

Assignment 3

Distributed Task Runner with MPI

CS3210 – 2023/24 Semester 1

1 Nov 2023

Learning Outcomes

This assignment allows you to apply your understanding of parallel programming with distributed-memory paradigms (MPI) to design and implement a distributed task runner for a fixed set of tasks on a network of machines (nodes).

1 Problem Scenario

1.1 Overview

A task runner is a program that schedules and executes tasks. In this assignment, you will **design and implement a parallel task runner using OpenMPI**, with the goal of **minimizing the time taken to finish executing tasks** while maximizing the average utilization of your MPI processes.

For simplicity, there are five distinct types of tasks that can be executed (further details in Section 6.1). When a task finishes executing, it may produce a number of further (child) tasks to be executed. Note that the implementation of the tasks are provided for you and you should not modify them; the focus of the assignment is to have parallel execution of the given tasks, and not to parallelize the computation within a task.

The set of initial tasks will be specified in a text file which should be a command line argument to your executable. One process (by convention, the MPI process with rank 0) will proceed to parse the file containing the initial tasks. Your assignment is to implement a way to distribute these tasks, and any subsequent tasks produced, among the MPI processes. Due to the different types of tasks, different distribution of tasks among different nodes can affect the overall performance of the task runner.

During the execution of the task runner system, each MPI process tracks the following execution metrics:

- Time this process was alive
- Time this process spent executing tasks
- Number of each type of task executed to completion by this process

After the execution of all the tasks, each MPI process reports all these metrics to the MPI process with rank 0. This process is responsible for printing the execution metrics of each process *in rank order* (including itself), prior to exiting.

2 Running the Program

You will be provided skeleton code which has a sequential version of the task runner running on the rank 0 MPI process (further detailed in Section 3). Additionally, you will be provided with Slurm scripts to run the program on our lab machines. Your goal in this assignment is to **implement the `run_all_tasks` function within the `runner.cpp` file – you should distribute tasks to your MPI processes and execute each task via the `execute_task` function that we have provided, until no more tasks remain to be executed**. The requirements for speedup, etc, are specified in Section 4.1. **Do not modify parts of the skeleton code which are marked as not to be modified, nor the settings within the Slurm scripts provided to you.** You can make modifications to test your code, but we will use the original Slurm scripts to grade your submission.

The starter code given contains the files specified in Table 1.

File name	Description
tests/tinkywinky.in tests/dipsy.in tests/lala.in tests/po.in tests/thesun.in	Sample input files containing initial tasks and their seeds.
tests/*.out	Reference sequential execution output and execution time.
config1.sh config2.sh config3.sh	Shell files to run your code via Slurm for each of the node configurations in Section 2.2.
main.cpp	Main entry point to the code. Do not edit the parts which are marked as such.
runner_seq.cpp	Sequential implementation of the task runner. Do not edit this code at all.
runner.cpp runner.hpp	These files constitute the main logic of the task runner, and where we expect most or all of your modifications to take place.
tasks.cpp tasks.hpp	Our implementation of the tasks run by the task runner. Do not edit either except the CHILD.DEPTH value for the bonus.
Makefile	The build script. You may edit this and include it in your submission.
generate_test_output.sh	Helper script to generate the reference output files via running our sequential implementation.
job.sh	Common code used by the Slurm job configs. You may edit it in your testing if you need more capabilities (e.g. perf) but please revert it to the original version for submission.
example.sh	Example commands to compile and run the code end-to-end with some of our test cases.
check.py	Python code to check if your implementation produces the same result as the reference sequential implementation. We will run this automatically for you at the end of the Slurm job.

Table 1: Files provided in the starter code.

2.1 Command-line Input

To compile and run your program, **use one of the three provided Slurm scripts** using `sbatch`. These scripts are named `config1.sh`, `config2.sh`, and `config3.sh`. The script takes in arguments in the format `H Nmin Nmax P Finit`, where:

- `H` - maximum depth of tasks generated in the task graph
 - The “depth” of a task is the “distance” along the parent – child chain to any of the initial tasks. For instance, if one of the initial tasks (starting at depth 0) generates a child task on completion, that child task is at a depth of 1, and if that child task generates another task on completion, the grandchild task is at a depth of 2.
- `Nmin` - minimum number of child tasks to generate upon completion of a task
- `Nmax` - maximum number of child tasks to generate upon completion of a task
- `P` - probability of generating each subsequent child task above `Nmin` tasks *to two decimal places*.
- `Finit` - the input file containing the initial tasks as specified in Section 6.2.1



Sample Program Execution

```
$ DEBUG=1 sbatch ./config1.sh 4 1 2 0.10 tests/tinkywinky.in
```

This sample execution reads the initial tasks from the file `tests/tinkywinky.in`. Upon completion of a task, if the depth of that task does not exceed 4, then 1 to 2 child tasks will be generated. The generation of the tasks beyond the minimum (in this case, the potential second child task) will each be generated with probability 0.10.

Note that this behavior (execution of tasks, creation of child tasks) is **already implemented for you!** However these details (and others) may help you optimize your code.



Cancelling your jobs

It is very common to introduce deadlocks in your MPI code. If you do so, your program will hang, you will hog the nodes, and your Slurm priority will decrease. If you notice your job running for longer than the expected time / not producing any output, you will need to cancel your job. To do so, run `scancel -u <your_username>` to cancel all your jobs. You can also cancel a specific job by running `scancel <job_id>`. You can find the job ID by running `squeue -u <your_username>`.

2.2 Configurations

We will test your code using four configurations (configs), three of which are publicly available to you via the provided Slurm scripts, and one is private to us:

- **Config 1** (`config1.sh`): One `i7-7770` node with 4 MPI processes, one `xs-4114` node with 10 MPI processes (total 14 MPI processes). The rank 0 process will be on the `xs-4114` node.
- **Config 2** (`config2.sh`): Two `xs-4114` nodes with 10 MPI processes each.
- **Config 3** (`config3.sh`): Two `i7-7770` nodes with 8 MPI processes each.
- **Config 4**: Secret. Please make sure your code is generalizable enough. This configuration will include a combination of 1-3 nodes (inclusive) in our `soctf` cluster, along with any number of MPI processes.

These configurations describe the number and type of nodes allocated to your job, along with the number of MPI processes allocated to each node. We specify `--map-by core` and `--bind-to core` for all MPI runs (see `job.sh`), and you are not allowed to change these settings.

2.3 Input File

We provide some sample input files (with the `*.in` extension) for testing purposes in the `tests/` subdirectory. You are encouraged to create your own input files as it's trivial to do so – you can simply have any list of initial tasks and seeds. The format of this file is specified in Section 6.2.1. Note that the overall behavior of this system (number and type of tasks generated) is more severely affected by the other parameters, and the input file is only used to specify the initial tasks.

2.4 Expected Output

2.4.1 Execution Metric vs Trace Output

There are two outputs that we expect that your code can generate: execution *metric* output (for performance measurement) and execution *trace* output (for correctness). We already handle the generation of these outputs for you in the starter code, and the formatting of these outputs should not be modified in your submission. For completeness, we provide the specification of these output formats in Section 6.2.2).

To generate only the execution *metric* output, you can either execute the Slurm script as described in Section 2.1 (this is the default) or set the environment variable `DEBUG=0`. To generate the execution *trace* output, set `DEBUG=1`. This also includes the execution metric output for completeness, but those values should not be used as all the debug printing will make your program slower. Additionally, for debugging purposes, you can also print the MPI rank of the process executing a task (and other debug info) by setting `DEBUG=2` although this will not be used for grading.

Example: if you want to run the program and print the execution *trace*, you can execute:

```
DEBUG=1 sbatch ./config1.sh 10 2 5 0.25 tests/tinkywinky.in
```



Do not modify the output formats, as it will be used to grade your submission. If additional text is output, it may cause your submission to be graded incorrectly. If you introduce logging prints for debug purposes, disable or remove them in your final submission.

2.4.2 Reference Output Files

We have pre-generated some *reference output files* (with the `*.out` extension) in the `tests/` subdirectory. These files are provided such that you do not need to re-run the sequential implementation for those cases, which might take a while to finish. These reference output files can be used to check the correctness of your programs. Note that these files have a specific filename format to enable automated testing: if the arguments to the program are e.g., `2 1 1 0.00 tests/tinkywinky.in`, the corresponding reference output file is `tests/tinkywinky_2_1_1_0.00.out`. For your convenience, we already help you check the correctness of your program by comparing the output of your program with the reference output file, if one exists. You can generate the reference output files by running `generate_test_output.sh` with the same arguments as your program (e.g., `./generate_test_output.sh 2 1 1 0.00 tests/tinkywinky.in` in this case).

While this should not be necessary (since `./generate_test_output.sh` already does this) if you would like to run our reference sequential executable directly, simply set `SEQ=1` e.g.,
`SEQ=1 sbatch ./config1.sh 4 1 2 0.10 tests/tinkywinky.in`.

3 Implementation Requirements

We provide a sequential implementation for the task runner in `runner.cpp`, **which is where most of your modifications should be made**. In this sequential implementation, the MPI process with rank 0 will add all the initial tasks to its own task queue (implemented as a C++ vector), and executes them serially by invoking `execute_task`, pushing any new child tasks generated back into its own queue. All other MPI processes block and do not contribute to task execution. The program terminates when all generated tasks are executed to completion, where the execution metrics from all MPI processes are printed.

Note that your task is to implement the `run_all_tasks` function within `runner.cpp`, which should schedule and run (through `execute_task`) the tasks among the MPI processes. You do not need to handle the implementation of the individual tasks themselves nor the generation of the children tasks. The further requirements for speed and correctness are specified in Section 4.1.

Each of your MPI processes (when they want to run a task) **must invoke** `execute_task` in a similar fashion to the invocation in the sequential implementation, i.e. with a reference each to that MPI process' `metrics_t` statistics struct, a `task_t` struct describing the task to execute, an integer to hold the number of new tasks created, and the task queue. `execute_task` will place the generated descendant tasks in the queue, and update the integer to reflect the number of descendant tasks. You are not allowed to change the `execute_task` implementation.

Focus on ensuring your program is correct and free of synchronization issues first before optimising your implementation for speed and utilization. Note that the randomly-generated nature of the task graph can lead to drastically varying performance with a different set of parameters and initial tasks. You may want to explore different approaches to distributing tasks amongst the processes, or implement additional heuristics to improve the efficiency of task scheduling.

3.1 Verifying Correctness

The easiest way to verify the correctness of your program is (if necessary) to generate a reference output file for your parameters as described earlier with `generate_test_output.sh` and then run your program with `DEBUG=1`. Our Slurm job script will automatically check for differences with `check.py` and you will see the output in the Slurm log.

While this should not be necessary, to *manually* verify the correctness of your program, run both the sequential implementation (using `SEQ=1` as mentioned above) and your program (using `DEBUG=1`) with the same parameters for a given input file. You can then remove unrelated lines that are not related to the actual program's output (e.g. Slurm-related output) and compare the log files with `diff`. If the sorted execution traces are identical, your program is correct for that configuration.

As the program makes use of a **deterministic generator**, the same arguments and input file should cause the execution trace obtained from `DEBUG=1` to have the same set of outputs (not necessarily in the same order). In particular, you can check that:

- the execution output of each task is correct
- the correct number of child tasks are generated upon completion of a task
- no tasks are generated with depth exceeding the specified limit
- each child task is generated with the correct seed `arg_seed` (if it has no dependencies), or has its seed correctly composed (if it has dependencies, bonus only)
- all tasks are executed without violating inter-task dependencies (bonus only)

For testing, you are provided multiple input files and reference output files from our sequential implementation. You are advised to produce new test cases for your program, with different arguments.

4 Grading

You are advised to work in groups of two students for this assignment (but you are allowed to work independently as well). You may have a different teammate compared to Assignments 1 and 2. You may discuss the assignment with others but in the case of plagiarism, both parties will be severely penalized. Cite your references or at least mention them in your report (what you referenced, where it came from, how much you referenced, etc.).

The grades are divided as follows:

- 8 marks – the OpenMPI implementation, split into:
 - 4 marks – correctness
 - 4 marks – performance
 - * 2 marks – meeting the minimum performance requirements below
 - * 2 marks – your speed relative to other submissions (likely based on clustering the submissions by performance)
- 5 marks – the report
- Up to 2 bonus marks

4.1 Program Requirements

Your OpenMPI implementation should (in addition to all requirements specified above):

- Be written in C or C++
- Produce an executable called `distr-sched` when we run `make` in the root directory of your repository – we will evaluate this executable. Note that this is already implemented for you.
- Run **at least 3x faster** than the sequential implementation *under specific conditions*.
 - **Runtime:** The official runtime of your program is printed in the last line of the *execution metric* output (see Section 6.2.2).
 - **Sequential Runtime:** The reference sequential runtime should be generated via `generate_test_output.sh` (i.e., 1 task on 1 xw-2245 node, while setting `SEQ=1`). We have already generated the reference sequential outputs with times for our given test cases, so unless you use other test cases, you won't have to run `generate_test_output.sh`.
 - **Minimum requirements to enforce speedup:** not all test cases and arguments can generate enough parallelism for speedup! Therefore we don't require 3x speedup for every possible combination of input files and arguments. Your program should achieve this 3x speedup **across all 4 configs** (see Section 2.2) for all of these possible arguments:
 - * 16 1 2 0.10 tests/tinkywinky.in
 - * 5 3 5 0.50 tests/dipsy.in
 - * 5 2 2 0.00 tests/lala.in
 - * 5 2 4 0.50 tests/po.in
 - * 12 0 10 0.16 tests/thesun.in

- We will automatically calculate the speedup if your code passes the correctness checks.
 - For convenience, we include all the calls to `sbatch` you might need in `example.sh`.
 - We reserve the right to change the value of the seeds in each input file to ensure that you are truly computing the result in each run
 - We can also include *similar-but-not-identical* test cases to these in our own grading to ensure you did not overly optimize for these specific scenarios.
- Only use MPI for parallel execution. Do not use any other parallelization constructs and paradigms like OpenMP and threading.
 - Do not use any external libraries without prior approval from the teaching team. Please email us if necessary (contact info in Section 5.1).

In your MPI implementation, you may assume that the delivery of messages between nodes will not be affected by network issues.

4.2 Report

Your report should contain:

- (1 mark) An outline and brief explanation of your implementation of the distributed task runner - include all assumptions, as well as any non-trivial implementation details.
- (2 marks) Performance measurement:
 1. A description of your measurement methodology, e.g. input parameters, sampling and data processing
 2. Reported execution metrics (average utilization, time taken) for your distributed implementation
 3. An analysis of the results, including your explanation for noteworthy observations

Include graphs to visualize your data for this section and link(s) to your supporting measurements (for example, a .csv file in the repository or a Google sheet). You should demonstrate how your distributed implementation scales with an increasing number of MPI processes for **at least two** of the following three parameters:

- Different configurations and numbers of each type of lab machine
- Different input task graph parameters
- Different sets of initial seed tasks (including the provided set of input files)

For your measurements, you are strongly encouraged to use node configurations beyond those specified by the teaching team. However, please **do not change the Slurm scripts for the default configurations** - instead, make additional Slurm scripts for your own testing.

- (2 mark) A description of **at least two** modifications attempted, with analysis and supporting performance measurements. These modifications should be made with the objective of increasing throughput as well as average utilization across nodes, even if they do not result in a direct speedup to your program. Include at least one graph for each modification and link to your supporting measurement data.

Your report should follow these specifications:

- Four pages maximum for main content (excluding appendix).

- All text in your report should be no smaller than 11-point Arial (any typeface and size is OK so long as it's readable English and not trying to bypass the page limit).
- All page margins (top, bottom, left, right) should be at least 1 inch (2.54cm).
- Have visually distinct headers for each content item above.
- It should be self-contained. If you write part of your report somewhere else and reference that in your submitted "report", we reserve the right to ignore any content outside the submitted document. An exception is referencing a document containing measurement data that you created as part of the assignment - we encourage you to do this.
- If headers, spacing or diagrams cause your report to *slightly* exceed the page limit, that's OK - we prefer well-organized, easily readable reports.

4.3 Bonus

You may obtain up to 2 bonus marks in total for this assignment. There are two ways to do so below.

4.3.1 Speed Contest

Similar to the first assignment, you may obtain up to 2 bonus marks by participating in the speed contest. For more details, please refer [here](#).

4.3.2 Complex Task Graphs

You may obtain up to 2 bonus marks by extending your implementation to handle the scheduling of a complex task graph (i.e., tasks now have dependencies that prohibit them from executing unless those tasks run first). For more details, please refer [here](#).

5 Admin

5.1 FAQ

Frequently asked questions (FAQ) received from students for this assignment will be answered [here](#). The most recent questions will be added at the beginning of the file, preceded by the date label. **Check this file before asking your questions.**

If there are any questions regarding the assignment, please use the Discussion Section on Canvas or email Theodore (theodoreleebrant@u.nus.edu).

5.2 Running OpenMPI on the Lab Machines

5.2.1 Lab Machines for Performance Testing

As described by the config files included in the starter code, we will be using our `soctf` lab machines for performance testing and final evaluation. The link to the documentation on how to use the lab's cluster can be accessed at <https://nus-cs3210.github.io>. Refer to Lab 4 and 5 for more details on how to run OpenMPI on the lab machines.

5.2.2 SoC Compute Cluster for Development and Debugging

We recognize that there might be long queues to run jobs on the lab machines during this assignment (at its peak). Unlike Assignment 1, each task runs for much longer, and also requires 2 nodes per job.

Therefore, if necessary, we recommend that you use the SoC Compute Cluster for development and debugging the correctness of your program. The link to the documentation on how to use the SoC Compute Cluster for this assignment is [here](#). **Do take note that you should still use the soctf cluster to take performance measurements for your report.**

5.3 Deadline and Submission

Assignment submission is due on **Fri, 17 Nov, 2pm**.

- **Canvas Quiz Assignment 3:** Take the quiz and provide the name of your repository and the commit hash corresponding to the a3-submission tag; if you are working in a team, only one team member needs to submit the quiz. If both of you submit, we will take the latest submission.
- **GitHub Classroom:** The implementation and report must be submitted through your GitHub Classroom repository. The GitHub Classroom for Assignment 3 can be accessed through <https://classroom.github.com/a/XdTltZ0m>.

The GitHub Classroom submission must adhere to the following guidelines:

- Name your team a3-e0123456 (if you work by yourself) or a3-e0123456-e0173456 (if you work with another student) – substitute your NUSNET number accordingly.
- Push your **code and report** to your team's GitHub Classroom repository.
- Your report must be a PDF named <teamname>.pdf. For example, a3-e0123456-e0173456.pdf. <teamname> should **exactly match** your team's name; if you are working in a pair, please DO NOT flip the order of your NUSNET numbers.
- Tag the commit that you want us to grade with a3-submission; if no such tag exists, we will use the most recent commit.

A penalty of 5% per day (out of your grade for this assignment) will be applied for late submissions.



Final check: Ensure that you have submitted to both **Canvas** and **GitHub Classroom**.

Canvas should contain your commit hash and repository name, and GitHub Classroom should contain your code and report.

6 Additional Details

The contents of this section is provided for your reference only – these specifies the details of the task runner which should not be modified for your submission.

6.1 Task Specification

6.1.1 The `task_t` Type

Each task is represented as a `task_t` struct containing the following attributes:

- `id`: the 32-bit unsigned integer ID of this task; set to `arg_seed` if the task has no dependencies
- `gen`: the generation (depth in the task graph) of this task
- `type`: an integer from $\{1, 2, 3, 4, 5\}$ denoting the task type
- `arg_seed`: the 32-bit unsigned integer seed for the arguments of this task
- `output`: the 32-bit unsigned integer output of this task
- `num_dependencies`: the number of dependent tasks (up to 4) which must execute to completion before this task (bonus only)
- `dependencies`: an array of (up to 4) 32-bit unsigned integers, each the ID of another task whose execution output is used to compose the seed of this task (bonus only)
- `masks`: an array containing the same number of 32-bit bitmasks used to compose the seed of this task (bonus only)

6.1.2 Task Types

There are five types of tasks:

1. **PRIMES** `<num>`: counts the number of primes up to and including `num` with the square-root method
2. **MATMULT** `<m>` `<n>` `<p>`: multiplies a $m \times n$ matrix by a $n \times p$ matrix to a $m \times p$ result matrix
3. **LCS** `<alph>` `<len1>` `<len2>`: computes the longest common subsequence of two strings of length `len1` and `len2` from an alphabet containing `alph` symbols
4. **SHA** `<str>`: computes the SHA-256 message digest of the string `str`
5. **BITONIC** `<n>`: sorts an array of 2^n integers from $[1, 2^{32} - 1]$ using bitonic sort

The arguments of these tasks are generated pseudo-randomly through the `arg_seed` of the task. The execution of the task would determine the output of the task. This output will be used for the check for correctness, as well as used by descendant tasks to compose their own argument seed.

6.2 Additional IO Details

The starter code already adheres to the following I/O specification and you should not modify it for your submission. The following is just for your reference.

6.2.1 Input: Initial Tasks

The input file contains on the first line the number of initial tasks, followed by 1 line for each task in the initial set of tasks. Each line represents a task and contains the following information, separated by space:

- `t` – Integer denoting the task type
- `S` – 32-bit unsigned integer seed for the arguments of this task



Sample Input

```
3
1 3210
3 57005
5 48779
```

The above input contains three tasks - the first is a **PRIMES** task with seed 3210, the second is a **LCS** task with seed 121418, and the third is a **BITONIC** task with seed 314159.

6.2.2 Output 1: Execution Metrics

This output is obtained by setting the `#define DEBUG` macro to 0 (or more easily, setting the environment variable `DEBUG=0`, or not setting it as all as this behavior is the default).

The execution metric output (to standard output) contains 1 line for each MPI process in the program execution. Each line contains the following information of a process:

- R - 32-bit integer; the rank of this MPI process
- T_{ex} - 64-bit integer; total time this MPI process spent executing tasks (in milliseconds)
- T_{all} - 64-bit integer; total time this MPI process was alive (in milliseconds)
- f - 32-bit floating-point number; fraction of time spent executing tasks by this MPI process
- C_i - five 32-bit integers separated by space; number of tasks of type $i \in [1, 5]$ executed to completion by this MPI process

in the format of "Rank R : T_{ex} ms of T_{all} ms (f) - completed: $C_1 C_2 C_3 C_4 C_5$ ".

There are also two summary lines printed out after the per-rank information:

1. A line that adds all the times and task counts, and averages the utilization, across all MPI processes. The format of this line is the same as above, except that the prefix is "Overall".
2. A line that prints out the final execution time of your program (to be precise: the total execution time of rank 0). The prefix of this line is "FINAL RUNTIME:".

Shown below is an example of the output produced by the starter code.



Sample Output

```
Rank 0: 19797ms of 19801ms (0.99975) - completed: 0 1 3 1 4
Rank 1: 0ms of 19801ms (0.00000) - completed: 0 0 0 0 0
Rank 2: 0ms of 19801ms (0.00000) - completed: 0 0 0 0 0
Rank 3: 0ms of 19801ms (0.00000) - completed: 0 0 0 0 0
Overall: 19797ms of 79204ms (0.24994) - completed: 0 1 3 1 4
FINAL RUNTIME: 19801ms
```

Note that the execution time you are trying to minimize is the runtime printed in the last line, for e.g., the **FINAL RUNTIME: 19801ms** here.

6.2.3 Output 2: Execution Trace

This output is obtained by setting the `#define DEBUG` macro to 1 (or more easily, setting the environment variable `DEBUG=1`).

The execution trace output contains 1 line for each task executed. Each line contains the following information of a task execution:

- *id*, corresponding to `id` in `task_t` (Sec.6.1.1)
- *G*, corresponding to `gen` in `task_t` (Sec.6.1.1)
- *T*, corresponding to `type` in `task_t` (Sec.6.1.1)
- *S*, corresponding to `arg_seed` in `task_t` (Sec.6.1.1)
- *O*, corresponding to `output` in `task_t` (Sec.6.1.1)
- *D*, the number of descendant tasks generated by this task

in the format of "EXECUTION TRACE: Task #*id*: depth *G*, type *T*, seed *S*, output *O* (*D* descendants)".

Below is a sample execution trace with `Nmin` and `Nmax` both set to 0 (i.e. no descendant tasks).



Sample Execution Trace

```
EXECUTION TRACE: Task #3210:  depth 0, type 1, seed 3210, output 1001896  
(0 descendants)
```

```
EXECUTION TRACE: Task #121418:  depth 0, type 3, seed 121418, output 18863  
(0 descendants)
```

```
EXECUTION TRACE: Task #314159:  depth 0, type 5, seed 314159, output  
2115765436 (0 descendants)
```