PP HW 5 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.

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`
    - `ucp_worker_create`
    - `ucp_ep_create`

Ans:

- `ucp_init()` is invoked within the `main()` function in `ucp_hello_world.c`. It's used to create UCP context `(ucp_context.h)` which holds global resources(e.g., network interfaces -> `ucp_fill_resources(self, config_env_prefix)`, memory registration caches -> `ucp_mem_rcache_init(context)`, etc.). It's the first call at the UCP level.
- `ucp_worker_create()` creates a `ucp_worker_h` object. A `ucp_worker` represents a communication resource (thread context, progress engine) used for sending/receiving data. The `ucp_worker_create()` function allocates memory for a worker object and initialize basic fields and data structures for EP matching, EP lookup, etc.
- `ucp_ep_create()` creates UCP endpoints. A UCP endpoint `(ucp_ep_h)` is a logical connection to a remote worker. The local worker and the remote worker's addresses are required to establish this communication link, and after the endpoint is created, it can be used to send or receive messages with the remote peer.
- To sum it up, the `ucp_context` is the highest-level object, created once via `ucp_init()`. In the context, one or more `ucp_worker` objects can be created with `ucp_worker_create()`. Then in each worker, one or more `ucp_ep` objects(endpoints) can be created with `ucp_ep_create()` to connect to remote peers.

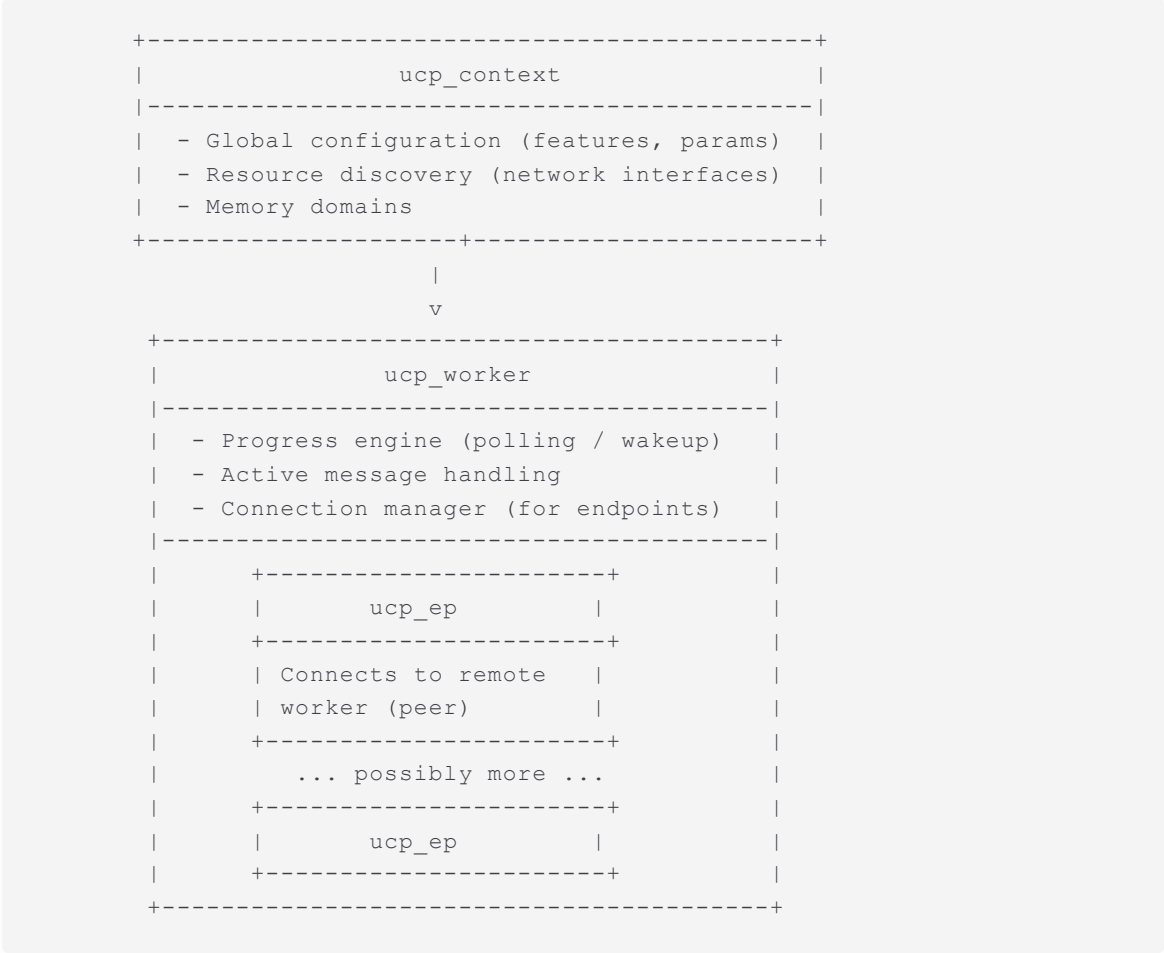
2. UCX abstracts communication into three layers as below. Please provide a diagram illustrating the architectural design of UCX.

- `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`

Ans:

- Diagram:

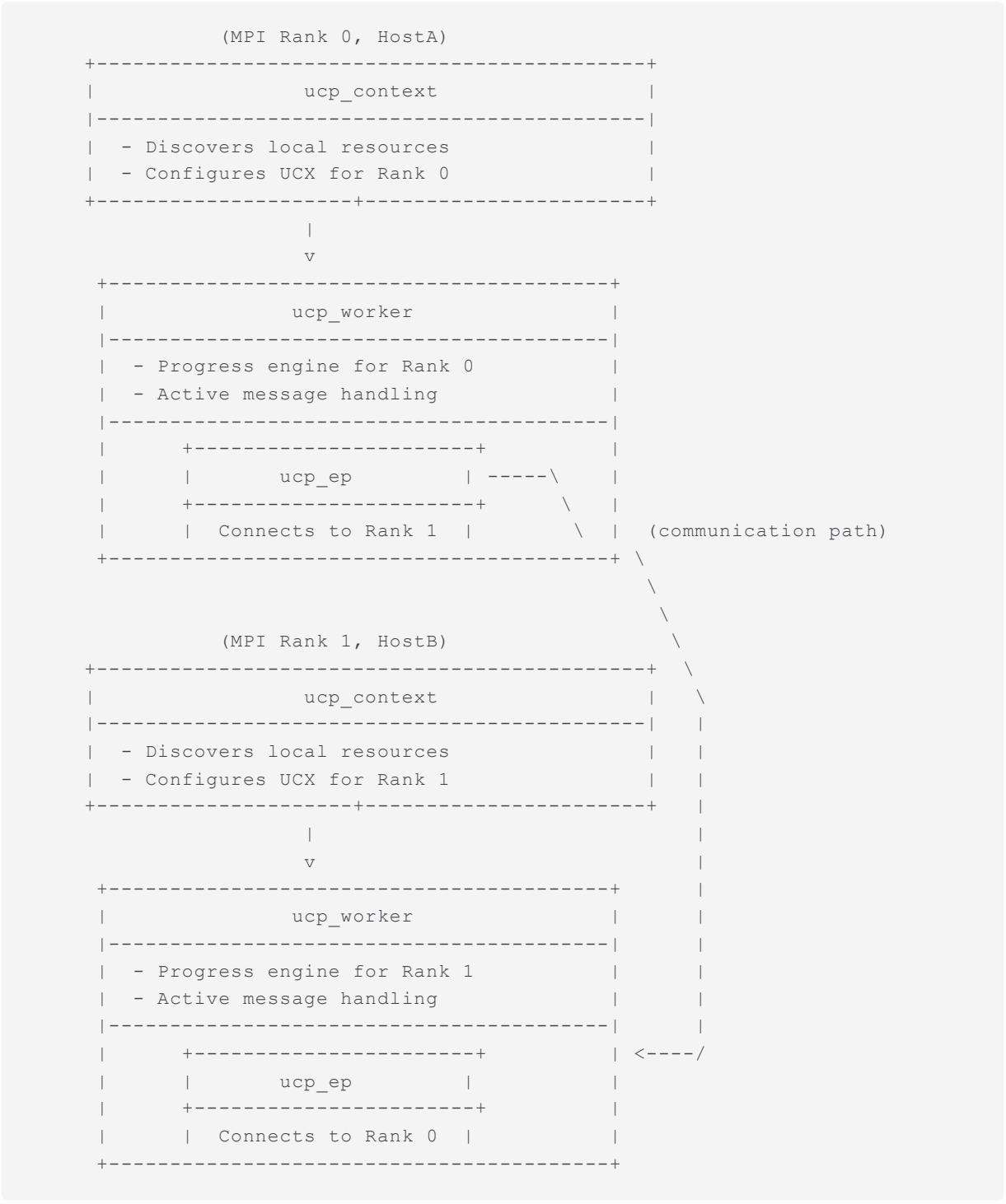


Explanation:

This is the diagram illustrating the architectural design of UCX. `ucp_context` is the top-level object of the UCX library. It stores global parameters, capabilities, features, and configurations that govern how UCX behaves across the entire application. Within a context, `ucp_workers` can be created. They're responsible for the actual progress of communication (polling or waking up on incoming messages). A UCX application might create one worker per thread or per process, depending on concurrency needs. Each worker operates independently, managing its own connections and progress. Each worker can maintain multiple endpoints(`ucp_ep`), one endpoint to each remote process/peer it

needs to communicate with. In an MPI environment, that might mean one endpoint for each MPI rank to talk to directly.

- `mpiucx --host HostA:1,HostB:1 ./send_recv.out`: Here is a diagram when the command is executed:



Explanation:

The diagram above is showing two processes (Rank 0 on HostA, Rank 1 on HostB). Each process has its own `ucp_context` and `ucp_worker`. Then, each creates a `ucp_ep` to

communicate with the peer. Both MPI processes have one UCX context, one worker, and endpoints each, connecting to each other.

3. Based on the description in HW5, where do you think the following information is loaded/created?
    - `UCX_TLS`
    - TLS selected by UCX

Ans:

- For `UCX_TLS`, I think it's loaded by the parser within the context, which is the function `ucs_config_parser_fill_opts()` located at `line 632` in `src/ucp/core/ucp_context.c`. It is then stored in a `ucp_config_t` data structure.

- Workflow for TLS selection:

```
ucp_ep_create()
   └──> ucp_wireup_init_lanes()
          └──> ucp_wireup_select_lanes()
                 └──> ucp_wireup_search_lanes()
```

`ucp_ep_create()` is located in `src/ucp/core/ucp_ep.c`, and is the main entry for creating a UCP endpoint. `ucp_ep_create_api_to_worker_addr()` in `src/ucp/core/ucp_ep.c` is then invoked when a remote worker's address is provided. After several calls, `ucp_wireup_init_lanes()` in `src/ucp/core/ucp_ep.c` is invoked to select and initialize the best transport “lanes” according to UCX_TLS, device capabilities, and scores. `ucp_wireup_select_lanes()` in `src/ucp/wireup/select.c` calls `ucp_wireup_search_lanes()` in the same file and completes the process of selecting the transport.

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?
2. How do the functions in these files call each other? Why is it designed this way?
3. Observe when Line 1 and 2 are printed during the call of which UCP API?
4. Does it match your expectations for questions **1-3**? Why?
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.

Ans:

1. I modified several files. I will list them in the order they were invoked.

- `UCS_CONFIG_PRINT_TLS`: Added to a structure located at `src/ucs/config/types.h` as a new flag.

```
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),
    // Added
    UCS_CONFIG_PRINT_TLS             = UCS_BIT(5)
} ucs_config_print_flags_t;
```

Those flags control which parts of the configuration to be printed. By adding the new flag, it allows the user to print only the `TLS` field.

- `ucp_worker_print_used_tls()`: Located at `src/ucp/core/ucp_worker.c`, this function is a diagnostic routine that logs which features/protocol lanes are used and also shows the configured transport selection `(UCX_TLS)`, helping confirm which “TLS” (transports) are active on a given endpoint.

```
static void
ucp_worker_print_used_tls(ucp_worker_h worker, ucp_worker_cfg_index_t cfg_i
{
    // A variable has to be declared at the top according to C90 standard.
    ucp_config_t *config;

    /* Existing code
        ...
    */

    // Read the configurations into config
    ucp_config_read(NULL, NULL, &config);

    // Print the configuration associating with the flag, which
    // is UCS_CONFIG_PRINT_TLS in this case
    ucp_config_print(config, stdout, NULL, UCS_CONFIG_PRINT_TLS);

    // "strb" variable is a "ucs_string_buffer_t" that builds up a textual
    // and transports (e.g., ud_verbs, rc_x, shm, etc.) the endpoint is act
    fprintf(stdout, "%s\n", ucs_string_buffer_cstr(&strb));

    // Release
    ucp_config_release(config);
}
```

The code above is a snippet of the parts I added. A variable `config` is declared for storing the configurations. It was populated with `ucp_config_read()`, then printed out using `ucp_config_print()` **(Line 1)**. To determine what element to be printed, a flag is used, which is `UCS_CONFIG_PRINT_TLS`.

Then I used `fprintf()` to print the `strb` variable. It is a `ucs_string_buffer_t` that builds up a textual description of which features (tag, active message, RMA, AMO, etc.) and transports (e.g., ud_verbs, rc_x, shm, etc.) the endpoint is actually using **(Line 2)**.

After retrieving the configuration information, release the object.

- `void ucs_config_parser_print_opts()`: Located at `src/ucs/config/parser.c`, it is responsible for printing configuration fields based on a set of bitmask flags (e.g., `UCS_CONFIG_PRINT_CONFIG`, `UCS_CONFIG_PRINT_TLS`, etc.).

```
1  void ucs_config_parser_print_opts(FILE *stream, const char *title, const vo
2                                    ucs_config_field_t *fields, const char *t
3                                    const char *prefix, ucs_config_print_flag
4  {
5      /* Existing code
6         ...
7       */
8
9      // TODO: PP-HW-UCX
10     if (flags & UCS_CONFIG_PRINT_TLS) {
11         // Josh: Add at least one prefix element, or the parser will crash
12         table_prefix_elem.prefix = table_prefix ? table_prefix : "";
13         ucs_list_add_tail(&prefix_list, &table_prefix_elem.list);
14
15         ucs_config_parser_print_opts_recurs(stream, opts, fields, flags, pr
16     }
17
18     /* Existing code
19        ...
20      */
21 }
```

Invoked by using the flag `UCS_CONFIG_PRINT_TLS`, I will only print the `TLS` configuration instead of the whole thing. The first two lines within the `if` block I implemented guarantee some prefix will exist in the list. It then called `ucs_config_parser_print_opts_recurs()` to iterate through the configurations and print the field requested.

- `static void ucs_config_parser_print_opts_recurs()`: Located in `src/ucs/config/parser.c`, this function iterates through all the fields recursively, printing out requested ones.

```c
static void
ucs_config_parser_print_opts_recurs(FILE *stream, const void *opts,
                                    const ucs_config_field_t *fields,
                                    unsigned flags, const char *prefix,
                                    ucs_list_link_t *prefix_list)
{
    /* Existing Code
     * ...
     */
    // Josh: The field name is declared within ucp_config_table[] in UCX-ls
    if ((flags & UCS_CONFIG_PRINT_TLS) &&
                strcmp(field->name, "TLS")) {
        continue;
    }
    ucs_config_parser_print_field(stream, opts, prefix, prefix_list,
                                  field->name, field, flags, NULL);
}
```

In the `if` block I implemented, it will check whether the current field name matches "TLS" when the `UCS_CONFIG_PRINT_TLS` is set. If the flag is set and the field name does not match, it continues to iterate through the fields. Otherwise, it is printed using `ucs_config_parser_print_field()`.

2. **Call Flow Diagram:**

```
ucp_ep_create()
   |
   └── ucp_ep_create_to_sock_addr()
         |
         └── ucp_ep_init_create_wireup()
               |
               └── ucp_worker_get_ep_config()
                     |
                     └── ucp_worker_print_used_tls()
                           |
                           ├── ucp_config_print()
                           |     |
                           |     └── ucs_config_parser_print_opts()
                           |           |
                           |           └── ucs_config_parser_print_opts_recurs()
                           |                 |
                           |                 └── ucs_config_parser_print_field()
                           |                       |
                           |                       └── (prints "UCX_TLS=ud_verbs")
                           |
                           └── (prints "cfg#0 tag(...)")
```

It's designed in a layered approach to make it straightforward and the same printing code is reusable for various UCX objects.

3. Both lines are printed when `ucp_ep_create()` API is invoked. For line 2, it's directly printed using `fprintf()` in `ucp_worker_print_used_tls()`, while for line 1, it's printed with `ucp_config_print()`, which then calls `ucs_config_parser_print_opts()` to iterate recursively until the `TLS` field is found.

4. Yes, it matched my expectations. Both of the output is printed within the `ucp_ep_create()` API, since the transport configuration is determined when an endpoint is created. `ucp_worker_print_used_tls()` within `ucp_ep_create()` invoked another series of functions to print the logs I needed.

5. Explanation of different variables:

- **Lanes:** A lane is a logical communication path within a UCP endpoint. For each lane the UCX tracks which transport interface is used and also included some metadate(bandwidth, priority, etc.)
- **tl_rsc:** Structure defined at `src/uct/api/uct.h` as `uct_tl_resource_desc_t`. This is a transport resource descriptor. It includes the transport name(`tl_name`), device name (`dev_name`), device type(`dev_type`), and a system-level-identifier. Each entry represents one resource, representing available network resources on the system.
- **tl_name:** Identifies which transport the resource uses. This tells UCX what kind of communication method is used(`InfiniBand RC` , `InfiniBand UD` , `TCP` , etc.).
- **tl_device:** Structure is defined at `src/uct/base/uct_iface.h` as `uct_tl_device_resource_t` . It contains a hardware device name (`char name[UCT_DEVICE_NAME_MAX]`) which identifies the specific device or interface, a device type (`uct_device_type_t type`) that tells UCX what kind of resource it is, and a system device identifier (`ucs_sys_device_t sys_device`) that captures system topology information such as NUMA node or PCI bus ID.
- **bitmap:** It is used throughout the program to store a set of flags or indices. Each bit in the bitmap corresponds to some resource index (e.g. lane index, device index, feature flag).
- **iface:** Refers to the `UCT interface handle` in UCX. It is the actual handle used to communicate over the given transport. For example, `UD Verbs` or `RC Verbs` or `TCP` will each have their own `uct_iface_t` .

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
```

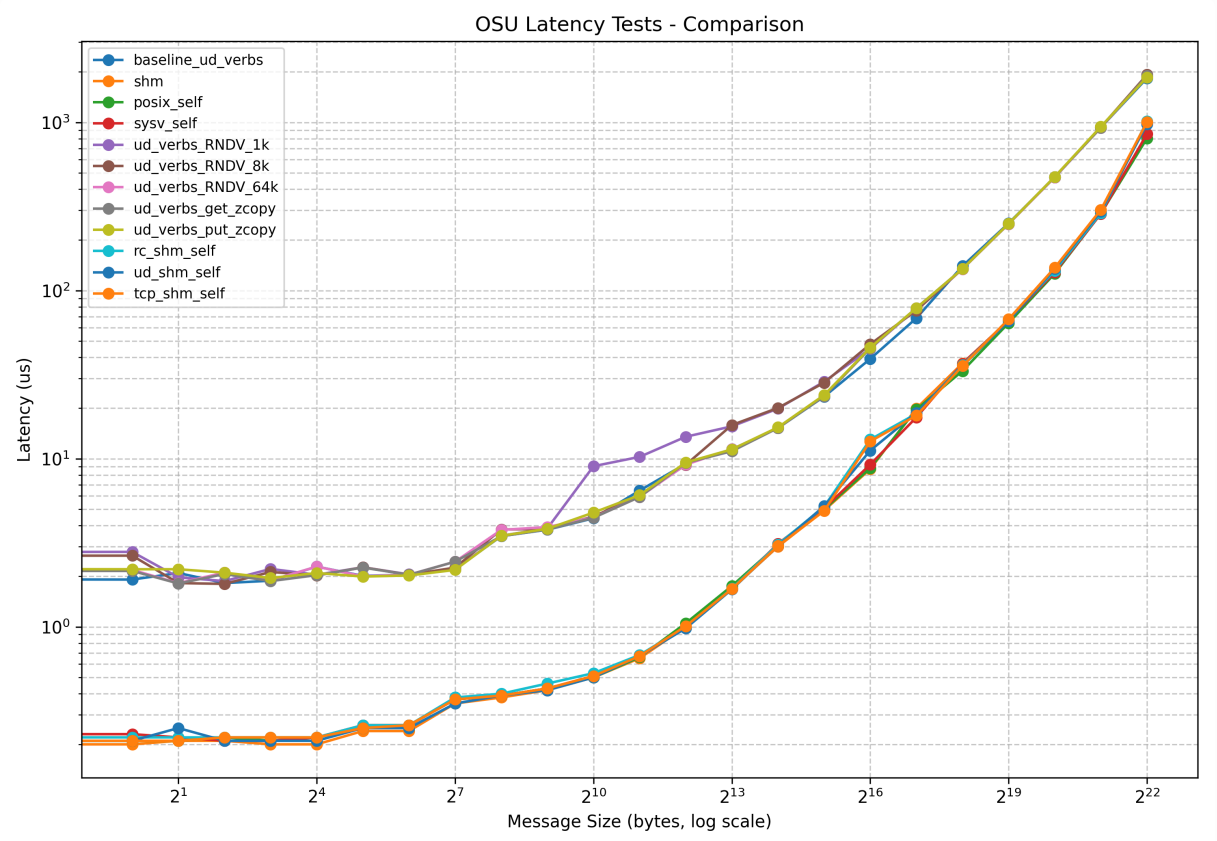
2. Please create a chart to illustrate the impact of different parameter options on various data sizes and the effects of different testsuite.
3. Based on the chart, explain the impact of different TLS implementations and hypothesize the possible reasons (references required).

Ans:
Below is the experiment table:

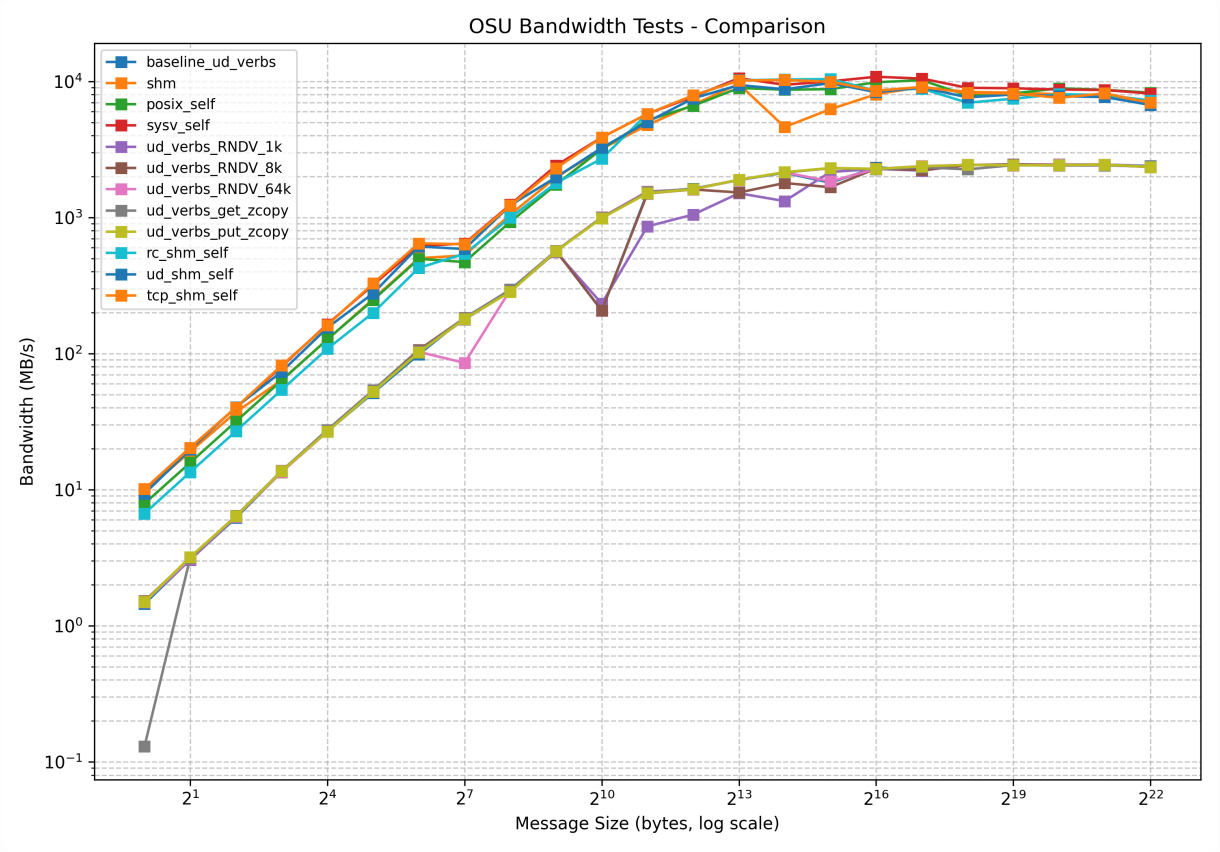
Experiment	UCX Environment Variables	Notes
1. Baseline	`UCX_TLS=ud_verbs`	Default config, no special tuning.
2. shm	`UCX_TLS=shm` `UCX_LOG_LEVEL=error`	Forces shared memory on node.
3. posix,self	`UCX_TLS=posix,self`	Single SHM transport (POSIX).
4. sysv,self	`UCX_TLS=sysv,self`	Single SHM transport (SysV).
5. ud_verbs, RNDV=1K	`UCX_TLS=ud_verbs` `UCX_RNDV_THRESH=1024`	Lower RNDV threshold.
6. ud_verbs, RNDV=8K	`UCX_TLS=ud_verbs` `UCX_RNDV_THRESH=8192`	Medium RNDV threshold.

7. ud_verbs, RNDV=64K	`UCX_TLS=ud_verbs` `UCX_RNDV_THRESH=65536`	Higher RNDV threshold.
8. ud_verbs, get_zcopy	`UCX_TLS=ud_verbs` `UCX_RNDV_SCHEME=get_zcopy`	RNDV scheme (GET).
9. ud_verbs, put_zcopy	`UCX_TLS=ud_verbs` `UCX_RNDV_SCHEME=put_zcopy`	RNDV scheme (PUT).
10. rc,shm,self	`UCX_TLS=rc,shm,self`	RC Verbs + shared memory + self.
11. ud,shm,self	`UCX_TLS=ud,shm,self`	UD Verbs + shared memory + self.
12. tcp,shm,self	`UCX_TLS=tcp,shm,self`	TCP + shared memory + self.

Latency:



Bandwidth:



I ran a total of 12 experiments (see the table above), each adjusting UCX environment variables to explore different transports (e.g., `ud_verbs`, `rc_verbs`, `shm`, `tcp`) and different rendezvous parameters (`UCX_RNDV_THRESH` and `UCX_RNDV_SCHEME`). Then I measured OSU MPI Latency and OSU MPI Bandwidth in two combined charts:

- **Latency:**
  - For small messages (`1-128 bytes`), shared-memory transports (`shm`, `posix_self`, `sysv_self`) consistently show the lowest latency, often around 0.2–0.3 microseconds, whereas baseline ud_verbs is closer to 2 microseconds.
  - For mid-range messages (`1KB-64KB`), Shared-memory remains faster. Rendezvous threshold changes (`1K`, `8K`, `64K`) begin to matter around these sizes for `ud_verbs`, but `shm` is still more efficient within a single node.
  - For Large messages (`≥1MB`): All methods scale up in latency, but `shm` tends to remain lower than networking transports for on-node communication.

- **Bandwidth:**
  - For small to medium messages: Shared-memory–based runs has higher bandwidth earlier, e.g., `6-10 GB/s` in the `4KB-64KB` range. In contrast, baseline `ud_verbs` typically lags behind at around `2-3 GB/s` in that same range.

- Rendezvous Tuning: Adjusting thresholds (`1K` , `8K` , `64K`) or switching between `get_zcopy` and `put_zcopy` can differ where large-message throughput peaks, but on a single node, `shm` still dominates most message sizes in terms of peak `MB/s` .
- Configurations like `rc` , `shm` , `self` or `ud` , `shm` , `self` allow fallback to `InfiniBand` or `UD` for multi-node runs, but remain high-performing for local shared-memory traffic as well.

Some Discussions: On a single node, shared memory transports (`shm` , `posix` , `sysv`) bypass NIC and kernel overhead, producing latencies under `0.3 µs` and bandwidths of `6–10 GB/s` in midrange sizes. In contrast, `ud_verbs` or `rc_verbs` incur additional InfiniBand overhead even on the same host. Tuning rendezvous thresholds can help avoid excess protocol handshaking for large messages, but small-message latency typically favors eager protocols.

Some Extras:

- Rendezvous: A rendezvous threshold is the message-size cutoff that determines when UCX switches from an eager protocol (which immediately sends data without waiting for the receiver) to a rendezvous protocol (which performs a handshake before transferring the bulk of the data). For smaller messages, the eager approach typically has lower latency because it avoids extra control exchanges. However, when messages exceed the threshold, rendezvous can reduce the number of memory copies or buffer usage for large data transfers, improving overall throughput. Setting the threshold too low causes mid-sized messages to incur handshake overhead unnecessarily, while setting it too high can delay the benefits of reduced data copying for large messages.

4. Experience & Conclusion

1. What have you learned from this homework?

Ans:

I learned the concept of UCX's internal structure, especially how `ucp_context` , `ucp_worker` , and `ucp_ep` cooperate and how the environment variables (like `UCX_TLS`) are parsed and applied. I also gained understanding of the choosing of transport type (`InfiniBand` , `shared memory`) for each endpoint. Most important of all, the command

```
grep -rn "function_name"
```

let me find the target function quickly without browsing through all the files.

2. How long did you spend on the assignment?

Ans:

Took me about a week to trace through the codes and finish the implementation. The report took me another two days or so.
