Comparative analysis between QuickThread and Intel® Threading Building Blocks (TBB)

This document is a comparative analysis of QuickThread, referred to as QT and Intel Threading Building Blocks, referred to at TBB. The format of this document is to follow the TBB tutorial document and make comparisons amongst Single Threaded (serial) programming TBB programming of the same task, and QT programming of the same task. By doing so, you will see the impact (burden), and benefits of each threading tool.

Benefits

- Both QT and TBB are task model threading tools
- Both QT and TBB are targeted towards performance
- Both QT and TBB are compatible with other threading packages
- Both QT and TBB emphasize scalable, data parallel, programming
- TBB relies on generic programming, QT generic programming is by preference
- QT has less programming effort to integrate into your applications
- QT has higher performance under heavy workloads.

Package Contents

Intel TBB includes dynamic shared library, C++ header files and C++ code examples for Windows and Linux.

QuickThread includes a static library, C++ and Fortran header files and C++ and Fortran code examples for Windows.

An additional benefit of QuickThread is the ability to incorporate it into your Fortran applications. This document will not cover Fortran integration as it is a comparative analysis of the C++ implementations of threading tools.

Debug and Versus Release Libraries

Intel TBB comes with debug and release versions of the dynamic shared runtime library.

QuickThread static library contains the thread scheduler and a few support functions in Debug and Release configurations (both x32 and x64 architectures). All language extensions interfacing to the scheduler and support routines are provided in source. User can compile these in Release or Debug as required.

Scalable Memory Allocator

Intel TBB has a scalable memory allocator.
QuickThread has a NUMA optimized scalable memory allocator.

Additional scalable memory allocators are available from various locations such as sourceforge.net or threadingbuildingblocks.org.

Installation Windows

Intel TBB (follow instructions)

QuickThread – Extract to folder of choice (e.g. C:\QuickThread or C:\Program Files\QuickThread)
Set environment variables if (as) desired (INCLUDE, LIB, etc…)

Initializing the Terminating the Library

Intel TBB

```
#include "tbb/task_scheduler_init.h"
using namespace tbb;
int main()
{
      task_scheduler_init init;
      ...
      return 0;
}
```

Intel TBB may have one or more **task_scheduler_init** objects

QuickThread

```
#include "QuickThread.h"
using namespace qt;
int main()
{
      qtInit init(-1,-1);     // use defaults
      ... // Your parallel code here
      return 0;
}
```

Or

```
#include "QuickThread.h"
using namespace qt;
int main()
{
      qtInit init(4,2); // 4 compute and 2 I/O threads
      ... // Your parallel code here
      return 0;
```

```
	}
```

Or

```
	#include "QuickThread.h"
	using namespace qt;
	int main()
	{
		qtInit init;      // delay initialization of thread pool
		// Optional configuration of init object
		// ...
		if(init.StartQT()) ReportError();
		... // Your parallel code here
		if(init.EndQT()) ReportError();
		return 0;
	}
```

Or

```
	#include "QuickThread.h"
	using namespace qt;
	int main()
	{
		... // (Some of) your serial code here
		for(int i=0; i<count; ++i)
		{
			qtInit init(-1,-1);     // use defaults
			... // Your parallel code here
		}
		... // Your serial code here
		return 0;
	}
```

QuickThread uses one **qtInit** object to specify and configure its thread pools.

In TBB the initialization object creates one class of thread pool (compute class).

In QuickThread a single **qtInit** object can declare two classes of thread pools:

Computation class
I/O class

The defaults for the numbers of threads are: the number of computational threads equals the number of cores (or hardware threads in the case of HT) available, and the default number of I/O class threads is 1. You can specify your preference to the numbers of threads per class in the ctor or together with additional optional preferences between the instantiation of the object qtInit (without arguments) and the function call to start the QuickThread thread pool.

Under normal programming requirements the number of threads created for the computation class of threads would never need to exceed the number of available cores (hardware threads). However, during transition from pre-QuickThread to QuickThread you may have a need to create additional computational threads. An example would be if some of your older pieces of code perform small amounts of I/O or WaitForSingleEvent-like messaging. Although you can specify more computational threads than you have hardware threads we recommend that you allocate additional I/O class threads for this purpose. The computational class threads are (usually) affinity pinned, while the I/O class threads are not and can roam from core to core.

parallel_for

Both QuickThread and TBB have a parallel_for template. Both support C++0x Lambda functions (which are not used in this analysis of parallel_for).

QuickThread has optional arguments:

qtPlacement permitting the programmer to select the preferred affinities of the thread team used for the parallel for and/or other characteristics of the task enqueue (to Compute or I/O class of thread, FIFO or quazi-LIFO).

qtControl permitting non-blocking parallel_for and/or completion task specification.

Serial code

```
void SerialApplyFoo( float a[], size_t n )
{
      for( size_t i=0; i<n; ++i )
            Foo(a[i]);
}
```

Intel TBB

parallel_for with grain size

```
...
#include "tbb/blocked_range.h"
// TBB requires you to create a class containing operator()
class ApplyFoo
{
      float *const my_a;
      public:
      void operator()( const blocked_range<size_t>& r ) const
      {
            float *a = my_a;
            for( size_t i=r.begin(); i!=r.end(); ++i )
                  Foo(a[i]);
      }
      ApplyFoo( float a[] ) :
      my_a(a)
      {}
};
...
#include "tbb/parallel_for.h"
void ParallelApplyFoo( float a[], size_t n )
{
      // TBB uses blocked_range object
      parallel_for(
            blocked_range<size_t>(0,n,IdealGrainSize),
            ApplyFoo(a) );
}
```

QuickThread

parallel_for with grain size

```
. . .
// QuickThread requires half open iteration space on for task
void ParallelApplyFoo(size_t iBegin, size_t iEnd,  float a[])
{
      // Call existing serial code
      SerialApplyFoo(&a[iBegin], iEnd-iBegin);
}
. . .

parallel_for( IdealGrainSize, ParallelApplyFoo, 0, n, a);
```

parallel_for without grain size

```
parallel_for( ParallelApplyFoo, 0, n, a);
```

QuickThread tasks are standard functions returning void, some contain required arguments and may contain a reasonable number of arbitrary user arguments (currently 9 arguments is upper limit). Control parameters, e.g `IdealGrainSize`, precede the functor in the argument list.

The task template for parallel_for has two required arguments: those of the half open range of the iteration space (iBegin and iEnd). In this example the optional user argument is the base of the array. For different functions this could be an STL vector.

The principal advantages of using QuickThread in this example are:

- Re-use of existing serial code
- Eliminates the requirement of creating an additional class and class functions for running the task.

QuickThread has advanced thread scheduling capabilities. You can specify a subset of all thread such as only the threads sharing a cache level (L1, L2, L3, M0) of that of the thread issuing the parallel_for. Or other cache related characteristics. Examples:

parallel_for bound to L2 cache of calling thread

```
parallel_for(L2$, ParallelApplyFoo, 0, n, a);
```

Specifies a parallel_for using the current thread plus any threads sharing the current thread's L2 cache.

On a typical current generation processor this would schedule two threads: the current thread and the one other thread sharing the L2 cache. If the processor has HT this might involve scheduling to four threads. Additional HT threads can be excluded from the thread pool at initialization time by use of an option.

parallel_for opportunistic scheduling

```
    parallel_for(Waiting_L2$, ParallelApplyFoo, 0, n, a);
```

Specifies a parallel_for using the current thread plus any threads sharing the current thread's L2 cache *provided* the other threads are available for work.

In scheduling for all threads sharing L2 cache the programmer is not taking into consideration the availability of those threads to perform the work. While at an outer layer of the application you may be reasonably assured that all the other threads sharing your thread's L2 cache are idle, as you traverse deeper into the application (nested thread levels) those additional threads sharing your thread's L2 cache may be busy performing other tasks. In this situation it would be counter productive to split the loop up into multiple pieces for those thread that are not available.

parallel_for current socket scheduling

```
    parallel_for(L3$, ParallelApplyFoo, 0, n, a);
```
or
```
    parallel_for(M0$, ParallelApplyFoo, 0, n, a);
```

parallel_for different socket scheduling

```
    parallel_for(L3$+ExcluedMyCacheLevel$,ParallelApplyFoo,0,n,a);
```
or
```
    parallel_for(M0$+ExcluedMyCacheLevel$,ParallelApplyFoo,0,n,a);
```

Including ExcluedMyCacheLevel$ you can select the socket with the most available threads.

parallel_for one thread per cache level scheduling

```
    parallel_for(OneEach_L2$,ParallelApplyFoo, 0, n, a);
```

QuickThread provides for task distribution in other ways. The above performs a parallel_for but to restrict the scheduling to one thread per each L2 cache.

You may ask: Why would you want to do this?

A good example of OneEach_L2$ distribution would be in dividing a matrix operation into tiles. The rows of the matrix can be distributed to each L2 and the columns as referenced by those rows, using parallel_for(L2$, …) would be performed by those threads sharing the L2 with each of the respective rows threads. In this manner you have control over locality of data and can improve cache utilization.

parallel_for non-blocking

By specifying a qtControl object in the argument list the parallel_for will not block. i.e. code execution of the main thread passes through the parallel_for while the other threads are working on the task(s) of the parallel_for.
```
    double* A = new double[nSize];
```

```
double* B = new double[nSize];
double* AB = new double[nSize];     // AB = A + B
double* C = new double[nSize];
double* D = new double[nSize];
double* CD = new double[nSize];     // CD = C + D
double* ABCD = new double[nSize];   // ABCD = AB + CD
. . .
{
      qtControl   qtControl;    // task team control object
      // enqueue first parallel_for without blocking
      parallel_for(&qtControl, &DoSum 0, nSize, A, B, AB);
      // enqueue second parallel_for without blocking
      parallel_for(&qtControl, &DoSum 0, nSize, C, D, CD);
      // do other work while both parallel for loops run
      DoOtherWork();            // runs concurrent with AB and CD
      // now wait for completion of both parallel for loops
      qtControl.WaitTillDone(); // or parallel_wait(&qtControl);
      // begin next parallel for (note use of qtControl)
      parallel_for(&qtControl, &DoSum, 0, nSize, AB, CD, ABCD);
      // something else to do while performing parallel_for
} // qtControl dtor performs implicit qtControl.WaitTillDone();
```

In the above example, the DoSum of AB is scheduled by the current thread. The current thread is a potential team member for the AB parallel loop but does not begin processing a portion of the AB sum. Instead, the thread continues and performs the enqueuing of the second parallel_for for the CD parallel loop but does not begin processing a portion of the CD sum. Instead, the thread continues and performs DoWork(). Upon return from DoWork() the current thread issues the `qtControl.WaitTillDone();` it will now be able to partake in the parallel_for loops requested for AB and CD. However, should any of the other threads performing the AB DoSum finish prior to the current thread issuing the `qtControl.WaitTillDone();` then that thread will process the current threads portion of the AB loop. And a similar situation with the CD loop.

Now consider the situation where you wish to control data locality and take advantage of cache placement. Assume in the above example that arrays A, B, AB are preferred to be run in the L2 cache of the main thread and that C, D, CD are preferred to run in a different L2 cache from the main thread, and ABCD has no preference to cache locality.

```
{
      qtControl   ABqtControl;
      // non-blocking parallel_for (CD = C + D)
      // *** perform this in an L2 cache other than mine
      parallel_for(
            L2$+ExcluedMyCacheLevel$,
            &ABqtControl,
            &DoSum 0, nSize, C, D, CD);
      // blocking parallel_for (AB = A + B)
      // restrict to threads sharing my L2 cache
      parallel_for(
            L2$,
            &DoSum 0, nSize, A, B, AB);
      // now wait for (CD = C + D)
      ABqtControl.WaitTillDone();
      // blocking parallel_for (ABCD = AB + CD)
      // using all threads
```

```
        parallel_for( &DoSum, 0, nSize, AB, CD, ABCD);
        // something else to do while performing parallel_for
    }
```

Distribution by thread affinity, thread availability, and flow-thru is not possible using TBB.

There are other ways to perform the scheduling which can be used to enhance the performance of your application. These techniques are covered in greater detail in the programmer's manual.

Grainsize

Both TBB and QuickThread have optional IdealGrainSize argument to their respective parallel_for statements. The TBB grain size is a cutoff point (below which parallelization does not occur). The QuickThread grain size is similar to the OpenMP Chunk size (and specifies a chunking factor).

Lambda functions

Both TBB and QuickThread (with C++0x compilers) support Lambda functions in parallel_for.

Bandwidth

Intel TBB, in some instances where the applied function Foo is relatively simple TBB may experience some bandwidth problems.

QuickThread, in a sub-set of these bandwidth problems uses of QuickThread's affinity scheduling may extract additional performance from the system.

Auto_partitioner

Intel TBB provides an auto_partitioner for use with blocked ranges.

QuickThread does not use an auto_partitioner to determine optimal grain sizes.

In place of auto_partitioner, QuickThread programmers can choose…

Dynamic partitioning

An example of dynamic partitioning is the ability to distribute a parallel_for across current thread plus any idle threads:

```
    parallel_for(Waiting$, ParallelApplyFoo, 0, n, a);
```

Or to a subset such as sharing L3 cache (same socket)

```
    parallel_for(Waiting_L3$, ParallelApplyFoo, 0, n, a);
```

The Waiting$ flag reduces the number of partitions (tasks) of the iteration space to that of number of waiting threads (in subset if specified) plus current thread (assuming no ExcludeMyCacheLevel$).

Additionally, the QuickThread **parallel_for_each**, **parallel_reduce** and **parallel_list** statements perform partitioning based upon the availability of idle threads (as well as optional placement capabilities).

```
        parallel_for_each(n, ParallelApplyFooEach, 0, a);
```

At start, the range 0:n-1 is partitioned into 1+ number of idle threads. Then as execution progresses the executing threads will monitor for additional waiting threads becoming available. When a thread becomes available, another partitioning occurs (i.e. dynamically).

The use of auto_partitioner in QuickThread may be of questionable value considering the enhanced scheduling capabilities, and in particular dynamic partitioning of QuickThread.

parallel_reduce

Serial summation function

```
        float SerialSumFoo( float a[], size_t n )
        {
              float sum = 0;
              for( size_t i=0; i!=n; ++i )
                    sum += Foo(a[i]);
              return sum;
        }
```

Intel TBB parallel_reduce

```
        // TBB requires you to create class to hold operator()
        class SumFoo
        {
              float* my_a;
              public:
              float sum;
              void operator()( const blocked_range<size_t>& r )
              {
                    float *a = my_a;
                    for( size_t i=r.begin(); i!=r.end(); ++i )
                          sum += Foo(a[i]);
              }
              SumFoo( SumFoo& x, split ) : my_a(x.my_a), sum(0) {}
              void join( const SumFoo& y ) {sum+=y.sum;}
              SumFoo(float a[] ) :
              my_a(a), sum(0)
              {}
        };
```

```
	float ParallelSumFoo( const float a[], size_t n )
	{
		SumFoo sf(a);
		parallel_reduce(
			blocked_range<size_t>(0,n,IdealGrainSize),
			sf );
		return sf.sum;
	}
```

QuickThread parallel_reduce

```
	parallel_reduce( fn, iBegin, iEnd, reduction [, optional args]);
```

This statement is similar to parallel_for in syntax excepting that it has one additional required argument following the iteration space variables. (optional qtPlacement and qtControl objects may preceed the arguments when applicable).

Unlike TBB where the reduction code is bound to the application data object as member function, QuickThread uses reduction objects that are not bound to the application data object. This permits higher reusibility of code.

The QuickThread reduction object can be simple or complex. The simple reduction objects can get reused.

For an example, consider a reduction object to produce a summation. The functionality of a reduction object is intuitively simple, it must:

a) initialize to required state (usually 0)
b) accept a next item
c) perform the reduction with other reduction object

An example of a simple reduction object would be for summation of an scalar type. The template function is relatively simple:

```
// Value reduction object
template<typename T>
struct ReduceSum
{
	T value;
	inline ReduceSum<T>() { value = 0; }
	inline void Reduce(T x) { value += x; }
	inline void Reduce(ReduceSum& o) { value += o.value; }
};
. . .
template<typename T>
void ParallelSum(
	long iBegin, long iEnd, // Half open range required first
	ReduceSum<T>& ret,      // reduction object next
	const T a[])            // optional arguments last
{
	for( long i=iBegin; i<iEnd; ++i )
		ret.Reduce(a[i]); // use reduction by values
}
```

```
. . .
ReduceSum<float> reduceSum;
parallel_reduce(
      ParallelSum<float>, 0, NumberOfFloats, reduceSum, Array);
```

With TBB the Array would have to be an encapsulation object containing Array or pointer/reference to Array. With QuickThread there is no encapslulation thus no requirement to alter code related to Array.

Also note that the reduction operators are performed without an AtomicAdd. The reason we can do this is the final reduction is performed after all threads have completed and the thread issuing the parallel_reduce performs the reduction on its return to the thread issueing parallel_reduce. This eliminates the relatively expensive _Interlocked… instruction used to complete the reduction.

Although QuickThread has a **parallel_reduce**, many reduction operations can be performed using thread safe programming practices with traditional functions (run as parallel tasks).

```
      // your existing serial function to produce sum
      float SerialSumFoo( float a[], size_t n )
      {
            float sum = 0;
            for( size_t i=0; i!=n; ++i )
                  sum += Foo(a[i]);
            return sum;
      }

      // add parallel slice function
      // to re-use serial function to produce sum
      void ParallelSumFoo(
            long iBegin, long iEnd, // Half open range required first
            const float a[],
            float*      out)
      {
            float sum = SerialSumFoo(&a[iBegin], iEnd- iBegin);
            AtomicAdd(out, sum); // thread safe atomic add of floats
                                 // in QuickThread.h
      }
      . . .
      float mySum = 0.0f;
      parallel_for(
            ParallelSumFoo, 0, NumberOfFloats, Array, &mySum);
```

Again, QuickThread provides for more re-usability of existing code.

In cases such as summation into float or double it would be more effective to pass the pointer to the eventual sum variable and perform the AtomicAdd function (supplied with QuickThread). Or if you prefer you could use a TBB atomic<float> object or use #pragma omp atomic statement.

Advanced Example

In this example the reduction object is compound. It contains a value and an index. The code is to find the index of the minimum value within an array.

Serial

```
long SerialMinIndexFoo(
	const float a[], long n )
{
	float value_of_min = FLT_MAX; // FLT_MAX from <float.h>
	long index_of_min = -1;
	for( long i=0; i<n; ++i )
	{
		float value = Foo(a[i]);
		if( value<value_of_min )
		{
			value_of_min = value;
			index_of_min = i;
		}
	}
	return index_of_min;
}
```

Intel TBB

```
class MinIndexFoo
{
	const float *const my_a;
public:
	float value_of_min;
	long index_of_min;
	void operator()( const blocked_range<size_t>& r )
	{
		const float *a = my_a;
		for( size_t i=r.begin(); i!=r.end(); ++i )
		{
			float value = Foo(a[i]);
			if( value<value_of_min )
			{
				value_of_min = value;
				index_of_min = i;
			}
		}
	}
	MinIndexFoo( MinIndexFoo& x, split ) :
	my_a(x.my_a),
	value_of_min(FLT_MAX), // FLT_MAX from <climits>
	index_of_min(-1)
	{}
	void join( const SumFoo& y )
	{
		if( y.value_of_min<x.value_of_min )
		{
			value_of_min = y.value_of_min;
			index_of_min = y.index_of_min;
		}
	}
	MinIndexFoo( const float a[] ) :
      my_a(a),
      value_of_min(FLT_MAX), // FLT_MAX from <climits>
      index_of_min(-1),
      {}
};

long ParallelMinIndexFoo( float a[], size_t n )
{
      MinIndexFoo mif(a);
      parallel_reduce(blocked_range<size_t>(0,n,IdealGrainSize), mif );
      return mif.index_of_min;
}
```

QuickThread

```
// Reduction object
struct ReduceMinIndexFoo
{
      float value_of_min; // your data here
      long index_of_min;
      ReduceMinIndexFoo()
      {     // default initialization
            value_of_min = FLT_MAX; // FLT_MAX from <float.h>
            index_of_min = -1;
      }
      void Reduce(float value, long index)
      {     // Reduction by values
            if( value<value_of_min )
            {
                  value_of_min = value;
                  index_of_min = index;
            }
      }
      void Reduce(ReduceMinIndexFoo& o)
      {     // Reduction by reference
            if( o.value_of_min<value_of_min )
            {
                  value_of_min = o.value_of_min;
                  index_of_min = o.index_of_min;
            }
      }
};
. . .
void fnParallelMinIndexFoo(    // Task function
      long iBegin, long iEnd, // Half open range required first
      ReduceMinIndexFoo& ret, // reduction object next
      const float a[])        // optional arguments last
{
      for( long i=iBegin; i<iEnd; ++i )
            ret.Reduce(Foo(a[i]), i); // use reduction by values
}
. . .

long ParallelMinIndexFoo(
      const float a[], size_t n)
{
      ReduceMinIndexFoo reduce;
      parallel_reduce(fnParallelMinIndexFoo, 0, n, reduce, a);
      return reduce.value;
}
```

Advanced Topic: Other Kinds of Iteration Spaces

Intel TBB permits you to produce and use custom written iteration objects (iterators) for use in your program.

QuickThread permits you to produce and use custom written iteration objects (iterators) for use in your program as well. However, due to the enhanced capability of QuickThread to permit you to specify placement and availability as well as dynamic partitioning you may find few uses for an interator object. Running a linked list might be one of the few cases where an iterator object might be advised. However, there are easier programming techniques for parallel processing of linked lists.

parallel_while

Serial

```
	void SerialApplyFooToList( Item*root )
	{
		for( Item* ptr=root; ptr!=NULL; ptr=ptr->next )
			Foo(ptr->data);
	}
```

Intel TBB parallel_while

```
class ItemStream
{
	Item* my_ptr;
	public:
	bool pop_if_present( Item*& item )
	{
		if( my_ptr )
		{
			item = my_ptr;
			my_ptr = my_ptr->next;
			return true;
		}
		else
		{
			return false;
		}
	};
	ItemStream( Item* root ) : my_ptr(root) {}
};

void ParallelApplyFooToList( Item*root )
{
	parallel_while<ApplyFoo> w;
	ItemStream stream;
	ApplyFoo body;
	w.run( stream, body );
}
```

QuickThread parallel_list

QuickThread uses the template name parallel_list to process linked lists.

Process a null terminated list using all threads:

```
      void ParallelApplyFooToList( Item* root )
      {
            parallel_list(Foo, root);
      }
```

Or simply make call inline in your code

```
      parallel_list(Foo, root);
```

The internal workings of parallel_list are (subject to optional qtPlacement) distribute the list to the current thread plus current number of idle threads. Then as each thread processes nodes, monitor for additional threads to become idle (subject to optional qtPlacement) and add them to the team of threads working on the list. The linked list is dynamically partitioned to available threads.

parallel_pipeline

Both TBB and QuickThread have parallel_pipeline
QuickThread has two classes of threads, TBB does not. QuickThread can take advantage of the two classes of threads in the parallel_pipeline by designating pipes as I/O class or compute class. Typically, the input end and output end of a pipeline are I/O. The dual class capability of QuickThread gives it a distinct advantage over TBB pipeline.

The following is a simple example of up casing the first letter of each word in a file.

Serial

```
// Buffer that holds block of characters
// and last character of previous buffer.
class MyBuffer
{
      static const long buffer_size = 10000;
      char* my_end;
      // storage[0] holds the last character of the previous buffer.
      char storage[1+buffer_size];
public:
      // Pointer to first character in the buffer
      char* begin() {return storage+1;}
      const char* begin() const {return storage+1;}
      // Pointer to one past last character in the buffer
      char* end() const {return my_end;}
      // Set end of buffer.
      void set_end( char* new_ptr ) {my_end=new_ptr;}
```

```
	// Number of bytes a buffer can hold
	long max_size() const {return buffer_size;}
	// Number of bytes in buffer.
	long size() const {return (long)(my_end-begin());}
};

class MyIoContext
{
public:
	FILE* input_file;
	FILE* output_file;
	char  last_char_of_previous_buffer;
	MyIoContext()
	{
		input_file = NULL;
		output_file = NULL;
	}
	bool openInput(const char* fileName)
	{
		last_char_of_previous_buffer = ' ';
		input_file = fopen(fileName, "rt");
		if(input_file) return true;
		return false;
	}
	bool closeInput()
	{
		if(input_file)
		{
			if(fclose(input_file))
			{
				input_file = NULL;
				return false;
			}
			input_file = NULL;
		}
		return true;
	}
	bool openOutput(const char* fileName)
	{
		output_file = fopen(fileName, "wt");
		if(output_file) return true;
		return false;
	}
	bool closeOutput()
	{
		if(output_file)
		{
			if(fclose(output_file))
			{
				output_file = NULL;
				return false;
			}
			output_file = NULL;
		}
		return true;
```

```
        }
};

MyBuffer    b;
MyIoContext io;
const int NumberOfLines = 100000;

void BuildFile()
{
        io.openOutput("QuickBrownFox.txt");
        int ret;
        for(int i=0; i<NumberOfLines; ++i)
        {
                ret = fprintf(io.output_file,
"the quick brown fox jumped over the lazy grey dog's back. %d\n", i);
                if(ret == EOF)
                        break;
        }
        io.closeOutput();
}

void ReadyFiles()
{
        io.openInput("QuickBrownFox.txt");
        io.openOutput("QuickBrownFoxUpcase.txt");
}

void  CloseFiles()
{
        io.closeInput();
        io.closeOutput();
}

void  UpcaseWords()
{
        // files already open
        char last_char_of_previous_buffer = ' ';
        for(;;)
        {
                size_t n = fread(
                        b.begin(), 1, b.max_size(), io.input_file );
                if( !n )
                        break; // end of file

                // insert last char of previous buffer
                b.begin()[-1] = last_char_of_previous_buffer;
                // remember for next time
                last_char_of_previous_buffer = b.begin()[n-1];
                // count read may be shorter than buffer
                b.set_end( b.begin()+n );

                // preamble to loop
                bool prev_char_is_space = (isspace(b.begin()[-1])!=0);
                // word upcase loop
                for( char* s=b.begin(); s!=b.end(); ++s )
```

```
            {
                  if( prev_char_is_space && islower(*s) )
                        *s = toupper(*s);
                  prev_char_is_space = (isspace(*s)!=0);
            }

            // output to file
            fwrite( b.begin(), 1, b.size(), io.output_file );
      }
}

void  SerialRunUpcaseWordsTest()
{
      BuildFile();
      ReadyFiles();
      UpcaseWords();
      CloseFiles();
}
```

TBB parallel_pipeline

```
// Filter that writes each buffer to a file.
class MyOutputFilter: public tbb::filter
{
      FILE* my_output_file;
      public:
      MyOutputFilter( FILE* output_file );
      /*override*/void* operator()( void* item );
};

MyOutputFilter::MyOutputFilter( FILE* output_file ) :
tbb::filter(/*is_serial=*/true),
my_output_file(output_file)
{
}

void* MyOutputFilter::operator()( void* item )
{
      MyBuffer& b = *static_cast<MyBuffer*>(item);
      fwrite( b.begin(), 1, b.size(), my_output_file );
      return NULL;
}

// Filter that changes the first letter of each word
// from lower case to upper case.
class MyTransformFilter: public tbb::filter
{
public:
      MyTransformFilter();
      /*override*/void* operator()( void* item );
};

MyTransformFilter::MyTransformFilter() :
```

```
tbb::filter(/*serial=*/false)
{}

/*override*/void* MyTransformFilter::operator()( void* item )
{
      MyBuffer& b = *static_cast<MyBuffer*>(item);
      bool prev_char_is_space = b.begin()[-1]==' ';
      for( char* s=b.begin(); s!=b.end(); ++s )
      {
            if( prev_char_is_space && islower(*s) )
                  *s = toupper(*s);
            prev_char_is_space = isspace(*s);
      }
      return &b;
}

class MyInputFilter: public tbb::filter
{
public:
      static const size_t n_buffer = 4;
      MyInputFilter( FILE* input_file_ );
      private:
      FILE* input_file;
      size_t next_buffer;
      char last_char_of_previous_buffer;
      MyBuffer buffer[n_buffer];
      /*override*/ void* operator()(void*);
};

MyInputFilter::MyInputFilter( FILE* input_file_ ) :
filter(/*is_serial=*/true),
next_buffer(0),
input_file(input_file_),
last_char_of_previous_buffer(' ')
{
}

void* MyInputFilter::operator()(void*)
{
      MyBuffer& b = buffer[next_buffer];
      next_buffer = (next_buffer+1) % n_buffer;
      size_t n = fread( b.begin(), 1, b.max_size(), input_file );
      if( !n )
      {
            // end of file
            return NULL;
      }
      else
      {
            b.begin()[-1] = last_char_of_previous_buffer;
            last_char_of_previous_buffer = b.begin()[n-1];
            b.set_end( b.begin()+n );
            return &b;
      }
}
```

```
// Create the pipeline
tbb::pipeline pipeline;

// Create file-reading writing stage
MyInputFilter input_filter( input_file );
//  and add it to the pipeline
pipeline.add_filter( input_filter );

// Create capitalization stage
MyTransformFilter transform_filter;
//  and add it to the pipeline
pipeline.add_filter( transform_filter );

// Create file-writing stage
MyOutputFilter output_filter( output_file );
//  and add it to the pipeline
pipeline.add_filter( output_filter );

// Run the pipeline
pipeline.run( MyInputFilter::n_buffer );

// Remove filters from pipeline before they are implicitly destroyed.
pipeline.clear();
```

QuickThread parallel_pipeline

QuickThread provides multiple types of pipelines. The programmer can select from:

a) Closed ends self running ring buffer (as used by this up-case example)
b) Open ends (input end waits for data, output end consumes buffer)
c) Half open end (one end open, one end closed)

The QuickThread pipeline also has the capability of flow control (covered in the programmer's guide).

Similar to TBB, the QuickThread pipeline pipes are passed a token consisting of a buffer (a buffer class of your design) together with a QuickThread pipeline header. To create your pipeline buffer token you may append your serial buffer to the QuickThread header.

```
class MyPipelineBuffer : public PipelineBuffer, public MyBuffer
{
};
```

There are two classes of pipes in the QuickThread pipeline: Compute and I/O. The compute class of pipe receives the buffer token (in this case your `MyPipelineBuffer` token). The I/O class of pipe is passed an I/O context of your design together with QuickThread header information. To create your pipeline I/O context you may append your serial I/O context to the QuickThread I/O context header.

```
class MyPipelineIoContext
```

```
: public PipelineIoContext, public MyIoContext
{
};
```

The input side of a QuickThread pipeline is typically an I/O class of pipe receiving the I/O context (after successful file open), plus a buffer token. With the exception of how to report the end of file (or read error), the task for reading into the buffer is ostensibly the same as your serial function. The principal difference being inserting the failure notification into the QuickThread header pre-pended to your buffer object.

```
// task for reading buffer
void PipelineReadBuffer(MyPipelineIoContext* io, MyPipelineBuffer* b)
{
      // read buffer
      size_t n = fread( b->begin(), 1, b->max_size(), io->input_file );
      if( n )
      {
            // have some data
            // Inform buff at to the sequence number
            // advance sequence number for next time
            // insert last char of previous buffer
            b->begin()[-1] = io->last_char_of_previous_buffer;
            // remember last char of this buffer for next time
            io->last_char_of_previous_buffer = b->begin()[n-1];
            // count read may be shorter than buffer
            b->set_end( b->begin()+n );
      }
      else
      {
            // EOF or read error
            // Set exit status to Fail$
            // Fail$ does not shutdown pipeline
            b->Status = Fail$;
      }
}
```

The compute class of QuickThread pipeline pipe receives just the buffer token. Typically compute class of pipes do not error out. If they do report an error, use `b->Status = Fail$;` to report the error. (There are additional status values available.)

```
// task to process buffer
void PipelineProcessBuffer(MyPipelineBuffer* b)
{
      bool prev_char_is_space = (isspace(b->begin()[-1])!=0);
      for( char* s=b->begin(); s!=b->end(); ++s )
      {
            if( prev_char_is_space && islower(*s) )
                  *s = toupper(*s);
            prev_char_is_space = (isspace(*s)!=0);
      }
}
```

The back end of the pipeline is typically an I/O class of pipe. The pipe task will receive the I/O context plus the buffer.

```
// task to write buffer
void PipeLineWriteBuffer(MyPipelineIoContext* io, MyPipelineBuffer* b)
{
      if(b->size() == 0) return; // ? nothing to write
      size_t itemsWritten = fwrite(
            b->begin(), 1, b->size(), io->output_file );
      if(itemsWritten != b->size())
            b->Status = ExitFail$;  // ExitFail$ - shutdown pipeline
}
```

Note the difference in error value on write being `ExitFail$` as opposed to the read pipe error `Fail$.` The distinction being that Fail$ does not shut down the pipeline, it simply removes the buffer token from the pipeline. Whereas ExitFail$ is considered terminal and it shuts down the pipeline.

Now that we have all the pieces of the pipeline we can put them together

```
void  ParallelRunUpcaseWordsTest()
{
      MyPipelineIoContext     pio;
      pio.openInput(  "QuickBrownFox.txt");
      pio.openOutput( "QuickBrownFoxUpcaseParallel.txt");

      qtPipeline<MyPipelineIoContext, MyPipelineBuffer>     pipeline;
      pipeline.addPipe( PipelineReadBuffer );
      pipeline.addPipe( PipelineProcessBuffer );
      pipeline.addPipe( PipeLineWriteBuffer );
      pipeline.run( &pio );
      // (run other code here if desired prior to waiting)
      pipeline.WaitTillDone();

      pio.closeInput();
      pio.closeOutput();
}
```

Note, the code to open and close the input and output files could have called your serial functions with the slight modification of passing in the pio object The above pipeline is the default ring buffer. The pipeline.run(&pio); is supplied the I/O context and will allocate a default number of buffers (tokens) based on the numbers of compute and I/O class thread counts. You can override the default number of buffers using:

```
      pipeline.initBuffers(numberOfBuffers);
```

When the pipeline terminates you can examine the completion status of the pipeline to determine if an error occurred.

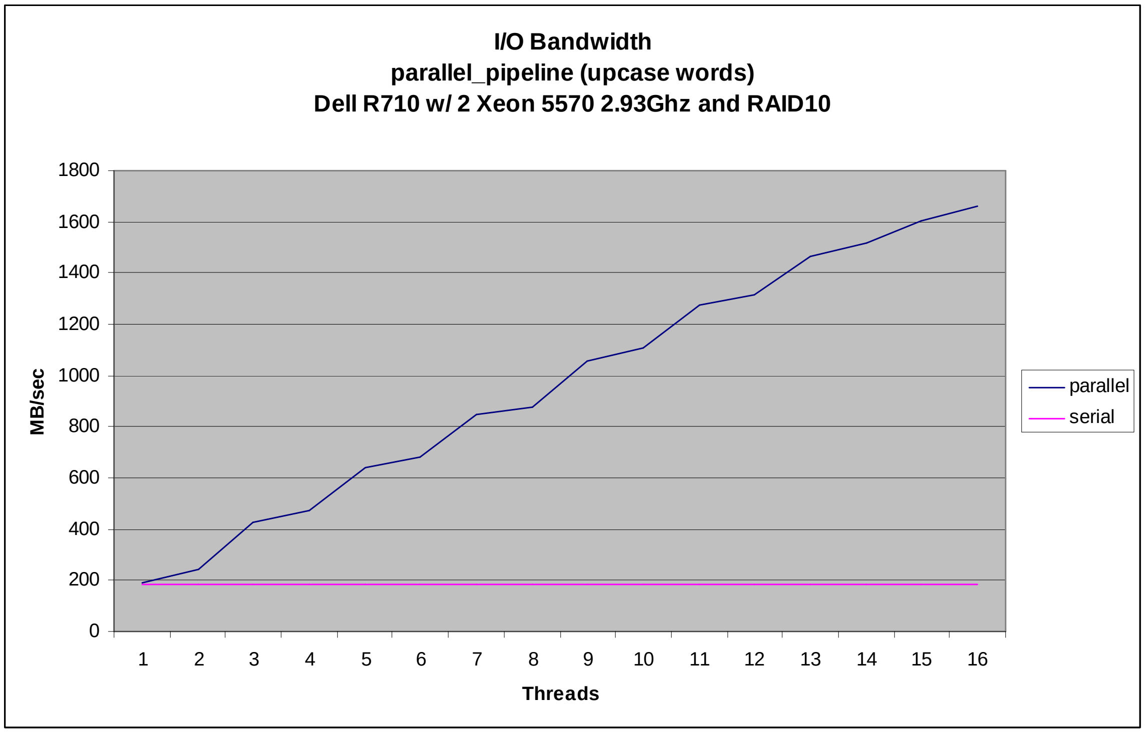
For QuickThread, when the first pipe is an I/O class of pipe it is assumed that I/O is sequential. That task will run only one instance of that task during the processing of the token (e.g. file read into buffer token). These buffer tokens will receive a sequence number. The output side of the pipeline, when an I/O class of pipe, will collate the buffers such that they are written in sequence. All interior pipes will run in parallel regardless as to if they are compute class or i/o class of pipe.

Throughput of pipeline

The same issues exist for both QT and TBB relating to throughput.

However, QuickThread has an advantage over TBB because QuickThread has two classes of threads Compute Class and I/O class. I/O typically stalls a thread (while waiting for read or write to complete). On TBB this will cause a loss of computational thread for duration of stall. With QuickThread, all computational threads in the pipeline never stall.

The following is a chart of QuickThread running a parallel_pipeline to up case words run on a dual processor Dell R710. Each processor was a 4-core with HT Xeon 5570 2.93GHz (total of 8 cores and 16 hardware threads available). Disk controller was a RAID10. File was approximately 659 MB in size. The bandwidth below was the sum of the read and write MB/sec.



Non-Linear Pipelines

TBB does not support non-linear pipelines.

QuickThread supports non-linear pipelines.

QuickThread pipelines have flow control capability. Pipes declared with return type of qtPipelineReturn have the capability of affecting flow through the pipeline in other manners than as what is available to the pipes returning void using a pipeline status of Fail$ or ExitFail$. Using a pipe that returns a status value together with branch control statement you can construct non-linear pipelines.

```
enum qtPipelineReturn
{
      ExitSuccess$ = 2,
      True$ = 1,
      Success$ = True$,
      Continue$ = 0,
      False$ = -1,
      Fail$ = False$,
      ExitFail$ = -2,
};
```

For pipes with return of `qtPipelineReturn` the return code, when not `ExitSuccess$` or `ExitFail$`, can be used to qualify the next pipe in the pipeline.

QuickThread pipelines have conditional execution of pipe as well as branch control

```
enum qtPipelineBranch
{
      IfTrue$,
      IfFalse$,
      Goto$,
      ReturnSuccess$,
      ReturnFail$,
};

...
pipeline.addPipe(            PipelineProcessBuffer );
pipeline.addPipe( IfFalse$,  PipelineProcessFailed );
pipeline.addPipe(            PipelineMoreProcessBuffer );
```

Or with flow control

```
const qtPipelineTag FoundIt$ = 999; // your arbitrary number
const qtPipelineTag MergeIt$ = 1234;// your arbitrary number

...
pipeline.addPipe(            PipelineReadBuffer );
pipeline.addPipe(            PipelineProcessBuffer );
pipeline.addPipe( IfTrue$,   FoundIt$ );
pipeline.addPipe(            PipelineMoreProcessBuffer );
pipeline.addPipe(            PipelineMoreTooProcessBuffer );
pipeline.addPipe( Goto$,     MergeIt$ );

pipeline.addPipe( FoundIt$,  PipelineFoundItProcessBuffer );
pipeline.addPipe(            PipelineMoreFoundItProcessBuffer );
pipeline.addPipe( MergeIt$,  PipelineMergeProcessBuffer );
```

. . .

The QuickThread pipelines are simplified state machines. When you choose to create a pipeline tag you may choose any arbitrary number as long as it has not been used (similar to tag number Fortran).

Additionally a pipeline pipe can acquire an additional buffer(s) for splitting or consume buffers for joining.

Summary of Loops

The high-level loop templates of TBB requires the creation of new class objects and offers minimal flexibility when running on larger systems.

QuickThread high-level templates make it easy to reuse much of your serial code as parallel tasks without requiring you to write new class objects. QuickThread provides for complete control over task scheduling (affinity or cache associated scheduling) as well as opportunistic scheduling (scheduling dependent upon the availability of threads).

QuickThread also has the capability of programming the loops in a non-blocking manner as well as the ability to program the loops with completion routines. That is: for non-blocking loops you have an option to specify that upon completion of the loop a particular task or series of tasks are to be scheduled.

Containers

QuickThread does not include concurrent containers. If you require concurrent containers we recommend that you use concurrent containers that are available as Open Source and are readily available on the internet. However, QuickThread has available concurrent_proxy_vector (see below).

concurrent_hash_map

QuickThread does not include concurrent_hash_map. We recommend that you use concurrent containers that are available as Open Source and are readily available on the internet.

concurrent_vector

QuickThread does not include concurrent_vector. We recommend that you use concurrent containers that are available as OpenSource and are readily available on the internet. However, QuickThread has available concurrent_proxy_vector (see below).

concurrent_proxy_vector

QuickThread provides a variation on the concurrent_vector called the concurrent_proxy_vector. (This name may change)

The concurrent_proxy_vector provides the same functionality as concurrent_vector (plus some additional functionality) however the internal workings of the concurrent_proxy_vector are quite different resulting in less use of locks. Meaning faster access, less interference, and no possibility of incomplete data.

The concurrent_proxy_vector is similar to a vector of pointers (proxies) to objects as opposed to a vector of the objects. The principal advantages are the pointer (proxy) is completely contained within a cache line and can be manipulated using Interlocked… instructions.

An article on the Intel software developer's blogs site:

http://software.intel.com/en-us/blogs/2009/04/09/delusion-of-tbbconcurrent_vectors-size-or-3-ways-to-traverse-in-parallel-correctly/

Describes the problems associated with the TBB concurrent_vector whereby the programmer must take into consideration the possibility of incomplete vectors. The TBB concurrent_vector can be incomplete, have holes in it and/or may have addressable areas in the process of being allocated.

The QuickThread concurrent_proxy_vector does not suffer these problems. Objects are allocated and constructed *before* an insertion attempt is made to the concurrent_proxy_vector. The concurrent_proxy_vector never contains objects under construction. The container for the proxies is contiguous, however, when a larger container is required, the current container can continue to be used by stale pointers. Thus there is no locking when enlarging the container. When the prior container(s) is(are) known to not have references then they are returned for recycling.

Clearing is Not Concurrency Safe

TBB has the requirement of never using `clear()` on concurrent_vector while operations might be in flight.

QuickThread (will) provide capability for using `clear()` while operations are in flight.

concurrent_queue

QuickThread does not provide concurrent_queue. We recommend that you use concurrent queues that are available as OpenSource and are readily available on the internet.

When Not to Use Queues

Same as for TBB

Mutual Exclusion

Lock_FIFO and LockLock

One of the locks provided with QuickThread is implemented with a **Lock_FIFO** object and scoped lock object **LockLock**. The QuickThread **Lock_FIFO** object is a fair lock. Meaning locks will be granted in the order in which they are attempted (First In First Out).

```
Lock_FIFO resourceLock;
...
{
      LockLock lock(resourceLock);  // Block until lock granted
      . . .
} // dtor of lock releases lock.
```

qt_pointerLock

```
void* qt_pointerLock(void** p);
```

The qt_pointerLock is a low overhead but unfair locking mechanism . Meaning threads contending for the pointer remain compute bound (in _mm_pause() loop) waiting to obtain the pointer. The first thread to re-attempt a lock following the unlock of the pointer will be the next thread to obtain the lock. This lock is performed by exchanging the contents of the pointer with 1 (and returning the prior value of the pointer). When, internal to qt_pointerLock, the return value is 1 the function issues an _mm_pause() then re-attempts the lock. This loop continues until lock is attained. And in which case the prior value of the pointer is returned (including NULL when prior value is NULL). The return value can be saved and/or use however you want. The user code, which is not using the qt_pointerLock must be aware of pointer indicating locked condition (containing 1). The unlock is performed by replacing the pointer with either the old pointer, new pointer or NULL. The `qt_pointerLock` is templated for arbitrary types

```
YourType* p = qt_pointerLock(&YourNodePointer);
if(p)
{
   // your protected code here
}
YourNodePointer = p;
```

Atomic Operations

TBB offers an atomic class QuickThread does not we suggest you use the TBB atomic class or use other Open Source atomic class templates.

Because QuickThread is compatible with OpenMP you can also elect to use #pragma omp atomic, or use the _Interlocked… series of intrinsic functions or use the QuickThread Atomic functions such as:

```
float AtomicAdd(float* pf, float v);
```

```
      double AtomicAdd(double* pd, double d);
```

Newer versions of C++ have atomic class therefore it was not deemed necessary to provide this capability with QuickThread.

Timing

QuickThread does not provide timing routines since these are generally available. Also, most programmers are now using

```
        unsigned __int64 __rdtsc(void);
```

(or you can use the OpenMP omp_get_wtime() function).

Memory Allocation

TBB has two allocators tbb_allocator<T> and cache_aligned_allocator<T>
QuickThread has one allocator qt_allocator<T>

The single QuickThread allocator can be declared to perform allocations without alignment considerations or cache aligned allocations.

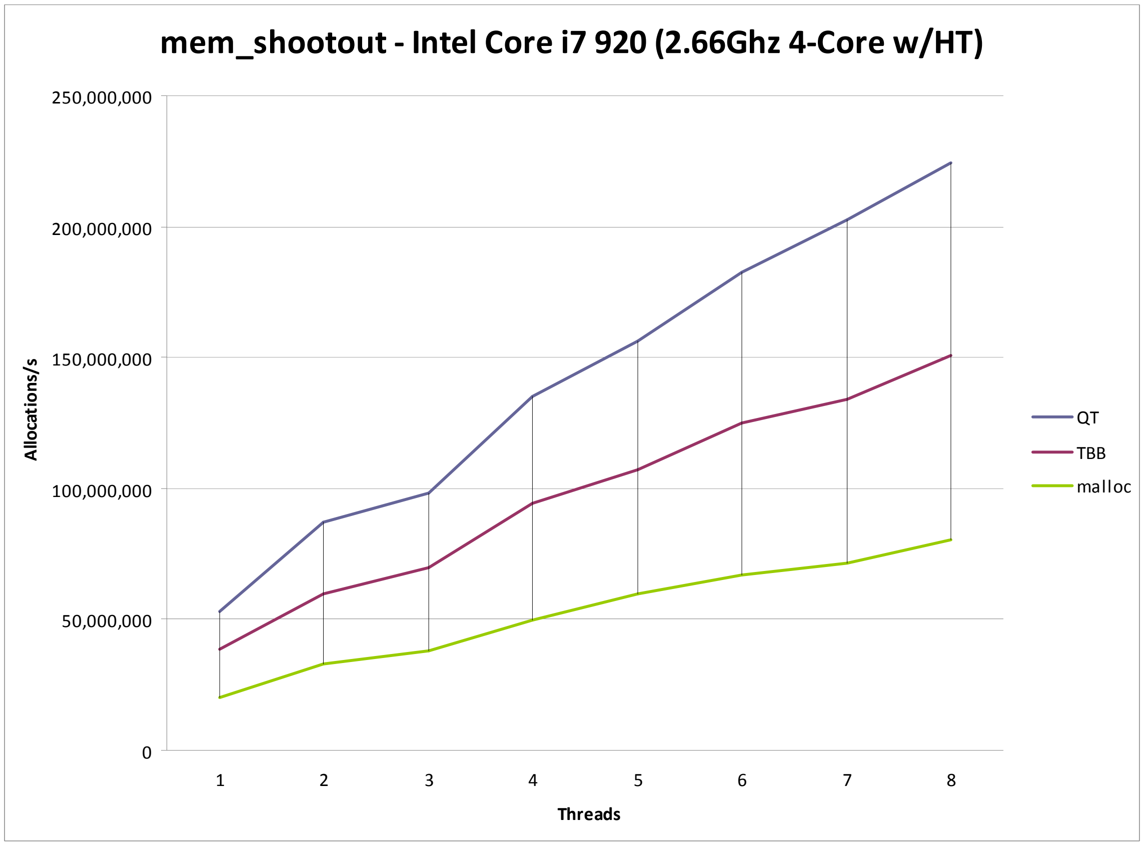
```
qt_allocator<Foo> FooAllocator;                // non-aligned allocator of Foo objects
qt_allocator<Foo> FooAlignedAllocator(64);     // cache aligned allocator of Foo objects
```

The QuickThread allocator uses a pool of pools concept with pool item granularity of sizeof(void*). The QuickThread allocator is NUMA enhanced for improved performance of memory access to object after allocation. The pool of pools is hierarchical and fast. Allocation is attempted in the following order

a) Thread's local pool of like sized prior allocations
b) Thread's NUMA node pool of pool of like sized prior allocations
c) Thread's NUMA node overflow list of pools of pools of like sized prior allocations
d) Thread's NUMA node overflow list of pools of like sized prior allocations
e) Thread's adjacent NUMA node(s) of b) c) d)
f) Thread's next hop NUMA node(s) of b) c) d)

Excepting for low memory conditions, when the thread that allocates an object is the same thread that deallocates the object the object will stay in the same NUMA pool of pools of like sized prior allocations. This is true too when the deallocating thread is pinned within the same NUMA node.

The following chart is produced from the average times through a comprehensive memory allocation/deallocation test suite mem_shootout written by Dmitriy V'jukov (you can find him on the Intel software forums and blogs site). There are a series of 4 tests with permutations resulting in 20 tests for each allocation system. The tests were run on a Core i7-920 independent of the TBB and QT thread schedulers. Allocations vary in strategy, size and order (small data set, large data set, similar sized, dissimilar sized, FIFO, LIFO, in order, out of order, random order, etc…).



Which Dynamic Libraries to Use

QuickThread is a relatively small collection of object files (library) that is linked statically into your application.

The Task Scheduler

Both QuickThread and TBB are task base threading systems.

Task-Based Programming

While both QuickThread and TBB are task-base threading systems the internal workings and programmer interface between the two task-based systems are quite different.

A QuickThread a task is any arbitrary function returning void. This function can be your original serial code function or a slightly modified version of your serial function (including thread safe programming practices and/or range of array as opposed to all of array).

TBB requires the use of adding a class object and using this as a wrapper to manage the tasks.

QuickThread offers a large degree of functionality on how to best schedule your tasks based on locality of data (in cache, in which cache) or lack there of, as well as availability of threads to perform work (opportunistic scheduling). QuickThread is NUMA aware and you can schedule based on NUMA placement.

While TBB claims benefit in scheduling where data is “hot in cache” TBB has no provisioning for which cache the data is “hot in”. On a single processor, dual core with shared L2 cache this may be satisfactory but on multi-processor or many-core system with multiple caches the TBB scheduler becomes more of a “hit or miss” situation with respect to the date being “hot in cache”.

With QuickThread you can specifically direct your task to a selected cache, cache level, or memory sub-system.

QuickThread also provides for the capability of synchronous tasks as well as completion tasks.

When Task-Based Programming is Inappropriate

TBB recommends that if your application has threads (code) that blocks frequently that you consider writing these tasks using separate threading system (e.g. separate Windows thread or pthread).

QuickThread has no Inappropriate Task-Based Programming

With QuickThread, the programmer has available two separate classes of threads: Compute Class and I/O class. Both classes of threads use the same QuickThread programming techniques and have the same functionality. The only distinction is you simply add the QuickThread placement directive IO$ or IOOnDone$ on the task function.

```
      parallel_task(IO$, fn, arg1, arg2);
```

or

```
      parallel_for(IO$, fn, iBegin, iEnd, arg1, arg2);
```

Upon initialization you may want to specify additional I/O class threads

```
      qtInit init(-1, 2); // all hw threads for compute, +2 I/O threads
```

Conclusions

The choice of threading model will greatly depend on the demands of the application and the amount of programming effort the programmer (or management) is willing to take to reach the level of performance desired. QuickThread is the only tool that provides forward-looking affinity based and NUMA capabilities into its threading syntax and does so in a highly efficient manner. With QuickThread you can write optimal multi-threaded applications for use on systems that span the full spectrum of IA and IA compatible platforms. Ranging from a single Atom processor with HT up through a multi-socket NUMA class system.

QuickThread Programming, LLC
85 Cove Lane
Oshkosh, WI 54902
USA
www.quickthreadprogramming.com