Part 1 - Solution:

1) What is a hash function? What are its properties? Name and describe the most popular hash functions.

Solution:

The meaning of the verb "to hash" – to chop or scramble something – provides a clue as to what hash functions do to data. That's right, they "**scramble**" data and convert it into a numerical value. And no matter how long the input is, **the output value is always of the same length**. Hash functions are also referred to as hashing algorithms or message digest functions. They are used across many areas of computer science, for example:

- To encrypt communication between web servers and browsers, and generate session IDs for internet applications and data caching
- To protect sensitive data such as passwords, web analytics, and payment details
- To add digital signatures to emails
- To locate identical or similar data sets via lookup functions

Properties:

Hash functions are designed so that they have the following **properties**:

One-way

Once a hash value has been generated, it must be **impossible to convert it back** into the original data. For instance, in the example above, there must be no way of converting "$P$Hv8rpLanTSYSA/2bP1xN.S6Mdk32.Z3" back into "susi_562#alone".

Collision-free

For a hash function to be collision-free, no two strings can map to the same output hash. In other words, every input string must generate a unique output string. This type of hash function is also referred to as a **cryptographic hash function**. In the example hash function above, there are no identical hash values, so there are **no "collisions"** between the output strings. Programmers use advanced technologies to prevent such collisions.

Lightning-fast

If it takes too long for a hash function to compute hash values, the procedure is not much use. Hash functions must, therefore, be very **fast**. In databases, hash values are stored in so-called hash tables to ensure fast access.

Message Digest (MD)

MD5 was the most popular and widely used hash function for quite some years.

- The MD family comprises hash functions MD2, MD4, MD5 and MD6. It was adopted as Internet Standard RFC 1321. It is a 128-bit hash function.
- MD5 digests have been widely used in the software world to provide assurance about integrity of transferred files. For example, file servers often provide a pre-computed MD5 checksum for the files, so that a user can compare the checksum of the downloaded file to it.

- In 2004, collisions were found in MD5. An analytical attack was reported to be successful only in an hour by using a computer cluster. This collision attack resulted in compromised MD5 and hence it is no longer recommended for use.

Examples of Popular Hash Functions

Secure Hash Function (SHA)

Family of SHA consists of four SHA algorithms; SHA-0, SHA-1, SHA-2, and SHA-3. Though from the same family, they are structurally different.

- The original version is SHA-0, a 160-bit hash function, was published by the National Institute of Standards and Technology (NIST) in 1993. It had few weaknesses and did not become very popular. Later in 1995, SHA-1 was designed to correct alleged weaknesses of SHA-0.
- SHA-1 is the most widely used of the existing SHA hash functions. It is employed in several widely used applications and protocols including Secure Socket Layer (SSL) security.
- In 2005, a method was found for uncovering collisions for SHA-1 within a practical time frame making long-term employability of SHA-1 doubtful.
- SHA-2 family has four further SHA variants, SHA-224, SHA-256, SHA-384, and SHA-512 depending on the number of bits in their hash value. No successful attacks have yet been reported on SHA-2 hash function.
- Though SHA-2 is a strong hash function. Though significantly different, its basic design still follows the design of SHA-1. Hence, NIST called for new competitive hash function designs.
- In October 2012, the NIST chose the Keccak algorithm as the new SHA-3 standard. Keccak offers many benefits, such as efficient performance and good resistance for attacks.

RIPEMD

The RIPEMD is an acronym for RACE Integrity Primitives Evaluation Message Digest. This set of hash functions was designed by the open research community and generally known as a family of European hash functions.

- The set includes RIPEMD, RIPEMD-128, and RIPEMD-160. There also exist 256, and 320-bit versions of this algorithm.
- Original RIPEMD (128 bit) is based upon the design principles used in MD4 and found to provide questionable security. RIPEMD 128-bit version came as a quick fix replacement to overcome vulnerabilities on the original RIPEMD.
- RIPEMD-160 is an improved version and the most widely used version in the family. The 256 and 320-bit versions reduce the chance of accidental collision, but do not have higher levels of security as compared to RIPEMD-128 and RIPEMD-160 respectively.

Whirlpool

This is a 512-bit hash function.

- It is derived from the modified version of Advanced Encryption Standard (AES). One of the designers was Vincent Rijmen, a co-creator of the AES.
- Three versions of Whirlpool have been released; namely WHIRLPOOL-0, WHIRLPOOL-T, and WHIRLPOOL.

2) What is a rainbow table (w.r.t. hashing)?

Solution:
Passwords are no longer stored online without encryption. When users set a password for their account on an online platform, the string **doesn't appear in plaintext** on any database or server. Instead, online services use various cryptographic mechanisms to encrypt their users' passwords: only the **hash value** of the password appears in the database itself.

The password can't be directly identified from the hash value, either – even if you know the crypto function. There's no way to recalculate the operation.

Then there are various ways which Hacker try:

1. Hackers use **brute force attacks** instead: with this, a computer program keeps guessing until it's found the correct character sequence.
2. **Password dictionaries**: These files – which can be obtained via the internet – contain numerous passwords that are either very popular or were captured during a previous attack on another system. This saves hackers time with the decryption: First, they try all passwords in the dictionary. Depending on the complexity of the passwords (length and used characters), this can cost some time and computing power, though.

Here comes the role of Rainbow Tables:

Rainbow tables go a step further than password dictionaries, as

1. They can also be found online, and can be used to crack passwords.
2. These are files, some of which can be multiple hundred gigabytes large, contain **passwords together with their hash values** along with the encryption algorithm used. It's not complete, though. **Instead, certain chains are created from which the actual values can easily be calculated, which reduces the storage requirements of the still massive tables. With rainbow tables, hash values found in a database can be used to sort your passwords into plaintext.**

Advantages:

1. Unlike brute-forcing, performing the hash function isn't the problem here (since everything is precomputed). With all of the values already computed, it's simplified to just a simple search-and-compare operation on the table.
2. The exact password string isn't needed to be known. If the hash is matched, it doesn't matter if the string isn't the password itself. It will be authenticated.

Disadvantages:

1. A large amount of storage is required for store tables.
2. With all of the values already computed, it's simplified to just a simple search-and-compare operation on the table.

3) Whenever the attacker gets a hashed password, can he directly know what the hash function is? Or can he somehow retrieve or investigate the hash and obtain more information?

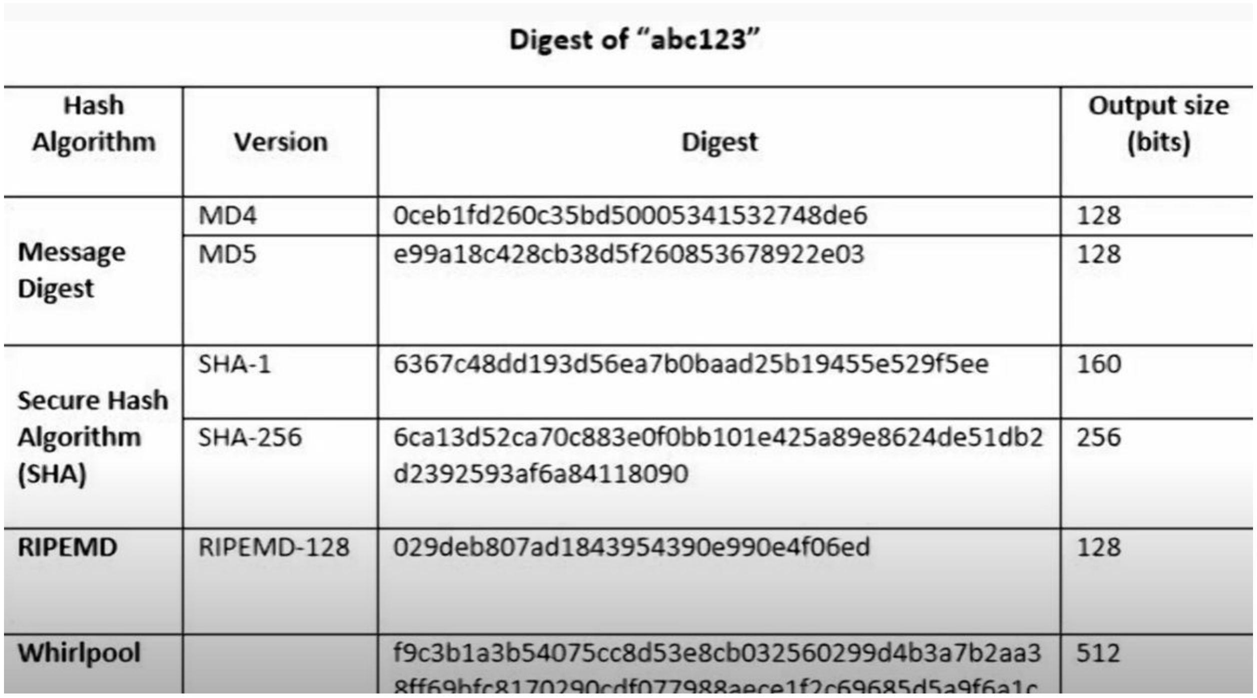
Solution:

By looking at the length, we can decide which algorithms to try. MD5 and MD2 produce 16-byte digests. SHA-1 produces 20 bytes of output. etc. There's nothing besides the length in the output of a cryptographic hash that would help narrow down the algorithm that produced it.

The investigative approach may look like as below:

1. The hash seems to contain only hexadecimal characters (each character represents 4bits)
2. For example if total count is 32 characters -> this is a 128-bits length hash.
3. So we can guess that standard hashing algorithms that comply with these specs are: haval, md2, md4, md5 and ripemd128.
4. Generally , the highest probability is that MD5 was used.

Below are shown various lengths of the hash value lengths for common hash algorithms:



Digest of "abc123"

Hash Algorithm	Version	Digest	Output size (bits)
Message Digest	MD4	0ceb1fd260c35bd50005341532748de6	128
	MD5	e99a18c428cb38d5f260853678922e03	128
Secure Hash Algorithm (SHA)	SHA-1	6367c48dd193d56ea7b0baad25b19455e529f5ee	160
	SHA-256	6ca13d52ca70c883e0f0bb101e425a89e8624de51db2 d2392593af6a84118090	256
RIPEMD	RIPEMD-128	029deb807ad1843954390e990e4f06ed	128
Whirlpool		f9c3b1a3b54075cc8d53e8cb032560299d4b3a7b2aa3 8ff69bfc8170290cdf077988aece1f2c69685d5a9f6a1c	512

However, there is no way to tell apart from the guesses like above.

Any hash function worth its salt will spread the hash values evenly throughout the entire output space, so if you have just a bunch of outputs, there's no way to tell hash functions apart.

Unless you can make some reasonable guesses about the input, and do some brute-forcing, of course.

So, there is nothing in a hash code that can tell you what algorithm was used to create it. Any strong hash code algorithm is specifically designed to not contain any traceable meta data in the hash code. There are no specific markers that identify the hash code as being a hash code, or what kind of algorithm was used.

Basically a hash algorithm tries to create a strong pseudo-random number using all the input as seed. The output is just as random as possible, there are no recognisable patterns that could be used to identify the input data or the algorithm.
If you have the input data that was used to create the hash code, you could recreate it and compare the results. However, if just a single bit in the input is different you get a completely different hash code, so the input has to be exactly the same.

4) How can the attacker eventually obtain the hash function so that he can brute-force or use a rainbow table to try any combination of characters through the hash in order to obtain the same hash value?

Solution:

It is to be seen that systems that use hash functions take an input password from the user and store the hash function from it. The system doesn't know the original password.

Attacker's obtain the hash only and after having the hashes the hacker just needs a password that generates the same hash. It doesn't matter if the password is the same as the original.

They can get the intuition of the hash function producing the desired length of hash from the hash value they grabbed. They can look through the rainbow table looking for that hash for such hash functions .Constructing a rainbow table requires two things: a hashing function and a reduction function. So accordingly we need to check the rainbow table corresponding to the expected hashing function.

5) When to use a Dictionary attack vs. a Rainbow Table attack? What are the resource requirements for each type of attack? Which uses more storage? Which requires more pre-computation? Which requires more analysis time? (per-hash vs. batch cracking)
Solution:
Dictionary Attack: It is typically a guessing attack which uses a precompiled list of options. Rather than trying every option, only try complete options which are likely to work.

Qualities:
- The dictionary or possible combinations is based upon some likely values and tends to exclude remote possibilities. It may be based on knowing key information about a particular target (family member names, birthday, etc.). The dictionary may be based on patterns seen across a large number of users and known passwords (e.g., what's the most globally likely answers). The dictionary is more likely to include real words than random strings of characters.
- The execution time of dictionary attack is reduced because the number of combinations is restricted to those on the dictionary list
- There is less coverage and a particularly good password may not be on the list and will therefore be missed

Real World Examples:
- Access to a secret club requires knowing the owner's name, you guess "Rob" or "Jake" rather than "computer"
- Given the same lock example above, you try a combinations equating to the birthday of the lock owner or the lock owner's friends and family

As explained earlier, **Password dictionaries** are the files – which can be obtained via the internet – contain numerous passwords that are either very popular or were captured during a previous attack on another system. This saves hackers time with the decryption: First, they try all passwords in the dictionary. Depending on the complexity of the passwords (length and used characters), this can cost some time and computing power, though.

Rainbow tables go a step further than password dictionaries, as they can also be found online, and can be used to crack passwords. These are files, some of which can be multiple hundred gigabytes large, contain **passwords together with their hash values** along with the encryption algorithm used. It's not complete, though. **Instead, certain chains are created from which the actual values can easily be calculated, which reduces the storage requirements of the still massive tables.** With rainbow tables, hash values found in a database can be used to sort your passwords into plaintext.

Choice between the rainbow and dictionary attack depends on the various factors like computation power,storage etc of the system.If the system has more computing power then it can use the brute force or dictionary attack (to save time in comparison to brute force by trying common passwords)

But by doing precomputation and if it has large storage then rainbow tables can be very helpful as it makes the attack more look up and less computation.

Rainbow Table attack requires more storage and precomputation.

Time Analysis:
It can be understood from the example below:
If you were to run a rainbow table attack and the 5th entry out of 500 million entries was your match, then all of the effort and time used to create the other 499,999,995 passwords may be considered wasted. However, if you are looking to break multiple passwords to reuse the table over multiple attacks, the time savings can add up.

In a brute force attack or dictionary attack, you need to spend time either sending your guess to the real system or to running through the algorithm offline. Given a slow hashing or encryption algorithm, this wastes time. Also, the work being done cannot be reused.

References:

[Secure Salted Password Hashing - How to do it Properly (crackstation.net)]

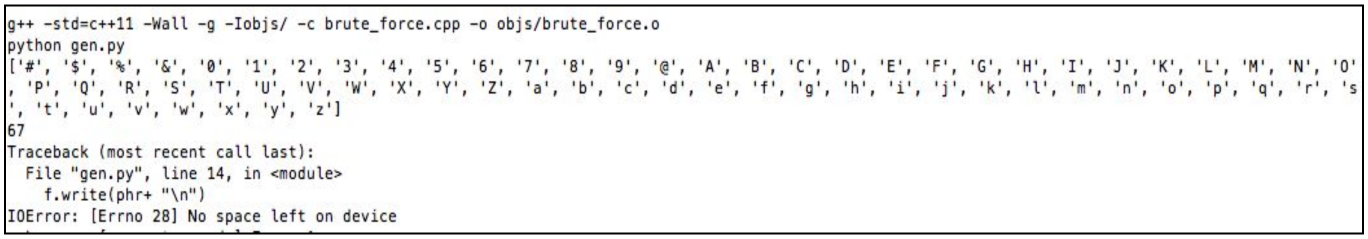
Part 2 - Solution:

1) We have implemented the serial password cracker code along with AVX Intrinsics and ISPC Tasks based parallelism code. Steps to execute the serial, AVX and ISPC Tasks based code is discussed in following sections
2) Implemented ISPC Tasks based password cracker code along with AVX Intrincics and Sequential/Serial methods to support any number of ISPC Tasks and below are the steps to execute the program.

<u>Steps to compile and execute the program:</u>

NOTE:

1. The same compiled code can be used to run for all 3 implementations - serial, AVX and ISPC Task.
2. Also, since the questions asks to support 1-8 ASCII Character length password crackers, generating all the possible combinations between chosen ASCII characters will take time.
3. We have implemented MD5 Hashing algorithm
4. We have added support for below ASCII Characters - 0 to 9, a to z and A to Z
5. Adding support for 1-8 ASCII Characters resulted in below error. As seen, there comes space constraints because the only reason is the program while compilation tries to store all the possible passwords between 1-8 characters for all the combinations of ASCII characters chosen.



```
g++ -std=c++11 -Wall -g -Iobjs/ -c brute_force.cpp -o objs/brute_force.o
python gen.py
['#', '$', '%', '&', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O'
, 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's
', 't', 'u', 'v', 'w', 'x', 'y', 'z']
67
Traceback (most recent call last):
  File "gen.py", line 14, in <module>
    f.write(phr+ "\n")
IOError: [Errno 28] No space left on device
```

6. We highly recommend to use passwords between 1-4 characters and that supports the ascii characters as discussed in point 4.
7. We highly recommend to use MD5 hash as we have implemented MD5 algorithm to get the correct decoded password.
8. Compiling program might take several minutes depending on machine storage and resources because it creates 4 files with all the combinations of ASCII characters chosen.

<u>**STEPS**</u>:

1. Navigate to Problem 2 folder and run ***make***
2. This will start the compilation process and generate 8 different text files having combinations of different ASCII characters which can be considered as passwords.
3. Execute the program: ***./brute_force_password_cracker   <md5 hash>   <number of tasks>***
4. NOTE: MD5 hash must be of length 32 and Number of tasks is required only for ISPC tasks execution

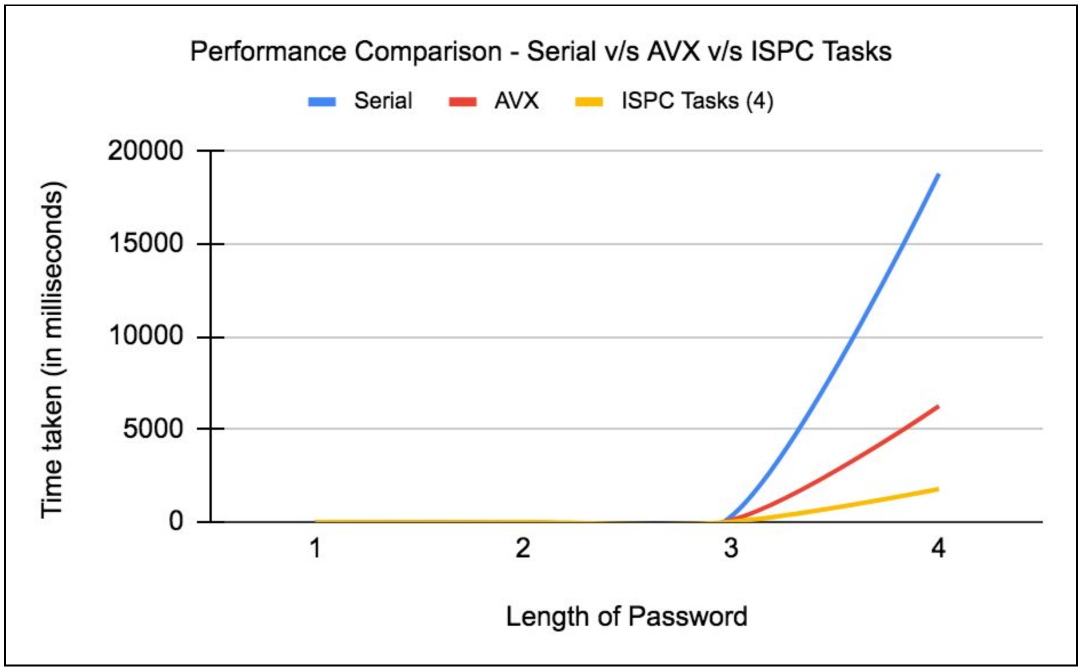
3) **Analysis**:

A. Performance Comparison between serial v/s AVX v/s ISPC Tasks based implementation:

Below is the table which shows performance i.e. time taken to crack password for a given hash value for each of the methods implemented - Serial, AVX and ISPC Tasks.

Password	Length of Password	Serial (ms)	AVX (ms)	ISPC Tasks (4) (ms)
Z	1	0	0	0
9Q	2	4	1	0
Ac4	3	305	101	29
zzzz	4	18784	6266	1790

Also, below is the graph based on the table above for better visualization of performance between all the methods implemented for password cracker.



Performance Comparison - Serial, AVX and ISPC Tasks

As seen in the above table and also the chart, ISPC Tasks based parallelism for cracking password of length 1-4 clearly outperformed other methods i.e. Serial and AVX intrinsics for cracking password from a given MD5 Hash value. If we want to calculate the speed up produced by ISPC Tasks with respect to Serial and AVX:

Speed up (Serial to ISPC Tasks) = (18784/1790) = ~**10.5**

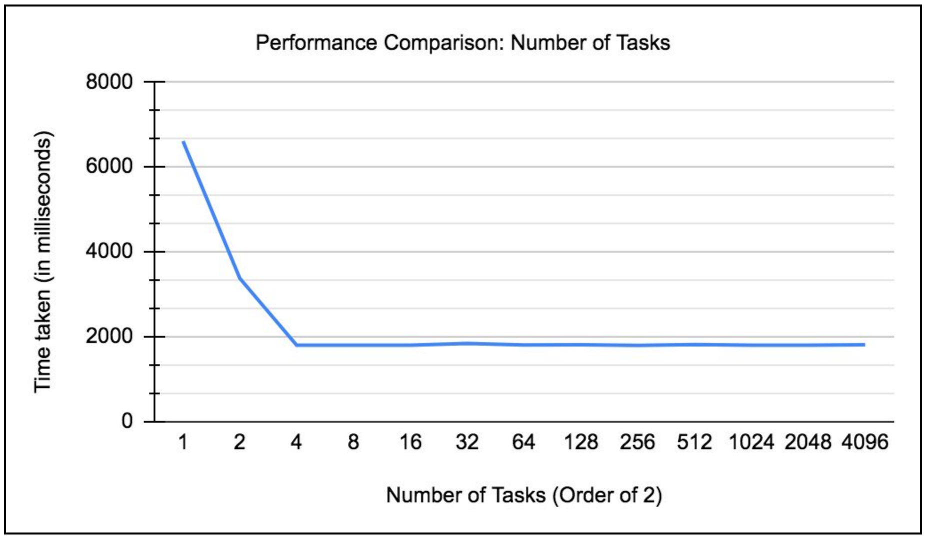
Speed up (AVX to ISPC Tasks) = (6266/1790) = ~**3.5**

B. Performance Comparison between different number of ISPC Tasks:

Below is the table which shows performance i.e. time taken to crack password for a given hash value for using a different number of ISPC tasks.

Time Taken (ISPC) (ms)	Number of Tasks
6608	1
3370	2
1792	4
1793	8
1795	16
1837	32
1797	64
1801	128
1788	256
1810	512
1793	1024
1795	2048
1801	4096

Also, below is the graph based on the table above for better visualization of performance between all the different number of ISPC Tasks used for password cracker.



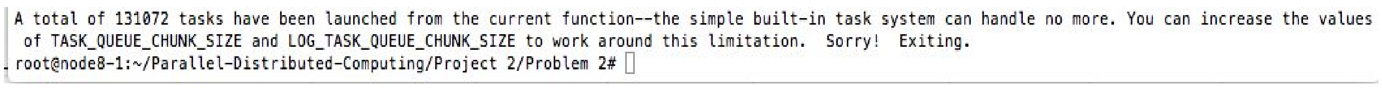
Performance Comparison - Different number of ISPC Tasks launched

As seen in the above table and also the chart, ISPC tasks help significantly improve performance of parallel execution of the program. Thus, when only 1 task is launched, time taken is the highest as seen and as we keep on increasing the number of tasks the time keeps on decreasing. But, after a particular increase in the number of tasks, the time doesn't decrease further.

As seen in the table above, when tasks = 4 are launched the performance reaches max. So, after increasing the tasks count, the performance remains almost the same. Performance with 4 tasks is also greater than with 2 tasks because ORBIT systems are of 4 cores and thus all cores will be running one task each resulting in best performance.

We also see the number of tasks when increased beyond 4, sometimes performs little low (only in milliseconds difference). For example, tasks 128, 512, 4096, etc. and the reason might be because the compiler will have to do the synchronization between different tasks that are launched and also manage the interleaving mechanism because we have only 4 cores running in the ORBIT system.

Now, we stopped after 4096 but directly used the number 1000000 for launching tasks to check the system limits. We received an error as shown below with the maximum number of tasks that can be launched which was **131072.**



```
A total of 131072 tasks have been launched from the current function--the simple built-in task system can handle no more. You can increase the values
 of TASK_QUEUE_CHUNK_SIZE and LOG_TASK_QUEUE_CHUNK_SIZE to work around this limitation.  Sorry!  Exiting.
root@node8-1:~/Parallel-Distributed-Computing/Project 2/Problem 2#
```

4) **<u>Speedup between number of ISPC tasks</u>**:

As seen in the above section, we already discussed the performance between different ISPC tasks that are launched for cracking passwords. Thus, we can see the speed up is surely achieved from tasks 1 to 4 but not further.

Speed up (1 to 4 tasks): (6608/1792) = **3.6875**

5) Testcase: **bv37qi#f**

The problem has a requirement to check whether the program decodes the hash value of the mentioned password successfully to check with other class groups. This is however not possible because the maximum password length can be 4 as discussed earlier. Because of hardware storage limits, the program doesn't store all the combinations of passwords for chosen ASCII characters. So, it would be great if the password is of lower length (max - 4) with ASCII Characters as 0-9, a-z and A-Z as discussed previously.