PP HW 5 Report Template

> - Please include both brief and detailed answers.
> - The report should be based on the UCX code.
> - Describe the code using the 'permalink' from GitHub repository.

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_worker_create`
    - `ucp_ep_create`

- `ucp_init()` :
    - 位置

      ```
      // main() 裡
      status = ucp_init(&ucp_params, config, &ucp_context);
      ```

        - ucp_params：使用者根據程式需求所設定的參數，例如要啟用哪種 UCP 功能集、每個 UCX Request 要額外配置多少空間、指定 UCX 分配 Request 時，會呼叫哪個初始化函式等。
        - config：先透過 `ucp_config_read()` 取得的 UCX 設定。
        - ucp_context：輸出參數，用來回傳初始化後的 `ucp_context` ，即整個 application 級別的 UCX 資源管理者。
    - 主要功能
        - 建立並初始化 UCP Application Context，任何和 UCX 相關的動作（建立 `ucp_worker` , `ucp_ep` 等）都必須在這個 Context 之上進行。
        - 檢查 UCX 的 API 版本相容性，確保使用的 API 與目前 UCX 函式庫版本一致或相容。
        - 偵測系統底層有哪些網路介面（TCP、InfiniBand、RoCE 等），以及不同的記憶體裝置，並根據可用的資源進行初始化，以便後續 UCP Worker 能直接使用這些網路或記憶體傳輸機制。

        - 不同應用程式（MPI、OpenSHMEM 等）都可能有不同的需求，`ucp_init()` 會幫忙整理、初始化 UCX 所需的結構，以符合這些應用的通訊需求。

- `ucp_worker_create()`：
    - 位置：

```
// main() 裡
status = ucp_worker_create(ucp_context, &worker_params, &ucp_worker);
```

        - `ucp_context`：傳入一個已初始化的 `ucp_context`。每個 worker 必須隸屬於某個 context，且只能隸屬此單一 context。但一個 context 可以有多個 worker。
        - `worker_params`：主要是設定 thread mode，例如：
            - `UCS_THREAD_MODE_SINGLE`：程式保證只有一個 thread 存取該 worker，不做額外的執行緒保護。
            - `UCS_THREAD_MODE_SERIALIZED` 或 `UCS_THREAD_MODE_MULTI`：若應用程式需要多個 thread 同時操作該 worker，就可能需設定更嚴格的模式，以防止資源衝突。
        - `ucp_worker`：輸出參數，函式完成後會回傳建立好的 worker 物件指標。
    - 主要功能
        - 這個 worker 物件會負責管理底層傳輸資源（如網路介面、記憶體管理等），並在應用程式需要收發資料時，進行事件排程、查詢進度（`ucp_worker_progress()`）等動作。
    - 在 `ucp_hello_world.c` 中，只有建立一個 worker，並在後續的 `run_ucx_client()` 或 `run_ucx_server()` 中，都用同一個 worker 來建立 endpoint 和處理收發。
    - 建立完 `ucp_worker` 後，程式又用 `ucp_worker_query()` 取得這個 worker 對應的通訊位址（`local_addr`），以便稍後交換給遠端。

- `ucp_ep_create()`：
    - function prototype

```
ucs_status_t ucp_ep_create(ucp_worker_h worker,
                           const ucp_ep_params_t *params,
                           ucp_ep_h *ep_p)
```

        - `worker`：指向本地端（local）的 worker。這個 endpoint 只能屬於該 worker，不可跨不同 worker 使用。
        - `params`：
            - 一個 struct（`ucp_ep_params_t`），裡面包含了：

        - 遠端地址 (peer address)：透過 OOB 或其他機制拿到的對端 UCP worker address。
        - 錯誤處理模式、錯誤處理 callback function。
        - 其他參數，如 user data 等。
    - `ep_p`：
        - 輸出參數，函式成功後會將「已建立的 endpoint handle」存到這裡供程式使用。

- 功能與特性
    - 利用傳入的對端地址（peer address）把本地端 worker 與遠端 worker 建立「UCX endpoint」連線。
    - 連線步驟可能在底層進行，例如 TCP/IP 或 InfiniBand 的連線協議。
    - `ucp_ep_create()` 是 non-blocking 的，呼叫後函式會很快返回，不會等到底層通訊連線完全完成。
    - 之後如果底層尚未完成連線，傳送/接收操作可能會有延遲，直到連線就緒才真正傳輸資料。
    - 程式通常需先用 TCP Socket（或其他 out-of-band 方式）交換彼此的 UCP worker address，才能在這裡傳進 `ucp_ep_create()`。

- 在 `run_ucx_client()` 的使用
    - 位置：

      ```
      // run_ucx_client() 裡
      status = ucp_ep_create(ucp_worker, &ep_params, &server_ep);
      ```

    - 在此之前，程式透過 OOB（out-of-band）Socket 連線取得了 server 端的 UCP address (`peer_addr`)。
    - `ep_params` 中的 `address = peer_addr`，表示要連線到對方的 UCX 位址。
    - 創建出的 `server_ep`（型別為 `ucp_ep_h`）即是「本地端 -> 伺服端」的通訊端點。之後的 `ucp_tag_send_nbx()`、`ucp_tag_recv_nbx()` 都會透過這個端點來進行傳輸。
    - 程式中馬上用 `ucp_tag_send_nbx()` 把本地的位址資訊 (`local_addr`) 傳送給伺服端，然後再進入等待接收資料的流程。

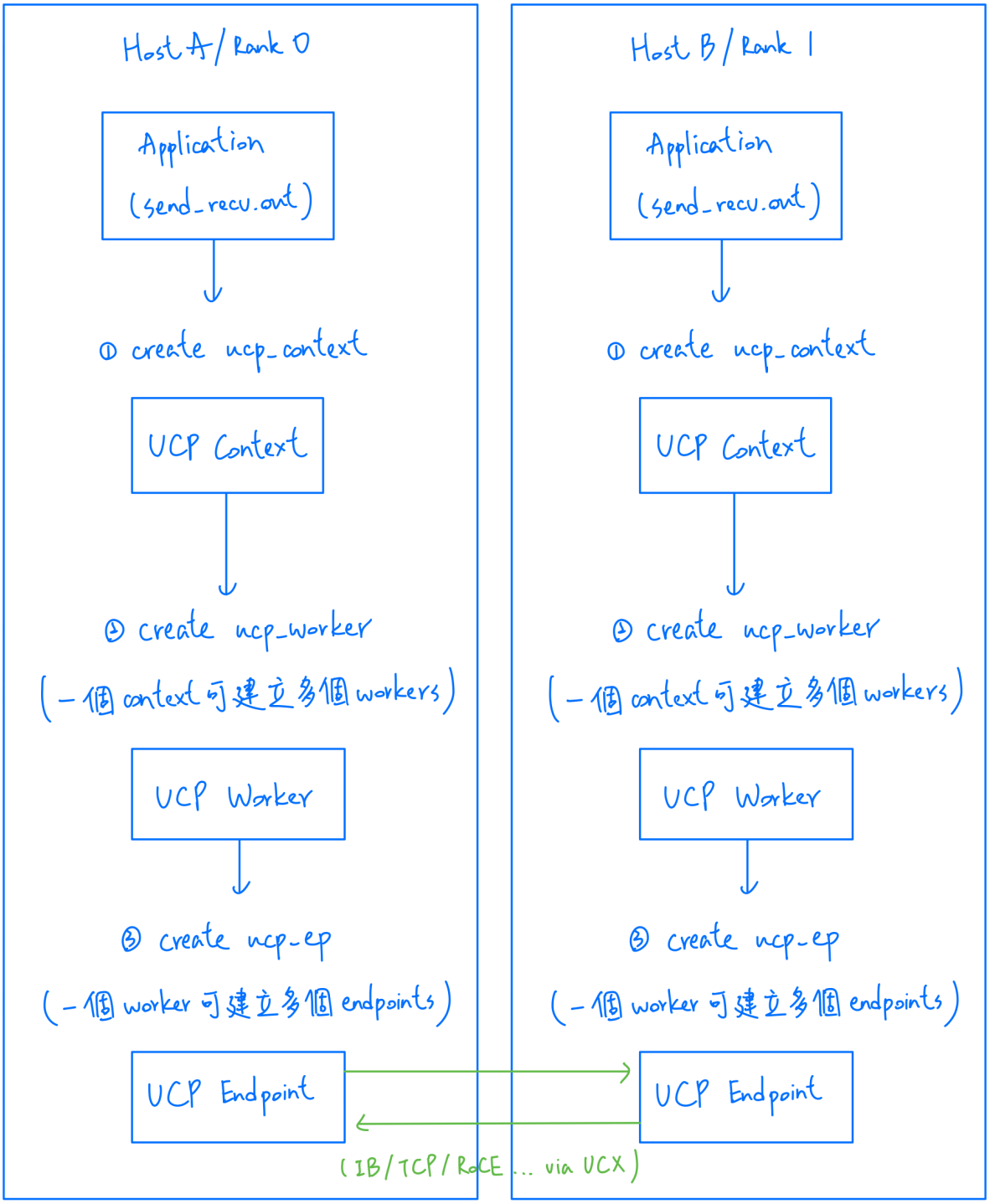
- 在 `run_ucx_server()` 的使用
    - 位置：

      ```
      // run_ucx_server() 裡
      status = ucp_ep_create(ucp_worker, &ep_params, &client_ep);
      ```

- 因為 client 已經建立好對 server 端的連線，所以 server 端可以用 `ucp_tag_probe_nb()`、`ucp_tag_msg_recv_nbx`，接收從客戶端送來的位址資訊，拿到 `peer_addr`（也就是客戶端的位址）。
- 接著一樣設定 `ep_params.address = peer_addr`，並帶有錯誤處理參數後，用 `ucp_ep_create()` 取得 `client_ep`。
- 創建出的 `client_ep` 即是「本地端 (server 端) -> client 端」的通訊端點。接下來，程式會用這個端點傳送測試字串給 client 端。

2. UCX abstracts communication into three layers as below. Please provide a diagram illustrating the architectural design of UCX.
    - `ucp_context`
    - `ucp_worker`
    - `ucp_ep`

> 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`



3. Based on the description in HW5, where do you think the following information is loaded/created?
   - `UCX_TLS`：我猜應該在 `ucp_config_read` 中被載入。
   - TLS selected by UCX：我猜應該在建立 endpoint 的時候被選擇，因為 endpoint 要傳輸就必須決定底層要使用哪種協議。

2. Implementation

> Please complete the implementation according to the spec
> Describe how you implemented the two special features of HW5.

1. Which files did you modify, and where did you choose to print Line 1 and Line 2?

   - UCX-lsalab/src/ucs/config/types.h

```
typedef enum {
    UCS_CONFIG_PRINT_CONFIG          = UCS_BIT(0),
    UCS_CONFIG_PRINT_HEADER          = UCS_BIT(1),
    UCS_CONFIG_PRINT_DOC             = UCS_BIT(2),
    UCS_CONFIG_PRINT_HIDDEN          = UCS_BIT(3),
    UCS_CONFIG_PRINT_COMMENT_DEFAULT = UCS_BIT(4),
    UCS_CONFIG_PRINT_TLS             = UCS_BIT(5)
} ucs_config_print_flags_t;
```

   我在 `types.h` 中加入了一個 `UCS_CONFIG_PRINT_TLS` flag，用於傳入 `parser.c` 中的 `ucs_config_parser_print_opts()`，若 flag 有被設置，function 就會印出目前系統設定的 `UCX_TLS`，也就是作業要求的第一行。

   - UCX-lsalab/src/ucs/config/parser.c

```
// TODO: PP-HW-UCX
if (flags & UCS_CONFIG_PRINT_TLS) {
    const char *val = getenv("UCX_TLS");
    if (val != NULL) {
        printf("UCX_TLS=%s\n", val);
    }
}
```

   我在 `parser.c` 中的 `ucs_config_parser_print_opts()` 中加入了以上程式碼，用於偵測 `UCS_CONFIG_PRINT_TLS` flag 是否有被設置，若有，function 就會用 `getenv()` 取得目前系統設定的 `UCX_TLS` 並印出來，也就是作業要求的第一行。

   - UCX-lsalab/src/ucp/core/ucp_worker.c

```
// 原本程式內容
ucs_string_buffer_rtrim(&strb, "; ");

// 加入兩行程式
ucp_config_print(NULL, stdout, NULL, UCS_CONFIG_PRINT_TLS);
printf("%s\n", ucs_string_buffer_cstr(&strb));

// 原本程式內容
ucs_info("%s", ucs_string_buffer_cstr(&strb));
```

我在 `ucp_worker.c` 中的 `ucp_worker_print_used_tls()` 加入了以上兩行程式，這個 function 的功能是在 UCX 建立 endpoint 後，將實際被啟用的 Lane、功能 (tag/rma/amo/...) 與底層傳輸 (self/shm/cma/verbs/...) 的資訊，彙整並印出的一個 function。

第一行呼叫了 `ucp_config_print()`，這是一個在 `UCX-lsalab/src/ucp/core/ucp_context.c` 中的 function，用來印出存在 data type 為 `ucp_config_t` 的 configuration information，這個 function 會去呼叫 `ucs_config_parser_print_opts()`，從而印出作業要求的第一行輸出。因為 `ucp_config_print()` 是需要 user 自己呼叫的，像在 `ucp_hwllo_world.c` 中就有呼叫，且作業要求在每一個 node 建立每個不同 configuration 的 endpoint 的時候都要印出來，所以才選擇在 `ucp_worker_print_used_tls()` 中印出。

第二行只是將這個 function 原本彙整的 string `strb` 用 `printf` 印出，可以看到下面一行，`ucs_info("%s", ucs_string_buffer_cstr(&strb));`，原本程式碼中就有在 `UCX_LOG_LEVEL` 設為 `info` 時將這個 string 印出，這也是我透過作業提示，將 `UCX_LOG_LEVEL` 設為 `info` 後發現的。

2. How do the functions in these files call each other? Why is it designed this way?

   - 要印出第一行的呼叫順序：`ucp_ep_create()` -> `ucp_ep_create_to_sock_addr()` -> `ucp_ep_init_create_wireup()` -> `ucp_worker_get_ep_config()` -> `ucp_worker_print_used_tls()` -> `ucp_config_print()` -> `ucs_config_parser_print_opts()` -> `printf()`
   - 要印出第二行的呼叫順序：`ucp_ep_create()` -> `ucp_ep_create_to_sock_addr()` -> `ucp_ep_init_create_wireup()` -> `ucp_worker_get_ep_config()` -> `ucp_worker_print_used_tls()` -> `printf()`

   為什麼這樣設計？我覺得是因為：

   UCX 主要分為三層：UCS、UCT、UCP，各有不同的職責：

   - UCP 是最上層 API，application 主要呼叫這層提供的函式。
   - UCS 負責系統設定、logging，以及其他底層服務。
   - UCT 封裝裝置層的通訊功能（verbs, shm, TCP 等）。
   - UCP 跟 UCT 可能都會使用到 UCS 的服務，像是 `ucp_worker_print_used_tls()` 所在的 `parser.c` 就位於 UCS，所有跟環境變數及系統參數有關的解析與列印都集中在 UCS 這層，讓 UCP 這層專注在高階功能（Endpoint 建立與 Wireup 的細節），這樣的分工使程式易於維護，每個 function 只專注於自己該做的事，不會混在一起。

3. Observe when Line 1 and 2 are printed during the call of which UCP API?

   - 從第二題中可以看出，兩行都是在呼叫到 `ucp_worker_print_used_tls()` 這個 UCP API 的時候被印出，而這個 API 又可以回溯到是在創建 endpoint `ucp_ep_create()` 時會被呼

叫到。

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

    - 是有符合的，我之前猜：
        - `UCX_TLS`：我猜應該在 `ucp_config_read` 中被載入。
        - TLS selected by UCX：我猜應該在建立 endpoint 的時候被選擇，因為 endpoint 要傳輸就必須決定底層要使用哪種協議。
    - `UCX_TLS` 應該真的在 `ucp_config_read` 中被載入，所以才能用 `ucp_config_print` 印出來。
    - TLS selected by UCX 確實是在建立 endpoint 的時候被選擇。

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：
      在 UCX 中，`lane` 代表一條與特定傳輸資源（transport resource）相對應的通訊路徑。一個 endpoint 可能包含多個 lanes，以達到平行傳輸的效果。每條 lane 會對應到一個特定的底層傳輸協定（由 UCX 在 wireup 過程中自動選擇）。例如，在程式中可以看到幾種針對不同功能的 lane 定義（如 `am_lane`、`tag_lane`、`cm_lane` 等），每個 lane 具有自己的 `rsc_index`，從而連結到真正的傳輸層資源（如 `tl_name`, `tl_device`）。
    - tl_rsc：
      `tl_rsc` 是 `uct_tl_resource_desc_t` 型別的 instance，用來描述網路資源。它裡頭包含了傳輸層名稱（`tl_name`）、裝置名稱（`dev_name`）、裝置類型（`dev_type`）等。當我們在 UCX 透過 `ucp_context->tl_rscs[...]` 取得這個 `tl_rsc` 時，就能知道底層對應的是哪一個網路介面或硬體裝置。
    - tl_name：
      這是一個字串，儲存了傳輸層的名稱（transport layer name）。例如，可能是 `tcp`、`rc`、`shm` 等，用以表示實際採用的通訊介面類型。
    - tl_device：
      我沒有直接找到這個變數，找到最接近的是 `uct_tl_device_resource_t` 這個 struct，用於表示底層的硬體裝置資訊（例如該裝置的名稱以及它的類型）。像是某張網卡介面、InfiniBand HCA port 或其他可用的傳輸裝置，都會在這裡被描述。
    - bitmap：
      `bitmap` 是一個用來標記（或篩選）哪些 Transport Layer 資源被啟用的資料結構。在 UCX 的程式碼裡，我們會看到程式使用這個 bitmap 來挑選或追蹤哪些網路資源已經初始化，或是在哪些資源上開啟了對應的 `iface`。
    - iface：
      `iface` 是在 UCT 層負責提供通訊操作的介面物件。它包含各種底層操作 function，像是發

送/接收資料、事件通知等。

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
------------------------------------------------------------------
```

1. 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
```

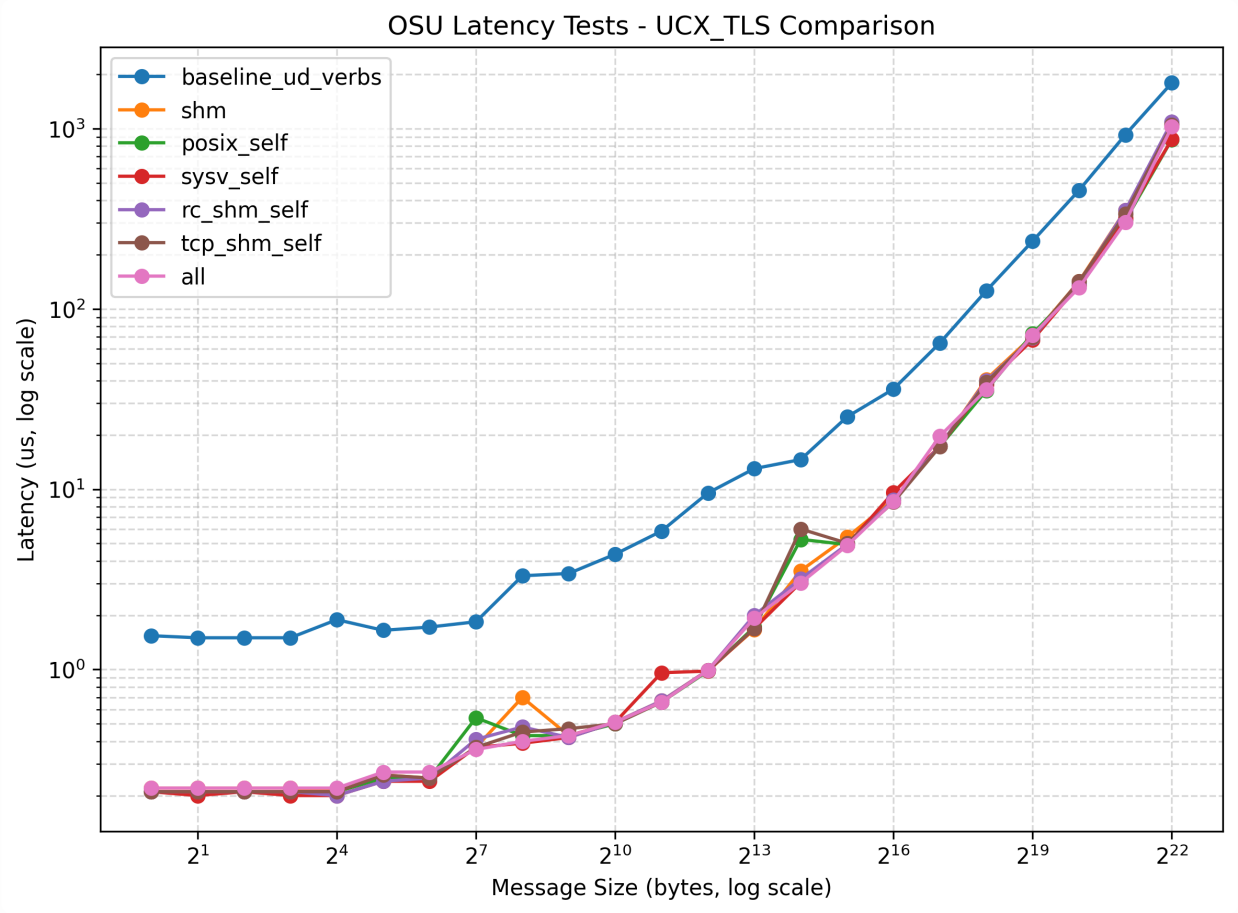
2. Please create a chart to illustrate the impact of different parameter options on various data sizes and the effects of different testsuite.
3. Based on the chart, explain the impact of different TLS implementations and hypothesize the possible reasons (references required).

實驗：測試不同 UCX_TLS 的效果

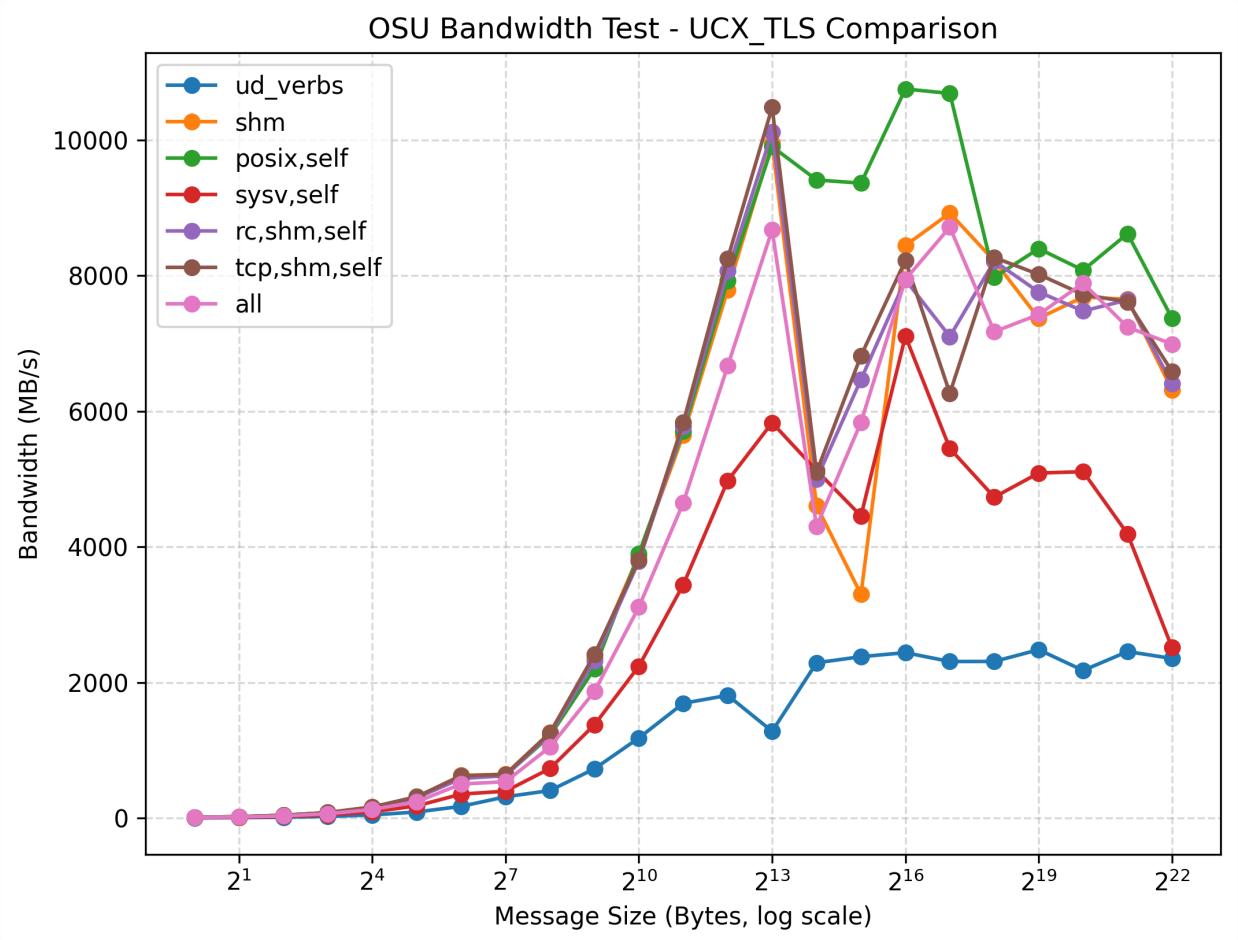
UCX_TLS 設置	Notes
`UCX_TLS=ud_verbs`	預設使用 UD over InfiniBand Verbs。
`UCX_TLS=shm`	強制使用 shared memory (shm)。
`UCX_TLS=posix,self`	單機共享記憶體傳輸：posix + self。
`UCX_TLS=sysv,self`	單機共享記憶體傳輸：SysV SHM + self。
`UCX_TLS=rc,shm,self`	同時啟用 RC (Reliable Connection) verbs、shm、self。
`UCX_TLS=tcp,shm,self`	同時啟用 TCP + shm + self。

`UCX_TLS=all`	讓 UCX 自動偵測並啟用所有可用傳輸層 (verbs, shm, tcp, self...)。

實驗結果



從上圖可以看到，其實除了 `ud_verbs` 之外，其他都有 shared memory 的選項，只是有些會指定特定的 shared memory 機制 (如 `posix`、`sysv`)，而有使用到 shared memory 的都比 `ud_verbs` 快很多，我推測可能是因為程式是在同一台機器上運行，並沒有跨節點，所以使用 shared memory 會快很多。



從上圖中可以看到，同樣可以使用 shared memory 的 bandwidth 都比 `ud_verbs` 大很多，我想也是因為程式在單機上運行，其中又以 `posix, self` 平均的表現最佳，我猜蔡可能是因為 posix 實作 shared memory 的方式在 linux 系統上有較好的優化。

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
```

4. Experience & Conclusion

1. What have you learned from this homework?
   透過這個作業我對 UCX 的運作機制更熟悉了，也練習到了要如何 trace 這樣大型的 repo，不過

這個 repo 真的有點大，我覺得其實還有很多地方沒有弄懂，不過至少之後如果要用到會知道要從哪裡開始深入理解了。

2. How long did you spend on the assignment?
   三個整天。
3. Feedback (optional)