UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

RELATÓRIO DE ANÁLISE DE DESEMPENHO USANDO PARALELISMO

Sadi Júnior Domingos Jacinto

Professor orientador: Odorico Machado Mendizabal

Florianópolis
2019

Sadi Júnior Domingos Jacinto

RELATÓRIO DE ANÁLISE DE DESEMPENHO USANDO PARALELISMO

Análise de desempenho de algoritmos de ordenação usando paralelismo requerido pelo professor da disciplina Programação Paralela e Distribuída, Odorico Machado Mendizabal, necessário para obtenção de nota.

Professor orientador: Odorico Machado Mendizabal

Florianópolis

2019

Conteúdo

1 PROBLEMÁTICA

O presente trabalho refere-se à implementação e análise de dois algoritmos de ordenação usando processamento paralelo e usando as linguagens de programação *C* e *Java*. Tal estudo objetiva descobrir as diferenças de desempenho entre cada um dos algoritmos de ordenação, além de também analisar a diferença de desempenho entre essas duas linguagens.

2 ALGORITMOS

Os algoritmos de ordenação usados neste trabalho foram:

2.1 *BUBBLE SORT*

O Bubble Sort é um algoritmo simples que é usado para ordenar um dado conjunto de n elementos fornecidos na forma de um *array* com n número de elementos. O Bubble Sort compara todos os elementos, um por um, e classifica-os com base em seus valores.

Se a matriz determinada tiver que ser classificada em ordem crescente, então a ordenação em bolha começará comparando o primeiro elemento da matriz com o segundo, se o primeiro elemento for maior que o segundo, ele trocará os elementos e, em seguida, siga em frente para comparar o segundo e o terceiro elemento, e assim por diante.

Se tivermos n elementos totais, precisamos repetir esse processo por n-*1* vezes.

É conhecido como Bubble sort, porque a cada iteração completa o maior elemento na matriz dada borbulha em direção ao último lugar ou ao índice mais alto, assim como uma bolha d'água sobe até a superfície da água.

A classificação ocorre percorrendo todos os elementos um por um e comparando-o com o elemento adjacente e trocando-os, se necessário.

No melhor caso, o algoritmo executa n operações relevantes, onde n representa o número de elementos do vetor. No pior caso, são feitas n^2 operações. A complexidade desse algoritmo é de ordem quadrática. Por isso, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.

2.2 *MERGE SORT*

A ideia básica do Merge Sort é criar uma sequência ordenada a partir de duas outras também ordenadas. Para isso, o algoritmo Merge Sort divide a sequência original em pares de dados, ordena estes pares na ordem desejada, depois reagrupa as sequências de pares já ordenados, formando uma nova sequência ordenada dos elementos, repetindo esse processa até ter toda a sequência ordenada.
O algoritmo segue três passos comuns aos algoritmos dividir-para-conquistar:

1. Dividir:
   Dividir os dados em subsequências pequenas. Este passo é realizado recursivamente, iniciando com a divisão do vetor de n elementos em duas metades, cada uma das metades é novamente dividida em duas novas metades e assim por diante, até que não seja mais possível a divisão (ou seja, sobrem n vetores com um elemento cada).

2. Conquistar:
   Classificar as duas metades recursivamente aplicando o algoritmo do *Merge Sort*.

3. Combinar:
   Juntar as duas metades em um único conjunto já classificado. Para completar a ordenação do vetor original de n elementos, faz-se o *merge* ou a fusão dos sub-vetores já ordenados.

A desvantagem do *Merge Sort* é que requer o dobro de memória, ou seja, precisa de um vetor com as mesmas dimensões do vetor que está sendo classificado.

A vantagem (ou desvantagem, dependendo do observador), é que a complexidade do *Merge Sort* ($O(n\ log\ n)$) é a mesma para o melhor, médio e pior caso, uma vez que, independente da situação dos dados no vetor, o algoritmo irá sempre dividir e intercalar os dados.

3 AMBIENTE DE TESTE

Os testes de desempenho foram executados em um computador com as seguintes configurações:

```
CPU:
  Topology: Quad Core model: Intel Core i5-8250U bits: 64
  type: MT MCP L2 cache: 6144 KiB
  Speed: 700 MHz min/max: 400/3400 MHz Core speeds (MHz):
  1: 800 2: 800 3: 800 4: 800 5: 800 6: 800 7: 800 8: 800
Graphics:
  Device-1: Intel UHD Graphics 620 driver: i915 v: kernel
  Device-2: NVIDIA GP108M [GeForce MX150] driver: nvidia
  v: 430.40
  OpenGL: renderer: GeForce MX150/PCIe/SSE2
  v: 4.6.0 NVIDIA 430.40
```

4 METODOLOGIA

Para facilitar os testes e tornar o processo mais dinâmico possível, ambas as aplicações (em C e em Java), utilizam o *input* do usuário para definir o número de linhas, colunas, *threads* e processos.

Após o *input* do usuário com tais informações, é criado uma matriz *NxM*, onde n é a quantidade de linhas e m é a quantidade de colunas requisitadas pelo usuário, com valores aleatórios limitados ao intervalo de 0 à 1000, usando as bibliotecas *rand()*, em C, e *Math.random()*, em Java.

4.1 *THREADS*

No processamento usando *threads*, cada *thread* é criada recebendo um parâmetro inteiro, que corresponde à linha da matriz que essa *thread* irá ordenar. Para saber qual a próxima linha a ser ordenada, a *thread* acrescenta ao seu parâmetro inicial o número total de *threads*, número esse definido pelo usuário no início da aplicação e que é armazenado em uma variável global. É importante frisar que, assim como o número total de *threads* que irão executar, também é armazenado em uma variável global o número total de linhas da matriz, dessa forma, toda vez que uma *thread* vai ordenar uma linha, ela primeiramente verifica se tal linha existe na matriz, caso exista a *thread* segue seu fluxo normal (ordenar a linha, calcular qual a próxima linha a ser ordenada, repetir), do contrário a *thread* é encerrada. Ao usar essa estratégia, pode-se descartar o uso de variáveis de controle (*mutex* e *synchronized*).

4.2 PROCESSOS

Já no processamento usando processos, existem dois processos principais:

- **writer.c**:
  Responsável por criar e escrever os dados na memória compartilhada. Também é responsável pelo *output* do resultado da ordenação e por excluir, ao final do processamento, a área de memória compartilhada.

- **reader.c**:
  Responsável por ler os dados da memória compartilhada e realizar a ordenação da matriz.

É importante esclarecer que foram criadas duas áreas de memória compartilhadas

- Uma que armazena as variáveis de controle:
  - Número de linhas (int);
  - Número de colunas (int);
  - Verificação se a matriz já foi preenchida (int);
  - Próxima linha a ser ordenada (int, inicialmente 0);
  - Verificação se o processamento já acabou (int) e
  - *Mutex*
- E outra que armazena a matriz.

Através da leitura das variáveis de controle, o "leitor" pode definir qual o tamanho exato da memória compartilhada usada pela matriz (novamente ressaltando que o tamanho da matriz é dinâmico, uma vez que se baseia nos *inputs* do usuário) e se a matriz já está populada (se não, o "leitor" entra em *loop* até que a matriz seja preenchida).

Através da variável que armazena a próxima linha a ser ordenada, os subprocessos conseguem saber qual linha da matriz que eles devem ordenar. E, graças ao *mutex*, nenhum processo tenta ordenar uma linha que já pertence à outro processo (ou que já tenha sido ordenada). Isso ocorre devido ao fato de, após conseguir o *mutex*, o processo adquire o número da linha que o mesmo irá ordenar e, antes de liberar o *mutex*, incrementa o valor da variável na memória compartilhada. Nessa aplicação também existe a verificação para que o processo não extrapole os limites da matriz.

4.3 A MATRIZ

O tamanho máximo de linhas e colunas para que o algoritmo funcione sem erro é de 25.000 linhas por 25.000 colunas. Excedido este tamanho, a aplicação sempre acaba sendo morta.

4.4 DEMAIS CONSIDERAÇÕES

O primeiro algoritmo que é executado é o *Bubble Sort*. Após o término do mesmo, a matriz ordenada é restaurada a seu estado original e o segundo algoritmo (*Merge Sort*) é executado. Essa estratégia foi usada para garantir que o acaso na geração de números aleatórios não favorecesse um algoritmo em detrimento de outro. Fora isso, o tempo de processamento é calculado separadamente para cada algoritmo, não contando no cálculo o tempo para gerar a matriz e restaura-lá. Vale ressaltar que os *outputs* existentes também não afetam o cálculo de desempenho dos algoritmos.

Quaisquer dúvidas, favor entrar em contato através do endereço eletrônico `sadijacinto@gmail.com`.

5 CONCLUSÕES

5.1 CONCLUSÕES A RESPEITO DA LINGUAGEM *C*

Com relação à *threads*, o limite máximo de *threads* possíveis de serem executadas sem erro, na máquina anteriormente detalhada, é de 30.000 *threads*. É possível executar com até 33.010 *threads*, porém, se esse valor for utilizado, apesar de nenhum erro ocorrer, a aplicação não consegue finalizar, ficando "congelada".

Agora com relação aos processos, o limite máximo de processos possíveis de serem executados sem erro é de 10.330 processos.

Finalmente, de forma geral e esperada, os algoritmos de ordenação executados em C apresentaram uma velocidade maior na ordenação das matrizes em comparação com os algoritmos executados em Java, tendo em mente que as matrizes usadas por ambas as linguagens possuíam o mesmo tamanho, embora não necessariamente possuíssem os mesmos dados.

5.2 CONCLUSÕES A RESPEITO DA LINGUAGEM *Java*

Em Java, o limite máximo de *threads* se mostrou superior em comparação à linguagem C, onde foi possível, embora com enorme perda de desempenho, executar 1.000.000 *threads*, tendo em mente que também se mostrou possível executar um número maior de *threads*, o que me leva a concluir que não existem limites para o número máximo de *threads* em Java, ou que, pelo menos, esse limite é extremamente alto.

5.3 CONCLUSÕES A RESPEITO DO ALGORITMO *BUBBLE SORT*

O algoritmo apresenta um desempenho inferior ao *Merge Sort* e, conforme o tamanho da matriz aumenta, o desempenho do algoritmo diminuí (considerando o mesmo número de *threads*).

Agora, com relação a linguagem, ao manter o mesmo tamanho de matriz e o mesmo número de *threads* em ambas as aplicações, notou-se um claro desempenho superior da linguagem C em relação à linguagem Java. Como, por exemplo, em uma matriz 100x100 e com 10 *threads* executando, em C a ordenação da matriz levou 0.018756 segundos, enquanto em Java a mesma ordenação levou 0.047326123 segundos.

Para maiores detalhes e mais casos de exemplo, favor ver os *logs* de cada aplicação.

5.4 CONCLUSÕES A RESPEITO DO ALGORITMO *MERGE SORT*

O algoritmo apresenta um desempenho muito bom, mesmo com matrizes enormes e poucas *threads*. Novamente a superioridade de desempenho da linguagem C sobre a linguagem Java se faz presente. Tomando como exemplo o caso relatado acima, enquanto a ordenação em C levou 0.003700 segundos, a ordenação em Java levou 0.010905595 segundos, o que, apesar de parecer ser pouco, é uma diferença colossal de desempenho.

5.5 DEMAIS CONSIDERAÇÕES

Como era de se esperar, quanto maior o tamanho da matriz, mais tempo se leva para ordena-lá. Além disso, percebeu-se que não vale a pena criar mais *threads* do que o limite físico de núcleos da máquina, uma vez que o ganho de desempenho ao fazer isso,

quando existe, não é muito significante, além, é claro, de poder ter o efeito oposto e acabar diminuindo o desempenho da aplicação.

Finalmente, pode-se perceber que o desempenho da aplicação aumenta se a matriz tiver poucos linhas, independente do tamanho das colunas, em detrimento a uma matriz com muitas linhas mas colunas pequenas.

Por fim, mas não menos importante, vale ressaltar que, caso reste alguma dúvida com relação ao desempenho das aplicações basta executar as aplicações anexadas você mesmo e fazer seus próprios testes e tirar suas próprias conclusões.