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

PROCESSAMENTO DISTRIBUÍDO USANDO MPI

Sadi Júnior Domingos Jacinto

Professor orientador: Odorico Machado Mendizabal

Florianópolis
2019

Sadi Júnior Domingos Jacinto

PROCESSAMENTO DISTRIBUÍDO USANDO MPI

Análise do uso de processamento distribuído utilizando MPI, 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 RESUMO

Muitos problemas interessantes de otimização não podem ser resolvidos de forma exata, utilizando a computação convencional (sequencial) dentro de um tempo razoável, inviabilizando sua utilização em muitas aplicações reais.

Embora os computadores estejam cada vez mais velozes, existem limites físicos e a velocidade dos circuitos não pode continuar melhorando indefinidamente. Por outro lado nos últimos anos tem-se observado uma crescente aceitação e uso de implementações paralelas nas aplicações de alto desempenho como também nas de propósito geral, motivados pelo surgimento de novas arquiteturas que integram dezenas de processadores rápidos e de baixo custo.

Dito isso, o presente relatório tem por objetivo analisar o desempenho de uma aplicação de ordenação de vetores de inteiros em sua implementação paralela, utilizando o MPI para troca de mensagens entre diferentes processos.

2 DEFINIÇÃO DO ESTUDO

O presente relatório buscará analisar o desempenho de uma aplicação responsável por ordenar um vetor de inteiros em sua implementação paralela em comparação com sua implementação sequencial.

Para isso, os seguintes critérios foram seguidos:

- A aplicação foi inteiramente escrita na linguagem C.
- Foi utilizada a biblioteca OpenMPI para realizar a comunicação entre os diferentes processos.
- O algoritmo de ordenação usado foi o *bucket sort*, para a classificação dos dados, e a função *qsort()* para a ordenação dos mesmos.
- Tanto a versão paralela quanto a versão sequencial ordenam o mesmo conjunto de dados (copiados para ambas as versões) e utilizam o mesmo número de *buckets*, com o objetivo de os resultados não serem prejudicados por diferenças nas implementações[1].
- Foram escritos duas aplicação para realizar essa análise, uma delas dando prioridade para o processamento dos dados e outra priorizando a otimização de memória.

Tais critérios serão melhor explicados ao longo do relatório.

[1] afora o paralelismo, é claro.

3 MPI (*MESSAGE PASSING INTERFACE*)

O MPI é um padrão de interface para a troca de mensagens em máquinas paralelas com memória distribuída. Apesar de alguns pensarem dessa forma, o MPI não é um compilador ou um produto específico.

No padrão MPI, uma aplicação é constituída por um ou mais processos que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre os processos. Inicialmente, na maioria das implementações, um conjunto fixo de processos é criado. Porém, esses processos podem executar diferentes programas. Por isso, o padrão MPI é algumas vezes referido como MPMD (*Multiple Program Multiple Data*).

Elementos importantes em implementações paralelas são a comunicação de dados entre processos paralelos e o balanceamento da carga. É importante frisar que o número de processos no MPI normalmente é fixo. Dito isso, tais processos podem usar mecanismos de comunicação ponto a ponto (operações para enviar mensagens de um determinado processo a outro), ou coletivas, na qual um grupo de processos pode invocar operações coletivas de comunicação para executar operações globais.

Sobre o MPI, o mesmo é capaz de suportar comunicação assíncrona e programação modular, através de mecanismos de comunicadores que permitem ao usuário MPI definir módulos que encapsulem estruturas de comunicação interna.

Finalmente, os algoritmos que criam um processo para cada processador podem ser implementados, diretamente, utilizando-se comunicação ponto a ponto ou coletivas, sendo que os algoritmos que implementam a criação de tarefas dinâmicas ou que garantem a execução concorrente de muitas tarefas, num único processador, precisam de um refinamento nas implementações com o MPI.

4 *BUCKET SORT*

O *Bucket sort* é uma técnica de classificação que classifica os elementos primeiro os dividindo em vários grupos chamados baldes[2]. Os elementos dentro de cada grupo são classificados usando qualquer um dos algoritmos de classificação adequados ou chamando recursivamente o mesmo algoritmo.

Vários *buckets*[3] são criados. Cada *bucket* é preenchido com um intervalo específico de elementos, que então são classificados usando qualquer outro algoritmo. Por fim, os elementos de cada *bucket* são reunidos para obter a matriz ou o vetor original classificado.

O processo de classificação pode ser entendido como uma abordagem de coleta dispersa. Os elementos são primeiro dispersos em *buckets* e, em seguida, os elementos dos *buckets* são classificados. Finalmente, os elementos são reunidos em ordem.

Esse algoritmo é especialmente útil, e fácil de implementar, em aplicações paralelas, onde cada *bucket* pode ser enviado para um processo, onde o mesmo ordenará o *bucket* e o reenviará, já ordenado, para o processo de origem, o que acarretará em um aumento de desempenho com relação à sua implementação sequencial[4].

4.1 CLASSIFICAÇÃO DOS *BUCKETS*

Os elementos são classificados através de uma função matemática que, dados os valores máximos e mínimos dos elementos, além do número de *buckets* disponível, determina para qual *bucket* tal elemento será enviado.

A função matemática que é usada para calcular em qual *bucket* o elemento deve ser inserido é:

$$\frac{elem - min}{\frac{max - min + 1}{buckets}}$$

Onde:

- elem: é o elemento que se deseja inserir em algum *bucket*;
- min: menor elemento presente no vetor ou matriz;
- max: maior elemento presente no vetor ou matriz e
- buckets: número total de *buckets* existentes.

4.2 COMPLEXIDADE

- **Pior Complexidade ($\theta(n^2)$)**:
  Quando a distribuição dos elementos não for uniforme, os mesmos provavelmente serão colocados no mesmo *bucket*. Isso pode resultar em alguns *buckets* com mais número de elementos do que outros, o que irá atrasar a execução consideravelmente.

  Isso faz com que a complexidade dependa do algoritmo de ordenação usado para ordenar os elementos do *bucket*.

  A complexidade se torna ainda pior quando os elementos estão na ordem inversa.

[2] ou *buckets* em inglês, daí o nome.
[3] deste ponto em diante, o termo *bucket* será utilizado ao invés do termo balde.
[4] pelo menos em teoria

- **Melhor Complexidade ($\theta(n+k)$)**:
  Ocorre quando os elementos são distribuídos uniformemente nos *buckets*, com um número quase igual de elementos em cada *bucket*.

  A complexidade se torna ainda melhor se os elementos dentro dos *buckets* já estiverem classificados.

  Se o algoritmo de ordenação utilizado for de tempo linear, a complexidade geral, na melhor das hipóteses, será linear, isto é, $\therefore (n+k)$m onde $\therefore (n)$ representa a complexidade para fazer os inserir os elementos nos *buckets* e $\theta(k)$ representa a complexidade para ordenar os elementos do *bucket*[5].

- **Caso Médio ($\theta(n)$)**:
  Isso ocorre quando os elementos são distribuídos aleatoriamente na matriz. Mesmo que os elementos não sejam distribuídos uniformemente, a classificação do *bucket* é executada em tempo linear.

[5] considerando que os algoritmos usados são de complexidade de tempo linear

5 qsort()

Trata-se de uma função disponibilizada pela biblioteca *stdlib.h* utilizada para ordenação de vetores. Possuí a sintaxe:

```
1 void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void
    * p1, const void* p2))
2
```

Onde:

- **base*:
  Ponteiro para o primeiro elemento do vetor;
- *nitems*:
  Número de elementos do vetor;
- *size*:
  Tamanho em *bytes* de cada elemento do vetor, é, obrigatoriamente, um inteiro positivo e
- **compar*:
  Função criada para o problema e que compara dois elementos.

A função **compar* é o que garante o polimorfismo dessa função, uma vez que se trata de um ponteiro para outra função, que é definida pelo desenvolvedor e que, portanto, pode ser implementada para ordenar qualquer tipo de dado.

A função que o ponteiro da função **qsort** apontar será chamada sempre com dois elementos e seu retorno ditará se os elementos serão trocados de lugar ou não. Para ficar mais claro, os valores de retorno da função explicitada são:

- <0:
  O elemento apontado por *p1* vai antes do elemento apontado por *p2*.
- 0:
  O elemento apontado por *p1* é equivalente ao elemento apontado por *p2*.
- >0:
  O elemento apontado por *p1* vai depois do elemento apontado por *p2*.

6 SOBRE A IMPLEMENTAÇÃO

7 RESULTADOS

8 CONCLUSÕES