

به نام خدا



# سیستم‌های بی‌درنگ

پروژه درس - زمان‌بندی منصفانه در سیستم‌های ابری

استاد :

دکتر سپیده صفری

نویسنده :

محمد هومان کشوری - هیربد بهنام

شماره دانشجویی :

۹۹۱۷۱۳۳۳ - ۹۹۱۰۵۶۶۷

## قسمت اول - Task Generator

در این قسمت دو کلاس Task، generate\_tasks را داریم که اولین کلاس به خود تسک‌ها اشاره می‌کند و دومی به ساخت تسک‌ها مطابق FFT<sup>۱</sup> و GE<sup>۲</sup> اشاره می‌کند. در generate\_tasks تابع داریم که مطابق زیر هستند:

```

1  @staticmethod
2  def FFT(m: int) -> dict[int, Task]:
3      total_graph_nodes = int(m * math.log2(m) + 2 * m - 1)
4      all_nodes: dict[int, Task] = {}
5      for node in range(1, total_graph_nodes + 1):
6          all_nodes[node] = Task(node)
7
8      for i in range(1, m - 1):
9          for j in range(2 ** (i - 1), 2 ** (i)):
10             for k in range(j * 2, j * 2 + 2):
11                 all_nodes[j].children.append(k)
12                 all_nodes[k].fathers.append(j)
13
14      base_pointer = 2 * m
15      forward = -1
16      for i in range(0, int(math.log2(m))):
17          for j in range(1, m + 1, 2 ** (i)):
18              forward *= -1
19              for k in range(j, j + 2 ** (i)):
20                  all_nodes[base_pointer + k - 1].fathers.append(
21                      base_pointer + k - m - 1
22                  )
23                  all_nodes[base_pointer + k - 1].fathers.append(
24                      base_pointer + k - m - 1 + forward * 2 ** (i)
25                  )
26                  all_nodes[
27                      base_pointer + k - m - 1 + forward * 2 ** (i)
28                  ].children.append(base_pointer + k - 1)
29                  all_nodes[base_pointer + k - m - 1].children.append(
30                      base_pointer + k - 1
31                  )
32              base_pointer += m
33
34      return all_nodes
35

```

تابع بالا یک عدد m را گرفته و مطابق آن یک دیکشنری از تسک‌ها تولید کرده و آنرا به عنوان خروجی می‌دهد. روابط بین تسک‌ها FFT DAG است.

```

1  @staticmethod
2  def GE(m: int) -> dict[int, Task]:
3      total_graph_nodes = int((m * m + m - 2) / 2)

```

<sup>۱</sup> Fast Fourier Transform  
<sup>۲</sup> Gaussian Elimination

```

4     all_nodes: dict[int, Task] = {}
5     for node in range(1, total_graph_nodes + 1):
6         all_nodes[node] = Task(node)
7
8     base_pointer = 1
9     while m > 1:
10         for i in range(base_pointer + 1, base_pointer + m):
11             all_nodes[i].fathers.append(base_pointer)
12             all_nodes[base_pointer].children.append(i)
13
14         for i in range(base_pointer + 1, base_pointer + m):
15             if i >= total_graph_nodes:
16                 break
17             all_nodes[i + m - 1].fathers.append(i)
18             all_nodes[i].children.append(i + m - 1)
19
20         base_pointer += m
21         m -= 1
22
23     return all_nodes
24

```

این تابع نیز مانند تابع قبلی یک عدد  $m$  را گرفته و مطابق آن تسک‌ها را و روابط میان آنها را ساخته و یک دیکشنری از تسک‌ها را خروجی می‌دهد.

```

1     @staticmethod
2     def draw_graph(graph: dict[int, Task]):
3         G = nx.DiGraph()
4
5         for node, attrs in graph.items():
6             G.add_node(node)
7             for child in attrs.children:
8                 G.add_edge(node, child)
9
10        pos = nx.drawing.nx_agraph.graphviz_layout(G, prog="dot")
11        nx.draw(G, pos, with_labels=True, arrows=True)
12        plt.show()
13

```

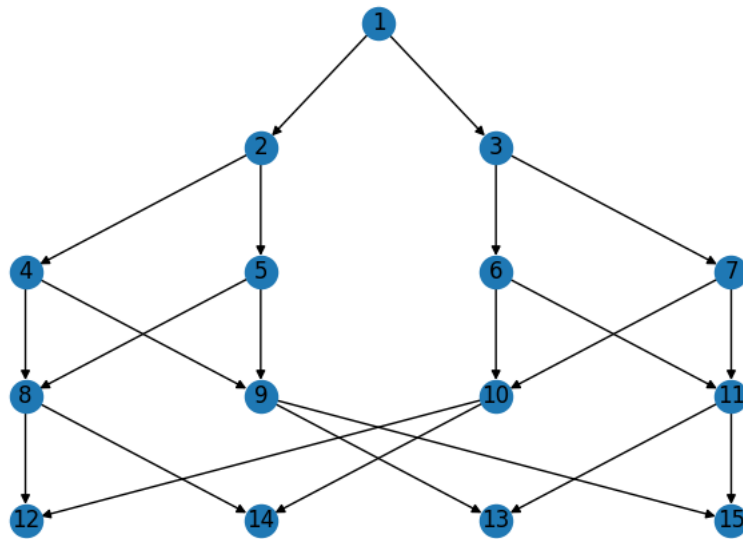
به واسطه این تابع نیز می‌توانیم تسک‌ها رسم کنیم. چند مثال از ورودی و خروجی متناظر آنها را مشاهده کنیم. ورودی برنامه و خروجی‌های متناظر آن :

```

1 python3 task_generator.py FFT 4
2

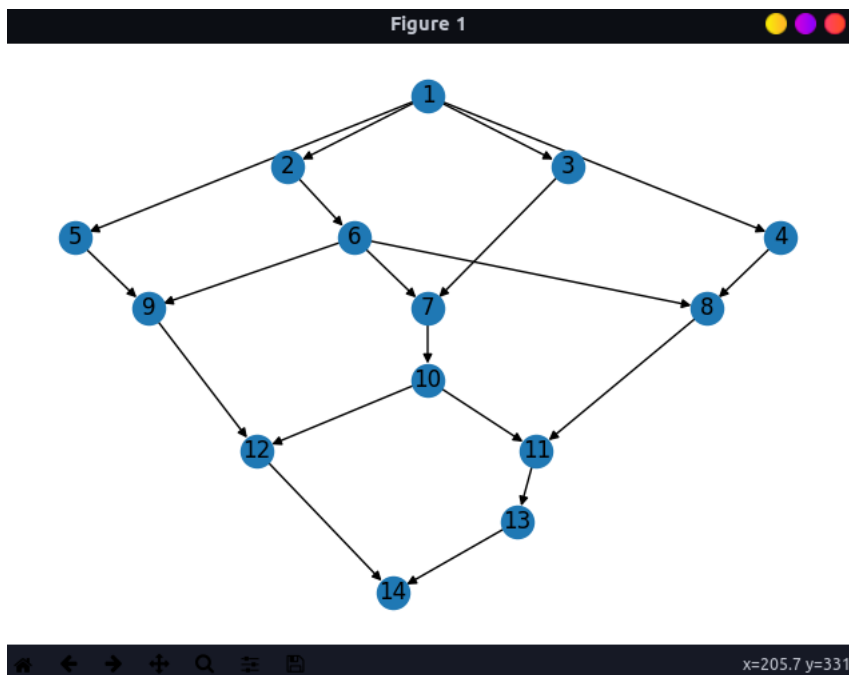
```

Figure 1



```
1 python3 task_generator.py GE 5
```

```
2
```



در کلاس Task نیز هر تسک ابتدا که تعریف می‌شود یک id و لیست پدران و فرزندان متناظر با آن نیز تعریف می‌شود و یک لیست که نشان می‌دهد هر تسک چقدر زمان اجرا دارد و یک دیکشنری که نشان می‌دهد هر تسک بر روی هر هسته چقدر به communication cost اضافه شد.

```

1 def __init__(self, id: int):
2     self.id = id
3     self.fathers: list[int] = []
4     self.children: list[int] = []
5     # This should be used like this:
6     # First index is the ID of the task which we want to comminate with
7     # (the child ID)
8     # The second index is the core ID which this task is currently on
9     # The third index is the core ID which child task should run on
10    self.communication_cost: dict[int, list[list[int]]] = {}
11    self.computation_times: list[int] = []

```

در توابع زیر نیز عملاً هزینه ارتباط و زمان محاسبه تسک به صورت رندوم برای هر cpu تعیین می‌شود.

```

1 def populate_cpu_dependant_variables(self, cpu_count: int):
2     for _ in range(cpu_count):
3         self.computation_times.append(random.randint(10, 100))
4         self.computation_times[-1] = min(
5             max(1, int(random.normalvariate(self.computation_times[-1],
6             5))),
7             self.computation_times[-1] + 25,
8         )
9         self.generate_random_communication_cost(cpu_count)
10
11 def generate_random_communication_cost(self, cpu_count: int):
12     self.communication_cost.clear()
13     for child_id in self.children:
14         costs = [[0] * cpu_count for _ in range(cpu_count)]
15         for cpu1 in range(cpu_count):
16             for cpu2 in range(cpu_count):
17                 if cpu1 == cpu2:
18                     continue
19                 costs[cpu1][cpu2] = random.randint(5, 25)
20     self.communication_cost[child_id] = costs

```

## قسمت دوم - HEFT

در ابتدا با توجه به فایل‌های PDF که آقای طوقانی برای بچه‌ها ایمیل کرده بودند باید الگوریتم HEFT را پیاده سازی کنیم به جای MinMax. برای این کار نیاز است که در ابتدا rank هر کدام از تسک‌های DAG را بدست بیاوریم. برای این کار کد زیر را نوشتیم:

```
1  @staticmethod
2  def rank(tasks: dict[int, Task]) -> list[tuple[float, Task]]:
3      # task-id -> rank
4      task_ranks: dict[int, float] = {}
5      for id in range(1, len(tasks) + 1):
6          if id in task_ranks:
7              continue
8          task_ranks[id] = HEFT.calculate_rank(tasks, task_ranks, id)
9      result: list[tuple[float, Task]] = []
10     for id, rank in sorted(
11         task_ranks.items(), key=lambda item: item[1], reverse=True
12     ):
13         result.append((rank, tasks[id]))
14     return result
15
16     @staticmethod
17     def calculate_rank(
18         tasks: dict[int, Task], task_ranks: dict[int, float], wanted:
19         int
20     ) -> float:
21         if wanted in task_ranks:
22             return task_ranks[wanted]
23         current_task = tasks[wanted]
24         if len(current_task.children) == 0:
25             return current_task.average_computation()
26         current_rank = current_task.average_computation() + max(
27             map(
28                 lambda child_task_id: current_task.average_communication
29                 (child_task_id)
30                 + HEFT.calculate_rank(tasks, task_ranks, child_task_id),
31                 current_task.children,
32             )
33         )
34         task_ranks[wanted] = current_rank
35         print(f"{wanted} -> {current_rank}")
36         return current_rank
```

به صورت خلاصه کاری که انجام می‌دهیم این است که تابع calculate\_rank را بر روی اولین تسک DAG صدا می‌زنیم و در نتیجه به صورت بازگشتی تمامی rank‌ها بدست می‌آید و آن را در آرایه‌ای که بر اساس rank مرتب شده است قرار می‌دهیم. در ادامه نیز تابع زمان بندی اصلی را باید تعریف کنیم. اکثر ایده‌هایی که گرفتیم از این پروژه برداشته شده بود. این تابع را در زیر مشاهده می‌کنید:

```

1  @staticmethod
2  def is_interval_occupied_in_time(
3      intervals: list[tuple[int, int]], interval: tuple[int, int]
4  ) -> bool:
5      # https://stackoverflow.com/a/3269471/4213397
6      return any(
7          map(lambda i: i[0] <= interval[1] and interval[0] <= i[1],
8              intervals)
9      )
10
11 @staticmethod
12 def find_gap(
13     scheduled_tasks: dict[int, ScheduledTask],
14     cpu_id: int,
15     fastest_start_time: int,
16     computation_cost: int,
17 ) -> int:
18     """
19     Finds the first time which we can schedule a task on a specific
20     core
21     """
22     occupied_intervals = sorted(
23         map(
24             lambda task: (task.start_time, task.
25                 computation_finish_time),
26             filter(
27                 lambda item: item.ran_cpu_id == cpu_id,
28                 scheduled_tasks.values()
29             ),
30         ),
31         key=lambda item: item[0], # sort by start time
32     )
33     candidate_start_time = fastest_start_time
34     # This is a horrible way to do it but whatever
35     while HEFT.is_interval_occupied_in_time(
36         occupied_intervals,
37         (candidate_start_time, candidate_start_time +
38             computation_cost),
39     ):
40         candidate_start_time += 1
41     return candidate_start_time
42
43 @staticmethod
44 def schedule(tasks: dict[int, Task], cpus: int) -> dict[int,
45     ScheduledTask]:
46     ranks = HEFT.rank(tasks)
47     # task_id -> task
48     scheduled_tasks: dict[int, ScheduledTask] = {}

```

```

43     for task in map(lambda rank: rank[1], ranks):
44         # cpu_id -> (start, finish) for each CPU core
45         cpu_runtimes: list[tuple[int, int]] = [(0, 0)] * cpus
46         for cpu_id in range(cpus):
47             processor_ready = 0 # when does this core can become
available for scheduling this task
48             for parent_id in task.fathers:
49                 assert parent_id in scheduled_tasks # sanity check
50                 communication_cost = tasks[parent_id].
communication_cost[task.id][
51                     scheduled_tasks[parent_id].ran_cpu_id
52                     ][cpu_id]
53                 start_delay = (
54                     communication_cost
55                     + scheduled_tasks[parent_id].
computation_finish_time
56                     )
57                 processor_ready = max(processor_ready, start_delay)
58                 # Now calculate when we can schedule this task
59                 cpu_start_time = HEFT.find_gap(
60                     scheduled_tasks,
61                     cpu_id,
62                     processor_ready,
63                     task.computation_times[cpu_id],
64                 )
65                 cpu_runtimes[cpu_id] = (
66                     cpu_start_time,
67                     cpu_start_time + task.computation_times[cpu_id],
68                 )
69                 # Now check what CPU yields the fastest one
70                 best_cpu_id = min(
71                     range(len(cpu_runtimes)), key=lambda x: cpu_runtimes[x
] [1]
72                 )
73                 scheduled_tasks[task.id] = ScheduledTask(
74                     task, best_cpu_id, cpu_runtimes[best_cpu_id][0]
75                 )
76         return scheduled_tasks
77

```

تابع اصلی در اینجا schedule است که تعداد تسک‌ها و CPU core ها را می‌گیرد و زمان بندی را انجام می‌دهد. خروجی تابع یک دیکشنری از task\_id به زمان‌هایی است که یک تسک شروع به انجام شدن می‌کند و پایان میابد و همچنین CPU که بر روی آن اجرا شده است. تعریف کلاس ScheduledTask را نیز می‌توانید در زیر ببینید:

```

1 class ScheduledTask:
2     def __init__(self, task: Task, cpu_id: int, start_time: int):
3         self.task_id = task.id
4         self.start_time = start_time
5         self.computation_finish_time = start_time + task.

```



```
computation_times[cpu_id]
self.ran_cpu_id = cpu_id
```

6

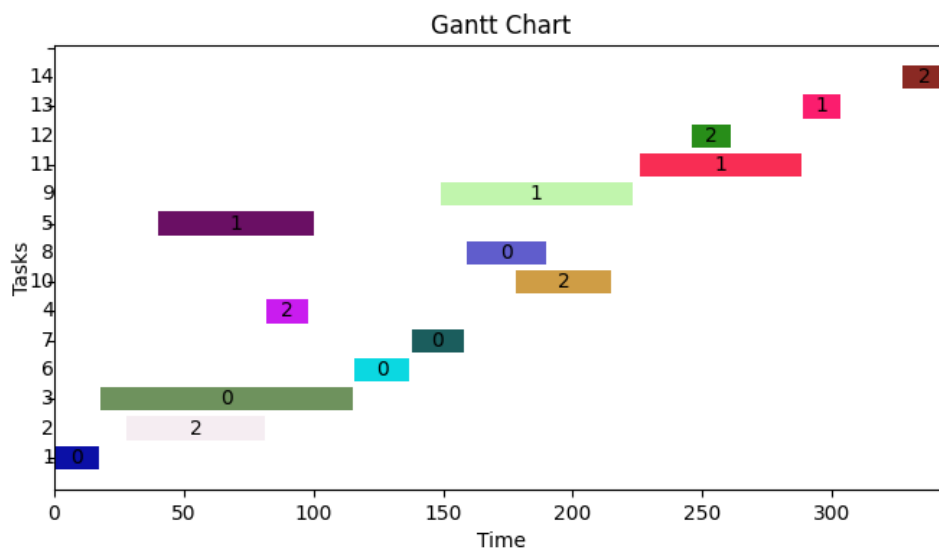
7

در تابع زمانبندی که ما نوشتیم عملاً برای هر واحد زمانی شبیه سازی انجام نمی‌شود. بلکه برای هر تسک به ترتیب rank حساب می‌شود که چه زمانی می‌تواند اجرا شود. این روش به نظر ما خیلی بهتر از زمان بندی برای هر ثانیه بود چرا که تسک‌های ما preemptive نیستند.

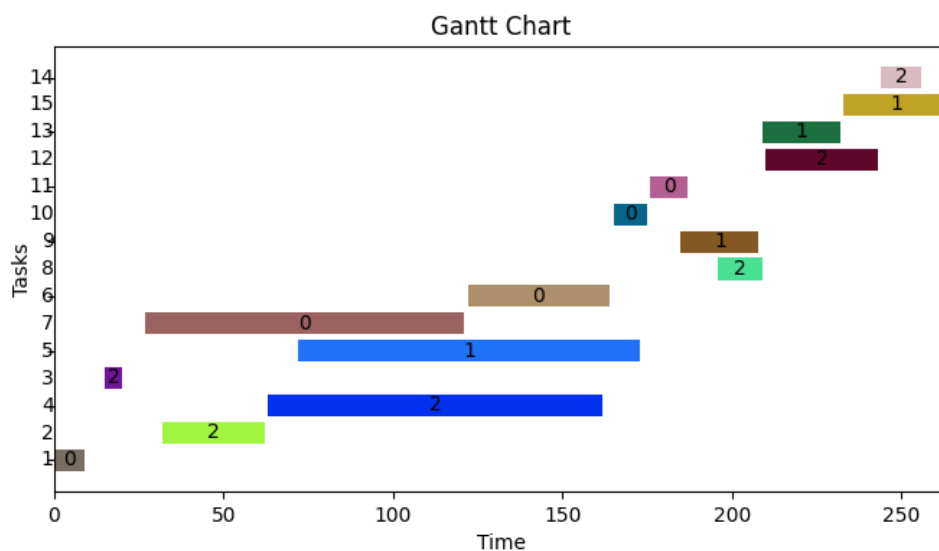
در داخل تابع schedule کاری که می‌کنیم این است که در ابتدا تسک‌ها را بر اساس رنک سورت می‌کنیم و هر تسک را چک می‌کنیم که زمانی که کارش در هر هسته تمام می‌شود کی است و در نهایت کمترین زمان را انتخاب می‌کنیم و آن تسک را بر روی آن هسته زمان بندی می‌کنیم. دقت کنید که همان طور که گفته شد در صورتی که بخواهیم اطلاعات را از پدرهای این تسک که بر روی هسته‌ی دیگری اجرا شدند بر روی هسته‌ای دیگر بیاوریم مجبور هستیم که کمی سربار پرداخت کنیم و این موضوع در زمان شروع تسک‌های فرزند لحاظ شده است.

### قسمت سوم - تست

در این قسمت دو ورودی کوچک و مختلف را تست و زمانبندی می‌کنیم. در ابتدا به کمک gaussian elimination گرافی با پارامتر ۵ می‌سازیم و آن را زمان بندی می‌کنیم. نتیجه‌ی آن در شکل زیر آمده است:



دقت کنید که محور عمودی نشان دهنده‌ی شماره تسکی است که در حال اجرا است و اعداد نوشته شده بر روی هر کدام از تسک‌ها هسته‌ای است که در آن اجرا می‌شود تسک. به عنوان مثال در شکل بالا همان طور که مشاهده می‌شود تسک ۳ بلافاصله بعد از اجرای تسک ۱ در همان هسته‌ی ۰ اجرا می‌شود چرا که نیازی به انتقال داده وجود ندارد. همچنین تسک شماره ۲ نیز با کمی تأخیر اجرا می‌شود که ناشی از انتقال داده است. در ادامه Fast Fourier Transform با پارامتر ۴ را اجرا می‌کنیم. شکل آن در زیر آمده است:



در ابتدا که ما این عکس را دیدیم فکر کردیم که زمان بند ما باگ دارد چرا که مقدار زیادی stack time بین تسک ۱ و ۳ وجود دارد و اصلا بر روی یک هسته یکسان اجرا نمی‌شوند. اما این زمان بندی درست است چرا که اگر اجرا را بر روی هسته ۰ ادامه می‌دادیم، زمان اجرای نهایی بسیار بیشتر می‌شد و در اینجا می‌صرفید که داده‌ها را بر روی یک هسته‌ی دیگر بفرستیم و بر روی یک هسته‌ی سریع‌تر اجرا کنیم تسک بعدی را.