

برنامه‌سازی موازی

گزارش پروژه‌ی ششم



سید پارسا حسینی‌نژاد 810196604

کیما خیبری 810196606

کد مسئله‌ی وزیر ی 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:

```

Microsoft Visual Studio Debug Console
# solutions 365596 time: 5299

C:\Users\ASUS\Desktop\PP_CA6\PP_CA6\Release\PP_CA6.exe (process 15100) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
  
```

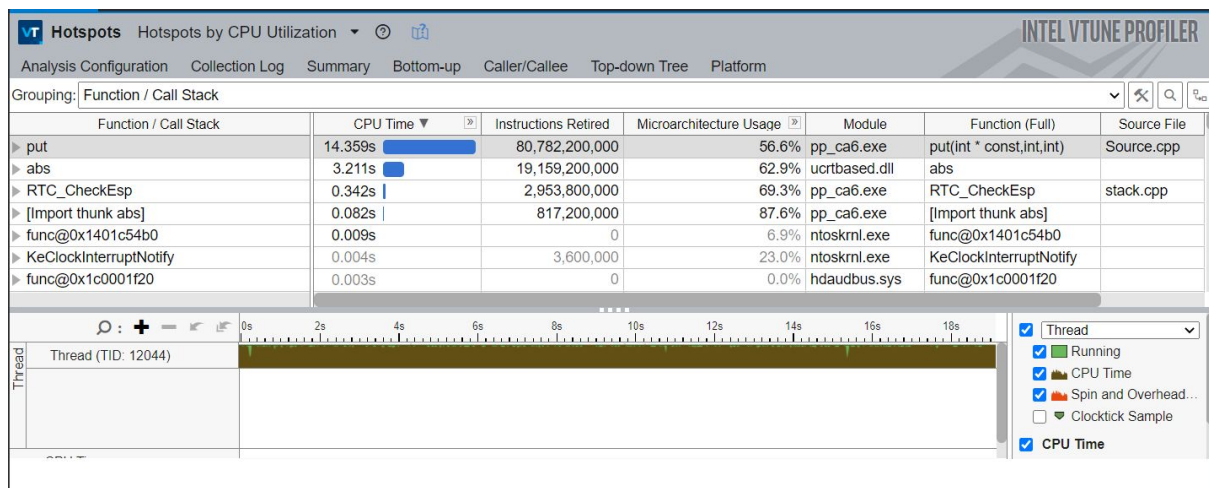
اجرا به وسیله ی ++intel c:

```

Microsoft Visual Studio Debug Console
# solutions 365596 time: 5309

C:\Users\ASUS\Desktop\PP_CA6\PP_CA6\Release\PP_CA6.exe (process 3948) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
  
```

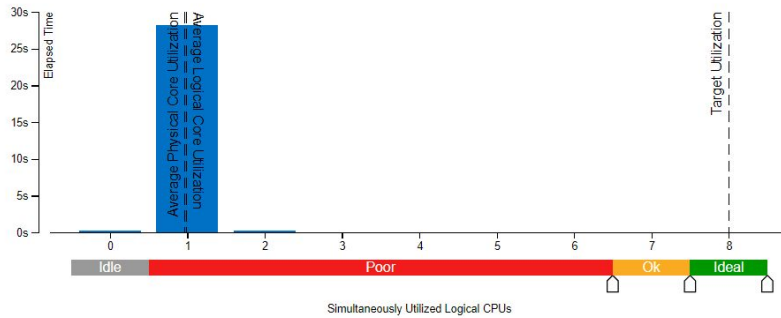
همچنین شکل زیر نشان‌دهنده‌ی cpu utilization برای اجرای سریال کد است. همانطور که میبینید به ما hotspot های کد نشان داده شده و گفته شده که در زمان اجرای کد هر تابع چقدر زمان برده است. تابع put بیشترین تکرار در اجرا و همچنین بیشترین زمان اجرا را دارد و بنابراین عملیات موازی‌سازی باید روی فراخوانی این تابع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



این تصویر نیز نشان‌دهنده‌ی نحوه‌ی تقسیم کار بین thread ها در برنامه‌ی ماست و ارزیابی میکند که تقسیم کار چقدر 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 جابجای 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.

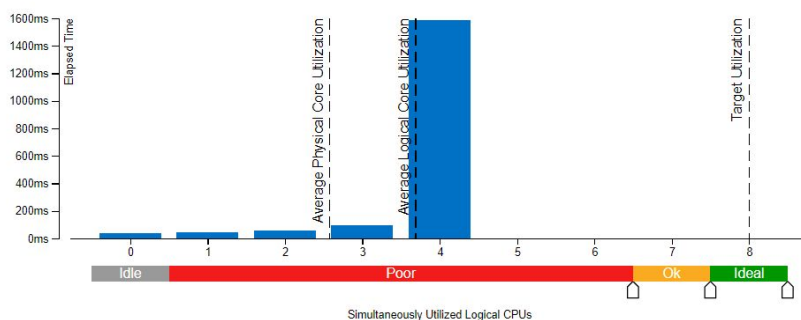
```
Microsoft Visual Studio Debug Console
# solutions 365596 time: 2213

C:\Users\ASUS\Desktop\PP_CA6\PP_CA6\Release\PP_CA6.exe (process 11128) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

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.

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.



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

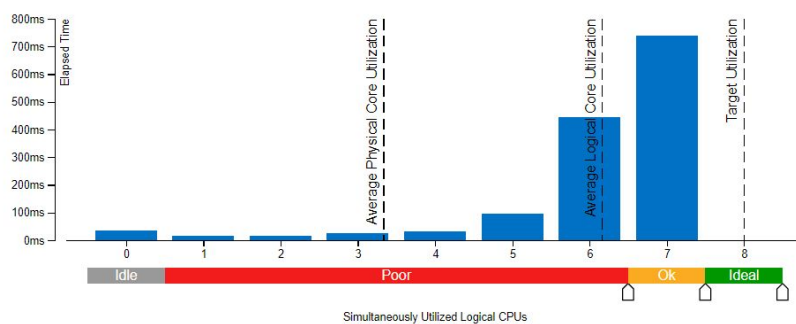
```
Microsoft Visual Studio Debug Console
Kimia Khabiri: 810196606 - Parsa Hoseinnejad: 810196604
# solutions 365596 time: 1563

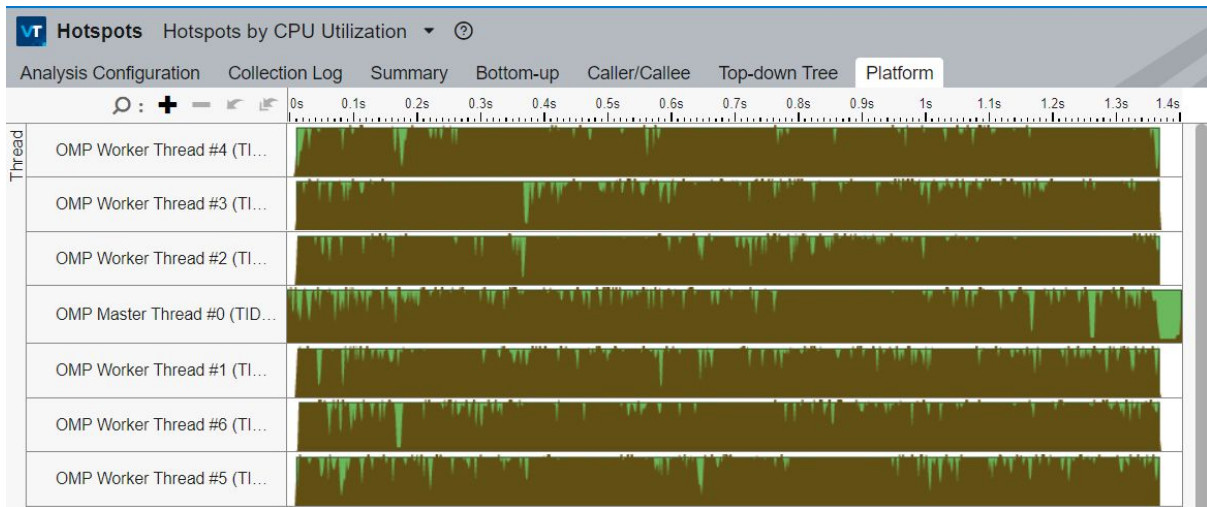
C:\Users\ASUS\Desktop\PP_CA6\PP_CA6\Release\PP_CA6.exe (process 12708) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

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 تقریباً هم اندازه ای انجام دهند.

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.





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.

The screenshot shows the 'Detect Memory Problems' window in Intel Inspector. The 'Problems' table lists two issues:

ID	Type	Sources	Modules	Object Size	State
P1	Mismatched allocation/deallocation	kmp_environment.cpp	libiomp5md.dll		New
P2	Memory leak	[Unknown]	ntdll.dll	5736	New

The 'Filters' panel on the right shows the following counts:

- Severity: Error (2 item(s))
- Type: Memory leak (1 item(s)), Mismatched allocation/deallocation... (1 item(s))
- Source: [Unknown] (1 item(s)), kmp_environment.cpp (1 item(s))
- Module: libiomp5md.dll (1 item(s))

The 'Code Locations: Memory leak' section shows the following details:

Description	Source	Function	Module	Object Size	Offset	Variable
Allocation site	ntdll.dll!0x72ed3	LdrDeleteEnclave	ntdll.dll	956		block

The 'Timeline' panel on the right shows a single event for 'OMP Worker Thread #1 (924)'.