به نام خداوند بخشنده مهربان

سیستم‌های بی‌درنگ

دکتر صفری

مبینا حیدری - نیکا قادری
بهار ۱۴۰۴

تاریخ گزارش: ۱۱ خرداد ۱۴۰۴ | **زمانبندی با استفاده از الگوریتم $Cuckoo$** | فاز اول

فهرست مطالب

۱ مقدمه

امروزه با افزایش پیچیدگی سامانه‌های محاسباتی، استفاده از پردازنده‌های چندهسته‌ای ($multi-core\ processors$) به امری رایج بدل شده است. این پردازنده‌ها امکان اجرای موازی چندین وظیفه را فراهم می‌کنند که این امر به‌ویژه در سامانه‌های بی‌درنگ ($real-time\ systems$) از اهمیت بالایی برخوردار است. در این سامانه‌ها، صحت عملکرد نه تنها به درستی نتایج محاسباتی، بلکه به زمان ارائه این نتایج نیز وابسته است و هرگونه تأخیر در اجرای وظایف می‌تواند منجر به نتایج نامطلوب یا حتی فاجعه‌بار گردد.

یکی از چالش‌های اساسی در بهره‌برداری مؤثر از پردازنده‌های چندهسته‌ای، زمان‌بندی ($scheduling$) بهینه وظایف است. هدف اصلی زمان‌بندی، تخصیص وظایف به هسته‌های پردازشی و تعیین ترتیب اجرای آن‌ها به گونه‌ای است که ضمن رعایت قیود زمانی هر وظیفه (مانند موعد زمانی یا ($deadline$))، معیارهای عملکردی سامانه نیز بهبود یابند. زمان‌بندی نامناسب می‌تواند منجر به گلوگاه‌های پردازشی، افزایش زمان پاسخ و کاهش کارایی کلی سیستم شود.

این پروژه به بررسی و پیاده‌سازی روش‌های زمان‌بندی وظایف بی‌درنگ متناوب بر روی پردازنده‌های چندهسته‌ای همگن ($homogeneous\ cores$) می‌پردازد. اهداف اصلی این پروژه شامل کمینه‌سازی زمان تکمیل کل وظایف یا ($makespan$)، متعادل‌سازی بار ($load\ balancing$) میان هسته‌ها، و تضمین کیفیت خدمات ($Quality\ of\ Service - QoS$) سامانه با رعایت موعدهای زمانی وظایف است. دستیابی به این اهداف در محیط‌های چندهسته‌ای پیچیدگی‌های خاص خود را دارد، زیرا علاوه بر تصمیم‌گیری در مورد اینکه کدام وظیفه چه زمانی اجرا شود، باید تصمیم گرفت که هر وظیفه بر روی کدام هسته اجرا گردد.

در فاز اول این پروژه، که مبنای گزارش حاضر است، ابتدا مجموعه‌ای از وظایف مصنوعی با دوره تناوب‌های مختلف و با بهره‌گیری از الگوریتم $(UUNIFAST)$ تولید می‌شوند. این وظایف دارای مشخصاتی نظیر زمان اجرا $(execution\ time)$، دوره تناوب $(period)$، موعد زمانی $(deadline)$ (که در این پژوهش برابر با دوره تناوب در نظر گرفته شده) و میزان بهره‌وری $(utilization)$ هستند. سپس، برای زمان‌بندی این وظایف بر روی هسته‌های پردازشی، از الگوریتم ژنتیک $(Genetic\ Algorithm)$ استفاده شده است. الگوریتم ژنتیک یک روش جستجوی فراابتکاری $(metaheuristic)$ قدرتمند است که با الهام از فرآیندهای تکامل طبیعی و انتخاب اصلح، به دنبال یافتن راه‌حل‌های بهینه یا نزدیک به بهینه برای مسائل پیچیده تخصیص و زمان‌بندی است.

گزارش حاضر به تشریح دقیق مراحل انجام‌شده در فاز اول پروژه، شامل نحوه تولید وظایف با الگوریتم $(UUNIFAST)$ و پیاده‌سازی الگوریتم ژنتیک برای زمان‌بندی آن‌ها بر روی سامانه‌های با تعداد هسته‌های ۸، ۱۶ و ۳۲ و با میزان بهره‌وری هدف برای هر هسته برابر با ۲۵.۰، ۵.۰، ۷۵.۰ و ۱ می‌پردازد. همچنین، نتایج به‌دست‌آمده از شبیه‌سازی‌ها، شامل نمودارهای کیفیت خدمات وظایف و سامانه، $(makespan)$، توزیع بهره‌وری هسته‌ها و قابلیت زمان‌بندی سامانه، ارائه و به‌تفصیل تحلیل خواهند شد.

مخزن گیتهاب این پروژه در آدرس https://github.com/NikaGhaderi/Cuckoo-Search-Task-Scheduling قابل مشاهده است.

۲ توضیح کد

۱.۲ تابع تولید وظایف (generate_tasks)

این تابع مسئولیت اصلی تولید مجموعه‌ای از وظایف بی‌درنگ متناوب را بر عهده دارد. اساس کار این تابع، استفاده از الگوریتم $(UUNIFAST)$ برای تقسیم عادلانه مجموع بهره‌وری خواسته شده بین تعداد مشخصی از وظایف است. پس از تعیین بهره‌وری هر وظیفه، سایر مشخصات آن مانند دوره تناوب، زمان اجرا و موعد زمانی نیز تعیین می‌گردند. در ادامه، بخش‌های مختلف این تابع به تفصیل و به صورت خط به خط تشریح می‌شوند.

۱.۱.۲ تعریف تابع و پارامترهای ورودی

```
def generate_tasks(num_tasks, total_utilization):
```

در این بخش، تابع generate_tasks تعریف می‌شود. این تابع دو آرگومان ورودی اصلی دریافت می‌کند:

- num_tasks: این پارامتر یک عدد صحیح است که تعداد کل وظایفی که باید توسط تابع تولید شوند را مشخص می‌کند.
- total_utilization: این پارامتر یک عدد اعشاری است که نشان‌دهنده مجموع بهره‌وری $(utilization)$ است که باید بین تمام num_tasks وظیفه تقسیم شود.

۲.۱.۲ پیاده‌سازی الگوریتم $(UUNIFAST)$ برای تقسیم بهره‌وری

در این بخش از کد، الگوریتم $(UUNIFAST)$ برای تولید بردار بهره‌وری وظایف پیاده‌سازی شده است.

```
    utilizations = []
    remaining_util = total_utilization

    for i in range(1, num_tasks):
        next_util = remaining_util * random.random() ** (1 / (num_tasks - i))
        utilizations.append(remaining_util - next_util)
        remaining_util = next_util
    utilizations.append(remaining_util)
```

مراحل اجرای این بخش به شرح زیر است:

- utilizations = []: یک لیست $(list)$ خالی با نام utilizations ایجاد می‌شود. این لیست در نهایت حاوی میزان بهره‌وری تخصیص‌یافته به هر یک از وظایف خواهد بود.
- total_utilization = remaining_util:

  متغیری با نام remaining_util تعریف شده و مقدار اولیه آن برابر با total_utilization (کل بهره‌وری مورد نظر) قرار داده می‌شود. این متغیر در هر مرحله از حلقه، میزان بهره‌وری باقیمانده برای تخصیص به وظایف بعدی را نگهداری می‌کند.
- for i in range(1, num_tasks): یک حلقه for برای تولید بهره‌وری (num_tasks − ۱) وظیفه اول اجرا می‌شود. متغیر i از ۱ تا ۱ − $texttt{num\_tasks}$ تغییر می‌کند.
- next_util = remaining_util * random.random() ** (1 / (num_tasks - i)):

  این خط، بخش کلیدی الگوریتم $(UUNIFAST)$ است.
  - random.random(): یک عدد اعشاری تصادفی در بازه نیمه‌باز [۰/۰ ,۱/۰) تولید می‌کند.
  - (1 / (num_tasks − i)): عدد تصادفی تولید شده به توان کسری $frac$۱num_tasks − i می‌رسد.
  - remaining_util * ...:

  نتیجه عبارت قبلی در remaining_util ضرب می‌شود. مقدار next_util در اینجا نشان‌دهنده میزان بهره‌وری است که *پس از* تخصیص بهره‌وری به وظیفه جاری (i)، برای سایر وظایف باقی خواهد ماند.
- utilizations.append(remaining_util - next_util): میزان بهره‌وری وظیفه جاری (i) برابر است با تفاضل بهره‌وری باقیمانده *قبل* از تخصیص به این وظیفه (remaining_util) و بهره‌وری باقیمانده *بعد* از تخصیص به این وظیفه (next_util). این مقدار محاسبه شده و به انتهای لیست $(list)$ utilizations اضافه می‌شود.
- remaining_util = next_util: مقدار بهره‌وری باقیمانده، برای استفاده در تکرار بعدی حلقه و برای تخصیص به وظیفه بعدی، به‌روزرسانی می‌شود.
- utilizations.append(remaining_util): پس از اتمام حلقه for (یعنی پس از تخصیص بهره‌وری به (num_tasks − ۱) وظیفه)، کل بهره‌وری باقیمانده در متغیر remaining_util به آخرین وظیفه (وظیفه $texttt{num_tasks}$ـام) اختصاص داده شده و به لیست $(list)$ utilizations اضافه می‌شود.

۳.۱.۲ ایجاد لیست وظایف و تعیین مشخصات هر وظیفه

پس از آنکه بهره‌وری هر وظیفه در لیست ($list$) `utilizations` مشخص شد، در این بخش سایر مشخصات هر وظیفه تعیین و ساختار نهایی وظایف ایجاد می‌شود.

```
tasks = []
for i, util in enumerate(utilizations):
    period = random.choice([10, 20, 40, 50, 100])  # Common periods
    execution = util * period
    deadline = period  # Implicit deadline
    tasks.append({
        'id': i,
        'execution': execution,
        'period': period,
        'deadline': deadline,
        'utilization': util
    })
```

جزئیات این بخش به شرح زیر است:

- $tasks = []$: یک لیست ($list$) خالی دیگر با نام `tasks` برای نگهداری مجموعه‌ای از دیکشنری‌های ($dictionary$) وظایف ایجاد می‌شود. هر دیکشنری ($dictionary$) در این لیست، معرف یک وظیفه و تمامی مشخصات آن خواهد بود.
- $for\ i,\ util\ in\ enumerate(utilizations)$: یک حلقه `for` بر روی لیست ($list$) `utilizations` (که حاوی بهره‌وری‌های محاسبه‌شده برای هر وظیفه است) اجرا می‌شود. تابع `enumerate` به ازای هر عضو لیست، هم اندیس آن عضو (`i`) و هم مقدار آن عضو (`util`) را برمی‌گرداند. اندیس `i` به عنوان شناسه (id) وظیفه استفاده خواهد شد.
- $period = random.choice([۱۰,\ ۲۰,\ ۴۰,\ ۵۰,\ ۱۰۰])$: دوره تناوب ($period$) برای وظیفه جاری به صورت تصادفی از میان یک لیست ($list$) از مقادیر متداول و از پیش تعریف‌شده ($[۱۰, ۲۰, ۴۰, ۵۰, ۱۰۰]$ واحد زمانی) انتخاب می‌شود.
- $execution = util * period$: زمان اجرای ($execution\ time$) وظیفه ($C_i$) با استفاده از رابطه اصلی بهره‌وری $U_i = C_i/T_i$ محاسبه می‌شود. بنابراین، $C_i = U_i \times T_i$. در اینجا، `util` همان U_i و `period` همان T_i است.
- $deadline = period$: موعد زمانی ($deadline$) برای هر وظیفه، برابر با دوره تناوب ($period$) آن در نظر گرفته می‌شود. این حالت به عنوان موعد زمانی ضمنی یا ($implicit\ deadline$) شناخته می‌شود.
- $tasks.append(\{...\})$: یک دیکشنری ($dictionary$) جدید برای وظیفه جاری ایجاد می‌شود. این دیکشنری ($dictionary$) شامل پنج زوج کلید–مقدار است:
  - i : $'id'$: شناسه یکتای وظیفه.
  - $execution$: $'execution'$: زمان اجرای محاسبه‌شده برای وظیفه.
  - $period$: $'period'$: دوره تناوب تخصیص‌یافته به وظیفه.
  - $deadline$: $'deadline'$: موعد زمانی وظیفه.
  - $util$: $'utilization'$: میزان بهره‌وری وظیفه.

  این دیکشنری ($dictionary$) سپس به انتهای لیست ($list$) `tasks` اضافه می‌شود.

۴.۱.۲ خروجی تابع

```
return tasks
```

در نهایت، تابع `generate_tasks` لیست ($list$) `tasks` را که حاوی تمامی وظایف تولیدشده به همراه مشخصات کامل آن‌ها (به صورت دیکشنری ($dictionary$)) است، به عنوان مقدار خروجی برمی‌گرداند.

۲.۲ کلاس زمان‌بند ژنتیک ($GeneticScheduler$)

این کلاس، تمامی منطق و عملیات مربوط به الگوریتم ژنتیک را برای مسئله زمان‌بندی وظایف بر روی هسته‌های پردازنده⁉ می‌کند. این الگوریتم سعی دارد با الهام از فرآیندهای تکاملی طبیعی، یک تخصیص بهینه از وظایف به هسته‌ها را پیدا کند به طوری که بهره‌وری هسته‌ها متوازن شده و سربار اضافی به حداقل برسد.

۱.۲.۲ متد سازنده ($\_\_init\_\_$)

متد سازنده ($constructor$) کلاس، مسئول مقداردهی اولیه به پارامترها و ویژگی‌های اصلی الگوریتم ژنتیک است.

```
class GeneticScheduler:
    def __init__(self, tasks, num_cores, pop_size=50, elite=0.2, mutation_rate=0.1, generations=100):
        self.tasks = tasks
        self.num_cores = num_cores
        self.pop_size = pop_size
        self.elite = int(elite * pop_size)
        self.mutation_rate = mutation_rate
        self.generations = generations
```

توضیحات خط به خط:

- $class GeneticScheduler$: تعریف کلاس با نام $GeneticScheduler$.
- $def \_\_init\_\_(self, tasks, num\_cores, pop\_size = ۵۰, elite = ۰/۲, mutation\_rate = ۰/۱, generations =$ ۱۰۰): تعریف متد سازنده $\_\_init\_\_$.
  - $self$: ارجاع به نمونه فعلی کلاس.
  - $tasks$: لیستی از دیکشنری‌ها که هر دیکشنری اطلاعات یک وظیفه را در خود دارد.
  - num_cores: تعداد هسته‌های پردازنده موجود.
  - $pop\_size = ۵۰$: اندازه جمعیت اولیه کروموزوم‌ها، با مقدار پیش‌فرض ۵۰.
  - $elite = ۰/۲$: درصد نخبگان جمعیت که مستقیماً به نسل بعد منتقل می‌شوند، با مقدار پیش‌فرض ۰/۲ (یعنی ۲۰ درصد).
  - $mutation\_rate = ۰/۱$: نرخ جهش برای هر ژن در کروموزوم، با مقدار پیش‌فرض ۰/۱ (یعنی ۱۰ درصد).
  - $generations = ۱۰۰$: تعداد نسل‌هایی که الگوریتم اجرا خواهد شد، با مقدار پیش‌فرض ۱۰۰.
- $self.tasks = tasks$: لیست وظایف ورودی در متغیر نمونه $self.tasks$ ذخیره می‌شود.
- $self.num\_cores = num\_cores$: تعداد هسته‌ها در متغیر نمونه $self.num\_cores$ ذخیره می‌شود.
- $self.pop\_size = pop\_size$: اندازه جمعیت در متغیر نمونه $self.pop\_size$ ذخیره می‌شود.
- $self.elite = int(elite * pop\_size)$: تعداد دقیق نخبگان با ضرب درصد نخبگان در اندازه جمعیت و تبدیل به عدد صحیح (int) محاسبه و در $self.elite$ ذخیره می‌شود.
- $self.mutation\_rate = mutation\_rate$: نرخ جهش در متغیر نمونه $self.mutation\_rate$ ذخیره می‌شود.
- $self.generations = generations$: تعداد نسل‌ها در متغیر نمونه $self.generations$ ذخیره می‌شود.

۲.۲.۲ متد ایجاد جمعیت اولیه ($initialize\_population$)

این متد مسئول ایجاد جمعیت اولیه از راه‌حل‌های تصادفی (کروموزوم‌ها) است.

```
    def initialize_population(self):
        return [np.random.randint(0, self.num_cores, len(self.tasks))
                for _ in range(self.pop_size)]
```

توضیحات:

- $def initialize\_population(self)$: تعریف متد.
- یک لیست ($listcomprehension$) برای ساخت جمعیت استفاده می‌شود. این لیست شامل $self.pop\_size$ تعداد کروموزوم است.
- هر کروموزوم با استفاده از $np.random.randint(۰, self.num\_cores, len(self.tasks))$ ساخته می‌شود:
  - این دستور یک آرایه ($array$) از اعداد صحیح تصادفی تولید می‌کند.
  - ۰: حد پایین (شامل) برای اعداد تصادفی (اندیس اولین هسته).
  - $self.num\_cores$: حد بالا (غیر شامل) برای اعداد تصادفی (تعداد کل هسته‌ها).
  - $len(self.tasks)$: طول آرایه، که برابر با تعداد وظایف است. هر عنصر آرایه نشان می‌دهد که وظیفه متناظر با آن اندیس، به کدام هسته تخصیص داده شده است.
- $return[...]$: لیست جمعیت اولیه برگردانده می‌شود.

۳.۲.۲ متد محاسبه برازندگی ($fitness$)

این متد میزان خوب بودن یک راه‌حل (کروموزوم) را ارزیابی می‌کند. هدف، حداقل کردن سربار و حداکثر کردن توازن بار بین هسته‌ها است.

```
def fitness(self, chromosome):
    core_utils = np.zeros(self.num_cores)
    for task_idx, core_idx in enumerate(chromosome):
        core_utils[core_idx] += self.tasks[task_idx]['utilization']

    overload_penalty = sum(max(0, util - 1) * 100 for util in core_utils)
    balance = np.std(core_utils) * 10
    return 1 / (1 + overload_penalty + balance)
```

توضیحات:

- $def\ fitness(self, chromosome)$: تعریف متد که یک کروموزوم ورودی می‌گیرد.
- $core\_utils = np.zeros(self.num\_cores)$: یک آرایه ($array$) به نام $core\_utils$ با طول برابر با تعداد هسته‌ها ایجاد و با صفر مقداردهی می‌شود. این آرایه برای ذخیره مجموع بهره‌وری وظایف تخصیص‌یافته به هر هسته استفاده می‌شود.
- $for\ task\_idx, core\_idx\ in\ enumerate(chromosome)$: حلقه‌ای که بر روی ژن‌های کروموزوم (تخصیص هر وظیفه به یک هسته) پیمایش می‌کند. $task\_idx$ اندیس وظیفه و $core\_idx$ شماره هسته‌ای است که وظیفه $task\_idx$ به آن تخصیص داده شده.
- $core\_utils[core\_idx] += self.tasks[task\_idx]['utilization']$:
  بهره‌وری وظیفه $task\_idx$ (که از $self.tasks[task\_idx]['utilization']$ خوانده می‌شود) به بهره‌وری کل هسته $core\_idx$ اضافه می‌شود.
- $overload\_penalty = sum(max(0, util - 1) * 100\ for\ util\ in\ core\_utils)$: محاسبه جریمه برای هسته‌هایی که بهره‌وری آن‌ها از ۱ (یا ۱۰۰ درصد) بیشتر شده است.
  - $max(0, util - 1)$: اگر بهره‌وری ($util$) یک هسته بیشتر از ۱ باشد، مقدار اضافه بار ($util - 1$) را برمی‌گرداند؛ در غیر این صورت ۰.
  - $100*$: مقدار اضافه بار در یک ضریب (۱۰۰) ضرب می‌شود تا جریمه سنگین‌تر شود.
  - $sum(...)$: مجموع جرایم برای تمام هسته‌ها محاسبه می‌شود.
- $balance = np.std(core\_utils)*10$: محاسبه معیاری برای توازن بار. این معیار بر اساس انحراف استاندارد ($standard\ deviation$) بهره‌وری هسته‌ها ($np.std(core\_utils)$) است. هرچه انحراف استاندارد کمتر باشد، بار متوازن‌تر است. این مقدار در ۱۰ ضرب شده تا تأثیر بیشتری در تابع برازندگی داشته باشد.
- $return 1/(1 + overload\_penalty + balance)$: مقدار برازندگی برگردانده می‌شود. این فرمول به گونه‌ای است که مقادیر بالاتر برازندگی نشان‌دهنده راه‌حل‌های بهتر (جریمه کمتر و توازن بهتر) هستند. افزودن ۱ به مخرج از تقسیم بر صفر جلوگیری می‌کند.

۴.۲.۲ متد انتخاب والدین ($select\_parents$)

این متد والدین را برای تولید نسل بعدی از میان جمعیت فعلی بر اساس برازندگی آن‌ها انتخاب می‌کند (روش چرخ رولت).

```
def select_parents(self, population, fitnesses):
    total_fitness = sum(fitnesses)
    probs = [f / total_fitness for f in fitnesses]
    parents_indices = np.random.choice(
        range(len(population)),
        size=len(population) - self.elite,
        p=probs
    )
    return [population[i] for i in parents_indices]
```

توضیحات:

- $def select\_parents(self, population, fitnesses)$: تعریف متد که جمعیت فعلی و لیست برازندگی‌های متناظر را به عنوان ورودی می‌گیرد.
- $total\_fitness = sum(fitnesses)$: مجموع کل برازندگی تمام کروموزوم‌های جمعیت محاسبه می‌شود.
- $probs = [f/total\_fitness \ for \ f \ in \ fitnesses]$: احتمال انتخاب هر کروموزوم به عنوان والد محاسبه می‌شود. این احتمال متناسب با برازندگی نسبی آن کروموزوم به کل برازندگی جمعیت است.
- $parents\_indices = np.random.choice(...)$: اندیس‌های والدین با استفاده از تابع $np.random.choice$ انتخاب می‌شوند.
  - $range(len(population))$: مجموعه‌ای از اندیس‌های ممکن (از ۰ تا تعداد کروموزوم‌ها منهای ۱).
  - $size = len(population) - self.elite$: تعداد والدینی که باید انتخاب شوند، برابر است با اندازه جمعیت منهای تعداد نخبگان (زیرا نخبگان مستقیماً به نسل بعد می‌روند).
  - $p = probs$: انتخاب‌ها بر اساس احتمالات محاسبه‌شده در $probs$ انجام می‌شود (انتخاب با جایگزینی).
- $return [population[i] \ for \ i \ in \ parents\_indices]$: لیست کروموزوم‌های والد انتخاب شده برگردانده می‌شود.

۵.۲.۲ متد ترکیب ($crossover$)

این متد عملگر ترکیب (تولید فرزند از والدین) را پیاده‌سازی می‌کند. در اینجا از ترکیب تک نقطه‌ای استفاده شده است.

```
def crossover(self, parent1, parent2):
    point = random.randint(1, len(parent1) - 1)
    child1 = np.concatenate((parent1[:point], parent2[point:]))
    child2 = np.concatenate((parent2[:point], parent1[point:]))
    return child1, child2
```

توضیحات:

- $def crossover(self, parent۱, parent۲)$: تعریف متد که دو کروموزوم والد را به عنوان ورودی می‌گیرد.
- $point = random.randint(۱, len(parent۱) - ۱)$: یک نقطه برش تصادفی در طول کروموزوم والد انتخاب می‌شود. این نقطه بین ژن اول و آخر (غیر شامل) است تا اطمینان حاصل شود که هر دو والد در تولید فرزند مشارکت دارند.
- $child۱ = np.concatenate((parent۱[: point], parent۲[point :]))$: فرزند اول با ترکیب بخش اول از $parent۱$ (تا قبل از نقطه برش) و بخش دوم از $parent۲$ (از نقطه برش تا انتها) ایجاد می‌شود. تابع $np.concatenate$ برای اتصال این دو بخش استفاده می‌شود.
- $child۲ = np.concatenate((parent۲[: point], parent۱[point :]))$: فرزند دوم با ترکیب بخش اول از $parent۲$ و بخش دوم از $parent۱$ ایجاد می‌شود.
- $return child۱, child۲$: دو فرزند تولید شده برگردانده می‌شوند.

۶.۲.۲ متد تکامل ($evolve$)

این متد اصلی، فرآیند الگوریتم ژنتیک را برای چندین نسل اجرا می‌کند تا به یک راه‌حل بهینه یا نزدیک به بهینه دست یابد.

```
def evolve(self):
    population = self.initialize_population()

    for _ in range(self.generations):
        fitnesses = [self.fitness(chromo) for chromo in population]

        elite_indices = np.argsort(fitnesses)[-self.elite:]
        new_population = [population[i] for i in elite_indices]

        parents = self.select_parents(population, fitnesses)
        random.shuffle(parents)

        for i in range(0, len(parents), 2):
            if i + 1 < len(parents):
                child1, child2 = self.crossover(parents[i], parents[i + 1])
                new_population += [self.mutate(child1), self.mutate(child2)]

        population = new_population

    fitnesses = [self.fitness(chromo) for chromo in population]
    return population[np.argmax(fitnesses)]
```

توضیحات:

- $def evolve(self)$: تعریف متد اصلی اجرای الگوریتم.
- $population = self.initialize\_population()$: جمعیت اولیه با فراخوانی متد $self.initialize\_population()$ ایجاد می‌شود.
- $for \_ \ in \ range(self.generations)$: حلقه اصلی الگوریتم که به تعداد $self.generations$ نسل تکرار می‌شود.
  - $fitnesses = [self.fitness(chromo) \ for \ chromo \ in \ population]$: برازندگی برای تمام کروموزوم‌های جمعیت فعلی با استفاده از متد $self.fitness()$ محاسبه می‌شود.
  - $elite\_indices = np.argsort(fitnesses)[-self.elite :]$: اندیس‌های کروموزوم‌های نخبه (با بالاترین برازندگی) شناسایی می‌شوند. $np.argsort(fitnesses)$ اندیس‌ها را بر اساس برازندگی مرتب‌شده برمی‌گرداند و $[-self.elite :]$ بخش آخر آن (نخبگان) را انتخاب می‌کند.
  - $new\_population = [population[i] \ for \ i \ in \ elite\_indices]$: کروموزوم‌های نخبه مستقیماً به جمعیت نسل بعدی ($new\_population$) اضافه می‌شوند (عملگر نخبه‌گرایی یا $elitism$).
  - $parents = self.select\_parents(population, fitnesses)$:
    والدین از جمعیت فعلی با استفاده از متد $self.select\_parents()$ انتخاب می‌شوند.
  - $random.shuffle(parents)$: لیست والدین به صورت تصادفی برهم زده می‌شود تا جفت‌گیری برای ترکیب به صورت تصادفی انجام شود.
  - $for \ i \ in \ range(۰, len(parents), ۲)$: حلقه‌ای برای انتخاب جفت والدین و تولید فرزندان. حلقه با گام ۲ حرکت می‌کند.
    - $if \ i + ۱ < len(parents)$: اطمینان از وجود والد دوم برای تشکیل زوج.
      - $child۱, child۲ = self.crossover(parents[i], parents[i+۱])$: دو فرزند از جفت والدین $parents[i]$ و $parents[i+۱]$ با استفاده از متد $self.crossover()$ تولید می‌شوند.
      - $new\_population += [self.mutate(child۱), self.mutate(child۲)]$: هر دو فرزند تولید شده ابتدا با متد $self.mutate()$ جهش داده شده و سپس به جمعیت جدید ($new\_population$) آضافه می‌شوند.
  - $population = new\_population$: جمعیت جدید ایجاد شده، جایگزین جمعیت فعلی برای نسل بعدی می‌شود.
- $fitnesses = [self.fitness(chromo) \ for \ chromo \ in \ population]$: پس از اتمام تمام نسل‌ها، برازندگی کروموزوم‌های جمعیت نهایی محاسبه می‌شود.
- $return \ population[np.argmax(fitnesses)]$: بهترین کروموزوم (با بالاترین برازندگی) از جمعیت نهایی انتخاب شده و به عنوان نتیجه الگوریتم برگردانده می‌شود. $np.argmax(fitnesses)$ اندیس کروموزومی که بیشترین برازندگی را دارد، برمی‌گرداند.

۷.۲.۲ متد جهش ($mutate$)

این متد عملگر جهش را بر روی یک کروموزوم اعمال می‌کند تا تنوع ژنتیکی در جمعیت حفظ شود و از همگرایی زودرس جلوگیری گردد.

```
def mutate(self, chromosome):
    for i in range(len(chromosome)):
        if random.random() < self.mutation_rate:
            chromosome[i] = random.randint(0, self.num_cores - 1)
    return chromosome
```

توضیحات:

- : $def mutate(self, chromosome)$: تعریف متد که یک کروموزوم را به عنوان ورودی می‌گیرد.
- : $for\ i\ in\ range(len(chromosome))$: حلقه‌ای که بر روی تمام ژن‌های کروموزوم (تخصیص هر وظیفه) پیمایش می‌کند.
- : $if random.random() < self.mutation\_rate$: برای هر ژن، یک عدد تصادفی در بازه $[۰, ۱)$ تولید می‌شود. اگر این عدد کوچکتر از نرخ جهش ($self.mutation\_rate$) باشد، آنگاه ژن جهش پیدا می‌کند.
- $chromosome[i] = random.randint(۰, self.num\_cores - ۱)$: در صورت وقوع جهش، مقدار ژن iـام (یعنی هسته تخصیص‌یافته به وظیفه iـام) با یک شماره هسته تصادفی جدید (بین ۰ و $self.num\_cores - ۱$) جایگزین می‌شود.
- $return chromosome$: کروموزوم جهش‌یافته (یا اصلی، اگر جهشی رخ نداده باشد) برگردانده می‌شود.

۳.۲ تابع محاسبه متریک‌ها ($calculate\_metrics$)

این تابع پس از اتمام فرآیند زمان‌بندی توسط الگوریتم ژنتیک (یا هر الگوریتم دیگری)، مجموعه‌ای از متریک‌های کلیدی را برای ارزیابی کیفیت تخصیص نهایی وظایف به هسته‌ها محاسبه می‌کند.

۱.۳.۲ تعریف تابع و پارامترهای ورودی

```
def calculate_metrics(tasks, assignment, num_cores):
```

تابع $calculate\_metrics$ با سه آرگومان ورودی تعریف می‌شود:

- $tasks$: یک لیست $(list)$ از دیکشنری‌های $(dictionary)$ وظایف، که هر دیکشنری $(dictionary)$ شامل مشخصات یک وظیفه مانند دوره تناوب و بهره‌وری است.
- $assignment$: یک لیست $(list)$ یا آرایه $(array)$ که نشان‌دهنده تخصیص وظایف به هسته‌ها است (همان کروموزوم بهینه). اندیس هر عنصر در این لیست $(list)$ شناسه وظیفه و مقدار آن عنصر، شناسه هسته‌ای است که آن وظیفه به آن تخصیص داده شده است.
- num_cores: تعداد کل هسته‌های پردازشی موجود در سیستم.

۲.۳.۲ محاسبه بهره‌وری هر هسته ($Core\ Utilization$)

در این بخش، مجموع بهره‌وری وظایف تخصیص داده شده به هر هسته محاسبه می‌شود.

```
    core_utils = [0] * num_cores
    for task_idx, core_idx in enumerate(assignment):
        core_utils[core_idx] += tasks[task_idx]['utilization']
```

توضیحات:

- $core\_utils = [۰] * num\_cores$: یک لیست $(list)$ با نام $core\_utils$ به طول num_cores ایجاد شده و تمام عناصر آن با مقدار اولیه ۰ مقداردهی می‌شوند. هر عنصر این لیست $(list)$ مجموع بهره‌وری یک هسته را ذخیره خواهد کرد.
- $for\ task\_idx, core\_idx\ in\ enumerate(assignment)$: یک حلقه بر روی لیست $assignment\ (list)$ اجرا می‌شود. تابع $enumerate$ برای هر عنصر، هم اندیس ($task\_idx$، که شناسه وظیفه است) و هم مقدار ($core\_idx$، که شناسه هسته تخصیص‌یافته است) را برمی‌گرداند.
- $core\_utils[core\_idx] += tasks[task\_idx]['utilization']$: بهره‌وری وظیفه $task\_idx$ (که از $tasks[task\_idx]['utilization']$ بازیابی می‌شود) به عنصر متناظر با هسته $core\_idx$ در لیست $core\_utils\ (list)$ اضافه می‌گردد. پس از اتمام حلقه، $core\_utils$ شامل بهره‌وری نهایی هر هسته خواهد بود.

۳.۳.۲ محاسبه هایپرپریود ($Hyperperiod$)

هایپرپریود، کوچکترین مضرب مشترک $(Least\ Common\ Multiple\ -\ LCM)$ دوره‌های تناوب تمام وظایف است و یک بازه زمانی مهم در تحلیل سیستم‌های بی‌درنگ متناوب محسوب می‌شود.

```
    hyperperiod = reduce(lambda a, b: a * b // gcd(a, b),
                         [t['period'] for t in tasks], 1)
```

توضیحات:

- $[t['period']\ for\ t\ in\ tasks]$: ابتدا یک لیست $(list)$ از تمام دوره‌های تناوب $(period)$ وظایف موجود در $tasks$ ایجاد می‌شود.
- $lambda a, b : a * b // gcd(a, b)$: این یک تابع لامبدا $(lambda)$ است که کوچکترین مضرب مشترک دو عدد a و b را با استفاده از فرمول $LCM(a,b) = (|a \times b|)/GCD(a,b)$ محاسبه می‌کند. تابع $gcd(a,b)$ بزرگترین مقسوم‌علیه مشترک a و b را برمی‌گرداند (فرض بر این است که از کتابخانه $math$ ایمپورت شده است). عملگر // تقسیم صحیح را انجام می‌دهد.
- $reduce(..., ۱)$: تابع $reduce$ (از کتابخانه $functools$) تابع لامبدای تعریف‌شده را به صورت تجمعی بر روی عناصر لیست $(list)$ دوره‌های تناوب اعمال می‌کند تا در نهایت یک مقدار واحد (هایپرپریود کل وظایف) به‌دست آید. مقدار ۱ به عنوان مقدار اولیه برای این عملیات تجمعی استفاده می‌شود.
- $hyperperiod$: متغیر $hyperperiod$ مقدار محاسبه‌شده هایپرپریود را ذخیره می‌کند.

۴.۳.۲ محاسبه زمان اتمام کل وظایف ($Makespan$)

در این پروژه، $Makespan$ به صورت ساده‌شده‌ای به عنوان حاصلضرب هایپرپریود در بیشترین میزان بهره‌وری مشاهده‌شده در میان هسته‌ها تعریف شده است.

```
makespan = hyperperiod * max(core_utils)
```

توضیحات:

- $max(core\_utils)$ $(list)$: بیشترین مقدار بهره‌وری از میان تمام هسته‌ها (که در لیست $core\_utils$ $(list)$ ذخیره شده‌اند) پیدا می‌شود.
- $makespan = hyperperiod * \max(core\_utils)$: مقدار $Makespan$ از حاصلضرب هایپرپریود در این بیشترین بهره‌وری هسته محاسبه می‌شود. این تعریف می‌تواند نشان‌دهنده طول زمانی باشد که شلوغ‌ترین هسته برای تکمیل کار خود در یک هایپرپریود نیاز دارد، یا اگر بهره‌وری بیش از ۱ باشد، نشان‌دهنده میزان بار اضافی است.

۵.۳.۲ محاسبه کیفیت خدمات ($Quality\ of\ Service\ -\ QoS$)

کیفیت خدمات در دو سطح وظیفه و سیستم محاسبه می‌شود.

```
task_qos = [100 if tasks[i]['utilization'] <= 1 else 0
            for i in range(len(tasks))]
system_qos = 100 if all(u <= 1 for u in core_utils) else 0
```

توضیحات:

- $task\_qos = [... \ for\ i\ in\ range(len(tasks))]$ $(list)$: کیفیت خدمات برای هر وظیفه به صورت جداگانه محاسبه و در یک لیست به نام $task\_qos$ ذخیره می‌شود.
  - $if\ tasks[i]['utilization'] <= ۱$: برای هر وظیفه i، بررسی می‌شود که آیا بهره‌وری ذاتی آن ($tasks[i]['utilization']$) کمتر یا مساوی ۱ است یا خیر. یک وظیفه با بهره‌وری بیشتر از ۱ ذاتاً قابل زمان‌بندی نیست.
  - در صورت برقرار بودن شرط، کیفیت خدمات آن وظیفه ۱۰۰ و در غیر این صورت ۰ در نظر گرفته می‌شود. (توجه: با توجه به نحوه تولید وظایف، این شرط معمولاً برای تمام وظایف برقرار است).
- $system\_qos = ۱۰۰ if all(u <= ۱\ for\ u\ in\ core\_utils)\ else\ ۰$: کیفیت خدمات کل سیستم محاسبه می‌شود.
  - $all(u <= ۱\ for\ u\ in\ core\_utils)$: این عبارت بررسی می‌کند که آیا بهره‌وری (u) *تمام* هسته‌ها (مقادیر موجود در $core\_utils$) کمتر یا مساوی ۱ است یا خیر.
  - اگر تمام هسته‌ها دارای بهره‌وری مجاز باشند (یعنی هیچ هسته‌ای دچار سربار نشده باشد)، کیفیت خدمات سیستم ۱۰۰ (نشان‌دهنده زمان‌بندی‌پذیری) و در غیر این صورت ۰ خواهد بود.

۶.۳.۲ مقادیر بازگشتی تابع

تابع در نهایت یک دیکشنری $(dictionary)$ شامل تمامی متریک‌های محاسبه‌شده را برمی‌گرداند.

```
return {
    'core_utils': core_utils,
    'makespan': makespan,
    'task_qos': task_qos,
    'system_qos': system_qos,
    'hyperperiod': hyperperiod
}
```

توضیحات:

- دیکشنری $(dictionary)$ بازگشتی شامل کلیدهای زیر است:
  - $'core\_utils'$: لیست $(list)$ بهره‌وری هر هسته.
  - $'makespan'$: مقدار محاسبه‌شده برای $Makespan$.
  - $'task\_qos'$: لیست $(list)$ کیفیت خدمات برای هر وظیفه.
  - $'system\_qos'$: کیفیت خدمات کل سیستم.
  - $'hyperperiod'$: مقدار هایپرپریود محاسبه‌شده.

۴.۲ تابع اجرای شبیه‌سازی ($run\_simulation$)

این تابع وظیفه اجرای کامل فرآیند شبیه‌سازی را برای مجموعه‌ای از پیکربندی‌های از پیش تعریف‌شده بر عهده دارد. برای هر پیکربندی، وظایف تولید شده، توسط الگوریتم ژنتیک زمان‌بندی می‌شوند و سپس متریک‌های عملکردی محاسبه و ذخیره می‌گردند.

۱.۴.۲ تعریف تابع، پیکربندی‌ها و مقداردهی اولیه نتایج

```
def run_simulation():
    configurations = [
        (8, 0.25), (8, 0.5), (8, 0.75), (8, 1.0),
        (16, 0.25), (16, 0.5), (16, 0.75), (16, 1.0),
        (32, 0.25), (32, 0.5), (32, 0.75), (32, 1.0)
    ]

    results = {}
```

توضیحات این بخش:

- $def run\_simulation()$: تابع $run\_simulation$ بدون هیچ آرگومان ورودی تعریف می‌شود.
- $configurations = [...]$: یک لیست ($list$) با نام $configurations$ تعریف می‌شود که شامل تمامی سناریوهای مورد بررسی در شبیه‌سازی است.
  - هر عنصر از این لیست ($list$) یک تاپل ($tuple$) دوتایی به فرم $(cores, util\_per\_core)$ است.
  - $cores$: نشان‌دهنده تعداد هسته‌های پردازنده در آن سناریو است (مقادیر ۸، ۱۶ یا ۳۲).
  - $util\_per\_core$: نشان‌دهنده میزان بهره‌وری هدف برای هر هسته در آن سناریو است (مقادیر ۰/۲۵، ۰/۵، ۰/۷۵ یا ۱/۰).
- $results = \{\}$: یک دیکشنری ($dictionary$) خالی با نام $results$ ایجاد می‌شود. این دیکشنری ($dictionary$) برای ذخیره‌سازی نتایج حاصل از شبیه‌سازی برای هر یک از پیکربندی‌ها استفاده خواهد شد. کلیدهای این دیکشنری ($dictionary$)، تاپل‌های ($tuple$) پیکربندی و مقادیر آن، دیکشنری‌هایی ($dictionary$) شامل جزئیات نتایج خواهند بود.

۲.۴.۲ حلقه پردازش پیکربندی‌ها، اجرای زمان‌بندی و محاسبه متریک‌ها

در این بخش، تابع بر روی هر یک از پیکربندی‌های تعریف‌شده پیمایش کرده، وظایف را تولید و زمان‌بندی می‌کند و سپس نتایج را ذخیره می‌نماید.

```
    for cores, util_per_core in configurations:
        total_util = cores * util_per_core
        tasks = generate_tasks(3 * cores, total_util)

        scheduler = GeneticScheduler(tasks, cores)
        assignment = scheduler.evolve()

        metrics = calculate_metrics(tasks, assignment, cores)
        results[(cores, util_per_core)] = {
            'tasks': tasks,
            'assignment': assignment,
            'metrics': metrics
        }
```

توضیحات گام به گام درون حلقه:

- $for cores, util\_per\_core \in configurations$: یک حلقه for برای پیمایش بر روی هر تاپل ($tuple$) $(cores, util\_per\_core)$ موجود در لیست ($list$) $configurations$ اجرا می‌شود.
- $total\_util = cores * util\_per\_core$: مجموع بهره‌وری کل ($total\ utilization$) برای سیستم در پیکربندی فعلی محاسبه می‌شود. این مقدار از حاصلضرب تعداد هسته‌ها ($cores$) در میزان بهره‌وری هدف برای هر هسته ($util\_per\_core$) به دست می‌آید.
- $tasks = generate\_tasks(۳ * cores, total\_util)$: تابع $generate\_tasks$ (که پیشتر توضیح داده شد) برای تولید مجموعه‌ای از وظایف فراخوانی می‌شود.
  - تعداد وظایف ورودی به این تابع برابر با $۳ * cores$ است، یعنی به ازای هر هسته، تقریباً ۳ وظیفه تولید می‌شود.
  - $total\_util$ نیز به عنوان مجموع بهره‌وری کل وظایف به تابع ارسال می‌گردد.
  - لیست ($list$) وظایف تولید شده در متغیر $tasks$ ذخیره می‌شود.

- $scheduler = GeneticScheduler(tasks, cores)$: یک نمونه $(instance)$ از کلاس $GeneticScheduler$ (که پیشتر توضیح داده شد) با نام $scheduler$ ایجاد می‌شود. وظایف تولیدشده $(tasks)$ و تعداد هسته‌های فعلی $(cores)$ به عنوان آرگومان به سازنده کلاس ارسال می‌شوند. سایر پارامترهای الگوریتم ژنتیک (مانند اندازه جمعیت، نرخ جهش و ...) از مقادیر پیش‌فرض تعریف‌شده در کلاس استفاده خواهند کرد.
- $assignment = scheduler.evolve()$: متد $evolve()$ از نمونه $scheduler$ فراخوانی می‌شود. این متد الگوریتم ژنتیک را اجرا کرده و بهترین تخصیص وظایف به هسته‌ها (کروموزوم بهینه) را برمی‌گرداند. نتیجه در متغیر $assignment$ ذخیره می‌شود.
- $metrics = calculate\_metrics(tasks, assignment, cores)$: تابع $calculate\_metrics$ (که پیشتر توضیح داده شد) برای محاسبه متریک‌های عملکردی بر اساس وظایف $(tasks)$، تخصیص به‌دست‌آمده $(assignment)$ و تعداد هسته‌ها $(cores)$ فراخوانی می‌شود. دیکشنری $(dictionary)$ حاوی متریک‌ها در متغیر $metrics$ ذخیره می‌شود.
- $results[(cores, util\_per\_core)] = \{...\}$: نتایج مربوط به پیکربندی فعلی در دیکشنری $(dictionary)$ $results$ ذخیره می‌شود.
  - کلید این ورودی، خود تاپل $(tuple)$ پیکرب renversement فعلی یعنی $(cores, util\_per\_core)$ است.
  - مقدار متناظر با این کلید، یک دیکشنری $(dictionary)$ دیگر است که شامل سه کلید ـ مقدار زیر می‌باشد:
    - $'tasks'$: لیست $(list)$ وظایف تولیدشده برای این پیکربندی.
    - $'assignment'$: تخصیص نهایی وظایف به هسته‌ها که توسط الگوریتم ژنتیک پیدا شده است.
    - $'metrics'$: دیکشنری $(dictionary)$ متریک‌های محاسبه‌شده برای این تخصیص.

۳.۴.۲ مقدار بازگشتی تابع

```
return results
```

توضیحات:

- $returnresults$: پس از اتمام حلقه و پردازش تمامی پیکربندی‌ها، تابع $run\_simulation$ دیکشنری $(dictionary)$ جامع $results$ را که حاوی تمامی داده‌ها و نتایج شبیه‌سازی برای همه سناریوها است، به عنوان خروجی برمی‌گرداند.

۳ تحلیل نمودارهای خروجی فاز اول

خروجی‌های بصری این فاز از پروژه به منظور ارزیابی عملکرد الگوریتم ژنتیک در زمان‌بندی وظایف بی‌درنگ بر روی پردازنده‌های چندهسته‌ای طراحی شده‌اند. این نمودارها و جداول، جنبه‌های مختلفی از کیفیت زمان‌بندی، بهره‌وری منابع و زمان‌بندی‌پذیری سیستم را تحت پیکربندی‌های متفاوت (تعداد هسته‌ها و بار کاری مختلف) به تصویر می‌کشند. در ادامه، هر یک از شش مولفه بصری ارائه شده در تصویر نتایج، تشریح می‌گردد.

اولین نمودار (واقع در بالا–چپ، بدون عنوان مشخص در تصویر نمونه، اما مربوط به کیفیت خدمات وظایف) به نمایش کیفیت خدمات ($Quality\ of\ Service\ -\ QoS$) برای تک‌تک وظایف می‌پردازد. در این نمودار، محور افقی می‌تواند نمایانگر شناسه وظایف و محور عمودی، میزان QoS تخصیص‌یافته به هر وظیفه باشد (معمولاً ٪۱۰۰ در صورتی که بهره‌وری ذاتی وظیفه کمتر یا مساوی ۱ باشد و ٪۰ در غیر این صورت). از آنجایی که در کد، این نمودار برای هر پیکربندی و با استفاده از نشانگرهای دایره‌ای رسم می‌شود، ممکن است هم‌پوشانی زیادی از نقاط مربوط به سناریوهای مختلف وجود داشته باشد. هدف اصلی این نمودار، بررسی این موضوع است که آیا وظایف تولید شده از ابتدا دارای مشخصات معتبری برای زمان‌بندی هستند یا خیر. با توجه به معیار ساده تعریف شده برای QoS وظیفه (صرفاً بر اساس بهره‌وری خود وظیفه)، انتظار می‌رود اکثر وظایف معتبر، QoS برابر با ٪۱۰۰ داشته باشند.

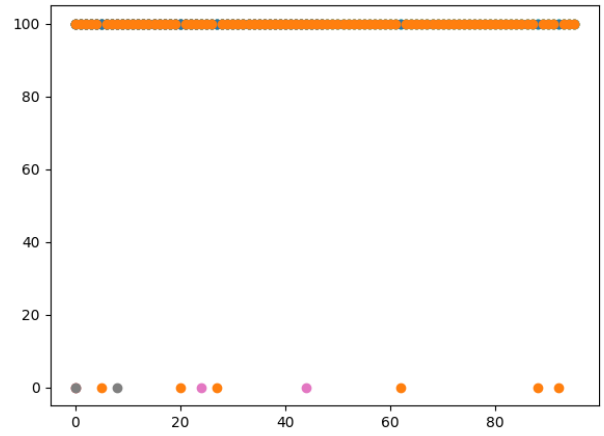
نمودار دوم که با عنوان ($System\ QoS$) مشخص شده است (واقع در بالا–راست)، کیفیت خدمات کلی سامانه را در برابر «میزان بهره‌وری به ازای هر هسته» ($Utilization\ per\ Core$) نشان می‌دهد. این نمودار خطی برای تعداد هسته‌های مختلف (۸، ۱۶ و ۳۲) به صورت مجزا و با رنگ‌های متفاوت رسم شده است. محور افقی، میزان بهره‌وری هدف برای هر هسته را از ۰/۲۵ تا ۱/۰ نمایش می‌دهد و محور عمودی، درصد کیفیت خدمات سامانه است. QoS سامانه ٪۱۰۰ است اگر بهره‌وری تمامی هسته‌ها پس از تخصیص وظایف، کمتر یا مساوی ۱ باقی بماند و در غیر این صورت ٪۰ خواهد بود. این نمودار اهمیت زیادی در درک آستانه زمان‌بندی‌پذیری سامانه با استفاده از الگوریتم ژنتیک دارد و نشان می‌دهد که الگوریتم تا چه میزان بارگذاری و با چه تعداد هسته‌ای قادر به یافتن یک تخصیص معتبر (بدون سربار بر روی هیچ هسته‌ای) بوده است.

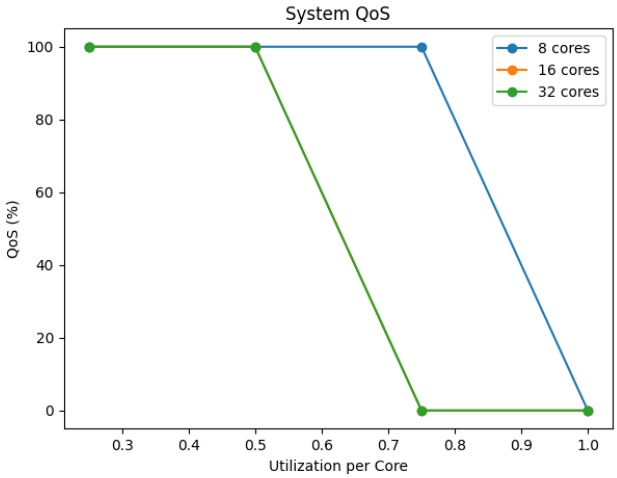
نمودار سوم، تحت عنوان ($Makespan\ Comparison$) (واقع در وسط–چپ)، به مقایسه مقدار $Makespan$ محاسبه‌شده برای حالات مختلف می‌پردازد. مشابه نمودار $System\ QoS$، این نمودار نیز به صورت خطی و برای تعداد هسته‌های مختلف (۸، ۱۶ و ۳۲) رسم شده و محور افقی آن «میزان بهره‌وری به ازای هر هسته» است. محور عمودی، مقدار $Makespan$ را نشان می‌دهد که در این پروژه به صورت ساده‌شده $hyperperiod \times \max(core\_utils)$ محاسبه شده است. این نمودار کمک می‌کند تا تأثیر افزایش بار سیستم و تعداد هسته‌ها بر روی این معیار از کارایی زمان‌بندی (که هدف، کمینه کردن آن است) مشاهده شود.

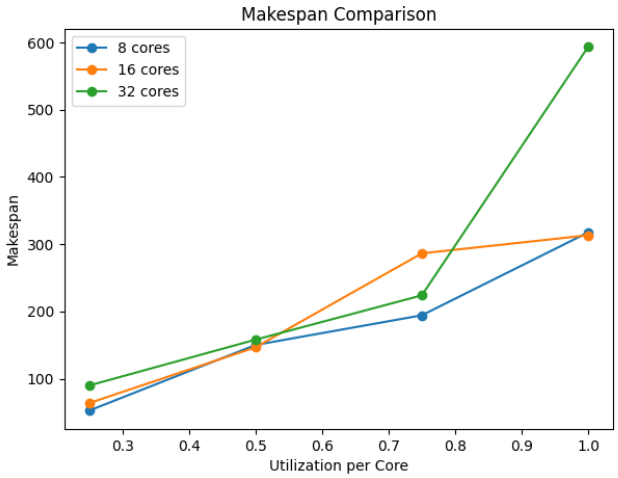
چهارمین نمودار، یک هیستوگرام با عنوان ($Core\ Utilization\ Distribution$) است (واقع در وسط–راست) که توزیع مقادیر بهره‌وری هسته‌ها را نمایش می‌دهد. این مقادیر از تمامی هسته‌ها و در تمامی پیکربندی‌های شبیه‌سازی‌شده جمع‌آوری شده‌اند. محور افقی، بازه‌های مختلف بهره‌وری و محور عمودی، فراوانی هسته‌هایی که بهره‌وری آن‌ها در آن بازه قرار گرفته را نشان می‌دهد. این نمودار برای ارزیابی کیفیت متعادل‌سازی بار ($load\ balancing$) توسط الگوریتم ژنتیک بسیار مفید است. یک توزیع باریک و متمرکز حول میزان بهره‌وری هدف برای هر هسته (و البته زیر مقدار ۱/۰) نشان‌دهنده عملکرد خوب الگوریتم در توزیع یکنواخت بار است.

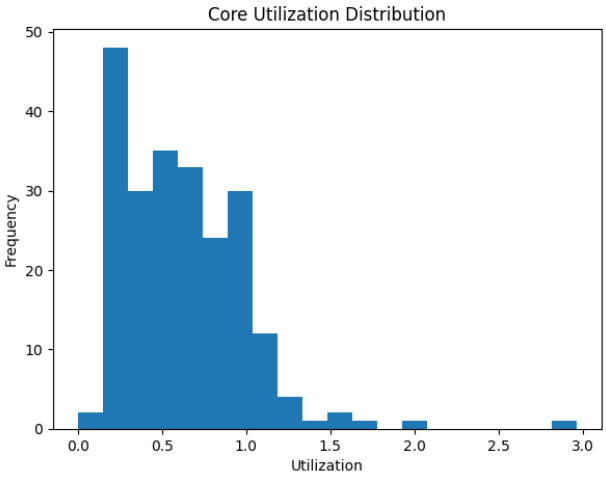
نمودار پنجم، یک نمودار میله‌ای با عنوان ($System\ Schedulability$) است (واقع در پایین–چپ) که قابلیت زمان‌بندی سیستم را برای هر یک از پیکربندی‌های خاص آزمایش‌شده نمایش می‌دهد. هر میله معرف یک پیکربندی مشخص (مثلاً $8c/0.25u$ به معنای ۸ هسته با بهره‌وری هدف ۰/۲۵ برای هر هسته) است و ارتفاع میله، QoS سیستم (٪۱۰۰ یا ٪۰) را برای آن پیکربندی نشان می‌دهد. این نمودار یک دید کلی و سریع از این که کدام سناریوها توسط الگوریتم ژنتیک با موفقیت زمان‌بندی شده‌اند، ارائه می‌دهد.

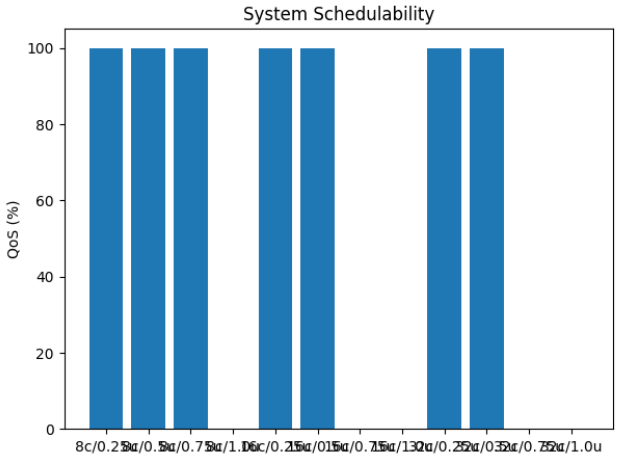
در نهایت، ششمین مولفه (واقع در پایین–راست)، جدولی با عنوان ($Sample\ Task\ Parameters$) است. این جدول، مشخصات کلیدی (شامل شناسه یا (ID)، زمان اجرا یا ($Exec$)، دوره تناوب یا ($Period$)، موعد زمانی یا ($Deadline$) و میزان بهره‌وری یا ($Util$)) را برای نمونه کوچکی از وظایف (در اینجا، ۵ وظیفه اول از اولین پیکربندی شبیه‌سازی‌شده) نمایش می‌دهد. هدف از این جدول، ارائه یک دید ملموس از نوع وظایفی است که در شبیه‌سازی‌ها مورد استفاده قرار گرفته‌اند.











Sample Task Parameters

ID	Exec	Period	Deadline	Util
0	1.6497934007887793	50	50	0.032995868015775 59
1	8.42312835505986	50	50	0.16846256710119722
2	2.3638896776196128	50	50	0.047277793552392255
3	11.073316762267593	100	100	0.11073316762267593
4	0.3937006155178424	10	10	0.03937006155178424