PP HW 5 UCX

ID: 112062520 / Name: 戴維恩

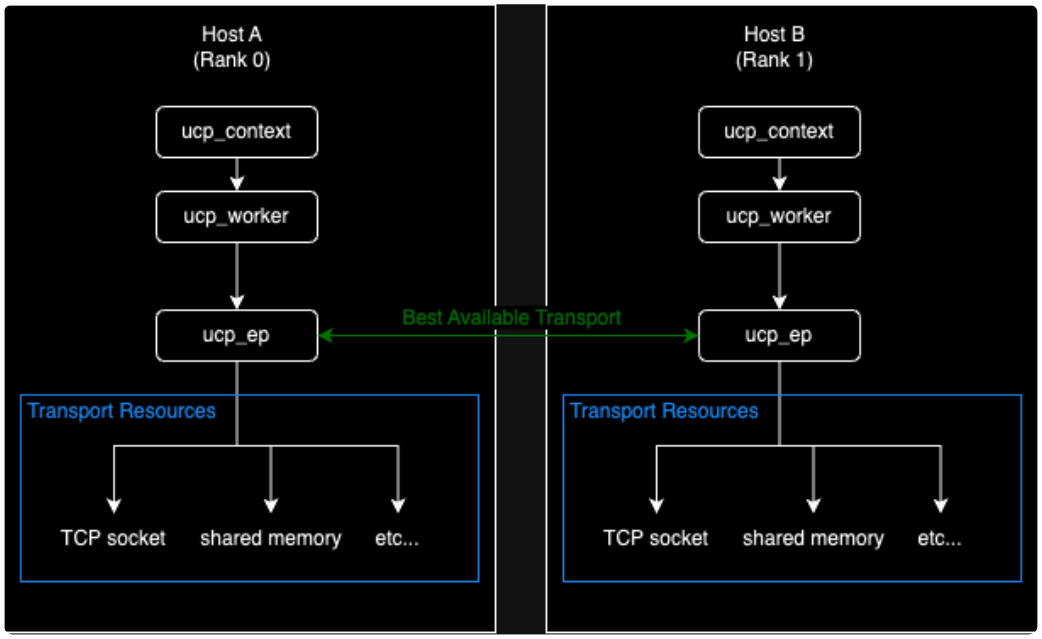
1. Overview

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`

> ANS:
> 首先每個 process 會需要建立自己的 `ucp_context` ，在程式的一開始使用 `ucp_config_read` ，並根據讀進來的 config 以及自定義的參數呼叫 `ucp_init` 以初始化 `ucp_context` ，接著 context 需要透過 `ucp_worker_create` 函數建立 `ucp_worker` 用以後續與其他人的資料傳輸。
> 在 `ucp_hello_world.c` 中可以看到，當 `server` 以及 `client` 各自的 worker 建好之後， `server` 會先用一些比較簡單的通訊方式把自己的 worker network address 送給 `client` ， `client` 就可以用收到的 `peer address` 建立 `ucp_ep` （呼叫 `ucp_ep_create` ），建好 ep 後也可以將自己的 network address 傳給 server，讓 server 也去建 ep，當兩邊的 ep 建好就可以使用 `UCX` 挑選的最佳溝通方式傳遞資訊。

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`

> ANS:
> 如果依照指令 `srun -N 2 ./send_recv.out` ，那麼架構圖如下。兩個 Node 各自有一個 process，每個 process 會建立一個 context，context 會去建立 worker，交換 network address 後建立 endpoint；硬體可能會有許多可以進行通訊的資源，例如 `TCP socket` /

> `shared memory` 等等，而 `UCX` 會挑選最佳溝通的方式透過 `ucp_ep` 進行資料傳輸。
>
> 
>

3. Based on the description in HW5, where do you think the following information is loaded/created?
    - `UCX_TLS`
    - TLS selected by UCX

> ANS:
> `UCX_TLS` 是 user 可以在執行程式的時候自行定義的 config 參數，因此在 `ucp_config_read` 時期就會被讀取到，並拿來初始化 `ucp_context`；而真正被 `UCX` 選到的 `TLS`，我認為是在建立 endpoint 的時候才會確定，因為在這時候才會拿到要溝通對象的 network address 等等，可能會需要根據溝通對象的資訊才能選擇最佳傳輸方式。

2. Implementation

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

> ANS:
> 我總共改了 `parser.c` / `ucp_worker.c` / `type.h` 這三個檔案，Line 1 是要印出在 config 裡面指定的環境變數 `UCX_TLS`；我將 API 函數 `ucp_config_print` 的功能擴充，讓他可以根據我自己定義的 flag 列印出 `UCX_TLS` 相關資訊。
> `ucp_config_print` 會去呼叫在 `parser.c` 中的 `ucs_config_parser_print_opts` 函數，用來根據指定 flag 印出對應 config 資訊的函數，我先在 `type.h` 裡面新增一個 flag `UCS_CONFIG_PRINT_TLS` 如下：
>
> ```
> // in type.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),
    // new flag here
    UCS_CONFIG_PRINT_TLS              = UCS_BIT(5)
} ucs_config_print_flags_t;
```

接著，在 `ucs_config_parser_print_opts` 裡面新增條件式，如果遇到這個 flag 就印出 config 中跟 UCX_TLS 有關的資訊：

```
// in parser.c
// in function 'ucs_config_parser_print_opts'
if (flags & UCS_CONFIG_PRINT_TLS) {
    printf("UCX_TLS=");
    // search for TLS in config fields
    for (; !ucs_config_field_is_last(fields); ++fields) {
        if (strcmp(fields->name, "TLS") == 0) {
            ucs_config_allow_list_t *tls = (ucs_config_allow_list_t*)((char*)
            // if UCX_TLS=all, print "all"
            if (tls->mode == UCS_CONFIG_ALLOW_LIST_ALLOW_ALL) {
                printf("all\n");
            } else {
                // if UCX_TLS!=all, print every specified resources
                ucs_config_names_array_t *array = &tls->array;
                for (unsigned i = 0; i < array->count; i++) {
                    if (i > 0) printf(",");
                    printf("%s", array->names[i]);
                }
                printf("\n");
            }
            break;
        }
    }
}
```

如此一來，我就可以用使用 API function `ucp_config_print` 去印出 `UCX_TLS` 的資訊。

至於 Line 2 的實作，是要印出最後被 UCX 選中的傳輸途徑，我在 `ucp_worker.c` 中發現一個 `ucp_worker_print_used_tls` 函數，他會在 API function `ucp_ep_create` 的途中被呼叫到，裡面已經整理了這個 endpoint 傳輸資料會用到的 resouce，因此我在這邊印出 Line 2 內容，順便呼叫印出 Line 1 所需要的 API call：

```
// in ucp_worker.c
// in function 'ucp_worker_print_used_tls'

ucp_config_t *cfg;
ucs_status_t status;
status = ucp_config_read(NULL, NULL, &cfg);
// print Line 1
ucp_config_print(cfg, stdout, NULL, UCS_CONFIG_PRINT_TLS);
ucp_config_release(cfg);

// print Line 2
```

```
// 'strb' has all the UCX selected tls information
printf("%s\n", ucs_string_buffer_cstr(&strb));
```

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

> ANS:
> 由於 Line 1 以及 Line 2 皆是在 `ucp_worker_print_used_tls` 這個函數中被印出的，因此下面列出從 API call 到這個函數的途徑：
> `ucp_ep_create` -> `ucp_ep_create_api_to_worker_addr`
> -> `ucp_ep_create_to_worker_addr` -> `ucp_wireup_init_lanes`
> -> `ucp_wireup_select_lanes` -> `ucp_worker_get_ep_config`
> -> `ucp_worker_print_used_tls`
>
> - 列印 Line 1 是從 `ucp_worker_print_used_tls` 呼叫 `ucp_config_print` ，接著到 `ucs_config_parser_print_opts` -> 取得 config `UCX_TLS` 並印出。
> - 列印 Line 2 是直接在 `ucp_worker_print_used_tls` 中以 `printf` 印出。

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

> ANS:
> 如上題的路徑所示，Line 1 以及 Line 2 是在 API call `ucp_ep_create` 中被印出。

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

> ANS:
> 我最後的實作方式跟原本預期的一樣，UCX 確實是在 endpoint 階段選擇最終使用的通訊方式。這次 trace code 主要看了 `ucp_context.c` / `ucp_worker.c` / `ucp_ep.c` 這三個檔案。
>
> - 在 context 階段，UCX 會去讀取 config 裡面設定的 resource，並在 `ucp_add_tl_resource_if_enabled` 函數中確認 resource 是否能使用，若是則加入 `context->tl_rscs` 中，且會有一個變數 `context->num_tls` 去紀錄總共能用的 TLS 數量，因此在這個階段還沒有真正的選擇最後使用的 resource。
> - 在 `ucp_worker_create` 階段，worker 會根據上一階段 context 列出來的那些 resource，去打開這些 resource 的 interface (`ucp_worker_add_resource_ifaces`)，這個階段也沒有真正的選擇過程。
> - 最後在 `ucp_ep_create` 階段，這裡的每一條 "lane" 都是用某一種 resource 去建立的通訊模式，而不同應用要用哪條 "lane" 去傳遞訊息，則是在 `ucp_wireup_select_lanes` 裡面被決定，因此最終的選擇過程在這個階段中被執行。

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.

ANS:

- **lanes**
  每個 endpoint 有可能會根據應用需要不一樣的 protocol，例如：`RMA` (remote memory access) / `AM` (active message) / `TAG` 等等，而為每個需要的 protocol 建立一個 lane (傳輸路徑)，用這種 protocol 進行傳輸；每條 lane 再根據 protocol 的特性，去選擇要用哪些底層 resource (`tl_rsc`)。例如，在幫 `lane_RMA` 選擇 resource 時，要選擇能支援 `PUT` / `GET` 這兩種 `RMA` 特有的 operation 的底層資源。
- **tl_rsc**
  `tl_rsc` 就是跟這個 resource 有關的所有資訊 (`uct_tl_resource_desc_t`)，裡面有：`tl_name` (這個資源的名字，例如：`ud_verbs`) / `dev_name` (使用的硬體名稱，例如：`ibp3s0:1`) 等等 `UCT` 底層跟這個資源有關係的資訊。
  - **tl_name**
    是 `tl_rsc` 的 member，代表資源的名稱，例如：`self` / `sysv` / `posix` / `ud_verbs` 等等。
  - **tl_device (dev_name)**
    也是 `tl_rsc` 的 member，代表資源使用的硬體，例如：`memory` / `ibp3s0:1`。
- **bitmap**
  是一個儲存 index & resource 對應關係的 map，裡面有許多 `tl_id` ，每個 `tl_id` 都會對應到一個 `tl_rsc`。
- **iface**
  就是 worker 裡面，每個 resource 的 interface。在 `ucp_worker_create` 階段，會把 context 階段列出來的每個 resource `context->tl_rscs` 的 interface 都打開 (`ucp_worker_iface_open`) ，並儲存在 `worker->ifaces` 裡面。

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

> ANS:
> 目前 config 預設的 `UCX_TLS` 變數是 `ud_verbs`，不過這種 resource 在對於 single-node 的溝通並沒有特別的優勢；如果是 `single-node` 的溝通應該可以選用 `sysv` 這種透過 shared memory 傳輸資料的優化資源；對於同一個 process 之間的溝通也可以用 `self` 。

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
```

> ANS:
> `UCX_TLS=ud_verbs`
>
> | Size | Latency (μs) | Bandwidth (MB/s) |
> |---|---|---|
> | 0 | 1.50 | - |
> | 1 | 1.75 | 2.90 |
> | 2 | 1.52 | 5.79 |
> | 4 | 1.50 | 11.78 |
> | 8 | 1.76 | 23.67 |
> | 16 | 1.65 | 45.42 |
> | 32 | 1.66 | 90.31 |
> | 64 | 1.98 | 171.65 |
> | 128 | 1.85 | 313.87 |
> | 256 | 3.07 | 405.30 |
> | 512 | 3.39 | 728.33 |
> | 1024 | 4.34 | 1184.78 |
> | 2048 | 5.56 | 1698.39 |
> | 4096 | 8.75 | 1822.29 |
> | 8192 | 13.03 | 2020.73 |
> | 16384 | 14.40 | 2253.07 |

32768	22.27	2367.32
65536	38.88	2183.16
131072	67.69	2439.95
262144	124.80	2384.54
524288	241.92	2478.80
1048576	462.30	2265.86
2097152	954.17	2461.26
4194304	1789.45	2363.62

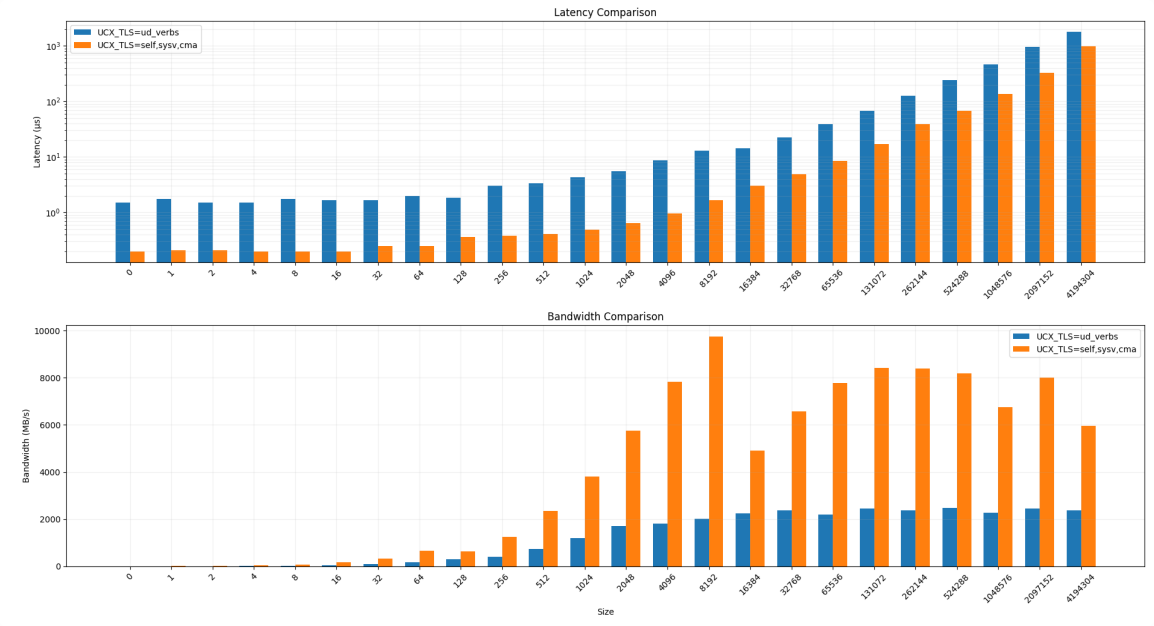
`UCX_TLS=self,sysv,cma`

Size	Latency (µs)	Bandwidth (MB/s)
0	0.20	-
1	0.21	9.70
2	0.21	20.25
4	0.20	40.58
8	0.20	82.44
16	0.20	165.95
32	0.25	327.50
64	0.25	650.66
128	0.36	630.60
256	0.38	1237.35
512	0.41	2353.26
1024	0.49	3813.47
2048	0.65	5761.74
4096	0.97	7827.96
8192	1.67	9748.68
16384	3.02	4902.73

32768	4.89	6560.68
65536	8.53	7783.76
131072	17.24	8424.85
262144	38.66	8395.70
524288	67.17	8188.99
1048576	136.15	6749.08
2097152	328.86	8004.64
4194304	972.70	5957.61

3. Please create a chart to illustrate the impact of different parameter options on various data sizes and the effects of different testsuite.

ANS:



4. Based on the chart, explain the impact of different TLS implementations and hypothesize the possible reasons (references required).

ANS:

- Latency

  `self` / `sysv` / `cma` 在 data size 較小的情況下，傳輸速度約快 7.5 倍（0.20-0.41 µs 對比 1.50-3.39 µs）；隨著 data size 增加，差距持續擴大（size = 4194304 時，972.70 µs 對比 1789.45 µs）。

- Bandwidth

  `self` / `sysv` / `cma` 在小 or 中型 data size 的情況下達到 3-4 倍頻寬；且 `self` /

> `sysv` / `cma` 頂峰值約 9700 MB/s，而 `ud_verbs` 僅能達到約 2400 MB/s。
>
> - Possible Reasons
>   - `self` / `sysv` / `cma` 使用直接記憶體傳遞資料（shared memory），完全繞過 network protocol。
>   - `ud_verbs` 需要完整的 network protocol 處理流程。
>   - reference: UCX document

4. Experience & Conclusion

1. What have you learned from this homework?

> 這次作業很深入的去了解 UCX 每個 API call 運作的方式，一般來說在實作一些 communication 應用的時候並不會接觸到那麼底層的東西，所以透過這次機會更深入的了解 process 之間的溝通管道，以及各種不同的 configuration，我覺得是蠻難得的機會。
>
> 底層通訊機制的方面，透過詳細理解每個階段的 TLS 處理，讓我更理解 process 間通訊的細節；從數據上來看，可以觀察到不同 transport methods 的優劣勢，也讓我深刻體會到系統底層設計對應用程式效能的重要影響。
>
> 整體而言，這次的作業讓我對系統程式有更深的認識，也提供了難得的機會去探索和理解常用通訊框架的內部運作機制。相信這些知識對未來在開發各種通訊應用時會有很大的幫助。

2. How long did you spend on the assignment?

> 這次要 trace 的 codebase 規模蠻大的，每個檔案都有幾千行程式碼，又橫跨多個不同的檔案和資料夾，因此花了不少時間在慢慢看裡面的每個 function 跟型別。trace code / 程式實作 / report 撰寫總共花了大概一週的時間。