# Intel® Threading Building Blocks

## Design Patterns

Document Number 323512-001US

World Wide Web: http://www.intel.com

## Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/#/en_US_01.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Atom Inside, Centrino Inside, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, Viiv Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

Copyright (C) 2010, Intel Corporation. All rights reserved.

## Revision History

| Version | Version Information | Date |
|---|---|---|
| 1.00 | Initial version. | 2010-Apr-4 |

## Contents

1 Introduction ........ 1
2 Agglomeration ........ 2
3 Elementwise ........ 5
4 Odd-Even Communication ........ 7
5 Wavefront ........ 8
6 Reduction ........ 12
7 Divide and Conquer ........ 16
8 GUI Thread ........ 20
9 Non-Preemptive Priorities ........ 24
10 Local Serializer ........ 27
11 Fenced Data Transfer ........ 31
12 Lazy Initialization ........ 34
13 Reference Counting ........ 37
14 Compare and Swap Loop ........ 39
General References ........ 41

# 1 Introduction

This document is a “cookbook” of some common parallel programming patterns and how to implement them in Intel® Threading Building Blocks (Intel® TBB). A cookbook will not make you a great chef, but provides a collection of recipes that others have found useful.

Like most cookbooks, this document assumes that you know how to use basic tools. The Intel® Threading Building Blocks (Intel® TBB) Tutorial is a good place to learn the basic tools. This document is a guide to which tools to use when.

A design pattern description is much more than a rote coding recipe.  The description of each pattern has the following format:

- **Problem** – describes the problem to be solved.
- **Context** – describes contexts in which the problem arises.
- **Forces** –  considerations that drive use of the pattern.
- **Solution** –  describes how to implement the pattern.
- **Example** – presents an example implementation.

Variations and examples are sometimes discussed.  The code examples are intended to emphasize key points and are not full-fledged code.  Examples may omit obvious const overloads of non-const methods.

Much of the nomenclature and examples are adapted from Web pages created by Eun-Gyu and Marc Snir, and the Berkeley parallel patterns wiki. See links in the General References section

For brevity, some of the code examples use C++0x lambda expressions. It is straightforward, albeit sometimes tedious, to translate such lambda expressions into equivalent C++98 code.  See the Section "Lambda Expressions" in the Intel® TBB tutorial on how to enable lambda expressions in the Intel® Compiler or how do the translation by hand.

# 2 Agglomeration

## Problem

Parallelism is so fine grained that overhead of parallel scheduling or communication swamps the useful work.

## Context

Many algorithms permit parallelism at a very fine grain, on the order of a few instructions per task. But synchronization between threads usually requires orders of magnitude more cycles. For example, elementwise addition of two arrays can be done fully in parallel, but if each scalar addition is scheduled as a separate task, most of the time will be spent doing synchronization instead of useful addition.

## Forces

- Individual computations can be done in parallel, but are small. For practical use of Intel® Threading Building Blocks (Intel® TBB), "small" here means less than 10,000 clock cycles.
- The parallelism is for sake of performance and not required for semantic reasons.

## Solution

Group the computations into blocks. Evaluate computations within a block serially.

The block size should be chosen to be large enough to amortize parallel overhead. Too large a block size may limit parallelism or load balancing because the number of blocks becomes too small to distribute work evenly across processors.

The choice of block topology is typically driven by two concerns:

- Minimizing synchronization between blocks.
- Minimizing cache traffic between blocks.

If the computations are completely independent, then the blocks will be independent too, and then only cache traffic issues must be considered.

If the loop is “small”, on the order of less than 10,000 clock cycles, then it may be impractical to parallelize at all, because the optimal agglomeration might be a single block,

## Examples

Intel® TBB loop templates such as `tbb::parallel_for` that take a *range* argument support automatic agglomeration.

When agglomerating, think about cache effects. Avoid having cache lines cross between groups if possible.

There may be boundary to interior ratio effects. For example, if the computations form a 2D grid, and communicate only with nearest neighbors, then the computation per block grows quadratically (with the block's area), but the cross-block communication grows with linearly (with the block's perimeter). Figure 1 shows four different ways to agglomerate an 8×8 grid. If doing such analysis, be careful to consider that information is transferred in cache line units. For a given area, the perimeter may be minimized when the block is square with respect to the underlying grid of cache lines, not square with respect to the logical grid.

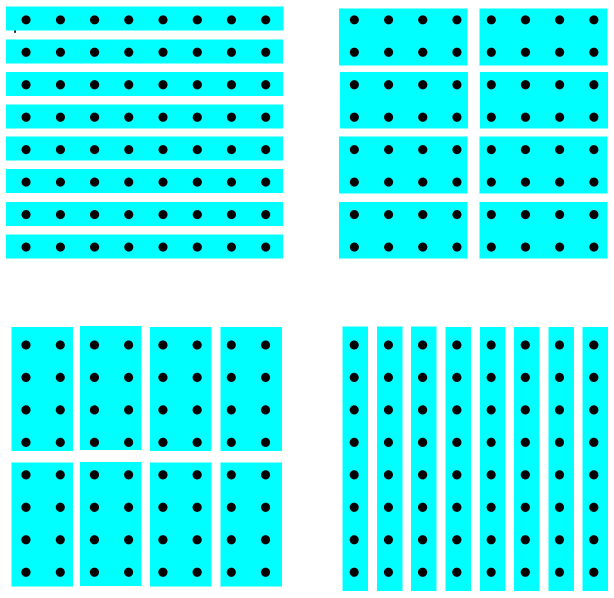

**Figure 1: Four different agglomerations of an 8×8 grid.**

Also consider vectorization. Blocks that contain long contiguous subsets of data may better enable vectorization.

For recursive computations, most of the work is towards the leaves, so the solution is to treat subtrees as a groups as shown in Figure 2.

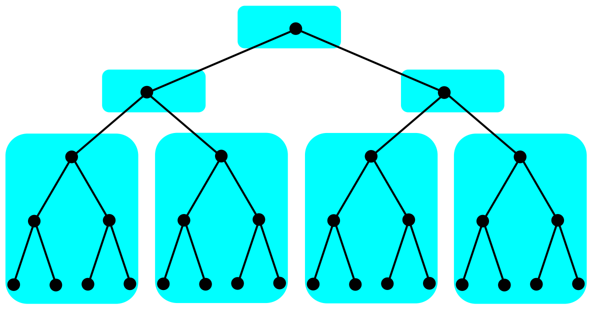

**Figure 2: Agglomeration of a recursive computation**

Often such an agglomeration is achieved by recursing serially once some threshold is reached. For example, a recursive sort might solve sub-problems in parallel only if they are above a certain threshold size.

## Reference

Ian Foster introduced the term "agglomeration" in his book *Designing and Building Parallel Programs* <http://www.mcs.anl.gov/~itf/dbpp>. There agglomeration is part of a four step “PCAM” design method:

1. **P**artitioning - break the program into the smallest tasks possible.
2. **C**ommunication – figure out what communication is required between tasks. When using Intel® TBB, communication is usually cache line transfers. Though they are automatic, understanding which ones happen between tasks helps guide the agglomeration step.
3. **A**gglomeration – combine tasks into larger tasks. His book has an extensive list of considerations that is worth reading.
4. **M**apping – map tasks onto processors. The Intel® TBB task scheduler does this step for you.

# 3 Elementwise

## Problem

Initiate similar independent computations across items in a data set, and wait until all complete.

## Context

Many serial algorithms sweep over a set of items and do an independent computation on each item. However, if some kind of summary information is collected, use the Reduction pattern instead.

## Forces

No information is carried or merged between the computations.

## Solution

If the number of items is known in advance, use `tbb::parallel_for`. If not, consider using `tbb::parallel_do`.

Use agglomeration if the individual computations are small relative to scheduler overheads.

If the pattern is followed by a reduction on the same data, consider doing the element-wise operation as part of the reduction, so that the combination of the two patterns is accomplished in a single sweep instead of two sweeps. Doing so may improve performance by reducing traffic through the memory hierarchy.

## Example

Convolution is often used in signal processing. The convolution of a filter *c* and signal *x* is computed as:

$$y_i = \sum_j c_j x_{i-j}$$

Serial code for this computation might look like:

```
// Assumes c[0..clen-1] and x[1-clen..xlen-1] are defined
for( int i=0; i<xlen+clen-1; ++i ) {
    float tmp = 0;
    for( int j=0; j<clen; ++j )
```

```
        tmp += c[j]*x[i-j];
    y[i] = tmp;
}
```

For simplicity, the fragment assumes that x is a pointer into an array padded with zeros such that *x*[*k*] for returns zero when *k*<0 or *k*≥*xlen*.

The inner loop does not fit the elementwise pattern, because each iteration depends on the previous iteration. However, the outer loop fits the elementwise pattern. It is straightforward to render it using `tbb::parallel_for` as shown:

```
tbb::parallel_for( 0, xlen+clen-1, [=]( int i ) {
    float tmp = 0;
    for( int j=0; j<clen; ++j )
        tmp += c[j]*x[i-j];
    y[i] = tmp;
});
```

`tbb::parallel_for` does automatic agglomeration by implicitly using `tbb::auto_partitioner` in its underlying implementation. If there is reason to agglomerate explicitly, use the overload of `tbb::parallel_for` that takes an explicit range argument. The following shows the example transformed to use the overload.

```
tbb::parallel_for(
    tbb::blocked_range<int>(0,xlen+clen-1,1000),
    [=]( tbb::blocked_range<int> r ) {
        int end = r.end();
        for( int i=r.begin(); i!=end; ++i ) {
            float tmp = 0;
            for( int j=0; j<clen; ++j )
                tmp += c[j]*x[i-j];
            y[i] = tmp;
        }
    }
);
```

# 4 Odd-Even Communication

## Problem

Operations on data cannot be done entirely independently, but data can be partitioned into two subsets such that all operations on a subset can run in parallel.

## Context

Solvers for partial differential equations can often be modified to follow this pattern. For example, for a 2D grid with only nearest-neighbor communication, it may be possible to treat the grid as a checkerboard, and alternate between updating red squares and black squares.

Another context is staggered grid ("leap frog") Finite Difference Time Domain (FDTD) solvers, which naturally fit the pattern. The code `examples/parallel_for/seismic/` uses such a solver.

## Forces

- Dependences between items form a bipartite graph.

## Solution

Alternate between updating one subset and then the other subset. Apply the elementwise pattern to each subset.

## Example

The example in `examples/parallel_for/seismic` demonstrates the principle. In it, two physical fields *velocity* and *stress* each depend upon each other, and so cannot all be updated simultaneously. However, the *velocity* calculations can be done independently as long as the *stress* values remain fixed, and vice-versa. So the code alternates updates of the *velocity* and *stress* fields. Each update is done using `tbb::parallel_for`.

## Reference

The document "Odd-Even Communication Group" <http://www.cs.uiuc.edu/homes/snir/PPP/patterns/oddeven.pdf> by Eun-Gyu Kim and Marc Snir describes the pattern.

# 5 Wavefront

## Problem

Perform computations on items in a data set, where the computation on an item uses results from computations on predecessor items. See reference for a discussion.

## Context

The dependences between computations form an acyclic graph.

## Forces

- Dependence constraints between items form an acyclic graph.
- The number of immediate predecessors in the graph is known in advance, or can be determined some time before the last predecessor completes.

## Solution

The solution is a parallel variant of topological sorting, using `tbb::parallel_do` to process items. Associate an atomic counter with each item. Initialize each counter to the number of predecessors. Invoke tbb::parallel_do to process the items that have no predessors (have counts of zero). After an item is processed, decrement the counters of its successors. If a successor's counter reaches zero, add that successor to the `tbb::parallel_do` via a "feeder".

If the number of predecessors for an item cannot be determined in advance, treat the information "know number of predecessors" as an additional predecessor. When the number of predecessors becomes known, treat this conceptual predecessor as completed.

If the overhead of counting individual items is excessive, aggregate items into blocks, and do the wavefront over the blocks.

## Example

Below is a serial kernel for the longest common subsequence algorithm. The parameters are strings `x` and `y` with respective lengths `xlen` and `ylen`.

```
int F[MAX_LEN+1][MAX_LEN+1];

void SerialLCS( const char* x, size_t xlen, const char* y, size_t ylen )
```

```
{
    for( size_t i=1; i<=xlen; ++i )
        for( size_t j=1; j<=ylen; ++j )
            F[i][j] = x[i-1]==y[j-1] ? F[i-1][j-1]+1 :
                                       max(F[i][j-1],F[i-1][j]);
}
```

The kernel sets F[i][j] to the length of the longest common subsequence shared by x[0..i-1] and y[0..j-1]. It assumes that F[0][0..ylen] and F[0..xlen][0] have already been initialized to zero.

Figure 3 shows the data dependences for calculating F[i][j].

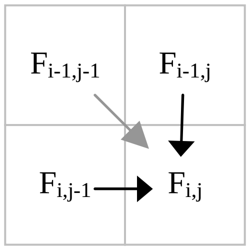

**Figure 3: Data dependences for longest common substring calculation.**

As Figure 4 shows, the gray diagonal dependence is the transitive closure of other dependences. Thus for parallelization purposes it is a redundant dependence that can be ignored.

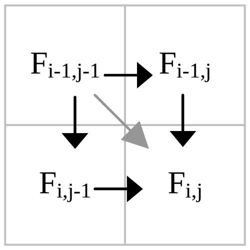

**Figure 4: Diagonal dependence is redundant.**

It is generally good to remove redundant dependences from consideration, because the atomic counting incurs a cost for each dependence considered.

Another consideration is grain size. Scheduling each F[i][j] element calculation separately is prohibitively expensive. A good solution is to aggregate the elements into contiguous blocks, and process the contents of a block serially. The blocks have the same dependence pattern, but at a block scale. Hence scheduling overheads can be amortized over blocks.

The parallel code follows. Each block consists of N×N elements. Each block has an associated atomic counter. Array Count organizes these counters for easy lookup. The code initializes the counters and then rolls a wavefront using `parallel_do`, starting with the block at the origin since it has no predecessors.

```
const int N = 64;
tbb::atomic<char> Count[MAX_LEN/N+1][MAX_LEN/N+1];

void ParallelLCS( const char* x, size_t xlen, const char* y, size_t ylen
) {
    // Initialize predecessor counts for blocks.
    size_t m = (xlen+N-1)/N;
    size_t n = (ylen+N-1)/N;
    for( int i=0; i<m; ++i )
        for( int j=0; j<n; ++j )
            Count[i][j] = (i>0)+(j>0);
    // Roll the wavefront from the origin.
    typedef pair<size_t,size_t> block;
    block origin(0,0);
    tbb::parallel_do( &origin, &origin+1,
        [=]( const block& b, tbb::parallel_do_feeder<block>& feeder ) {
            // Extract bounds on block
            size_t bi = b.first;
            size_t bj = b.second;
            size_t xl = N*bi+1;
            size_t xu = min(xl+N,xlen+1);
            size_t yl = N*bj+1;
            size_t yu = min(yl+N,ylen+1);
            // Process the block
            for( size_t i=xl; i<xu; ++i )
                for( size_t j=yl; j<yu; ++j )
                    F[i][j] = x[i-1]==y[j-1] ? F[i-1][j-1]+1 :
                                               max(F[i][j-1],F[i-1][j]);
            // Account for successors
            if( bj+1<n && --Count[bi][bj+1]==0 )
                feeder.add( block(bi,bj+1) );
            if( bi+1<m && --Count[bi+1][bj]==0 )
                feeder.add( block(bi+1,bj) );        }
    );
}
```

A regular structure simplifies implementation of the wavefront pattern, but is not required. The parallel preorder traversal in `examples/parallel_do/parallel_preorder` applies the wavefront pattern to traverse each node of a graph in parallel, subject to the constraint that a node is traversed after its predecessors are traversed. In that example, each node in the graph stores its predecessor count.

## Reference

The longest common substring example is adapted from “Wavefront Pattern” <http://www.cs.illinois.edu/homes/snir/PPP/patterns/wavefront.pdf> by Eun-Gyu Kim and Marc Snir.

# 6 Reduction

## Problem

Perform an associative reduction operation across a data set.

## Context

Many serial algorithms sweep over a set of items to collect summary information.

## Forces

The summary can be expressed as an associative operation over the data set, or at least is close enough to associative that reassociation does not matter.

## Solution

Two solutions exist in Intel® Threading Building Blocks (Intel® TBB). The choice on which to use depends upon several considerations:

- Is the operation commutative as well as associative?
- Are instances of the reduction type expensive to construct and destroy? For example, a floating point number is inexpensive to construct. A sparse floating-point matrix might be very expensive to construct.

Use `tbb::parallel_reduce` when the objects are inexpensive to construct. It works even if the reduction operation is not commutative. The Intel® TBB Tutorial describes how to use `tbb::parallel_reduce` for basic reductions.

Use `tbb::parallel_for` and `tbb::combinable` if the reduction operation is commutative and instances of the type are expensive.

If the operation is not precisely associative but a precisely deterministic result is required, use recursive reduction and parallelize it using `tbb::parallel_invoke`.

## Examples

The examples presented here illustrate the various solutions and some tradeoffs.

The first example uses t `tbb::parallel_reduce` to do a + reduction over sequence of type T. The sequence is defined by a half-open interval [first,last).

```
T AssocReduce( const T* first, const T* last, T identity ) {
    return tbb::parallel_reduce(
        // Index range for reduction
        tbb::blocked_range<const T*>(first,last),
        // Identity element
        identity,
        // Reduce a subrange and partial sum
        [&]( tbb::blocked_range<const T*> r, T partial_sum )->float {
            return std::accumulate( r.begin(), r.end(), partial_sum );
        },
        // Reduce two partial sums
        std::plus<T>()
    );
}
```

The third and fourth arguments to this form of parallel_reduce are a built in form of the agglomeration pattern. If there is an elementwise action to be performed before the reduction, incorporating it into the third argument (reduction of a subrange) may improve performance because of better locality of reference.

The second example assumes the + is commutative on T. It is a good solution when T objects are expensive to construct.

```
T CombineReduce( const T* first, const T* last, T identity ) {
    tbb::combinable<T> sum(identity);
    tbb::parallel_for(
        tbb::blocked_range<const T*>(first,last),
        [&]( tbb::blocked_range<const T*> r ) {
            sum.local() += std::accumulate(r.begin(), r.end(), identity);
        }
    );
    return sum.combine( []( const T& x, const T& y ) {return x+y;} );
}
```

Sometimes it is desirable to destructively use the partial results to generate the final result. For example, if the partial results are lists, they can be spliced together to form the final result. In that case use class `tbb::enumerable_thread_specific` instead of combinable. The ParallelFindCollisions example in Chapter 7 demonstrates the technique.

Floating-point addition and multiplication are almost associative. Reassociation can cause changes because of rounding effects. The techniques shown so far reassociate terms non-deterministically. Fully deterministic parallel reduction for a not quite associative operation requires using deterministic reassociation. The code below demonstrates this in the form of a template that does a + reduction over a sequence of values of type T.

```
template<typename T>
T RepeatableReduce( const T* first, const T* last, T identity ) {
    if( last-first<=1000 ) {
        // Use serial reduction
        return std::accumulate( first, last, identity );
    } else {
        // Do parallel divide-and-conquer reduction
        const T* mid = first+(last-first)/2;
        T left, right;
        tbb::parallel_invoke(
            [&]{left=RepeatableReduce(first,mid,identity);},
            [&]{right=RepeatableReduce(mid,last,identity);}
        );
        return left+right;
    }
}
```

The outer if-else is an instance of the agglomeration pattern for recursive computations. The reduction graph, though not a strict binary tree, is fully deterministic. Thus the result will always be the same for a given input sequence, assuming all threads do identical floating-point rounding.

The final example shows how a problem that typically is not viewed as a reduction can be parallelized by viewing it as a reduction. The problem is retrieving floating-point exception flags for a computation across a data set. The serial code might look something like:

```
    feclearexcept(FE_ALL_EXCEPT);
    for( int i=0; i<N; ++i )
        C[i]=A[i]*B[i];
    int flags = fetestexcept(FE_ALL_EXCEPT);
    if (flags & FE_DIVBYZERO) ...;
    if (flags & FE_OVERFLOW) ...;
    ...
```

The code can be parallelized by computing chunks of the loop separately, and merging floating-point flags from each chunk. To do this with `tbb:parallel_reduce`, first define a "body" type, as shown below.

```
struct ComputeChunk {
    int flags;              // Holds floating-point exceptions seen so far.
    void reset_fpe() {
        flags=0;
        feclearexcept(FE_ALL_EXCEPT);
    }
    ComputeChunk () {
        reset_fpe();
    }
    // "Splitting constructor" called by parallel_reduce when splitting a range into subranges.
    ComputeChunk ( const ComputeChunk&, tbb::split ) {
        reset_fpe();
    }
    // Operates on a chunk and collects floating-point exception state into flags member.
    void operator()( tbb::blocked_range<int> r ) {
        int end=r.end();
        for( int i=r.begin(); i!=end; ++i )
            C[i] = A[i]/B[i];
    // It is critical to do |= here, not =, because otherwise we
    // might lose earlier exceptions from the same thread.
        flags |= fetestexcept(FE_ALL_EXCEPT);
    }
    // Called by parallel_reduce when joining results from two subranges.
    void join( Body& other ) {
        flags |= other.flags;
    }
};
```

Then invoke it as follows:

```
    // Construction of cc implicitly resets FP exception state.
    ComputeChunk cc;
    tbb::parallel_reduce( tbb::blocked_range<int>(0,N), cc );
    if (cc.flags & FE_DIVBYZERO) ...;
    if (cc.flags & FE_OVERFLOW) ...;
    ...
```

# 7 Divide and Conquer

## Problem

Parallelize a divide and conquer algorithm.

## Context

Divide and conquer is widely used in serial algorithms. Common examples are quicksort and mergesort.

## Forces

- Problem can be transformed into subproblems that can be solved independently.
- Splitting problem or merging solutions is relatively cheap compared to cost of solving the subproblems.

## Solution

There are several ways to implement divide and conquer in Intel®Threading Building Blocks (Intel® TBB). The best choice depends upon circumstances.

- If division always yields the same number of subproblems, use recursion and `tbb::parallel_invoke`.
- If the number of subproblems varies, use recursion and `tbb::task_group`.
- If ultimate efficiency and scalability is important, use `tbb::task` and continuation passing style.

## Example

Quicksort is a classic divide-and-conquer algorithm. It divides a sorting problem into two subsorts. A simple serial version looks like:[1]

```
void SerialQuicksort( T* begin, T* end ) {
```

[1] Production quality quicksort implementations typically use more sophisticated pivot selection, explicit stacks instead of recursion, and some other sorting algorithm for small subsorts. The simple algorithm is used here to focus on exposition of the parallel pattern.

```
    if( end-begin>1  ) {
        using namespace std;
        T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
        swap( *begin, mid[-1] );
        SerialQuicksort( begin, mid-1 );
        SerialQuicksort( mid, end );
    }
}
```

The number of subsorts is fixed at two, so `tbb::parallel_invoke` provides a simple way to parallelize it. The parallel code is shown below:

```
void ParallelQuicksort( T* begin, T* end ) {
    if( end-begin>1 ) {
        using namespace std;
        T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
        swap( *begin, mid[-1] );
        tbb::parallel_invoke( [=]{ParallelQuicksort( begin, mid-1 );},
                              [=]{ParallelQuicksort( mid, end );} );
    }
}
```

Eventually the subsorts become small enough that serial execution is more efficient. The following variation, with changed parts in blue, does sorts of less than 500 elements using the earlier serial code.

```
void ParallelQuicksort( T* begin, T* end ) {
    if( end-begin>=500 ) {
        using namespace std;
        T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
        swap( *begin, mid[-1] );
        tbb::parallel_invoke( [=]{ParallelQuicksort( begin, mid-1 );},
                              [=]{ParallelQuicksort( mid, end );} );
    } else {
        SerialQuicksort( begin, end );
    }
}
```

The change is an instance of the Agglomeration pattern.

The next example considers a problem where there are a variable number of subproblems. The problem involves a tree-like description of a mechanical assembly. There are two kinds of nodes:

- Leaf nodes represent individual parts.
- Internal nodes represent groups of parts.

The problem is to find all nodes that collide with a target node. The following code shows a serial solution that walks the tree. It records in `Hits` any nodes that collide with `Target`.

```
std::list<Node*> Hits;
Node* Target;

void SerialFindCollisions( Node& x ) {
    if( x.is_leaf() ) {
        if( x.collides_with( *Target ) )
            Hits.push_back(&x);
    } else {
        for( Node::const_iterator y=x.begin(); y!=x.end(); ++y )
            SerialFindCollisions(*y);
    }
}
```

A parallel version is shown below.

```
typedef tbb::enumerable_thread_specific<std::list<Node*> > LocalList;
LocalList LocalHits;
Node* Target;     // Target node

void ParallelWalk( Node& x ) {
    if( x.is_leaf() ) {
        if( x.collides_with( *Target ) )
            LocalHits.local().push_back(&x);
    } else {
        // Recurse on each child y of x in parallel
        tbb::task_group g;
        for( Node::const_iterator y=x.begin(); y!=x.end(); ++y )
            g.run( [=]{ParallelWalk(*y);} );
        // Wait for recursive calls to complete
        g.wait();
    }
}

void ParallelFindCollisions( Node& x ) {
    ParallelWalk(x);
    for(LocalList::iterator i=LocalHits.begin(); i!=LocalHits.end(); ++i)
        Hits.splice( Hits.end(), *i );
}
```

The recursive walk is parallelized using class `task_group` to do recursive calls in parallel.

There is another significant change because of the parallelism that is introduced. Because it would be unsafe to update `Hits` concurrently, the parallel walk uses variable `LocalHits` to accumulate results. Because it is of type `enumerable_thread_specific`, each thread accumulates its own private result. The results are spliced together into Hits after the walk completes.

The results will *not* be in the same order as the original serial code.

If parallel overhead is high, use the agglomeration pattern. For example, use the serial walk for subtrees under a certain threshold.

# 8 GUI Thread

## Problem

A user interface thread must remain responsive to user requests, and must not get bogged down in long computations.

## Context

Graphical user interfaces often have a dedicated thread ("GUI thread") for servicing user interactions. The thread must remain responsive to user requests even while the application has long computations running. For example, the user might want to press a "cancel" button to stop the long running computation. If the GUI thread takes part in the long running computation, it will not be able to respond to user requests.

## Forces

- The GUI thread services an event loop.
- The GUI thread needs to offload work onto other threads without waiting for the work to complete.
- The GUI thread must be responsive to the event loop and not become dedicated to doing the offloaded work.

## Related

Non-Preemptive Priorities

Local Serializer

## Solution

The GUI thread offloads the work by firing off a task to do it using method `task::enqueue`. When finished, the task posts an event to the GUI thread to indicate that the work is done. The semantics of enqueue cause the task to eventually run on a worker thread distinct from the calling thread. The method is a new feature in Intel® Threading Building Blocks (Intel® TBB) 3.0.

Figure 5 sketches the communication paths. Items in black are executed by the GUI thread; items in blue are executed by another thread.

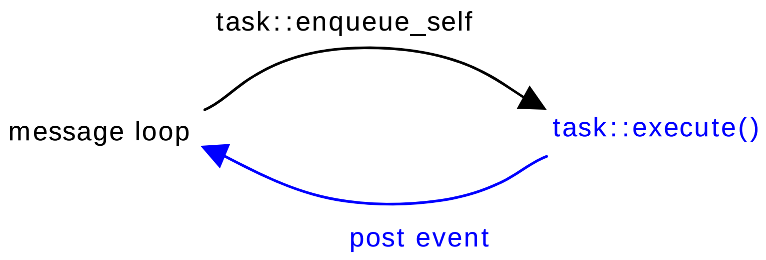

**Figure 5: GUI Thread pattern**

## Example

The example is for the Microsoft Windows* operating systems, though similar principles apply to any GUI using an event loop idiom. For each event, the GUI thread calls a user-defined function `WndProc`. to process an event. The key parts are in bold font.

```
// Event posted from enqueued task when it finishes its work.
const UINT WM_POP_FOO = WM_USER+0;

// Queue for transmitting results from enqueued task to GUI thread.
tbb::concurrent_queue<Foo> ResultQueue;

// GUI thread's private copy of most recently computed result.
Foo CurrentResult;

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM
lParam) {
    switch(msg) {
        case WM_COMMAND:
            switch (LOWORD(wParam)) {
                case IDM_LONGRUNNINGWORK:
                    // User requested a long computation. Delegate it to another thread.
                    LaunchLongRunningWork(hWnd);
                    break;
                case IDM_EXIT:
                    DestroyWindow(hWnd);
                    break;
                default:
                    return DefWindowProc(hWnd, msg, wParam, lParam);
            }
            break;
        case WM_POP_FOO:
            // There is another result in ResultQueue for me to grab.
            ResultQueue.try_pop(CurrentResult);
            // Update the window with the latest result.
            RedrawWindow( hWnd, NULL, NULL, RDW_ERASE|RDW_INVALIDATE );
            break;
        case WM_PAINT:
            Repaint the window using CurrentResult
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc( hWnd, msg, wParam, lParam );
    }
    return 0;
}
```

The GUI thread processes long computations as follows:

1. The GUI thread calls LongRunningWork, which hands off the work to a worker thread and immediately returns.

2. The GUI thread continues servicing the event loop. If it has to repaint the window, it uses the value of CurrentResult, which is the most recent Foo that it has seen.

When a worker finishes the long computation, it pushes the result into ResultQueue, and sends a message WM_POP_FOO to the GUI thread.

3. The GUI thread services a WM_POP_FOO message by popping an item from ResultQueue into CurrentResult. The try_pop always succeeds because there is exactly one WM_POP_FOO message for each item in ResultQueue.

Routine LaunchLongRunningWork creates a root task and launches it using method task::enqeueue. The task is a root task because it has no successor task waiting on it.

```
class LongTask: public tbb::task {
    HWND hWnd;
    tbb::task* execute() {
        Do long computation
        Foo x = result of long computation
        ResultQueue.push( x );
        // Notify GUI thread that result is available.
        PostMessage(hWnd,WM_POP_FOO,0,0);
        return NULL;
    }
public:
    LongTask( HWND hWnd_ ) : hWnd(hWnd_) {}
};

void LaunchLongRunningWork( HWND hWnd ) {
    LongTask* t = new( tbb::task::allocate_root() ) LongTask(hWnd);
    tbb::task::enqueue(*t);
}
```

It is essential to use method `task::enqueue` and not method `task::spawn`. The reason is that method `enqueue_self` ensures that the task eventually executes when resources permit, even if no thread explicitly waits on the task. In contrast, method spawn may postpone execution of the task until it is explicitly waited upon.

The example uses a `concurrent_queue` for workers to communicate results back to the GUI thread. Since only the most recent result matters in the example, and alternative would be to use a shared variable protected by a mutex. However, doing so would block the worker while the GUI thread was holding a lock on the mutex, and vice versa. Using `concurrent_queue` provides a simple robust solution.

If two long computations are in flight, there is a chance that the first computation completes after the second one. If displaying the result of the most recently requested computation is important, then associate a request serial number with the computation. The GUI thread can pop from `ResultQueue` into a temporary variable, check the serial number, and update `CurrentResult` only if doing so advances the serial number.

See Non-Preemptive Priorities for how to implement priorities. See Local Serializer for how to force serial ordering of certain tasks.

# 9 Non-Preemptive Priorities

## Problem

Choose the next work item to do, based on priorities.

## Context

The scheduler in Intel® Threading Building Blocks (Intel® TBB) chooses tasks using rules based on scalability concerns. The rules are based on the order in which tasks were spawned or enqueued, and are oblivious to the contents of tasks. However, sometimes it is best to choose work based on some kind of priority relationship.

## Forces

- Given multiple work items, there is a rule for which item should be done next that is *not* the default Intel® TBB rule.
- Preemptive priorities are not necessary. If a higher priority item appears, it is not necessary to immediately stop lower priority items in flight. If preemptive priorities are necessary, then non-preemptive tasking is inappropriate. Use threads instead.

## Solution

Put the work in a shared work pile. Decouple tasks from specific work, so that task execution chooses the actual piece of work to be selected from the pile.

## Example

The following example implements three priority levels. The user interface for it and top-level implementation follow:

```
enum Priority {
    P_High,
    P_Medium,
    P_Low
};

template<typename Func>
void EnqueueWork( Priority p, Func f ) {
    WorkItem* item = new ConcreteWorkItem<Func>( p, f );
    ReadyPile.add(item);
}
```

The caller provides a priority *p* and a functor *f* to routine `EnqueueWork`. The functor may be the result of a lambda expression. `EnqueueWork` packages *f* as a `WorkItem` and adds it to global object `ReadyPile`.

Class `WorkItem` provides a uniform interface for running functors of unknown type:

```
// Abstract base class for a prioritized piece of work.
class WorkItem {
public:
    WorkItem( Priority p ) : priority(p) {}
    // Derived class defines the actual work.
    virtual void run() = 0;
    const Priority priority;
};

template<typename Func>
class ConcreteWorkItem: public WorkItem {
    Func f;
    /*override*/ void run() {
        f();
        delete this;
    }
public:
    ConcreteWorkItem( Priority p, const Func& f_ ) :
        WorkItem(p), f(f_)
    {}
};
```

Class `ReadyPile` contains the core pattern. It maintains a collection of work and fires off tasks that choose work from the collection:

```
class ReadyPileType {
    // One queue for each priority level
    tbb::concurrent_queue<WorkItem*> level[P_Low+1];
public:
    void add( WorkItem* item ) {
        level[item->priority].push(item);
        tbb::task::enqueue(*new(tbb::task::allocate_root()) RunWorkItem);
    }
    void runNextWorkItem() {
        // Scan queues in priority order for an item.
        WorkItem* item=NULL;
        for( int i=P_High; i<=P_Low; ++i )
            if( level[i].try_pop(item) )
                break;
        assert(item);
        item->run();
    }
};

ReadyPileType ReadyPile;
```

The task enqueued by add(item) does *not* necessarily execute that item. The task executes runNextWorkItem(), which may find a higher priority item. There is one task for each item, but the mapping resolves when the task actually executes, not when it is created.

Here are the details of class RunWorkItem:

```
class RunWorkItem: public tbb::task {
    /*override*/tbb::task* execute(); // Private override of virtual
method
};
…
tbb::task* RunWorkItem::execute() {
    ReadyPile.runNextWorkItem();
    return NULL;
};
```

RunWorkItem objects are fungible. They enable the Intel® TBB scheduler to choose when to do a work item, not which work item to do. The override of virtual method task::execute is private because all calls to it are dispatched via base class task.

Other priority schemes can be implemented by changing the internals for ReadyPileType. A priority queue could be used to implement very fine grained priorities.

The scalability of the pattern is limited by the scalability of ReadyPileType. Ideally scalable concurrent containers should be used for it.

# 10 Local Serializer

## Context

Consider an interactive program. To maximize concurrency and responsiveness, operations requested by the user can be implemented as tasks. The order of operations can be important. For example, suppose the program presents editable text to the user. There might be operations to select text and delete selected text. Reversing the order of "select" and "delete" operations on the same buffer would be bad. However, commuting operations on different buffers might be okay. Hence the goal is to establish serial ordering of tasks associated with a given object, but not constrain ordering of tasks between different objects.

## Forces

- Operations associated with a certain object must be performed in serial order.
- Serializing with a lock would be wasteful because threads would be waiting at the lock when they could be doing useful work elsewhere.

## Solution

Sequence the work items using a FIFO (first-in first-out structure). Always keep an item in flight if possible. If no item is in flight when a work item appears, put the item in flight. Otherwise, push the item onto the FIFO. When the current item in flight completes, pop another item from the FIFO and put it in flight.

The logic can be implemented without mutexes, by using `concurrent_queue` for the FIFO and `atomic<int>` to count the number of items waiting and in flight. The example explains the accounting in detail.

## Example

The following example builds on the Non-Preemptive Priorities example to implement local serialization in addition to priorities. It implements three priority levels and local serializers. The user interface for it follows:

```
enum Priority {
    P_High,
    P_Medium,
    P_Low
};
```

```
template<typename Func>
void EnqueueWork( Priority p, Func f, Serializer* s=NULL );
```

Template function `EnqueueWork` causes functor *f* to run when the three constraints in Table 1 are met.

**Table 1: Implementation of Constraints**

| Constraint | Resolved by class... |
|---|---|
| Any prior work for the `Serializer` has completed. | `Serializer` |
| A thread is available. | `RunWorkItem` |
| No higher priority work is ready to run. | `ReadyPileType` |

Constraints on a given functor are resolved from top to bottom in the table. The first constraint does not exist when s is NULL. The implementation of EnqueueWork packages the functor in a `SerializedWorkItem` and routes it to the class that enforces the first relevant constraint between pieces of work.

```
template<typename Func>
void EnqueueWork( Priority p, Func f, Serializer* s=NULL ) {
    WorkItem* item = new SerializedWorkItem<Func>( p, f, s );
    if( s )
        s->add(item);
    else
        ReadyPile.add(item);
}
```

A `SerializedWorkItem` is derived from a `WorkItem`, which serves as a way to pass around a prioritized piece of work without knowing further details of the work.

```
// Abstract base class for a prioritized piece of work.
class WorkItem {
public:
    WorkItem( Priority p ) : priority(p) {}
    // Derived class defines the actual work.
    virtual void run() = 0;
    const Priority priority;
};

template<typename Func>
class SerializedWorkItem: public WorkItem {
    Serializer* serializer;
    Func f;
    /*override*/ void run() {
        f();
        Serializer* s = serializer;
        // Destroy f before running Serializer's next functor.
        delete this;
        if( s )
```

```
            s->noteCompletion();
    }
public:
    SerializedWorkItem( Priority p, const Func& f_, Serializer* s ) :
        WorkItem(p), serializer(s), f(f_)
    {}
};
```

Base class `WorkItem` is the same as class `WorkItem` in the example for Non-Preemptive Priorities. The notion of serial constraints is completely hidden from the base class, thus permitting the framework to extend other kinds of constraints or lack of constraints. Class `SerializedWorkItem` is essentially `ConcreteWorkItem` from the other example, extended with a `Serializer` aspect.

Virtual method `run()` is invoked when it becomes time to run the functor. It performs three steps:

1. Run the functor
2. Destroy the functor.
3. Notify the `Serializer` that the functor completed, and thus unconstraining the next waiting functor.

Step 3 is the difference from the operation of ConcreteWorkItem::run. Step 2 could be done after step 3 in some contexts to increase concurrency slightly. However, the presented order is recommended because if step 2 takes non-trivial time, it likely has side effects that should complete before the next functor runs.

Class `Serializer` implements the core of the Local Serializer pattern:

```
class Serializer {
    tbb::concurrent_queue<WorkItem*> queue;
    tbb::atomic<int> count;            // Count of queued items and in-flight item
    void moveOneItemToReadyPile() {  // Transfer item from queue to ReadyPile
        WorkItem* item;
        queue.try_pop(item);
        ReadyPile.add(item);
    }
public:
    void add( WorkItem* item ) {
        queue.push(item);
        if( ++count==1 )
            moveOneItemToReadyPile();
    }
    void noteCompletion() {            // Called when WorkItem completes.
        if( --count!=0 )
            moveOneItemToReadyPile();
    }
};
```

The class maintains two members:

- A queue of WorkItem waiting for prior work to complete.
- A count of queued or in-flight work.

Mutexes are avoided by using `concurrent_queue<WorkItem*>` and `atomic<int>` along with careful ordering of operations. The transitions of count are the key understanding how class `Serializer` works.

- If method add increments count from 0 to 1, this indicates that no other work is in flight and thus the work should be moved to the ReadyPile.
- If method `noteCompletion` decrements count and it is *not* from 1 to 0, then the queue is non-empty and another item in the queue should be moved to `ReadyPile`.

Class `ReadyPile` is explained in the example for Non-Preemptive Priorities.

If priorities are not necessary, there are two variations on method `moveOneItem`, with different implications.

- Method `moveOneItem` could directly invoke `item->run()`. This approach has relatively low overhead and high thread locality for a given `Serializer`. But it is unfair. If the `Serializer` has a continual stream of tasks, the thread operating on it will keep servicing those tasks to the exclusion of others.
- Method `moveOneItem` could invoke `task::enqueue` to enqueue a task that invokes `item->run()`. Doing so introduces higher overhead and less locality than the first approach, but avoids starvation.

The conflict between fairness and maximum locality is fundamental. The best resolution depends upon circumstance.

The pattern generalizes to constraints on work items more general than those maintained by class Serializer. A generalized `Serializer::add` determines if a work item is unconstrained, and if so, runs it immediately. A generalized `Serializer::noteCompletion` runs all previously constrained items that have become unconstrained by the completion of the current work item. The term “run” means to run work immediately, or if there are more constraints, forwarding the work to the next constraint resolver.

# 11 Fenced Data Transfer

## Problem

Write a message to memory and have another processor read it on hardware that does not have a sequentially consistent memory model.

## Context

The problem normally arises only when unsynchronized threads concurrently act on a memory location, or are using reads and writes to create synchronization. High level synchronization constructs normally include mechanisms that prevent unwanted reordering.

Modern hardware and compilers can reorder memory operations in a way that preserves the order of a thread's operation from its viewpoint, but not as observed by other threads. A serial common idiom is to write a message and mark it as ready to ready as shown in the following code:

```
bool Ready;
std::string Message;

void Send( const std::string& src ) {    // Executed by thread 1
    Message=src;
    Ready = true;
}

bool Receive( std::string& dst ) {      // Executed by thread 2
    bool result = Ready;
    if( result ) dst=Message;
    return result;                    // Return true if message was received.
}
```

Two key assumptions of the code are:

a. Ready does not become true until Message is written.

b. Message is not read until Ready becomes true.

These assumptions are trivially true on uniprocessor hardware. However, they may break on multiprocessor hardware. Reordering by the hardware or compiler can cause the sender's writes to appear out of order to the receiver (thus breaking condition a) or the receiver's reads to appear out of order (thus breaking condition b).

## Forces

- Creating synchronization via raw reads and writes.

## Related

Lazy Initialization

## Solution

Change the flag from `bool` to `tbb::atomic<bool>` for the flag that indicates when the message is ready. Here is the previous example, with modifications colored blue.

```
tbb::atomic<bool> Ready;
std::string Message;

void Send( const std::string& src ) {    // Executed by thread 1
    Message=src;
    Ready = true;
}

bool Receive( std::string& dst ) {       // Executed by thread 2
    bool result = Ready;
    if( result ) dst=Message;
    return result;                       // Return true if message was received.
}
```

A write to a `tbb::atomic` value has *release* semantics, which means that all of its prior writes will be seen before the releasing write. A read from `tbb::atomic` value has *acquire* semantics, which means that all of its subsequent reads will happen after the acquiring read. The implementation of `tbb::atomic` ensures that both the compiler and the hardware observe these ordering constraints.

## Variations

Higher level synchronization constructs normally include the necessary *acquire* and *release* fences. For example, mutexes are normally implemented such that acquisition of a lock has *acquire* semantics and release of a lock has *release* semantics. Thus a thread that acquires a lock on a mutex always sees any memory writes done by another thread before it released a lock on that mutex.

## Non Solutions

Mistaken solutions are so often proposed that it is worth understanding why they are wrong.

One common mistake is to assume that declaring the flag with the `volatile` keyword solves the problem. Though the `volatile` keyword forces a write to happen immediately, it generally has no effect on the visible ordering of that write with respect to other memory operations. An exception to this rule are processors from the Intel® Itanium® processor family, which by convention assign acquire semantics to `volatile` reads and release semantics to volatile writes.

Another mistake is to assume that conditionally executed code cannot happen before the condition is tested. However, the compiler or hardware may speculatively hoist the conditional code above the condition.

Similarly, it is a mistake to assume that a processor cannot read the target of a pointer before reading the pointer. A modern processor does not read individual values from main memory. It reads cache lines. The target of a pointer may be in a cache line that has already been read before the pointer was read, thus giving the appearance that the processor presciently read the pointer target.

# 12 Lazy Initialization

## Problem

Perform an initialization the first time it is needed.

## Context

Initializing data structures lazily is a common technique. Not only does it avoid the cost of initializing unused data structures, it is often a more convenient way to structure a program.

## Forces

- Threads share access to an object.
- The object should not be created until the first access.

The second force covers several possible motivations:

- The object is expensive to create and creating it early would slow down program startup.
- It is not used in every run of the program.
- Early initialization would require adding code where it is undesirable for readability or structural reasons.

## Related

Fenced Data Transfer

## Solutions

A parallel solution is substantially trickier, because it must deal with several concurrency issues.

**Races:** If two threads attempt to simultaneously access to the object for the first time, and thus cause creation of the object, the race must be resolved in a way that both threads end up with a reference to the same object of type T.

**Memory leaks:** In the event of a race, the implementation must ensure that any extra transient T objects are cleaned up.

**Memory consistency:** If thread X executes `value=new T()`, all other threads must see stores by `new T()` occur before the assignment `value=` .

**Deadlock:** What if the constructor of `T()` requires acquiring a lock, but the current holder of that lock is also racing to access the object for the first time?

There are two solutions. One is based on double-check locking. The other relies on compare-and-swap. Because the tradeoffs and issues are subtle, most of the discussion is in the following examples section.

## Examples

An Intel® TBB implementation of the "double-check" pattern is shown below:

```
template<typename T, typename Mutex=tbb::mutex>
class lazy {
    tbb::atomic<T*> value;
    Mutex* mut;
public:
    lazy() : value(NULL) {}
    ~lazy() {delete value;}
    T& get() {
        if( !value ) {                      // Read of value has acquire semantics.
            Mutex::scoped_lock lock;
            if( !value ) value = new T();   // Write of value has release semantics
        }
        return *value;
    }
};
```

The name comes from the way that the pattern deals with races.  There is one check done without locking and one check done after locking. The first check handles the presumably common case that the initialization has already been done, without any locking. The second check deals with cases where two threads both see an uninitialized value, and both try to acquire the lock. In that case, the second thread to acquire the lock will see that the initialization has already occurred.

If `T()` throws an exception, the solution is correct because `value` will still be NULL and the mutex unlocked when object `lock` is destroyed.

The solution correctly addresses memory consistency issues. A write to a `tbb::atomic` value has *release* semantics, which means that all of its prior writes will be seen before the releasing write. A read from `tbb::atomic` value has *acquire* semantics, which means that all of its subsequent reads will happen after the acquiring read.  Both of these properties are critical to the solution. The releasing write ensures that the construction of `T()` is seen to occur before the assignment to value. The acquiring read ensures that when the caller reads from `*value`, the reads occur after the "`if(!value)`" check. The release/acquire is essentially the Fenced Data Transfer pattern, where the "message" is the fully constructed instance `T()`, and the "ready" flag is the pointer value.

The solution described involves blocking threads while initialization occurs. Hence it can suffer the usual pathologies associated with blocking. For example, if the thread first acquires the lock is suspended by the OS, all other threads will have to wait until that thread resumes. A lock-free variation avoids this problem by making all contending threads attempt initialization, and atomically deciding which attempt succeeds.

An Intel® TBB implementation of the non-blocking variant follows. It also uses double-check, but without a lock.

```
template<typename T>
class lazy {
    tbb::atomic<T*> value;
    Mutex* mut;
public:
    lazy() : value(NULL) {}
    ~lazy() {delete value;}
    T& get() {
        if( !value ) {
            T* tmp = new T();
            if( value.compare_and_swap(tmp,NULL)!=NULL )
                // Another thread installed the value, so throw away mine.
                delete tmp;
        }
        return value;
    }
};
```

The second check is performed by the expression `value.compare_and_swap(tmp,NULL)!=NULL`, which conditionally assigns `value=tmp` if `value==NULL`, and returns true if the old `value` was NULL. Thus if multiple threads attempt simultaneous initialization, the first thread to execute the compare_and_swap will set value to point to its T object. Other contenders that execute the compare_and_swap will get back a non-NULL pointer, and know that they should delete their transient T objects.

As with the locking solution, memory consistency issues are addressed by the semantics of `tbb::atomic`. The first check has *acquire* semantics and the compare_and_swap has both *acquire* and *release* semantics.

## Reference

A sophisticated way to avoid the acquire fence for a read is Mike Burrow's algorithm <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2660.htm>.

# 13 Reference Counting

## Problem

Destroy an object when it will no longer be used.

## Context

Often it is desirable to destroy an object when it is known that it will not be used in the future. Reference counting is a common serial solution that extends to parallel programming if done carefully.

## Forces

- If there are cycles of references, basic reference counting is insufficient unless the cycle is explicitly broken.
- Atomic counting is relatively expensive in hardware.

## Solution

Thread-safe reference counting is like serial reference counting, except that the increment/decrement is done atomically, and the decrement and test "count is zero?" must act as a single atomic operation. The following example uses tbb::atomic<int> to achieve this.

```
template<typename T>
class counted {
    tbb::atomic<int> my_count;
    T value;
public:
    // Construct object with a single reference to it.
    counted() {my_count=1;}
    // Add reference
    void add_ref() {++my_count;}
    // Remove reference.  Return true if it was the last reference.
    bool remove_ref() {return --my_count==0;}
    // Get reference to underlying object
    T& get() {
        assert(my_count>0);
        return my_value;
    }
};
```

It is incorrect to use a separate read for testing if the count is zero. The following code would be an incorrect implementation of method `remove_ref()` because two threads might both execute the decrement, and then both read `my_count` as zero. Hence two callers would both be told incorrectly that they had removed the last reference.

```
        --my_count;
        return my_count==0; // WRONG!
```

The decrement may need to have a *release* fence so that any pending writes complete before the object is deleted.

There is no simple way to atomically copy a pointer and increment its reference count, because there will be a timing hole between the copying and the increment where the reference count is too low, and thus another thread might decrement the count to zero and delete the object. Two way to address the problem are “hazard pointers” and “pass the buck”. See the references at the end of this chapter for details.

## Variations

Atomic increment/decrement can more than an order of magnitude more expensive than ordinary increment/decrement. The serial optimization of eliminating redundant increment/decrement operations becomes more important with atomic reference counts.

Weighted reference counting can be used to reduce costs if the pointers are unshared but the referent is shared. Associate a *weight* with each pointer. The reference count is the sum of the weights. A pointer *x* can be copied as a pointer *x*' without updating the reference count by splitting the original weight between x and x'. If the weight of x is too low to split, then first add a constant W to the reference count and weight of x.

## References

D. Bacon and V.T. Rajan, “Concurrent Cycle Collection in Reference Counted Systems” in *Proc. European Conf. on Object-Oriented Programming* (June 2001). Describes a garbage collector based on reference counting that does collect cycles.

M. Michael, “Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects” in IEEE Transactions on Parallel and Distributed Systems (June 2004). Describes the “hazard pointer” technique.

M. Herlihy, V. Luchangco, and M. Moir, “The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures” in *Proceedings of the 16th International Symposium on Distributed Computing* (Oct. 2002). Describes the “pass the buck” technique.

# 14 Compare and Swap Loop

## Problem

Atomically update a scalar value so that a predicate is satisfied.

## Context

Often a shared variable must be updated atomically, by a transform that maps its old value to a new value. The transform might be a transition of a finite state machine, or recording global knowledge. For instance, the shared variable might be recording the maximum value that any thread has seen so far.

## Forces

- The variable is read and updated by multiple threads.
- The hardware implements "compare and swap" for a variable of that type.
- Protecting the update with a mutex is to be avoided.

## Related

Reduction

Reference counting

## Solution

The solution is to atomically snapshot the current value, and then use `atomic<T>::compare_and_swap` to update it. Retry until the `compare_and_swap` succeeds. In some cases it may be possible to exit before the `compare_and_swap` succeeds because the current value meets some condition.

The template below does the update `x=F(x)` atomically.

```
// Atomically perform x=F(x).
template<typename F, typename T>
void AtomicUpdate( atomic<T>& x, F f ) {
    int o;
    do {
        // Take a snapshot
        int o = x;
        // Attempt to install new value computed from snapshot
    } while( x.compare_and_swap(o,f(o))!=o );
}
```

It is critical to take a snapshot and use it for intermediate calculations, because the value of X may be changed by other threads in the meantime.

The following code shows how the template might be used to maintain a global maximum of any value seen by RecordMax.

```
// Atomically perform UpperBound = max(UpperBound,y)
void RecordMax( int y ) {
    extern atomic<int> UpperBound;
    AtomicUpdate(UpperBound, [&](int value){return std::max(value,y);} );
}
```

When y is not going to increase UpperBound, the call to AtomicUpdate will waste time doing the redundant operation compare_and_swap(o,o). In general, this kind of redundancy can be eliminated by making the loop in AtomicUpdate exit early if F(o)==o. In this particular case where F==std::max<int>, that test can be further simplified. The following custom version of RecordMax has the simplified test.

```
// Atomically perform UpperBound =max(UpperBound,y)
void RecordMax( int y ) {  .
    extern atomic<int> UpperBound;
    do {
        // Take a snapshot
        int o = UpperBound;
        // Quit if snapshot meets condition.
        if( o>=y ) break;
        // Attempt to install new value.
    } while( UpperBound.compare_and_swap(y,o)!=o );
}
```

Because all participating threads modify a common location, the performance of a compare and swap loop can be poor under high contention. Thus the applicability of more efficient patterns should be considered first. In particular:

- If the overall purpose is a reduction, use the reduction pattern instead.
- If the update is addition or subtraction, use atomic<T>::fetch_and_add. If the update is addition or subtraction by one, use atomic<T>::operater++ or atomic<T>::operator--. These methods typically employ direct hardware support that avoids a compare and swap loop.

***CAUTION:*** If use compare_and_swap to update links in a linked structure, be sure you understand if the “ABA problem” is an issue. See the Internet for discourses on the subject.

# General References

This section lists general references. References specific to a pattern are listed at the end of the chapter for the pattern.

- E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns* (1995).
- Berkeley Pattern Language for Parallel Programming, http://parlab.eecs.berkeley.edu/wiki/patterns
- T. Mattson, B. Sanders, B. Massingill. *Patterns for Parallel Programming* (2005).
- ParaPLoP 2009, http://www.upcrc.illinois.edu/workshops/paraplop09/program.html
- ParaPLoP 2010, http://www.upcrc.illinois.edu/workshops/paraplop10/program.html
- Eun-Gyu Kim and Marc Snir, “Parallel Programming Patterns”, http://www.cs.illinois.edu/homes/snir/PPP/index.html