Parallel and Distributed Programming

Project # 1: ISPC Programming

Group 9

Akhil Velagapudi
Yunqi Shen
Daniel Youssif
Antonio Rodrigues

URL: https://github.com/akhilles/ISPC

Contribution Breakdown:

All members contributed equally.

Table of Contents

Part 1: Square Root ISPC Program

For this program we started with the basic sequential code of execution and then we ran several tests to ensure that our code was working properly.

This gave us the baseline for comparing **performance** gains or losses due to ISPC and SIMD implementations.

The sequential part of the program is shown in Figure 1.

```
143     int size=20000000;
144     float * nums=(float *) malloc(sizeof(float)*size);
145     if (nums == NULL){
146     printf("malloc error");
147     return 0;}
148
149     float  * roots=(float *) malloc(sizeof(float)*size);
150     if (roots == NULL){
151     printf("malloc error");
152     return 0;}
153
154     //Serial implementation to find all roots
155     float starttime = clock();
156     srand(time(0));
157
158     for (int i=0; i<20000000; i++){
159
160     float r1=rand();
161     float r2=rand();
162     float x=(fmin(r1,r2)/fmax(r1,r2))*8;
163
164
165     float root=sqrt(x);
166     nums[i]=x;
167     float est=x;
168     while ( abs(root-est)>.0001){
169
170        est=est-(est*est-x)/(2*est);
171
172     }
173     roots[i]=est;
174     }
175     float endtime = clock();
176     printf("roots %f num %f ",roots[19999999],nums[19999999]);
177     float elapsedtime = endtime - starttime;
178     printf("elapsed time with sequential execution: %f seconds \n ",((float) elapsedtime)/CLOCKS_PER_SEC);
```

Figure 1 - Sequential Program

After compiling the program and running it we obtained the following output shown in Figure 2.

```
d
tony@tony-VirtualBox:~/Parallel Programming/Project 1/ISPC-master/part 1$ ./randispc
Last record information: root 1.410964 number 1.990814
 Elapsed time with sequential execution: 1.433680 seconds
```

Figure 2 - Sequential Program Benchmark

Then we proceeded to implement the ISPC part of the problem without recurring to the use of tasks. And the code is shown in Figure 3.

```
42 export void sr(uniform float  nums[], uniform float roots[], uniform int size){
43   varying struct RNGState state;
44    seed_rng(&state,500+programIndex);
45
46     foreach(i=0...size)
47     {float r1=random(&state);
48     float r2=random(&state);
49     float x=(min(r1,r2)/max(r1,r2))*8;
50
51     float root=sqrt(x);
52     //print("sqrt % ",x);
53     nums[i]=x;
54     float est=x;
55     while ( abs(root-est)>.0001){
56
57        est=est-(est*est-x)/(2*est);
58        //print("error =%f",est-root);
59
60     }
61     roots[i]=est;
62     //print("root % est %", root, est);
63     }
```

Figure 3 - ISPC Program Implementation

After compiling the program and running it we obtained the following output shown in Figure 4.

```
Last record information: root 2.757287 number 7.602471
Elapsed time with ISPC: 309.924988 milliseconds
```

Figure 4 - ISPC Program Benchmark

Finally we implemented ISPC with tasks starting with one task and defining the limit initially to 8 tasks. The code is shown in Figure 5.

```
 1 task void srt(uniform float * nums, uniform float * roots, uniform int size){
 2   varying struct RNGState state;
 3    seed_rng(&state,500+programIndex);
 4
 5     foreach(i=0...size)
 6     {float r1=random(&state);
 7     float r2=random(&state);
 8     float x=(min(r1,r2)/max(r1,r2))*8;
 9
10     float root=sqrt(x);
11     //print("sqrt % ",x);
12     nums[i]=x;
13     float est=x;
14     while ( abs(root-est)>.0001){
15
16        est=est-(est*est-x)/(2*est);
17        //print("error =%f",est-root);
18
19     }
20     roots[i]=est;
21     //print("root % est %", root, est);
22     }
23
24
25 }
26
27
28 export void sroot(uniform float  nums[], uniform float  roots[], uniform int size, uniform int ntasks){
29   uniform int i=0;
30     uniform int s=size/ntasks;
31   while (i<ntasks-1){
32
33   launch srt(&nums[s*i],&roots[s*i],s);
34
35   i=i+1;
36   }
37   launch srt(&nums[s*i],&roots[s*i],size-s*i);
38   sync;
39
```

Figure 5 - ISPC Program Implementation with tasks

After some testing under VirtualBox with 1 processor and 4 processors enabled we did not see any noticeable difference in processing times, so we decided to increase even further the number of tasks by doubling the number of them at every iteration as shown by the ispc program call being nested in a regular for loop which doubles the parameter to designate the number of tasks to be created. This is shown in Figure 6.

```
189
190     for (int tsk = 1; tsk<300000; tsk=tsk*2){
191     starttime = clock();
192     sroot(nums, roots, size,tsk);
193     // printf("num %f",roots[90]);
194     //sravx(nums, roots, size, 4);
195     //sravxtasks(nums, roots, size, 4);
196     roots[1999];
197     nums[1999];
198     endtime = clock();
199     printf("Last record information: root %f number %f \n ",roots[19999999],nums[19999999]);
200     elapsedtime = endtime - starttime;
201     printf("Elapsed time with ISPC and %d task(s): %f milliseconds \n ",tsk,((float) elapsedtime*1000)/CLOCKS_PER_SEC);
202
203     }
```

Figure 6 - For loop implemented to launch ISPC Program with different number of tasks

After compiling and running the program we obtained the following results shown in Figure 7.

```
Last record information: root 2.757287 number 7.602471
Elapsed time with ISPC and 1 task(s): 323.523987 milliseconds
Last record information: root 2.591058 number 6.713522
Elapsed time with ISPC and 2 task(s): 324.057007 milliseconds
Last record information: root 2.497119 number 6.235571
Elapsed time with ISPC and 4 task(s): 346.079987 milliseconds
Last record information: root 2.624953 number 6.890303
Elapsed time with ISPC and 8 task(s): 337.583008 milliseconds
Last record information: root 1.457687 number 2.124838
Elapsed time with ISPC and 16 task(s): 353.210999 milliseconds
Last record information: root 1.714115 number 2.937920
Elapsed time with ISPC and 32 task(s): 329.510010 milliseconds
Last record information: root 2.268534 number 5.146242
Elapsed time with ISPC and 64 task(s): 370.907990 milliseconds
Last record information: root 2.790841 number 7.788600
Elapsed time with ISPC and 128 task(s): 372.880005 milliseconds
Last record information: root 2.671745 number 7.138123
Elapsed time with ISPC and 256 task(s): 327.369995 milliseconds
Last record information: root 2.542817 number 6.465877
Elapsed time with ISPC and 512 task(s): 368.822021 milliseconds
Last record information: root 2.632282 number 6.928830
Elapsed time with ISPC and 1024 task(s): 333.528015 milliseconds
Last record information: root 2.697299 number 7.275311
Elapsed time with ISPC and 2048 task(s): 336.947998 milliseconds
Last record information: root 1.179292 number 1.390729
Elapsed time with ISPC and 4096 task(s): 329.198029 milliseconds
Last record information: root 2.012116 number 4.048609
Elapsed time with ISPC and 8192 task(s): 337.971985 milliseconds
Last record information: root 2.043046 number 4.174038
Elapsed time with ISPC and 16384 task(s): 332.028992 milliseconds
Last record information: root 1.830286 number 3.349948
Elapsed time with ISPC and 32768 task(s): 359.137970 milliseconds
Last record information: root 2.597244 number 6.745614
Elapsed time with ISPC and 65536 task(s): 349.824982 milliseconds
Last record information: root 1.219615 number 1.487460
Elapsed time with ISPC and 131072 task(s): 395.241974 milliseconds
```

Figure 7 - ISPC Program with tasks Benchmark

Results.

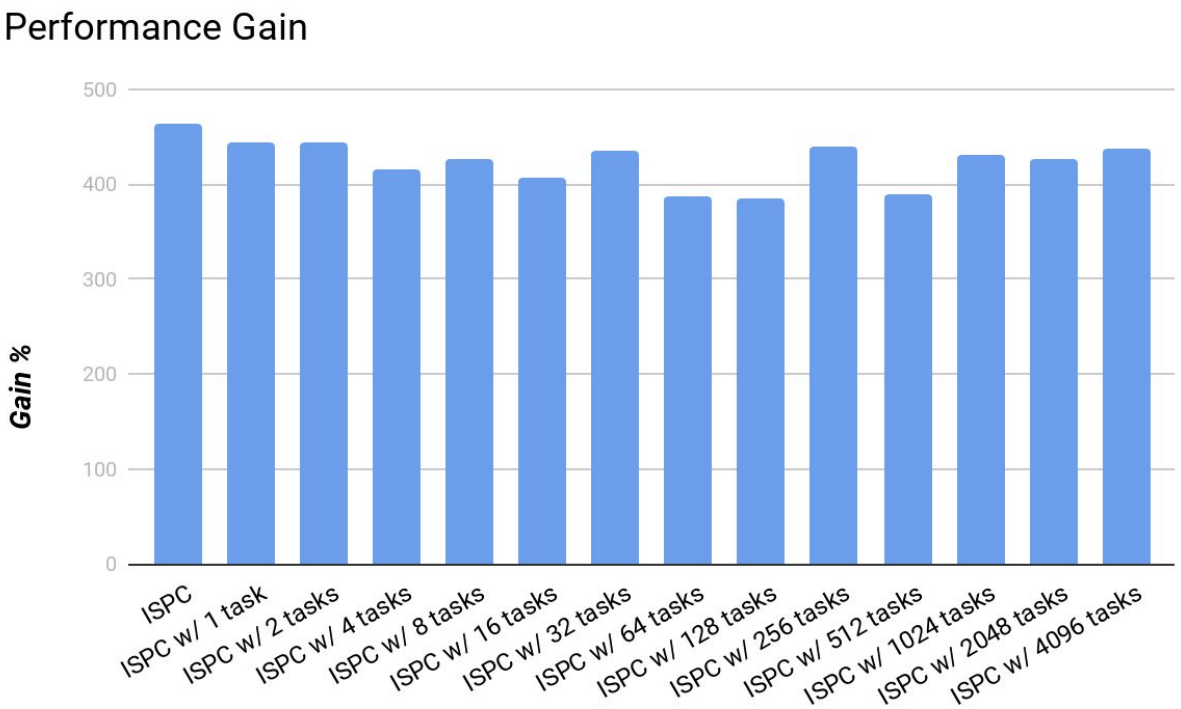
Based on the collected information we can calculate the approximate speedup due to ISPC when compared to Sequential Execution by using the following formula.

$$Speedup = \frac{Sequential\ time}{ISPC\ time} * 100 = \frac{1.433680\ s}{309.925\ ms} * 100 = 463.6\%$$

When we implemented tasks we did not see a very noticeable difference between 1 task and thousands of tasks, leading us to believe that our application is memory intensive and further benchmark improvements might not be feasible. Also we notice that creating tasks slightly decreases performance possibly due to communication overheads. Therefore we calculate the performance improvement compared to the sequential program by applying the same formula we used previously.

$$Speedup = \frac{Sequential\ time}{ISPC\ with\ 1\ task\ time} * 100 = \frac{1.433680\ s}{323.524\ ms} * 100 = 443.1\%$$

Here we calculated the performance improvement against the best time obtained by the tests we performed. We found that as you create more tasks the performance gets slightly affected by the amount of tasks spawned.



The previous chart only shows the relative speedup when compared to the sequential program. Also we decided not to include the data for more than 4096 spawned tasks as it is not meaningful.

In conclusion, we do see the benefits of SIMD and ISPC processing versus sequential processing even on our test machine powered by an old first generation Intel i7 quad core manufactured in 2010.

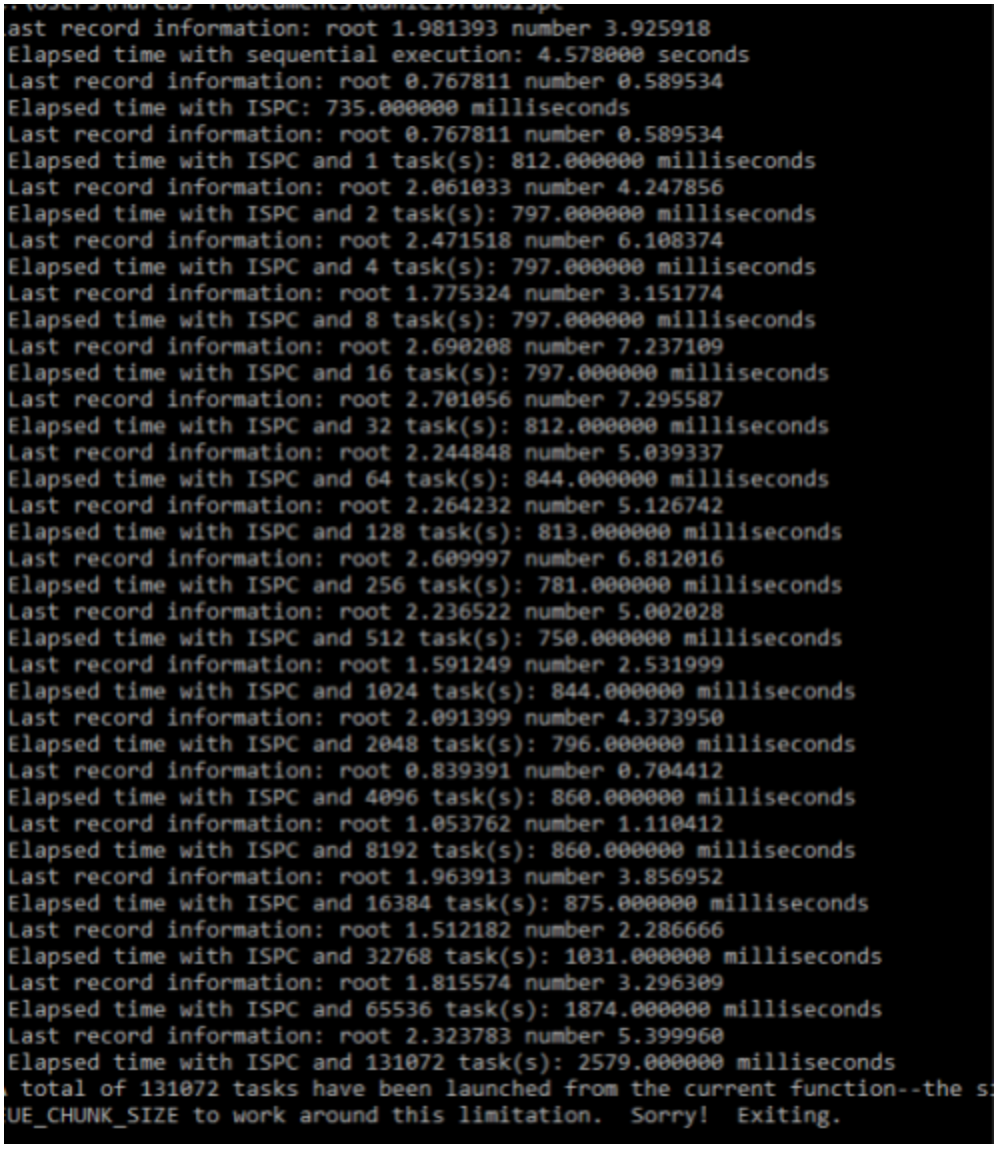
The machine we used to test this module is powered by an Intel core i7-960 quad core processor with Hyperthreading and supporting up to SSE4.2 instructions and has a 4-wide SIMD capability.

AVX

ISPC compiled to avx1-i32x8

```
ast record information: root 1.318992 number 1.739739
Elapsed time with sequential execution: 4.063000 seconds
Last record information: root 2.705659 number 7.320471
Elapsed time with ISPC: 687.000000 milliseconds
Last record information: root 2.705659 number 7.320471
Elapsed time with ISPC and 1 task(s): 750.000000 milliseconds
Last record information: root 1.940515 number 3.765599
Elapsed time with ISPC and 2 task(s): 735.000000 milliseconds
Last record information: root 1.016949 number 1.034185
Elapsed time with ISPC and 4 task(s): 750.000000 milliseconds
Last record information: root 2.075762 number 4.308785
Elapsed time with ISPC and 8 task(s): 750.000000 milliseconds
Last record information: root 1.233812 number 1.522292
Elapsed time with ISPC and 16 task(s): 703.000000 milliseconds
Last record information: root 2.301981 number 5.299110
Elapsed time with ISPC and 32 task(s): 750.000000 milliseconds
Last record information: root 2.467377 number 6.087922
Elapsed time with ISPC and 64 task(s): 766.000000 milliseconds
Last record information: root 1.937936 number 3.755595
Elapsed time with ISPC and 128 task(s): 734.000000 milliseconds
Last record information: root 1.669550 number 2.787223
Elapsed time with ISPC and 256 task(s): 719.000000 milliseconds
Last record information: root 2.147917 number 4.613544
Elapsed time with ISPC and 512 task(s): 718.000000 milliseconds
Last record information: root 1.323554 number 1.751795
Elapsed time with ISPC and 1024 task(s): 750.000000 milliseconds
Last record information: root 2.237758 number 5.007555
Elapsed time with ISPC and 2048 task(s): 750.000000 milliseconds
Last record information: root 2.476794 number 6.134480
Elapsed time with ISPC and 4096 task(s): 782.000000 milliseconds
Last record information: root 1.455491 number 2.118443
Elapsed time with ISPC and 8192 task(s): 828.000000 milliseconds
Last record information: root 0.725065 number 0.525718
Elapsed time with ISPC and 16384 task(s): 813.000000 milliseconds
Last record information: root 0.860662 number 0.740645
Elapsed time with ISPC and 32768 task(s): 921.000000 milliseconds
Last record information: root 2.246706 number 5.047682
Elapsed time with ISPC and 65536 task(s): 1390.000000 milliseconds
Last record information: root 2.759278 number 7.613453
Elapsed time with ISPC and 131072 task(s): 2157.000000 milliseconds
 total of 131072 tasks have been launched from the current function--
UE_CHUNK_SIZE to work around this limitation.  Sorry!  Exiting.
```

avx1-i32x16



There is not much of a meaningful difference between the two compilation targets.

Sequential vs. AVX speed up avx1-i32x8 =4.063sec/.687 sec= **5.914**

Sequential vs. AVX speed up avx1-i32x16=4.578sec/.735 sec=**6.229**

Then AVX intrinsics was implemented directly:

```
C:\Users\Marcus Y\Documents\daniel>randavx
Last record information: root 2.153895 number 4.639261
 Elapsed time with sequential execution: 3.641000 seconds
 Last record information: root 2.081035 number 4.330704
 Elapsed time with AVX: 1203.000000 milliseconds
 Last record information: root 0.643847 number 0.414538
 Elapsed time with AVX and 1 task(s): 1187.000000 milliseconds
 Last record information: root 2.239003 number 5.013130
 Elapsed time with AVX and 2 task(s): 1156.000000 milliseconds
 Last record information: root 1.787228 number 3.194183
 Elapsed time with AVX and 4 task(s): 1204.000000 milliseconds
 Last record information: root 1.367964 number 1.871325
 Elapsed time with AVX and 8 task(s): 1187.000000 milliseconds
 Last record information: root 2.813221 number 7.913995
 Elapsed time with AVX and 16 task(s): 1094.000000 milliseconds
 Last record information: root 1.860080 number 3.459899
 Elapsed time with AVX and 32 task(s): 1219.000000 milliseconds
```

Speedup of Sequential vs. AVX =3.641sec/1.203 sec= **3.027**

Here we see a speedup of roughly 3x compared to the sequential, but the manual avx is still considerably slower than the automatically generated avx by ISPC. This could be due to implementation differences; for example, in the manual implementation some of the logic required to decide when to stop iterating to determine the square root was done sequentially due to difficulty in using the Intrinsics API to do so. Also, ISPC uses its specific random number generator, whereas the manual code was using a C based random number generator and hence that particular function call may receive no vectorization benefit. We also see the pattern of tasks having no significant impact on performance as exhibited in the ISPC implementation.

Note. The AVX tests were done a machine with AMD A4 5000 CPU 4 core at 1.5GHz per core (64 bit).

Part 2: Password Cracking ISPC Program

Research on Hash and Password Cracking

Hash Function and Properties

A **hash function** is basically a function that maps an arbitrary size of data to a fixed size of data, which are strings or characters in terms of password. It is designed to act as a "one-way function". One data structure that implements hash function is hash table, which is widely used for rapid data lookup. Hash tables map the search key to a list of data.

A good hash function will

- always generate the same hash value for a given input key,
- map the expected inputs as evenly as possible over its output range
- have a defined output range

Such hashing is commonly used to accelerate data searches.[5]On the other hand, cryptographic hash functions produce much larger hash values, in order to ensure the computational complexity of brute-force inversion.[2] For example, SHA-1, one of the most widely used cryptographic hash functions, produces a 160-bit value. Producing fixed-length output from variable length input can be accomplished by breaking the input data into chunks of specific size. Hash functions used for data searches use some arithmetic expression which iteratively processes chunks of the input (such as the characters in a string) to produce the hash value.[5] In cryptographic hash functions, these chunks are processed by a one-way compression function, with the last chunk being padded if necessary. In this case, their size, which is called *block size*, is much bigger than the size of the hash value.[2] For example, in SHA-1, the hash value is 160 bits and the block size 512 bits.from Wikipedia

- change hash values for each run of the program
- have data normalization features such as ignoring case differences of letters
- have a continuous data
- be not invertible

Password Hash Generation

Cryptographic Algorithms (public-key cryptography)

- **AES**
  - has become one of the most popular algorithm
  - is fast
- **SHA1**
  - one of the most secure hash algorithm

- **DES**
    - is no longer secure as it uses 56-bit key
    - is easy to be cracked
- **RSA**
    - asymmetric cryptography, ensures security with public key open
    - is slow and can only handle limited amount of data
- **MD5**
    - is used to check for data integrity

Public-key cryptography, also known as *asymmetric cryptography*, is a type of cryptographic algorithm. In such algorithm, there is a pair of a public key and a private key. Those two keys are involved in Math. Data encrypted by one key, can only be decoded using the other key. Just by knowing one of the keys, the adversary cannot work out the other one. Therefore, making one of the key public will not compromise the secrecy of the other one.

Adversary's Approach to the given Password Hash

The adversary can guess the hash function by looking at the pattern. One way to investigate **which hash function is used** is to take take all different algorithms and hash/encode your plain text and see which of the output matches the pattern of the given hash.

However, the given hash could be hashed from a **salted** password, meaning that extra strings or characters are inserted to password before it gets hashed. One way is to still use brute force to crack and at the end, try to exclude the known salts.

Dictionary attack vs. Rainbow Table attack

Dictionary attack

- A guessing attack which uses precompiled list of options
- Resource requirement
    - based on knowing key information about a particular target such as family member names, birthday (involved with social engineering), or based on patterns seen across a large number of users and known passwords.
    - requires pre-generation of dictionary first
    - requires more analysis time, as it compares hashes between the given hash and all those in the dictionary entries, so it is basically batch cracking

Rainbow Table attack

- A rainbow table is a precomputed table for reversing cryptographic hash functions with hash of a plain-text password up to a certain length consisting of a limited set of characters. It is an example of space/time trade off.
- Resource requirement
    - more storage and less computer processing time than brute-force attack.
    - requires pre-computation of hash chains.
    - requires less analysis time, as it just compares values in the table with the given hash
- Cannot crack salted passwords

When to use Dictionary attack vs. Rainbow Table attack?

If the password is salted and the adversary knows the salt, it should use dictionary attack. A rainbow table is generally an offline-only attack. In a brute force attack or dictionary attack, you need to spend time either sending your guess to the real system to running through the algorithm offline. Given a slow hashing or encryption algorithm, this wastes time. Also, the work being done cannot be reused. Therefore, if the adversary would like to reuse a table to break multiple passwords, it is better to use rainbow table to save time. However, if it is one-time, rainbow table attack might wastes time on creating unused password candidates, and a brute-force/dictionary attack is prefered.

Report of Code

The hash function implemented is MD5.

```
void md5(uint8_t initial_msg[8], int initial_len, unsigned int *h0, unsigned int *h1, unsigned int *h2, unsigned int *h3) {

    unsigned int a, b, c, d, i;

    unsigned int w[16];
    memset(w, 0, 64);

    if (initial_len == 2) {
        w[0] = 0x800000 | (initial_msg[1] << 8) | (initial_msg[0]);
    } else if (initial_len == 3) {
        w[0] = 0x80000000 | (initial_msg[2] << 16) | (initial_msg[1] << 8) | (initial_msg[0]);
    } else if (initial_len == 4) {
        w[0] = (initial_msg[3] << 24) | (initial_msg[2] << 16) | (initial_msg[1] << 8) | (initial_msg[0]);
        w[1] = 0x80;
    } else if (initial_len == 5) {
        w[0] = (initial_msg[3] << 24) | (initial_msg[2] << 16) | (initial_msg[1] << 8) | (initial_msg[0]);
        w[1] = 0x8000 | (initial_msg[4]);
    } else if (initial_len == 6) {
        w[0] = (initial_msg[3] << 24) | (initial_msg[2] << 16) | (initial_msg[1] << 8) | (initial_msg[0]);
        w[1] = 0x800000 | (initial_msg[5] << 8) | (initial_msg[4]);
    } else if (initial_len == 7) {
        w[0] = (initial_msg[3] << 24) | (initial_msg[2] << 16) | (initial_msg[1] << 8) | (initial_msg[0]);
        w[1] = 0x80000000 | (initial_msg[6] << 16) | (initial_msg[5] << 8) | (initial_msg[4]);
    } else if (initial_len == 8) {
        w[0] = (initial_msg[3] << 24) | (initial_msg[2] << 16) | (initial_msg[1] << 8) | (initial_msg[0]);
        w[1] = (initial_msg[7] << 24) | (initial_msg[6] << 16) | (initial_msg[5] << 8) | (initial_msg[4]);
        w[2] = 0x80;
    }

    w[14] = 8*initial_len;

    for (i = 0; i < 16; i++){
        //cout << hex << w[i] << " ";
    }
    cout << endl;


    // Initialize hash value for this chunk:
    a = 0x67452301;
    b = 0xefcdab89;
    c = 0x98badcfe;
    d = 0x10325476;

    FF (a, b, c, d, w[ 0], S11, 0xd76aa478); /* 1 */
    FF (d, a, b, c, w[ 1], S12, 0xe8c7b756); /* 2 */
    FF (c, d, a, b, w[ 2], S13, 0x242070db); /* 3 */
    FF (b, c, d, a, w[ 3], S14, 0xc1bdceee); /* 4 */
    FF (a, b, c, d, w[ 4], S11, 0xf57c0faf); /* 5 */
    FF (d, a, b, c, w[ 5], S12, 0x4787c62a); /* 6 */
    FF (c, d, a, b, w[ 6], S13, 0xa8304613); /* 7 */
    FF (b, c, d, a, w[ 7], S14, 0xfd469501); /* 8 */
    FF (a, b, c, d, w[ 8], S11, 0x698098d8); /* 9 */
    FF (d, a, b, c, w[ 9], S12, 0x8b44f7af); /* 10 */
    FF (c, d, a, b, w[10], S13, 0xffff5bb1); /* 11 */
    FF (b, c, d, a, w[11], S14, 0x895cd7be); /* 12 */
    FF (a, b, c, d, w[12], S11, 0x6b901122); /* 13 */
    FF (d, a, b, c, w[13], S12, 0xfd987193); /* 14 */
```

Sequential crack in a recursive way.

```
    II (b, c, d, a, w[ 1], S44, 0x85845dd1); /* 56 */
    II (a, b, c, d, w[ 8], S41, 0x6fa87e4f); /* 57 */
    II (d, a, b, c, w[15], S42, 0xfe2ce6e0); /* 58 */
    II (c, d, a, b, w[ 6], S43, 0xa3014314); /* 59 */
    II (b, c, d, a, w[13], S44, 0x4e0811a1); /* 60 */
    II (a, b, c, d, w[ 4], S41, 0xf7537e82); /* 61 */
    II (d, a, b, c, w[11], S42, 0xbd3af235); /* 62 */
    II (c, d, a, b, w[ 2], S43, 0x2ad7d2bb); /* 63 */
    II (b, c, d, a, w[ 9], S44, 0xeb86d391); /* 64 */

    *h0 = 0x67452301 + a;
    *h1 = 0xefcdab89 + b;
    *h2 = 0x98badcfe + c;
    *h3 = 0x10325476 + d;

}

void recurse_crack(uint8_t vals[8], int length, int depth, unsigned int h0, unsigned int h1, unsigned int h2, unsigned int h3
    ){
    if (depth+1 == length) {
        for (uint8_t a = 32; a < 127; a++){
            vals[depth-1] = a;

            unsigned int aa, bb, cc;
            aa = bb = cc = 0;

            if (length == 2){
                aa = 0x800000 | (vals[0] << 8);
            } else if (length == 3) {
                aa = 0x80000000 | (vals[0] << 16) | (vals[1] << 8);
            } else if (length == 4) {
                aa = (vals[0] << 24) | (vals[1] << 16) | (vals[2] << 8);
                bb = 0x80;
            } else if (length == 5) {
                aa = (vals[0] << 24) | (vals[1] << 16) | (vals[2] << 8);
                bb = 0x8000 | vals[3];
            } else if (length == 6) {
                aa = (vals[0] << 24) | (vals[1] << 16) | (vals[2] << 8);
                bb = 0x800000 | (vals[3] << 8) | vals[4];
            } else if (length == 7) {
                aa = (vals[0] << 24) | (vals[1] << 16) | (vals[2] << 8);
                bb = 0x80000000 | (vals[3] << 16) | (vals[4] << 8) | vals[5];
            } else if (length == 8) {
                aa = (vals[0] << 24) | (vals[1] << 16) | (vals[2] << 8);
                bb = (vals[3] << 24) | (vals[4] << 16) | (vals[5] << 8) << vals[6];
                cc = 0x80;
            }

            cracked(length, aa, bb, cc, h0, h1, h2, h3);
        }
    } else {
        for (uint8_t a = 32; a < 127; a++){
            vals[depth-1] = a;
            recurse_crack(vals, length, depth+1, h0, h1, h2, h3);
        }
    }
}
```

Implement ispc.

```
void crack_ispc(uint8_t initial_msg[8], int len, bool serial, int tasks) {
    unsigned int h0, h1, h2, h3, aa, bb, cc;
    md5(initial_msg, len, &h0, &h1, &h2, &h3);
    aa = 0;
    bb = 0;
    cc = 0;

    if (serial) {
        uint8_t vals[8];
        recurse_crack(vals, len, 1, h0, h1, h2, h3);
    } else {
        ispc::launch_tasks(len, h0, h1, h2, h3, tasks);
    }
```

Description of Experiment

We try to crack a 4-char password on a 16-core computer. We run 10 iterations so that it is easier to compare the time.

After compiling the program and running it with single core sequential without ispc implementation we obtained the following output.

```
CRACKED
 Sequential Implementation.Time elapsed 9.62679 secs
```

Then, we proceed to implement ISPC with single core without tasks. The output is as follows:

```
ISPC and no tasks: 7.75961 secs
```

We find that there is an improvement of performance using ispc.
Finally, we proceed to implement ISPC with tasks starting with 1 task. We decided to try number of tasks from 1 to 20. and see if there is a speed improvement. The outputs are in the following:

```
tasks 1:        5.51627 secs
tasks 2:        2.97694 secs
tasks 3:        2.15212 secs
tasks 4:        1.77128 secs
tasks 5:        1.44373 secs
tasks 6:        1.20192 secs
tasks 7:        1.75295 secs
tasks 8:        1.60551 secs
tasks 9:        1.46112 secs
tasks 10:       1.33766 secs
tasks 11:       1.28896 secs
tasks 12:       1.17496 secs
tasks 13:       1.49318 secs
tasks 14:       1.36604 secs
tasks 15:       1.34626 secs
tasks 16:       1.31046 secs
tasks 17:       1.28015 secs
tasks 18:       1.17163 secs
tasks 19:       1.36725 secs
tasks 20:       1.33014 secs
```

Result

Based on the collected information we can calculate the approximate speedup due to ISPC when compared to Sequential Execution using single core by using the following formula.

$$Speedup = \frac{Sequential\ time}{ISPC\ without\ task\ \ time} = \frac{9.62679s}{7.75961s} * 100\% = 124.06\%$$

Then, when we implement ispc with 1 task, we get more speedup as follows:

$$Speedup = \frac{Sequential\ time}{ISPC\ with\ 1\ task\ \ time} = \frac{9.62679s}{5.51627s} * 100\% = 172.52\%$$

When we implemented tasks we did not see a very noticeable difference between 6 tasks and any further more tasks beyond 6, leading us to believe that our application is memory intensive and further benchmark improvements might not be feasible. Also we notice that creating more than 6 tasks slightly decreases performance possibly due to communication overheads.

The maximum speedup we can achieve is when having 6 tasks, which is in the following:

$$Speedup = \frac{Sequential\ time}{ISPC\ with\ 1\ task\ \ time} = \frac{9.62679s}{1.20192s} * 100\% = 801.20\%$$

Here we calculated the performance improvement against the best time obtained by the tests we performed. We found that the speedup doesn't have a linear relationship with number of tasks implemented, and as you create more tasks the performance beyond 6 tasks gets slightly affected by the amount of tasks spawned.

In conclusion, there are performance improvement of SIMD and ISPC processing versus sequential processing, while implementing more tasks than 6 doesn't improve the performance with ISPC.

Research on existing software tools for password cracking

- Aircrack-ng
    - Doesn't not support CUDA or parallel execution
- Hashcat
    - Supports parallel execution over multiple cores

For reference, the following is 4-char password cracking implemented on an alternate 16-core machine:

```
tasks 1:        5.61968 secs
tasks 2:        3.87465 secs
tasks 3:        4.43633 secs
tasks 4:        3.85909 secs
tasks 5:        4.19067 secs
tasks 6:        3.87011 secs
tasks 7:        4.0766 secs
tasks 8:        3.85401 secs
tasks 9:        4.03296 secs
tasks 10:       3.85628 secs
tasks 11:       3.98727 secs
tasks 12:       3.8556 secs
tasks 13:       3.95803 secs
tasks 14:       3.85318 secs
tasks 15:       3.94934 secs
tasks 16:       3.86277 secs
tasks 17:       3.92684 secs
tasks 18:       3.85566 secs
tasks 19:       3.91638 secs
tasks 20:       3.85816 secs
```