PP HW 4

> 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.

資應所碩二 郭芳妤 111065531

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`

目的：創建 UCP context（ucp_context_h）

細節：

- 在 ucp.h 中定義了此 function（permalink），可以看到他做了以下幾件事：
  1. check API version compatibility
     - 在 ucp_init 這個 routine 其實是調用了 ucp_inti_version 這個 function 來確認上述的 API version compatibility（實作於 core/ucp_context.c 中）
  2. discover available network interface
  3. initialize the network resource（for discovering of the network and memory related devices）
     - 呼叫 ucp_init 需要的參數：
       - `ucp_params`：使用者定義的參數
       - `ucp_context`：欲 setup 的 application context（也是我們最終會設定好的 UCP objects）

  簡言之，他所做的事情就是初始化所有特定 application（MPI, OpenSHMEM） 需要的資訊。

- `ucp_worker_create` 、 `ucp_worker_query`

> 💡 目的：create a worker object、Get attributes specific to a particular worker.
>
> 細節：
>
> - 在 ucp.h 中定義此 function：主要執行了 allocate 和初始化 worker object，實作於 ucp_worker.c
> - 一個 worker 只會對應到唯一 context（但一個 context 可以 create 多個 worker 以實現溝通資源的 concurrent access），因此在初始化 worker 時就要把 context 當作參數傳入
> - 在 worker 創建完成後，可以看到程式 call 了 `ucp_worker_query()` 取得 worker attribute（permalink），其中包括一個重要資訊為 `worker_attr.address` 作為我們的 local address（稍後 endpoint 需要使用到的 address）

- `ucp_ep_create`

💡 目的：create and connect an endpoint

程式細節：

1. 觀察他的傳入參數有 worker 和 ep 的指標，這是代表我們要在指定的 local worker 上創建一個 endpoint，這個 endpoint 要和 remote worker 建立 connection（remote worker 的 address 就是另一個傳入參數：const `ucp_ep_params_t` * `params` ）
2. connection establishment [permalink]：
    a. 先判斷是 client 或是 server（by `client_target_name` ）：如果 client_target_name = Null 為 server，反之為 client
    b. server 傳自己的 local address length 和 Local address
    c. client 先接收到 server 傳過來的 address lenghth 並以此 malloc 出 peer_address，接著再接收 server 傳過來的 address 為 peer_address 做設定
3. 最後，依據 client/server 呼叫在 `run_ucx_client()` 和 `run_ucx_server()` [permalink]
    - 在 run_ucx_* 中就會呼叫 `ucp_ep_create()`，其中傳入的 address 就是在上一步 connection establishment 時設定的 local/peer address
    - ucp_ep_create() 實作於 [permalink]，主要目的是為 worker 建立通信的節點

2. What is the specific significance of the division of UCP Objects in the program? What important information do they carry?
    a. **區分 UCP object 的意義**：之所以需要區分三個層級的 UCP Objects 是希望以更具架構的方式管理一個 application 需要的 communication 資源。可以想像成 context 是一個提供數種資源的通信環境，在這裡可以實現資源的共享（對底下的 workers）以及對外隔離，底下的 worker 負責管理 communication 進度和操作、而其下的 ep 則是 communication 的端點（可以理解為實際通信的 source and target），透過這種架構設計，可以讓資源以及 communication event 的管理更加便利。
    b. **UCP object 攜帶的訊息**：

- `ucp_context` context 代表 application 層級，ucp_context 提供了 isolation mechanism，允許 application 可以獨立管理其 communication resource（共享或隔離），由 ucp_context.h 所定義的 data 可以發現：context 主要攜帶的訊息是「可用資源」及其數量，包括：
  - thread_mode（object 的 thread sharing mode）
  - memory_types
  - Enabled protocols
  - memory domains
  - 可用的 network interface 等等

  這些資訊會影響下層創建 worker create 或者 endpoint create 的方式，在可用資源描述上多用 bitmap 的資料結構來呈現。
- `ucp_worker` worker 是代表實際 communication 的主體，我們可以參考 ucp_worker.h 的這一段來得知 worker 具體攜帶的訊息。可以發現它主要攜帶的訊息包括：
  - 自己的 worker id（uuid）
  - 對上的 ucp_context（back reference）
  - 通信對象的 id
  - 可用資源：依據 context 的設定，有些資源可能會在 create worker 時就決定好（存於 atomic_tls）
  - 對下的 endpoint list（包含自己的 internal ep 與通信的 ep），以及用來管理維護這些 ep 的資訊
- `ucp_ep` ep 為真正通信的端點，可以參考 ucp_ep.h 的這一段 得知 ep 具體攜帶的訊息，包括：
  - 對上的 worker（back reference）
  - Reference counter：用來確認 ep 資源是否需要存在（Couter = 0 時 ep 可被 destroy）
  - Remote connection 的 sequence
  - transport entry：表示此 ep 的 <transport/device> name pairs

3. Based on the description in HW4, where do you think the following information is loaded/created?

- `UCX_TLS` 為系統決定的 transport layers，可以看到如果沒有在一開始指定 context configuration（config = NULL），在 **context initialization** 時會 call `ucs_config_parser_fill_opts()` 來使用預設的 setting → 為驗證這件事，在 context init 完成時加入 `ucs_config_parser_print_env_vars_once(context->config.env_prefix)` 可以發現 UCX 相關的環境變數都已經被設定好，包括：

  ```
  UCX_* env variables:
  UCX_LOG_LEVEL=info
  UCX_NET_DEVICES=ibp3s0:1
  UCX_TLS=ud_verbs
  UCX_WARN_UNUSED_ENV_VARS=n
  ```

- `TLS selected by UCX` **endpoint 被 create 時決定**。因爲真正通信的端點是 end point，所以最兩個 endpoint 要用什麼樣的 protocol 來進行 communication 除了取決於自己有什麼資源、還取決於對方擁有的資源，因此會在 create endpoint 時，還要透過層層 function call 才能確認最終要用何種 network device 和 transport layers 進行通信。

  📎 可以進一步確認 UCX selected TLS 被選擇的路徑（code trace）：

  → `ucp_ep.c` ucp_ep_create()

  → `ucp_ep.c` upc_ep.c ucp_ep_create_api_to_worker_addr()

  → `ucp_ep.c` ucp_ep_create_to_worker_addr()：initialize transport endpoints

  → `wireup.c` ucp_wireup_init_lanes()

  → `select.c` ucp_wireup_select_lanes()

2. Implementation

> Describe how you implemented the two special features of HW4.

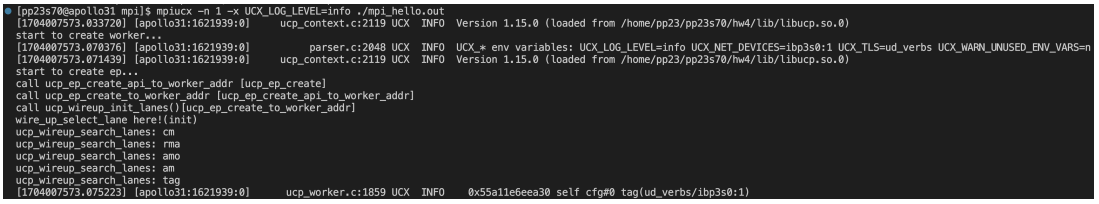
1. Which files did you modify, and where did you choose to print Line 1 and Line 2?

   ▼ **觀察與 call print function 的選擇**

   首先，我們先使用 `UCX_LOG_LEVEL=info` 觀察：可以發現兩個資訊被 ucs_info() 印出相關資訊的時機不同：

- UCX_TLS（specified by system）雖然在 worker create 的階段被 print 出來（printed by `ucs_config_parser_print_env_vars_once(context->config.env_prefix)`），但往上 trace code 可以發現在 ucp_init() 時，就已經透過呼叫ucs_config_parser_fill_opts 來填入所需要的 configuration 了。
- UCX 最終選擇的 TLS transport method 則是在 ep_create() 時才被選定並印出來（printed by `ucp_worker_print_used_tls()`）

因此思考這兩者被設定的時機應該是不同的，接著進一步 trace code 去印證這件事：可以看到 context init 後 system 就選定了 ud_verbs，但 create ep 後還執行了以下 fucntions 才決定最終的 TLS transport method：



因此知道至少要創建 ep 之後才有辦法 Line2 information，為求方便本次實作將 line1 資訊的時間延後到與 line2 一同 print 出，並實作於 `ucp_worker_print_used_tls()`，也就原本的 ucs_info() 印出 final TLS transport method 處。

```
static void ucp_worker_print_used_tls(ucp_worker_h work
    char *TLS_info = NULL;
    ...
    ucs_info("%s", ucs_string_buffer_cstr(&strb));
  /*HW4: print line 1 */
  TLS_info = ucs_config_parser_print_TLS("UCX_TLS", TLS
  ucp_config_print(NULL, stdout, TLS_info, UCS_CONFIG_P
  /*HW4: print line 2 */
  ucp_config_print(NULL, stdout, ucs_string_buffer_cstr
}
```

▼ **實作細節（modified file & modified content）- `path/file` 修改內容**

1. `src/ucs/config/types.h`：加入 UCS_CONFIG_PRINT_TLS = UCS_BIT(5) 設定

```
typedef enum {
        ...
    UCS_CONFIG_PRINT_TLS             = UCS_BIT(5),
} ucs_config_print_flags_t;
```

2. `src/ucs/config/parser.c` ：觀察 TODO: PP-HW4 的提示，在 `ucs_config_parser_print_opt()` 中加入以下 code，表示傳入的 Flag 為 UCS_CONFIG_PRINT_TLS 時，要 print 出 title（char *） 內容

```
// TODO: PP-HW4
if (flags & UCS_CONFIG_PRINT_TLS) {
      fprintf(stream, "%s \n", title);
}
```

3. **實作 feature1 - 印出 UCX_TLS** ：

   a. `src/ucs/config/parser.c` ：觀察 `ucs_config_parser_print_env_vars()` ，發現它可以針對傳入的 prefix 印出有此前綴的訊息，因此我仿照此函數的邏輯寫了一個類似的 function，這個 function 的 input prefix 為 UCX_TLS，使用迴圈找到符合的 env_var 以後，回傳此時的 TLS 設定，存在 TLS_info 這個 char pointer 中：

```
/**
    HW4: print TLS function
*/
char* ucs_config_parser_print_TLS(const char *pre
{
    char **envp, *envstr;
    size_t prefix_len;
    char *var_name;
    char *saveptr;
    char *TLS_info = NULL;
    prefix_len      = strlen(prefix);

    pthread_mutex_lock(&ucs_config_parser_env_var
    for (envp = environ; *envp != NULL; ++envp) {
        envstr = ucs_strdup(*envp, "env_str");
        if (envstr == NULL) {
```

```
            continue;
        }
        var_name = strtok_r(envstr, "=", &saveptr
        if (!var_name || strncmp(var_name, prefix
            ucs_free(envstr);
            continue; /* Not UCX */
        }
        else{
            TLS_info = malloc(strlen(envstr)+strl
            strcpy(TLS_info, envstr);
            strcat(TLS_info, "=");
            strcat(TLS_info, saveptr);
        }
    }

    pthread_mutex_unlock(&ucs_config_parser_env_v
    return TLS_info;
}
```

b. `/src/ucp/core/ucp_worker.c` 修改這個 file 裡 `ucp_worker_print_used_tls()` 的內容：在 return 之前多加上兩行 code 以 print 出我們想要的訊息：

```
/*HW4: print line 1 */
char *TLS_info = NULL;
...
TLS_info = ucs_config_parser_print_TLS("UCX_TLS",
ucp_config_print(NULL, stdout, TLS_info, UCS_CONF
```

在上一段落 (a) 有描述我們如何獲得 UCX_TLS 的訊息，因此在獲得 UCX_TLS 的資訊字串以後，只要 call `ucp_config_print()`，將欲 print 的內容作為 title 傳入即可。

4. **實作 feature2 - 印出 final transport method**：因為觀察到 `ucp_worker_print_used_tls()` 在使用 `UCX_LOG_LEVEL=info` 時印出的資訊就是我們想要的內容，因此直接在此 function 內部 call `ucp_config_print()`，將原本印在 ucs_info 的資訊作為 title 傳入即可。

```
/*HW4: print line 2 */
```

```
ucp_config_print(NULL, stdout, ucs_string_buffer_cst
```

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

   在實作中牽涉到的 functions 如下：

   - File - ucp/core/ucp_worker.c
     - `ucp_worker_print_used_tls`
   - File - ucs/config/parser.c
     - `ucs_config_parser_print_TLS`
     - `ucp_config_print`

   呼叫順序如下：

   create worker → create endpoint → … `ucp_worker_print_used_tls()` 裡面會呼叫以下兩個 function 分別 print 出 line 1 & 2：

   → [Line 1] `ucs_config_parser_print_TLS` → `ucp_config_print`

   - 前者是為了取得後面要放入 `ucp_config_print` 的字串內容（UCX_TLS=ud_verbs）

   → [Line 2] `ucp_config_print`

   - 因為在 `ucp_worker_print_used_tls()` 前面已經處理了需要印出的資訊，所以直接使用其其處理好的字串作為 ucp_config_print 的 input（char *title）即可

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

   在實作時，我是在 `ucp_worker_print_used_tls()` 這個 UCP API 中直接呼叫兩次 `ucp_config_print` 來印出內容，往上 trace code 可以發現 `ucp_worker_print_used_tls()` 這個 function 是在 ep create 之後才被呼叫的，因此總結印出 Line1 & Line2 的時機為 create endpoint 時。

📎 trace code:

`ucp_ep.c` ucp_ep_create()

→ `ucp_ep.c` upc_ep.c ucp_ep_create_api_to_worker_addr()

→ `ucp_ep.c` ucp_ep_create_to_worker_addr()：initialize transport endpoints

→ `wireup.c` ucp_wireup_init_lanes()

→ `ucp_worker.c` ucp_worker_get_ep_config

→ `ucp_worker.c` ucp_worker_print_used_tls()

→ 印出我們想要的資訊（Line1+Line2）

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

   在 1-3 我們已經透過 code trace 驗證 Line1 的資訊得知 UCX_TLS 其實在 Context init 時就已經被決定（並且是可以 print 出來的狀態），但如果在當下就 print 該資訊的話，其順序會是

   ```
   Line1
   Line1
   Line2
   Line2
   Line2
   Line2
   ```

   與我們期待的 Line1 Line2 交錯不同，這是因為如果我們在 create context 時就印出 Line1，那依照 send receive 的架構，只會有兩個 context，Line1 的資訊就只會有兩個。為符合作業要求的格式，我們在實作時將 UCX_TLS 的資訊 delayed 到 create ep 時才印出，這是實作上和資訊被 load 入時的差異。

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`：指可用的傳輸通道
   - `tl_rsc`：為 UCT resource descriptor（傳輸層包含的資源描述）
     - 結構定義於：ucp_context.h → 被包在 ucp_tl_resource_desc_t 成為一個抽象化後的結構

    - 在 ucp_context 中的其中一個變數 `*tl_rscs` 就是以此結構構成一個 array，儲存 context 的 communication resources。
- `tl_name`：為傳輸層名稱。
    - 整個結構定義於：uct.h
    - 上方描述的 `tl_rsc` 就是由此結構定義而來，tl_name 則是這此結構中以 char array 的形式呈現 transport name，因此如果想要知道一個 context 底下使用的 transport 是哪一種，讀取的方法就是 tl_rsc.tl_name。
- `tl_device`： transport device（傳輸層裝置）
    - 結構定義於：uct_iface.h
    - 主要攜帶的資訊為硬體 device name、UCX device type、system device (將 device bus id 轉為 integer)

    ▼ device type 包含以下幾種：

```
typedef enum {
    UCT_DEVICE_TYPE_NET,     /**< Network devices */
    UCT_DEVICE_TYPE_SHM,     /**< Shared memory devi
    UCT_DEVICE_TYPE_ACC,     /**< Acceleration devic
    UCT_DEVICE_TYPE_SELF,    /**< Loop-back device *
    UCT_DEVICE_TYPE_LAST
} uct_device_type_t;
```

- `各種 bitmap` ：以 bit 方式表達此資源是否可以使用。這邊以 tl_bitmap 舉例：tl_bitmap 為 transport layer 資源的 cached map，提供 worker 所有可用的 transport layer 資訊。
- `iface`
    - 結構定義於：ucp_worker.h
    - worker interface 的資訊：worker 的變數 **ifaces 表示指向這些 interface 的 array，array 中每一個 element 都是由 ucp_worker_iface_t 結構組成，即把 uct 層的 interface 資訊包裝起來提供給 ucp_worker 使用

3. Optimize System

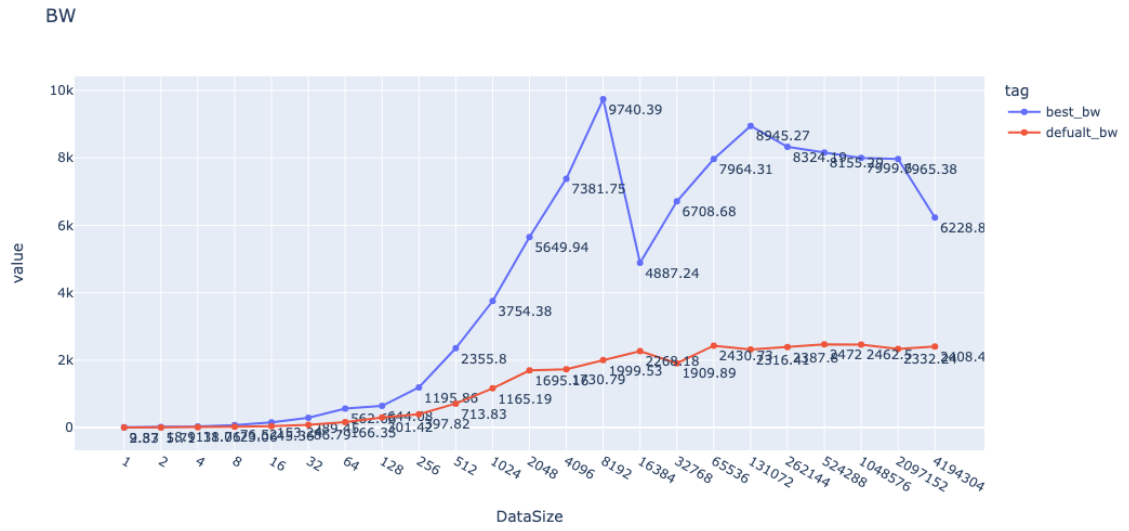
1. Below are the current configurations for OpenMPI and UCX in the system. Based on your learning, what methods can you use to optimize single-node performance by setting UCX environment variables?

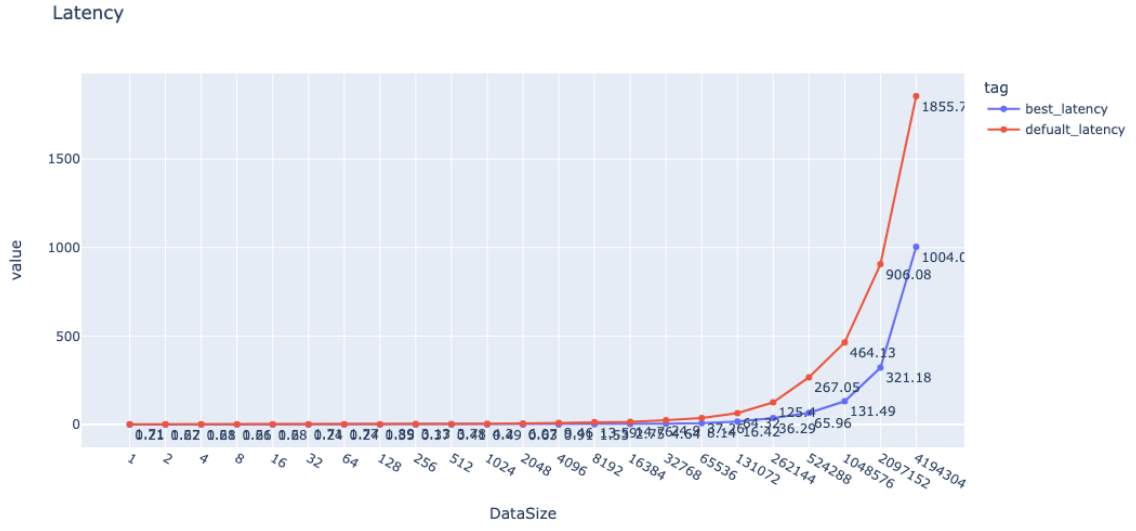
```
-------------------------------------------------------------
/opt/modulefiles/openmpi/4.1.5:

module-whatis   {Sets up environment for OpenMPI located i
conflict        mpi
module          load ucx
setenv          OPENMPI_HOME /opt/openmpi
prepend-path    PATH /opt/openmpi/bin
prepend-path    LD_LIBRARY_PATH /opt/openmpi/lib
prepend-path    CPATH /opt/openmpi/include
setenv          UCX_TLS ud_verbs
setenv          UCX_NET_DEVICES ibp3s0:1
-------------------------------------------------------------
```

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

```
module load openmpi/4.1.5
mpiucx -n 2 $HOME/UCX-lsalab/test/mpi/osu/pt2pt/standard/o
mpiucx -n 2 $HOME/UCX-lsalab/test/mpi/osu/pt2pt/standard/o
```

- 由上述圖表可以看到，在指定 `UCX_TLS=sm` 執行兩個測試檔案時，可以有比 default 更好的 bandwidth 以及 latency 表現（latency 下降 and bandwidth 提升）。當我們選用 shared memory 以後可以發現 UCX select 的最終 transport 為 `UCX_TLS=sm`，同時找到可用的 <transports/devices> pair 為 `sysv/memory` `cma` `/memory`，這些都是 UCX 提供的 optimized shared memory 機制。這是由於我們今天是在同一個 node 上面進行 communication，又 shared memory 有 low latency 和 high bandwidth 的特性，因此在這個 case 底下使用 shared memory 會有相較 default 的 ud_verbs 更好的效果。





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?

   這是第一次需要這麼仔細的 trace 一個大型的 project，第一個困難是因為我對通信架構沒有很熟悉，所以裡面的很多專有名詞（包括概念和可以使用的資源種類）都需要查很久（有些甚至很難查到XD），再來是因為 UCX 是大型的 project，所有的 code file 基本上都是千行起跳，在呼叫各類資源或 UCP API 的時候很常因為名稱很像而弄混，但在實作兩個 feature 的時候會因為需要 follow project 的架構而體認到這種大型專案不能任意妄為的寫 code，許多實作的 function 在命名、call API 的方法都要保持一致性，而且這個專案提供了很完整的 debug 訊息（一開始對實作毫無頭緒的時候就是依賴 UCX_LOG_LEVEL 拯救）。

2. Feedback (optional)

   很感謝助教用心出的這份作業，真的在從實作到 trace code 都可以感受到他們之間之環環相扣，是必須了解 UCX 的架構後才能寫得出來的一份作業，但這份作業其實比預想的要花時間，如果和 Final Project 放在相同的三週做可能有點勉強 >< 如果可以會希望有更多的時間可以完成這份作業

參考資料

https://cloud.tencent.com/developer/article/2353126