HW5 Parallel Programming

學號: 112062532　　**姓名:** 温佩旻

1. Overview

> In conjunction with the UCP architecture mentioned in the lecture, please read ucp_hello_world.c

1. Identify how UCP Objects (`ucp_context` , `ucp_worker` , `ucp_ep`) interact through the API, including at least the following functions:

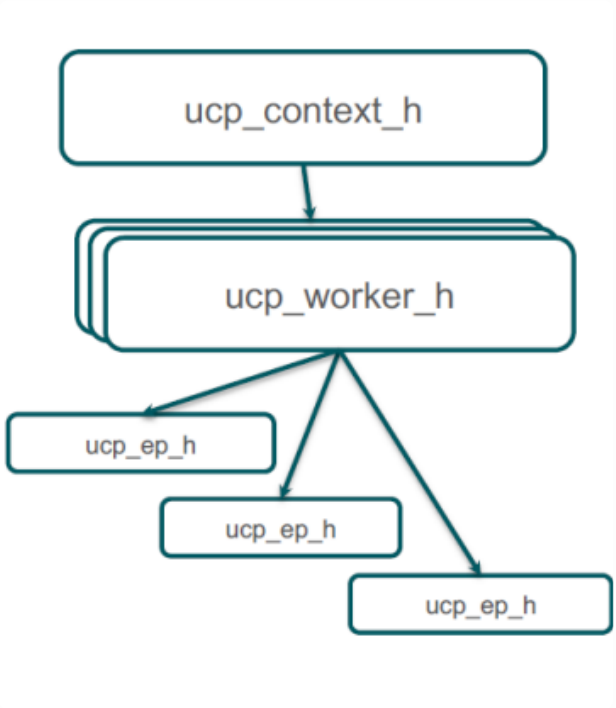
   - `ucp_init`
     `ucp_context_h` 是負責管理 UCP application 的global context的物件指標，透過call ucp_init API，並傳入 ucp_params_t，可以創建並初始化 `ucp_context_h` 。
   - `ucp_worker_create`
     透過call ucp_worker_create，可以基於已透過 ucp_init 初始化的 `ucp_context` ，以及設定好的 `worker_params` （用來指定 thread mode 的參數），來創建並初始化 `ucp_worker` object。
   - `ucp_ep_create`
     `ucp_ep` 代表與遠端節點的connection，透過 ep_params（包含client的 UCX address與其他資訊）以及已經初始化的 `ucp_worker` ，來初始化 `ucp_ep` object，建立與遠端節點的連接。

   完成上述三個API的初始化後，application就可以利用 `ucp_worker` 來和不同的client communication，其中一個 `ucp_worker` 可以看成是負責處理通訊的一個thread，而 `ucp_ep` 表示的是該thread和其他process的connection。

2. UCX abstracts communication into three layers as below. Please provide a diagram illustrating the architectural design of UCX.

   - `ucp_context`
     提供application的global context，管理 UCX 的初始化和資源配置，包括記憶體註冊和底層硬體的資訊，每個application只需創建一個 `ucp_context` ，負責生成 `ucp_worker` 。
   - `ucp_worker`
     代表communication的執行單元，負責管理與傳輸相關的資源，通常一個thread對應到一個 `ucp_worker` ，每個worker可以有多個 `ucp_ep` ，每個worker只能用一種transport方法，要處理怎麼對endpoint去做傳輸。

- `ucp_ep`
  UCP endpoint，表示從本地 worker 到遠端 worker 的連接，當每個worker需要和遠端進行communication時，就會建立一個 `ucp_ep` ，每個worker可以有很多個 `ucp_ep` ，和不同host連接，主要用於發送和接收data，並由 `ucp_worker_h` 負責建立。



> Please provide detailed example information in the diagram corresponding to the execution of the command `srun -N 2 ./send_recv.out` or `mpiucx --host HostA:1,HostB:1 ./send_recv.out`

- `mpiucx --host HostA:1,HostB:1 ./send_recv.out`
  指定 HostA 和 HostB 各執行 1 個process，HostA 和 HostB 各創建自己的 `ucp_context_h` 和 `ucp_worker_h` ，HostA 與 HostB 之間創建對應的 `ucp_ep_h` 來建立connection，通過 `ucp_ep_h` 進行點對點的資料交換。當Host A 的 `ucp_worker_h` 使用 `ucp_ep_h` 發送data時，data會經由 RDMA 傳輸到Host B 的 `ucp_worker_h` ，由該 worker 處理後上傳至應用層。

3. Based on the description in HW5, where do you think the following information is loaded/created?
   - `UCX_TLS`
     TLS是所有目前系統能夠選擇的transport 方法，所以我認為應該是 `ucp_context` 在初始化階段的時候load進來的。
   - TLS selected by UCX
     這邊是指實際被選擇的TLS，我認為應該是 `ucp_worker` 處理的，因為每個 `ucp_worker` 可以選用不同的transport 方法。

2. Implementation

1. Which files did you modify, and where did you choose to print Line 1 and Line 2?
    - **print Line 1:**
        - 修改 ucp_worker.c 這份檔案。

```
2034  ucs_status_t ucp_worker_get_ep_config(ucp_worker_h worker,
2035                                        const ucp_ep_config_key_t *key,
2036                                        unsigned ep_init_flags,
2037                                        ucp_worker_cfg_index_t *cfg_index_p)
2038  {
2039      ucp_context_h context = worker->context;
2040      ucp_worker_cfg_index_t ep_cfg_index;
2041      ucp_ep_config_t *ep_config;
2042      ucp_memtype_thresh_t *tag_max_short;
2043      ucp_lane_index_t tag_exp_lane;
2044      unsigned tag_proto_flags;
2045      ucs_status_t status;
2046      ucp_config_t *config;
2047
2048      ucs_assertv_always(key->num_lanes > 0,
2049                         "empty endpoint configurations are not allowed");
2050
2051      /* Search for the given key in the ep_config array */
2052      ucs_array_for_each(ep_config, &worker->ep_config) {
2053          if (ucp_ep_config_is_equal(&ep_config->key, key)) {
2054              ep_cfg_index = ep_config - worker->ep_config.buffer;
2055              goto out;
2056          }
2057      }
2058
2059      /* Create new configuration */
2060      ucs_array_append(ep_config_arr, &worker->ep_config,
2061                       return UCS_ERR_NO_MEMORY);
2062      if (ucs_array_length(&worker->ep_config) >= UCP_WORKER_MAX_EP_CONFIG) {
2063          ucs_array_pop_back(&worker->ep_config);
2064          ucs_error("too many ep configurations: %d (max: %d)",
2065                    ucs_array_length(&worker->ep_config),
2066                    UCP_WORKER_MAX_EP_CONFIG);
2067          return UCS_ERR_EXCEEDS_LIMIT;
2068      }
2069
2070      ep_config = ucs_array_last(&worker->ep_config);
2071      status    = ucp_ep_config_init(worker, ep_config, key);
2072      if (status != UCS_OK) {
2073          return status;
2074      }
2075
2076      ep_cfg_index = ucs_array_length(&worker->ep_config) - 1;
2077
2078      if (ep_init_flags & UCP_EP_INIT_FLAG_INTERNAL) {
2079          /* Do not initialize short protocol thresholds for internal endpoints,
2080           * and do not print their configuration
2081           */
2082          goto out;
2083      }
2084
2085      if (context->config.ext.proto_enable) {
2086          if (ucp_ep_config_key_has_tag_lane(key)) {
2087              tag_proto_flags = UCP_PROTO_FLAG_TAG_SHORT;
2088              tag_max_short   = &ep_config->tag.offload.max_eager_short;
2089              tag_exp_lane    = key->tag_lane;
2090          } else {
2091              tag_proto_flags = UCP_PROTO_FLAG_AM_SHORT;
2092              tag_max_short   = &ep_config->tag.max_eager_short;
2093              tag_exp_lane    = key->am_lane;
2094          }
2095
2096          ucp_worker_ep_config_short_init(worker, ep_config, ep_cfg_index,
2097                                          UCP_FEATURE_TAG, UCP_OP_ID_TAG_SEND,
2098                                          tag_proto_flags, tag_exp_lane,
2099                                          tag_max_short);
2100
2101          ucp_worker_ep_config_short_init(worker, ep_config, ep_cfg_index,
2102                                          UCP_FEATURE_AM, UCP_OP_ID_AM_SEND,
2103                                          UCP_PROTO_FLAG_AM_SHORT, key->am_lane,
2104                                          &ep_config->am_u.max_eager_short);
2105      } else {
2106          ucp_config_read(NULL, NULL, &config);
2107          ucp_config_print(config, stdout, NULL, UCS_CONFIG_PRINT_TLS);
2108          ucp_worker_print_used_tls(worker, ep_cfg_index);
2109      }
```

圖 1

我在ucp_worker.c 中的 ucp_worker_get_ep_config這個function中新增 `ucp_config_read` 來取得UCX_TLS相關資訊，接著會 call `ucp_config_print` ，透

過 `parser.c` 的 ucs_config_parser_print_opts 這個function將取得的TLS資料印出。

- 修改 parser.c 這份檔案。

```
1853  void ucs_config_parser_print_opts(FILE *stream, const char *title, const void *opts,
1854                                    ucs_config_field_t *fields, const char *table_prefix,
1855                                    const char *prefix, ucs_config_print_flags_t flags)
1856  {
1857      ucs_config_parser_prefix_t table_prefix_elem;
1858      UCS_LIST_HEAD(prefix_list);
1859      char str[100];
1860
1861      if (flags & UCS_CONFIG_PRINT_DOC) {
1862          fprintf(stream, "# UCX library configuration file\n");
1863          fprintf(stream, "# Uncomment to modify values\n");
1864      }
1865
1866      if (flags & UCS_CONFIG_PRINT_HEADER) {
1867          fprintf(stream, "\n");
1868          fprintf(stream, "#\n");
1869          fprintf(stream, "# %s\n", title);
1870          fprintf(stream, "#\n");
1871          fprintf(stream, "\n");
1872      }
1873
1874      if (flags & UCS_CONFIG_PRINT_CONFIG) {
1875          table_prefix_elem.prefix = table_prefix ? table_prefix : "";
1876          ucs_list_add_tail(&prefix_list, &table_prefix_elem.list);
1877          ucs_config_parser_print_opts_recurs(stream, opts, fields, flags,
1878                                              prefix, &prefix_list);
1879      }
1880
1881      // TODO: PP-HW-UCX
1882      if (flags & UCS_CONFIG_PRINT_TLS) {
1883          ucs_config_parser_get_value((void *)opts, fields, "TLS", str, 100 * sizeof(char));
1884          fprintf(stream, "UCX_TLS=%s\n", str);
1885      }
1886
1887      if (flags & UCS_CONFIG_PRINT_HEADER) {
1888          fprintf(stream, "\n");
1889      }
1890  }
```

圖 2

在ucs_config_parser_print_opts 中完成上圖的TODO，印出UCX_TLS

- 修改 types.h 這份檔案。

```
88  typedef enum {
89      UCS_CONFIG_PRINT_CONFIG           = UCS_BIT(0),
90      UCS_CONFIG_PRINT_HEADER           = UCS_BIT(1),
91      UCS_CONFIG_PRINT_DOC              = UCS_BIT(2),
92      UCS_CONFIG_PRINT_HIDDEN           = UCS_BIT(3),
93      UCS_CONFIG_PRINT_COMMENT_DEFAULT  = UCS_BIT(4),
94      UCS_CONFIG_PRINT_TLS              = UCS_BIT(5)
95  } ucs_config_print_flags_t;
```

圖 3

由於在ucp_worker.c 和 parser.c 都會用到 `UCS_CONFIG_PRINT_TLS` ，所以需要在 types.h 中的 ucs_config_print_flags_t 加入 `UCS_CONFIG_PRINT_TLS` ，如上圖所示。

- **print Line 2:**
  修改ucp_worker.c這份檔案。
  從助教的提示中可以觀察到在 run `mpiucx -x UCX_LOG_LEVEL=info -np 2 ./mpi_hello.out` 的時候， `ucp_worker.c` 的ucp_worker_print_used_tls function 中就有印出TLS selected by UCX的資訊，所以在print Line 2的部分就直接在 `ucs_info("%s", ucs_string_buffer_cstr(&strb));` 的後面加上一行 `printf("%s\n", ucs_string_buffer_cstr(&strb));` 就能印出SPEC指定的line 2資訊。

2. How do the functions in these files call each other? Why is it designed this way?
   在 `ucp_worker.c` 的 `ucp_worker_get_ep_config` 中會先call `ucp_config_read` (圖1的2106行) 來取得UCS_TLS的資訊，接著會 call `ucp_config_print` (圖1的2107行) ，`ucp_config_print` 裡面會再call `ucs_config_parser_print_opts` (圖2)，將TLS的資訊印出，接著 `ucp_worker.c` 的 `ucp_worker_get_ep_config` 會再call `ucp_worker_print_used_tls` (圖1的2108行) ，將目前所使用的TLS資訊印出。

   由於前面說到的，透過助教的提示中可以觀察到在 run `mpiucx -x UCX_LOG_LEVEL=info -np 2 ./mpi_hello.out` 的時候，`ucp_worker.c` 的ucp_worker_print_used_tls 會印出目前所使用的TLS的資訊，因此只需要用同一個function所獲的的資訊直接印出Line 2就好。接著透過trace code發現 `ucp_worker_get_ep_config` 會 call `ucp_worker_print_used_tls` ，因此我在 call `ucp_worker_print_used_tls` 之前將全部可用的TLS的資訊印出，完成Line 1的實作。

3. Observe when Line 1 and 2 are printed during the call of which UCP API?
   從上面的敘述中可以發現我的Line 1和Line 2都寫在ucp_worker_get_ep_config 這個function當中，從這個function回朔回去，可以觀察到最初call的UCP API 是 ucp_ep_create 這個API，在 `ucp_ep_create` 中會call ucp_ep_create_to_sock_addr，接著再從 `ucp_ep_create_to_sock_addr` 中 call ucp_ep_init_create_wireup，最後在 `ucp_ep_init_create_wireup` function 裡面call ucp_worker_get_ep_config，然後就可以印出Line 1和Line 2。

4. Does it match your expectations for questions 1-3? Why?

   - `UCX_TLS`
     在前面我推測這是會在 `ucp_context` 中被 load 進來，因為 `UCX_TLS` 是 global資訊，trace code 後也確實是如此，在 ucp_init function 中會將透過 `ucp_config_read` 得到的UCX_TLS讀到 `ucp_context` 。
     不過實作中因為在 `ucp_worker_get_ep_config` 才會取得真正被取用的TLS，所以我將

`ucp_config_read` 及 UCX_TLS 印出的地方寫在 `ucp_worker_get_ep_config` 的 `ucp_worker_print_used_tls` 之前。

- TLS selected by UCX
  在前面我推測是 `ucp_worker` 處理的，但 trace code 後發現 `ucp_worker` 是提供了所有可用的 TLS（例如 TCP、RDMA等），`ucp_ep` 才從這些可用的 TLS 中選擇最終要用的協議。從 ucp_worker_get_ep_config 傳入的參數 key ，查找 `worker->ep_config` 中是否已有相同的 key，如果找到，代表已經配置過了，用 `ep_cfg_index` 獲得現有配置的 index，則最終協議已經由之前的初始化決定，如果沒有找到則透過 ucp_ep_config_init 來創建新配置，根據 key 初始化每個Lane的協議，並存入 `ep_config` ，作為該端點的最終協議。

5. In implementing the features, we see variables like lanes, tl_rsc, tl_name, tl_device, bitmap, iface, etc., used to store different Layer's protocol information. Please explain what information each of them stores.

   - **lanes:** 每個 Lane 對應到 UCX 中的一個傳輸層協議（如: TCP、RDMA）以及具體的設備（如 eth0 或 mlx5_0），用來決定數據傳輸中使用的傳輸協議和設備，例如: lanes[0] 對應 TCP，lanes[1] 對應 RDMA。
   - **tl_rsc:** transport layer的相關資訊，包括tl_name、tl_device。
   - **tl_name:** transport layer的名字，像是ud_verbs、tcp、rdma。
   - **tl_device:** transport layer可以使用的具體device名稱，像是eth0、mlx5_0。
   - **bitmap:** 表示可用資源或 Lane 的狀態，每一bit對應於一個資源（如 Lane 或傳輸協議）的啟用或禁用狀態，能用來快速檢查哪些 Lane 或資源是可用的。
   - **iface:** 用來管理數據的發送和接收，每個 iface 由 Lane 綁定到特定的Transport Layer和device，例如iface 可能是 TCP 的 socket entity 或 RDMA 的 QP（Queue Pair）。

3. Optimize System

1. Below are the current configurations for OpenMPI and UCX in the system. Based on your learning, what methods can you use to optimize single-node performance by setting UCX environment variables?

```
-------------------------------------------------------------------
/opt/modulefiles/openmpi/ucx-pp:

module-whatis   {OpenMPI 4.1.6}
conflict        mpi
module          load ucx/1.15.0
prepend-path    PATH /opt/openmpi-4.1.6/bin
prepend-path    LD_LIBRARY_PATH /opt/openmpi-4.1.6/lib
prepend-path    MANPATH /opt/openmpi-4.1.6/share/man
prepend-path    CPATH /opt/openmpi-4.1.6/include
setenv          UCX_TLS ud_verbs
```

```
setenv           UCX_NET_DEVICES ibp3s0:1
-----------------------------------------------------------------------
```

可以看到apollo預設是將UCX_TLS設定為 ud_verbs，ud_verbs 會使用RDMA，而RDMA需要到 remote 將memory copy過來，進而產生 memory copy 的 overhead，這邊因為是single-node，我認為可以考慮設成shared memory，這樣可以減少 memory copy 的 overhead，這樣應該會比走網路的速度要來的快。

2. Please use the following commands to test different data sizes for latency and bandwidth, to verify your ideas:

```
module load openmpi/ucx-pp
mpiucx -n 2 $HOME/UCX-lsalab/test/mpi/osu/pt2pt/osu_latency
mpiucx -n 2 $HOME/UCX-lsalab/test/mpi/osu/pt2pt/osu_bw
```

圖4、圖5是使用UCX預設的 `UCX_TLS=ud_verbs` 去測試 osu_latency(圖4) 及 osu_bandwidth(圖5) 的結果。圖6、圖7則是將UCX_TLS設定為all，讓UCX去選擇最好的TLS，可以看到UCX都是選擇shared memory，等同於 `UCX_TLS=sm`，這邊可以觀察到使用 `UCX_TLS=all` 能讓latency減少2~3倍，也讓 bandwidth 增加3倍多，根據這些結果表示了在 single-node 上採用 shared memory 確實可有效減少 latency 及增加 bandwidth。

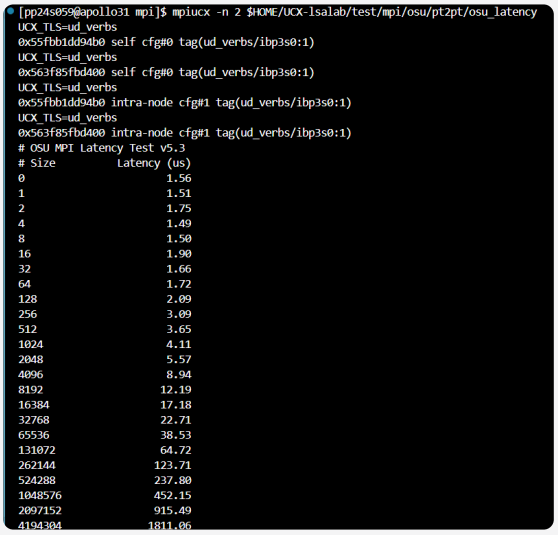


圖 4

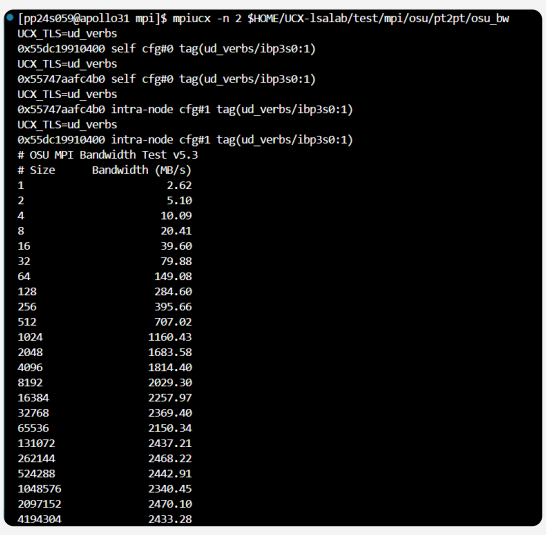
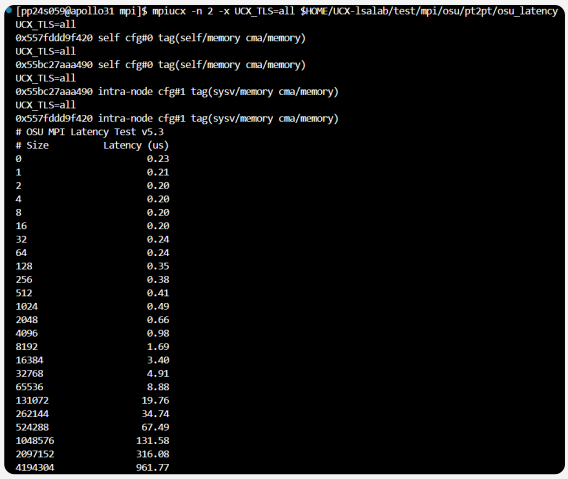
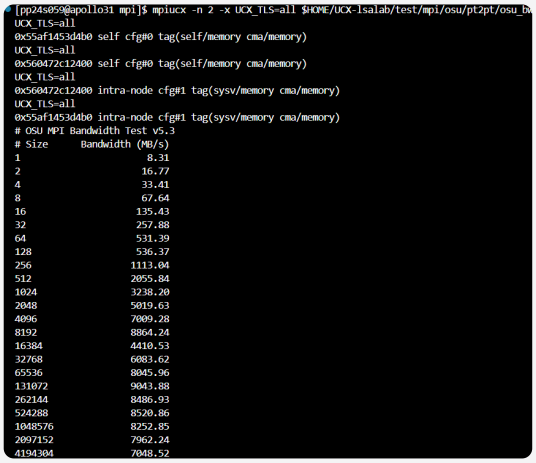


圖 5



```
[pp24s059@apollo31 mpi]$ mpiucx -n 2 -x UCX_TLS=all $HOME/UCX-lsalab/test/mpi/osu/pt2pt/osu_latency
UCX_TLS=all
0x557fddd9f420 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x55bc27aaa490 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x55bc27aaa490 intra-node cfg#1 tag(sysv/memory cma/memory)
UCX_TLS=all
0x557fddd9f420 intra-node cfg#1 tag(sysv/memory cma/memory)
# OSU MPI Latency Test v5.3
# Size          Latency (us)
0                       0.23
1                       0.21
2                       0.20
4                       0.20
8                       0.20
16                      0.20
32                      0.24
64                      0.24
128                     0.35
256                     0.38
512                     0.41
1024                    0.49
2048                    0.66
4096                    0.98
8192                    1.69
16384                   3.40
32768                   4.91
65536                   8.88
131072                 19.76
262144                 34.74
524288                 67.49
1048576               131.58
2097152               316.08
4194304               961.77
```

圖 6



```
[pp24s059@apollo31 mpi]$ mpiucx -n 2 -x UCX_TLS=all $HOME/UCX-lsalab/test/mpi/osu/pt2pt/osu_b
UCX_TLS=all
0x55af1453d4b0 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x560472c12400 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x560472c12400 intra-node cfg#1 tag(sysv/memory cma/memory)
UCX_TLS=all
0x55af1453d4b0 intra-node cfg#1 tag(sysv/memory cma/memory)
# OSU MPI Bandwidth Test v5.3
# Size      Bandwidth (MB/s)
1                       8.31
2                      16.77
4                      33.41
8                      67.64
16                    135.43
32                    257.88
64                    531.39
128                   536.37
256                  1113.04
512                  2055.84
1024                 3238.20
2048                 5019.63
4096                 7009.28
8192                 8864.24
16384                4410.53
32768                6083.62
65536                8045.96
131072               9043.88
262144               8486.93
524288               8520.86
1048576              8252.85
2097152              7962.24
4194304              7048.52
```

圖 7

size	latency_ud_verbs (us)	latency_all (us)
0	1.56	0.23
1	1.51	0.21
2	1.75	0.2
4	1.49	0.2
8	1.5	0.2
16	1.9	0.2
32	1.66	0.24
64	1.72	0.24
128	2.09	0.35
256	3.09	0.38
512	3.65	0.41
1024	4.11	0.49
2048	5.57	0.66
4096	8.94	0.98
8192	12.19	1.69
16384	17.18	3.4
32768	22.71	4.91
65536	38.53	8.88
131072	64.72	19.76
262144	123.71	34.74
524288	237.8	67.49
1048576	452.15	131.58
2097152	915.49	316.08
4194304	1811.06	961.77

圖 8 `UCX_TLS=ud_verbs` 和 `UCX_TLS=all` latency 比較

size	bandwidth_ud_verbs (MB/s)	bandwidth_all (MB/s)
1	2.62	8.31
2	5.1	16.77
4	10.09	33.41
8	20.41	67.64
16	39.6	135.43
32	79.88	257.88
64	149.08	531.39
128	284.6	536.37
256	395.66	1113.04
512	707.02	2055.84
1024	1160.43	3238.2
2048	1683.58	5019.63
4096	1814.4	7009.28
8192	2029.3	8864.24
16384	2257.97	4410.53
32768	2369.4	6083.62
65536	2150.34	8045.96
131072	2437.21	9043.88
262144	2468.22	8486.93
524288	2442.91	8520.86
1048576	2340.45	8252.85
2097152	2470.1	7962.24
4194304	2433.28	7048.52

圖 9 `UCX_TLS=ud_verbs` 和 `UCX_TLS=all` bandwidth 比較

接著我觀察到 `UCX_TLS=sm` 時同時啟用了 `cma/memory` 和 `sysv/memory`，所以測試了 `UCX_TLS=sm` 和 `UCX_TLS=sysv` latency 和 bandwidth 的比較，將結果用圖10、圖11兩張表格呈現，可以發現在 single-node 上，數據量越大， `UCX_TLS=sysv` 在 latency 和 bandwidth 上的表現相對於 `UCX_TLS=sm` 的表現就越好，因此除了設定 `UCX_TLS=all` 能提高 performance，設定`UCX_TLS=sysv` 在資料量大的時候表現會更不錯。

size	latency_sm (us)	latency_sysv (us)
0	0.2	0.2
1	0.2	0.21
2	0.2	0.2
4	0.23	0.2
8	0.23	0.2
16	0.21	0.2
32	0.25	0.24
64	0.25	0.23
128	0.36	0.36
256	0.38	0.38
512	0.41	0.41
1024	0.5	0.49
2048	0.66	0.65
4096	0.97	0.98
8192	1.67	1.68
16384	2.99	3.01
32768	4.95	4.91
65536	8.68	8.72
131072	17.17	17.11
262144	38.04	33.43
524288	66.26	66.13
1048576	131.74	123.49
2097152	316.5	281.65
4194304	964	762.26

圖 10 `UCX_TLS=sm` 和 `UCX_TLS=sysv` latency 比較

size	bandwidth_sm (MB/s)	bandwidth_sysv (MB/s)
1	10.11	9.31
2	20.41	18.84
4	40.29	40.33
8	82.34	81
16	165.33	164.19
32	310.66	312.74
64	587.2	635.48
128	622.28	472.82
256	1264.47	1223.02
512	2379.38	2340.15
1024	3819.78	3855.84
2048	5744.72	5762.63
4096	7766.99	8213.92
8192	9955.32	10214.5
16384	4921.55	9532.82
32768	6684.11	10067.38
65536	8125.87	10601.41
131072	8270.03	10583.68
262144	8610.91	8792.06
524288	8313.76	8870.25
1048576	8179.07	8798.73
2097152	8308.15	8668.98
4194304	6947.23	7732.45

圖 11 `UCX_TLS=sm` 和 `UCX_TLS=sysv` bandwidth 比較

3. Please create a chart to illustrate the impact of different parameter options on various data sizes and the effects of different testsuite.
   因為 `UCX_TLS=all` 選出來的結果基本上就是 `UCX_TLS=sm` ，所以這邊我比較 `UCX_TLS=sm` 、`UCX_TLS=sysv` 、`UCX_TLS=ud_verbs` 三種設定隨著 size 不同，latency 和 bandwidth 的相關性為何。

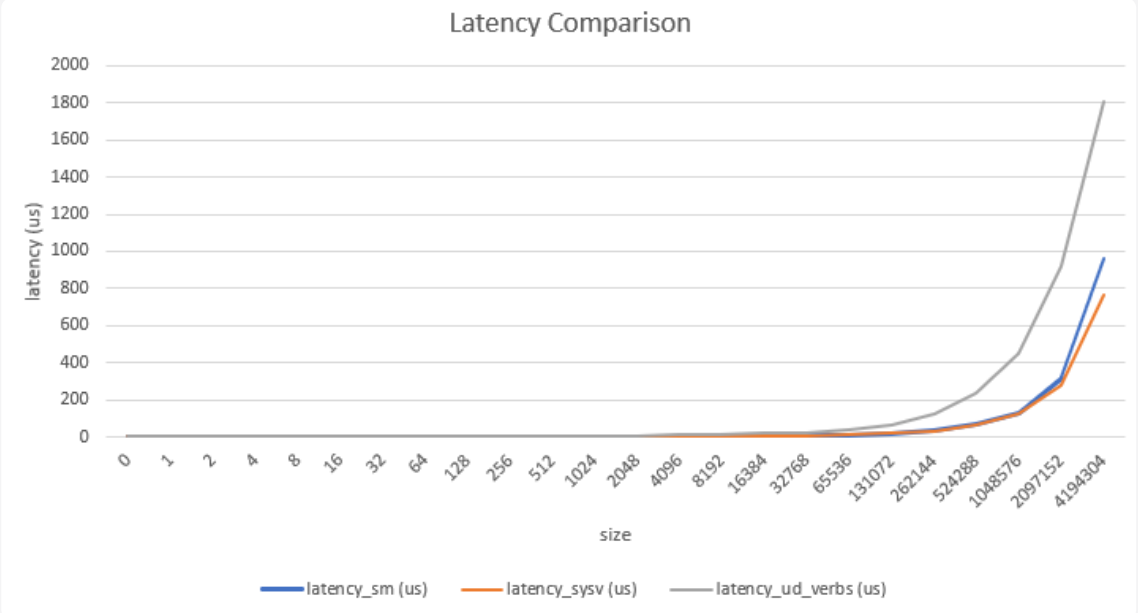


圖 12 `UCX_TLS=sm`、`UCX_TLS=sysv`、`UCX_TLS=ud_verbs` latency 比較

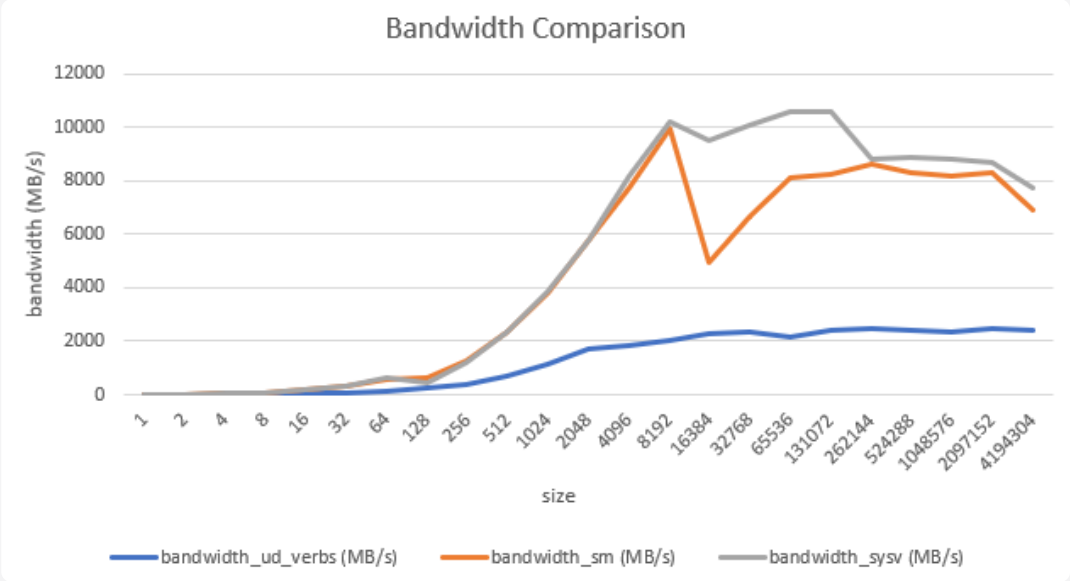


圖 13 `UCX_TLS=sm`、`UCX_TLS=sysv`、`UCX_TLS=ud_verbs` bandwidth 比較

4. Based on the chart, explain the impact of different TLS implementations and hypothesize the possible reasons (references required).
   從圖12可以發現，`UCX_TLS=ud_verbs` 隨著數據大小增加，latency會急劇上升，尤其是數據量愈大攀升的幅度越明顯，相較之下，`UCX_TLS=sm` 和 `UCX_TLS=sysv` 就相對穩定一些，這樣的原因是因為 ud_verbs 是基於Unreliable Datagram 協議，數據傳輸需要經過網路(如RDMA網卡)，在大數據傳輸下，網路傳輸延遲、網路資源競爭等等的開銷會更明顯，所以相比於shared memory就會慢的許多。
   同時也可以發現在大數據量下，latency_sm 比 latency_sysv 略高，顯示 sysv 在大數據下的延遲更穩定一些，這可能是因為 `UCX_TLS=sm` 同時啟用了多種shared memory機制，如 `cma/memory` 和 `sysv/memory`，在大數據的傳輸時，CMA 可能會觸發頻繁的context switch，導致額外的overhead，進而使 latency_sm 略高於 latency_sysv，而在 `UCX_TLS=sysv` 的模式下，專注於 System V shared memory，所以memory可能更容易被系統優化，能避免額外的context switch，所以在大數據的情況下 `UCX_TLS=sysv` 表現較好。
   從圖13可以發現到 `UCX_TLS=ud_verbs` 隨著數據量提高 bandwidth 上升的幅度並不明顯，可以和上述latency結果呼應，由於延遲過高，單位時間內能夠處理的數據量就會受到限制，原因也是因為網路層的限制，加上不保證數據順序和可靠性，在大數據量下會增加數據包重傳的可能性，更進一步降低有效bandwidth。
   而 `UCX_TLS=sm` 則是看起來波動比較明顯，這可能是因為上面提到的， UCX 在 `sm` 模式下會去嘗試多種 shared memory 機制，在不同數據量會切換不同機制，導致性能不穩定，另外可以觀察到 `UCX_TLS=sysv` 在大概中等數據量的時候 bandwidth 就已經明顯高於另外兩個了，顯示出 sysv 協議在大數據傳輸時表現較穩定。

Advanced Challenge: Multi-Node Testing

This challenge involves testing the performance across multiple nodes. You can accomplish this by utilizing the sbatch script provided below. The task includes creating tables and providing explanations based on your findings. Notably, Writing a comprehensive report on this exercise can earn you up to 5 additional points.

- For information on sbatch, refer to the documentation at Slurm's sbatch page.
- To conduct multi-node testing, use the following command:

```
cd ~/UCX-lsalab/test/
sbatch run.batch
```

圖14是用run.batch來跑 Multi-Node，設定 `UCX_TLS=all` 在 latency 的結果，圖15是single-node和multi-node 的latency 表格，可以觀察到，`UCX_TLS=all` 在 multi-node 環境下執行時，Latency 明顯比 single-node 環境高，這是因為在 Single-Node 的環境下UCX使用的是shm協議，不需要透過網路進行傳輸，而是直接通過 shared memory 來進行傳輸，但在 Multi-Node 的環境下，UCX會選擇 RDMA (rc_verbs) 和 TCP 作為主要的傳輸協議，這需要通過InfiniBand 或 Ethernet TCP/IP 來傳輸數據，也就是在 Multi-Node 還需要網卡與CPU之間的 memory 交換，會產生額外的 CPU overhead。

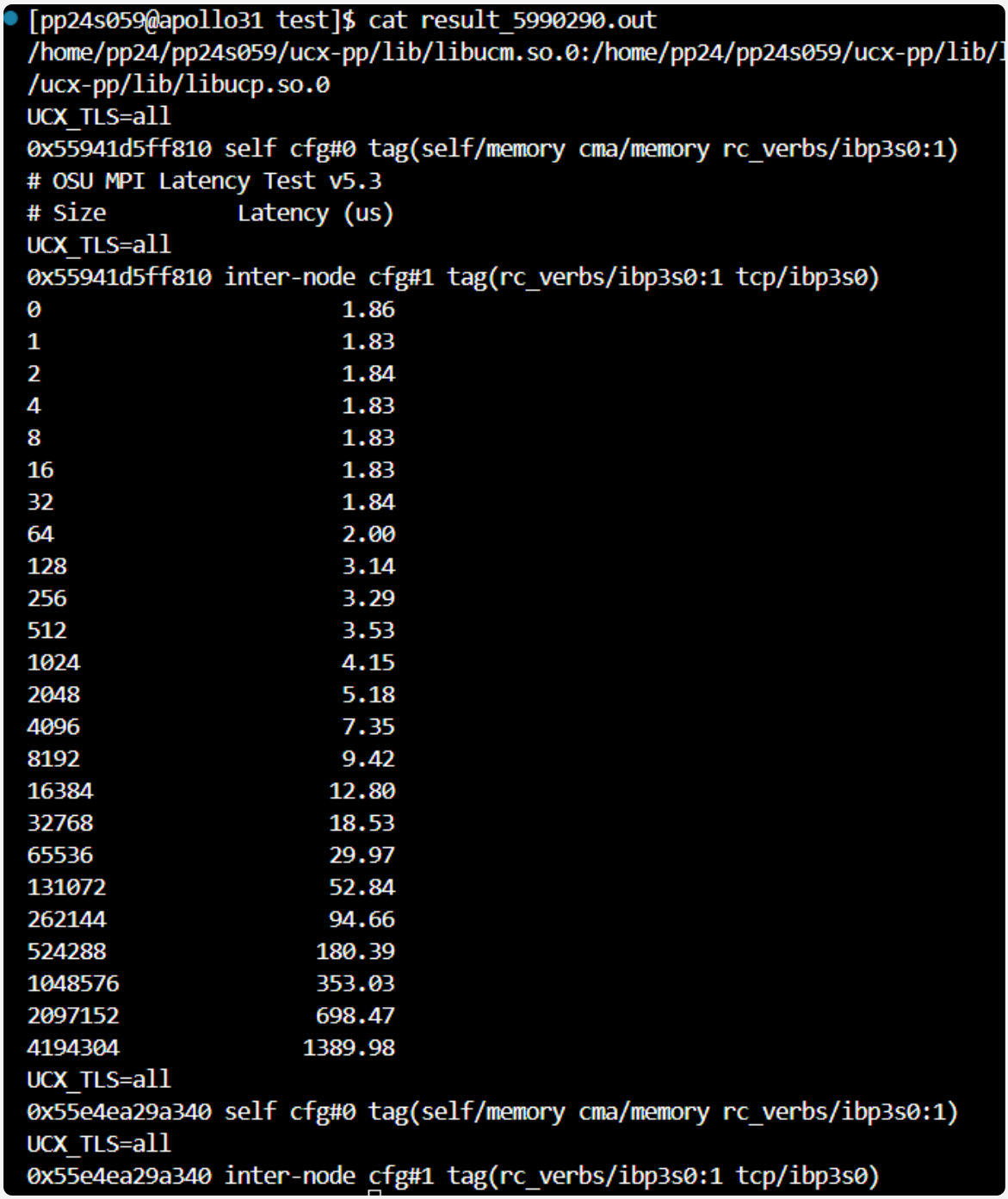


圖 14 Multi-Node 設定 `UCX_TLS=all` 在osu_latency表現

size	multi node_latency_all (us)	single node_latency_all (us)
0	1.86	0.23
1	1.83	0.21
2	1.84	0.2
4	1.83	0.2
8	1.83	0.2
16	1.83	0.2
32	1.84	0.24
64	2	0.24
128	3.14	0.35
256	3.29	0.38
512	3.53	0.41
1024	4.15	0.49
2048	5.18	0.66
4096	7.35	0.98
8192	9.42	1.69
16384	12.8	3.4
32768	18.53	4.91
65536	29.97	8.88
131072	52.84	19.76
262144	94.66	34.74
524288	180.39	67.49
1048576	353.03	131.58
2097152	698.47	316.08
4194304	1389.98	961.77

圖 15 Single-Node 和 Multi-Node 設定 `UCX_TLS=all` latency 比較

圖16是用run.batch來跑 Multi-Node，設定 `UCX_TLS=all` 在 bandwidth 的結果，圖17是single-node 和 multi-node 的 bandwidth 表格，這邊呈現的結果大致上可以和上面 latency 的結果相呼應，在所有 size 下，Single-Node 的 bandwidth 都顯著高於 Multi-Node ，主要原因和上述一樣，因為 Multi-Node 會用到網路傳輸，即使網路 bandwidth 有被充分利用，但上限也遠低於 shared memory 傳輸，導致 bandwidth 上升幅度有限。

```
[pp24s059@apollo31 test]$ cat result_5990648.out
/home/pp24/pp24s059/ucx-pp/lib/libucm.so.0:/home/pp24/pp24s059/ucx-pp/lib/
/ucx-pp/lib/libucp.so.0
UCX_TLS=all
0x564581c3a510 self cfg#0 tag(self/memory cma/memory rc_verbs/ibp3s0:1)
# OSU MPI Bandwidth Test v5.3
# Size      Bandwidth (MB/s)
UCX_TLS=all
0x564581c3a510 inter-node cfg#1 tag(rc_verbs/ibp3s0:1 tcp/ibp3s0)
1                       2.90
2                       6.18
4                      12.35
8                      24.58
16                     49.45
32                     98.84
64                    183.08
128                   234.82
256                   455.30
512                   888.33
1024                 1488.85
2048                 2142.82
4096                 2501.86
8192                 2765.50
16384                2863.19
32768                2922.03
65536                2948.62
131072               2963.75
262144               2972.53
524288               3031.20
1048576              3033.68
2097152              3035.27
4194304              3035.81
UCX_TLS=all
0x562771567800 self cfg#0 tag(self/memory cma/memory rc_verbs/ibp3s0:1)
UCX_TLS=all
0x562771567800 inter-node cfg#1 tag(rc_verbs/ibp3s0:1 tcp/ibp3s0)
```

圖 16 Multi-Node 設定 `UCX_TLS=all` 在osu_bandwidth 表現

size	multi node bandwidth_all (MB/s)	single node bandwidth_all (MB/s)
1	2.9	8.31
2	6.18	16.77
4	12.35	33.41
8	24.58	67.64
16	49.45	135.43
32	98.84	257.88
64	183.08	531.39
128	234.82	536.37
256	455.3	1113.04
512	888.33	2055.84
1024	1488.85	3238.2
2048	2142.82	5019.63
4096	2501.86	7009.28
8192	2765.5	8864.24
16384	2863.19	4410.53
32768	2922.03	6083.62
65536	2948.62	8045.96
131072	2963.75	9043.88
262144	2972.53	8486.93
524288	3031.2	8520.86
1048576	3033.68	8252.85
2097152	3035.27	7962.24
4194304	3035.81	7048.52

圖 17 Single-Node 和 Multi-Node 設定`UCX_TLS=all` bandwidth 比較

4. Experience & Conclusion

1. What have you learned from this homework?
   透過這次trace UCX code的作業讓我更了解UCX每個layer的功能和傳輸的架構，原本聽老師上課時感覺很抽象的，比較難理解，但透過實際trace code後會有更具體的了解。
   這次作業也讓我體會到，要提升程式的執行效率，不是只有不斷優化 GPU code，相反的，底層的硬體架構和通訊傳輸機制可能更加重要。

2. How long did you spend on the assignment?
   大約花一個禮拜