CS160 – Winter 2018 – Programming Assignment #5

Due 1159pm, Mar 4, 2018 (SUNDAY)

The goal of this assignment is to build a message passing implementation with a limited API for use within a threaded program. That is, to perform message passing between threads within a single program. Included in your starter code a message-passing benchmark the uses the API, that you will implement. using Github classroom for turn in your final codes so that you have flexibility in designing your final codes..

msgbench.c

Your are given msgbench that, when compiled with your header file and linked with your message passing code will properly benchmark the performance of your implementation. If you look at msgbench, it starts two threads and performs a ping-pong test. Its usage is

```
usage: msgbench <size> <trials> [validate]
```

If anything appears as the third argument, it will validate the round-trip message. After each iteration.

You might notice the statement

```
assert(sbuf[i] == rbuf[i]);
```

This is program "assertion" meaning that it must be true. In the even the assertion is false, the program immediately aborts and dumps a core file.

cs160mp.h

You are given a header file called `cs160mp.h`. It defines the prototypes for the functions that you must implement. The semantics of these calls are intended to mirror similar calls in MPI

int MP_Init(int nthreads)	Initialize any internal data structures needed to implement message passing. Called once per program.
int MP_Finalize()	Clean up an allocated memory. Indicates no more message passing will be performed. Called after all message passing is complete. Called once per program
int MP_Size()	Return the number of threads passed to MP_Init()
int MP_Rank(pthread_t thread);	Give a rank (0 .. MP_Size()) to the thread with a particular id.
int iSend(void *buf, int len, int src, int dest, int tag, MSGRQST * request);	Send a memory buffer (no type) of particular length from src to dest using a specific tag. Asynchronous.
int iRecv(void *buf, int len, int src, int dest, int tag, MSGRQST * request);	Receive into a memory buffer of size len from src to dest (src may be wildcard) using tag (tag my be wildcard). Asynchronous
int msgWait(MSGRQST * request);	Wait for a specific message request to complete
int msgWaitAll(MSGRQST *requests, int nqrsts);	Wait for an array of message requests to complete
int getRank(MSGRQST *request);	Read the rank of a (completed) message request
int getTag(MSGRQST *request);	Read the tag of a (completed) message request
int getLen(MSGRQST *request);	Read the length of a (completed) message request

Note: in iSend,iRecv the integer id of both the src and the dest are used. This is because there is no per-thread initialization required. Instead a process-level initialization is used. Threads can find out their rank with MP_Rank.

Predefined Constants

ANY_SRC	Receive can match any Sender
ANY_TAG	Receive can match any Tag
MSGRQST_NULL	A "NULL" message request such that msgWait would immediately return if passed MSGRQST_NULL. If part of an array of MSGRQSTs passed to msgWaitAll, it would ignore this particular request in the array.

Note: ANY_SRC, ANY_TAG are already defined "sanely" in your starter code. MSGRQST_NULL and MSGRQST (see below) will need to be re-defined for your implementation.

Valid TAGS are non-negative

MSGRQST

You must define the MSGRQST structure in cs160mp.h . It can contain whatever fields you need for you implementation.

How to go about designing your solution

There numerous ways to design your solution. What is below is just one way to go about this. This broad outline will be talked about in lecture.

To start, let's make some observations

- Transferring actual data from one buffer to another is a memory copy. `memcpy` is a good way to do this. Type of data (int, long, float, double, …) isn't important since all threads are using the same memory and data representations.
- Data transfer is asynchronous. This is the part that requires much care, thought, and design when building your implementation.
- Any thread can call any of the key functions (iSend, iRecv, msgWait, msgWaitAll) at any time and in any relative order

Broad approach

Since messaging is asynchronous the matching iRecv for an iSend or a matching iSend for an iRecv may not yet have been called. This suggests a queue-based approach. That is, an iSend request is place on a queue, and an iRecv request is placed on a queue.

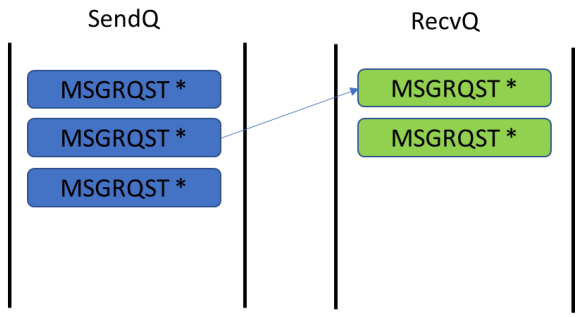
When is data actually transferred? When msgWait or msgWaitAll has been called.

Queue-based approach

There are many design possibilities. Once could have a single program-wide message queues that has all outstanding send and recv requests. One could have *two* program-wide message queues: one for send requests and one for receive requests. One could also have per-thread message queues.

The solution suggested below is for two program-wide message queues. A per-thread queuing system would allow more concurrency, but is logically much more complex. Why two queues? It simplifies logic for message completion.

Suppose that the only things iSend/iRecv actually do are: 1) check sanity of arguments and 2) fill out and place the MSGRQST on either the sendQ or recvQ. That makes these functions very straightforward. It also means that in the send of MPI-like implementation, all message passing is of the rendevouz type (no Eager mode). This also simplifies some logic. Let's look at a possible state of the queues



SendQ and RecvQ, arrow indicates a match of posted iSend and iRecv

The tricky part of this is the logic of message completion. Completion can occur whenever any thread calls `msgWait` or `msgWaitAll`. The hard part is how to wake up a thread that has blocked because when it called `msgWait` or `msgWaitAll`, the message(s) it was looking for have not yet completed. One way is to use a condition variable and broadcast to wake up any thread that might be blocked inside of `msgWait`. You also want a "failsafe" mode that allows a thread to "wake itself" while waiting. You *could* do this by busy waiting in a tight loop, but a nicer way is to look at `pthread_cond_timedwait()` and set a time out of a few milliseconds.

A number of other items that you will probably want to think about is when waiting for a message (or array of messages), do you want one thread to complete as many messages as

completed by either thread (in other words, either the sender could copy the data to the receiver or the receiver could copy the data from the sender).

Remember, when a message can complete, both the send and -a- matching receive must have been posted. Only when both are there, can the actual data be copied.

Before your write code

Sketch out how you want to do message completion. Try to figure out how your code could "go wrong". Think especially about critical sections. Mutexes or read/write locks can be used, depending upon your solution.

Implementing queues – you could do linked lists, but it's just fine to use arrays for your queues. If you use arrays for queues, you can easily expand the size of a `malloc'ed` arrary with `realloc()`. Do not engineer a solution that can have a maximum number of requests.

The largest logic issue in this assignment is figuring out what parts of message completion might be completed by another thread – in just an iSend and iRecv pair, the message could your queues of message requests, `realloc()` is pretty convenient when you need more space.

Creating a library (aka archive)

You are to create a static library called libcs160mp.a that contains all the object files you need to implement your solution. This is so, if you desire, you can implement your solution in multiple files. We will create test codes that link against your libcs160mp.a

Creating an archive. Suppose you have two object files called cs160mp.o and msgcompletion.o, the linux `ar` command can be used to create an archive.

```
$ ar rcs libcs160mp.a cs160mp.o msgcompletion.o
```

Then, to create msbench code you could do the following

```
$ cc -o msgbench msgbench.c libcs160mp.a -lpthread
```

Requirements

- You need to provide a Makefile the has targets
  - `libcs160mp.a`
  - `msgbench`
  - `clean`
- Graph the performance of msgbench with range of input sizes from 32 bytes to 1Mbyte (1024*1024 bytes). Turn in your (labeled) graph with name `msgbench-perf.pdf`. You may choose which message sizes to benchmark. For example, you might choose 32 byte

intervals through 4K, 256 byte intervals through 64K, 512Byte intervals through 256K and 1K intervals through 1M. Those are a guideline, pick your own our use these.

- Your code should have no memory leaks. We likely will use valgrind to check this for your code (valgrind is available on ieng6. It's not available on TSCC).
- **You may NOT change the API definition above (see `cs160mp.h` in the starter code)**

- Your Makefile shoud properly build the targets above
- `$ make clean` should remove all object files, libraries, and executables
- `$ make` should be the equivalent of "`make libcs160mp.a`"

Hints/Notes

- Msgbench is just ONE code. Try creating some test cases that might, for example have one thread create 10 iSends with tags counting upwards and the 10 iRecvs with tags counting downwards. This would test message completion.
- Your MP_Init() will initialize some variables that are "global" to your message passing implementation. This is expected.
-

FAQs/Other questions.

- Your program should not freeze or hang for any set of reasonable inputs.
- Think about "shared variables" and how "synchronized" your threads need to be to insure identical numerical results to the single threaded code
- Semantics of the send/recv operations are similar to MPI
- Message matching – the length of buffers is NOT a criterion for message matching. Only tags and sources are used for message matching. Wildcarding of tag and sender are allowed only in receives.
- If two or more outstanding receives match a particular send, your code may choose to complete the message using any matching receive.
- You may use `assert`() to capture any user errors in calling the API. Buffers should be non-null. Tags are nonnegative, ids (src, dest) must be in the range 0..(MP_Size()-1).
- If the receive buffer does not have enough space for the matching send, copy on the amount of data that will fit. This is a user error, but should generate no error (or memory overwrite).

What to turn in

- Makefile
- All of your *.c and *.h files needed to compile your code. NO object files, library archives or executable files should be included
- Your graph of msgbench performance over the range of 32 bytes to 1MB (1024*1024) bytes called `msgbench-perf.pdf`.

Turning your program

Just like PR2,PR3,PR4 . Don't forget to push your final changes in Github.

What you will be graded on

- Correctness of your code. If msgbench with validation completes on all inputs, you likely have a pretty good implementation.
- Indentation – Your code must be indented. Choose a style and stick with it.
- Comments – You must comment your code in a reasonable way. At a minimum, your name, ucsd email address, student ID must appear in comments. All your routines must be commented with a brief description of what the routine does, what it returns and a description of the parameters.
- We will test your code against various small benchmarking programs.
- Your library code should not print anything to standard output/standard error.
- Absence of memory leaks. (Validated with valgrind)

Example of running valgrind on ieng6

```
$ valgrind ./msgbench 65536 100 validate
==23844== Memcheck, a memory error detector
==23844== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et
al.
==23844== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright
info
==23844== Command: ./msgbench 65536 100 validate
==23844==
starting initialize
starting bench
65536 bytes (100 trials) 0.003600 secs/trial  3.64089e+07 Bytes/s
==23844==
==23844== HEAP SUMMARY:
==23844==     in use at exit: 0 bytes in 0 blocks
==23844==   total heap usage: 8 allocs, 8 frees, 197,924 bytes
allocated
==23844==
==23844== All heap blocks were freed -- no leaks are possible
==23844==
==23844== For counts of detected and suppressed errors, rerun with: -v
==23844== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
0)
[root@rocks-76 PR5]#
```

UPDATES/CORRECTIONS

Significant updates/corrections to the assignment will be put in a pinned message on Piazza.