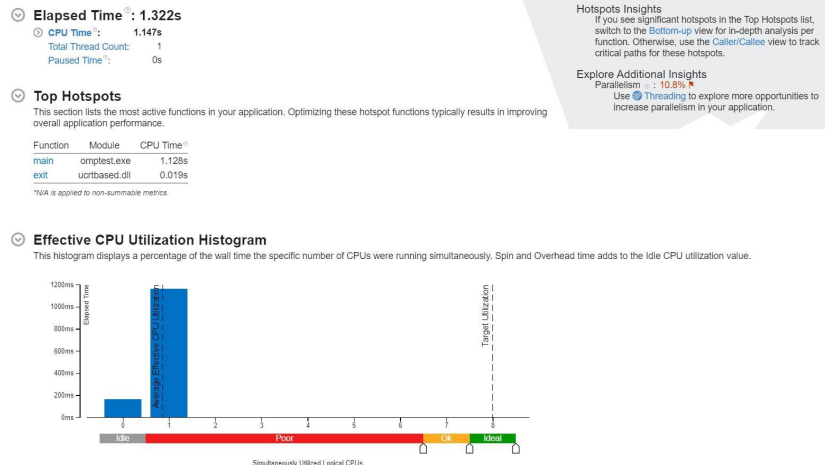
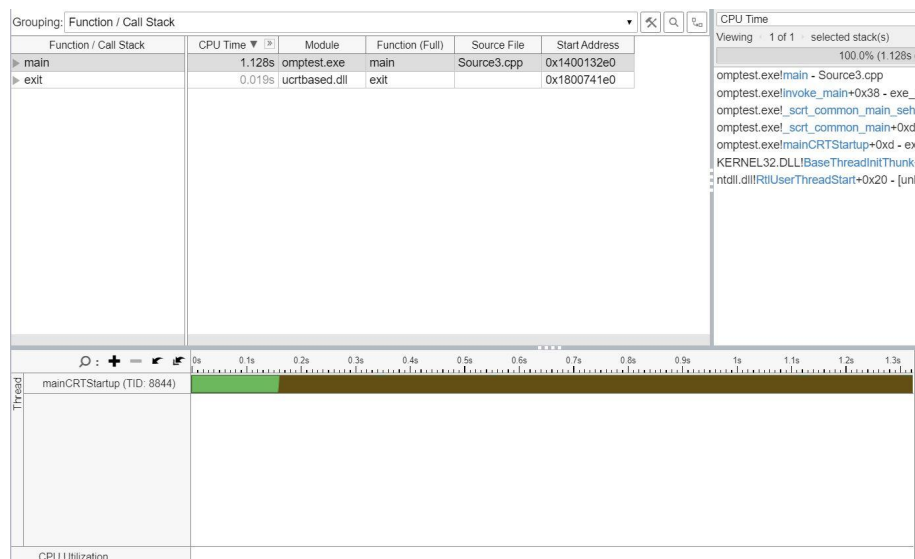


اجرای حالت سریال:

ابتدا کد سریال را اجرا کردیم.



همانطور که مشاهده می کنید، برنامه از یک هسته استفاده کرده و آن را به خوبی استفاده نمی کند.



برنامه به غیر از بخش اول در بقیه اوقات به صورت غیر مناسبی از یک هسته استفاده می کند.

Source	Assembly				
S. ▲		Source	🔥 CPU Time: Total	CPU Time: Self	
21		// repeat experiment several times			
22		for (i = 0; i < 6; i++)			
23		{			
24		// get starting time			
25		starttime = timeGetTime();			
26		// reset check sum & running total			
27		sum = 0;			
28		total = 0.0;			
29		// Work Loop, do some work by looping VERYBIG times			
30		for (j = 0; j < VERYBIG; j++)			
31		{			
32		// increment check sum			
33		sum += 1;			
34		// Calculate first arithmetic series			
35		sumx = 0.0;			
36		for (k = 0; k < j; k++)			
37		sumx = sumx + (double)k;	4.1%	48.628ms	
38		// Calculate second arithmetic series	53.2%	610.368ms	
39		sumy = 0.0;			
40		for (k = j; k > 0; k--)			
41		sumy = sumy + (double)k;	5.4%	62.502ms	
42		if (sumx > 0.0) total = total + 1.0 / sqrt(sumx);	35.6%	408.240ms	
43		if (sumy > 0.0) total = total + 1.0 / sqrt(sumy);			
44		}			
45		// get ending time and use it to determine elapsed time			
46		elapsedTime = timeGetTime() - starttime;			
47		// report elapsed time			
48		printf("Time Elapsed %10d mSecs Total = %lf Check Sum = %ld\n",			
49		(int)elapsedTime, total, sum);			
50		}			
51		// return integer as required by function header			

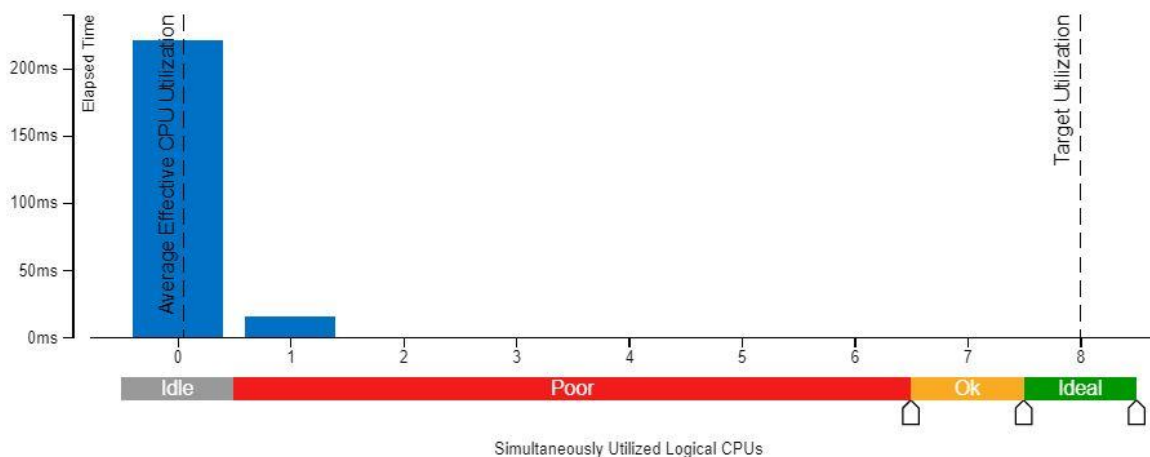
همانطور که در آزمایش‌های قبلی اشاره شد، اکثر زمان اجرای برنامه در دو خط ۳۷ و ۴۱ می‌گذرد.

اجرای موازی سازی با مشکل:

پس از قرار دادن pragma در بالای حلقه، حلقه موازی سازی شده ولی تعداد زیادی race condition بوجود خواهد آمد.

Effective CPU Utilization Histogram

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



همانطور که مشاهده می‌کنید، تقریباً اصلاً استفاده مناسبی از CPUها نشده است.

تشخیص مشکلات کد موازی سازی شده:

پس از اجرای کد موازی سازی شده و تحلیل آن توسط inspector مشاهده می کنیم:

The screenshot shows the Intel Inspector interface with the 'Locate Deadlocks and Data Races' window. The 'Problems' list on the left contains seven entries (P1-P7) all identified as 'Data race' with a severity of 'Error'. The 'Filters' pane on the right shows counts for Severity (Error: 7), Type (Data race: 7), Source (Source31.cpp: 6), Module (ompctest.exe: 6, vcomp140d.dll: 4), State (New: 7), and Suppressed (Not suppressed: 7). The main pane displays 'Code Locations: Data race' with a table of read and write operations. The 'Timeline' pane on the right shows a sequence of memory accesses, including 'vcomp_atomic_div_r8 (10312)' and 'vcomp_atomic_div_r8 (14616)'.

ID	Type	Sources	Modules	State
P1	Data race	[Unknown]; Source31.cpp	ompctest.exe; vcomp140d.dll	New
P2	Data race	[Unknown]; Source31.cpp	ompctest.exe; vcomp140d.dll	New
P3	Data race	[Unknown]; Source31.cpp	ompctest.exe; vcomp140d.dll	New
P4	Data race	Source31.cpp	ompctest.exe	New
P5	Data race	Source31.cpp	ompctest.exe	New
P6	Data race	Source31.cpp	ompctest.exe	New
P7	Data race	[Unknown]	vcomp140d.dll	New

Description	Source	Function	Module	Variable
Read	Source31.cpp:36	main\$omp\$1	ompctest.exe	sum
Write	Source31.cpp:36	main\$omp\$1	ompctest.exe	sum

لیست مشکلات و مکان‌هایی که race condition پیش آمده است در عکس مشخص است.

مشکلاتی که در کد وجود داشت عبارت‌اند از:

This screenshot provides a detailed view of the source code for the detected data race. The 'Data race' window is active, showing two threads: 'Read - Thread vcomp_atomic_div_r8 (14616)' and 'Write - Thread vcomp_atomic_div_r8 (10312)'. Both threads are executing code from 'Source31.cpp'. The code includes a parallel loop using OpenMP pragmas. The 'Call Stack' pane on the right shows the execution path, starting from 'ompctest.exe!main\$omp\$1' and going through 'vcomp140d.dll' functions like 'vcomp_atomic_div_r8' and 'vcomp_for'.

```
// reset check sum & running total
sum = 0;
total = 0.0;
// Work Loop, do some work by looping VERVBIG times
#pragma omp parallel for
for (j = 0; j < VERVBIG; j++)
{
    // increment check sum
    sum += 1;
    // Calculate first arithmetic series
    sumx = 0.0;
    for (k = 0; k < j; k++)
        sumx = sumx + (double)k;
    // Calculate second arithmetic series
    sumy = 0.0;
    for (k = j; k > 0; k--)
        sumy = sumy + (double)k;
}
```

The screenshot shows Intel Inspector's 'Data race' analysis. It identifies a race between a 'Read - Thread vcomp_atomic_div_r8 (14616)' and a 'Write - Thread vcomp_atomic_div_r8 (10312)'. Both threads are from 'omptest.exe\main\$omp\$1 - Source31.cpp:39'. The code snippets show nested loops over 'j' and 'k', with 'sumx' and 'sumy' being updated. The 'Read' thread is at line 39, and the 'Write' thread is at line 39. The call stack for both threads shows the same sequence of calls: 'omptest.exe\main\$omp\$1 - Source31.cpp:39', 'vcomp140d.dll\vcomp_atomic_div_r8', 'vcomp140d.dll\vcomp_fork', and 'vcomp140d.dll\vcomp_atomic_div_r8'.

The screenshot shows Intel Inspector's 'Data race' analysis. It identifies a race between two 'Write - Thread vcomp_atomic_div_r8 (14616)' threads. Both threads are from 'omptest.exe\main\$omp\$1 - Source31.cpp:40'. The code snippets show nested loops over 'j' and 'k', with 'sumx' and 'sumy' being updated. The first 'Write' thread is at line 40, and the second 'Write' thread is at line 38. The call stack for both threads shows the same sequence of calls: 'omptest.exe\main\$omp\$1 - Source31.cpp:40', 'vcomp140d.dll\vcomp_atomic_div_r8', 'vcomp140d.dll\vcomp_fork', and 'vcomp140d.dll\vcomp_atomic_div_r8'.

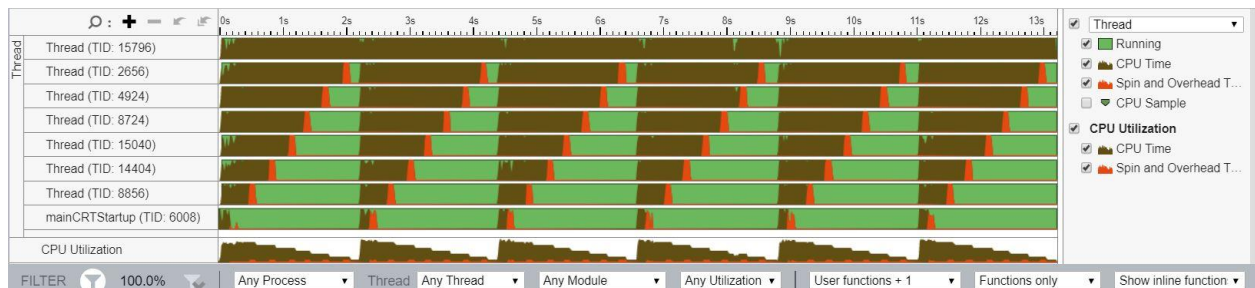
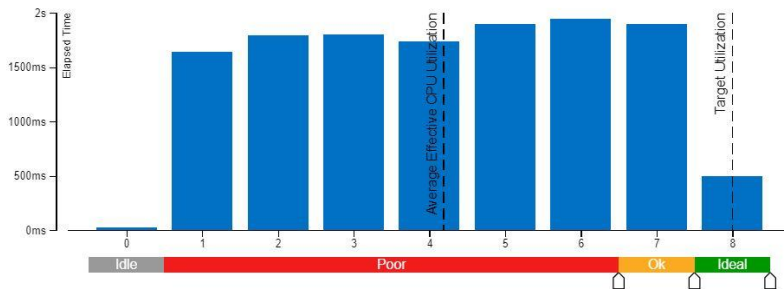
همانطور مشکلی که بوجود آمده با private کردن sumx, sumy و k درست می شود، همچنین برای درست شدن checksum باید مقدار sum را به صورت جمع و total را داخل reduction قرار داد.

اجرای کد موازی سازی درست شده:

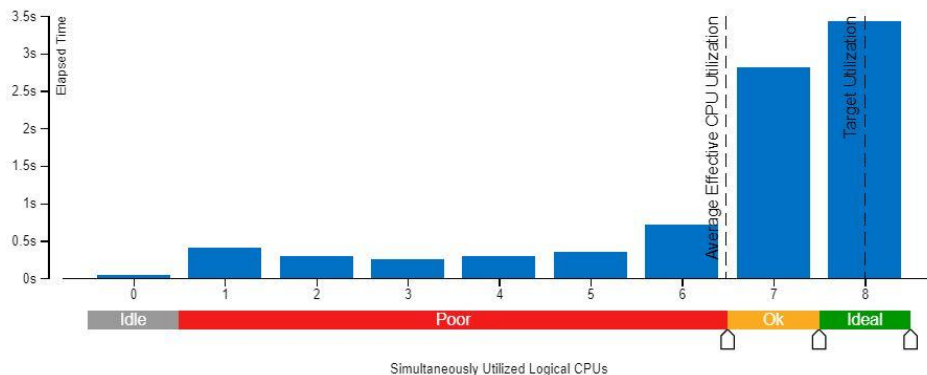
پس از درست کردن کد، آن را اجرا می کنیم.

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.

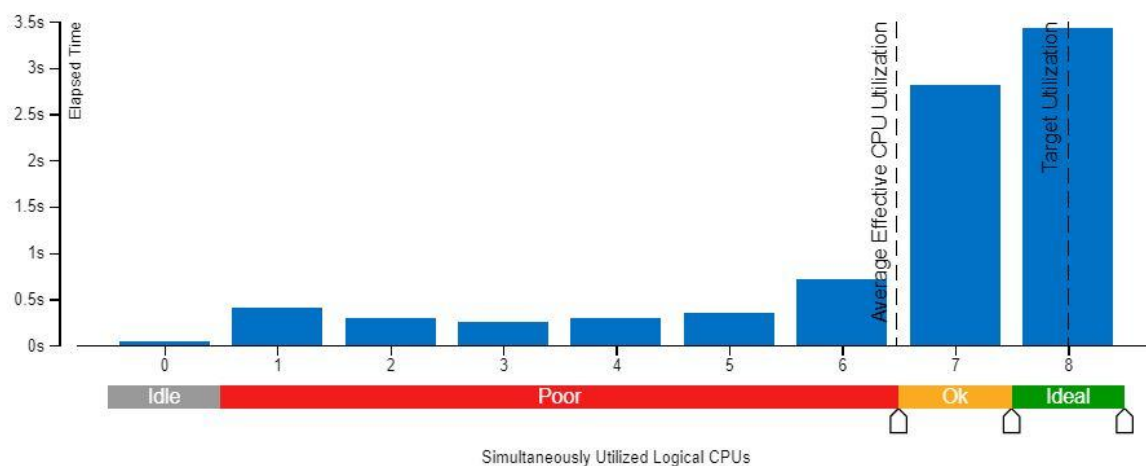


همانطور که می بینید از همه ی CPU ها استفاده شده اما نه به خوبی، دلیل این اتفاق این است که OpenMP به صورت مساوی تقسیم می کند بازه ی حلقه را اما بازه های با طول مساوی در برنامه، حجم محاسبات مساوی نمی خواهند و این حجم صعودی می باشد، به همین دلیل اگر با استفاده از schedule، بازه ها را به تعداد بیشتر با طول کمتر تقسیم کنیم، تعادل بین هسته ها بهتر رعایت می شود. اجرای کد با schedule dynamic با بازه های ۲۰۰۰ تایی:



امیرحسین پاشایی هیر ۹۷۳۱۰۱۳ - محمد مهدی منتظر ۹۷۳۱۱۲۰

اجرای کد با schedule dynamic با بازه‌های ۱۰۰۰ تایی:



همانطور که می‌بیند، استفاده از CPUها بهتر شد، همچنین بازه‌های ۱۰۰۰ تایی بهتر از ۲۰۰۰ تایی

می‌باشد زیرا تعداد بازه‌ها بیشتر شده و تعادل بین هسته‌ها بهتر رعایت می‌شود.