Introduction to Parallel and Distributed Computing

Project 2- Report

Group -17

Team members:

Saurabh Singh; S.singh@rutgers.edu

Rachit Shah; rachit.shah@rutgers.edu

Ridip De; ridip.de@rutgers.edu

William Cheng; william.cheng@rutgers.edu

1) **Sequential version of square root program with Newton's Method**:
   - Regular C program with smarter version of newtons method to calculate square roots.
   - Successfully runs, 30000000, 45000000 and 60000000 random real numbers between 0 and 5

a) Please look for the folder named "Single_Core_Seq". The folder contains the following files:
   - SquareRootFunction.cpp
   - SquareRootFunction.out

b) How to run: Please execute the following commands:
   - To compile:

   gcc -g SquareRootFunction.cpp -o SquareRootFunction.out
   - To Run:

   ./SquareRootFunction.out

c) Following table shows the time taken in milliseconds by the program to run on different array sizes.

Number of Elements in the array	15000000	30000000	45000000	60000000
Time taken (In milliseconds)	372.312 ms	743.436 ms	1117.706 ms	1480.293 ms

d) Observations:
   - Since there is no penalization involved of any kind, the time increases in the same proportion as the elements increases.
   - As the elements get doubled from 15000000 to 30000000 the time also goes up (exactly doubles) from 372 ms to 743 ms and 743/372 = 2. This trend can be observed for 45000000 and 60000000 elements too.

2) Pure SIMD version (Using just "For-each" construct of ISPC) with Newton's method:
   - Program uses only the "For-Each" construct of ISPC, hence uses pure SIMD for penalization.
   - Successfully runs, 30000000, 45000000 and 60000000 random real numbers between 0 and 5

a) Please look for the folder named "Single_Core_Ispc". The folder contains the following files:
   - SingleCoreIspc.out
   - SingleCoreIspcSqrt.ispc
   - SingleCoreIspcSqrt.ispc.h
   - SingleCoreIspcSqrt.ispc.o
   - SingleCoreSqrt.cpp

b) How to run: Please execute the following commands:
   - To compile:

   g++ -g SingleCoreSqrt.cpp SingleCoreIspcSqrt.ispc.o -o SingleCoreIspc.out
   - To Run:

   ./SingleCoreIspc.out

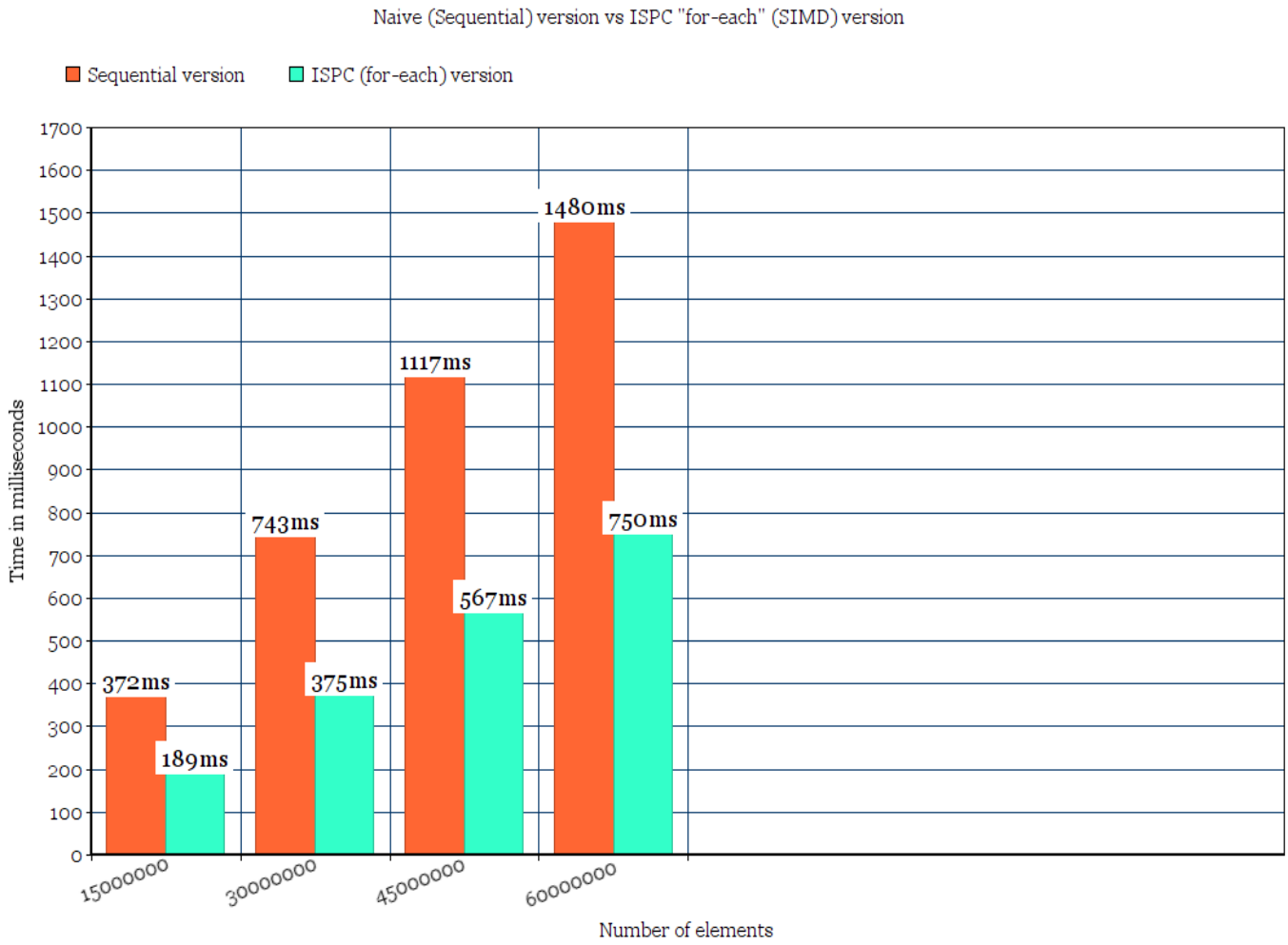
c) Following table shows the time taken in milliseconds by the program to run on different array sizes.

Number of Elements in the array	15000000	30000000	45000000	60000000
Time taken (In milliseconds)	189.151 ms	375.306 ms	567.485 ms	750.400ms

d) Observations:

- Because of the SIMD penalization, the time taken is half the time that was taken by naïve sequential implementation.
- We get **a speed up of 2x over the naive sequential** implementation.
- As the elements get doubled from 15000000 to 30000000 the time also goes up (exactly doubles) from 189 ms to 375 ms and 375/189 = 2. This trend can be observed for 45000000 and 60000000 elements too.

e) Following is the bar graph showing the time comparisons:



3) <u>Multicore version (using ISPC for each with "launch" construct) with Newton's method</u>:

- Parallel ISPC program using launch and for-each constructs.
- Successfully runs 15000000, 30000000, 45000000 and 60000000 random real numbers between 0 and 5

a) Please look for the folder named “Multiple_Core_Ispc”. The folder contains the following files:
   - MultiCoreIspc.out
   - MultipleCoreIspcSqrt.ispc.h
   - MultipleCoreSqrt.cpp
   - tasksys.o
   - MultipleCoreIspcSqrt.ispc
   - MultipleCoreIspcSqrt.ispc.o
   - tasksys.cpp

b) How to run: Please execute the following commands:
   - To compile:

```
g++ MultipleCoreSqrt.cpp MultipleCoreIspcSqrt.ispc.o tasksys.o -lpthread -o MultiCoreIspc.out
```

   - To Run: (provide the argument NUMER_OF_TASKS)

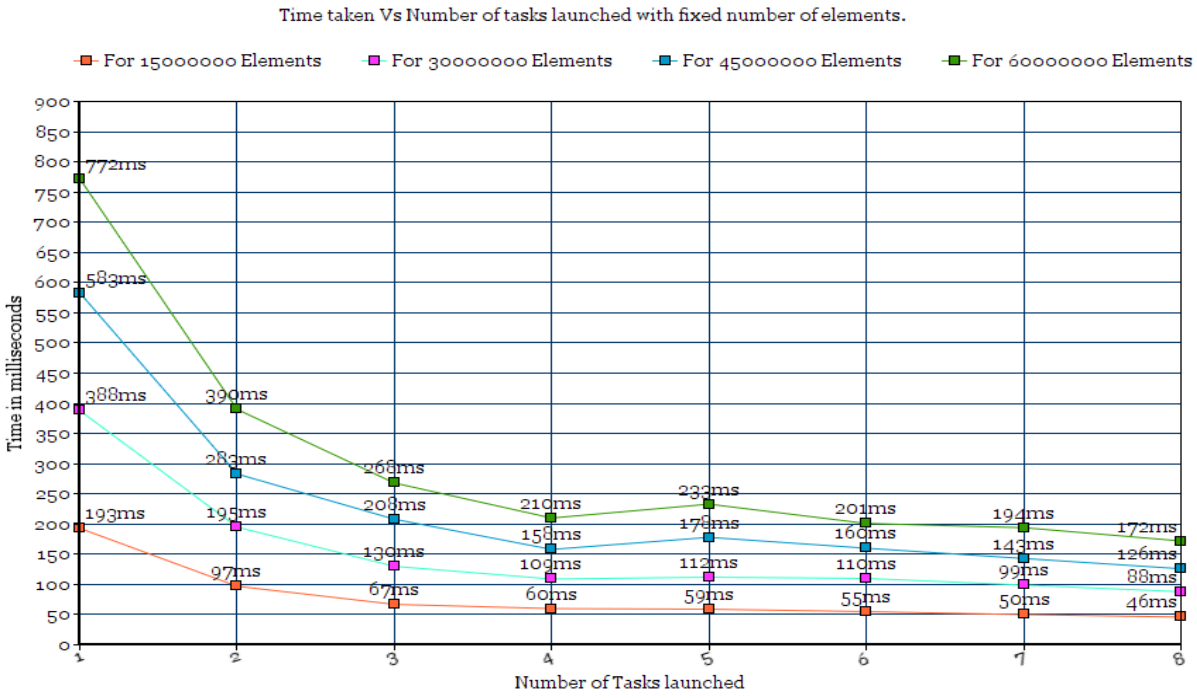
   ./MultiCoreIspc.out NUMER_OF_TASKS

c) Following table shows the time taken in milliseconds by the program to run on different array sizes and number of tasks launched.

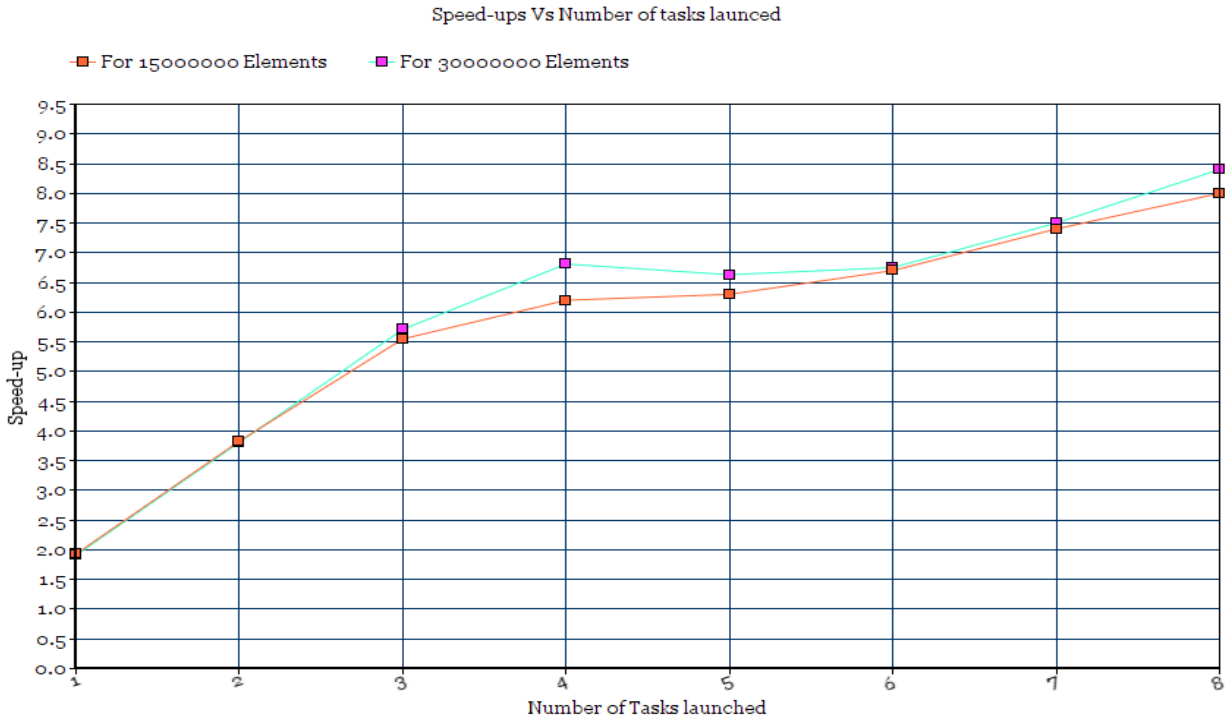
Number of Tasks launched	15000000 elements	30000000 elements	45000000 elements	60000000 elements
1	193.256 ms	388.085 ms	583.016 ms	772.322 ms
2	97.659 ms	195.490 ms	283.093 ms	390.339 ms
3	67.609 ms	130.807 ms	208.607 ms	268.128 ms
4	60.903 ms	109.330 ms	158.806 ms	210.291 ms
5	59.794 ms	112.223 ms	178.762 ms	233.082 ms
6	55.626 ms	110.752 ms	160.597 ms	201.421 ms
7	50.625 ms	99.350 ms	143.857 ms	194.112 ms
8	46.697 ms	88.708 ms	126.822ms	172.735ms

d)) Observations:
   - The results with only one task launched are almost similar to only using “for-each” construct. Which makes sense since with only one task launched, we have pure SIMD.
   - We get **a speed up of 2x over the naive sequential** implementation if we only launch one task.
   - The time reduces every time we increase the number of tasks launched

e) Following is the line graph showing the time comparisons:



f) Following is the line graph showing the speed-ups for 15000000 and 30000000 elements:



4) **AVX intrinsic version with Newton's method**:

- The program uses AVX intrinsic with packed single precision floating point data types.
- Successfully runs for 15000000, 30000000, 45000000 and 60000000 random real numbers between 0 and 5

a) Please look for the folder named "AVX". The folder contains the following files:

- AVX_Newtons_Sqrt.cpp
- AVX_Newtons_Sqrt.out
- AVX_Default_Sqrt.cpp
- AVX_Default_Sqrt.out

How to run: Please execute the following commands:

b) To compile:

```
g++ AVX_Newtons_Sqrt.cpp -march=native -o AVX_Newtons_Sqrt.out
```

c) To Run: (provide the argument NUMER_OF_TASKS)

```
./AVX_Newtons_Sqrt.out
```

Following table shows the time taken in milliseconds by the program to run on different array sizes and number of tasks launched.

Number of Elements in the array	15000000	30000000	45000000	60000000
Time taken (In milliseconds)	355.335 ms	696.203 ms	1060.370 ms	1415.375 ms

d) Observations:

- AVX does not support the absolute value functions and direct Boolean value comparison functions. To implement those, we had to do some extra logical work in the code (Please see the code). This perhaps is the reason that resulted in the AVX version with Newton's method ending up not so efficient

5) EXTRA CREDIT (As discussed with the professor):

AVX version that uses the default "_mm256_rsqrt_ps()" function:

- Program uses AVX intrinsic and the built in **"_mm256_rsqrt_ps()"** function to compute the square roots.
- Successfully runs 15000000, 30000000, 45000000 and 60000000 random real numbers between 0 and 5

a) Please look for the folder named "AVX". The folder contains the following files:

- AVX_Newtons_Sqrt.cpp
- AVX_Newtons_Sqrt.out
- AVX_Default_Sqrt.cpp
- AVX_Default_Sqrt.out

b) How to run: Please execute the following commands:
To compile:

```
g++ AVX_Default_Sqrt.cpp -march=native -o AVX_Default_Sqrt.out
```

To Run:

```
./AVX_Default_Sqrt.out
```

c) Following table shows the time taken in milliseconds by the program to run on different array sizes and number of tasks launched

Number of Elements in the array	15000000	30000000	45000000	60000000
Time taken (In milliseconds)	12.608 ms	26.899 ms	38.065 ms	52.505 ms

d) Observations:
The built-in function performs much better.

Appendix:

1) All experiments were performed on the machine – decorator.cs.rutgers.edu (iLabs machine -Computer Science dept)
2) Following are the specs of the machine:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 60
Model name:            Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
Stepping:              3
CPU MHz:               3896.453
BogoMIPS:              6784.18
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
```

L3 cache: 8192K
NUMA node0 CPU(s): 0-7

Note: The machine has 4 physical cores but because of hyperthreading shows 8 logical cpu cores.