BODIES

While the PPL and the ConcRT runtime aim to have minimal overhead, nothing is completely free. Loop constructs within the library are designed to support a set of features desirable for generalized use. Features such as *load balancing*, *cancellation*, and *cooperative blocking* have incrementally small overheads.

If an iteration of the parallel loop is medium or coarse grained, the overhead from supporting these features is negligible. Some loop parallelizations, however, have iterations which are so fine grained that the overhead of the runtime dwarfs the actual workload. In these cases, the parallelization of the loop may result in a performance that is worse than serial execution! Consider the following:

```
//
// Consider three std::vector<int> of equal size: a, b, c
//

parallel_for(0, size, [&a, &b, &c](int i){
    c[i] = a[i] + b[i];
    });
```

Figure 13

The parallel loop in Figure 13 is simply adding two large vectors. Here, the workload is nothing more than two memory reads, an addition, and a memory write. This is not enough to amortize the runtime overhead. Either the per iterate workload needs to be made coarser (e.g.: by manually chunking) or the algorithm used for parallelization needs to be changed. While the Visual Studio 2010 PPL does not provide a loop algorithm amenable to utilization in such scenarios, the sample pack (currently located at http://code.msdn.microsoft.com/concrtextras) does. *parallel_for_fixed* is a loop construct designed to forgo the extensive feature set of *parallel_for* and provide the a construct amenable to the parallelization of loops with small bodies. It makes a determination at invocation time of how to divide up the loop and statically schedules that many tasks. No effort is made for things such as *load balancing*. This operates very similarly to OpenMP's parallel for construct. Modifying the example of Figure 13 to the below will result in significantly increased performance.

```
//
// Consider three std::vector<int> of equal size: a, b, c
//

Concurrency::samples::parallel_for_fixed(0, size, [&a, &b, &c](int i){
    c[i] = a[i] + b[i];
    });
```

Figure 14

Note that while this version of a parallel loop does significantly reduce runtime overhead, it does not support any of the features that *parallel_for* does. It does not load balance (either between iterations or on blocking) and has no direct support for per-iterate cancellation. For some loops where the distribution is balanced, these features are not required, and each iteration is very fine grained, it is the right construct to utilize.

Consider carefully the size of the loop body when parallelizing a loop. If the workload is very small, consider carefully the decision to parallelize it. If parallel execution is still required, utilize constructs such as parallel_for_fixed to do the parallelization!

REPEATED BLOCKING

The scheduler underneath the PPL is cooperative by nature. It continues to run a given thread until that thread blocks, yields, or finishes executing. If a thread does block and there is work (e.g.: other iterations of a parallel loop) but no threads ready to run, the scheduler will by its nature attempt to create a new thread to pick up work.

For computational workloads with occasional blocking, this is an ideal strategy. For a workload which is dominated by repeated blocking, it is not. Such repeated blocking will lead the scheduler to create a large number of threads very rapidly.

Consider the following:

```
Concurrency::event e;
parallel_for(…, [&e](){
    do_something();
    e.wait();
    });
```

Figure 15

Each iterate of the parallel loop does something and waits for some event to become signaled externally. As the event waited upon is a synchronization primitive that is aware of the Concurrency Runtime, the scheduler is aware that the blocking has occurred and will attempt to create a new thread. The same is true no matter what runtime synchronization primitive the code above utilized.

While the example of Figure 15 may seem somewhat contrived, the concept behind it is certainly not. On platforms where user mode scheduling is enabled, any blocking in the system is considered cooperative blocking. The same problem that manifests with runtime synchronization primitives can manifest itself from repeated I/O within a parallel loop. Imagine using a *parallel_for_each* to download a series of files in parallel from high-latency network connections:

```
std::vector<std::string> urls;
parallel_for_each(urls.begin(), urls.end(), [](std::string url){
    download(url);
    });
```

Figure 16

If the network connection is high-latency, there will be long blocking operations in the download and the scheduler will create new threads to pick up other iterations. The number of threads created by the runtime will be bounded by

$$\frac{IO\ Latency}{Thread\ Creation\ Latency}$$

subject to some level of throttling decided by the scheduler. Because of the scenarios to which the scheduler is tuned, the number of threads created in this scenario will be far more than intended.

Do not repeatedly block on the same synchronization construct within a parallel loop!

BLOCKING BEFORE CANCELLATION

One of the many uses of parallel loops – particularly parallel loops which terminate early – is to parallelize some type of search. Consider the parallel search described in Figure 12 of the *Breaking Out of Loops* section. The search routine executes the following:

```
    if (pfn(curNode) &&
        InterlockedCompareExchangePointer(result,
                                           curNode,
                                           NULL) == NULL)
    {
        rootGroup->cancel();
    }
```

Figure 17

Immediately after determining that the current node matches the predicate, it performs a CAS operation and cancels the tree. Imagine instead that the code for returning the result were something more akin to the below:

```
if (pfn(curNode) && InterlockedExchange(&firstResult, TRUE) == FALSE)
{
    Answer *pAnswer = new Answer(curNode);
    //
    // Stash the answer somewhere
    //
    rootGroup->cancel();
}
```

Figure 18

Here, the answer to return is prepared via some other method which performs a heap allocation to return some complex data structure. At first glance, this seems like a perfectly reasonable approach. It does, however, come with a major caveat that can lead to very poor parallelized search performance. The call to *new* to perform the heap allocation may block. When user mode scheduling is enabled in the runtime, the runtime becomes responsible for rescheduling the thread when the thread unblocks. One of the runtime's heuristics is that recently unblocked threads are preferred over older ones. With many threads in the search potentially blocking and unblocking due to the fork/join nature of parallel loops, it is quite likely that there other threads executing search code will unblock between the time the thread executing *new* unblocks and the time the runtime becomes able to schedule a new thread. If this occurs, a substantial amount of wasted work can be performed searching for a result when one has already been found!

There are several possibilities to alleviate this problem in practice:

- During a search operation which performs cancellation, minimize blocking operations. If possible, avoid them.

- If blocking operations (or API calls into arbitrary code are necessary), minimize these -- ***<u>especially</u>*** in between checking the predicate and calling *cancel* (which prevents other threads from continuing expression of parallelism). Simply rewriting the code of Figure 18 like the below will eliminate the issue:

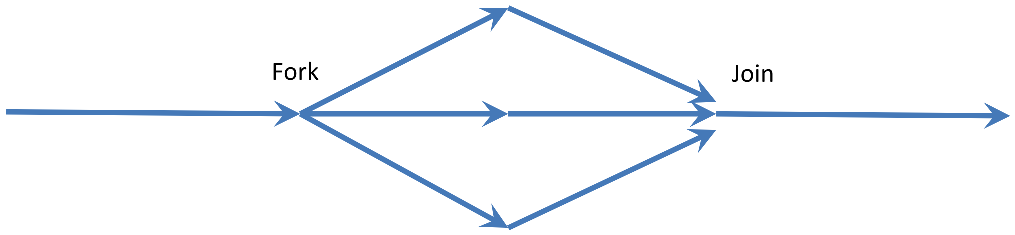
```
if (pfn(curNode) && InterlockedExchange(&firstResult, TRUE) == FALSE)
{
    rootGroup->cancel();
    Answer *pAnswer = new Answer(curNode);
    //
    // Stash the answer somewhere
    //
}
```

Figure 19

*In a cancellable algorithm, evaluate carefully any blocking calls placed before the method call to **cancel**. Consider one of the above recommendations for any such calls in your code.*

FORK-JOIN

A more general pattern which commonly shows up in the application of parallelism is that of "fork-join." From a single thread of execution, work is "forked" out to multiple threads where it executes in parallel. The original thread waits for all of the work to complete – the "join" point. This looks like what is shown below:



The pattern of "fork-join" is really a super-set of the parallel loops as discussed earlier. At the time a *parallel_for* or *parallel_for_each* is executed, iteration through the loop is potentially forked out to multiple threads which execute the iterations of the loop. The original thread executing the *parallel_for* or *parallel_for_each* algorithm does not proceed until all the work is complete (subject to potential load balancing, of course).

PARALLEL INVOKE

The Parallel Patterns Library provides several constructs which simplify the expression of fork-join parallelism other than parallel loops. The first of these is *parallel_invoke:*

```
template<typename _Function1, typename _Function2, …>
void parallel_invoke(const _Function1& _Func1,
                     const _Function2& _Func2,
                     …);
```

Figure 20

The *parallel_invoke* algorithm executes the functors passed to it in parallel as determined by the runtime. It is the preferred method by which to express fork-join parallelism that is static in its decomposition. The PPL also provides several more generalized constructs -- *structured_task_group* and *task_group* -- which can also be utilized to express fork-join parallelism albeit in a less simple manner.

PATTERNS

DIVIDE-AND-CONQUER (RECURSIVE DECOMPOSITION)

One of the most well-known patterns in programming is that of divide-and-conquer. Consider a classic example of a naïve serial quicksort:

```
template<typename T>
void quicksort(T* data, size_t l, size_t r)
{
    if (r - l <= 1)
        return;

    size_t pivot = partition(data, l, r);
    quicksort(data, l, pivot);
    quicksort(data, pivot + 1, r);
}
```

Figure 21

Each call to the **quicksort** method partitions the array into two segments: elements less than the pivot and elements greater than (or equal to) the pivot. After doing so, it recursively calls itself to continue to perform the same action until the array has been sorted.

The same notion can be applied to fork-join parallelism. In the above example, after the partition stage is complete, the left and right subsections of the array can be sorted totally independently. Thus, the quicksort could be rewritten as:

```
template<typename T>
void quicksort(T* data, size_t l, size_t r)
{
    if (r - l <= 1)
        return;

    size_t pivot = partition(data, l, r);

    parallel_invoke([=](){ quicksort(data, l, pivot); },
                    [=](){ quicksort(data, pivot + 1, r); });

}
```

Figure 22

Here, each call to quicksort forks off the left and right subsections in parallel and waits for the sorting to complete. As the quicksort unfolds recursively, there is increasing decomposition of the problem (more and more things can potentially execute in parallel).

There are several things to be aware of with the naïve sort example of the above. Many serial sorting implementations which are based upon recursive algorithms such as quicksort will stop the divide-and-conquer strategy at either some depth or at some chosen size of the array and switch to an alternative sort. This type of thinking is necessary in the parallel expression of the algorithm as well. The expression of parallelism through constructs such as *parallel_invoke* has some overhead associated with it. While the Parallel Patterns Library and the Concurrency Runtime strive to keep this overhead small, as the size of the workload gets smaller and smaller (the quicksort array smaller and smaller), the overhead becomes a greater portion of execution time and is that much more difficult to amortize. As well, in the serial algorithm, unbounded recursion would present potential for stack overflow. This is even more true in a version which utilizes *parallel_invoke* in unbounded fashion as the Concurrency Runtime will place its own data structures on the stack to express the opportunity for parallel execution.

Another issue to consider in parallelizing algorithms such as the above is that there is still some serial portion of the sort algorithm. While the two sub-arrays can be sorted in parallel, the partition step in the above example is still purely serial. Given that a portion of the algorithm executes serially, the scalability of the algorithm will be bounded by how much execution is serial.

For problems which lend themselves to being expressed in divide-and-conquer fashion, consider parallel_invoke as a parallelization strategy.

ANTI-PATTERNS

FREQUENT JOIN COLLAPSING

The “fork-join” style of parallelism makes it very easy to take a piece of code which today executes serially and gain performance benefit from parallel execution by compartmentalizing the parallelism. Imagine that profiling some application led to the realization that there was a particularly hot path in performing an image filter:

```
void PerformFilter(ImageData* img)
{
    for(int y = 0; y < img->ySize; y++)
    {
        for(int x = 0; x < img->xSize; x++)
        {
            // pixel filter
        }
    }
}
```

Figure 23

With a realization that the iterations of this loop are independent, the *parallel_for* algorithm can be applied for a painless parallelization of the algorithm:

```
void PerformFilter(ImageData* img)
{
    parallel_for(0, img->ySize; [=](int y)
    {
        for(int x = 0; x < img->xSize; x++)
        {
            // pixel filter
        }
    });
}
```

Figure 24

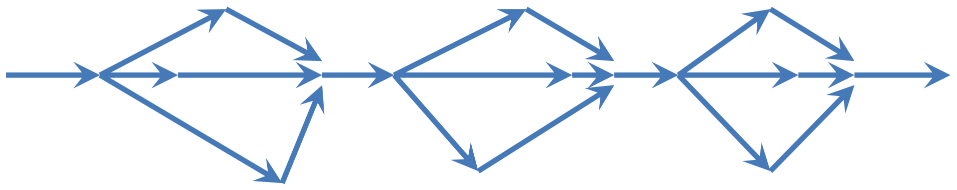
Recompiling and rerunning the application shows a significant improvement in performance. It seems that this optimization was a large win. Yet, as the application scales up to machines with many cores, the performance increase does not scale with it. Why?

Consider the context in which **PerformFilter** might be called. Imagine that the application were doing something similar to the below:

```
void ProcessImages()
{
    ImageData *img = GetNextImageBlock();
    while(img != NULL)
    {
        PerformFilter(img);
        …
        img = GetNextImageBlock();
    }
}
```

Figure 25

While the inner hot loop performing the actual processing was parallelized, the parallelism was compartmentalized within **PerformFilter**. The effect of the loop within **ProcessImages** of Figure 25 is that the application is repeatedly forking and joining parallelism. This might lead to something like the below:



The repeated forking and joining is causing significant overhead – particularly if there is ***any*** load imbalance in the forked work. Even with the load balancing inherent in a *parallel_for*, the balancing will not be perfect. The more cores in the system, the more overhead a join amongst them has. As well, the more cores in the system, the more small load imbalances will be magnified in such a scenario.

It is almost always beneficial to consider the introduction of parallelism at the highest level possible (e.g.: outer loops before inner ones, etc…). Expressing parallelism at very low levels can lead to inefficiencies for a variety of reasons. At low levels and inner loops, the workload size becomes smaller – hence it becomes more difficult to amortize the runtime overhead. As well, it can lead to patterns such as the inefficient repeated fork/join above. In this example, it would be beneficial to step back a level higher and consider where parallelism could be applied. The fork-join style could be applied recursively by parallelizing the **GetNextImageBlock** iteration which could significantly reduce the number and degree of involved joins. Likewise, other patterns of parallelism could be considered. Pipelining, Data Flow, and Messaging, for example, might be patterns to consider applying in such circumstances.

Express parallelism at the highest level possible.

Give careful consideration to repeated use of fork-join constructs – especially if expressing low level parallelism. The synchronization involved in the repeated joins is a detriment to scalability.

FREE FORM TASK PARALLELISM

At times, the strict nature of “fork-join” parallelism can be too restrictive for a particular problem.

The Parallel Patterns Library and the Concurrency Runtime provide a number of task-oriented constructs that allow a more free-form expression of parallelism.

TASK GROUPS

The PPL’s *parallel_invoke* algorithm and the *structured_task_group* construct on which it is based allow for a strict “fork-join” style of parallelism. The PPL supports a similar construct, the *task_group*, which is a superset of the functionality provided by its more structured cousins.

The *task_group* class acts as a container of tasks which can execute in parallel and are either waited on or cancelled altogether. It has much more free-form capabilities allowing tasks to be scheduled to the group concurrently from multiple threads as well as free-form interleaving with other PPL constructs. This allows it to be used to express some algorithms and semantics which do not naturally fit into the “now-fork-now-join” nature of *parallel_invoke*.

While the *task_group* construct does allow for many more general forms of expression, it also incurs higher runtime overhead due to the semantics that it supports. If the nature of a problem allows it to be easily expressed with a *parallel_invoke*, such will generally perform better with fine grained workloads.

LIGHT WEIGHT TASKS

Many platforms have the notion of a thread-pool where work is submitted to a first-in / first-out queue and executed by an arbitrary thread within the pool. The Concurrency Runtime supports this notion through what it terms *light weight tasks*. These *light weight tasks* provide a way to have a particular method called with supplied context in FIFO fashion when a thread is available within the runtime. The API for scheduling such tasks is the *ScheduleTask* method and it looks as follows:

```
void ScheduleTask(TaskProc _Proc, void *_Data);
```

Figure 26

One of the advantages of the *ScheduleTask* method is that it allows more specific placement of work. Variants of the *ScheduleTask* method shown in Figure 26 appear on various constructs in the Concurrency Runtime. The *CurrentScheduler* class, the *Scheduler* class, and the *ScheduleGroup* class all provide this method. Separating categories of work out in *ScheduleGroup* objects allows more control over scheduling for advanced usage.

PATTERNS

SINGLE WAITER

Often times, there are algorithms where a set of forked parallel work must be waited in aggregate but recursively added work deep within the algorithm does not need to be both forked and joined. Consider the quicksort example of Figure 22. Each level of the quicksort forks off two tasks: one which sorts the left side of the array and one which sorts the right side of the array. It then waits on the completion of both of these tasks. Strictly speaking, the wait is not necessary in every recursive call to the sort. It is true that all of the sub-sorting must be completed prior to declaring the sort itself finished; however – each level does not need to perform this type of wait. Consider the following:

```
template<typename T>
void quicksort_helper(T* data, size_t l, size_t r, task_group *tg)
{
    if (r - l <= 1)
        return;

    size_t pivot = partition(data, l, r);

    tg->run([=](){ quicksort(data, l, pivot, tg); });
    tg->run([=](){ quicksort(data, pivot + 1, r, tg); });
}

template<typename T>
void quicksort(T* data, size_t l, size_t r)
{
    task_group tg;
    quicksort_helper(data, l, r, &tg);
    tg.wait();
}
```

Figure 27

This version of quicksort is very similar to the original; however, instead of waiting at each level of the quicksort, the parallelism is simply scheduled and the helper routine allowed to return. The group of work is waited upon in aggregate at the bottom of **quicksort**.

Depending on the type and granularity of work being performed, there are several advantages of this pattern. Perhaps the largest is that there is no join point during each call to **quicksort** (or **quicksort_helper** here). As there is no explicit join during each invocation of **quicksort_helper**, the calls for the left and right sub-arrays will never occur inline. That is to say, **quicksort_helper** will never call **quicksort_helper** recursively on the same stack. Hence, the thread's stack will not grow as the decomposition increases in depth as with a typical recursive algorithm.

One important thing to note about using more free form patterns like this is that they involve higher runtime overhead than a utilization of *parallel_invoke* or other forms of structured parallelism. The free-form nature of *task_group* is optimized around patterns such as those shown above where each parallel task adds new work back to the group to which it was scheduled. The nature of *task_group* allows it to be used in arbitrary ways which may not be able to take advantage of runtime optimizations and will incur higher overhead.

> *Consider utilizing a single high level task_group in divide-and-conquer style problems where each level does not require a join operation and the work performed is sufficient to amortize additional overhead.*

ANTI-PATTERNS

INCOHERENT SCHEDULING DEPENDENCIES

The Concurrency Runtime scheduler which underlies the PPL and Asynchronous Agents libraries provides a number of mechanisms to cede some level of scheduling control to client code. One such mechanism is the ability to create *schedule groups* and target individual work items to a particular schedule group instead of letting the scheduler decide where to place the work item. Manually targeting work items in this manner can allow for better utilization of cache in some scenarios. As with many lower level forms of control, this one comes with potential performance traps if not used carefully.

While the Concurrency Runtime scheduler makes every effort to guarantee ***forward progress*** amongst work items and threads it schedules, it makes no attempt to be ***fair*** in doing so with its default policy. While the lack of fairness in scheduling frequently leads to better cache utilization and higher overall performance, it can create a performance problem with certain types of dependency between scheduled tasks. Consider the below code sample:

```
void SpinWait(volatile long *data)
{
    int spinCount = 4000;
    for(;;)
    {
        if (*data != 0)
            break;

        if (--spinCount == 0)
        {
            Context::Yield();
            spinCount = 4000;
        }
    }
}

void SetData(volatile long *data)
{
    InterlockedExchange(data, 1);
}

ScheduleGroup *group1 = CurrentScheduler::CreateScheduleGroup();
ScheduleGroup *group2 = CurrentScheduler::CreateScheduleGroup();

long data = 0;

for (int i = 0; i < 32; i++)
    group1->ScheduleTask(reinterpret_cast<TaskProc>(SpinWait), &data);

group2->ScheduleTask(reinterpret_cast<TaskProc>(SetData), &data);

…
```

Figure 28

In this example, two schedule groups **group1** and **group2** are created. A number of tasks are scheduled on **group1** that perform a spin wait / *Context::Yield* until a message is available. As the tasks cooperatively yield to the runtime periodically through the *Context::Yield* API, one might expect this would be a reasonable spin-wait. The message that these tasks are waiting on, however, will be sent by a task scheduled to **group2**. The way in which this work has been scheduled created a cross-group scheduling dependency.

As mentioned earlier, the Concurrency Runtime scheduler is not fair. It prefers to keep a given core working on tasks or threads from the group it was last working on before getting work from elsewhere. Since the 32 tasks scheduled above spin, yield, and are always ready to run (they are never blocked), there will always be work in **group1** for the scheduler to find. The result is something that looks to the Concurrency Runtime scheduler much like a priority inversion might to an OS scheduler. Until the scheduler realizes that forward progress is not being made in this scenario, a lot of CPU time will get wasted performing the spin wait. The realization of lack of forward progress will eventually be made and the scheduler will pick up the task from **group2** that leads to progress. This realization, however, may come after some time has passed. If many tasks are scheduled with spin or yield dependencies in this manner, the performance of resulting code may significantly degrade.

Because the tasks of Figure 28 are deeply inter-related (they spin wait on each other, for instance), it would be far more beneficial from a scheduling perspective for them to be placed within a single schedule group. There is a far greater sense of fairness applied to yielding threads within a single schedule group than across schedule groups.

Code written to the Asynchronous Agents library should pay particular note to this problem. There are some scenarios which are easily and naturally expressed as a series of agents which yield to each other. Frequent yielding is generally the vehicle by which this issue manifests itself.

Minimize scheduling dependencies between work in multiple schedule groups. Such dependencies look to the Concurrency Runtime as a priority inversion would to an operating system scheduler!

UNCOOPERATIVE TASKS

The scheduler which underlies the Parallel Patterns Library and Asynchronous Agents Library is one which is cooperative by nature. There is a determination of how many threads will run concurrently and the scheduler strives to keep exactly that many threads running. As long as a given task is running on a thread, it will continue to run that task until the scheduler is explicitly aware that a blocking or yielding operation has taken place. By default, this translates into code utilizing a PPL cooperative synchronization primitive or explicitly calling an API such as *Context::Yield*.

What this also means is that if you have a task (or an agent) which performs some long running work, it may prevent other tasks in the system from starting. For example:

```
event finishedSignal;

void MyTask1(void *data)
{
    for(size_t count = 100000000; count > 0; --count)
    {
        // just waste some time.
    }
}

void MyTask2(event *finishedSignal)
{
    finishedSignal->set();
}

int main(int argc, char **argv)
{
    CurrentScheduler::SetDefaultSchedulerPolicy(
        SchedulerPolicy(2,
                        MinConcurrency, 1,
                        MaxConcurrency, 1)
        );

    CurrentScheduler::ScheduleTask(reinterpret_cast<TaskProc>(MyTask1),
                                   NULL);
    CurrentScheduler::ScheduleTask(reinterpret_cast<TaskProc>(MyTask2),
                                   &finishedSignal);

    finishedSignal.wait();
}
```

Figure 29

In Figure 29 above, two tasks are scheduled on a scheduler that is directed to only utilize one core on the system. **MyTask1** wastes time and spins. **MyTask2** signals an event which **main** waits for. If this example is executed, the event will not be signaled for a significant period of time. The single core will continue to execute **MyTask1** until it blocks or yields. As it does neither for a protracted period, **MyTask2** will not start until **MyTask1** finishes.

There are times where this behavior is perfectly fine; however, sometimes having such long running and uncooperative tasks can be a detriment to the desired progress within the system. There are several ways of dealing with this problem:

- Long running tasks can be further decomposed into sub-tasks which are not long running.

- Tasks, agents, etc… which are long running and wish other tasks to make progress against them can cooperatively block or yield. For example:

```
void MyTask1(void *data)
{
    for(size_t count = 100000000; count > 0; --count)
    {
        // doing something
        if (count % 100000 == 0)
            Context::Yield();
    }
}
```

Figure 30

Depending on what features of the runtime and its libraries are used, this strategy can increase progress in the system if desired. The *Context::Yield* call will, however, only yield to three different things -- another runnable thread on the scheduler to which the current thread belongs (regardless of what type of work that thread is running), a light weight task scheduled via the *ScheduleTask* API that has not yet started execution on a thread, or another operating system thread (via *SwitchToThread* like semantics). It will not yield to any work scheduled via the PPL's *task_group* or *structured_task_group* constructs that has not yet started execution on a thread.

- Programmatic oversubscription can be utilized for the period that the task is long running. For more information about this, please see Oversubscribing During Blocking Operations.

Be careful when writing single tasks with long bodies that do not cede control to the scheduler. Consider one of the alternative suggestions above.

PIPELINING, DATA FLOW, AND MESSAGING

Another classic pattern that turns up when considering the application of parallelism is that of pipelining / streaming or data-flow driven programming models. Consider having some data which flows through multiple independent stages of computation or transform. While each stage individually may not be able to exploit parallelism, the pipeline as a whole can exploit parallelism through concurrent execution of its stages so long as more than one bit of data flows through the pipeline.

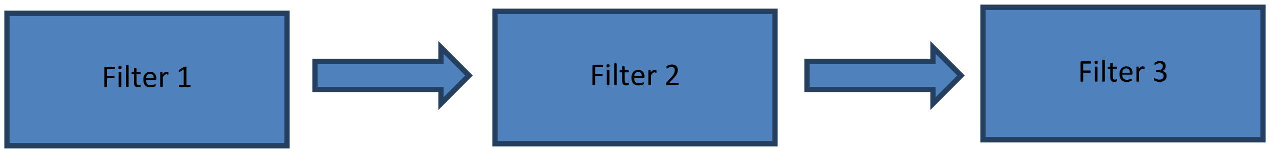
As an analogy, consider an assembly line manufacturing widgets. Each individual widget produced by the assembly line is built in a specified and serial sequence: step A then step B then step C, etc… Even though each individual widget on the assembly line is built in a serial fashion, there is parallelism overall because multiple widgets are on the assembly line at the same time. When the assembly line starts up, a single widget goes through step A. As that widget moves on to stage B, a second widget falls into place on the line going through step A. This happens concurrently to the stage B on the first widget.

The Asynchronous Agents Library provides a means of constructing such pipelines to take advantage of this style of parallelism. At the highest level, there are two fundamental constructs provided by the library: *messaging blocks* and *agents*. Both are constructs designed for a programming model based upon messaging. *Message blocks* are passive data-flow oriented messaging constructs laid out in an imperatively static fashion. *Agents* are generally more active participants in the messaging framework capable of more dynamically controlled messaging.

DATA FLOW

The components of the Asynchronous Agents Library are patterned to communicate via message passing from sources to targets. Sources produce a particular type of data and targets consume a particular type of data. Objects within the pipeline are either sources, targets, or both.

Consider a simple example: a set of filters performed to process image data:



The Asynchronous Agents Library can be utilized to construct this pipeline quite simply:

```
transformer<ImageData*, ImageData*> filter1([](ImageData* data){
    return PerformFilter1Action(data);
    });

transformer<ImageData*, ImageData*> filter2([](ImageData* data){
    return PerformFilter2Action(data);
    });

transformer<ImageData*, ImageData*> filter3([](ImageData* data){
    return PerformFilter3Action(data);
    });

filter1.link_target(&filter2);
filter2.link_target(&filter3);
```

Figure 31

As data gets sent to **filter1**, the runtime will invoke the functor provided to the transformer object which in turn calls **PerformFilter1Action**. The value returned from that method will be propagated by the runtime into **filter2** and so on.

Data can be sent into the pipeline via either the *send* or *asend* methods as shown below:

```
ImageData data(…);
send(&filter1, &data);
```

Figure 32

Assuming that multiple pieces of data are being sent into the pipeline, **filter1**, **filter2**, and **filter3** may have their transform functors running concurrently.

Short of repeatedly calling send on the thread which set up the pipeline, how might data flow into the network? As agents allow for more free form and dynamic expression of messaging outside the confines of what a static message block can do, they are the natural fit for this type of problem. Consider an agent as follows:

```
class ImageReaderAgent : public agent
{
public:

    ImageReaderAgent(ITarget<ImageData *> *imgTarget) :
        target(imgTarget)
    {
    }

protected:

    virtual void run()
    {
        while(…)
        {
            ImageData* data = ReadImageData();
            send(target, data);
        }

        done();

    }

private:

    ITarget<ImageData *> *target;
};
```

Figure 33

Once this agent is started, a thread will be dispatched into the agent's *run* method. This will repeatedly push image data into the network until some condition (not shown) is met. With an agent performing this work, the main portion of processing (see Figure 32) could be modified to do something akin to:

```
ImageReaderAgent readerAgent(&filter1);
readerAgent.start();
agent::wait(&readerAgent);
```

Figure 34

AD-HOC MESSAGING

One of the advantages of *agents* over static *message blocks* is that they allow a more free form expression of messaging than the static data-flow pipeline. Because agents are typically active control flow constructs, they can execute code and receive messages from multiple sources in a dynamic fashion as determined by code-flow and internal state rather than a static layout inherent in pure data-flow messaging with message blocks. As a very simple agent, the example of Figure 33 read data from some source and pushed it into the static data-flow network to perform a predetermined set of imaging transformations.

The active nature of this *agent*, for instance, makes it far easier than static message blocks to implement something like a protocol based on messaging. An *agent* body may decide to receive messages from one source, then another based upon the first message received. For instance:

```
virtual void run()
{
    while(…)
    {
        ActionType action = receive(actionPort);
        switch(ActionType)
        {
            case ActionA:
                ActionAProtocolHandler(); // drives its own receives
                break;

            case ActionB:
                ActionBProtocolHandler(); // drives its own receives
                break;

            default:
                break;
        }
    }

    done();
}
```

Figure 35

Similarly, the combination of messaging blocks and the control flow oriented model shown above makes a powerful construct:

```
virtual void run()
{
    join<DataType> dataJoin(2);
    source1->link_target(&dataJoin);
    source2->link_target(&dataJoin);

    while(…)
    {
        //
        // Wait on a control change message (e.g.: stop, cancel) as
        // well as two sources to “mix”:
        //
        auto msgChoice = make_choice(controlPort, &dataJoin);

        int result = receive(&msgChoice);
        switch(result)
        {
            case 0:
            {
                ControlMessage cm = msgChoice.value<ControlMessage>();
                ProcessControl(cm);
                break;
            }
            case 1:
            {
                 join<DataType>::_OutputType mixData =
                     msgChoice.value<join<DataType>::_OutputType>();
                 ProcessData(mixData);
                 break;
            }
            default:
                break;
        }
    }

    source1->unlink_target(&dataJoin);
    source2->unlink_target(&dataJoin);

    done();
}
```

Figure 36

PATTERNS

STATE ISOLATION

One of the many benefits of the agent (or actor) based programming model is that the channels of communication between concurrent components are governed by well-defined messages instead of arbitrary and ad-hoc shared memory synchronization. Ideally, each object in this model – be it an active agent or a passive message block – operates on its incoming message(s) with only local state information and sends message(s) onward. Isolating state in this manner has several key advantages:

- It greatly reduces shared memory contention. There are no longer a series of objects contending on locks and spend significant periods blocking. Likewise, the objects involved are far less likely to be contending on shared lines of cache. Both of these factors can lead to greater scalability.

- It reduces the risk of deadlock. Since the channels of communication are well-defined and isolated and the framework provides higher level semantics such as joins, there is often no explicit locking in code written to this model. Without the explicit locking, there is much less chance for traditional deadlock involving cyclic acquisition of mutual exclusion objects. As well, if the communication between components falls into a more data-flow oriented model, the chance of deadlock is greatly reduced.

- It greatly reduces the risk of race conditions. Given that communication is explicit and through well-defined channels at defined points controlled by the messaging infrastructure, there is little chance of classic race conditions – colliding writes, observing intermediate state with violated invariants, etc…

Although agents and messaging presents an infrastructure designed for and well suited to complete state isolation, the underlying implementation language is still C++. There is nothing that prevents some construct from stepping outside the bounds of suggested practice and explicitly utilizing shared state. An agent may, for instance, touch some object or take some lock in its *run* method. A messaging block such as a *transform* may do the same in its implementation functor. Doing this side-steps many of the benefits achieved from going to the agents and messaging model in the first place.

While the agents model may, in many circumstances, provide better scalability than a task or data parallel solution based on the actual utilization of shared memory, it is not a panacea. There is always runtime overhead involved in the messaging infrastructure. Similarly, the ideal practice of state isolation at times implies more frequent copies of data that works its way through the pipeline. If he messages being passed and the processing being performed at each stage are fine grained, the overhead of the paradigm may outweigh the benefit. Keep in mind that it is entirely plausible to garner the benefits of both models – utilizing agents, messaging, and isolated state at higher levels, and using task level or data parallelism within components to further decompose the problem for the runtime. The Concurrency Runtime is designed and suited to scheduling compositions of these multiple types of parallelism.

Isolate and encapsulate state within agents and messaging constructs. Let them directly access only their own data and state.

DEFINED INTERACTION POINTS

As discussed earlier, it is a good practice to keep all of an agent's state local to the agent itself and maintain isolation from other objects in the system. As the purpose of an agent, however, is typically message based communication with other objects in the system, there must be some degree of interaction between agents. How that interaction is initiated and proceeds is up to the discretion of the implementer of an agent. As with the notion of state isolation, it is a best practice to keep the interaction points with an *agent* well-defined and intentional.

As an example, consider how the Asynchronous Agents Library defines the interaction point of status which is generic to all agents:

```
class agent
{
public:

    ISource<agent_status> * status_port();

protected:

    overwrite_buffer<agent_status> _M_status;

};
```

Figure 37

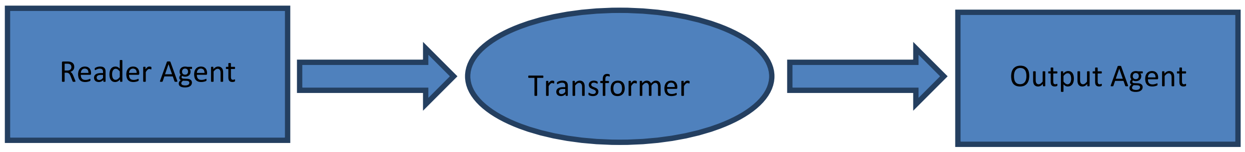
Here, the agent class has exposed the notion of a *port* – a well-defined interaction point on the agent that data flows in to or, in this case, out of. The external client of the agent can get no more access to the underlying mechanism (the overwrite buffer) than intended. The status port is an *ISource* – a source of agent_status messages only.

Defining interaction points with agents in this manner and the protocol around which they are utilized keeps the agent's state isolated while allowing only the necessary messaging to flow. Note that while it is best practice to define the protocol of communication with agents, there is no mechanism by which such can be automatically verified. It is up to the implementation of a particular agent.

Strongly define the interaction points with agents. Consider utilization of a model such as ports which codifies this concept.

MESSAGE THROTTLING

The Asynchronous Agents Library makes it easy to construct pipelines where the individual constructs in the pipeline can execute concurrently with other constructs in the pipeline. Consider a simple example as illustrated below:



Here, an agent reading data from some source (e.g.: file / network) is sending it to another construct to transform it, after which it is sent to some second agent to output it (e.g.: to a file / device / etc…). As a simple example, consider this some filtering of image data. Naively, the reader agent might have a run loop that looks something similar to what follows:

```
void MyReaderAgent::run()
{
    while(!finished())
    {
        Packet *p = ReadPacket();
        send(readerTarget, p);
    }

    done();
}
```

Figure 38

The reader agent is pulling packets as fast as it can from its source (disk, for example) and sending them on to be transformed. Depending on how much data will flow through this pipeline and how long the processing takes in comparison to the reading and packetizing, it is quite possible that the pipeline will build up a significant number of messages waiting to be processed. This may cause a significant waste of memory and other resources as the consumer (the transformer and output agent in this case) falls further and further behind the producer.

Instead of producing arbitrary amounts of data as fast as possible, the producer should be throttled so as to not overrun the consumer with data. There are several ways to accomplish this; however, the Visual Studio 2010 libraries do not provide single constructs for this.

THE SLIDING WINDOW PATTERN

Often times, the producer of data – **MyReaderAgent** in the example of Figure 38 – is sending data downstream without explicit knowledge of how that data will be used or how fast it will be consumed. Without the application of a throttling mechanism, problems will begin to present themselves as messages build up. One technique which can be applied to solve this is the sliding window pattern. Consider a modification of **MyReaderAgent**:

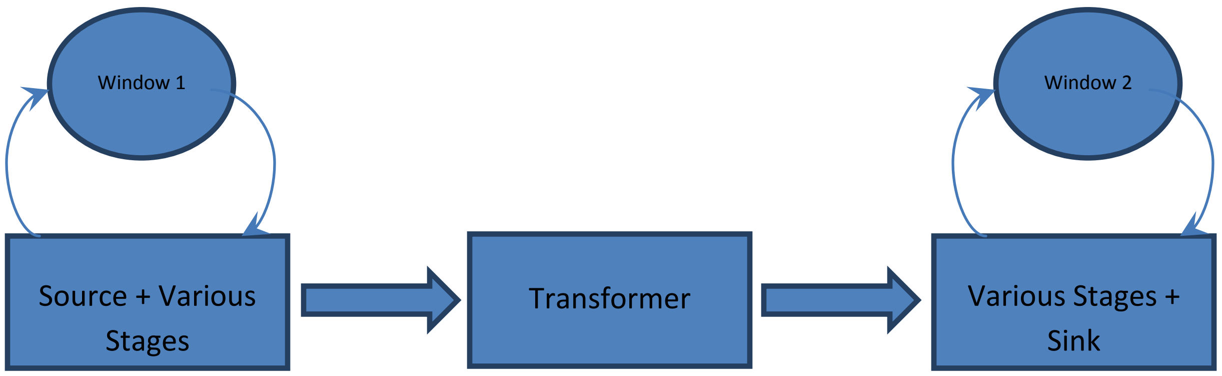
```
void MyReaderAgent::run()
{
    while(!finished())
    {
        throttleSemaphore.wait();
        Packet *p = ReadPacket();
        Send(readerTarget, p);
    }

    done();
}
```

Figure 39

Instead of arbitrarily sending data downstream as fast as the agent can produce it, a semaphore (**throttleSemaphore**) is waited upon before creating a message and sending it downstream. If this semaphore is initialized to have a maximum value of N and the final component in the stream releases the semaphore after receipt of the message, a stream of at most N messages will be present in the pipeline at any given time. This prevents the overrun.

What makes this pattern effective is that the entire pipeline must use it. If there are components in the pipeline performing non 1:1 transformations, they can apply a sliding window to their own portion of the pipeline. Everything in the pipeline must participate in some capacity. If there is a single point in the pipeline which does not and contains an unbounded queue as many messaging constructs do, an overrun of messages can build at that point if the consumption rate is below the production rate. Consider:



The source and sink segments in the above example both utilize the sliding window pattern within the confines of their portion of the pipeline. There is, however, a transformer connected between them which does not. As the Asynchronous Agents Library's messaging blocks all contain ***unbounded*** buffers to hold incoming messages which have yet to be handled by their respective functionality, a backlog of messages can build up here at either the transformer itself or as the transformer sends the messages on to the sink segment. All of the effort to utilize the sliding window pattern as a throttling mechanism within two segments of the pipeline is for naught. In aggregate, the pipeline itself is not throttled.

Another interesting application of this pattern involves allocation of messages or message data as it flows through the pipeline. Consider the producer of data in the example of Figure 38 (**MyReaderAgent**). In this case, and many like it, a packet of complex data is allocated and moves through the pipeline rather than a simple value type. When this happens, the various stages of the pipeline must agree on the allocation and deallocation pattern for that packet. One technique that can be used to combine an agreement on memory allocation with an agreement on throttling is to extend the notion of the sliding window pattern to that of an allocator pattern. Instead of simply allocating the packet from the heap, filling it, and sending it on, the allocation can come from an explicit object with responsibility for encapsulating the allocation and deallocation pattern – an *allocator* if you will. As well as encapsulating the notion of memory allocation, the allocator itself can be responsible for the encapsulation of the sliding window pattern for the segment of the pipeline which utilizes it. Consider:

```
Packet *Allocator::GetPacket()
{
    allocSemaphore.wait();
    return new Packet; // or from a pool
}

void Allocator::FreePacket(Packet *p)
{
    delete p; // or pool it
    allocSemaphore.signal();
}

void MyReaderAgent::run()
{
    while(!finished())
    {
        Packet *p = allocator.GetPacket();
        ReadPacket(p);
        send(readerTarget, p);
    }

    done();
}
```

Figure 40

Here, when the reader agent requests allocation of a new packet, the allocator itself blocks on the semaphore intended to implement our sliding window. If there are too many packets in flight at the time the reader tries to acquire a new one, the allocator will block the reader agent's thread and the system will automatically throttle itself.

Note that in utilizing the sliding window pattern or allocator pattern, it would be ideal if the semaphore used were a cooperative synchronization primitive that the Concurrency Runtime was aware of. The Visual Studio 2010 Parallel Patterns Library does not provide such a synchronization primitive though the sample pack does.

Limit the number of active messages flowing through a pipeline at any given time. It needs to be high enough to ensure an appropriate level of concurrency without being so high that it floods the consumer(s) and is a resource drain.

ANTI-PATTERNS

VERY FINE GRAINED MESSAGING

While the utilization of messaging constructs has a series of definite advantages, it also comes with overhead. Each of the messaging constructs in the Asynchronous Agents Library communicates via a carefully orchestrated protocol between sources (implementing the *ISource* interface) and targets (implementing the *ITarget* interface). As well, the propagation of messages between a given source and target is typically asynchronous. The fact that a protocol is involved here – and one that is partially asynchronous – would imply that runtime overhead for messaging is higher than that of the task parallel constructs within the Parallel Patterns Library. In fact, it is.

If the action that a particular messaging construct performs is very small, the overhead from the communication protocol to push those messages through the pipeline may dwarf the action that the block performs. Thus, the work that messaging constructs perform should be medium to coarse grained and not as fine grained as might be considered with task parallel constructs. It is true that some messaging constructs which perform little work in and of themselves may be necessary in the pipeline (joins, for instance). The actual work being performed within the pipeline (e.g.: the image filtering performed in the example of Figure 31) should be coarse enough to amortize the communication cost between necessary components such as buffers, joins, choices, etc…

Don't send messages resulting in very fine grained work. The work resulting from message flow needs to be high enough to amortize the overhead of the messaging constructs.

LARGE MESSAGE DATA

The preferred pattern of isolating state within a network of communicating agents of message blocks makes it very natural to pass message data by value. This alleviates many concerns about accidental sharing or mutating of data as well as memory management concerns. As messages are received or passed between agents, however, a copy of the message payload occurs. For simple value types, the implications of the copy are not typically of great concern. For complex or large types, this copy can carry a significant impact to the performance of the network.

Consider the example of Figure 33 where chunks of imaging data were being passed amongst the constructs in the pipeline. If the image data were frequently copied, this would carry significant performance implications on the overall application.

Do not pass large message payloads by value through a messaging network. Consider alternative means.

CARELESS USE OF PAYLOAD POINTERS

While the performance implications of passing large message payloads by value through a network may lead to passing the data by reference (or pointer), there are several issues which need to be carefully considered when doing this.

Once a payload is passed by reference or pointer through the pipeline, the lifetime of the object becomes a consideration. Typically, the sender of a message does not wait until the message has reached its final destination and been discarded before moving on to other work. This implies that such messages will be allocated from a particular allocator and have some form of associated ownership semantics. At the payload's end-of-life, the object must be returned to the same allocator from which it was allocated.

If careful discipline is followed or a given section of code has complete control over the entire messaging network, an allocator pattern can be utilized to provide careful object lifetime semantics. When the payload has reached its end-of-life, it is simply returned to the allocator from whence it came. This pattern, however, assumes that there is sole ownership of the payload data at any given point in time. Depending on the network through which the data is flowing, this may not always be the case.

Consider a messaging block which sends the message it received out to every target it is connected to. An *overwrite_buffer* is an example of one such block. If there are multiple targets connected to such a block and a pointer is passed as payload data, the payload's lifetime loses its sole ownership semantics. Multiple branches of the messaging network will have shared ownership of the payload. Such a payload cannot be released to its allocator until every shared owner mutually agrees that the payload has reached end-of-life.

There are several ways to address this problem. First, the problem can be avoided by careful control of the messaging network or a discipline of requiring single-ownership semantics through the network. If this is untenable for the problem space, solutions such as *shared_ptr* can be utilized to provide shared ownership semantics. Instead of passing **T*** throughout the network, **shared_ptr<T>** can be utilized instead.

Do not pass pointers as message payloads or embed pointers in message payloads without very careful consideration of ownership semantics in the network.

PASSING AND SHARING DATA

While there are many benefits to isolating the data utilized by multiple concurrent tasks, there is still often a need to share data between the tasks in some capacity – even if it is shared in a read-only form. When such sharing does take place, thought must be given to both the manner of passing the data to the task and the manner of sharing within the task. Passing and utilizing shared data in concurrent or asynchronous tasks haphazardly can lead to some common problems which can affect both the performance and correctness of the task bodies.

PATTERNS

COMBINABLE

There are many situations in which the multiple threads that work together to execute an algorithm in parallel wish to update some form of shared state. Consider first a simple serial example – a simple reduction computing the sum of *f(a[i])* for some array *a*:

```
int reduction;

for (size_t i = 0; i < arraySize; i++)
{
    reduction += f(a[i]);
}
```

Figure 41

Exempting the update of the shared state variable **reduction**, the iterations of the loop are completely independent of each other – there are no loop carry dependencies. It would not seem difficult to parallelize this loop using the PPL's primitives. The difficulty here is to handle the shared state update of the **reduction** variable. Naively, one could use a lock:

```
critical_section cs;
int reduction;

parallel_for((size_t)0, arraySize, [&cs, &reduction, &a](size_t i) {
    critical_section::scoped_lock holder(&cs);
    reduction += f(a[i]);
    });
```

Figure 42

The problem with the approach illustrated in Figure 42 is that the multiple threads executing the *parallel for* will spend a significant amount of time contending on the critical section **cs**. A significant portion of each thread's lifetime will be spinning or blocking waiting for a turn to update the shared variable. This will not scale. In fact, in this loop where all the work is done under the auspices of a critical section, the entire execution will serialize on the lock!

Some may notice that the shared variable **reduction** is of a type that hardware can increment atomically. The loop shown above could be rewritten as:

```
int reduction;

parallel_for((size_t)0, arraySize, [&reduction, &a](size_t i) {
    InterlockedExchangeAdd(
        reinterpret_cast<volatile LONG *>(&reduction),
        f(a[i])
        );
    });
```

Figure 43

While the threads involved in the loop no longer contend on a lock spending large portions of their life spinning or blocking, this still will not scale. The example of Figure 43 has merely traded one scalability bottleneck for another. Instead of contention on a lock that we see explicitly within the code, we have the overhead of significant communication from cache coherency mechanisms. The cache line containing the **reduction** variable will ping-pong from core to core being continually invalidated from the repeated atomic writes.

The way to achieve scalability is to remove the dependence on shared-write data between the threads involved in the loop. The PPL's combinable construct provides a convenient way to do just this. Consider a rewritten version of Figure 43:

```
combinable<int> reduction;

parallel_for((size_t)0, arraySize, [&reduction, &a](size_t i) {
    reduction.local() += f(a[i]);
    });

int finalReduction = reduction.combine(std::plus<int>());
```

Figure 44

Instead of updating a single shared variable, the example of Figure 44 updates **reduction.local()**. The PPL's combinable construct internally keeps a private copy of its data type for every thread which has utilized the given combinable object. Each thread involved in the loop is operating with its own private local data structures until the loop joins. After the join point, the private versions are combined together through a call to the *combine* method of the combinable construct. As there is no longer write-shared data between threads involved in the parallel for, this version of the loop will scale to a much higher degree than the prior examples.

Use the combinable construct to provide isolated thread-local copies of shared variables. This prevents the performance impact of having shared mutable data.

ANTI-PATTERNS

CARELESS REFERENCE CAPTURES

Visual Studio 2010 introduces the concept of lambdas into the C++ language by way of the C++0x lambda specification. This feature alone makes programming models such as the PPL much more expressive as code can be conveniently co-located rather than being pulled out into an explicit functor. Consider a *parallel_for* loop which multiplies a vector by a scalar:

```
template<typename T>
void MultiplyByScalar(std::vector<T>& v, const T& scalar)
{
    class MultFunctor
    {
    public:

        MultFunctor(std::vector<T>& srcVector, const T& srcScalar)
        {
            v = srcVector;
            scalar = srcScalar;
        }

        void operator()(size_t i)
        {
            v[i] *= scalar;
        }

    private:
        std::vector<T>& v;
        T scalar;
    } f(v, scalar);

    parallel_for(0, v.size(), f);
}
```

Figure 45

Now consider the same implementation with lambdas introduced:

```
template<typename T>
void MultiplyByScalar(std::vector<T>& v, const T& scalar)
{
    parallel_for(0, v.size(), [&v, =scalar](size_t i){
        v[i] *= scalar;
        });
}
```

Figure 46

By not requiring the explicit creation of a functor object and manual passing of variables used in the functor, the resulting code is greatly simplified and its intent is that much clearer. This ease of expressiveness does, however, come with risks. In Figure 46 above, the capture of **v** and **scalar** and the type of those captures is explicit (**v** by reference and **scalar** by value). It would be perfectly legal, however, to write the following:

```
template<typename T>
void MultiplyByScalar(std::vector<T>& v, const T& scalar)
{
    parallel_for(0, v.size(), [&](size_t i){
        v[i] *= scalar;
        });
}
```

Figure 47

Here, the capture of v and scalar are implicit by reference since the default capture semantic for the lambda is by reference. Since the actual capture of variables is implicit, care must be taken to ensure that the capture semantic is correct. Consider the following:

```
void DoAsyncAction(task_group tg)
{
    Object o = SomeInitialization();
    tg.run([&](){
        o.Action();
        });
}
```

Figure 48

Here, the capture of **o** is implicit by reference due to the default capture semantic of the lambda. By the time the lambda executes asynchronously on the task_group, however, it is likely that **DoAsyncAction** has unwound and destructed **o**. Unless there is a guarantee that **o** is still valid by the time the lambda executes (e.g.: by executing a *wait* call on **tg** as shown below in Figure 49), the capture of **o** should not be by reference.

```
void DoAsyncAction(task_group tg)
{
    Object o = SomeInitialization();
    tg.run([&](){
        o.Action();
        });
    …
    tg.wait();
}
```

Figure 49

Given the precarious nature of implicit reference capture in conjunction with asynchronous execution, consideration should be given to having the captured variables and their capture semantics explicit within lambdas used in asynchronous invocations.

Carefully consider every by reference capture within lambdas used as task bodies – especially in non-fork-join constructs! Make sure that the variable is guaranteed to be alive by the time the task body executes!

FALSE SHARING

One of the key tenets of writing parallelized code intended to scale well is to avoid shared data wherever possible. This is especially true for shared mutable data. While it may not be trivial, identifying places in parallelized code where such sharing happens explicitly is usually clear from the code. Consider a rather contrived example:

```
volatile LONG count = 0;

parallel_invoke(
    [&count](){
        for(int i = 0; i < 1000000; i++)
            InterlockedIncrement(&count);
    },
    [&count](){
        for(int j = 0; j < 1000000; j++)
            InterlockedIncrement(&count);
    });
```

Figure 50

It is very clear from this example that the variable **count** is shared between both forked tasks and that both are frequently modifying the same mutable data. The communication between the two cores or processors executing the two tasks will be significant. In fact, the parallel version above may run slower than two adjacent serial loops!

Imagine that we try to eliminate the sharing with a naïve modification of Figure 50:

```
LONG count1 = 0;
LONG count2 = 0;

parallel_invoke(
    [&count1](){
        for(int i = 0; i < 1000000; i++)
            count1++;
    },
    [&count2](){
        for(int j = 0; j < 1000000; j++)
            count2++;
    });

LONG count = count1 + count2;
```

Figure 51

It would seem that there is no longer shared modification of mutable state. From the program's perspective, this is true. However, from the perspective of the hardware, things are somewhat more complicated. In the example of Figure 50, imagine that two processors, A and B execute the two tasks. When processor A writes to **count**, the cache coherency protocol between the two processors kicks in and B's cached copy of **count** is invalidated. The next time B performs the atomic increment, it must fetch a new copy of **count** and modify it. B's modification of **count** will in turn invalidate A's cached copy.

As these updates and invalidations occur, the cache coherency protocol is not moving the single machine word containing **count** back and forth between the two processor's caches. For performance reasons, all of the updates are done at much coarser granularity. It is a group of bytes that is being invalidated and fetched into the caches. This group is known as a *cache line* and is typically around 64 or 128 bytes.

In the example of Figure 51, the declarations of **count1** and **count2** are adjacent on the forking thread's stack. Because they are only 8 bytes in total, it is exceedingly likely that they are on the same *cache line*. If this is indeed the case, as processor A writes to **count1**, it will invalidate the cached copy of **count2** within processor B's cache. Likewise, as processor B writes to **count2**, it will invalidate A's cached copy of **count1**. Even though there is no shared modification of state from the program's perspective, there is shared modification of a resource – a *cache line* – from the hardware's perspective. This is known as *false sharing* and it is something to be avoided! The code of Figure 51 will likely run no faster than that of Figure 50.

While the identification of the problem of *false sharing* can be exceedingly difficult, its remedy is often not so. There are several ways that the code of Figure 51 could be modified to remove the *false sharing*. For example, the *combinable* construct could be utilized for thread local copies of **count** which are then combined after the join point. The PPL's *combinable* will create the thread local copies in such a way as to make *false sharing* unlikely.

Similarly, the declarations of **count1** and **count2** could be padded to guarantee that they do not sit on the same *cache line*. Consider:

```
LONG count1;
LONG pad[CACHE_LINE_SIZE / sizeof(LONG)];
LONG count2;
```

Figure 52

This particular technique is often used on data structures where certain fields are meant to be shared read only while others are shared mutable. The fields are either arranged in such a way that infrequent fields are placed between frequent shared read fields and the mutable fields or suitable padding is inserted.

In any case, false sharing is something of which you always need to be cognizant. Make sure that state intended to be isolated to a particular thread/task or intended to be shared read-only is not on a cache line shared with other mutable state.

> *Carefully consider the placement of mutable data passed to parallel constructs. Avoid placing isolated mutable data for multiple tasks on the same cache lines!*

SYNCHRONIZATION

In many patterns of parallelism, a series of tasks – be they tasks created as part of a fork join construct, agents, etc… – work in a coordinated fashion on some problem. In any such pattern, there is an inevitability of some form of communication or synchronization between the tasks. Even in something as simple as a fork-join construct, there is a synchronizing point at the end of the construct. The join is, by its very nature, synchronization between all of threads executing tasks for the construct. Likewise, in messaging constructs, receipt of a message is a form of synchronization. The agent or construct performing the receive blocks until it the requisite message arrives. As well, there are other patterns which may require some form of synchronization – mutable access to some shared resource, for instance.

When considering the appropriate mechanism and method of synchronization to utilize in any of these scenarios, there are some issues which need to be considered in terms of how that synchronization interacts with the Concurrency Runtime and PPL layers.

PATTERNS

COOPERATIVE SYNCHRONIZATION PRIMITIVES

There are many circumstances under which parallel or concurrent code must arbitrate access to some resource, wait for some event to be signaled, or otherwise block for a period of time. While the Windows API provides a number of synchronization primitives for this purpose, utilizing these primitives in code parallelized using the

Concurrency Runtime has a problem. These primitives are below the level of the scheduler and unless user mode scheduling is enabled, the runtime is completely unaware of their use. Depending on the code in question, this may lead to underutilization of CPU resources or, in extreme cases, deadlock.

The PPL provides a series of cooperative synchronization primitives similar in nature to those found in the Windows API:

- *critical_section* –a non-reentrant fair mutual exclusion object similar to the Windows API's CRITICAL_SECTION

- *reader_writer_lock* – a non-reentrant writer biased reader writer lock

- *event* – a manual reset event similar in nature to the Windows API's manual reset event

Whenever possible, code targeting the PPL or Concurrency Runtime should utilize these locking primitives instead of the Windows API counterparts. Using these primitives makes the scheduler explicitly aware that a blocking operation is taking place and allows it to schedule additional work in lieu of the blocked thread.

If user mode scheduling is enabled, the Concurrency Runtime's scheduler is explicitly aware of any blocking performed through the use of Windows API synchronization primitives. While at face value, this seems to mitigate the need to use cooperative synchronization primitives, there is a significant performance benefit to using the cooperative synchronization primitives even in conjunction with user mode scheduling. Any blocking call to a Windows API synchronization primitive must make a transition in and out of kernel mode. Cooperative synchronization primitives on a user mode scheduler operate entirely in user mode with no kernel mode context switch. This allows the granularity of work being performed to be significantly finer grained.

Use the synchronization primitives within the Parallel Patterns Library and Concurrency Runtime whenever possible.

OVERSUBSCRIBING DURING BLOCKING OPERATIONS

When code that is interacting with the PPL and Concurrency Runtime can utilize the runtime's cooperative synchronization primitives, it should. There may, however, still be interactions with third-party components or components which are unaware of the runtime and utilize Windows API synchronization primitives. Such use may be explicit – as in calling *WaitForSingleObject* – or may be implicit – as synchronously reading from a file which may be on a high-latency network share.

Whenever parallel code makes API calls that block (or may potentially block) for long periods of time, that code should make the runtime explicitly aware of the blocking operation through oversubscription notification. For example:

```
//
// About to read from a file (which may be on the network)
//
Context::Oversubscribe(true);
if (ReadFile(…))
{
}
Context::Oversubscribe(false);
```

Figure 53

The calls to *Context::Oversubscribe* which enclose the *ReadFile* API call inform the scheduler that a long blocking operation may potentially occur in the region. Since the scheduler is now explicitly aware of potential blocking, it may choose to inject more concurrency to prevent underutilization of CPU resources during the blocking operation.

While in many cases, such as in the example above, forgetting to call the *Context::Oversubscribe* method may only result in reduced performance, there are more extreme cases where it may result in deadlock. Consider two tasks as presented below in Figure 54 and Figure 55:

```
void Task1()
{
    WaitForSingleObject(hEvent, INFINITE);
}
```

Figure 54

```
void Task2()
{
    Concurrency::wait(100);
    SetEvent(hEvent);
}
```

Figure 55

Consider the possibility that these tasks are queued for execution on a single core system. If **Task2** runs first, it will perform a cooperative blocking operation (waiting for 100mS) leading to execution of **Task1**. Once **Task1** executes, it blocks on an event. Unfortunately, the event will never get signaled because the scheduler is unaware that **Task1** has blocked and will never reschedule **Task2** on the single core that it manages.

This problem can be addressed in several manners:

- Avoid the use of Windows API synchronization primitives in code operating in conjunction with the PPL and Concurrency Runtime. This may not always be practical given the large numbers of components unaware of the PPL and Concurrency Runtime's synchronization primitives.

- Annotate blocking operations that are not cooperative with calls to *Context::Oversubscribe* as shown in Figure 53.

- Alter the scheduler policy to indicate a minimum concurrency level of the number of threads or tasks interdependent on non-cooperative synchronization primitives. This may not be practical in many circumstances as the requisite number may not be known or may result in unnecessary oversubscription when the dependency chains do not occur in practice.

When making blocking method calls based on non-runtime-aware synchronization primitives, annotate the blocking operations with calls to Context::Oversubscribe.

ANTI-PATTERNS

SPIN WAITING

A common pattern used in threaded code is that of a spin wait. There are times where it is known that another thread is in a small region of critical code and it is beneficial to spin wait for that thread rather than taking the overhead of blocking and context switching.

Consider a starting point for a very naïve implementation of a simple "lock-free yet not wait-free" stack implemented in an array:

```
void Stack::Push(void *data)
{
    long slot = InterlockedIncrement(&top);
    for(;;)
    {
        if (InterlockedCompareExchange(&store[slot],
                                       data,
                                       NULL) == NULL)
            break;
    }
}

void *Stack::Pop()
{
    void *data;
    long slot = InterlockedDecrement(&top);
    for(;;)
    {
        data = InterlockedExchange(&store[slot], NULL);
        if (data != NULL)
            break;
    }
}
```

Figure 56

The intent here is that the manipulation of **top** and the write into **store** should be atomic. Since the writes cannot be made atomic on today's hardware architectures, the snippet above attempts to utilize a spin loop to guard against the other side being in the middle of the opposite operation.

Consider what happens when this code is executed on a machine with a single processor core. Imagine two threads, one pushing and one popping. If the thread executing the push operation is preempted immediately after the write to **top** and the popper runs, the popper will reach the spin loop and continue to spin until the pusher writes to its slot in the array. As the machine only has a single processor core, this will not happen until the popper's thread quantum expires and the pusher gets a time slice.

The fact that this progresses at all is an artifact of the preemptive nature of the kernel scheduler. The Concurrency Runtime, however, is a cooperative scheduler. Threads managed by the concurrency runtime run until they make an explicit call to block or yield. If the pusher and popper discussed above were tasks instead of threads and executed on a single processor core machine under the Concurrency Runtime with user mode scheduling, the scenario described would play out very differently. Instead of the popper spinning until its thread quantum expired, the popper would potentially spin forever! The pusher would never get a chance to execute and livelock would ensue!

With our threaded example running under the kernel scheduler, the performance issue can be addressed via periodic calling of the *SwitchToThread* Windows API (or by falling back to blocking after a period of time). With the

Concurrency Runtime, a loop such as illustrated above should utilize a yielding method such as *Context::Yield*. This should only be done where spin loops truly cannot be avoided.

> *Avoid spin waiting whenever possible! If unavoidable, ensure that any such spin cooperatively yields to the Concurrency Runtime scheduler.*

GENERAL

Many of the patterns discussed earlier in this document can be considered to apply to the general development of parallel code. There are, however, some concerns, patterns, and anti-patterns that apply more specifically to the Concurrency Runtime, Parallel Patterns Library, and Asynchronous Agents Library and their present implementation details. What follows is a discussion of some of these more specific patterns and anti-patterns that do not easily fit into broad parallelization patterns.

PATTERNS

SUBALLOCATION FOR SHORT LIVED LOCAL MEMORY

As programs attempt to make use of increasing numbers of cores, it is increasingly likely that the decomposition of parallel tasks will become finer grained. With coarse grained tasks, the allocation of memory does may not become a significant concern. As tasks become increasingly fine grained, however, any memory allocation performed within them can become a bottleneck as heap contention increases.

In order to attempt to alleviate this problem, the Concurrency Runtime exposes a suballocator object which caches small allocations in a per-thread manner to largely eliminate heap contention. If the memory is allocated in a manner where it is allocated and freed on the same thread – particularly where these allocations and deallocations are frequent, this suballocator can be a significant performance win. For example, the Concurrency Runtime utilizes its own suballocator for the allocation of certain tasks as those tasks come and go frequently and are short lived.

The suballocator can be accessed as shown below:

```
Structure *s = reinterpret_cast<Structure *>
                    (Concurrency::Alloc(sizeof(Structure)));

Concurrency::Free(s);
```

Figure 57

The *Concurrency::Alloc* and *Concurrency::Free* methods work much like the standard library *malloc* and *free* methods. If they are used for C++ object allocations, consideration must be given to constructor and destructor calls. This can be done either via a placement new or, more expressively, through a base class providing an implementation of *operator new* and *operator delete*:

```
class SuballocatorObject
{
public:

    void *operator new(size_t size)
    {
        return Concurrency::Alloc(size);
    }

    void operator delete(void *obj) throw()
    {
        Concurrency::Free(obj);
    };
};
```

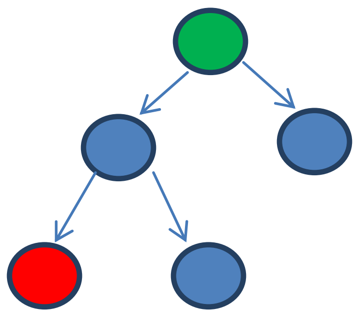
Figure 58

Any object which derives from the **SuballocatorObject** class illustrated in Figure 58 will automatically utilize the suballocator for its memory.

Utilize the Concurrency Runtime's suballocator when making small short-lived local allocations!

EXCEPTION SAFETY

The Concurrency Runtime utilizes exception propagation as a means of cancelling trees of work – particularly within nested patterns of fork/join parallelism. Consider a parallel search similar to that shown in Figure 12 of *Breaking Out of Parallel Loops*. Each level of the search recursively invokes a *parallel_for_each* to scan the next level of the search space. Again, consider a very simple search space with the leaf node in red being the one which matches the search predicate and the root node in green being the root of the search where cancellation takes place.



Once the *cancel* method is called on the *structured_task_group* representing the root of the search, any thread which is participating in the search by being contained in the sub-tree rooted at the search node in green above will be canceled by the runtime.

How the runtime affects cancellation of these threads has a key impact and implication for all code which interacts with the PPL and Concurrency Runtime. Whenever such a thread is canceled, it is flagged by the runtime as such. When that thread next executes code which is considered an *interruption point*, the runtime checks this flag and causes an exception to be thrown. While that exception is both thrown and caught within the Concurrency Runtime, it has the effect of unwinding the stacks of any parallel execution forked from the original search (in its sub-tree). As the Concurrency Runtime does not specify which points are interruption points and which are not, any interaction with the PPL, Asynchronous Agents, or Concurrency Runtime APIs must be considered for possible interruption. Likewise, calling into any code which might possibly interact with the PPL, Asynchronous Agents, or Concurrency Runtime APIs must also be considered for interruption transitively. Consider the following snippet of rather arbitrary code:

```
void Object::Action()
{
    int *tempBuffer = new int[size];
    parallel_for(0, size, [&](){
        //
        // Perform some action requiring the temporary buffer
        // Action is nothrow.
        //
        });
    delete tempBuffer;
}
```

Figure 59

Suppose that the manipulation performed within the *parallel_for* loop above is guaranteed not to throw exceptions – it only operates on base types and performs no allocations. From the outset, this code ***seems*** to be safe. Unfortunately, it is not. The call to *parallel_for* is an *interruption point*. If this routine is being called somewhere in a cancellable algorithm (e.g.: the parallel search of Figure 12), the call to *parallel_for* itself may throw an exception that would result in tempBuffer leaking.

As with C++ exception safety in general, the problem gets worse if state manipulations were performed to the object before the parallel loop that temporarily violate some object invariant with the expectation that the invariant would be restored after the loop.

Since any parallel algorithm can be canceled, unless you are explicitly aware of all the potential callers of a routine and can guarantee that none of them will ever call a *cancel* method, the routine must be exception safe around utilization of the runtime. At a minimum, they should provide a weak guarantee of exception safety. Depending on the semantics of the code, a stronger guarantee may be required.

Write any code which interacts with APIs provided by the Concurrency Runtime, Parallel Patterns Library, or Asynchronous Agents Library in an exception safe way!

LIMITING CONCURRENCY WHEN NECESSARY

Sometimes in the course of examining the scalability of a parallel algorithm, it is discovered that the implementation will only scale well to N cores after which performance falls off for a variety of reasons. This can present a problem for a library writer trying to exploit parallelism within the library. Consider, as an example, the version of quicksort presented in Figure 22. This particular implementation could be modified to limit the runtime's usage of cores as shown below:

```
Scheduler* qsScheduler = Scheduler::Create(
    SchedulerPolicy(2,
                    MinConcurrency, 1,
                    MaxConcurrency, 8)
    );

…

template<typename T>
void quicksort(T* data, size_t l, size_t r)
{
    qsScheduler->Attach();

    if (r - l <= 1)
        return;

    size_t pivot = partition(data, l, r);

    parallel_invoke([=](){ quicksort(data, l, pivot); },
                    [=](){ quicksort(data, pivot + 1, r); });

    CurrentScheduler::Detach();

}
```

Figure 60

The region between the calls to **qsScheduler->Attach()** and **CurrentScheduler::Detach()** will be controlled via **qsScheduler**. As this scheduler was created with a policy dictating that it uses at most 8 cores, the *parallel_invoke* calls within the sort implementation will be forked out to at most 8 cores. The actual number of cores here will be governed by a resource manager underlying the runtime which will potentially start load balancing between a scheduler in use before calling the library and the one utilized by the library. Once the **CurrentScheduler::Detach()** call happens, the runtime reverts back to utilizing the previously assigned scheduler.

It is important to note in such scenarios that it may be beneficial to cache the scheduler utilized by the library and attach and detach it rather than creating a new explicit scheduler every time the library call is made. Also, for the purposes of code brevity, the attachment and detachment are not governed by an RAII pattern. The runtime, unfortunately, does not provide any RAII idiom for the attachment and detachment of schedulers.

Utilize scheduler policy to limit concurrency when an algorithm is known to only scale to a certain point.

ANTI-PATTERNS

PARALLELISM WITHIN OBJECT DESTRUCTORS AND CLEANUP CODE

As discussed earlier in the *Exception Safety* section, any code which interacts with the PPL, Asynchronous Agents Library, or the Concurrency Runtime must deal with the possibility of interruption. Since, at present, no code can opt out of participating in the cancellation mechanism, this places implicit restrictions on where parallelism can be expressed with these libraries.

Consider the following:

```
class Object
{
    …

    ~Object()
    {
        parallel_invoke([&](){ containedObject1.Cleanup(); },
                        [&](){ containedObject2.Cleanup(); });

        containedObject3.Cleanup();
    }

    …
};
```

Figure 61

Here, Object's destructor performs some cleanup in parallel. Conceptually, it seems fine that this cleanup is done in parallel. In practice, however, this is problematic. Object's destructor might run in one of several contexts:

- It may run as the result of stack unwinding in normal control flow (exiting the containing scope, a delete call, etc…)

- It may run as the result of stack unwinding due to an exception flowing (e.g.: an exception causes unwind past Object's containing scope).

- It may run during either of the above two scenarios after the *cancel* method is called at some fork point in the work tree above the current execution location

During normal control flow, it is indeed true that Object's destructor will clean up the two objects in parallel. In the other two scenarios; however, the *parallel_invoke* call is subject to cancellation. The Concurrency Runtime has no idea that this invocation is happening for cleanup purposes in an object destructor. It assumes that more parallelism is being forked from a canceled point and that this forking is useless work. The *parallel_invoke* call is, in fact, an *interruption point*. This has some interesting implications.

First, if the destructor is running during normal control flow after a cancellation and the *parallel_invoke* call happens to be the first *interruption point* that the thread has hit, the call to *parallel_invoke* will throw! The Concurrency Runtime has no idea that the call is coming from a destructor! Throwing out of a destructor in this manner is strongly discouraged in C++. The object will be left in a partially destructed state with, at the very least, containedObject3 never being cleaned up! If the destructor happens to run during exception flow, the Concurrency Runtime is aware of this and will not collide exceptions. The *parallel_invoke* call, in this case, is effectively ignored. Here, containedObject3 is cleaned up; however, containedObject1 and containedObject2 may not be depending on when the cancellation was observed.

If any piece of code has mandatory execution requirements for cleanup or exception safety, parallelism expressed by PPL, Asynchronous Agents, or other Concurrency Runtime primitives should not be used.

***Never** utilize any cancellable parallelism construct within an object destructor or cleanup code!*

ACKNOWLEDGEMENTS

The author would like to thank the following people for their feedback on drafts of this paper: Atilla Gunal, Steve Gates, Niklas Gustafsson, Mohamed Ameen Ibrahim, Artur Laksberg, Thomas Petchel, and Krishnan Varadarajan.