Documentação Trabalho Prático 3 de AED's 3

Aluno: Marcus Vinícius de Oliveira **Matricula:2014144804**

1. Introdução

Dada uma matriz MxN contendo a população de cada cidade, devemos descobrir qual a população máxima a recuperar usando estratégia de programação dinâmica e seguindo as seguintes restriçoẽs de escolha de cidades:

- Cada vez que uma cidade é escolhida, as linhas abaixo e acima são anuladas juntamente com as cidades adjacentes.
- Deve-se obter a soma máxima da população, dada restrição acima

2. 1. Modelagem do Problema e sua solução

Dada as restrições, chegamos a seguinte conclusão:

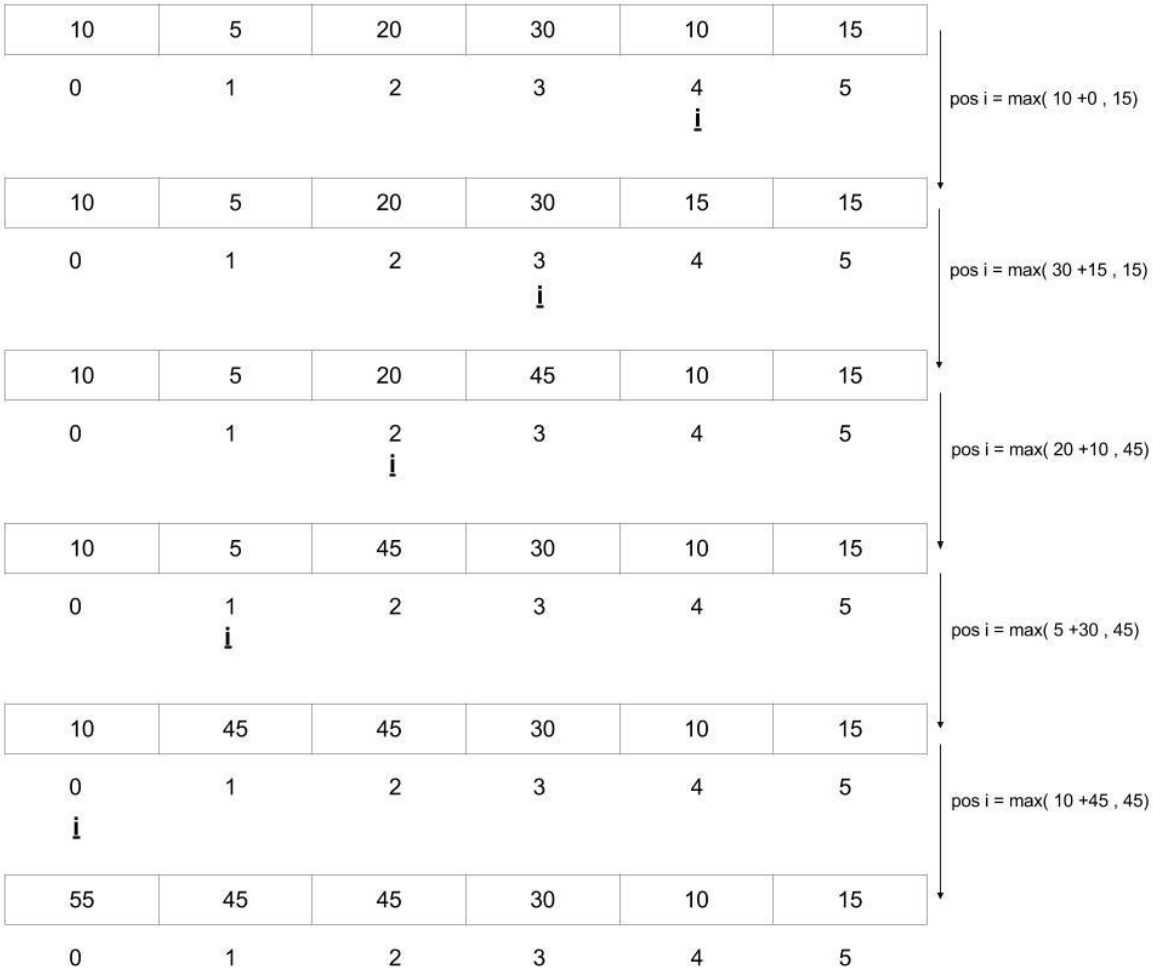
- Não se pode escolher duas cidades adjacentes.

Além disso, observamos que é muito custoso resolver o problema instanciando a matriz. Por isso adotamos a ideia de resolvermos as linhas separadamente e posteriormente juntar os resultados.

Para a solução de uma linha usamos a seguinte estratégia:

```
for (i = Numero_de_elementos_no_vetor -2; i>=0; if--){
  vetor[i] = escolhe_maior_entre (vetor[i+2] + vetor[i], vetor[i+1]);
}
```

Ou seja, o funcionamento se resume ao dilema: é melhor somar vetor[i] com vetor[i+2] ou propagar vetor [r+1]? Veja o exemplo:



O algoritmo é simples e funcional. Ao final, o elemento [0] guarda a maior soma possível do vetor e está deve ser guardada em novo vetor 'vetor_final' com tamanho igual ao número de linhas da matriz.

Ao final, quando o algoritmo tiver rodado em todas as linhas da matriz, o vetor_final estará completo com todas as maiores somas possíveis das linhas da matriz. Dessa forma, basta rodar o algoritmo novamente, desta vez para o vetor_final e então obteremos a maior soma possível da matriz.

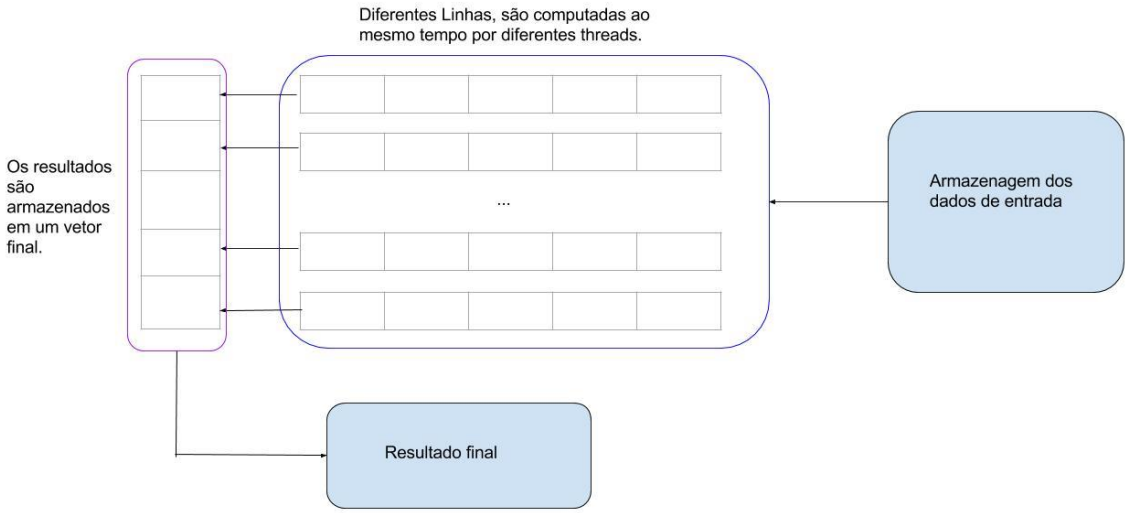
2. 2. Programação Dinâmica

Para ganhar desempenho, podemos paralelizar os processos descritos no tópico anterior. Pela natureza da solução do problema, o processo de paralelização é bem simples. Basta criar threads com a biblioteca phreads.h, criar uma rotina com o algoritmo anterior e distribuir as linhas da matriz para as threads.

Desta forma, a soma máxima de uma linha, poderá ser calculada ao mesmo tempo que outra. Isso resulta em um ganho em desempenho do programa.

Vale lembrar que o número de threads é passado como parâmetro na chamada do programa e o desempenho do mesmo é dependente do número de núcleos que o processador que o está executando possui.

Abaixo um esquema simplificado do funcionamento do programa.



3. Análise de complexidade

A análise de complexidade do programa é dependente apenas da análise do algoritmo que calcula a soma máxima de uma linha. Dado o algoritmo:

```
for (i = Numero_de_elementos_no_vetor -2; i>=0; if--){
  vetor[i] = escolhe_maior_entre (vetor[i+2] + vetor[i], vetor[i+1]);
}
```

Temos o dilema, é melhor somar vetor[i] com vetor[i+2] ou propagar vetor [i+1]? Essa pergunta é respondida em O(1), pois vetor[i+2] e vetor [i+1] já foram calculados anteriormente e ambos são subproblemas ótimos.

Uma vez que o algoritmo é repetido mais m vezes + 1 (onde m é o máximo de linhas e 1 é o vetor_final), temos que a complexidade é de O(n *m) para um programa serial. Para um programa paralelo, temos: O(n*m)/qt_threads (onde qt_threads é a quantidade de threads que serão utilizadas).

4. 1. Análise Experimental

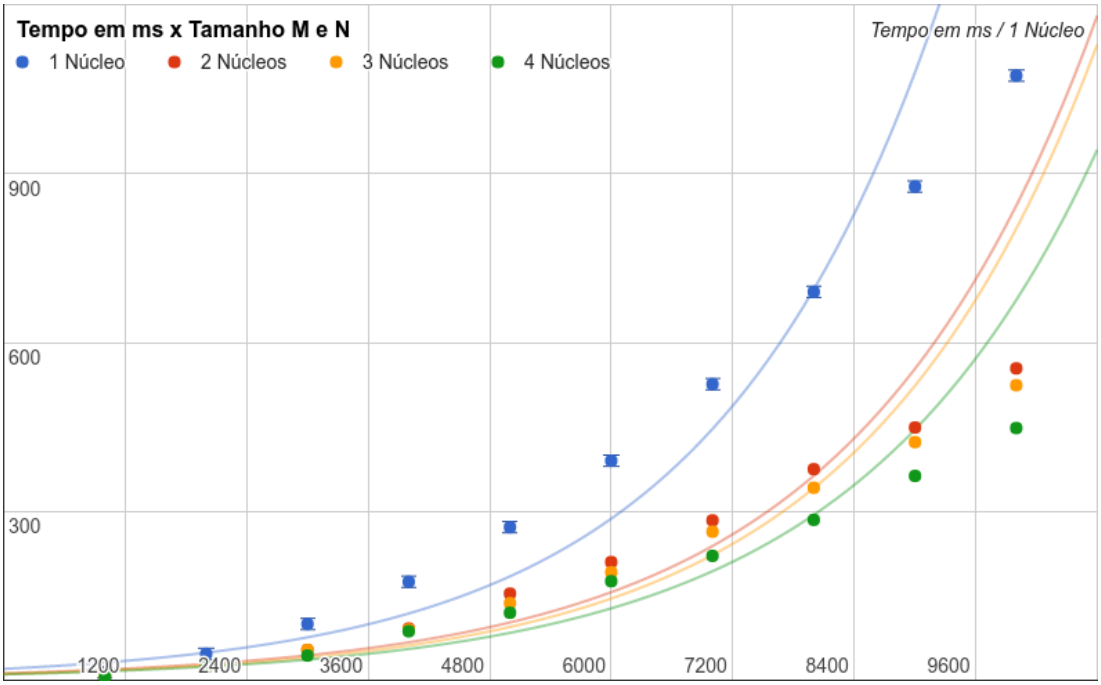
A medição do tempo da análise foi feita utilizando um técnica que utiliza a biblioteca sys/time.h. Tal técnica foi descoberta através de pesquisas em forum stackoverflow e seu uso está sinalizado no código.

Em nossa análise experimental, fixamos n = m para facilitar a criação de casos teste, bem como os estudos dos resultados. Criamos 10 testes com n e m variando de 1000 a 10000. Também variamos o número de threads de 1 a 4. Os resultados podem ser conferidos na tabela abaixo:

Tamanho de M e N	Físicos		Emulados	
	1 Núcleo	2 Núcleos	3 Núcleos	4 Núcleos
1000	14	11	8	7
2000	47	23	24	25
3000	100	54	54	44
4000	175	92	90	87
5000	272	154	137	120
6000	390	210	192	176
7000	526	284	264	221
8000	690	375	342	285
9000	877	449	423	363
1000	1074	554	524	448

4. 2 Analise dos resultados

Para uma melhor visualização foi criado o seguinte gráfico:



Como podemos notar, o resultado experimental é totalmente compatível com o teórico. O fato de considerarmos n=m torna a complexidade O(n*m)/qt_thread em O(n^2)/qt_thread.

O fato das séries não estarem em perfeita concordância com suas linhas de tendência não contradiz nossa analise teórica. Muito provavelmente esse fenômeno ocorra, devido aos ruídos do sistema utilizado (gerenciamento de processamento, memoria e etc pelo SO). Mesmo com tal interferência, é possível notar sua tendência exponencial.

Outro fato a se notar é que assim como na análise teórica, o número de threads é inversamente proporcional ao tempo de execução. Importante notar que tal fato não se aplica da mesma maneira quando adicionamos threads com núcleos virtuais (núcleo 3 e 4). Neste caso, embora o desempenho aumente, a equação O(n*m)/qt_thread não é verdadeira. A equação só é verdadeira caso as treads estejam ligadas à núcleos físicos.

5. Conclusão

Neste trabalho resolvemos o problema de dominação mundial aplicando nosso conhecimento em programação dinâmica. Embora tenha ocorrido uma dificuldade inicial na modelação do problema, uma rápida pesquisa na wiki da Maratona de programação UFMG, resolveu tal dificuldade.

Trabalhar com programação dinâmica foi divertido e não houve grandes problemas ou dificuldades na utilização da biblioteca pthread.h. Além disso, é notável a melhora de desempenho ao utilizar tal técnica.

6. Referências

1. https://virtual.ufmg.br/20162/pluginfile.php/251291/mod_resource/content/3/pthreads.pdf
2. http://wiki.maratona.dcc.ufmg.br/index.php/SAMER08C
3. http://ncc.furg.br/seminarios/pthreads.pdf
4. http://homepages.dcc.ufmg.br/~coutinho/pthreads/ProgramandoComThreads.pdf
5. http://stackoverflow.com/questions/12722904/how-to-use-struct-timeval-to-get-the-execution-time