به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

برنامه نویسی موازی

گزارش کار پروژه‌ی چهارم

نرم افزار Intel Parallel Studio

نام و نام خانوادگی	امیرحسین ثمودی – آرمین قاسمی
شماره دانشجویی	810100198 – 810100108
تاریخ ارسال گزارش	1403/10

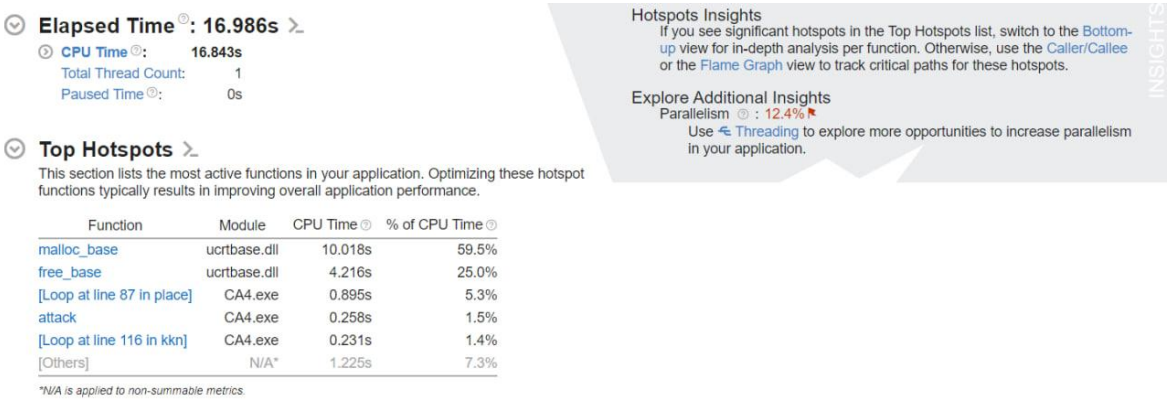
فهرست گزارش سوالات

بخش اول: مسئله K اسب

در این سوال قصد داریم کد داده شده که مسئله K اسب را به صورت بازگشتی حل کرده است را با متودولوژی طراحی 4 مرحله ای برنامه موازی به کمک نرم افزار Intel Parallel Studio موازی سازی کنیم.

گام اول: آنالیز برنامه سریال

ابتدا از طریق Intel VTune Profiler یک تحلیل HotSpot برای برنامه سریال ایجاد میکنیم. در اینجا برای اینکه زمان اجرای برنامه را افزایش دهیم تا تحلیل بر روی آن انجام شود پارامتر های مسئله را تغییر میدهیم. سایز بورد را 7 در 6 قرار داده و تعداد اسب هارا 21 میگذاریم.



Elapsed Time: 16.986s

CPU Time: 16.843s
Total Thread Count: 1
Paused Time: 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
malloc_base	ucrtbase.dll	10.018s	59.5%
free_base	ucrtbase.dll	4.216s	25.0%
[Loop at line 87 in place]	CA4.exe	0.895s	5.3%
attack	CA4.exe	0.258s	1.5%
[Loop at line 116 in kkn]	CA4.exe	0.231s	1.4%
[Others]	N/A*	1.225s	7.3%

*N/A is applied to non-summable metrics.

Hotspots Insights

If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee or the Flame Graph view to track critical paths for these hotspots.

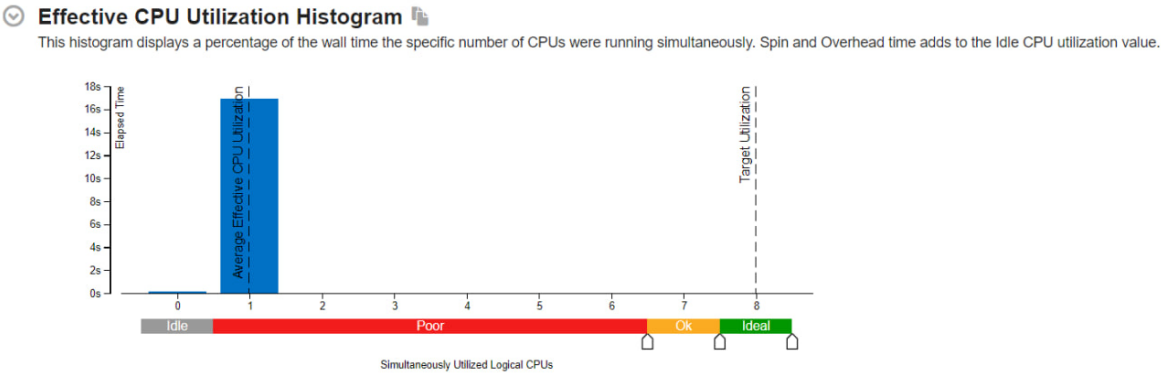
Explore Additional Insights

Parallelism : 12.4%

Use Threading to explore more opportunities to increase parallelism in your application.

INSIGHTS

شکل 1: خلاصه آنالیز برنامه سریال

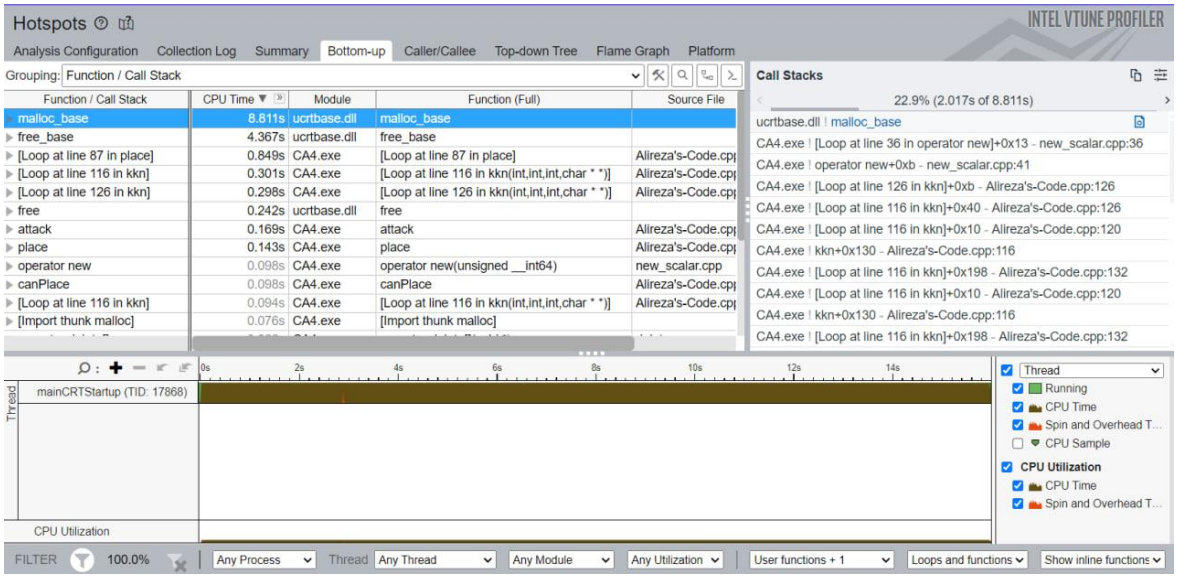


شکل 2: زمان اجرای برنامه روی هر ترد در برنامه موازی

تحلیل: همانطور که در شکل 1 مشخص است بیشتر زمان اجرا صرف اختصاص دادن حافظه و پاک کردن آنها میشود. (حدود 85 درصد زمان اجرا)

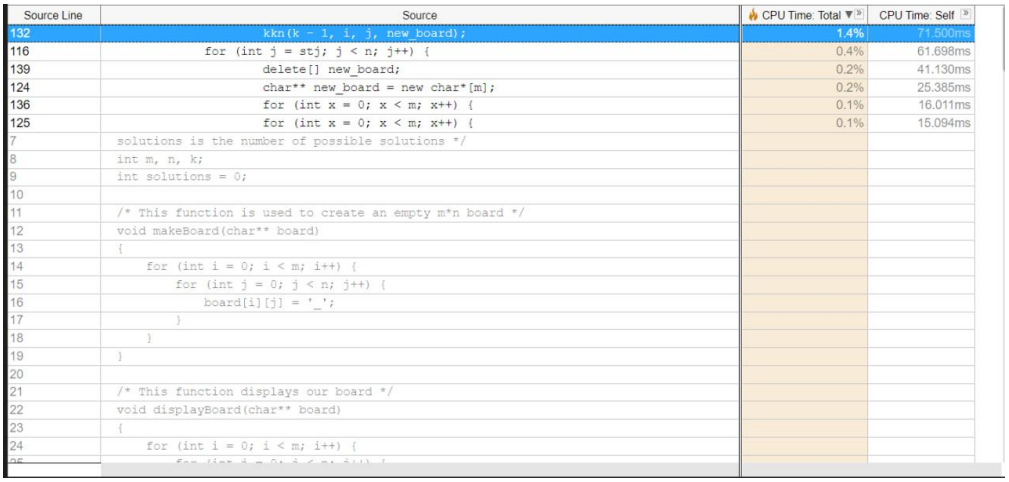
بخش های دیگر کد که زمان اجرای قابل توجهی دارند یکی حلقه اصلی در تابع Knn است (که برروی همه خانه های برد iterate میکند و همچنین حلقه داخل تابع place که مقادیر برد قبلی را روی برد جدید ایجاد شده کپی میکند.)

در شکل 2 میبینیم که تقریبا اجرای برنامه بر عهده یک ترد است که با توجه به ساختار سریال کد همین انتظار را نیز داشتیم)



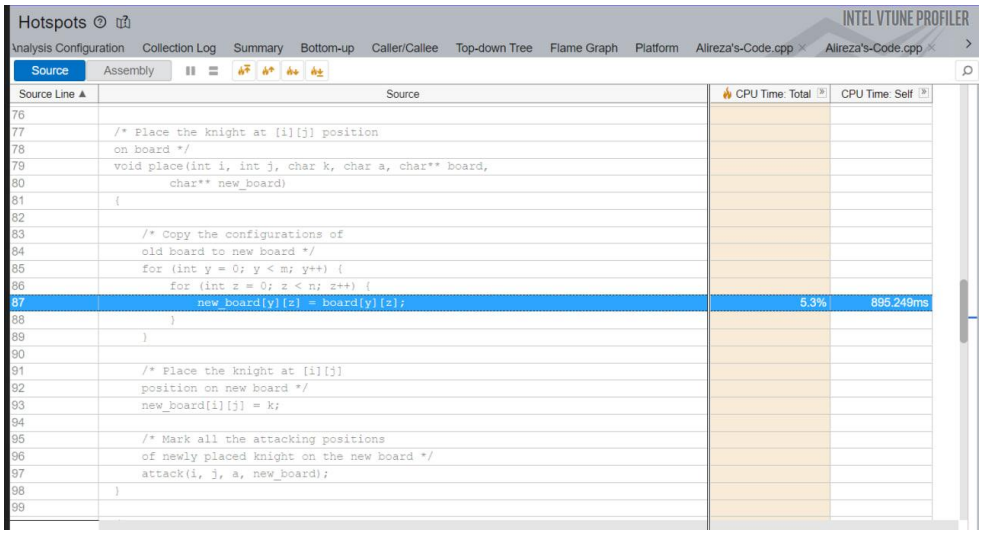
شکل 3: زمان اجرای توابع مختلف

شکل 3 زمان اجرای توابع مختلف را با جزئیات بیشتری نشان میدهد. با کلیک بر روی هر کدام بخش متناظر آن در کد را نشان میدهد.



شکل 4: اجرای تابع **Knn**

شکل 4 بخشی از کد داخل تابع Knn را نشان میدهد که زمان اجرای قابل توجهی به خود اختصاص میدهد.(همانطور که اشاره شده این بخش در یک حلقه تو در تو قرار دارد.)



شکل 5: تابع **Place**

شکل 5 هم بخشی از تابع Place که داخل یک حلقه تو در تو برد را کپی میکند را به عنوان یکی دیگر از Hotspot های برنامه نشان میدهد.

گام دوم: موازی سازی به کمک ساختار های OpenMP

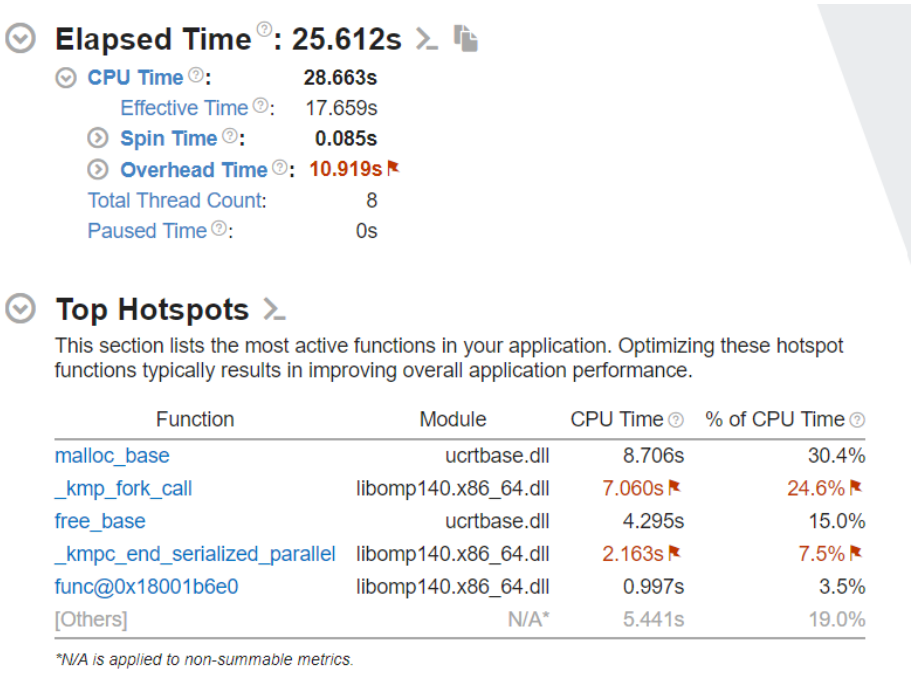
حال که Hotspot های برنامه را شناسایی کرده ایم سعی میکنیم که آنهارا موازی کنیم و نحوه تاثیرگذاری آنرا بر عملکرد برنامه ببینیم.

ابتدا با موازی کردن تابع Place شروع میکنیم.

```
#pragma omp parallel for schedule(static)
for (int y = 0; y < m; y++) {
    for (int z = 0; z < n; z++) {
        new_board[y][z] = board[y][z];
    }
}
```

شکل 6: ساختاری موازی داخل تابع **Place**

حال برنامه را دوباره آنالیز میکنیم.



شکل 7: آنالیز برنامه موازی شده

همانطرو که در شکل 7 میبینیم ساختار موازی ای که اضافه کرده ایم نه تنها باعث بهبود عملکرد نشده بلکه به دلیل عملیات های Synchronization یی که بین ترد ها باید انجام بگیرد زمان اجرا به مراتب بدتر شده است. دلیل این امر میتواند این باشد که حجم محاسبات این بخش آنقدری زیاد نیست که موازی کردن و تحمیل Overhead ناشی از آن به صورت کلی زمان اجرا را کاهش دهد.

پس به سراغ بخش دیگری برای موازی سازی میرویم.

```
#pragma omp parallel for schedule(static)
for (int i = sti; i < m; i++) {
    for (int j = stj; j < n; j++) {

        /* Is it possible to place knight at
osition on board? */
        if (canPlace(i, j, board)) {

            /* Create a new board and place the
            new knight on it */
            char** new_board = new char*[m];
            for (int x = 0; x < m; x++) {
                new_board[x] = new char[n];
            }
            place(i, j, 'K', 'A', board, new_board);
```

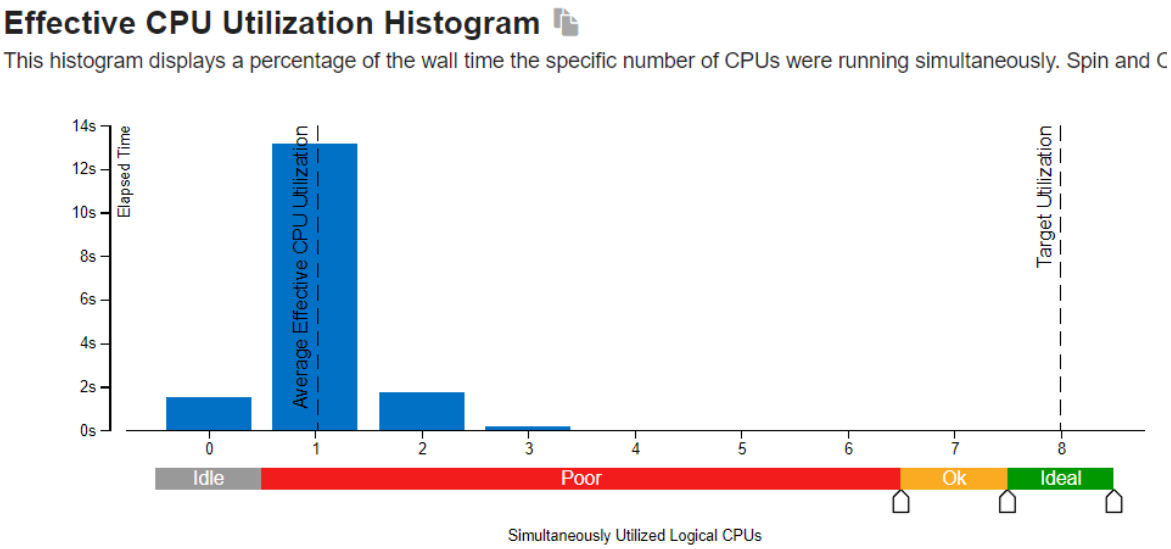
شکل 8: موازی سازی تابع **Knn**

اینبار حلقه داخل تابع Knn را موازی میکنیم.

```
if (k == 0) {
    displayBoard(board);
    #pragma omp atomic
    solutions++;
    return;
}
```

شکل 9: جلوگیری از **data race**

همچنین برای جلوگیری از data race (به دلیل اینکه ترد های مختلف به متغیر گلوبال solutions دسترسی همزمان write نگیرند از ساختار atomic استفاده میکنیم.



شکل 10: زمان اجرا روی ترد های مختلف

مشاهده میکنیم که وضعیت نسبت به حالت قبل کمی بهتر شد ولی باز هم استفاده بهینه ای از ترد ها صورت نگرفته و بخش قابل توجهی از زمان اجرا بر روی یک ترد است.

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
malloc_base	ucrtbase.dll	9.046s	48.3%
free_base	ucrtbase.dll	4.420s	23.6%
[Loop at line 89 in place]	CA4.exe	1.043s	5.6%
func@0x1800090eb	VCOMP140.DLL	0.737s	3.9%
[OpenMP fork]	VCOMP140.DLL	0.638s	3.4%
[Others]	N/A*	2.842s	15.2%

*N/A is applied to non-summable metrics.

شکل 11: **Hotspot** های برنامه

با بررسی Hotspot های برنامه مشخص میشود که باز هم بخش زیادی از زمان اجرا به allocate کردن حافظه جدید اختصاص دارد. این امر به این دلیل میباشد که در هر Recursive Call تابع یک بورد جدید ساخته میشود و قابلیت موازی سازی مارا محدود میکند.

اینبار به سراغ موازی سازی در سطح Task ها می رویم. ساختار کد را به صورت زیر تغییر میدهیم.

```
#pragma omp task
kkn(k - 1, i, j, new_board);
#pragma omp taskwait
```

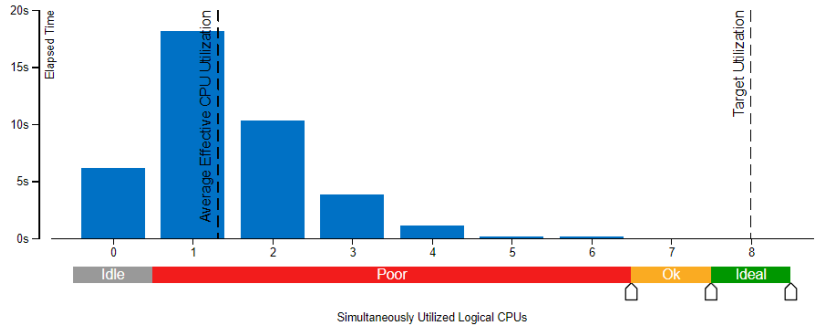
همچنین تابع main را نیز باید به صورت زیر تغییر دهیم:

```
#pragma omp parallel
{
#pragma omp single
    {
        kkn(k, 0, 0, board);
    }
}
```

حال برنامه جدید را آنالیز می نماییم.

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and



با دیدن نمودار بالا ممکن است تصور کنیم اوضاع اجرای برنامه موازی بهتر شده است.

Elapsed Time: 39.742s

CPU Time:	272.167s
Effective Time:	52.383s
Spin Time:	217.905s
Overhead Time:	1.880s
Total Thread Count:	8
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
_kmpc_barrier	libomp140.x86_64.dll	196.589s	72.2%
func@0x18002fc30	libomp140.x86_64.dll	48.952s	18.0%
malloc_base	ucrtbase.dll	11.327s	4.2%
free_base	ucrtbase.dll	5.274s	1.9%
func@0x18003260f	libomp140.x86_64.dll	3.089s	1.1%
[Others]	N/A*	6.936s	2.5%

*N/A is applied to non-summable metrics.

ولی ریپورت بالا نشان میدهد که زمان اجرای برنامه به شدت بدتر شده است. یک دلیل میتواند زمان بالای لازم برای مپ کردن تسک های ایجاد شده مجازی روی هسته های CPU باشد (زمان بالای Spin time نیز همین را نشان میدهد.)

پس در نهایت همان ساختار دوم را به عنوان ساختار موازی برنامه مان انتخاب میکنیم.

گام سوم: دیباگ کردن و برطرف کردن مشکلات

در این بخش باید مشکلات احتمالی ای که در اثر موازی کردن برنامه ایجاد شده اند را شناسایی و برطرف کنیم. البته خروجی کد موازی ما کاملا با خروجی برنامه سریال مطابقت دارد که نشان میدهد کد موازی مشکلی به برنامه اضافه نکرده است.

```
K  A  K  A  K  A
A  K  A  K  A  K
K  A  K  A  K  A
A  K  A  K  A  K
K  A  K  A  K  A
A  K  A  K  A  K
K  A  K  A  K  A

A  K  A  K  A  K
K  A  K  A  K  A
A  K  A  K  A  K
K  A  K  A  K  A
A  K  A  K  A  K
K  A  K  A  K  A
A  K  A  K  A  K


Total number of solutions : 2
```

شکل بالا خروجی برنامه موازی را نشان میدهد که دقیقا خروجی برنامه سریال نیز همین است.

یکی از مشکلات احتمالی ای که میتوانست ایجاد شود در دسترسی ترد های مختلف به متغیر solutions بود که در بخش قبل دیدیم با اضافه کردن ساختار atomic و تغییر مقدار این متغیر داخل آن از دسترسی چند ترد به این متغیر به طور همزمان جلوگیری کرده ایم.

گام چهارم: بهبود برنامه موازی

در بخش دوم ساختار های مختلف موازی سازی را بر روی برنامه امتحان کردیم و مشاهده کردیم که بهترین حالت همان بود که حلقه تو در تو داخل تابع Knn را موازی اجرا میکرد. حال میتوانیم پارامتر هایی

که در دست داریم مانند تعداد ترد ها و یا نوع Scheduling را تغییر داده و تاثیر آنها را بر اجرای برنامه موازی بررسی نماییم.

نوع Scheduling را از Static به dynamic تغییر میدهیم.

Elapsed Time: 22.532s

CPU Time:	25.379s
Effective Time:	17.793s
Spin Time:	0.080s
Overhead Time:	7.506s
Total Thread Count:	8
Paused Time:	0s

Hotspots Insights
If you see significant hotspots
up view for in-depth analysis
or the Flame Graph view to tr

Explore Additional Insights
Parallelism : 9.9%
Use Threading to expl
in your application.

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

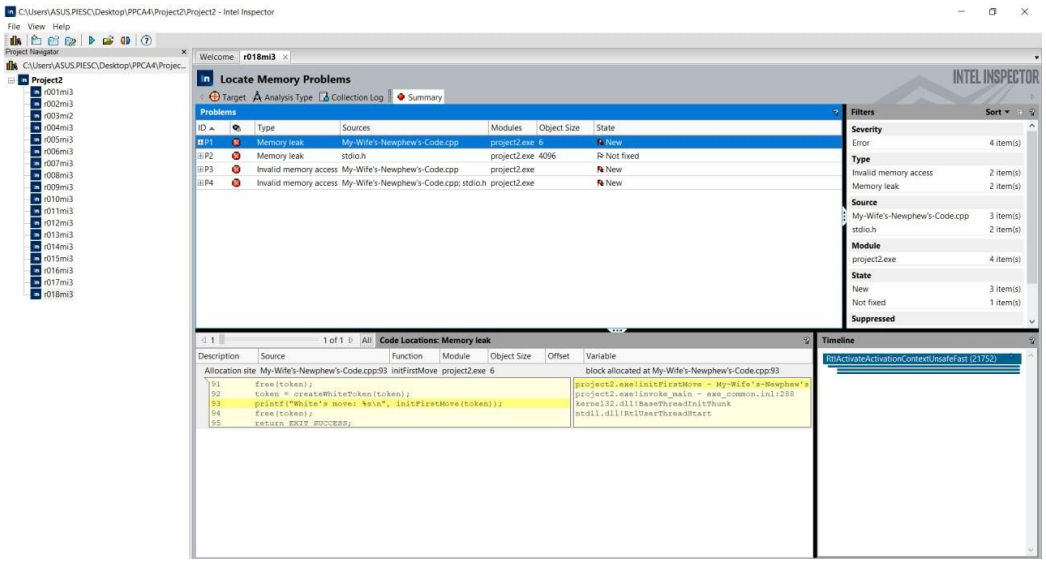
Function	Module	CPU Time	% of CPU Time
malloc_base	ucrtbase.dll	8.855s	34.9%
free_base	ucrtbase.dll	4.445s	17.5%
_kmp_fork_call	libomp140.x86_64.dll	4.010s	15.8%
_kmpc_end_serialized_parallel	libomp140.x86_64.dll	1.181s	4.7%
[Loop at line 89 in place]	CA4.exe	1.122s	4.4%
[Others]	N/A*	5.767s	22.7%

*N/A is applied to non-summable metrics.

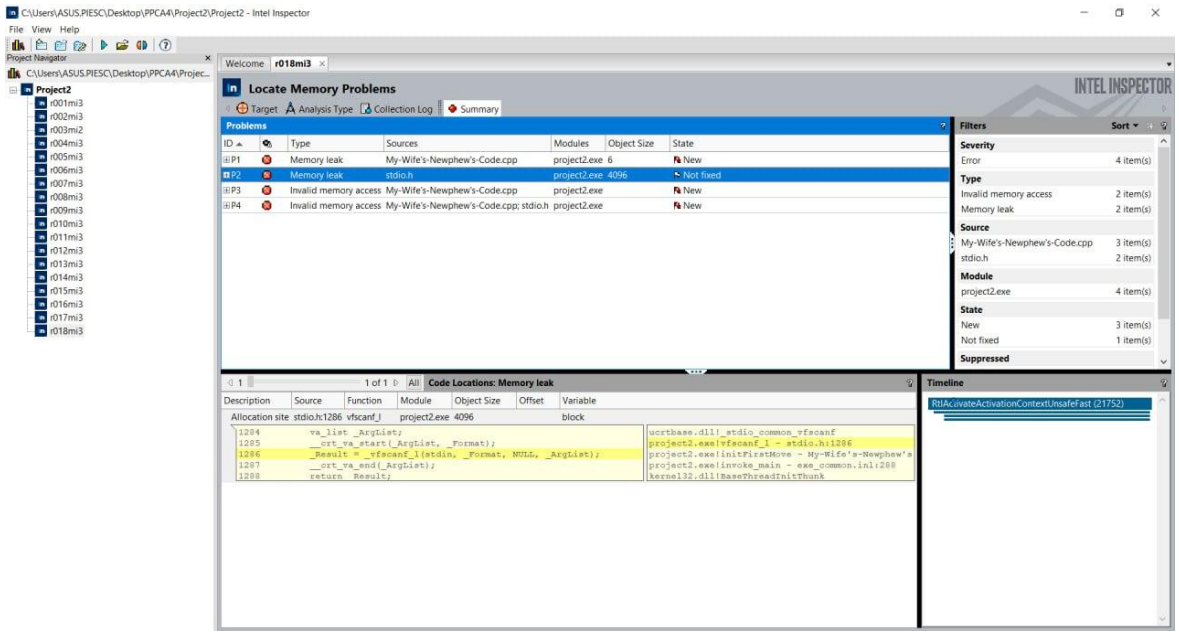
همانطور که میبینیم با این کار وضعیت اجرای برنامه بدتر شد و Overhead زمانی تحمیل شده به برنامه بخاطر نوع Scheduling افزایش یافته است.

بخش دوم: کد خواهرزاده Danny Rensch

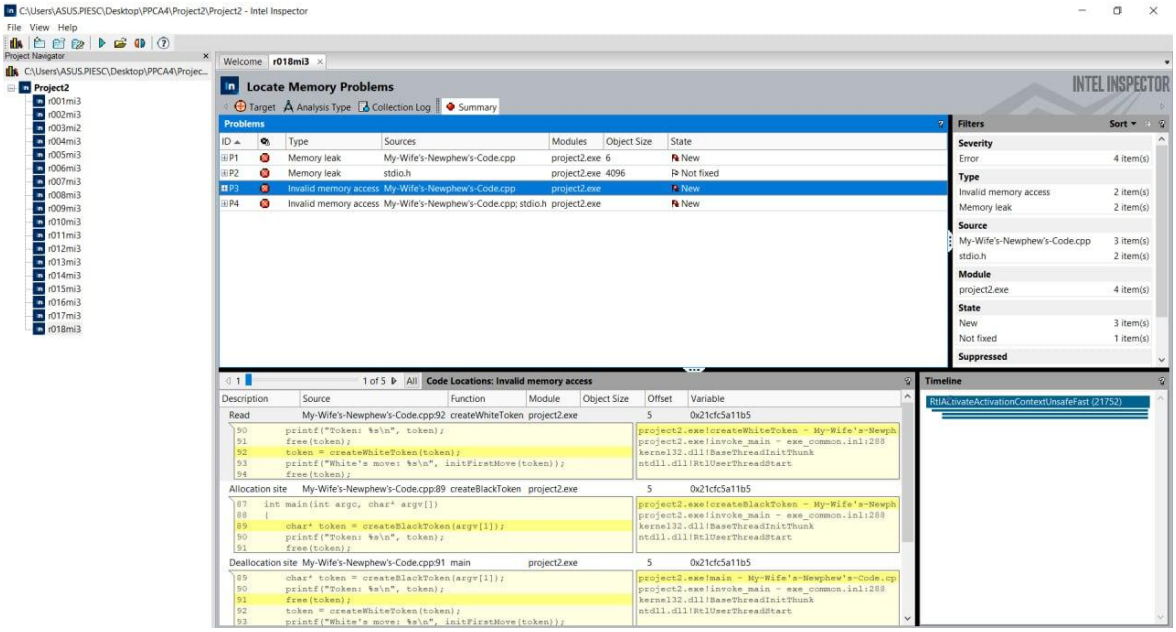
ابتدا کد داده شده را توسط ابزار Intel Inspector آنالیز میکنیم تا مشکلات مربوط به حافظه کد را پیدا کنیم.



شکل 12: مشکل 1

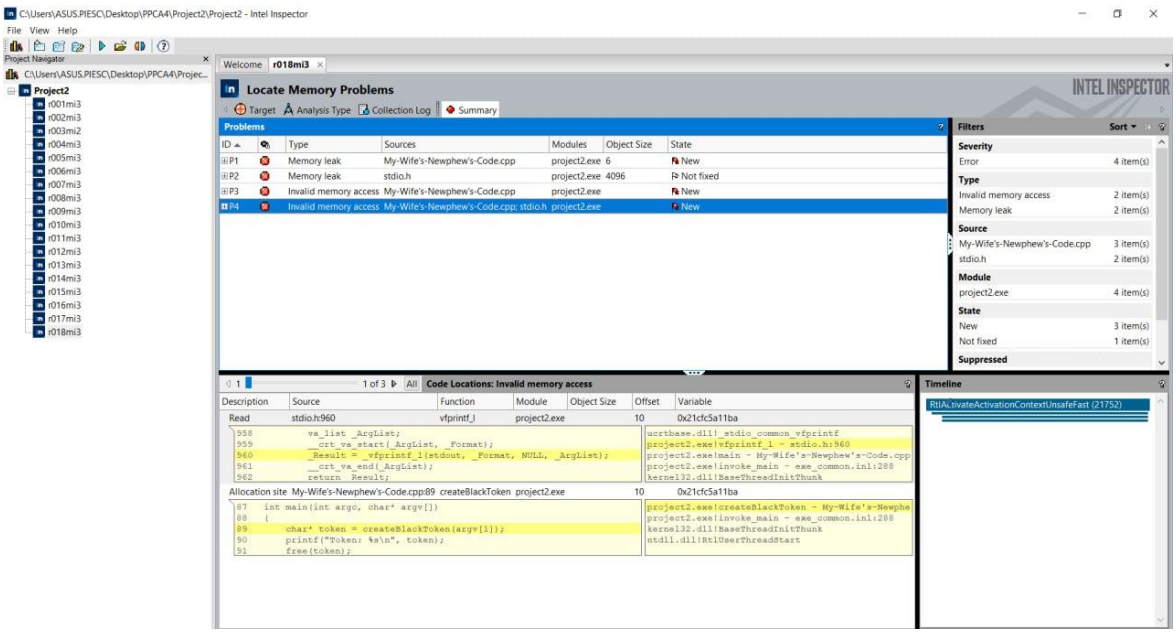


شکل 13:مشکل 2



شکل 14:مشکل 3

...



شکل 15:مشکل 4

مشکلات بالا توسط ابزار اینتل شناسایی شده اند. مشکل دوم ربطی با کد ما ندارد و در کتابخانه stdio.h شناسایی شده است.

مشکل اول در خط 93 به دلیل استفاده از پوینتری که قبلا free کرده ایم ایجاد شده است.

برطرف کردن مشکلات کد:

تابع createBlackToken:

در صورتی که name برابر Null باشد در تابع main اقدام به free کردن مموری ای میکنیم که اصلا اختصاص داده نشده است پس کد را به صورت زیر تغییر میدهیم تا نال بودن name را هندل کند.

```
char* createBlackToken(const char* name)
{
    if (name == NULL) return NULL;

    char* tokenHolder = (char*)malloc(TOKEN_SIZE);
    if (!tokenHolder) return NULL;

    for (int i = 0; i < TOKEN_SIZE; ++i)
    {
        srand(time(0));
        tokenHolder[i] = rand() % 255;
    }
    tokenHolder[TOKEN_SIZE - 1] = '\0';
    return tokenHolder;
}
```

همچنین یک null character به انتهای آن اضافه کرده تا برای چاپ مشکلی نداشته باشد.

تابع createWhiteToken:

این تابع previousToken را برمیگرداند و در بعضی مواقع free مینمود که باتوجه به free شدن در مین دوبار free میشد.

```
char* createWhiteToken(const char* previousToken)
{
    if (previousToken == NULL) return NULL;

    char* currentToken = (char*)malloc(TOKEN_SIZE);
    if (!currentToken) return NULL;

    for (int i = 0; i < TOKEN_SIZE; ++i)
    {
        if (i < (int)strlen(CHESS_TOKEN))
            currentToken[i] = CHESS_TOKEN[i];
        else
            currentToken[i] = previousToken[i] + 1;
    }
    currentToken[TOKEN_SIZE - 1] = '\0';
    return currentToken;
}
```

تابع initFirstMove:

این تابع whiteToken را free میکرد بی آنکه تابع main از آن با خبر باشد که باعث دستری به پوینتر پاک شده میشد.

یک newToken داخل آن تعریف شده و این توکن برگشت داده شده است.

تابع main:

```c
int main(int argc, char* argv[])
{
    if (argc < 2)
    {
        fprintf(stderr, "Usage: %s <name>\n", argv[0]);
        return EXIT_FAILURE;
    }

    char* blackToken = createBlackToken(argv[1]);
    if (blackToken)
    {
        printf("Black's token: %s\n", blackToken);
        char* whiteToken = createWhiteToken(blackToken);
        if (whiteToken)
        {
            char* newToken = initFirstMove(whiteToken);
            free(whiteToken);
            whiteToken = newToken;
            printf("White's move: %s\n", whiteToken);
            free(whiteToken);
        }
        free(blackToken);
    }

    return EXIT_SUCCESS;
}
```

این تابع به صورت بالا تغییر کرد تا توکن برگشتی تابع initFirstMove به درستی freeشود.

خروجی تحلیلگر با کد ادیت شده به صورت زیر است