QuickThread

QuickThread is a pending trademark of

QuickThread Programming, LLC
James G. Dempsey
85 Cove Lane
Oshkosh, WI 54902
USA

Additional information can be obtained by email at info@quickthreadprogramming.com or at the website http://www.quickthreadprogramming.com.

QuickThread

QuickThread is a runtime library and programming paradigm for writing multithreaded applications in 32-bit and 64-bit environments using C++, Fortran and mixed language programs.

QuickThread is affinity capable supporting thread affinity, data binding affinity and NUMA support.

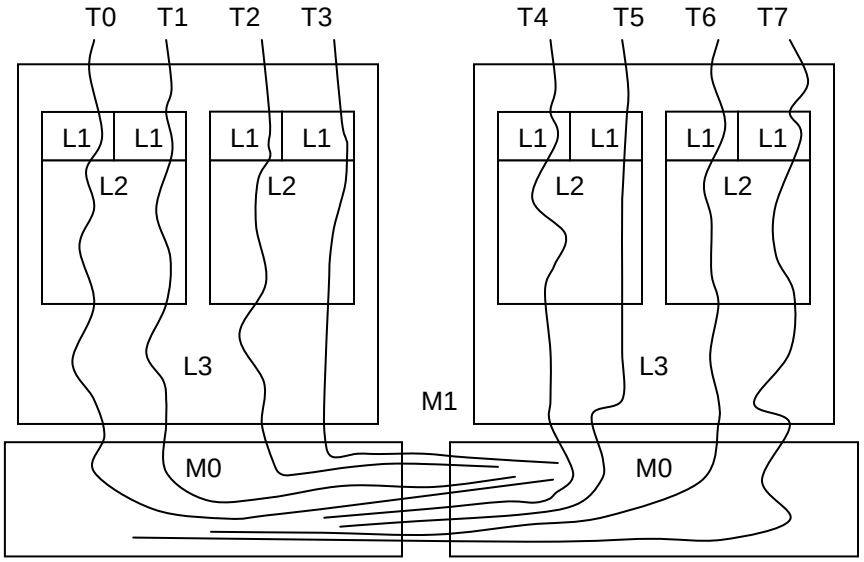
QuickThread is a tasking system using thread pools. Providing exceptional control over task scheduling with respect to cache levels, core placement, and thread availability.

The design goal of QuickThread is to produce a minimal overhead mechanism for distributing work in a multi-socket, multi-core, multi-threaded environment.

Parallel constructs

parallel_task	Schedule a single task.
parallel_invoke	Invoke multiple different tasks (C++0x Lambda functions only)
parallel_distribute	Schedule a task team to work on different portions of same task..
parallel_for	Schedule a task team to run across an iteration space divided up evenly to team members (or chunked up to team members)..
parallel_for_each	Schedule a task team across an iteration space divided upon demand by each team member number.
parallel_reduce	Schedule a task team across an iteration space divided upon demand by each team member number while performing reduction operation.
parallel_list	Schedule a task team to process a singly linked list of objects.
parallel_pipeline	Schedule a task team to process a sequence of steps (pipes) contained within a vector (pipeline).

Conceptual programming technique



The above figure depicts an idealized system with eight threads (T0-T7), running on two processors, each processor with four cores, three level cache, two memory systems. Two core pairs within each processor sharing one of two L2 cache within the processor, all cores within each processor sharing a processor common L3 cache. And each processor with direct access to a local RAM (M0) and one hop access to RAM local to the other processor (M1 obversely). The above diagram can be expanded to include additional processor packages and memory systems as well as additional memory hop levels (M2, M3).

In the idealized system, each thread has independent data distributed amongst the various cache and memory levels and where the programming goal is to keep as many of the thread's data (and instruction) accesses as close to its L1 as possible. When the programmer has the means to control the execution of the application in a manner complementary to this idealized system, then the application will experience maximum performance.

In practice, the generally used threading tools do not provide the programmer with the means to control the program execution towards this idealized system. That is until now.

One of the techniques employed by most of the threading tools which provides a limited measure of this control, was the switch in programming practice from:

using a dedicated software thread per task

to

using a pool of threads (typically with one software thread per hardware thread)

Then using a task scheduler within the application that schedules tasks to available threads from the thread pool. This technique exchanged a costly operating system thread context switch with a comparatively low cost task context switch within the application.

Additionally, when using the thread pool tasking technique, the programmer can use thread affinity to pin the software thread to a specific hardware thread (or set of threads). Using thread pinning, and when the operating system interrupts an application thread, or context switches to

another application, or system task, then upon resumption of application thread, there is the benefit of a higher probability of some portion of the previously cached application data still being present in its cache system.

The remaining control technique for the programmer to approach the idealized system is the means to choreograph not only the task scheduling but also the task placement, interaction with other tasks, and data placement control. QuickThread offers this level of control.

QuickThread offers the programmer the means to:

- Allocate data objects from a particular NUMA node (e.g. with most available RAM or least estimated computation load).
- Direct execution of task or task slices for data objects allocated with placement to be restricted to, or have preference to, run on threads within the NUMA node of that data object.
- Hot-in-cache programming considerations to direct execution of task or task slices to be restricted to, or have preference to, run on threads sharing a specific cache level with the current thread (thread issuing the task en-queue).
- Not-in-cache programming considerations to direct execution of task or task slices to be restricted to, or have preference to, run on threads sharing a specific cache level on the processor with the most idle hardware threads at that cache level.
- Opportunistic-in-cache task scheduling whereby loops can be conditionally split-up into multiple task slices only when, and to the extent of, threads sharing a specific cache level with the current thread are available (else the loop is run as a single task or diminished number of tasks together with the current thread).
- Include (by direct call of task as function call by current thread) or exclude current thread in task slice-up.
- Slice-up and distribute a task to a primary thread slice, one each, per requested cache level.
- Slice-up primary thread slice into secondary thread slices within the cache level of the primary thread slice.
- Opportunistic, as threads become available scheduling to reduce unnecessary thread scheduling calls.

With QuickThread the programmer can exert extraordinary level of control by the inclusion of a single placement directive on the parallel_for and other parallel directives.

The conceptual programming technique for QuickThread is a messaging system whereby you to throw objects and arguments at functions (C++) or subroutines (Fortran). These throw requests are placed into a queue (one of several queues). The queued subroutines, when run, can throw (en-queue) additional subroutines and arguments or perform work or do both.

The general queuing technique is neither strictly LIFO nor FIFO. QuickThread will en-queue the work requests in a manner that defaults to be hot-in-cache friendly (the programmer optionally can use cache directed en-queuing of work requests as well as FIFO en-queuing).

When thread affinity is not used, the application programmer can select between a compute class queue and an I/O class queue. There is a third queue called the compute class overflow queue which may be used in rare circumstances (i.e. to not block en-queue requests by other threads while the primary compute class queue is being allocated additional free nodes performed by the first thread to cause the overflow).

Affinity, when enabled, will, at programmer's direction, pin compute class threads to one or more execution cores. Then the programmer has the choice of using affinity directed task en-queuing of compute class tasks, or non-affinity directed task en-queuing of compute class tasks. I/O class threads are not affinity pinned.

Tasks

Tasks in QuickThread are standard functions (C++), and/or class member functions (C++),) that have a void return, or subroutines in Fortran, and take from 0 to 9 arguments. There is no distinction between a task and function/subroutine as these functions or subroutines can be called directly as well as having task requests thrown at them. For example, the parallel_for template, when creating N tasks will, en-queue N-1 tasks and then directly call the en-queued function (thus saving the overhead of one task en-queuing operation).

From the function's viewpoint there is no distinction between being called as a task and being called directly from within the application.

There are no special considerations when writing functions callable from the task scheduler other than for the standard multi-threaded programming requirement in making the function thread-safe.

At this time, C++ exception handling is not performed between the en-queued task and the en-queuing task. Future revisions may address this issue.

Tasks begin execution upon call (by QuickThread task manager) and terminate upon return. Tasks do not directly make request for work to do, rather they begin life upon receipt of requests for work to do (with optional list of arguments). A task is not a thread waiting in an idle loop, rather, a task is code waiting for a function call.

Using threading model, as opposed to tasking model, the application would start many threads with each initially entering a wait state (e.g. WaitForSingleEvent). Standard threading models tend to incur a higher degree of interaction with the operating system than does a task pool system such as QuickThread.

C++ Programming with QuickThread

The C++ programmer includes the QuickThread header file plus desired template headers.

```
#include <QuickThread.h>
#include <parallel_task.h>          // Optional desired template
#include <parallel_distribute.h>    // Optional desired template
#include <parallel_for.h>           // Optional desired template
#include <parallel_list.h>          // Optional desired template
#include <parallel_reduce.h>        // Optional desired template
#include <parallel_pipeline.h>      // Optional desired template

using namespace qt;                 // Optional namespace
```

And links in the appropriate QuickThread.lib file (x32 or x64 for target O/S)

For the C++ programmer, QuickTread uses the namespace “qt” although some of the Fortran entry points are visible as QUICKTHREADsomenamehere. The programmer has the choice of using the verbose fully qualified names or shorthand templates. If the programmer prefers, they are free to rename the templates or add there own templates.

Program Initialization

There are several methods to initialize your application for using QuickThread.

Default initialization

```
// YourApp.cpp
#include <QuickThread.h>
using namespace qt;

int main(void)    // or with arguments
{
      // Worker Threads = # hardware threads + 1 I/O thread
      qtInit      qtInit(-1);
      // ... your program here
      return YourReturnCode;
}
```

Worker thread count

```
// YourApp.cpp
#include <QuickThread.h>
using namespace qt;

int main(void)
{
      // Worker Threads = 3 threads + 1 (or more) I/O threads
      qtInit      qtInit(3);
      // ... your program here
      return YourReturnCode;
}
```

Worker and I/O thread counts

```
// YourApp.cpp
#include <QuickThread.h>
using namespace qt;

int main(void)
{
      // Worker Threads = 3 threads + 2 I/O threads
      qtInit      qtInit(3,2);
      // ... your program here
      return YourReturnCode;
}
```

Specialty initialization

```
// YourApp.cpp
#include <QuickThread.h>
using namespace qt;
```

```
int main(void)
{
	qtInit	qtInit;	// default ctor
	// default ctor initialized qtInit to default settings
	// but does not start the QuickThread thread pool
	// Optional configuration of the qtInit object here
	// ...
	// Start QuickThread
	if(qtInit.StartQT())
		return YourThreadingErrorReport();

	// ... your program here

	// End QuickThread
	If(qtInit.EndQT())
		return YourThreadingErrorReport();
	return YourReturnCode;
}
```

Alternate specialty initialization

```
// YourApp.cpp
#include <QuickThread.h>

// The Main-level task is the only task that returns a completion code
int MainTask(void* context)
{
	// do something with context if desired
	DoWork(); // or body of your Main Task
	return YourReturnCode;
}

// The C++ program entry point
int _tmain(int argc, _TCHAR* argv[])
{
	GetCommandLineArguments(argc, argv);
	// Declart a qtInit object
	qt::qtInit qtInit;
	// Modify qtInit as desired

	// Launch Main Task
	// with optional argument context
	// return error code
	return qtInit.QueueMain(MainTask, NULL);
}
```

The choice of technique for initialization is left for the programmer to decide.

Task evocation

Tasks on QuickThread are any function returning void

```
void foo(a1T a1, a2T a2, …)
```

This may be a static function or class member function. Functions may have from 0 to 9 arguments.

Functions may be overloaded (same name, different argument lists). In cases where the compiler is unable to disambiguate the function signature a cast may be required on the function name argument.

Internally, tasks on QuickThread, are evoked by way of a **`qtControl`** structure (described later). This **`qtControl`** structure is either implicit or explicit at programmer preference.

Most of your task evocations will likely be by way of the supplied templates. These templates are provided in source form and thus have the provision for user extensibility. The major templates names are:

parallel_task
parallel_distribute
parallel_for
parallel_for_each
parallel_list
parallel_reduce
parallel_pipeline
parallel_wait

The templates will accomidate either static functions or member functions.

Most templates accept a variable number of arguments, some optional, some required, and follow this generalized organization:

```
parallel_{template_suffix} (
      {enqueuing instructions and arguments ouside control of task}
      {address of object when enquing member function as task}
      {address of task}
      {arguments to task}
      );

      Foo(1, “hello world”);
```

becomes

```
      parallel_task(Foo, 1, “hello world”);
```

and

```
      Obj->Fee(1,2,3);
```

becomes

```
      parallel_task(Obj, &Object::Fee, 1, 2, 3);
```

parallel_task

Static function (non-member function)

```
parallel_task(
      [qtPlacement,]
      [&qtControl,]
      &fn,
      [, a1[, a2[, a3[, a4[, a5[, a6[, a7[, a8[, a9]]]]]]]]]
      );
```

`qtPlacement`	an optional placement and/or selection directive (see **qtPlacement**)
`&qtControl`	an optional address of QuickThread Control structure (see **qtControl**)
`&fn`	a required address of Task function name
`a1` through `a9`	are optional function arguments.

Member function

```
parallel_task(
      [qtPlacement,]
      [&qtControl,]
      &Obj,
      &ClassName::fn,
      [, a1[, a2[, a3[, a4[, a5[, a6[, a7[, a8[, a9]]]]]]]]]
      );
```

`qtPlacement`	an optional placement and/or selection directive (see **qtPlacement**)
`&qtControl`	an optional address of QuickThread Control structure (see **qtControl**)
`&Obj`	a required address of object
`&ClassName::fn`	a required address of function name
`a1` through `a9`	optional function arguments.

The distinction between member function and static function is the presense or absence of the address of the object of the class of the member function.

Lambda function (C++0x)

```
parallel_task(
      [qtPlacement,]
      [&qtControl,]
      [&]()
      {
        // ... function body
      }
      );
```

`qtPlacement`	an optional placement and/or selection directive (see **qtPlacement**)
`&qtControl`	an optional address of QuickThread Control structure (see **qtControl**)
`[&]( ){ ... }`	required Lambda function (may use = as well as & and &x,=y, …)

The en-queued tasks are standard functions and/or member functions that can be called directly or enqueue using parallel_task. Lambda functions (C++0x) have no function name and take their arguments by way of a hidden object containing reference (&) or value (=) of current scoped variables.

Note

The following parallel_task examples serve as a tutorial relating to use of qtPlacement and qtControl calling variations and are applicable to other parallel_... templates. The other template descriptions will not repete this information.

Read this section to learn the nuances of task enqueuing that will apply to the other parallel_... templates.

Examples:

```
// static (free) functions
SimpleStaticFunction(1, msg, 101);
parallel_task(&SimpleStaticFunction, 1, msg, 101);
```

Note, most C++ compilers will permit you to drop the & on the static function name. Scoped qualified member function names require the presense of the &.

```
parallel_task(SimpleStaticFunction, 1, msg, 101);
```

When **parallel_task** is issued witout the address of a qtControl structure the address of the Task's current level default control structure is used. The enqueued task will run asynchronous from the code that issued the **parallel_task**.

Using explicit qtControl

```
parallel_task(&qtControl, SimpleStaticFunction, 1, msg, 101);
```

Using explicit cache level directive (to Task's current level default control structure)

```
parallel_task(L2$, SimpleStaticFunction, 1, msg, 101);
```

Using explicit cache level directive to explicit qtControl

```
parallel_task(L2$, &qtControl, SimpleStaticFunction, 1, msg, 101);

// and static member functions (no required &Obj)
CBaseClass::StaticMemberFunction(2, msg, 102);
parallel_task(&CBaseClass::StaticMemberFunction, 2, msg, 102);

// Simple member function (requires address of object)
a.SimpleMemberFunction( 0, msg, 100);
parallel_task(&a, &CBaseClass::SimpleMemberFunction, 0, msg, 100);

// virtual member functions
b.SimpleVirtualFunction(4, msg, 104);
parallel_task(&b, &CBaseClass::SimpleVirtualFunction, 4, msg, 104);

parallel_task(&c, &CDerivedClass::TrickyVirtualFunction, 7, msg, 107);

// parallel_task Lambda function
Total = 0;
parallel_task(
      [&]()
      {
            for(int i=0; i<nVectors; ++i)
                  Total += intVector[i];
      }
);
// *** Caution Total not complete yet
```

Special Note for Overloaded Functions

At times you may be required to cast a function when functions are overloaded and ambiguous.

```
void Foo(int i);
void Foo(long i);
void Foo(double d);
…
char  arg;  // When arg is char, which Foo do you mean?
…
parallel_task( (void(*)(long))Foo, arg); // use void Foo(long i);
```

`parallel_task` is non-blocking. Meaning execution generaly continues past the parallel_task statement while the en-queued task runs (or is scheduled to run).

When `parallel_task` is issued *without* the **qtControl** argument, a default thread context **qtControl** is used. Each thread maintains a task level default **qtControl** structure. You can use **parallel_wait();** (with a null **qtControl**) to insert a barrier for all sub-tasks spawned by way of the thread's task-level default thread context **qtControl**. There is an implicit **parallel_wait();** at end of task for any pending sub-tasks en-queued using a task level default **qtControl** structure.

The optonal **qtControl** argument provides additional control over the task enqueuing. A primary function is to provide object and/or task level placement and thread synchronization. Your task can use multiple **qtControl** objects for coordinating multiple task lists.

The **qtPlacement** argument specifies restrictions on which spawned threads are to be run. When used in conjunction with the **qtControl** argument this provides the programmer considerable control with respect to performance considerations. Examples:

a) Queue the task to the thread that co-resides with the issueing thread's L2 cache.
b) Locate an L3 cache which has the most available threads and condition the **qtControl** structure to enqueue/deque to/from those groups of threads.
c) Condition the **qtControl** structure to distribute current en-queued task and subsequent en-queued tasks to one thread per each L2 cache. Thus providing for each en-queued task to enqueue a sub-task to threads sharing its L2 cache using (a) above

Using thread private task-level default qtControl

```
      // primary tasks first, then secondary tasks
      parallel_task( Task_A );
      parallel_task( Task_B );
      parallel_task( Task_C );
      parallel_wait();
      parallel_task( Task_A2 );
      parallel_task( Task_B2 );
      parallel_task( Task_C2 );
      parallel_wait();
      xxx
```

```
|<->|----Task_A------|         |(prior tasks)|<->|----Task_A2-----|xxx|
|<->|----Task_B-----------|                   |<->|----Task_B2---|
|<->|----Task_C----------------|              |<->|---Task_C2--|

|<------------------------ latency time to run xxx -------------->|
|<------------------------ total elapse time ------------------------>|
```

Notes, "`|<->|`" depicts possible skew in start of task."`|(prior tasks)|`" depicts potential completion time for prior task enqueued using default **qtControl**, and "`|xxx|`" depicts code that follows second parallel_wait.

Using specific **qtControl** object.

```
{     // primary tasks first, then secondary tasks
      qtControl   qtControl;
      parallel_task(&qtControl, Task_A );
      parallel_task(&qtControl, Task_B );
      parallel_task(&qtControl, Task_C );
      parallel_wait(&qtControl); // or qtControl.WaitTillDone();
      parallel_task(&qtControl, Task_A2 );
      parallel_task(&qtControl, Task_B2 );
      parallel_task(&qtControl, Task_C2 );
}
xxx
```

```
|<->|--------Task_A------|         |<->|----Task_A2------|xxx|
|<->|--------Task_B-----------|    |<->|----Task_B2---|
|<->|--------Task_C----------------|<->|---Task_C2--|

|<------------------ latency time to run xxx ----------->|
|<------------------ total elapse time --------------------->|
|<------------------ prior total elapse time ----------------------->|
```

The first example has no consideration for additional pending tasks en-queued by the current task and where there is no requirement to suspend execution.

In the second example additional tasks en-queued by the current task may be pending. The programmer uses the local **qtControl** to en-queue and synchronize a local list of tasks. When the **qtControl** leaves scope, the dtor performs an implicit wait. The advantage of the second method is `Task_A2, Task_B2, Task_C2` do not have to wait for pending task level sub-tasks en-queued prior to the entry of the scope of the local **qtControl** structure.

Through use of multiple **qtControl** objects you can handle multiple task queues concurrently and improve your synchronization appropriately.

```
{     // primary tasks first, partial overlap with secondary tasks
      qtControl   qtControlA, qtControlB, qtControlC;
      parallel_task(&qtControlA, Task_A );
      parallel_task(&qtControlB, Task_B );
      parallel_task(&qtControlC, Task_C );
      parallel_wait(&qtControlA); // or qtControlA.WaitTillDone();
      parallel_task(&qtControl, Task_A2 );
      parallel_wait(&qtControlB); // or qtControlB.WaitTillDone();
      parallel_task(&qtControl, Task_B2 );
      parallel_wait(&qtControlC); // or qtControlC.WaitTillDone();
      parallel_task(&qtControl, Task_C2 );
      xxx
}
yyy

|<->|--------Task_A------|<->|----Task_A2------|
|<->|--------Task_B-----------|<->|----Task_B2---|
|<->|--------Task_C----------------|xxx|<->|----Task_C2--|yyy|

|<------------------ total elapse time ---------------------->|
|<--- latency time to run xxx ---->|
|<------------ prior latency time to run xxx ----------->|
|<------------ prior total elapse time --------------------->|
|<------------ prior prior total elapse time ----------------------->|
```

Using completion nodes may eliminate skew

```
{
      // Using completion node format
      // primary tasks first, complete overlap with secondary tasks
      // inside of scope
      qtControl   qtControlA, qtControlB, qtControlC;
      parallel_task(&qtControlA, Task_A);
      parallel_task(OnDone$, &qtControlA, Task_A2);
      parallel_task(&qtControlB, Task_B );
      parallel_task(OnDone$, &qtControlB, Task_A2);
      parallel_task(&qtControlC, Task_C );
      parallel_task(OnDone$, &qtControlA, Task_A2);
      xxx
}
yyy

|xxx|
|<->|--------Task_A------|----Task_A2------|
|<->|--------Task_B-----------|----Task_B2---|
|<->|--------Task_C----------------|---Task_C2--|yyy|
```

```
// completion nodes with controls outside of scope
qtControl   qtControlA, qtControlB, qtControlC;
{     // primary tasks first, complete overlap with secondary tasks
      parallel_task(&qtControlA, Task_A);
      parallel_task(OnDone$, &qtControlA, Task_A2);
      parallel_task(&qtControlB, Task_B );
      parallel_task(OnDone$, &qtControlB, Task_A2);
      parallel_task(&qtControlC, Task_C );
      parallel_task(OnDone$, &qtControlA, Task_A2);
      xxx
}
yyy

|xxx|yyy|
|<->|--------Task_A------|----Task_A2------|
|<->|--------Task_B-----------|----Task_B2---|
|<->|--------Task_C---------------|---Task_C2--|
```

In the last example, when the qtControl objects are instantiate outside the scope of the task (as static object or passed in as an argument to the task). Then the en-queued task can return with pending sub-tasks.

By varying your use of the qtControl you can significantly alter latencies as well as core utilization. Control of latencies means control of “Hot in cache”.

Completion routines

When **parallel_task** is issued with **qtPlacement** attribute containing **OnDone$** the en-queued task is placed into a FIFO list associated with the **qtControl** object. When all pending tasks complete that are/were en-queued via the specified control object, then the tasks en-queued using the **OnDone$** attribute are processed in FIFO order. The **OnDone$ qtPlacement** attribute can be used to serialize tasks whenever that is in your design requirement.

Proper use of the **qtControl**(s) and the **parallel_task** with placement attribute of **OnDone$** can improve concurrency. N.B. The programmer is free to use **qtPlacement** or other techniques to schedule the completion task to a thread other than the one it waits on.

As you can see, you have considerable control over the sequencing of the tasks. And consequently must take care in your programming to attain the desired results.

Example code outlines for `qtControl` existing outside the scope of the task:

```
void foo(qtControl* qtControl)
{
      //... other code here
      parallel_task(qtControl, Task_A);
      parallel_task(qtControl, Task_B);
      parallel_task(qtControl, Task_C);
      //... other code here
      // exit Task foo with
      // Task_A, Task_B, Task_C pending, running, or complete
}
```

```
void SomeFunction(someArgs)
{
      qtControl qtControl;    // SomeFunction scoped control
      //... other code here
      // use local control as argument to foo
      // Task_A, Task_B, Task_C enqueued via above qtControl
      // use default task level control for control of Task foo
      parallel_task(foo, &qtControl);
      // return while foo pending/running/complete
      // Task_A, Task_B, Task_C pending/running/complete
      // ... run other code here
      // wait for all pending default task level control Tasks
      // including Task foo to complete
      // but do not wait for pending tasks on qtControl
      // i.e. Task_A, Task_B, Task_C may be pending/running/complete
      parallel_wait();
      // ... other code here
      // wait for Task_A, Task_B, Task_C to complete
      // (as well as additional tasks enqueued via this qtControl)
      parallel_wait(&qtControl); // or qtControl.WaitTillDone();
      // ... other code here
} // void SomeFunction(someArgs)

void SomeOtherFunction(someArgs)
{
      qtControl qtControl;
      //... other code here
      // use local control as argument to foo
      // Task_A, Task_B, Task_C enqueued via above qtControl
      // use local control for control of Task foo
      parallel_task(&qtControl, foo, &qtControl);
      // return while foo pending/running/complete
      // Task_A, Task_B, Task_C pending/running/complete
      // ... other code here
      // wait for foo and Task_A, Task_B, Task_C to complete
      // (as well as additional tasks enqueued via this qtControl)
      parallel_wait(&qtControl);
      // ... other code here
} // void SomeOtherFunction(someArgs)

void OtherSomeOtherFunction(someArgs)
{
      // code elsewhere
      qtControl qtControl;
      //... other code here
      // use local control as argument to foo
      // Task_A, Task_B, Task_C enqueued via above qtControl
      // use local control for control of Task foo
      parallel_task(&qtControl, &foo, &qtControl);
      // return while foo pending/running/complete
      // Task_A, Task_B, Task_C pending/running/complete
      // ... other code here
      // implicit parallel_wait(&qtControl);
      // on dtor of local qtControl
      // waits for foo and Task_A, Task_B, Task_C to complete
      // (as well as additional tasks enqueued via this qtControl)
} // void OtherSomeOtherFunction(someArgs)
```

parallel_invoke

The parallel_invoke template is used to invoke two or more Lambda function tasks (C++0x)

```
parallel_invoke(
      [qtPlacement,]
      [*](){ ...Lambda0... },
      [*](){ ...Lambda1... }
      ...); // additional Lambda functions here
```

Where

qtPlacement	Optional qtPlacement argument
[*]	Standard Lambda context ([&], [=], [], [&arg1,=arg2,…)
()	no calling parameters
`{ ...Lambda0... }`	Body of Lambda function 0
`{ ...Lambda1... }`	Body of Lambda function 1
`...`	Additional Lambda functions follow

Example:

```
void rect_interact(int i0, int i1, int j0, int j1)
{
    // if there's room,
    // further subdivide the rectangle into four smaller ones
    int di = i1 - i0; int dj = j1 - j0;
    if (di > grainsize && dj > grainsize)
    {
      int im = i0 + di/2;
      int jm = j0 + dj/2;

      // recursively call ourself to...
      // invoke two tasks for two non-conflicting rectangles
      // return when done
      parallel_invoke(
            [&]() {rect_interact(i0, im, j0, jm);},
            [&]() {rect_interact(im, i1, jm, j1);});

      // recursively call ourself to...
      // invoke two tasks for other two non-conflicting rectangles
      // return when done
      parallel_invoke(
            [&]() {rect_interact(i0, im, jm, j1);},
            [&]() {rect_interact(im, i1, j0, jm);});
    }
    else
    {
      // otherwise all we have left is a strip
      // that can be handled locally
      for (int i = i0; i < i1; ++i)
          for (int j = j0; j < j1; ++j)
                addAcc(i, j);
    }
}
```

and

```
void
body_interactQT(int i, int j, int level)
{
  // split the interaction triangle into upper and lower triangles
  // (which can be executed in parallel) and the adjacent rectangle
  // (which will be further split)
  int d = j - i;
  if (d > 1)
  {
    int k = d/2 + i;
    if(level == 0)
    {
      // using qtPlacement of OneEach_L2$ has the effect of
      // scheduling current thread
      // plus one of a group of L2 caches that is not in my L2
      // first choice will be one of the other L2 cache threads
      // that is waiting. Second choice is closest other L2
      parallel_invoke_v1(
          OneEach_L2$,
          [&](){body_interactQT(i,k,level+1);},
          [&](){body_interactQT(k,j,level+1);});
    }
    else
    {
      // for all other levels (1, 2, ...)
      // schedule to closest cache level
      parallel_invoke_v1(
          [&](){body_interactQT(i,k,level+1);},
          [&](){body_interactQT(k,j,level+1);});
    }
    rect_interactQT(i, k, k, j);
  }
  // if d is 1 or 0 then we can skip it.
}
```

parallel_distribute

The **parallel_distribute** template is used to distribute tasks by way of a selection criterion.

Static function (non-member function)

```
parallel_distribute(
      qtPlacement,
      [&qtControl,]
      &fn
      [, a3[, a4[, a5[, a6[, a7[, a8[, a9]]]]]]]
      );
```

`qtPlacement`	required placement directive
`&qtControl`	address of optional QuickThread Control structure
`&fn`	address of required function or member function name
`a3` through `a9`	optional function arguments.

Note, arguments a1 and a2 are missing as they are implicit. Tasks (argument `fn`) are of the following format:

```
void foo(intptr_t teamMemberNumber, intptr_t membersInTeam[, args]);
```

Lambda function (C++0x)

```
parallel_distribute(
      qtPlacement,
      [&qtControl,]
      [&](intptr_t teamMemberNumber, intptr_t membersInTeam)
      {
            // ...
      }
      );
```

`qtPlacement`	required placement directive
`&qtControl`	address of optional QuickThread Control structure
`&fn`	address of required function or member function name
`[&](...){...}`	Lambda function taking two args

Common:

The **teamMemberNumber** is 0-based and in the range of 0:**membersInTeam**-1. When the thread issuing the **parallel_distribute** is a member of the team its **teamMemberNumber** is 0.

The **parallel_distribute** will use the required **qtPlacement** as a selection criteria for the members of a thread pool. The size of the thread pool could range from 0 to all threads of class. When **qtControl** is missing from the argument list, one will be provided by the template and the

effect of which is the **parallel_distribute** blocks until all threads complete. When **qtControl** is supplied, the **parallel_distribute** does not block. Consider:

```
      parallel_distribute( OneEach_L1$, foo);
```

On a system with HyperThreading, the above would select a team of threads consisting of one of the HT threads from each pair (or sub-group) of threads that share an L1 cache with all L1 caches scheduled. On a 4 core HT system with 8 hardware threads (2 threads per core), a team of 4 threads will be selected, one thread from each core.

Assume your needs are to divide a process up across cores, and then within each core:

```
// one task per core
parallel_distribute( OneEach_L1$, OnePerCore);
...

void OnePerCore(size_t teamMemberNumber, size_t membersInTeam)
{
      for(  size_t row = teamMemberNumber;
            row < numberOfRows;
            row += membersInTeam)
      {
            // only with my HT companion thread(s)
            parallel_for( L1$, DoColumns, 0, numberOfColumns, row);
      }
}
```

The above using Lambda function

```
// one task per core
parallel_distribute(
      OneEach_L1$,
      [&](size_t teamMemberNumber, size_t membersInTeam)
      {
      for(  size_t row = teamMemberNumber;
            row < numberOfRows;
            row += membersInTeam)
            {
                  // only with my HT companion thread(s)
                  parallel_for(L1$,DoColumns,0,numberOfColumns, row);
            }
      }
);
```

Notes: `OneEach_L1$` was used for the first (outer) selection criteria, and `L1$` was used for the second (inner) criteria. L1$ means only threads sharing L1 of thread issuing statement. You can distribute across each/any cache level (L1, L2, L3) as well as NUMA node distances (M0, M1, M2, M3). Additionally you can distribute depending on availability of threads (`Waiting_L1$`)

```
      parallel_for( Waiting_L1$, DoColumns, 0, numberOfColumns, row);
```

In the above scenario (`OnePerCore`) on 4 core HT system the above statement will, depending on availability of other thread sharing L1, will produce either one en-queue for a task to perform half the loop plus a direct call to perform the other half of the loop, or perform a direct call to perform the complete loop.

Loop Parallelization

parallel_for

Static (non-member) function:

```
void parallel_for(
      [qtPlacement,]
      [&qtControl,]
      [iChunk,]
      &fn,
      iBegin,
      iEnd
      [, a3[, a4[, a5[, a6[, a7[, a8[, a9]]]]]]]
      );
```

Member function:

```
void parallel_for(
      [qtPlacement,]
      [&qtControl,]
      [iChunk,]
      &Obj, // of class or class derived from containing fn
      &ClassName::fn,
      iBegin,
      iEnd
      [, a3[, a4[, a5[, a6[, a7[, a8[, a9]]]]]]]
      );
```

Lambda Function (C++0x)

```
void parallel_for(
      [qtPlacement,]
      [&qtControl,]
      [iChunk,]
      iBegin,
      iEnd,
      [&](iBeginT iBegin, iEndT iEnd)
      {
            // preamble if required
            for(iBeginT i=iBegin; i<iEnd; ++i)
            {
                  // body of your loop
            }
            // postamble if required
      } // closure of Lambda function
      );
```

`qtPlacement`	optional placement directive (used with thread affinity)
`&qtControl`	optional QuickThread Control structure
`iChunk`	optional chunking value
`&Obj`	Address of class object when used for member function
`&fn`	address of required function or member function name
`iBegin,iEnd`	are required iteration space fields *(half-open interval)*
`a3` through `a9`	optional function arguments.
[&]	or [&var1, =var2, etc …]
`iBeginT, iEndT`	typedef of iterator such as int, long (generally `iBeginT` same as `iEndT`)

`iBegin` and `iEnd` define the *half-open interval*, expressed in documentation as `[iBegin,iEnd)`. Where `iBegin` is the first of the interval and `iEnd` is one after the last of the interval. The typical programming practice for C++ is:

```
for(int i = iBegin; i < iEnd; ++i)
```

`fn` declares a slice function with two required numeric arguments (integer generally `intptr_t`), and up to 7 optional arguments (a3:a9) which can be scalars, objects, references or pointers. The specified function is run as one or more tasks (with potentially the first or last task run by way of an inline function call).

When issued *without* a **qtControl** object, the template uses a template-level default qtControl within the **parallel_for** function. This type of **parallel_for** is blocking (all threads complete before return).

When issued *with* a **qtControl** object, the parallel_for is non-blocking. Meaning it may return from the parallel_for while execution continues on the **parallel_for**. Use **parallel_wait([qtControl]);** or **qtControl.WaitTillDone();** for synchronization.

Special note for Lambda functions

Lambda functions create a hidden temporary object on the stack of the caller. It is the programmers responsibility to preserve this object for the duration of the call. parallel_for without qtControl is blocking therefore will preserve temporary object for duration of call. parallel_for with qtControl has to be used with care such that WaitTillDone (either implicitly or explicitly) is called prior to exiting the scope of the parallel_for. Also, Lambda functions that modify objects (values) in the scope of the parallel_for must do so in a thread safe manner:

```
std::vector<int> intVector;
// ...
long total = 0;
parallel_for_v1(
      0, nVectors,
      [&total,&intVector](int iBegin,int iEnd)
      {
            long _total = 0;  // local subtotal
            for(int i=iBegin; i<iEnd; ++i)
                  _total += intVector[i];
            AtomicAdd(total, _total); // shared total
      }
);
```

Sample tasks for use with **parallel_for** are of the format:

```
void ArraySum1D(        // arbitrary name
      int iBegin,       // required integral type
      int iEnd,         // required integral type
      double* inA,      // optional arguments
      double* inB,      // optional arguments
      double* outC)     // optional arguments
{
      for(int i= iBegin; i < iEnd; ++i)
            outC[i] = inA[i] + inB[i];
}

      // . . .
      parallel_for(
            // fn, iBegin, iEnd,      inA,    inB,    outC
            ArraySum1D, 0, ArraySize, ArrayA, ArrayB, ArrayC);
      // returns when done

void ArraySum2D(
      int nRows,
      int nCols,
      double* inA,
      double* inB,
      double* outD)
{
      qtControl   qtControl;
      for(int row=0; row<nRows; ++row)
      {
            int iBegin = row * nCols;
            int iEnd = iBegin + nCols;
            parallel_for(
                  &qtControl,
                  ArraySum1D, // use 1D function as task
                  iBegin,
                  iEnd,
                  &ArrayA[iBegin],
                  &ArrayB[iBegin],
                  &ArrayC[iBegin]);
            // returns while pending (or may be complete)
      }
      // dtor implicitly waits till done
}
```

Simple parallel_for

When `qtControl, qtPlacement` and `iChunk` are omitted, the default behavior is to divide up the iteration space up into as many pieces as you have compute class threads. Then en-queue number of compute class threads-1 task requests and directly calling the slice function for the last slice.

Example:

```
// slice function
void DoSum(intptr_t iFrom, intptr_t iEnd, double* A, double* B, double* C)
{
      for( intptr_t i = iFrom; i < iTo; ++i)
            A[i] = B[i] + C[i];
}

      . . .
      double* A = new double[nSize];
      double* B = new double[nSize];
      double* C = new double[nSize];
      . . .
      parallel_for(&DoSum, 0, nSize, A, B, C);
      . . .
```

When, for example, nSize = 1000, and number of threads = 4, threads will run with arguments

```
  0,  250, A, B, C      (thread a)
250,  500, A, B, C      (thread b)
500,  750, A, B, C      (thread c)
750, 1000, A, B, C      (thread d)  (a != b != c != d)
```

Simple parallel_for with iChunk

```
// same slice function
void DoSum(intptr_t iFrom, intptr_t iEnd, double* A, double* B, double* C)
{
      for( intptr_t i = iFrom; i < iTo; ++i)
            A[i] = B[i] + C[i];
}

      . . .
      double* A = new double[nSize];
      double* B = new double[nSize];
      double* C = new double[nSize];
      . . .
      intptr_t iChunk = 100;
      parallel_for(iChunk, &DoSum, 0, nSize, A, B, C);
```

When, for example, nSize = 1000, iChunk=100, and number of threads = 4, threads will run with arguments

```
  0,  100, A, B, C      (thread a)
100,  200, A, B, C      (thread b)
200,  300, A, B, C      (thread c)
300,  400, A, B, C      (thread d)  (a != b != c != d)
400,  500, A, B, C      (thread first of above to finish)
500,  600, A, B, C      (thread second of above to finish)
600,  700, A, B, C      (thread third of above to finish)
700,  800, A, B, C      (thread forth of above to finish)
800,  900, A, B, C      (thread next available)
900, 1000, A, B, C      (thread next available)
```

Including the iChunk parameter the parallel_for divides the range into iChunk sized pieces (last chunk may be less than specified size). If nSize / iChunk is less than the number of available threads then the lesser number of threads are scheduled. When iChunk <= nSize then the slice function is called directly (bypassing the scheduler).

The chunking of a parallel_for is advantageous when the amount of processing is not uniform across the iteration space or when not all of the threads scheduled for the work distribution are immediately available for performing the work, or may get preempted during work (by O/S running other applications).

Simple parallel_for with qtControl

Example:

```
// same slice function
void DoSum (int iFrom, int iEnd, double* A, double* B, double* C)
{
      for( int i = iFrom; i < iTo; ++i)
            A[i] = B[i] + C[i];
}

      . . .
      double* A = new double[nSize];
      double* B = new double[nSize];
      double* AB = new double[nSize];
      double* C = new double[nSize];
      double* D = new double[nSize];
      double* CD = new double[nSize];
      . . .
      {
            qtControl   qtControl;
            parallel_for(&qtControl, DoSum 0, nSize, A, B, AB);
            parallel_for(&qtControl, DoSum 0, nSize, C, D, CD);
      }
```

The above two parallel_for statements run concurrently. The blocking occurs within the dtor of the qtControl structure called at the close brace. The scoping of the qtControl structure is under the programmer's control. Alternately, this example could have omitted the use of the local qtControl structure and used the parallel_wait(); however, the current task may have had additional sub-tasks pending and you may not wish to wait for those additional tasks to complete as well.

Should you wish to block on a qtControl structure without letting it go out of scope you can call the member function WaitTillDone().

```
      double* A = new double[nSize];
      double* B = new double[nSize];
      double* AB = new double[nSize];
      double* C = new double[nSize];
      double* D = new double[nSize];
      double* CD = new double[nSize];
      double* ABCD = new double[nSize];
      . . .
      {
            qtControl   qtControl;
            parallel_for(&qtControl, &DoSum 0, nSize, A, B, AB);
            parallel_for(&qtControl, &DoSum 0, nSize, C, D, CD);
            qtControl.WaitTillDone(); // parallel_wait(&qtControl);
            parallel_for(&qtControl, &DoSum, 0, nSize, AB, CD, ABCD);
            // something else to do while performing parallel_for
      }
```

The reason for coding this way is it eliminates an additional call to the ctor of the **qtControl** object.

parallel_for with placement

You may find it advantageous to specify placement restrictions on the parallel_for (see `qtPlacement`):

Example 1:

Assume you have a loop that has a relatively small number of iterations and you know that most of the data for the loop is “hot in L2 cache” of the current thread, and further you know that the execution time will benefit using multiple threads *only* when the other thread(s) sharing the current thread’s L2 cache are *waiting* to run.

```
parallel_for(Waiting_L2$, DoSum, 0, nSize, A, B, C);
```

Example 2:

Assume you have a loop that has a relatively small number of iterations and you know that most of the data for the loop is *not in cache* (in RAM) and further you know that the execution time will benefit using multiple threads only when the other thread(s) are available and also share the same L2 cache with each other.

```
parallel_for(Waiting_NotInCache_L2$, fnFoo, 0, nSize, A, B, C);
```

Example 3:

You have a relatively long running loop but each instance of the running loop (each slice) will run optimally when all the data resides within one L3 cache

```
parallel_for(NotInCache_L3$, fnFoo, 0, nSize, A, B, C);
```

Assuming your system has two quad core processors each with L3 cache capability. The processor with the most available threads at L3 would be selected and the number of threads scheduled will be the number of HW threads sharing the same L3 (4 in this case).

parallel_for_each

Task

```
void fn(int iPos[, optional args])
{
      // ...
}
```
The function fn is run in parallel one element at a time using optional arguments.

Static function

```
void parallel_for_each(
      [qtPlacement,]
      [&qtControl,]
      iEnd,       // half open termination point
      &fn,        // void fn(intptr_t i[, ...]);
      iBegin      // beginning and varying position
      [, a3[, a4[, a5[, a6[, a7[, a8[, a9]]]]]]]); // optional args
```

Member function

```
void parallel_for_each(
      [qtPlacement,]
      [&qtControl,]
      iEnd,       // half open termination point
      [&Obj,]     // object when member function
      &Object::fn,// void Object::fn(intptr_t i[, ...]);
      iBegin      // beginning and varying position
      [, a3[, a4[, a5[, a6[, a7[, a8[, a9]]]]]]]); // optional args
```

Lambda Function (C++0x)

```
void parallel_for_each(
      [qtPlacement,]
      [&qtControl,]
      iEnd,       // half open termination point
      [&](intptr_t i)
      {
            // i=item number
      }
);
```

parallel_for_each schedules or runs a (or multiple) dispatching task(s). The dispatching task(s) will run making calls across the iteration space (iBegin, iEnd) to the supplied function (with optional args). As the dispatching task runs, it monitors for the availability of additional threads to join the team processing the **parallel_for_each**. Monitoring is at beginning of task, then periodically during dispatching. When an available thread or threads are observed, the remaining

iteration space of the observing thread is divided, a portion is held for the current thread, and the remainder is en-queued for the idle thread. The new task is a recursive call of the current distribution task and it runs in parallel with the first task. Only when additional threads become available is the iteration space split.

Note, the various dispatching task(s) may run out of iteration space at differing times. When a busy dispatching task notices a new idle thread it will split its remaining iteration space and en-queues a new task.

The number of task en-queuing operations for **parallel_for_each** is generally larger than **for parallel_for**. However, when the execution time varies across the iteration space, a non-chunking **parallel_for** might not fully utilize all cores (i.e. some threads finish early while others finish late). A **parallel_for** with chunking can improve the utilization of all cores but at the expense of incurring a fixed number of additional task en-queue/de-queue. Use of **parallel_for_each** is thread availability demand related number of additional task en-queue/de-queue operations. As to which method is best, this will depend on your application.

The criteria for using **parallel_for_each** are:

a) The work per object is variable and significant with respect for the cost of intermittent task en-queue/de-queue operations.
b) Your requirement is to distribute objects using qtPlacement into specified cache systems such that the object sub-tasks can subsequently schedule within that cache.
c) The availability of threads to schedule is uncertain at the issuance of the **parallel_for_each** and may change during the execution of the control loop.

Example:

```
void  DoObject(intptr_t iObj);      // Function to do per object

void  DoObjectsParallel()
{
      // Create 1 or more instances of DoObject task
      // split up across (for each) L2 cache on system
      // (one task per L2).
      //
      //                qtPlacement, iEnd,     fn,       iBegin
      parallel_for_each(OneEach_L2$, nObjects, DoObject, 0);
}

void DoObject(intptr_t iObj)
{
      // Obtain an object from the Object List
      // The Object contains three matricies
      // A and B are to be multiplied and stored in output
      Object& Obj = ObjectList[iObj];

      // Perform the matrix multiplication row at a time
      // using a thread pool of threads within my L2
      parallel_for(
            L2$,              // only threads sharing our thread's L2
            matmultStripe,    // function to perform striped matmul
            0, Obj.Height,    // split across number of threads on L2
            Obj.inputA, Obj.inputB,
            Obj.output,       // results
            Obj.Width, Obj.Height);
```

```
 }
```

Note, `matmultStripe` would be optimal for this Object when all cores using the selected L2 would be available. This is not normally the case. As such, not all stripes will finish at the same time. When the array size is relatively large and the system is working hard, then consider using the iChunk form of the parallel_for.

Example using iChunk follows:

```
void DoObject(intptr_t iObj)
{
      // Obtain an object from the Object List
      Object& Obj = ObjectList[iObj];
      parallel_for(
            L2$,              // only our thread's L2
            Obj.Height / 4,   // iChunk = 1/4 of total
            &matmultStripe,   // function to perform striped matmul
            0, Obj.Height,    // split into iChunk sizes
            Obj.inputA, Obj.inputB,
            Obj.output,       // results
            Obj.Width, Obj.Height);
 }
```

parallel_reduce

Static Function

```
void parallel_reduce(
      [qtPlacement,]
      [&qtControl,]
      &fn         // void fn(intptr_t iBegin, intptr_t iEnd[, ...]);
      iBegin,     // beginning and varying position
      iEnd,       // half open termination point
      Reduction   // reduction object
      [, a3[, a4[, a5[, a6[, a7[, a8[, a9]]]]]]]); // optional args
```

Member Function

```
void parallel_reduce(
      [qtPlacement,]
      [&qtControl,]
      &Obj,       // Address of an object
      &Object::fn // void Object::fn(intptr_t iBegin, intptr_t iEnd...
      iBegin,     // beginning and varying position
      iEnd,       // half open termination point
      Reduction   // reduction object
      [, a3[, a4[, a5[, a6[, a7[, a8[, a9]]]]]]]); // optional args
```

Lambda Function (C++0x)

```
void parallel_reduce(
      [qtPlacement,]
      [&qtControl,]
      [&](intptr_t iBegin, intptr_t iEnd, ReductionT Reduction)
      {
            // ...
      }
);
```

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).

The 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

And example of a simple reduction object would be for summation of a 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);
```

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.

In the following example the reduction object is compound. It contains a value and an index.

Serial Code

```
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;
}
```

Using parallel_reduce

```
// 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;
}
```

parallel_list

Static Function

```
void parallel_list(
      [qtPlacement,]
      [&qtControl,]
      &fn,        // void fn(intptr_t i[, ...]);
      root        // pointer to beginning of null terminated list
      [,a2[,a3[,a4[,a5[,a6[,a7[,a8[,a9]]]]]]]]); // optional args
```

Member Function

```
void parallel_list(
      [qtPlacement,]
      [&qtControl,]
      &Obj,       // address of an Object
      &Object::fn,// void Object::fn(intptr_t i[, ...]);
      root        // pointer to beginning of null terminated list
      [,a2[,a3[,a4[,a5[,a6[,a7[,a8[,a9]]]]]]]]); // optional args
```

Lambda Function (C++0x)

```
void parallel_list(
      [qtPlacement,]
      [&qtControl,]
      [&](Item* node)
      {
            // process one node in the list
      }
);
```

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

Serial

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

Parallel

Process a null terminated list using all threads:

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

or simply insert parallel_list inline in your code

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

Depending on the selection chriteria (optional **qtPlacement**), one or more threads will be enguaged to process the singly linked list of objects (linked on Object->next beginning with root).

```
parallel_list(L1$, Foo, root);
```

The above would restrict the processing of the list to those thread(s) sharing the current threads L1 cache and then only subject to availability of that thread.

parallel_pipeline

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

a) Closed ends self running ring buffer
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. Pipelines can be thought of as a sequence of abstract steps to be performed.

The description of a **parallel_pipeline** is best made by way of an example. The majority of the work in using the **parallel_pipeline** is in the setup.

The following example serial program creates a text file containing lower case words plus interspersed sequence numbers. Once created, the task is to read the file and up-case the first letter of each word and write the resultant file to the disk while maintaining the original sequence.

Serial program

```
// 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();
}

// The UpcaseWords function is the portion of the application
// we wish to parallize by use of parallel_pipeline.
// First study the serial version of the code.
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();
}
```

Parallel version

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
{
};
```

The composit class consists of a standardized header (`PipelineBuffer`) and an arbitrary context (`MyBuffer`). Prefix format provides for abstracton as the internal workings of the pipeline need not know about the contents of your context.

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.

In the case of the Serial example above you would convert the read file portion, together with the initialization of state variables, into a task. The serial code:

```
            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 );
```

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
            // 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.).

The following from the Serial code

```
            // 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);
            }
```

Is converted into a task

```
// task to process buffer
void PipelineProcessBuffer(MyPipelineBuffer* b)
{
      // 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);
	}
}
```

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. Convert the Serial statements

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

into a task

```
// 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 );
	// (you may run other code here 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 a pipeline, 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.

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; // arbitrary (unique) number
const qtPipelineTag MergeIt$ = 1234;

...
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.

concurrent_proxy_vector

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

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 single Interlocked… instruction.

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 concurrent_vector whereby the programmer must take into consideration the possibility of incomplete vectors. The 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 symptoms. Objects are allocated and constructed *before* an insertion attempt is made to the **concurrent_proxy_vector**. Insertion into the vector is performed with a single atomic instruction..

The **concurrent_proxy_vector** never contains objects under construction. The container for the proxies is contiguous providing fast indexing [] operators, however, when a larger container is

required, a container expansion operation occures. The design of the **concurrent_proxy_vector** is such that the current (old) container can continue to be used during the re-allocation of the newer (larger) container. Thus there is no blocking when enlarging the container. When the prior container(s) is(are) known to not have references then they are returned for recycling.

It is safe to delete items in the middle of the **concurrent_proxy_vector** container provided you make a minor change to your usage part of the program to accommodate for the possibility of a NULL reference (pointer) being returned during iteration. The preferred technique would be to augment the iterator ++() operator to skip over the deleted items. However, use of [] operator might return a deleted reference. It is relatively simple to test for a NULL reference.

Memory Allocation

QuickThread has one allocator **qt_allocator<T>**

The single QuickThread allocator can be declared to perform allocations without alignment considerations as well as 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 member functions of the allocator are

```
pointer allocate(size_type n);                  // allocate n objects
pointer allocate(size_type n, const void* u);   // allocate n objects or to alignment restriction
                                                // whichever is larger
void deallocate( pointer p, size_type n);       // deallocate n objects or to alignment restriction
                                                // whichever is larger
size_type max_size();                           // largest meaningful value for allocate
void construct( pointer p, const T& t);         // constructor
void destroy( pointer p );                      // destructor
```

The following allocation/deallocation functions are also callable;

```
void* qt_malloc(size_t size);                   // allocate size number of bytes
void* qt_calloc(size_t nitems, size_t size);    // allocate n * size number of bytes
void* qt_realloc( void* block, size_t size);    // reallocate (and copy)
void  qt_free( void* block );                   // return memory
```

It is recommended that you use the qt_allocator because the code is optimize for use with the qt_allocator.

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 are hierarchical and fast. Allocation is attempted in the following order

a) Thread's local pool of compatibly sized prior allocations
b) Thread's NUMA node pool of pool of compatibly sized prior allocations
c) Thread's NUMA node overflow list of pools of pools of compatibly sized prior allocations
d) Thread's NUMA node overflow list of pools of compatibly 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)

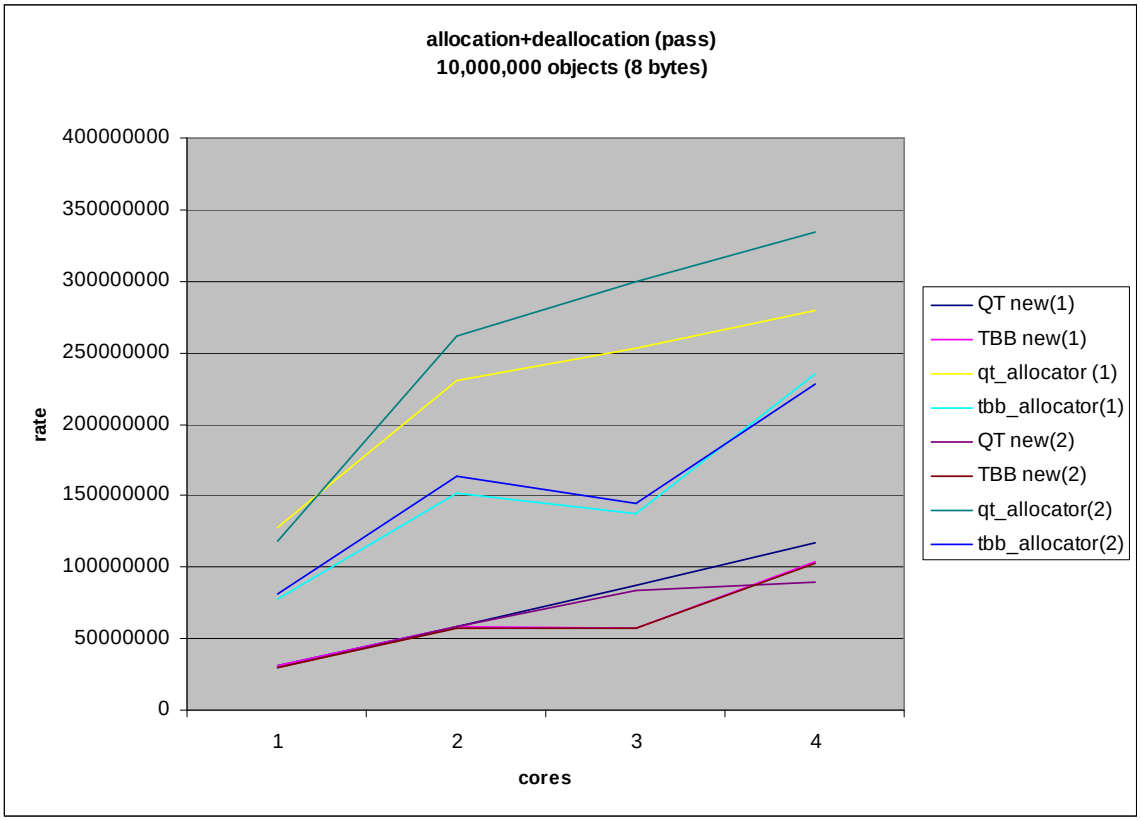
Excepting for low memory conditions, when the thread that allocates an object is the same thread that de-allocates the object the object will stay in the same NUMA pool of pools of compatibly sized prior allocations. This is true too when the de-allocating thread is pinned to the same NUMA node. Allocation from a thread in one NUMA pool, followed by de-allocation of that object from a thread in a separate NUMA pool is permitted; however, subsequent allocation of that object may yield non-optimal memory access. The most efficient place to sort out the de-allocation is within the application as opposed to within the de-allocation routine.

Although the NUMA considerations appear to make the overly complex for use in smaller (and non-NUMA) systems the code paths are relatively short and significantly faster than other scalable allocators.

NOTICE

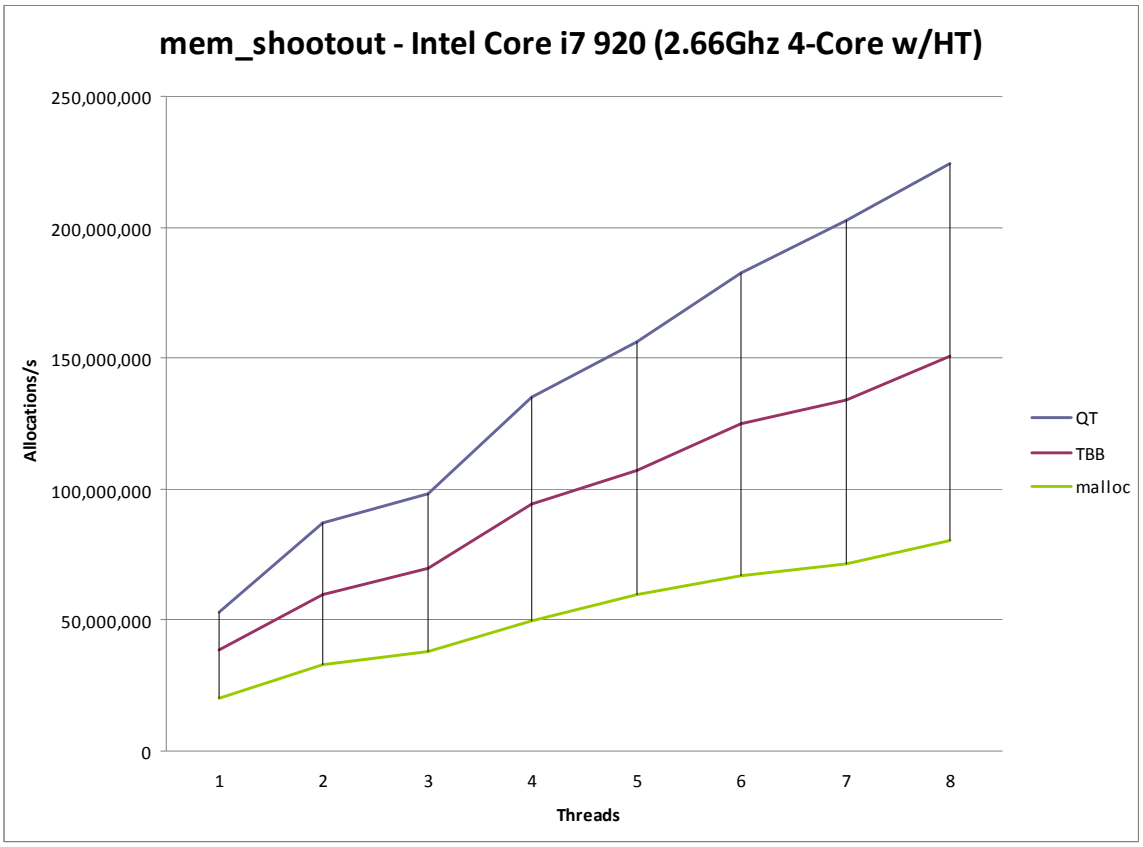
The **qt_allocator** can be used by auxiliary threads created by the user application. As well as being used prior to initialization of QuickThread. (thus permitting static constructors to have access to the allocator). These threads are not part of the QuickThread thread pool(s). As such, when these threads terminate they do so without QuickThread interaction. The **qt_allocator** maintains data objects in thread local storage (other features of QuickThread may do so as well). Some of the thread local storage context is allocated memory – some from the C++ heap, and some from the qt_allocator pool of pools. In order to reclaim this memory it is necessary to make a call to **qt::ExitingThread();** just prior to exiting your non-QuickThread thread. There is no requirement to make this call should your application terminate shortly thereafter When your application repeatedly starts and stops non-QuickThread threads and when these threads use the **qt_allocator** (or other QuickThread support features), then it is recommended you call **qt::ExitingThread();** just prior to exiting your thread.

Test program results on an Intel Q6600 allocating 10,000,000 8-byte objects then de-allocating those objects, then repeating the allocation and de-allocation. Thus illustrating faster 2nd allocation.



The bottom group of lines are the test program using the standard C++ new and delete for the objects. The 3 core performance hit for TBB new may be an anomaly. The middle two lines are the TBB tbb_allocator first pass allocation/deallocation (1) and second pass allocation/deallocatoin (2). The upper two lines are the QT qt_allocator first pass allocation/deallocation (1) and second pass allocation/deallocatoin (2).

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 an Intel 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…).



qtInit

```
struct qtInit
{
    // nWorkerThreads = -1;
    //
    // The number of compute intensive threads.
    //
    // Most applications run best when the number of Worker Threads
    // is equal to the number of cores available to the application.
    // Using a larger number of Worker Threads than the number of
    // cores available to the application generaly introduces extra
    // operating system overhead to perform thread context switching.
    //
    // If you are unable to export program I/O to an IoThread then the
    // use of additional Worker Threads may be warranted.
    //
    //   -2 = Set number of worker threads to number of available cores
    //              excluding HT companion thread(s)
    //   -1 = Set number of worker threads to number of available cores
    //    0 = No worker threads
    //   +n = Number of worker threads the application desires
    //
    int_Native  nWorkerThreads;

    // nIoThreads = -1;
    //
    // Number I/O threads
    //
    // -1 = max(1, (Number of available processors - nWorkerThreads))
    // 0 =  No I/O threads
    // +n = Number of I/O threads
    int_Native  nIoThreads;

    // QueueSize = 0; // 0 = Use default QueueSize (512)
    //                           // 0:32768 are valid entries
    //
    // When the queue size is too small, such as
    // less than the number of cores available to the application,
    // then the worker threads may become starved for work.
    // When the queue size is too large, then the object latency may
    // become too large. The object latency is the time interval
    // between queueing the work for an object and when work begins.
    //
    // 0 = Use default QueueSize (512)
    //
    int_Native QueueSize;

    // StackLevelMax = 0;  // 0 = Use default (1024)
    // values enforced [128:32768]
    // StackLevelMax is a task nesting level max count.
    // StackLevelMax is not task stack size

    int_Native StackLevelMax;
```

```
// DefaultSpinWait = 512;
//
// SpinWait is used during a WaitTillDone()
// Each qtControl has a private SpinWait
// Each thread has a private DefaultSpinWait
// The task pool has a DefaultSpinWait
// qtInit.DefaultSpinWait sets the task pool-wide DefaultSpinWait
// Until thread issues qtControl.DefaultSpinWait(n) thread uses
// task pool-wide DefaultSpinWait
// Threads may on a case by case bases directly set
// qtControl.SpinWait
//
// SpinWait is used by the qtControl.WaitTillDone();
// (implicitly by parallel_task_wait())
// The WaitTillDone() is used for task synchronization.
// Depending on the nature of the enqueued tasks the programmer
// may wish to handle the busy wait condition differently.
// Options are:
//
//  a) Task steal when busy
//  b) Wait when busy
//  c) Run in _mm_pause loop while busy
//  d) Run in SwitchToTask loop while busy
//  e) Run in Sleep(0) loop while busy
//
// Option a) Task steal when busy includes a count
// indicating the number of times an attampt at task
// steal fails prior to suspending the thread.
//
// This is similar to the OpenMP BusyWait
//
// When (while) qtControl indicates busy:
//
// SpinWait = 1:n   Attempt task steal
//                   perform task when found,
//                  else
//                   after SpinWait number of failed attempts
//                        suspend thread.
//
// SpinWait = 0     Suspend thread without task stealing
// SpinWait = -1    Run in _mm_pause loop while busy
// SpinWait = -2    Run in SwitchToTask loop while busy
// SpinWait = -3    Run in Sleep(0) loop while busy
//
// Note, at thread pool startup, for non-master threads
// each has a root level qtControl object that is set to
// busy until shutdown. These root level qtControl objects
// cannot be set with SpinWait < 1
// To do so would inhibit them from performing work
// Once in a stolen task the SpinWait can be adjusted to
// defaults or according to specific requirememts/
int_Native DefaultSpinWait;

// nAffinityGrouping = 1;
//
// nAffinityGrouping is the cache level granularity
//
```

```
    // 0 = Thread Processor Affinity not used
    // 1 = L1 granularity (generally 1 core per L1)
    // 2 = L2 granularity (often 2 cores per L2)
    // 3 = L3 granularity (often 4, 6, or 8 cores per L3)
    //           (diminishes to cores per L2 when L3 not present)
    // 4 = Cores per package
    // ...
    //
    //
    // The most effective grouping depends on the application
    // as well as the archetecture of the system.
    // Using Affinity introduces additional overhead in the queue
    // and dequeue of work nodes in the system.
    // It is best to experiment with nAffinityGrouping to
    // find the best setting for the application on a given system.
    //
    // The recommended order for experimentation:
    //
    //   nAffinityGrouping = 0;     // Base level (affinity off)
    //   nAffinityGrouping = 1;     // Individual core level
    //   nAffinityGrouping = 2;     // # cores sharing L2
    //   nAffinityGrouping = 3;     // # cores sharing L3
    //   nAffinityGrouping = 4;     // # cores per package
    //
    int_Native nAffinityGrouping;

    // Status = 0;
    //
    // Initialization status
    //
    // 0 = Success
    // 1 = Initialized more than once
    // 2 = nWorkerThreads is invalid
    // 3 = nIoThreads is invalid
    // 4 = No Threads requested (nWorkerThreads + nIoThreads equals 0)
    // 5 = QueueSize is invalid
    // 6 = nAffinityGrouping is invalid
    int_Native Status;
    qtInit();
    qtInit(int_Native _nWorkerThreads, int_Native _nIoThreads = -1);
    ~qtInit() {EndQT();};
    // Call application from shell
    // (starts QT thread pool and runs to completion)
    int     QueueMain(FnMain* MainCode, PVOID context);
    // alternately
    // start QT thread pool and return immediately
    int StartQT();
    int EndQT();
};
```

Notes,

QueueSize – Do not assume larger queue size means better performance. Not all work nodes are equal. In particular, the thread scheduler will limit the scheduling of work nodes to a thread if that work node were en-queued from that thread and at a higher nesting level of the work stealing call stack. Not obeying this rule usually results in the work stealing stack to grow infinately large. Too large of QueueSize may result in unnecessary overhead in filtering out inappropriate work

nodes. QueueSize is an upper limit on the number of pending work nodes en-queued via individual qtControl structures (explained later). The application may have an unlimited number of these qtControl structures (subject to memory capacity). When this upper limit is reached for a particular qtControl structure, the thread issuing the enqueuing operation enters into task stealing mode (calling downwards in the task stealing call stack). Upon completion of the the stolen task, the number of pending work nodes is checked against QueueSize. If the number of pending work nodes is less than QueueSize then the work stealing mode terminates, else the enqueuing task continues in work stealing mode. Generally, the best QueueSize is one that has a few work nodes pending upon return from task stealing mode. This number is application dependent and not pre-determinable.

Consider nAffinityGrouping as experimental for groupings other than 0 or 1.

qtPlacement

When affinity is used, task en-queuing can specify an optional placement restriction.

```
typedef unsigned short qtPlacement;
const qtPlacement L0$ = 0;          // All threads sharing my L0
const qtPlacement L1$ = 1;          // All threads sharing my L1
const qtPlacement L2$ = 2;          // All threads sharing my L2
const qtPlacement L3$ = 3;          // All threads sharing my L3
const qtPlacement M0$ = 4;          // All threads sharing my M0
const qtPlacement M1$ = 5;          // All threads sharing my M1
const qtPlacement M2$ = 6;          // All threads sharing my M2
const qtPlacement M3$ = 7;          // All threads sharing my M3
const qtPlacement Level$ = 7;
const qtPlacement AllThreads$ = 7;
const qtPlacement Waiting$ = 8;     // Select waiting threads only
const qtPlacement Waiting_L0$ = Waiting$ + L0$;
const qtPlacement Waiting_L1$ = Waiting$ + L1$;
const qtPlacement Waiting_L2$ = Waiting$ + L2$;
const qtPlacement Waiting_L3$ = Waiting$ + L3$;
const qtPlacement Waiting_M0$ = Waiting$ + M0$;
const qtPlacement Waiting_M1$ = Waiting$ + M1$;
const qtPlacement Waiting_M2$ = Waiting$ + M2$;
const qtPlacement Waiting_M3$ = Waiting$ + M3$;
const qtPlacement AllWaitingThreads$ = Waiting$ + AllThreads$;
const qtPlacement NotInCache$ = 16; // Indicate data not in cache
const qtPlacement NotInCache_L0$ = NotInCache$ + L0$;
const qtPlacement NotInCache_L1$ = NotInCache$ + L1$;
const qtPlacement NotInCache_L2$ = NotInCache$ + L2$;
const qtPlacement NotInCache_L3$ = NotInCache$ + L3$;
const qtPlacement NotInCache_M0$ = NotInCache$ + M0$;
const qtPlacement NotInCache_M1$ = NotInCache$ + M1$;
const qtPlacement NotInCache_M2$ = NotInCache$ + M2$;
const qtPlacement NotInCache_M3$ = NotInCache$ + M3$;
const qtPlacement Waiting_NotInCache_L0$ = Waiting$+NotInCache$+L0$;
const qtPlacement Waiting_NotInCache_L1$ = Waiting$+NotInCache$+L1$;
const qtPlacement Waiting_NotInCache_L2$ = Waiting$+NotInCache$+L2$;
const qtPlacement Waiting_NotInCache_L3$ = Waiting$+NotInCache$+L3$;
const qtPlacement Waiting_NotInCache_M0$ = Waiting$+NotInCache$+M0$;
const qtPlacement Waiting_NotInCache_M1$ = Waiting$+NotInCache$+M1$;
const qtPlacement Waiting_NotInCache_M2$ = Waiting$+NotInCache$+M2$;
const qtPlacement Waiting_NotInCache_M3$ = Waiting$+NotInCache$+M3$;
const qtPlacement OneEach$ = 32;    // Choose one thread per cache
const qtPlacement OneEach_L0$ = OneEach$ + L0$;
const qtPlacement OneEach_L1$ = OneEach$ + L1$;
const qtPlacement OneEach_L2$ = OneEach$ + L2$;
const qtPlacement OneEach_L3$ = OneEach$ + L3$;
const qtPlacement OneEach_M0$ = OneEach$ + M0$;
const qtPlacement OneEach_M1$ = OneEach$ + M1$;
const qtPlacement OneEach_M2$ = OneEach$ + M2$;
const qtPlacement OneEach_M3$ = OneEach$ + M3$;
const qtPlacement ExcludeMyCacheLevel$ = 64;    // Exclude my thread
const qtPlacement OnDone$ = 128;    // enqueue when qtControl done
const qtPlacement IO$ = 256;        // enqueue to I/O queue
const qtPlacement IOOnDone$ = IO$+OnDone$;//.. I/O when qtControl done
```

```
const qtPlacement FIFO$ = 512; // enqueue in FIFO order
const qtPlacement IsSplitable$ = 1024;
// parallel_list permited to split after first division

struct qtPlacementBitFields
{
      unsigned Level : 3;           // 0:2 Cache, 4:7 RAM
      unsigned Waiting : 1;                     //  8
      unsigned NotInCache : 1;                  // 16
      unsigned OneEach : 1;                     // 32
      unsigned ExcludeMyCacheLevel : 1;   // 64
      unsigned OnDone : 1;                      // 128
      unsigned IO : 1;                          // 256
      unsigned FIFO : 1;                              // 512
      unsigned IsSplitable : 1;                 // 1024
      unsigned reserved : 16-11;
};
```

The qtPlacement values are optional and when used, provide the programmer with additional tuning capability.

As an example, assume you have just finished a code section and know that the data for the next processing step is contained in the current thread's L2 cache and you also know that the next step will run faster with multiple threads but only if the multiple threads share the L2 cache with the current thread and further know the next processing step will only run faster if the additional threads scheduled are waiting for work (i.e. not busy working on something else). For this circumstance use `Waiting_L2$.`

```
  parallel_for(Waiting_L2$, Foo, 0, n, A, B, C);
```

On system without HT the above loop will be split into two only when the other thread sharing the current thread's L2 cache is available. Otherwise the loop will be run with one thread (the current thread via direct function call).

A second example would be if:

a) you know that the data is not currently in cache and
b) the runtime is relatively long and
c) that the working set fits nicely within an L3 cache (the system has multiple processors with L3).

By specifying `NotInCache_L3$` all the threads sharing the L3 of the processor with the most waiting L3 threads will be scheduled to run the task(s). The waiting threads will begin first, with the expectation that the remaining threads sharing that L3 will begin shortly.

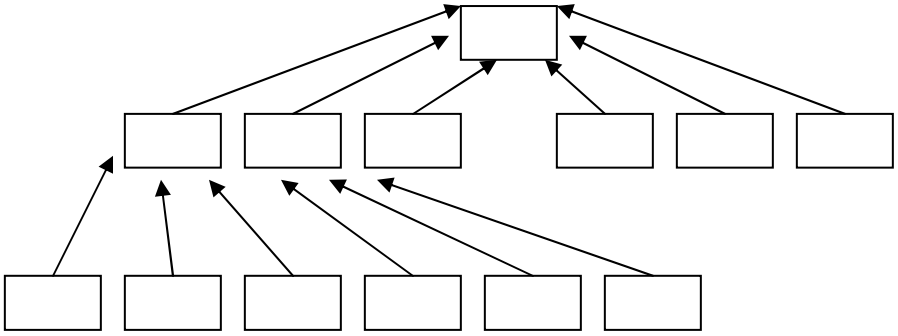
qtControl

QuickThread applications communicate with the task scheduler by way of a control structure (of type qtControl). The programmer can create any number of qtControl objects. The template parallel_task.h will implicitly use a thread private, task level, default qtControl object resulting in non-blocking effect, whereas parallel_for.h will default to a scoped qtControl object resulting in a blocking effect. The programmer can provide an explicit qtControl object to override default behavior.

Because QuickThread.h deffinition of the qtControl structure is rather large, only the important parts of this structure will be summarised below. Consult QuickThread.h for a complete description.

```
// When we are a child qtControl
// this points to our parent qtControl
struct qtControl* Parent;
```

The programmer can construct dependency trees of their qtControl structures. An example of use would be where a sub-task uses its own qtControl structures but also wishes to notify it's parent control structure that task requests are pending. In this manner the parent task need only to monitor the parent's qtControl object as opposed to each sub-task control structure (which may be transient).



Whenever qtControl structures are linked there is additional overhead only for the first en-queue and last de-queue via that control structure. This overhead is typically less than individually polling each decendent qtControl structure.

```
// Count is incremented at the beginning of
// the insertion. Therefore the queue may
// appear empty while Count > 0.
// Count includes an additional tick when
// there exists any QueueOnDoneList nodes.
// A qtControl node is idel when
// Count == 0
volatile int_Native Count;
```

The Count value reflects the number of pending requests (including potentially the one in the process of being en-queued, plus one tick when there are pending completion nodes). The Count is incremented by way of QueueWork or similar member functions. The count will not reflect the number of pending completion nodes instead there is only 1 tick for any number of completion nodes. The count also reflects (by increment) any dependent qtControl (with non-zero Count), linked to this qtControl.

```
Affinities  Affinity;
```

Where:

```
struct Affinities
{
 uint_Native Queue;     // first order preference affinities
 uint_Native DeQueue;   // required affinities
 uint_Native Waiting;   // affinities of threads waiting for done
 uint_Native SelectAffinities(
                qtPlacement Placement);
 uint_Native SelectAffinitiesSharingCacheLevel(
                qtPlacement Placement);
 uint_Native SelectAffinitiesSeperatedByCacheLevel(
                qtPlacement Placement);
};
```

The Affinity.Queue and Affinity.DeQueue are used when the Affinity option is enabled. The AffinityQueue is a bitmask of the processor affinities deemed to be the primary set of processors to run the next en-queued tasks (via the qtControl structure). The Affinity.DeQueue is a bitmask of the processor affinities permitted to deque a task request. Generally Affinity.Queue is a sub-set (or complete set) of Affinity.DeQueue. Threads will check their primary queues first, then secondary queues next.

Caution, it is recommended that you not rely upon Affinity.Queue and Affinity.DeQueue remaining as a bit mask. Use the member functions to preset these fields. When Windows 7 is released, the operating system will be able to support more hardware threads than the number of bits available in the uint_Native word length. Future versions of QuickThread will likely encode values into these field (much similar to the qtPlacement encoding).

```
// Queue On Done FIFO list
QuickThreadCallNodePointer    QueueOnDoneListHead;
QuickThreadCallNodePointer    QueueOnDoneListTail;
```

The qtControl structure can maintain a singly linked list (FIFO) of task completion request nodes. The programmer has the option of specifying a completion routine (or several completion routines) to be run when the tasks en-queued via the qtControl structure have completed.
It should be noted that should you enqueue a completion task to an “empty” qtControl structure that the completion task begins immediately. Therefor the recommended programming practice is to queue the completion requests after queuing all the work requests. You can continue queuing additional work request while completion requests are pending, however, should you lag in your en-queueing operations, then a (the, some) completion node(s) may run prior to completion of the en-queuing phase of your task.. Each subsequent enqueuing of completion tasks via the qtContro,l while the current completion node is being processed, will be linked into the completion node list. The completion nodes have the use once characteristic.

What this means is you can use the completion node task list of a qtControl to serialize steps that would otherwise be performed in parallel. This can be an attractive feature when running older sections of code that are not thread safe.

```
uint_Native Status;
```

The Status word is intended to pass error codes back up through the qtControl node tree (assuming you use one)

```
union
{
    uint_Native   Mode;
    struct
    {
      unsigned HalfOpenRange : 1;   // used by parallel_for
      unsigned CallFirstIteration : 1;// used by QueueDo
      unsigned CallLastIteration : 1; // used by QueueDo
      unsigned AffinityForAllocation : 1; //
      unsigned AffinityIsTaskNumber : 1;  //
      unsigned reserved : 3;
      unsigned Placement : 16;      // qtPlacement
    };
} ModeFields;
```

The Mode field is used by the template library and internally by QuickThread to alter the behavior of qtControl member functions.

HalfOpenRange is use to convert Fortran style closed range into C++ half open range (see parallel_for.h).

CallFirstIteration and CallLastIteration are used by QueueDo to specify if (when) the enqueuing task perform a direct function call to perform the first and/or last iteration.

AffinityForAllocation indicates the next en-queued tasks are to run synchronously at the specified Affinity.Queue/Affinity.Dequeue affinity. This is used on NUMA systems to assure completion of allocation at designated node prior to use.

AffinityIsTaskNumber When set, the Affinity.DeQueue holds the thread pool task number of the task permitted to read the en-queued nodes. As opposed to the bit mask of the affinity.

Caution, the Mode bit fields are subject to change.

```
int_Native SpinWait;
```

The SpinWait is used to control how and if the thead is to task steal when the current qtControl object is busy. The default SpinWait is setable at initialization time using the SetDefaultSpinWait member function of qtInit. The default for DefultSpinWait = 128.

SpinWait is used during a WaitTillDone(). Each qtControl has a private SpinWait. Each thread has a private DefaultSpinWait. The task pool has a DefaultSpinWait. qtInit.DefaultSpinWait(n) sets the task pool-wide DefaultSpinWait. Until a thread issues qtControl.DefaultSpinWait(n) the thread uses the task pool-wide DefaultSpinWait. Threads may on a case by case bases directly set qtControl.SpinWait=n.

SpinWait is used by the qtControl.WaitTillDone(); function (implicitly by parallel_wait()). The WaitTillDone() is used for task synchronization. Depending on the nature of the en-queued tasks the programmer may wish to handle the busy wait condition differently.

The SpinWait value can be used to provide the following options:

a) Task steal when busy
b) Wait when busy
c) Run in _mm_pause loop while busy
d) Run in SwitchToTask loop while busy
e) Run in Sleep(0) loop while busy

SpinWait = 1:n Task steal when busy uses a + count indicating the number of times an attampt at task steal fails prior to suspending the thread. This is similar to, but more flexible than, the OpenMP BusyWait.

SpinWait = 0 Suspend thread without task stealing
SpinWait = -1 Run in _mm_pause loop while busy
SpinWait = -2 Run in SwitchToTask loop while busy
SpinWait = -3 Run in Sleep(0) loop while busy

Note, at thread pool startup, for non-master threads, each has a root level qtControl object that is set to busy until shutdown. These root level qtControl objects cannot be set with SpinWait < 1. To do so would inhibit them from performing work. Once in a stolen task the SpinWait can be adjusted to defaults or according to specific requirememts.

```
qtControl();
qtControl(qtControl& rParent);
qtControl(qtControl* pParent);
~qtControl() { WaitTillDone(); };
```

The ctors of qtControl creates a wiped qtControl structure. You may construct a qtControl object with a reference to and address of a parane qtControl object.

The dtor of qtControl performs an implicit WaitTillDone function. i.e. when the qtControl goes out of scope, or is otherwise deleted.

```
void WaitTillDone();
```

The code waits in task stealing mode if there are any pending requests or completion requests that had been queued via the qtControl structure.

```
void SuggestAffinity();
```

The SuggestAffinity function is used prepare the qtControl structure to enqueue work to the least loaded processor(s).

```
void SuggestAffinity(int_Native Charge);
```

The SuggestAffinity function is used prepare the qtControl structure to enqueue work to the least loaded processor(s). Additionally the Charge is added to the Affinity Charge total for the thread.

```
void SuggestAffinity(qtPlacement Placement);
```

Suggest affinities using the Placement restrictions.

```
void SuggestAffinity(qtPlacement Placement, int_Native Charge);
```

Suggest affinities using the Placement restrictions. Additionally the Charge is added to the Affinity Charge total for the thread.

```
void SuggestAffinityForAllocation();
```

The SuggestAffinityForAllocation is similar to SuggestAffinity except that the first run of the next task en-queued via the qtControl object will only be run on the affinity suggested.

```
void SuggestAffinityForAllocation(int_Native Charge);
```

The SuggestAffinityForAllocation is similar to SuggestAffinity except that the first run of the next task en-queued via the qtControl object will only be run on the affinity suggested. . Additionally the Charge is added to the Affinity Charge total for the thread.

```
void SuggestAffinityForAllocation(qtPlacement Placement);
```

The SuggestAffinityForAllocation uses the Placement restrictions.

```
void SuggestAffinityForAllocation(qtPlacement Placement, int_Native Charge);
```

The SuggestAffinityForAllocation uses the Placement restrictions. Additionally the Charge is added to the Affinity Charge total for the thread.

```
void ChargeAffinity(int_Native Charge);
```

Use ChargeAffinity to add charge points to your thread's affinity charge.

```
uint_Native SelectAffinities(qtPlacement Placement)
{
      ModeFields.Placement = Placement;
      return Affinity.SelectAffinities(Placement);
};
```

Use SelectAffinities to perform both the initialization of the ModeFields placement member and to condition the Affinity for the desired placement

The various CountOfWaitingThreads are used by the parallel_for templates to aid in scheduling decisions.

```
// CountOfWaitingThreads();
// System wide count of waiting threads
// Sets AffinityQueue/AffinityDeQueue to 0
// (non-affinity queue, any thread dequeue)
int_Native CountOfWaitingThreads();


// Leaves AffinityQueue/AffinityDeQueue as is
// counts waiting threads at AffinityDeQueue
int_Native CountOfAffinityDeQueueWaitingThreads();

// CountOfWaitingThreads(int iCacheLevel);
// Count of waiting threads withing the specified
// cache level of the program issueing the call.
// Cache level 0 is the running thread so you will
// expect to receive 0 always.
// Cache level 1 is the running thread however
// on a system with HT the L1 cache is shared
```

```
// with the cores HT sibling(s)
// Cache level 2 is typically available on all
// processors.
// Cache level 3 may or may not be present.
// On systems without L3 cache, the count of waiting
// threads will be 0
// Cache Level 4 represents the RAM within the system
// or within the NUMA node of the processor executing
// the call.
// Cache level 5 will be one level out NUMA node
// Cache level 6 will be two levels out NUMA node
// Cache level 7 will be three levels out NUMA node
//
// In all forms of calls when the count reurns > 0
// the AffinityQueue and AffinityDeQueue of the
// of the control object are setup such that
// subsequent enqueueing on that control object
// will go the specified cache level
int_Native CountOfWaitingThreads(int iCacheLevel);

// CountOfWaitingThreads(int iCacheLevel, int iAPICid);
// When iAPICid >= 0 the iCacheLevel is first used to
// determine a mask to apply to the iAPICid. The masked
// iAPICid is then used to identify one or more
// threads, cores, processors, sockets for counting
// of available threads at that cache level.
// Example, on a Q6600 quad core withou HT where
// two cores share one L2 and the other two cores share
// the second L2 then the APICid mask is 11111110.
//
// When iAPICid < 0 it is an alternate method
// -1 = Current core and one cache level up if available.
// -2 = Current core and nearest cache level neighbor.
// -3 = First selection w1th most available cores at
//      specified level
int_Native CountOfWaitingThreads(int iCacheLevel, int iAPICid);

int_Native SetDefaultSpinWait(int_Native newSpinWait);
```

The programmer may call these member functions to obtain the number of worker threads and number of I/O threads.

```
int_Native nWorkerThreads();
int_Native nIoThreads();
```

To determine if the current thread is in the run set of the affinities of the qtControl

```
// returns non-zero when current thread in runset
int_Native inRunSet();
```

Functions to retrieve the size in bytes of the running thread’s Cache or memory level.

```
int_Native CacheLevelSize(int iCacheLevel);
int_Native CacheLevelLineSize(int iCacheLevel);
int_Native CLFLUSHCacheLineSize();
```

Note, these were the values returned at time of thread initialization using CPUID. L0 generally does not report a size. For processors without an L3 the value returned will be 0. This can be a method for you to determine if the L3 is present. Most systems will have all similar processors, but this may change. If a memory level is returned, this would reflect the physical memory available to that processor which is not necessarily the virtual amount of memory available to the thread. When affinity is not used (or for I/O threads) the thhreads may move about so if the processors have different cache schemes, the floating threads may not hold current values.

The `void Queue...` functions are generally called from the templates and not directly form the user application.

```
void QueueWork(QuickThreadCallNode_v2* cn_v2);
void QueueWork(QuickThreadCallStack& CallStack);
void QueueWork(&fn[,a1[,a2[,a3[,a4[,a5[,a6[,a7[,a8[,a9]]]]]]]]]);
```

QueueWork is use to enqueue a compute class task. You may issue the call using a CallNode, CallStack object or by supplying a function and up to 9 arguments. The function (task) called has the following format:

```
void fn(T1 a1[, T2 a2[, T3 a3[,a4. . .]]]);
```

The function argument list is to match those used of the function en-queued.

```
void QueueIO(QuickThreadCallStack& CallStack);
void QueueIO(&fn[,a1[,a2[,a3[,a4[,a5[,a6[,a7[,a8[,a9]]]]]]]]]);
```

QueueIO is use to enqueue an I/O class task. You may issue the call using a CallStack object or by supplying a function and up to 9 arguments.

```
void QueueOnDone(QuickThreadCallStack& CallStack);
void QueueOnDone(&fn[,a1[,a2[,a3[,a4[,a5[,a6[,a7[,a8[,a9]]]]]]]]]);
```

QueueOnDone is use to enqueue compute class completion requests. You may issue the call using a CallStack object or by supplying a function and up to 9 arguments

```
void QueueIOOnDone(QuickThreadCallStack& CallStack);
void QueueIOOnDone(&fn[,a1[,a2[,a3[,a4[,a5[,a6[,a7[,a8[,a9]]]]]]]]]);
```

QueueIOOnDone is use to enqueue I/O class completion requests. You may issue the call using a CallStack object or by supplying a function and up to 9 arguments.

```
void QueueDo(QuickThreadCallStackDo& CallStack);
void QueueDo(&fn,iFrom,iTo[,a3[,a4[,a5[,a6[,a7[,a8[,a9]]]]]]]);
```

QueueDo is used to enqueue a closed range (Fortran style) do loop. The function and inclusive from and to range values are required arguments, the remaining arguments are optional. Note, the QueueDo (when not using HalfOpenRange) permits do loops to run either forward or backwards. When the programmer desires to specify the C++ half open style, they must indicate this by setting the `HalfOpenRange` mode indicator and for loops only run forward. See parallel_for for examples of use for C++. Subject to restrictions within the qtControl structure the range of the QueueDo will be divided into as many groups as there are threads available in the selected group. When no restrictions are specified then the number of groups is the number of compute class threads. The placement option (described elsewhere) may specify differing subsets of all available threads (e.g. those sharing L3 with current thread).

```
void QueueDoChunk(QuickThreadCallStackDoChunk& CallStack);
void QueueDoChunk(iChunk,&fn,iFrom,iTo[,a3[,a4[,a5[,a6[,a7[,a8[,a9]]]]]]]);
```

QueueDoChunk divides the range into Chunk sized pieces.

```
void QueueDoChunkTemporal(QuickThreadCallStackDoChunk& CallStack);
void QueueDoChunkTemporal(iChunk,&fn,iFrom,iTo[,a3[,a4[,a5[,a6[,a7[,a8[,a9]]]]]]]);
```

QueueDoChunkTemporal is equivilent to QueueDoChunk with the exception that each chunk is queued sequentially as opposed to in parallel. At the end of processing of each chunk, the programmer can code for a QueueWork using the chunk parameters to a secondary function. Using this technique it becomes relatively easy to construct a pipeline. Example:

```
// global scope
qtControl qtcPipe1, qtcPipe2, qtcPipe3;

...
qtcPipe1.QueueDoChunkTemporal(iChunk,Pipe1,iFrom,iTo,a3,a4);

qtcPipe1.WaitTillDone();
qtcPipe2.WaitTillDone();
qtcPipe3.WaitTillDone();
...
void Pipe1(intptr_t iFrom, intptr_t iTo, double* a3, double* a4)
{
	// **  QueueDo... uses Fortran style closed range
	for(intptr i=iFrom; i<=iTo;++i)
	{
		...
	}
	qtcPipe2.QueueWork(Pipe2,iFrom,iTo,a3,a4);
}
void Pipe2(intptr_t iFrom, intptr_t iTo, double* a3, double* a4)
{
	// **  QueueDo... uses Fortran style closed range
	for(intptr i=iFrom; i<=iTo;++i)
	{
		...
	}
	qtcPipe3.QueueWork(Pipe3,iFrom,iTo,a3,a4);
}
void Pipe3(intptr_t iFrom, intptr_t iTo, double* a3, double* a4)
{
	// **  QueueDo... uses Fortran style closed range
	for(intptr i=iFrom; i<=iTo;++i)
	{
		...
	}
}
```

Miscelleaneous Library Functions

ChargeAffinity

```
void ChargeAffinity(uint_Native AffinityMask, int_Native Charge);
```

Places an affinity charge onto the threads specified in the AffinityMask. The charge is a weighted value of your design.

Lock_FIFO

```
// The QuickThread Lock_FIFO is intended for use in
// situations where lock is to be held for short durations
// and where the access to the Lock is on FIFO basis.
// If the lock is held by a different thread for longer than
// a relatively short duration a SwitchToThread() is performed.
// If after SwitchToThread() the lock is still held by a different
// thread then task stealing occures until lock is owned.
// Note, this technique does not suspend the thread while attempting
// to obtain the lock. If no additional tasks are available to
// run the thread will be compute bound alternately testing for
// ownership of lock and testing for other task to run (i.e. the
// thread enters a non-productive compute loop).
// If you require longer term locks consider using critical sections.
//
// The Lock structure must persist for the duration of use.
__declspec( align(_QT_CACH_LINE_SIZE)) struct Lock_FIFO
{
      volatile int_Native LastLock;
      volatile int_Native OwnerOfLock;
      volatile int_Native TransferLock;
      volatile int_Native RecursionLevel;
      Lock_FIFO() { LastLock =
                        OwnerOfLock =
                              TransferLock =
                                    RecursionLevel = 0; };
      ~Lock_FIFO(){
#ifdef _ASSERT
      _ASSERT(LastLock==NULL);
#endif
  };
  void Acquire();
  void Release();
};
```

LockLock

```
// An SEH safe wrapper to Acquire and Release a Lock
struct LockLock
{
      Lock_FIFO* pLock_FIFO;
      // NULL ctor - use DefaultLock_FIFO, and Acquire
      LockLock()
      { pLock_FIFO = &DefaultLock_FIFO; pLock_FIFO->Acquire(); }
```

```
        // Reference to Lock_FIFO ctor
        // Use specified Lock_FIFO, and Acquire
        LockLock(Lock_FIFO& someLock)
        { pLock_FIFO = &someLock; pLock_FIFO->Acquire(); }
        // dtor performs Release() on Lock
        ~LockLock() { pLock_FIFO->Release(); }
};
```

qt_pointerLock

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

The qt_pointerLock is intended for short term locking of pointers to objects. Code referencing such pointers must be aware of the possibility of a pointer being in the locked state (pointing to location 1). And if so, defer use of pointer until it becomes unlocked.

The qt_pointerLock is not FIFO (fair). The first thread to obtain and lock the pointer (or recently released pointer) is the thread that holds the lock.

AtomicAdd

Functions to perform atomic add of float and double variables.

```
float AtomicAdd(float* pf, float v);
double AtomicAdd(double* pd, double d);
```

get_qtControl

```
qtControl* get_qtControl();
```

Function to obtain the threads current default qtControl object. Note, this object changes as you nest in/out of nested task levels. Do not assume the default qtControl object is persistent throught the task.

qtYield

```
void qtYield();
```

Use qtYield to perform one instance of task stealing. Generally you should code such that your tasks do not block/wait in a compute loop. When you cannot avoid this your choices are to

a) Burn CPU cycles
b) Issue _mm_pause()
c) Issue SwitchToThread()
d) Issue Sleep(nnn)
e) issue qtYield()

There are pros and cons to each method.

qt_get_num_threads

```
int_Native qt_get_num_threads();    // worker thread pool size
```

This is a non-member function that you can use to obtain the number of worker threads.

qt_get_num_io_threads

```
int_Native qt_get_num_io_threads(); // io thread pool size
```

This is a non-member function that you can use to obtain the number of I/O threads.

qt_get_thread_num

```
int_Native qt_get_thread_num();
```

Returns a 0-based QuickThread thread number. Compute class threads are numbered 0 to number of worker threads -1 and I/O class threads follow. Undefined result returned when call is made by thread that is not one of the QuickThread threads.

Note, the number returned is the internal QuickThread thread number which is not necessarily the the thread team member number resulting from parallel_distribute.

qt_get_thread_ID

```
int_Native qt_get_thread_ID();
```

Returns an operating system identifier for the thread. On Windows, this is the handle for the thread.

qt_get_thread_AffinityMask

```
int_Native qt_get_thread_AffinityMask();
```

Gets the current threads affinity mask (assuming there is one).

qt_index

```
size_t      qt_index(size_t size);
```

Used to produce the QuickThread scalable memory allocator index for allocations of size number of bytes. The current value is

```
      ((size-1) / sizeof(void*))
```

however, do not rely on the index being derrived from this formula. Future revisions of QuickThread may alter the relationship of the size to index.

```
// qt fast malloc/calloc/free
// these are index based allocations as opposed to byte allocations
// where index = (size-1) / size;
```

```
// 0 = size in range of 1:sizeof(void*)
// 1 = size in range of sizeof(void*)+1:2*sizeof(void*)
// ...
void* qt_malloc_index(size_t index);
void* qt_calloc_index(size_t nitems, size_t index);
void  qt_free_index(void* block, size_t index);
```

Fortran Programming

QuickThread can be used in Fortran as well as in mixed language programming. The Fortran API to QuickThread makes use of Generic Interfaces and as of this writing will require a bit more work for the programmer by requiring them to properly declare the subroutine interfaces to their application as used by the calls to the QuickThread library. The C++ programmers avoid this procedure because of the template capability of C++. The programming team of QuickThread is intending to extend the Fortran PreProcessor to provide an equivalent functionality to the C++ template capability. This feature enhancement, when available, will greatly reduce the amount of effort in introducing QuickThread into your Fortran applications. Until then, you will be required to do a little bit of extra work in copying your subroutine interface blocks into the generic interfaces to access the QuickThread library.

Program Initialization

The recommended technique for QuickThread initialization is by way of a customizable template subroutine titled YourQueueMainTemplate.f90 and distributed with QuickThread. It is suggested you copy this subroutine to you application project and rename it as QueueMain.f90. The conversion process is relatively simple. You convert your current “program” into a subroutine of different name, then create a new “program” of the original name of application. The new program is what is called a shell program. The primary function of the shell program is to call the customizable initialization routine QueueMain, passing in your program’s replacement subroutine name. Example:

Before modification:

```
program YourProgram
    use MyTypes
    use MyInterfaces
    ProgramName = 'YourProgram'
    call Init
    call DoWork
    call Finish
end program YourProgram
```

After modification

```
program YourProgram
    use MyTypes
    use MyInterfaces
    integer :: Status
    Status = QueueMain(YourProgramAsSubroutine)
end program YourProgram

subroutine YourProgramAsSubroutine
    use MyTypes
    use MyInterfaces
    implicit none
    ReportName = 'YourProgram'
    call DoInit
    call DoWork
    call Finish
end subroutine YourProgramAsSubroutine
```

QueueMain(yourApplicationAsSubroutine)

The functional requirements of the customizable template subroutine are to specify the threading and tunable parameters for QuickThread and to start the application.

```
! QueueMain.f90
!
! This is an interface between your application and QuickThread
!
! This function performs 2 things
!
! 1) Initialize QuickThread
! 2) Starts the application
!
! Initialization is performed by the function
! call QuickThreadInit(qtInit)
! Returns .true. on success, .false. on failure
!
! QuickThreadInit requires a T_qtInit type structure (qtInit)
! The fields and default initialization of the T_qtInit structure are:

! T_qtInit
! Initialization structure passed to QuickThreadInit
! Must be initialized by the user
!
! type T_qtInit
!    sequence
    ! nWorkerThreads = -1;
      !
    ! The number of compute intensive threads.
    !
    ! Most applications run best when the number of Worker Threads
    ! is equal to the number of cores available to the application.
    ! Using a larger number of Worker Threads than the number of
    ! cores available to the application generaly introduces extra
    ! operating system overhead to perform thread context switching.
    !
    ! If you are unable to export program I/O to an IoThread then the
    ! use of additional Worker Threads may be warranted.
    !
    !   -2 = Set number of worker threads to number of available cores
      !             excluding HT companion thread(s)
    !   -1 = Set number of worker threads to number of available cores
    !    0 = No worker threads
    !   +n = Number of worker threads the application desires
    !
!    integer(INT_PTR) ::  nWorkerThreads = -2

    ! nIoThreads = -1;
    !
    ! Number I/O threads
    !
    ! -1 = max(1, (Number of available processors - nWorkerThreads))
    ! 0 =  No I/O threads
    ! +n = Number of I/O threads
```

```
!    integer(INT_PTR) ::  nIoThreads = -1

    ! QueueSize = 0 ! 0 = Use default QueueSize (512)
    !
    ! When the queue size is too small, such as
    ! less than the number of cores available to the application,
    ! then the worker threads may become starved for work.
    ! When the queue size is too large, then the object latency may
    ! become too large. The object latency is the time interval
      ! between queueing the work for an object and when work begins.
    !
    ! 0 = Use default QueueSize (512)
    !
!    integer(INT_PTR) :: QueueSize = 0

    ! nAffinityGrouping = 1
    !
    ! nAffinityGrouping is the cache level granularity
    !
    ! 0 = Thread Processor Affinity not used
    ! 1 = L1 granularity (generally 1 core per L1)
    ! 2 = L2 granularity (often 2 cores per L2)
    ! 3 = L3 granularity (often 4, 6, or 8 cores per L3)
      !           (diminishes to cores per L2 when L3 not present)
    ! 4 = Cores per package
    ! ...
    !
      !
      ! The most effective grouping depends on the application
      ! as well as the archetecture of the system.
      ! Using Affinity introduces additional overhead in the queue
      ! and dequeue of work nodes in the system.
      ! It is best to experiment with nAffinityGrouping to
    ! find the best setting for the application on a given system.
    !
    ! The recommended order for experimentation:
    !
    !   nAffinityGrouping = 0;      ! Base level (affinity off)
    !   nAffinityGrouping = 1;      ! Individual core level
    !   nAffinityGrouping = 2;      ! # cores sharing L2
    !   nAffinityGrouping = 3;      ! # cores sharing L3
    !   nAffinityGrouping = 4;      ! # cores per package
    !
!    integer(INT_PTR) :: nAffinityGrouping = 1

    ! Status = 0
    !
    ! Initialization status
    !
    ! 0 = Success
    ! 1 = Initialized more than once
    ! 2 = nWorkerThreads is invalid
    ! 3 = nIoThreads is invalid
    ! 4 = No Threads requested (nWorkerThreads + nIoThreads equals 0)
    ! 5 = QueueSize is invalid
    ! 6 = nAffinityGrouping is invalid
!    integer(INT_PTR) :: Status = 0
```

```
!     int   QueueMain(FnMain* MainCode, PVOID context);
! end type T_qtInit

! integer :: Status = 0
!
! Initialization status
!
!   0 = Success
!   1 = Initialized more than once
!   2 = nWorkerThreads is invalid
!   3 = nIoThreads is invalid
!   4 = No Threads requested (nWorkerThreads + nIoThreads equals 0)
!   5 = QueueSize is invalid
!   6 = nAffinityGrouping is invalid

integer(DWORD) function QueueMain(MainCode)
    use kernel32
    use QuickThreadInterfaces
    use MyCommon
    use MyTypes
    implicit none
    external :: MainCode
    type(T_qtInit) :: qtInit


    ! code
    qtInit.nWorkerThreads = -1  ! use number of available processors

    ! Specify Affinity usag.
    qtInit.nAffinityGrouping = 1

    ! Do once-only initialization for QuickThread
    if(.not.QuickThreadInit(qtInit)) then
        write(*,*) 'QuickThreadInit failure = ', qtInit.Status
        QueueMain = qtInit.Status
        return
    endif

    ! Queue MainCode
    !
    QueueMain = QuickThreadQueueMain(MainCode)
    if(QueueMain .ne. 0) then
        write(*,*) 'QuickThreadQueueMain failure = ', qtInit.Status
        return
    endif
end function QueueMain
```

QuickThread Interfaces

QuickThreadInit(qtInit)

QuickThreadInit is a function is to be used once by the initialization function QueueMain. The calling argument qtInit is an initialization structure of type `T_qtInit`. This function returns .true. for success and .false. or failure.

QuickThreadQueueMain(MainCode)

QuickThreadQueueMain is a function is to be used by the initialization function QueueMain. The purpose of this function is to start your main code after initialization parameters have been specified. The return code of this function is whatever status may have been passed back by your application via QuickThread. 0 indicates no error.

T_qtControl

One of the key components of QuickThread is the use of control structures. The use of the control structure is flexible. A control structure may be used for process sequencing, object sequencing, object tree declarations, affinity binding, completion routine queuing and status information.

The persistence requirements of control structures are such that the memory allocated for the control structure must remain valid for the duration of all pending operations on that control structure. You may elect to create control structures in stack local storage, but if you do so, you must also use the Wait Until Done method of programming. It is more efficient to place the control structures in, or adjacent to, your objects and then use the Throw And Go method of programming. This is less stack-intensive. If you wish to postpone the decision of using control structures on a per-object basis then you can create one, or a few, for use on a by-process basis.

If your preference is to not insert a foreign looking control structure into your objects then the control structures for the objects can be placed outside the objects as long as your code knows how to locate the control structure for the object. Example: Object(n) uses qtObjectControlStructure(n)

The contents of the control structure may vary as QuickThread evolves. The current type definition of the control structure is as follows:

```
type T_qtControl
    sequence
    union
        map
              ! When we are a child qtControl
              ! this points to our parent qtControl
            type(T_qtControl), pointer :: Parent
        end map
        map
            integer(PVOID) :: Parent_p_void = 0
        end map
    end union
      ! Count is incremented at the beginning of
      ! the insertion. Therefore the queue may
      ! appear empty while Count > 0
      integer(INT_PTR) :: Count = 0
      type(T_QuickThreadAffinityQueueDeQueue) :: Affinities
```

```
  ! Queue On Done FIFO list
  integer(PVOID) :: QueueOnDoneListHead = 0
  integer(PVOID) :: QueueOnDoneListTail = 0
  integer(INT_PTR) :: Status = 0
  integer(INT_PTR) :: Mode = 0
  ! unsigned WaitingThread : 1;       // used by WaitTillDone();
  ! unsigned HalfOpenRange : 1;       // used by parallel_for
  ! unsigned CallFirstIteration : 1;  // used by QueueDo
  ! unsigned CallLastIteration : 1;   // used by QueueDo
  ! unsigned reserved : 4;
  ! qtPlacement Placement : 8;
  ! qtControl();
  ! ~qtControl() { WaitTillDone(); };
  ! void WaitTillDone();
  ! uintptr_t SuggestAffinity();
  ! uintptr_t SuggestAffinityForAllocation();
!
  ! CountOfWaitingThreads();
  ! System wide count of waiting threads
  ! Sets AffinityQueue/AffinityDeQueue to 0
  ! (non-affinity queue, any thread dequeue)
  ! intptr_t CountOfWaitingThreads();
!
  ! Leaves AffinityQueue/AffinityDeQueue as is
  ! counts waiting threads at AffinityDeQueue
  ! intptr_t CountOfAffinityDeQueueWaitingThreads();
!
  ! CountOfWaitingThreads(int iCacheLevel);
  ! Count of waiting threads withing the specified
  ! cache level of the program issueing the call.
  ! Cache level 0 is the running thread so you will
  ! expect to receive 0 always.
  ! Cache level 1 is the running thread however
  ! on a system with HT the L1 cache is shared
  ! with the cores HT sibling(s)
  ! Cache level 2 is typically available on all
  ! processors.
  ! Cache level 3 may or may not be present.
  ! On systems without L3 cache, the count of waiting
  ! threads will be 0
  ! Cache Level 4 represents the RAM within the system
  ! or within the NUMA node of the processor executing
  ! the call.
  ! Cache level 5 will be one level out NUMA node
  ! Cache level 6 will be two levels out NUMA node
  ! Cache level 7 will be three levels out NUMA node
  !
  ! In all forms of calls when the count reurns > 0
  ! the AffinityQueue and AffinityDeQueue of the
  ! of the control object are setup such that
  ! subsequent enqueueing on that control object
  ! will go the specified cache level
  ! intptr_t CountOfWaitingThreads(int iCacheLevel);
  !
  ! CountOfWaitingThreads(int iCacheLevel, int iAPICid);
  ! When iAPICid >= 0 the iCacheLevel is first used to
  ! determine a mask to apply to the iAPICid. The masked
```

```
      ! iAPICid is then used to identify one or more
      ! threads, cores, processors, sockets for counting
      ! of available threads at that cache level.
      ! Example, on a Q6600 quad core withou HT where
      ! two cores share one L2 and the other two cores share
      ! the second L2 then the APICid mask is 11111110.
      !
      ! When iAPICid < 0 it is an alternate method
      ! -1 = Current core and one cache level up if available.
      ! -2 = Current core and nearest cache level neighbor.
      ! -3 = First selection w1th most available cores at
      !      specified level
      ! intptr_t CountOfWaitingThreads(int iCacheLevel, int iAPICid);
      ! intptr_t nWorkerThreads();
      ! intptr_t nIoThreads();
end type T_qtControl
```

Parent is an optional pointer to an upper level control structure. NULL indicates there is no dependency on this control structure. When the Parent pointer is not NULL then it points to a control structure that is dependent on this control structure. The control structures are intended to be linked into and inverted dependency tree.

For example, if you have an Object with multiple SubComponents then you code the dependencies by inserting into the Parent pointer of each SubComponent control structure the address of the control structure of the Object. Likewise, you are free to create a control structure for the list of Objects and then point the Parent pointer of each Object control structure at the list of objects control structure. In this manner you can construct trees for effective thread processing on: Lists of Objects, Objects, SubObjects, SubSubObjects, ….

You can also elect to use control objects to construct process (code) dependency trees. It is simply a matter of interpretation.

Count is the count of pending work requests. 0 means no work pending. The Count is incremented upon queuing work via this control structure as well as incremented when any control structure, which this control structure is dependent on, has work requests queued via the dependency control structure. Count is decremented as work completes for this control structure and when the queue dependency work requests are completed. When Count reaches 0 then work is complete. Generally your application need not monitor the Count field.

Status may be used by your application to pass abnormal termination status codes back up the control structure tree.

Affinities – Contains **AffinityQueue** and **AffinityDeQueue**

Affinities.AffinityQueue – When processor affinity option is selected during initialization, then during subsequent application use you may place into the AffinityQueue field a bit mask identifying the processor queues in which to queue work items when queuing via this control structure. Helper functions are supplied with QuickThread to provide a suggested AffinityQueue bit mask.

Affinities.AffinityDeQueue – Place into AffinityDeQueue a bit mask of a list of processors permitted to receive the queued work items..

QueueOnDoneListHead, QueueOnDonListTail – This is a FIFO list of work requests (which is not to be used by the customer). When a completion work node is queued via a control structure the completion code work node is linked into these locations.

Status – Completion status can be inserted here.

Mode – Various flags and bit field are set by support routines. Users will generally not directly set these fields.

QuickThreadWaitTillDone(qtControl)

The QuickThreadWaitTillDone subroutine is passed a reference to a control structure. The Count of the control structured is examined to see if work is pending (+n) or has completed (0). When pending, an additional work request is processed by the current thread by proxy of some other work request. When this intervening work request completes the control structure is re-examined. If complete, QuickThreadWaitTillDone returns, if not an additional work request is performed. This repeats until done condition.

Using QuickThreadWaitTillDone introduces a stacking effect into the application. Depending on the queuing nature of the application you may experience a stack overflow problem. The stack overflow problem can be mitigated in one of three ways:

Increase the stack size
Specify a QueueSize in the initialization structure for the call to QuickThreadInit(qtInit)
Use queuing of completion routines

The use of completion routines is recommended.

QuickThreadSuggestAffinity(qtControl, qtPlacement, Charge)

This function is used to obtain a suggestion for processor affinity bit masks for AffinityQueue and AffinityDequeue bit masks in the qtControl structure (modified by call). The qtPlacement argument is a T_qtPlacement value loaded with a placement constant. You have considerable control over thread scheduling using qtPlacement such as schedule to the thread that shares the L2 with the current thread or schedule for a socket that is not the socked of the current thread. The Charge field is an arbitrary weight value to asses thread(s) selected by the suggestion. The purpose of the charge is to evenly distribute work using anticipated load (charge) values. In a NUMA architecture system or when you want to try to distribute work based on memory allocation you could use the number of bytes of memory to be allocated by the suggested thread(s) as a charge amount. Or, if you know the computational requirements are significantly different you could choose a value based on an anticipation of work load. The use Affinity Charge is left to you.

QuickThreadChargeAffinity(Charge)

This subroutine is called with an integer value, of your preference, of what to bill the current thread's AffinityCharge in excess of that charged via QuickThreadSuggestAffinity. You are not required to use this subroutine because QuickThreadSuggestAffinity billed the thread of the suggested affinity charge points.

QuickThread_Initialized()

Logical function that can be used to determine if QuickThread is already initialized.

QuickThread_nWorkerThreads()

Integer function that returns the number of worker threads.

QuickThreadQueueWork(&
[qtPlacement, &]
[qtControl, &]
aSub[,args])

The QuickThreadQueueWork subroutine is used to *queue to computational class thread,* an application subroutine with optional arguments. The programmer may supply an optional placement directive and/or an optional QuickThread control structure.

Care should be taken as to passing arguments by reference or by value. If passing by reference (default) then the referenced item must persist between the time of the en-queue and through the time of the execution. Additionally, if pass by reference, the value of the referenced item must remain what you intend it to be between the time of the en-queue and through the time of execution.

You must declare interfaces (edit local copy of QuickThreadInterfaces.f90) to enforce argument passing rules. Note, the queued subroutine must be declared either by interface or by external. Generally you pass objects by reference and pass numeric arguments by values as these typically change while you are queuing up work requests. However, in the case of shared variables, you may pass variables by reference (e.g. a reduction variable).

Additional notes. When passing the reference to a variable to multiple concurrent threads care must be taken to perform atomic operations when the shared variables are written to. Failure to follow thread-safe programming practices may result in erroneous results.

The QuickThreadInterfaces.f90 distributed in the QuickThread\F90 should be copied to your application directory and then expanded to incorporate the interface declarations for the user subroutines and arguments. Failure to follow this recommendation may result in calling subroutines with incorrect arguments. Similar modifications to QuickThreadInterfaces.f90 should be made for the remainder en-queuing functions.

QuickThreadQueueIO(qtControl, aSub[,args])

The QuickThreadQueueIO subroutine is used to *queue to IO class threads* an application subroutine with optional arguments via a QuickThread control structure. Care should be taken as to passing arguments by reference (default) or by value. If passing by reference then the referenced item must persist between the time of the queue and the time of the execution. Additionally, if pass by reference, the value of the referenced item must remain what you intend it to be between the time of the queue and the time of execution. Feel free to declare interfaces to enforce argument passing rules. Note, the queued subroutine must be either declared by interface or declared as external. Generally you pass objects by reference and pass integer arguments by values as these typically change while you are queuing up work requests.

See note in QuickThreadQueueWork regarding adding subroutine interfaces to QuickThreadInterfaces.f90.

QuickThreadQueueOnDone(qtControl, aSub[,args])

The QuickThreadQueueOnDone subroutine is used to specify a *computational class* completion routine to be executed upon completion of work queued via the control structure and its dependencies. Call QuickThreadQueueOnDone after queuing work via the control structure. Calling prior to queuing work will result in immediate execution of the completion routine (as the Count in the control structure would indicate Done).

QuickThreadQueueIOOnDone(qtControl, aSub[,args])

The QuickThreadQueueIOOnDone subroutine is used to specify an *IO class* completion routine to be executed upon completion of work queued via the control structure or its dependencies. Call QuickThreadQueueIOOnDone after queuing work via the control structure.

QuickThreadQueueDo(&
& qtControl, aSub, iFrom, iTo[,args])

The QuickThreadQueueDo is used to distribute iterative work to the available computational class threads. The distributed sub ranges *may execute out of order*. The subroutine referenced must have the range variables as its first two arguments. The range variables supplied to QuickThreadQueueDo must specify the full range of interest. The range variables supplied to referenced subroutine (when called) will be a subset (or potentially the complete range) of the queued range. The range progression may be either ascending (iFrom .lt. iTo) or descending (iFrom .gt. iTo) or unitary (iFrom .eq. iTo). The queued range is divided by the number of computational threads available and queue requests are made accordingly. Due to the queued range not necessarily being evenly divisible by the number of computational threads each computational thread may receive different span of their sub range.

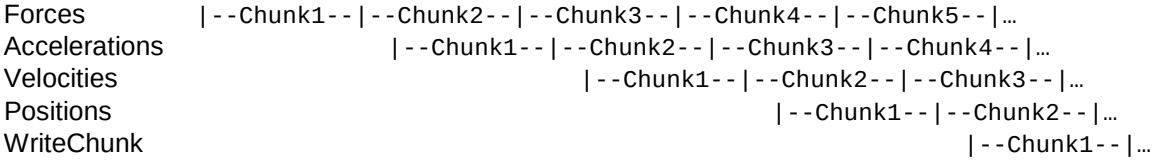
QuickThreadQueueDoChunk(&
& iChunkSize, pvControl, aSub, iFrom, iTo[,args])

The QuickThreadQueueDoChunk is used to distribute iterative work in iChunkSize groupings. The distributed sub ranges *may execute out of order*. The subroutine referenced must have the range variables as its first two arguments. The range variables supplied to QuickThreadQueueDoChunk must specify the full range of interest. The range variables supplied to referenced subroutine (when called) will be a subset (or potentially the complete range) of the queued range. The range progression may be either ascending (iFrom .lt. iTo) or descending (iFrom .gt. iTo) or unitary (iFrom .eq. iTo). The queued range is divided by the specified chunk size and queue requests are made accordingly. Due to the queued range not necessarily being evenly divisible by the chunk size each computational thread may receive different span of their sub range.

QuickThreadQueueDoChunkTemporal(&
& iChunkSize, pvControl, aSub, iFrom, iTo[,args])

The QuickThreadQueueDoChunkTemporal is used to distribute iterative work in iChunkSize groupings. The distributed sub ranges *will execute serially in order*. This method of queuing is useful in establishing a pipeline effect with data that have temporal computational requirements. The subroutine referenced in the QuickThreadQueueDoChunkTemporal is responsible for queuing the next phase of the pipeline (generally via QuickThreadQueueWork).

Consider a finite element analysis application with multiple objects where each object consists of a wireframe. Each wireframe may vary in complexity. Some wire frames may be large some may be small. For each integration step the temporal order of computation might be: Compute the forces, compute the accelerations, compute the velocities, and compute the positions. When the wireframe is large you might want to pipeline the operation:



Pipelines are useful in reducing latency times. In the above diagram, the writing of Chunk1 data could conceivably begin during the Forces computational phase of Chunk5 of the data set.

Examples

The examples in this document are portions of the example applications distributed with QuickThead. The example applications contain an initialization and test control program module that performs performance timed runs of three version of the problem. A timed session for: Single Threaded, OpenMP, and QuickThread. The example applications will have tuning parameters near the top of the source file. It is recommended that you make your first test runs with the distributed tuning parameters before you experiment with the values of the tuning parameters.

The purpose of the example applications is to provide you with an overview of the QuickThread techniques. In many of the examples the use of OpenMP parallel programming techniques or the use of a multi-threading library (e.g. MKL) might yield easier coding or superior performance than the QuickThread example.

QuickThread programming techniques requires more effort by the programmer, but repays that effort with faster run times.

SimpleArray

This example illustrates how you can perform an operation on three arrays. The operation is A=B+C for varying sized arrays.

Example of suggested changes to user object derived types
Before changes

```
module ApplicationModule

    ! sub-object derived type
    type T_SubObject
        integer :: id
        real(8), pointer :: Array(:)
    end type T_SubObject

    type T_Object
        integer :: id
        integer :: count
        type(T_SubObject) :: A, B, C
    end type T_Object

    ! define a pointer type to derived type
    type T_ObjectPointer
        type(T_Object), pointer :: p
    end type T_ObjectPointer

    ! declare static instance of a pointer to an array of pointers your
objects
    ! (not yet allocated)
    type(T_ObjectPointer), pointer :: ObjectArray(:)

end module MyTypes
```

```
subroutine DoObjects
    use QuickThread
    use MyCommon
    use MyTypes
    use MyInterfaces
    implicit none

    ! local variables
    integer :: ObjectNumber
    type(T_Object), pointer :: Object

    do ObjectNumber=1, NumberOfObjects
        Object => ObjectArray(ObjectNumber).p
        call QuickThreadQueueWork(ObjectArray_qtControl, DoObject, Object)
    end do
end subroutine DoObjectsFromTo
```

Bad example

```
subroutine Foo
    use QuickThread
    implicit none
    type(T_QuickThreadControlStructure) :: qtControl
    integer :: i
    external :: DoWork

    do i=1,100
        call QuickThreadQueueWork(qtControl, DoWork, i)
    end do
    call QuickThreadWaitTillDone(qtControl)
end subroutine Foo
```

Good example:

```
subroutine Foo
    use QuickThread
    implicit none
    type(T_QuickThreadControlStructure) :: qtControl
    integer :: i

    interface
    subroutine QuickThreadQueueWork(qtControl, aSub, index)
    use QuickThread
    type(T_QuickThreadControlStructure) :: qtControl
    external :: aSub
!DEC$ ATTRIBUTES VALUE :: index
    integer :: index
    end subroutine
    end interface

    external :: DoWork

    do i=1,100
        call QuickThreadQueueWork(qtControl, DoWork, i)
    end do
    call QuickThreadWaitTillDone(qtControl)
end subroutine Foo
```

The first example (bad) calls DoWork passing “I” by reference.

Example 2: OpenMP outer level parallelization

Same program as non-parallel version in example 1 except for the following highlighted changes to ProcessObjects:

```
! ProcessObjects.f90
subroutine ProcessObjects()
  use omp_lib
  use YourTypesAndInterfaces
  integer :: iObjectNumber
!$OMP PARALLEL DO SCHEDULE(DYNAMIC, 1)
  do iObjectNumber=1, NumberOfObjects
    Call ProcessObject(ObjectArray(iObjectNumber))
  end do
!$OMP END PARALLEL DO
end subroutine ProcessObjects()
```

Example 3: OpenMP inner level parallelization

Same program as non-parallel version in example 1 except for the following highlighted changes to ProcessObject:

```
subroutine ProcessObject(Object)
  use omp_lib
  use YourTypesAndInterfaces
  type(T_Object) :: Object
  integer :: i
!$OMP PARALLEL DO
  do i=1,Object.ArraySize
    Object.Velocity(i) = Object.Velocity(i) + (Object.Acceleration(i)*dT)
  end do
!$OMP END PARALLEL DO
end subroutine ProcessObject
```

Example of Pipeline

```
subroutine DoObject(Object)
    use ApplicationModule
    type(T_Object) :: Object    ! your object type
    ! nChunkElements = chunk size for elements (specified in your ApplicationModule)
    call QuickThreadDoChunkTemporal( &
      & nChunkElements, Object.qtControl, DoForces, 1, Object.NumberOfElements, Object)
end subroutine DoObject

subroutine DoForces(iFrom, iTo, Object)
    use ApplicationModule
!DEC$ ATTRIBUTES VALUE :: iFrom
    integer :: iFrom
!DEC$ ATTRIBUTES VALUE :: iTo
    integer :: iTo
    type(T_Object) :: Object    ! your object type
    integer :: i
    do i=iFrom, iTo
        Object.Force(i) = DoForce(Object, i)
    end do
    ! queue next phase in pipeline
    call QuickThreadQueueWork(Object.qtControl, DoAccelerations, iFrom, iTo, Object)
end subroutine DoForces

subroutine DoAccelerations(iFrom, iTo, Object)
    use ApplicationModule
!DEC$ ATTRIBUTES VALUE :: iFrom
    integer :: iFrom
!DEC$ ATTRIBUTES VALUE :: iTo
    integer :: iTo
    type(T_Object) :: Object    ! your object type
    integer :: i
    do i=iFrom, iTo
        Object.Acceleration(i) = Object.Force(i) / Object.Mass
    end do
    ! queue next phase in pipeline
    call QuickThreadQueueWork(Object.qtControl, DoVelocities, iFrom, iTo, Object)
end subroutine DoForces

subroutine DoVelocities(iFrom, iTo, Object)
    use ApplicationModule
!DEC$ ATTRIBUTES VALUE :: iFrom
    integer :: iFrom
!DEC$ ATTRIBUTES VALUE :: iTo
    integer :: iTo
    type(T_Object) :: Object    ! your object type
    integer :: i
    do i=iFrom, iTo
        Object.Velocity(i) = Object.Velocity(i) + Object.Acceleration(i) * DeltaT
    end do
    ! queue next phase in pipeline
    call QuickThreadQueueWork(Object.qtControl, DoPositions, iFrom, iTo, Object)
end subroutine DoForces

subroutine DoPositions(iFrom, iTo, Object)
    use ApplicationModule
!DEC$ ATTRIBUTES VALUE :: iFrom
    integer :: iFrom
!DEC$ ATTRIBUTES VALUE :: iTo
    integer :: iTo
    type(T_Object) :: Object    ! your object type
    integer :: i
    do i=iFrom, iTo
        Object.Position(i) = Object.Position(i) + Object.Velocity(i) * DeltaT
    end do
    ! end of pipeline, nothing else to queue
end subroutine DoForces
```

Example 4: OpenMP outer and inner level parallelization

Same program as non-parallel version in example 1 except for the following highlighted changes to ProcessObjects and ProcessObject:

```
! ProcessObjects.f90
subroutine ProcessObjects()
  use omp_lib
  use YourTypesAndInterfaces
  integer :: iObjectNumber

  ! Enable OpenMP nesting
  call OMP_SET_NESTED(.true.)

!$OMP PARALLEL DO SCHEDULE(DYNAMIC, 1)
  do iObjectNumber=1, NumberOfObjects
    Call ProcessObject(ObjectArray(iObjectNumber))
  end do
!$OMP END PARALLEL DO
end subroutine ProcessObjects()

subroutine ProcessObject(Object)
  use omp_lib
  use YourTypesAndInterfaces
  type(T_Object) :: Object
  integer :: i
!$OMP PARALLEL DO
  do i=1,Object.ArraySize
    Object.Velocity(i) = Object.Velocity(i) + (Object.Acceleration(i)*dT)
  end do
!$OMP END PARALLEL DO
end subroutine ProcessObject
```

As you can see from examples 2, 3 and 4 integrating OpenMP into the application is relatively easy and benign. The ease of integration was made with a compromise to absolute performance.

The following examples will illustrate the QuickThread methods of programming.

Example 5 QuickThread outer and inner level parallelization

```
! YourQuickThreadInterfaces.inc
! Fortran PreProcessor #include file
! Used to redefine interfaces in a manner that maintains
! a transparency to your source code
!
! Compile with _QuickThread defined to enable QuickThread
! Compile with _QuickThread undefined to disable QuickThread

#ifdef _QuickThread
! Enable QuickThread
#define QuickThreadModule use QuickThread
#define QuickThreadLocals type(T_QuickThreadNode) :: qtControlNode
#define QuickThreadInterface(name) \
  !DEC$ ATTRIBUTES ALIAS : '_QUICKTHREADQUEUEWORK' :: name
! Rename interfaces for subroutines modified for QuickThread
! ProcessObjects()
#define ProcessObjects ProcessObjects(qtControlNode)
! ProcessObject(Object)
#define ProcessObject(Object) qtProcessObject(qtControlNode, Object)
#else
! _QuickThread not defined
! Disable QuickThread by setting macros to void
! (not equivalent to undefined)
#define QuickThreadModule
#define QuickThreadLocals
#define QuickThreadInterface(name)
#endif
```

Remaining pages of manual have been omitted.