PP HW4 Report

112164513 陳彥凱

> - 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`
     - 建立與初始化：首先透過 ucp_init(&ucp_params, config, &ucp_context) 初始化 `UCP context` 。 這個函數配置 UCX 環境，設定如記憶體、網絡介面和傳輸協議等資源。
     - 角色： `UCP context` 代表整個 UCX 通訊環境，是通訊操作前必須建立的全域物件，它也負責檢查 API 版本兼容性。
   - `ucp_worker_create`
     - 創建 `worker` ：在有了 `UCP context` 之後，程式透過 ucp_worker_create(ucp_context, &worker_params, &ucp_worker) 創建 `UCP worker` 。 這些 `worker` 與特定的 thread 或 application context 相關聯。
     - 角色： `UCP worker` 處理通訊和計算操作。 它是在 UCX 應用程式中進行通訊操作的執行單位。每個 `worker` 和唯一一個 appication context 關聯，但同時一個 application context 可以創建多個 `worker` 以實現對通信資源的並行訪問。
   - `ucp_ep_create`
     - 創建 `endpoint` ：ucp_ep_create(ucp_worker, &ep_params, &server_ep) 用於在特定的 `worker` 上創建 `UCP endpoint` 。 這需要遠端 `worker` 的地址來建立連接。在 ep_params 中就帶有所有有關 client 的必要資訊。
     - 角色： `UCP endpoint` 建立了兩個 `UCP workers` 之間的連接，允許資料通訊。

   總結來說，在程式中這些物件的互動大致如下：

   - 程式首先初始化 `UCP context` ，這是和 UCX 進行任何互動的起點。
   - 接著，創建一個或多個 `UCP worker` 。 每個 `worker` 代表一個通訊執行單位，可以獨立進行資料的接收和發送操作。
   - 然後，為了與遠端進行通訊，程式在本地 `worker` 上創建 `UCP endpoint` 。 這些 `endpoint` 直接與遠端 `worker` 的 `endpoint` 進行通訊。

   在這個過程中， `UCP context` 管理全域的配置和資源， `UCP worker` 處理實際的通訊操作，而 `UCP endpoint` 則是通訊的具體通道。 透過這種分層的設計，UCX 能夠提供高效且可擴展的通訊機制。

2. What is the specific significance of the division of UCP Objects in the program? What important information do they carry?

   三者各自承載了不同的資訊和功能，所以需要拆分成三個層次，每個層次向下擁有特定的物件，以對開發者更友善。

   - `ucp_context` :
     - 承載的資訊：它包含了全域配置，如記憶體管理、傳輸資源、網路介面和協定配置。
     - 對開發者的好處： 通過提供一個統一的 `context` 環境，使得開發者不需要在每個操作中重複配置這些基本資訊，簡化了初始化和資源管理的過程。
   - `ucp_worker`
     - 承載的資訊： `worker` 負責處理具體的通信任務。 它管理了與特定執行線程或進程相關的事件和資源，如消息佇列和網路事件。而在處理傳輸的過程，包含了初始化 connection、分配 worker 資源、interrupt 的處理和事件處理完後的終止等等。
     - 對開發者的好處： 開發者可以根據應用程式的需求，靈活創建和管理多個 `worker` ，實現多個 processes 或 threads 中的平行處理和 load balance。
   - `ucp_ep`
     - 承載的資訊： `endpoint` 表示與特定遠端實體的連接，承載了與這個特定連接相關的狀態和資源，如遠端位址和連接狀態。例如建立 client endpoint 或 server endpoint。
     - 對開發者的好處： 允許開發者精確控制與特定遠端實體的通信，使得數據傳輸更加直接和高效。

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:

在UCX（Unified Communication X）框架中，`UCX_TLS` 選項的設置通常發生在初始化階段，特別是在創建 `ucp_context` 物件時。這是因為 `ucp_context` 負責管理全域配置和資源，包括記憶體、傳輸資源和網路介面。

當使用 UCX 時，可以通過以下步驟配置和傳遞 `UCX_TLS` 資訊：

i. **讀取和設置配置（在 `ucp_context` 建立之前）：**
- 在創建 `ucp_context` 之前，需要讀取和（可選地）設置 UCX 配置。這通常通過調用 `ucp_config_read` 函數完成。在這個階段，可以透過環境變數或直接修改配置來指定 `UCX_TLS` 的值。

ii. **初始化 UCP Context：**
- 建立 `ucp_context` 時，UCX 將使用之前讀取和設置的配置。這包括 `UCX_TLS` 設定，決定了 UCX 將使用哪些傳輸層安全協定和網路介面。
- 在 `ucp_hello_world.c` 中，調用 `ucp_init` 函數來初始化 UCP context。這個函數根據提供的配置（包括 `UCX_TLS` 設置）初始化 UCX 環境。

iii. **UCX 環境啟動：**
- 一旦 `ucp_context` 被成功創建，它會啟動 UCX 環境，並使用之前指定的 `UCX_TLS` 配置。

因此，`UCX_TLS` 的配置是在創建 `ucp_context` 時處理的，這是 UCX 框架中最高層次的物件，負責整個環境的設置和資源管理。透過在這個層次上設置 `UCX_TLS`，可以確保整個 UCX 環境都遵循相應的傳輸層配置。

TLS selected by UCX:

雖然 `UCX_TLS` 的初始設置是在 `ucp_context` 層級進行的，但具體到每個 `endpoint` 的配置和使用，則是在 `ucp_worker` 層級和 `ucp_ep` 配置過程中實現的。在這個過程中，會根據全域設置（如 `UCX_TLS`）和特定的 `endpoint` 需求來調整和優化通信方式。這樣的設計使 UCX 能夠在保持全域配置的同時，為不同的通信連接提供靈活的調整和優化。

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?

- 我在 `worker.c` 裡面 ucp_worker_get_ep_config() 最後 call ucp_worker_print_used_tls() 的這個 else block 的一開頭先印出 Line 1，並且在 `ucp_worker_print_used_tls()` 最後把 debug 資訊印到 `UCX_LOG_LEVEL=info` ucs_info("%s", ucs_string_buffer_cstr(&strb)); 的這個地方，額外用 `printf("%s\n", ucs_string_buffer_cstr(&strb));` 來印出line2。

- 這裡我先在 src/ucs/config/types.h 中額外的建立一個新的 `UCS_CONFIG_PRINT_TLS` 這個 flag，透過這個 flag 來設定為要印出作業要求的 Line1。

```
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;
```

- 接下來的做法是先建一個空的 `config`，然後使用 `ucp_config_read()` 去把 `worker->context->config.env_prefix` 中的內容讀出來存到 `config` 裡面，接著再用 `ucp_config_print(config, stdout, NULL, UCS_CONFIG_PRINT_TLS)` 去調用我們在 `parser.c` 裡寫好的 todo 部分，並且把 output 輸出到 `stdout`，才能在 terminal 上看到 output。

in worker.c,

```
} else {
        ucp_config_t *config;
        ucp_config_read(worker->context->config.env_prefix, NULL, &config);
        ucp_config_print(config, stdout,
                         NULL, UCS_CONFIG_PRINT_TLS);
        ucp_worker_print_used_tls(worker, ep_cfg_index);
    }
```

in parser.c,

```
// TODO: PP-HW4
if (flags & UCS_CONFIG_PRINT_TLS) {

    char *env_value;

    // Retrieve the value of the UCX_TLS environment variable
    env_value = getenv("UCX_TLS");
    if (env_value != NULL) {
        printf("UCX_TLS=%s\n", env_value);
    }
}
```

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

- 互相調用的流程如同第一點所提到的，這裡另外說明在 `worker.c` 中 `worker` 會透過 ucp_worker_get_ep_config() 來讀取各個 `endpoint` 的設定，具體來說在這個 function 他會藉由 ucp_worker_print_used_tls(worker, ep_cfg_index) 將對應的 `endpoint` config 資訊 (Line2) 給印出來，也就是實際上 UCX 所選用的傳輸方式，Line1 則是原本 `context` 中設定好的環境內容。
- 為何要用這種有層次的架構，我認為原因在於 `context` 中儲存的是 global 資訊，而實際上各 `endpoint` 所選用的設定可能會各自有所不同，所以兩者才會拆分開來變成各自的一個 function，這種設計使得代碼模組化並且重用性更高，易於維護和更新。
- 另一個分層架構的原因則可能是類似於 UCP Object 分工的機制，在 `worker` 中會記錄一些過程中用到的參數，然而在 `worker` 中不用理會使用者想印出的 `configs` 是哪些，只要選擇適當的 `flag`，讓 `parser.c` 程式幫忙印出即可，同一時間 `worker` 可以繼續往下執行其他工作，達成平行化效果。

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

- 在我們的實作中如同第一點的說明，Line1 是透過 `ucp_config_print()` 所印出來，而 Line2 則是透過 `ucp_worker_print_used_tls()` 所印出來。
- 而在我們使用 `UCX_LOG_LEVEL=info` 進行 debug 的測試中，可以看到 Line1 在 ucp_context.c:2119 就被印出來了，實際上這一行是在 ucp_version_check()，也就是在檢查 version 的時候；而 Line2 則是在 ucp_worekr.c 中 `ucp_worker_print_used_tls()` 這個 function 的最後面被印出來的。

```
[pp23s88@apollo31 mpi]$ mpiucx -n 2 -x UCX_TLS=all -x UCX_LOG_LEVEL=info ./send_recv.out
[1704616739.153440] [apollo31:2230101:0]     ucp_context.c:2119 UCX  INFO  Version 1.15.0 (loaded from /home/pp23/pp23s88/hw4/lib
[1704616739.171131] [apollo31:2230113:0]     ucp_context.c:2119 UCX  INFO  Version 1.15.0 (loaded from /home/pp23/pp23s88/hw4/lib
[1704616739.220423] [apollo31:2230113:0]          parser.c:2057 UCX  INFO  UCX_* env variables: UCX_TLS=all UCX_LOG_LEVEL=info UC
[1704616739.220460] [apollo31:2230101:0]          parser.c:2057 UCX  INFO  UCX_* env variables: UCX_TLS=all UCX_LOG_LEVEL=info UC
[1704616739.221248] [apollo31:2230113:0]     ucp_context.c:2119 UCX  INFO  Version 1.15.0 (loaded from /home/pp23/pp23s88/hw4/lib
[1704616739.221254] [apollo31:2230101:0]     ucp_context.c:2119 UCX  INFO  Version 1.15.0 (loaded from /home/pp23/pp23s88/hw4/lib
UCX_TLS=all
0x561f037bbb30 self cfg#0 tag(self/memory cma/memory)
[1704616739.225223] [apollo31:2230101:0]      ucp_worker.c:1857 UCX  INFO    0x561f037bbb30 self cfg#0 tag(self/memory cma/memory
UCX_TLS=all
0x5608beb77bb0 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x561f037bbb30 intra-node cfg#1 tag(sysv/memory cma/memory)
[1704616739.225399] [apollo31:2230101:0]      ucp_worker.c:1857 UCX  INFO    0x561f037bbb30 intra-node cfg#1 tag(sysv/memory cma/
[1704616739.225395] [apollo31:2230113:0]      ucp_worker.c:1857 UCX  INFO    0x5608beb77bb0 self cfg#0 tag(self/memory cma/memory
UCX_TLS=all
0x5608beb77bb0 intra-node cfg#1 tag(sysv/memory cma/memory)
[1704616739.225561] [apollo31:2230113:0]      ucp_worker.c:1857 UCX  INFO    0x5608beb77bb0 intra-node cfg#1 tag(sysv/memory cma/
Process 0 sent message 'Hello from rank 0' to process 1
Process 1 received message 'Hello from rank 0' from process 0
```

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

- 以上的操作流程基本上就是遵循 **1-3** 的預想去實作的，這裡我們可以清楚的看到我們先印出 `context` 中的 `UCX_TLS` 全域設定，接著再取得各個 `endpoint` 的資訊印出 `TLS selected by UCX`，也就是 UCX 實際選用的傳輸方式。

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：通常存儲有關 endpoint 的不同傳輸通道（lanes）的資訊。 每個 lane 代表一個特定的傳輸方式或路徑。
- tl_rsc：代表特定的傳輸層資源，可能是指向具體網路介面或硬體資源的指標，和資源配置有關。
- tl_name：傳輸層的名稱，例如「rdma」或「tcp」。
- tl_device：指代用於該傳輸層的具體設備，如特定的網卡或介面。

- bitmap：通常用於表示一組資源或配置的狀態，每一位代表不同的設置或資源，像我們實作中用到的 `UCS_CONFIG_PRINT_TLS` flag 即為 bitmap 的一種應用
- iface：介面（interface）的縮寫，通常指代特定傳輸層的介面或端點，可能是和網路介面有關的訊息。

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
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/osu_latency
mpiucx -n 2 $HOME/UCX-lsalab/test/mpi/osu/pt2pt/standard/osu_bw
```

在這個情境下我們只需要對在同一個 node 上的傳輸做優化，所以使用 shared memory 相關的方式會是比預設的 `ud_verbs` 更快的設定，基於這個想法我們直接在 bash 使用 `export UCX_TLS=all` 指令，設定讓 UCX 自動的選擇它認為最快的傳輸方式(self/memory cma/memory sysv/memory)，並且在傳輸時對兩者各自進行 load balance。 根據 `mpiucx -n 2 -x UCX_TLS=all ./send_recv.out` 的結果我們可以看到：

```
[pp23s88@apollo31 mpi]$ mpiucx -n 2 -x UCX_TLS=all ./send_recv.out
UCX_TLS=all
0x559b98b1de50 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x564f6c492dc0 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x559b98b1de50 intra-node cfg#1 tag(sysv/memory cma/memory)
UCX_TLS=all
0x564f6c492dc0 intra-node cfg#1 tag(sysv/memory cma/memory)
Process 1 received message 'Hello from rank 0' from process 0
Process 0 sent message 'Hello from rank 0' to process 1
```

在 process 內以及 process 之間的傳輸，UCX 分別選了 (self/memory cma/memory) 以及 (sysv/memory cma/memory) 兩組設定。

1. self 傳輸層：

- self 傳輸層專門用於進程內的通信。
- 當訊息的發送和接收都在同一個 process 內時會使用 `self` 傳輸層。
- 這種情況通常出現在多線程應用程式中，其中同一個 process 的不同 thread 需要相互通信。
- 使用傳輸層可以避免不必要的數據複製和網路堆疊的使用，從而提高進程內通信的效率。

2. intra-node 傳輸層：

- intra-node 傳輸層用於同一 node 上不同 process 間的通信。
- 它涵蓋了 node 裡面不同 process 之間的通信場景，比如使用共用記憶體（如 POSIX 共用記憶體、System V 共用記憶體等）。
- 在同一實體機或虛擬機上運行的不同進程之間進行通信時， 傳輸層提供了高效的數據交換機制。intra-node
- 它優化了節點內通信，減少了跨網路通信的開銷，適用於多進程並行應用程式。

設定完成前後都分別使用 `osu_latency` 以及 `osu_bw` 去觀察傳輸速度以及延遲的變化，由此我們得到以下的兩組 output。對其進行比較後可以發現，在設定完 `UCS_TLS=all` 之後的 Bandwidth 明顯的上升、Latency 明顯的下降了，也就證明了這個做法可以優化系統的預設設定。

Original			Using `UCX_TLS=all`		
# Size	Latency (us)	Bandwidth (MB/s)	# Size	Latency (us)	Bandwidth (MB/s)
1	2.04	2.88	1	0.23	10.01
2	1.53	5.89	2	0.22	20.09
4	2.11	11.72	4	0.22	40.39
8	1.53	23.42	8	0.22	80.73
16	1.65	45.52	16	0.22	160.82
32	1.9	48.51	32	0.27	314.09
64	1.75	116.36	64	0.27	600.3
128	1.9	282.24	128	0.39	641.28
256	3.15	403.21	256	0.41	1215.34
512	3.95	681.36	512	0.45	2334.87
1024	4.28	1189.42	1024	0.53	3854.95
2048	5.75	1707.94	2048	0.91	5527.51
4096	9.15	1744.61	4096	1.01	8066.33
8192	12.42	2003.74	8192	1.74	10121.98
16384	16.76	2267.66	16384	3	4990.58
32768	22.51	1708.62	32768	4.97	6743.98
65536	36.33	2423.13	65536	9.97	7785.66
131072	62.64	2212.32	131072	17.62	7428.13
262144	124.23	2374.53	262144	38.3	8259.9
524288	242.09	2489.26	524288	69.38	7550.12
1048576	458.43	2220.91	1048576	142.1	7580.71
2097152	930.72	2414.17	2097152	339.54	7514.84
4194304	1807.49	2344.78	4194304	1006.37	6855.85

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
```

在 multinode 的情況下，無論我選用預設的 `UCX_TLS=ud_verbs` 或者改成 `UCX_TLS=all` / `UCX_TLS=sm,mm`，UCX在多節點傳輸的情況下都會幫我選擇使用 `rc_verbs/ibp3s0:1 tcp/ibp3s0`，因此前一題針對單節點的優化，在這裡的測試沒辦法使得效能提升。以下為使用 run.sbatch 的指令，以及 log file 的內容。

```
#!/bin/bash
#SBATCH --job-name=$USER-ucx
#SBATCH --output=result_%J.out
#SBATCH --nodes=2
#SBATCH --ntasks=2
#SBATCH --time=00:01:00
```

```
#SBATCH --mem=1000
#SBATCH --partition=test

module load openmpi/4.1.5

ucx_lib_directory="$HOME/hw4/lib"

ld_preload_paths=$(find "$ucx_lib_directory" -name 'libucp.so.0' -o -name 'libuct.so.0' -o -name 'libucm.so.0' -o -name 'libucs.so.0'

echo $ld_preload_paths

srun --export LD_PRELOAD=${ld_preload_paths}:${LD_PRELOAD} ./mpi/osu/pt2pt/standard/osu_latency ./mpi/send_recv_multinode.out
srun --export LD_PRELOAD=${ld_preload_paths}:${LD_PRELOAD} ./mpi/osu/pt2pt/standard/osu_bw ./mpi/send_recv_multinode.out
```

```
/home/pp23/pp23s88/hw4/lib/libucm.so.0:/home/pp23/pp23s88/hw4/lib/libucs.so.0:/home/pp23/pp23s88/hw4/lib/libuct.so.0:/home/pp23/pp23s
0x55fd027b1d10 self cfg#0 tag(self/memory cma/memory rc_verbs/ibp3s0:1)
0x55acb8e4e630 self cfg#0 tag(self/memory cma/memory rc_verbs/ibp3s0:1)
# OSU MPI Latency Test v7.3
# Size          Latency (us)
# Datatype: MPI_CHAR.
0x55acb8e4e630 inter-node cfg#1 tag(rc_verbs/ibp3s0:1 tcp/ibp3s0)
1                       2.09
2                       2.06
4                       2.05
8                       2.06
16                      2.06
32                      2.06
64                      2.20
128                     3.38
256                     3.51
512                     3.80
1024                    4.39
2048                    5.45
4096                    7.67
8192                    9.82
16384                  13.13
32768                  18.80
65536                  30.12
131072                 53.11
262144                 95.36
524288                182.24
1048576               355.11
2097152               702.34
4194304              1396.76
0x55fd027b1d10 inter-node cfg#1 tag(rc_verbs/ibp3s0:1 tcp/ibp3s0)
0x557ea7fb6660 self cfg#0 tag(self/memory cma/memory rc_verbs/ibp3s0:1)
# OSU MPI Bandwidth Test v7.3
# Size      Bandwidth (MB/s)
# Datatype: MPI_CHAR.
0x55f1a44657f0 self cfg#0 tag(self/memory cma/memory rc_verbs/ibp3s0:1)
0x557ea7fb6660 inter-node cfg#1 tag(rc_verbs/ibp3s0:1 tcp/ibp3s0)
1                       3.04
2                       6.17
4                      12.03
8                      24.85
16                     49.60
32                     99.78
64                    183.83
128                   256.31
256                   491.18
512                   920.72
1024                 1528.61
2048                 2169.38
4096                 2525.93
8192                 2705.35
16384                2841.17
32768                2910.54
65536                2943.64
131072               2957.27
262144               2965.92
524288               3028.15
1048576              3032.17
2097152              3034.64
```

```
4194304              3035.67
0x55f1a44657f0 inter-node cfg#1 tag(rc_verbs/ibp3s0:1 tcp/ibp3s0)
```

- 在 output 中可以看到這時變成了 inter-node，UCX 在多節點時幫我選擇的是 `rc_verbs/ibp3s0:1` `tcp/ibp3s0` 這兩組設定，具體來說使用了 InfiniBand 的 rc_verbs (Reliable Connection) 和 TCP/IP 網路傳輸。
- 至於為何不管怎麼設定 `UCX_TLS`，UCX 都會幫我選擇特定的傳輸方式，原因可能是 UCX 在選擇傳輸層（TLS）時會考慮到當前的硬體和網路環境。如果預設配置已經選擇了最適合當前環境的傳輸層，那麼即使我將 `UCX_TLS` 設定更改後，UCX 在實際操作中可能仍然選擇了相同的傳輸方式。這代表實際使用的傳輸層在兩種配置下是一樣的，從而導致幾乎相同的 latency/ bandwidth 表現，也就是多節點的情況下效能並沒有提升。

4. Experience & Conclusion

1. What have you learned from this homework?

   在完成這次作業的過程中，我對 UCX 的整體架構有了更深入的理解。它不但是一個高效的通信框架，還能夠根據不同的硬體和網路環境自動選擇最佳的傳輸方式。在 trace code 的過程中我也將上課時老師提到的抽象概念對應到實作中，從而對它整體的系統設計概念有了具體的認識。

   這次的作業都圍繞在 UCX_TLS 這個環境變數上，我們透過它掌握了如何配置和優化 UCX 環境；而為了在解決作業中遇到的各種問題，也練習到如何查看 log file、找出 types 定義在整份專案的哪個位置、各個 API 需要代入什麼樣的參數、查閱官方文檔.....，以上這些經驗都使得我訓練到快速熟悉一個大型專案的能力，這些學習的過程對於未來進入業界，或是在實驗室做計畫各方面都會是很實用的經驗。

2. Feedback (optional)

   這次的作業形式跟前三份不太一樣，主要是以 trace code 為主，我覺得是一個很好的訓練，感謝助教辛苦的規劃。