INFO-F404: Parallel exhaustive search with progressive saves

Raymond Lochner Najim Essakali

1 Introduction

This project deals with the implementation of a multi-process program that searches for collisions in SHA1 using OpenMPI. Massively parallel implementations of single-process programs are sometimes needed when we try to calculate a fast space of possibilities and solutions where we are considerably limited when using one single processor. This project further briefly handles the implementation of progressive saves, where one can access intermediate results on an ongoing calculation.

2 Program description

Since the point of this project is to use multiple processing units, we have to design an algorithm that imitates the algorithm of the following algorithm, which is designed for one processing unit.

Listing 1: Single processor example

```
for each current_bits in bytes
    hash = SHA1(current_bits)
    LSB = get_lsb(hash)
    for each current_bits2 in bytes
        hash2 = SHA1(current_bits2)
        LSB2 = get_lsb(hash2)
        if (LSB equal LSB2)
            print " current_bits and current_bits2 share LSB of the hash "
```

We decided to apply parallelization in the inner for loop of the previous program. Meaning that we want to split up all the possible values in

```
    for each current_bits2 in bytes
```

which can get quite large when choosing the byte value accordingly large at run-time. We opted for a master-slave architecture with one master-node and N-1 slave-nodes. The master-node is responsible for selecting these chunks and sending relevant information such as the LSB of the hash and the bytes to search in for the slave-nodes when asked for more work. As the size is user dependent on a number of bytes, two chunks representing one *space* of 4 bytes is able to be represented like this :
$[x][x][0][0] - [x][x][255][255]$ which represents one chunk of the total *bit space.*
All the chunks possible of 4 bytes are hence of the form:
[0][0][0][0] – [0][0][255][255]
[0][1][0][0] – [0][1][255][255]
...
[255][255][0][0] – [255][255][255][255]

Indeed, we opted to make chunks in our architecture this size.
The sole task of the slave-nodes on the other hand is to ask the master-node for work and do the computation shown in the inner for loop of the previous code example.

In theory and with a computer with a very very large amount of memory, it would be ideal to only calculate the SHA of each possible bit computation once and store the bit combination together with the LSB of the corresponding hash in a data structure such as a map, but we are unfortunately

bound by these constraints and have to compute the SHA and compare the LSB of the hash multiple times.
Indeed, if we opt to use a larger amount of RAM (currently, the program does barely use any), it is possible to reduce SHA calculations by simply storing some LSB of already calculated hashes in a map like data structure. This is because calculating the hash with SHA is more expensive than comparing two *unsigned char[]*.

This gives us the basic program structure of our solution for the master and slave-nodes:

Listing 2: Multi processor code

```
master-node()
    for each current_bits in bytes
        hash = SHA1(current_bits)
        LSB = get_lsb(hash)
        for each chunk in bytes
            wait for idle-slave-node
            send chunk, LSB

slave-node()
    while there is work
        get chunk, LSB
        for each bit in chunk
            hash = SHA1(current_bit)
            LSB2 = get_lsb(hash)
            if (LSB equal LSB2)
                print " current_bits and current_bits2 share LSB of the hash "
```

This leads us directly to the communication implementation of the program

3 Communication protocol

In this master-slave-node architecture, the master-node talks to all the slave-nodes and the slave-nodes are only talking to the master-node. In the working stage, the master-node chunks up the iterations into blocks. He waits for slave-nodes to send a *idle* message which follows by the master sending 4 messages: The active work flag, meaning that there is still work to do be done, followed by the current looped byte *word*, the chunk start, the chunk end and finally the LSB of the hash.
The slave-node sends the master-node a *give me work* flag and starts to receive the data in the same way the master-node sends the data. When all possible bit iterations have been finished in the master-node, he responds to *give me work* requests with a 0 flag, meaning that it has finished. This is followed by the run-time of the slave-node which is being stored by the master-node. This is matched on the slave-node side. When all slave-nodes have finished and send their working time, the master-node follows the slave-nodes into terminating.
The master-node obtains the messenger-id with *status.MPI_SOURCE*. The slave-nodes are simply ignoring the status on every interaction with MPI.

4 Performance

Running the program on 4 processors, we can see in the activity monitor, as well as by adding idle timers in the slave-nodes code, that there is no bottleneck on the master-node.
Since we are checking every possible bit iteration by every possible bit iteration, we have an algorithm of $O(2^{(n)})$ complexity. This is also represented by the time results in Table 1 and Table 2. There we can clearly see how exponentially the time increases when we add 1 byte.

5 Difficulties

The *chunking* of the total search space proved to be a challenge to implement as the communication architecture depended also on it. Deciding between the master-slave-node and a round based system was difficult. As we tried multiple possibilities of splitting up the total search space, it seemed more than difficult to create n-node equal spaces that evenly divide between them. This is why we decided to use a incrementing space method which the master-node uses to chunk the space up into evenly large parts.

Obtaining intermediate results was a problem when we first designed the architecture of communication between master and slave node. Initially, the slave-node would send the found results to the master-node. However, we realized that with an increasing amount of slave-nodes, this would push too much work onto the master-node and so we decided to let slave-nodes write immediately to file when they found matching LSB. We thought this would not be possible, as there would be cases where 2 or more nodes are trying to write to the same file, but thankfully the library open and write operations forbid this.

As with almost all our meetings with C and C++, most of the time spend for this project was to correctly implement the methods that deal with unsigned char*.
Many hours of bug findings were spend on trying to find out why 1 line of particular code does not work or why values were being copied into the next iteration of a loop when they were uniquely created in this particular loop with malloc().

Table 1: mpirun -n 4 ./main -b 1 -n 16

original word	LSB of hash	found word	matching LSB	matching hash
0a	ddf0	90	ddf0	c4595d8f743731cbc1ca0bb34be79a40d771ddf0

Process 0 : 0.007478 seconds
Process 1 : 0.007478 seconds
Process 2 : 0.007433 seconds
Process 3 : 0.007484 seconds

Table 2: mpirun -n 4 ./main -b 2 -n 24

original word	LSB of hash	found word	matching LSB	matching hash
0b16	5a76f2	be2c	5a76f2	91ac7e1350c89ae971340bc3293f5b55f85a76f2
...	...	...	...	...
febf	50a261	2400000000	50a261	55ff58cecb462333b36a7903202e4c3f7c50a261

Process 0 : 403.862364 seconds
Process 1 : 403.862363 seconds
Process 2 : 403.853168 seconds
Process 3 : 403.850892 seconds

6 Results

Running the program on 1 byte with 16 LSB (Table 1) gives us one collision and a running time of less than a 100th of a second. Running the program on 2 bytes with 24 LSB (Table 2) gives us many collision and takes almost 7 minutes. This leads up to the conclusion that we have to chose 2 bytes for the job that is supposed to exit itself without interruption.