HW5

> - 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` (初始化context)
        - **ucp_init**會呼叫**ucp_init_version**，裡面會初始化一個context並把param和config塞進context，用指標的方式把context傳回去
        - 塞param

```
status = ucp_fill_config(context, params, config);
if (status != UCS_OK) {
    goto err_free_ctx;
}
```

```
status = ucp_fill_resources(context, config);
if (status != UCS_OK) {
    goto err_thread_lock_finalize;
}
```

        - 初始化context lock用來保護ucp_mem_和ucp_rkey_

```
UCP_THREAD_LOCK_INIT(&context->mt_lock);
```

    - `ucp_worker_create` (初始化worker，並把context塞進去)

- 初始化一個worker，負責管理endpoints，並且把context和一些資訊塞進worker

```
ucs_status_t ucp_worker_create(ucp_context_h context,
                               const ucp_worker_params_t *params,
                               ucp_worker_h *worker_p)
{
    ucs_thread_mode_t thread_mode, uct_thread_mode;
    unsigned name_length;
    ucp_worker_h worker;
    ucs_status_t status;

    worker = ucs_calloc(1, sizeof(*worker), "ucp worker");
    if (worker == NULL) {
        return UCS_ERR_NO_MEMORY;
    }

    worker->context              = context;
    worker->uuid                 = ucs_generate_uuid((uintptr_t)worker);
    worker->flush_ops_count      = 0;
    worker->inprogress           = 0;
    worker->rkey_config_count    = 0;
    worker->num_active_ifaces    = 0;
    worker->num_ifaces           = 0;
    worker->am_message_id        = ucs_generate_uuid(0);
    worker->rkey_ptr_cb_id       = UCS_CALLBACKQ_ID_NULL;
    worker->num_all_eps          = 0;
    ucp_worker_keepalive_reset(worker);
    ucs_queue_head_init(&worker->rkey_ptr_reqs);
    ucs_list_head_init(&worker->arm_ifaces);
    ucs_list_head_init(&worker->stream_ready_eps);
    ucs_list_head_init(&worker->all_eps);
    ucs_list_head_init(&worker->internal_eps);
    kh_init_inplace(ucp_worker_rkey_config, &worker->rkey_config_hash);
    kh_init_inplace(ucp_worker_discard_uct_ep_hash, &worker->discard_uct_ep_hash);
    worker->counters.ep_creations         = 0;
    worker->counters.ep_creation_failures = 0;
    worker->counters.ep_closures          = 0;
    worker->counters.ep_failures          = 0;

    /* Copy user flags, and mask-out unsupported flags for compatibility */
    worker->flags = UCP_PARAM_VALUE(WORKER, params, flags, FLAGS, 0) &
                    UCS_MASK(UCP_WORKER_INTERNAL_FLAGS_SHIFT);
    UCS_STATIC_ASSERT(UCP_WORKER_FLAG_IGNORE_REQUEST_LEAK <
                      UCS_BIT(UCP_WORKER_INTERNAL_FLAGS_SHIFT));
```

- worker的thread mode會在上層由使用者決定

```
worker_params.field_mask  = UCP_WORKER_PARAM_FIELD_THREAD_MODE;
worker_params.thread_mode = UCS_THREAD_MODE_SINGLE;
```

- `ucp_ep_create` (初始化endpoint，並把worker塞進去)
  - server端會呼叫**run_ucx_server**，裡面會呼叫**ucp_ep_create**創建endpoint來進行傳輸

```
ep_params.field_mask      = UCP_EP_PARAM_FIELD_REMOTE_ADDRESS |
                            UCP_EP_PARAM_FIELD_ERR_HANDLING_MODE |
                            UCP_EP_PARAM_FIELD_ERR_HANDLER |
                            UCP_EP_PARAM_FIELD_USER_DATA;
ep_params.address         = peer_addr;
ep_params.err_mode        = err_handling_opt.ucp_err_mode;
ep_params.err_handler.cb  = failure_handler;
ep_params.err_handler.arg = NULL;
ep_params.user_data       = &ep_status;

status = ucp_ep_create(ucp_worker, &ep_params, &client_ep);
```

- client端也會呼叫**run_ucx_server**

```
ep_params.field_mask      = UCP_EP_PARAM_FIELD_REMOTE_ADDRESS |
                            UCP_EP_PARAM_FIELD_ERR_HANDLING_MODE |
                            UCP_EP_PARAM_FIELD_ERR_HANDLER |
                            UCP_EP_PARAM_FIELD_USER_DATA;
ep_params.address         = peer_addr;
ep_params.err_mode        = err_handling_opt.ucp_err_mode;
ep_params.err_handler.cb  = failure_handler;
ep_params.err_handler.arg = NULL;
ep_params.user_data       = &ep_status;

status = ucp_ep_create(ucp_worker, &ep_params, &server_ep);
```

- peer_address的取得
  - server會主動把worker的local address的長度送到同一個port底下的client。若client有順利收到，server會再傳一個worker實際的local address給client。而收到的這個server worker local address在client端就是peer_address。
  - server的worker local address取得是透過**ucp_worker_query**

```
status = ucp_worker_query(ucp_worker, &worker_attr);
CHKERR_JUMP(status != UCS_OK, "ucp_worker_query\n", err_worker);
local_addr_len = worker_attr.address_length;
local_addr     = worker_attr.address;
```

```
oob_sock = connect_common(NULL, server_port, ai_family);
CHKERR_JUMP(oob_sock < 0, "server_connect\n", err_peer_addr);

ret = send(oob_sock, &local_addr_len, sizeof(local_addr_len), 0);
CHKERR_JUMP_RETVAL(ret != (int)sizeof(local_addr_len),
                   "send address length\n", err_peer_addr, ret);

ret = send(oob_sock, local_addr, local_addr_len, 0);
CHKERR_JUMP_RETVAL(ret != (int)local_addr_len, "send address\n",
                   err_peer_addr, ret);
```

  - server獲得client的peer_address方法則要在**run_ucx_server**時，透過ucp_tag_msg_recv_nbx接收client端發送的資訊

```
request = ucp_tag_msg_recv_nbx(ucp_worker, msg, info_tag.length,
                               msg_tag, &recv_param);
```

- 收到的peer_address會放在buffer msg + 1的位置

```
memcpy(peer_addr, msg + 1, peer_addr_len);
```

- client在呼叫**run_ucx_client**時會傳入自己透過**ucp_worker_query**得到自己的worker local address

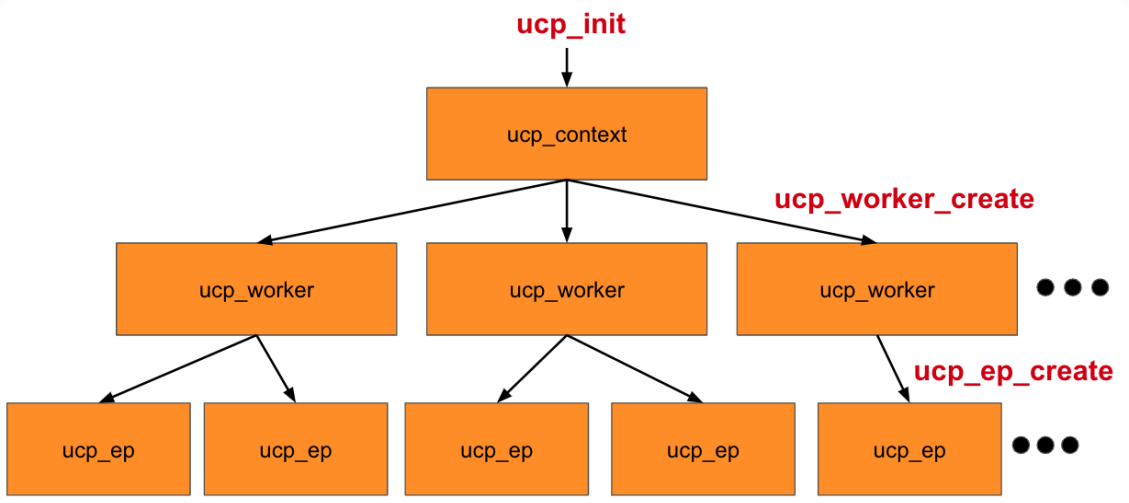
```
ret = run_ucx_client(ucp_worker,
                     local_addr, local_addr_len,
                     peer_addr, peer_addr_len);
```

- 在**run_ucx_client**中會主動傳這個worker local address給server

```
ucp_tag_send_nbx(server_ep, msg, msg_len, tag,
                 &send_param);
```

- 呼叫**ucp_ep_create**時，會把和自己對接的端點地址(i.e. peer_address)塞進去。之後就可以用這個endpoint進行溝通

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`



> Please provide detailed example information in the diagram corresponding to the execution of the command `srun -N 2 ./send_recv.out` or `mpiucx --host HostA:1,HostB:1 ./send_recv.out`

```
ucp_worker.c:2748 UCX  DEBUG destroy worker 0x55e198a88bf0
ucp_worker.c:2734 UCX  DEBUG worker 0x55e198a88bf0: destroy all endpoints
ucp_worker.c:2734 UCX  DEBUG worker 0x55e198a88bf0: destroy internal endpoints
ucp_worker.c:229  UCX  DEBUG worker 0x55e198a88bf0: remove active message handlers
ucp_worker.c:2748 UCX  DEBUG destroy worker 0x55f18227ebd0
ucp_worker.c:2734 UCX  DEBUG worker 0x55f18227ebd0: destroy all endpoints
ucp_worker.c:2734 UCX  DEBUG worker 0x55f18227ebd0: destroy internal endpoints
ucp_worker.c:229  UCX  DEBUG worker 0x55f18227ebd0: remove active message handlers
```

> worker通訊結束後會把自己創建出來的endpoints destroy掉，也把自己destroy掉。

3. Based on the description in HW5, where do you think the following information is loaded/created?
    - `UCX_TLS`
        - UCX_TLS是從環境變數environ抓到的

          ```
          /* Process environment variables */
          extern char **environ;
          ```

        - **ucs_config_parser_print_env_vars**呼叫時，會把環境變數全部parse出來放到**used_vars_strb**裡，並log出來。

          ```
          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, prefix_len)) {
                  ucs_free(envstr);
                  continue; /* Not UCX */
              }

              iter = kh_get(ucs_config_env_vars, &ucs_config_parser_env_vars, var_name);
              if (iter == kh_end(&ucs_config_parser_env_vars)) {
                  if (ucs_global_opts.warn_unused_env_vars) {
                      ucs_string_buffer_appendf(&unused_vars_strb, "%s", var_name);
                      ucs_config_parser_append_similar_vars_message(
                                  prefix, var_name, &unused_vars_strb);
                      ucs_string_buffer_appendf(&unused_vars_strb, "; ");
                      ++num_unused_vars;
                  }
              } else {
                  ucs_string_buffer_appendf(&used_vars_strb, "%s ", *envp);
                  ++num_used_vars;
              }

              ucs_free(envstr);
          }
          ```

        - 檢查可用資源會呼叫**ucp_is_resource_enabled**，裡面會呼叫**ucp_is_resource_in_transports_list**來檢查是否在transports list裡面，或是確認是

否為輔助資源。

```
ucp_is_resource_in_transports_list(const char *tl_name,
                                   const ucs_config_allow_list_t *allow_list,
                                   const ucs_string_set_t *aux_tls,
                                   uint8_t *rsc_flags, uint64_t *tl_cfg_mask)
{
    uint8_t search_result;

    if (allow_list->mode == UCS_CONFIG_ALLOW_LIST_ALLOW_ALL) {
        return 1;
    }

    ucs_assert(allow_list->array.count > 0);
    search_result = ucp_transports_list_search(tl_name, &allow_list->array,
                                               tl_cfg_mask);

    if (allow_list->mode == UCS_CONFIG_ALLOW_LIST_ALLOW) {
        /* Enable the transport, if UCX_TLS=tl_name, or alias={tl_name} and
         * UCX_TLS=alias. */
        if (search_result & UCP_TRANSPORTS_LIST_SEARCH_RESULT_PRIMARY) {
            return 1;
        }

        /* Enable the transport as an auxiliary, if UCX_TLS=tl_name:aux, or
         * alias={tl_name} and UCX_TLS=alias:aux, or alias={tl_name:aux} and
         * UCX_TLS=alias. */
        if (search_result & (UCP_TRANSPORTS_LIST_SEARCH_RESULT_AUX_IN_MAIN |
                             UCP_TRANSPORTS_LIST_SEARCH_RESULT_AUX_IN_ALIAS)) {
            *rsc_flags |= UCP_TL_RSC_FLAG_AUX;
            return 1;
        }

        return 0;
    }
```

- TLS selected by UCX
  - **ucp_worker_print_used_tls**會列出endpoint的config，透過數值的config map回對應的config。

```
ucp_worker_print_used_tls(ucp_worker_h worker, ucp_worker_cfg_index_t cfg_index)
```

```
ucp_worker_add_feature_rsc(context, key, tag_lanes_map, "tag", &strb);
ucp_worker_add_feature_rsc(context, key, rma_lanes_map,
                           !rma_emul ? "rma" : "rma_am", &strb);
ucp_worker_add_feature_rsc(context, key, amo_lanes_map,
                           !amo_emul ? "amo" : "amo_am", &strb);
ucp_worker_add_feature_rsc(context, key, am_lanes_map, "am", &strb);
ucp_worker_add_feature_rsc(context, key, stream_lanes_map, "stream", &strb);
ucp_worker_add_feature_rsc(context, key, ka_lanes_map, "ka", &strb);
```

2. Implementation

> Please complete the implementation according to the spec
> Describe how you implemented the two special features of HW5.

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

- 修改ucp_worker.c
  - print Line 1和Line 2都是在**ucp_worker_print_used_tls**最底下的地方，可以直接用包好的變數。
    - ucp_config_print()用來印出Line 1，printf()直接印出Line 2。

```
ucp_config_print(NULL, stdout, NULL, UCS_CONFIG_PRINT_TLS);
printf("%s\n", ucs_string_buffer_cstr(&strb));
```

- 修改parser.c
  - Line 1的參數處理是寫在**ucs_config_parser_print_opts**的判斷式裡
    - 當**ucp_config_print**傳入**UCS_CONFIG_PRINT_TLS**這個專門用來印TLS參數的flag就必須要把Line 1該印出來的參數整理好。至於參數的整理是參考同一份檔案底下的**ucs_config_parser_print_env_vars**。

```
if (flags & UCS_CONFIG_PRINT_TLS) {
    int num_used_vars;
    char* match_char = "UCX_TLS";
    ucs_string_buffer_t used_vars_strb;
    char **envp, *envstr;
    // size_t prefix_len;
    char *var_name;
    khiter_t iter;
    char *saveptr;
    // prefix_len      = strlen(opts);
    num_used_vars   = 0;
    ucs_string_buffer_init(&used_vars_strb);
    pthread_mutex_lock(&ucs_config_parser_env_vars_hash_lock);
    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, opts, prefix_len)) {
        //     ucs_free(envstr);
        //     continue; /* Not UCX */
        // }

        iter = kh_get(ucs_config_env_vars, &ucs_config_parser_env_vars, var_name);
        if (iter != kh_end(&ucs_config_parser_env_vars) && !strncmp(*envp, match_char, strlen(match_char))) {
            ucs_string_buffer_appendf(&used_vars_strb, "%s ", *envp);
            ++num_used_vars;
        }

        ucs_free(envstr);
    }
    pthread_mutex_unlock(&ucs_config_parser_env_vars_hash_lock);
    if (num_used_vars > 0) {
        ucs_string_buffer_rtrim(&used_vars_strb, " ");
        printf("%s\n", ucs_string_buffer_cstr(&used_vars_strb));
    }
    ucs_string_buffer_cleanup(&used_vars_strb);
}
```

- 修改type.h
  - 由於**UCS_CONFIG_PRINT_TLS** flag是額外添加的，原本並沒有map到一個數值，所以需要在type.h裡寫出來。

- 這裡把flag對到UCS_BIT(5)的位置

```
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;
```

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

- 主要是worker在create endpoint時會呼叫**ucp_ep_create**，接著呼叫**ucp_ep_create_to_sock_addr**來create目標的socket address，其中會呼叫**ucp_ep_init_create_wireup**來協商endpoint之間的通訊，裡面會呼叫**ucp_worker_get_ep_config**來取得全域endpoints的共同config，最底下呼叫了**ucp_worker_print_used_tls**用來log出目前endpoint所使用的TLS
  - 而我增加print Line 1和Line 2的程式就是在**ucp_worker_print_used_tls**被執行到的。其中的**ucp_config_print**又會呼叫**ucs_config_parser_print_opts**執行相對應flag該做的操作。
  - 會這樣設計是因為在create出endpoint並決定好使用的TLS後會需要把這個決定好的結果print出來，所以是把這些要印出來的資料寫在**ucp_worker_print_used_tls**裡面。Line 2需要印出決定的TLS結果，剛好在**ucp_worker_print_used_tls**裡面有包好對endpoint的TLS。Line 1需要印出來的是UCX_TLS，也就是指定可以使用的TLS。而這些資訊都被寫在環境變數裡，需要把他們parse出來，所以就把print Line 1的程式寫在parser.c裡面。再加上如果是想增加額外功能而不動到原本程式的架構，所以多增加了一個**UCS_CONFIG_PRINT_TLS** flag專門印出Line 1。**ucp_config_print**中的**ucs_config_parser_print_opts**能針對傳入的flag做不同操作，所以就把新加的功能寫在裡面。

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

- 當程式呼叫**ucp_ep_create**時會呼叫**ucp_ep_create**，接著呼叫**ucp_ep_create_to_sock_addr**來create目標的socket address，其中會呼叫**ucp_ep_init_create_wireup**來協商endpoint之間的通訊，裡面會呼叫**ucp_worker_get_ep_config**來取得全域endpoints的共同config，最底下呼叫了**ucp_worker_print_used_tls**用來log出目前endpoint所使用的TLS，最後才會呼叫寫在最下面用來print Line 1和Line 2的**ucp_config_print**和**printf**。

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

- 大致上是相符的。由於選擇適合的TLS來傳輸本來就是在endpoint之間做協調，只是沒想過是在create的當下就順便決定了，原本以為會需要額外呼叫別的function來建立連線和選擇適合的TLS。

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`
  - 用來存通訊用的通道
  - lane會連接iface和endpoint，也會需要和目標endpoint連接

```
ucp_wireup_connect_lane_to_iface(ucp_ep_h ep, ucp_lane_index_t lane,
                                 unsigned path_index,
                                 ucp_worker_iface_t *wiface,
                                 const ucp_address_entry_t *address)
```

```
ucp_wireup_connect_lane_to_ep(ucp_ep_h ep, unsigned ep_init_flags,
                              ucp_lane_index_t lane, unsigned path_index,
                              ucp_rsc_index_t rsc_index,
                              ucp_worker_iface_t *wiface,
                              const ucp_unpacked_address_t *remote_address)
```

```
ucs_status_t ucp_wireup_connect_remote(ucp_ep_h ep, ucp_lane_index_t lane)
```

- `tl_rsc`
  - 用來儲存transport layer的資源。
  - 本身的型態是**uct_tl_resource_desc_t**，是一種resource descriptor。
  - 裡面存了關於transport和device的資訊。

```
typedef struct uct_tl_resource_desc {
    char                  tl_name[UCT_TL_NAME_MAX];   /**< Transport name */
    char                  dev_name[UCT_DEVICE_NAME_MAX]; /**< Hardware device name */
    uct_device_type_t     dev_type;     /**< The device represented by this resource
                                             (e.g. UCT_DEVICE_TYPE_NET for a network interface) */
    ucs_sys_device_t      sys_device;   /**< The identifier associated with the device
                                             bus_id as captured in ucs_sys_bus_id_t struct */
} uct_tl_resource_desc_t;
```

  - **ucp_tl_resource_is_same_device**可以比較兩個傳入的resource是否是同一個硬體資源。

```
(ucp_tl_resource_is_same_device(&context->tl_rscs[i].tl_rsc, resource)
```

- `tl_name`
  - 用來儲存transport layer的實際名稱。
  - 型態是array of char。
  - 是**uct_tl_resource_desc_t**的成員。
  - 比對資源時，通常會使用字串直接比對。

```
search_result = ucp_transports_list_search(tl_name, &allow_list->array,
                                           tl_cfg_mask);
```

- `tl_device`

- 用來儲存transport layer的硬體device。
- 型態是**uct_tl_device_resource_t**，是一種resource descriptor。

```
typedef struct uct_tl_device_resource {
    char                    name[UCT_DEVICE_NAME_MAX]; /**< Hardware device name */
    uct_device_type_t       type;       /**< The device represented by this resource
                                             (e.g. UCT_DEVICE_TYPE_NET for a network interface) */
    ucs_sys_device_t        sys_device; /**< The identifier associated with the device
                                             bus_id as captured in ucs_sys_bus_id_t struct */
} uct_tl_device_resource_t;
```

- 通常不會直接傳tl_device，而是傳**uct_tl_device_resource**。

- `tl_bitmap`
  - 用來儲存可用的resource或iface。
  - resource上限是128種(128 bit的bit string)。
  - **ucp_worker_add_resource_ifaces**會去查看目前context有沒有cache最佳的資源和iface，如果有就可以直接使用，如果沒有就呼叫**ucp_worker_select_best_ifaces**來尋找最佳的iface。
  - 決定好的設定會被寫進tl_bitmap，其他worker就可以直接沿用，就不需要再呼叫**ucp_worker_select_best_ifaces**尋找最好的iface。

```
static ucs_status_t ucp_worker_add_resource_ifaces(ucp_worker_h worker)
{
    ucp_context_h context = worker->context;
    ucp_tl_resource_desc_t *resource;
    uct_iface_params_t iface_params;
    ucp_rsc_index_t tl_id, iface_id;
    ucp_worker_iface_t *wiface;
    ucp_tl_bitmap_t ctx_tl_bitmap, tl_bitmap;
    unsigned num_ifaces;
    ucs_status_t status;

    /* If tl_bitmap is already set, just use it. Otherwise open ifaces on all
     * available resources and then select the best ones. */
    ctx_tl_bitmap  = context->tl_bitmap;
    if (!UCS_BITMAP_IS_ZERO_INPLACE(&ctx_tl_bitmap)) {
        num_ifaces = UCS_BITMAP_POPCOUNT(ctx_tl_bitmap);
        tl_bitmap  = ctx_tl_bitmap;
    } else {
        num_ifaces = context->num_tls;
        UCS_BITMAP_MASK(&tl_bitmap, context->num_tls);
    }

    worker->ifaces = ucs_calloc(num_ifaces, sizeof(*worker->ifaces),
                                "ucp ifaces array");
    if (worker->ifaces == NULL) {
        ucs_error("failed to allocate worker ifaces");
        status = UCS_ERR_NO_MEMORY;
        goto err;
    }

    worker->num_ifaces = num_ifaces;
    iface_id           = 0;
```

```
if (UCS_BITMAP_IS_ZERO_INPLACE(&ctx_tl_bitmap)) {
    /* Context bitmap is not set, need to select the best tl resources */
    UCS_BITMAP_CLEAR(&tl_bitmap);
    ucp_worker_select_best_ifaces(worker, &tl_bitmap);
    ucs_assert(!UCS_BITMAP_IS_ZERO_INPLACE(&tl_bitmap));

    /* Cache tl_bitmap on the context, so the next workers would not need
     * to select best ifaces. */
    context->tl_bitmap = tl_bitmap;
    ucs_debug("selected tl bitmap: " UCT_TL_BITMAP_FMT "(%zu tls)",
              UCT_TL_BITMAP_ARG(&tl_bitmap),
              UCS_BITMAP_POPCOUNT(tl_bitmap));
}
```

- `iface`
  - 用來儲存transport layer的communication interface context。

- iface的type是**uct_iface_t**，主要的資訊都塞在**uct_iface_ops_t**裡。

```
typedef struct uct_iface {
    uct_iface_ops_t           ops;
} uct_iface_t;
```

```
    /* interface - synchronization */
    uct_iface_flush_func_t              iface_flush;
    uct_iface_fence_func_t              iface_fence;

    /* interface - progress control */
    uct_iface_progress_enable_func_t    iface_progress_enable;
    uct_iface_progress_disable_func_t   iface_progress_disable;
    uct_iface_progress_func_t           iface_progress;

    /* interface - events */
    uct_iface_event_fd_get_func_t       iface_event_fd_get;
    uct_iface_event_arm_func_t          iface_event_arm;

    /* interface - management */
    uct_iface_close_func_t              iface_close;
    uct_iface_query_func_t              iface_query;

    /* interface - connection establishment */
    uct_iface_get_device_address_func_t iface_get_device_address;
    uct_iface_get_address_func_t        iface_get_address;
    uct_iface_is_reachable_func_t       iface_is_reachable;

} uct_iface_ops_t;
```

- **ucp_worker_select_best_ifaces**會決定最好的iface給worker。

```
ucp_worker_select_best_ifaces(ucp_worker_h worker, ucp_tl_bitmap_t *tl_bitmap_p)
{
    ucp_context_h context     = worker->context;
    ucp_tl_bitmap_t tl_bitmap = UCS_BITMAP_ZERO;
    ucp_rsc_index_t repl_ifaces[UCP_MAX_RESOURCES];
    ucp_worker_iface_t *wiface;
    ucp_rsc_index_t tl_id, iface_id;

    /* For each iface check whether there is another iface, which:
     * 1. Supports at least the same capabilities
     * 2. Provides equivalent or better performance
     */
    for (tl_id = 0; tl_id < context->num_tls; ++tl_id) {
        wiface = worker->ifaces[tl_id];
        if (!ucp_worker_iface_find_better(worker, wiface, &repl_ifaces[tl_id])) {
            UCS_BITMAP_SET(tl_bitmap, tl_id);
        }
    }

    *tl_bitmap_p       = tl_bitmap;
    worker->num_ifaces = UCS_BITMAP_POPCOUNT(tl_bitmap);
    ucs_assert(worker->num_ifaces <= context->num_tls);
```

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

1. 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
```

- process內通訊一定是用process內部直接的memory access來傳輸會最快，所以UCX選擇了self/memory。而輔助的通訊是透過cma/memory。
- 單一節點內不同process之間通訊，UCX則是選擇了sysv/memory，也就是使用System V的share memory機制作為傳輸方法。而輔助的通訊是透過cma/memory。

```
[pp24s086@apollo31 UCX-lsalab]$ mpiucx -n 2 -x UCX_TLS=all  $HOME/UCX-lsalab/test/mpi/osu/pt2pt/osu_latency
UCX_TLS=all
0x556ebd518470 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x563e181243c0 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x563e181243c0 intra-node cfg#1 tag(sysv/memory cma/memory)
UCX_TLS=all
0x556ebd518470 intra-node cfg#1 tag(sysv/memory cma/memory)
# OSU MPI Latency Test v5.3
# Size          Latency (us)
0                       0.20
1                       0.20
2                       0.20
4                       0.20
8                       0.20
16                      0.20
32                      0.24
64                      0.24
128                     0.36
256                     0.38
512                     0.42
1024                    0.50
2048                    0.66
4096                    0.98
8192                    1.93
16384                   3.01
32768                   5.00
65536                   8.68
131072                 17.66
262144                 37.89
524288                 65.77
1048576               132.96
2097152               334.99
4194304               955.57
```

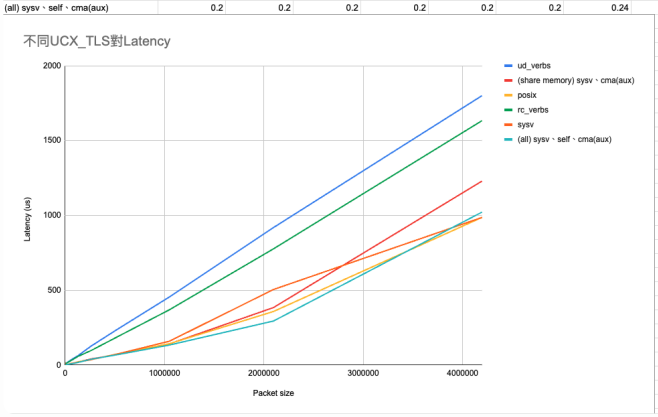
```
[pp24s086@apollo31 UCX-lsalab]$ mpiucx -n 2 -x UCX_TLS=all  $HOME/UCX-lsalab/test/mpi/osu/pt2pt/osu_bw
UCX_TLS=all
0x55a8f7294470 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x55ba62aa03c0 self cfg#0 tag(self/memory cma/memory)
UCX_TLS=all
0x55a8f7294470 intra-node cfg#1 tag(sysv/memory cma/memory)
UCX_TLS=all
0x55ba62aa03c0 intra-node cfg#1 tag(sysv/memory cma/memory)
# OSU MPI Bandwidth Test v5.3
# Size      Bandwidth (MB/s)
1                     10.04
2                     20.32
4                     40.82
8                     82.01
16                   134.92
32                   321.84
64                   648.49
128                  642.83
256                 1219.61
512                 2036.75
1024                3285.43
2048                5780.32
4096                8271.85
8192               10502.24
16384               5063.28
32768               6853.41
65536               8554.76
131072              9293.93
262144              7710.10
524288              8392.90
1048576             8047.68
2097152             7836.19
4194304             6906.06
```

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

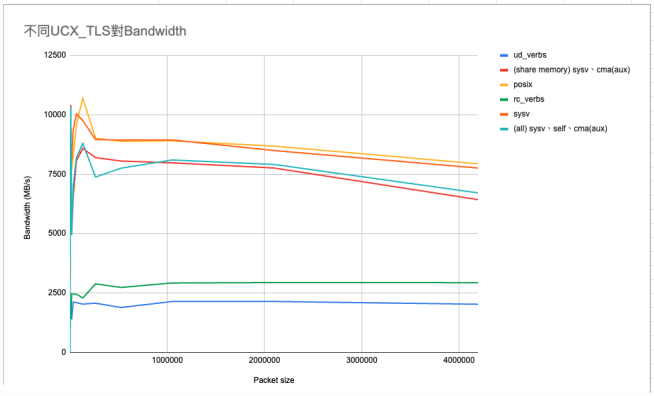
下圖針對5種UCX_TLS對Latency和Bandwidth作圖。

- 圖中因為share memory(UCX_TLS=shm)有大於一種方法，所以會有cma作為輔助方法。
- 如果不限制方法(UCX_TLS=all)，UCX會在process內選擇memory通訊，intra-node選擇sysv通訊。
- 圖中也能看出UCX選擇的不一定是對於所有大小封包的最佳組合，但也是latency比較低，bandwidth比較高的選擇。

Packet size	0	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
ud_verbs	1.71	1.96	1.66	1.55	1.56	1.63	1.66	2	1.85	3.08	3.44	4.4	5.65	11.15	11.71	14.47
(share memory) sysv、cma(aux)	0.2	0.2	0.2	0.19	0.2	0.19	0.23	0.23	0.34	0.37	0.43	0.5	0.69	1.06	1.66	2.93
posix	0.21	0.2	0.21	0.2	0.2	0.2	0.24	0.25	0.36	0.37	0.41	0.49	0.68	1.39	1.74	3.07
rc_verbs	1.47	1.44	1.69	1.48	1.47	1.47	1.72	1.61	3.03	2.96	3.56	4.14	6.31	9.46	12.21	13.54
sysv	0.2	0.19	0.2	0.19	0.2	0.19	0.24	0.24	0.38	0.37	0.4	0.49	0.65	0.95	1.88	3.43
(all) sysv、self、cma(aux)	0.2	0.2	0.2	0.2	0.2	0.2	0.24	0.24	0.36	0.38	0.42	0.51	0.66	0.99	1.69	3.04



Packet size	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768
ud_verbs	1.5	3.11	6.23	13.42	26.57	53.22	10.48	181.19	290.73	561.9	993.55	1530.56	1590.07	1863.63	1381.67	2115.63
(share memory) sysv、cma(aux)	9.44	20.44	39.96	81.95	165.26	303.45	639.33	637.19	1237.48	2372.51	3638.15	5798.3	8022.22	10415.75	4938.88	6563.06
posix	9.29	18.47	37.26	74.19	147.53	289.14	574.07	496.81	958.91	1837.03	2964.85	5327.7	6391.74	8276.95	7677.36	8250.73
rc_verbs	1.46	2.98	5.96	5.59	27.41	55.33	106.36	174.16	312.8	584.51	990.45	1617.6	2085.09	1432.29	2455.03	2447.02
sysv	9.24	15.25	31.36	63.87	126.95	252.5	504.86	612.48	1202.21	2243.82	3102.62	4817.4	7520.23	8932.81	8081.92	9387.21
(all) sysv、self、cma(aux)	7.77	15.94	31.95	64.27	156.25	302.54	505.58	535.51	1057.19	2292.92	3837.06	5832	8218.56	10299.71	4953.45	7042.44



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

由於rc_verbs和ud_verbs不是使用share memory比較適用於多節點之間的通訊，所以latency比較大，bandwidth比較小就不討論了。

- POSIX
  - posix在沒有資源競爭時不需要用system call進入kernel。相較之下，sysv不論有沒有資源競爭都會呼叫system call，因此表現就會比posix差一點點。
  - Reference
- SysV

    - sysv需要自行維護inter-process communication的key-value pair。如果process沒有主動釋放資源，kernel裡就會持續佔用這些資源，並沒有garbage collection機制。
    - posix訪問和管理inter-process communication就如同訪問file system一樣方便。process結束後也會有garbage collection機制清理memory。
    - Reference
- cma
    - 節點內通訊時，一個process可以透過system call，直接把資料寫進另一個process的memory裡，只是因為要進kernel，所以會花比較多時間。
    - Reference

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
```

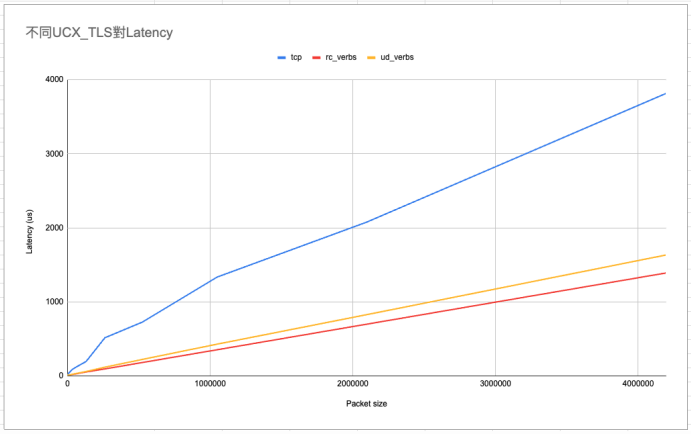
- 在多節點間通訊，UCX會選擇使用rc_verbs作為節點間的溝通方法。
- 因為是跨節點的溝通，所以輔助的通訊方法就不會有用share memory或是呼叫system call的機會，只剩下verbs和tcp。
- 因此跨節點的溝通會採用verbs的方法，讓endpoint先跟destination worker要資源做綁定，建立好的endpoint就可以不用透過OS，直接和對方的endpoint做溝通。

```
/home/pp24/pp24s086/ucx-pp/lib/libucm.so.0:/home/pp24/pp24s086/ucx-pp/li
UCX_TLS=all
0x5601946921f0 self cfg#0 tag(self/memory cma/memory rc_verbs/ibp3s0:1)
# OSU MPI Latency Test v5.3
# Size          Latency (us)
UCX_TLS=all
0x5601946921f0 inter-node cfg#1 tag(rc_verbs/ibp3s0:1 tcp/ibp3s0)
0                       1.86
1                       1.84
2                       1.83
4                       1.83
8                       1.81
16                      1.81
32                      1.83
64                      1.98
128                     3.14
256                     3.31
512                     3.58
1024                    4.13
2048                    5.20
4096                    7.37
8192                    9.41
16384                  12.78
32768                  18.52
65536                  29.90
131072                 52.70
262144                 94.74
524288                180.49
1048576               353.30
2097152               699.01
4194304              1391.13
UCX_TLS=all
0x55eba17aa3c0 self cfg#0 tag(self/memory cma/memory rc_verbs/ibp3s0:1)
UCX_TLS=all
0x55eba17aa3c0 inter-node cfg#1 tag(rc_verbs/ibp3s0:1 tcp/ibp3s0)
```

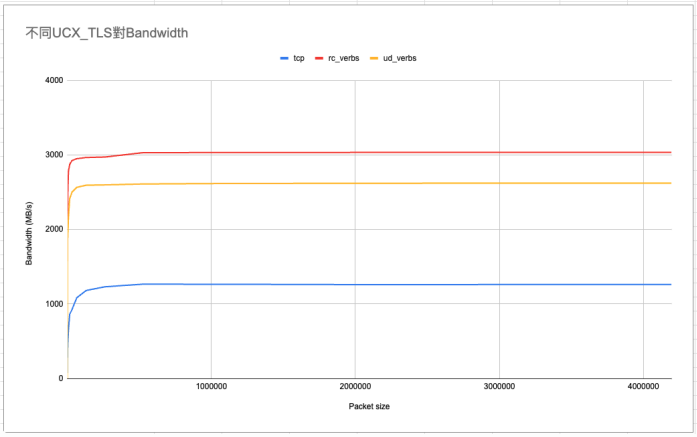
```
/home/pp24/pp24s086/ucx-pp/lib/libucm.so.0:/home/pp24/pp24s086/ucx-pp/li
UCX_TLS=all
0x55d2e85720e0 self cfg#0 tag(self/memory cma/memory rc_verbs/ibp3s0:1)
# OSU MPI Bandwidth Test v5.3
# Size      Bandwidth (MB/s)
UCX_TLS=all
0x55d2e85720e0 inter-node cfg#1 tag(rc_verbs/ibp3s0:1 tcp/ibp3s0)
1                       3.02
2                       6.21
4                      12.38
8                      24.89
16                     49.73
32                     97.48
64                    183.18
128                   259.15
256                   495.57
512                   930.01
1024                 1519.79
2048                 2178.69
4096                 2525.60
8192                 2766.09
16384                2862.89
32768                2922.48
65536                2950.19
131072               2964.90
262144               2972.57
524288               3031.19
1048576              3033.71
2097152              3035.17
4194304              3035.82
UCX_TLS=all
0x5587654303f0 self cfg#0 tag(self/memory cma/memory rc_verbs/ibp3s0:1)
UCX_TLS=all
0x5587654303f0 inter-node cfg#1 tag(rc_verbs/ibp3s0:1 tcp/ibp3s0)
```

- 實驗的部分是針對三種跨節點通訊的方法(tcp, rc_verbs, ud_verbs)對Latency和bandwidth作圖。
- 實驗結果發現verbs的表現比tcp好很多，主要原因是verbs在建立完連線後，可以直接bypass OS，讓endpoints直接做溝通。
- rc_verbs表現又比ud_verbs好一點，原因是rc_verbs提供高可靠性的連線，ud_verbs則不會針對每個傳輸對象建立連線。ud_verbs傳送的是datagram，會根據IP去走該走的路由，而不會建立專屬通道，因此比較耗時。
- 圖中也可以看出當packet size較小時，packet size若變成2倍，bandwidth也會變成接近兩倍。隨著packet size逐漸變大，就能明顯看出每一種通訊協定收斂到的極限值。
- Reference

Packet size	0	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
tcp	28.67	11.89	11.95	11.83	11.89	11.88	11.9	12.02	12.08	12.21	12.57	13.24	17.71	20.48	36.58	52.16
rc_verbs	1.64	1.62	1.62	1.6	1.61	1.61	1.62	1.75	2.91	3.06	3.33	3.9	4.97	7.12	9.15	12.53
ud_verbs	2.42	1.67	1.68	1.67	1.67	1.8	1.82	1.89	1.98	3.16	3.42	3.99	5.2	7.7	9.78	13.31



Packet size	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768
tcp	0.22	0.34	0.62	1.48	3.37	7.01	13.43	26.93	51.47	103.27	198.2	365.48	504.23	655.82	862.12	925.85
rc_verbs	3.15	6.6	13.14	26.34	52.82	105.51	193.99	268.01	520.03	1009.95	1587.34	2216.02	2549.36	2783.97	2873.43	2925.75
ud_verbs	2.89	5.88	11.84	23.64	43.57	88.88	140.57	308.88	438.43	790.13	1300.69	1846.83	1976.59	2169.1	2414.52	2501.06



4. Experience & Conclusion

1. What have you learned from this homework?

- 從這次的作業中我學到了，一個unified communication是如何在呼叫相同API call的前提下，在底層根據通訊環境和使用情境幫忙評估適合使用的通訊協定。同時也透過trace code驗證老師上課所講到的UCX和verbs架構，context、worker和endpoint之間的關係。也透過觀察debug的輸出行為知道有哪些UCX_TLS是會被加入list裡面，而哪些又因為不適合而被剔除，剩下的就變成輔助用的協定。

2. How long did you spend on the assignment?

- 大約先花2小時在trace code，然後又回去看了老師的UCX介紹影片複習連線建立流程。之後再重新檢查自己的理解是不是對的。然後又花了大約4小時trace名稱看起來比較重要的function，跟他們是如何被呼叫到的。也有搭配-x UCX_LOG_LEVEL=debug來查看function call的呼叫流程。

3. Feedback (optional)

- 感覺可以把report和demo的評分標準講清楚。不然常常會花很多時間做無關緊要的是，結果分數不如預期。或是demo把該講的都講了，結果評分都在baseline以下。

- 感覺可以把report和demo的評分完全分離。不然如果因為沒做很特別的優化，就沒有特別寫在報告裡，demo時也把所有做的事都在implementation的階段都說完了，所以被問到還有沒有做額外的優化也只能說沒有。結果report和demo都有優化說明的評分，所以兩個都拿了0分。