Parallel Graph

Objectives

- Learn how to design and implement parallel programs
- Gain skills in using synchronization primitives for parallel programs
- Get a better understanding of the POSIX threading and synchronization API
- Gain insight on the differences between serial and parallel programs

Statement

Implement a generic thread pool, then use it to traverse a graph and compute the sum of the elements contained by the nodes. You will be provided with a serial implementation of the graph traversal and with most of the data structures needed to implement the thread pool. Your job is to write the thread pool routines and then use the thread pool to traverse the graph.

Support Code

The support code consists of the directories:

- `src/` is the skeleton parallel graph implementation. You will have to implement missing parts marked as `TODO` items.
- `utils/` utility files (used for debugging & logging)
- `tests/` are tests used to validate (and grade) the assignment.

Implementation

Thread Pool Description

A thread pool contains a given number of active threads that simply wait to be given specific tasks. The threads are created when the thread pool is created. Each thread continuously polls the task queue for available tasks. Once tasks are put in the task queue, the threads poll tasks, and start running them. A thread pool creates **N** threads upon its creation and does not destroy (join) them throughout its lifetime. That way, the penalty of creating and destroying threads ad-hoc is avoided. As such, you must implement the following functions (marked with `TODO` in the provided skeleton, in `src/os_threadpool.c`):

- `enqueue_task()`: Enqueue task to the shared task queue. Use synchronization.
- `dequeue_task()`: Dequeue task from the shared task queue. Use synchronization.
- `wait_for_completion()`: Wait for all worker threads. Use synchronization.
- `create_threadpool()`: Create a new thread pool.
- `destroy_threadpool()`: Destroy a thread pool. Assume all threads have been joined.

You must also update the `os_threadpool_t` structure in `src/os_threadpool.h` with the required bits for synchronizing the parallel implementation.

Notice that the thread pool is completely independent of any given application. Any function can be registered in the task queue.

Since the threads are polling the task queue indefinitely, you need to define a condition for them to stop once the graph has been traversed completely. That is, the condition used by the `wait_for_completion()` function. The recommended way is to note when no threads have any more work to do. Since no thread is doing any work, no other task will be created.

Graph Traversal

Once you have implemented the thread pool, you need to test it by doing a parallel traversal of all connected nodes in a graph. A serial implementation for this algorithm is provided in `src/serial.c`. To make use of the thread pool, you will need to create tasks that will be put in the task queue. A task consists of 2 steps:

1. Add the current node value to the overall sum.
2. Create tasks and add them to the task queue for the neighbouring nodes.

Implement this in the `src/parallel.c` (see the `TODO` items). You must implement the parallel and synchronized version of the `process_node()` function, also used in the serial implementation.

Synchronization

For synchronization you can use mutexes, semaphores, spinlocks, condition variables - anything that grinds your gear. However, you are not allowed to use hacks such as `sleep()`, `printf()` synchronization or adding superfluous computation.

Input Files

Reading the graphs from the input files is being taken care of the functions implemented in `src/os_graph.c`. A graph is represented in input files as follows:

- First line contains 2 integers `N` and `M`: `N` - number of nodes, `M` - numbed or edges
- Second line contains `N` integer numbers - the values of the nodes.
- The next `M` lines contain each 2 integers that represent the source and the destination of an edge.

Data Structures

Graph

A graph is represented internally by the `os_graph_t` structure (see `src/os_graph.h`).

List

A list is represented internally by the `os_queue_t` structure (see `src/os_list.h`). You will use this list to implement the task queue.

Thread Pool

A thread pool is represented internally by the `os_threadpool_t` structure (see `src/os_threadpool.h`). The thread pool contains information about the task queue and the threads.

Requirements

Your implementation needs to be contained in the `src/os_threadpool.c`, `src/os_threadpool.h` and `src/parallel.c` files. Any other files that you are using will not be taken into account. Any modifications that you are doing to the other files in the `src/` directory will not be taken into account.

Operations

Building

To build both the serial and the parallel versions, run `make` in the `src/` directory:

```
student@so:~/.../content/assignments/parallel-graph$ cd src/

student@so:~/.../assignments/parallel-graph/src$ make
```

That will create the `serial` and `parallel` binaries.

Testing and Grading

Testing is automated. Tests are located in the `tests/` directory.

```
student@so:~/.../assignments/parallel-graph/tests$ ls -F
Makefile  checker.py  grade.sh@  in/
```

To test and grade your assignment solution, enter the `tests/` directory and run `grade.sh`. Note that this requires linters being available. The easiest is to use a Docker-based setup with everything installed and configured. When using `grade.sh` you will get grades for checking correctness (maximum `90` points) and for coding style (maxim `10` points). A successful run will provide you an output ending with:

```
### GRADE


Checker:                                                         90/ 90
Style:                                                           10/ 10
Total:                                                          100/100


### STYLE SUMMARY
```

Running the Checker

To run only the checker, use the `make check` command in the `tests/` directory:

```
student@so:~/.../assignments/parallel-graph/tests$ make check
[...]
SRC_PATH=../src python checker.py
make[1]: Entering directory '...'
rm -f *~
[...]
TODO
test1.in                        ...................... failed ...   0.0
test2.in                        ...................... failed ...   0.0
test3.in                        ...................... failed ...   0.0
[...]

Total:                                                            0/100
```

Obviously, all tests will fail, as there is no implementation.

Each test is worth a number of points. The maximum grade is `90`.

A successful run will show the output:

```
student@so:~/.../assignments/parallel-graph/tests$ make check
[...]
SRC_PATH=../src python checker.py
test1.in                        ...................... passed ...   4.5
test2.in                        ...................... passed ...   4.5
test3.in                        ...................... passed ...   4.5
test4.in                        ...................... passed ...   4.5
test5.in                        ...................... passed ...   4.5
test6.in                        ...................... passed ...   4.5
test7.in                        ...................... passed ...   4.5
test8.in                        ...................... passed ...   4.5
test9.in                        ...................... passed ...   4.5
test10.in                       ...................... passed ...   4.5
test11.in                       ...................... passed ...   4.5
test12.in                       ...................... passed ...   4.5
test13.in                       ...................... passed ...   4.5
test14.in                       ...................... passed ...   4.5
test15.in                       ...................... passed ...   4.5
test16.in                       ...................... passed ...   4.5
test17.in                       ...................... passed ...   4.5
test18.in                       ...................... passed ...   4.5
test19.in                       ...................... passed ...   4.5
test20.in                       ...................... passed ...   4.5

Total:                                                             90/100
```

Running the Linters

To run the linters, use the `make lint` command in the `tests/` directory:

```
student@so:~/.../assignments/parallel-graph/tests$ make lint
[...]
cd .. && checkpatch.pl -f checker/*.sh tests/*.sh
[...]
cd .. && cpplint --recursive src/ tests/ checker/
[...]
cd .. && shellcheck checker/*.sh tests/*.sh
```

Note that the linters have to be installed on your system: `checkpatch.pl`, `cpplint`, `shellcheck`. They also need to have certain configuration options. It's easiest to run them in a Docker-based setup with everything configured.

Fine-Grained Testing

Input tests cases are located in `tests/in/`. If you want to run a single test, use commands such as below while in the `src/` directory:

```
$./parallel ../tests/in/test5.in
-38

$ ./serial ../tests/in/test5.in
-38
```

Results provided by the serial and parallel implementation must be the same for the test to successfully pass.

Previous
« Memory Allocator

Next
Minishell »