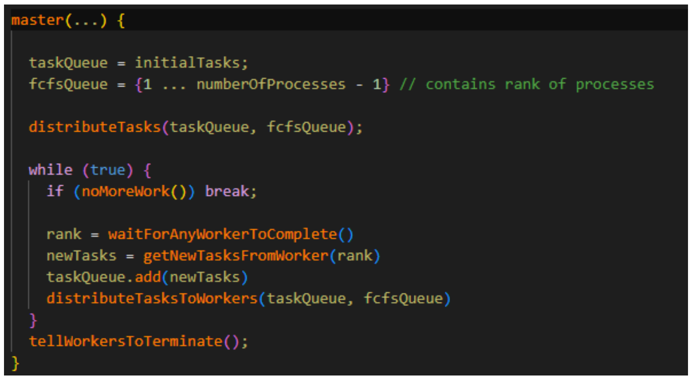
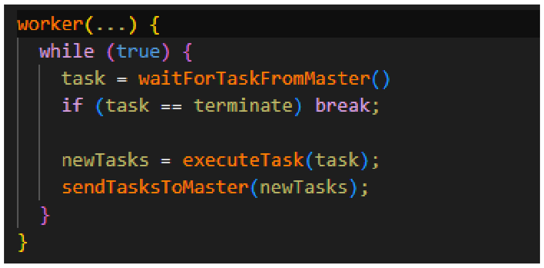
Implementation Outline

Our implementation follows the master-worker pattern, with the rank 0 process serving as the master and the remaining processes as workers. The master is responsible for task distribution. It first distributes the initial set of tasks. Whenever a worker completes a task, it will send its newly generated tasks to the master. In response, the master will add the new tasks to its task queue and distribute tasks to available workers. This is shown in the pseudocode below.



```
master(...) {

  taskQueue = initialTasks;
  fcfsQueue = {1 ... numberOfProcesses - 1} // contains rank of processes

  distributeTasks(taskQueue, fcfsQueue);

  while (true) {
    if (noMoreWork()) break;

    rank = waitForAnyWorkerToComplete()
    newTasks = getNewTasksFromWorker(rank)
    taskQueue.add(newTasks)
    distributeTasksToWorkers(taskQueue, fcfsQueue)
  }
  tellWorkersToTerminate();
}
```

The distribution of tasks is done in a First Come First Serve (FCFS) fashion ,using `fcfsQueue`, to achieve workload equalization. Whenever a worker completes a task, it is added to the back of the queue and will wait to receive another task. Each worker is only allocated a single task at any given time. Workers simply wait to receive a task from the master, execute the task, send the new tasks to the master and once again wait for another task. The pseudo-code below depicts this process.



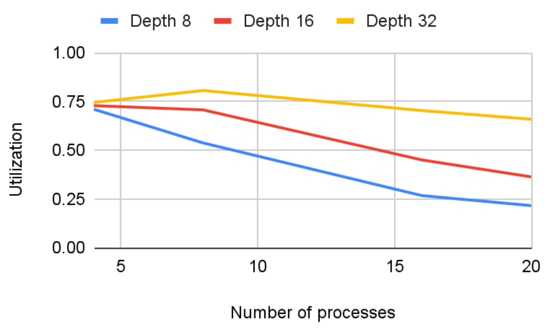
```
worker(...) {
  while (true) {
    task = waitForTaskFromMaster()
    if (task == terminate) break;

    newTasks = executeTask(task);
    sendTasksToMaster(newTasks);
  }
}
```

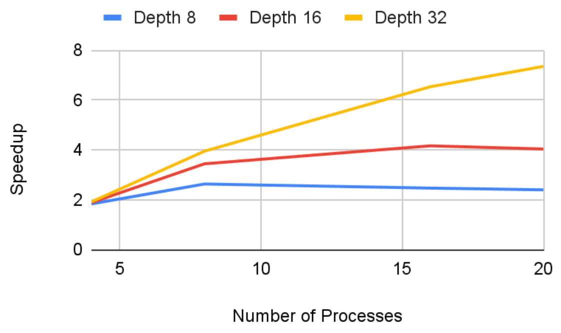
It is important to note that our implementation does not consider the existence of any dependencies between tasks.

Performance Measurement

To assess the performance of our implementation, we measured the average utilization and speedup across an increasing number of processes, while considering various conditions involving varying numbers of nodes and tasks. For this section, we define utilization as the fraction of time spent executing tasks by an MPI process.Speedup is the runtime speedup compared to the provided sequential implementation. Conveniently, both of these performance metrics are provided by the sample code. Notably, average utilization will be lower due to master’s exclusion from task execution, despite it doing useful scheduling work.

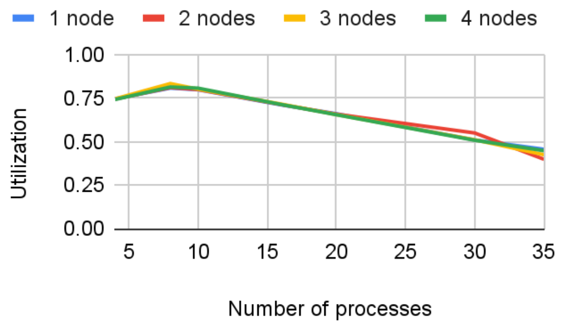
When we varied the number of tasks, we used a configuration with two `xs-4114` nodes, where the MPI processes are split evenly between both nodes. We used `tinkywinky.in` as the initial seed file with Nmin = 1, Nmax = 2 and P = 0.10 and used depth to control the number of tasks. The results are summarized by the graphs below and the relevant data can be found in table A and B in `data.csv`.

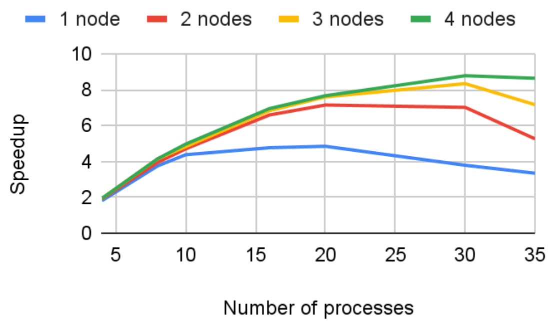




Utilization and speedup is generally higher at higher numbers of tasks, due to higher opportunities for parallelism. However, utilization decreases when the number of processes increases, likely because the greater number of processes results in greater communication overhead. The change in speedup as the number of processes increases is different depending on the depth. It seems that as long as the number of tasks is sufficient, as in the case of a depth of 32, an increase in the number of processes leads to a noticeable increase in speedup.

When we varied the number of nodes, we only used `xs-4114` nodes. We used `tinkywinky.in` with the parameters H=32, Nmin=1, Nmax=2 and P=0.10. The results are summarized by the graphs below and relevant data can be found in table E and F in `data.csv`:





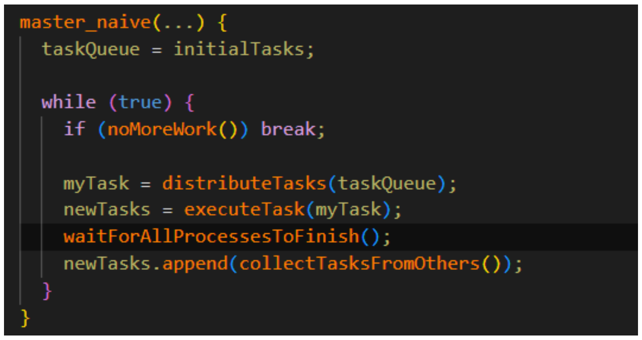
Speedup increases as the number of processes increase, only decreasing at a point where the number of processes exceeds the number of cores in the configuration. In particular, the configuration with 4 nodes experienced a slight decrease in speedup after crossing 30 processes despite having 40 cores. We hypothesized that this was due to the static number of tasks that cannot justify the increase in communication overhead as the number of processes increases. To test this, we ran the 4 node configuration at a depth of 40 with 35 processes. The result was a speedup of 13.4 compared to the speedup of 8.63 shown in the graph. Based on both investigations, we conclude that our implementation scales well with an increasing number of processes, given enough hardware resources and tasks.

Modifications

Over the course of this assignment, we came up with a total of 3 different implementations: `run_all_tasks_naive`, `run_all_tasks` and `run_all_tasks_master` in chronological order. `run_all_tasks` is our final implementation that we described in the first section. We refer to `run_all_tasks_naive` as the naive implementation, `run_all_tasks` as the final implementation and `run_all_tasks_master` as the master implementation.

`run_all_tasks_naive`->`run_all_tasks`

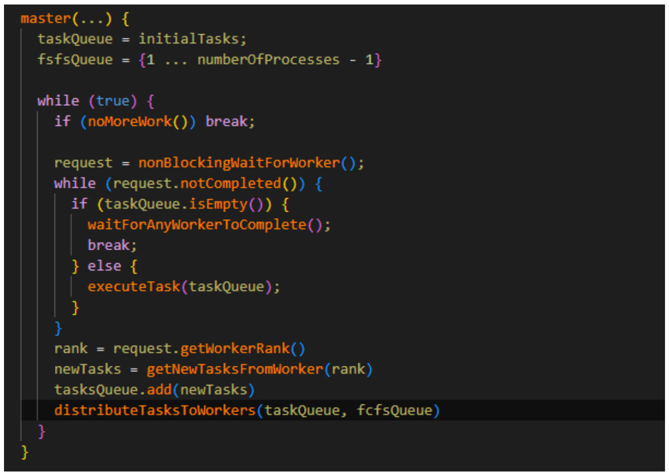
Our naive implementation uses a hybrid of master-worker and parbegin-parend. The rank 0 process will distribute tasks among the processes, including itself. It will wait until task execution for all processes complete, gather the new tasks from the processes and distribute tasks again. The pseudocode for the master is shown below and the worker is the same as the first section.



```
master_naive(...) {
  taskQueue = initialTasks;

  while (true) {
    if (noMoreWork()) break;

    myTask = distributeTasks(taskQueue);
    newTasks = executeTask(myTask);
    waitForAllProcessesToFinish();
    newTasks.append(collectTasksFromOthers());
  }
}
```

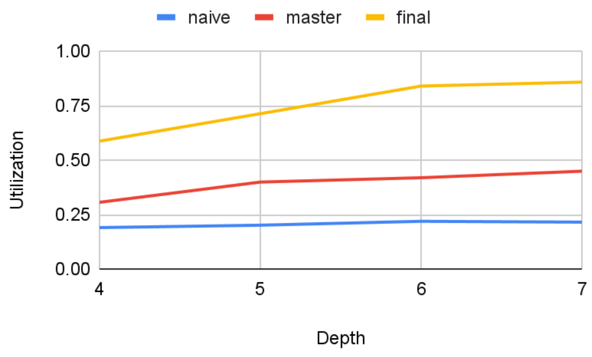
`run_all_tasks`->`run_all_tasks_master`

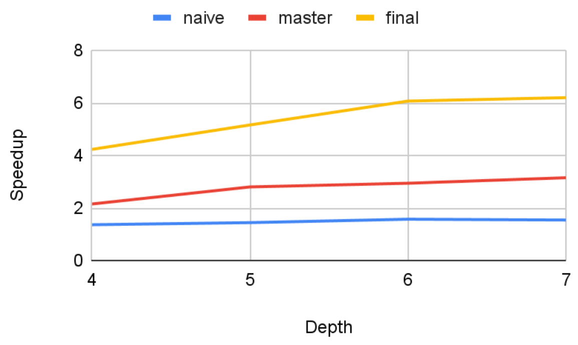
In our final implementation, we observed that our master process does not execute any tasks. The idea behind this modification is to let the master execute tasks while waiting for workers to finish executing their tasks, to achieve higher resource efficiency. This is implemented by making use of a non-blocking receive function. The pseudocode for the master is shown below.



```
master(...) {
  taskQueue = initialTasks;
  fsfsQueue = {1 ... numberOfProcesses - 1}

  while (true) {
    if (noMoreWork()) break;

    request = nonBlockingWaitForWorker();
    while (request.notCompleted()) {
      if (taskQueue.isEmpty()) {
        waitForAnyWorkerToComplete();
        break;
      } else {
        executeTask(taskQueue);
      }
    }
    rank = request.getWorkerRank()
    newTasks = getNewTasksFromWorker(rank)
    tasksQueue.add(newTasks)
    distributeTasksToWorkers(taskQueue, fcfsQueue)
  }
}
```

Performance measurements were taken with two `i7-7700` nodes, using `dipsy.in` as the seed file and with Nmin=3, Nmax=3 and P=0.00. The results are shown below and the relevant data can be found in tables C and D in `data.csv`.





Utilization for the naive implementation is rather low, due to the fact that there is a lot of idle time as processes wait for all other processes to finish for every batch of distributed tasks. Utilization for the master implementation is also quite low compared to the final, most likely due to idle time for the workers as the master is executing tasks instead of distributing them. In both cases, speedup is much lower compared to the final implementation, hence it is what we chose as our implementation.

Conclusion

In conclusion, our distributed task runner implementation uses a master-worker pattern, consisting of a single master process that exclusively distributes tasks to workers and worker processes that executes any received task and sends the result back to the master. Our implementation scales well with an increasing number of processes as long as there are enough tasks to offset the communication overhead between the processes and enough processing elements to avoid too much overcommitment.