Trabalho Prático 3
Plano de Dominação Global do professor WM. Jr

Gilson Urbano Ferreira Dias - 2015430959
guf@ufmg.br

1. Introdução

O problema proposto é: dado uma matriz MxN onde M é o número de linhas e N o número de colunas, essa matriz representa cidades e seu número de habitantes o objetivo é dominar o maior número de habitantes possível utilizando uma arma de dominação, mas essa arma tem um efeito colateral, se a cidade $C(i,j)$ é escolhida, as cidades $\{C_{i,j+1}; C_{i,j-1}; C_{i+1,k}; C_{i-1,k}\}\forall k \in \{1, \dots, M\}$ são destruídas, em outras palavras, as cidades das colunas ao lado e todas as cidades da linha acima e abaixo a escolhida são destruídas. O modo encontrado para solucionar o problema foi encontrar a maior quantidade de cidades de cada linha da matriz colocando esse resultado em um vetor e encontrar a maior quantidade desse vetor isso é feito pela função sumLine(), ela é melhor explicada em Funções e Procedimentos, abaixo temos um exemplo da solução do problema.

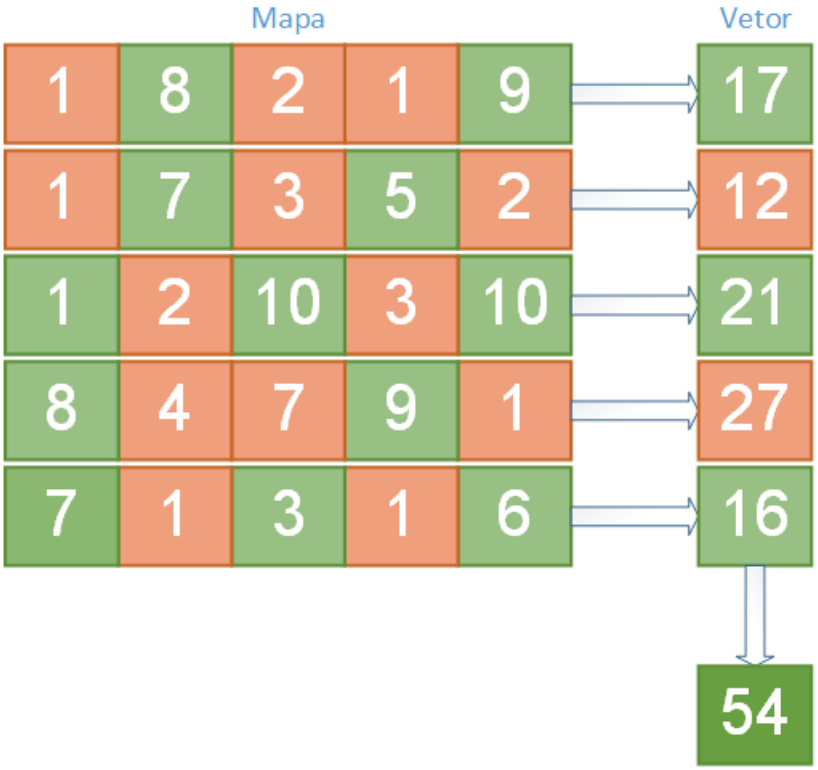

Figura 1 - Solução do Problema

Outro objetivo é paralelizar o programa utilizando as técnicas de programação dinâmica para solucionar o problema, a solução proposta facilita essa paralelização pois encontrar o número máximo de habitantes em cada linha da matriz pode ser efetuada de maneira independente, pois as linhas não são dependentes entre elas, então pode se utilizar a função que soluciona as linhas de maneira paralela entre as linhas da matriz.

2. Solução do Problema

A implementação foi criada a partir da definição do Tipo Abstrato de Dados (TAD), com três estruturas, a primeira de nome Control armazena as variáveis M, N, que irão controlar e definir a execução do programa, indicando: Quantidades de Linhas e quantidade de colunas, respectivamente, na segunda estrutura Grid temos a representação do mapa, a estrutura Arg é utilizada para unir os parâmetros em um único tipo isso é feita devido a limitação imposta pela biblioteca que será utilizada para que as threads funcionem de maneira correta.

2.1 Funções e Procedimentos

As funções foram declaradas da seguinte forma no Tipo Abstrato de Dados:

void allocGrid(Grid *map, Control variables); : Essa função aloca memória dinamicamente para que seja recebida a entrada que representa o mapa, durante a alocação o mapa é inicializado com valor zero em todas suas posições.

void freeGrid(Grid *map, Control variables); : Função que será utilizada no final da execução do programa e tem o objetivo de liberar a memória alocada pela função allocGrid();

void getGrid(Grid *map, Control variables); : E a função que pela entrada padrão obtém o mapa que representa as cidades e o salva para uso futuro.

int max(int a, int b); : Função simples que apenas retorna o maior número entre dois inteiros;

void *sumLine(void *arg); : Procurando a solução do problema a ideia foi de encontrar o ótimo de cada linha ou seja o número máximo de cidades possíveis em cada linha da matriz que representa a cidade, colocar o ótimo em um vetor, com isso podemos aplicar o mesmo processo efetuado nas linhas no vetor com isso obtemos a solução ótima do problema. Para encontrar o número máximo de cidades em cada linha é feita a seguinte comparação: se linha[i] + linha[i + 2] é maior do que linha[i + 1], ou seja caso a soma dessas posições das linhas seja maior do que linha[i + 1] colocamos o resultado da soma em linha[i] senão linha[i+1] é propagada, a seguinte equação de recorrência descreve a ideia para a linha inteira da matriz:

$$linha[i] = \max(linha[i] + linha[i+2],\ linha[i+1)\text{, com } i \text{ variando de } n-3 \text{ a } 0$$

2.2 Programa Principal

No programa principal são declaradas algumas variáveis importantes, uma delas, que será utilizada ao longo de toda a execução, é a variables do tipo Control. É feita a leitura da entrada padrão, pegamos o mapa e armazenamos.

Após a leitura da entrada padrão, alocamos variáveis para que possamos encontrar o número máximo de cidades de cada linha da matriz, são criadas a threads e passada a função que soluciona o problema, isso de acordo com a

limitação imposta do número de threads, o processo é repetido até que todas as linhas sejam trabalhadas, por fim criamos um vetor com o máximo de cada linha e repetimos o processo então é colocado na saída padrão o número máximo de cidades encontradas por fim a memória que foi alocada durante a utilização é liberada.

2.3 Organização do Código, Decisões de Implementação e Detalhes Técnicos

O código está dividido em três arquivos principais: TP3.c e TP3.h implementam o Tipo Abstrato de Dados enquanto o arquivo main.c implementa o programa principal.

O compilador utilizado foi o GCC 5.4.0 no Sistema operacional UBUNTU 16.04.

3. Análise de Complexidade

void allocGrid(Grid *map, Control variables); : Função simples que tem dois loops aninhados que alocam e inicializam, com isso tem uma uma complexidade de tempo O(M*N), onde M*N é o tamanho da entrada, e O(M*N) para espaço;

void freeGrid(Grid *map, Control variables); : Com apenas um loop para efetuar a liberação de memória temos uma complexidade de tempo O(M) e O(1) para espaço.

void getGrid(Grid *map, Control variables); : Assim como na allocGrid() essa função tem complexidade O(M*N), onde M*N é o tamanho da entrada a ser lida e O(1) para espaço.

int max(int a, int b); : Temos apenas comparação nessa função com isso temos a complexidade O(1) para espaço e tempo;

void *sumLine(void *arg); : Essa é a função principal que soluciona o problema ela percorre as linhas da matriz e encontra o maior número da habitantes, pensando em termos de linha ela tem uma complexidade O(N) para tempo e O(1) para espaço, mas para resolver o problema ela precisa ser chamada M + 1 vezes com isso chegamos a complexidade O(M*N) para tempo e O(1) para espaço.

4. Avaliação Experimental

4.1 Testes

Vários testes foram realizados com o programa de forma a verificar o seu perfeito funcionamento. Os testes foram realizados em Intel Core i5, 6Gb de memória com um Intel Core i5-4210U rodando a 1.70GHz, e sistema operacional Ubuntu 16.04 LTS.

Os testes foram repetidos dez vezes para cada entrada e então foi feita uma média do tempo de execução para que fosse gerado os gráficos a seguir.

O primeiro teste tem como objetivo ser a base para futuras comparações, nele temos o número de threads igual a 1 durante todo o teste, variamos o número de linhas e colunas da matriz de entrada, o resultado como podemos observar na figura 2 é um gráfico com forte tendência quadrática o que já se era de esperar pela complexidade do programa, foi traçada uma linha de tendência polinomial de grau 2 na cor vermelha que modelou muito bem a situação.

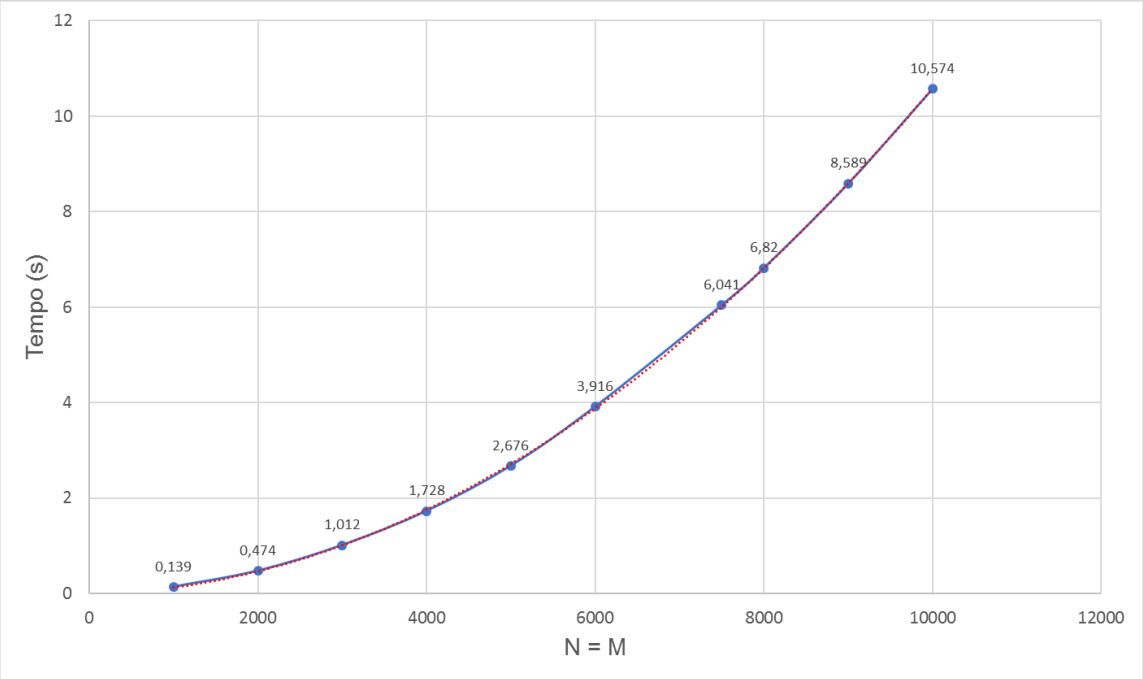

Figura 2 - Número de Threads = 1

Para o segundo teste foi verificado que o número de threads no computador de testes é igual a quatro, esse foi o número fixado para os threads, então foi repetido o teste anterior com o novo número de threads, Figura 3.

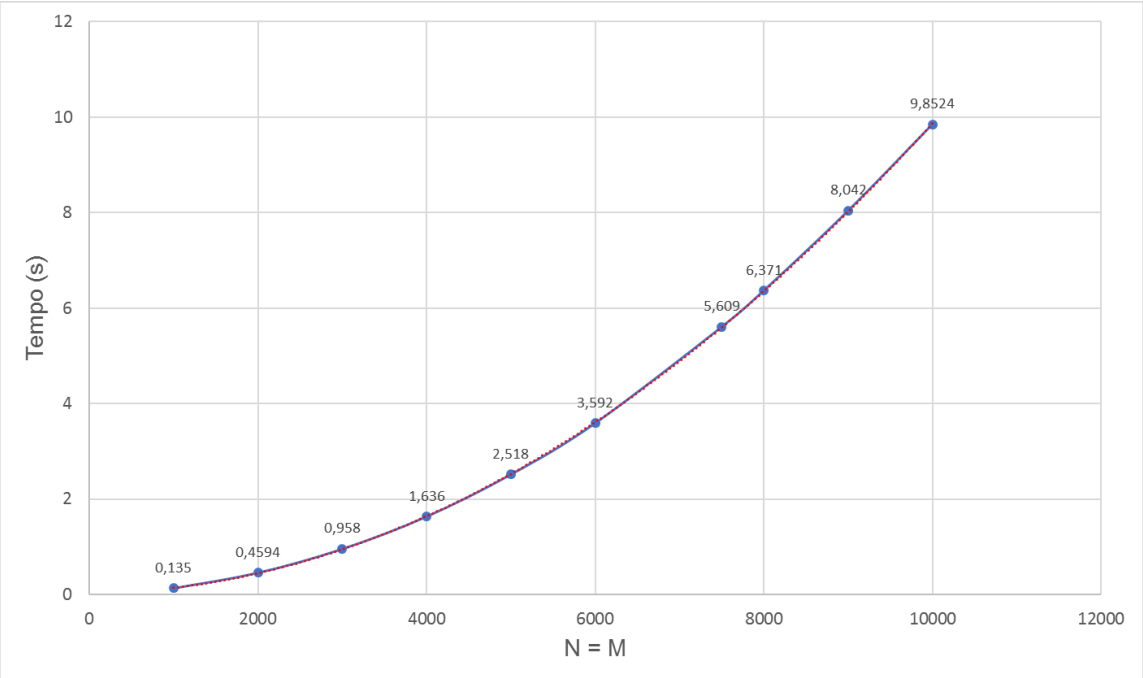

Figura 3 - Número de Threads = 4

O resultado foi um gráfico parecido com o apresentado na figura 2, a diferença foi a redução no tempo de execução em alguns casos mais de meio segundo para a mesma entrada, mostrando que utilizar o número máximo de threads do computador tem o efeito de acelerar na execução do programa, nesse gráfico também foi colocada uma linha de tendência polinomial de grau 2 e modelou muito bem o gráfico obtido como já era esperado.

Um terceiro teste foi feito para verificar até que ponto aumentar o número de threads tem o efeito de fazer com que o tempo de execução do programa diminua, para isso utilizamos a entrada com N e M igual 10000 cidades geradas aleatoriamente, começamos com uma thread e repetimos até o número de 10 threads.

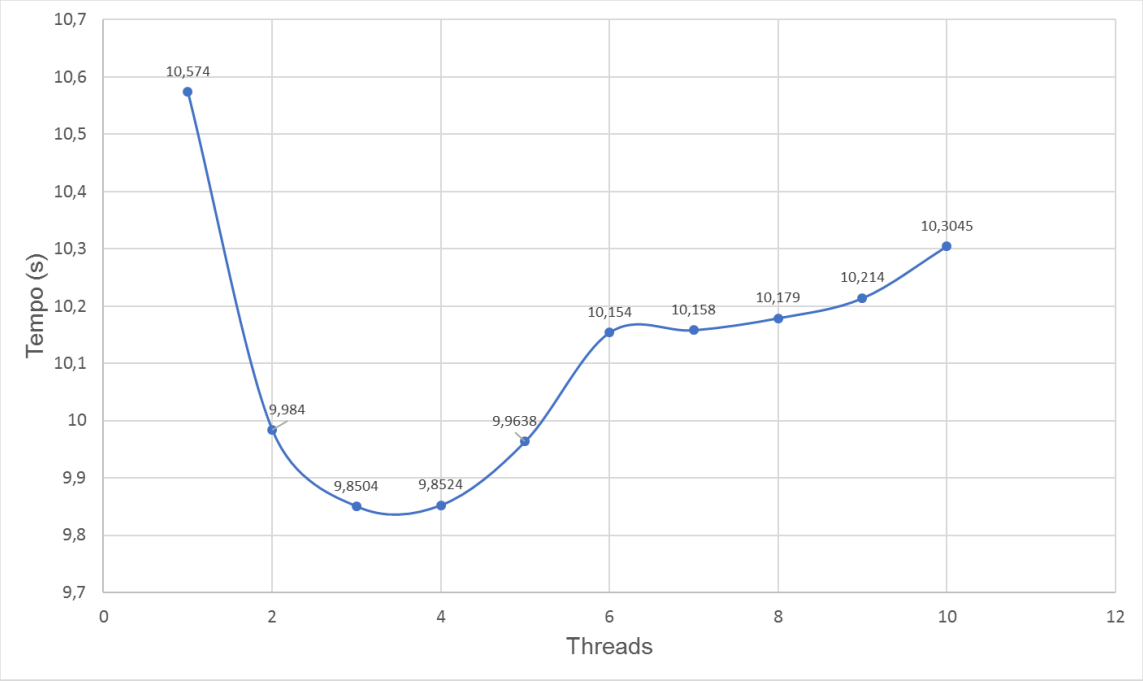

Figura 4 - N = M = 10000

O resultado é um gráfico com formato bem peculiar, mas podemos verificar alguns detalhes interessantes o tempo de execução diminui até chegar ao número de quatro threads que é o máximo que o computador de testes suporta, depois desse número o tempo de execução volta a aumentar, conferimos então que ultrapassar o número máximo de threads disponíveis no computador não garante ganho na queda do tempo de execução muito pelo contrário pode chegar a ocorrer um aumento no tempo de execução.

5. Conclusão

A implementação ocorreu de forma tranquila, as técnicas de programação paralela haviam sido estudadas durante as aulas de Algoritmos e Estruturas de Dados III, foi necessário algumas pesquisas e revisões sobre o assunto a fim de verificar alguns detalhes sobre as estruturas de dados a serem utilizadas e manipulação das funções propostas, sendo que a maior dificuldade foi entender o funcionamento da biblioteca pthread.h para que fosse possível a paralelização

do programa, mas por fim o trabalho pode ser concluído, com o resultado ficando dentro do esperado.

Para obter os valores e gerar os gráficos dos testes, foram feitas dez medidas do tempo de execução para cada entrada, e então foi feita a média simples desses tempos, a entradas utilizadas nos testes foram geradas de forma aleatória.

O resultado que pode ser conferido nos testes é que o número de threads influência no tempo de execução do programa sendo que o número ideal é o número máximo de threads que o computador possui, ultrapassar esse número não garante redução no tempo de execução, também é importante garantir a independência das atividades paralelizadas para que não ocorra conflito entre os threads caso isso ocorra é necessário um maior cuidado no tratamento dessas situações.

6. Referências

[1] Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 4ª Edição, Editora Pionera, 1999

[2] André Backes, Linguagem C: Completa e Descomplicada, 1ª Edição, Elsevier, 2012

[3] Kyle Loudon, Mastering Algorithms with C, 1ª Edição, Editora O'Reilly

[4] Dennis M. Ritchi, C Programming Language, 2ª Edição

[5] Robert Sedgewick, Algorithms, Algorithms in C, 1990, ADDISON-WESLEY

[6] Thomas H. Cormen ..., Algoritmos - Teoria e Prática, 2ª Edição, Elsevier, 2002

[7] Jon Kleinberg, Eva Tardos, Algorithm design, 1ª Edição, Pearson Education, 2006

[8] Jayme Luiz Szwarcfiter e Lilian Markenzon, Estruturas de Dados e Seus Algoritmos, 1994, LTC