Trabalho Prático 3: Plano de Dominação Global do Professor WM. Jr

Francis Carlos dos Santos

Matrícula: 2012022167

franciscarlossantos@gmail.com

1 Introdução

Encontrar a solução ótima para problemas de elevado grau de complexidade, ou seja, problemas no qual o espaço de soluções é extremamente grande,é extremamente custoso computacionalmente. Para muitos casos de aplicações práticas heurísticas são aplicadas ou algum tipo de computação evolutiva para se conseguir um resultado bem bom mas, que não necessariamente é ótimo. Para problemas em que existe obrigatoriedade de se conseguir encontrar o ótimo são mais desafiadores e uma da técnica bastante empregada é a de Programação Dinâmica. Outra estratégia boa para se resolver esse tipo de problema seria combinar a solução em PD juntamente com aguma técnica de computação em paralelo.

No trabalho proposto, temos uma situação em que o professor WM quer dorminar todas as cidades do planeta, porém para cada cidade dominada as cidades vizinhas explodem. O conquinstador WM quer consquinstar o maior numero de cidades de tal forma a controlar o maior número de pessoas possível. Portanto, o objetivo deste trabalho é combinar as técnicas de programação dinâmica e programação em paralelo de forma a encontrar o maior numero de cidades possivél.

2 Solução do Problema

2.1 Entendimento do Problema e Abordagem

O problema consistirá de uma entrada, uma espécie de mapa, passada por meio de uma matrix de N linhas e M colunas, além de receber o numero de threads a serem utilizadas por meio da linha de comando. A saída consitirá apenas do valor ótimo impresso ao fim da execução. A entrada é lida de maneira sequencial. A princípio foi pensada uma solução basicamente de força bruta, o objetivo principal era de conseguir entender melhor o problema. Porém essa solução, obviamente se monstrou não factível pela demora causada pelo grande número de casos. No próximo passo tentei aplicar as idéias de uma lógica gulosa para facilitar o problema, essa lógica iria funcionar comparando os números e vendo se era maior que o seu adjacente até que o vetor fosse todo lido. Ao fim teriamos um vetor indicando as posições das cidades que fariam com que aquele vetor tivesse a soma máxima. Essa lógica tinha um problema bem básico que era, o aumento da complexidade para analisar casos em que o a entrada viria já ordenada. A partir desse momento todo o pensamento foi de se utilizar sempre as somas entre os números, afim de percorrer o vetor uma única vez e de uma forma que a ordem não influenciaria. Primeiramente Utilizou-se duas somas. A idéia básica era a de se comparar sempre dois valores e armazenar sempre o maior valor. Como mostrado na Figura 1. O primeiro passo é ler os primeiros valores. Em **(A)** temos a primeira iteração, nela armazena-se a soma dos dois extremos lidos e comparamos o primeiro valor com o segundo, pois nessa iteração consideramos que o primeiro valor lido tenha valor máximo. Na tabela da esquerda pode ser observada a soma máxima encontrada para cada solução ao decorrer da resolução. Em **(B)** temos a leitura do proximo valor, setas brancas indicam a leitura, e a comparação entre o a soma armazenada da ultima iteração 3 e o valor máximo encontrado na ultima iteração, nessa iteração a soma parcial armazeada é 9. Na parte **(C)** e ultima iteração o valor 9 é lido a nova soma $8 + 9$ é armazenada e a comparação entre a soma da última iteração e o máximo da atual é feita, nesse caso é interessante perceber que apesar de parecer reduntante, essa parte do código não está relicaionada com a posição dos números no vetor e sim com o máximo encontrado na execução anterior . Em **(D)** a função já saiu do laço, mas realiza uma última comparação entre a ultima soma computada, e o valor máximo, *Max(9,17)*. Pelo diagrama é simples perceber que a comparação para decidir qual soma é maior é sempre feita utilizando o valor da solução anterior, ou seja, basicamente estamos consultando uma tabela com soluçções de sub-problemas.

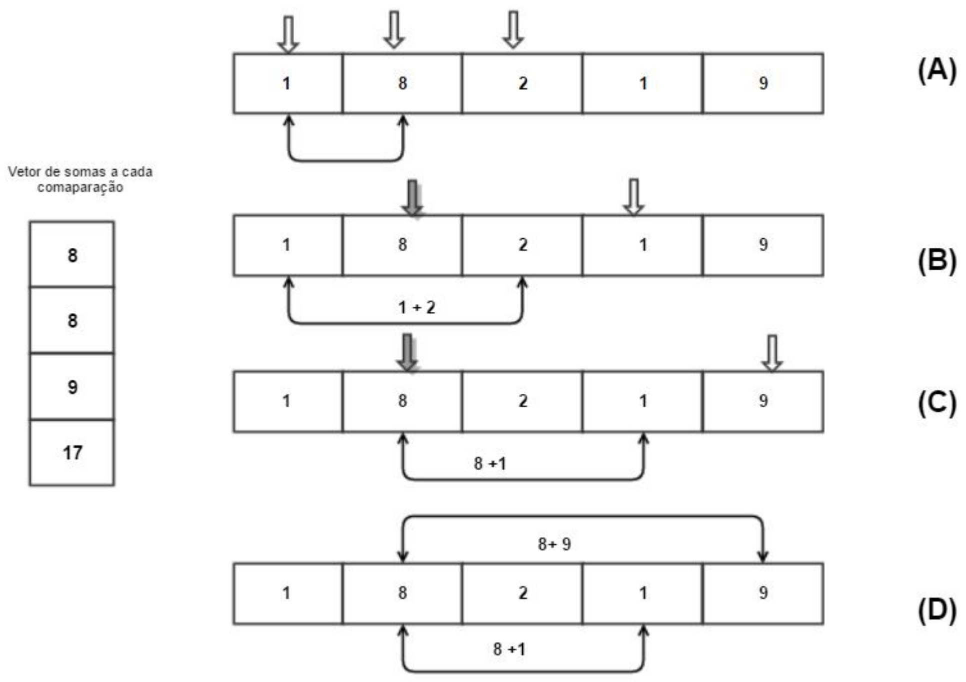

Figura 1: Esquema de solução para uma linha da entrada

A partir dessa solução pode-se inferir uma analise da recorrência, como pode ser observado a esquação de recorrencia mostrada na equação da Figura 2. Pode-se observar que os dois primeiros valores são passados como resultados diretos da função. A partir desse ponto todo as soluções são calculadas a partir dos maximos entre os antecendentes enquanto a soma a ser usada para comparação no próximo laço é também feita. A partir desse desenvolvimento foi possível chegar a equação de recorrência mostrada na Figura

3. Como explicado anteriormente fica claro a relação com os resultados dos subproblemas anteriores. A condição mostrada,, resulta do entendimento que para fazer a comparação precisamos de pelo menos dois numeros, logo essa equação de recorrência representa a solução para o espaço de solução de $2 < n < N$. Uma vez que para valores menores que três a única operação realizada seria uma unica comparação.

T(0) = s(0)

T(1) = s(1)

T(2) = max(v(1),s(0)) e s(0)+s(2)

T(3) = max(T(2),v(0)+s(2)) e T(2) + s(3)

T(4) = max(T(3),max(T(2)) +s(2)) e T(3) + s(4)

T(5) = max(T(4),T(3)+s(4))

Figura 2: Dedução da equação de recorrências

T(0) = s(0)

T(1) = s(1)

T(n) = max{ T(n-1),T(n-2) } e T(n-2)+s(n), 2<n < N

Figura 3: Equação de recorrências

2.2 Estrutura de Dados Escolhida

Com o objetivo de simplificar a solução do problema uma única estrutura de dados foi criada, com o fim de facilitar a implementação das threads e não comprometer o funcionamento do programa. A estrutura *dados* mostrada abaixo é utilizada na forma de um vetor de structs, embora essa ecolha tenha aumentado o custo de espaço pois para cada linha da matriz essa estrutura será alocada, ela facilita muito a construção da solução uma vez que pode ser utilizada diretamente para realizar a solução em paralelo e ainda evita a utilização de primitivas de sincronização, uma vez que a chance de escrita e leitura da mesma variável ao mesmo tempo é descartada. A struct possui também um paramêtro *Soma*, que serve para armazenar o máximo obtido para cada linha da estrutura *dados*.

2.3 Fluxo de funcionamento

O algoritmo proposto possui um fluxo de funcionamento muito simples, após a leitura da entrada o algoritmo busca as soluções linha por linha utilizado paralelismo de dados, isso por que a mesma operação é utilizada em varios blocos diferentes de dados. A solução é

Algoritmo 1: Estruturas de Dados dados

```
typedef struct{
  long int N,M
  int *populacao,thread_ID,soma

}dados;
```

então computada e em uma função similar *MaxPop* e o ótimo é calculado e impresso na saída padrão.

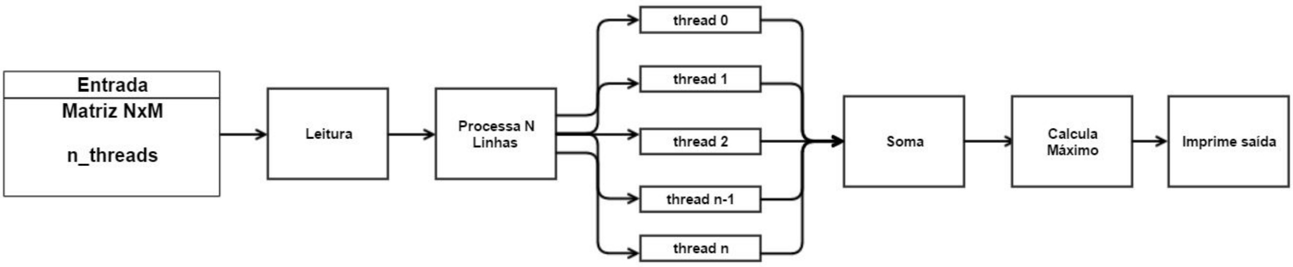

Figura 4: Esquema de funciomaneto do algoritmo.

3 Análise de Complexidade

Nesta seção, será apresentada a análise do custo teórico de tempo e de espaço para as principais funções do algoritmo.

3.1 Análise Teórica do Custo Assintótico de Tempo

Sejam N e M o numero de linhas e o numero de colunas na entrada rescpectivamente. A complexidade geral do algoritmo pode ser encontrada fazendo-se uma analise das principais funções. As funçoes *free* e *malloc* serão consideradas como tendo um tempo de execeução $O(1)$. Consideramos também todos os processos de atribuição e criação das threads, como tendo tempo complexidade de $O(1)$.

- **LeEntrada**: A função *LeEntrada* é resposánvel por ler a entrada e armazena-la na struct *dados*. Essa função perorre as $\mathsf{N} \times \mathsf{M}$. Portanto sua complexidade assíntotica de tempo é $O(\mathsf{N} \times \mathsf{M})$.
- **processa_threads**: Talvez essa seja a função mais importante do algoritmo, é responsavel por gerenciar a criação e finalização das threads. Para realizar essas operações essa função conta com um loop mais externo responsável por garantir que o as operações sejam executadas $\mathsf{N} \div \mathtt{n\_threads}$ vezes, sendo *n_threads* o numero de threads passado como paramâmetro. A ideia é que se o número de threads for múltiplo do número de linhas da matriz, cada thread terá que executar exatamente $\mathsf{N} \div \mathtt{n\_threads}$ e caso contrário cada thread executará o mesmo número de operações mais $\mathsf{N}\%\mathtt{n\_threads}$. Com isso cara loop executará no maximo N vezes. Logo N para a criação das threads e mais N vezes para verificar se o processo da thread está terminado. nesse caso temos que a complexidade é dada por $O($*max(N+M)*$)$.
- **MaxPop_thread**: Essa é a função chamada pela thread logo ela é executada no maximo N vezes e possui uma complexidade de $O(\mathrm{M})$,logo a complexidade assíntótica de tempo dessa função é $O(\mathrm{M})$. É simples perceber que para todos os processos a complexidade dessa função é entao $O(\mathsf{N} \times \mathsf{M})$
- **SumStruct**: Essa função tem como objetivo realizar o somatório de todos os ótimos encontrados para cada linha da entrada. Logo é evidente que sua complexidade é $O(\mathrm{N})$.
- **MaxPop**: Essa função é exatamente igual a que realiza as operações através das threads, porém essa é executada uma única vez portanto sua complexidade é $O(\mathrm{N})$.

Conclui-se assim que a complexidade assintótica de tempo do algoritmo é $O(|\mathsf{N} \times \mathsf{M}|)$.

3.2 Análise Teórica do Custo Assintótico de Espaço

Considerando a estrutura *dados* como a príncipal estrutura do algoritmo, pois essa é a única em que o espaço varia de acordo com a entrada. O maior peo dentro dessa estrutura vem da alocação da própria estrutua N vezes mas seu vetor interno *populacao*, que tem tamanho M. Como é facil perceber essa estrutura está intimamente relacionada com a matriz, ou mapa, da entrada. Portanto a complexidade teórica assintótica de espaço será $O(|\mathsf{N} \times \mathsf{M}|)$.

4 Análise Experimental

A análise experimental da implementação é mostrada pelas figuras, e. Para realizar os experimentos foi utilizado um gerador de casos teste. O tempo de execução foi obtido pelo terminal ao fim da execução do progama. O espaço utilizado em cada execução foi obtido por meio da utilização do Valgrind. Os testes foram realizados em uma máquina com processdor Core i7-4720HQ 2.6GHz e 8GB de memória ram.
Os testes foram realizados com o intuito de testar como o algoritmo responde a diferentes variações no número de linhas, número de colunas e com o número de threads. Na Figura 5 podemos observar um gráfico mostrando o tempo de execução relativo a variação do tamnaho da entrada e do número de threads. A linha azul mostra o caminho esperado, que foi indicado na análise teórica assintótica de espaço. É possível observar que as três curvas tem basicamente a mesma inclinação o que concorda com o resultado esperado. Embora seja possível notar alguma alteração grande nas curvas, tais fatos se deve aos processos relacionados a execução das threads, que são chamadas sistêmicas.

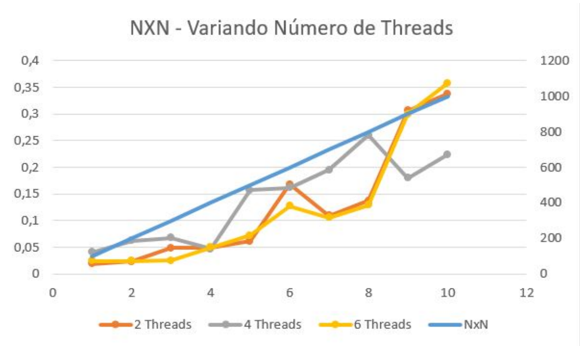

Figura 5: Gráfico mostrando a resposta do tempo(s) com a variação do número de threads e do tamanho da entrada.

Na Figura 6, podemos perdeber como o espaço alocado para o algoritmo varia de forma, em forma de parabola, como esperado pela análise téorica assintótica de espaço. Foi possível perceber também que o número de threads tem influência muito baixa na quantidade de memória utilizada, pelo menos para esse algoritmo. Nesse teste o tamanho da entrada foi fixado em NxN com o número de threads fixo. Outro teste utilizado foi a

Figura 6: Gráfico mostrando a variação de memória, em bytes, alocada para o programa com a variação no tamanho da entrada.

variação do número de colunas utilizando o número de linhas fixo em 10 e com threads nos tamanhos de 2,4,6. É possível perceber que para problemas muito pequenos a divisão de trabalho não apresenta grandes benefícios, isso pode ser explicado pelo fato de que, se a entrada for pequena e a divisão for grande o processador armazenará os dados na memória cache, o que pode causar maior tempo de execução. Após uma pequena anomalia os gráficos apresentaram um comportamento bastante estável e conforme o esperado quase linaer. Esse gráfico nos mostra também que o número de colunas da matriz de entrada não apresenta grande influência no tempo de execução do problema.

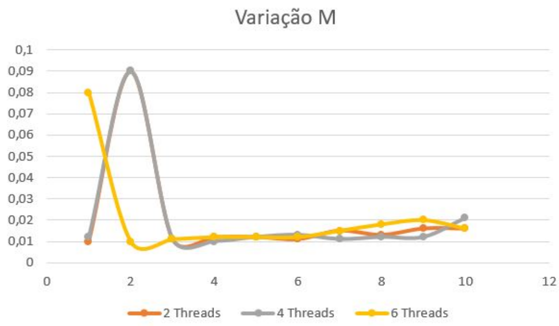

Figura 7: Resposta à variação do número de colunas da matriz de entrada.

Por último avaliamos a resposta do tempo em relação a variação do número de linhas. É fácil perceber a característica parábolica da curva do tempo. Assim como esperado pela análise teórica. Os paramêtros utilizados para esse teste experimento foram, 2,4 e 6 threads e um número fixo de 100 colunas na matriz de entrada. Ainda, durante a analise

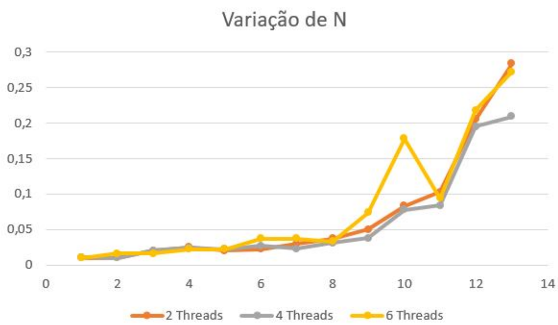

Figura 8: Gráfico mostrando a resposta do tempo(s) com a variação do número de linhas e threads.

experimental foram feitos testes utilizando o Valgrind. Para os exemplos *TOYs* passados, todos os testes obtiveram sucesso. Para entradas muito pequenas foram observadas pequenas leaks de 4 bytes de memória, os quais não consegui identificar a fonte.

5 Conclusões

Nesse trabalho Tivemos que implementar uma solução em programação dinâmica para um problema muito interessante. Não é interessante por ser díficil ou aplicavél e sim porque foi interessante pensar e caminhar por várias soluções até encontrar uma satisfatória. O problema proposto, era dado uma matriz NxM conseguir maxímizar o número de pessoas dominadas pelo WM e implementar tal solução de utilizando também a programação paralela. Um algoritmo solução foi implementado, e o problema foi resolvido, acredito, com êxito. Desde o principio o foco foi o de minimizar o número de comparações para encontrar a solução. A implementação em paralelo de certa forma, moldou de certa forma, a estrtura utilizada. A complexidade teórica assíntótica de espaço e tempo do algoritmo implemnetado é $O(\mathsf{N} \times \mathsf{M})$.

Esse trabalho foi o mais desafiador, na minha opinião, não por ser o mais dificíl mas por intuitivamente parecer existir tantas opções de solução e ao mesmo tempo tão poucas. Não fosse o tempo extra longo devido ás ocupações acredito que dificilmente conseguiria resolver o problema no fim de semestre com outras disciplinas.

6 Referências Bibliográgicas

Robert Sedgewick and Kevin Wayne. 2011. Algorithms (4th ed.). Addison-Wesley Professional.

Thomas T. Cormen, Charles E. Leiserson, and Ronald L. Rivest. 1990. Introduction to Algorithms. MIT Press, Cambridge, MA, USA.