Greedy Pipeline Scheduling for Online Dispersed Computing

Diyi Hu, Pranav Sakulkar, Bhaskar Krishnamachari
October 2, 2017

1. Introduction:

Dispersed Computing leverages the computing capabilities of heterogeneous resources dispersed across the network. With effective task scheduling, performance of computationally intensive applications can be significantly improved in dispersed computing environments. Task scheduling is the process of mapping tasks in the application to the target computing environment. Tasks of an application are described by a Directed Acyclic Graph (DAG). In the DAG, nodes represent tasks and edges represent dependency constraints among tasks. The edge weights represent the amount of data packets required by the successor task. The target computing environment is a network of compute nodes [1]. A classic scheduling algorithm, HEFT [2], sorts tasks by a metric called upward rank, and then greedily assigns tasks to compute nodes with earliest finish time. HEFT minimizes the makespan, which is defined as the end to end latency of the task graph. However, it is not optimized to schedule jobs that arrive in an online fashion for pipelined execution. In such case, makespan has an additional component: queuing delay, which could be relieved by the use of additional compute resources. This technical report proposes a greedy heuristic that aims to improve HEFT for online jobs by duplicating tasks to additional compute nodes.

2. Simulator:

We develop a simulator in Python to simulate the execution of tasks on a network of compute nodes. The inputs of our simulator are a task graph (DAG), its mapping to different nodes across the network, the communication profiles of the node-node links, the execution profiles of the nodes and the job arrival rate. The outputs of our simulator are the average end-to-end latency of the jobs and the average execution time of each task in the task graph.

The simulator evaluates the given task mapping by considering the computation delay, queueing delay and communication delay associated with each compute node.

3. Greedy Scheduling Algorithm:

Inputs to the algorithm:

1) Task Graph
2) Execution profiles of the compute nodes
3) Communication profiles of the data links
4) Arrival rate of the jobs
5) Total number of jobs to simulate

Routine:

1) Obtain an initial scheduling by HEFT
2) Iterate until the candidate nodes converge:
    a) Call the simulator with current scheduling to find the bottleneck task by execution time
    b) Find the bottleneck task by the execution time
    c) For each available candidate node, create an allocation that duplicates the bottleneck task on this node, and call the simulator to obtain the average end-to-end latency.
    d) Select an allocation that minimizes the average end-to-end latency and use it as the current schedule if the evaluated average latency for this allocation is lower than that of the current one

The termination condition is satisfied when there is no improvement on the average end-to-end delay.

By finding the bottleneck and greedily selecting the new node, we expand our set of chosen compute nodes. This means that a new node is selected by our algorithm only if it helps in reducing the average latency of the schedule. If one task is performed on multiple compute nodes, the input of the task is fed in a round-robin scheme. The output rate of its parent task is unchanged.

Fig.1 illustrates the code structure of our implementation.

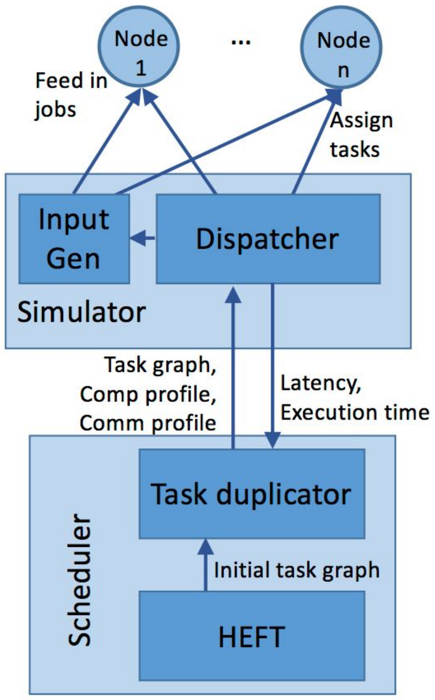

Figure 1: code structure

4. Experiments:

We perform simulations for task-graph shown in Fig.2 with 10 compute nodes. We use HEFT as the baseline of our algorithm. HEFT takes as input the task graph, execution profiles of the nodes and the link profiles to generate a mapping of the tasks to compute nodes.

We assume that no more than one task can be assigned to one compute node. We set the execution and communication profile such that the system is computation bound. Tasks B and C are much more computationally expensive than the other tasks.

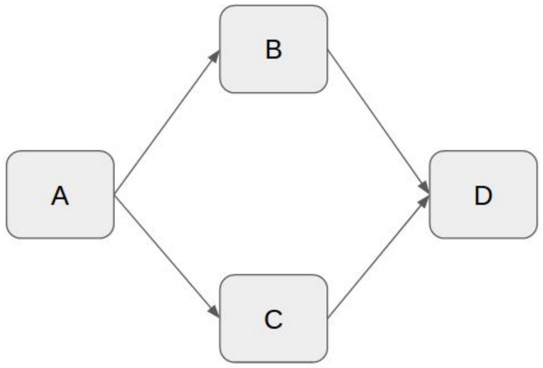

Figure 2: Task-Graph Structure

5. Results:

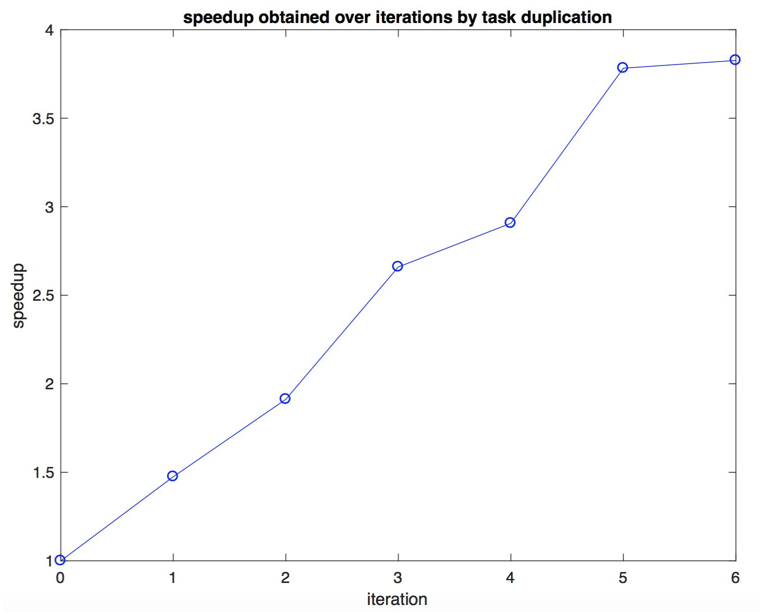

Figure 3: Speedup compared to HEFT

1. The speedup over the HEFT scheduling (Fig.3) is computed by
   $speedup \ = \ average\ end-to-end\ delay\ in\ i^{th}\ iteration\ /\ HEFT\ end-to-end\ delay$

2. Initially, HEFT algorithm schedules each task on one compute node. Hence, 4 compute nodes out of 10 are occupied while the remaining are available for task duplication.
3. In the first iteration of our greedy algorithm, task B is identified as the bottleneck task. The 6 available nodes are traversed one by one. The node that minimizes the average end-to-end delay is selected for task B duplication. This gives a speedup of 1.48x, due to the decreased execution time and queuing delay of task B.
4. Since task D depends on the computationally expensive tasks B and C, duplicating only one of D's predecessors cannot significantly speed up task D. Thus, in the second iteration, task C is determined as the bottleneck task and gets duplicated. The speedup then reaches 1.91x. This shows the advantage of our algorithm. By iteratively identifying the bottleneck task and duplicating it to the node which leads to maximum latency reduction, we effectively boost the performance.
5. After 6 iterations, the final speedup obtained by task duplication is 3.83x.

6. Conclusions

In this technical report, we proposed a greedy scheduling algorithm for dispersed computing with online inputs. The algorithm used the scheduling of HEFT as initialization. Then it iteratively identified the bottleneck task and duplicated the task to the node which led to maximum latency reduction. We simulated our algorithm on a computation bound system with 10 compute nodes and 4 tasks with pipelined online jobs. Experiments showed that our algorithm achieved 3.83x speedup over HEFT.

Our ongoing work is focused on improving the simulator where each packet for a job has an associated ID. Only packets of the same job can be processed together.

In the future, we will evaluate our scheduling algorithm by larger and more diverse DAGs. We will further refine the algorithm (e.g. to exploit simulated annealing to avoid local optimum) or propose new algorithms that analytically identify the bottleneck task and the optimum duplication node. The ultimate goal is to implement the algorithm in conjunction with CIRCE and evaluate it on real environment.

Acknowledgement
This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001117C0053. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] Aleksandra Knezevic, Quynh Nguyen, Jason A. Tran, Pradipta Ghosh, Pranav Sakulkar, Bhaskar Krishnamachari, and Murali Annavaram, "DEMO: CIRCE – A runtime scheduler for DAG-based dispersed computing," The Second ACM/IEEE Symposium on Edge Computing (SEC) 2017.

[2] Topcuoglu, Haluk, Salim Hariri, and Min-you Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing." IEEE transactions on parallel and distributed systems 13.3 (2002): 260-274.