PP HW 4 Report

> - 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.
> - Note some variable have premalink and some have note. They are in same color.

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.

- `ucp_context`
  - `ucp_config_read`: Reads configuration settings.
  - `ucp_init`: This routine creates and initializes a UCP application context. And must be called before any other UCP function call in the application. In detail, this routine checks API version compatibility then discovers the available network interfaces and initializes the network resources required for discovering the network and memory-related devices. This routine is responsible for the initialization of all information required for a particular application scope, for example, MPI application, OpenSHMEM application, etc.
  - `ucp_config_print`: Prints the configuration settings.
  - `ucp_cleanup`: The cleanup process releases and shuts down all resources associated with the application context.
- `ucp_worker`
  - `ucp_worker_create`: This routine allocates and initializes a worker object. Each worker is associated with one and only one application context. At the same time, an application context can create multiple workers to enable concurrent access to communication resources. For example, the application can allocate a dedicated worker for each application thread, where every worker can progress independently of others.
  - `ucp_worker_get_address`: This routine returns the address of the worker object. This address can be passed to remote instances of the UCP library to connect to this worker.
  - `ucp_worker_arm`: Turn on event notification for the next event. This routine needs to be called before waiting on each notification on this worker, so will typically be called once the processing of the previous event is over, as part of the wake-up mechanism. The worker must be armed before waiting on an event (must be re-armed after it has been signaled for re-use) with `ucp_worker_arm`.
  - `ucp_worker_progress`: Progress all communications on a specific worker. This routine explicitly progresses all communication operations on a worker.
  - `ucp_worker_destroy`: This routine releases the resources associated with a UCP worker.
- `ucp_ep`
  - `ucp_ep_create`: This routine creates and connects an endpoint on a local worker for a destination address that identifies the remote worker. This function is non-blocking, and communications may begin immediately after it returns. If the connection process is not completed, communications may be delayed. The created endpoint is associated with one and only one worker.

  - `ucp_ep_create_connected`: Creates a connected endpoint.
  - `ucp_ep_connect`: Connects the endpoint to a remote peer.
  - `ucp_ep_destroy`: Destroys the endpoint.
  - `ucp_ep_query`: Queries information about the endpoint.

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

- `ucp_context`
  - Significance:
    - UCP application context is an opaque handle that holds a UCP communication instance's global information. It represents a single UCP communication instance.
    - It encapsulates configuration settings and resources that are shared across multiple communication workers. And is responsible for managing global communication parameters and options.
  - Important information:
    - Communication resources `*tl_rscs`
    - Memory `*tl_mds`
    - Config environment prefix used to create the context `*env_prefix`
    - How many endpoints are expected to be created `est_num_eps`

- `ucp_worker`
  - Significance:
    - UCP worker is an opaque object representing the communication context. The worker represents an instance of a local communication resource and the progress engine associated with it.
    - The progress engine is a construct that is responsible for asynchronous and independent progress of communication directives. The progress engine could be implemented in hardware or software.
    - The worker object abstracts an instance of network resources such as a host channel adapter port, network interface, or multiple resources such as multiple network interfaces or communication ports.
    - Although the worker can represent multiple network resources, it is associated with a single UCX application context. All communication functions require a context to perform the operation on the dedicated hardware resource(s) and an endpoint to address the destination.
  - Important information:
    - List of all endpoints (except internal endpoints) `all_eps`
    - Array of pointers to interfaces, one for each resource `**ifaces`
    - Array of Connection managers, one for each component `*cms`
    - Number of requests to create or close endpoint `counter`

- `ucp_ep`
  - Significant:
    - All UCP communication routines address a destination with the endpoint handle.

        - The endpoint handle is associated with only one `ucp_context_h`. UCP provides the `ucp_ep_create` routine to create the endpoint handle and the `ucp_ep_destroy` routine to destroy the endpoint handle.
    - Important attributes:
        - Communication source
        - Destination
        - Endpoint handle.

The reason why instead of a big ucp struct to include all attributes, the developer chooses to split responsibility into three layers of objects is that they have a level of dependency. For example, an application context can create multiple workers to enable concurrent access to communication resources, while each worker can only be owned by an application context.

3. Based on the description in HW4, where do you think the following information is loaded/created?
    - `UCX_TLS`
        - UCX_TLS is a global variable that we can get from `extern char **environ`. It represents a comma-separated list of transports to use. In parser.c we can print it by iterating each global variable.
    - TLS selected by UCX
        - UCX_TLS is information defined by the user. This config is defined in here. In `ucp_fill_resources()` under `ucp_init()` the config will config will be load into context. In specific `tl_bitmap` and `num_tls` variable. And when the worker is created, all resources will be open as an interface in `ucp_worker_add_resource_ifaces()` according to bitmap. And the best tls will be selected by `ucp_worker_select_best_ifaces()`.

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? There are 3 files I have modified to finish homework.

- parser.c (In `ucs_config_parser_print_opts`)

```
if (flags & UCS_CONFIG_PRINT_TLS) {
    char **envp, *envstr;
    char *var_name;
    char *saveptr;

    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 (strncmp("UCX_TLS", var_name, 7) == 0)
            printf("%s\n", *envp);

        ucs_free(envstr);
    }

    pthread_mutex_unlock(&ucs_config_parser_env_vars_hash_lock);
}
```

- types.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),
    UCS_CONFIG_PRINT_TLS = UCS_BIT(5)
} ucs_config_print_flags_t;
```

- ucp_worker.c (In `ucp_worker_print_used_tls`)

```
    ucs_string_buffer_rtrim(&strb, "; ");
    // Here I print two lines
    ucp_config_print(NULL, stdout, NULL, UCS_CONFIG_PRINT_TLS);
    printf("%s\n", ucs_string_buffer_cstr(&strb));
    ucs_info("%s", ucs_string_buffer_cstr(&strb));
}
```

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

- parser.c is a generic file that can print various attributes and logs of the UCX. Function `ucs_config_parser_print_opts` is a sub-function of `ucp_config_print` and is responsible for printing the options names, values, and documentation by designated flags. And in the `ucp_worker.c`, `ucp_worker_print_used_tls` is a function that will be called to output the tls configuration to the log file, and will be called when `ucp_worker_get_ep_config` is triggered to load configuration. So I choose to put `ucp_config_print` with a new flag option to print the UCX_TLS variable inside `ucp_worker_print_used_tls`, right before printing the output of the tls configuration.

3. Observe when Lines 1 and 2 are printed during the call of which UCP API? It will be printed during the call `ucp_worker_get_ep_config` to get the config that is shared among all ucp endpoints. And usually `ucp_worker_get_ep_config` is called by `ucp_wireup_init_lanes` when init new lanes.

4. Does it match your expectations for questions **1-3**? Why? Yes, all of the UCX transport protocols actively utilized by the program have been outputted.

5. In implementing the features, we see variables like lanes, tl_rsc, tl_name, tl_device, bitmap, iface, etc., used to store different Layer protocol information. Please explain what information each of them stores.

- `lanes`: Lanes are the independent communication channels or paths. Defined in `ucp_ep_config_key`. It may be associated with parallelism or concurrency in data transmission. And each lanes may use one protocol selected by UCX.
- `tl_rsc`: "tl_rsc" might be "transport layer resource". It is defined in `ucp_context`. It could store information related to the communication resources allocated or associated with a specific transport layer.
- `tl_name`: "tl_name" likely stands for "transport layer name." Defined in `uct_tl_resource_desc`. It could store the name or identifier of a stand-alone communication resource such as an HCA port, network interface, or multiple resources such as multiple network interfaces or communication ports.
- `tl_device`: I don't find any variable name that is `tl_device`. The nearest type I find is `uct_tl_device_resource`. This struct is an internal resource descriptor of a transport hardware device.
- `bitmap`: A "bitmap" is a data structure that represents a set of bits, where each bit typically corresponds to a specific attribute or state. It could be used to store information about the capabilities, features, or availability of certain resources. For example, the variable `tl_bitmap` is used to store available communication transpose layer resources.
- `iface`: "iface" is defined in `uct_ep`. It provides a handful of transport interface operations. Every operation exposed in the API must appear in this API to allow the creation interface/endpoint with custom operations.

3. Optimize System

1. Below are the system's current configurations for OpenMPI and UCX. Based on your learning, what methods can you use to optimize single-node performance by setting UCX environment variables?

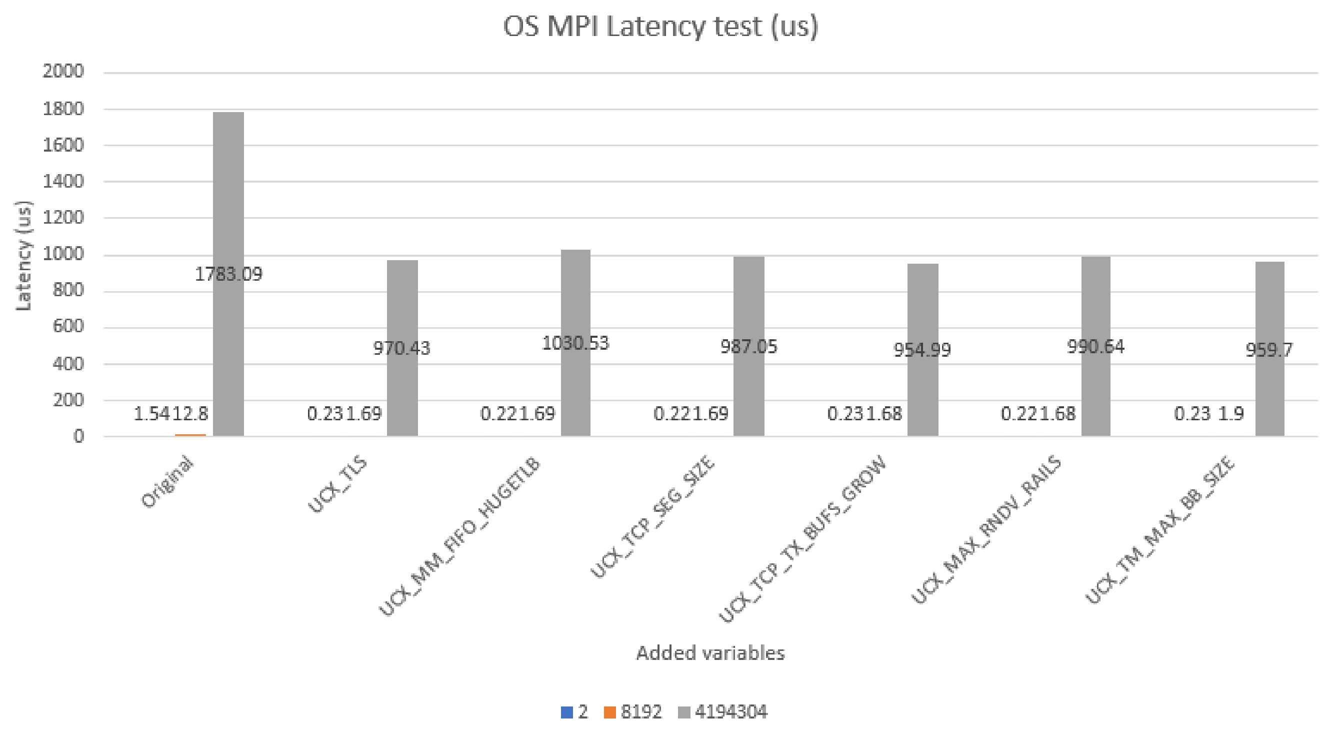
```
-------------------------------------------------------------------
/opt/modulefiles/openmpi/4.1.5:

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

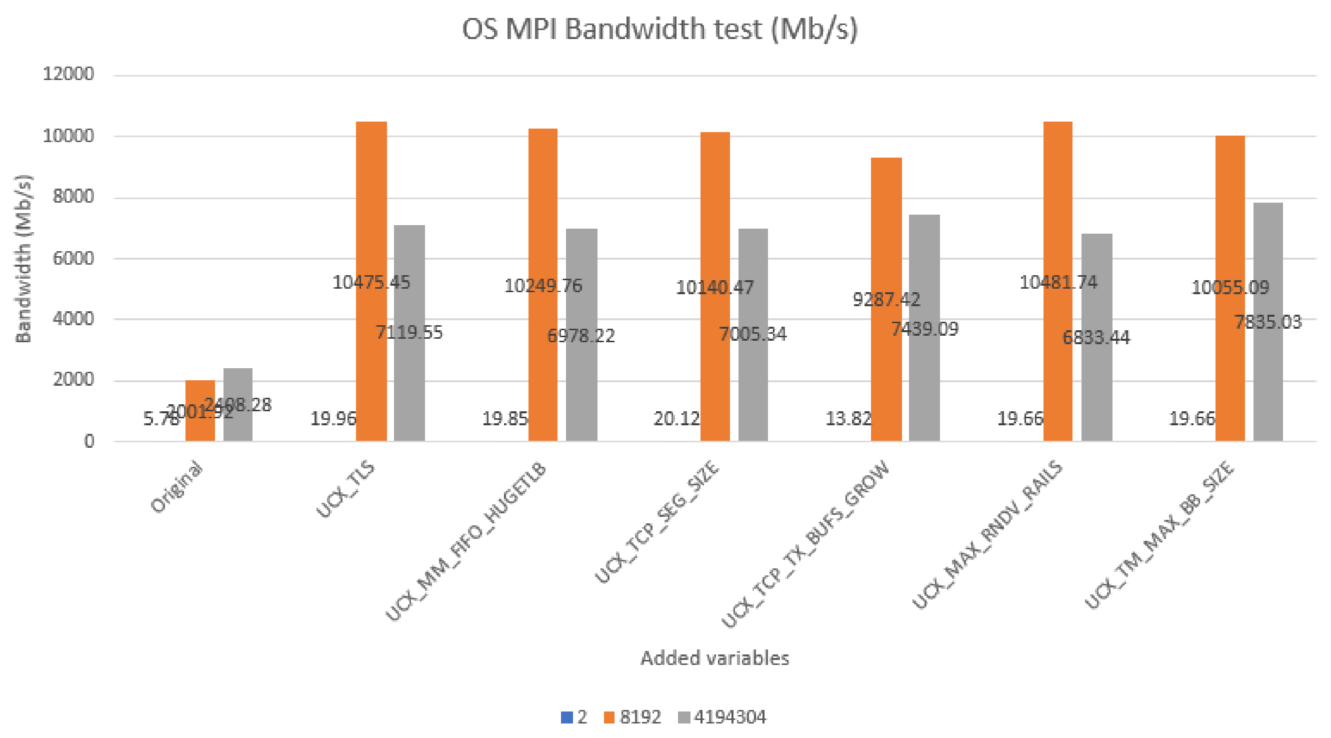
> Please use the following commands to test different data sizes for latency and bandwidth, to verify your ideas:

```
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
```

Note: All available environment variables is in here.

Result (Note three number of MPI_CHAR have been used to test):





Configurations experimented (The environment variable changes are cumulative):

- `UCX_TLS=all`: Changing allow transportation method from ud_verbs to all greatly increases communicate efficiency.
- `UCX_MM_FIFO_HUGETLB=y`: Enable using huge pages for internal shared memory buffers. Originally I think it can decrease latency by increasing hit rate. But maybe the data is too small, so the benefit of

the hit rate is not as great as the overhead to create a bigger TLB, so the performance has decreased.
- `UCX_TCP_SEG_SIZE=64K`: Increase the size of a copy-out buffer from 8K to 64K. The result is just about the same as the previous one.
- `UCX_TCP_TX_BUFS_GROW=0`: How many buffers are added every time the send memory pool grows? 0 means the value is chosen by the transport. I changed the value from 8 to 0. The result shows a slight improvement, I think it is because buffer growth times have decreased.
- `UCX_MAX_RNDV_RAILS=4`: Maximal number of devices on which a rendezvous operation may be executed in parallel. I changed the value from 2 to 4. But the result got worse, maybe four RNDV rails are just too many for this size of data.
- `UCX_TM_MAX_BB_SIZE=16k`: Maximal size for posting "bounce buffer" (UCX internal preregistered memory) for tag offload receives. When a message arrives, it is copied into the user buffer (similar to the eager protocol). The size values have to be equal to or less than the segment size. Also, the value has to be bigger than UCX_TM_THRESH to take effect.

4. Experience & Conclusion

1. What have you learned from this homework? I have learned about how to trace big open source project and the underlying implementation of the UCX.