برنامه‌سازی موازی

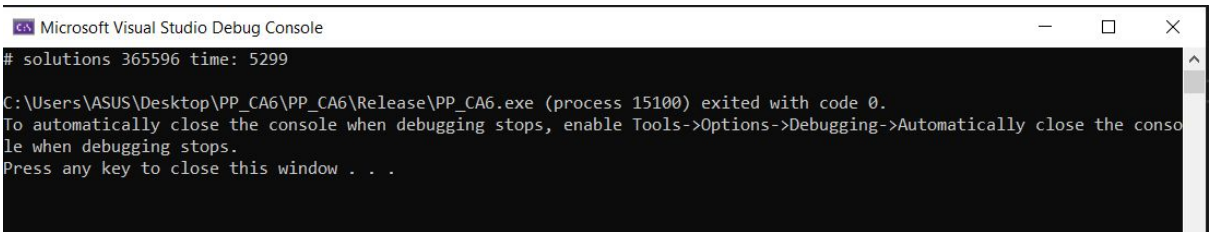
گزارش پروژه‌ی ششم

سید پارسا حسینی‌نژاد 810196604

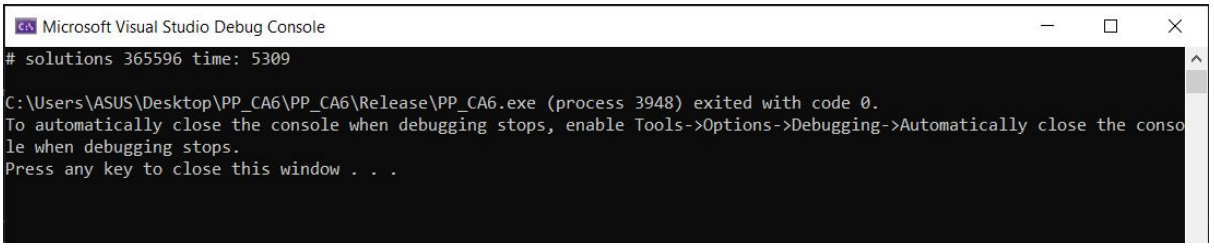
کیمیا خبیری 810196606

کد مسئله‌ی وزیری که به ما داده شده بود را با دو کامپایلر visual c++ و intel c++ اجرا کردیم که نتایج این اجرا در دو شکل زیر قابل مشاهده است.

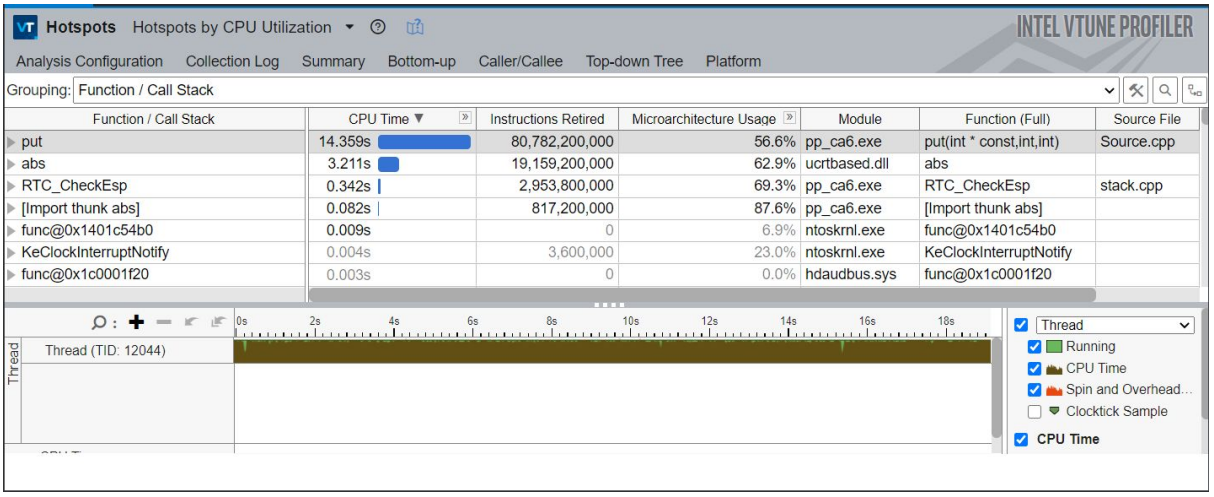
اجرای به وسیله ی visual c++:



اجرا به وسیله‌ی intel c++:



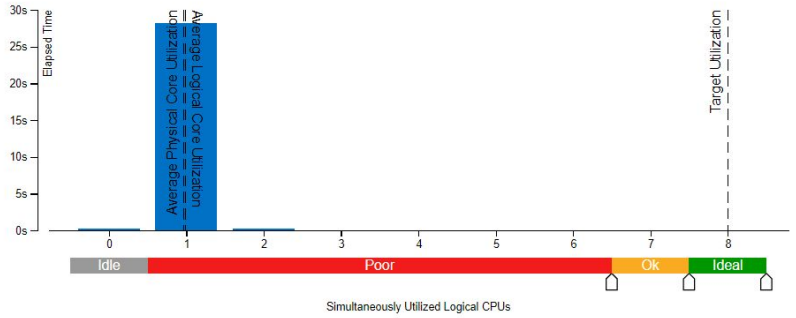
همچنین شکل زیر نشان‌دهنده‌ی cpu utilization برای اجرای سریال کد است. همانطور که میبینید به ما hotspot های کد نشان داده شده و گفته شده که در زمان اجرای کد هر اجرای هر تابع چقدر زمان برده است. تابع put بیشترین تکرار در اجرا و همچنین بیشترین زمان اجرا را دارد و بنابراین عملیات موازی‌سازی باید روی فراخوانی این تابع انجام پذیرد.



این تصویر نیز نشان‌دهنده‌ی نحوه‌ی تقسیم کار بین thread ها در برنامه‌ی ماست و ارزیابی میکند که تقسیم کار چقد خوب یا بد انجام گرفته است. به عنوان مثال زمانی که یک thread داریم از قابلیت ۸ هسته ای بودن پردازنده استفاده نشده و این یک تقسیم کار ضعیف است.

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



عملیات موازی سازی را روی تابع solve اعمال کردیم. این تابع شروع به جستجو برای راه حل های مختلف مسئله‌ی چند وزیر می کند و در هر مرحله این جستجو را از جابه‌جا کردن جایگاه وزیر در سطر اول شروع میکند. این وظیفه را میان thread ها پخش میکنیم به این شکل که هر thread وظیفه ی پیدا کردن راه حل با شروع از جایگاه های محدودتری از وزیر در سطر اول را داشته باشد. همچنین به هر thread متغیر solution و queen مجزا می دهیم که thread ها در انجام عملیاتشان از یکدیگر مستقل باشند. جواب نهایی حاصل جمع solutions های پیدا شده توسط thread های مجزاست.

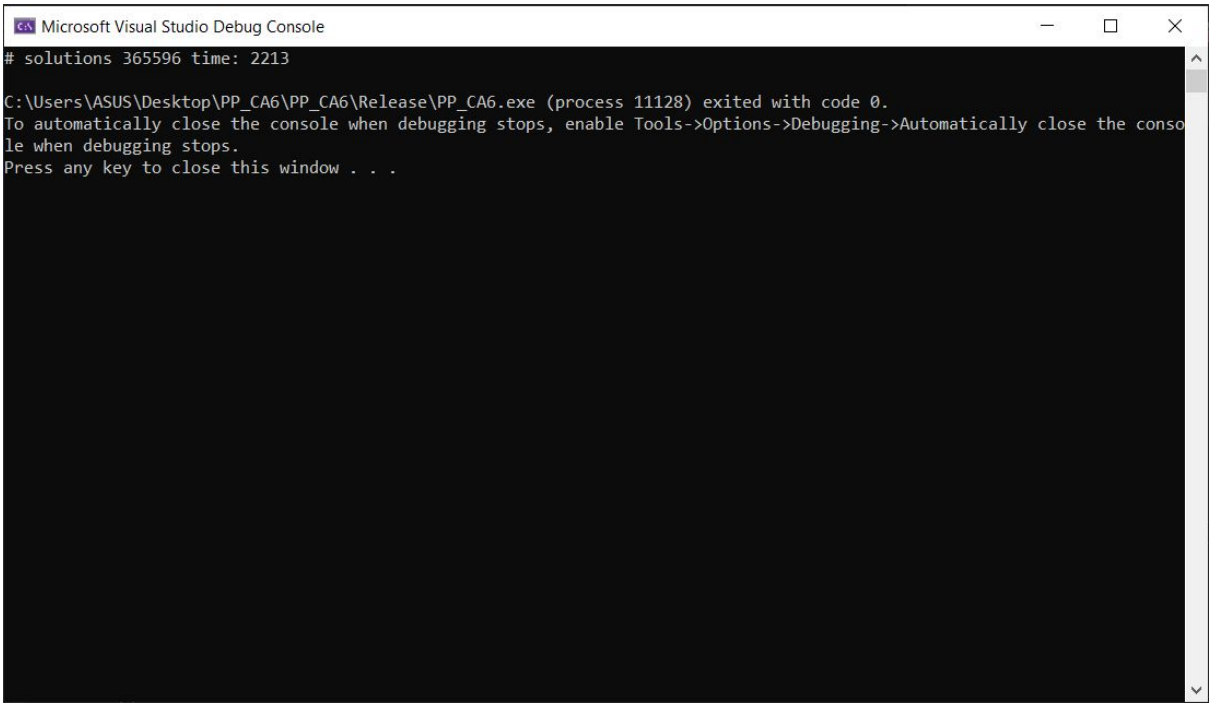
تابع put را نیز اندکی تغییر دادیم تا به جای شمارش روی متغیر گلوبال solution، تعداد را روی متغیر solution مخصوص به thread خودش جمع آوری کند.

```
#define N  14
#define NUM_OF_THREADS 7

int put(int Queens[], int row, int column, int* solutions)
{
    int i;
    for (i = 0; i < row; i++) {
        if (Queens[i] == column || abs(Queens[i] - column) == (row - i))
            return -1;
    }
    Queens[row] = column;

    if (row == N - 1) {
        (*solutions)++;
    }
    else {
        for (i = 0; i < N; i++) { // increment row
            put(Queens, row + 1, i, solutions);
        }
    }
    return 0;
}
```

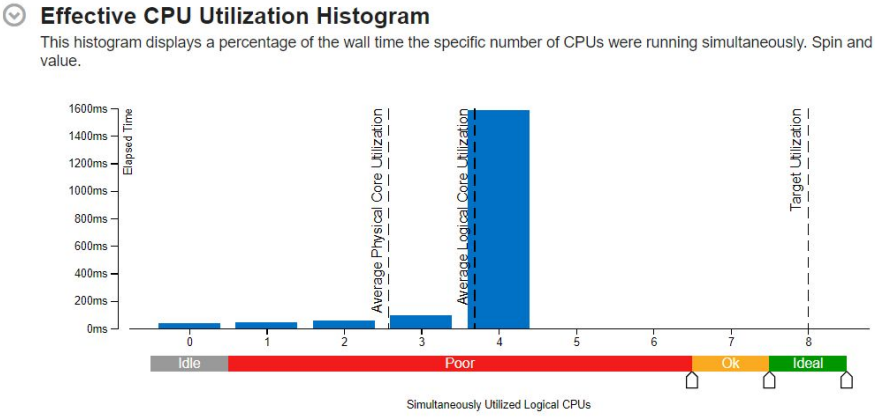
```
int solve(int Queens[]) {
    int i, localSolutions[NUM_OF_THREADS] = {0}, localQueens[NUM_OF_THREADS][N], solutions = 0;

    #pragma omp parallel num_threads(NUM_OF_THREADS) shared(localSolutions, localQueens) private(i)
        #pragma omp for schedule(dynamic)
            for (i = 0; i < N; i++) {
                put(localQueens[omp_get_thread_num()], 0, i, &localSolutions[omp_get_thread_num()]);
            }

    for (i = 0; i < NUM_OF_THREADS; i++)
        solutions += localSolutions[i];

    return solutions;
}
```

```
int main()
{
    int Queens[N], diff, solutions;
    struct timeb start, end;

    ftime(&start);
    solutions = solve(Queens);
    ftime(&end);

    diff = (int)(1000.0 * (end.time - start.time)
        + (end.millitm - start.millitm));

    printf("# solutions %d time: %u \n", solutions, diff);

    return 0;
}
```

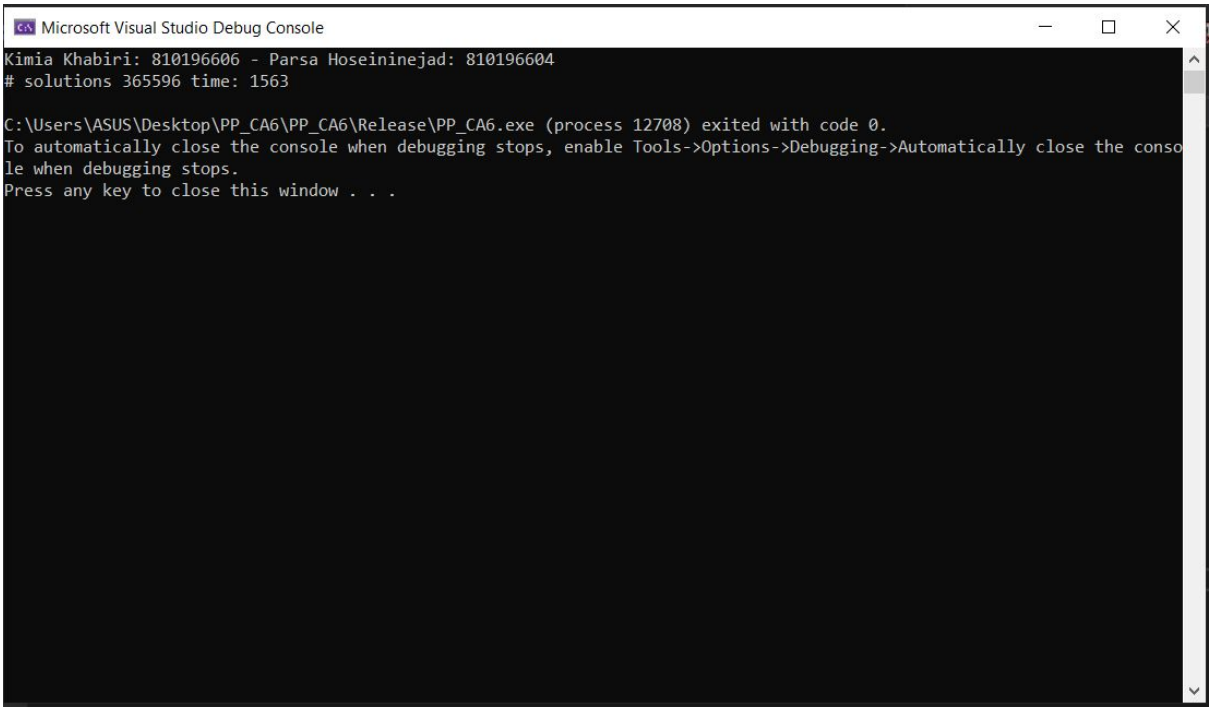
شکل زیر زمان اجرای کد با استفاده از 4 thread را نشان می دهد.



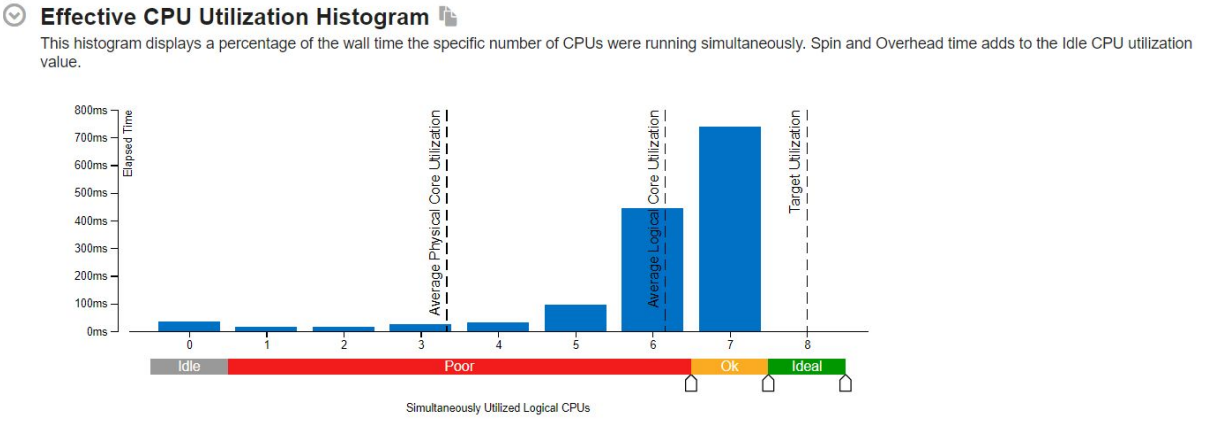
شکل زیر cpu utilization هنگامی که تعداد thread ها 4 باشد را نشان میدهد. همانطور که میبینید این تعداد همچنان ضعیف ارزیابی شده و باید با ایجاد تغییراتی در کد و یا تعداد thread ها tune شود.

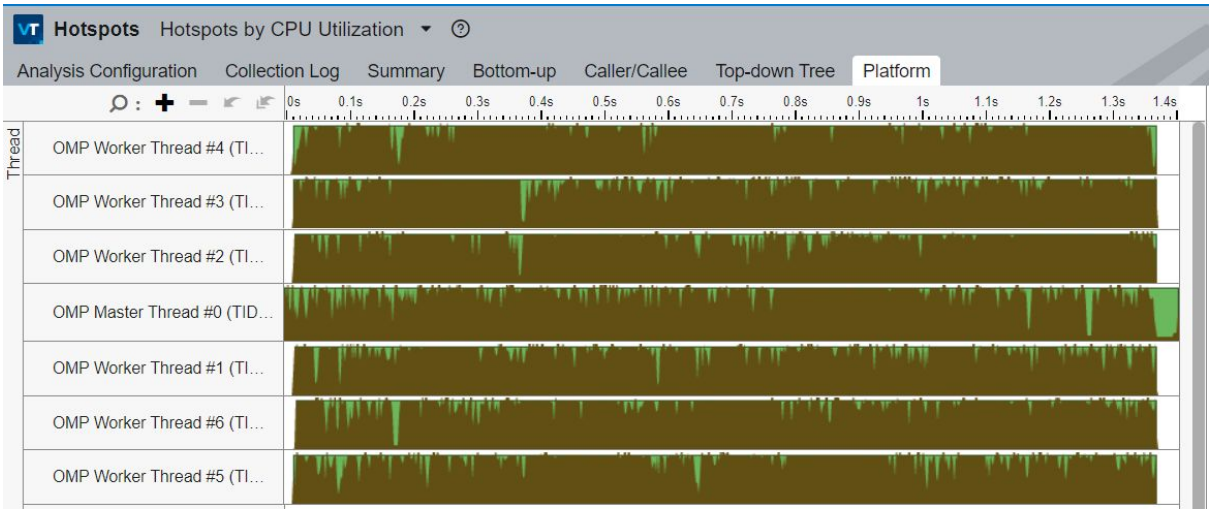


سپس تعداد thread ها را به ۷ thread افزایش دادیم و شکل زیر زمان اجرا پس از اعمال این تغییر است.



در این شکل تعداد thread ها را به ۷ تا افزایش دادیم. همانطور که میبینیم cpu utilization از وضعیت poor به وضعیت ok بهبود یافته است. عدد ۷ به نظر عدد مناسبی برای تعداد thread ها میرسد. چرا که ۱۴ (تعداد وزیرها) بر ۷ بخش پذیر است و بنابراین میتوان پیش‌بینی کرد که thread ها کار تقریبا هم اندازه ای انجام دهند.





عیب های برنامه نیز که با استفاده از inspector گزارش شده در شکل زیر آمده است. همانطور که میبینید با موازی سازی برنامه دو مشکل mismatch allocation/deallocation و memory leak اتفاق افتاده است.