Implementation of a distributed task runner

The function first checks if num_procs is less than or equal to 2. If so, it calls run_all_tasks_seq to run tasks sequentially to avoid the overhead of message passing. Otherwise, the function operates differently depending on the rank of the process:

- Master Process (rank 0): Reads the initial tasks from the input file and sends one task to each worker process using MPI_Send and marks the workers busy. It keeps track of the status of each worker (busy or free) using a boolean vector. It then enters a loop and does the following:
    1. Assign new tasks to free workers using MPI_Send
    2. Receive messages from workers with MPI_Recv and add new tasks to the queue

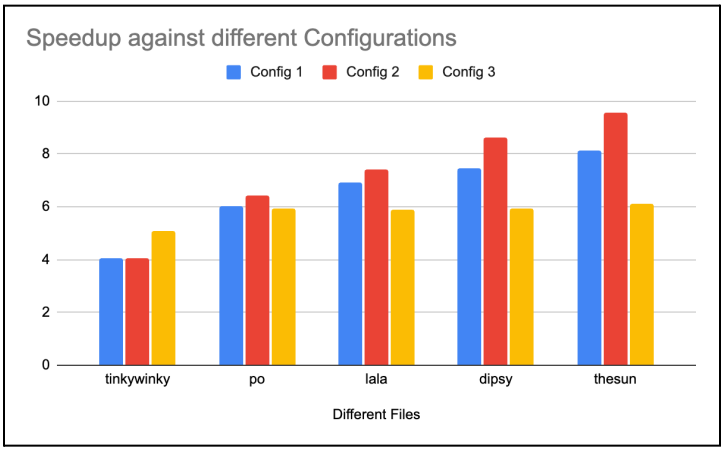
  The loop terminates when the task queue is empty and all workers are free. A termination message is then sent to all workers.

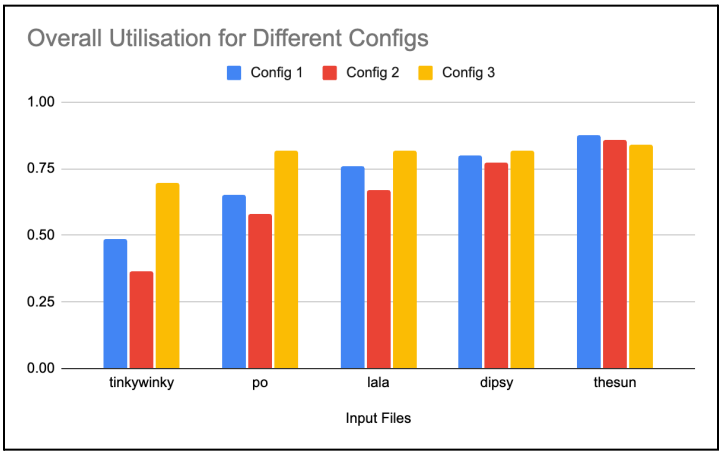
- Worker Processes: Worker processes receive tasks from the master process, execute them, and send the resulting children tasks back to the master process. They continue this process until they receive a termination signal from the master process.

To send tasks, we created a new MPI type for the task_t struct (code shown in appendix).

Performance Measurement

Performance Comparison Across Different Configurations





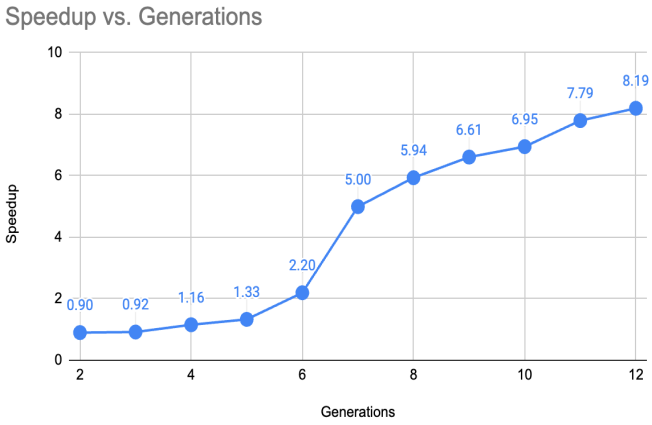
We first compared the speedup and utilisation for the three provided configs using the arguments provided under the minimum requirements section in the assignment pdf. For easier comparison, we arranged the input files in increasing order of total tasks executed.

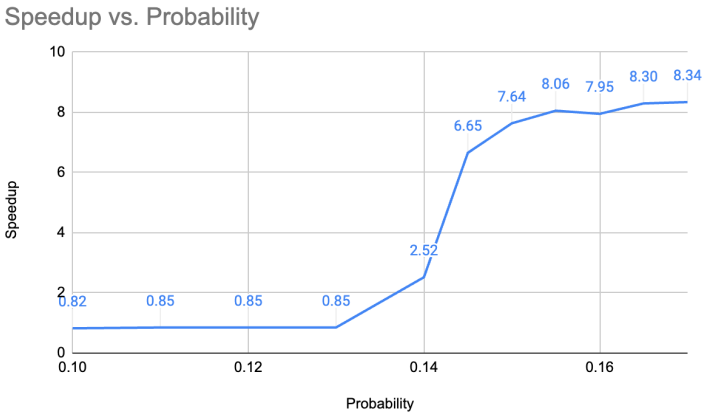
Comparing the configs, Config 2 (20 MPI processes) produced the best speedup followed by configs 1 (14 MPI processes) and 3 (8 MPI Processes). This is consistent with the number of processes allocated in each config. This increase in speed up for the config with a higher number of available processes is due to the increased potential for exploiting parallelism.

We also observed an increase in overall utilisation as the total number of tasks executed increases. This was a surprising observation. Since the task allocation works by sending one message per task, we expected the communication overhead to be a roughly constant proportion of the overall time taken.

Upon closer examination of the data, we found that it is due to the task graph parameters. In the cases of "tinkywinky," "lala," and "po", there were insufficient tasks at any point in time for allocation to all processes. This led to processes of higher rank to idle, decreasing the overall utilisation. As the number of tasks generated increases, the differences in utilisation become less apparent due to the better utilisation of all processes available. Additionally, there could be a higher overhead associated with creating such processes and having them wait for resources to come. The greater number of idle processes would also lead to higher overhead through context switching.

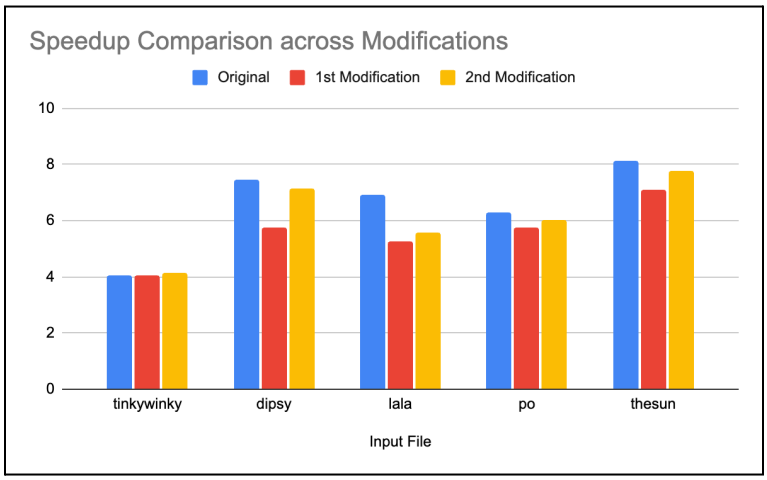
Varying the Task Graph Parameters

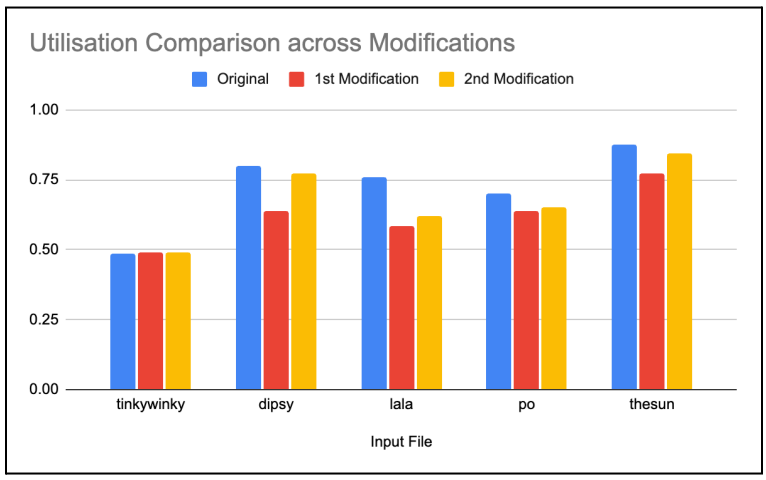




Next, we measured the speedup while varying the number of generations and probability in the task graph parameters. For both experiments, we observed a small initial speedup followed by a steep increase and then a slower increase.
The initial small speedup is due to the low number of tasks. In those cases, only one process was sufficient to execute all tasks at any point in time. This means that openMPI was pure overhead in these cases.
The subsequent increase in speedup is due to the increase in tasks. Both parameters directly cause an increase in available tasks for execution, leading to a greater potential for parallel execution. The slower increase hints at the point of saturation.

Attempted Modifications





First Modification

In the first implementation, the main process would wait for one task to finish before dispatching the next one to an available worker. We tried to improve this by enabling the main process to assign tasks to workers even while they were busy executing. This attempted to hide the communication overhead by using Isend instead of Send to efficiently distribute tasks among all workers when the `task_buffer` was not empty. To determine the next worker to assign a task to, we monitored the backlog of tasks for each worker and selected the one with the least pending tasks.

To enable the worker processes to send the completed tasks back to the main process without blocking, we implemented multiple task buffers. Each time a task was executed, a new buffer was used to guarantee the safe and non-blocking use of Isend for sending newly generated tasks back to the master. We cycle through 10 buffers and if a buffer is reused, we perform a wait to ensure that it is safe for reuse.

This modification performed worse as shown in the graphs above. A possible explanation for this is that we failed to hide the communication overhead by overlapping it with computation. This modification allowed the master process to send the tasks in a non-blocking manner. However, the master process does not perform any heavy computation and thus we are in fact not hiding the communication overhead at all. Furthermore, the extra wait operation for the workers might have caused extra overhead, resulting in poorer performance.

Second Modification

Building upon our initial modification, a crucial aspect of this enhancement was the thoughtful redesign of the worker's execution strategy. After executing a task, the worker initiates a non-blocking `Isend` operation to transmit the newly generated tasks. In this modification, we keep one child task and send the rest. This deliberate choice allows the worker to seamlessly

transition into executing another task, adopting a parallel execution and communication approach.

This strategy aimed to boost overall performance by enabling workers to carry out concurrent operations—sending a child task and executing another task in parallel. Simultaneously, at the master node, the task buffer was ensured to dispatch all available tasks to each node, aiming to maximise the utilisation of all workers effectively.

However, this modification did produce the expected improvements. The performance and utilisation, although better than the first modification, performed worse than the original. The continuous execution done by each worker may have the unintended consequence of leaving fewer tasks for the remaining workers. This could be a contributing factor to the decreased utilisation and performance compared to the original design. Furthermore, the increased number of Isend operations introduced with this modification possibly leads to degraded performance and diminished utilisation due to the additional time spent on data transmission.

Appendix

Types

runner.cpp

```
// MPI message definitions for task_t
constexpr int task_t_num_blocks = 8;
constexpr int task_t_lengths[task_t_num_blocks] = { 1, 1, 1, 1, 1, 1, 4, 4 };
constexpr MPI_Aint task_t_displs[task_t_num_blocks] = {
   offsetof(struct task_t, id),
   offsetof(struct task_t, gen),
   offsetof(struct task_t, type),
   offsetof(struct task_t, arg_seed),
   offsetof(struct task_t, output),
   offsetof(struct task_t, num_dependencies),
   offsetof(struct task_t, dependencies),
   offsetof(struct task_t, masks) };
inline const MPI_Datatype task_t_types[task_t_num_blocks] = { MPI_UINT32_T, MPI_INT,
MPI_INT, MPI_UINT32_T, MPI_UINT32_T, MPI_INT, MPI_UINT32_T, MPI_UINT32_T };
MPI_Datatype MPI_TASK_T;
```

```
// Create the custom MPI data types
 MPI_Type_create_struct(task_t_num_blocks, task_t_lengths, task_t_displs,
task_t_types,
   &MPI_TASK_T);
 MPI_Type_commit(&MPI_TASK_T);
```

Modified Runner Files

Modification1: runner_mod1.cpp

Modification2: runner_mod2.cpp

Reproduction Instructions

The results are tabulated here: CS3210 Assignment 3 Data
A simple bash script named batchrun.sh is included and can be run using `./batchrun.sh`. This will submit all the slurm jobs we used for our testing. To run the modified files, simply rename them to runner.cpp and run batchrun.sh again. For reference, the parameters are shown below:

```
configurations=(
 # Default configs
 "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"
 "16 1 2 0.10 tests/tinkywinky.in"

  # Vary Generations
 "2 0 10 0.16 tests/thesun.in"
 "3 0 10 0.16 tests/thesun.in"
 "4 0 10 0.16 tests/thesun.in"
 "5 0 10 0.16 tests/thesun.in"
 "6 0 10 0.16 tests/thesun.in"
 "7 0 10 0.16 tests/thesun.in"
 "8 0 10 0.16 tests/thesun.in"
 "9 0 10 0.16 tests/thesun.in"
 "10 0 10 0.16 tests/thesun.in"
 "11 0 10 0.16 tests/thesun.in"
 "12 0 10 0.16 tests/thesun.in"

 # Vary Probability
 "12 0 10 0.10 tests/thesun.in"
 "12 0 10 0.11 tests/thesun.in"
 "12 0 10 0.12 tests/thesun.in"
 "12 0 10 0.13 tests/thesun.in"
 "12 0 10 0.14 tests/thesun.in"
 "12 0 10 0.145 tests/thesun.in"
 "12 0 10 0.15 tests/thesun.in"
 "12 0 10 0.155 tests/thesun.in"
 "12 0 10 0.16 tests/thesun.in"
 "12 0 10 0.165 tests/thesun.in"
 "12 0 10 0.17 tests/thesun.in"
)
```