AK - plan projektu

Porowski Wiktor

8 września 2021

Spis treści

1 ETAP 1

1.1 Wprowadzenie

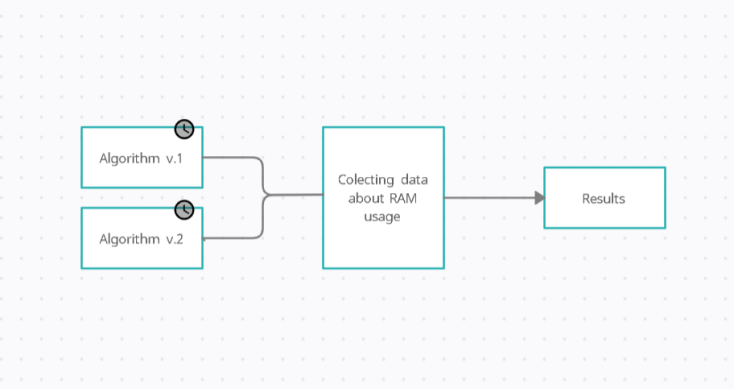
Celem projektu jest badanie konsumpcji pamięci RAM przez oprogramowanie tzn. stworzenie kilku wersji algorytmu w mniej lub bardzij optymalny sposób zużywającego zasoby RAM.

1.2 Plan projektu

Eksperyment będzie polegał na uruchamianiu różnych wersji algorytmu i obserwacji zużycia RAM przez dany proces. Możliwe będzie

to np. za pomocą narzędzi w Linux.
Następnie porównanie wyników z teorytecznymi przewidywaniami a także badanie rezultatów optymalizacji.
Przydatnym narzędziem może okazać się także debbuger (np. do obserwacji stosu przy wywołaniach rekurencyjnych).
Wybrane problemy to:
1. Obliczanie n-tej liczby ciągu Fibonacciego.
Algorytm ma za zadanie obliczyć wartośc liczby na wskazanej pozycji w ciągu Fibonacciego. (skupienie się na iteracyjnej oraz rekurencyjnej wariancji rozwiązania problemu).
2. BFS-przeczhodzenie grafu w szerz. Dzięki różnym sposobom reprezentacji grafów w pamięci będzie możliwa optymalizacja.
Językiem który zostanie użyty do implementacji algorytmów będzie C++.
System operacyjny - Linux.
Literatura - "Wprowadzenie do algorytmów" Thomas Cormen, "Architektura komputerów" Janusz Biernat, inne źródła.

1.3 Graf pomocniczy

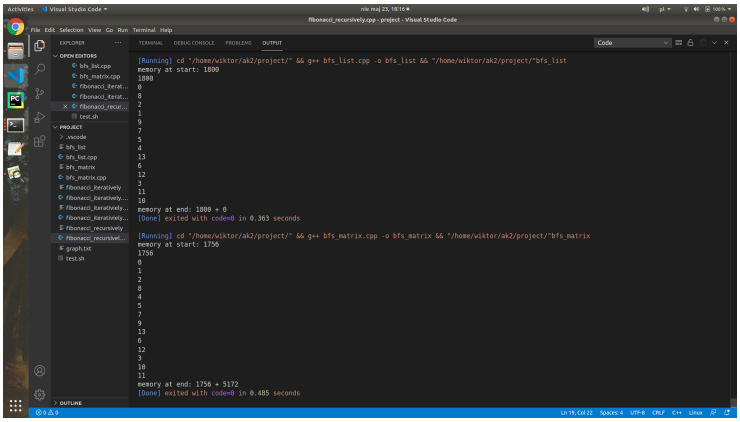


Rysunek 1: Architektura programu

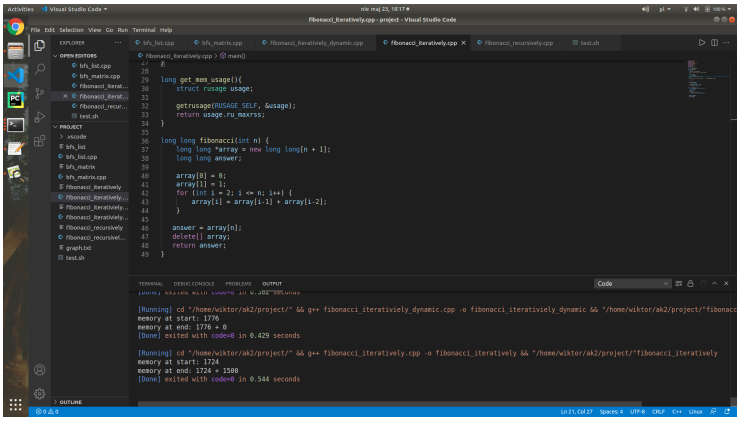
2 ETAP 2

2.1 Proof of concept

Został napisany kod źródłowy (dołączony w katalogu projektu) oraz zostały dokonane wstępne pomiary zużycia RAM dla różnych wersji algorytmów rozwiązujących dany problem.
Pomiary są wykonywane przez program podczas jego działania i są przekazywane na konsole.



Rysunek 2: BFS



Rysunek 3: n-ty wyraz ciągu fibonacciego

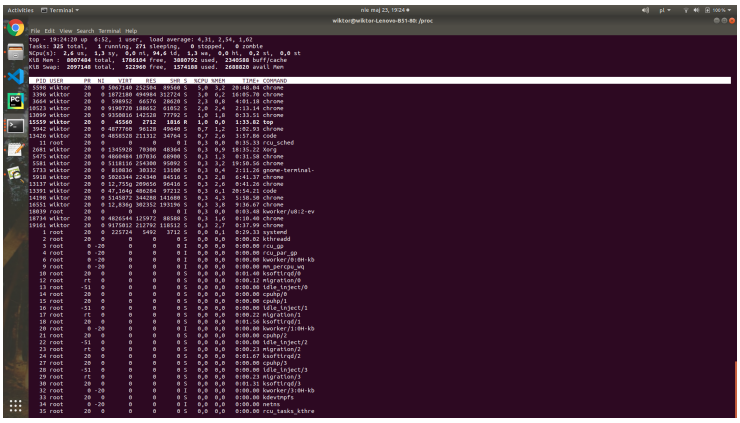
2.2 Opis testów

Mówiąć o pamięci potrzebnej dla naszego programu mówimy o pamięci potrzebnej na kod, zmienne, dodatkowe alokacje pamięci np (używając malloc) a także współdzielone biblioteki.
Uzasadnione jest aby badając zużycie RAM naszego programu pominąć współdzielone biblioteki ponieważ nie są to zasoby pamięciowe potrzebne specjalnie dla naszego procesu.
Wiele procesów w systemie operacyjnym korzysta z współdzielonych zasobów jak bibliteki.
Tak więć założeniem co do pomiarów jest że chcemy zbadać maksymalną ilość pamięci jaką nasz program może użyć pomijając współdzielone biblioteki. W tym celu możemy skorzystać z getrusage() - zwraca statystyki wykorzystania zasobów dla procesu czyli suma zasobów używanych przez wszystkie wątki wprocesu.
Zgodnie z założeniami nteresujece dla nas jest ru-maxrss czyli maksymalny używany rozmiar.

Rysunek 4: getrusage()

Praca krokowa w przypadku getrusage():
-Przed rozpoczęciem algorytmu zmierzyć używane zasoby (uwzględnia współdzielone biblioteki)
-Wykonać algorytm
-Zmierzyć zasoby po wykonaniu algorytmu (gdy odejmiemy stan początkowy będziemy mieli informacje o faktycznych zasobach używanych przez nasz algorytm.

W dalszym badaniu RAM używanym przez nasz proces może przydać się pseudoplik /proc/ lub polecenie możliwe jest wydzielenie tak zwanej virtualnej pamięci potrzebnej dla naszego programu (uwzględniając współdzielone zasoby) - VIRT.
a także ubecnie używane zasoby specyficzne dla naszego procesu - RES.



Rysunek 5: /proc/

Praca krokowa w przypadku tej metody:
-Uruchomić różne wersje algorytmów jako osobne procesy
-Znaleźć ich PID
-Dokonać obserwcji z użyciem /proc/.

3 ETAP 3

3.1 Test

Plik mp4 z wykonania testów znajduje się w katalogu.

Test 1:
bfs-list 2440 + 0 kb
bfs-matrix 2440 + 4520 kb
fibonacci-iterativiely-dynamic 2440 + 0 kb
fibonacci-recursively BRAK DANYCH*

*W przypadku elementu ciągu fibonacciego liczonego rekurencyjnie wywołań rekurencyjnych na stosie jest tak dużo że nie jest możliwe otrzymanie wyniku w rozsądnym czasie podczas gdy iteracyjna wersja nie ma z tym problemu.

Test 2:
bfs-list RES-1769 VIRT-14124
bfs-matrix RES-6932 VIRT-17952
fibonacci-iterativiely-dynamic RES-1732 VIRT-13988
fibonacci-recursively RES-1868 VIRT-13988

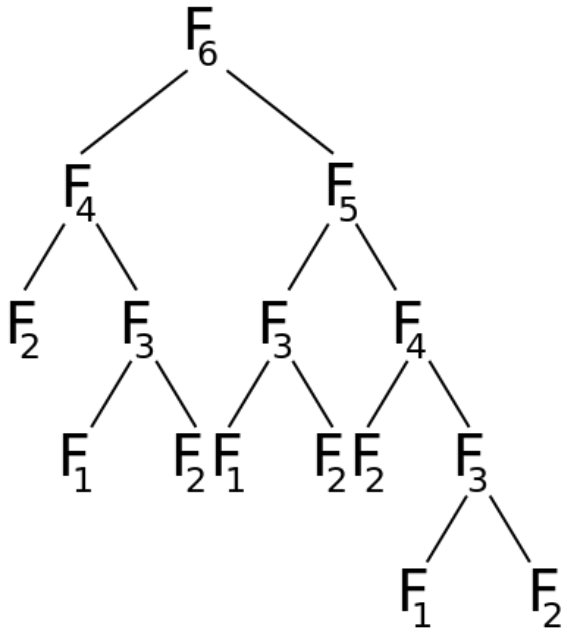
4 ETAP 4

4.1 Omówienie testu i wnioski

W przypadku elementu ciągu fibonacciego liczonego rekurencyjnie widać że zasoby pamięciowe programu są takie same jak w iteracyjnej wersji.
Uzasadnienie:
W przypadku podejścia iteracyjnego (fibonacci-iterativiely-dynamic) ilość wymaganego miejsca pozostaje taka sama. Stąd złożoność pamięci jest O (1).
W przypadku rekurencyjnej implementacji fibonacciego wymagana przestrzeń jest proporcjonalna do maksymalnej głębokości drzewa rekurencji, ponieważ jest to maksymalna liczba elementów, które mogą znajdować się w niejawnym stosie wywołań funkcji. Głębo-

kość drzewa rekurencyjneego w tym wypadku jest O(n).
Maksymalny wyraz ciągu który możemy policzyczyć wykorzystując największy podstawowy typ danych w c++ (long long)
to 92 wyraz = 7 540 113 804 746 346 429 następne wyrazy ciągu nie zmieszczą się w long long.
Tak więc wracając do złożoności pamięciowej O(n) nasze n maksymalnie jest równe 92 jest to za mało aby dodatkowa potrzebna pamięć (na stosie) była widoczna w tej metodzie pomiarowej.

Skoro głębokość drzewa to tylko O(n) dlaczego więc nie otrzymujemy wyniku w rozsądnym czasie?
głębokość drzewa to faktycznie O(n) jednak liczba wszystkich elementów drzewa - czyli wartości które musimy przetworzyć to $O(2^n)$



Rysunek 6: rekurencja

Jeżeli chodzi o korzyśći z wykorzystania listowej implementecji grafu w stosunku do macierzowej będą one szczególnie wyraźne dla

grafów rzadkich ponieważ implementacja macierzowa posiada złożoność pamięciową $O(V^2)$ (V - liczba wierzchołków)
natomiast listowa O(E) (E - liczba krawędzi). Dla rzadkiego grafu spójnego w przybliżeniu E rowne V tak więc wtedy O(V).
Dla grafu bardzo gęstego gdzie E rowne V^2 korzyści się niwelują.