Relatório
T1 – Programação Paralela
Por:
Guilherme Costa Pateiro - GRR20197152
João Gabriel Borges de Macedo - GRR20190429

1. Recorte o kernel (parte principal) do algoritmo e explique em suas palavras o funcionamento sequencial do trecho.

```
int passeio_cavalo_par2(int **tabuleiro, int x, int y, int jogada){
    #pragma omp parallel
    #pragma omp single
    {
    #pragma omp taskloop firstprivate(tabuleiro, jogada) grainsize(1)
    for (int i=0;i<8;i++){
        task++;
            int **tabuleiro2 = copiamatriz(tabuleiro);
            int x2 = x + xMove[i];
            int y2 = y + yMove[i];
            if (jogada_valida(x2,y2, tabuleiro2)){
                tabuleiro2[x2][y2] = jogada+1;
                log("jogada = %i || tread = %i || i = %i || endereco do tabuleiro = %p \n", jogada , omp_get_thread_num(), i ,tabuleiro2);
                passeio_cavalo_seq(tabuleiro2, x2,y2, jogada+1);
                tabuleiro2[x2][y2] = 0;
            }
                    // libera a memória da matriz
            for (int i=0; i < N; i++)
                free (tabuleiro2[i]);
            free (tabuleiro2);
            task--;
        }
    }
    return 0;
}
```

O algoritmo tem como entrada uma tabela de tamanho M por N, as coordenadas atuais dentro da tabela e um número representando a jogada atual.
No início de cada iteração, primeiro se verifica se a jogada atual é a jogada máxima, o que indicaria que um resultado foi encontrado e o algoritmo pode ser parado.
Caso não seja o movimento final, será escolhido um novo movimento dentre os 8 possíveis, as coordenadas serão atualizadas e será verificado se a nova posição é válida (nunca foi ocupada e está dentro dos limites da tabela). Por fim será chamado o algoritmo recursivamente, agora com uma tabela atualizada e com o incremento da jogada atual.

O algoritmo utilizado para o trabalho se assemelha a uma busca em profundidade em um grafo com fator de ramificação 8, por causa de sua recursão, pode se dizer que cada jogada aumenta a profundidade da árvore.

2. Explique qual a estratégia final (vitoriosa) de paralelização você utilizou.

A estratégia escolhida se baseia em paralelizar a busca atribuindo a uma task um movimento ou conjunto de movimentos do nível inicial da árvore. Esse nível foi escolhido por ser o que possui o retorno mais lento, pois há, no máximo, um conjunto de 8^((M*N) - 2) movimentos dentro desse primeiro movimento.
Posteriormente, uma segunda camada de paralelização é gerada na profundidade (M*N)/4; essa profundidade foi escolhida pois, para problemas feitos nos testes, valores mais avançados geram muitos retornos, o que aumenta o overhead e diminui a eficiência.
A ideia é aumentar ao máximo a ramificação da árvore maximizando a probabilidade de uma task encontrar uma resposta viável.
Por causa dos limites físicos dos computadores utilizados, mais camadas de paralelização não foram possíveis, porém a estratégia de subdividir uma jogada e atribuir um conjunto de movimentos a uma task poderia ser feito até o tamanho da árvore do problema. Caso todas os níveis da árvore fossem paralelizados, e realmente fossem rodados em paralelo, haveriam até 8^(M*N) tasks e o problema seria resolvido em O(M*N)

A cada task, inicialmente, é atribuído uma carga de 8^((M*N)-2) instruções, com o segundo nível de paralelização, cada task recebe até 8^((M*N)-2)/4 passos para serem executados.

3. Descrever a metodologia que você adotou para os experimentos a seguir. Não esqueça de descrever também a versão do SO, kernel, compilador, flags de compilação, modelo de processador, número de execuções, etc.

A metodologia escolhida foi de fazer 2 casos de teste para tabelas de tamanho N*N com N = 5,6,7 . Problemas de tamanhos 8x8 ou maiores não foram possíveis de serem feitos por causa de seus tamanhos, por ser teoricamente 8^15 vezes maior que o 7x7.

O primeiro caso de teste coloca o ponto inicial da tabela na coordenada (0,0). Esse caso de teste causa uma ramificação de apenas 2 movimentos possíveis na posição inicial, sendo que as threads posteriores serão utilizadas no segundo nível de ramificação. Ou seja, iniciar na posição (0,0) faz com que haja mais threads no segundo nível de ramificação.

O segundo caso de teste coloca a posição inicial nas coordenadas (N/2,N/2), ou o meio da tabela, nessa posição é garantido que haverá a ramificação dos 8 possíveis movimentos na jogada = 1 , diminuindo a ramificação no segundo nível.

Para cada tamanho e caso de teste foram executadas 20 vezes o algoritmo(tempos disponíveis na pasta data), depois foi tirado a média dos tempos e seus desvios padrões utilizando um programa auxiliar (presente em media.cpp e no executável media) os resultados estão presentes na pasta estatística.

Da forma como foi implementado, por só possuir 2 níveis de paralelização, existe um limite de máximo de 64 tasks que serão lançadas ao mesmo tempo , caso fosse colocado um terceiro nível de paralelização esse limite se tornaria 512 tasks

O hardware para rodar o algoritmo paralelizado (foi utilizada a cpu2 do Dinf):
Processador: AMD EPYC 7401 (24 núcleos físicos, 32 threads)
RAM : 197Gb
Nenhuma configuração de limitação de clock foi feita, por exemplo desligar o turboboost.

A Compilação é feita através do makefile, internamente o makefile utiliza as flags "-fopenmp -Wall -O3"
-fopenmp = indica que a biblioteca openmp deve ser ligada com os fontes
-Wall = indica que todos os warnings devem ser mostrados
-O3 = indica que o compilador deve otimizar o máximo possível o código executável

4. Com base na execução sequencial, meça e apresente a porcentagem de tempo que o algoritmo demora em trechos que você não paralelizou (região puramente sequencial).

Já que a paralelização ocorre na primeira iteração do algoritmo, a única parte não paralelizada é a leitura dos args, e entradas do programa, assim como a alocação inicial da tabela e algumas configurações iniciais. Porém o número de iterações dentro da parte paralela é muito variável.

Fazendo a medição de tempo da parte sequencial, a parte sequencial é de aproximadamente 2*10^-5 segundos e apresenta crescimento de tempo praticamente constante.

Pegando as métricas de tempo recebidas, a porcentagem de tempo gasto em áreas puramente sequenciais pode variar de de 2,5%(5x5 com início 0x0) até 0.000000001% (7x7 com início no N/2xN/2)

5. Aplicando a Lei de Amdahl, crie uma tabela com o speedup máximo teórico para 2, 4, 8 e infinitos processadores. Não esqueça de explicar a metodologia para obter o tempo paralelizável e puramente sequencial.

Utilizando como base o tempo de 2*10^-5 segundos para a execução da parte sequencial, e os tempos encontrados na execução paralela chegamos em porcentagens diferentes baseado no tamanho da entrada:
5x5: 97,5% de área paralela
6x6: 99,99% de área paralela
7x7: 99,9999999999% de área paralela
Essa foram as métricas utilizadas para formar a seguinte tabela

Processadores	2	4	8	Infinitos
5x5	1.9512195	3.7209302	6.8085106	40
6x6	1.9980019	3.9880358	7.9443892	10^4
7x7	1.9999999	3.9999999	7.9999999	10^{12}

A fração paralelizável do algoritmo cresce extremamente rapidamente com o tamanho do tabuleiro (visto que a complexidade do mesmo é exponencial).

6. Apresente tabelas de speedup e eficiência. Para isso varie o número de threads entre 1, 2, 4 e 8. Varie também o tamanho das entradas, tentando manter uma proporção.

média de tempos para tabelas com inicio 0 x 0						
thread(s)	1	2	4	8	16	32
5x5	0,00151545	0,00087715	0,00098695	0,00126185	0,00163885	0,0026565
6x6	0,108879	0,110205	0,111632	0,110582	0,113046	0,114705
7x7	2,37236	0,0177265	0,018683	0,0183655	0,0190972	0,0220448

média de tempos para tabelas com inicio n/2 x n/2						
thread(s)	1	2	4	8	16	32
5x5	0,00064985	0,00056895	0,00075165	0,0011682	0,001397	0,0020949
6x6	0,732215	0,737332	0,0137158	0,00081505	0,00114085	0,0021856
7x7	256,613	6,62609	6,66124	1,16165	1,16631	1,1799

Speedup relativo para tabelas com inicio em 0 x 0						
thread(s)	1	2	4	8	16	32
5x5	1	1,72770	1,53549	1,20097	0,92470	0,57047
6x6	1	0,98797	0,97534	0,98460	0,96314	0,94921
7x7	1	133,83127	126,97961	129,17481	124,22554	107,61540

Eficiência para tabelas com inicio em 0 x 0						
thread(s)	1	2	4	8	16	32
5x5	1	0,86385	0,38387	0,15012	0,05779	0,01783
6x6	1	0,49398	0,24383	0,12307	0,06020	0,02966
7x7	1	66,91563	31,74490	16,14685	7,76410	3,36298

Speedup relativo para tabelas com inicio em n/2 x n/2						
thread(s)	1	2	4	8	16	32
5x5	1	1,14219	0,86456	0,55628	0,46518	0,31021
6x6	1	0,99306	53,38478	898,36820	641,81531	335,01784
7x7	1	38,72767	38,52331	220,90389	220,02126	217,48708

Eficiência para tabelas com inicio em n/2 x n/2						
thread(s)	1	2	4	8	16	32
5x5	1	0,57110	0,21614	0,06954	0,02907	0,00969
6x6	1	0,49653	13,34620	112,29602	40,11346	10,46931
7x7	1	19,36383	9,63083	27,61299	13,75133	6,79647

7. Analise os resultados e discuta cada uma das duas tabelas. Você pode comparar os resultados com speedup linear ou a estimativa da Lei de Amdahl para enriquecer a discussão.

A eficiência do algoritmo depende fortemente da posição inicial, visto que há uma grande variância de tempo de execução dependendo do tamanho do tabuleiro (a mesma posição inicial pode ou não ser ótima em tamanhos diferentes). Falando exclusivamente dos testes com posição inicial (0,0), é evidente que, para posições iniciais boas, a paralelização tem pouco efeito, tal como nos testes 5x5, em que houve um speedup, mesmo que menor, ou os testes 6x6, em que o overhead superou o ganho da paralelização. No entanto, para posições iniciais ruins, como nos testes 7x7, o speedup é muito maior do que o projetado pela Lei de Amdahl. Assim, fica claro que a paralelização é extremamente efetiva no pior caso do problema (e portanto, acelera a resolução dele no caso médio).

8. Seu algoritmo apresentou escalabilidade forte, fraca ou não foi escalável? Apresente argumentos coerentes e sólidos para suportar sua afirmação.

O algoritmo não apresentou escalabilidade notável, os valores de speedup do algoritmo variam de maneiras drásticas e com pouca previsibilidade, observando a tabela, há 2 casos que ocorrem normalmente.

O primeiro é de haver pouca variação na execução conforme o número de threads aumenta, ainda tendendo a piorar já que o aumento do número de execuções paralelas aumenta o overhead. Nesses casos, observando as tabelas de eficiência, como o speedup não tem grandes variações a eficiência diminui uma vez que o número de núcleos está dobrando.
Esse caso pode ser observado em ambas execuções do 5x5 e na execução 0x0 do tabuleiro 6x6.

O segundo caso que ocorre é o de haver saltos extremos no speedup, mesmo com pouca alteração no número de threads que estão sendo usadas, tal característica condiz com o caráter superlinear da estratégia de paralelização.

Esse caso pode ser observado em ambas execuções do 7x7 e na execução com coordenadas iniciais (N/2,N/2) do tabuleiro 6x6.

9 . Pense sobre cada um dos resultados. Eles estão coerentes? Estão como esperados? A análise dos resultados exige atenção aos detalhes e conhecimento.

Os resultados encontrados, condizem com a ideia do algoritmo ser superlinear. O principal influenciador para a superlinearidade é o fato de existirem muitas respostas corretas e saídas válidas para uma única entrada, assim, a ramificação gerada pela paralelização convém escolher um caminho que chega em uma resposta muito mais rapidamente que o programa sequencial. Isso já era esperado, vendo que problemas com múltiplas saídas podem ser superlineares.

A estratégia escolhida tem como objetivo ramificar a árvore de busca e se mostrou muito efetiva em vários casos, principalmente com o crescimento da entrada. No entanto, há o risco da ramificação não encontrar uma resposta mais rapidamente que o programa sequencial. Nesses casos, a paralelização pode gerar perda de desempenho e um uso desnecessário de recursos computacionais além de piorar até o caso com 1 thread, pois haverá um overhead mínimo.

Além disso, pequenas variações na entrada geram grandes mudanças no tempo. Por exemplo, mudar a coordenada inicial do problema pode modificar drasticamente o tempo de execução.

Um ponto importante que deve ser comentado é que a escolha da ramificação secundária ocorrer na jogada = M*N/4 foi feita pois, para os problemas apresentados, essa profundidade permitiria que as tasks ficassem muito tempo executando (e assim diminuindo o overhead) e ao mesmo tempo uma profundidade que fosse possível que as tasks terminassem sua execução e pudessem ser utilizadas novamente, maximizando a ramificação da busca. Para tamanhos maiores, a abordagem correta seria trazer ramificação secundária para mais próximo da jogada = M*N