Password Recovery Using MPI and CUDA

David Apostal*, Kyle Foerster†, Amrita Chatterjee*, Travis Desell*
*Department of Computer Science, University of North Dakota
david.apostal@my.und.edu, amrita.chatterjee@my.und.edu, travis.desell@cs.und.edu
†Department of Electrical Engineering, University of North Dakota
kyle.foerster@my.und.edu

Abstract*—Using passwords to verify a user's identity is the most widely deployed method for electronic authentication. When system administrators need to recover lost passwords or test accounts for easily guessable passwords, it can require millions of hash function and string comparison operations. These operations can be computationally expensive but are easily parallelizable because each password can be tested independently. Therefore, using high performance computing (HPC) can greatly reduce the time required to perform password recovery or analysis. Due to the high level of fine-grained parallelism of this type of problem, GPU computing using Compute Unified Device Architecture (CUDA) can be used to further improve performance. The scale of HPC can be further increased through the use of multiple GPUs, but this requires communication between the GPU devices and can reduce the overall performance due to increased communications latency. In this work, a well established HPC framework, MPI, was used to minimize the amount of latency and handle the communication between the devices. This allowed for a course-grained division of the problem using MPI where each device applies a fine-grained division of the problem using CUDA to perform the actual calculations. This paper describes three dictionary-based password recovery algorithms that use both Message Passing Interface (MPI) and CUDA. In this approach the hashed values of known words are computed and compared with hash values of unknown user passwords. The algorithms differed in GPU memory utilization, and how the data was divided and distributed among the MPI nodes and GPU devices. A *divided dictionary algorithm* split the dictionary of potential passwords over the GPUs and copied the password database to each GPU. A *divided password database algorithm*, split the password database and copied the potential passwords. A *minimal memory algorithm* split the password database, and sequentially processed individual passwords on the GPUs. The divided dictionary and the divided password database algorithms performed well, resulting in a speedup of 17x and 12x over a single processor using 8 GPUs across 4 compute nodes, respectively. Illustrating the cost of communication latency between MPI nodes and GPUs, the minimal memory algorithm performed significantly slower than a single CPU. The algorithms are shown to scale well to multiple GPUs, so this password recovery system could be used for much larger systems for larger databases. In addition to recovering lost passwords, this work could be used to help improve the security of computer systems by identifying accounts with weak or common passwords. The framework described may also be useful for other research that needs to process large amounts of data with similar characteristics using MPI and CUDA.

I. Introduction

Electronic authentication is the process of confirming that a user identity presented to an information system. A widely deployed method for confirming a user identity is based on passwords. A person submits her userid and password to a computer system. The computer system processes the password with a hash function and compares the hash function output with a hash value associated with that userid from a password database. If the two hash values match, she is granted access to the system. Millions of people follow this protocol to confirm their identity multiple times each day.

The hash functions used to authenticate users are called cryptographic hash functions. These functions read a string of characters of any length, perform a hash operation on the string, and then return an encoded, hashed value as a fixed-length string. A good cryptographic hash function is one-way, which means the original input string cannot be recovered by reversing the function. This one-way characteristic has led to passwords being stored as hashed values instead of in their plain-text format; so even if the password database is compromised, the plain-text passwords are still somewhat secure.

One problem with password authentication is that some passwords are weak; they contain little randomness from one character to the next. When people create their own passwords, it is natural for them to choose words that are easy to remember. Passwords are sometimes based on familiar names, common words, or are simple sequences like "12345." Those types of passwords are easily guessed by attackers.

A dictionary-based password recovery application [1]–[5] performs a large number of hash and string comparison operations. A password dictionary is filled with familiar names, common words, and simple character sequences. With a large dictionary or a large number of entries in a password database, password recovery is computationally expensive. However, these types of analysis have a high level of parallelism; testing one word from the dictionary is independent of testing another. This makes dictionary-based password recovery well suited for a HPC environment, like MPI [6] or CUDA [7]. The parallelism can be taken even further and implemented in a hybrid MPI+CUDA environment, which allows for an initial course-grained division of the problem using MPI and a fine-grained division of the problem using CUDA.

In the realm of password recovery there are several approaches that have been deployed [1]–[4], [8]–[11], each with their own advantages and disadvantages. These approaches are discussed further in Section II.

In this paper we compare the memory usage and run times for multiple password recovery algorithms. We designed and implemented algorithms using an MPI+CUDA hybrid approach to password recovery; with each algorithm based around the dictionary attack method of password recovery. This MPI+CUDA approach allows for the division of a large dictionary or a password database between multiple MPI nodes, which is then further divided among the GPUs of each node. This allows for multiple GPUs to perform calculations at the same time. An analysis of the three algorithms was done to identify the different strengths and weaknesses with respect to execution time and memory usage and help determine the most efficient data distribution strategy for a hybrid MPI+CUDA computing environment.

These password recovery algorithms have three high-level steps:

I. Distribute the dictionary and user password data to MPI nodes and GPU devices.
II. Calculate hash values for the dictionary words.
III. Compare the calculated hash values of dictionary words with the hash values from a password database file.

This processing is similar to the general processing required in some other fields of research:

I. Distribute the data.
II. Optionally transform the data.
III. Perform pair-wise operations on the independent data.

A similar hybrid approach to HPC may be useful in other domains with large amounts of data.

Our algorithms are distinguished by how they distribute data to the GPU devices. The algorithms are:

- A *divided dictionary algorithm* which divides the dictionary among the MPI nodes. Each MPI node further divides the dictionary among the available GPU devices. The password database is copied to all GPU devices.
- A *divided password database algorithm* which divides the password database among the MPI nodes. Each MPI node divides its portion of the database among the available GPU devices. The entire dictionary is copied to all GPU devices.
- A *minimal memory algorithm*, similar to the divided password database algorithm, where each GPU gets a unique subset of the password database. However, instead of loading the entire dictionary into each GPU's memory, the minimal memory algorithm ensures low memory usage by processing a single dictionary word at a time.

Although each algorithm recovered the same number of passwords, results from experiments showed distinct differences in program execution times and memory usage. Compared to a single processor, using 8 GPUs across 4 compute nodes, the divided dictionary algorithm was 17x faster, the divided database algorithm was 12x faster and the minimal memory algorithm was 7x slower. The divided dictionary and divided password database algorithms were also shown to scale well to multiple GPUs. The divided dictionary algorithm used 98% more GPU memory on average than the minimal memory algorithm. The divided password database algorithm used 87% more GPU memory on average than the divided dictionary algorithm.

These variations among the algorithms are further described in the sections III and IV.

II. Related Work

There has been significant research in the areas relating password strength to system security, improving the performance of password recovery systems, as well as high performance distributed computing approaches to a variety of problems including password recovery.

A. Passwords and System Security

One area of research focuses on improving the strength of systems that use password-based authentication protocols. System strength can be improved by understanding what makes one password better than another password. Dell'Amico *et al.* measured password strength in terms of the search space required for attackers to guess some of the passwords [2]. It has been shown that dictionary attacks are most effective at discovering weak passwords; dictionary word mangling is useful after the dictionary has been fully tested. Finally, Markov-based techniques were most useful in breaking strong passwords [2].

Another approach for improving system strength involves employing policies appropriate to the operating environment. Users in different environments face different threats and have different security needs. The collection of policies used in a small office may be very different from the policies used in a security-conscious business or an environment with highly classified government data. It is reasonable that password complexity and password aging policies are different in these environments. Teat *et al.* analyzed each policy set's resistance to an attacker using different computing resources: a single personal computer, a botnet, and cloud-based resources [12]. The effective security on each policy set was also compared against the attacker's cost of using each resource or how long each resource would take to crack a password.

B. Serial Dictionary Recovery Systems

Some researchers have studied ways to improve the performance of serial password recovery systems.

A larger dictionary generally increases the likelihood of recovering one or more passwords. However, this also increases the number of hash and string compare operations to perform. Weir *et al.* developed a system to generate dictionary word lists based on the probability distribution of user passwords [1]. After analyzing an appropriate set of training passwords, the system determines the probabilities of grammar production rules and the probabilities of special characters and digit characters. This approach allows one to generate dictionary words based on the probabilities of observed patterns of passwords. The result is a probability-based ordering of dictionary words and word-mangling templates.

John the Ripper (JtR) is an open-source software package designed to attack (or crack) passwords in a number of formats such as MD5 and SHA1. It is a useful tool for recovering forgotten passwords and for detecting weak passwords [5]. One attack mode JtR supports is dictionary attack.

Markov chains have been shown to perform well compared to JtR and brute force exhaustive search techniques for password recovery [10]. Markov chains can be further improved using various optimization techniques described by Marechal [10]. Though the results found that JtR was faster, the Markov chain process is more successful in cracking passwords. The Markov process runs for a specified time and it can be easily distributed. But, JtR does not have a predefined time limit and can run until all the passwords are recovered.

C. Course-Grained HPC

In the field of high performance computing there are multiple tool kits available for use such as Message Passing Interface (MPI) and OpenMP. This section describes work to improve the performance of MPI on grid systems as well as efforts to utilize course-grained HPC technologies in the area of password recovery.

Foster *et al.* describe an implementation of high performance computational grids in different domains [13]. MPICH-G is a grid-enabled implementation of MPI. Several problems were encountered in the development of the MPICH-G. For example, a user cannot have same user ids at the two sites [13]. It is essential to develop grid-enabled tools in order to eliminate the limitations. It is stated that MPICH-G can be distributed broadly. Another grid-enabled MPI implementation is constructed from MPICH and Globus. MPICH has been implemented in a variety of platforms as the architecture of MPICH features a layered design [13]. The interrelation of the components of the Globus toolkit made this toolkit to be used widely.

Performance of password recovery can also be improved by increasing the number of processors used. Bernaschi *et al.* used a loosely coupled architecture based on volunteer computing to implement a dictionary attack against a private key ring generated by GnuPG software [3]. The architecture organizes heterogeneous volunteer nodes into levels. The first level has a single "root" node. The second and third levels are organized into working groups. Nodes in each level have specific responsibilities.

Pellicer *et al.* used Berkeley Open Infrastructure for Network Computing (BOINC) on a small test bed to search possible five-character passwords hashed with the MD4 algorithm [14]. They observed less than linear increases in performance due to work unit scheduling and network overhead.

JtR has been integrated with Message Passing Interface (MPI) [4] and with BOINC [9]. BOINC-based password recovery application may be further improved through the use of a task server to handle the distribution of work as further described in Anderson *et al.* [15].

Sykes *et al.* used JtR and MPI to create two algorithms with different goals [4]. In their first algorithm a single password to be cracked is distributed to each MPI nodes and the dictionary is carefully divided among the nodes so that each node is equally likely to crack the password. The second algorithm divides a large password database file among the MPI nodes. If any node is underutilized the system can reallocate the uncracked passwords.

D. Fine-Grained HPC

MPI and OpenMP provide an efficient way of coarsely dividing work up between multiple computers, cores, or threads where each individual unit performs calculations on a subset of the data. On the other hand, CUDA is better suited to a more fine-grained parallelism of the problem as further demonstrated in Pennycook *et al.* [16], where multiple versions of the LU benchmark are compared on multiple architectures.

Hash function performance has also been the focus of some research. Keccak-f[1600] is a finalist in the NIST SHA-3 competition. Cayrel *et al.* described results of implementing a version of the Keccak-f[1600] hash function for GPU devices [17]. The Keccak family of hash functions is based on a sponge construction which consists of two phases: an absorbing phase and a squeezing phase. The absorbing phase reads a message to be hashed. The squeezing phase outputs the hash value. The input phase of the Keccak function was found to be largely a sequential process. This limited the number of parallel threads that the Keccak function could use.

Cayrel *et al.* also described their results using an alternative GPU implementation called leaf interleaving [17]. In leaf interleaving GPU kernels are organized like leaves of a tree. A message to be hashed is divided among the kernel leaves. After a leaf hashes its portion of the message, the hash result is passed to the leaf node's parent node.

A table of precomputed hash values used for password recovery is called a rainbow table. Brute force exhaustive searches and the generation of rainbow tables are well-suited for different types of HPC. This is explained further in Gomez *et al.* [18] where a comparison between a sequential version and three multiprocessing methods; threaded, MPI, and CUDA, of both tasks were run using the MD5, SHA-1, and NTLM/MD4 hash algorithms. These results showed that the Rainbow table generation performed the best in an MPI environment, while the brute force attack performed the best in the CUDA version. CUDA implementation of password recovery with Rainbow tables is further described by Graves [8].

E. Hybrid HPC

Course- and fine-grained HPC have normally been used independent of each other. However, they may be combined as a hybrid framework of parallelism. A course division of the problem is done at the CPU level, using MPI or OpenMP. This level divides the data or calculations into smaller subsets, handling distribution of data and gathering of results. At the next level GPUs can then apply a fine-grained type of parallelism to the smaller subset of the problem [19]. This type of abstract parallelism has lead to implementations of MPI+CUDA hybrid programs becoming more common as demonstrated by an analysis of the performance difference between an OpenMP and MPI+CUDA program to perform two-layer shallow water systems numerical modeling using a first order Roe type finite volume scheme [11], which demonstrated the advantages to using a MPI+CUDA hybrid approach over an OpenMP approach and a discontinuous Galerkin Time Domain analysis performed in Dosopoulos *et al.* [20].

One important factor to take into account when attempting to create a parallel version of a problem or simulation is making sure the underlying calculations or algorithms used have a high degree of parallelism as demonstrated in Karunadasa *et al.* [21] using GPU clusters. The need to use a highly parallel algorithm or calculation is further demonstrated by the comparison between the classic Roe mathematical model and an improved Roe scheme, IR-Roe, as described further by Asuncion *et al.* [11] using MPI and GPUs. The changes made to their underlying model resulted in significant speed-ups when compared to the old model. This comparison demonstrated that when porting a problem to an HPC solution it is necessary to take the hardware or tool kits performance limits into consideration. Modifications to the mathematical model or algorithm can result in even larger gains after implementation on an HPC framework, as demonstrated between the classic Roe model and the improved IR-Roe model.

III. Approach

In order to determine the most efficient use of a hybrid MPI+CUDA system for password recovery, three different algorithms were developed. The algorithms are similar in that they are based on a dictionary approach to password recovery. In the dictionary approach hash values for a list of words are computed. Those computed hash values are then compared against the hash values stored in a password database. If the computed hash value for a dictionary word matches the stored hash value for an account in the database, then the dictionary word is the password for the account.

The algorithms differ primarily in how the dictionary and password database are distributed among the GPU devices. The remainder of this section will describe the hash function used, our dictionary and password database data, and the three algorithms.

A. Hash Function

The cryptographic hash function used in this project is a version of SHA-1 implemented in C and obtained under a Freeware Public License [22]. For the CPU version of the program, a driver program was written to implement and use the SHA-1 C code obtained from Jones *et al.* [22].

In the MPI+CUDA version of the password recovery program, the SHA-1 C code was ported over as device functions to allow for their usage within the CUDA kernel used to generate the dictionary word hashes.

B. Input Data

The input data for each algorithm consists of a dictionary of potential passwords and a password database. The dictionary is comprised of word lists from multiple online sources. Some word lists are purported to contain actual passwords aggregated from postings on various Internet forums. One word list is simply a list of English words; one is a list of German words; and one is a list of US cities.

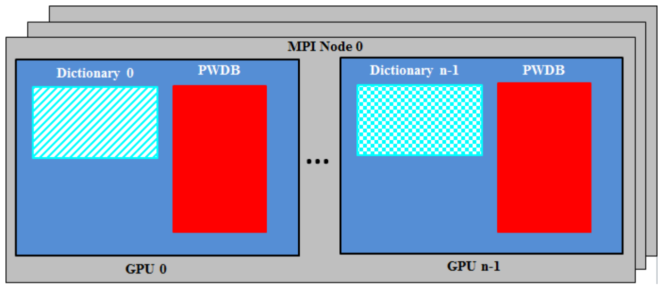

Fig. 1. Divided Dictionary Algorithm. The dictionary of potential passwords is divided among the MPI nodes. Each MPI node subdivides its words among its available GPU nodes. The same password database is loaded onto each GPU.

The password database loosely resembles an `/etc/passwd` file on some Unix systems. Each record, representing one account, has two fields. The first field holds the account user name. The user names in the file are unique. The second field stores the hash value of the user's password. The hash values were set such that a known number of accounts had passwords known to be in the dictionary. The remaining accounts had passwords known not to exist in the dictionary. This aided in verifying the correctness of each algorithm.

C. Algorithms

Our hypothesis was that each algorithm will require different amounts of memory and processing time due to their differences in how they divided the dictionary and password database data. The scalability of the different algorithms was also of interest. Section IV examines their scalability, performance and memory usage in detail.

1) Divided Dictionary Algorithm: The divided dictionary algorithm (shown in Figure 1) divides the larger dictionary file between the MPI nodes which then further divide the dictionary words between the GPUs on each MPI node. Each GPU has a different portion of the dictionary. The full password database file is then loaded onto each GPU. A kernel function is repeatedly called for each of the user accounts to test for matches between the password database hash and the dictionary word hashes. Any matches represent the recovered password for the user account. The main steps of this algorithm are described below.

I. Evenly divide the dictionary words among the MPI nodes. Each MPI node further divides its words among its available GPU devices.
II. The GPU devices calculate hash values for each of its dictionary words and stores them in global memory.
III. The contents of the password database are copied to each GPU device. Each GPU has the same account data.

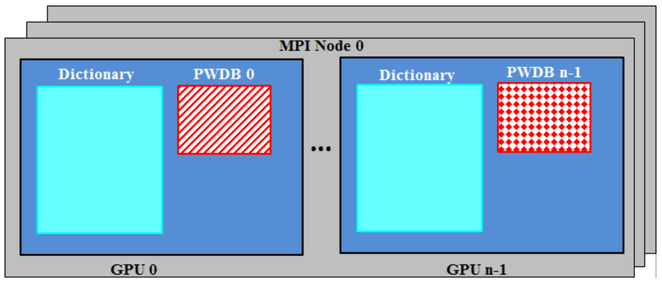

Fig. 2. Divided Password Database Algorithm. The password database is divided among the MPI nodes. Each MPI node subdivides that user account data among its available GPU devices. The entire dictionary is loaded onto each GPU.

IV. Finally, the GPU tests for matches between the calculated hash value of each dictionary word with the stored hash value from the password database.

2) Divided Password Database Algorithm: The divided password database algorithm (shown in Figure 2) evenly divides the password database file between the MPI nodes which then further divide the user account data between the GPUs on each MPI node. The full dictionary file is then loaded onto each GPU and their respective hash values are calculated. A kernel function is then repeatedly called for each of the user accounts to test for matches between the password database hash value and the dictionary word hashes. Any matches represent the recovered password for a given user account as outlined below.

I. Distribute all of the dictionary words to each MPI node and its available GPU devices.
II. The GPU devices calculate hash values for each dictionary word.
III. Evenly divide the contents of the password database among each MPI node. Each MPI node then divides its account data among the GPU devices.
IV. Finally, the GPU tests for matches between the calculated hash value of each dictionary word and the stored hash value from the password database.

3) Minimal Memory Algorithm: The minimal memory algorithm (shown in Figure 3) repeatedly calculates the hash value for a dictionary word and then tests for a match in the password database. The intent of this algorithm is to be a low memory solution; it differs greatly when compared to the other algorithms. The minimal memory algorithm divides the password database file between the MPI nodes which then further divide the password database between the GPUs on each MPI node. The choice to divide the password database between the MPI nodes instead of the dictionary file is based on the assumption that the password database will be significantly smaller than the dictionary file. The algorithm processes the data as outlined below.

I. Evenly divide the password database among the MPI nodes. Each MPI node further divides its user accounts among the GPU devices.
II. For each dictionary word,

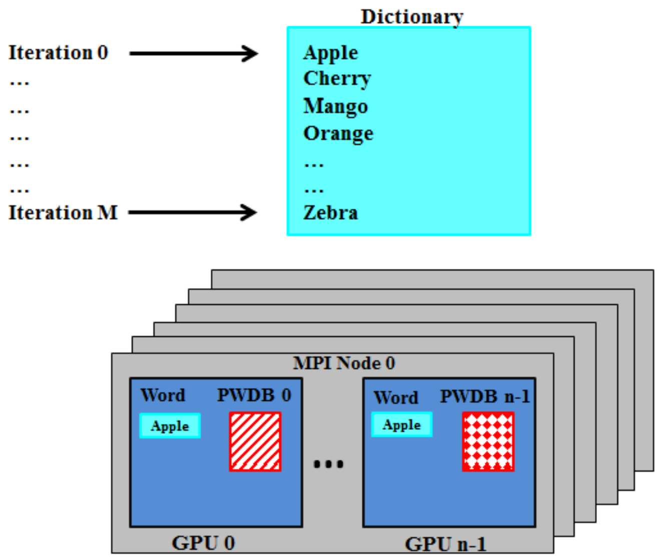

Fig. 3. Minimal Memory Algorithm. The password database is divided evenly among the MPI nodes and then subdivided among the GPU devices. One by one, a single word from the dictionary is loaded onto the GPUs, hashed, and compared with the user account hash values.

a. Load and distributed the word among the GPUs.
b. Calculate the word's hash value.
c. Compare the calculated hash value against the loaded user account hash values.

IV. Results

The MPI+CUDA password recovery program was implemented and tested on a four-node cluster. The MPI implementation was MPICH2 version 1.4.1p1. Each MPI node was an x2200 compute node with dual 64-bit, 4-core processors. Each cluster node had two Tesla C1060 GPUs and CUDA 3.0.

Two observations effected the design and results. First, the dictionaries used for the testing of the password recovery program are expected to be relatively large, and the division of the dictionaries would remain the same between executions of the program with the same number of GPUs. Second, if MPI is used to handle the loading and distribution of the dictionaries between the nodes, it incurs a large amount of network communication latency and causes the program to take a considerably larger amount of time to run. To address this issue, the dictionaries were divided and distributed to the MPI nodes using a synchronous file system. This allowed each individual MPI node to load its associated file and remove the potential network latency associated with distributing data.

Program options specify the number of MPI nodes to use and which data distribution algorithm to execute. There is no option to specify the number of GPUs to use; the program uses as many GPUs as are available on each MPI node. MPI handles the initialization and communication of the MPI nodes as well as any file IO associated with the loading of the dictionary and password database files. The MPI thread on each node also handles the initialization and creation of the GPU threads because CUDA 3.0 requires each GPU to have its own controlling thread to handle initialization, memory operations, and kernel calls. The threads are synchronized to allow for the loading, distribution, and retrieval of data between the MPI thread and GPU threads.

We analyzed each algorithm in terms of scalability, elapsed time and memory usage. Elapsed times are reported for the total execution time for each algorithm as well as average times the GPUs to perform individual steps. The individual steps are: loading the data into GPU memory, computing hash values for the dictionary words, and comparing hash values for dictionary words with hash values in the password database file. Average GPU memory usage is also reported.

A. *Elapsed Time*

In terms of total execution time, the minimal memory algorithm was significantly slower than the other algorithms. The minimal memory algorithm loads a single dictionary word, computes its hash value, and compares that hash value with each value in a subset of the password database. The slow performance is due to repeated CUDA memory operations. Table I shows results of the minimal memory algorithm for different numbers of GPUs. The columns represent the times in seconds to load the password database, processing every word in the entire dictionary, and the total runtime of the program. Processing a word requires loading a word in the GPU, calculating the hash value for that word, comparing the hash value with the hash values in the password database. These tests used a dictionary of 2,151,220 words and a password database of 1,000 accounts.

TABLE I
MINIMAL MEMORY ALGORITHM ELAPSED TIMES (SEC)

GPUs	Load PWDB	Process Dictionary	Total
1	0.000139	683.498	684.794
2	0.000186	717.484	718.952
3	0.000153	664.074	665.472
4	0.000155	662.807	664.306
5	0.000618	659.600	661.015
6	0.000164	660.300	661.725
7	0.000178	656.557	659.335
8	0.000107	653.205	656.291

Metrics for the divided dictionary and divided password database algorithms are shown in Table II and Table III respectively. The tables show the times in seconds to load the dictionary, generate hash values for each dictionary word, compare the dictionary word hash values with hash values in the password database, and the total runtime of the program. These tests used a dictionary of 8,604,880 words and a password database of 1000 accounts. For the divided dictionary algorithm the dictionary data is divided among the MPI nodes. Each MPI node then divides the data among the number of available GPU devices installed on the MPI node. In the divided password database algorithm the entire dictionary is loaded into the memory of each GPU. In tests with three, five, and seven GPU devices one MPI node has only one GPU device while the other MPI nodes have two GPU devices. In these special cases one GPU device performs more work than the other GPUs.

TABLE II
DIVIDED DICTIONARY ALGORITHM ELAPSED TIMES (SEC)

GPUs	Load	Generate	Compare	Total
1	1.435	0.584	23.712	27.037
2	1.421	0.321	12.345	15.406
3	0.701	0.157	6.178	14.392
4	0.699	0.156	6.625	8.869
5	0.466	0.107	4.124	10.120
6	0.468	0.105	3.983	4.556
7	0.353	0.077	3.223	8.349
8	0.353	0.075	3.248	5.191

TABLE III
DIVIDED PASSWORD DATABASE ALGORITHM ELAPSED TIMES (SEC)

GPUs	Load	Generate	Compare	Total
1	1.435	0.581	23.682	27.025
2	1.882	0.579	11.808	15.714
3	1.897	0.584	5.897	15.152
4	1.927	0.589	6.161	10.050
5	1.908	0.590	3.925	11.239
6	1.887	0.584	3.962	8.061
7	1.880	0.587	3.134	9.868
8	1.930	0.581	2.813	7.456

Figures show side-by-side comparisons of the divided dictionary and divided password database algorithms for each metric. Figure 4 shows the total program execution time when each algorithm was run with different numbers of GPU devices. This is the elapsed real time between start and finish times as reported by the unix time command. Results for tests with three, five, and seven GPUs show the effect of data not divided evenly among the GPU devices; one GPU processed twice as much data as the others. In this test, the divided dictionary algorithm performs slightly better than the divided password database algorithm due to the decreased number of hash operations and comparisons. This is largely due to the fact that the dictionary is considerably larger than the password database.

Figure 5 shows the average times to load the dictionary for different numbers of GPU devices. The times for the divided dictionary algorithm trend downward because the number of dictionary words processed by each GPU decreases. For the divided password database algorithm the load times for two or more GPU devices is higher than the load time for one GPU. This is because an extra string copy operation is required for each dictionary word for the second GPU device on an MPI node.

The times to generate hash values for each dictionary word using a SHA-1 hash function are shown in Figure 6. The times for the divided dictionary algorithm decrease because the dictionary is divided among each GPU device. The times for the divided password database algorithm are consistent because each GPU has to calculate hash values for the same number of dictionary words.

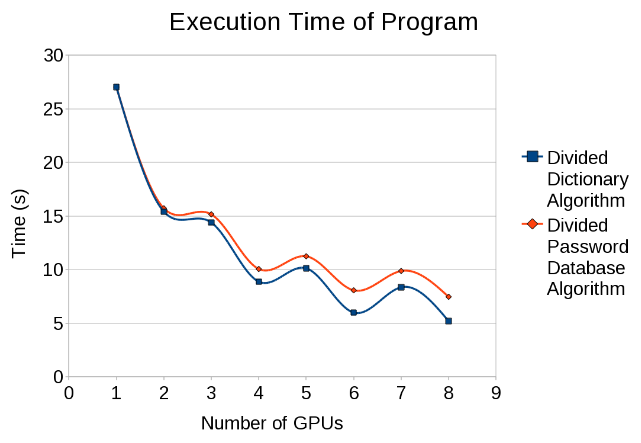

Fig. 4. Execution Time of Program

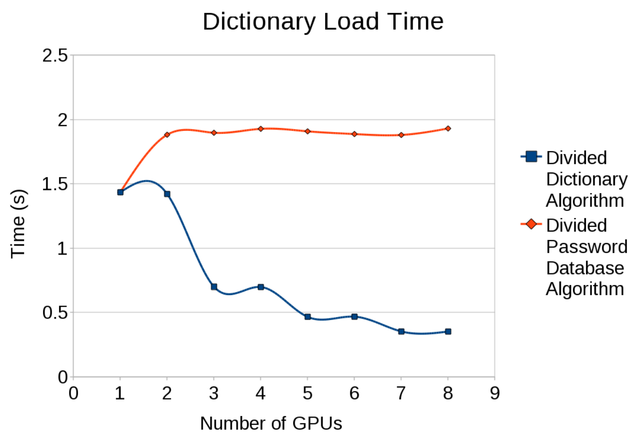

Fig. 5. Average GPU Dictionary Load Time

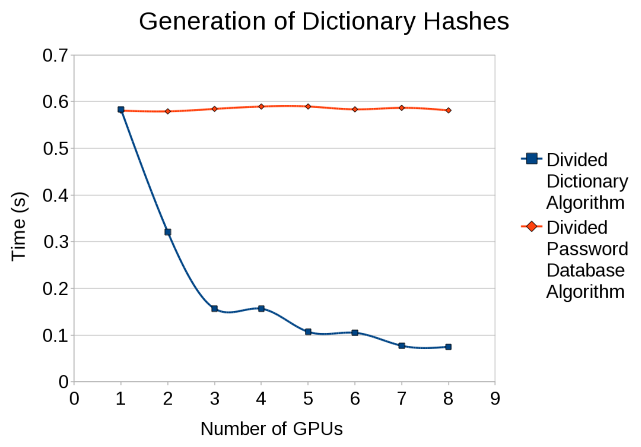

Fig. 6. Average GPU Time to Generate Dictionary Hashes

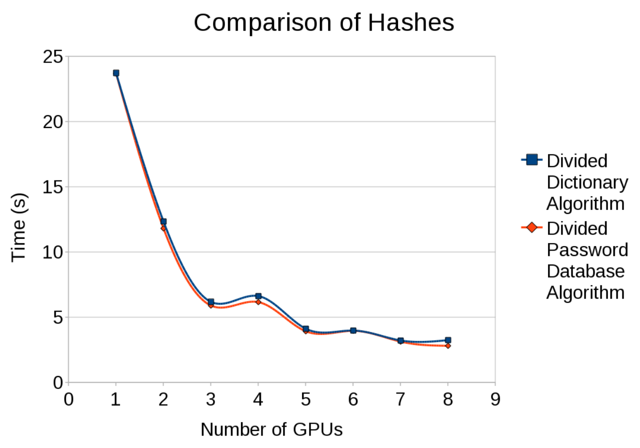

Fig. 7. Average GPU Time to Compare Hashes

Figure 7 shows the times to compare hash values of dictionary words with hash values found in the password database file.

When comparing the results for the divided dictionary and divided password database algorithms, it is clear that the increased execution time for the divided password database algorithm is due to the increased load and hash generation time of the dictionary due to the dictionaries increased size.

A serial CPU version of the password recovery program was also implemented. This program loads the entire dictionary into memory and then generates their associated hashes and stores them in memory. After the dictionary has been loaded into memory, the program loads the password database one account at a time and compares its hash value against the dictionary hashes, if a match is found the resulting recovered password is displayed.

The serial version of the password recovery program took 89.758 seconds to execute with a dictionary load time of 2.043 seconds, hash generation time of 5.856 seconds, and comparison time of 78.570 seconds. The minimal memory algorithm was 7x slower than the serial version. However, the divided dictionary and divided password database algorithms were 17x and 12x faster than the serial password recovery program. These results demonstrate that through the use of HPC computing using GPUs, the execution time of a password recovery program using a dictionary-based attack can be significantly reduced.

B. Memory Usage

In the minimal memory algorithm the amount of memory used by a GPU is approximately 1.048MB. This is because in this algorithm a single dictionary word is hashed, and the hash value is compared with those entries from the password database previously loaded into the GPU memory. Also, there is no need to store the hashed value after the comparisons.

The differences in memory required by the divided dictionary and divided password database algorithms are shown in Figure 8. Memory usage for the divided dictionary algorithm

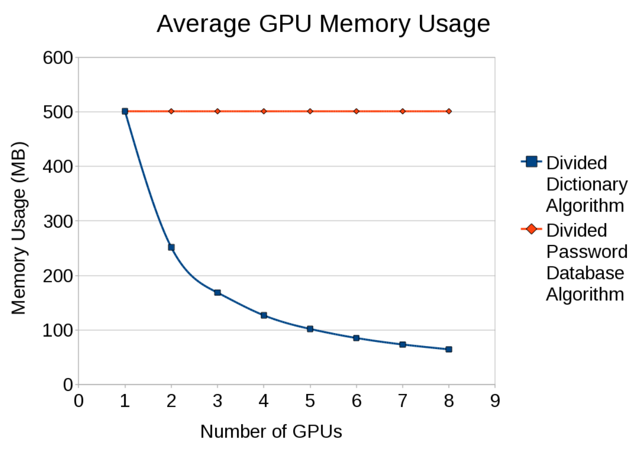

Fig. 8. Average Memory Usage Per GPU

decreases because the dictionary is divided among the available GPU devices. In the divided dictionary algorithm each GPU calculates and stores hash values for each dictionary word before testing for matches in the password database. Therefore, memory usage required by a GPU is twice the number of dictionary words distributed to the GPU.

The memory usage for the divided password database algorithm appears to be equal for all GPUs in Figure 8. However, Table IV shows a small decrease in memory usage as the number of GPUs increases. The reason that memory usage for the divided dictionary algorithm decreases at a greater rate than the divided password database algorithm is because the dictionary is larger than the password database.

TABLE IV
MEMORY USAGE: DIVIDED DICTIONARY VS DIVIDED PASSWORD DATABASE

GPUs	Divided Dictionary	Divided PWDB
1	501.219	501.219
2	251.678	501.204
3	168.498	501.199
4	126.907	501.197
5	101.952	501.195
6	85.317	501.194
7	73.434	501.193
8	64.522	501.193

V. FUTURE WORK

There are a number of areas where the current MPI+CUDA implementation for password recovery could be improved or expanded.

In another computing environment GPU devices may not be able to load all of the data at one time. This could be the result of fewer GPUs per MPI node or larger dictionary or password database files. The implementation should be modified so that it can process subsets of the data and still complete the pair-wise analysis of dictionary and password database entries without operator intervention.

Limitations in CUDA 3.0 affected the design and implementation of the password recovery application. For example, each GPU requires its own controlling thread, and memory context is lost when the controlling thread exits. Using newer versions of CUDA make it easier to create multiple GPU/threaded programs. Also, a new CUDA feature called GPUDirect [23] may improve performance of MPI send and receive functions. With this improved MPI performance, the password recovery application could use MPI instead of the file system to distribute data to each MPI node.

It would also be possible to expand on the current program by creating an implementation that further divides the work between the idle CPU processor cores and the GPUs. This would require addressing two issues. Because the GPUs run faster than the CPU cores, the initial division of data would reflect the difference in relative performance. Also, it may be necessary to perform dynamic load balancing if any of the processing units are being underutilized.

The implementation should be evaluated against the needs of other problem or simulation domains that have a high degree of parallelism. This would help to identify design decisions specific to password recovery and allow modifications to make a more general framework. Other fields may require processing of even larger data sets. This could help to demonstrate the extensibility of the framework by requiring more MPI nodes and more GPU devices per node.

Obtaining and implementing an actual user password database would allow for a better and more realistic analysis of the three data distribution strategies that were tested and could be further improved by implementing a larger dictionary specifically tailored towards password recovery; this database would not include random words, but would include commonly used phrases, sequences of characters, and passwords to further improve the success rate when recovering passwords from an actual user password database.

It would be interesting to develop MPI+CUDA implementations of other password recovery methods such as Markov chains or rainbow tables. Do these methods show similar scalability and performance gains? This would also allow for further comparisons and analysis of the dictionary approach with different password recovery methods.

The current implementation of the dictionary attack method used by the each algorithm performs sequential string comparisons for each thread on the GPU; this could be further improved by implementing a parallel version of string compares on the GPU to allow for faster string comparisons.

The strategies for data distribution could also be combined into a single approach that takes the dictionary and user password databases into account and uses the most appropriate algorithm to perform the calculations. This would allow the program to perform the password recovery using the algorithm with the best performance for a given dictionary and user password database size.

To help prevent wasted computation time, the program could be modified to stop once a match is found. This would prevent the program from waiting for all of the MPI nodes to finish performing hash comparisons for all of the dictionary words and instead stop when a match is found. This type of modification would be implemented on the MPI side of the program and would allow for each MPI node to indicate its current status and weather or not it has found a match. If a node has found a match all of the nodes could stop execution and return their results.

VI. Conclusion

Password-based authentication systems and their problems will most likely be the most common identity verification system for quite some time; and people will continue to use weak passwords like "12345" as long as they don't understand their important role in the security of computer systems.

This project demonstrates that significant scalability and performance gains are possible with a hybrid HPC approach to password recovery using MPI+CUDA. These tools can be easy to use and will likely be adopted by both sides in the arms race between system administrators and hackers.

Of the three algorithms analyzed the divided dictionary algorithm was the best approach for distributing data to each GPU. Dividing the larger dictionary, rather than the smaller password database among the GPUs ensured performance gains as the number of GPUs increased.

The divided password database algorithm stayed close with the divided dictionary algorithm in overall program execution time. However, dividing the smaller password database among the GPUs resulted in considerable duplication of processing by each GPU. Each GPU loaded the same dictionary, computing and storing hash values for the same words. With larger datasets or more GPUs this approach would fall further behind the execution time of the divided dictionary algorithm.

Creating programs with lower memory requirements is often laudable goal. However, the minimal memory algorithm actually worked against the GPU processing power resulting in poor execution times. This is demonstrated by the fact that a serial version of the password recovery program performed better in all metrics than the minimal memory algorithm.

In general, this work shows that MPI and CUDA can be combined to develop an efficient password recovery system that can scale to multiple compute nodes with multiple GPUs. Further, this work opens up many avenues for improving performance and scalability for large scale hybrid computing systems, as well as utilizing and testing modern improvements in GPU technology.

References

[1] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek, "Password cracking using probabilistic context-free grammars," in *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009, pp. 391–405.

[2] M. Dell'Amico, P. Michiardi, and Y. Roudier, "Password strength: An empirical analysis," in *29th IEEE Conference on Computer Communications (INFOCOM 2010)*, 2010, pp. 1–9.

[3] M. Bernaschi, M. Bisson, E. Gabrielli, and S. Tacconi, "An architecture for distributed dictionary attacks to cryptosystems," *Journal of Computers*, vol. 4, pp. 378–386, 2009.

[4] E. R. Sykes, M. Lin, and W. Skoczen, "Mpi enhancements in john the ripper," *Journal of Physics: Conference Series*, vol. 256, no. 1, 2010.

[5] A. Peslyak. John the ripper. [Online]. Available: http://www.openwall.com/john/

[6] MPICH. [Online]. Available: http://www.mcs.anl.gov/research/projects/mpich2/

[7] NVIDIA Corporation. CUDA. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[8] R. Graves, I. S. U. Electrical, and C. Engineering, *High performance password cracking by implementing rainbow tables on nVidia graphics cards (IseCrack)*. Iowa State University, 2008. [Online]. Available: http://books.google.com/books?id=unIx0fWdMUgC

[9] J. R. Crumpacker, "Distributed password cracking," Master's thesis, Naval Postgraduate School, December 2009.

[10] S. Marechal, "Advances in password cracking," *Journal in Computer Virology*, vol. 4, pp. 73–81, 2008.

[11] M. de la Asuncin, J. M. Mantas, M. J. Castro, and E. Fernndez-Nieto, "An mpi-cuda implementation of an improved roe method for two-layer shallow water systems," *Journal of Parallel and Distributed Comput. (2011)*, 2011.

[12] C. Teat and S. Peltsverger, "The security of cryptographic hashes," in *Proceedings of the 49th Annual Southeast Regional Conference*, ser. ACM-SE '11, 2011, pp. 103–108.

[13] I. Foster and N. T. Karonis, "A grid-enabled MPI: message passing in heterogeneous distributed computing systems," in *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–11.

[14] S. Pellicer, Y. Pan, and M. Guo, "Distributed md4 password hashing with grid computing package boinc," in *GCC*, ser. Lecture Notes in Computer Science, H. Jin, Y. Pan, N. Xiao, and J. Sun, Eds., vol. 3251. Springer, 2004, pp. 679–686.

[15] D. P. Anderson, E. Korpela, and R. Walton, "High-performance task distribution for volunteer computing," in *Proceedings of the First International Conference on e-Science and Grid Computing*, ser. e-Science '05, Dec. 2005, pp. 196–203.

[16] S. Pennycook, S. Hammond, S. Jarvis, and G. Mudalige, "Performance analysis of a hybrid mpi/cuda implementation of the nas-lu benchmark," *SIGMETRICS Performance Evaluation Review (2011)*, pp. 23–29, 2011.

[17] P.-L. Cayrel, G. Hoffmann, and M. Schneider, "Gpu implementation of the keccak hash function family," in *Information Security and Assurance*, ser. Communications in Computer and Information Science, T.-h. Kim, H. Adeli, R. J. Robles, and M. Balitanas, Eds. Springer Berlin Heidelberg, 2011, vol. 200, pp. 33–42.

[18] J. Gomez, F. Montoya, R. Benedicto, A. Jimenez, C. Gil, and A. Alcayde, "Cryptanalysis of hash functions using advanced multiprocessing," *Distributed Computing and Artificial Intelligence: 7th International Symposium (2010)*, vol. 79, pp. 221–228, 2010.

[19] C.-T. Yang, C.-L. Huang, and C.-F. Lin, "Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters," *Computer Physics Communications (2011)*, vol. 182, pp. 266–269, 2011.

[20] S. Dosopoulos, J. D. Gardiner, and J.-F. Lee, "An mpi/gpu parallelization of an interior penalty discontinuous galerkin time domain method for maxwells equations," *Radio Science (2011)*, vol. 46, 2011.

[21] N. P. Karunadasa and D. N. Ranasinghe, "On the comparative performance of parallel algorithms on small gpu/cuda clusters," *International Conference on High Performance Computing (2009)*, 2009.

[22] P. E. Jones. SHA1-c. [Online]. Available: http://www.packetizer.com/security/sha1/

[23] NVIDIA Corporation. GPUDirect. [Online]. Available: http://developer.nvidia.com/gpudirect