Nachiappan C14148805 20th Sept. 2018

PARALLEL PROGRAMMING ASSIGNMENT

1. BLAS SAXPY:
Basic Linear Algebra Subproblems, like SAXPY, can utilize low level routines to run faster. It can also take advantage of vector registers/ SIMD instructions to execute more quickly.

ISPC uses tasks to parallelize across cores, and SPMD/SIMD vector lanes to optimize operations across a single core.

SAXPY has two operations per vector set of x and y- a multiplication and addition. The given SAXPY program has predictable computation requirement, data access and does not have dependencies besides the fact that the multiplication must take place before the addition.

When, the given SAXPY program is run on a single core CPU vs on an 8 core CPU, the speedup is approximately 4 times for both the tasks and non-task implementations respectively. Hence, ISPC's ability to parallelize across cores is effective in producing a good speed-up.

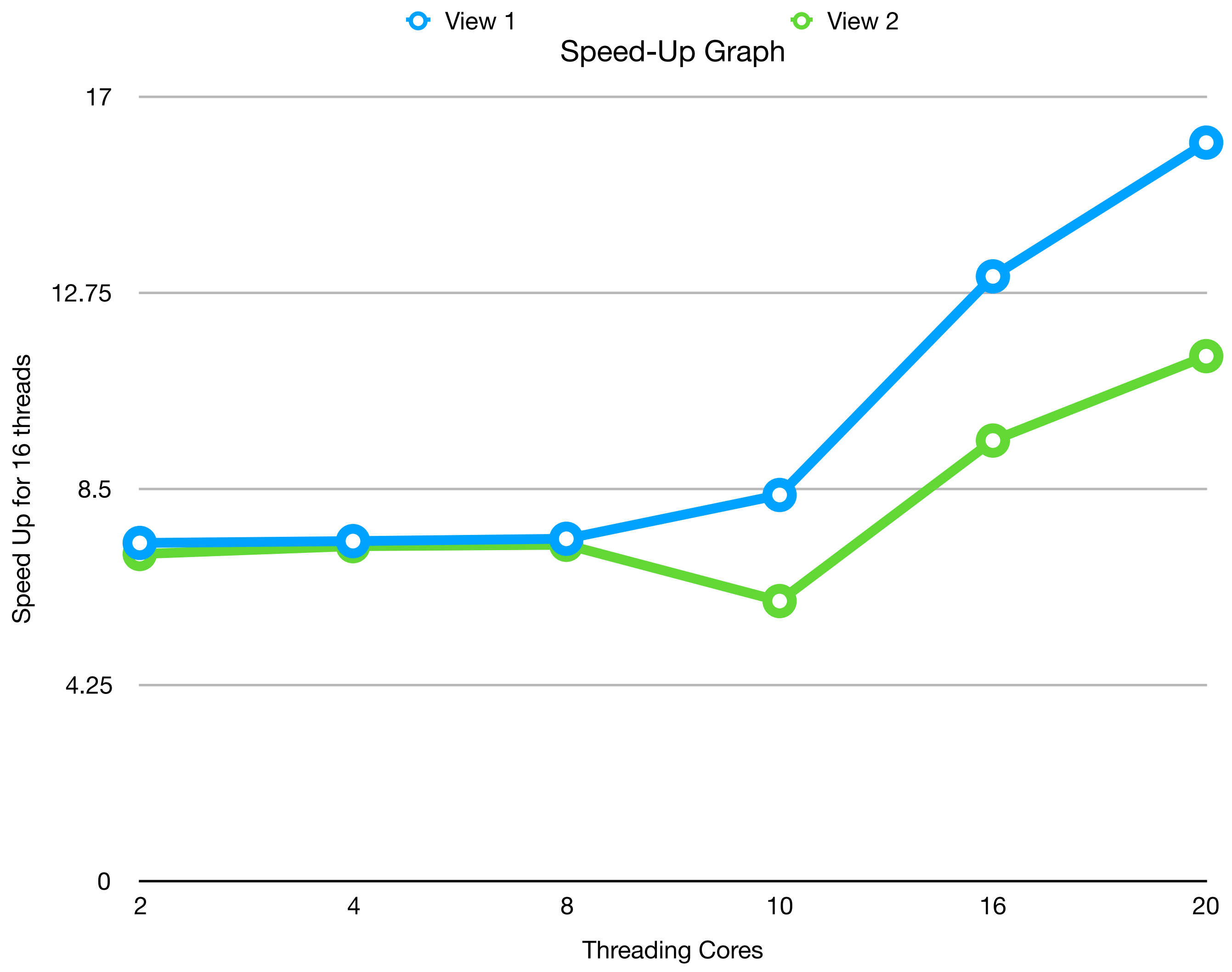
However, the second aspect of ISPC- its ability to use SPMD vector lanes to optimize across a single core also becomes useful in our program. This is why despite using tasks, which allows for asynchronous execution of our program, our speed-up is only about 2.2-2.5 (2,4 or 8 Core CPU) when split into 64 tasks.

Changing the number of tasks, with different CPU configurations, seemed to yield no positive result. The SAXPY program is already optimized to utilize ISPC well and hence is unlikely to yield a linear speed up, even if changed.

2. Threads Mandelbrot:
i)
Using threads in Mandelbrot, we see a speed-up in processing time. Using 2 threads, the speedup is 1.99 and 1.70 for views 1 & 2 respectively over serial implementation.

ii)
When we use a differing number of cores, we see a marginal improvement in performance, but there is a steep jump in the performance when moving from 8 threading cores to 16 (E5 2665 processor).
The following graph shows the speed up for 16 threads implementation across different number of cores over a sequential implementation.
The table below shows the speedup for (optimized) 10 and 20 threading cores using the E5-2670v2 processor.

Threading Cores	View 1 Speedup	View 2 Speedup
10	8.37	6.07
20	16.01	11.38



iii:
The time taken for a thread to execute goes down with the increase in the number of cores. When the number of cores increases, there is a positive speedup- except for an anomaly with view 2 - 10 core (E5 2670 V2). The speedup increases dramatically from 10 cores onwards. Some of the over performance of 2 and 4 cores can be attributed to CPU scaling (CPU Percent) on the palmetto network.

iv:
Table showing the thread count to speedup on an 8 core E5 2665 CPU-

Threads	View 1 Speedup	View 2 Speedup
1	1	1
2	2.00	1.71
4	3.82	2.93

Threads	View 1 Speedup	View 2 Speedup
8	7.19	5.32
16	13.11	9.55

Achieving speedup:
In order to achieve speed up on both the views close to 8.5 on an entire E5-2665 CPU, changes need to be made to the threading code. The un-optimised speed-up on 16 threads is 7.35 and 5.81 respectively for the 2 views.

On reviewing the thread times for execution, it is obvious that certain thread ids almost always complete more quickly. For example, even in multiple iterations, the 1st and last threads complete first in a 4 thread implementation for view 1. This is despite thread 4 being launched last. After juxtaposing thread execution (for which data is split by height) with the image generated after execution, one notices that the threads corresponding to the outer darker areas on the picture tend to execute faster, than the white areas.
Since these images tend to be roughly symmetrical vertically, there is a possibility of splitting the image into two baskets of tasks- one computationally intensive, the other which is not. Doing so, we can pass one from each basket to each thread for execution. To illustrate for 4 threads; split the image into 2 vertically- then each into 4. Pass the 1st part of each into the 1st thread, 2nd part of each to the 2nd, etc. We can hence expect each thread to finish at roughly the same time.

The observed speed-up on a full E5 2665 processor is 13.11 and 9.55 for the 2 views respectively. Steep jump is observed going from 8 to 16 threads vs 4 to 8 as seen in the table above.

3. ISPC Mandelbrot:
Using 4-wide SSE vector instruction, we would expect a speed-up of 4 using a single core (non-task) ISPC implementation. However, since some parts of the Mandelbrot image are more computation intensive, we only get a speedup of 3.04 (approximately). In the case that a pixel passes a certain if condition, the Mandelbrot code will add additional computation. In this case linear speed-up is lost. For example, in view 1, the middle section is more computationally intensive than the top section- as seen in thread execution times in the previous question's result.

i: On running Mandelbrot with tasks, the speedup is approximately 2.16 for 2 threads, 2.7 for 4 threads, and 7.93 for 8 threads, over without tasks ISPC implementation for view 1. A 16 thread implementation of the same, only offered a marginal speed-up compared to 8 threads.

ii: The 8 thread implementation has a speed-up of 25.69 & 22.53 over the sequential version for 8 threads in View 1 and 2 respectively. Given the increase in performance from 2 to 4 then 8 threads, the 8 thread implementation appears most appropriate. Further increases in threading yields smaller improvements in performance for the CPU configuration (1 entire e5-2665 processor) or even a marginal decline. Hence, the 8 thread implementation is accepted as optimal on a 16 thread machine.

CPU	View 1 Speedup	View 2 Speedup
2	23.99	15.81
4	24.23	15.76
8	25.69	22.53
16	25.38	22.33

Threads	View 1 Speedup	View 2 Speedup
2	6.73	4.99
4	8.51	7.71
8	13.86	12.59
16	25.38	22.33