
# Parallel Brute Force Dictionary Attack Using OpenMPI and OpenMP

CPE 512

KYLE RAY

December 8, 2017

# Contents

## Abstract

In this experiment, a popular brute force attack known as the Dictionary Attack was implemented and then parallelized. The parallelization was performed using a message-passing interface (MPI) to distribute the dictionary amongst a cluster of computers and an API known as OpenMP which utilizes compiler pragmas as well as interface calls to parallelize existing serial code. Also, three different hashing algorithms (MD5, SHA1, SHA256) were tested in this experiment to see the timing affects a more complex algorithm might have on finding the password. It was observed that the MPI implementation performed poorly for each hashing algorithm in most cases worse than the serial implementation; whereas, the OpenMP implementation had great speed up and efficiency. The experiment shows that while parallelization does make an impact on performing this attack, the method that is employed is very important.

# Introduction

In this day and age most of our daily life is performed in the digital realm. To protect our digital identity, which can grant access to everything from our bank accounts to our social security number, we must have a secure method to access this information. Passwords have been used since the earliest days of computing and there have been many techniques developed to keep these passwords secure. In this project, I would like to explore a form of securing passwords known as hashing and one of the attacks used to crack it.

Hashing is a one-way mathematical transformation of an input to a constant sized output string. Hence a user gives a password, which then passes through a "hashing" algorithm and is eventually stored. There are many different hashing algorithms out there which range in complexity. Some of the most common hashing techniques are MD5, SHA-1, SHA-256, and so on.

Example MD5 Hashed Password:

Password = "Parallel_Programming_1s_fUn"

Hash = 6243ebf30a0642f7dad0ec17cc916df9

The purpose of this experiment was to determine the benefits that a hacker would have if they were to implement a parallel version of a very well-known penetration attack. The penetration attack used is the Brute Force Dictionary Attack. The attack assumes that the hacker possesses some password hashes from the victim and is now trying to determine the human readable password via confirming the password that created the hash. To do this the hacker will use a known Dictionary of either previously cracked passwords or just popular words in general, apply the hashing algorithm to each word and check each hash against the victim's hash. Once a matching hash is found the hacker will more than likely have access to any of the victim's accounts using this password.

Performing this attack can be quite a process intensive task if the dictionary is large and parallelism can be utilized to break these tasks down into smaller subtasks and thus decrease the overall time it takes to successfully find the password.

# Environment

The experiment was carried out on the Jetson cluster located at the University of Alabama in Huntsville.

## Test Layout

To perform the experiment a popular dictionary of passwords known as rockyou.txt was incorporated. The list was trimmed down from around fifteen million passwords to just ten million, removing most of the passwords with spaces. The dictionary can be found on the SkullSecurity website.

The three hashing algorithms (MD5, SHA1, SHA256) were taken from zedwood.com, license file included with source code. See appendix for implementation source code.

The parallel implementations were both run with the following conditions. They programs were tasked each with finding the words telva, khrisdaniel, and arisfarias which are near the front, middle, and end respectively of the dictionary. The intent of the test is to simulate three possible searches starting from the beginning of the list. This task was run on each implementation with 1, 2, 4, and 8 processes/threads. The times were recorded and checked against a base serial version to determine the overall speed up and efficiency of each implementation.

The application begins by reading in the dictionary file line by line and storing it in a container. In the MPI case the program will scatter the list as evenly as possible to the number of processes desired and each process will perform the attack using the hashing algorithm specified. The OpenMP version utilizes the work-sharing construct for the for loop to divide the work amongst the number of threads specified. Once the password is found it is displayed to the user along with the hashed representation. The time to find the password is also displayed.

Below is an example output from each of the implementations.

**MPI Implementation:**

```
krr0010@jetson:~/jetson/Project/MPI_Run$ mpirun -np 4 OpenMPI_Password_Crack 0
rockyou.txt telva
Reading Dictionary
Finished reading Dictionary with 10000000 passwords!
Starting the timer.
Scattering...
Pass found by process 1
Password is telva
Hash is bb95c8f4455a2385e3c5af1a301c6240
CLEAN UP: Process 1 is finished processing
CLEAN UP: Process 2 is finished processing
CLEAN UP: Process 3 is finished processing
CLEAN UP: Process 0 is finished processing
Time to find the password is 18.0492 seconds
Total execution time: 50.8507 seconds
```

**OpenMP Implementation:**

```
krr0010@jetson:~/jetson/Project/OpenMP_Run$ ./OpenMP_Password_Crack 0
rockyou.txt telva 4
Reading Dictionary
```

```
Finished reading Dictionary with 10000000 passwords!
Starting the Timer
Processing...
Pass found by process 3
Password is telva
Hash is bb95c8f4455a2385e3c5af1a301c6240
Time to find the password is 4.48877 seconds
```

The MPI version was originally implemented in a way that would stop processing after the password had been found by a process, this was achieved by keeping a flag and having every process check that flag to see if they needed to stop. In the end this implementation worked but was dreadfully slow. The blocking communication and the time to communicate are far too costly to be utilized in this sense especially because the loops are performing at least *listSize/numtasks* with the list size being ten million. The final decision was to remove this and just keep the execution time of the process that found the password because in the end, if a hacker were to use this method, once the password is output to the console the job is done.

## Results

This portion of the report contains the data collected when running the experiment. The table below shows the serial run time of each test case.

**Table 1: Serial Run Times for Each Test Scenario**

| Serial MD5 | Run Time (s) |
|---|---|
| Front of Dictionary (telva) | 19.7664 |
| Middle of Dictionary (khrisdaniel) | 34.8885 |
| End of Dictionary (arisfarias) | 56.9507 |
| | |
| Serial SHA1 | Run Time (s) |
| Front of Dictionary (telva) | 29.3506 |
| Middle of Dictionary (khrisdaniel) | 64.5736 |
| End of Dictionary (arisfarias) | 99.7225 |
| | |
| Serial SHA256 | Run Time (s) |
| Front of Dictionary (telva) | 36.5515 |
| Middle of Dictionary (khrisdaniel) | 75.9972 |
| End of Dictionary (arisfarias) | 110.581 |

The algorithms are ordered by complexity with SHA256 being the most complex hashing algorithm tested.

## MPI RESULTS

This section houses the data collected from the MPI test runs.

**Table 2: MPI Run Times for Each Test Scenario**

| MPI MD5 | NP = 1 | NP = 2 | NP = 4 | NP = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 23.0993 | 44.5034 | 24.8596 | 21.1322 |
| Middle of Dictionary (khrisdaniel) | 43.7216 | 36.4375 | 27.1765 | 19.9634 |
| End of Dictionary (arisfarias) | 63.7789 | 61.895 | 34.5734 | 16.8332 |

| MPI SHA1 | NP = 1 | NP = 2 | NP = 4 | NP = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 40.2165 | 113.334 | 45.8712 | 31.0657 |
| Middle of Dictionary (khrisdaniel) | 77.8335 | 96.8681 | 64.3992 | 38.7767 |
| End of Dictionary (arisfarias) | 111.519 | 130.166 | 60.6241 | 31.4162 |

| MPI SHA256 | NP = 1 | NP = 2 | NP = 4 | NP = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 43.5325 | 111.886 | 46.3524 | 41.2534 |
| Middle of Dictionary (khrisdaniel) | 80.7348 | 94.4664 | 64.7486 | 32.6261 |
| End of Dictionary (arisfarias) | 113.118 | 129.271 | 75.7362 | 41.5177 |

Comparing the results from Table 2 with the results from Table 1 we can see that the MPI version performed poorly in the cases where less than eight processes were used and that the word was found near the middle or end of the file. In all other cases the serial version performed better. The values in Table 2 **DO** take into account the scattering of the dictionary to the other processes, so this incurs quite a bit of communication overhead as well as the overhead of creating the processes. Refer to the next two tables for the speed up and efficiency of this implementation.

**Table 3: MPI Speed Up for Each Test Scenario**

| MPI MD5 S(p) | NP = 1 | NP = 2 | NP = 4 | NP = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 0.855714242 | 0.444154829 | 0.795121402 | 0.935368774 |
| Middle of Dictionary (khrisdaniel) | 0.797969425 | 0.957488851 | 1.283774585 | 1.74762315 |
| End of Dictionary (arisfarias) | 0.892939514 | 0.920117942 | 1.647240364 | 3.383236699 |

| MPI SHA1 S(p) | NP = 1 | NP = 2 | NP = 4 | NP = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 0.729814877 | 0.258974359 | 0.639848096 | 0.944791201 |
| Middle of Dictionary (khrisdaniel) | 0.829637624 | 0.666613674 | 1.002708108 | 1.66526806 |
| End of Dictionary (arisfarias) | 0.894219819 | 0.76611788 | 1.644931636 | 3.174238132 |

| MPI SHA256 S(p) | NP = 1 | NP = 2 | NP = 4 | NP = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 0.839637053 | 0.326685197 | 0.788556795 | 0.88602394 |
| Middle of Dictionary (khrisdaniel) | 0.941318985 | 0.804489215 | 1.173727308 | 2.329337555 |
| End of Dictionary (arisfarias) | 0.977572093 | 0.855420009 | 1.460081177 | 2.663466425 |

Although the MPI implementation reduced the execution in a few cases the speed up achieved was not desirable compared to the number of resources used and this can be seen by viewing the efficiency table below.

**Table 4: MPI Efficiency for Each Test Scenario**

| MPI MD5 E(p) | NP = 1 | NP = 2 | NP = 4 | NP = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 0.855714242 | 0.222077414 | 0.19878035 | 0.116921097 |
| Middle of Dictionary (khrisdaniel) | 0.797969425 | 0.478744425 | 0.320943646 | 0.218452894 |
| End of Dictionary (arisfarias) | 0.892939514 | 0.460058971 | 0.411810091 | 0.422904587 |

| MPI SHA1 E(p) | NP = 1 | NP = 2 | NP = 4 | NP = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 0.729814877 | 0.129487179 | 0.159962024 | 0.1180989 |
| Middle of Dictionary (khrisdaniel) | 0.829637624 | 0.333306837 | 0.250677027 | 0.208158508 |
| End of Dictionary (arisfarias) | 0.894219819 | 0.38305894 | 0.411232909 | 0.396779766 |

| MPI SHA256 E(p) | NP = 1 | NP = 2 | NP = 4 | NP = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 0.839637053 | 0.163342599 | 0.197139199 | 0.110752992 |
| Middle of Dictionary (khrisdaniel) | 0.941318985 | 0.402244608 | 0.293431827 | 0.291167194 |
| End of Dictionary (arisfarias) | 0.977572093 | 0.427710005 | 0.365020294 | 0.332933303 |

Looking at the table above we can see that in most cases where more than one process is used that the efficiency declines significantly.

It seems that, from this experiment, the process of hashing a string and doing string compares doesn't really suit the type of parallelism offered by using MPI.

## OPENMP RESULTS

Not only was the OpenMP version much easier to implement, the results achieve were vastly better than the MPI version implemented. This implementation achieved a reasonable decrease in execution time every time more threads were utilized.

**Table 5: OpenMP Run Times for Each Test Scenario**

| OpenMP MD5 | NT = 1 | NT = 2 | NT = 4 | NT = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 17.6082 | 8.92389 | 6.66862 | 6.08179 |
| Middle of Dictionary (khrisdaniel) | 36.9943 | 18.1401 | 12.3859 | 10.8101 |
| End of Dictionary (arisfarias) | 57.7918 | 27.8238 | 13.8588 | 13.7383 |

| OpenMP SHA1 | NT = 1 | NT = 2 | NT = 4 | NT = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 35.398 | 17.1591 | 9.55503 | 9.41605 |
| Middle of Dictionary (khrisdaniel) | 69.4872 | 35.6905 | 19.5332 | 19.4156 |
| End of Dictionary (arisfarias) | 106.239 | 51.7842 | 27.6193 | 27.9677 |

| OpenMP SHA256 | NT = 1 | NT = 2 | NT = 4 | NT = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 37.0772 | 17.312 | 8.97647 | 8.78002 |
| Middle of Dictionary (khrisdaniel) | 76.7078 | 37.414 | 27.3433 | 20.387 |
| End of Dictionary (arisfarias) | 112.131 | 56.9685 | 39.4119 | 26.3841 |

Comparing Table 5 with the serial run times of Table 1 we can clearly see that this implementation is much faster. It's interesting, but makes sense, that the execution time for the case that the word is near the front and the application utilizes four and eight threads is near the same for each implementation. It seems that there is a limit for this case and adding more threads will not be as efficient.

**Table 6: OpenMP Speed Up for Each Test Scenario**

| OpenMP MD5 S(p) | NT = 1 | NT = 2 | NT = 4 | NT = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 1.122567895 | 2.214998168 | 2.964091521 | 3.250095778 |
| Middle of Dictionary (khrisdaniel) | 0.943077717 | 1.923280467 | 2.816791674 | 3.227398451 |
| End of Dictionary (arisfarias) | 0.985446032 | 2.046834005 | 4.109352902 | 4.145396446 |

| OpenMP SHA1 S(p) | NT = 1 | NT = 2 | NT = 4 | NT = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 0.82915984 | 1.710497637 | 3.071743364 | 3.117082004 |
| Middle of Dictionary (khrisdaniel) | 0.929287696 | 1.809265771 | 3.305838265 | 3.325861678 |
| End of Dictionary (arisfarias) | 0.938661885 | 1.925732173 | 3.610609248 | 3.565631067 |

| OpenMP SHA256 S(p) | NT = 1 | NT = 2 | NT = 4 | NT = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 0.985821475 | 2.111338956 | 4.071923596 | 4.163031519 |
| Middle of Dictionary (khrisdaniel) | 0.990736275 | 2.031250334 | 2.779371912 | 3.727728454 |
| End of Dictionary (arisfarias) | 0.986176882 | 1.941090252 | 2.805776935 | 4.191198487 |

As far as speed up, looking at the table above, the cases using two and four threads have close to ideal speed up compared to the other scenarios.

**Table 7: OpenMP Efficiency for Each Test Scenario**

| OpenMP MD5 E(p) | NT = 1 | NT = 2 | NT = 4 | NT = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 1.122567895 | 1.107499084 | 0.74102288 | 0.406261972 |
| Middle of Dictionary (khrisdaniel) | 0.943077717 | 0.961640234 | 0.704197919 | 0.403424806 |
| End of Dictionary (arisfarias) | 0.985446032 | 1.023417003 | 1.027338226 | 0.518174556 |

| OpenMP SHA1 E(p) | NT = 1 | NT = 2 | NT = 4 | NT = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 0.82915984 | 0.855248818 | 0.767935841 | 0.38963525 |
| Middle of Dictionary (khrisdaniel) | 0.929287696 | 0.904632886 | 0.826459566 | 0.41573271 |
| End of Dictionary (arisfarias) | 0.938661885 | 0.962866087 | 0.902652312 | 0.445703883 |

| OpenMP SHA256 E(p) | NT = 1 | NT = 2 | NT = 4 | NT = 8 |
|---|---|---|---|---|
| Front of Dictionary (telva) | 0.985821475 | 1.055669478 | 1.017980899 | 0.52037894 |
| Middle of Dictionary (khrisdaniel) | 0.990736275 | 1.015625167 | 0.694842978 | 0.465966057 |
| End of Dictionary (arisfarias) | 0.986176882 | 0.970545126 | 0.701444234 | 0.523899811 |

Overall OpenMP is the most efficient implementation as well as very easy to implement compared to the MPI version.

## Conclusion

After the experiment it is clear to see that a hacker could benefit from using parallel processing to perform the brute force dictionary attack. The experiment also demonstrated that some tools are better for the job than others in the case where MPI was very tedious to use because of all the synchronization and message passing and OpenMP involved changing a few lines of code in the serial version to achieve a substantial performance increase.

## Future Work

Optimization is a never-ending endeavor. The parallel implementations in this experiment were nowhere near totally optimized and there is a lot of room for improvement in this area. Also, the actual searching algorithm could be changed from brute force to achieve better results, possibly starting at the center of the dictionary and performing a divide and conquer methodology could increase performance overall for any test case.

In addition, the hashing algorithms that were used in this experiment are for the most part outdated. MD5 is not widely used anymore for security because it is considered to be a weak hashing algorithm and easily cracked. Future work could include testing some of the more complex hashing algorithms such as SHA-512 or RIPEMD-320.

# Appendix

## MPI SOURCE CODE

```
/*
* MPI_Password_Crack.cpp
*
* Program designed to utilize the message passing interface (MPI)
* to parallelize the process of password cracking using the dictionary
* attack.  This will take in a text file of passwords as well as the real
* password we are looking for, this is to simulate when a hacker has already
* dumped hashes from the victim machine and is now trying to crack the hash,
* and perform the appropriate hashing technique to each word in the dictionary.
* If the hacker is lucky the victim's actual password will be in the dictionary
* file chosen and the hacker will be able to crack the hash and compromise the
* victim's system.  Performing this attack serially takes a very long time, so
* this program faciliates a study to see how much of a speed up a hacker could
* gain by using MPI.
*
* Note:
*   The hashing algorithms used in this study are from zedwood.com
*   Refer to the license file attached.
*
* Compile
*   mpic++ md5.cpp sha1.cpp sha256.cpp MPI_Password_Crack.cpp -o
MPI_Password_Crack
*
* Run:
*   mpirun -np NUM_PROCESSES MPI_Password_Crack DICTIONARY ACTUAL_PASSWORD
*
* Author: Kyle Ray
* CPE 512 Intro to Parallel Programming
* Project
* December 5, 2017
*/

#include <iostream>
#include <vector>
#include <fstream>
#include <cstdlib> // atoi
using namespace std;

// Hashing Function Include
#include "md5.h"
#include "sha1.h"
#include "sha256.h"

// MPI
#include <mpi.h>

// Prototypes
int readDictionary(string filename, std::vector<string>& passwords);
void scatter(std::vector<string>& passwords, std::vector<string>&
local_passwords, int num_passes, int root, int rank, int numtasks);

int main(int argc, char* argv[])
{
  // Variables
  int numtasks, rank, num;
  double tot_finish;
  double start, finish;
```

```
  int hash_mode; // 0 - MD5, 1 - SHA-1, 2 - SHA-256
  MPI_Status status;

  // Start up the MPI Processes
  MPI_Init(&argc, &argv); // initialize MPI environment
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks); // get total number of MPI
processes
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get unique task id number

  if (rank == 0)
  {
    if (argc < 3)
    {
      std::cout << "Usage: mpirun -np numtasks programName hashmode
dictionaryFile passwordToFind" << endl;
      MPI_Abort(MPI_COMM_WORLD, MPI_ERR_INFO);
    }
  }

  // Store the Hash Mode
  hash_mode = atoi(argv[1]);

  // Actual Password
  std::string actual_pass;
  switch (hash_mode)
  {
    case 0:
      actual_pass = md5(argv[3]);
      break;
    case 1:
      actual_pass = sha1(argv[3]);
      break;
    case 2:
      actual_pass = sha256(argv[3]);
      break;
  }

  // Password Container
  std::vector<string> passwords;

  // Done Flag
  bool pass_found = false;

  // Total number of passwords, used for sizing
  int total_num_passes = 0;

  // Root needs to read in and parse the dictionary file
  // Should then send the size that each process should allocate to hold their
part
  // Scatter the dictionary to other tasks.

  if (rank == 0)
  {
    std::cout << "Reading Dictionary" << endl;
    if (!readDictionary(argv[2], passwords))
    {
      // Something went wrong with the file
      MPI_Abort(MPI_COMM_WORLD, MPI_ERR_FILE);
    }

    // Set the total number of passwords to make each process aware
    total_num_passes = passwords.size();
```

```
    // Display Size of Dictionary
    std::cout << "Finished reading Dictionary with " << total_num_passes << "
passwords!" << endl;
  }

  MPI_Bcast(&total_num_passes, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Local Set of Passwords
  std::vector<string> local_passwords;

  // Setup a barrier to make sure that all of the processes enter the region at
the same time
  // for timing analysis
  MPI_Barrier(MPI_COMM_WORLD);

  if (rank == 0)
  {
    std::cout << "Starting the timer. " << endl;
    std::cout << "Scattering...\n";
    start = MPI_Wtime();
  }

  // Scatter the passwords to the processes
  scatter(passwords, local_passwords, total_num_passes , 0, rank, numtasks);

  // Flag for a process to report when done, allows other processes to know
when to stop
  bool someone_done;

  int pass_count = 0;
  double loc_finish = 0;

  // Check each local pass hash and if we find the password then quit and let
all the other processes know
  // Start the main loop
  // 1.) Hash the current password in the MPI process list
  // 2.) Compare the hashed string with the password hash given by the user
  // 3.) If it is a match, then quit and let the other processes know.
  // 4.) Else, process the next password.
  for (int i = 0; i < local_passwords.size(); i++)
  {
    std::string check_pass;
    switch (hash_mode)
    {
      case 0:
        check_pass = md5(local_passwords[i]);    // MD5 algorithm by
zedwood.com
        break;
      case 1:
        check_pass = sha1(local_passwords[i]);   // SHA1 algorithm by
zedwood.com
        break;
      case 2:
        check_pass = sha256(local_passwords[i]); // SHA256 algorithm by
zedwood.com
        break;
    }

    // If we have a match then set the flag
    if (actual_pass == check_pass)
    {
```

```
      pass_found = true;
      loc_finish = MPI_Wtime();
      std::cout << "Pass found by process " << rank << endl
        << "Password is " << local_passwords[i] << endl
        << "Hash is " << check_pass << endl;
    }

    // Let other processes know if the password is found or not
    // TODO: Could probably use non blocking send and receives to do this
somehow, just to make it more efficient.
    // Although, all of the loops should be somewhere around the same time
complexity, so it might not matter too much.
    // The communication takes too much time because it is executing every
loop.
    //MPI_Status status;
    //for (int mpitask = 0; mpitask < numtasks; mpitask++)
    //{
    //  // If not current process, send flag to other processes.
    //  if (mpitask != rank)
    //  {
    //    MPI_Send(&pass_found, 1, MPI_C_BOOL, mpitask, rank, MPI_COMM_WORLD);

    //    // Receive flag from other processes
    //    MPI_Recv(&someone_done, 1, MPI_C_BOOL, mpitask, mpitask,
MPI_COMM_WORLD, &status);

    //    // If password is found by a process, then set exit condition
    //    if (someone_done)
    //      pass_found = true;
    //  }
    //}

    //pass_count++;

    // If the password has been found, no more processing necessary
    if (pass_found)
      break;
  }

  if (!pass_found)
    loc_finish = MPI_Wtime();

  MPI_Reduce(&loc_finish, &finish, 1, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
  std::cout << "CLEAN UP: Process " << rank << " is finished processing " <<
endl;//<< pass_count << " number of passwords ending on " <<
local_passwords[pass_count-1] << endl;

  MPI_Barrier(MPI_COMM_WORLD);

  // Display the results
  if (rank == 0)
  {
    tot_finish = MPI_Wtime();
    std::cout << "Time to find the password is " << (finish - start) << "
seconds" << endl;
    std::cout << "Total execution time: " << (tot_finish - start) << " seconds"
<< endl;
  }


  // Terminate MPI Program -- perform necessary MPI housekeeping
  // clear out all buffers, remove handlers, etc.
```

```
  MPI_Finalize();
}

// Method to read in the dictionary text file and store the passwords
// into a vector.
int readDictionary(string filename, std::vector<string>& passwords)
{
  // Create file stream object and open the file
  fstream dictFile(filename.c_str());

  // Keep a count of the number of passwords, needed for equal distribution
  int count = 0;

  if (dictFile.fail())
  {
    dictFile.close();
    return count;
  }

  // Read and store each password
  while (!dictFile.eof())
  {
    string line;
    getline(dictFile, line);
    passwords.push_back(line);
    ++count;
  }

  dictFile.close();
  return count;
}

// Method to equally scatter the dictionary to each MPI process
// TODO: Find a faster way to send strings.
// These blocking send and receives take a very long time.
void scatter(std::vector<string>& passwords, std::vector<string>&
local_passwords, int num_passes, int root, int rank, int numtasks)
{
  MPI_Status status;

  // Root handles the distribution of the dictionary file
  if (rank == root)
  {
    int base = num_passes / numtasks;
    int extra = num_passes % numtasks;

    int begin_element = 0;

    for (int mpitask = 0; mpitask < numtasks; mpitask++)
    {
      int num_passes_by_rank = mpitask < extra ? base + 1 : base;
      if (mpitask == root)
      {
        for (int i = 0; i < num_passes_by_rank; i++)
        {
          local_passwords.push_back(passwords[begin_element]);
          ++begin_element;
        }
      }
      else
      {
        for (int i = 0; i < num_passes_by_rank; i++)
```

```
        {
          char* pass = const_cast<char*>(passwords[begin_element].c_str());
          MPI_Send(pass, passwords[begin_element].length(),
            MPI_CHAR, mpitask, mpitask, MPI_COMM_WORLD);
          ++begin_element;
        }
      }
    }
  }

  // Other processes receive their share of the dictionary
  if (rank != root)
  {
    int base = num_passes / numtasks;
    int extra = num_passes % numtasks;
    int num_passes_by_rank = rank < extra ? base + 1 : base;
    for (int i = 0; i < num_passes_by_rank; i++)
    {
      // Probe the message
      MPI_Probe(root, rank, MPI_COMM_WORLD, &status);
      int len = 0;
      MPI_Get_count(&status, MPI_CHAR, &len);
      char* buf = new char[len];

      MPI_Recv(buf, len, MPI_CHAR,
        root, rank, MPI_COMM_WORLD, &status);

      string buf_string(buf, len);
      local_passwords.push_back(buf_string);
    }
  }
}
```

## OPENMP SOURCE CODE

```
/*
* OpenMP_Password_Crack.cpp
*
* Program designed to utilize OpenMP parallel library
* to parallelize the process of password cracking using the dictionary
* attack.  This will take in a text file of passwords as well as the real
* password we are looking for, this is to simulate when a hacker has already
* dumped hashes from the victim machine and is now trying to crack the hash,
* and perform the appropriate hashing technique to each word in the dictionary.
* If the hacker is lucky the victim's actual password will be in the dictionary
* file chosen and the hacker will be able to crack the hash and compromise the
* victim's system.  Performing this attack serially takes a very long time, so
* this program faciliates a study to see how much of a speed up a hacker could
* gain by using OpenMP.
*
* Note:
*   The hashing algorithms used in this study are from zedwood.com
*   Refer to the license file attached.
*
* Compile
*   g++ md5.cpp sha1.cpp sha256.cpp OpenMP_Password_Crack.cpp -o
OpenMP_Password_Crack -fopenmp
*
* Run:
*   ./OpenMP_Password_Crack HASH_MODE DICTIONARY ACTUAL_PASSWORD NUM_THREADS
*
* Author: Kyle Ray
* CPE 512 Intro to Parallel Programming
* Project
* December 5, 2017
*/

#include <iostream>
#include <vector>
#include <fstream>
#include <cstdlib> // atoi
using namespace std;

// Hashing Function Include
#include "md5.h"
#include "sha1.h"
#include "sha256.h"

// OpenMP
#ifdef _OPENMP
#include <omp.h>
#endif

// Prototypes
int readDictionary(string filename, std::vector<string>& passwords);

int main(int argc, char* argv[])
{
  // Variables
  double start, finish;

  if (argc < 5)
  {
    std::cout << "Usage: programName hashMode dictionaryFile passwordToFind
numberOfThreads" << endl;
```

```
    exit(EXIT_FAILURE);
  }

  int hash_mode = atoi(argv[1]); // 0 - MD5, 1 - SHA-1, 2 - SHA-256

  // Actual Password
  std::string actual_pass;
  switch (hash_mode)
  {
  case 0:
    actual_pass = md5(argv[3]);
    break;
  case 1:
    actual_pass = sha1(argv[3]);
    break;
  case 2:
    actual_pass = sha256(argv[3]);
    break;
  }

  // Password Container
  std::vector<string> passwords;

  // Done Flag
  bool pass_found = false;

  // Total number of passwords, used for sizing
  int total_num_passes = 0;

  // Root needs to read in and parse the dictionary file
  // Should then send the size that each process should allocate to hold their
part
  // Scatter the dictionary to other tasks.
  std::cout << "Reading Dictionary" << endl;
  if (!readDictionary(argv[2], passwords))
  {
    // Something went wrong with the file
    exit(EXIT_FAILURE);
  }

  // Set the total number of passwords to make each process aware
  total_num_passes = passwords.size();

  // Display the number of Passwords
  cout << "Finished reading Dictionary with " << total_num_passes << "
passwords!" << endl;

  cout << "Starting the Timer " << endl;
  cout << "Processing...\n";
  // Start the timer
  start = omp_get_wtime();


  // Check each local pass hash and if we find the password then quit
  int num_threads;
  if (argc < 5)
    num_threads = 1;
  else
    num_threads = atoi(argv[4]);

#pragma omp parallel for num_threads(num_threads) schedule(static, 1)
  for (int i = 0; i < passwords.size(); i++)
```

```
  {
    std::string check_pass;
    switch (hash_mode)
    {
    case 0:
      check_pass = md5(passwords[i]);
      break;
    case 1:
      check_pass = sha1(passwords[i]);
      break;
    case 2:
      check_pass = sha256(passwords[i]);
      break;
    }

    // If we have a match then set the flag
    if (actual_pass == check_pass)
    {
      pass_found = true;
      finish = omp_get_wtime();
      std::cout << "Pass found by process " << omp_get_thread_num() << endl
        << "Password is " << passwords[i] << endl
        << "Hash is " << check_pass << endl;
    }

    // Let the other processes know that we are done.
    if (pass_found)
    {
      i = passwords.size();
    }
  }

  // Display the results
  std::cout << "Time to find the password is " << (finish - start) << "
seconds" << endl;

  return 0;
}

// Method to read in the dictionary text file and store the passwords
// into a vector.
int readDictionary(string filename, std::vector<string>& passwords)
{
  // Create file stream object and open the file
  fstream dictFile(filename.c_str());

  // Keep a count of the number of passwords, needed for equal distribution
  int count = 0;

  if (dictFile.fail())
  {
    dictFile.close();
    return count;
  }

  // Read and store each password
  while (!dictFile.eof())
  {
    string line;
    getline(dictFile, line);
    passwords.push_back(line);
    ++count;
```

```
  }

  dictFile.close();
  return count;
}
```