Relatório
T2 – Programação Paralela
Por:
Guilherme Costa Pateiro – GRR20197152

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

```c
void max_min_avg(double *vetor, int tamanho, double *max, double *min, double *media){
    double soma = 0;
    *max = *min = vetor[0];
    for(int i=0; i< tamanho; i++){
        soma += vetor[i];
        if (vetor[i] > *max)
            *max = vetor[i];
        if (vetor[i] < *min)
            *min = vetor[i];
    }
    *media = soma / tamanho;
}
```

O kernel do algoritmo é o procedimento max_min_avg() que descobre o maior, menor valores de um vetor e a media dos elementos desse vetor, sua complexidade é de O(n) sendo ‘n’ o valor da variável ‘entrada’ passada para a função. O procedimento se baseia em, para cada elemento do vetor de entrada comparar com o menor e o maior para possivelmente substitui-los, e posteriormente esse elemento do vetor é somado com um uma variável ‘soma’ que guarda a soma de todos os elementos do vetor, no final ‘soma’ é dividida pelo numero de elementos do vetor para se obter a media.

Esse trecho, mesmo sendo a parte mais importante do programa, não foi alterado , sendo utilizado uma estrategia de executar esse procedimento com varias partes do vetor ao mesmo tempo, tal escolha se baseou no fato que ‘tamanho só afeta o calculo dentro de uma janela enquanto paralelizar a parte externa a esse procedimento permite calcular varias janelas ao mesmo tempo

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

```c
if (rank == 0) {
    for(int i=1; i < num_proc; ++i) {
            MPI_Send(&serie[(i*(tam_serie/num_proc))], (tam_serie/num_proc) + tam_janela , MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
    }
    double max, min, media;
    for(int i = 0; (i <= tam_serie/num_proc) && (((rank*(tam_serie/num_proc)) + i + tam_janela) < tam_serie) ; i++){
        max_min_avg(&serie[i],tam_janela, &max, &min, &media);
        //printf("janela %d - max: %lf, min: %lf, media: %lf\n", i, max, min, media);
    }
}
else {
    double max, min, media;
    MPI_Recv(vet_parcial,tam_serie/num_proc + tam_serie , MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for(int i = 0; (i <= tam_serie/num_proc) && (((rank*(tam_serie/num_proc)) + i + tam_janela) < tam_serie) ; i++){
        max_min_avg(&vet_parcial[i],tam_janela, &max, &min, &media);
        //printf("janela %d - max: %lf, min: %lf, media: %lf\n", i, max, min, media);
    }
    free(vet_parcial);
}
```

A estrategia utilizada se baseia em:
- o calculo do vetor inteiro é feito sequencialmente e no momento do carregamento de dados, como a leitura dos dados é feita lendo cada numero da entrada de cada vez, no momento da entrada eles são utilizados para tirar o mínimo. máximo e media do vetor, quando o vetor terminal de ser carregado, essas métricas já existem.

- o vetor é então quebrado em vetores menores e distribuídos entre as threads, o tamanho do vetor distribuído para cada thread é o tamanho de seu chunk + o tamanho da janela que será gerada pelo ultimo elemento do chunk, nota-se que, no envio de chunks para as threads, há a sobreposição de elementos do vetor entre 2 ou mais threads, por causa do tamanho extra da janela, essa estrategia foi adotada para garantir que não haveria necessidade de enviar outros pedaços de chunks para outras threads, diminuindo o overhead ao custo de aumentar a transmissão inicial.
Essa estrategia também server para evitar o possível erro de haver uma janela que é maior que um chunk, por exemplo se o tamanho do chunk fosse 100 e o tamanho da janela 150 haveriam casos onde o inicio do calculo ocorrei em uma thread, uma thread intermediaria teria que gerar valores para seu pedaço inteiro e o final seria calculado por uma terceira thread.
Uma comparação teve que ser criada para verificar se a janela que esta sendo calculada é a ultima presente no vetor.

-o método de transmissão escolhido é o mpi_send // mip_recv pois ele permite um controle mais preciso do tamanho dos chunks e ao mesmo tempo é o método que considerei mais fácil para fazer a sobreposição de chunks que foi falada.

3. Descreve 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.

O método utilizado para tirar as métricas se baseia em escolher um tamanho de vetor e janelas que são potencias de 2 , isso facilita a visualização e permite criar uma relação entre tamanho da entrada e o de janelas. Além disso como numero de threads escolhido também é uma potencia de 2, essa escolha de tamanho de entrada garante que sempre o número de threads será divisível com o numero de

É importante citar que o programa só funciona perfeitamente se o tamanho da entrada for divisível pelo numero de threads que serão atibuidas para o programa, o tamanho de janelas pode ter qualquer tamanho entre 1 e o tamanho da entrada sem perdas. Divisões com resto geram perda de janelas.

Os tamanhos escolhidos para janelas foram de 128, 2048, 16384, 32768. Os tamanhos de entrada eram de 16384, 32768 , 65536, 131072, 262144, 524288, 1048576. O tamanho de 16384 não foram usados nos teste com janela de tamanho 32768.

O tempo foi tirado colocando um timer na linha anterior do mpi_init() e terminando pouco antes de chamar o mpi_finalize(), antes de tirar a métrica foi colocado uma barrira para garantir que todas as threads foram finalizadas antes de parar o tempo.

Para cada tamanho e para cada tamanho de janela foram feitos 20 testes com 1, 2 ,4, 8, 16, 32 threads, totalizando 3240 excussões entre todos os testes

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 nenhuma flag de otimização é passada para o compilador mpicc

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).

Como o carregamento no arquivo é feito sequencialmente então há O(N) tempo feito sequencialmente , sendo N o tamanho da entrada, a parte paralela é executada K(N-K) vezes, sendo K o tamanho da janela, no caso mínimo K = 1 ou K = N a parte paralela é feita em N passos ou seja só 50% do programa é paralelizado, no caso máximo de K = N/2 a parte paralela faz $N^2/2-N$ passos, tornando a parte paralela em aproximadamente $N^2/2$ vezes maior que a parte sequencial, quando mais perto K estiver de N/2 mais próximo do máximo o programa estará, a parte sequencial, no caso de 1048576 é de aproximadamente 0,16segundos

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.

Para a geração dessas métricas foi feito analisando o caso com maior e menor porcentagem paralelizável.
Para o caso K = 1 , foi aplicado a porcentagem de 50% sequencial e 50% com paralelismo.
Para o caso K = N/2 , foi aplicado a taxa de 99,90% de paralelismo

o calculo feito foi o de:

1/(<porcentagem_sequencial> + 1/<numero de treads>*<porcentagem_paralela>)

	2	4	8	∞
K=1	1,33	1,6	1,77	2
k=N/2	1,98	3,98	7,94	1000

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. Veja um exemplo de tabela:

As tabelas a seguir mostram as medições de tempo obtidas pelo programa desenvolvido:
A coluna a esquerda indica o tamanho da entrada e a linha superior o numero de treads.

eficiência janela de 16384						
-	1	2	4	8	16	32
16384	1,00000	0,49914	0,24140	0,11258	0,04872	0,01975
32768	1,00000	0,49735	0,40535	0,29104	0,17407	0,08175
65536	1,00000	0,71887	0,63251	0,50663	0,34620	0,18166
131072	1,00000	0,84996	0,79358	0,69484	0,53755	0,33181
262144	1,00000	0,92373	0,89365	0,82007	0,69951	0,44726
524288	1,00000	0,95335	0,93075	0,87562	0,79005	0,47203
1048576	1,00000	0,98296	0,95895	0,91380	0,84363	0,71587

eficiência janela de 128						
-	1	2	4	8	16	32
16384	1,00000	0,50983	0,24528	0,11502	0,05043	0,02089
32768	1,00000	0,51887	0,25306	0,11996	0,05272	0,02188
65536	1,00000	0,53086	0,26211	0,12551	0,05542	0,00000
131072	1,00000	0,55826	0,28850	0,14154	0,06290	0,02694
262144	1,00000	0,59828	0,32087	0,16293	0,07440	0,03247
524288	1,00000	0,65024	0,37699	0,19912	0,09509	0,04143
1048576	1,00000	0,69734	0,42773	0,23592	0,11600	0,05439

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.

As tabelas a seguir mostram as medições de tempo obtidas pelo programa desenvolvido: A coluna a esquerda indica o tamanho da entrada e a linha superior o numero de treads. Todos os tempos estão em segundos.

janela de 128						
-	1	2	4	8	16	32
16384	0.27436	0.269069	0.279636	0.298176	0.340033	0.410346
32768	0.289891	0.279351	0.286381	0.30208	0.343679	0.414101
65536	0.315398	0.297063	0.300823	0.314128	0.355677	0.418748
131072	0.38047	0.340763	0.329697	0.336007	0.378051	0.441286
262144	0.494928	0.413628	0.385609	0.379703	0.415768	0.476395
524288	0.749481	0.576309	0.49702	0.470498	0.492632	0.565358
1048576	1.22275	0.876719	0.714669	0.647851	0.65882	0.702573

Nota-se que nessa tabela, o tempo necessário para se executar o programa é consideravelmente mais curto, como será demostrado nas próximas tabelas, isso se deve ao fato que 128 está muito próximo ao caso mínimo de K = 1, por esse motivo seu speedup máximo teórico é baixo.

É importante apontar o fato que o algoritmo paralelo é pior se o sequencial para os tamanhos menores que 262 mil e que o algoritmo demostrou melhor desempenho com 4 ou 2 treads nesses tamanhos menores

O maior sppedup alcançado foi 1,74 na coluna do 1milhao comparando 1 com 32 treads

janela de 2048						
-	1	2	4	8	16	32
16384	0.395236	0.343023	0.314783	0.315458	0.353554	0.415118
32768	0.543488	0.415315	0.351536	0.331911	0.362725	0.421672
65536	0.816885	0.562531	0.431524	0.381019	0.393301	0.45571
131072	1.41679	0.860197	0.59709	0.472181	0.450049	0.505935
262144	2.56371	1.45955	0.90141	0.648916	0.544995	0.592515
524288	4.93893	2.68115	1.54471	0.999654	0.764561	0.797307
1048576	9.60212	5.12382	2.85051	1.75612	1.20171	1.13033

Com uma janela 16 vezes maior, o sppedup cresce de maneira bem mais notável chegando a 8,49 na comparação de 1 e 32 treads e 1 milhão de elementos

O Algoritmo também apresentou melhoras no desempenho de tamanhos menores, o tamanho de 16 mil já apresentou melhorias do algoritmo paralelo com 2,4,8 e 16 treads. Com 32 treads o algoritmo se tornou melhor com 32 mil elementos.

janela de 16384						
-	1	2	4	8	16	32
16384	0.266795	0.267255	0.276303	0.296216	0.342221	0.422096
32768	1.36986	1.37716	0.844872	0.588337	0.491838	0.523651
65536	3.56246	2.47782	1.40807	0.878957	0.643145	0.61284
131072	8.11912	4.77618	2.55775	1.46061	0.94400	0.764663
262144	17.1075	9.26004	4.78583	2.60762	1.52852	1.19531
524288	34.85	18.2776	9.36071	4.97504	2.75695	2.30718
1048576	69.8550	35.5329	18.2113	9.55558	5.1752	3.04939

O speedup máximo encontrado foi de 22,97 , comparando 1 e 32 treads com 1 milhão de elementos

Um fato Importante é que no caso de 16 mil elementos, o tamanho do vetor é igual ao tamanho da janela, caindo assim , no menor caso do algoritmo.

Analisando a linha do 16 mil, percebe-se que todos os casos paralelizados apresentaram métricas piores que a versão com somente 1 tread.

Nessa mesma tabela, a linha de 32 mil elementos cai no maior caso do algoritmo, as métricas paralelas apresentaram resultados melhores, com exceção do caso com 2 treads, os resultados baixos são justificados com o tamanho pequeno do vetor.

janela de 32768						
-	1	2	4	8	16	32
32768	0.269193	0.272571	0.283355	0.301269	0.346235	0.412295
65536	4.75695	4.83818	2.55384	1.4666	0.932476	0.729181
131072	13.3861	9.04025	4.71711	2.55597	1.50191	1.24244
262144	30.9109	17.7379	9.16954	4.83921	2.67117	1.74223
524288	66.8387	35.8024	18.3708	9.46837	5.04339	3.60183
1048576	137.8490	71.4770	36.5129	18.8014	9.86571	5.49608

O speedup máximo encontrado foi de 25,10 , comparando 1 e 32 treads com 1 milhão de elementos

Novamente, nessa tabela temos estatísticas de maior e menor caso:

É importante notar que o tempo do algoritmo sequencial é 18,26 vezes mais rapido com 32 mil elementos que com 64mil , se tornando a maior perda de desempenho em comparação com a mudança de tamanho da entrada entre todas as tabelas

Com 16 mil elementos, todos os algoritmos paralelos demostraram pera de desempenho. Com 32 mil elemento, a paralelização com 2 treads demostrou perda de desempenho, porém houve um speedup 6,6 com 32 treads , ainda muito inferior que o limite teórico de 31,03 que esse caso geraria aplicando a lei de amdahl

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

eficiência janela de 16384						
-	1	2	4	8	16	32
16384	1,00000	0,49914	0,24140	0,11258	0,04872	0,01975
32768	1,00000	0,49735	0,40535	0,29104	0,17407	0,08175
65536	1,00000	0,71887	0,63251	0,50663	0,34620	0,18166
131072	1,00000	0,84996	0,79358	0,69484	0,53755	0,33181
262144	1,00000	0,92373	0,89365	0,82007	0,69951	0,44726
524288	1,00000	0,95335	0,93075	0,87562	0,79005	0,47203
1048576	1,00000	0,98296	0,95895	0,91380	0,84363	0,71587

Como visto nessa tabela de eficiência, quando maior o numero de treads, maior deve ser a entrada para que a eficiência se mantenha, indicando que o algoritmo tem escalabilidade fraca.

Porém, como o aumento do vetor ou da janela não seguem um padrão de crescimentos de custo linear e sim um custo parabólico, essas métricas podem tender a uma eventual perda do desempenho uma vez que o aumento da janela diminui o custo computacional, e como foi demostrado nas tabelas acima, ha uma tendencia de piora nos casos mínimos do algoritmo.

Então, com as métricas que possuo com tamanhos de janelas fixos e modificando somente o tamanho do vetor , ou seja proporcionalmente tornando o custo do algoritmo consideravelmente mais próximo do caso mínimo , a tendencia ter uma escalabilidade

fraca. Porém admito que seria necessário rodar um teste com entrada de tamanho fixo e modificar o tamanho da janela para obter uma resposta.

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

O tem uma entrada de N leituras e depois ele faz N-i janelas de tamanho i. a Formula que gera a complexidade do algoritmo é de N+(N-I)I , a complexidade desse algoritmo é uma parábola, o que faz com que ele seja $O(N^2)$ e $\Omega(N)$, o valor máximo é alcançado quando I = N/2; Nos testes feitos, alguns casos de mínimo e máximo foram gerados , porem por se tratar de um algoritmo “rápido” com somente algumas milhões ou bilhões de instruções, um tamanho de vetor e janela maiores deveriam ser testados para gerar métricas mais claras e estatística mais precisas, uma vez que só foram feito casos com janelas fixos.

Outra métrica que não foi muito explorada é o overhead gerado pela transição dos vetores, porem com o crescimento da entrada, ha um aumento drástico no consumo de memoria o que pode gerar saturação de gerar perda de desempenho. Por exemplo, 1 bilhão de doubles ocupam aproximadamente 1Gb de memoria, como o algoritmo foi construído, haveria mais de 2 vezes os dados da entrada em memoria, o que não caberia na cache do processador, diferente dos 1,05 milhão de doubles que foram colocados nesse teste no máximo, que ocupam 4Mb e provavelmente ocupavam pouco mais de 8Mb de memoria, o suficiente para ser colocado em memoria cache do processador, tirando proveito da velocidade dessas memorias.

Por fim, esse algoritmo apresenta resultados, melhores conforme o tamanho da entrada aumenta , provavelmente métricas com entradas maiores iriam tender ao speedup máximo teórico, lembrando que há um custo de memoria, que não foi explorado, podendo atrapalhar nas métricas do algoritmo.