Finding Hash Collisions using MPI on HPC Clusters

Melisa Cantu  
Mathematics Dept.  
Texas State University  
San Marcos, TX 78666, U.S.A.  
Email: mgc54@txstate.edu

Joon Kim  
Computer Science Dept.  
College of Staten Island, CUNY  
Staten Island, NY 11314, U.S.A.  
Email: joon.kim@cix.csi.cuny.edu

Xiaowen Zhang  
Computer Science Dept.  
College of Staten Island, CUNY  
Staten Island, NY 11314, U.S.A.  
Email: xiaowen.zhang@csi.cuny.edu  
Corresponding Author

Abstract—In cryptography, a hash function is a very important cryptographic primitive with a wide range of applications. There are three required properties for a good hash function, i.e., collision, pre-image, and second pre-image resistance. In this paper, we try to contest these properties on a popular and widely used hash function called MD5 - and its two simplified versions that we made. The birthday attack technique was used to test MD5’s general collision resistance, while the brute force method was used in the search for pre-image and second pre-image collisions. We calculated the Hamming distance to monitor the progress in our search for a collision; the smaller the Hamming distance the better. Our input domain for the MD5 hash function consisted of hexadecimal bit-strings and strategically generated ASCII character strings. Since finding hash collisions demands much more computing power and storage, we wrote C parallel programs in conjunction with the Message Passing Interface (MPI) library that runs over multiple processors / cores in the heavily used CUNY HPC cluster called Penzias. Multiple search / sort / merge algorithms were tested, not only to reduce time and space complexities, but also to improve performance. Hash distributions, numerous arbitrary meaningless and a few meaningful collisions were found.

Keywords—Hash function; MD5; hash function collisions; MPI; HPC.

I. INTRODUCTION

A hash function, short for cryptographic hash function in this paper, takes a long arbitrary size message as input and generates a short fixed size bit string as output. The fixed output bit string is called message digest, fingerprint, hash value, or simply hash. Hash functions compress a long string to a short string; inevitably, there will be a lot of collisions in which two or more different strings map to the same hash. However, for cryptographic purposes, a good hash function should run fast and satisfy collision, pre-image, and second pre-image resistance. These properties make hash functions the most useful cryptographic primitive. Applications of hash functions include, but are not limited to, digital signature, message integrity check, message authentication code (MAC), secret sharing schemes [3], pseudo-random number generator, etc.

The break of any of these hash function properties could cause dire consequences in practical applications. For instance, what happens if we can break the collision resistance property? In digital signatures, if an attacker can find a meaningful modified contract that has the same hash value as the original contract which was signed by the authority, then he can go to the court and show the judge that the authority has signed the modified contract. In this paper, for a given will in which Alice is the sole beneficiary, we are able to forge another will in which Edger becomes the beneficiary. The two wills have the same hash.

Parallel methods in finding hash collisions can be found in the literature. Oorschot and Wiener [4] proposed to use pseudo-random walks for each processor to find a distinguished point. All distinguished points are sent back to a master processor. A collision is detected when the same distinguished point appears twice in the master processor. The use of distinguished points can parallelize Pollard’s rho method [5] to achieve linear speedup. In his 2010 random graphs presentation Shamir [6] summarized some interesting cycle detection algorithms which include Floyd’s tortoise and hare, its improved version by Nivasch, stack algorithm, and multicollision by Joux and Lucks.

The rest of the paper is organized as follows. In Section II, we briefly introduce iterative hash, hash function properties, MD5 and its simplified versions, and CUNY HPC facilities. In Section III, after the introduction of pigeonhole principle and birthday attack, we describe our four major algorithms for finding hash collisions. We selectively show some experiment results in Section IV and conclude the paper in Section V.

II. HASH FUNCTIONS AND ENVIRONMENT

A cryptographic hash function [8] takes an input string of arbitrary length and generates an output string of fixed length, called hash. Most popular hash functions are iterative, and they are based on Merkle-Damgard construction.

In iterative hash function, the first step is to pad a message \( M \) of arbitrary length into multiple fixed-length blocks of \( s \)-bit. Then hash function iteratively compresses each block with compression function \( C \) to a shorter fixed-length output string, [i.e., \( C : \{0, 1\}^{s+t} \rightarrow \{0, 1\}^{t} \), where \( s > t > 0 \)], until all blocks are processed [2]. Let \( M = m_1 | m_2 | \ldots | m_n \), and \( h_i = C(h_{i-1}, m_i) \), where \( 1 \leq i \leq n \) represents concatenation. Let \( h_0 = IV \), the initial vector (value) of the hash function, then \( h_n = h \) is the final hash of \( M \), \( m_i \) and \( h_i \) are \( s \) and \( t \) bit long, respectively. \( h_i = C(h_{i-1}, m_i) \) states that \( C \) takes a \( s \)-bit
message block $m_i$ and a $t$-bit intermediate hash $h_{i-1}$ as input, and then compresses them to a $t$-bit output as a new intermediate hash $h_i$. The entire process can be written as $H(IV, M) = h$, or just $H(M) = h$.

A. Hash Function Properties

Hash functions are usually required to satisfy the following properties [1], [8]:

1. Collision resistance: the difficulty of finding two different messages that hash to the same hash. I.e., it is computationally infeasible to find two messages $x, x'$, such that $x \neq x'$ but $H(x) = H(x')$.

2. Pre-image resistance: the difficulty of finding a message itself given its hash. I.e., given the hash $h$, it is computationally infeasible to find the message $x$ such that $H(x) = h$.

3. Second pre-image resistance: the difficulty of finding a second message, given the first message, where they generate the same hash. I.e., given $x$, it is computationally infeasible to find $x'$ such that $x \neq x'$ but $H(x') = H(x)$.

B. Hash Function MD5 and Simplified MD5

MD5 is an iterative hash function, designed by Ronald Rivest of MIT. MD5 uses Merkle-Damgard construction. The block size $s$ equals 512 bits, and hash size $t$ is 128 bits. An arbitrary length message $M$ is padded and broken up into multiple blocks of 512-bit. The underlying compression function, a non-linear function, created by logical XOR, AND, OR, NOT, and SHIFT operations, compresses a 512-bit into a 128-bit output string stored in four 32-bit words $A, B, C$ and $D$. The last iteration for the last message block generates the hash for the entire message $M$.

Let $H$ be hash function MD5, $h$ be its hash, i.e. $H(M) = h$. $h$ is a 128-bit string, is represented in four 32-bit words $A, B, C$, and $D$, then the hash $h$ is

$$ h = A||B||C||D. \quad (1) $$

Let us define MD5 quarter- / half-hash hash function $H_4 / H_2$, which takes an arbitrary length message $M$ and hashes it to a 32- / 64-bit hash $h_4 / h_2$; i.e., $H_4(M) = h_4$ and $H_2(M) = h_2$. The size of $h_4 / h_2$ is a quarter / half of MD5 hash. Both $h_4$ and $h_2$ are defined from MD5’s original four words as follows. We call $H_4$ and $H_2$ simplified MD5 hash functions. In Eq. 2, $\oplus$ is bitwise XOR.

$$ h_4 = A \oplus B \oplus C \oplus D, \quad h_2 = (A||B) \oplus (C||D). \quad (2) $$

C. CUNY HPC Facilities and Libraries

In the CUNY HPC (City University of New York High Performance Computing) system we use Penzias, which is a distributed memory cluster with the following configuration:

- Job Mix: 128 or fewer cores
- Nodes: 60, Cores per node: 12
- Memory per node / core: 48 GB / 4 GB (gigabyte)
- Chip type: Sandy Bridge 2.2 GHz
- Modules and libraries: Portable Batch System(PBS), Intel C Compiler, and RSA MD5\(^1\) source code in C

III. METHODS FOR FINDING MD5

A. Pigeonhole Principle and Birthday Attack

The pigeonhole principle states that if there are $N$ containers for $N + 1$ objects, at least one container will hold two objects [1]. Similarly, for a cryptographic hash function with a 128-bit hash, in MD5 case, a collection of $2^{128} + 1$ unique messages will guarantee that at least two of those messages have the same hash. Based on the pigeonhole principle, to find the pre-image or second pre-image collision of a given hash, brute force method is used to try all possible combinations. MD5 function will require to try, by average, half of $2^{128}$ hash function calls. A simplified MD5 function requires to try, by average, half of $2^{32}$ calls for a quarter-hash and $2^{64}$ calls for a half-hash.

Suppose $k$ people were interviewed for their birthdays. We label the $k$ people by $0, 1, 2, \ldots, k - 1$, and assume that all birthdays are equally likely among $N = 365$ possible days. Person 0 takes a birthday out of $N$. If all people have different birthdays, then person 1 must have a birthday that differs from person 0; i.e., person 1 takes a birthday on any of the remaining 364 days. Similarly, person 2 takes a birthday on any of the remaining 363 days, and so on. The probability\(^2\) of at least two of the $k$ people having the same birthday is complementary to all $k$ birthdays being different; it can be calculated in the following way:

$$ Pr(k) = 1 - \frac{365}{365} \cdot \frac{364}{365} \cdots \frac{365 - k + 1}{365}. \quad (3) $$

Setting $Pr(k)$ in Eq. 3 to 0.5, we find $k = 23$, roughly $\sqrt{N}$. What do birthdays have to do with cryptographic hash functions? Suppose a hash function $H$ produces a hash that is $t$ bits long. Then there are $N = 2^t$ different possible hashes. Suppose that all hashes are equally likely. Since $\sqrt{N} = 2^t/2$, the birthday problem implies that there is a 50% chance of finding a collision if $2^t/2$ different messages are hashed [7].

The birthday attack is used to reduce the number of trials tested in finding collisions. The MD5 function requires to try $2^{64}$ hash function calls. For MD5 half- and quarter-hash, the trials are $2^{32}$ and $2^{16}$, respectively. Comparing to brute force, the birthday attack only needs $\sqrt{N}$ hash trials for finding a collision with over 50% probability.

\(^1\)https://people.csail.mit.edu/rivest/Md5.c
\(^2\)https://en.wikipedia.org/wiki/Birthday_problem
B. Algorithm Description

1) Hexadecimal bit string for second pre-image collisions

A second pre-image collision is to find a second message, given a first one, where their hashes are the same. To keep matters simple, a 64-bit binary number - 0x1234567890abcdef - was chosen as the first message, \( m_1 \). The finding of a second message, \( m_2 \), consisted of incrementing the previous binary number (starting from the 64-bit binary number 0) until a collision of the increment’s hash against the first message’s hash was found, see Fig. 1. We use Algorithm 1 to find full-collisions for hexadecimal bit strings.

<table>
<thead>
<tr>
<th>Message</th>
<th>Half-hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1234567890abcdef</td>
<td>e47e443e0fedecca</td>
</tr>
<tr>
<td>0x0000000000000000</td>
<td>e8807f799c787a74</td>
</tr>
<tr>
<td>0x0000000000000001</td>
<td>0fe1f46f701ad4ad</td>
</tr>
<tr>
<td>0x92020030a154193d</td>
<td>e8807f799c787a74</td>
</tr>
<tr>
<td>0x0000000000000001</td>
<td>0fe1f46f701ad4ad</td>
</tr>
</tbody>
</table>

Figure 1: Second pre-image brute force attack

To monitor the progress of finding a collision, the Hamming distance - number of bits which differ between two binary strings - of the increment’s hash against the first message’s hash was calculated, then updated for every next best Hamming distance found. In the following algorithms, \( d(h_0, h_i) \) represents Hamming distance between \( h_0 \) and \( h_i \).

The separation of workload for this algorithm was dependent on the number of processors being used. The workload was equally divided among the processors with each processor receiving its specific domain of inputs to hash. For example when 4 processors were assigned to the job, the task was divided into equal four segments. All inputs beginning with the most significant byte from 00 to 3F, 40 to 7F, 80 to BF, and C0 to FF were assigned to processor 0, 1, 2, and 3, respectively. In this way the program did not need modification as long as the number of processors was a power of 2 (from 1 to 256).

2) Character string for second pre-image collisions

While using 128-bit binary inputs is a perfectly viable method in finding hash collisions (see Fig. 1 and Algorithm 1), a more common input for hash functions would consist of character strings. For finding a character string second pre-image collision, the first message was a sequence of non-repeating ASCII characters. Permutations of a modified first message were hashed in order to find the second message whose hash would collide with the first’s.

This algorithm’s workload was divided among different processors according to their rank. To achieve workload balance, which allows that each processor has roughly an equal amount of work, the character of the first message located at the index equaling a processor’s rank was replaced with the last character in the array, creating a modified version of the first message. See the following example for illustration. Then each processor starts their permutation with their modified first message. Because each processor has a unique non-repeating (but the same length) of the first message to permute to get their second message, and therefore hash the second message, the workload balance is achieved. Algorithm 2 is for finding half-collision for character strings.

Example:

1st message: a b c d e f g h i
Array index: 0 1 2 3 4 5 6 7 8

Processor Rank Inputs
0   0   i   b   c   d   e   f   g   h
1   1   a   i   c   d   e   f   g   h
2   2   a   b   i   d   e   f   g   h
3   3   a   b   c   i   e   f   g   h

Data: Character Strings
Result: Updated closest Hamming distance

first message initialization
\( h_0 = H_2(m_0); \)  //first message hash
for each processor do
initialize Hdistance = 64
for each string permutation do
\( h_i = H_2(i); \)
if \( d(h_0, h_i) < Hdistance \) then
update Hdistance
end
end

Algorithm 2: Finding half-hash collisions

3) Hexadecimal bit string collisions by birthday attack

The birthday attack algorithm is for finding two different messages that hash to the same hash rather than finding a second message that collides with a specific target.

While finding hexadecimal bit string collisions using the brute force method was successful, a different algorithm...
performing the birthday attack was implemented in order to improve running time. However, compared to the second pre-image - which required constant memory usage of \( O(1) \) - the birthday attack algorithm’s downside was its greater amount of memory needed, which is \( O(n) \).

In finding a MD5 quarter-hash collision by birthday attack using a domain of all possible 4 byte inputs, each iteration of a message and hash required 8 bytes at minimum which amounted to 32 GB in total. Since each core in the Penzias cluster has access to only 4 GB memory, the development of the program was undertaken very conservatively with small increments in the input domain. One significant hindrance to the development was the amount of time it took to wait in queue for jobs that required 16 or more cores.

Each processor was assigned its own unique input domain and an output range of hashes to be sorted and divided into predetermined sections for sending and receiving between processors. The simple quick-sort algorithm proved to be fast in the initial sorting of the hashes due to the random nature of the cryptographic hash function. Then a binary search was utilized to mark the positions of the sorted hashes at the predetermined sections to be shared.

Those positions were shared using the MPI_alltoall() function; they were the information needed to acquire the parameters for the MPI_alltoallv() function (vector version), which was utilized for the actual transmission of the sections of sorted hashes. This ensured that each processor had a portion of the total output in memory that could not have a collision with a portion in the memory of a different processor.

After receiving its portion of the hashes, each processor sorted its new list using the merge sort which proved to be much faster than quick sort for this already partially sorted list. After the second sort, collisions were checked and reported by comparing each hash with its neighboring hash sequentially. Please see Algorithm 3 for the details.

4) Meaningful passage collisions by birthday attack

While finding hash collisions using sequential numbers or permutations of ASCII characters as inputs have their academic and password security checking values, collisions between longer meaningful passages will show a more realistic result in the difficulty of exploiting a cryptographic hash function. Algorithm 4 is for finding meaningful passage collisions.

The following implementation uses synonyms in certain key words to generate exponentially multiple combinations of paragraphs that have the same meaning. For example:

- Unless differently / otherwise specifically said / indicated in this Will, any inheritance / provision for my children includes / considers the following children...

The above paragraph has 4 words with 2 synonyms each, which can make \( 2^4 = 16 \) unique combinations. Likewise, to generate 4 billion unique messages that have the same meaning, only 32 pairs of synonyms are required.

**Algorithm 3: Finding quarter-hash collisions**

```plaintext
Data: Hexadecimal strings \( m_i \), and its hash \( h_i \)
Result: Two hexadecimal strings with the same hash

Total_domain
Processor_domain = Total_domain / # of processors

for each processor do
    local_list
    for each \( m_i \) in Processor_domain do
        \( h_i = H(m_i) \)
        \( m_i, h_i \to \) local_list
    end
    quicksort(local_list)
    MPI_Alltoallv(local_list)
    mergesort(local_list)
    for each \( h_i \) in local_list do
        if \( h_i = h_{i+1} \) then
            collision found
        end
    end
end
```

In the following digital signature exploit example, a collision between passages of different meanings are required. Two lists are generated: one with a will specifying Alice as the beneficiary and another specifying Edgar as the beneficiary. If Edgar can find a matching pair from the lists that maps to the same hash, and provided he can get his victim to digitally sign the first passage, it would be equivalent to getting a signature on the second passage.

**Algorithm 4: Finding meaningful passage collisions**

```plaintext
Data: Paragraphs
Result: 2 colliding paragraphs \( P_1 \) & \( P_2 \)

seed() : binary integer, 1:1 with all paragraphs
\( P_1() \) // Paragraph for Alice/Bob
\( P_2() \) // Paragraph for Edgar
\( List_1 \) // consisting of \( P_1()\)'s
\( List_2 \) // consisting of \( P_2()\)'s

for each paragraph Generating_seed do
    put \( P_1() \) and its hash in \( List_1 \)
    put \( P_2() \) and its hash in \( List_2 \)
end

sort(List_1) and (List_2) by hash value

for each smallest item in List_1 & List_2 do
    if same hash is found in corresponding list then
        report collision
    else
        discard paragraph with the smaller hash
    end
end
```
Table I: Full-hash collision, target 0x00...0 (128)

<table>
<thead>
<tr>
<th>Processor</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>2016-7-02 at 03:48:09, Hamming distance: 67</td>
</tr>
<tr>
<td></td>
<td>Hexstring: 000000000000000000000010000000</td>
</tr>
<tr>
<td></td>
<td>Hash: 765f8817e81795086e5e166c1e936c0e</td>
</tr>
<tr>
<td>14</td>
<td>2016-07-02 at 03:48:09, Hamming distance: 64</td>
</tr>
<tr>
<td></td>
<td>Hexstring: 0000080e000000000000000001000000</td>
</tr>
<tr>
<td></td>
<td>Hash: 515de995200d1812b121473c392100e</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>2016-07-22 at 01:34:31, Hamming distance: 21</td>
</tr>
<tr>
<td></td>
<td>Hexstring: 0000805000000000e902000012a695f6</td>
</tr>
<tr>
<td></td>
<td>Hash: 43c75732864bd9177951752e226810e7</td>
</tr>
</tbody>
</table>

Table II: Half-hash collision, target 0x1234567890abcdef, its half-hash e47c443e0fedeca

<table>
<thead>
<tr>
<th>Processor</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2016-07-15 at 18:23:28, Hamming distance: 40</td>
</tr>
<tr>
<td></td>
<td>Hexstring: 000000000010000000</td>
</tr>
<tr>
<td></td>
<td>Half-hash: 0fe1f6f701ad4ad</td>
</tr>
<tr>
<td></td>
<td>Hexstring: 0000000d0080000000</td>
</tr>
<tr>
<td></td>
<td>Half-hash: e790f468cb9885a</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>2016-07-29 at 12:53:58, Hamming distance: 4</td>
</tr>
<tr>
<td></td>
<td>Hexstring: 92020030a154193d</td>
</tr>
<tr>
<td></td>
<td>Half-hash: e47ac43e0fed6cc2</td>
</tr>
</tbody>
</table>

Table III: Quarter-hash collision, target 0x00...0 (128)

<table>
<thead>
<tr>
<th>Processor</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2016-07-14 at 03:44:28, Hamming distance: 14</td>
</tr>
<tr>
<td></td>
<td>Hexstring: 01000000000000000000000000000000</td>
</tr>
<tr>
<td></td>
<td>Quarter-hash: c274278c</td>
</tr>
<tr>
<td>9</td>
<td>2016-07-14 at 03:44:28, Hamming distance: 9</td>
</tr>
<tr>
<td></td>
<td>Hexstring: 1b000000000000000000000000000000</td>
</tr>
<tr>
<td></td>
<td>Quarter-hash: e02aaa6cb</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>2016-07-14 at 03:44:58, Hamming distance: 1</td>
</tr>
<tr>
<td></td>
<td>Hexstring: fe9f9354</td>
</tr>
<tr>
<td></td>
<td>Quarter-hash: f03208cb</td>
</tr>
<tr>
<td>0</td>
<td>2016-07-14 at 03:45:56, Hamming distance: 0</td>
</tr>
<tr>
<td></td>
<td>Hexstring: 000f1e09</td>
</tr>
<tr>
<td></td>
<td>Quarter-hash: f03208c9</td>
</tr>
</tbody>
</table>

IV. EXPERIMENT RESULTS

In this section we will show some results obtained from the algorithms described in Section III-B.

A. Hexadecimal bit string second pre-image collisions

The first message (i.e., target) is set as 128 binary 0’s (0x00...0) to run Algorithm 1. Passing the input to the MD5 hash function returns a hash of 4ae71336e44bf9bf79d2752e234818a5. While the target could be any possible bit string, setting it to the smallest made the code programming simpler and ensured that false positives (identical hashes due to identical inputs) were easily avoidable.

Running the initial code on personal laptops gave discouraging results. While the processors may be able to run instructions in the order of $2^{32}$ every second, it initially took around 15 minutes to run an MD5 hash function through that many 128-bit hexadecimal binary strings. The best Hamming distances in that time frame were in their low 30’s. Compiling the code using aggressive optimization proved valuable in shortening the running time while providing identical results.

The target Hamming distance for the whole project was 11, and rudimentary arithmetic thinking suggests if that target could be attained using a birthday attack, than a similar second pre-image attack should be able to achieve around 22.

Due to space limitations, we can only give some result excerpts by using 16 processors on Penzias. Tables I, II, and III show that given a target hexadecimal string, their second pre-image (near-) collisions are obtained for full-, half-, and quarter-hash MD5.

A representation of the quarter-hash collision running time based on the number of processors used is shown in Fig. 2.

B. Hexadecimal bit string collisions by birthday attack

The biggest challenge in running a successful birthday attack on a hash function is the memory requirements in storing all the different inputs and their hashes. Not only did the input domain have to be split as previously, but it also had to make the most use of each processors’ available memory. To ensure collisions between processors were not missed, each processor first sorted its own local list, split its output list by ranges that were automatically determined by the number of processors used, and sent and received its assigned output space.

Assuming the MD5 hash function is a good pseudo-
random function, in theory the expected number of birthday collisions is determined using the following formula,

\[
\text{No.Collisions} \approx \frac{\text{InputStates}^2}{\text{OutputStates}} / 2.
\]  

We test these properties on the birthday attacks of MD5 quarter-hash collisions for hexadecimal bit string by birthday attack. Our results almost overlap the theoretical values in Eq. 4, see Fig. 3.

### C. Meaningful passages collisions by birthday attack

Using the method described in Section III-B 4), we were able to find the following two passages collided for MD5 quarter-hash; their quarter-hash is a9e2a7f6.

At the time of reading this Will, I am married to Jane Doe. My children are as follows. Unless differently specifically said in this Will, any inheritance for my children includes the following children, as well as any child of mine hereafter born or adopted: Alice and Bob Doe. I generally, intentionally and with full knowledge fail to provide for Edgar Doe in this Will.

At the time of executing this Will, I am married to Jane Doe. The names of my children are as follows. Unless differently specifically said in this Will, any inheritance for my children includes the following children, as well as any child of mine hereafter born or adopted: Alice and Bob Doe in this Will.

### V. Conclusions and Future Work

Based on the original MD5, we created half- and quarter-hash MD5 formulas in our paper for collision searching use. The major contributions are the four algorithms, namely algorithms for hexadecimal bit string / ASCII character string second pre-image full-, half-, and quarter-hash collisions by brute force; birthday attack algorithms for finding hexadecimal bit string / ASCII character string quarter-hash collisions; and birthday attack algorithm for finding meaningful passage quarter-hash collisions. We designed a method to generate exponentially many different passages that have the same meaning. In the experiments, we chose the number of processors as a power of two, this makes the distribution of workload equally among processors easier. In the future we will investigate tree structured merge sort for saving memory and the communication cost among processors.

### Acknowledgments

This work is supported, in part, by NSF/DoD REU Site\(^3\) (Award No.: 1359266) at CUNY-College of Staten Island. We want to thank CUNY HPC Center\(^4\) for providing MPI training and their facilities.

### References


\(^3\)http://www.cs.csi.cuny.edu/REU
\(^4\)https://cunyhpc.csi.cuny.edu