به نام خدا

تمرین کامپیوتری چهارم - intel parallel studio

امیرحسین راحتی 810100144

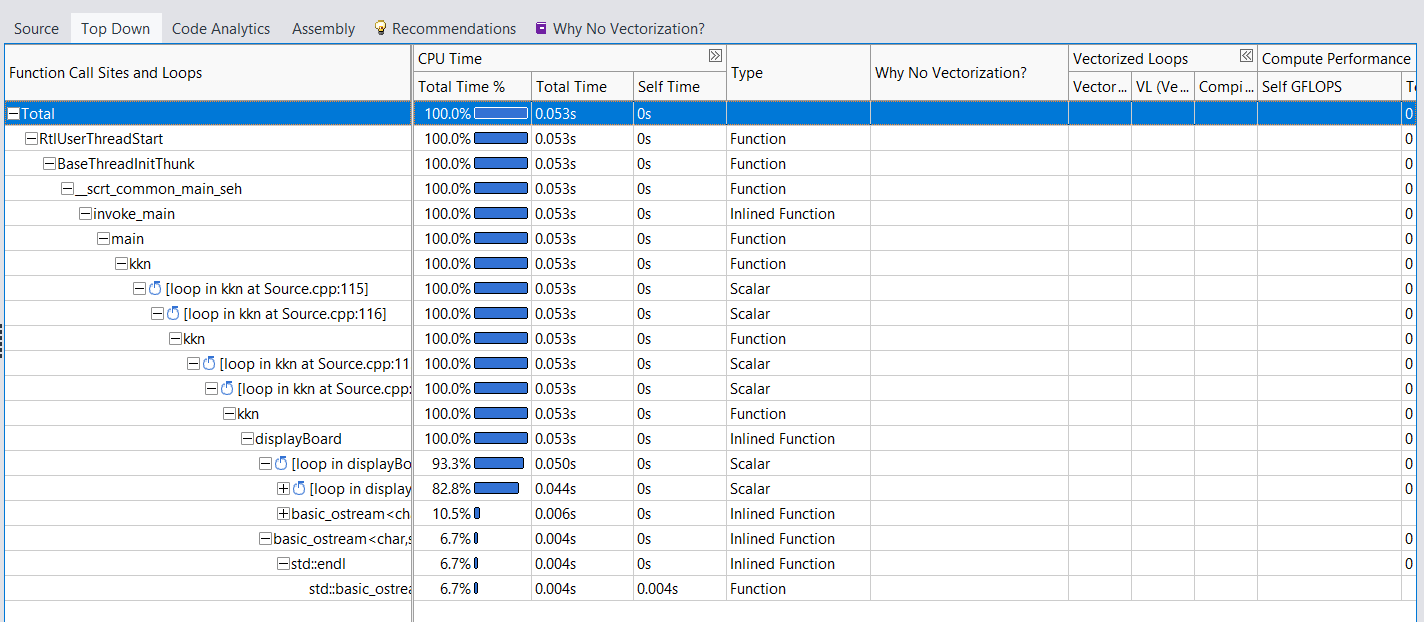
فربد عظیم محسنی 810100187

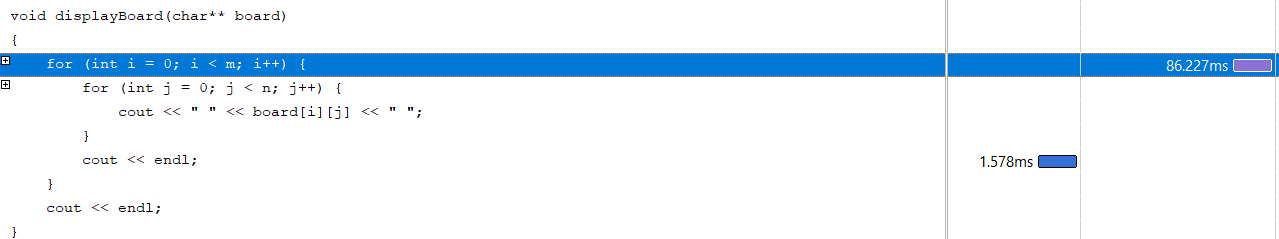
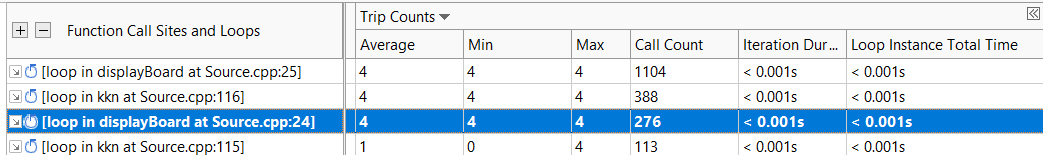
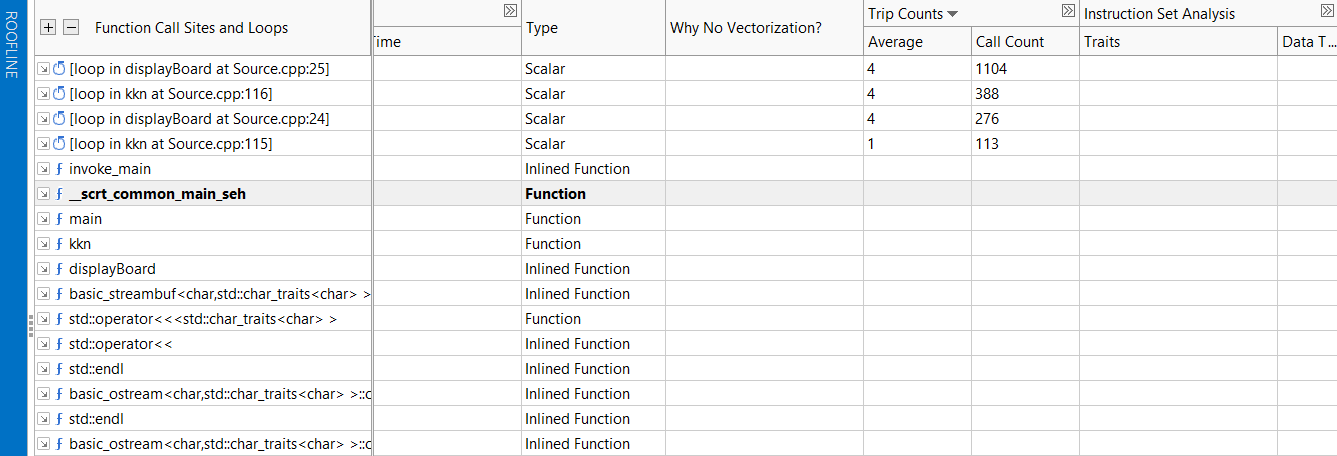
سوال اول

مرحله Analyze:

گزارش hot spot های کد در ابتدا و قبل از موازی سازی به شکل زیر است:

در کد داده شده امکان هیچ نوع بهینه سازی از نوع وکتوری امکان پذیر نمی باشد چرا هیچ عملیات ریاضی انجام نمی شود.

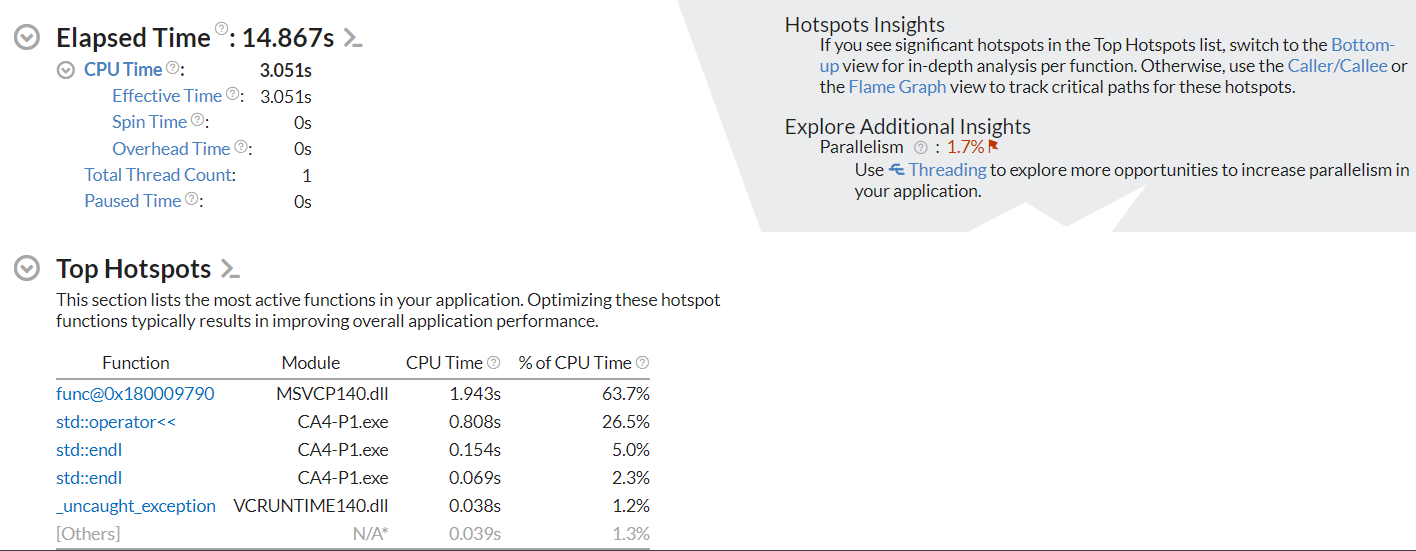
و برنامه advisor intel نیز نمی تواند کد را وکتوری کند چرا که اصلا برنامه ما قابلیت آن را ندارد.

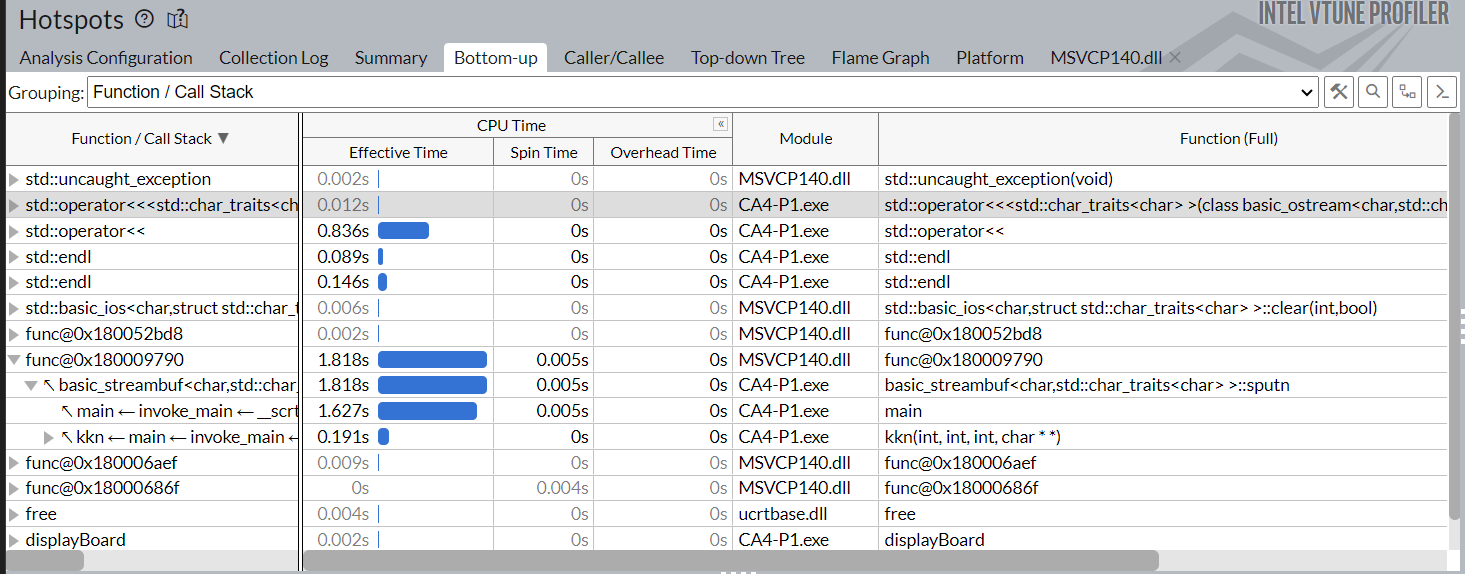
ابزار advisor تعداد تکرار های حلقه را نیز به ما نشان می دهد:

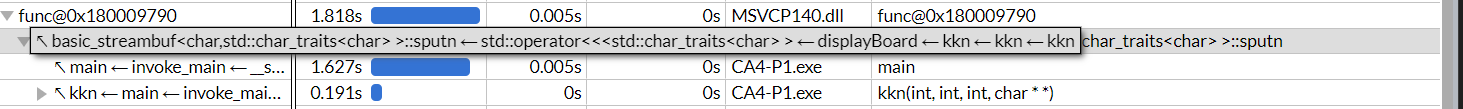
همانطور که مشخص است تابع displayBoard به تعداد دفعات زیادی اجرا می شود و هر بار صدا شدن آن حدود 86 میلی ثانیه طول می کشد و احتمالا با موازی کردن این تابع gain ما خیلی زیاد شود.

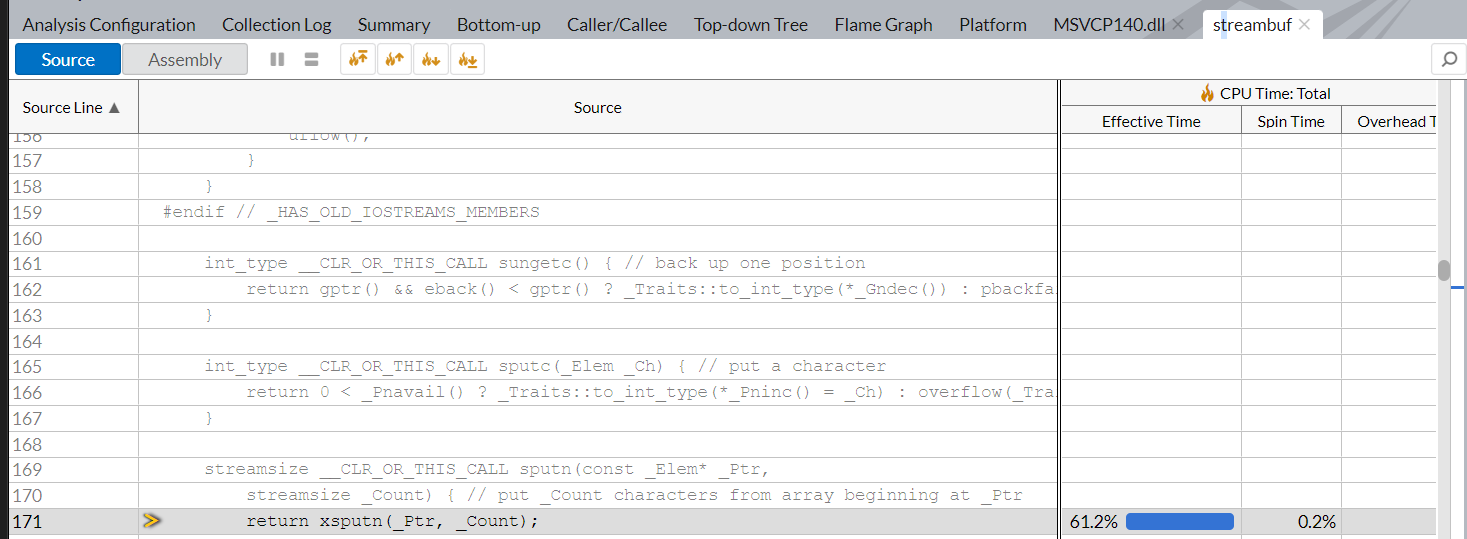
در حالت کلی اجرای برنامه ما در حالت پایه و با یک نخ به اندازه 0.88 ثانیه طول کشیده است.

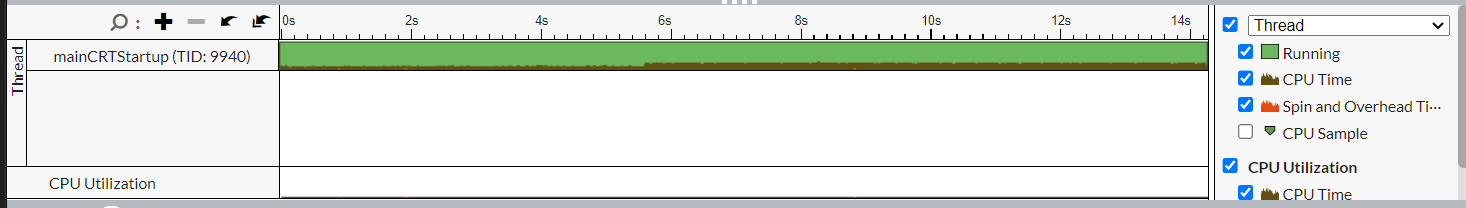


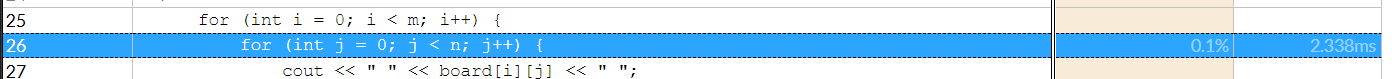
حال از ابزار VTune برای بررسی دقیق تر کد استفاده می کنیم. برای اینکه VTune بتواند کد ما را تحلیل کند لازم است زمان اجرای برنامه افزایش یابد. خلاصه تحلیل VTune با پارامتر های m = 6, n = 6 , k = 4 به شکل زیر می باشد:

همانطور که مشخص است بیشتر زمان برنامه متعلق به چاپ کردن است و CPU time ما فقط 3 ثانیه از برنامه بوده است. بررسی هات اسپات های کد به ما نشان خواهد داد که در کدام قسمت کد بیشترین زمان صرف شده است:

بررسی هات اسپات های کد نشان می دهد که بیشتر وقت cpu صرف نوشتن بر روی serial port می شود چرا که به تعداد دفعات زیادی function call های مربوط به آن در کد وجود دارد:

همچنین همانطور که مشخص است cpu time ما بسیار پایین است:





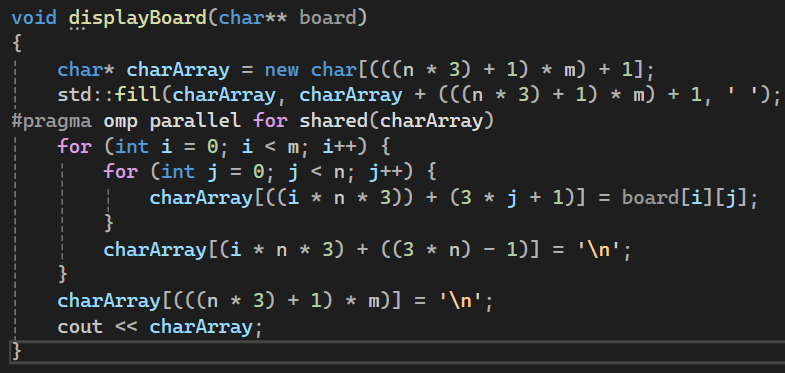
این تنها قسمتی از کد ما است که بخشی از هات اسپات است و بقیه هات اسپات ها بخشی از source نیستند

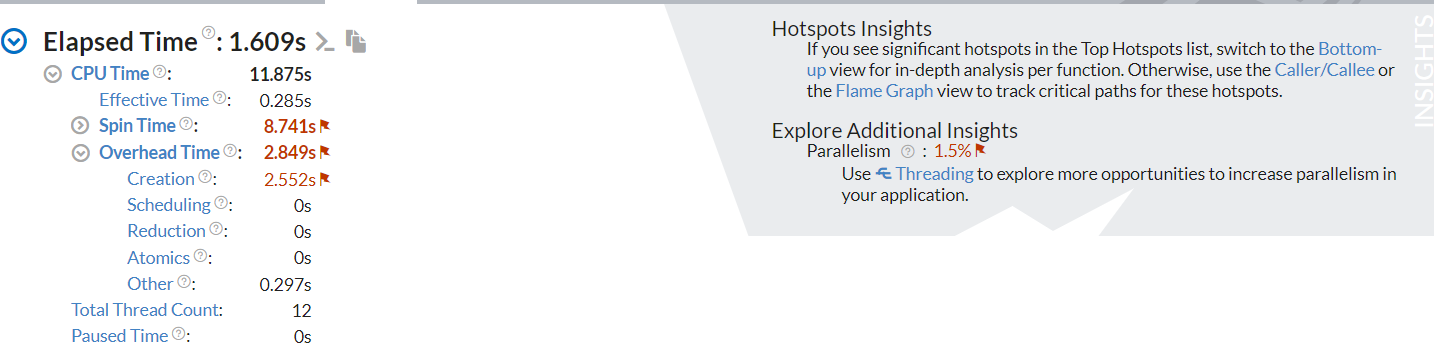
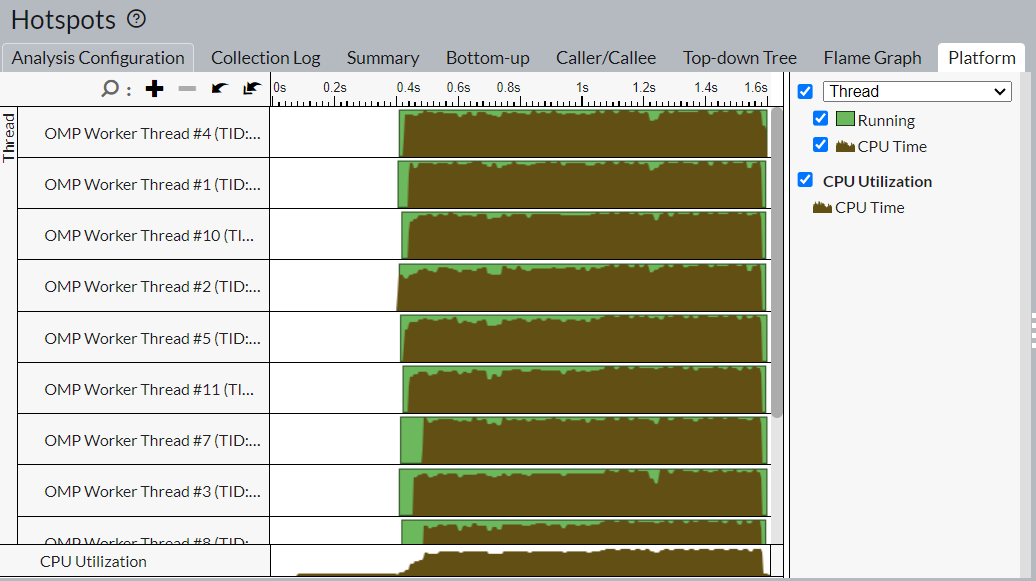
مرحله Construct:

تحلیل های بخش قبل نشان می دهد که برای optimization باید موارد زیر را انجام دهیم:

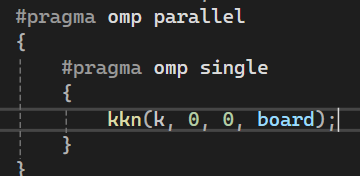
* به جای تک به تک نوشتن بر روی بافر خروجی، ابتدا یک رشته کامل بسازیم و سپس آن را بنویسیم تا cpu time بیشتری بگیریم
* حلقه های kkn را با openMP موازی کنیم
* حلقه ها placeBoard را با openMP موازی کنیم

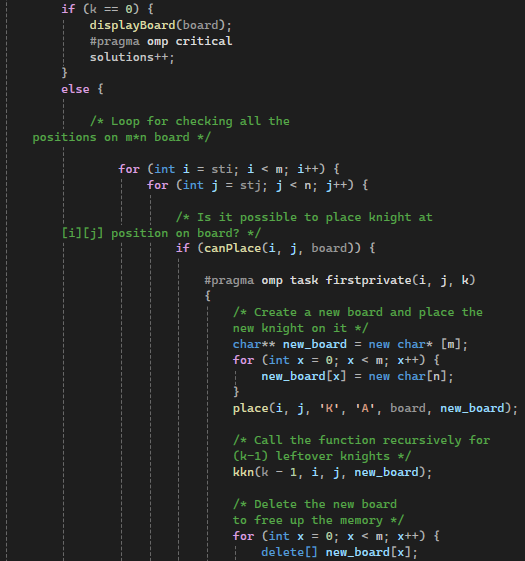
موازی کردن displayBoard:

با تغییر کد به شکل زیر به صورتی که ابتدا کل board در یک آرایه جدید نوشته شود و بعد کلا چاپ شود به شکل زیر پیاده سازی شده است و برای موازی کردن لوپ بیرونی از openMP استفاده شده است و در کل cpu time ما بسیار بالا رفته است:

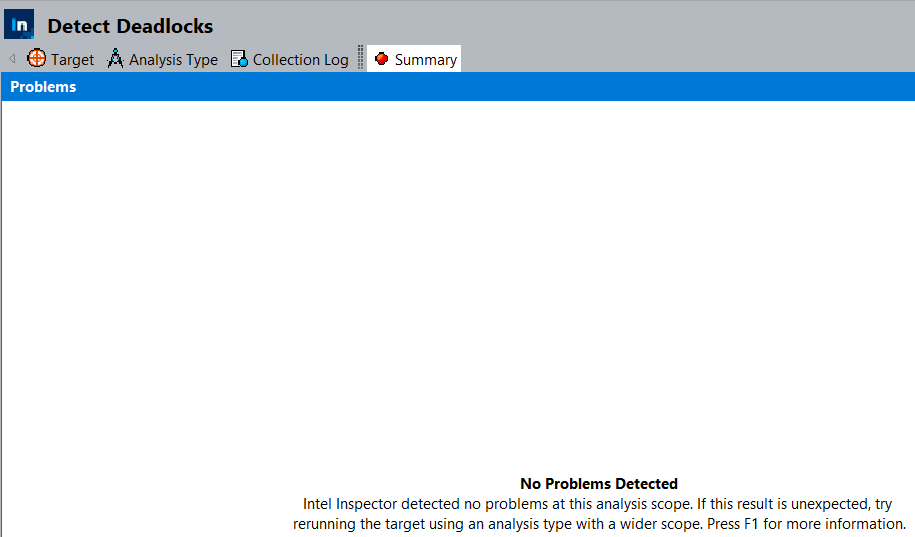
و اجرای زمان برنامه بسیار کاهش می یابد:

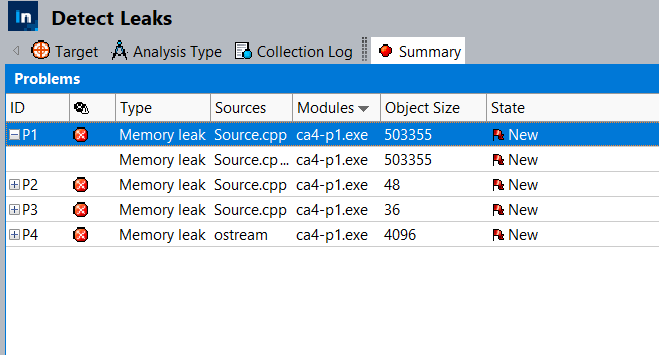
در نهایت هات اسپات دیگر کد که مربوط به اجرا بازگشتی تابع kkn است را با استفاده از OpenMP task موازی میکنیم.

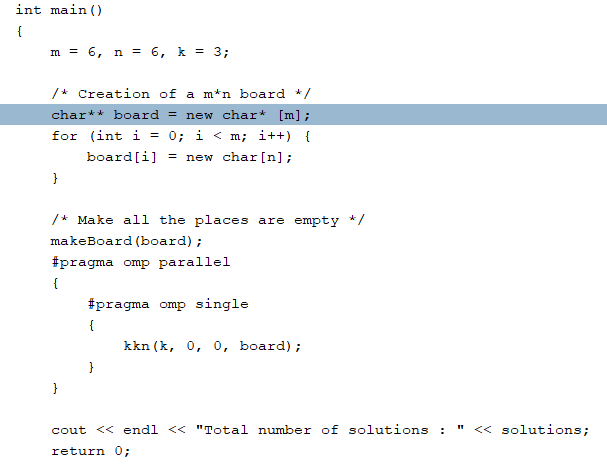
موازی کردن kkn:



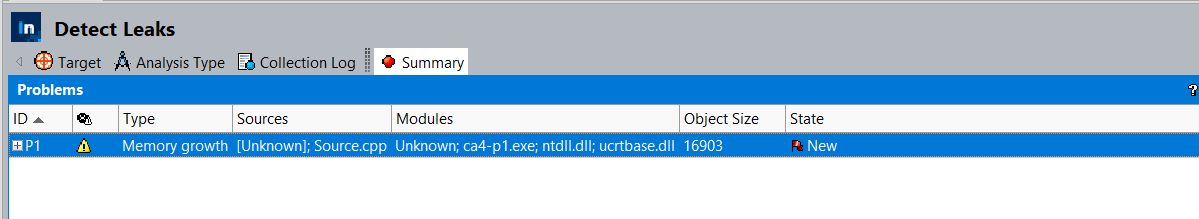
با استفاده از openMP task ما تکرار های تابع kkn را با thread ها واگذار می کنیم. استفاده از این قابلیت البته می تواند خوب نباشد چرا که متغیر solutions گلوبال است و مجبوریم از critical section داشته باشیم که این خود overhead زیادی دارد و در قسمت آخر این مشکل را حل خواهیم کرد.

مرحله Debug:

در قسمت threading هیچ deadlock وجود ندارد.



مشکل اول این است که board حافظه خود را آزاد نمی کند.

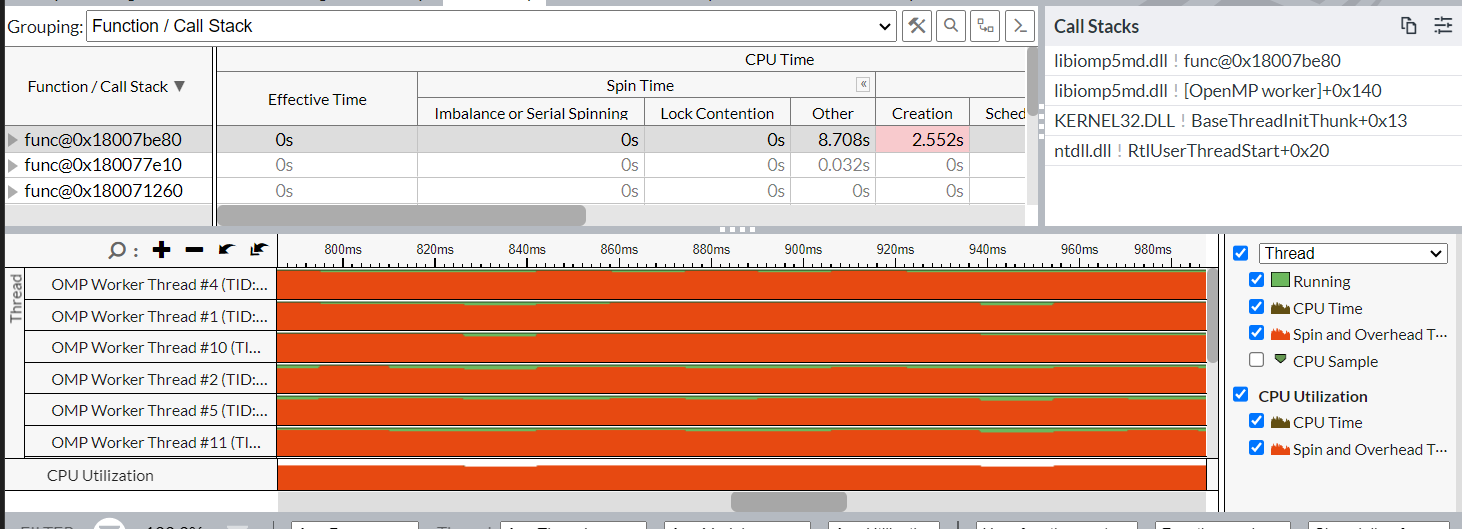
پس از اصلاح این بخش دیگر هیچ اروری دریافت نمی کنیم:

مرحله tuning:

در این مرحله سعی می کنیم قسمت های موازی برنامه را سریعتر بکنیم.

در قسمت موازی سازی displayBoard مشکلی که به وجود می آید این است که overhead ساخت نخ ها خیلی زیاد است و ترد های ما کار زیادی انجام نمیدهند.

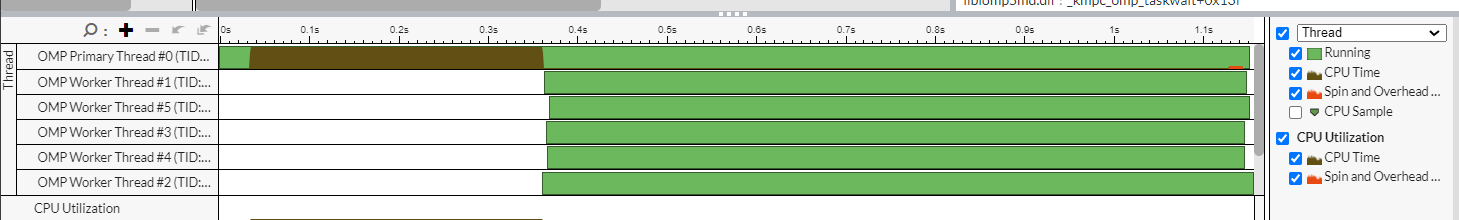
هات اسپات مربوط به overhead موازی سازی openMP در displayBoard:

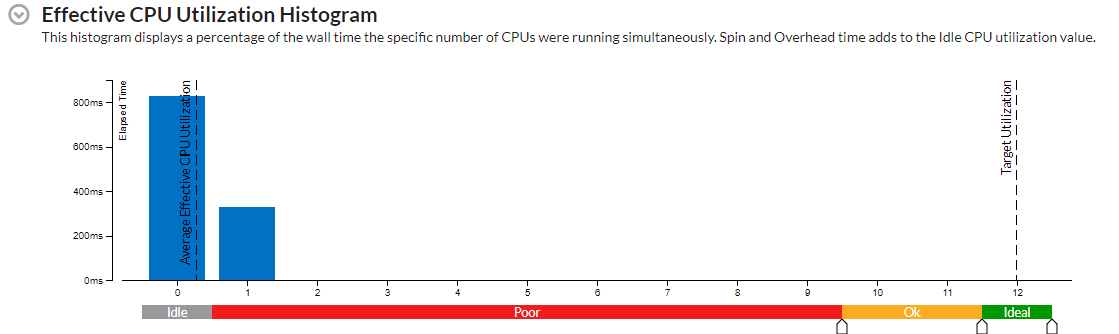


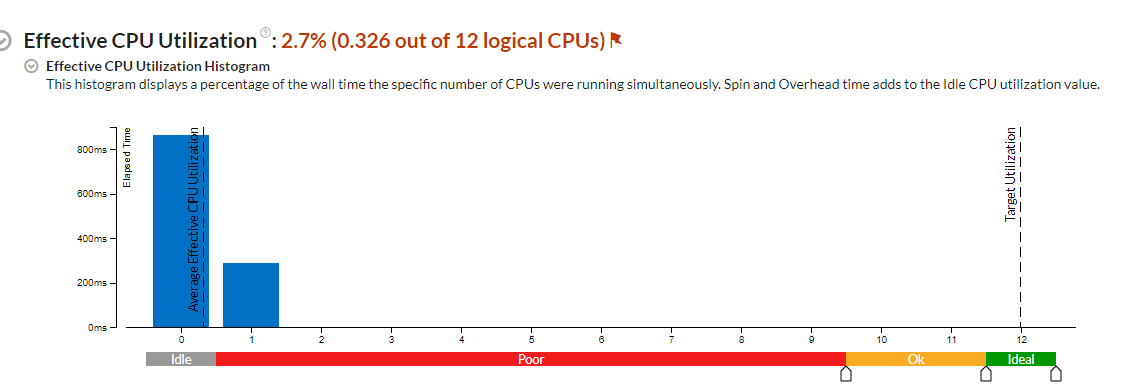
با برداشتن کد موازی این بخش، سرعت به شکل زیر تغییر میکند:



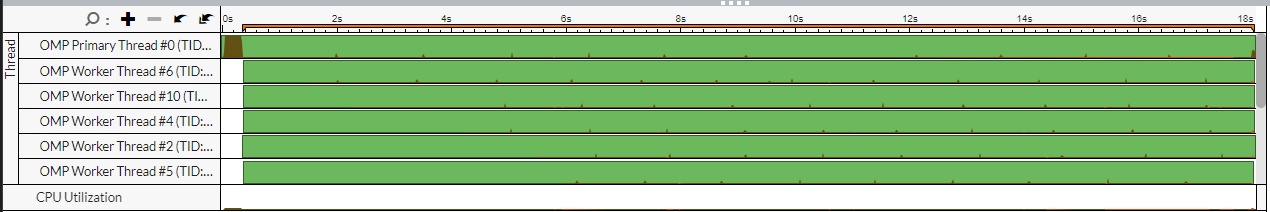
به نظر می رسد عملا فرقی ندارد که این تابع با openMP موازی شود یا نه و ما داریم الکی هزینه می کنیم بدون اینکه هیچ gain به دست آوریم.

در کل برای تابع kkn هم موازی سازی ما زیاد به صرفه نبوده است چرا که بیشتر زمان برنامه نخ ها در حالت idle هستند:  
و cpu utilization ما اصلا خوب نیست:

کار دیگری که میتوانیم بکنیم این است که displayBoard را به یک نخ دیگر واگذار کنیم و کلا دو نخ داشته باشیم که یکی مسئولیت اجرای برنامه اصلی و دیگری نیز مشغول اجرای displayBoard باشد که عملا چون آن عملیات صرفا چاپ کردن است تفاوت زیادی در utilization ایجاد نمی کند و یک thread همچنان بیکار است.



حتی با تغییر پارامتر های دیگر نیز نمی توان از این بهینه تر کرد چرا که ذاتا برنامه ما cpu محور نیست و بیشتر وقتش را صرف io کردن و یا فراخوانی های بازگشتی می کند.



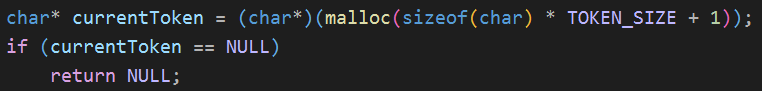
سوال دوم

در این قسمت، به بررسی مشکلات حافظه قطعه کد دوم می پردازیم .

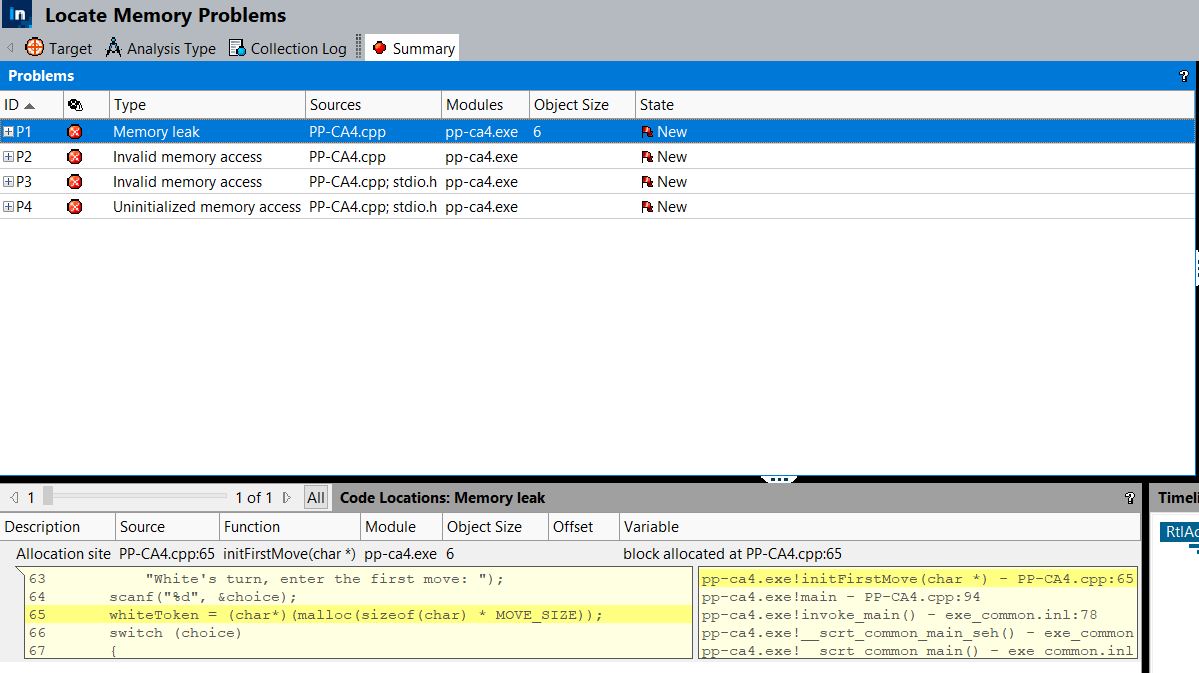
ابتدا با ابزار inspector به بررسی مشکلات حافظه ، ددلاک ، دیتا ریس و memory leak میپردازیم . با اجرای برنامه در محیط inspector ، خروجی تحلیل به شکل زیر است که چهار مشکل را نشان می دهد.

بخش Locate memory problems:

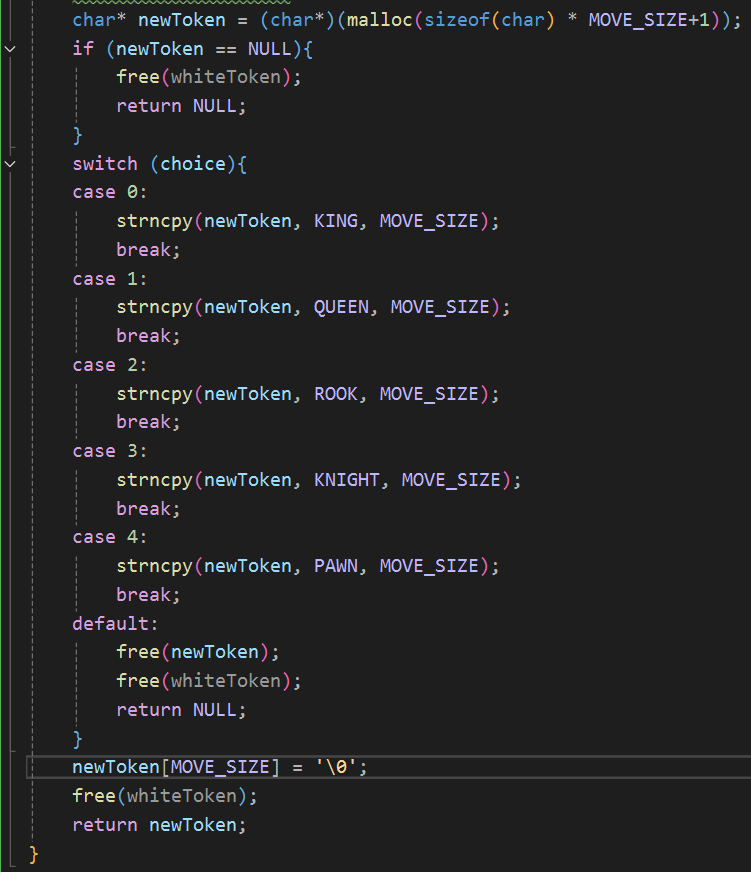
یک مشکلی که به دفعات در کد تکرار شده ، عدم چک کردن تخصیص درست مموری است که باید در یک شرط آن را چک کنیم . این کار به عنوان نمونه در شکل زیر انجام داده شده است:

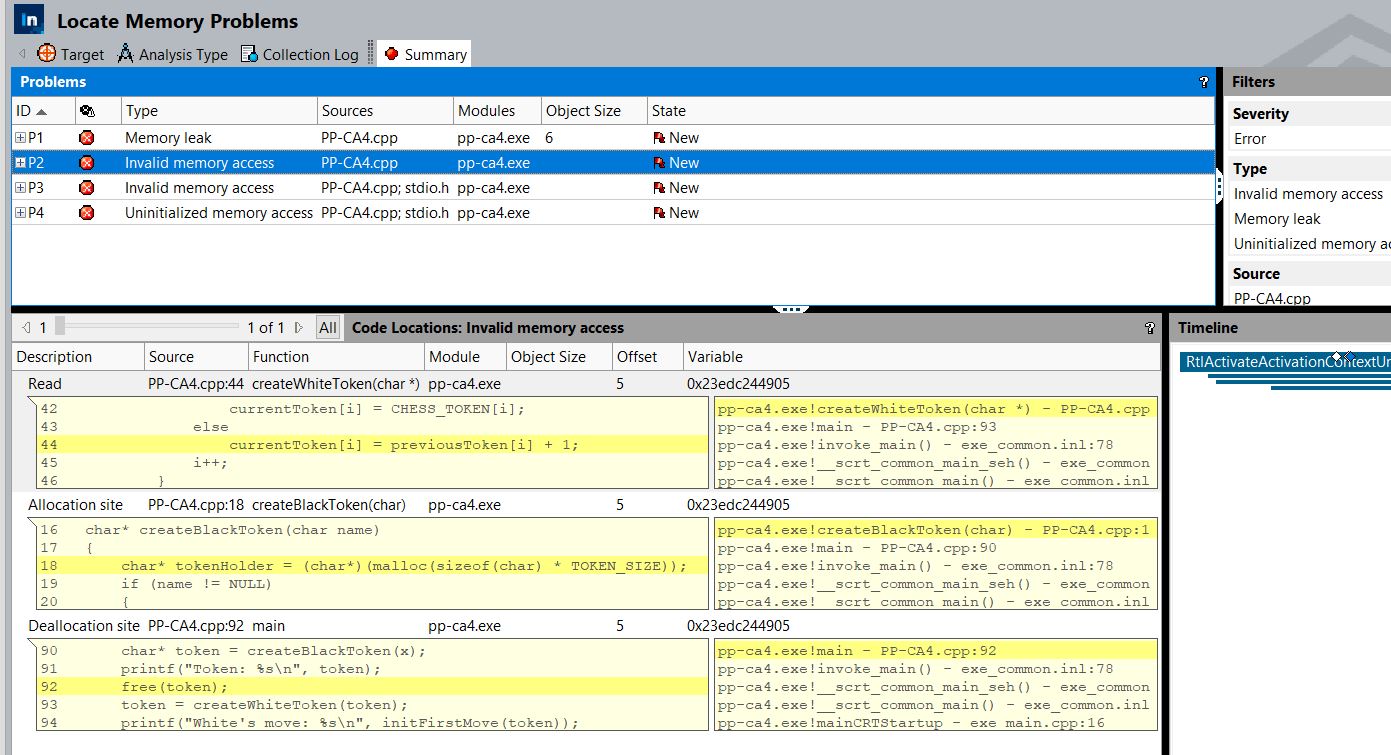


خروجی تحلیل inspector:

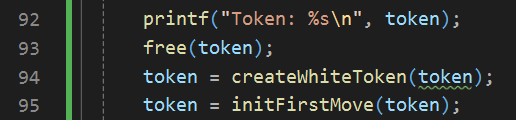


در اینجا ، پوینتر whiteToken به عنوان آرگومان به تابع پاس داده شده و از قبل allocate شده بود ولی ما بدون آزاد کردن آن مجددا آن را allocate کردیم که memory leak ایجاد میکند. برای حل این مشکل ، پوینتر قبلی را آزاد میکنیم و یک پوینتر جدید allocate میکنیم را کار را ادامه میدهیم و در نهایت آن را برمیگردانیم . سپس آن را در main آزاد میکنیم :





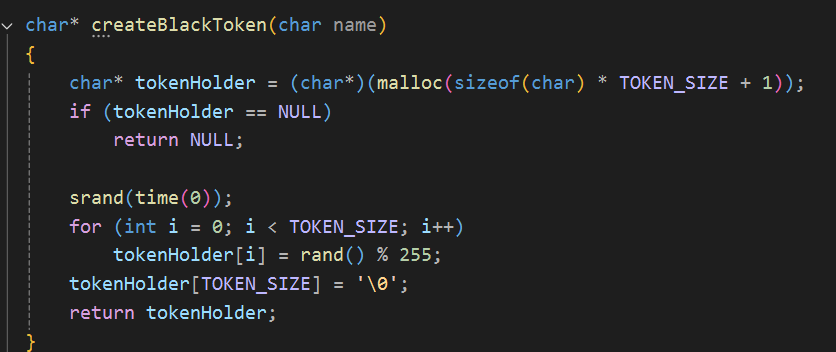
Free کردن token در شکل زیر باعث مشکل در تابع بعدی می شود . آن را حذف میکنیم

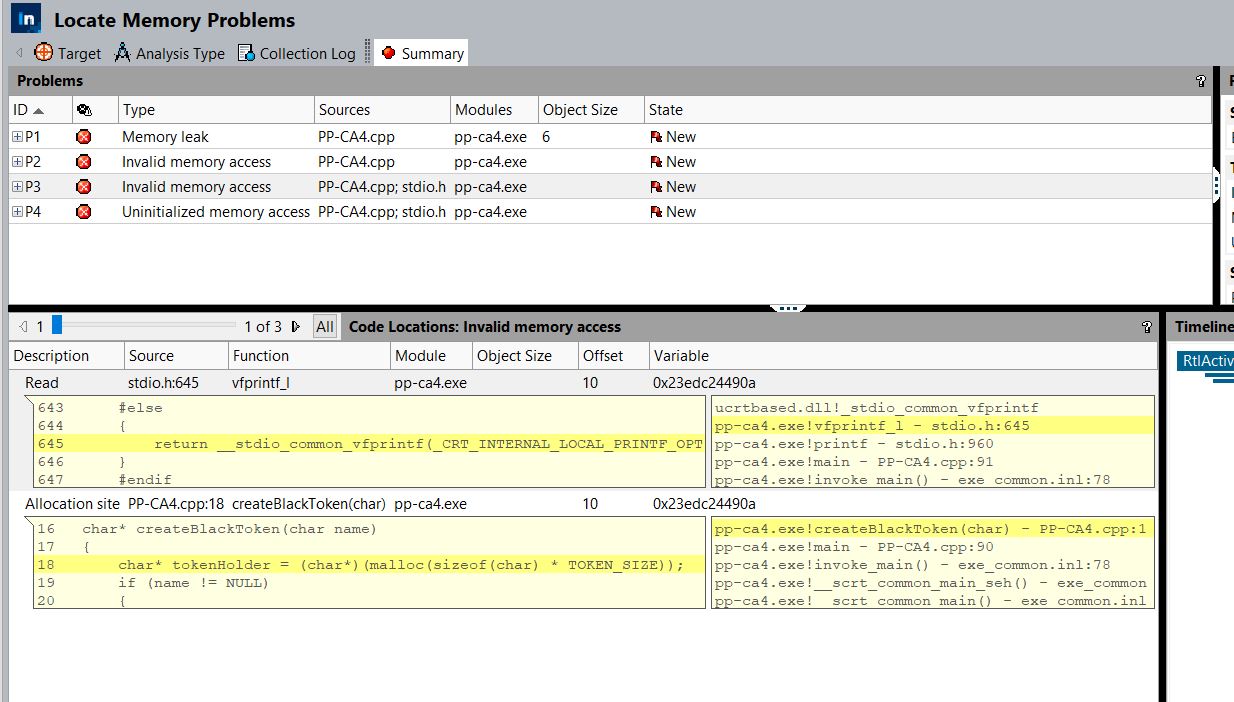


برای حل مشکل خط 44 ، تابع createWhiteToken را به همراه چند checking اضافه بهبود می دهیم . همچنین حافظه ای که در تابع createBlackToken اختصاص دادیم و ریترن کردیم ، اینجا به عنوان previousToken است . باید پس از اتمام کار آن را free کنیم که در این تابع انجام شده است . اینگونه مشکل خط 18 هم حل میشود.



سپس در تابع ، نتیجه تخصیص حافظه را چک میکنیم . تابع را کمی بهتر میکنیم . تابع جدید به شکل زیر است.

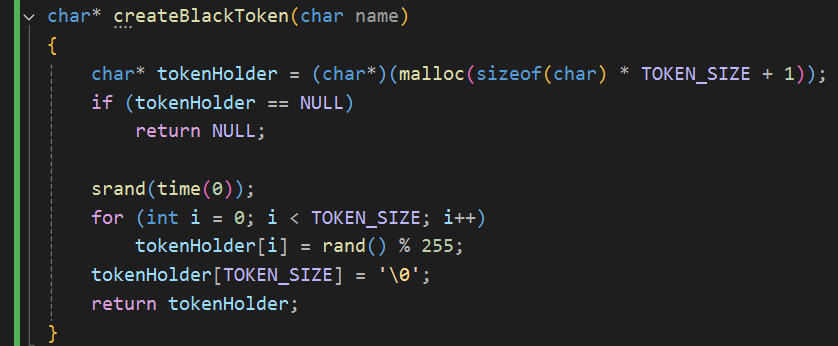


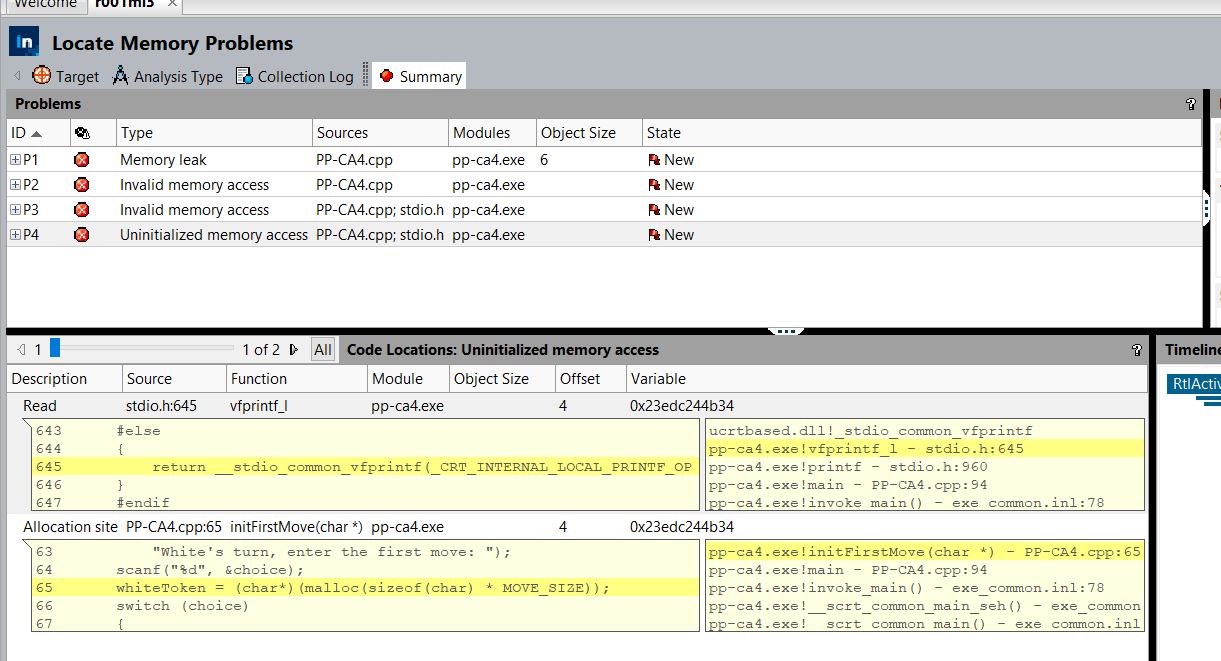


این مشکل به این دلیل است که در حلقه فور در داخل createBlackToken , مقدار i طوری پیش میرود که در پیمایش آخر میخواهد به حافظه ای خارج از محدوده تخصیص داده شده دسترسی پیدا کند:

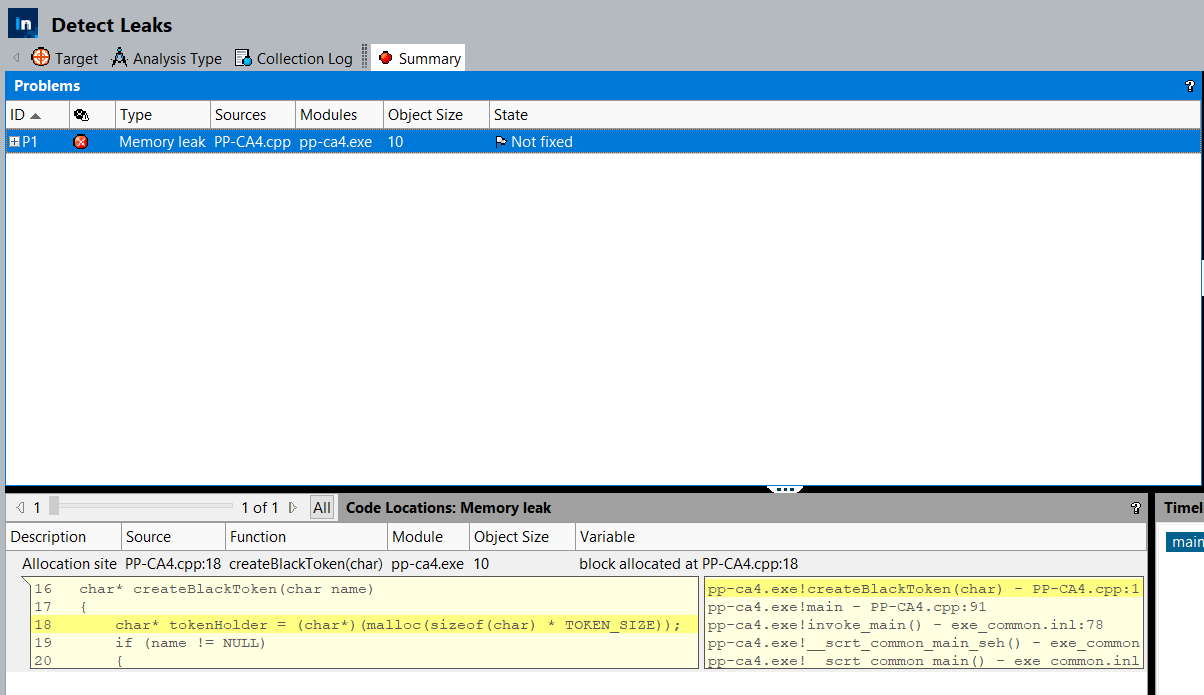


حالت اصلاح شده آن به شکل زیر است که این مشکل را حل میکند :

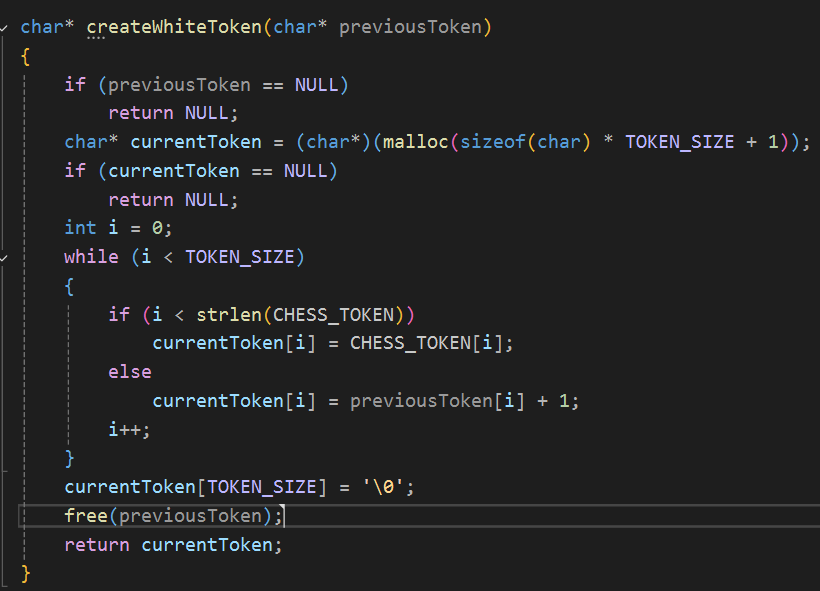




این مشکل هم ریختن مقادیر در newToken در سوئیچ و با چک کردن نتیجه تخصیص حافظه و آزاد کردن آن در پایان ، حل می شود.

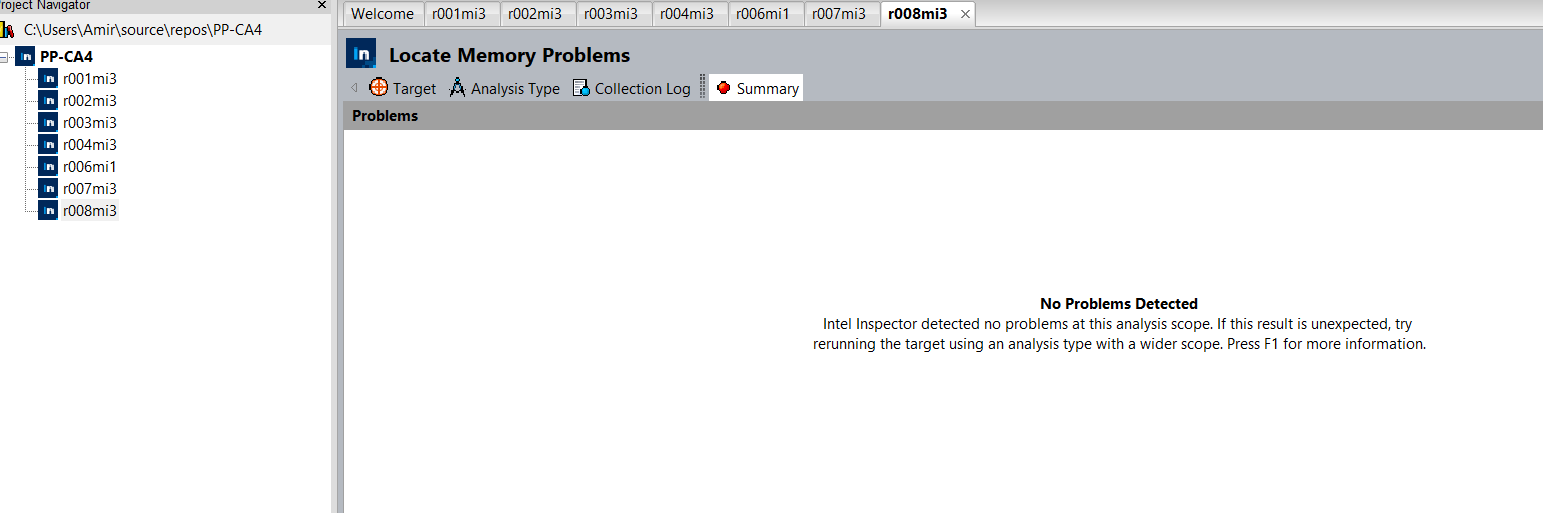


این مشکل با free کردن حافظه previousToken که در بخش های قبل توضیح داده شد حل میشود. این previousToken همان حافظه ای است که در اینجا به عنوان tokenHolder تخصیص داده میشود و بعد ریترن میشود.

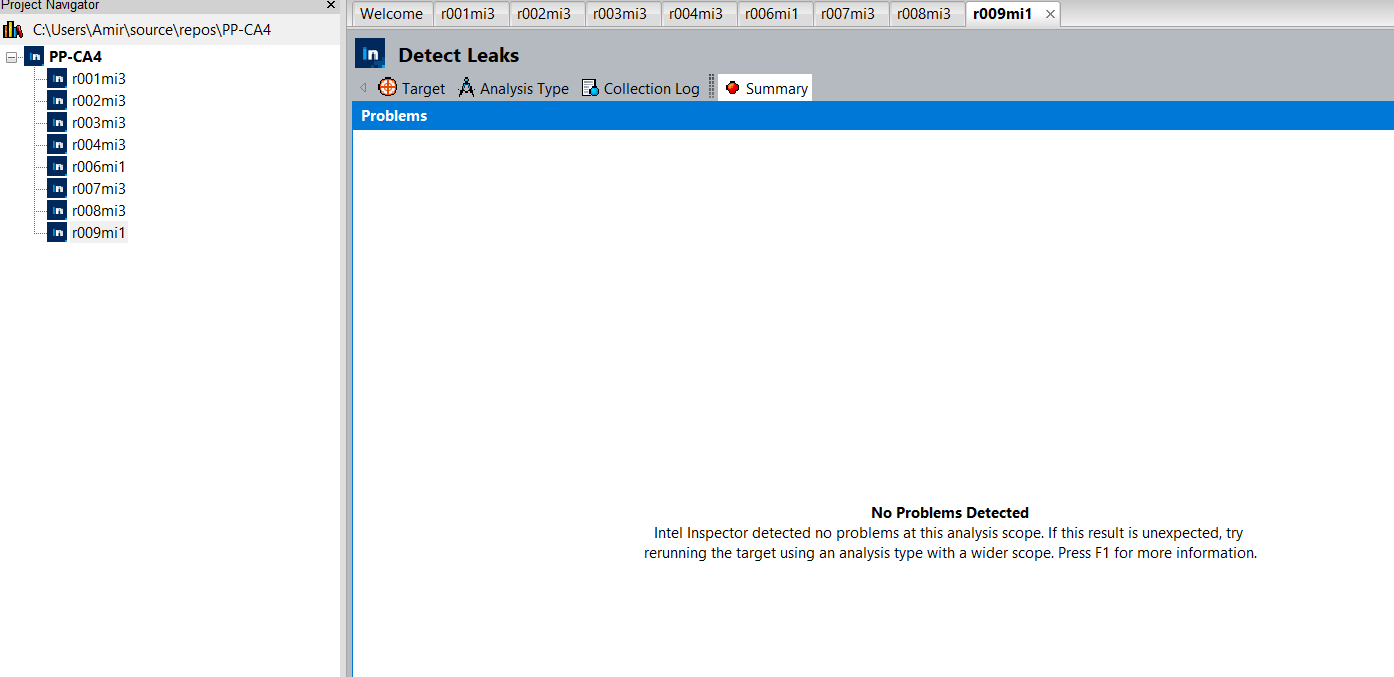


در پایان ، با تحلیل دوباره کد توسط inspector مشاهده میکنیم که این مشکلات حل شده اند :

بخش locate Memory problems:



بخش Detect leaks :



قطعه کد داده شده ، می توانست از نظر از نظر طراحی کمی بهتر عمل کند . برای مثال name که به عنوان argv پاس داده میشود ، فقط برای یک چک کردن ساده استفاده می شود که تصمیم درستی نیست . مقادیر پاس داده شده به توابع به درستی چک نمیشوند که null نباشند و همچنین برای حالتی که تخصیص حافظه با خطا مواجه میشود ، چک نمیشود و برای آن اقدامی صورت نمی گیرد. تابع initFirstMove می توانست کمی بهتر نوشته شود و به دو یا چند تابع شکسته شود. شیوه مدیریت حافظه ها که در تابع main انجام داده است ، می توانست بهتر انجام شود و از پاس داده شدن مکرر پوینتر ها و اینکه حافظه ای که در یک تابع ایجاد شود و در تابع دیگری free شود جلوگیری شود. در نهایت به این قطعه کد از نظر ما نمره 60 تعلق میگیرد.