Lab 6: Designing a multithreaded runtime environment for parallel computing

Master M1 MOSIG – Université Grenoble Alpes (Ensimag & UFR IM2AG)

2021

1 Introduction

In this lab, you are going to design and implement a *parallel runtime environment* — this is essentially a software library allowing application developers to write efficient parallel code in a relatively simple and portable way. More precisely:

- The notion of *"runtime environment"* here refers to an *"active"* library, i.e., a library that creates and schedules its own threads (through interactions with the operating system) in order to manage the activities of the application that it supports[1]. In this document lab, we will use the expressions "runtime" and "library" interchangeably.
- By "simple and portable", we essentially mean that the design and implementation of the runtime API and internals remove some burden from the application programmers, such as:
  - the necessity to tweak the application code when there is a change regarding the characteristics of the underlying machine (for example, the number of CPUs) and/or regarding the characteristics of the tasks (e.g., number of tasks and their size/duration);
  - the necessity to write long and/or complex pieces of code for the tasks and their management (for instance, in this lab, we will study an API allowing to easily parallelize the execution of some types of algorithms based on `for` loops).[2]

You will start from a provided skeleton that does not support concurrent/parallel execution of tasks. You will gradually introduce additional features and support for efficient parallel execution.

During the different stages of the project, you will have to identify the concurrency issues related to the use of multiple threads, and to propose appropriate solutions to handle these issues.

[1] Some types of runtimes also have different or more advanced features, like the ability to perform garbage collection for heap memory, and/or to translate the code of an application from one binary/executable format to another, and/or to load and run multiple applications concurrently. These aspects are outside of the scope of this lab assignment.

[2] Note that the API of many runtimes is also made more convenient through the use of new constructs provided via extensions of the programming language(s) used for the implementations of the tasks. Such extensions are sometimes also named *"syntactic sugar"*, as they make the life of programmers sweeter, thanks to more concise and easier-to-debug code. This aspect is out of the scope of this lab assignment (to keep the required amount of work within a reasonable level). Hence, the runtime API that we will study is not as convenient as it could be.

Your implementation will be based on the POSIX threads interface (*pthreads*). To solve concurrency issues, you are allowed to use any synchronization mechanisms introduced during the lectures. This includes: pthread mutexes, condition variables, semaphores, barriers (see `man pthread_barrier_init`), read-write locks (see `man pthread_rwlock_init`) and atomic operations[3].

The main goal of this lab is to design and implement a runtime that behaves correctly, despite the pitfalls introduced by the usage of multiple concurrent threads. Obviously, the performance of the parallel runtime also matters, but to a lesser extent. In other words, in the context of this lab, our two main evaluation criteria will be, in decreasing order of importance:

1. The correctness of the code, that is, the fact that its behavior complies with the provided specification and that it does not have concurrency bugs (regarding safety and/or liveness properties).
2. The efficiency of the code, especially regarding its support for parallelism. In particular, an implementation that never executes any task in parallel is not acceptable.

2 Important information

- This assignment will be graded.
- The assignment is to be done by groups of at most **2** students.
- Deadline: Monday November 29, 2021 at 22:00.
- The assignment is to be turned in on Moodle (report + source code).

2.1 Collaboration and plagiarism

You are encouraged to discuss ideas and problems related to this project with the other students. You can also look for additional resources on the Internet. However, we consider plagiarism very seriously. Hence, if any part of your final submission reflects influences from external sources, you must cite these external sources in your report. Also, any part of your design, your implementation, and your report must come from you and not from other students. We will run tests to detect similarities between source codes. Additionally, we will allow ourselves to question you about your submission if part of it looks suspicious, which means that you must be able to explain each line you submitted. In case of plagiarism, your submission will not be graded and appropriate actions will be taken.

2.2 Evaluation of your work

The main points that will be evaluated for this work are (ordered by decreasing priority):

[3] A list of built-in atomic operations for gcc is available at `https://gcc.gnu.org/onlinedocs/gcc-4.4.0/gcc/Atomic-Builtins.html`

- The understanding of the problem.
- The design of the proposed solution.
- The correctness of the code.
- The quality of the code.
- The number of implemented features.

2.3 Your submission

Your submission is to be turned in on Moodle as an archive named with the last name of the two students involved in the project: `Name1_Name2_lab6.tar.gz`.

The archive will include:

- A **concise** report, either in `txt`, `md` (MarkDown) or `pdf` format[4], which must include the following sections:
  - The name of the participants.
  - For each stage of the project:
    - The answers to the questions raised in this document (when applicable).
    - A short description of each concurrency issue that you have identified.
    - A few words about the solution that you designed to solve it (Note that a simple figure is sometimes better than a long paragraph).
    - Any additional information that you consider necessary to understand your code.
    - A list of tests that you pass successfully, and the known bugs if any.
    - A brief description of the new tests that you have designed, if any.
  - (Optional) A feedback section including any suggestions you would have to improve this lab.
- **A version of your source code for each stage of the project** (except for stage 0), in a directory `stage_X` for stage X. **Each version must be self-contained**.

2.4 Expected achievements

Considering the time that is given to you to work on the assignment, here is a scale of our expectations:

- An **acceptable work** is one in which at least Stage 1 has been implemented and works correctly.
- A **good work** is one in which Stages 1 and 2 have been implemented and work correctly.
- An **excellent work** is one in which Stages 1–4 have been implemented and extensively tested.

[4] Other formats will be rejected.

2.5 Tentative schedule

We provide below a indicative schedule that you should try to follow in order to properly manage the work and time needed for completing the project.

- Before the second lab session, you should at least have completed the study of Stage 0 and started working on Stage 1.
- Before the third lab session, you should at least have completed Stage 1 and started working on Stage 2.

3 Description of the runtime

3.1 Main principles

To create parallel programs, we propose to implement a runtime that allows developers to create tasks that can be executed in parallel. The API of the runtime has already been defined. Also, you are provided with a basic version of the runtime that is able to execute tasks sequentially. Your job will be to modify this runtime so that tasks can be executed in parallel, while maintaining the semantics of the API. Of course, while modifying the runtime, several concurrency issues will appear. You will have to identify and solve these issues.

The main principles of the runtime are as follows:

- A task is an instance of executable block of code. A task is executed by a thread. The block of code associated with a task is defined by a function, called *task routine*. Input and output parameters can be associated with a task.
- The runtime relies on a pool of *worker threads*. Tasks can be executed by any worker thread. However, a task cannot be executed concurrently by several workers.
- Two main cases should be distinguished in the creation of tasks:
  - Tasks created by the main thread of the program are independent and can be executed in any order.
  - A task T_p can create new tasks. In this case, a parent-child *dependency* relation is created between the task T_p and the newly created tasks. After the child tasks have been submitted, the task T_p is interrupted and will resume only when the all the children have finished executing. This will be illustrated in the example shown in Figure 2 and discussed in Section 3.3.
- In all cases, the *creation* of a task is done in 3 steps: i) the allocation and initialization of a task instance; ii) the association of input and output parameters to the new task instance; iii) the submission of the task instance for execution.
- The execution of tasks is *asynchronous*: tasks are not necessarily executed immediately when they are submitted to the runtime.

- The main thread is provided with a `waitall()` function that it should call every time it wants to ensure that all previously submitted tasks are completed (blocking call).

3.2 The API

In this section, we describe the main functions in the API of the runtime.

Runtime startup/shutdown functions:

- `runtime_init()`: To be called at the initialization of the application, in order to set up the runtime internals (thread pool, etc.).
- `runtime_finalize()`: To be called before terminating the application. Waits for all tasks to be executed and does some cleanup steps.

Essential functions for task management:

- `task_t* create_task(task_routine_t f)`: Creates a new task object that will execute function `f`.
- `void submit_task(task_t *t)`: Submits a task to be executed. Between the creation and the submission of a task, the programmer can define input and output parameters for the task (see the description of the corresponding primitives in Section 13).
- `void task_waitall(void)`: Blocks the calling thread until all tasks that have been submitted are completed. This function must be called by the main thread of the program (not within a task).

Additional functions: The other functions of the runtime API are described in Section 13.

3.3 Programming with tasks

Note: *During this lab, you will not have to write parallel applications that use tasks (except if you decide to design extra tests). However, it is important that you understand the semantic of the creation and of the execution of tasks, to be able to implement correct parallel versions of the runtime. The examples presented in this section illustrate these points.*

Figure 1 provides a first code example with 3 tasks: task `t1` can execute in parallel with task `t2` whereas task `t3` can only execute when the two previous tasks have terminated (because of the call to `task_waitall()`). Lines 12-15 show an example of creation of a task, initialization of an input parameter, and submission of the task.

The task routine `print_message()` (lines 1-6) has two parameters. The first is a pointer to the task object associated with the execution of the routine. It allows accessing the input and output parameters associated with the task. The second one, called `step`, is only used for tasks with dependencies, as illustrated in Figure 2.

Figure 2 presents a code where a parent task creates a child task. In our library, a task that creates other tasks is executed in multiple steps:

```
1   task_return_value_t print_message(task_t *t, unsigned int step)
2   {
3     char *msg= (char*) retrieve_input(t);
4     printf("%s\n", msg);
5     return TASK_COMPLETED;
6   }
7
8   int main(void)
9   {
10    runtime_init();
11
12    task_t *t1 = create_task(print_message);
13    char *in = attach_input(t1, sizeof(char) * 64);
14    strncpy(in, "hello", 64);
15    submit_task(t1);
16
17    task_t *t2 = create_task(print_message);
18    in = attach_input(&(t2, sizeof(char) * 64);
19    strncpy(in, "hello again", 64);
20    submit_task(t2);
21
22    task_waitall();
23
24    task_t *t3 = create_task(print_message);
25    in = attach_input(t3, sizeof(char) * 64);
26    strncpy(in, "bye", 64);
27    submit_task(t3);
28
29    runtime_finalize();
30
31    return 0;
32  }
```

Figure 1: A simple code with tasks

1. A first block of code is executed, which can create and submit one or several child tasks (lines 43-49), and then, the task routine returns `TASK_TO_BE_RESUMED`.
2. The parent task is put to wait while the child tasks are executed.
3. Once all child tasks have been fully executed, the second block of code of the parent task is executed. (lines 51-54).

The different blocks of a task routine are defined using a `switch` statement, where the parameter `step` of the task routine defines the block to execute.

```
33  task_return_value_t child_routine(task_t *t, unsigned int step)
34  {
35    printf("Hi␣from␣child\n");
36    return TASK_COMPLETED;
37  }
38
39  task_return_value_t parent_routine(task_t *t, unsigned int step)
40  {
41    switch(step){
42
43    case 1:;
44       printf("Hello␣from␣parent␣(before␣child)\n");
45
46       task_t *child = create_task(child_routine);
47       submit_task(child);
48
49       break;
50
51    case 2:;
52       printf("Hello␣from␣parent␣(after␣child)\n");
53
54       return TASK_COMPLETED;
55    }
56
57    return TASK_TO_BE_RESUMED;
58  }
59
60  int main(void)
61  {
62    runtime_init();
63
64    task_t *t = create_task(parent_routine);
65    submit_task(t);
66
67    runtime_finalize();
68    return 0;
69  }
```

Figure 2: A code with task dependencies

4 Provided material

You are provided with a basic skeleton for the runtime. Note, however, that this skeleton only supports sequential execution of the tasks. The skeleton includes a Makefile to compile all the source files. Feel free to create additional source files to structure your code if necessary.

The provided code also includes a few examples of test applications (see Section 11).

The provided source files are listed below. The ones whose name is in **bold font** will have to be modified/extended by you to complete the lab. The other files should not require modifications

(unless you want to introduce extra debugging traces).

- **`Makefile.config`**: main settings for the runtime
- **`tasks_types.h`**: data types and data structures for the management of tasks and their state.
- **`tasks.*`**: implementation of the main functions of the runtime API
- **`tasks_implem.*`**: runtime engine in charge executing the tasks (thread pool, etc.)
- **`tasks_queue.*`**: instantiation of and operation on task queues
- `tasks_io.c`: functions related to the management of the input/output data of the tasks
- `parallel_for.*`: management of parallel `for` loops
- `utils.*`: various utility functions, in particular for the generation of random numbers
- `debug.h`: function used for debug traces
- `tests/*`: See Section 11

5 Stage 0: Getting to know the provided skeleton

This version corresponds to the provided code base, compiled with the `WITH_DEPENDENCIES` flag disabled[5].

This version is fully functional but: (1) it does not support tasks dependencies (due to the above disabled flag) and (2) it does not support the parallel execution of the tasks.

5.1 Running the code

- To compile the runtime and the tests, simply run `make` in the `src` directory.
  - Every time you modify a configuration parameter in the file `Makefile.config`, you should recompile the code.
- To execute a test, simply run: `./tests/name_of_the_test.run`
  - For instance, you can run: `./tests/simple_test.run`

[5]This means that the portions of code delimited by the following blocks are ignored: `#ifdef WITH_DEPENDENCIES ... #endif`

5.2 Digging into the code

To help you start getting familiar with the provided code base, try to answer the following questions (include these answers in your report):

1. The provided version of the runtime uses a (single) queue to keep track of the tasks . Notice that this queue is managed with a LIFO (*last in, first out*) policy.
   (a) When is a task enqueued?
   (b) When is a task dequeued?
2. A given task is associated with a current status. The list of possible statuses is defined in `task_types.h` (`enum task_status`). Based on your observation of the code and your understanding of the runtime, answer the following questions:
   (a) What does the `INIT` status correspond to?
   (b) What does the `READY` status correspond to?
   (c) What does the `RUNNING` status correspond to?
   (d) What does the `TERMINATED` status correspond to?

5.3 About memory management

While browsing and testing the provided code, you will observe that it may cause memory leaks. Namely, the heap memory blocks allocated to store the state of tasks (`task_t`) as well as input/output task parameters (`task_param_t`) are never freed.

This is a choice, which has been made to simplify the code. We will ignore these memory leaks during the first stages of the lab. Managing memory in a proper way will be done in one of the bonus stages (see Section 10).

6 Stage 1: Supporting parallel execution

In this stage, you will have to modify the code in order to introduce support for parallel execution of the tasks. More precisely, the design should be based on the following characteristics:

- It creates a pool of worker threads (configured with a fixed number of threads, see the `THREAD_COUNT` constant defined in `Makefile.config`).
- A single queue of submitted tasks is shared between all the workers. This queue must implement a LIFO policy.
- Each thread repeatedly fetches a new task from the queue and processes it.

Note: Like in the previous stage (Stage 0), dependencies between tasks are not considered in this stage. Hence, the `WITH_DEPENDENCIES` flag must remain disabled.

Before designing and implementing your new version of the runtime, answer the following questions in your report:

1. When should the threads be created?
2. What are the different steps executed by a worker thread?
3. To which famous synchronization problem corresponds the problem to be solved for the shared queue?
4. What should happen if the queue gets full?
5. At what moment can the `task_waitall()` function return?

Remark: For this stage (and the next ones), the definition and usage of the `active_task` variable(s) will deserve some attention. Comments in Section 12.3 may be helpful in this regard.

7 Stage 2: Dealing with dependencies

Important *Do not forget to create a new copy of you code for each stage*

In this stage, you will keep the thread pool principle from the previous stage and you will introduce support for the management of dependencies between tasks.
Therefore, do not forget to set the `WITH_DEPENDENCIES` flag, as well as to read and understand the corresponding blocks in the provided code base.

This new feature will require to handle resizing (growth) of the task queue when this queue becomes full. Besides, like in the previous stage, the task queue will be managed with a LIFO algorithm.

Answer the following questions in your report.

1. Explain why the task queue resizing operation is necessary. Also briefly describe how this operation must be performed. Note that functions such as `memcpy()` and/or `realloc()` (provided by the C standard library) may be useful for this purpose.
2. What does the `WAITING` task status correspond to?
3. What happens after the task routine for a task returns? To answer this question, 4 cases need to be discussed depending whether the task routine returns `TASK_COMPLETED` or `TASK_TO_BE_RESUMED` and whether there can be dependencies between tasks or not.

8 Stage 3: Improved parallelism

The previous design may suffer from a performance limitation stemming from the use of a single task queue. In this stage, you will modify the runtime according to the following principles:

- There is one task queue per thread (instead of a single shared queue).
- Every time a task must be stored in a queue, a decision is made to choose the target queue; this decision is based on a simple "round robin" algorithm (i.e., we feed the *first* queue, then the *second* queue, etc.).

9 Stage 4: Work stealing

An alternative to the design described in Stage 3 is to introduce the following principles. This approach is usually called *"work stealing"*.

- There is one task queue per thread.
- The new tasks created by a given task T_i are stored in the task queue of the thread that currently executes T_i.
- An idle thread (i.e., a thread whose task queue is empty) can attempt to steal a task from another task queue.[6] More precisely, a steal must be performed at the other end of the task queue (i.e., opposite to the side where new tasks are enqueued in LIFO mode).
- The choice of the target queue (i.e., the queue to steal from) can be based on different strategies, such as round robin or random. Choose the strategy that seems the most appropriate and briefly justify your decision.

In your report, also try to provide a short justification for the design characteristics that we have specified for the management of a task queue: LIFO accesses for operations performed by the worker thread that owns the queue and FIFO steals performed by the other workers.

10 Bonus stages

We describe below two optional stages, which can be completed in any order:

10.1 Stage B1: Performance evaluation and analysis

Using some of the provided test programs (and possibly additional programs that you have written), perform an empirical comparison of the designs described in Stages 2, 3 and 4. More precisely, there are several goals:

1. Comparing the achieved performance (execution time) of the designs for a given workload and studying the evolution of the performance when the number of threads increases.
2. Providing an explanation of the observed trends.

Note regarding the "work stealing" design (Stage 4): In order to improve the performance of this design, it may be useful to enhance it with a *backoff* approach: after a failed stealing attempt (empty target queue), an idle thread should wait for a short amount of time before trying to perform a new steal (`usleep()` can be used here). More precisely, the duration of the waiting time should be a (short) randomly generated value. And in case the next stealing attempt also fails, the range of values used for the next random number generation should be increased. And so on. Briefly explain why the enhancement may improve performance.

[6] It may also make sense to steal several tasks at once but we will not consider this approach here.

Note regarding performance evaluation: For a performance evaluation to be meaningful, it is important to run without generating any debug message, and to compile with the maximum level of optimization (add flag `-O3`).

10.2 Stage B2: Memory management

So far, we have explicitly chosen (for simplicity, as discussed in 5.3) to avoid freeing the dynamically allocated data structures that become useless when some tasks are completed. However, this choice is obviously unrealistic in practice, if we need to support long running programs with a large number of tasks. In this new version, you are asked to introduce the required support for properly freeing the `task_t` and `task_param_t` structures (dynamically allocated by the runtime) as soon as the become useless.

In addition to your code, provide a brief description of your solution and the methodology to test it.

Remarks:

- For the tasks directly created from the main function (as opposed to from another task), the outputs of the tasks cannot be freed before the call to the `runtime_finalize()` function.
- Unlike the other stages, this stage requires to introduce modifications in `task_io.c`.

11 Testing the code

11.1 Preliminary remarks

Important comment about tests: Testing multi-threaded applications is difficult. As multi-threading implies non-determinism, successfully running a test does not mean your program is correct. On the other hand, failing to run a test means that your program is not correct (assuming that the test itself is correct).

To have more chances to detect bugs, it can be good to test your code under *extreme configurations*. An extreme configuration can be, for instance, when the number of tasks is very large and/or when the tasks have a very short execution time.

11.2 Overview

You are provided with a set of test programs, briefly described below.

The source code of these programs should not be modified to work with the parallel version(s) of the runtime that you will implement in this lab assignment.

Provided tests:

- **Test without task dependencies** (useful for all stages):

- `simple_test.c`: This test is a slightly modified version of the example shown in Figure 1. It creates several tasks with attached input and also imposes some ordering constraints between them, via `task_waitall()`.
- `simple_test_outputs.c`: This test creates a number of parallel tasks each with an output parameter, and eventually checks that all the produced outputs are consistent.
- `pi.c`: This test computes the value of the *pi* constant, using a numerical method implemented with a "parallel for" loop.[7] The test perfoms some checks regarding the validity/plausibility of the result.

- **Test with task dependencies** (useful for Stages 2, 3, 4, B1, B2):
  - `test_dependencies.c`: This test is a slightly modified version of the example shown in Figure 2. It checks the correct management of a simple parent-child dependency between two tasks.
  - `test_dependencies_outputs.c`: This tests creates a number of parallel child tasks each with an output parameter, and checks the proper completion of the tasks and the validity of their outputs.
  - `fibo.c`: This test performs a computation producing the Fibonacci number for a given input integer. The code uses recursive child tasks, with an input and an output. The test verifies the correctness of the results by comparing it to the result produced by a sequential algorithm.
  - `nqueens.c`: This test computes the solution to the *"N queens"* problem[8] according to the size of the considered chessboard. It uses parallel child tasks with input and output. The test verifies the correctness of the results.

You are of course free to introduce additional tests to validate your code.

11.3 Testing on the servers of the University

To stress your server and run *meaningful* performance tests, it can be good to run tests on servers with a significant number of CPU cores. To this end, you have access to a server of the university: `im2ag-mandelbrot.univ-grenoble-alpes.fr`.

- You can connect to it using `ssh`:

  ```
  ssh LOGIN@im2ag-mandelbrot.univ-grenoble-alpes.fr
  ```

- You can copy files there using `scp`:

  ```
  scp myfile LOGIN@im2ag-mandelbrot.univ-grenoble-alpes.fr:./
  ```

  (use the `-r` option of `scp` to copy a directory)

Note that you may also be able to access machines of the lab rooms of Ensimag. See the following documentation (in French): `https://intranet.ensimag.grenoble-inp.fr/fr/informatique`.

[7] You are not required to understand the mathematical aspects of this algorithm to run this test.

[8] `https://en.wikipedia.org/wiki/Eight_queens_puzzle`

12 Additional technical details

12.1 Debug traces

You are provided with a macro named `PRINT_DEBUG` (see `debug.h`) to control *the level of verbosity*, i.e., the amount of traces produced and displayed by your code. Different levels can be defined via the value passed as a first argument to `PRINT_DEBUG`. The desired level of verbosity to be enabled can be configured via the value assigned to the `VERBOSE` constant in `Makefile.config`. For example, if `VERBOSE` is set to `10`, only the calls to `PRINT_DEBUG` with a value ≤ 10 will be enabled by the compiler (The values 1, 10, 100 and 1000 are used in the provided code). Besides, setting `VERBOSE` to `0` disables all the traces.

12.2 Random number generation

Helper functions to simplify the generation of helper functions are provided in `utils.h` and `utils.c`. Note that `rand_generator_init()` is already called in `runtime_init()`.

12.3 Thread-local variables

The C compiler provides support for simplifying the declaration and usage of per-thread variables. An example is given in the provided code with the `active_task` variable defined in `task.c` as follows: `__thread task_t *active_task;`
The `__thread` keyword means that instead of allocating and using a single global variable, the compiler will allocate and use one distinct copy per thread (and each thread will only be able to access its own copy).

13 Other functions of the runtime API

In Section 3.2, we introduced the most important functions of the runtime API. The remaining ones are described below.

Note regarding the runtime initialization: For applications that rely on task dependencies, the `main` function must call `runtime_init_with_deps` instead of `runtime_init` (the latter is called internally by the former).

Functions to manipulate inputs and outputs of tasks:

- `void* attach_input(task_t *t, size_t s)`: Allocates a new input parameter of size `s` for a task given as parameter and returns the address of this new parameter, for the programmer to initialize it.
- `void* attach_output(task_t *t, size_t s)`: Same as `attach_input()`, but for an output parameter.

- `void* retrieve_input(task_t *t)`: Gets the address of the next input parameter associated with a task.
- `void* retrieve_output(task_t *t)`: Same as `retrieve_input()` but for an output parameter.
- `void* retrieve_output_from_dependencies(task_t *t)`: Gets the address of the output generated by a child task.

Helper function to implement programs based on parallel `for` loops:

- `void parallel_for_with_reduction(task_routine_t f, long int begin_index, long int end_index, long int incr, long int nb_blocks, void *output, size_t output_size, reduce_function_t rf)`: Creates a parallel `for` loop that executes function `f`. The `for` loop iterates from `begin_index` to `end_index` with an increment of `incr`. The `for` loop is divided into `nb_blocks` tasks that can be executed in parallel. The partial results generated by all tasks are aggregated using the `rf` reduction function. The final result, of size `output_size`, is stored at address `output`.