PP HW 4 Report

112062673吳文雯

link to hackmd: https://hackmd.io/@Vickiewu/HkC_PGuLp

> - 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`

    簡要說明: context是關於一個application的資訊，通常會對應到一個process，而一個context之下可以有一個或多個worker(對應到thread)，用來管理傳輸相關的資源及程序，一個worker之下可以有多個end point，ep是實際上負責處理send任務的角色。因此，要先有context才能建立worker，有了worker才能夠創造ep，底下會以client的角度來進一步說明:

    透過 `ucp_init` 會創造並初始化一個針對這個application的 ucp context，且會透過這個函式取得指向context的指標(https://github.com/NTHU-LSALAB/UCX-lsalab/blob/ce5c5ee4b70a88ce7c15d2fe8acff2131a44aa4a/examples/ucp_hello_world.c#L575C14-L575C14)。接著會用這個剛取得的context創造worker(https://github.com/NTHU-LSALAB/UCX-lsalab/blob/ce5c5ee4b70a88ce7c15d2fe8acff2131a44aa4a/examples/ucp_hello_world.c#L587C5-L587C5)，在 `ucp_hello_world.c` 中是使用single thread來對應一個worker，而透過 `ucp_worker_create` 這個函式，就能夠取得指向這個worker的指標，也就是這個worker的handler。接著worker藉由 `ucp_worker_query` 取得worker address，這個特殊的address structure能夠在不同的transportation layer傳輸，目的是讓server知道這個worker的位置(https://github.com/NTHU-LSALAB/UCX-lsalab/blob/ce5c5ee4b70a88ce7c15d2fe8acff2131a44aa4a/examples/ucp_hello_world.c#L592C14-L592C30)。接著worker會透過out-of-bound的方式(例如socket)取得server的address(https://github.com/NTHU-LSALAB/UCX-lsalab/blob/ce5c5ee4b70a88ce7c15d2fe8acff2131a44aa4a/examples/ucp_hello_world.c#L612)。

    再來進入 `run_ucx_client` 函式，在這裡會透過 `ucp_ep_create` 來創造並初始化ep，這裡除了需要傳入他所屬的worker object，在參數的部分還需要給定剛才透過out-of-bound取得的server

address，這是因為ep是負責跟server互動的管道，且他負責進行send的任務，因此必須要知道傳遞的目的位置(https://github.com/NTHU-LSALAB/UCX-lsalab/blob/ce5c5ee4b70a88ce7c15d2fe8acff2131a44aa4a/examples/ucp_hello_world.c#L251)。當ep建立完成後，就需要透過ep將他所屬的worker的位置(也就是先前透過`ucp_worker_query`取得的worker address)透過`ucp_tag_send_nbx`傳送給server (https://github.com/NTHU-LSALAB/UCX-lsalab/blob/ce5c5ee4b70a88ce7c15d2fe8acff2131a44aa4a/examples/ucp_hello_world.c#L266C31-L266C47) 接著才能夠傳輸訊息。

2. What is the specific significance of the division of UCP Objects in the program? What important information do they carry?
    - `ucp_context`
    - `ucp_worker`
    - `ucp_ep`

- `ucp_context`帶有關於整個application溝通所需的資源及資訊，包含conifguration、記憶體管理、可用的network interface、transport protocols等等(https://github.com/NTHU-LSALAB/UCX-lsalab/blob/ce5c5ee4b70a88ce7c15d2fe8acff2131a44aa4a/src/ucp/core/ucp_context.h#L270)。在`ucp_init`的過程中，除了會建立context，也會將所需資源存在context之中(https://github.com/NTHU-LSALAB/UCX-lsalab/blob/ce5c5ee4b70a88ce7c15d2fe8acff2131a44aa4a/src/ucp/core/ucp_context.c#L1614)。
- `ucp_worker`負責管理傳輸過程相關資源與程序，因此他需要有很多資源與機制來配合，例如request memory pool、address_name、以及他所擁有的end points的相關資訊等。(https://github.com/NTHU-LSALAB/UCX-lsalab/blob/ce5c5ee4b70a88ce7c15d2fe8acff2131a44aa4a/src/ucp/core/ucp_worker.h#L269)
- `ucp_ep`代表一個connection，帶有要傳輸的對象的ep id、記憶體位址等傳輸所需資訊，負責進行send任務(https://github.com/NTHU-LSALAB/UCX-lsalab/blob/ce5c5ee4b70a88ce7c15d2fe8acff2131a44aa4a/src/ucp/core/ucp_ep.h#L539)。
- 讓context-worker-end point這樣的階層式設計有以下好處: 1.資源管理: 整個application都相同的資訊可以放在context，讓底下的worker及end point共享，不必再重複儲存或配置。而屬於同一個worker的end point也可以共享相同的worker相關資訊，只有在每個connection各自不同的資訊才需要存放在end point。 2.scalability & isolation: 不同的worker可以同時運用自己擁有的end points獨立進行傳輸任務，因此可以同時利用多個processor的資源，且worker間也不會互相影響。

整體而言，這樣的架構可以讓開發者更容易根據要處理的資源或任務是屬於哪個層級(整個application，一個worker (thread)，或是單一connection)來進行管理。

3. Based on the description in HW4, where do you think the following information is loaded/created?
    - `UCX_TLS`
    - TLS selected by UCX

- UCX_TLS 包含了所有這個程式可用的Tranport相關資訊，這個資訊對於整個process而言都是相同的，因此應該存放於context的部分。在這段程式碼中可以看到，所有可用的tls資訊都被儲存在

context中(https://github.com/NTHU-LSALAB/UCX-lsalab/blob/ce5c5ee4b70a88ce7c15d2fe8acff2131a44aa4a/src/ucp/core/ucp_context.c#L1614)。

- 由於worker負責管理傳輸相關的資源與流程，因此應該由worker在run time來選擇最適合的TLS，TLS selected by UCX應該由worker來create。這段程式碼就是worker挑選最佳的tls的過程，而被挑選到的tls會被存在worker->ifaces的陣列中(https://github.com/NTHU-LSALAB/UCX-lsalab/blob/ce5c5ee4b70a88ce7c15d2fe8acff2131a44aa4a/src/ucp/core/ucp_worker.c#L991)。

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?
    - 我修改了 `ucs/config/type.h` ，在 `ucs_config_print_flags_t` 中新增一個flag: `UCS_CONFIG_PRINT_TLS` 。
    - 在 `ucs/config/parser.c` 的 `ucs_config_parser_print_opts` 中實作 `if (flags & UCS_CONFIG_PRINT_TLS){}` 的段落，Line 1 及 Line 2 都是透過這裡印出的。
        - 首先透過 `ucs_config_parser_print_field` 把config中關於TLS的資訊印出(Line 1)。
        - 接著使用fprintf印出傳入函式的title字串，也就是Line 2。
    - 最後修改了 `ucp/core/_ucp_worker.c` 中的 `ucp_worker_print_used_tls` 函式。
2. How do the functions in these files call each other? Why is it designed this way?
    - 在ucp_worker_print_used_tls函式裡，原本會把所使用的tls相關資訊log下來，因此我就在這個函式的最後呼叫 `ucp_config_print` ，接著根據它所需的參數格式放置對應參數。首先透過 `ucp_config_read` 取得config，stream的部分輸入stdout以顯示在螢幕上，要印出的字串就是前面已經組合好的 `ucs_string_buffer_cstr(&strb)` (也就是 Line 2)，最後flag部分就使用新增的 `UCS_CONFIG_PRINT_TLS` 。
    - 在 `ucp_config_print` 中，會再呼叫 `ucs_config_parser_print_opts` 並傳入它所需參數，而由於我們傳入的flag是 `UCS_CONFIG_PRINT_TLS` ，因此會透過前面實作的 `if (flags & UCS_CONFIG_PRINT_TLS){}` 段落印出相關資訊。
    - 之所以會這樣設計是因為(根據slide p.37)ucs是負責提供data structure support或system utilities等服務的架構，因此像是印出訊息這樣的任務，統一交由ucs相關的函式來處理，會更為統一與簡便。而前面所新增的flag也是定義在ucs之下，也是因為相同的原因。
3. Observe when Line 1 and 2 are printed during the call of which UCP API? Line 1 及 Line 2 是透過 `ucp_config_print` ，其中又呼叫了 `ucs_config_parser_print_opts` ，這是最終印出訊息的函式。
4. Does it match your expectations for questions **1-3**? Why? 第1~3題跟我的預期大致相同，由於ucs是負責提供服務的架構，因此由他底下的函式來印出訊息是很合理的設計。而由於worker的主要任務就是負責管理傳輸相關的過程，因此有預期會由他選擇適當的傳輸方式。不過關於Line 1，原本想過因為是整個application都相同的資訊，可能會放在context，並從context的部分呼叫print，但後來發現worker在呼叫 `ucp_config_print` 時傳入的config就能夠取得這個資訊，因此在todo的地方加上 `ucs_config_parser_print_field` 就能夠把Line 1一併印出來。

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.

    - lane: 代表一個communication path。
    - tl_src: 包含check sum of tl_name,memory domain index, device index等溝通資源。
    - tl_name: 紀錄transport protocol 的名稱。
    - tl_device: transport layer的network devices，包含hardware name, network interface及bus id。
    - bitmap: 是一個由bits組成的data structure，以tl_bitmap來說，長度是64bits，用來記錄使用了哪些resources。
    - iface: 包含所有tranport interface operations(put,get,active message,etc.)

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 in /opt/openmpi}
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 這裡指定只能用ud_verbs
setenv          UCX_NET_DEVICES ibp3s0:1 設定只能用這個ifiniband網卡傳輸    觀察這兩點看如何更改
-------------------------------------------------------------------
```

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

```
module load openmpi/4.1.5
會告訴你系統上傳輸1byte 資料latency多久
mpiucx -n 2 $HOME/UCX-lsalab/test/mpi/osu/pt2pt/standard/osu_latency
在系統上不同大小資料頻寬多大
mpiucx -n 2 $HOME/UCX-lsalab/test/mpi/osu/pt2pt/standard/osu_bw
可以做成表格證明你修改後獲得多少性能提升
```

我使用 `export setenv UCX_TLS all` 讓傳輸協定的選擇不只侷限於ud_verbs，而是可以在所有的選擇中，選出ucx評估最佳的tls來使用。而在UCX_NET_DEVICES的部分也利用 `export setenv UCX_NET_DEVICES all` 來讓ucx可選擇較佳的網路來傳輸，修改前後的比較如下:

ps. 1.這裡的TLS=all指的是修改後的UCX_TLS加上原本的UCX_NET_DEVICES設定 2.這裡的NET_DEVICES=all指的是UCX_TLS=all加上NET_DEVICES=all的設定 **Latency:**

Datatype	original	TLS=all	NET_DEVICES=all
1	1.55	0.24	0.22
4	1.76	0.21	0.21
16	1.67	0.22	0.22
64	1.76	0.26	0.26
256	3.11	0.39	0.47
1024	4.18	0.54	0.51
4096	8.78	1.13	1.03
16384	14.62	3.00	3.47
65536	36.52	10.78	8.84
262144	123.79	38.91	40.12
1048576	449.72	137.90	139.35
4194304	1779.18	1009.69	1035.82

Bandwidth:

Datatype	original	TLS=all	NET_DEVICES=all
1	2.52	9.84	9.98
4	10.89	39.61	40.05
16	42.13	158.53	160.51
64	159.67	568.87	630.48
256	388.03	1194.03	1151.49
1024	1201.72	3733.45	3842.62
4096	1763.35	7787.77	7476.87
16384	2271.68	5012.37	5137.69

Datatype	original	TLS=all	NET_DEVICES=all
65536	2430.85	8164.07	5232.64
262144	2466.02	8395.39	8633.10
1048576	2488.75	7880.52	8008.98
4194304	2428.82	7078.08	6914.91

從表格可以看出，當UCX_TLS修改為all之後，latency大幅減少，bandwidth也大幅增加，顯示這個修改有助於提升系統效能，能夠在runtime根據現況選擇最佳的protocol來傳輸。另外，當修改UCX_NET_DEVICES=all之後，latency跟bandwidth都只再獲得微幅提升，這可能是因為原本設定的inifiniband已經夠好了，才看不出顯著的效果。

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? 透過這次的作業了解到課堂上所教授關於ucp、uct及ucs的架構是如何透過程式碼實現，在trace code的過程中也更加具體地理解context、worker及end point是如何一層一層地被創造與管理，以及他們如何完成各自的任務。另外，透過optimization的實作也學習到如何調整ucx的設定以取得更好的效能，整體而言藉由這次的作業對於ucx確實有了更加深入與具體的認識與理解。
2. Feedback (optional)