Nome: Beatriz Viana Costa

NºUSP: 13673214

Relatório EP1 - Algoritmos e Estruturas de Dados I (MAC0121):

Para a implantação do Exercício Programa foram criados três documentos: a EP1.h, que se trata da biblioteca, onde é definido um tipo e também onde se encontram os cabeçalhos das funções que serão utilizadas, o EP1funcoes.c, onde se encontram as definições das funções e por fim o EP1.c, que é basicamente o *main.*

A forma usada para tentar otimizar o cálculo da Conjectura de Collatz foi a alocação dinâmica de um vetor do tamanho do intervalo dado pelo usuário, dessa forma, a medida que a conjectura é calculada para cada número do intervalo em ordem crescente, o número de passos necessários para este número convergir para 1 é guardado no vetor. Assim, posteriormente, quando um número coincidir com um que já teve sua conjectura calculada, ou seja, quando ele for menor do que o número de entrada atual, a função *procura* é chamada, e a partir do índice do número procurado, retorna o número de passos deste que é somado aos passos já realizados no número de entrada.

Temos então as especificações de cada função:

1. *void \*cria(int inicio, int fim):* Rebe o início e o fim do intervalo e assim aloca um vetor que será usado posteriormente para guardar a quantidade de passos.
2. *void destroi(info \*v):* É utilizada no final do programa. Recebe o vetor que foi alocado anteriormente e libera sua memória.
3. *int conjec\_collatz(int n):* Recebe o número atual do intervalo e retorna o resultado após a função ter sido aplicada somente uma vez.
4. *int procura(info \*v, int n, int inicio):* Recebe o vetor onde a quantidade de vezes que a conjetura de Collatz foi aplicada é guardada, o número (n) o qual se deseja encontrar a quantidade anterior e o início do intervalo (*inicio*).

Encontramos o índice (i) através da relação *i = n - inicio* e então retornamos o número de passos.

1. *int calcula(int n, int inicio, info \*v):* Recebe o vetor alocado, o início do intervalo e o número atual. Antes de fazer qualquer alteração no número recebido, a função o iguala à outra variável (*x*), pois o número de entrada deve ser guardado para uma posterior comparação. O índice (*i*) onde o número de passos deve ser guardado também já é calculado e é do formato *i = n – inicio*, onde *n* é o número atual, fornecido na entrada e *inicio* o começo do intervalo.

Depois entramos em um laço no qual permanecemos enquanto número não converge em 1. Este laço é composto por mais duas condicionais. A primeira é cumprida se o número atual (após as modificações da conjectura) é menor que o número de entrada, neste caso a função *procura* é chamada, e o número de passos é retornado e somado à quantidade já realizados.

Se não o número atual entra em outro laço onde a função *conjec\_collatz* é chamada e o número de passos é acrescido.

A linha de comando utilizada para verificar o tempo de execução foi *time./<TESTE>*, que retorna três valores *real*, *user* e *sys*. O *real* se refere ao tempo total decorrido durante a medição, o *user* ao tempo em que a CPU ficou ocupada processando as instruções do programa e já o *sys* é o tempo gasto pela CPU no modo kernel.

O teste foi realizado no intervalo [1, 113.382], onde ocorreu *segmentation fault* por *core dumped,* ou seja, o programa tentou acessar memória que não lhe pertencia. E os tempos retornados foram:

|  |  |
| --- | --- |
| *real* | 0m7,244s |
| *user* | 0m0,142s |
| *sys* | 0m0,293s |

De maneira geral o programa otimizado levou o mesmo tempo para ser executado em relação ao programa sem otimização alguma, sendo alguns centésimos de segundo mais rápido. Contudo, nos casos onde o intervalo era pequeno, como por exemplo [1, 100], o programa sem otimização era muito mais rápido, levando apenas dois segundos (*real)* para ser executado enquanto o otimizado levava em média 8 segundos.