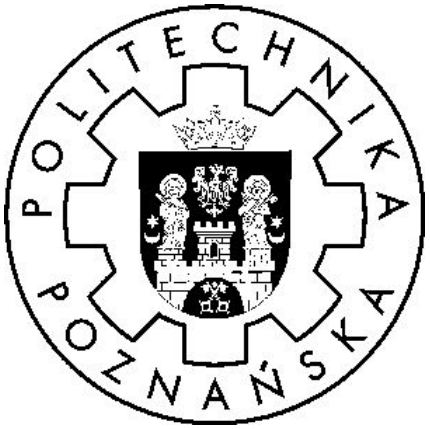

Łamanie haseł metodą brutalnej siły

Przetwarzanie Rozproszone, lab. nr 3

Zakres ćwiczenia

Głównym celem ćwiczenia jest demonstracja łatwości, z jaką można rozproszyć niektóre algorytmy, a równocześnie, że rozproszenie nie zawsze musi mieć sens. W ramach ćwiczenia napiszesz wersję sekwencyjną i rozproszoną prostego algorytmu łamania haseł z użyciem podejścia brutalnej siły (*brute force*). Następnie zmierzysz czasy potrzebne do złamania hasła o określonej długości. Powinieneś/powinnaś zaobserwować, że dla prostych haseł, wersja sekwencyjna może być szybsza od rozproszonej, jednakże już dla haseł pięcioliterowych widać przewagę podejścia rozproszonego.

Łamanie haseł

Metoda brutalnej siły w łamaniu haseł jest bardzo prosta. Załóżmy, że mamy dane hasło $H \in S$, gdzie S oznacza zbiór wszystkich potencjalnych haseł. Hasło zostało zaszyfrowane przy pomocy funkcji F, w efekcie czego otrzymujemy zakodowane hasło F(H) = X. Aby odgadnąć hasło, bierzemy wszystkie elementy $E \in S$ i każdy z nich kodujemy funkcją F. Jeżeli F(E) =X, oznacza to, że E=H.

Zwykle hasła kodowane są w ten sposób, by ich złamanie metodą brutalnej siły było niemożliwe lub przynajmniej bardzo trudne obliczeniowo - to znaczy, by zbiór S był obszerny (np. wymagania, by hasła były długie, zawierały znaki specjalne i cyfry…), oraz by metoda szyfrowania F była trudna obliczeniowo. Ponieważ jednak zajęcia trwają tylko półtorej godziny, specjalnie wybraliśmy taką metodę szyfrowania i takie ograniczenia, by można było złamać hasła jak najszybciej (złamanie najprostszego hasła powinno zająć mniej niż sekunda).

W naszym przypadku hasła będą należeć do zbioru obejmującego *n* literowe ciągi składające się z samych małych liter, natomiast jako metody szyfrowania użyjemy funkcji *crypt* z biblioteki o tej samej nazwie. Dodatkowo, funkcja *crypt* bierze dwa argumenty: pierwszy, to hasło do zakodowania, a drugi, to tzw. *salt*, czyli dwubajtowy ciąg mający

zwiększyć trudność odgadnięcia hasła. Efektem jest hasło w postaci na przykład bahAZ9Hk7SCf6, przy czym w tym zaszyfrowanym haśle dwa pierwsze bajty zawsze są równe argumentowi *salt* (w naszym przypadku, ***ba*** *w* ***ba***hAZ9Hk7SCf6) i powinny zostać usunięte z hasła przed jego zapisaniem. Dla ułatwienia odgadnięcia hasła, my tej czynności nie wykonaliśmy.
W efekcie, wiemy, że “bahAZ9Hk7SCf6” jest efektem wykonania funkcji *crypt*(H, “ba”). Jeżeli dalej powiemy, że H jest ciągiem czteroznakowym (same małe litery), wystarczy w pętli sprawdzić wszystkie możliwe czteroliterowe ciągi, tzn. wykonać X = *crypt*(“aaaa”, “ba”), sprawdzić czy X równa się “bahAZ9Hk7SCf6”, jeżeli nie sprawdzić ciąg “aaab”, “aaac” i tak dalej aż do “zzzz”.

Zadanie do samodzielnego wykonania

Dany jest szkielet programu do szyfrowania, a w nim w komentarzu lista haseł do odgadnięcia. Są to, kolejno, hasła cztero, pięcio i sześcioliterowe. Należy najpierw napisać wersję sekwencyjną: zmodyfikować szkielet tak, by w pętli szyfrował kolejne ciągi (“aaaa”, “aaab”...”zzzz”) i za każdym razem sprawdzał przy pomocy strcmp, czy rezultatem szyfrowania jest hasło. Jeżeli tak, hasło jest odgadnięte. W wersji sekwencyjnej należy nie używać żadnych funkcji MPI.

Zakładając, że program ten nazywać się będzie crack_seq.c, wersję sekwencyjną kompilujemy przy pomocy polecenia:

```
gcc -lcrypt crack_seq.c -o crack_seq
```

Następnie mierzymy czas do złamania hasła:

```
time ./crack_seq
```

Czynności te powtarzamy dla haseł pięcio i sześcioliterowych. Ponieważ hasło sześcioliterowe łamie się stosunkowo długo, w tym czasie można rozpocząć prace nad wersją rozproszoną. Należy zastanowić się, jak rozproszyć program, następnie skompilować przy pomocy `mpicc -lcrypt` i pomierzyć czas za pomocą

```
time mpirun -np <liczba_procesow> ./crack
```

Pamiętaj o kilku wskazówkach:

1. W wersji sekwencyjnej trzeba usunąć wszystkie funkcje MPI
2. W wersji równoległej można przyjąć na sztywno liczbę procesów np. 26
3. W wersji równoległej, jeden proces może odgadnąć hasło na samym początku, a inne mogą wciąż próbować odgadnąć. Zawyża to czasy w stosunku do wersji sekwencyjnej - zastanów się, jak to obejść.
4. Nie należy ani alokować miejsca, ani tym bardziej potem zwalniać pamięci związanej z ciągiem x (rezultatem funkcji crypt). Pamięcią zarządza wewnętrznie biblioteka crypt
5. Pamiętaj, że porównujemy łańcuchy funkcją strcmp, która zwraca ZERO, gdy ciągi są równe. Nie używamy == !!

6. Zmienna stro zawiera hasło w postaci zaszyfrowanej, zmienna cmp zawiera kolejne ciągi, które będziemy szyfrować. Zmienna cmp to tablica o rozmiarze zawsze o jeden większym niż rozmiar poszukiwanego hasła (bo łańcuchy w C kończą się znakiem o kodzie 0, więc musimy na ten znak także zarezerwować miejsce).

UWAGA! Do zadania dołączony jest plik Makefile, w którym zawarte są polecenia kompilacji programu.