MuPC Test Suite v1.0

1. Introduction

The test cases in the MuPC Test Suite were written to test the different components of the MuPC RTS. None of the test cases check the error detection capabilities of the compiler. However, some of these test cases do check the error detection capabilities of the Run−Time System (RTS). For instance, the test case *test_barrier2.c* checks that the RTS does produce a run−time error when a programmer tries to call a **upc_notify(1)** followed by a **upc_wait(2)**; which is illegal according to the UPC Specification.

The MuPC Test Suite contains 50 test cases. These test cases have been named according to the components that they test so that users can differentiate the test cases easily. Below is an explanation of the names given to the test cases:

1) test_getput*.c − test cases that check the correctness of *get* and *put* operations.
2) test_barrier*.c − test cases that check the correctness of the synchronization operations.
3) test_locks*.c − test cases that check the correctness of the locking operations.
4) test_memory*.c − test cases that check the correctness of the memory operations.
5) test_string*.c − test cases that check the correctness of the string handling operations.
6) test_int_*.c − test cases that integrates several different components.
7) test_app_*.c − application programs written using UPC code.
8) test_stress_*.c − test cases that stress test the different components of MuPC.
9) test_globalexit.c − test case that tests the global exit function.

Section 2 lists all the test cases together with their descriptions. Further information about a particular test case can be obtained from the *header* of the test case. Please refer to the README to obtain more information about the test suite. The file *BugList* included with the test suite package contains a list of bugs found when testing MuPC.

2. List of Test Cases

2.1 Test cases for *get* and *put* operations

1. **test_getput1.c**: gets and puts with and without affinity.

   Function tested: get and *put* operations for all data type.

   Description:
   - A 2−part test case that test gets and puts with affinity followed by test of gets and puts with no affinity.
   - 1st part:
     - Initalize shared array "Idata" concurrently.
     - Each thread updates data in "Idata" that it has affinity to.
     - Thread 0 performs error checking at end.
   - 2nd part:
     - Reinitialize shared array "Idata".
     - Each thread updates data in "Idata" that it has no affinity to.

    - Thread 0 performs error checking at end.
    - Thread 0 determines the success of test case.

2. **test_getput2.c**: gets and puts with interleaved data types.

   Function tested: *get* and *put* operations for all data type.

   Description:
   - Test gets and puts with interleaved data types. Data types currently supported by MuPC include char, short, integer, long, float, and double. Long double has not been implemented.
   - Each thread initializes the portion of the 6 arrays that they have affinity to concurrently (each array for each data type).
   - Each element in the 6 arrays gets updated.
   - Thread 0 performs error checking at end.

3. **test_getput3.c**: gets and puts with uniform offset affinity.

   Function tested: *get* and *put* operations for all data type.

   Description:
   - A 2−part test case that tests gets and puts with uniform offset of affinity.
   - This test case is similar to *test_getput1*.c, except that the declaration of shared array "Idata" is different. This gives us different memory access pattern.
   - 1st Part (gets and puts with affinity to thread):
     - All threads initialize and update array "Idata" concurrently.
     - Thread 0 performs error checking.
   - 2nd Part (gets and puts with no affinity to thread):
     - All threads reinitialize and update array "Idata".
     - Thread 0 performs error checking.
   - Thread 0 determine the success of test case in general.

4. **test_getput4.c**: volume test MuPC by concurrently doing many gets and puts.

   Function tested: *get* and *put* operations for all data type.

   Description:
   - Volume test MuPC RTS by issuing many gets and puts from all threads concurrently.
   - All threads initialize and updates a[MYTHREAD][i] concurrently.
   - Thread 0 performs error checking at end.

2.2 Test cases for synchronization operations.

1. **test_barrier1.c**: test of gets and puts with notify and wait (works for n=pow(2, x), x=0,1,2,...)

   Function tested: upc_notify, upc_wait

Description:
- Integration test of gets and puts functions with split−barrier (notify and wait).
- Initialize and update data concurrently.
- Calls upc_notify and upc_wait. Perform local computation in between the split−barriers.
- Calls upc_notify and upc_wait for a specified number of time inside a for loop. Acts as a kind of stress test.
- Thread 0 performs error checking at end.

2. **test_barrier2.c**: test if appropriate run−time error is generated.

   Function tested: upc_notify, upc_wait

   Description:
   - Perform test on the upc_notify() and upc_wait() functions.
   - This primarily tests if upc_notify and upc_wait would generate a runtime error should the expressions used in the two functions doesn't match.
   - We first initialized an array "a".
   - Calls upc_notify and upc_wait with different expressions.
   - Test case passes if run−time error's generated, and fails if the printfs are executed.

3. **test_barrier3.c**: test of synchronization of upc_barrier.

   Function tested: upc_barrier

   Description:
   - Check if synchronization does happen in a barrier.
   - Each thread sets the value of sync[MYTHREAD] with the value 1.
   - Each thread executes the barrier statement with local variable "localVar" in the expression.
   - Each thread sums up the elements in array sync, and the value should be equal to THREADS.
   - Each thread performs error checking.
   - Repeat the steps above with the only change of shared variable "shVar" as the expression for the barrier statement.
   - Thread 0 determines if test case passes.

4. **test_barrier4.c**: test of synchronization of split−barrier.

   Function tested: upc_notify, upc_wait.

   Description:
   - Check if synchronization does happen in a split−barrier.
   - Each thread sets the value of sync[MYTHREAD] with the value 1 right before executing the upc_notify with local variable "localVar" as expression.
   - Each thread executes the upc_notify with "localVar" as expression.
   - Each thread increments the value of sync[MYTHREAD] in between the upc_notify and upc_wait.
   - Each thread executes the upc_wait with "localVar" as expression.
   - Each thread sums up the elements in array sync, and the value should be equal to

(2*THREADS).
- Each thread performs error checking.
- Repeat the steps above with the only change of shared variable "shVar" as the expression for the split−barrier.
- Thread 0 determines if test case passes.

2.3 Test cases for synchronization operations.

1. **test_locks1.c**: test the use of upc_all_lock_alloc and other locking mechanisms.

   Function tested: upc_all_lock_alloc, upc_lock_init, upc_lock, upc_unlock

   Description:
   - Allocate locks using the colelctive upc_all_lock_alloc function.
   - Use allocated locks to protect a critical region.
   - Perform error checking at end.

2. **test_locks2.c**: test of upc_global_lock_alloc and other locking mechanisms.

   Function tested: upc_global_lock_alloc, upc_lock_init, upc_lock, upc_unlock

   Description:
   - Tests the correctness of upc_global_alloc. Also test if upc_lock_init, upc_lock, and upc_unlock works.
   - Allocate locks using global_lock_alloc, and perform all necessary initialization.
   - Use the two locks allocated to protect 2 critical regions.
   - Perform error checking at end.

3. **test_locks3.c**: test the use of lock_attempt, and other locking mechanism.

   Function tested: upc_all_lock_alloc, upc_lock_init, upc_lock_attempt, upc_unlock

   Description:
   - Test the correctness of the upc_lock_attempt() function.
   - First allocate lock, and perform necessary initializations.
   - Lock critical region using upc_lock_attempt.
   - Perform accumulation in critical region.
   - Thread 0 performs error checking.

4. **test_locks4.c**: test of locks and gets and puts of a specified DTYPE data type.

   Function tested: upc_global_lock_alloc, upc_lock_init, upc_lock, upc_unlock

   Description:
   - Intended originally to test and see if locking and unlocking a lock twice is allowed by the MuPC RTS. Locking: Similar to Compaq’s RTS, MuPC hangs when a thread tries to lock a "lock" twice. Unlocking: Program ran fine when a thread unlocks a lock twice.

Output is correct too. No warning or message appeared.
- A simple test case that allocates a lock and then updates the value of the lock.
- Critical region protected by the lock just simply increments and decrements the value of counter continuously.
- Thread 0 performs error checking at end.

5. **test_locks5.c**: test of implied synchronization of upc_all_lock_alloc.

   Function tested: upc_all_lock_alloc

   Description:
   - Check if there’s an implied synchronization before all threads execute the collective lock allocation function − upc_all_lock_alloc.
   - Each thread set the value of sync[MYTHREAD] with the value 1.
   - Each thread allocates the lock collectively.
   - Each thread sums up the elements in array sync, and the value should be equal to THREADS.
   - Each thread performs error checking.
   - Thread 0 determines if test case passes.

2.4 Test cases for memory operations.

1. **test_memory1.c**: test the correctness of upc_all_alloc.

   Function tested: upc_all_alloc

   Description:
   - Test the correctness of upc_all_alloc() dynamic memory allocation function.
   - Each thread initialized sync[MYTHREAD] to 1 just before reaching upc_all_alloc.
   - A shared array "a" is then allocated using the collective memory allocation function.
   - Each thread sums up the values of array "sync" and they should get a value of THREADS.
   - Each thread performs error checking.
   - The shared array "a" is initialized and updated by all threads using upc_forall.
   - Finally thread 0 perform the necessary error checking to determine what error has occurred.

2. **test_memory2.c**: test the correctness of upc_global_alloc function.

   Function tested: upc_global_alloc

   Description:
   - Test the correctness of upc_global_alloc(). There are 2 parts in this test case.
   - 1st part:
     - All threads will perform global memory allocation, initialization, simultaneously.
     - Thread 0 performs error checking at end.
   - 2nd part:
     - Each thread initializes the elements in array b with value of the array a of its

  neighbour (MYTHREAD + 1).
  - Thread 0 performs error checking at end.
- Thread 0 determines the success of this test case.

3. **test_memory3.c**: test the correctness of upc_local_alloc.

   Function tested: upc_local_alloc

   Description:
   - Tests the correctness of upc_local_alloc().
   - Test case first allocates THREAD number of array a dynamically using upc_local_alloc(), then thread 0's array is initialized.
   - Thread i then fills up the values in its array with values from thread (i − 1).
   - After all the arrays have been filled, thread 0 perform the necessary error checking.

4. **test_memory4.c**: test if the RTS allows a user to allocate more than the max memory.

   Function tested: upc_global_alloc

   Description:
   - Test if allocating more than the maximum allowed memory would produce an error.
   - MuPC only allows 8MB of memory to be used/allocated in each thread. Allocating size of memory that exceeds 8MB is not allowed.
   - This is a simple test case that allocates more than 8MB of memory using upc_global_alloc.

5. **test_memory5.c**: test that upc_threadof functions properly.

   Function tested: upc_threadof

   Description:
   - This is a simple test case that tests the correctness of the upc_threadof function. This function is part of the MuPC RTS because it was implemented by the programmer.
   - First check that upc_threadof returns the right affinity for each of the elements in array "ptr" which was declared with a blocking factor of 1.
   - Secondly, check that upc_threadof returns the proper affinity for each of the elements in array "ptr2" which was declared with a blocking factor of BLOCK.
   - Thread 0 performs error checking at end.

6. **test_memory6.c**: test that upc_phaseof functions properly.

   Function tested: upc_phaseof

   Description:
   - This is a simple test case that tests the correctness of the upc_phaseof function. This function is part of the MuPC RTS because it was implemented by the programmer.
   - Each thread first checks that the phase of each element in array "ptr" which was declared with block factor of 1, is always 0.

- Then, each thread checks that the phase of each element (returned by the upc_phaseof function) in array "ptr2" declared with block factor of BLOCK is correct.
- Thread 0 then performs assignment and casting of pointers, and check that upc_phaseof still returns the right phase.
- Thread 0 performs error checking at end.

7. **test_memory7.c**: test that upc_addrfield functions properly.

   Function tested: upc_addrfield

   Description:
   - This is a simple test case that tests the correctness of the upc_addrfield function. This function is part of the MuPC RTS because it was implemented by the programmer.
   - Each thread checks if the value returned from upc_addrfield is appropriate.
   - Thread 0 performs error checking at end.

8. **test_memory8.c**: test of implied synchronization of upc_all_alloc.

   Function tested: upc_all_alloc

   Description:
   - Check if there's an implied synchronization before all threads execute the collective memory allocation function − upc_all_alloc.
   - Each thread set the value of sync[MYTHREAD] with the value 1.
   - Each thread allocates memory collectively.
   - Each thread sums up the elements in array sync, and the value should be equal to THREADS.
   - Each thread performs error checking.
   - Thread 0 determines if test case passes.

2.5 Test cases for string handling operations.

1. **test_string1.c**: test correctness of upc_memcpy.

   Function tested: upc_memcpy

   Description:
   - Test the correctness of the upc_memcpy() functionality.
   - Two source arrays "src" and "zeroSrc" are initialized.
   - Each thread takes turn to copy the content of "src" to "dst" array.
   - Thread 0 performs error checking and copies the content of "zeroSrc" to "dst". This reinitializes the content of "dst" to 0.
   - Thread 0 determines the success of test case.

2. **test_string2.c**: test correctness of upc_memget.

   Function tested: upc_memget

Description:

- Test correctness of upc_memget() function. There are 2 parts to this test case. The first execute a single upc_memget(), while the 2nd executes a specified time of upc_memget().
- 1st Part:
  - Thread 0 initializes content of array "masterCopy".
  - Each thread copies the content of "masterCopy" to its local array "localCopy".
  - Each thread performs error checking and store results in err[MYTHREAD].
  - Thread 0 determines if the 1st part passes or not.
- 2nd Part:
  - Thread 0 reinitializes content of array "masterCopy" with different values.
  - Each thread performs multiple get from "masterCopy" to local array "localCopy".
  - Each thread performs error checking and store results in err[MYTHREAD].
  - Thread 0 determines if 2nd part passes.
- Thread 0 determines if test case passes.

3. **test_string3.c**: test correctness of upc_memput.

   Function tested: upc_memput

   Description:
   - Test correctness of upc_memput() function.
   - Each thread first initializes local array "src".
   - Each thread takes turn to copy the content of its local array "src" to shared array "dst".
   - Thread 0 performs error checking and determines if test case passes.

4. **test_string4.c**: test the functionality of upc_memset.

   Function tested: upc_memset

   Description:
   - Test correctness of upc_memset() function.
   - Each thread initializes a local array "compare[]" to be used for comparison purposes.
   - Then, upc_memset each element in "a" to VALUE+MYTHREAD.
   - Each thread performs error checking by comparing "a" to array "compare".
   - Thread 0 determines success of test case at end.

2.6 Integration test cases involving several different components.

1. **test_int_allfunc.c**: a test case that includes all the functions implemented in MuPC.

   Function tested: all functions.

   Description:
   - This program incorporates all the functions currently implemented in MuPC.
   - This is not an application program, but simply a test program that tries to make use of all functions and generate lots of communications between threads. This test case is an

extension of test_alltype.c.
- Thread 0 allocates memory for 6 arrays of diff data types using upc_local_alloc.
- Thread 0 initializes the contents of the 6 arrays with upc_memset. This is not necessary, but we incorporate it anyway.
- Thread 0 fills in the 6 arrays with some values.
- Each thread copies the contents of the 6 arrays from thread 0 into its own.

2. **test_int_alltype.c**: a more complex test case that integrates the different data types.

   Function tested: all *get* and *put* functions, upc_local_alloc, upc_memcpy, upc_memset.

   Description:
   - This program tries to integrate all the different data types, this includes char, short, long, int, float, and double.
   - 6 arrays of all data types are allocated using upc_local_alloc on thread 0.
   - The allocated arrays are then initialized first with upc_memset, then given some values.
   - Each thread then copies the arrays from thread 0 into their own arrays.
   - Error checking is performed at end.

3. **test_int_barlocks.c**: integrates memory alloc., locking, and split−barrier functions.

   Function tested: upc_all_lock_alloc, upc_global_alloc, upc_memset, upc_lock_init, upc_lock, upc_unlock, upc_notify, upc_wait.

   Description:
   - Integrates memory allocation functions with upc_notify, upc_wait, and locking functions.
   - Test case first perform necessary memory and lock allocations; and initialization.
   - All threads perform initialization of a[MYTHREAD] in between the 1st split−barrier.
   - All threads perform error checking on array a[MYTHREAD+1] in the 2nd split−barrier to make sure that synchronization occurred in the previous split−barrier.
   - Each thread takes turn to enter a critical region where each thread accumulates the value of a[MYTHREAD] in the array acc.
   - Thread 0 performs error checking at end.

4. **test_int_memlocks.c**: integration test of locking, memory alloc, and memory transfer funcs.

   Function tested: upc_local_alloc, upc_global_alloc, upc_all_lock_alloc, upc_lock_init, upc_lock, upc_unlock, upc_memset, upc_memget, upc_memput, upc_memcpy.

   Description:
   - Test the interaction of the locking, memory allocation, and memory transfer functions.
   - In this test case, each thread will take turns to enter a critical region where it will perform memory transfer operations from one array to another.
   - First, memory and lock is allocated, and initialized.
   - Each thread enters the critical region, and perform copying of blocks of data from one array to another.
   - Thread 0 performs error checking at end.

5. **test_int_memstring.c**: tests upc_all_alloc, and memory transfer functions.

   Function tested: upc_all_alloc, upc_memget, upc_memput

   Description:
   - Allocate shared array "a" and "b[i]" using upc_all_alloc, and initialize array "a".
   - Each thread copies the content of shared array "a" into its local array, modifies the content of the local array, and then write to shared array.
   - Thread 0 performs error checking at end.

6. **test_int_multilocks.c**: test multiple locks and multiple memory allocations.

   Function tested: upc_all_alloc, upc_memset, up_all_lock_alloc, upc_lock_init, upc_lock, upc_unlock

   Description:
   - Test the robustness of MuPC's locking mechansism. Most test cases only make use of single lock. We attempt to use multiple locks in this test case, nesting them, while updating 3 shared arrays in the critical regions.
   - First allocate memory for 3 shared arrays, a[0], a[1], and a[2], and initialize them.
   - Allocate 3 locks and initialize them.
   - Create 3 critical regions protected by the 3 locks. These critical regions overlap each other causing the 3 locks to be nested inside one another.
   - Perform error checking at end.

7. **test_int_precision.c**: test loss of precision in floating−point numbers during data transfer.

   Function tested: upc_memcpy, upc_memget, upc_memput, *get* and *put* operations for **float** and **double** data type.

   Description:
   - Tests if a loss of precision occurs when transferring floating−point numbers.
   - Change the constant DTYPE to test either "float" or "double" data type.
   - Initialize an array origin with data type DTYPE.
   - Transfer data using all possible methods − normal gets and puts, upc_memcpy, upc_memput, and upc_memget.
   - Perform error checking at end.

2.7 Application programs.

1. **test_app_matmult.c**: matrix multiplication program.

   Function tested: *get* and *put* operations for all data types.

   Description:
   - Matrix multiplication program. This program performs the following matrix

multiplication operation. User can change the data type by simply changing the DATA_T macros.

a[ROW x colA] . b[colA x colB] = c[ROW x colB]

- Matrix a is divided up among the threads so that row 0 has affinity to thread 0, and row i has affinity to thread i.
- Matrix b is divided up among the threads so that column 0 has affinity to thread 0 and column i has affinity to thread i.
- Matrix c is divided in the same fashion as matrix a.

2. **test_app_prime.c**: counts the number of prime in a given range, and fine largest prime.

   Function tested: UPCRTS_GetBytes, UPCRTS_GetSyncInteger, UPCRTS_PutInteger.

   Description:
   - This program calculates the total prime numbers in a given range and the highest prime in this range. The method used to determine if a number is prime is known as the "trial divison" method. This program is only suitable for finding primes in the range of 1 to 60M.
   - Program first calculate the amount of work each thread is responsible for.
   - Each thread gets their portion of work, and calculate the number of primes.
   - Thread 0 finds the largest prime and total number of primes.

3. **test_app_wave1.c**: decompose vibrating string into points and update its amplitude.

   Function tested: upc_local_alloc, upc_memset, *get* and *put* operations for **double** data type.

   Description:
   - A vibrating string is decomposed into points. Each thread is responsible for updating the amplitude of a number of points over time. This is not an embarassingly parallel program because each thread has to send the endpoint values of its portion of the string to its neighbour.
   - To change the number of points and steps, users can change the POINTS and STEPS constants.
   - This program first divide−up the workload amongst the threads.
   - Each thread performs initialization of the portion of sine curve it’s responsible for.
   - Each thread performs update concurrently on its portion of curve.
   - Each thread calculates subtotal of all the points in its portion of curve.
   - Thread 0 sums up all the subtotals, the total should be close to 0.

4. **test_app_wave2.c**: decompose vibrating string into pts and update its amplitude. (alt ver.)

   Function tested: *get* and *put* operations for **double** data type.

   Description:
   (Alternate version of test_wave.c. This test case reflects shared programming techniques better. However, it uses less MuPC functions. This version should give better performance if performance is a concern.)

   - A vibrating string is decomposed into points. Each thread is responsible for updating the

amplitude of a number of points over time. This is not an embarassingly parallel program because each thread has to send the endpoint values of its portion of the string to its neighbour.
- To change the number of points and steps, users can change the POINTS and STEPS constants.
- This program first divide−up the workload amongst the threads.
- Each thread performs initialization of the portion of sine curve it’s responsible for.
- Each thread performs update concurrently on its portion of curve.
- Each thread calculates subtotal of all the points in its portion of curve.
- Thread 0 sums up all the subtotals, the total should be close to 0.

2.8 Test cases that stress test MuPC’s components.

1. **test_stress_01.c**: stress test barriers and notify’s and wait’s.

   Function tested: upc_notify, upc_wait, upc_barrier.

   Description:
   - Stress test barriers and split−barriers.
   - Call COUNT number of barriers and split barriers in for loop, while performing checks for barrier synchronization.
   - Perform error checking at end.

2. **test_stress_02.c**: stress test upc_memputs.

   Function tested: upc_memput.

   Description:
   - Stress test upc_memput by doing many memputs as fast as possible, i.e. without operations that require synchronization in between the memputs, such as barriers.
   - Each thread initialize its own local array.
   - Copy value of local array to array "a" using upc_memput.
   - Perform error checking at end.

3. **test_stress_03.c**: stress test upc_memget.

   Function tested: upc_memget.

   Description:
   - Stress test upc_memget by doing many memgets as fast as possible, i.e. Without operations that require synchronization in between the memgets, such as barriers.
   - Thread 0 initializes array "a".
   - Every thread performs CONST number of upc_memget from shared array "a" to local array "local".
   - Error checking at end.

4. **test_stress_04.c**: alternative stress test of upc_memputs.

Function tested: upc_memput, upc_memset.

Description:
- Stress test upc_memput and the internal buffering system by doing large memputs.
- If block of message to be "put" is larger than the maximum transferable block, it'll be split into multiple section and sent one after another.
- To really stress this, one can change the MAX_BLOCK_XFER constant from 10,000 to 10. In this case, a 10K block would have to be chopped into 1000 parts. However, the user needs access to the source code of MuPC.
- Initialize a shared array "a" using upc_memset to something != 0.
- Each thread initialize local array "local" using memset to 0.
- Perform upc_memput of a large volume.
- Perform error checking at end.

5. **test_stress_05.c**: alternative stress test of upc_memget.

   Function tested: upc_memget, upc_memset.

   Description:
   - Stress test upc_memget and the internal buffering system by doing large memgets.
   - If block of message to be "get" is larger than the maximum transferable block, it'll be split into multiple section and retrieved one after another.
   - To really stress this, one can change the MAX_BLOCK_XFER constant from 10,000 to 10. In this case, a 10K block would have to be chopped into 1000 parts. However, the user needs access to the source code of MuPC.
   - Thread 0 first initializes a shared array "a" to 0.
   - Every thread initializes its own local array to a value that's non−zero.
   - Each thread then performs upc_memget of large chunks of memory.
   - Each thread perform error checking. Thereafter, thread 0 will sum up the error value of each thread, the total should equal to 0.

6. **test_stress_06.c**: stress test of upc_memcpy.

   Function tested: upc_local_alloc, upc_memcpy, upc_memset.

   Description:
   - Stress test upc_memcpy function and the internal buffering system by calling upc_memcpy many times, with as little synchronization in between calls.
   - Thread 0 allocates and initialize destination array with non−zero values.
   - Each thread allocates shared array src[MYTHREAD] with affinity to it.
   - Each thread copies the content of its array CONST number of times to the shared dst array. In every iteration, each thread would do a upc_memset to src[MYTHREAD] to change the values of the array.
   - Thread 0 performs error checking at end. All values in dst should be 0 at this point.

7. **test_stress_07.c**: alternative stress test of upc_memcpy.

   Function tested: upc_local_alloc, upc_memcpy, upc_memset.

Description:

- Stress test upc_memcpy function and the internal buffering system, by copying large amount of memory. The stress here is not on the number times, but the amount of data copied.
- Thread 0 allocates a shared array "src" and sets the whole array to 0.
- Each thread allocates shared array dst[MYTHREAD] with affinity to itself. Also, each shared array is initialized using upc_memset non−zero values.
- Each thread copies content of "src" to "dst[MYTHREAD]" concurrently. After the upc_memcpy, each dst[MYTHREAD] should contain 0 values.
- Each thread performs error checking and sends results to thread 0. Thread 0 decides if test case passes or fails.

8. **test_stress_08.c**: stress test of upc_memset.

   Function tested: upc_local_alloc, upc_memset.

   Description:

   - Stress test upc_memset function and the internal buffering system by calling upc_memset many times, with as little synchronization in between calls.
   - Each thread allocates a shared array "dst[MYTHREAD]".
   - Each thread initializes its own shared array using upc_memset CONST number times.
   - Each thread performs error checking. Final values in dst[MYTHREAD] should be 0.
   - Thread 0 determines the success of test case.

9. **test_stress_09.c**: alternative stress test of upc_memset.

   Function tested: upc_local_alloc, upc_memset.

   Description:

   - Stress test upc_memset function and the internal buffering system by calling upc_memset many times, with as little synchronization in between calls.
   - Each thread allocates a shared array "dst[MYTHREAD]".
   - Each thread initializes its neighbour’s shared array using upc_memset CONST number times.
   - Each thread performs error checking. Final values in dst[MYTHREAD] should be 0.
   - Thread 0 determines the success of test case.

10. **test_stress_10.c**: stress test of upc_all_alloc.

    Function tested: upc_all_alloc

    Description:

    - Stress test upc_all_alloc by calling the function numerous times. The number of times called is limited by the 8MB of memory pool allocated for each thread.
    - Each thread collectively performs allocation of CONST number of arrays, each with size of SIZE.
    - Some of these allocated arrays is used to verify that they exist and that the pointers returned are valid.

- Thread 0 performs error checking at end.

11. **test_stress_11.c**: stress test of upc_global_alloc.

    Function tested: upc_global_alloc

    Description:
    - Stress test upc_global_alloc by calling the function numerous times. The number of times called is limited by the 8MB of memory pool allocated for each thread.
    - Each thread allocates memory using upc_global_alloc CONST number of times and store the pointer returned each time in an array.
    - Each thread makes use of some of the memory allocated.
    - Each thread performs error checking.
    - Thread 0 performs determines if test case passes.

12. **test_stress_12.c**: stress test of upc_local_alloc

    Function tested: upc_local_alloc

    Description:
    - Stress test upc_local_alloc by calling the function numerous times. The number of times called is limited by the 8MB of memory pool allocated for each thread.
    - Each thread allocates memory using upc_local_alloc CONST number of times and store the pointer returned each time in an array.
    - Each thread makes use of some of the memory allocated.
    - Each thread performs error checking.
    - Thread 0 performs determines if test case passes.

13. **test_stress_13.c**: stress test of upc_all_lock_alloc, upc_lock_init.

    Function tested: upc_all_lock_alloc, upc_lock_init, upc_lock, upc_unlock.

    Description:
    - Stress test upc_all_lock_alloc by calling the function numerous times. The number of times called is limited by the 8MB of memory pool allocated for each thread.
    - Since a lock has to be initialized in MuPC in order to be used, this also stress test upc_lock_init.
    - Each thread collectively allocates CONST number of locks using upc_all_lock_alloc.
    - One of the locks in then used to determine if all threads received the valid pointer.
    - Error checking is performed at end by thread 0.

14. **test_stress_14.c**: stress test of upc_lock and upc_unlock.

    Function tested: upc_lock, upc_unlock, upc_all_lock_alloc, upc_lock_init.

    Description:
    - Stress test upc_lock and upc_unlock by calling the functions numerous times. The number of times can be changed by modifying the constant CONST.

- A lock, lock_1, is first allocated collectively.
- A critical region protected by lock_1 is executed CONST number of times.
- Perform error checking at end.

2.9 Miscellaneous test cases.

1. **test_globalexit.c**: test the functionality of the **upc_global_exit()**.

   Function tested: upc_globalexit.

   Description:
   - Test correctness of **upc_global_exit()** function.