# CS542000 Parallel Programming HW4 Report

- 學號：108022138
- 姓名：楊宗諺

> - 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`

      The primary purpose of ucp_init() is to initialize the UCP context object. During the execution of ucp_init() function, it calls the ucp_init_version() function to first confirm the compatibility of the API version. Subsequently, it handles memory allocation for the UCP context object and initializes various relevant parameters of the object such as global unique identifier (uuid), memory registration cache (rcache), vfs, and others.

    - `ucp_worker_create`

      After invoking the ucp_init() function to create and initialize the UCP context object, the resulting object is stored as a pointer and subsequently passed as a parameter to the ucp_worker_create() function, along with the initialized worker parameters, including field mask and thread mode.

      During the execution of the ucp_worker_create() function, its main responsibilities include memory allocation for the worker object and the initialization of other worker parameters. It can also be observed here that the created worker object utilizes the passed UCP context object pointer to establish a connection with the associated UCP context.

    - `ucp_ep_create`

      Once the Server and Client sides have both created their workers, the Server side proceeds to transmit metadata which includes its worker address and corresponding address length to the Client side through Out-of-Band socket-based communication. Following the completion of this transmission, both the Server and Client sides enter the phase of creating endpoints for communication, which involves the Client side transmitting information about its worker address to the Server side, and the Server side transmitting a test string to the Client side.

      It is noteworthy that the Server side and the Client side subsequently call different functions. Specifically, the Server side invokes the run_ucx_server() function, while the Client side calls the run_ucx_client() function. Although the responsibilities of these two functions are similar, there are subtle differences. The following analysis is divided into two sections to elaborate on each side.

        - Client side - run_ucx_client()

During the execution of the run_ucx_client() function, the Client side, having obtained the Server side's worker address through a prior socket transmission, can directly utilize this peer address to establish an endpoint for transmitting its worker address to the Server side. The establishment of the endpoint involves memory allocation and parameter initialization for the endpoint, as well as the construction and selection of lanes.

    - Server side - run_ucx_server()

      Initially, the Server side receives the worker address information transmitted from the Client side. Subsequently, it utilizes this address information to establish an endpoint and follows the same pattern to transmit a test string to the Client side.

  It's worth noting that the communication between both the Server and Client sides no longer follows the socket transmission pattern; instead, it utilizes UCP (Unified Communication Protocol) endpoint communication.

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

   - `ucp_context`

     Looking at its corresponding struct, the UCP context encompasses various pieces of information. This includes memory domain resources, mappings, and indexes for multiple memory domains. Additionally, it contains details about available communication resources and the enabled TLS protocols.

   - `ucp_worker`

     UCP worker contains more than just basic information like a unique ID (uuid), worker address, and its connection to a specific context. It serves as a dashboard that keeps track of all the endpoints it manages. This includes details about each endpoint's configurations, the total number of endpoints, and even statistics regarding the success or failure of endpoint creations. In simpler terms, it's like a control center overseeing and managing various aspects of all the underlying endpoints.

   - `ucp_ep`

     UCP endpoint encompasses various details such as the sequence number of the remote connection, cached values, and its affiliation with a specific UCP worker. It also includes configuration information, stored in the form of a worker configuration index. The exact configuration can be accessed through this index in the config array within its associated worker object.

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

   - `UCX_TLS`

     As the UCP context stores information about available TLS protocols, I believe this information should be prepared before creating the UCP context object. I speculate that system environment variable settings might be configured here.

- TLS selected by UCX

  Because communication connections between different endpoints may utilize different TLS protocols, I believe the selection of TLS protocols should occur during the establishment of UCP endpoints.

## 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?

   - src/ucs/config/types.h

     Added the `UCS_CONFIG_PRINT_TLS` flag in the configuration printing flags.

   - src/ucs/config/parser.c

     Continuing with the modification of the ucs_config_parser_print_opts() function in parser.c, an if-statement for printing TLS has been added. Within this statement, all supported TLS protocols are extracted from environment variables and printed.

   - src/ucp/core/ucp_worker.c

     In the ucp_worker_print_used_tls() function, the ucp_config_print() function is invoked. This function subsequently calls the ucs_config_parser_print_opts() function to print Line 1. Following that, using fprintf in conjunction with a string buffer containing selected TLS information, Line 2 is printed.

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

   At the outset, in the ucp_worker_print_used_tls() function located in `src/ucp/core/ucp_worker.c`, the ucp_config_print() function is called with `UCS_CONFIG_PRINT_TLS` as a printing flag parameter. This function then proceeds to invoke the ucs_config_parser_print_opts() function in `src/ucs/config/parser.c`, ultimately printing the available TLS protocols in the system based on the content of environment variables.

   Given that environment variables and most printing functions are located in `src/ucs/config/parser.c`, in `src/ucp`, if there is a need to print any configuration, it is necessary to utilize the ucp_config_print() function to call the printing functions in `src/ucs` and effectively print the configuration.

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

   During the creation process of an endpoint, the initial step involves the retrieval of the remote address and the ucp_ep_create() function call in `src/ucp/core/ucp_ep.c`.

   Following this, through a series of function chain calls, the ucp_worker_get_ep_config() function will invoke the ucp_worker_print_used_tls() function, connecting to the previously mentioned implementation to print out Line 1 and Line 2.

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

Yes, this matches my expectations. Because communication between different endpoints may adopt different TLS protocols, and the information implied in Line 2 includes the TLS protocol currently selected for communication, placing the printout of Line 2 after the creation of the endpoint is relatively appropriate.

As for Line 1, the TLS protocols available in the system should ideally be configured before endpoint creation. Therefore, Line 1 doesn't necessarily need to be printed after the endpoint creation, but for the sake of implementation convenience, I chose to print both Line 1 and Line 2 together within the ucp_worker_print_used_tls() function.

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

     lanes represent communication channels between endpoints, and communication between two endpoints may be composed of multiple lanes. Each lane may also carry different TLS protocols.

   - tl_rsc

     tl_rsc is a resource descriptor object that represents the network resource. This information encompasses details such as the transport name, hardware device name, the device represented by the resource, and the identifier associated with the device.

   - tl_name

     tl_name represents the transport name, and this information is stored within the tl_rsc object.

   - tl_device

     tl_device is an internal resource descriptor object of a transport device, containing information such as the hardware device name, device type, and system device.

   - bitmap

     tl_bitmap is a bitmap type for representing which TL resources are in use.

   - iface

     iface primarily serves as an interface that stores various pieces of information such as memory domain, active worker, active message handlers, and error handler.

## 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
```

Due to the current system environment settings limiting the use of ud_verbs in the UCX_TLS configuration, I opted to optimize by enabling all available TLS protocols. This allows the UCX framework to autonomously select the most suitable transmission method. The table below illustrates the measured latency and bandwidth for different data sizes before and after the optimization.

| Data Size | Latency (us) - Before Optimization | Latency (us) - After Optimization |
|---|---|---|
| 1 | 1.73 | 0.28 |
| 2 | 1.77 | 0.27 |
| 4 | 1.69 | 0.28 |
| 8 | 1.91 | 0.30 |
| 16 | 1.83 | 0.28 |
| 32 | 1.82 | 0.31 |
| 64 | 2.16 | 0.32 |
| 128 | 2.02 | 0.47 |
| 256 | 3.60 | 0.50 |
| 512 | 3.74 | 0.56 |
| 1024 | 4.72 | 0.61 |
| 2048 | 6.34 | 0.71 |
| 4096 | 9.81 | 1.39 |
| 8192 | 13.77 | 1.72 |
| 16384 | 15.49 | 3.03 |

| Data Size | Latency (us) - Before Optimization | Latency (us) - After Optimization |
|---|---|---|
| 32768 | 23.44 | 4.92 |
| 65536 | 39.42 | 8.88 |
| 131072 | 66.70 | 22.57 |
| 262144 | 126.07 | 41.05 |
| 524288 | 238.52 | 73.02 |
| 1048576 | 456.61 | 142.02 |
| 2097152 | 925.98 | 345.61 |
| 4194304 | 1816.75 | 1085.06 |

From the experimental results, it can be observed that there is a significant improvement in latency for different data sizes after optimization. And next, let's take a look at the bandwidth performance.

| Data Size | Bandwidth (MB/s) - Before Optimization | Bandwidth (MB/s) - After Optimization |
|---|---|---|
| 1 | 0.28 | 9.56 |
| 2 | 0.27 | 19.35 |
| 4 | 0.28 | 38.85 |
| 8 | 0.30 | 78.44 |
| 16 | 0.28 | 156.44 |
| 32 | 0.31 | 307.27 |
| 64 | 0.32 | 614.86 |
| 128 | 0.47 | 611.86 |
| 256 | 0.50 | 1147.70 |
| 512 | 0.56 | 2240.55 |
| 1024 | 0.61 | 3847.90 |
| 2048 | 0.71 | 5713.45 |
| 4096 | 1.39 | 7798.33 |
| 8192 | 1.72 | 9961.55 |
| 16384 | 3.03 | 4864.03 |
| 32768 | 4.92 | 6751.53 |
| 65536 | 8.88 | 8119.61 |
| 131072 | 22.57 | 8639.28 |
| 262144 | 41.05 | 8226.28 |

| Data Size | Bandwidth (MB/s) - Before Optimization | Bandwidth (MB/s) - After Optimization |
|---|---|---|
| 524288 | 73.02 | 8123.49 |
| 1048576 | 142.02 | 7755.89 |
| 2097152 | 345.61 | 7435.01 |
| 4194304 | 1085.06 | 7265.96 |

The results above indicate that allowing the UCX framework to autonomously choose suitable TLS protocols can significantly enhance the transmission bandwidth.

### 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
```

The following are the results of latency and bandwidth experiments conducted for both single node and multiple nodes.

| Data Size | Latency (us) - Multiple Nodes | Latency (us) - Single Node |
|---|---|---|
| 1 | 1.91 | 0.22 |
| 2 | 1.88 | 0.21 |
| 4 | 1.82 | 0.23 |
| 8 | 1.85 | 0.22 |
| 16 | 1.86 | 0.21 |
| 32 | 1.86 | 0.25 |
| 64 | 2.00 | 0.26 |
| 128 | 3.15 | 0.37 |
| 256 | 3.32 | 0.39 |
| 512 | 3.58 | 0.43 |
| 1024 | 4.15 | 0.51 |
| 2048 | 5.22 | 0.68 |

| Data Size | Latency (us) - Multiple Nodes | Latency (us) - Single Node |
|---|---|---|
| 4096 | 7.41 | 1.00 |
| 8192 | 9.51 | 1.69 |
| 16384 | 12.84 | 3.02 |
| 32768 | 18.59 | 4.99 |
| 65536 | 29.84 | 8.80 |
| 131072 | 52.86 | 17.97 |
| 262144 | 94.64 | 36.20 |
| 524288 | 180.74 | 68.26 |
| 1048576 | 353.03 | 136.20 |
| 2097152 | 698.36 | 339.16 |
| 4194304 | 1389.97 | 1086.51 |

| Data Size | Bandwidth (MB/s) - Multiple Nodes | Bandwidth (MB/s) - Single Node |
|---|---|---|
| 1 | 2.99 | 9.83 |
| 2 | 6.14 | 19.84 |
| 4 | 12.32 | 40.00 |
| 8 | 24.48 | 80.43 |
| 16 | 49.63 | 161.19 |
| 32 | 99.32 | 313.58 |
| 64 | 184.53 | 621.78 |
| 128 | 260.02 | 606.21 |
| 256 | 505.40 | 1261.24 |
| 512 | 968.75 | 2365.25 |
| 1024 | 1484.27 | 3887.74 |
| 2048 | 2194.92 | 5846.44 |
| 4096 | 2518.75 | 7992.58 |
| 8192 | 2749.99 | 10495.47 |
| 16384 | 2857.75 | 5057.40 |
| 32768 | 2916.34 | 6886.99 |
| 65536 | 2945.86 | 8346.24 |
| 131072 | 2961.85 | 8877.12 |

| Data Size | Bandwidth (MB/s) - Multiple Nodes | Bandwidth (MB/s) - Single Node |
|---|---|---|
| 262144 | 2940.03 | 8511.21 |
| 524288 | 3032.17 | 8452.69 |
| 1048576 | 3034.36 | 8323.47 |
| 2097152 | 3035.78 | 8331.44 |
| 4194304 | 3036.38 | 7854.95 |

From the above experimental results, it can be observed that utilizing multiple nodes does not perform as well in terms of latency and bandwidth compared to transmissions on a single node. Notably, the TLS protocols chosen by the UCX framework differ between single and multiple nodes. In the case of a single node, intra-node communication is employed, coupled with sysv/memory and cma/memory protocols. On the other hand, for multiple nodes, inter-node communication is selected, utilizing rc_verbs and TCP protocols.

However, this observation is quite reasonable. Given that multiple nodes require cross-node transmissions, network communication protocols such as TCP/IP or InfiniBand are typically needed. This aligns with the communication protocols chosen by the UCX framework, namely RDMA and TCP. However, these types of transmissions usually incur higher latency and lower bandwidth compared to in-node transmissions.

As for in-node transmissions, communication can be accomplished within a single node, allowing for the selection of communication methods with lower latency and higher bandwidth, such as shared memory. It also corresponds to the protocols chosen by the UCX framework in the case of a single node.

## 4. Experience & Conclusion

1. What have you learned from this homework?

   Through this assignment, I gained a deeper understanding of how the current UCX system architecture implements the communication mechanisms on both the server and client sides. The details include the construction of UCP context, worker, and endpoint, outlining the specific communication-related information each of them encompasses. It also involves understanding how these components collaborate, delineating their respective roles, and interacting with each other through function APIs.

2. Feedback (optional)

   Tracing code in such a large-scale project is extremely exhausting... 😦