15418 Project Proposal

Jason Yuan (laiy), Rui Zhou (ruizhou)

1 Parallel Graph Coloring for Task-Scheduling Simulation

2 Url

`https://jyuan2003.github.io/task-scheduler`

3 Summary

We will implement and optimize a parallel scheduler that dynamically maps tasks to different processes. We will do this by encoding the computation and communication of tasks into a graph that allows queries of edge and vertex update and viewing task assignment as a coloring of the vertices. We will analyze the performance of our implementation under OpenMP and MPI, as well as the speedup for various different graph structures corresponding to different computation-communication patterns in real-life parallel programming.

4 Background

In parallel programming, the speedup of the code can often depend on the effectiveness of the task assignment policy. On the one hand, it is often desirable to ensure that each process gets about the same amount of computation, so that the process with the most tasks assigned to it does not become a bottleneck to the total speedup of the program; on the other hand, the computation of task i may produce a result that is required in the computation of task j, so assigning task i and j to different processes can induce overhead such as synchronization or communication. One challenge to implementing an effective task scheduler is that the requirement to balance workload and the requirement to minimize data dependency across processes can often go against each other. Another issue is that the workload and dependency of a task can be dynamically changing (a good example of which is the Barnes-Hut algorithm), so the scheduler will also need to dynamically adjust its task scheduling.

In this project, we will implement a parallel task scheduler as follows: given N tasks and P processes, we construct a graph G in which each task i is represented as a vertex v_i, and task i's data dependency on task j is represented as an edge (v_i, v_j). Given P parallel processes, process i is represented by the color P_i, and given a coloring χ of the graph, the cost of vertex v_i is defined by

$$\begin{aligned} c(v_i) &= \text{computation time of task } i + \sum_{(v_i,v_j) \text{ s.t. } \chi(v_i)\neq\chi(v_j)} \text{data dependency on task } j \\ &:= t(v_i) + \sum_{(v_i,v_j),\chi(v_i)\neq\chi(v_j)} t(v_i, v_j) \end{aligned}$$

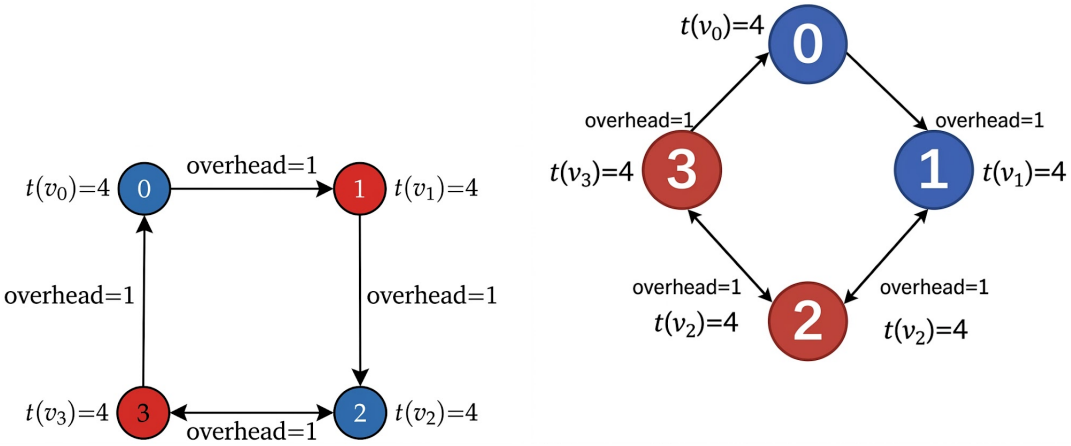
That is, we sum up the dependency on the tasks that are assigned to a different process and the computation time of the task itself. For each P_i, the total time of process i is

$$t(P_i) = \sum_{\chi(v_j)=P_i} c(v_j)$$

and the total time of the program is $t(G) = \max_i t(P_i)$. We will approximate an optimal scheduling of tasks as follows:

```
initially color G
partition G into NUM_THREADS parts
//each thread does the following in parallel
for (int i = 0; i < NUM_IT; i++){
    for v_j in the partition{
        for (int k = 0; k < P; k++){
            color v_j with P_k
            compute t(G) under this coloring
        }
        update the color of v_j to the one that gives lowest cost
    }
}
```

Note that when the color of v_j is updated, the only changes that this update makes to $t(G)$ are at v_j and its neighboring vertices, so we only need to go through the neighbors of v_j to compute the new value of $t(G)$. Below is a toy example of the result after running one iteration of the algorithm:



To handle update queries, we simply modify the algorithm by changing the graph structure for each update and modify the cost computation accordingly.

This algorithm can greatly benefit from parallel programming when the number of tasks N is large, since computation and update of a vertex only involves its neighboring vertices without affecting the cost of other vertices, so there is a lot of room for parallelism.

5 The Challenge

One major challenge to the performance of our implementation is that the amount of computation required for updating a vertex depends on its degree, which may vary greatly across different vertices. Therefore, we need to partition the vertices in a way such that the total degree for each scheduling thread is close enough. Additionally, under a message-passing model, there will be severe communication overhead if the vertices computed by a scheduling thread has many neighbors from other threads, so our partition also needs to guarantee that each scheduling thread's vertices approximately form a continuous cluster, and this can sometimes go against workload balancing.

Another problem is that the graph is dynamically updated, so even if the initial partition of the graph is good for performance, it may become suboptimal in the long run; for example, in the case of Barnes-Hut, the task of a star may start out inside one cluster, but as time evolves the computation of the star depends more heavily on data from another cluster, in which case the star should be re-scheduled to a different process. For

us, this means that the program needs to dynamically move the computation of vertices from one scheduling thread to another, which would require careful synchronization.
Also, accesses to the data of the neighbors of a vertex may be scattered across the memory if implemented naively, which would give poor locality.

6 Resources

We will write our code from scratch, since the algorithm itself is straight-forward. We will be using GHC machines to test the performance, and PSC machines for high thread-counts to test scalability. We will also need to generate task graphs for different types of parallel programming problems, which involves searching for instances of those problems like Barnes-Hut or Ocean simulation.

7 Goals and Deliverables

Plan to Achieve:

- Organize the graph structure and improve locality
- Implement the task-scheduler using both OpenMP and MPI, and compare the performance of the two implementations (both in computation time and in speedup).
- Dynamically re-allocate vertices to achieve workload balancing but avoiding too much synchronization overhead.
- Analyze the performance and speedup of our implementation on various types of task graphs for different problems

Hope to Achieve:

- Compare the performance of our implementation to some of the existing scheduling algorithm (both the computation time and the cost of the schedule produced)
- Study some of the existing implementations and use them as guide to analyze potential optimization.

8 Platform Choice

We will use both GHC machines and PSC clusters, since they support both multi-threading and multi-process execution, which are necessary for our OpenMP and MPI implementations in C++. Since the update of a vertex only depends on its neighboring vertices, which can be done inherently in parallel, OpenMP and MPI provide a perspective on the trade off between parallel computation and thread synchronization cost or inter-process communication overhead, respectively.

9 Schedule

- Week 1: Generate graphs with varying topologies, edge densities, and sizes and re-organize the graph structure to improve locality and storage through graph sharding and compression. Setup the experiment pipeline.
- Week 2: Develop both synchronous and asynchronous OpenMP implementations and experiment with different work assignment strategies, in-place or out-of-place updates, and different types of graphs.

- Week 3: Develop both synchronous and asynchronous MPI implementations and experiment with different work assignment strategies, in-place or out-of-place updates, different types of graphs, and different interconnect topologies.
- Week 4: Debug and continue to optimize current OpenMP and MPI implementations. Benchmark their scalability on PSC machines.
- Week 5: Summarize and analyze the final results.