-

CS240A Applied Parallel Computing

Final Report

Parallelization of John the Ripper Using MPI

BY
Andy Pippin
Brent Hall
Wilson Chen

{pippin,brendon,wilson}@cs.ucsb.edu

TABLE OF CONTENTS

INTRODUCTION

Overview

This project describes an effort to parallelize John the Ripper [1] (John), an open-source password cracking software package. This type of work is relevant to Information Technology (IT) security for different reasons. With increasing availability of high performance computing resources, it is important to understand the abilities of would be attackers. By reinforcing back of the envelope calculations with experimental data, system administrators can better gauge the security of their systems and plan accordingly. Second, tools that result from projects like this give administrators the ability to efficiently check the security of their systems. They can scan for weak passwords that are likely to be cracked by others using similar tools.

UNIX Passwords

Most UNIX systems use a username/password combination as the only real barrier preventing unwanted access to system resources. It follows that the security of the system depends quite heavily on the security of the password/authentication system. Almost all accounts on a UNIX system have a password assigned to it. The passwords are not stored in plaintext, but instead, they are encoded ("hashed") using a modified encryption algorithm. By default, most UNIX systems use the *crypt()* library function [2], which is based on the standard Data Encryption System (DES). This hashing process is similar to regular encryption, but it is a one-way hash function. This means that it is a straightforward process to take a plain-text word and obtain its hash, but it is considered computationally infeasible to do the reverse. Passwords can be up to 8 characters in length. The algorithm takes the 7 lower order bits from each character and packs them into a 56-bit string. This key is used by the algorithm to hash a constant string normally consisting of all zeros. A randomly generated two character "salt" is used to perturb this encryption in one of 4096 different ways. The result is the hash consisting of 13 printable ASCII characters. The first two characters are the salt that was used to help generate the hash.

UNIX uses the /etc/passwd file to store login information. It has entries of the form:

```
smithj:wOWQfwTXWRB.I:561:561:Joe Smith:/home/smithj:/bin/bash
```

The fields are delimited by colons and represent the following:

- login id
- password hash
- numeric user id
- numeric group id
- full user name
- home directory
- shell

When a user attempts to login, the password entered is hashed using the same algorithm. The result is compared with the password hash in the corresponding entry for the user. If there is a match, the user is authenticated. In older or distributed environment, the password file can be obtained by any user. This is a tremendous security risk and the security community developed shadow password to safeguard the password file. Shadowed password files are used by default on most systems now, and this hides the password hash information from everyone but the root user.

Cracking UNIX Passwords

Password cracking programs can be easily found on the Internet. John is one of the more popular and powerful programs, but they all operate in similar ways. They presume that the user has a list of password hashes and the user wants to know the corresponding passwords. Cracking programs successively generate different candidate passwords; encrypt them with the same algorithm used in the password file, and compare the results. If the hashes are the same, then the password has been cracked. The main difference between different cracking programs is the way they generate the candidate passwords.

They normally use one of two approaches. Brute-force style attacks are the most powerful, but also the most time consuming. This technique works by hashing every possible character combination that is a valid password. This method is considered infallible - it will eventually crack any given password. However, given that there are about 95 valid characters and assuming passwords up to 8 characters in length, there are about 6 quadrillion different passwords. It would take a 1.5 GHz single processor machine approximately 2000 years to generate every possible hash (using John).

Many users however, do not use random strings of characters as passwords. Dictionary words or phrases are commonly used and this drastically reduces the size of the password space. Dictionary attacks take advantage of this fact and employ a large file containing words, which are used by the program to generate candidate passwords. Rules can also be incorporated that modify the words in the dictionary by adding different punctuation, changing capitalization, etc. This method is much quicker at identifying so called 'weak' passwords, but can't easily crack passwords that contain random characters. John the Ripper is capable of performing both brute force and dictionary attacks.

Previous Work

The idea of applying high performance computing resources to password cracking is not new. The San Diego Super Computer Center [3] has performed two related projects. Both focused on UNIX passwords generated with the *crypt()* function. The first, Tablecrack, started in 1997. Over the course of two weeks, they used a number of workstations to pre-compute the hashes to 51 million common passwords. This generated 1.1 terabytes of data, and was stored on SDSC's large archival tape storage system. These stored hashes represent weak passwords, and SDSC used this system for a couple of years to identify weak passwords. The storage requirement for this project was considered huge in 1997, but is not too significant by today's standards. SDSC's next project was called Teracrack. One of the main purposes of this project was to demonstrate that the *crypt()* function should be considered obsolete for generating password hashes. They used Blue Horizon to calculate the hashes of 50 million common passwords, which took 81 minutes and 2 terabytes of storage. They extrapolated these results to predict that an attacker, perhaps using a large number of compromised machines, could duplicate their results in a couple of days.

There have been other projects that have used hardware resources far more readily accessible to the public. The `dkbf` (Distributed Keyboard Brute Force) program [4] is a parallel implementation of `L0phtCrack` using MPI. It was designed to crack Windows LANMAN and NT hashes. In the documentation for their program (which was written in 2000) they estimated that using a cheap 8-node cluster ($^\$$3000) any NT password can be cracked within 35 days. This would be both considerably faster and cheaper today. In a similar project, R. Lim of the

University of Nebraska-Lincoln parallelized John using MPI [5]. In his implementation, he focused on the brute-force mode by giving different parts of the key-space to the processes involved. He achieved the performance he expected, but suggested that more work could be done to take advantage of John's various cracking modes.

JOHN-MPI

Our Approach and Parallelization Schemes

Our intent was to parallelize John to perform dictionary attacks. In contrast to Lim's work, which is better for a concentrated attack on a single password; our program will be better suited for quickly scanning an entire password file for weak passwords. This makes it more useful to administrators as a security tool. We used the Message Passing Interface (MPI) for inter-process communication, and concentrated on passwords generated with the UNIX *crypt()* function, though John is capable of handling MD5 and many other hashing schemes. Password cracking is inherently very parallelizable; the hashing attempts are all independent of each other. John is already quite good at performing serial dictionary attacks. We wanted to find the best way to partition the problem for efficient parallelization. We considered the three different techniques described below.

Because John uses a large word list file to perform the dictionary crack, this word list can be broken up into equal pieces and distributed to each process involved. Each process would then try to crack each password using only the words it has. Uncracked passwords could then be passed onto the next process. This technique would also provide a means for concentrating a dictionary attack on a single password, by giving the same hash to all processes simultaneously. Because each process only has a part of the dictionary, memory requirements are relaxed. Proper coordination to ensure load balance between processes is essential, but difficult. It is impossible to tell beforehand how long each process will spend trying to crack each hash, and there is a very good chance that processes will be idle waiting for one another.

Instead of splitting the dictionary we could split the password hashes amongst the processes involved. Each process would then effectively perform a serial crack on the reduced password list assigned to it. This would require that each process keep an entire copy of the dictionary in memory, but the coordination between processes would be greatly reduced. Load balancing is still an issue, because it is unlikely that each process was assigned a list of equal strength passwords. Some processes are likely to finish with their list before others, and sit idle. It would be very difficult to know the difficulty of passwords beforehand (that is the point of running John in the first place).

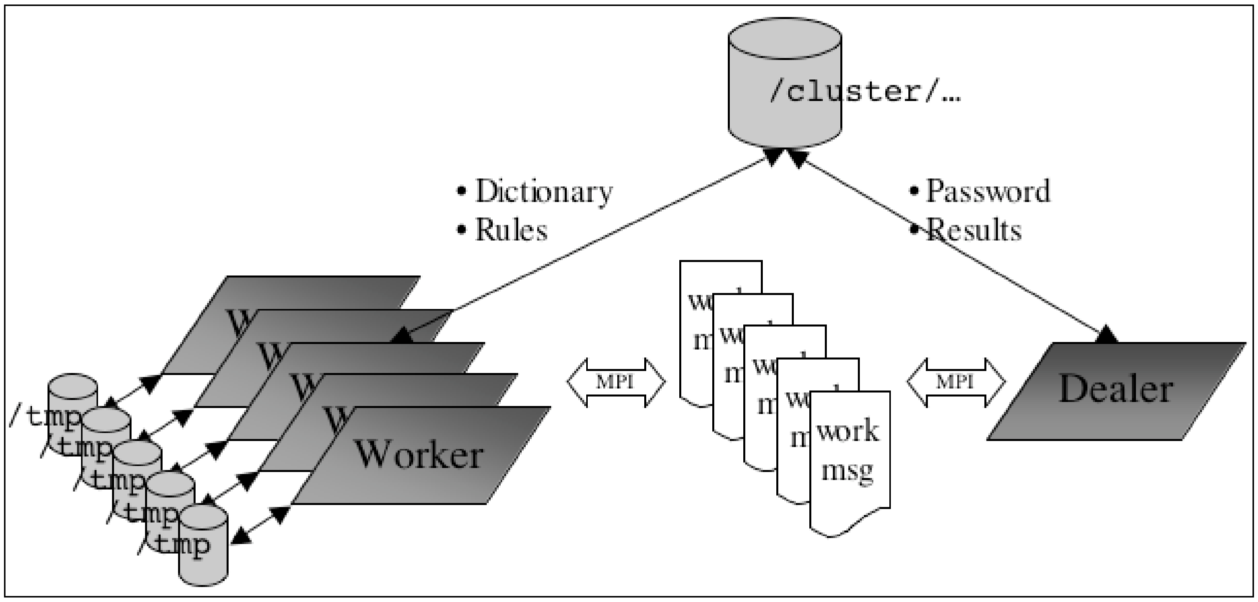
Finally, a work-sharing scheme could be used. One process could act as a dealer, handing out password hashes to worker processes, who would then proceed with the cracking attempt as usual. This requires each worker to have the entire word list in memory, but load balancing is ensured because once a process finishes with a cracking attempt, it is given more to do by the dealer. At the end of the password list some processes may sit idle while others finish with the last password hashes. The dealer process also may become a performance bottleneck if it is overloaded and workers are sitting idle while waiting for more work. This is the method we chose to parallelize John. We felt that the parallelization issues described above would be easiest to overcome in this case. The particulars of the MPI interface would also be better suited to an approach of this nature.

Topology

Most clusters usually have some disk storage available to all nodes as well as some local disk storage for temporary use. The root node (the "dealer") uses the global storage to store the passwords to be cracked as well as the results of the attempts. In order to keep the results consistent, the compute nodes (the "workers") use the central storage for the wordlist and attack rules. The workers also use the local storage to simulate a global storage in order to keep the changes to John as small as possible.

When using `cluster2.cs.ucsb.edu` (cluster2), the global, shared storage is located at `/cluster` with the local storage being located at `/tmp`. MPI communicates to each node in the cluster via a 1 Gb/s Ethernet switch.

Figure 1: John-MPI Topology



Modus Operandi

There are several distinct operational modes: initialization of MPI for both the dealer and worker nodes, attempts to crack passwords, and termination.

Initialization

Like most MPI programs, john-mpi has a single process that parses the command line options and then invokes *MPI_Init()*. Once all processes have started, each determines its rank within the MPI communicator. Rank 0 becomes the dealer, while all others become workers. The dealer will not perform any password cracking and is reserving itself to handle the bulk of the communication between processes.

When the dealer starts up, it reads in the specified password file, and sorts it based on the password hash. Since the encrypting salt is prepended to the password hash, this groups all password hashes with the same salt together. By passing all of the passwords with the same salt to the same process, this allows it to encrypt each word in the dictionary once and compare it against multiple passwords. After sorting the entries, it creates a linked list of "work messages". Each work message can contain up to 20 different password entries. In a weak attempt to increase security, the login identifier is not transmitted with the hash.

Each worker starts up and immediately calls *MPI_Recv()* on the dealer in order to get something to work on. While the dealer is working on the sorting and preparation of work messages, these processes simply block.

Runtime

Once the worker gets a work message, it writes the contents of that message to a temporary password file on its local disk, and then calls the serial John to start the cracking process. The first step in the serial John is to load the dictionary and rules, then loop through the password file attempting to crack each entry. If a match is found, the word is written to a results file (`john.pot`). If there are no more passwords to crack, the serial portion of John then invokes its shutdown, and exits. Our code then picks up any results, puts them back into the work message, adds some timing information, and sends the results back to the dealer. Then calls *MPI_Recv()* again, blocking until more work is available.

The dealer simply accepts the results from the workers, logs the result, and gives it some more work.

Termination

Once all passwords are processed, the dealer sends to each worker that is requesting more work a special shutdown message. Once all the workers are told to shutdown, the dealer then computes the final statistics, writes them to the results file and then enters a barrier and waits for all processes to finish up. Upon receiving the shutdown message, the worker cleans up and then enters the barrier. Once all processes are in the barrier, all processes exit and the MPI communicator group shuts down gracefully.

RESULTS

Expected

The performance metric used for John is "cracks per second". This is not actually the number of passwords cracked, but really should be called "attempts per second", since it refers to the number of times per second a dictionary word is encrypted and compared to the password to see if there is a match.

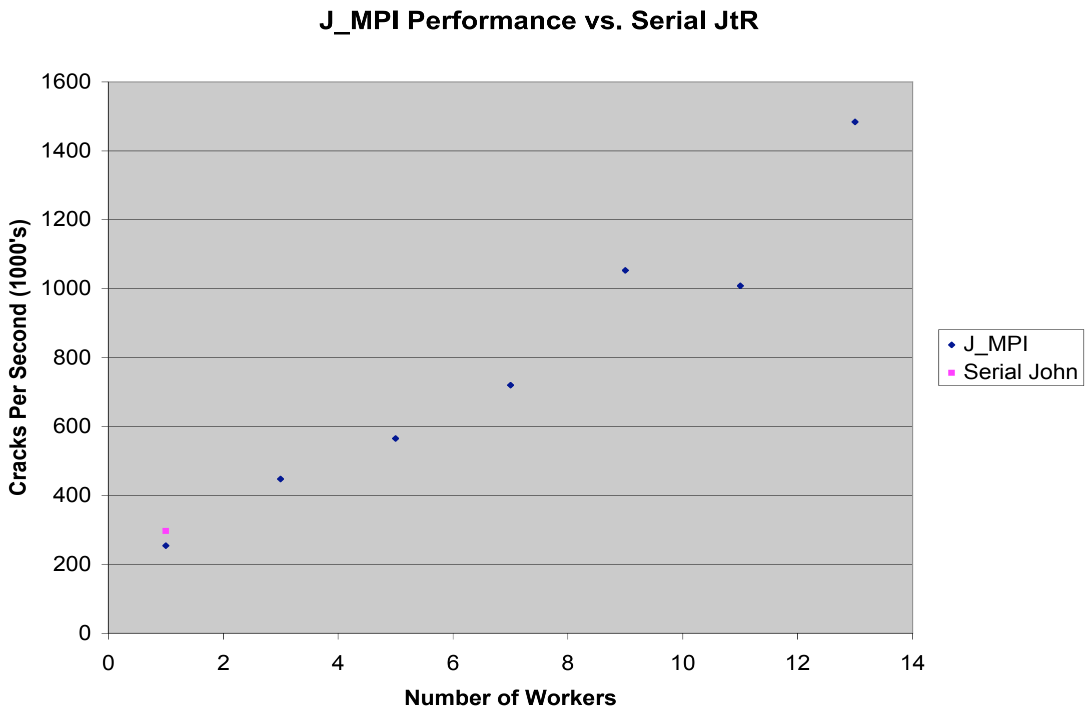
We expected to achieve a linear speedup of the cracks per second with respect to the number of processes applied to the job. This implementation of a parallel John the Ripper is embarrassingly parallel, with minimal communication, and a single synchronization at the very end. Communication only happens between the dealer and workers. The messages themselves are quite small.

Since the startup time of the serial John requires loading the dictionary file for each set of passwords, it was a concern that this would slow the program down. However, initial testing showed that the time spent cracking the passwords overwhelmed the startup and shutdown times.

Actual

We ran the first test on cluster2 with 2, 4, 6, 8, 10, 12, and 14 processes. `Cluster2` is a "beowulf" cluster of 32 systems. Each node has two 2.6GHz Intel® Xeon™ processors with 3GB of memory attached by a 533MHz front-side bus. Access to the nodes is controlled by the Portable Batch System (PBS) [6]. Each compute node in the cluster initializes two processes (one for each CPU). Since the dealer does not do any password cracking itself, one node was subtracted from the performance computation.

The first test run was done with 15 known and easy passwords.



The regression line on the above graph demonstrates our expected linear speed-up. As expected, the two process (1 dealer, 1 worker) test ran slightly slower than a single serial test. While the speedup wasn't 100% for each process added to the test, the speedup was consistent – a little more than 1/3 improved performance for each node added.

Table 1: Test #1

Nodes	Serial	1	3	5	7	9	11	13
$^{cracks}/_{sec}$	296947	253996	447864	564928	719649	1053290	1008511	1483690

Speedup	86% 0.86x	151% 0.50x	190% 0.38x	242% 0.35x	355% 0.39x	340% 0.31x	500% 0.38x

A question was raised whether or not the dealer would be become a bottleneck. Additional timing was added to calculate the amount of time spent by the dealer just waiting for input. With 3 workers on the above list of easy passwords, the total run time was 17 seconds. The dealer spent 12.4 of those seconds blocked waiting for input.

The second test was an attempt to validate the security of the existing College of Engineering (CoE) passwords. Besides periodically running the entire password file through John, they also test all new passwords against it. Running John against the CoE password file should not expose any weak passwords, our test differing only by the word list being used.

In order to minimize security risks, we cleaned the CoE password file, removing all information except the password hash. There were 3,893 password entries in the file with 2,391 unique salts. Unfortunately, that was still too much work for my allowable CPU quota, and the job was killed after about 15 hours:

```
PBS: job killed: cput job total 173522 secs exceeded limit 172800 secs
```

172,800 seconds is 2880 minutes, or 48 hours. This run was able to process 274 passwords, with only one successful crack.

Another test with fewer passwords (200 entries with 196 unique salts) was performed. Due to an error in our code, we were not able to get the total cracks per second metric, but it was able to complete the file in 2:08:03 wall clock time, with a total of 30:59:27 compute time. During this run, five accounts were found with weak passwords, three of which were already shutdown. The dealer spent 91% of its time waiting for a worker to communicate.

CONCLUSIONS

When first investigating this project, there were two primary goals: converting a serial version of John into a parallel version while making minimal modifications ("wrapping" the code) and providing a tool for system administrators to quickly check their accounts for easy passwords.

The changes made to the serial code were minimal. By replacing the entry point (`main` with `worker_main`) and adding some helper functions to get information out of John directly, this was achieved. Some other changes to the recovery system were necessary as John does not expect simultaneous invocations. Adding the process id to all files allowed multiple instances to be run on the same machine (as they are on `cluster2`). Although this invalidated John's recovery system, this was acceptable since the parallel wrapper could implement its own.

When running John against the College of Engineering's password file, it validated that the existing security measures are sufficient for preventing against a dictionary attack.

FUTURE WORK

The original goal of this project was to perform minimal modifications to John. This was successful, but at a performance cost. Each time the worker tries to crack a password, it needs to re-initialize itself. This causes a delay as the worker loads and indexes the dictionary and the rules. This should be done only once. Also, John outputs its performance statistics (cracks per second) to the console via standard error. This forces our program to grab hold of standard error and redirect (possibly vital) system information. It would be better to grab hold of that information directly, allowing debugging and error messages be displayed to the console. Both of these tasks would require more in-depth modification to John's code than we felt wise to do at this point in time.

We used the latest version of John that was tested, and known to be stable (version 1.6). However, a newer development version has been released (1.6.37) that is supposed to have many performance improvements in it. We made no effort in trying to improve the performance of the serial code, however. Since this implementation is almost embarrassingly parallel, we felt it was best not to modify the serial password encryption implementation and focus on making John v1.6 parallel.

Version 1.6 is unable to compile on machines with 64-bit architecture due to some low-level bit twiddling to simulate 64-bit operations. As 64-bit machines become more and more common, replacing the software emulation of the 64-bit arithmetic with a hardware implementation may gain significant improvements.

REFERENCES

[1] Solar Designer. John the Ripper, 2003. `http://www.openwall.com/john/`

[2] UNIX crypt(3) Man Page

[3] Perrine, T. and Kowatch, D. Teracrack: Password Cracking using Teraflop and Petabyte Resources. SDSC, 2003.

[4] D4 B0rg., dkbf. `http://dkbf.sourceforge.net/`

[5] Lim, R. Parallelization of John the Ripper (John) using MPI. UNL, 2004

[6] OpenPBS is a flexible batch queuing system developed for NASA in the early to mid-1990's. http://www.openpbs.org/