STATIC LIST SCHEDULING FOR DAGS: A COMPARATIVE STUDY BETWEEN ALGORITHMS

Naman Sharma
[redacted]
EE5903 Real-Time Systems Simulation Report
[redacted]
[redacted]

ABSTRACT

Scheduling of precedence constrained tasks in a system with heterogenous processors is a complex problem with requirements on both high performance and real-time constraints. This simulation report presents a comparison of two different algorithms (randomHEFT [1] and IPEFT [2]) to solve the static scheduling problem for DAGs. We examine the effect of various parameters of the DAG on the performance of the two algorithms. This leads us to identify the better algorithm out of the two and the pitfalls of each of the algorithms.

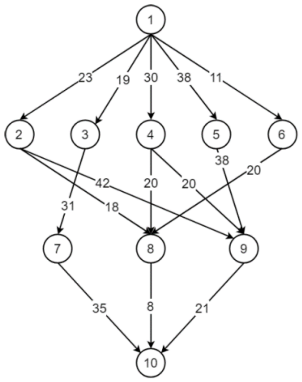
1 Introduction

Cloud Computing is one of the most important parts of any digital pipeline. Over the years, more and more enterprise systems have recognized cloud computing to be the primary source of computing power, to the extent that a "cloud-first" approach to system design is the most obvious choice [3]. More recently, we have also had the concept of Edge Computing gaining traction, where data is directly processed near to the point of generation rather than at the cloud. According to a Frost & Sullivan analysis, 5.6 billion IoT devices will be utilizing the power of edge computing by 2020 in several different use cases [4]. In a cloud computing environment, we usually have multiple heterogeneous computing nodes, where some CPU nodes may even be connected to their own dedicated GPUs. These resources are required to perform diverse computational tasks with a range of computational complexity. This complexity is only further aggravated in the use case of edge computing, where network architectures are not well defined, with nodes joining and leaving a network dynamically. In such cases, communication costs may also vary by significant amounts.

In such environments, task scheduling becomes an important step to achieve high performance. The primary objective of task scheduling is to assign tasks to the available processors and define the order of the execution of tasks to minimize the overall completion time. The preemptive scheduling problem is a wellknown NP complete problem [5] and the heterogeneity of processors only adds another layer of complexity to finding the optimal solution. Hence most solutions involve finding sub-optimal solutions using various heuristics. The task-scheduling problem can be further classified into static scheduling or dynamic scheduling. In the case of static scheduling, information such as the execution and communication times between tasks in known beforehand. Methods of solving the static scheduling problems are further divided into three major groups: task-clustering scheduling algorithms, task-duplication scheduling algorithms and list-scheduling algorithms.

List scheduling algorithms have two main phases. The first phase is *task prioritization*, where each task is associated with a priority signifying its importance. This is followed by *task allocation* where tasks are selected with decreasing priority and allocated to the available processors for completion. In general, list scheduling methods have proved to be the most widely used due to their ease of implementation, low complexity (generally quadratic in relation to the number of tasks) and efficient solutions [6].

In this simulation report we will be comparing the performance of two different algorithms: IPEFT (proposed in [2]) and RandomHEFT (proposed in [1]). These two algorithms will be benchmarked against the commonly used HEFT algorithm [7]. The remainder of the report is organized as follows: Section 2 formally introduces the problem and some definitions used. Section 3 then continues to briefly describe the two algorithms considered in the simulation. Section



Computation Cost Matrix										
Task	1	2	3	4	5	6	7	8	9	10
p_1	17	24	18	8	10	35	45	43	22	40
p_2	12	19	19	25	44	16	11	5	16	14
p_3	27	28	9	16	44	32	6	31	8	34

Figure 1: An static scheduling problem can be described using two components: (a) a DAG (b) a Computation Cost matrix

4 & 5 looks at the simulation parameters and the results of the simulation, with the pitfalls of each algorithm being discussed specifically in Section 5.1. We then conclude this simulation report in Section 6.

2 Problem Formulation

We define the heterogeneous set of processors P, where each processor p is fully interconnected. Each processor can perform communication and computation concurrently and the tasks are assumed to be non-preemptive, which means that a task once started runs uninterrupted until finished. These processors can be used to perform a set of inter-dependent tasks, represented by a Directed Acyclic Graph (DAG), $\mathcal{G} = (V, E)$ where vertex $v_i \in V$ is the i-th task and edge $e_{i,j} \in E$ is the task dependency constraint indicating that v_i must finish before v_j can begin. The weights $\bar{c_{i,j}}$ on the edges represent the communication cost that is incurred between tasks v_i and v_j, averaged over the possible combinations of processors on which the tasks are run. If both tasks are computed at the same processors, the communication cost is assumed to be negligible. In addition to the DAG $\mathcal{G}$, the computational cost matrix W is also required. The dimensions of W are $(v \times p)$ where $w_{i,j} \in W$ represents the execution time of task v_i on processor p_j. Figure 1 shows an example of a DAG.

Based on the above formulation, we can define a number of terms commonly used in DAG Scheduling algorithms:

- $\mathbf{pred(v_i)}$ (or $\mathbf{succ(v_i)}$) : The set of immediate predecessors (or successors) of task v_i in a DAG
- **Makespan:** The Actual Finish Time (AFT) of the last task in the DAG: makespan $= \max\{AFT(v_{exit})\}$
- **Earliest Start Time:** EST for task v_i on processor p_i is given by:

$$EST(v_i, p_j) = \max\left\{ \begin{array}{c} T_{available}(p_j) \\ \max_{v_m \in pred(v_i)} AFT(v_m) + c_{m,i} \end{array} \right\}$$

  where $T_{available}(p_j)$ is the earliest available time for p_j and the second max term refers to the earliest time by which p_j has all the data required to do task v_i
- **Earliest Finish Time (EFT):** EFT is simply defined as the EST plus the computation cost of v_i on p_j

$$EFT(v_i, p_j) = EST(v_i, p_j) + w_{i,j}$$

- **Critical Node:** A critical node is one for which the average earliest and latest start time are equal

3 Algorithm Description

3.1 RandomHEFT

Like all standard list scheduling algorithms, the Random Crossover algorithm proposed in [1] consists of 2 phases: task prioritization and processor selection.

Task Prioritization

In this phase the authors introduce a new ranking metric for different tasks, $weight(v_i)$. It is defined as the absolute ratio of the difference between the fastest execution $(w_{i,j})$ and the slowest execution time $(w_{i,k})$ of a task over the speedup.

$$weight(v_i) = \left| \frac{w_{i,j} - w_{i,k}}{w_{i,j}/w_{i,k}} \right|$$

The rank for each task v_i is then calculated recursively using eq. 1.

$$rank(v_i) = weight(v_i) + \max_{v_j \in succ(v_i)} \{\bar{c_{i,j}} + rank(v_j)\} \tag{1}$$

Task Allocation

In the standard HEFT algorithm, the scheduler takes the tasks in the decreasing order of their rank and allocates then to the processor that has the smallest EFT for that task. THis corresponds to the algorithm trying to achieve global maximum. However, the authors of [1] argue that sometimes allocating the task to the processor with the fastest execution time (corresponding to a local minimum) can lead to a smaller global makespan. This is referred to as *crossover*. The authors define a threshold τ (eq. 2) where is a random number $r \in [0.1, 0.3]$ is greater than the threshold, RandomHEFT performs a cross-over.

$$\tau = \frac{weight(v_i)}{weight_{abstract}} \quad \text{,where } weight_{abstract} = \left| \frac{EFT(v_i, p_j) - EFT(v_i, p_k)}{EFT(v_i, p_j)/EFT(v_i, p_k)} \right| \tag{2}$$

where p_j (resp. p_k) provides the global (resp. local) minimum. $\tau \in [0, 1]$ and if $\tau = 1$ then the algorithm behaves in the same way as HEFT[7].

3.2 IPEFT

The widely used vanilla PEFT algorithm [8] considers an Optimistic Cost Table (OCT) to define the ranking between tasks. In [2] the authors suggest an improved IPEFT algorithm that uses a Pessimistic Cost Table (PCT) and a Critical Node Cost Table (CNCT) to perform ranking.

Task Prioritization

The $PCT(v_i, p_j)$ defines the longest path from $succ(v_i)$ to the exit task when v_i runs on p_j.

$$PCT(v_i, p_j) = \max_{v_k \in succ(v_i)} \left\{ \begin{array}{c} \max_{p_m} \{PCT(v_k, p_m) + w_{k,m} + \bar{c_{i,k}}\} \\ \bar{c_{i,k}} \end{array} \right\} \tag{3}$$

Based on eq. 3 the tasks are given a rank as follows:

$$rank_{PCT}(v_i) = \frac{\sum_{k=1}^{p} PCT(v_i, p_k)}{p} + \bar{w_i}$$

Task Allocation

The processor allocation depends on two different metrics: $CNP(v_i)$ and $CNCT(v_i, p_j)$. $CNP(v_i)$ is a binary variable that takes the value TRUE if v_i is the parent of a critical node but not a critical node itself. If $CNP(v_i)$ is TRUE (resp. FALSE), $CNCT(v_i, p_j)$ represents the maximum value of the shortest path from the critical successors (resp. all successors) of task V_I to the end task.

Once the CNCT table is generated, the authors propose a modified definition of EFT_{CNCT}:

$$EFT_{CNCT}(v_i, p_j) = \begin{cases} EFT(v_i, p_j), & \text{if } CNP(v_i) = \text{TRUE} \\ EFT(v_i, p_j) + CNCT(v_i, p_j), & \text{otherwise} \end{cases} \tag{4}$$

Based on eq. 4 task v_i runs on the processor with the minimum EFT_{CNCT}.

Parameter	Value
n	$[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250, 300, 350, 400]$
fat	$[0.1, 0.4, 0.8]$
density	$[0.2, 0.8]$
regularity	$[0.2, 0.8]$
jump	$[1, 2, 4]$
CCR	$[0.1, 0.25, 0.5, 0.8, 1, 2, 5, 8, 10, 15, 20, 25, 30]$
β	$[0.1, 0.2, 0.5, 0.75, 1, 2]$
p	$[4, 8, 16, 32]$

Table 1: Set of simulation parameters.

4 Simulation Parameters

For the simulation, a large number of DAGs were created using DAGGEN [9]. This allowed us to create DAGS with different parameters to better compare the two algorithms. The parameters controlled were:

- **n:** The number of nodes in the DAG / tasks in the applications
- **fat:** Used to control the width and height of the DAG. The width of each level is sampled from a uniform distribution of $fat \times \sqrt{n}$. The height of the DAG is then increased until n number of nodes is achieved.
- **density:** Controls the number of edges between two levels of a DAG
- **regularity:** A low value of regularity means that there is an increased irregularity in the number of tasks in each level.
- **jump:** Defines the maximum number of levels an edge is allowed to jump

In addition to the above parameters, we included additional parameters as a part of the parsing process to model a heterogenous processing environment. These additional parameters are:

- **CCR:** Communication to Computation Ratio defines the ratio between the sum of the edge weights and the sum of the node weights in a DAG.
- β**:** A higher value of β leads to a higher degree of heterogeneity in the processors. A random average computing cost $\bar{w_i}$ is first selected randomly between $[20, 50]$ for each task i. The cost of task i on processor j is then sampled such that:

$$\bar{w_i} \times \left(1 - \frac{\beta}{2}\right) \leq w_{i,j} \leq \bar{w_i} \times \left(1 + \frac{\beta}{2}\right)$$

- **p:** Number of processors

In our simulations the different permutations of parameters used are tabulated in Table 1. This leads to a total of $179,712$ DAGs. Each DAG is then assigned random node and edge weights to create 2 different graphs. Hence, we have $359,424$ graphs.

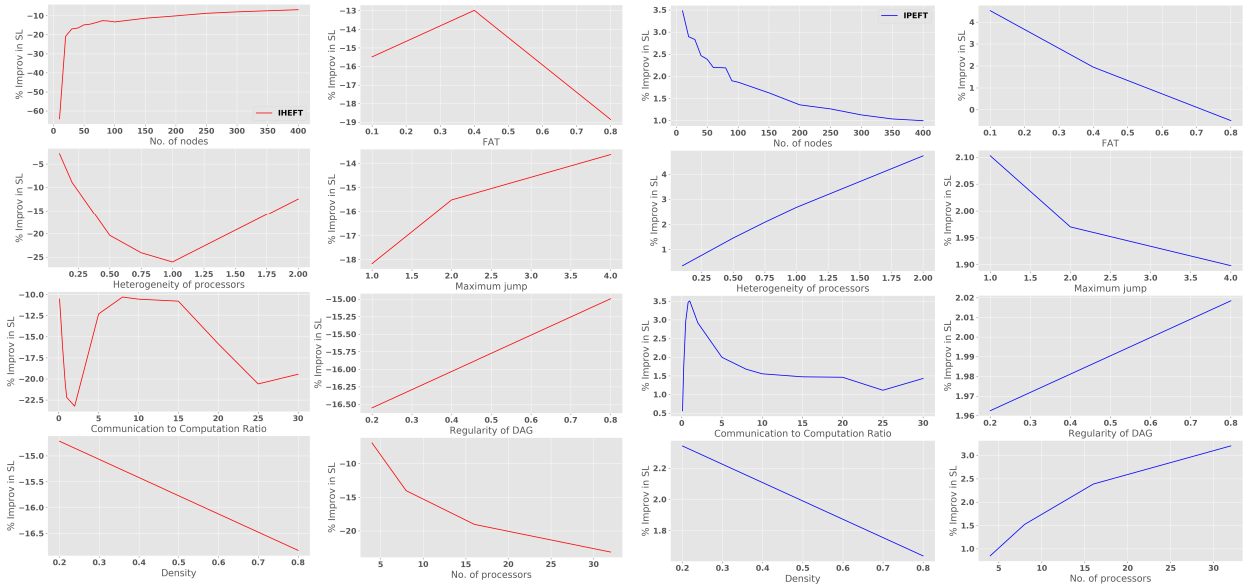
5 Simulation Results

We compare the performance of the two algorithms with the performance of the HEFT algorithm. For this purpose, we also programmed a HEFT Scheduler which is similar to the one implemented in [10] with minor changes. The metric for comparison is the percentage improvement in Scheduled Length or makespan (eq. 5). A larger value of this metric is better.

$$\% \text{ Improvement in SL} = \left(1 - \frac{SL_{algo}}{SL_{HEFT}}\right) \tag{5}$$

In the case of RandomHEFT, only 32.95% of the DAGs had an equal or smaller makespan than the HEFT algorithm. On the other hand, the IPEFT algorithm performed much better, with 73.11% DAGs having a makespan of equal or lower to that of HEFT.

Fig. 4 and 5 show a more detailed analysis of how the parameters affect the total makespan of the scheduled DAGs. When considering each parameter, the results were aggregated for the other parameters. Therefore, every element is



(a) RandomHEFT

(b) IPEFT

Figure 2: Average % improvement in Scheduled Length for (a) RandomHEFT (b) IPEFT

plotted as a boxplot to show the distribution of values. In a boxplot, the line inside the box corresponds to the median value, with the limits of the box height corresponding to the 25^{th} and 75^{th} percentile.

In Fig. 4 we see that RandomHEFT consistently performs worse than vanilla HEFT on average. Another important general observation is that the percentage improvement fluctuates by large amounts for RandomHEFT, with as much as 40% improvement or 80% deterioration. This may be attributed to the random nature fo the algorithm, where running the same DAG multiple times can lead to different makespan. In contrast to RandomHEFT, the IPEFT algorithm (Fig. 5) consistently performs better that HEFT on average. Moreover, the variation in the results for IPEFT is also smaller.

We now look at how each of the simulation parameters affects the percentage improvement in Scheduled Length. For this analysis, we will focus on the average improvement as shown in Fig. 2.

1. **No. of tasks:** The randomHEFT algorithm performs better as the number of tasks increases. We see a sharp improvement between 0 and 50 tasks. This suggests that randomHEFT is not an algorithm of choice when the number of tasks is small. The trend supports the claims of the authors in [1]. However, the algorithm performs worse on average than the vanilla HEFT which is contradictory to the claims on the authors.

   The % improvements in SL for the IPEFT algorithm decrease with the increase in number of tasks. This supports the claims of the authors in [2].

2. **FAT:** Both randomHEFT and IPEFT perform better on thin DAGs (smaller FAT). This may be due to the fact that more interdependencies between tasks provide for a smaller solution set for the schedule, which is more rigid in terms of the sequence of tasks.

3. **Heterogeneity:** For the randomHEFT algorithm, we do not see a definitive trend between % improvement in SL and heterogeneity of the processors.

   For the IPEFT algorithm, we see that it performs better in a more heterogeneous cluster of processors. This improvement is also quite significant. In Fig. 5 we see that increased heterogeneity widens the boxplots only above the median value and not below. This is a desired property.

4. **Maximum Jump:** We see that the randomHEFT algorithm performs better when the jumps between tasks is higher. Conversely, the IPEFT algorithm performs worse as the jump increases. However, we believe that for both algorithms this trend is not statistically significant. This is confirmed in the boxplot figures (Fig. 4 and 5) where the distributions remain very similar over the different values of the maximum allowed jump.

5. **CCR:** For randomHEFT, we see that the performance decreases for $CCR \leq 1$, i.e., the tasks take more time to finish processing than the time it takes for communication between processors. However, randomHEFT performs better for $CCR > 1$. The performance degrades again for very high values of CCR. The authors

claim that the algorithms performs better for higher values of CCR, which is partially supported by our simulations. This is because the algorithm limits the amount of cross-overs in cases when the communication cost is very high. However, the authors did not provide results with very high values of CCR (> 10).

For IPEFT, we see that the performance increases until $CCR = 1$ and then decreases and plateaus for large value of CCR. This is contradictory tot he claims of the authors, who suggest that the IPEFT algorithm has larger improvements for larger values of CCR.

6. **Regularity:** Both algorithms perform better when the DAGs are more regular (similar number of tasks in each level of the DAG). However, the improvements are not very significant, as seen in the boxplot figures.
7. **Density:** Both algorithms perform better when the density of the DAGs is low (low no. of edges between each level).
8. **No. of Processors:** The performance of randomHEFT decreases as the number of processors increases. Conversely, the IPEFT performs better as the number of processors increases.

Time Complexity comparison

In Fig. 3 we see that all three algorithms have a similar time complexity ($\mathcal{O}(t^2p)$ where t is no. of tasks and p is no. of processors). RandomHEFT algorithm has a very similar time complexity to the HEFT algorithm. However, we see that the IPEFT algorithm has approximately twice the time complexity. This does not appear in the Big-O notation but can be seen in the graph. This is because the IPEFT algorithm computes two tables of size (t^2p) rather than just one (like in randomHEFT).

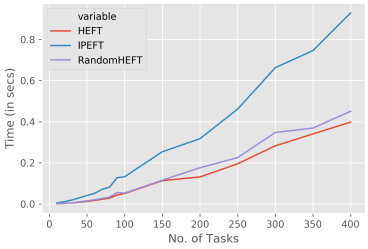


Figure 3: Time complexity of the three algorithms

5.1 Pitfalls of Algorithms

The randomHEFT algorithm suffers in its performance due to its random nature. We see that it is unable to perform better than the vanilla HEFT algorithm (contrary to the claim of its authors). We believe that this algorithm could perform much better if it used another metric to decide whether to perform cross-over rather than randomly deciding to do so. However, we see that it is able to provide very significant improvements in some cases. This convinces us that the algorithm can be improved to perform much better.

The IPEFT algorithm consistently performs better than both the vanilla HEFT and the randomHEFT. This algorithm is suitable for cases with small number of tasks and large number of extremely heterogenous processors. However, this algorithm suffers from a greater time complexity than the other algorithms.

6 Conclusion

In this simulation report we compared two algorithms in terms of the percentage improvement in the scheduled length. We based this comparison on multiple parameters and provided results with an extensive number of simulations. Our results suggest that the IPEFT algorithm is superior to the randomHEFT anf vanilla HEFT algorithms.

References

[1] S. AlEbrahim and I. Ahmad, "Task scheduling for heterogeneous computing systems," *The Journal of Supercomputing*, vol. 73, no. 6, pp. 2313–2338, jun 2017. [Online]. Available: http://link.springer.com/10.1007/s11227-016-1917-2

[2] N. Zhou, D. Qi, X. Wang, Z. Zheng, and W. Lin, "A list scheduling algorithm for heterogeneous systems based on a critical node cost table and pessimistic cost table," *Concurrency Computation*, 2017.

[3] D. M. Smith, "Cloud Computing Primer for 2019," *Gartner*, 2019.

[4] Frost & Sullivan, "Intelligence at the Edge—An Outlook on Edge Computing," Tech. Rep., 2017.

[5] J. D. Ullman, "NP-complete scheduling problems," *Journal of Computer and System Sciences*, 1975.

[6] H. Arabnejad, "List Based Task Scheduling Algorithms on Heterogeneous Systems - An overview," Universidade do Porto, Tech. Rep., 2013.

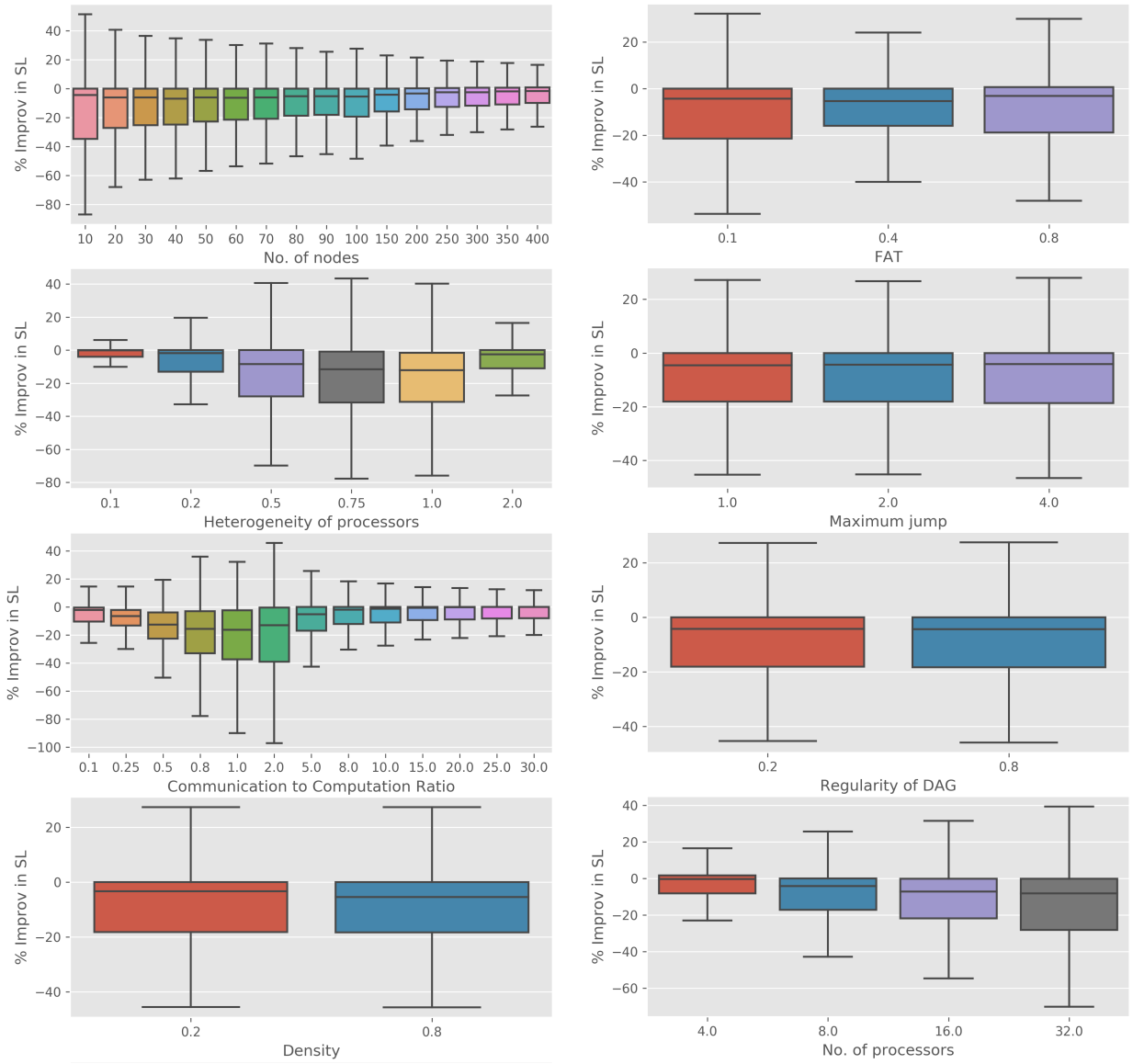


Figure 4: Effect of various parameters on total Schedule Length for RandomHEFT

[7] H. Topcuoglu, S. Hariri, and M. Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, 2002.

[8] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[9] F. Suter, "DAGGEN: A synthethic task graph generator," 2013. [Online]. Available: https://github.com/frs69wq/daggen

[10] Oyld, "HEFT," 2014. [Online]. Available: https://github.com/oyld/heft

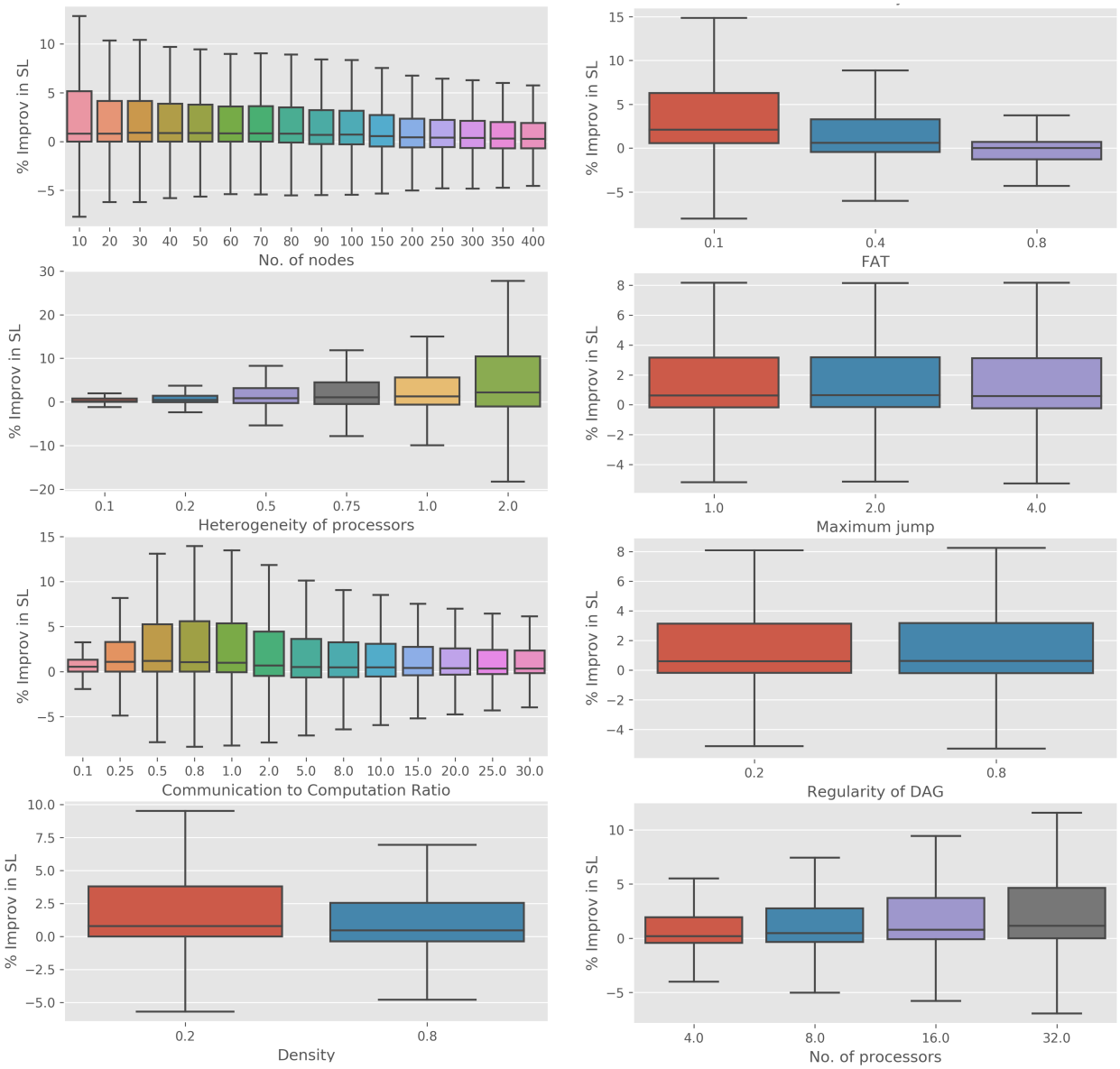


Figure 5: Effect of various parameters on total Schedule Length for IPEFT