hw4_report

110590007白宸安

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

在ucp.h中

https://github.com/NTHU-LSALAB/UCX-lsalab/blob/7c0362c97c8fe9cbeaacaac90271dde0210ac529/src/ucp/api/ucp.h#L2057

ucp_init會呼叫ucp_init_version來進行初始化。

而ucp_init_version是ucp_context中進行初始化的部分。

https://github.com/NTHU-LSALAB/UCX-lsalab/blob/7c0362c97c8fe9cbeaacaac90271dde0210ac529/src/ucp/core/ucp_context.c#L2123

ucp_init是用來初始化ucp_context，一個program需要一個ucp_context。

ucp_worker_create

在ucp.h中

https://github.com/NTHU-LSALAB/UCX-lsalab/blob/7c0362c97c8fe9cbeaacaac90271dde0210ac529/src/ucp/api/ucp.h#L2140

有定義ucp_worker_create函式，實作是在ucp_worker.c中

https://github.com/NTHU-LSALAB/UCX-lsalab/blob/7c0362c97c8fe9cbeaacaac90271dde0210ac529/src/ucp/core/ucp_worker.c#L2311

ucp_worker_create是用來創建一個worker，每個ucp_context下可以有多個worker。

ucp_ep_create

在ucp.h中

https://github.com/NTHU-LSALAB/UCX-lsalab/blob/7c0362c97c8fe9cbeaacaac90271dde0210ac529/src/ucp/api/ucp.h#L2575

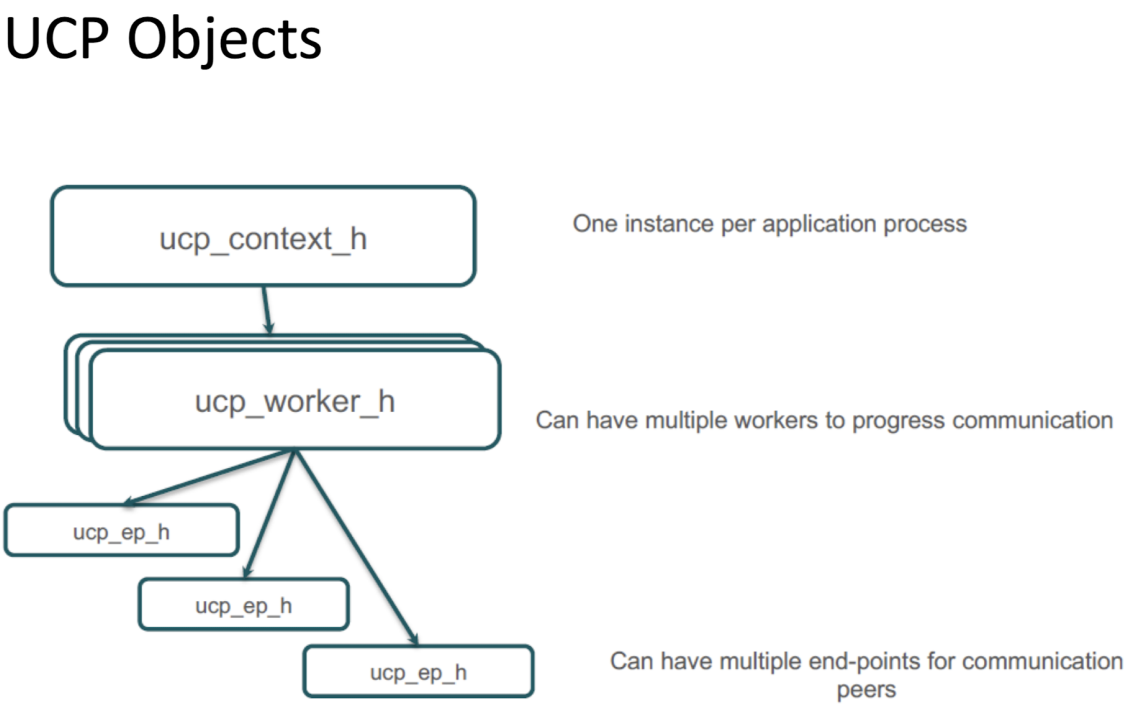
定義了ucp_ep_create函式，實作是在ucp_ep.c中

https://github.com/NTHU-LSALAB/UCX-lsalab/blob/7c0362c97c8fe9cbeaacaac90271dde0210ac529/src/ucp/core/ucp_ep.c#L1176

ucp_ep_create是用來在worker下創建end points，每個worker下可以有多個end points。

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

為什麼要這麼設計？



`ucp_context`

這個負責管理application的global context，把global information和有關communication的設定都包裝好。

重要性：裡面包含了memory allocation設定、configuration設定、其他application level的參數、global device context等。

`ucp_worker`

負責communication和處理engine context，也要管理跟這些operation有關的資源。一個context下可以有多個worker一起處理。要handle transport相關的resource、interrupts、connection建立、completion events等。

重要性：因為有分層，所以能有多個worker，這樣就能進行平行化，比如1個thread當1個worker，能更有效率地處理communication task。

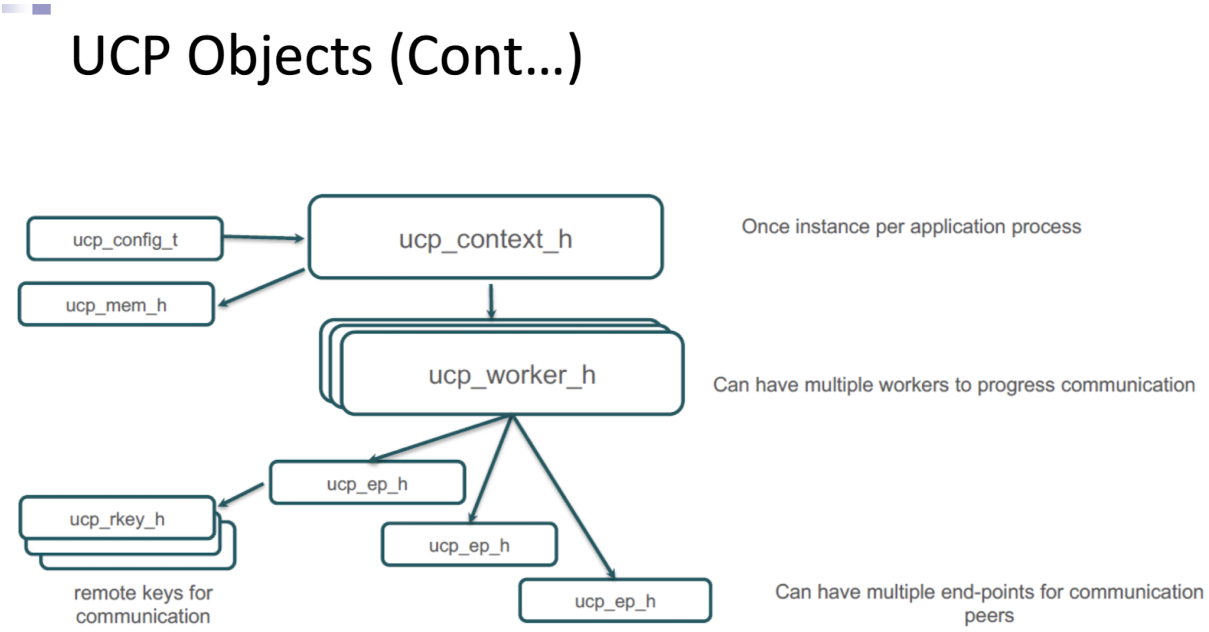
`ucp_ep`

代表兩個process間的一條connection。所有的send operation都是藉由end point來進行。用來讓process間建立連線，如此能實現平行計算。一個worker下一樣能有多個end point。

重要性：有了ucp_ep這個object，就能讓process間建立起direct communication channel，讓彼此傳輸時間更快，而重要的點在於因為能建立多個end point，所以也能更有效地善用平行的效能。

總結，切成三種UCP objects的好處在於能夠分層管理，如果全部混雜成一個class會變得難以管理。另外還有能使用多個worker和多個end point，這樣能發揮平行計算的優勢。

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



`UCX_TLS`

根據作業spec的描述，UCX_TLS是可以透過flag來指定的環境變數，所以應該會在parser.c裡面處理環境變數的部分。在trace code時也有發現這樣的現象。

TLS指得是Transport Layer Security，因為是和建立連線時有關，所以認為應該是在ucp_worker裡面。

TLS selected by UCX

```
mpiucx -n 2 -x UCX_TLS=all ./send_recv.out
```

如果有在跑程式時下參數，UCX_TLS就會使用參數給的，另外UCX會自動選擇transport protocols

比如沒有指定時

```
mpiucx -np 1 ./mpi_hello.out
UCX_TLS=ud_verbs
0x56544467c8f0 self cfg#0 tag(ud_verbs/ibp3s0:1)
Hello world from processor apollo31, rank 0 out of 1 processors
```

apollo31會預設使用ud_verbs

而有指定時

```
mpiucx -n 2 -x UCX_TLS=all ./send_recv.out
UCX_TLS=all
0x5580604e1a90 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x559a90bb7a00 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x5580604e1a90 intra-node cfg#1 tag(sysv/memory cma/memory)
UCX_TLS=all
0x559a90bb7a00 intra-node cfg#1 tag(sysv/memory cma/memory)
Process 0 sent message 'Hello from rank 0' to process 1
Process 1 received message 'Hello from rank 0' from process 0
```

UCX會去選擇self/memory和sysv/memory

因為也是建立連線時選擇，所以認為應該也是在ucp_worker裡面實作。

2. Implementation

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

有修改了ucp_worker.c、parser.c、types.h這三個檔案

`ucp_worker.c`

這邊主要改了兩個部分，在

https://github.com/NTHU-LSALAB/UCX-lsalab/blob/7c0362c97c8fe9cbeaacaac90271dde0210ac529/src/ucp/core/ucp_worker.c#L1855

加上這些部分

```
// modify here
ucs_info("%s", ucs_string_buffer_cstr(&strb));

//ucp_config_print(NULL, stdout, NULL, UCS_CONFIG_PRINT_TLS);
ucs_config_parser_print_env_vars_once(context->config.env_prefix);
fprintf(stdout, "%s\n",  ucs_string_buffer_cstr(&strb));
```

其中的ucs_config_parser_print_env_vars_once(context->config.env_prefix);會將Line 1資訊印出。

而fprintf會將Line 2的資訊印出。

另外一個地方在

https://github.com/NTHU-LSALAB/UCX-lsalab/blob/7c0362c97c8fe9cbeaacaac90271dde0210ac529/src/ucp/core/ucp_worker.c#L2504

這邊改成

```
// modify here
//ucs_config_parser_print_env_vars_once(context->config.env_prefix);
```

讓create worker時不要呼叫到印出env_vars的部分，因為UCX_TLS是包含在env_vars裡面。

`parser.c`

這邊改了三個部分

首先是ucs_config_parser_print_env_vars

https://github.com/NTHU-LSALAB/UCX-lsalab/blob/7c0362c97c8fe9cbeaacaac90271dde0210ac529/src/ucs/config/parser.c#L1995

改成，讓它不要因為沒有要印info就直接把ucs_config_parser_print_env_vars給return，如此後續才能印出UCX_TLS=xxx。

```
// modify here
// if (!ucs_config_parser_env_vars_track()) {
//     return;
// }
```

接著一樣是改ucs_config_parser_print_env_vars

https://github.com/NTHU-LSALAB/UCX-lsalab/blob/7c0362c97c8fe9cbeaacaac90271dde0210ac529/src/ucs/config/parser.c#L2019

這邊加上

```
// modify here
if (strcmp(var_name, "UCX_TLS") == 0) {
    fprintf(stdout, "UCX_TLS=%s\n", saveptr);
    //char *ucs_ucx_tls_value = NULL;
    //ucs_ucx_tls_value = malloc(strlen(saveptr) + 1);
    //strcpy(ucs_ucx_tls_value, saveptr);;
}
```

從輸出info的部分擷取出UCX_TLS，再把這項資訊用fprintf(stdout)印出。

也就是說現在呼叫ucs_config_parser_print_env_vars這個function時就會將UCX_TLS先印出一次，後續如果有要印出info資訊一樣會再印一次。

最後是改ucs_config_parser_print_env_vars_once()

https://github.com/NTHU-LSALAB/UCX-lsalab/blob/7c0362c97c8fe9cbeaacaac90271dde0210ac529/src/ucs/config/parser.c#L2068

這邊改成

```
const char   *sub_prefix = NULL;
//int          added;
ucs_status_t status;

//modify here
//fprintf(stdout, "once\n");

/* Although env_prefix is not real environment variable put it
 * into table anyway to save prefixes which was already checked.
```

```
 * Need to save both env_prefix and base_prefix */
// ucs_config_parser_mark_env_var_used(env_prefix, &added);
// //fprintf(stdout, "once_end\n");
// if (!added) {
//     return;
// }

ucs_config_parser_print_env_vars(env_prefix);
//fprintf(stdout, "env_prefix\n");

status = ucs_config_parser_get_sub_prefix(env_prefix, &sub_prefix);
if (status != UCS_OK) {
    return;
}

if (sub_prefix == NULL) {
    return;
}

// ucs_config_parser_mark_env_var_used(sub_prefix, &added);
// if (!added) {
//     return;
// }

ucs_config_parser_print_env_vars(sub_prefix);
//fprintf(stdout, "sub_prefix\n");
```

基本上就是讓ucs_config_parser_print_env_vars_once在不用印出info時，也能夠呼叫到ucs_config_parser_print_env_vars，就能夠印出UCX_TLS。

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

Line 1

- 在ucp_ep.c中

在end point初始化時會開始路線。

ucp_ep_create → ucp_ep_create_to_sock_addr → ucp_ep_init_create_wireup → ucp_worker_get_ep_config

- 在ucp_worker.c中

ucp_worker_get_ep_config → ucp_worker_print_used_tls → ucs_config_parser_print_env_vars_once

- 在parser.c中

ucs_config_parser_print_env_vars_once → ucs_config_parser_print_env_vars → fprintf(stdout, "UCX_TLS=%s\n", saveptr)

這條路會印出Line 1

Line 2

- 在ucp_ep.c中

在end point初始化時會開始路線。

ucp_ep_create → ucp_ep_create_to_sock_addr → ucp_ep_init_create_wireup → ucp_worker_get_ep_config

- 在ucp_worker.c中

ucp_worker_get_ep_config → ucp_worker_print_used_tls → fprintf(stdout, "%s\n", ucs_string_buffer_cstr(&strb));

這條路會印出Line 2

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

根據2的描述，Line 1和Line 2會從以上路線被print out，也就是在ucp_ep_create這個UCP API被call以後會一路呼叫function，最終達到印出Line 1和Line 2。

因為同一個worker下建立end points時會使用相同的configuration，所以會去呼叫worker來取得config資訊，也就會呼叫到印出相關config的function，接著修改這些function後就可以達到要印出Line 1和Line 2的結果。

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

應該算有符合預期，因為會印出環境參數的部分確實會印出UCX_TLS的資訊，而實作上也確實有修改到處理環境變數的parser.c。

UCX_TLS selected by UCX這部分，認為是在建立連線的時候所設定，所以在ucp_worker.c裡面做修改，也確實符合預期，會印出Line 2資訊的地方確實在ucp_worker.c裡面。

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：

表示用來communication的連線，worker底下end point所建立起的連線

tl_rsc：

Transport Layer Resource，指跟Transport Layer有關的資源，比如buffer、socket、system device、tl_name、tl_device等。

使用ucp_tl_resource_desc_t這個type來儲存以上資訊。

tl_name：

Transport Layer Name，指Transport Layer的名稱或identifier。

tl_device：

Transport Layer Device，指特定的硬體device或network device等。

bitmap：

tl相關資源的map。

因為code中有用到ucp_tl_bitmap_t這個type。

ifaces：

interface，會根據bitmap的tl resource來開啟interface，如果bitmap還沒設定，就會開啟所有available resource的interface並選擇最好的那個，把它存進context的bitmap裡，之後的worker就能直接選用。

interface指得是worker到特定communication的接口，比如network interface或shared memory等。

3. Optimize System

根據觀察，發現將UCX_TLS設定為all就會有較好的效果，因為是在single node，不需要走網路線，但apollo31會預設使用ud_verbs。

```
setenv UCX_TLS all
```

設定成all時，會讓UCX自動去看有哪些tls可以用，並選擇最好的，因為不用跨節點，不用走到網路線，所以選用memory溝通會比走網路的速度還快。

以下是osu_latency測試結果

可以發現使用UCX_TLS=all確實能讓latency降低2~3倍。

```
mpiucx -n 2 ~/hw4/UCX-lsalab/test/mpi/osu/pt2pt/standard/osu_latency
UCX_TLS=ud_verbs
0x55e8803d5dc0 self cfg#0 tag(ud_verbs/ibp3s0:1)
UCX_TLS=ud_verbs
0x55d1950d4e50 self cfg#0 tag(ud_verbs/ibp3s0:1)
UCX_TLS=ud_verbs
0x55d1950d4e50 intra-node cfg#1 tag(ud_verbs/ibp3s0:1)
UCX_TLS=ud_verbs
0x55e8803d5dc0 intra-node cfg#1 tag(ud_verbs/ibp3s0:1)
# OSU MPI Latency Test v7.3
# Size          Latency (us)
# Datatype: MPI_CHAR.
1                       1.53
2                       1.52
4                       1.52
8                       1.52
16                      1.65
32                      1.67
64                      1.74
128                     2.07
256                     3.11
512                     3.47
1024                    4.16
2048                    5.84
4096                    8.99
8192                   13.06
16384                  16.61
```

```
32768                    22.38
65536                    38.32
131072                   64.61
262144                  126.63
524288                  240.11
1048576                 454.40
2097152                 922.11
4194304                1838.45
```

```
mpiucx -n 2 -x UCX_TLS=all ~/hw4/UCX-lsalab/test/mpi/osu/pt2pt/standa
UCX_TLS=all
0x5613a0e99dc0 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x563790565e50 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x563790565e50 intra-node cfg#1 tag(sysv/memory cma/memory)
UCX_TLS=all
0x5613a0e99dc0 intra-node cfg#1 tag(sysv/memory cma/memory)
# OSU MPI Latency Test v7.3
# Size          Latency (us)
# Datatype: MPI_CHAR.
1                       0.21
2                       0.21
4                       0.21
8                       0.21
16                      0.22
32                      0.27
64                      0.25
128                     0.41
256                     0.43
512                     0.47
1024                    0.55
2048                    0.70
4096                    1.02
8192                    1.70
16384                   3.01
32768                   4.92
65536                   8.61
131072                 17.24
262144                 37.96
524288                 69.38
1048576               133.76
```

```
2097152                320.03
4194304                964.10
```

以下是osu_bw測試結果

可以看到用UCX_TLS=all可以讓bandwidth增加約3倍多。

```
mpiucx -n 2 ~/hw4/UCX-lsalab/test/mpi/osu/pt2pt/standard/osu_bw

UCX_TLS=ud_verbs
0x55adc8a9bea0 self cfg#0 tag(ud_verbs/ibp3s0:1)
UCX_TLS=ud_verbs
0x557277c1ae10 self cfg#0 tag(ud_verbs/ibp3s0:1)
UCX_TLS=ud_verbs
0x55adc8a9bea0 intra-node cfg#1 tag(ud_verbs/ibp3s0:1)
UCX_TLS=ud_verbs
0x557277c1ae10 intra-node cfg#1 tag(ud_verbs/ibp3s0:1)
# OSU MPI Bandwidth Test v7.3
# Size      Bandwidth (MB/s)
# Datatype: MPI_CHAR.
1                       2.90
2                       5.89
4                      11.14
8                      21.56
16                     42.19
32                     84.24
64                    153.59
128                   291.79
256                   385.10
512                   707.76
1024                 1156.93
2048                 1664.35
4096                 1753.63
8192                 1308.08
16384                2247.82
32768                2359.63
65536                2432.64
131072               2316.47
262144               2327.24
524288               2490.46
1048576              2310.35
```

```
2097152              2462.19
4194304              2372.77
```

```
mpiucx -n 2 -x UCX_TLS=all ~/hw4/UCX-lsalab/test/mpi/osu/pt2pt/standa
UCX_TLS=all
0x55b8a6dd3ea0 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x564f1ea29e10 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x564f1ea29e10 intra-node cfg#1 tag(sysv/memory cma/memory)
UCX_TLS=all
0x55b8a6dd3ea0 intra-node cfg#1 tag(sysv/memory cma/memory)
# OSU MPI Bandwidth Test v7.3
# Size      Bandwidth (MB/s)
# Datatype: MPI_CHAR.
1                     9.60
2                    19.22
4                    38.58
8                    77.35
16                  154.26
32                  302.86
64                  602.28
128                 603.59
256                1195.09
512                2287.82
1024               3712.59
2048               5378.16
4096               7719.59
8192               9825.28
16384              4791.72
32768              6488.04
65536              7948.92
131072             8913.28
262144             7599.19
524288             7861.24
1048576            8093.03
2097152            8074.20
4194304            6945.18
```

Advanced Challenge: Multi-Node Testing

使用run.batch來測試，發現multi-node下，會比single-node使用預設的UCX_TLS時latency在低一些，但比使用UCX_TLS=all的single-node來的高。

Multi-node版的也有嘗試改export UCX_TLS=all和export UCX_NET_DEVICES=ibp3s0:1，但結果跟原先的一樣。

```
0x5646b174e5d0 self cfg#0 tag(self/memory cma/memory rc_verbs/ibp3s0:
# OSU MPI Latency Test v7.3
# Size          Latency (us)
# Datatype: MPI_CHAR.
0x558c87731c90 self cfg#0 tag(self/memory cma/memory rc_verbs/ibp3s0:
0x5646b174e5d0 inter-node cfg#1 tag(rc_verbs/ibp3s0:1 tcp/ibp3s0)
1                       2.12
2                       2.10
4                       2.07
8                       2.09
16                      2.08
32                      2.09
64                      2.19
128                     3.36
256                     3.53
512                     3.78
1024                    4.35
2048                    5.42
4096                    7.61
8192                    9.67
16384                  12.98
32768                  18.74
65536                  30.02
131072                 53.06
262144                 94.86
524288                180.55
1048576               353.24
2097152               698.56
4194304              1390.00
0x558c87731c90 inter-node cfg#1 tag(rc_verbs/ibp3s0:1 tcp/ibp3s0)
```

發現的可能原因是因為inter-node connection會走到tcp。

去查看一下機器的網卡，發現機器是使用Ethernet，因此UCX在inter-node時還是得走tcp那條，這會是一個原因，如果改成infiniband或許會更快。

```
lspci | grep -i net
04:00.0 Ethernet controller: Intel Corporation 82576 Gigabit Network
```

```
(rev 01)
04:00.1 Ethernet controller: Intel Corporation 82576 Gigabit Network
(rev 01)
```

4. Experience & Conclusion

1. What have you learned from this homework?

從這次作業學到最多的應該是UCX的概念，相較於一般七層式的網路架構，UCX能直接bypass OS，達成更好的傳輸效率。

此外也了解到，整個平行系統不單單只是平行程式，很多變快的細節其實是在更加底層的layer比如UCX、網卡、硬體等，要達到好的performance不能僅去優化程式本身，還有很多底層細節其實也要注意。

這個作業感覺讓我對超算的知識有更多了解，期望之後的比賽能活用這次學到的經驗。

2. Feedback (optional)

感謝助教幫忙改作業，新年快樂！