Assignment 3
Distributed Task Scheduling with MPI

CS3210 – 2020/21 Semester 1

28 Oct 2020

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 scheduling library for a fixed set of tasks on a network of machines (nodes).

1 Problem Scenario

The primary focus of CS3210 thus far has been on shared-memory systems, with Assignment 1 exploring various frameworks for shared-memory programming. However, shared-memory systems scale poorly with an increasing number of processors, due to increased contention, increased latency and memory bandwidth limitations. Hence, large computing systems (e.g. datacenters or supercomputers) employ a distributed memory or distributed-shared memory organisation, with reliance on a fast network of interconnects for inter-processor communication and memory access.

In this assignment, you will parallelize the scheduling and execution of a set of tasks over different processes executing on multiple lab nodes, with the goal to **maximise both the task throughput and average utilisation across all your MPI processes**. The set of tasks are seeded from an initial pool and are dynamically updated during execution to include new tasks generated when tasks are completed. For simplicity, there are only five distinct types of tasks, which are detailed in Section 1.2.

1.1 Requirements

In your distributed task scheduling library, each MPI process simulates the role of a worker thread, with no explicitly designated master thread. All worker threads are allowed to (i) schedule (issue) tasks to other threads and (ii) execute tasks assigned to them.

Your scheduling program will be launched with `mpirun` with a `hostfile` (`--hostfile` flag), which lists the hostnames of the lab machines the MPI processes will be launched on.

By default, MPI assigns one MPI process to each slot on a given node in the `hostfile` before proceeding to the next node in the `hostfile`. Therefore, if your program is launched with fewer MPI processes (`-np` flag) than the total number of slots across all nodes in the `hostfile`, some of the nodes may not be utilised. You can control the process rank-processor mapping and binding by providing a `--rankfile` when launching your program.

The task generation parameters are supplied as command-line arguments, whilst the initial set of seed tasks are read in from `stdin`. **As a convention, this will be done by the MPI process with rank `0`.**

The MPI processes of your program should proceed to distribute the tasks amongst themselves for execution. Upon completion of a task, a pre-specified routine is invoked to dynamically generate new child tasks, each with its own arguments and without any dependent tasks. **To increase overall throughput of the system, you should continue to distribute these new child tasks.**

Each MPI process tracks the following execution metrics:

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

At the end of execution, 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.**

1.2 Types of Tasks

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)

There are five type of tasks, each with their own `type` integer (index of the task as shown below) and arguments. The arguments of each task are generated pseudo-randomly using the `arg_seed` of the task. The task is then computed to its result, which determines the (arbitrary) 32-bit unsigned task `output`. This `output` will be checked for correctness and is used by dependent descendant tasks to compose their own argument seed.

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

Do not modify any sub-routines that are invoked as tasks for execution (however inefficient they may appear) or invoked for task generation (specifically `execute_task` and `generate_desc_tasks`).

1.3 Inputs and Outputs

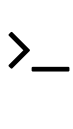
This section is provided for your reference only - the provided starter code already adheres to this input and output format and **you should not modify it for your submission**.

The program accepts four command-line arguments, which in order are:

- `H` - maximum depth of tasks generated in the task graph
- `Nmin` - minimum number of child tasks to generate upon completion of a task
- `Nmax` - maximum number of child tasks to generate upon completiion of a task
- `P` - probability of generating each child task above the minimum number

The program will be executed with `mpirun` with `N` processes launched on the hosts listed in the `hostfile` as `mpirun -np <N> --hostfile <hostfile> ./distr-sched <H> <Nmin> <Nmax> <P> < tasks.in`.

Failing to execute the program with exactly `<N>` processes will result in deductions.



Sample Program Execution

```
$ mpirun -np 80 --hostfile xeons ./distr-sched 10 1 3 0.25 < tasks.in
```

The above executes the program with 80 MPI processes using the nodes in `xeons` hostfile, with the initial tasks read from `tasks.in` and 1 to 3 child tasks generated upon completion of a task (if the child task would have a depth not exceeding 10). Child tasks above the minimum number are generated with probability $1/4$.

The input file `tasks.in` contains on the first line the number of initial seed tasks, followed by 1 line for each seed 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 121418
5 314159
```

The above input contains three tasks - one each of **PRIMES**, **LCS** and **BITONIC**.

The 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

Shown below is the format 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
```

Do not modify the above output format, 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.

1.4 Starter Code

We provide a skeleton implementation for the distributed task scheduling program, which will launch multiple MPI processes only on the current node. Here, the MPI process with rank 0 reads and parses the command-line arguments and input file containing the initial tasks.

This same process then adds all the initial tasks to its own task queue (implemented as a linked list), and executes them serially by invoking `execute_task`, pushing any new child tasks generated back into its own queue. All other tasks 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.

Your submission should invoke `execute_task` in a similar fashion to the invocation in the skeleton implementation, i.e. with a pointer to each of: the `task_t` struct, that MPI process' `metrics_t` struct, an integer and a `task_t` buffer. `execute_task` will place the generated descendant tasks in the buffer, and update the integer to reflect the number of descendant tasks.

The program can also be run in `DEBUG` mode by modifying the value of the `#define DEBUG` macro to `1`. **In `DEBUG` mode, no execution metrics are printed - instead, an execution trace is printed, containing the details of each task executed**. You can optionally also print the MPI rank of the process executing a task by setting the value of the macro above to 2 (this rank is not used for checking correctness).

Shown below is a sample execution trace with `#define DEBUG 1` for the above sample input, where `Nmin` and `Nmax` are both set to 0 (so no descendant tasks are ever generated).

Sample Execution Trace

```
Task #3210: depth 0, type 1, seed 3210, output 1001896 (0 descendants)
Task #121418: depth 0, type 3, seed 121418, output 18863 (0 descendants)
Task #314159: depth 0, type 5, seed 314159, output 2115765436 (0
descendants)
```

As the program makes use of a deterministic generator, you should use `DEBUG` mode to ensure the correctness of your program - namely:

- 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)

To verify the correctness of your program, run both the skeleton implementation and your program with the same parameters for a given input file, and redirect `stdout` to a file. This will produce two execution traces, which you should then sort and compare with `diff`. If the sorted execution traces are identical, your program is correct for that configuration.

For testing, you are provided four input files (which you can redirect `stdin` to). To generate the corresponding `DEBUG` mode execution traces, run the skeleton implementation on that with the specified parameters. Due to the non-deterministic nature of distributed scheduling and I/O, we do not provide execution metrics nor the detailed `#define DEBUG 2` execution traces (which print the rank of each MPI process executing a task). You are advised to produce new test cases for your program.

Your distributed implementation should be correct, and execute faster on the lab machines.

1.5 Optimizing your Solution

Focus on ensuring your program is correct and free of synchronisation issues first before optimising your implementation for process utilisation or throughput. 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.

In your submission, you are advised to retain your alternative implementations (even if they are slower or not correct), and explain the changes you have made in each.

Distinguish any alternative implementations you include in your submission clearly from the final distributed implementation to be graded.

For those who seek a challenge, you may obtain **bonus marks** by extending your distributed task scheduling program to handle more complex task graphs generated by the `generate_desc_tasks` function. For more details, see Section 2.2.

2 Admin Issues

2.1 Running your Programs

For performance measurements, run your program at least 3 times and take the shortest execution time. You should only use the machines in the lab for your measurements:

- `soctf-pdc-001` - `soctf-pdc-008`: (Xeon Silver 4114)
- `soctf-pdc-009` - `soctf-pdc-016`: (Intel Core i7-7700K)
- `soctf-pdc-018` - `soctf-pdc-019`: (Dual-socket Xeon Silver 4114)
- `soctf-pdc-020` - `soctf-pdc-021`: (Intel Core i7-9700)

You should demonstrate how your distributed implementation scales with an increasing number of MPI processes for:

- 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)

To analyse the improvements in performance, compare the average process utilisation and task throughput of your final implementation against the skeleton implementation and other prototype implementations you have tried. You should select parameters and inputs that have meaningful execution times when solved by your final implementation.

If you are computing the speedup of your distributed implementation, compute it using the provided skeleton implementation as the baseline.

2.2 Bonus

If you are implementing any extension, distinguish it from your final distributed implementation by organising the extended implementations into a separate sub-folder. In addition to the extensions listed on the next page, you may propose (via email) other extensions for consideration for bonus credit. **This should be done prior to the assignment deadline - any unacknowledged extensions will not be graded.**

You may obtain up to 3 bonus marks by extending your implementation to handle the following:

- **Schedule a complex task graphs - up to 3 bonus marks**: Modify the value of the `#define CHILD_DEPTH` macro from 1 to 4 - this allows the `generate_child_tasks` function to generate descendant tasks with depth up to 4 greater than the depth of the task that just completed execution
  - Some descendant tasks will possess dependencies and thus will not be generated with an `arg_seed` - instead, you should compose it as the logical OR of the results of applying the corresponding bitmask (logical AND) to the `output` of each dependent task
    - Suppose you have a task T dependent on two tasks T_1, T_2, with corresponding outputs R_1, R_2 and bitmasks B_1, B_2 - then the `arg_seed` of T is `((R1 & B1) | (R2 & B2))`
  - In other words, the function will now generate a directed acyclic graph (DAG) of depth up to 4 rooted at the current task, with each vertex representing a descendant task

2.3 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 post on the LumiNUS forum or email Keven (keven@comp.nus.edu.sg) or Richard (e0191783@u.nus.edu).

3 Submission Instructions

You are allowed to work in groups of maximum two students for this assignment, and can discuss the assignment with others as necessary. **However, in the case of plagiarism (from other students or online sources), all parties involved will be severely penalised.**

The deadline for this assignment is **Mon, 16 November at 11am**.

3.1 Report

Your report should include:

- An outline and brief explanation of your distributed task scheduling program - include all assumptions, as well as any non-trivial implementation details
- A description of your measurement methodology, e.g. input parameters, sampling and data processing
- Execution time measurements and reported execution metrics for your distributed implementation - you may present your data as graphs if you feel they provide better insight
- An analysis of the results, including your explanation for noteworthy observations
- **A separate section for each modification investigated or extension implemented**, detailing the changes and their impact on the overall performance and execution metrics

There is no minimum or maximum page length for the report. Be **comprehensive**, yet **concise**.

3.2 Deliverables

This assignment is worth 14% of your CS3210 grade (about 10% implementation and 4% report). Late submissions are accepted with a penalty of 5% of your assignment grade per day.

Submit your assignment to the folder **Assignment 3** under **LumiNUS Files** by **Mon, 16 Nov at 11am**. Submit **one ZIP archive per student** with your **student number** (A0123456Z.zip - if you worked by yourself, or A0123456Z_A0173456T.zip - if you worked with another student) containing:

1. Your C/C++ code, including the modified `distr_sched.c` and any new header or C/C++ files you have added
2. **Outputs for each of the four (4) provided configurations (input files and parameters).**
3. **Three (3) additional sample configurations (input files and parameters) with their corresponding outputs.**
4. `README` file, minimally with instructions on how to compile and execute the code. You may substitute the `README` with a `Makefile` or a collection of scripts.
5. Report in PDF format (a3_report.pdf)

Note that for group submissions, only the **most recent submission** (from any group member) will be graded, and both students receive the same grade.