Parallel Square Root Computation

### Code Files:

Code files for all the three parts of this problem can be found in their respective folders i.e. Part 1, Part 2 and Part 3.

### Performance Comparisons:

In this section, we describe the performance comparisons between serial and parallel methods for finding square root computation using ISPC on a single CPU core (no tasks launched) and multi-core (with tasks). Also, we explain execution of ISPC code with different launch tasks i.e. different threads so as to partition computation accordingly.

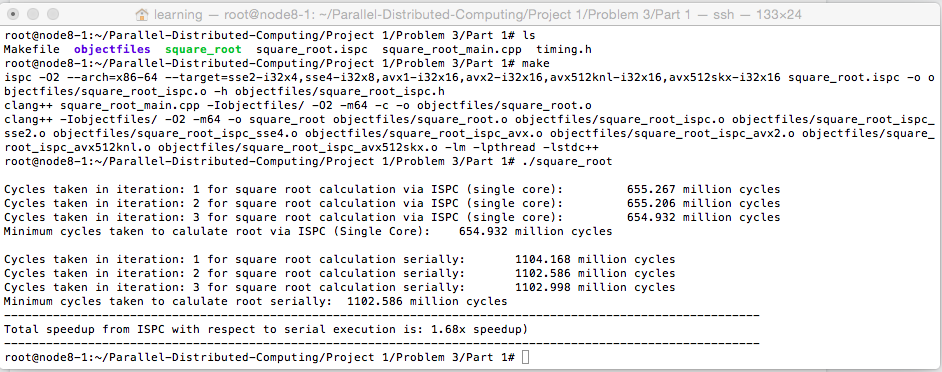
Also, we will see the effects of Intrinsics AVX instructions on the speed up by comparing the speeds between manually written AVX program with AVX instructions, AVX instructions generated from ISPC program and SSE4 instructions generated from ISPC program.

#### Comparison between serial and SIMD parallel program

NOTE: Please refer to the code files present in **Part 1** folder.

##### **Results:**

Below is the screenshot of results of executing serial and SIMD parallel execution using ISPC on single core:

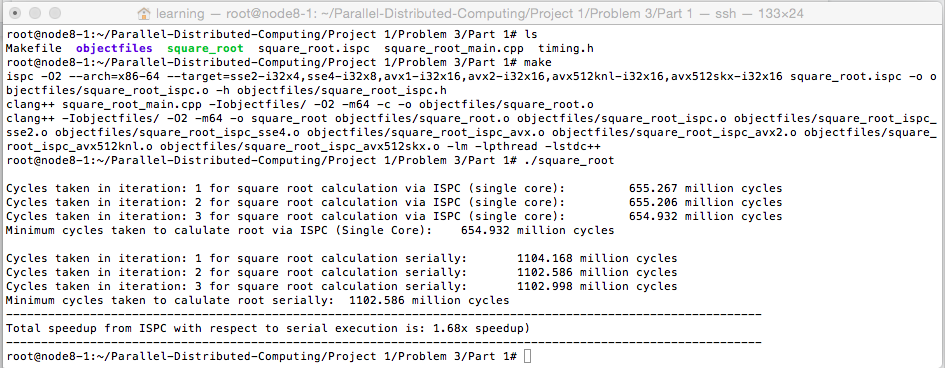


As we can see in the screenshot above, we observed a **speedup of 1.68x** when computation is performed by SIMD parallelization that uses ISPC on a single CPU core compared to that of serial execution for finding the square roots of 20 million numbers.

##### **Instructions to execute the code:**

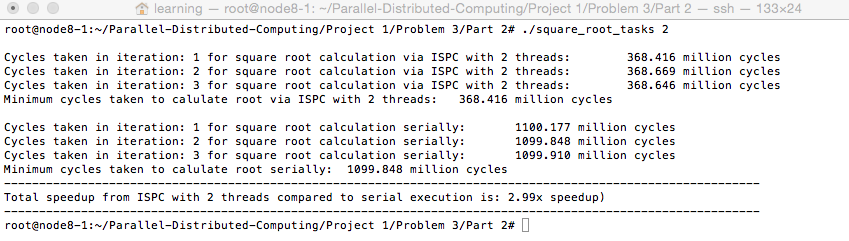
1. Move inside the Part 1 folder and you will see a bunch of files which do performance comparison between serial and SIMD parallel execution using ISPC on a single CPU core.
   1. ***square\_root\_main.cpp:*** This is the main file which generates random numbers and executes the functions that calculates square roots of the numbers generated using Newton’s method. It calls serial square root function and ISPC square root function on single core.
   2. ***square\_root.ispc:*** This is the file which contains ispc function that calculates square root of 20 million numbers.
   3. **Makefile:** This file contains all the instructions that are required to generate ***square\_root\_ispc.h*** (header file consisting of ISPC function) and other object files of different target instructions set. Basically, it will compile all the code files and generate an executable file as well i.e. ***square\_root***
2. Now, execute make file using below command:  
   ***make***
3. After compiling the code files, run the executable file i.e. ***square\_root*** to get the results:  
   ***./square\_root***
4. In each of the methods (serial and parallel), we are calculating the square roots 3 times and choose the lowest time among those iterations for better comparison.

##### Comparison between serial, ISPC on single core and multi-core (using tasks) program NOTE: Please refer to the code files present in **Part 2** folder. **Results:**

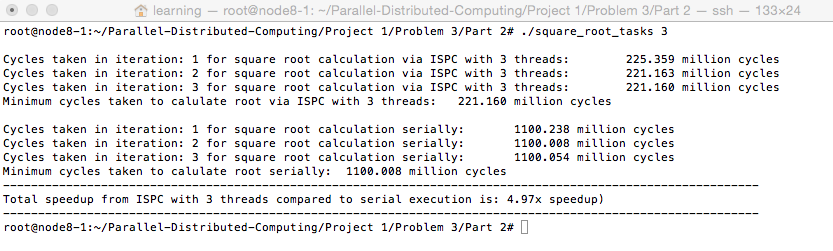
Below is the screenshot of results of executing serial, ISPC on single core (with no tasks) and ISPC on multi-core (using tasks)  
  
**Serial v/s ISPC (single core i.e. 1 thread/task):  
**

*Speedup by ISPC (1-thread) i.e. single core:* ***1.68x***

**Serial v/s ISPC (multicores - 2 threads/tasks):**

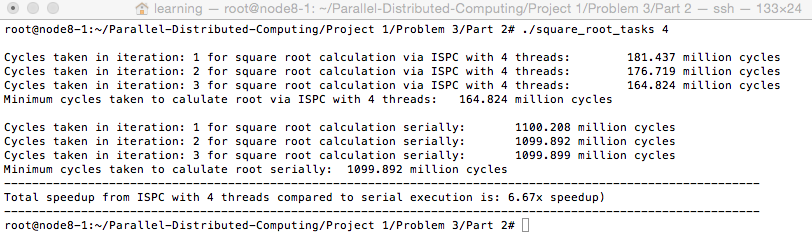
****

*Speedup by ISPC (2-threads):* ***2.99x***

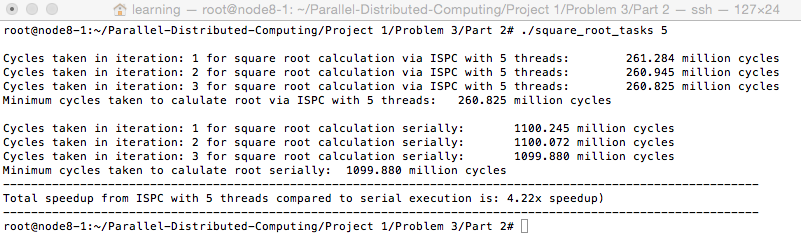
**Serial v/s ISPC (multicores - 3 threads/tasks):  
**

*Speedup by ISPC (3-threads):* ***4.97x***

**Serial v/s ISPC (multicores - 4 threads/tasks):**

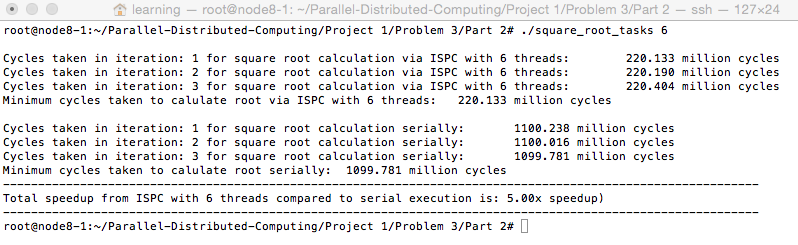
*****Speedup by ISPC (4-threads) :* ***6.67x***

**Serial v/s ISPC (multicores - 5 threads/tasks):**

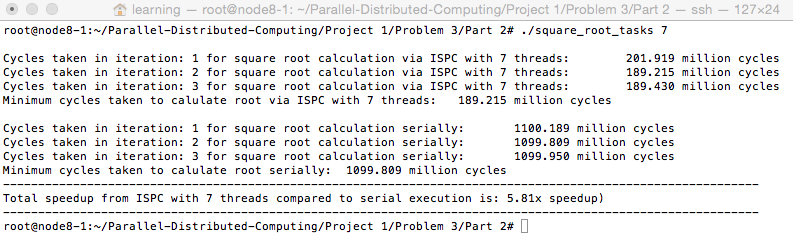
****

*Speedup by ISPC (5-threads) :* ***4.22x***

**Serial v/s ISPC (multicores - 6 threads/tasks):**

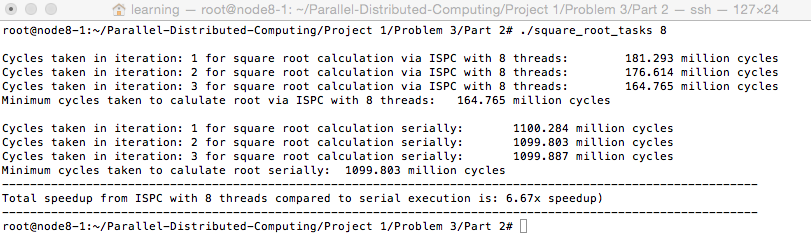
****

*Speedup by ISPC (6-threads) :* ***5.00x***

**Serial v/s ISPC (multicores - 7 threads/tasks):  
**

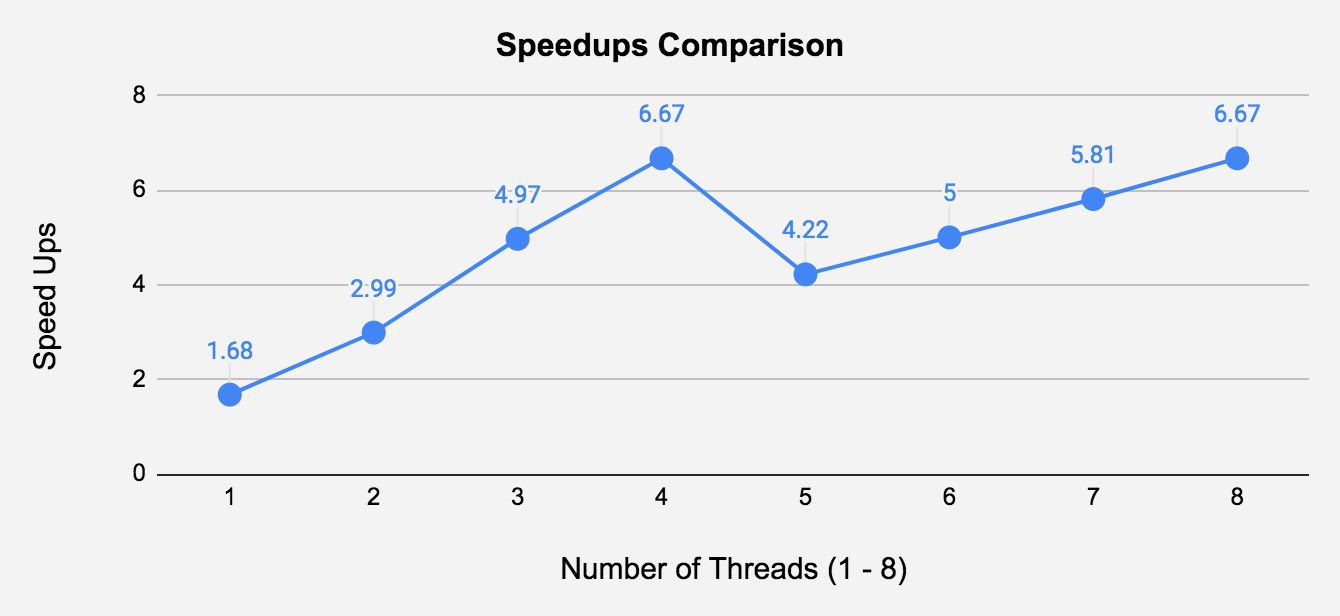
*Speedup by ISPC (7-threads) :* ***5.81x***

**Serial v/s ISPC (multicores - 8 threads/tasks):**

****

*Speedup by ISPC (7-threads) :* ***6.67x***

##### Graph for speedups comparison:



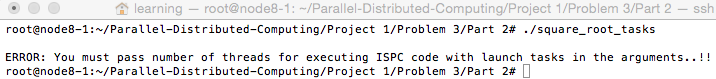
##### Discussion:

In this section, we went through the effects of threads while performing parallel execution of ISPC program that computes square root of 20 million numbers and the speed up with respect to sequential execution method for performing square root.

1. When the number of threads/tasks is 1, the speedup is the same as that of Part 1 of this problem i.e. SIMD parallelization using ISPC execution which is very much valid as it runs computation on the same single core with only 1 thread/worker/task doing the needful.
2. As we increase the number of threads i.e. launch more tasks the speed up increases linearly but not consistently from 1 to 8 threads. It reaches the peak speed up for 4 threads as seen in the above graph thus achieving speed up of **6.67x** with respect to sequential execution.
3. When we increase threads before 4, we see a linear drop in speed up for 5 threads and again as well increase the threads it increases and reaches the same peak at 8 threads i.e. same as 4 threads thus achieving speed up of **6.67x.**
4. Thus, the performance decreases at some point and one of the reasons might be because of either thread inactivity resulting in under utilization of hardware.
5. One more major reason which we think that achieved the highest speedup at 4 and 8 threads is the machine has 4 cores and each core executes 1 thread. Thus, at a given point in time all threads would be running in case of 4 threads thus utilizing 4 cores fully and also for 8 threads, 4 threads in one clock cycle and other 4 in other clock cycles would be running in 4 cores thus fully utilizing it.

##### **Instructions to execute the code:**

1. Move inside the Part 2 folder and you will see a bunch of files which do performance comparison between serial and ISPC on single and multicores by using threads/workers/tasks for computing the square root of 20 million numbers.
   1. ***square\_root\_main.cpp:*** This is the main file which generates random numbers and executes the functions that calculates square roots of the numbers generated using Newton’s method. It calls serial square root function and also launches tasks for computation by calling ISPC square root function based on the user inputted number of threads.
   2. ***square\_root\_tasks.ispc:*** This is the file which contains ispc function that calculates square root of 20 million numbers and accepts a number of threads as primary parameter which is used to launch the corresponding number of tasks.
   3. **Makefile:** This file contains all the instructions that are required to generate ***square\_root\_tasks\_ispc.h*** (header file consisting of ISPC function) and other object files of different target instructions set. Basically, it will compile all the code files and generate an executable file as well i.e. ***square\_root\_tasks***
2. Now, execute make file using below command:  
   ***make***
3. After compiling the code files, run the executable file i.e. ***square\_root\_tasks*** to get the results:  
   ***./square\_root\_tasks 2  
   NOTE:*** *If you don’t pass the number of threads as an argument while running the executable file, you will get an error message and you won’t be able to proceed further. Check the below error message:*

******

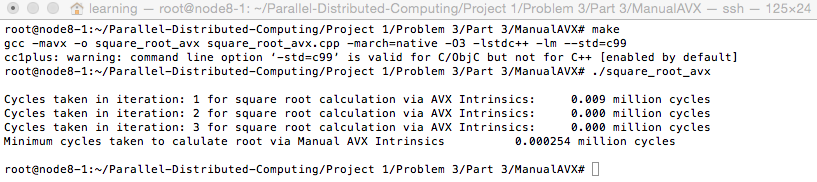
1. In each of the methods (serial and parallel), we are calculating the square roots 3 times and choose the lowest time among those iterations for better comparison

#### Effects of AVX Intrinsics (manual and from ISPC) on speedup

In this section we will look at the effect of AVX Intrinsics on performance and compare the performance of programs written manually in AVX with that of AVX instruction set generated from ISPC on different targets and also with SSE4 target.

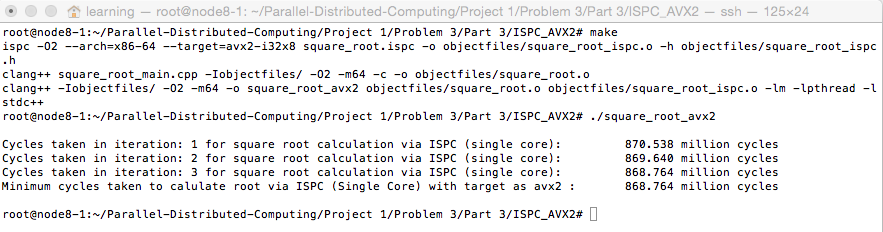
##### Results:

**Manual AVX Program:**

****

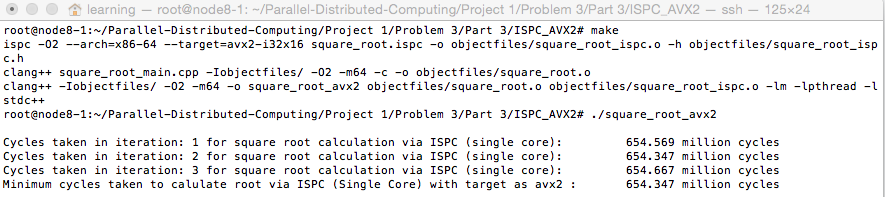
As seen in the above screenshot, we can see that it takes very less machine cycles to complete the entire execution of computing the square root of 20 million numbers compared to that of earlier programs. In this program, we have used the inbuilt AVX Intrinsics library that loads the data in the vectors and computes the square root via the AVX square root function. And that might be one of the reasons why we can see the best performance until now.

**AVX generated by ISPC: (avx2-i32x8)**

****

As seen in the above screenshot, we can see that the performance in terms of minimum machine cycles taken to calculate the square root of numbers is low. One of the major reasons why the performance is lower for AVX instructions generated by ISPC compilers with that manually written AVX program by using inbuilt functions is because AVX code is very faster than that of one generated by ISPC compiler.

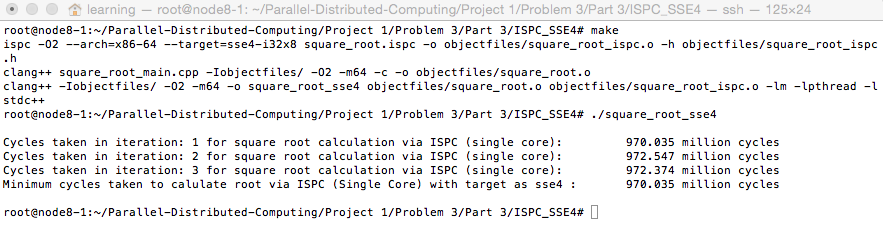
**AVX generated by ISPC: (avx2-i32x16)**

****

Now, as seen in above image the AVX instruction set program generated by ISPC compiler for target as avx2-i32x16 performs better than the one generated for target as avx2-i32x8. This is very obvious as we have moved to a higher configuration of target where we are asking the compiler to divide the computation among 16 gang sizes or i.e. 16 instances in a gang compared to that of 8 in earlier cases with 32-bit masking.

However, the performance is still lower than that of AVX code written manually by using the AVX instructions and inbuilt functions due to the same reasons that were mentioned earlier.

**SSE4 generated by ISPC: (sse4-i32x8)**



As seen in the above image, the performance of the program for the instruction set target as SSE4 is lowest compared to that of AVX code written manually or generated by ISPC compiler. One of the reasons is because gang size is only 8 but that is not of much importance because even with size 8 the performance is lower than avx2-i32x8 target instruction set. The only primary reason is AVX code or program is very fast as it deals with vectors that speeds up the program execution because all the vectors work in parallel for performing the computation.

##### Discussion:

1. Majoring of the points is already discussed in each topic of performance comparison between manually written AVX, ISPC generated AVX and SSE4 instruction sets.
2. Manually written AVX programs work the fastest among every other program followed by AVX programs generated by ISPC compilers.
3. Some reasons are AVX programs are designed to be faster than ISPC programs and AVX programs deal with vectors that help in speedup during execution.
4. Irrespective of this, ISPC is widely used because of the ease of programming and also understanding the program is very easy.

##### **Instructions to execute the code:**

Move inside the Part 3 folder and you will see 3 other folders - ISPC\_AVX2, ISPC\_SSE4 and ManualAVX

***ManualAVX***:

This folder contains the manually written AVX code that we had discussed in the very beginning of this topic.

1. Move inside the ManualAVX folder and you will see a bunch of files which will compute square root of 20 million numbers via Manually written AVX program with the help of inbuilt libraries.
   1. ***square\_root\_avx.cpp:*** This is the main file which generates random numbers and executes the functions that calculates square roots of the numbers generated using Newton’s method by using vectors provided by AVX Intrinsics.
   2. **Makefile:** This file contains all the instructions that are required to generate object files of the avx program and also executable files. Basically, it will compile all the AVX Intrinsics code and generate an executable file as well i.e. ***square\_root\_avx***
2. Now, execute make file using below command:  
   ***make***
3. After compiling the code files, run the executable file i.e. ***square\_root\_avx*** to get the results:  
   ***./square\_root\_avx***
4. We are calculating the square roots 3 times and choose the lowest time among those iterations for better comparison.

***ISPC\_AVX2***:

This folder contains the ISPC program that can be compiled to generate ***avx2-i32x-8*** and ***avx2-i32x-16*** target instructions sets for AVX.

1. Move inside the ISPC\_AVX2 folder and you will see a bunch of files which will compute the square root of 20 million numbers via ISPC program that can be compiled in different AVX target instructions sets using ***Makefile***.
   1. square\_root\_main.cpp: This is the main file which generates random numbers and executes the functions that calculate square roots of the numbers generated using Newton’s method by SIMD based ISPC program.
   2. Makefile: This file contains all the instructions that are required to generate object files of the ISPC program and also executable files. Basically, it will compile all the ISPC code and generate an executable file as well i.e. square\_root\_avx2
   3. This executable file will have a target instruction sets of AVX2 i.e. ***avx2-i32x-8*** and ***avx2-i32x-16***
   4. *For generating* ***avx2-i32x-8*** *instructions set target related executable file, make sure the* ***target*** *is assigned as* ***avx2-i32x-8*** *and similarly for* ***avx2-i32x-16*** *the* ***target*** *is assigned as* ***avx2-i32x-16*** *in the* ***Makefile.*** *Thus, you will need to change the* ***target*** *variable in Makefile for each case manually.*
2. Now, execute make file using below command:

**make**

1. After compiling the code files, run the executable file i.e. square\_root\_avx2 to get the results:  
   **./square\_root\_avx2**
2. We are calculating the square roots 3 times and choose the lowest time among those iterations for better comparison.

***ISPC\_SSE4***:

This folder contains the ISPC program that can be compiled to generate ***avx2-i32x-8*** target instruction set for SSE4.

1. Move inside the ISPC\_SSE4 folder and you will see a bunch of files which will compute the square root of 20 million numbers via the ISPC program that can be compiled in the SSE4 target instruction set using ***Makefile***.
   1. square\_root\_main.cpp: This is the main file which generates random numbers and executes the functions that calculate square roots of the numbers generated using Newton’s method by SIMD based ISPC program.
   2. Makefile: This file contains all the instructions that are required to generate object files of the ISPC program and also executable files. Basically, it will compile all the ISPC code and generate an executable file as well i.e. square\_root\_sse4
   3. This executable file will have a target instruction set of SSE4 i.e. ***sse4-i32x-8***
2. Now, execute make file using below command:  
   ***make***
3. After compiling the code files, run the executable file i.e. *square\_root\_sse4* to get the results:  
   ***./square\_root\_sse4***
4. We are calculating the square roots 3 times and choose the lowest time among those iterations for better comparison.

### Machine Specifications:

We won’t be 100% sure about this as we had used ***node8-1*** from the ***grid*** ***machine*** in the Orbit Systems for doing Problem 3.

**CPU**: 4 Core and 1 thread per core

**Memory**: 8 GB

**OS**: Ubuntu

**Image**: mariasfirstimage.ndz