Josh Wretlind

MATH/CSCI 440

Parallel Scientific Computing

12/08/13

**Peforming SHA3/Keccak in parallel**

*Motivation:*

Cryptographic hashing functions serve a couple of purposes for software systems. One purpose is to verify if a message or file has been transferred with no modification. The sender would both provide a hash of the file, along with the file itself. Then, when the recipient receives the file, they can hash what they received, and then compare the hash with the received file. If they are the same, the file is fully intact.

Another purpose of cryptographic hashing is for storing passwords. The designer of the software system would create a login page, and the user’s credentials would be sent to the server. Then, the server will add in a random salt to the user’s password, has the resulting combination, and store that information in the database. Then, whenever a user would like to log back in, they send their password to the server, the server adds in the salt, and hashes the resulting mixture. Then, the server will compare the value with the value stored in the database.

As the use of the internet grows, the use of hashing functions has grown as well. For large scale systems, the servers must perform this hashing operation many times a second. Therefore, it is an important algorithm to understand and very widely used.

However, given that the size of the input to a hash function is much larger than the space for the output, there are guaranteed to be collisions – that is, multiple different files or messages that wind up hashing to the same output. Due to this feature, there is a tradeoff made when designing a hashing algorithm.

This tradeoff is known as the “performance vs. security” tradeoff. What this tradeoff says is that if you have a very performant algorithm, that is, one that hashes very quickly either in serial or parallel, however machines will be able to test large sections of the input space quickly. On the other hand, if a system is very secure but not too performant, then very few developers or security experts will use the algorithm, especially with how often these algorithms are used.

This presents an interesting case for parallelization, as the assumption of “faster algorithms are better” is not necessarily true here. . Additionally, Keccak/SHA3 is a new algorithm that has recently been accepted as the next standard, so it is helpful to see what can be parallelized within the algorithm, and discover how well it parallelizes.

*Overview of Algorithm and Parallelization:*

SHA3/Keccak performs its manipulations on what is called a “state array”. This array is a 5x5xW array of bits, with W being equivalent to a lane width. What a lane width is a string of consecutive bits from within the message or input to the hash function. SHA3 specifies that a lane width is 64, to associate with 64 bit CPUs, however Keccak specifies that the width of a lane can be more general than that. For my analysis of SHA3/Keccak, I have used the SHA3 standard for lane width. Both standards specify a variable l, which is simply the log2 of the lane width.

In addition to the message to be hashed, there is also a number associated with a SHA3 function. Typically, these are listed as SHA3-X(ex: SHA3-512). This number specifies how many bits the output of the hash function should be. I will refer to this number as the digest length.

The capacity of the hash function is specified to be twice the digest length. Then, the rate of bits to be taken into the state array (called r here) is specified to be the difference between the total number of bits in the state array and the capacity of the hash function.

Then, the message is converted into bits. This conversion to bits represents the first parallelization that I was able to do. The message is broken into pieces and distributed out to the hosts working on the problem, then each host converts their message into bits(in little endian notation, as specified in the Keccak specifications), and then sent back to the master host. This host then will pad the message so that it fits into an even number of r – bit blocks. The pattern for how the message is a one, followed by zero or more zero’s, and then with a trailing one. The padded converted message is then broadcast back out to all of the hosts.

Then, for each r bit block in the message, an exclusive or operation is performed on the first r bits of the state array. This operation is performed in parallel as well, with each host performing all of the columns in the rows which they had been given. The lanes and rows are then sent back to the master, and broadcast back out to the hosts. This is to ensure that all of the hosts have the current state array.

Then, for each block, after the bits have been xor’ed into the state array, we perform 12 plus twice the value of l of a function called a “Keccak Round”. Each round permutates the state array, and represents most of the bit manipulation within Keccak. These rounds perform their manipulations on the previous round’s state array. At the end of the manipulations, the resultant state array becomes the new state array.

Each Keccak Round has 5 steps to it. These steps are known as the Theta, Pi, Rho, Chi, and Iota steps. Each one does something different to the state array. For some of these steps, I have been able to perform some parallelization on their manipulations.

The first step which the state array is taken into is called the Theta step. In this step, an array of the odd-parity of each column in the state array is developed. Then, a new array is developed by xor’ing the parity array previous to the current column as well as a copy of the parity array immediately after the current that has been bit shifted right by 1 bit. Then, this new array is xor’ed with the state array and stored back into the state array. This is performed on all of the bits in the bit array. I was able to parallelize the development of the parity array as well as the final xor’ing of the shifted array back into the state array, in a similar fashion to how I parallelized the xor operation of the next r bits into the state array.

This leads into the 2nd and 3rd steps of the Keccak Round. These steps are called the Pi and Rho steps respectively. Each word in the state array is rotated by a different triangular number, and then is placed in a different column in the matrix according to a set pattern.

The fourth step in a Keccak Round is called the Chi step. In this step, each row is recombined according to a set Boolean expression. This recombined bit is then set in the state array. I was able to parallelize this step by having each host only calculate a certain number of rows, send them back to the master, and then the master then broadcasts the new array back out to the hosts. The constants are values from a linear feedback shift register, and specified in the algorithm specifications.

Then, after all of the r bit blocks have been taken into the state array, with the specified number of Keccak round’s happening in between xor’ing new r bit blocks into the state, it is time to generate our output.

How the output hash is generated by SHA3 is that you take the first r bits of the state array, and append them to the output hash. Then, you perform the series of Keccak rounds on the state array again, and then take the first r bits out again. This is done until you reach whatever digest length you need.

*Results and Analysis:*

I performed my code using the text of John Milton’s “Paradise Lost” as my input message. Each copy of the text is about 500kB, so in order to increase the message size/see how the algorithm faired with larger sizes I concatenated additional copies of the text. I was able to test with up to an input of 1.5 MB before having issues with running at larger sizes.

Additionally, for each input length, I performed both SHA3-256 as well as SHA3-512 on the input message, in order to test for any difference in terms of having a different digest length.

Here are my results for SHA3-256 on 500kB of message size

For that same input size, here is my speedup relative to 2 processors:

Unfortunately, my speedup hasn’t improved by too much compared to when I gave my presentation. This is still primarily due to the cost of needing to have the Keccak Round function so synchronized. However, there is relative speedup for N=3 and N=4 processors here, which provides some reference point to obtain fairly reasonable values from.

Here is the absolute Efficiency for SHA3-256 using a 500kB message:

The efficiency here does not seem out of ordinary for code that has parallel slowdown rather than speedup.

For SHA3-256 with 500kb of input data, here is my Isoefficiency metric

The only real outlier on this graph seems to be the case for P=7. I cannot think of any particular reason why that would be especially bad, so it seems like it would just be normal variance.

The average Karp-Flatt Metric, using the absolute speedup, was 11.343, however, the average for the relative speedup was 1.57901, with the average for the cases where we did see relative speedup being .963597. For measuring the experimental fraction of code, the case where we did see relative speedup gives the most reasonable answer, with 96.3597% of my code.

Now I will turn to see what happens with SHA3-512 with my message at 1.5MB in sizze, so that we can compare the two.

Here is the absolute speedup I saw with SHA3-512 with 1.5 MB message size:

Interestingly, with this case, there is more variance towards the high end of the processor spectrum. This could be due to how, as your digest length grows, the function has to perform more iterations of Keccak Round and thus, there is more communication time, making the algorithm more vulnerable to network conditions.

On top of that, I noticed that our absolute speedup is lower for the lower counts of processors. This is a trend that I had noticed whenever I performed SHA3-512, and similar to above, my best judgment says that since there are more times that one would have to perform the Keccak Round method, there is more synchronization, and thus the speedup is less.

Additionally, here is the relative speedup for SHA3-512 with 1.5 MB of input:

While it is less clear on this graph, there is only one processor value that we saw relative speedup in this case. This was a trend among SHA3-512 results that I had gathered. Similar to the discussion above about why the speedup is generally less in this case, it seems to be the case that those additional synchronizations are impacting performance here as well.

The Karp-Flatt metric for this case using the case where we did see relative speedup was .972308, which is in line with the SHA3-256 results. When I averaged all of the Karp-Flatt metrics from all of the relative speedups that I had seen, the result came out to be roughly 0.965866123, corresponding with a 96.586% serial code fraction. Again, this is most likely do to a combination of the cost of having to synchronize all of the hosts, as well as the fact that a lot of the operations handled by SHA3 use the information produced immediately prior to the current iteration.

The efficiency that I got using SHA3-512 with 1.5MB of message size is the following:

These results align well with the efficiency results for SHA3-256. Given that my code suffered from parallel slowdown, it is to be expected that my efficiency would get very low quickly, and it had done exactly that.

Here is the isoefficiency metric for SHA3-512 with 1.5 MB of input data:

This graph follows the same pattern as the SHA3-256 one, but without the sudden dip at N=7. This adds to my suspicion that whatever caused N=7 to wind up fairly low was a onetime event, rather than something inherent in how I parallelized the code.

*Conclusion:*

SHA3 presents one of the more interesting cases for attempting parallelization. This is due to how the algorithm is designed, given the performance-security tradeoff. Due to this tradeoff, it is designed to make it costly to parallelize. Given the parallelization that I performed, the biggest cost is most likely synchronization and the reliance on the previously calculated state array.