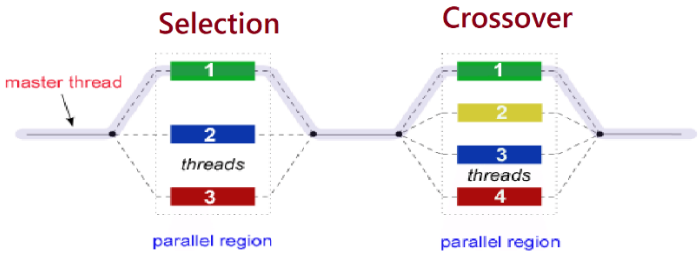
پیاده‌سازی موازی الگوریتم ژنتیک

استاد درس: دکتر عبدالرضا سوادی

اعضای گروه: محمدحسین حسینی، وحید رمضانی دشت‌بیاض، سروش فعال

الگوریتم ژنتیک

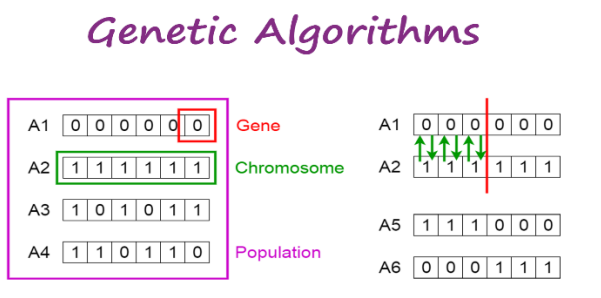
الگوریتم ژنتیک یک الگوریتم جستجو مبتنی بر هیوریستیک است که از نظریه‌ی تکامل داروین الگوبرداری شده است. مراحل اصلی الگوریتم شامل مرحله‌ی انتخاب(Selection)، تولید مثل(Crossover) و جهش(Mutation) است. مراحل ذکر شده کاملا ترتیبی هستند. به این معنی که شرط لازم برای انجام یک مرحله از الگوریتم، پایان یافتن مرحله‌ی قبلی است.



به طور کلی بیشترین حجم پردازش در دو مرحله‌ی انتخاب و تولید مثل انجام می‌گیرد که نقطه‌ی عطف تاثیر چشم‌گیر موازی‌سازی، در بهبود زمان اجرای الگوریتم، نیز در همین قسمت‌هاست.

برنامه شامل چند قسمت فرعی از جمله آماده‌کردن جمعیت اولیه(init_pop)، محاسبه‌ی برازندگی کروموزوم‌ها(fitness)، مرتب‌سازی جمعیت(sort) و ... است که در این قسمت‌ها نیز تا حدودی نخ‌ها مورد استفاده قرارگرفته و باعث بهبود زمان اجرا شده‌اند.

الگوریتم برای اجرا نیاز به تعیین و تنظیم چندین فاکتور خاص دارد. این فاکتورها شامل اندازه‌ی جمعیت(Population)، حداکثر دورها(Max Iteration) و احتمال جهش(Mutation Probability) هستند که به صورت Hard-code در برنامه تنظیم شده‌اند. اجرای برنامه با مجموعه فاکتورهای مختلف نتایج مختلفی را از نظر دقت عملکرد الگوریتم و همچنین زمان اجرای الگوریتم به همراه خواهد داشت. در انتها نتایج اجرای برنامه با تعدادی مجموعه مختلف فاکتورها گزارش و تحلیل شده‌است.



فاکتورهای موثر در الگوریتم

فاکتور اندازه‌ی جمعیت(Population): این شاخص نشان‌دهنده‌ی اندازه‌ی جمعیتی است که الگوریتم بر رویشان اجرا می‌شود. مشخص است که با افزایش مقدار این شاخص زمان اجرای الگوریتم نیز افزایش خواهد داشت.

فاکتور حداکثر دورها(Max Iteration): این شاخص حداکثر تعداد دورهایی که الگوریتم طی می‌کند تا متوقف شود را مشخص می‌کند. واضح است که با افزایش مقدار این شاخص زمان اجرای الگوریتم نیز افزایش خواهد داشت.

فاکتور احتمال جهش(Mutation Probability): این شاخص احتمال رخداد جهش را به درصد بیان می‌کند. این شاخص در بهبود دقت و عملکرد الگوریتم تاثیر مثبت دارد اما در زمان اجرای الگوریتم تغییری ایجاد نمی‌کند.

مراحل اصلی الگوریتم

انتخاب(Selection): در این مرحله بر اساس استراتژی چرخ شانس به تعداد جمعیت، از نسل فعلی، با جایگذاری، کروموزوم‌ها را انتخاب می‌کنیم. کروموزوم با برازندگی بیشتر شانس انتخاب بالاتری دارد. امکان انتخاب تکراری یک کروموزوم نیست وجود دارد. این مرحله به صورت زیر پیاده‌سازی و موازی‌سازی شده‌است.

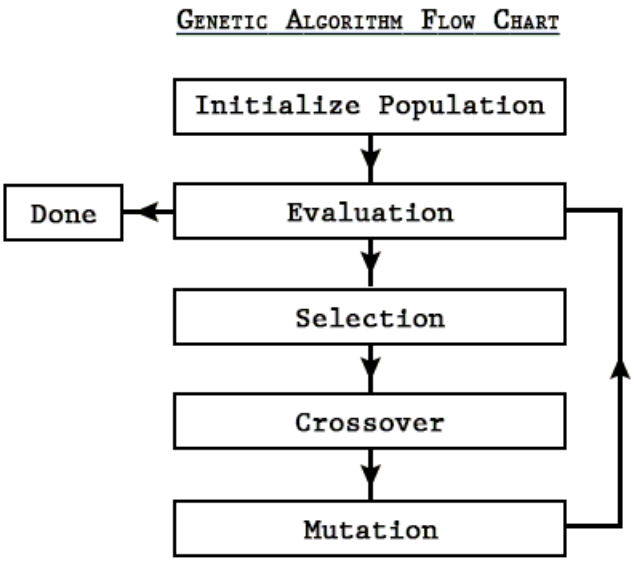
```
#pragma omp parallel for
    for (int j = 0; j < POPULATION; j++)
    {
        double random = (double)rand() / RAND_MAX;
        double cumulative_prob = 0;
        for (int i = 0; i < POPULATION; i++)
        {
            cumulative_prob += popcurrent[i].prob;
            if (random < cumulative_prob)
            {
                selected[j] = popcurrent[i];
                break;
            }
        }
    }
```

تولیدمثل(Crossover): در این مرحله ابتدا جمعیت انتخاب شده از مرحله‌ی قبل دریافت می‌شود و سپس هر دو کروموزوم با انجام یک تولید مثل، دو کروموزوم جدید را تولید می‌کنند تا نسل بعدی تشکیل شود. انتظار می‌رود کروموزوم فرزند، چون حاصل تولید مثل والدین با برازندگی بالاست، دارای برازندگی بالا باشد، اما به دلیل وجود فاکتور شانس و احتمال در عمل تولید مثل، عکس این اتفاق نیز ممکن است.

```
void crossover(chrom popnext[POPULATION])
{ // crossover function takes a pointer to array of chromes
    int random;
    random = rand();
    chrom temp_child;
    random = ((random % (GENE_COUNT - 1)) + 1); // random cross over for first child (child of first
#pragma omp parallel for
    for (int i = 0; i < random; i++)
    {
        temp_child.bit[i] = popnext[0].bit[i];
    }
#pragma omp parallel for
    for (int i = random; i < GENE_COUNT - 1; i++)
    {
        temp_child.bit[i] = popnext[POPULATION - 1].bit[i];
    }
#pragma omp parallel for
    for (int chrom_counter = 0; chrom_counter < POPULATION; chrom_counter++)
    {
        random = rand();
        random = ((random % (GENE_COUNT - 1)) + 1);
        for (int i = random; i < GENE_COUNT; i++)
        {
            popnext[chrom_counter].bit[i] = popnext[chrom_counter + 1].bit[i];
        }
    }
    popnext[POPULATION - 1] = temp_child; // the last popnext chrom now becomes the new child
```

جهش(Mutation): در این مرحله به صورت تصادفی و با احتمالی ناچیز یک تغیر در یکی از ژن‌های یکی از کروموزوم‌های موجود در جمعیت اتفاق می‌افتد. این مرحله شامل حلقه نیست و مرتبه زمانی $O(1)$ دارد و در زمان اجرای الگوریتم تاثیری ندارد.

```
void mutation(chrom popnext[POPULATION])
{
    srand((unsigned)time(NULL));
    int prob = rand() % 101;
    if (prob <= MUTATION_FAC)
    {
        int chrom = rand() % (POPULATION);
        int gene = rand() % (GENE_COUNT);
        popnext[chrom].bit[gene] = (popnext[chrom].bit[gene] == 1) ? 0 : 1;
        popnext[chrom].fit = fitness(popnext[chrom]);
    }
}
```



سایر قسمت‌های برنامه

تشکیل جمعیت اولیه(init_pop): در این قسمت از برنامه، جمعیتی از کروموزوم‌ها به اندازه‌ی فاکتور Population از داخل یک فایل استاتیک و از قبل تهیه شده خوانده‌می‌شوند. ایده‌ی خواندن جمعیت از داخل فایل استاتیک، برخلاف اصل الگوریتم که در هر بار اجرا داده‌های اولیه به صورت تصادفی تولید می‌شوند، به هدف کاهش اثر متغیرهای تصادفی در عملکرد الگوریتم برای سهولت امکان مقایسه‌ی نتایج اجراهای مختلف الگوریتم بوده‌است.

مرتب‌سازی(Sort): در این قسمت نیز از یک الگوریتم ساده‌ی مرتب‌سازی با مرتبه زمانی $O(n^2)$ برای مرتب کردن جمعیت بر اساس برازندگی کروموزوم‌ها کمک گرفته‌شده‌است.

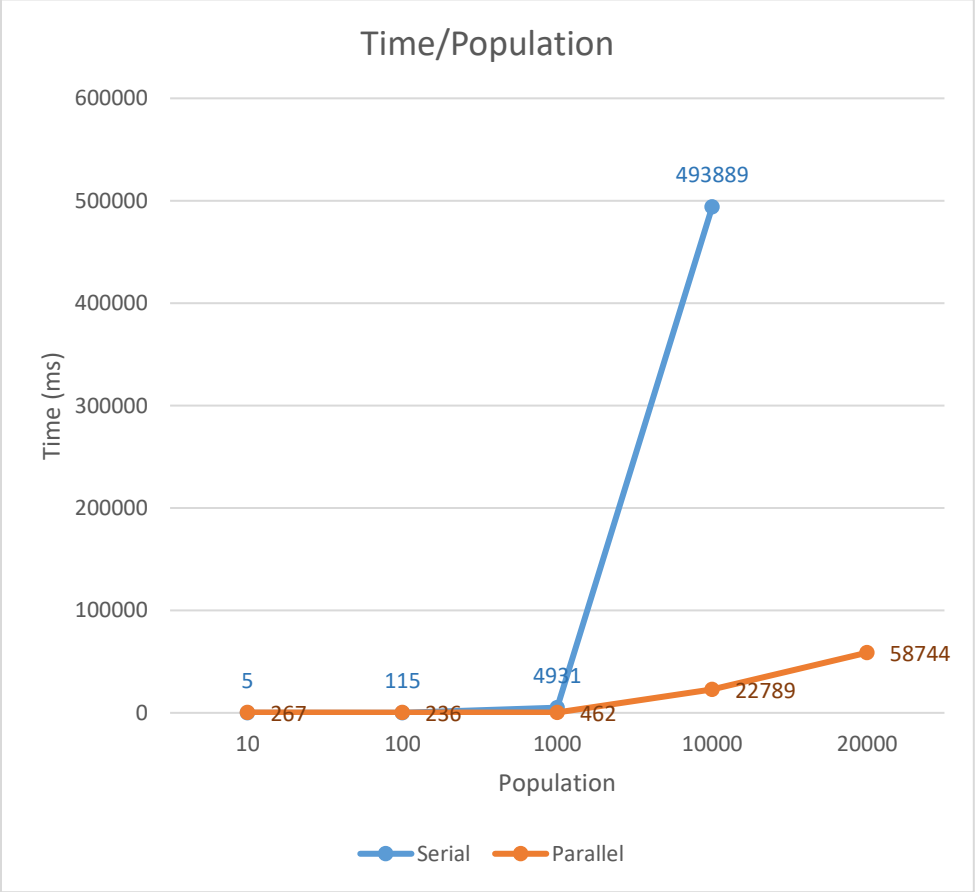
محاسبه برازندگی(Fitness): این بخش از برنامه نیز برازندگی را برای هر کروموزوم محاسبه می‌کند.

Operating System:	Windows 11 Home 64-bit (10.0, Build 22000)
Language:	English (Regional Setting: English)
System Manufacturer:	LENOVO
System Model:	82JW
BIOS:	HHCN27WW
Processor:	AMD Ryzen 7 5800H with Radeon Graphics (16 CPUs), ~3.2GHz
Memory:	16384MB RAM

تحلیل عملکرد الگوریتم در دو حالت سری و موازی – فاکتور جمعیت

در ادامه عملکرد الگوریتم در حالات سری و موازی با فاکتورهای مختلف بررسی می‌شود. در شکل 1 شاهد دو نمودار هستیم که نمودار آبی رنگ مربوط به حالت سری و نارنجی رنگ مربوط به حالت موازی می‌باشند. در این شکل فاکتور جمعیت اولیه(بدون تغییر دادن حداکثر دور[1]های الگوریتم) دستخوش تغییر شده و زمان اجرای الگوریتم برای هر جمعیت در هردو حالت سری و موازی برحسب میلی‌ثانیه(ms) توسط دو نمودار مختلف نمایش داده شده است.

همانطور که در شکل مشاهده می‌شود برای جمعیت‌هایی که از یک حد مشخص(10000) بیشتر هستند زمان اجرا در حالت سری فوق العاده بالا خواهد بود و عملا اجرای الگوریتم دیگر ممکن نخواهد بود که در حالت موازی این مشکل تا حد خوبی حل شده است و می‌توان برای جمعیت‌های اولیه بیشتر هم الگوریتم را اجرا نمود.



شکل 1

[1] Iteration

تحلیل عملکرد الگوریتم در دو حالت سری و موازی – فاکتور حداکثر دور

در شکل 2 نیز با ثابت نگه‌داشتن جمعیت اولیه، حداکثر دورهای الگوریتم تغییر کرده و زمان های اجرای متفاوت برحسب میلی‌ثانیه اندازه‌گیری و نمایش داده شده‌اند. نمودار آبی مربوط به حالت سری و نمودار نارنجی مربوط به حالت موازی می‌باشد.

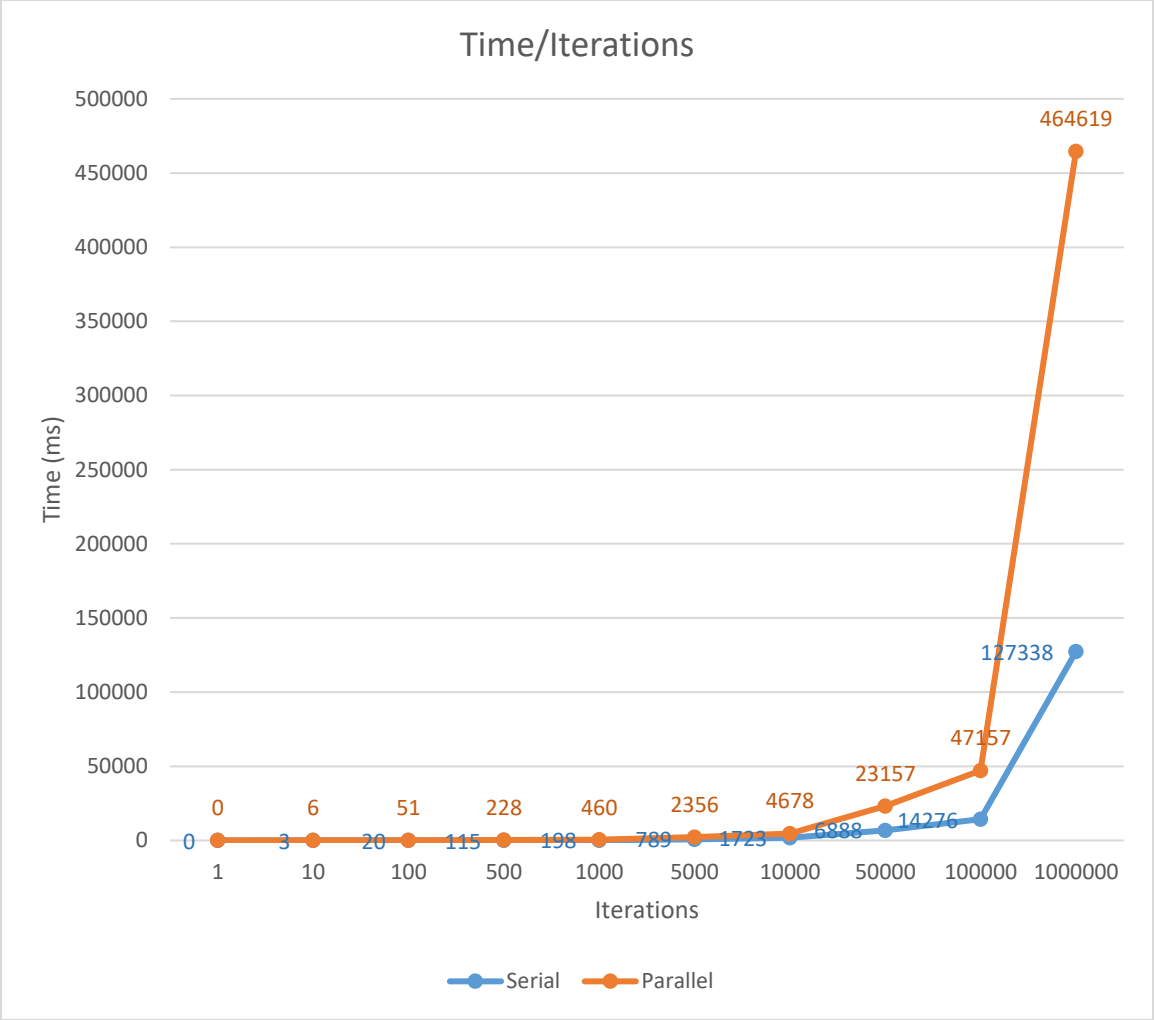
همانطور که در نمودار مربوط به حالت سری مشاهده می‌شود رابطه زمان اجرا با حداکثر دورها تقریبا خطی می‌باشد.

در مورد نمودار مربوط به حالت موازی اما اتفاقی که می‌افتد این است که شاهد افزایش زمان اجرا نسبت به حالت سری هستیم؛ دلیل این موضوع این است که سربارهای مربوط به ایجاد thread بیش از حد زیاد می‌شوند و موجب افزایش زمان نسبت به حالت سری می‌شود؛ لازم به ذکر است که حلقه اصلی الگوریتم که در داخل آن عملگرهای مختلف الگوریتم ژنتیک تا زمان رسیدن به حداکثر تعداد دورها روی جمعیت اعمال می‌شوند، قابل موازی‌سازی نمی‌باشد زیرا مراحل اجرای الگوریتم ژنتیک ترتیبی[۲] می‌باشد و در هر دور باید روی جمعیتی که در دور قبل ایجاد شده است، عملیات مجددا صورت گیرد و اگر دورهای حلقه موازی اجرا شوند آنگاه این ترتیب رعایت نخواهد شد؛ با این تفاسیر اگر هم‌چنان حلقه اصلی الگوریتم را موازی‌سازی کنیم، بهبود زمان خواهیم داشت، اما دقت الگوریتم کاهش خواهد یافت و در نهایت هم‌گرایی الگوریتم با مشکل روبرو خواهد شد.

با توجه به عدم امکان موازی‌سازی حلقه اصلی برنامه، در تمام مراحل داخلی الگوریتم(اعم از عملگرهای انتخاب، تولیدمثل و...) موازی‌سازی تا حد امکان صورت گرفته است.

---

[2] sequential


Time/Iterations
500000
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
Time (ms)
464619
127338
47157
23157
14276
6888
4678
2356
1723
789
460
228
198
115
51
20
6
3
0
0
1
10
100
500
1000
5000
10000
50000
100000
1000000
Iterations
Serial
Parallel

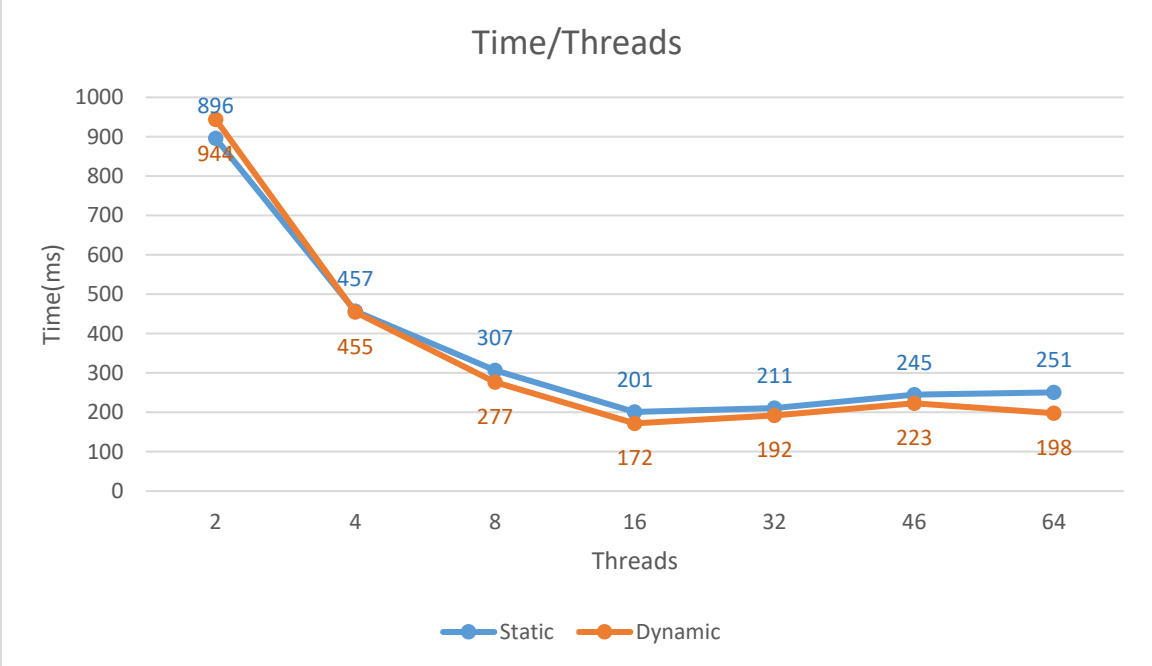
شکل 2

تحلیل عملکرد الگوریتم در دو حالت ایستا[3] و پویا[4] با نخ[5]های مختلف

در ادامه تحلیل دیگری بر روی الگوریتم صورت گرفته است و زمان اجرا با تعداد نخ‌های مختلف اندازه‌گیری شده‌است.(شکل 3)

در حالت ایستا منظور از تعداد نخ‌ها، تعداد نخ‌هایی است که برای اجرای برنامه بکار گرفته می‌شوند و در حالت پویا نیز با توجه به اینکه تعداد نخ‌ها به شکل خودکار تعیین می‌شوند، منظور از تعداد نخ‌ها حداکثر تعداد نخ‌هایی می‌باشد که می‌توانند برای اجرای برنامه بکار گرفته شوند که یعنی یک سقف برای نخ‌های قابل ایجاد ساخته می‌شود.

همان‌طور که در نمودار مربوط به شکل 3 دیده می‌شود زمان اجرای حالت ایستا برای تعداد نخ‌های کم(کمتر از 4) بهتر از حالت پویا بوده است و پس از اینکه زمان‌های اجرای دو حالت ایستا و پویا در هنگام وجود 4 نخ باهم تقریبا برابر می‌شود، حالت پویا برای تعداد نخ‌های بیشتر از 4 زمان اجرای کمتری نسبت به حالت ایستا دارد و هرچه تعداد نخ‌ها بیشتر می‌شود نیز هم‌چنان زمان اجرای حالت پویا از حالت ایستا کمتر است



شکل 3

[3] static
[4] dynamic
[5] thread

نکات و چالش‌های پیاده‌سازی

در پیاده‌سازی پروژه براساس چالش‌های مختلفی که در سر راه تیم ما قرار گرفت، تصمیماتی مبنی بر تغییر و نحوه‌ی پیاده‌سازی گرفته‌شد.

از جمله این چالش‌ها، که در بالاتر نیز اشاره‌ای به آن شد، **ذات ترتیبی** مراحل اصلی الگوریتم ژنتیک است. در اجرای الگوریتم ژنتیک باید ترتیب مراحل حفظ شود زیرا هر مرحله به خروجی مرحله‌ی قبل نیاز دارد. در این سناریو برای پیاده‌سازی موازی الگوریتم با مشکل مواجه هستیم و راه حل این مشکل موازی‌سازی هر مرحله به صورت جداگونه است، که این زوش در پیاده‌سازی ما پیش گرفته‌شده‌است.

یکی دیگر از نکات قابل توجه که در مسیر پروژه به چشم آمد، **تاثیر منفی زیاد شدن تعداد نخ‌ها** بیشت از حدی مشخص، بر زمان اجرای الگوریتم است. قابل مشاهده‌است که وقتی تعداد نخ‌ها از مقدار مشخصی بیشتر می‌شود، بازدهی اجرا کاهش می‌یابد.

از مقایسه‌ی نمودارهای به‌دست آمده برای اجرای الگوریتم با **نخ‌های ایستا(Static) و پویا(Dynamic)** نکات جالبی برداشت می‌شود که در قسمت مربوطه به آن پرداخته‌شده‌است.

روند نمایی نمودارهای ترسیم شده نیز از نکاتی است که باید به آن توجه داشت.

ضمیمه

تمامی کدهای پیاده‌سازی این پروژه در ریپازیتوری سایت گیت‌هایت(Github repository) در لینک زیر قابل مشاهده و دستیابی است.

https://github.com/smhhoseinee/genetic_algorithm_implementation_in_c

مراجع

- Artifitial Intelligence: A modern approach, Stuart Russell, Peter Norvig
- A “Hands-on” introduction to OpenMP,Tim Mattson, Bronis R. de Supinski
- https://www.codeproject.com/Tips/10417/Genetic-Algorithm-2
- https://www.geeksforgeeks.org/genetic-algorithms/
- A Summary of Research on Parallel Genetic Algorithms, Erick Cantú-Paz
- https://www.openmp.org/resources/