

The Sandia Qthreads Application Programming Interface User Guide

March 27, 2021

1 Introduction

The Qthreads Application Programming Interface (API) supports multithreaded programming using the lightweight threading model. The API supports the expression of task parallelism through the creation and synchronization of lightweight threads, called *qthreads*, as well as parallel loops that are implemented as qthreads. The qthreads are lightweight threads in that their stack space is minimal – containing only the minimum state required to execute the task – and context switching is inexpensive compared to pthreads and other heavyweight threads.

Qthreads are scheduled and executed within control structures called *shepherds*. Each shepherd comprises a queue of qthreads awaiting execution and one or more *workers*. Each worker is a long-lived heavyweight thread, pinned to a core, that executes qthreads from its shepherd's queue. While a qthreads program may generate thousands of qthreads, the number of worker threads is typically at most the number of cores or hardware threads in the system. The number of workers in each shepherd and the number of shepherds can be specified through environment variables and their configuration can be mapped to the underlying hardware topology as discovered by run time mechanisms such as the hwloc library. An example configuration would be one shepherd per socket in a two- or four-socket multicore system, with one worker per hardware thread. Another configuration would be one shepherd per core. At the extremes, a configuration could have one shepherd per hardware thread with only one worker per shepherd, or only a single shepherd comprising all the workers in the system.

2 Installation

First, we need to grab the repository: `git clone git://github.com/Qthreads/qthreads.git`

Next, let's change into the qthreads directory:

```
cd qthreads
```

Before compiling the Qthreads library, we need to setup some environment variables:

```
export QTHREAD_INC=<home-directory>/qt-install/include/  
export QTHREAD_LIB=<home-directory>/qt-install/lib/  
export LD_LIBRARY_PATH=<home-directory>/qt-install/lib/:$LD_LIBRARY_PATH
```

The `QTHREAD_INC` environment variable should point to the directory containing the Qthreads header files. The `QTHREAD_LIB` directory should point to the directory containing the Qthreads library. To run the resulting executable, ensure that the Qthreads library directory is in `LD_LIBRARY_PATH`.

Now, let's build the library:

```
./autogen.sh  
./configure --prefix<home-directory>/qthreads/qt-install  
make  
make install
```

If the above instructions worked, inside the qt-install directory you will have 3 directories: include, lib, and share.

To run the examples below, please change into the examples directory:

```
cd userguide/examples
```

3 Simple Qthreads Example Programs

3.1 Hello, world

The following code shows a simple “hello, world” example program using the qthreads API:

```
#include <stdio.h>
#include <qthread/qthread.h>

static aligned_t greeter(void *arg)
{
    printf("Hello, world!\n");

    return 0;
}

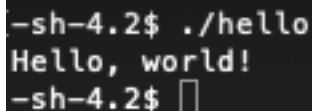
int main(int argc, char *argv[])
{
    aligned_t return_value = 0;

    qthread_initialize();
    qthread_fork(greeter, NULL, &return_value);
    qthread_readFF(NULL, &return_value);

    return return_value;
}
```

After initializing qthreads using the `qthread_initialize` API call, the program generates a single qthread through the `qthread_fork` API call. The first argument to the call is the function to be executed by the qthread. The second argument to the call is an argument of type `void*` to be passed to the specified function. The third argument to the call is a pointer of type `aligned_t*` to the location into which the return value of the function will be placed upon completion of the qthread. In the case of this program, the call is to the function called `greeter`, which does not use its argument and simply prints “Hello, world!”, then returns 0. The `qthread_readFF` API call waits for the return value produced by the completion of the qthread. Its two arguments are of type `aligned_t*`. The first argument specifies a destination address to store the value that is read (unused in this program), and the second specifies the source address for the value. As a library-based API, Qthreads does not require any special compiler support. The following is an example command to compile the program:

```
cc -o hello -I$QTHREAD_INC -L$QTHREAD_LIB -lqthread hello.world.c
./hello
```



```
-sh-4.2$ ./hello
Hello, world!
-sh-4.2$
```

3.2 A safer hello, world

In the following example, the original qthreads "hello, world" program is supplemented with some safety checking added:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <unistd.h>
#include <qthread/qthread.h>

static aligned_t greeter(void *arg)
{
    printf("Hello_World!\n");

    return 0;
}

int main(int argc, char *argv[])
{
    aligned_t return_value = 0;
    int status;

    status = qthread_initialize();
    assert(status == QTHREAD_SUCCESS);

    status = qthread_fork(greeter, NULL, &return_value);
    assert(status == QTHREAD_SUCCESS);

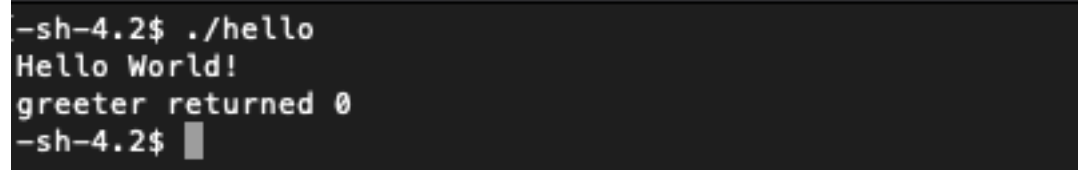
    status = qthread_readFF(NULL, &return_value);
    assert(return_value == QTHREAD_SUCCESS);

    printf("greeter returned %lu\n",
           (unsigned long) return_value);

    return EXIT_SUCCESS;
}
```

As shown above, each qthreads API call returns an integer indicating the success or failure of the call. In this version of the example, the value returned from the qthread execution is also printed.

```
cc -o hello -I$QTHREAD_INC -L$QTHREAD_LIB -lqthread hello.worldsafe.c
./hello
```



```
-sh-4.2$ ./hello
Hello World!
greeter returned 0
-sh-4.2$ █
```

3.3 Hello, World with multiple qthreads

Now consider a more interesting variation on the “hello, world” example. The following example generates ten qthreads, each of which prints the “hello, world” message:

```
#include <stdio.h>
#include <qthread/qthread.h>

static aligned_t greeter(void *arg)
{
    printf("#%lu:_Hello,_world!\n", (unsigned long) (uintptr_t) arg);

    return 0;
}

int main(int argc, char *argv[])
{
    aligned_t return_value[10] = { 0 };
    unsigned long i;

    qthread_initialize();

    for (i = 0; i < 10; i++)
        qthread_fork(greeter, (void *)i, &return_value[i]);

    for (i = 0; i < 10; i++)
        qthread_readFF(NULL, &return_value[i]);

    return 0;
}
```

In this example, each qthread prefaces its “hello, world” message with the value of its argument. Note that the messages may be printed in any order, since all the qthreads are generated before waiting on any of the return values. Calling `qthread_readFF` immediately after generating each qthread would guarantee that they execute in order, but would also serialize the execution.

```
cc -o hello -I$QTHREAD_INC -L$QTHREAD_LIB -lqthread hello.worldmultiple.c
./hello
```

```
-sh-4.2$ ./hello
#9: Hello, world!
#8: Hello, world!
#7: Hello, world!
#6: Hello, world!
#5: Hello, world!
#4: Hello, world!
#3: Hello, world!
#2: Hello, world!
#1: Hello, world!
#0: Hello, world!
-sh-4.2$
```

3.4 Hello, world with Qthreads parallel loop

In the previous example, the loop iterations generate qthreads. The Qthreads API provides a higher-level loop construct for this pattern, as shown in the following example:

```
#include <stdio.h>
#include <qthread/qloop.h>
#include <qthread/qthread.h>

static void greeter(const size_t startat,
                   const size_t stopat,
                   void *arg)
{
    size_t i;

    for (i = startat; i < stopat; i++)
        printf("#%lu:_Hello,_world!\n", i);
}

int main(int argc, char *argv[])
{
    qthread_initialize();

    qt_loop(0, 10, greeter, NULL);

    return 0;
}
```

The `qt_loop` API call takes four arguments. The first two arguments specify start and end of the loop bounds. The third and fourth arguments specify the function to be executed and the argument to be passed to it. The form of the function provided to `qt_loop` is different from the form of the function provided to `qthread_fork`: In addition to the `void*` argument, two arguments indicate the start and end of the set of loop iterations to be handled by the loop specified inside the function. The Qthreads run time system packages the loop iterations as qthreads and supplies the start and end loop bounds for each thread's instance of the `greeter` function as it is generated. The `qt_loop` call also waits for all the loop iterations to complete. The resulting output is the same as the previous example.

```
cc -o hello -I$QTHREAD_INC -L$QTHREAD_LIB -lqthread hello_world_loop.c
./hello
```

```
-sh-4.2$ ./hello
#0: Hello, world!
#1: Hello, world!
#2: Hello, world!
#3: Hello, world!
#4: Hello, world!
#5: Hello, world!
#6: Hello, world!
#7: Hello, world!
#8: Hello, world!
#9: Hello, world!
-sh-4.2$
```

3.5 Optimizing loop execution

If the amount of work performed in each iteration of a loop is uniform, more optimal performance typically may be achieved by dividing the iteration space into a number of qthreads equal to the number of available workers on the system, with each qthread containing an equal (or near equal) number of iterations. The `qt_loop_balance` API call offers this functionality. Its use is shown in the following example:

```
#include <stdio.h>
#include <qthread/qloop.h>
#include <qthread/qthread.h>

static void greeter(const size_t startat,
                   const size_t stopat, void *arg)
{
    size_t i;

    for (i = startat; i < stopat; i++)
        printf("#%lu: Hello, world!\n", i);
}

int main(int argc, char *argv[])
{
    qthread_initialize();

    qt_loop_balance(0, 10, greeter, NULL);

    return 0;
}
```

The only change to the code from the previous example is that the call has been changed from `qt_loop` to `qt_loop_balance`. The arguments to both calls are the same. The internal implementation of `qt_loop_balance` in the Qthreads run time system performs the division of the iteration space into qthreads based on the number of available workers. Thus, the same code may be executed on systems with different numbers of hardware threads, or on different numbers of threads on the same machine, without any changes to the code.

```
cc -o hello -I$QTHREAD_INC -L$QTHREAD_LIB -lqthread hello.world.loop.optimized.c
./hello
```

```
-sh-4.2$ ./hello
#0: Hello, world!
#1: Hello, world!
#2: Hello, world!
#3: Hello, world!
#4: Hello, world!
#5: Hello, world!
#6: Hello, world!
#7: Hello, world!
#8: Hello, world!
#9: Hello, world!
-sh-4.2$ █
```

4 Calculating a Fibonacci number

The following example illustrates naive brute-force calculation of the n -th Fibonacci number using recursive task parallelism. The program begins by generating a single qthread to execute the `fib` function with the argument indicating which Fibonacci number (n) to calculate. The `fib` function then calls itself recursively with arguments $n-1$ and $n-2$ until the base case of the recursion is satisfied.

```
cc -o fib -I$QTHREAD_INC -L$QTHREAD_LIB -lqthread fib.c
time ./fib
time ./fib 24
```

```
#include <stdio.h>
#include <stdlib.h>
#include <qthread/qthread.h>

static aligned_t fib(void *arg)
{
    aligned_t x = 0, y = 0;
    unsigned long n = (unsigned long) (uintptr_t) arg;

    if (n < 2) return n;

    qthread_fork(fib, (void *) n-1, &x);
    qthread_fork(fib, (void *) n-2, &y);

    qthread_readFF(NULL, &x);
    qthread_readFF(NULL, &y);

    return (x + y);
}

int main(int argc, char *argv[])
{
    aligned_t return_value = 0;
    unsigned long n = 0;

    if (argc > 1)
        n = atol(argv[1]);

    qthread_initialize();

    qthread_fork(fib, (void *) n, &return_value);

    qthread_readFF(NULL, &return_value);

    printf("fib(%ld)_=%ld\n", n, return_value);

    return 0;
}
```



```

-sh-4.2$ time ./fib
fib(0) = 0

real    0m0.003s
user    0m0.000s
sys     0m0.002s
-sh-4.2$ time ./fib 24
fib(24) = 46368

real    0m0.050s
user    0m0.048s
sys     0m0.000s
-sh-4.2$ █

```

4.1 A faster Fibonacci

The program in the previous example generates a single qthread to execute each addition in the Fibonacci calculation. Since the generation and execution of each qthread has an associated overhead cost, it is not recommended to perform so little computational work in each qthread. In the following example program, a sequential calculation is introduced below a certain threshold. Because far fewer qthreads are created to solve a given problem, time spent on overhead costs is greatly reduced compared to the previous example program.

```

cc -o fib -I$QTHREAD_INC -L$QTHREAD_LIB -lqthread fib-fast.c
time ./fib
time ./fib 24

```

```

-sh-4.2$ time ./fib
fib(0) = 0

real    0m0.003s
user    0m0.000s
sys     0m0.002s
-sh-4.2$ time ./fib 24
fib(24) = 46368

real    0m0.003s
user    0m0.000s
sys     0m0.003s
-sh-4.2$ █

```

```

#include <stdio.h>
#include <stdlib.h>
#include <qthread/qthread.h>

#define THRESHOLD 20

unsigned long fib_serial(unsigned long n)
{
    int x, y;

    if (n < 2) return n;

    x = fib_serial(n-1);
    y = fib_serial(n-2);

    return (x + y);
}

static aligned_t fib(void *arg)
{
    aligned_t x = 0, y = 0;
    unsigned long n = (unsigned long) (uintptr_t) arg;

    if (n < THRESHOLD)
        return (aligned_t) fib_serial(n);

    qthread_fork(fib, (void *) n-1, &x);
    qthread_fork(fib, (void *) n-2, &y);

    qthread_readFF(NULL, &x);
    qthread_readFF(NULL, &y);

    return (x + y);
}

int main(int argc, char *argv[])
{
    aligned_t return_value = 0;
    unsigned long n = 0;

    if (argc > 1)
        n = atol(argv[1]);

    qthread_initialize();

    qthread_fork(fib, (void *) n, &return_value);

    qthread_readFF(NULL, &return_value);

    printf("fib(%ld) = %ld\n", n, return_value);

    return 0;
}

```

4.2 Fibonacci by preconditioned tasks

Many algorithms can be structured as a set of qthreads partially ordered by dependence relationships. Qthreads represents dependences between qthreads in terms of *preconditions*. The following example program demonstrates their use.

```
#include <stdio.h>
#include <stdlib.h>
#include <qthread/qthread.h>

typedef struct {
    aligned_t n;
    aligned_t result;
} f_arg_t;

typedef struct {
    f_arg_t fargs[2];
    aligned_t *target;
} fr_arg_t;

static aligned_t fib_result(void *arg)
{
    aligned_t *target = ((fr_arg_t *)arg)->target;
    aligned_t r0      = ((fr_arg_t *)arg)->fargs[0].result;

    r0 += ((fr_arg_t *)arg)->fargs[1].result;

    qthread_writeEF(target, &r0);
    free(arg);

    return 0;
}

static aligned_t fib(void *arg)
{
    aligned_t n      = ((f_arg_t *)arg)->n;
    aligned_t *result = &((f_arg_t *)arg)->result;

    if (n < 2) {
        qthread_writeEF_const(result, n);
        return 0;
    }

    fr_arg_t *fibs = malloc(sizeof(fr_arg_t));
    f_arg_t *f1    = &fibs->fargs[0];
    f_arg_t *f2    = &fibs->fargs[1];

    f1->n = n - 1;
    f2->n = n - 2;
    qthread_empty(&f1->result);
    qthread_empty(&f2->result);

    fibs->target = result;

    // Collect results of sub-actions
```

```

    qthread_fork_precond(fib_result, fibs, NULL, 2,
                        &f1->result, &f2->result);

    // Fork off recursive actions
    qthread_fork(fib, f1, NULL);
    qthread_fork(fib, f2, NULL);

    return 0;
}

int main(int argc, char *argv[])
{
    unsigned long n = 0;
    f_arg_t args = { 0, 0 };

    if (argc > 1)
        n = atol(argv[1]);

    args.n = (aligned_t) n;

    qthread_initialize();

    qthread_empty(&args.result);

    qthread_fork(fib, &args, NULL);

    qthread_readFF(NULL, &args.result);

    printf("fib(%lu) = %lu\n", n, args.result);

    return 0;
}

```

The code first defines two structure types, `f_arg_t` and `fr_arg_t`. Pointers to structures of these types will be passed as arguments to the two functions `fib_result` and `fib`, respectively. We will examine these functions in reverse order. The arguments provided to the `fib` function specify the Fibonacci number to calculate (`n`) and a location where the result will be placed (`result`). These arguments are unpacked from the `arg` structure into local variables. Unless `n` is trivial, the remaining code in the function sets up and generates new `qthreads` to recursively call the `fib` function and to combine the results using the `fib_result` function. A new argument structure `fibs` is allocated for the call to `fib_result`, and that structure contains within it an array of two argument structures for calls to `fib`. Pointers to these two structures are saved, their input fields are set to `n-1` and `n-2`, and the result fields are set to empty. The second field of the `fibs` argument structure is the target location where the sum of the combined results will be placed. This is set to the `result` field of `fib`.

The `qthread_fork_precond` API call generates a `qthread` to execute the `fib_result` function only when its preconditions are fulfilled. The first three arguments to the call are the same as those of `qthread_fork`: the function, the pointer to the argument data to be passed to the function, and a pointer to the return value. This example program collects the results through pointers in the argument structures, and thus does not use the third argument. The fourth argument specifies the the number of preconditions that must be fulfilled before the run time will schedule for execution. The remaining arguments specify those preconditions, in this case, the completion of the two recursive calls to `fib`. The `qthread_fork_precond` call is followed by the two calls to `qthread_fork` that generate those recursive calls to `fib`. At this point,

the function returns. Thus the qthread executing `fib` at run time does not wait for the recursive calls to complete. Instead, the qthread generated to execute `fib_result` waits on the recursive calls, only beginning execution once they are finished.

Finally, we have the code for the `main` function of the program, which is similar to that of the previous program. The primary difference is that an argument structure is initialized for the first `fib` task. The `n` field takes the Fibonacci number to be calculated, as specified by the command line argument. The `result` field is initially zero, and at the completion of all the qthreads generated, it contains the final result.

```
cc -o fib -I$QTHREAD_INC -L$QTHREAD_LIB -lqthread fib_preconditioned.c
time ./fib
time ./fib 24
```

```

#include <stdio.h>
#include <stdlib.h>
#include <qthread/qthread.h>

typedef struct {
    aligned_t n;
    aligned_t result;
} f_arg_t;

typedef struct {
    f_arg_t    fargs[2];
    aligned_t *target;
} fr_arg_t;

static aligned_t fib_result(void *arg)
{
    aligned_t *target = ((fr_arg_t *)arg)->target;
    aligned_t  r0      = ((fr_arg_t *)arg)->fargs[0].result;

    r0 += ((fr_arg_t *)arg)->fargs[1].result;

    qthread_writeEF(target, &r0);
    free(arg);

    return 0;
}

static aligned_t fib(void *arg)
{
    aligned_t  n      = ((f_arg_t *)arg)->n;
    aligned_t *result = &((f_arg_t *)arg)->result;

    if (n < 2) {
        qthread_writeEF_const(result, n);
        return 0;
    }

    fr_arg_t *fibs = malloc(sizeof(fr_arg_t));
    f_arg_t  *f1    = &fibs->fargs[0];
    f_arg_t  *f2    = &fibs->fargs[1];

    f1->n = n - 1;
    f2->n = n - 2;
    qthread_empty(&f1->result);
    qthread_empty(&f2->result);

    fibs->target = result;

    // Collect results of sub-actions
    qthread_fork_precond(fib_result, fibs, NULL, 2,
                        &f1->result, &f2->result);

    // Fork off recursive actions
    qthread_fork(fib, f1, NULL);
    qthread_fork(fib, f2, NULL);

```

```
-sh-4.2$ time ./fib
fib(0) = 0

real    0m0.003s
user    0m0.000s
sys     0m0.001s
-sh-4.2$ time ./fib 24
fib(24) = 46368

real    0m0.065s
user    0m0.064s
sys     0m0.000s
-sh-4.2$
```

```
    return 0;
}

int main(int argc, char *argv[])
{
    unsigned long n = 0;
    f_arg_t args = { 0, 0 };

    if (argc > 1)
        n = atol(argv[1]);

    args.n = (aligned_t) n;

    qthread_initialize();

    qthread_empty(&args.result);

    qthread_fork(fib, &args, NULL);

    qthread_readFF(NULL, &args.result);

    printf("fib(%lu) = %lu\n", n, args.result);

    return 0;
}
```

5 Accumulators and Reductions

One common design pattern is that of an *accumulator* or a *reduction*. These are used when we want to reduce a large amount of data to a value. For example, one might want to compute the maximum value from a large number array. There is a function for this kind of approach in qthreads called `qt_loopaccum_balance`. A simple example of using `qt_loopaccum_balance` is given below, which computes the sum of an array.

```
#include <stdio.h>
```

```

#include <stdlib.h>
#include <qthread/qloop.h>

static void qtds_acc(void *a, const void *b){
    *(double*)a += *(double*)b;
}

static void qtds_worker(const size_t startat,
                       const size_t stopat,
                       void* array,
                       void *ret){
    size_t i;
    double sum = 0;
    for(i = startat; i < stopat; i++){
        sum += ((double*)array)[i];
    }
    *(double*)ret = sum;
}

double qt_double_sum_example(double *array, size_t length){
    double ret;
    qt_loopaccum_balance(0,
                        length,
                        sizeof(double),
                        &ret,
                        qtds_worker,
                        array,
                        qtds_acc);

    return ret;
}

int main(){
    qthread_initialize();
    double array[100];
    size_t i;
    for(i=0; i<100; i++){
        array[i] = rand();
    }
    double ret = qt_double_sum_example(array, 100);
    printf("Sum=%f\n", ret);
    return 0;
}

```

The work is broken up into chunks, then each chunk is summed in the `qtds_worker` function, and then the sums are reduced using `qtds_acc`. Both the worker and the accumulator functions will be assigned to workers automatically by the qthreads runtime. Note the types of the arguments for the worker. The second argument is `const`, to make it clear that the value must be accumulated in the location pointed to by the first argument.

5.1 Sincs

There may be use cases when you want to write your own reduction. For example, perhaps you might need some sort of tree-reduction. One tool in the qthreads toolbox that can be helpful for this sort of application is the `sinc`. In the following example, we use `sincs` to build an n -ary tree of depth d and sum the leaves in parallel.


```

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <unistd.h>
#include <qthread/qthread.h>
#include <qthread/sinc.h>

const int n = 30;
const int d = 4;

typedef struct v_args_s {
    size_t    depth;
    qt_sinc_t *sinc;
} v_args_t;

static void incr(void* target, const void *src){
    *(int*)target += *(int*)src;
}

static aligned_t buildAndReduce(void* arg_){
    size_t i;
    v_args_t* arg = (v_args_t*)arg_;
    qt_sinc_t sinc;
    aligned_t result, initial = 0;
    qt_sinc_init(&sinc, sizeof(aligned_t), &initial, incr, n);
    if(arg->depth > 1){
        v_args_t args = { arg->depth - 1, &sinc };
        for(i=0; i<n; i++)
            qthread_fork(buildAndReduce, &args, NULL);
        qt_sinc_wait(&sinc, &result);
        qt_sinc_submit(arg->sinc, &result);
    } else{
        result = 1;
        qt_sinc_submit(arg->sinc, &result);
    }
    return 0;
}

int main(){
    aligned_t result, initial = 0;
    qthread_initialize();
    qt_sinc_t sinc;
    qt_sinc_init(&sinc, sizeof(aligned_t), &initial, incr, n);
    v_args_t arg = {d, &sinc};
    size_t i;
    for(i=0; i<n; i++)
        qthread_fork(buildAndReduce, &arg, NULL);
    qt_sinc_wait(&sinc, &result);
    printf("Incremented_%d_times\n", (int)result);
    return 0;
}

```

The sinc structure contains a function that works similarly to the function bound in the loopaccum example. The first argument is a pointer to the accumulator, and the second is a pointer to the value we want to ac-

accumulate. Note we create a new `sinc` at each node in the tree, waiting on its children. The `qt_sinc_submit` function applies the accumulator function, in this case, `incr` and adjusts the semaphore in the `sinc` structure such that when the correct number of submissions have been made (in this case `n`), any threads waiting on the `sinc` (in this case only one) can continue.

6 Workers and Shepherds

As described in the introduction, the workers are usually bound one per core, while the shepherds (shared work queues) are often bound somewhere higher in the memory hierarchy. While `qthreads` will try and do The Right Thing[®], there may be times where the user needs to set the number of workers or shepherds. The recommended way to do this is with environment variables. For example: running with `QT_NUM_SHEPHERDS=8` `QT_NUM_WORKERS_PER_SHEPHERD=2` will yield 8 shepherds and 16 workers. You can check the number of workers and shepherds with the api calls `qt_num_workers()` and `qt_num_shepherds()`. For example:

```
#include <stdio.h>
#include <stdlib.h>
#include <qthread/qthread.h>

int main(int argc, char *argv[]){
    qthread_initialize();

    printf("Number_of_workers:_%d\n", qthread_num_workers());
    printf("Number_of_shepherds:_%d\n", qthread_num_shepherds());

    return 0;
}
```

6.1 Worker ID

While not recommended for most use cases, there may be situations where one would want to use worker id explicitly. In the following example, we again do a simple sum reduction, this time explicitly taking into account the number of workers, and assigning each worker to a subset of the array to be summed.

```
#include <stdio.h>
#include <stdlib.h>
#include <qthread/qthread.h>

typedef struct Vector {
    int n;
    int *array;
} Vector;

static aligned_t sum(void *arg){
    int i;
    int *array = ((Vector*) arg)->array;
    int n = ((Vector*)arg)->n;
    int result = 0;
    qthread_worker_id_t id = qthread_worker(NULL);

    for(i=0; i<n; i++)
```

```

        result += array[id * n + i];

    return result;
}

int main(int argc, char *argv[]){
    srand(1);
    int elems_per_worker = 100;
    qthread_initialize();
    int n = qthread_num_workers();
    int* array = malloc(n * elems_per_worker * sizeof(int));
    aligned_t* return_value = malloc(n * elems_per_worker * sizeof(int));
    int i, results=0;

    printf("Number_of_workers:_%d\n", n);
    printf("Number_of_array_elements:_%d\n", n * elems_per_worker);

    Vector input;
    input.n = n;
    input.array = array;

    for(i=0; i<n*elems_per_worker; i++)
        array[i] = rand();

    for(i=0; i<n; i++)
        qthread_fork(sum, &input, return_value + i);

    for(i=0; i<n; i++){
        qthread_readFF(NULL, return_value + i);
        results += return_value[i];
    }

    printf("Result:_%d\n", results);

    return 0;
}

```

7 Atomics and Blocking

Qthreads has a number of tools for doing atomic and blocking operations. An illustrative example of a non-blocking atomic operation is the `qthread_incr` function. The increment function is a thread-safe version of `*operand += incr;`

In the following program we have a number of tasks that all increment a single variable, with the desired property that the variable is incremented `n` times.

```

#include <stdio.h>
#include <qthread/qthread.h>

const int n = 30;
aligned_t master = 0;

static aligned_t incr(void *arg){

```

```

    qthread_incr(&master, 1);
    return 0;
}

int main() {
    qthread_initialize();
    aligned_t rets[n];
    size_t i;
    for(i=0; i<n; i++)
        qthread_fork(incr, NULL, rets + i);

    for(i=0; i<n; i++)
        qthread_readFF(NULL, rets + i);

    printf("Incremented_%d_times\n", (int)master);
    return 0;
}

```

7.1 Barriers

Qthreads has a number of mechanisms for creating barriers as well. For example, we could implement the above incrementer using a barrier instead of the `readFF` function calls.

```

#include <stdio.h>
#include <qthread/qthread.h>
#include <qthread/barrier.h>

const int n = 30;
qt_barrier_t *wait_on_me;
aligned_t master = 0;

static aligned_t incr(void *arg) {
    qthread_incr(&master, 1);
    qt_barrier_enter_id(wait_on_me, 0);
    return 0;
}

int main() {
    qthread_initialize();
    size_t i;
    wait_on_me = qt_barrier_create(n+1, REGION_BARRIER);
    for(i=0; i<n; i++)
        qthread_fork(incr, NULL, NULL);
    qt_barrier_enter(wait_on_me);
    qt_barrier_destroy(wait_on_me);
    printf("Incremented_%d_times\n", (int)master);
    return 0;
}

```

Instead of the parent task blocking on each full empty bit in order, each task is blocked until all tasks have entered the barrier. Then each is allowed to continue, including the parent task. Note because we want the parent task to wait on the barrier as well, the first argument (which is the number of tasks we want to wait on) is `n+1`.

7.2 High Performance Asynchronous Behaviour

One application of lightweight threads is low overhead asynchronous behaviour. For example, if we want a server to do some expensive computation asynchronously, qthreads is a good option. In the following example, we have a tcp server that reads in an integer and computes the fibonacci sequence at that index. While this could be done with a fork or pthread, this approach will allow significantly higher throughput by reducing the overhead of spawning the task.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <linux/sched.h>
#include <sched.h>
#include <qthread/qthread.h>

#define BUFSIZE 1024
#define PORT 5000

void error(char *msg) {
    perror(msg);
    exit(1);
}

int fib(int x){return x < 2 ? 1 : fib(x-1) + fib(x-2);}

typedef struct {
    int x;
    int childfd;
} argtype;

aligned_t process(void* arg){
    argtype *a = (argtype*) arg;
    char buf[BUFSIZE];
    sprintf(buf, "fib_%d:_%d\n", a->x, fib(a->x));
    write(a->childfd, buf, strlen(buf));
    return 0;
}

static void handleRequest(int childfd){
    char buf[BUFSIZE];
    while(1){
        bzero(buf, BUFSIZE);
        if(read(childfd, buf, BUFSIZE) == 0) break;
        printf("Working_on_%s\n", buf);
        argtype *newArg = malloc(sizeof(argtype));
        newArg->x = atoi(buf);
        newArg->childfd = childfd;
        //if(fork() == 0) process(newArg);
        qthread_fork(process, (void*)newArg, NULL);
        qthread_flushsc();
    }
}

int main(int argc, char **argv) {
```

```

qthread_initialize();
int parentfd, childfd;
struct sockaddr_in serveraddr; /* server's addr */
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
serveraddr.sin_port = htons(PORT);

parentfd = socket(AF_INET, SOCK_STREAM, 0);
int optval = 1;
setsockopt(parentfd, SOL_SOCKET, SO_REUSEADDR,
            (const void *)&optval , sizeof(int));

if (parentfd < 0)
    error("ERROR_opening_socket");

if (bind(parentfd, (struct sockaddr *) &serveraddr, sizeof(serveraddr)) < 0)
    error("ERROR_on_binding");

if (listen(parentfd, 1024) < 0)
    error("ERROR_on_listen");

while (1) {
    childfd = accept(parentfd, NULL, NULL);
    if (childfd < 0) error("ERROR_on_accept");
    handleRequest(childfd);
}
return 0;
}

```

Note there is an unfamiliar API call here as well: `qthread_flushsc`. The current default behaviour is to buffer tasks before moving them into the queue. This flushes said buffer, which is necessary in cases like this to begin work. To test this example, simply telnet or netcat to localhost port 5000 and enter some numbers (remember it's naive fibonacci, so don't enter large numbers if you want a result!). To get interesting asynchronous behaviour, responses will likely start slowing down around $n = 40$.