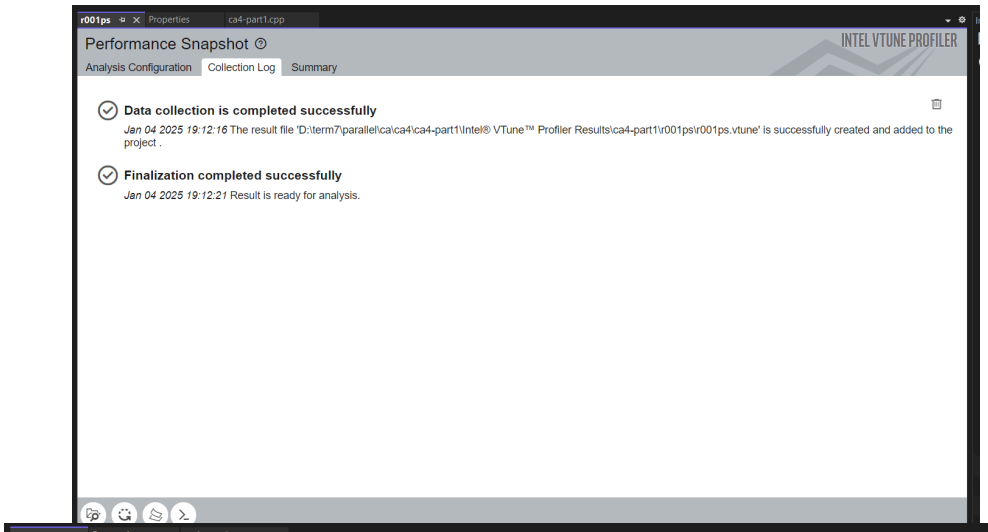
گلبو رشیدی 810100148
پروا شریفی 810100171

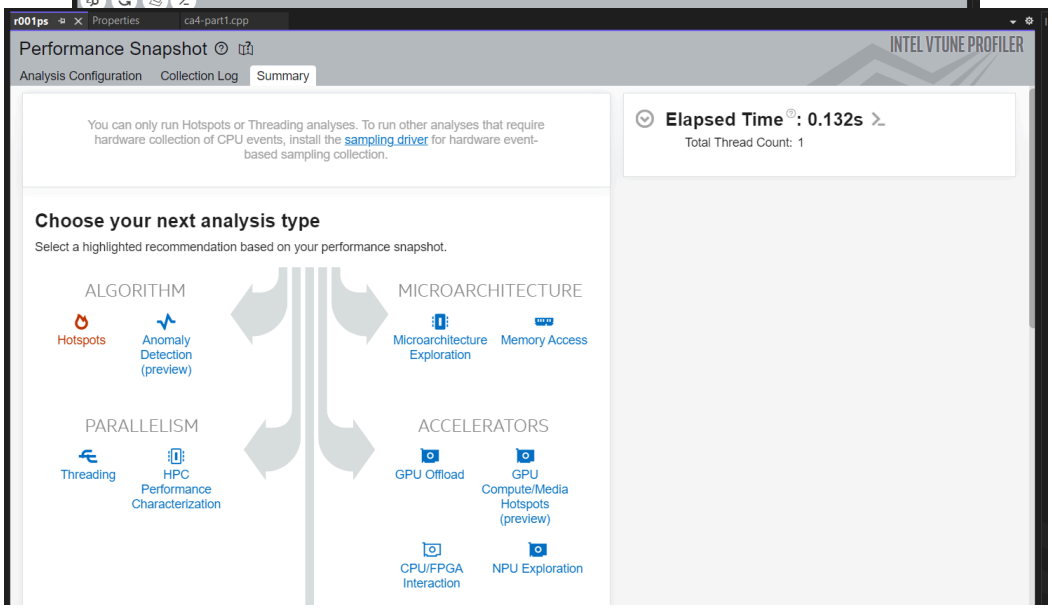
بخش اول

زمان اجرای برنامه با استفاده از کامپایلر اینتل:

```
Execution with:
m:4 ,n:4 ,k:4
Total number of solutions : 412
Execution time: 1299656 microseconds
```

مرحله 1: آنالیز





انتخاب کردن تحلیل hotspot

خروجی های قابل توجه:

Elapsed Time: 103.964s

CPU Time: 32.297s

Total Thread Count: 1

Paused Time: 0s

Top Hotspots

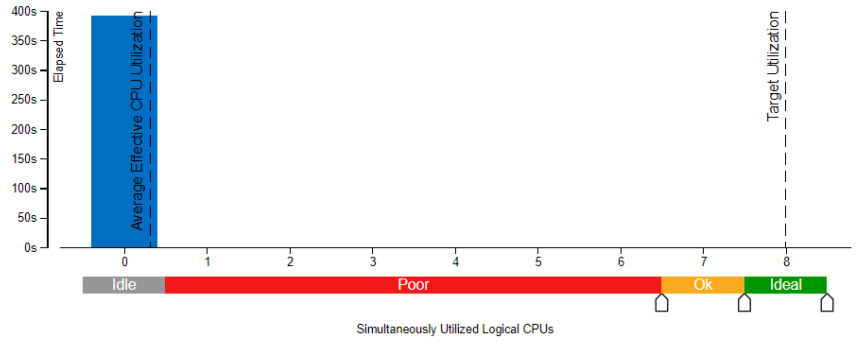
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

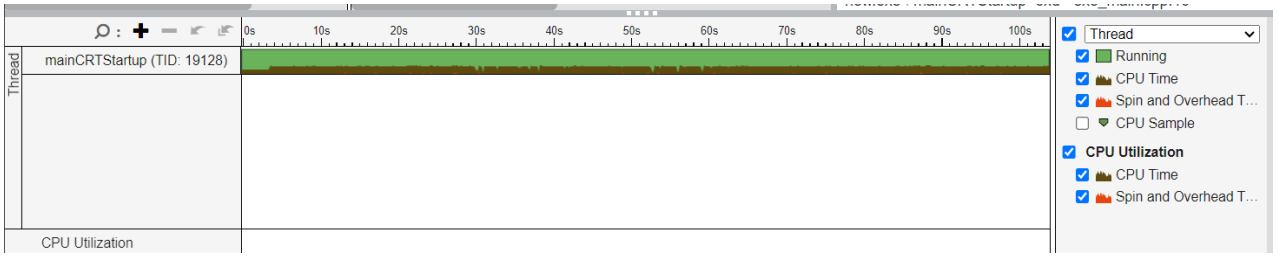
Function	Module	CPU Time	% of CPU Time
std::operator<<<struct std::char_traits<char> >	new.exe	19.822s	61.4%
std::operator<<<struct std::char_traits<char> >	new.exe	9.977s	30.9%
std::basic_ostream<char,struct std::char_traits<char> >::operator<<	MSVCP140D.dll	2.414s	7.5%
displayBoard	new.exe	0.033s	0.1%
malloc	ucrtbased.dll	0.023s	0.1%
[Others]	N/A*	0.027s	0.1%

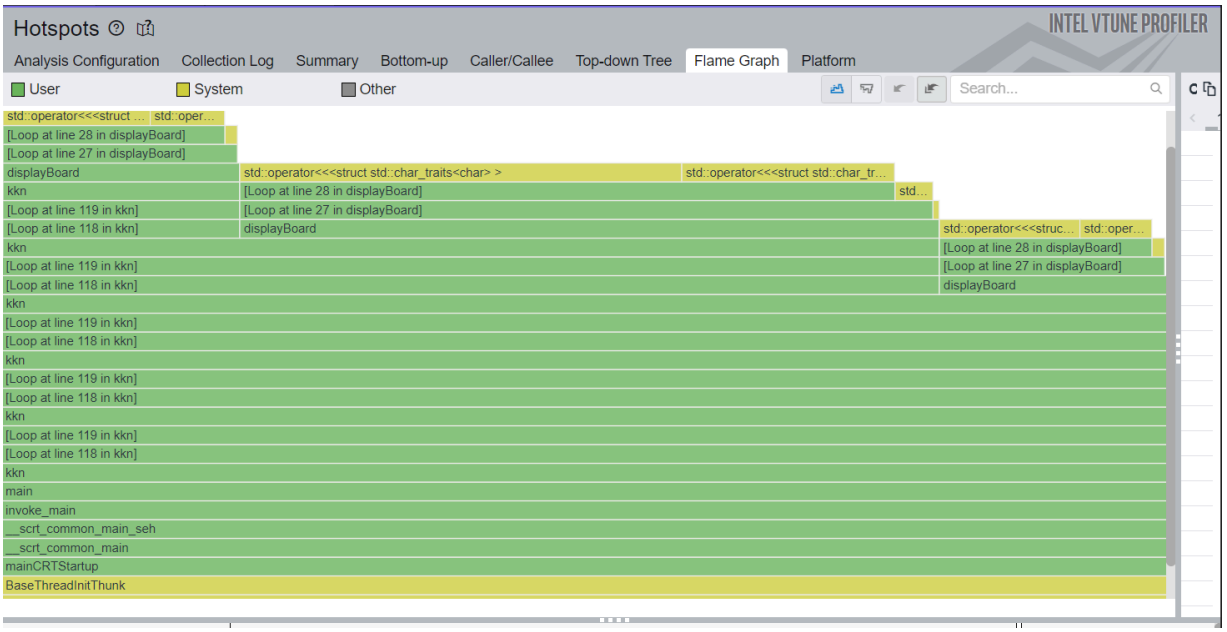
*N/A is applied to non-summable metrics.

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.







آنالیز Hotspot های برنامه:

توضیح جدول:

- function: نام تابع که CPU time صرف اجرای آن شده است.
- CPU time: total: کل زمان صرف شده توسط CPU در تابع مورد نظر و تمام عملکردهای فرزند آن.
- CPU time: self: زمانی که CPU صرف اجرای تابع مورد نظر کرده است، به استثنای زمان صرف شده در توابع فرزند آن.
- Module: ماژولی که تابع مورد نظر در آن قرار دارد.
- Function full: سیگنچر کامل تابع مورد نظر. این توضیحات دقیق تر میتوانند به ما در برخورد با hotspot کمک کنند.
- Source File: محلی که تابع مورد نظر در آن قرار دارد.
- Start Address: آدرس حافظه که در آن تابع شروع می شود. این فیلد معمولا کمتر مورد استفاده است، مگر برای اسمبلی یا بهینه‌سازی‌های سطح پایین.

Hotspot های شناسایی شده:

1.

std::operator<<<struct std::char_traits<char> >	19.822s	new.exe	std::operator<<<struct std::char_traits<char> >(class std::basic_ostr
displayBoard	19.822s	new.exe	displayBoard(char * *)
kkn	19.822s	new.exe	kkn(int,int,int,char * *)
kkn	19.822s	new.exe	kkn(int,int,int,char * *)
kkn	19.822s	new.exe	kkn(int,int,int,char * *)
main	3.584s	new.exe	main
kkn	16.239s	new.exe	kkn(int,int,int,char * *)
kkn	16.239s	new.exe	kkn(int,int,int,char * *)
main	16.239s	new.exe	main

2.

std::operator<<<struct std::char_traits<char> >	9.977s	new.exe	std::operator<<<struct std::char_traits<char> >(class std::basic_ostr
displayBoard	9.977s	new.exe	displayBoard(char * *)
kkn	9.977s	new.exe	kkn(int,int,int,char * *)
kkn	9.977s	new.exe	kkn(int,int,int,char * *)
kkn	9.977s	new.exe	kkn(int,int,int,char * *)
main	1.588s	new.exe	main
kkn	8.390s	new.exe	kkn(int,int,int,char * *)
kkn	8.390s	new.exe	kkn(int,int,int,char * *)
main	8.390s	new.exe	main

.3

std::basic_ostream<char,struct std::char_traits<char> >::operator<<	2.414s	MSVCP140D.dll	std::basic_ostream<char,struct std::char_traits<char> >::operator<<
displayBoard	2.414s	new.exe	displayBoard(char * *)
kkn	2.414s	new.exe	kkn(int,int,int,char * *)
kkn	2.414s	new.exe	kkn(int,int,int,char * *)
kkn	2.414s	new.exe	kkn(int,int,int,char * *)
main	0.400s	new.exe	main
kkn	2.014s	new.exe	kkn(int,int,int,char * *)
kkn	2.014s	new.exe	kkn(int,int,int,char * *)
main	2.014s	new.exe	main
invoke_main	2.014s	new.exe	invoke_main()

.4

displayBoard	0.033s	new.exe	displayBoard(char * *)
kkn	0.033s	new.exe	kkn(int,int,int,char * *)
kkn	0.033s	new.exe	kkn(int,int,int,char * *)
kkn	0.033s	new.exe	kkn(int,int,int,char * *)
main	0.010s	new.exe	main
invoke_main	0.010s	new.exe	invoke_main()
kkn	0.022s	new.exe	kkn(int,int,int,char * *)
kkn	0.022s	new.exe	kkn(int,int,int,char * *)
main	0.022s	new.exe	main
invoke_main	0.022s	new.exe	invoke_main()

.5

malloc	0.023s	ucrtbased.dll	malloc
operator new	0.023s	new.exe	operator new(unsigned __int64)
operator new[]	0.023s	new.exe	operator new[](unsigned __int64)
kkn	0.023s	new.exe	kkn(int,int,int,char * *)
kkn	0.023s	new.exe	kkn(int,int,int,char * *)
kkn	0.023s	new.exe	kkn(int,int,int,char * *)
main	0.015s	new.exe	main
invoke_main	0.015s	new.exe	invoke_main()
kkn	0.008s	new.exe	kkn(int,int,int,char * *)
main	0.008s	new.exe	main
invoke_main	0.008s	new.exe	invoke_main()

همچنین در این لینک جدول بخش top-down tree آنالیز قرار دارد.

مرحله 2: پیاده‌سازی و انجام موازی سازی

با توجه به آنالیز hotspot های برنامه، می بینیم که بخش قابل توجهی از زمان اجرا توسط تابع std::operator<< برای خروجی کنسول در برنامه استفاده شده است. با تحقیق در این زمینه به این نتیجه رسیدیم که با استفاده از buffer output به جای std::cout، میزان overhead کمتری خواهیم داشت و زمان اجرا بهبود خواهد یافت. در واقع، استفاده از std::cout، از نظر ترد ایمن نیست و همچنین استفاده مکرر از آن (در حلقه های برنامه) موجب slow down شدن اجرا می‌شود. پس راه حل ما برای این مشکل این است که خروجی را در حافظه بافر کنیم و فقط پس از تکمیل تمام محاسبات، آن را در کنسول بنویسیم.

```cpp
/* This function displays our board */
void displayBoard(char** board)
{
    std::ostringstream buffer;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            buffer << " " << board[i][j] << " ";
        }
        buffer << "\n";
    }
    buffer << "\n";
    #pragma omp critical
    std::cout << buffer.str();
}
```

پس از این تغییر برنامه را اجرا میکنیم(با استفاده از کامپایلر اینتل) و مشاهده میکنیم که سرعت اجرای برنامه به شدت بهبود یافته است

همچنین از نتایج آنالیز درمیابیم که زمان زیادی صرف حلقه های تابع knn میشود. پس با موازی سازی این بخش سعی میکنیم تا اجرای برنامه را بهبود ببخشیم.

در این تابع دو حلقه for تو در تو داریم پس از pragma omp parallel for استفاده میکنیم.

همچنین اپدیت کردن مقدار solution باید atomic باشد تا data race پیش نیاید.

★ با توجه به تحقیقات انجام شده، دریافتیم که میتوان از pragma omp parallel for collapse(2) استفاده کرد.(به دلیل وجود حلقه تو در تو) ولی پس از استفاده به جواب غلط رسیدیم و سپس دریافتیم با توجه به بزرگ نبودن متغیرهای حلقه و اینکه حلقه داخلی به حلقه خارجی مربوط است،استفاده از collapse نتیجه جالبی نخواهد داشت.

https://stackoverflow.com/questions/28482833/understanding-the-collapse-clause-in-openmp

★ موارد دیگری همچون کد زیر نیز امتحان شد ولی در تسریع اجرا تفاوتی ایجاد نشد.

```cpp
#pragma omp parallel for num_threads(m)
for (int x = 0; x < m; x++) {
    new_board[x] = new char[n];
}
place(i, j, 'K', 'A', board, new_board);
kkn(k - 1, i, j, new_board);
#pragma omp parallel for num_threads(m)
for (int x = 0; x < m; x++) {
    delete[] new_board[x];
}
```

تابع knn در نهایت:

```
void kkn(int k, int sti, int stj, char** board, int depth_threshold = 2) {
    if (k == 0) {
        displayBoard(board);

        #pragma omp atomic
        solutions++;
    }
    else {
        #pragma omp parallel for schedule(dynamic)
        for (int i = sti; i < m; i++) {
            for (int j = stj; j < n; j++) {
                if (canPlace(i, j, board)) {
                    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);
                    kkn(k - 1, i, j, new_board);
                    for (int x = 0; x < m; x++) {
                        delete[] new_board[x];
                    }
                    delete[] new_board;
                }
            }
            stj = 0;
        }
    }
}
```

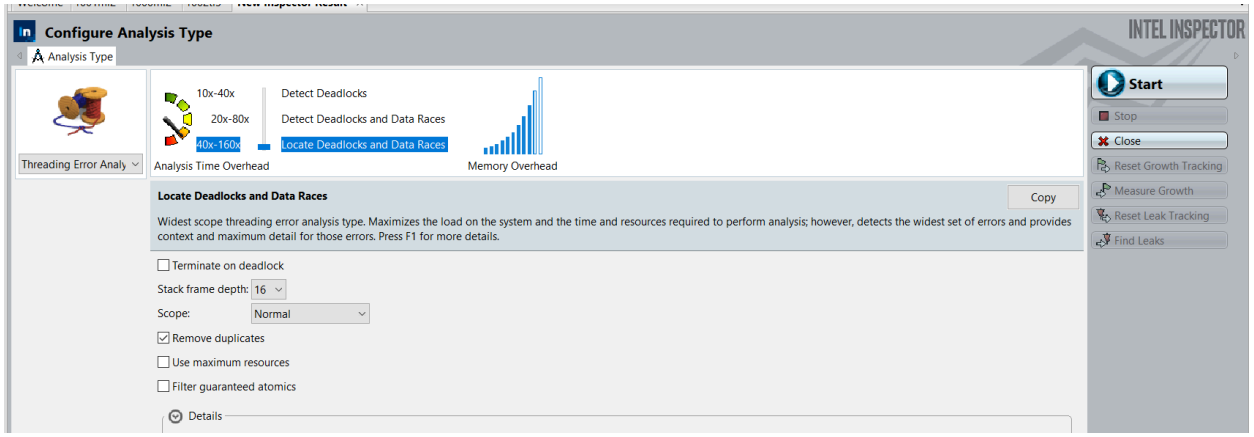
در تابع displayBoard نیز از #pragma omp parallel for استفاده میکنیم.

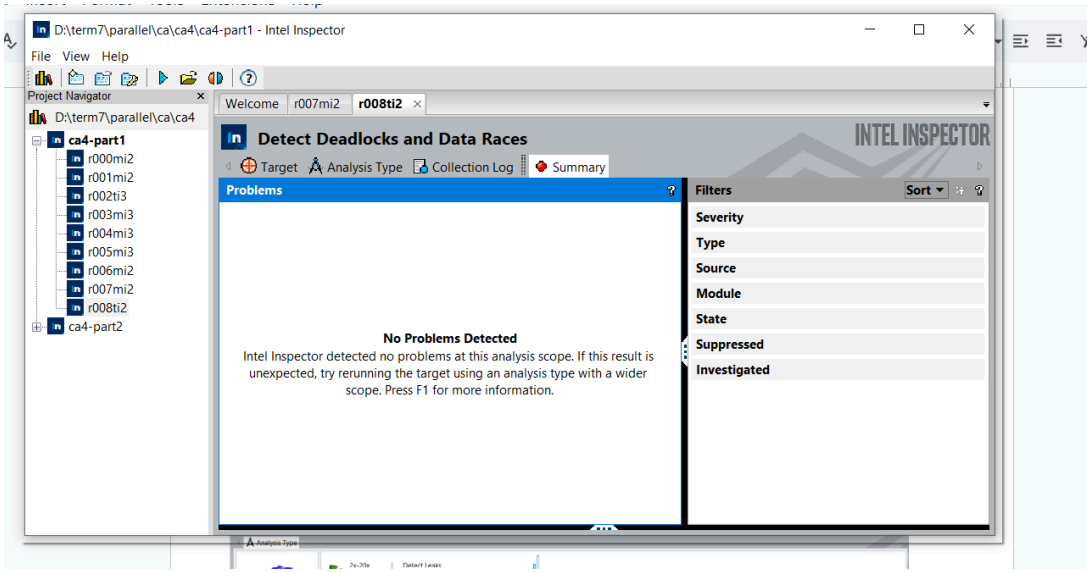
```
void displayBoard(char** board)
{
    std::ostringstream buffer;
    #pragma omp parallel for
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            buffer << " " << board[i][j] << " ";
        }
        buffer << "\n";
    }
    buffer << "\n";
    #pragma omp critical
    std::cout << buffer.str();
}
```

نتیجه ران کردن:

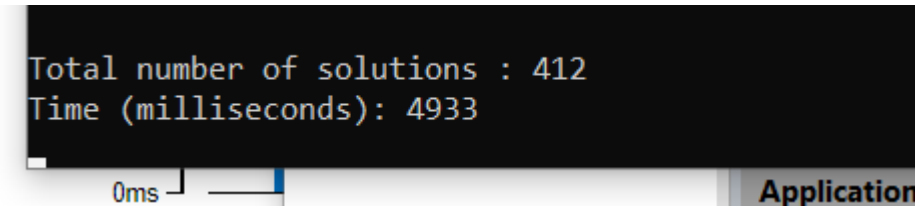
مرحله 3: دیباگ باگ های احتمالی

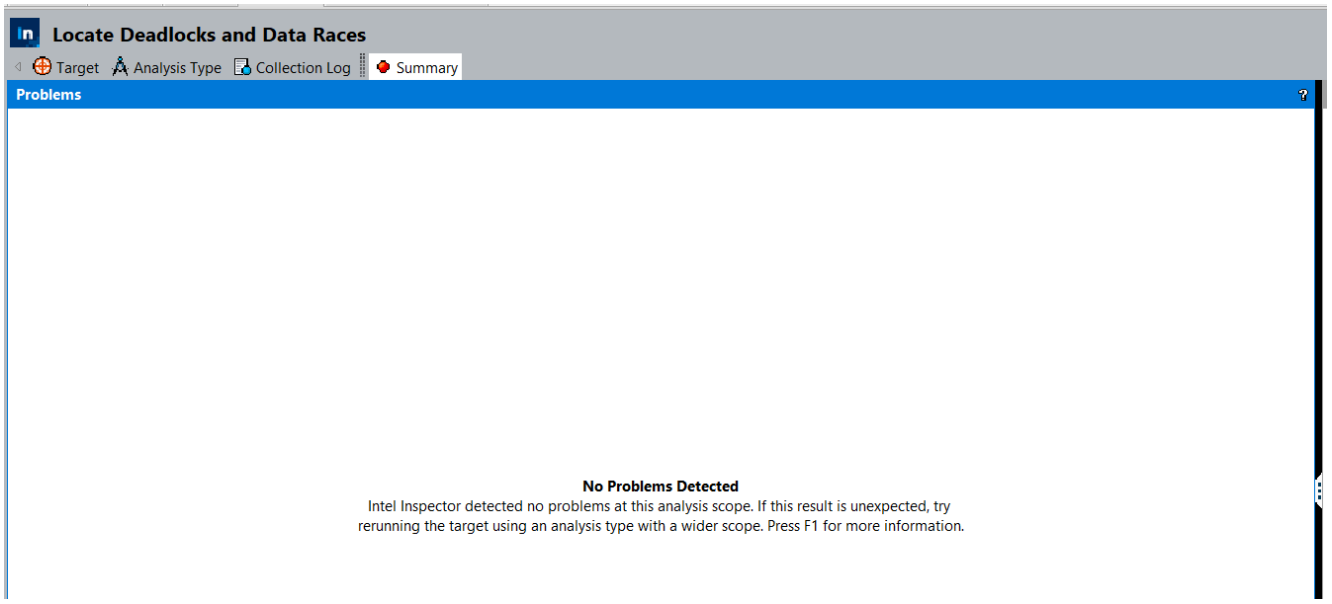
استفاده از intel inspector:

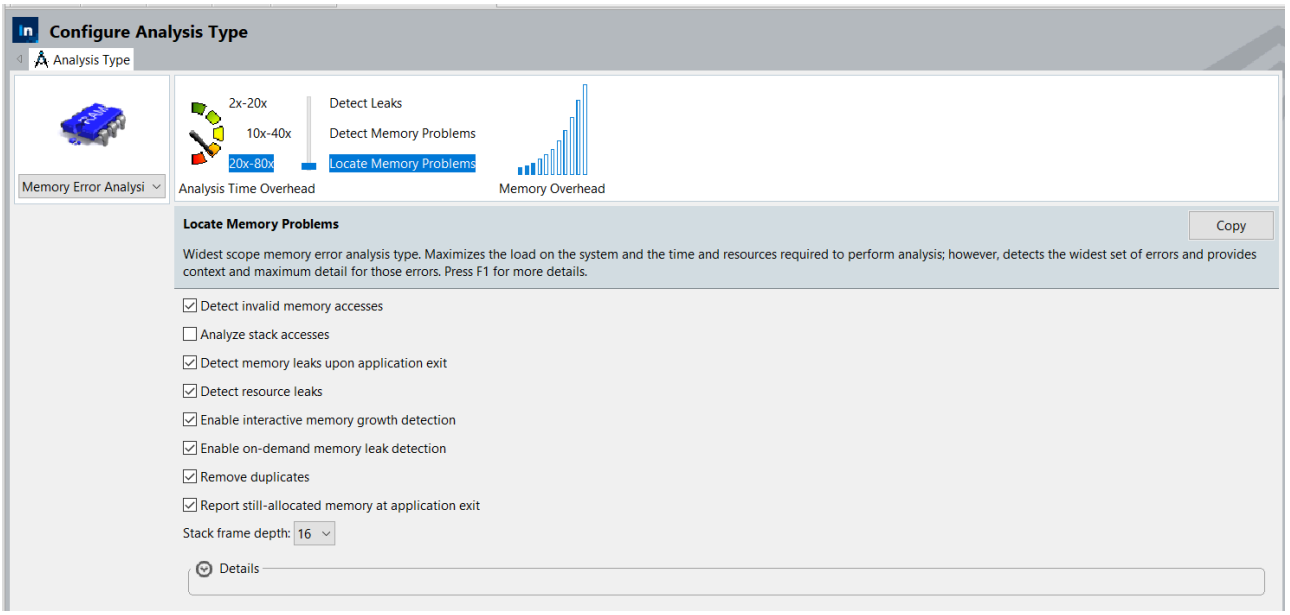

Configure Analysis Type
INTEL INSPECTOR
Analysis Type
Threading Error Analy
10x-40x
20x-80x
40x-160x
Detect Deadlocks
Detect Deadlocks and Data Races
Locate Deadlocks and Data Races
Analysis Time Overhead
Memory Overhead
Start
Stop
Close
Reset Growth Tracking
Measure Growth
Reset Leak Tracking
Find Leaks
Locate Deadlocks and Data Races
Copy
Widest scope threading error analysis type. Maximizes the load on the system and the time and resources required to perform analysis; however, detects the widest set of errors and provides context and maximum detail for those errors. Press F1 for more details.
Terminate on deadlock
Stack frame depth: 16
Scope: Normal
Remove duplicates
Use maximum resources
Filter guaranteed atomics
Details


D:\term7\parallel\ca\ca4\ca4-part1 - Intel Inspector
File View Help
Project Navigator
D:\term7\parallel\ca\ca4
ca4-part1
r000mi2
r001mi2
r002ti3
r003mi3
r004mi3
r005mi3
r006mi2
r007mi2
r008ti2
ca4-part2
Welcome
r007mi2
r008ti2
Detect Deadlocks and Data Races
INTEL INSPECTOR
Target
Analysis Type
Collection Log
Summary
Problems
Filters
Sort
Severity
Type
Source
Module
State
Suppressed
Investigated
No Problems Detected
Intel Inspector detected no problems at this analysis scope. If this result is unexpected, try rerunning the target using an analysis type with a wider scope. Press F1 for more information.

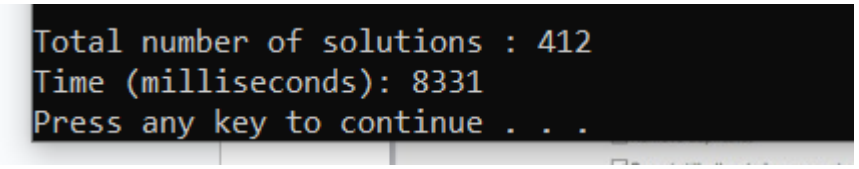
پس از اجرا:

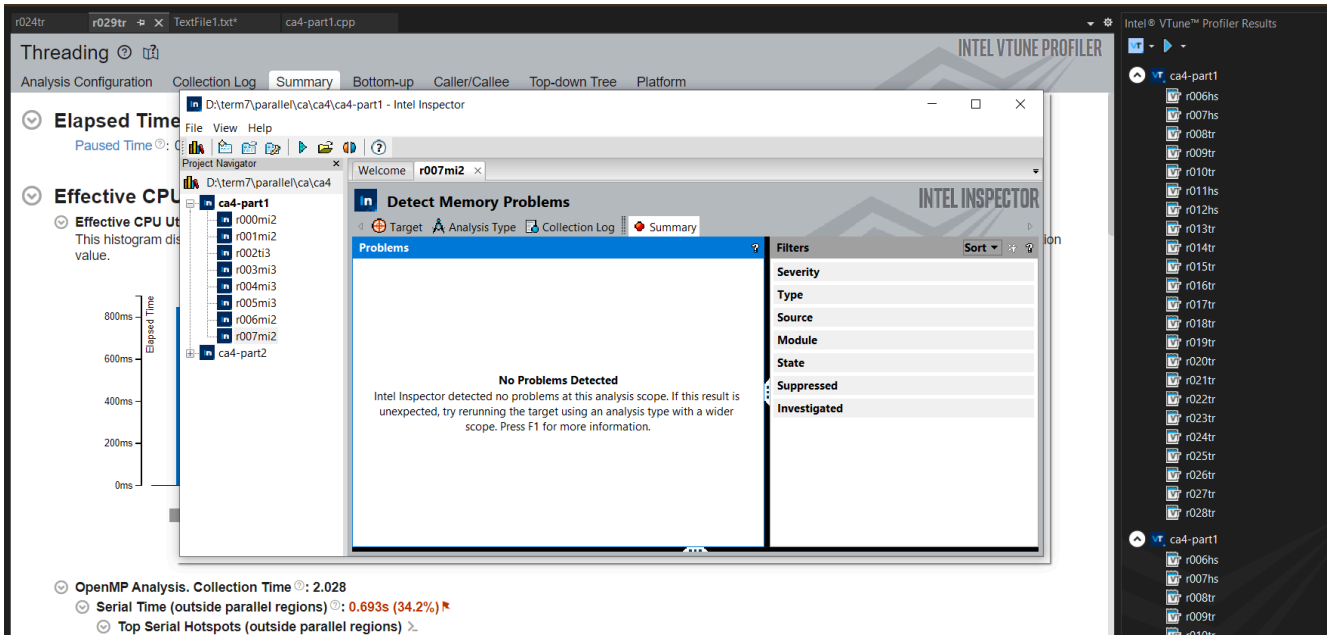

Total number of solutions : 412
Time (milliseconds): 4933
0ms
Application


Locate Deadlocks and Data Races
Target
Analysis Type
Collection Log
Summary
Problems
No Problems Detected
Intel Inspector detected no problems at this analysis scope. If this result is unexpected, try rerunning the target using an analysis type with a wider scope. Press F1 for more information.

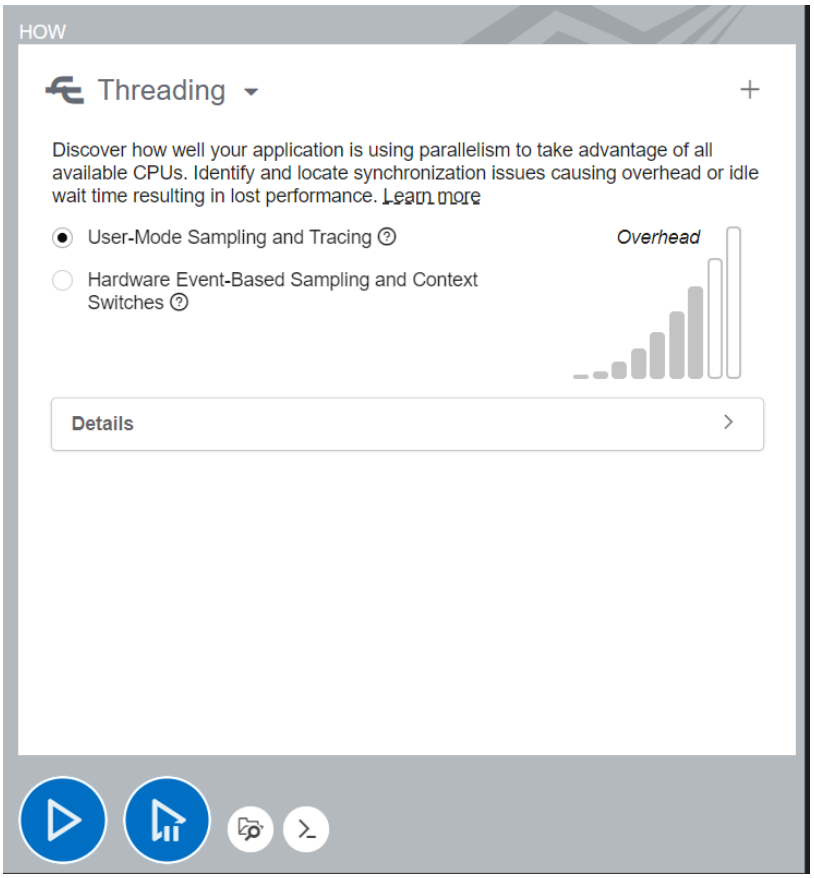


پس از اجرا:





مرحله 4: تنظیم برنامه موازی



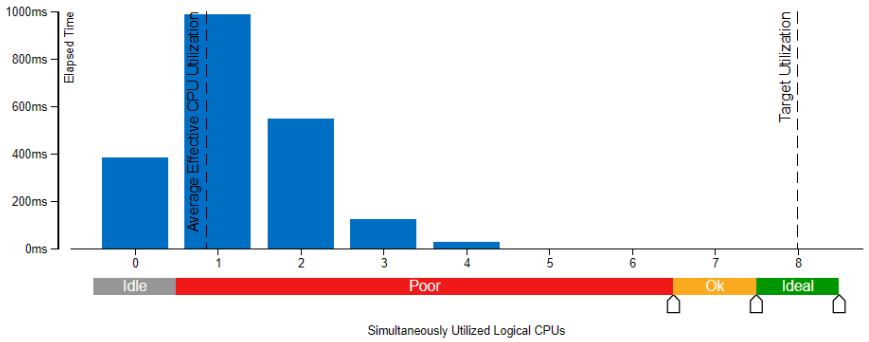
Elapsed Time: 2.073s

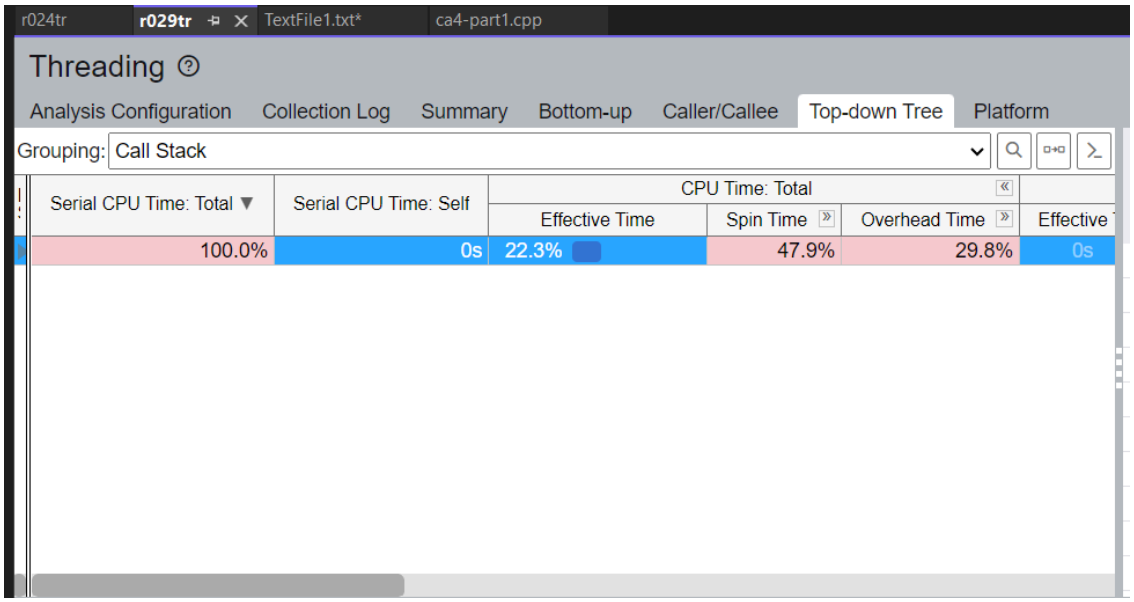
Paused Time: 0s

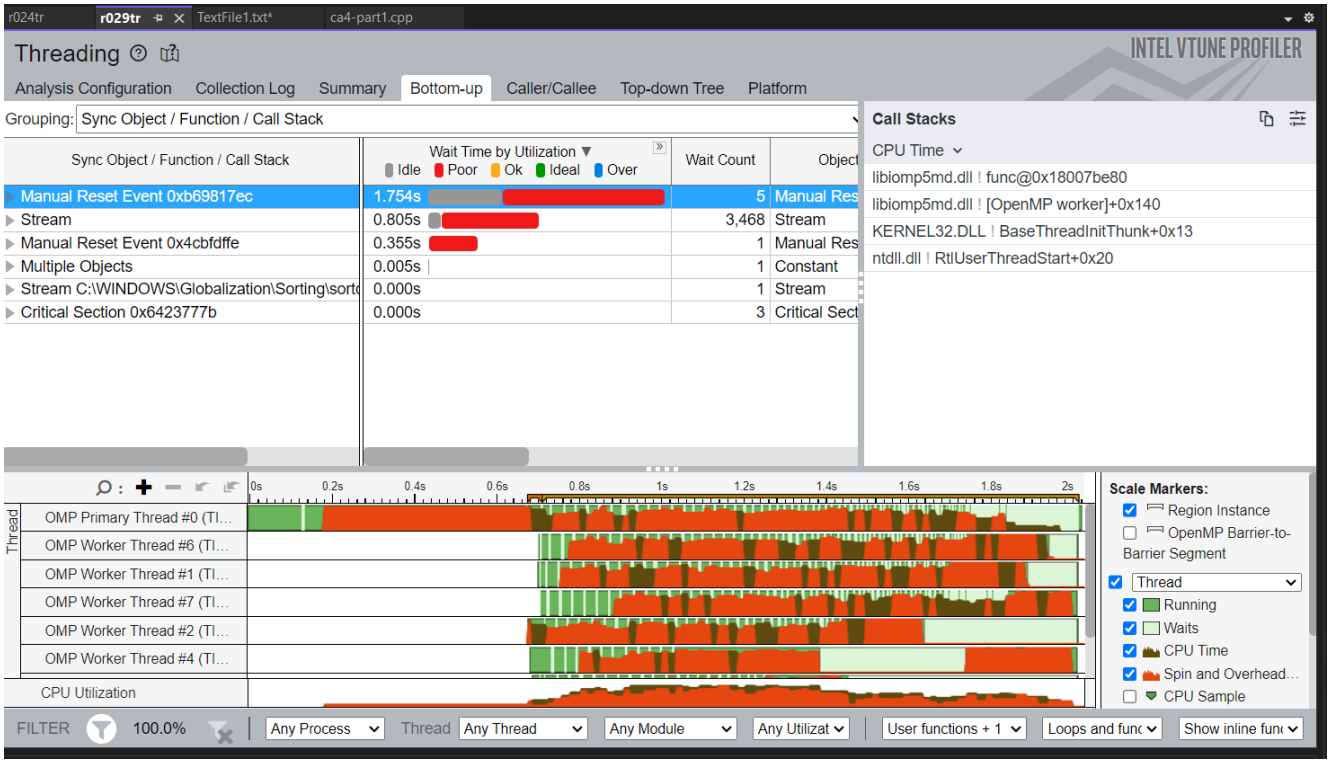
Effective CPU Utilization: 10.8% (0.863 out of 8 logical CPUs)

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.



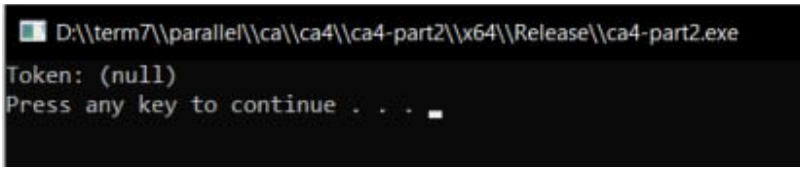

r024tr
r029tr
TextFile1.txt*
ca4-part1.cpp
Threading
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
Grouping: Call Stack
Serial CPU Time: Total
Serial CPU Time: Self
CPU Time: Total
Effective Time
Spin Time
Overhead Time
Effective
100.0%
0s
22.3%
47.9%
29.8%
0s


r024tr
r029tr
TextFile1.txt*
ca4-part1.cpp
Threading
INTEL VTUNE PROFILER
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
Grouping: Sync Object / Function / Call Stack
Sync Object / Function / Call Stack
Wait Time by Utilization
Idle Poor Ok Ideal Over
Wait Count
Manual Reset Event 0xb69817ec
1.754s
5
Stream
0.805s
3,468
Manual Reset Event 0x4cbfdffe
0.355s
1
Multiple Objects
0.005s
1
Constant
Stream C:\WINDOWS\Globalization\Sorting\sort
0.000s
1
Stream
Critical Section 0x6423777b
0.000s
3
Critical Sect
Call Stacks
CPU Time
libiomp5md.dll ! func@0x18007be80
libiomp5md.dll ! [OpenMP worker]+0x140
KERNEL32.DLL ! BaseThreadInitThunk+0x13
ntdll.dll ! RtlUserThreadStart+0x20
0s
0.2s
0.4s
0.6s
0.8s
1s
1.2s
1.4s
1.6s
1.8s
2s
Thread
OMP Primary Thread #0 (TI...
OMP Worker Thread #6 (TI...
OMP Worker Thread #1 (TI...
OMP Worker Thread #7 (TI...
OMP Worker Thread #2 (TI...
OMP Worker Thread #4 (TI...
CPU Utilization
Scale Markers:
Region Instance
OpenMP Barrier-to-Barrier Segment
Thread
Running
Waits
CPU Time
Spin and Overhead...
CPU Sample
FILTER
100.0%
Any Process
Thread
Any Thread
Any Module
Any Utilizat
User functions + 1
Loops and func
Show inline func

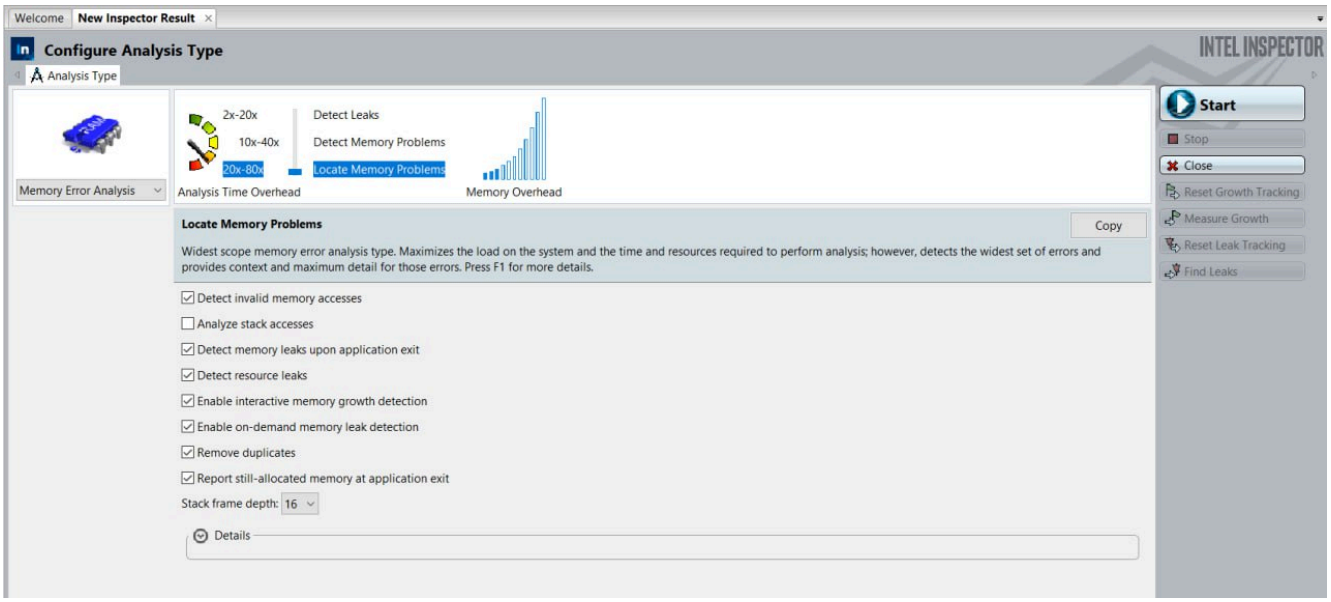
بخش دوم

مرحله 1: آنالیز

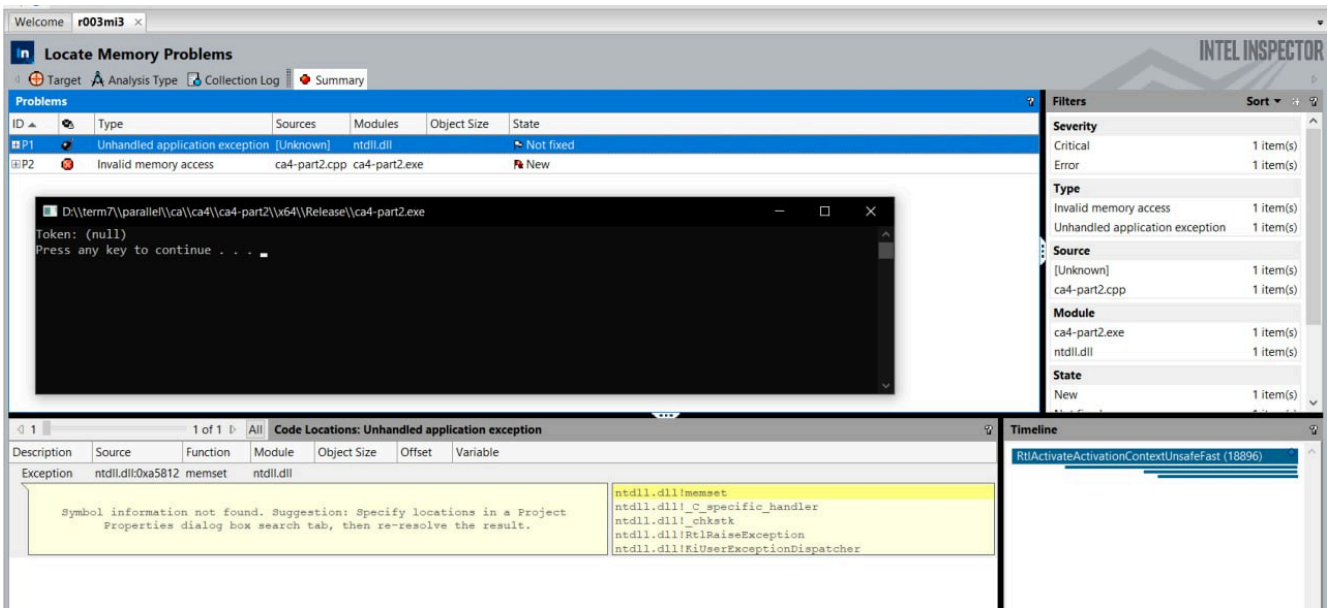
خروجی کد داده شده در فایل پروژه:



آنالیز مموری در intel parallel studio:



مشکلات مموری:



مرحله 2: پیاده‌سازی و رفع مشکلات

دسترسی به حافظه نامعتبر (Invalid Memory Access)

در نسخه اصلی کد، زمانی که `previousToken` برابر با `NULL` بود، تابع `createWhiteToken` سعی می‌کرد به این مقدار اشاره کند و به `previousToken` دسترسی پیدا کند، که باعث خطای دسترسی نامعتبر به حافظه می‌شد.
در نسخه جدید، ابتدا بررسی می‌شود که آیا `previousToken` برابر با `NULL` است یا خیر. در صورت `NULL` بودن، حافظه اختصاص داده‌شده آزاد می‌شود و `NULL` بازگردانده می‌شود

نشت حافظه (Memory Leaks)

در تابع `initFirstMove`، زمانی که حافظه جدیدی به `whiteToken` اختصاص داده می‌شد، حافظه قبلی بدون آزادسازی بازنویسی می‌شد، که باعث نشت حافظه می‌شد.
در نسخه جدید، این مشکل با آزادسازی حافظه‌های قبلی (در صورت نیاز) و مدیریت صحیح تخصیص حافظه برطرف شده است:

```
free(WhiteToken);
```

استفاده از `srand` در حلقه

در نسخه اولیه کد، تابع `srand` برای تولید اعداد تصادفی در هر بار اجرای حلقه فراخوانی می‌شد. این باعث می‌شد مقدار تصادفی به درستی تولید نشود.

در نسخه اصلاح‌شده، `srand` تنها یک بار در ابتدای برنامه فراخوانی می‌شود:

```
srand((unsigned int)time(0));
```

بهینه‌سازی تخصیص و آزادسازی حافظه

تخصیص و آزادسازی حافظه در نسخه اصلی کد بهینه نشده بود و احتمال دسترسی به حافظه آزادشده (Use-After-Free) وجود داشت.

در نسخه جدید، تخصیص و آزادسازی حافظه با دقت بیشتری انجام شده و تمامی حافظه‌های تخصیص داده‌شده در پایان برنامه آزاد می‌شوند:

```
free(blackToken);

free(whiteToken);

free(firstMove);
```

عدم مقداردهی پایان رشته (Null-Termination)

در نسخه اصلی کد، توکن‌ها به صورت صحیح با کاراکتر `0\` پایان‌دهی نمی‌شدند. این موضوع باعث رفتار غیرمنتظره هنگام چاپ رشته‌ها یا دسترسی به آن‌ها می‌شد.

در نسخه جدید، اطمینان حاصل شده است که تمام توکن‌ها (مانند `currentToken` و `whiteToken`) به صورت صحیح پایان‌دهی شوند:

```
currentToken[TOKEN_SIZE - 1] = '\0';

whiteToken[MOVE_SIZE - 1] = '\0';
```

تغییرات دیگر انجام شده شامل:

مدیریت صحیح ورودی‌های کاربر: ورودی کاربر بررسی شده و در صورت نامعتبر بودن، مقدار پیش‌فرض (Pawn) انتخاب می‌شود

نمایش صحیح توکن‌ها: به جای `s%` از یک حلقه برای چاپ کاراکترهای توکن استفاده شده است

مرحله 3:اجرا

پس از انجام تغییرات مورد نیاز در کد و مراجعه به intel parallel studio خواهیم دید که تمام خطاها برطرف شده است.