Ng Choon Siong (A0196600H)

CS3210 Parallel Computing Assignment 3

Implementation Explanation

The distributed task runner is designed to efficiently distribute and execute tasks across multiple processors using the Message Passing Interface (MPI). The architecture is divided into a master process and several worker processes, with the master handling task distribution and the workers executing the tasks. The master process is used exclusively for reading the input tasks from I/O and distributing them to the worker processes, and for load balancing and exit management. The worker processes execute task, sending excess tasks to the master for forwarding, and signaling readiness to receive tasks to the master when it has no task to execute.

Key Components and Implementation Details

Task Distribution: Initially, the master process reads tasks from an input file and distributes them among worker processes. This initial distribution is done in a round-robin fashion.

Dynamic Load Balancing: After distributing the initial set of tasks, the master acts as a load balancer by maintaining a queue of tasks. When a worker finishes a task and is ready for more work, it sends a ready signal to the master. The master then redistributes tasks from its queue to ready workers, ensuring a balanced workload among all processes.

Task Execution and Generation: Workers execute tasks and may generate new tasks (child tasks). These child tasks can be added to the worker's local queue or sent back to the master, depending on the worker's current load. This design allows for dynamic creation of tasks.

Completion and Termination: The system uses a special 'exit task' to signal workers to terminate. The master process determines when all tasks are completed and all workers have signaled readiness and sends this exit signal to all workers.

Performance measurement methodology

Independent Variables

The independent variables that were changed are the following:

1. Computer Architecture: The different sets of configuration files (config1.sh, config2.sh, config3.sh) were used to run the code across different architectures to test for correctness. Additional config files numbered five to eight were also created.
    a. 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.
    b. Config 2 (config2.sh): Two xs-4114 nodes with 10 MPI processes each.
    c. Config 3 (config3.sh): Two i7-7770 nodes with 8 MPI processes each.
    d. Config 5: Three xs-4114 nodes with 10 MPI processes each.
    e. Config 6: Four xs-4114 nodes with 10 MPI processes each.
    f. Config 8: Two xs-4114 nodes with 5 MPI processes each.
2. Input Files: The provided input files `dipsy.in` (1 initial parent task) and `lala.in` (5 initial parent tasks) were used.
3. Task parameters: Different sets of parameters were used to investigate the effects of changing the maximum task depth H, maximum number of child tasks N_{max} and child task spawning probability P.

Dependent Variables

The dependent variables are the runtime, normalized speed v (see calculation below) and average worker task utilization. As the master process was only used for task distribution and load balancing, its utilization is 0. Hence, I only used the average worker task utilization.

Average child task count $\mathbb{E}(N)$

For the given parameters, the effective child task count was also calculated. Let $N =$ number of child tasks generated and $S =$ total number of tasks. For a configuration where $N_{min} = n_{min}, N_{max} = n_{max}, P = p$, $N - n_{min} \sim B(n_{max} - n_{min}, p)$ for $N \in [n_{min}, n_{max}\,] \cap \mathbb{Z}$. The effective number of child tasks is therefore $\mathbb{E}(N) = \mathbb{E}(n_{min}) + \mathbb{E}(N - n_{min})$ by linearity of expectation, which simplifies to $n_{min} + p(n_{max} - n_{min}) = p \cdot n_{max} + (1 - p) \cdot n_{min}$. I used this to calculate the effective child task count $\mathbb{E}(N)$.

Average total number of tasks $\mathbb{E}(S)$ **and speed** v

Then, to calculate the average total number of tasks $\mathbb{E}(S)$, let us consider the total number of tasks at depth i, S_i. Consider the expected number of tasks at depth $h - 1$ $\mathbb{E}(S_{h-1})$. For each additional depth, the expected number of tasks is the expected number of tasks from the previous level multiplied by the expected number of child task generated per task $E(N)$. Thus, we have the following recurrence relation:

$$\mathbb{E}(S_h) = \mathbb{E}(S_{h-1}) + \mathbb{E}(S_{h-1}) \cdot \mathbb{E}(N) = \mathbb{E}(S_{h-1})[\mathbb{E}(N) + 1]$$

This is a geometric progression which simplifies to $\mathbb{E}(S_h) = (N + 1)^h + \mathbb{E}(S_0)$. Let $H = h$. Then, $\mathbb{E}(S) = \mathbb{E}(S_h) = (N + 1)^h + \mathbb{E}(S_0)$. The average total number of tasks was then used to approximate the input and parameter normalized speed of the parallel execution $v = \frac{\mathbb{E}(S)}{\text{time taken}}$. As different input files and many different parameters were used, this speed was used to provide a common basis of comparison.

Experiments

Two experiments were performed. In each experiment, some independent variables are changed to measure their effect on all dependent variables.

- Experiment 1: Config files 2, 5, and 6 were used to vary the number of processes and number of nodes. The maximum task depth H and the probability of generating child tasks P were varied as well. All runs used the `dipsy.in` input file.
- Experiment 2: Config files 2, and 8 were used to vary the number of processes. The input files `lala.in` and `dipsy.in` were used.

Results

Experiment 1: Effect of task depth H and the probability of generating child tasks P

Findings (see Figure 1 below)

- As the number of processes increase, on average, the runtime decreases, the speed increases, and the worker utilization decreases. These are all expected, as more processes mean more resources and faster execution. At the same time, there won't be enough tasks for all processors to execute in parallel, leading to a lower worker utilization.
- As the child task probability P increases, the runtime, the speed and the worker utilization all increase. These are all expected as having more child tasks overcomes the issue of having more processors than available tasks, increasing utilization and speed. However, while the speed increases as P increases from 0.25 to 0.75, but falls slightly as P further increases to 1. This is likely due to the limit of parallelization being reached, with increasing overheads due to dynamic load balancing being employed to redistribute extra tasks.
- As the task depth H increases, the number of tasks available increases exponentially[1]. This means more available tasks which lead to the same effects as increasing the child task probability on the runtime, the speed, and the worker utilization for similar reasons.

[1] $\mathbb{E}(S_h) = (N + 1)^h + \mathbb{E}(S_0). \in \Theta\left(N^h\right)$ (based on the calculations earlier)

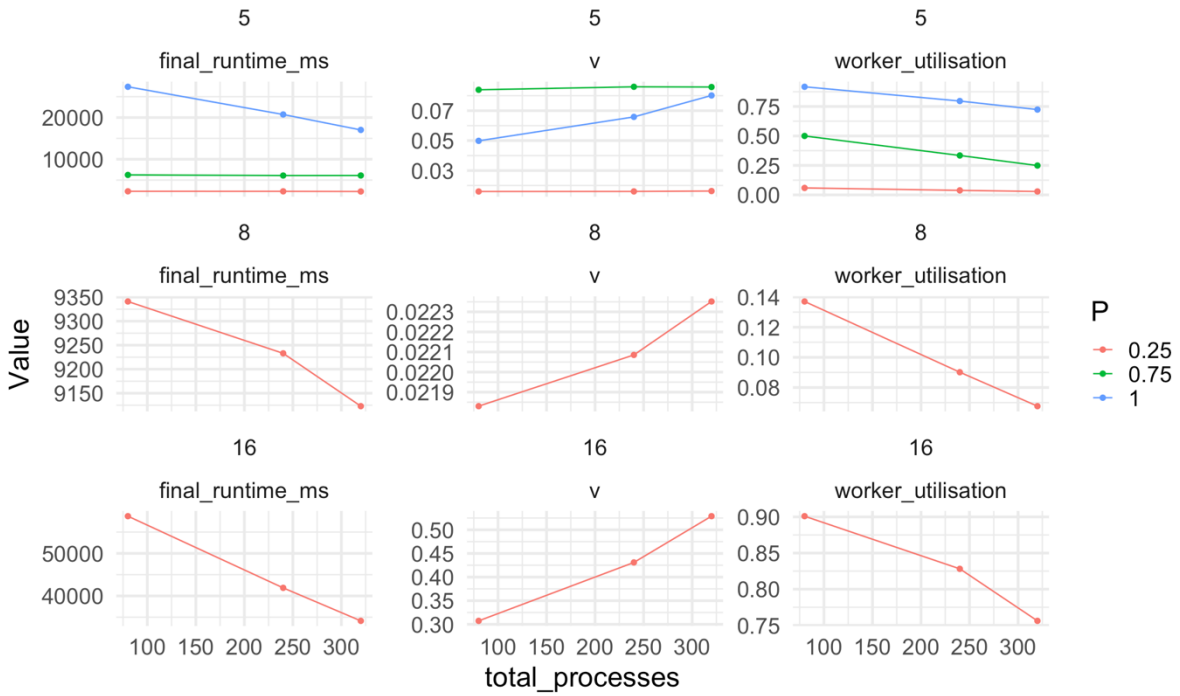

Figure 1 Effect of varying task depth H, child task probability P and number of processes on the runtime, speed and worker utilization. The task depths H are 5, 8 and 16. All runs are on dipsy.in with 1 to 4 tasks.

Experiment 2: Effect of task depth H and the number of initial tasks (different input files)

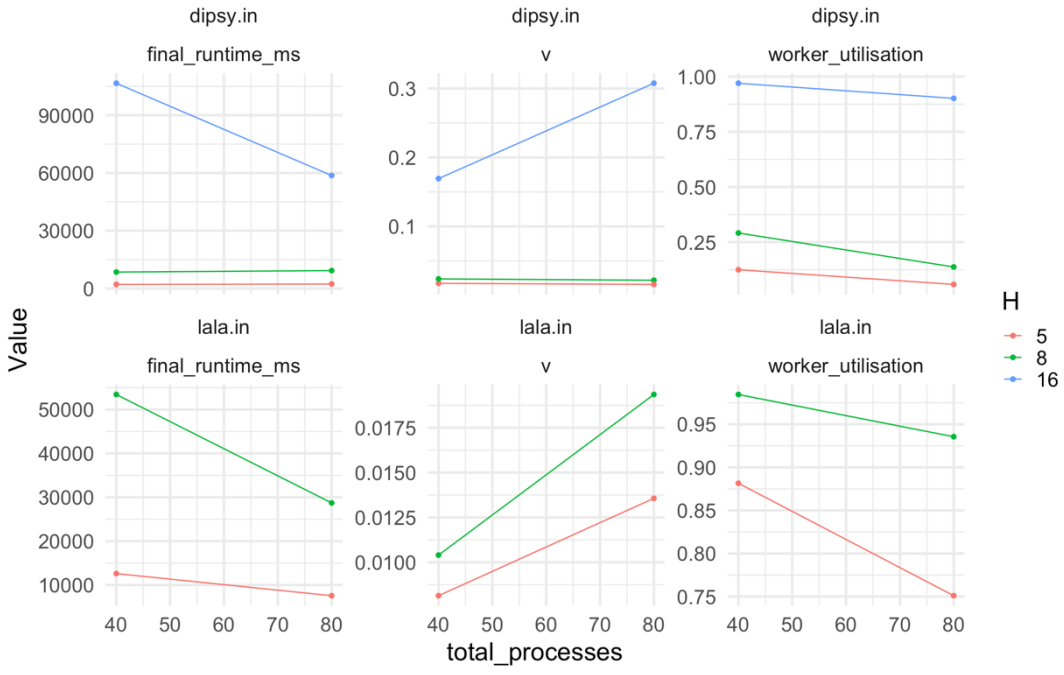

Figure 2 Effect of varying task depth H, and number of processes, and the input file on the runtime, speed and worker utilization. The task depths H are 5, 8 and 16. The file dipsy.in has 1 initial parent task and the file lala.in has 5 initial parent tasks.

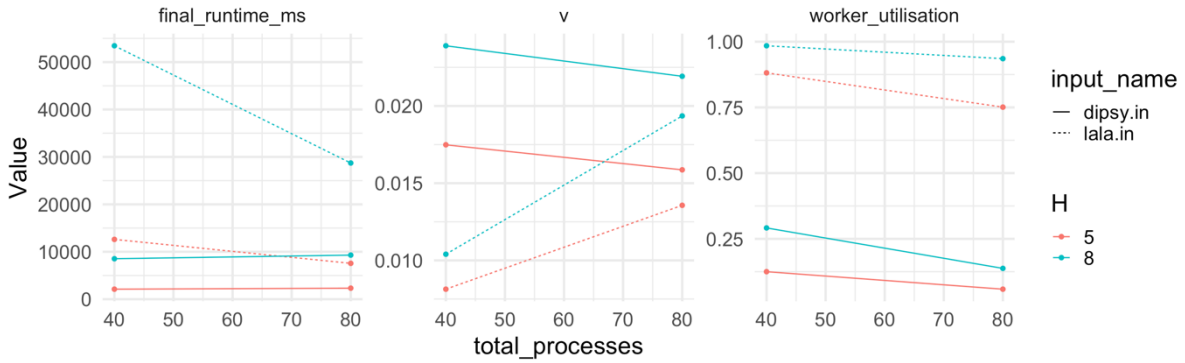

Figure 3 Effect of the same independent variables on the same dependent variables. The task depths are now 5 and 8 with different input files shown on the same graph.

Findings

- Increasing the number of initial tasks (from 1 in `dipsy.in` to 5 in `lala.in`) slowed down the execution but increased worker utilization (Figure 3). The increased worker utilization was expected due to the better parallelization from having multiple task but the reduced speed was unexpected. This could be because the additional tasks hit the limit on parallelization while leading to increased overheads from load balancing and overshadowing the improved worker utilization. Tellingly, the speed gap between the different input files also shrank as the number of processes increase, suggesting that the processor contention was relieved by increasing the number of processes.
- Like before, increasing the task depth H lead to an increase in all the runtime, the speed and the worker utilization (Figure 2) due to better use of parallelization.

Modifications

In my final code, the master performed task distribution and load balancing exclusively. This means the master's task utilization is zero. I tried to improve the performance further by allowing the master to execute tasks when there are tasks available but no worker processor available. Unexpectedly, this led to the task utilization by the master skyrocketing to 95+% while drastically reducing worker utilization. This was likely due to the master spending too much time executing tasks that it neglected its load balancing duties after it starts executing a task.

To attempt to prevent the master from abandoning its load balancing duties, I incorporated an execution limit policy in another modification. In addition to only allowing the master to execute a task only when there are tasks and no workers available, I added an extra condition where the workers have to collective execute a total number of redistributed tasks equal to the number of processes before the master can execute a task instead of redistributing it. This helped somewhat but was still significantly slower. I did not incorporate both modifications ultimately as the execution slowed significantly afterwards. Some measurements are given below.

Config1.sh, $N_{min} = 1, N_{max} = 4$, `tinkywinky.in`					
Version	**Parameters**	**Master Utilisation**	**Overall Utilisation**	**Execution time**	**Slurm_ID**
Original	$H = 5, P = 0.50$	0	0.81634	18608ms	130072
	$H = 10, P = 0.25$	0	0.86313	54790ms	130074
Modification 1	$H = 5, P = 0.50$	0.98913	0.20575	79887ms	130064
	$H = 10, P = 0.25$	*0.97635*	*0.21908*	*225651ms*	130066
Modification 2	$H = 5, P = 0.50$	*0.40768*	*0.68076*	*22584ms*	130077
	$H = 10, P = 0.25$	*0.29372*	*0.81776*	*58022ms*	130078

Data

All data and summary data are available in the Github Repository.