Extract Features Paralelizado com MPI

Universidade Federal do Paraná - **UFPR**
Professor Daniel Alfonso Gonçalves de Oliveira

Aluno Erick Eckermann Cardoso - GRR20186075

1. **Kernel do algoritmo sequencial**

O kernel do algoritmo se encontra no meio para o final do código. Sua função é alocar e ler o vetor de números de um arquivo que foi passado como parâmetro, depois calcular as janelas. Segue o código:

```
/Começo Região paralelizável - Kernel
double *serie = (double *) malloc(sizeof(double)*tam_serie);
read_serie(argv[1], serie, tam_serie);
printf("tamanho da serie: %d, tamanho da janela: %d\n",tam_serie, tam_janela);
double max, min, media;
max_min_avg(serie,tam_serie, &max, &min, &media);
printf("serie total - max: %lf, min: %lf, media: %lf\n", max, min, media);

for(int i = 0; i <= tam_serie - tam_janela; 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);
}
//Fim da região paralelizável
```

O arquivo de série temporal é fornecido como um argumento de linha de comando e lido para um vetor de double. Em seguida, é chamada a função max_min_avg para calcular o máximo, mínimo e média da série inteira.

Um laço então é feito, para poder imprimir o resultado dos cálculos de cada janela. O laço anda de um em um, do valor 0 até tam_serie - tam_janela, e indica o índice inicial para cada janela, em relação ao vetor original. A parte do vetor que começa com o índice do laço indica qual janela está. Suas características são calculadas e impressas a cada laço. Assim, todas as janelas disponíveis no vetor principal são impressas.

2. **Estratégia de paralelização utilizada**

A estratégia de paralelização utiliza o MPI e se baseia no algoritmo do produtor consumidor, onde cada processo atua como consumidor e fica responsável por calcular o máximo, mínimo e média das janelas que recebe. O processo principal (rank 0) atua como produtor e é responsável por ler a série temporal e distribuir as janelas para os outros processos. Os resultados são então impressos na saída padrão.

O processo principal (rank 0) atua como coordenador, distribuindo as janelas para serem processadas pelos outros processos. Ele usa as funções MPI_Send e MPI_Recv para enviar a posição de início de cada janela para os outros processos e receber uma mensagem de cada processo indicando quais estão prontos para receberem janelas. Quando todas as janelas foram processadas, rank 0 envia uma mensagem final para os outros processos para informá-los que eles podem sair.

Os outros processos aguardam uma mensagem de rank 0 e calculam o máximo, mínimo e média para a parte da série temporal especificada pela janela. Os resultados são então impressos na saída padrão.

O programa termina com a função MPI_Barrier, que aguarda que todos os processos terminem e o tempo de execução é calculado e impresso na saída padrão.

3. Metodologia de experimentos e especificaçẽos técnicas

- Para colher os tempos de execução para cada número de processadores, foi-se executado 20 vezes o programa e colhido a média dos tempos de execução.
- Foram realizados testes de escalabilidade fraca e escalabilidade forte. Foi criado um script em Python para automatizar os testes e cálculos.
- Para os testes, foram escolhidas entradas em que o programa sequencial demorou mais de 10 segundos para ser executado. Para o teste de escalabilidade forte consistiu num vetor de 1 milhão de números aleatórios com uma janela de tamanho 10 mil .
- O código foi executado com a **flag de compilação** -O3, utilizando o **seguinte compilador**:

  gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0

  Não foi alterada a configuração de DVFS.

- A **CPU** utilizada para os testes foi a descrita abaixo (lshw):
  produto: Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz, **8 cores**
  fabricante: Intel Corp.
  ID físico: 1
  tamanho: 3309MHz
  capacidade: 3600MHz
  largura: 64 bits

- O **Kernel** e a **versão do SO** utilizados foram os seguintes:
  - OS: Ubuntu 20.04.5 LTS
  - Kernel: 5.15.0-60-generic

4. Tempo de execução da região com código sequencial

Para medir o tempo da região puramente sequencial, utilizei de uma entrada no algoritmo sequencial em que o tempo de execução fosse maior que 10 segundos. A entrada consistiu num vetor de 1 milhão de números aleatórios com uma janela de tamanho 10 mil.

Foi definida a parte sequencial a parte do código em que as características das janelas são calculadas, ou seja, o laço **for** onde o índice de cada janela anda de um em um. Todo o código anterior a esse foi considerado como região puramente sequencial.

De acordo com as medições, a porcentagem de tempo que é decorrida na região **puramente sequencial** em relação ao tempo de execução total é de aproximadamente **0,7%** do tempo, ou seja, a **região paralelizável** ocupará então **99.3%** do tempo de execução.

5. **Tabela speedup máximo teórico - Lei de Amdahl**

	2 CPU	**4 CPU**	**6 CPU**	**8 CPU**	**16 CPU**	**MAX TEO.**
Speedup	1,98	3,91	5,79	7,62	14,47	142,85

A metodologia utilizada para medir o tempo puramente sequencial e o tempo paralelizável foi descrita na questão anterior. Os tempos foram medidos utilizando a biblioteca clock**,** e para medir o tempo sequencial, foram somados os tempos antes e depois da região paralelizável.

6. **Tabelas de Speedup e Eficiência reais**

- **Tabela para o teste de escalabilidade forte**

Foi escolhida uma entrada em que o tempo de execução no algoritmo sequencial ultrapassasse os 10 segundos. Essa entrada foi fixada então foi se alterando o número de threads. Segue o resultado:

	Vetor x Janela	1 CPU	2 CPU	4 CPU	8 CPU	16 CPU
Speedup	1000000 x 10000	1 (0.34)	0,72 (0,24)	1,69 (0.13)	2,52 (0.15)	2,66 (0.09)

OBS:: os valores representam o speedup, ou seja, a divisão do tempo de execução com 1 processo pelo tempo de execução de n processos. Os valores entre parênteses representam o desvio padrão das médias das execuções.

- **Tabela para o teste de escalabilidade fraca**

	Vetor x Janela	1 CPU	2 CPU	4 CPU	8 CPU	16 CPU
Speedup	1000000 x 100	1 (0.03)	0.21 (0.37)	0.47 (0.08)	0.49 (0.05)	0.50 (0.06)
	1000000 x 1000	1 (0.08)	0.37 (0.27)	0.84 (0.08)	1.00 (0,05)	1.03 (0.06)
	1000000 x 10000	1 (0.34)	0,72 (0,24)	1,69 (0.13)	2,52 (0.15)	2,66 (0.09)
	1000000 x 100000	1 (1.79)	0.91 (0.23)	2.18 (0.39)	3.66 (0.82)	3.62 (0.38)

7. **Análise dos resultados e discussão das tabelas**

Pode-se perceber que houve sim uma certa escalabilidade e aumento de desempenho com a solução de paralelização utilizada, porém esse desempenho somente aparece para entradas muito grandes, onde o tempo de execução do algoritmo sequencial é maior ou igual a 10 segundos. Por exemplo, para as entradas **1000000 x 10000** e **1000000 x 100000** os tempos de execução do algoritmo sequencial são em média **12 segundos** e **1 minuto**, respectivamente, e é onde houve o maior speedup registrado.

Então, para o caso em que o vetor de características é muito grande, vale a pena sim paralelizar. O problema encontrado aqui é que a complexidade do algoritmo sequencial e seus cálculos são extremamente simples, tempo um tempo de de execução linear O(n), onde para entradas pequenas o algoritmo executa muito bem. Para essas mesmas entradas, o overhead de se paralelizar se torna muito grande, para cálculos muito simples. A partir do momento em que a entrada se torna suficientemente grande, o que não é difícil acontecer no problema de extração de características no aprendizado de máquina, o overhead da paralelização acaba sendo compensado e a eficiência da solução acaba se mostrando.

Houve então um speedup extremamente considerável para essas entradas grandes, onde com 8 processos o algoritmo chega a executar até 3 vezes mais rápido em algumas situações. A solução está longe de alcançar o limite teórico disposto pela **Lei de Amdahl,** porém acredito que tendo um processador mais potente e com mais threads, com uma entrada suficientemente grande, talvez até falando em tamanho de Gigabytes, haveria então uma melhora extremamente considerável no speedup. Porém, na situação atual, mesmo sendo pouco aumento, à medida que aumentamos as threads o speedup aumenta.

Há uma deficiência porém ao utilizar somente 2 threads, onde realmente em nenhuma situação haveria algum speedup. A solução de paralelização se baseia no algoritmo do produtor consumidor. O processo 0 é responsável por calcular as características do vetor inteiro e após isso, comunicar para os outros processos quais janelas irão calcular, à medida que processos vão se disponibilizando para isso. Com dois processos isso acaba gerando somente overhead pois o processo 0 irá mandar mensagens somente para um processo, que é o 1. Então toda vez o processo 0 tem que esperar o processo 1 terminar para poder enviar mensagem sobre a próxima janela. Seria mais rápido então seguir a solução sequencial nesse caso. Mas a partir de 2 processos, já começa a haver vantagens.

8. Escalabilidade do Algoritmo Paralelo

Apresenta sim escalabilidade forte, porém de forma disfarçada. Na tabela de resultados do teste de escalabilidade forte, encontrada no ponto 6, percebe-se que à medida que dobrou-se o número de processos, entre 2, 4 e 8, o speedup seguia a mesma ordem de crescimento, dobrando a cada vez que o número de processos era dobrado. Porém para com 16 processos, acredito eu que por motivos de overhead e limitações do meu processador, o speedup seguiu aumentando porém numa ordem muito mais baixa.

Porém, como mencionado anteriormente, acredito que aumentando a entrada e melhorando a capacidade do processador, que no momento estou limitado ao que tenho em casa, essa escalabilidade se tornaria muito mais visível.

9. Considerações finais

Pode-se perceber que houve sim uma certa escalabilidade e aumento de desempenho com a solução de paralelização utilizada, porém esse desempenho somente aparece para entradas muito grandes, onde o tempo de execução do algoritmo sequencial é maior ou igual a 10 segundos.

O problema de extração de características no aprendizado de máquina geralmente lida com entradas muito maiores que a que simulamos nesse trabalho, e também são extraídas características dessas entradas que são muito mais custosas ao computador. Para esse problema, paralelizar é essencial, pois lida com ordens de grandeza em que seria impossível de executar sem a paralelização, ou pelo menos renderia um longo tempo de espera até o término. Portanto, pode-se afirmar que a solução foi sim eficiente e cumpriu o propósito de paralelizar, apresentando resultados com entradas grandes.