Gabriela Lemos Lúcidi Pinhão e Andrew Cruz

Sistemas Distribuídos – Trabalho 2

A linguagem utilizada para a implementação de programas *multithreading* foi a C++.

Primeiro programa – somadorSL.cpp

O primeiro programa tem como objetivo somar os valores de um vetor cujo comprimento é da ordem de centenas de milhões de valores. Para contornar isso, foram utilizadas threads de forma que o vetor original pudesse ser dividido em várias parcelas, as somas destas fossem calculadas e finalmente ocorresse a soma dos resultados obtidos.
Quando uma thread acaba de calcular a soma, ela deve guardar o resultado na variável *soma*, que é compartilhada entre as threads. Esse problema foi resolvido com a utilização de um spinlock implementado com a ajuda da instrução atômica test_and_set oferecida pela biblioteca <atomic> do C++.

```
class Spinlock
{
   std::atomic_flag locked = ATOMIC_FLAG_INIT;
public:
   void aquire()
   {
      while(locked.test_and_set()){}
   }
   void release()
   {
      locked.clear();
   }
};
```

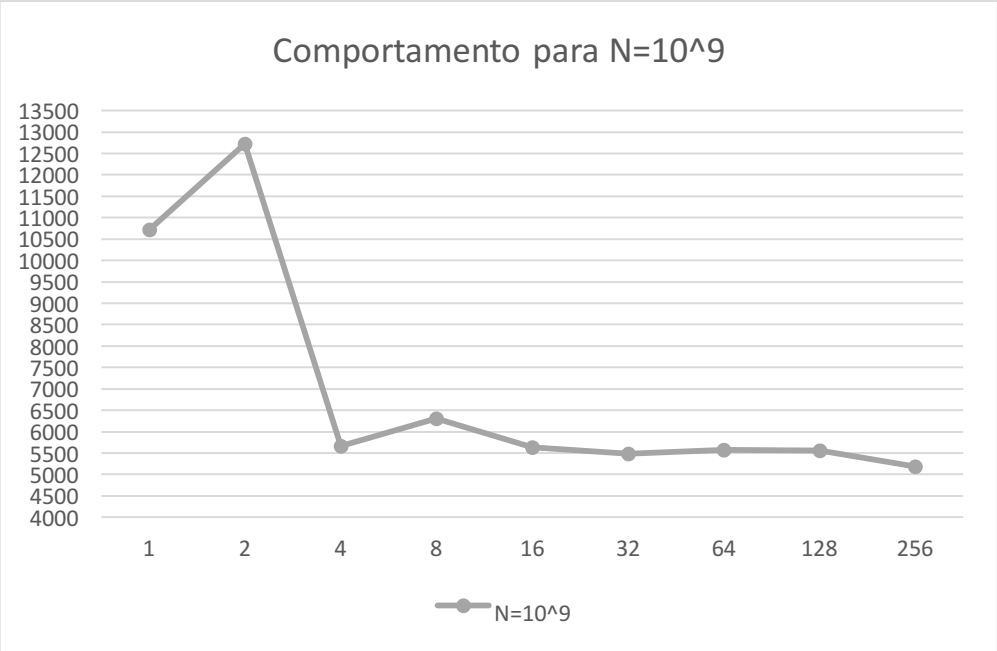
Módulos auxiliares:

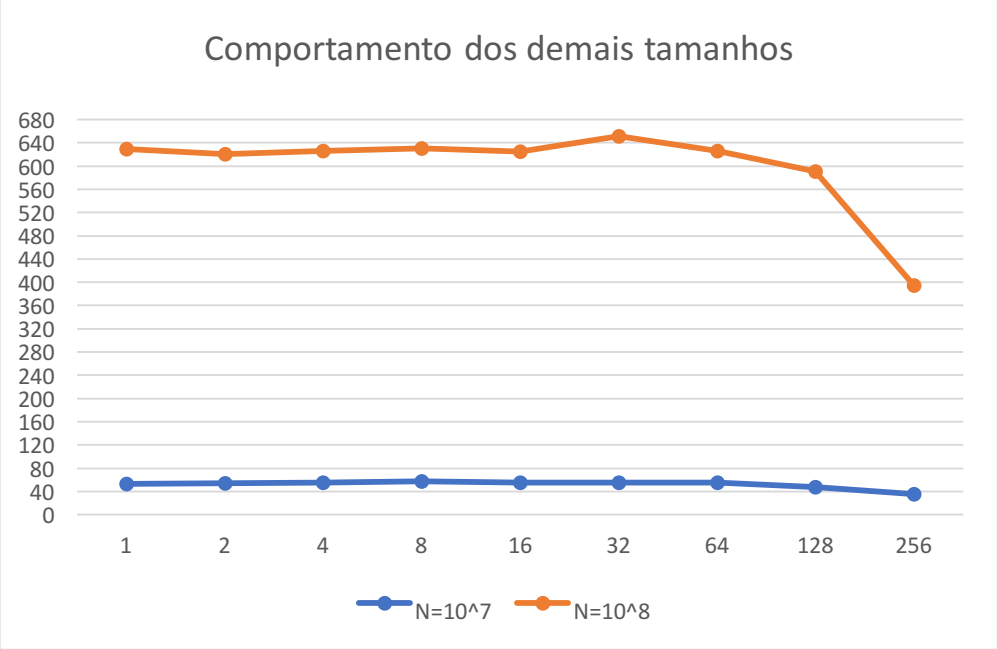
void ***new_thread***(void *arg): toda vez que uma thread é criada, essa função é executada. Ela associa a thread a um trecho do vetor e a incumbe de realizar a soma dos valores contidos nele. O resultado é armazenado na variável utilizando o spinlock definido anteriormente para evitar condições de corrida. A thread então é encerrada.

A função *main* recebe como parâmetros o número de threads a serem usadas (2^n, com n=1,2,...,8) e o tamanho do vetor a ser processado (entre 10^7 e 10^9). O vetor *valores* é redimensionado para o tamanho 10^n onde n é o parâmetro de entrada,o vetor então é preenchido com valores entre -100 e 100;
O programa divide o vetor em parcelas de tamanhos iguais e posteriormente é criado um vetor que irá armazenar as K threads. As threads então são criadas e armazenadas nesse vetor.

Agora as threads sofrem um join e passam a rodar, tendo o tempo gasto calculado da seguinte forma: o programa registra o PC time antes do join e após ele, depois realiza (*t_fim – t_inicio).*

Para fins de experimento, o programa foi rodado pelo menos dez vezes para cada combinação de N e K de modo que fosse possível calcular uma tempo médio mais confiável. Os gráficos a seguir apresentam os resultados encontrados:





De acordo com os gráficos, quando o N=10^9, o tempo de processamento dispara entre uma e quatro threads. Após isso, o gráfico se estabiliza entre 5 e 6 segundos. Para N=10^7 o tempo gasto foi praticamente constante. Esse comportamento pode ser explicado pela baixa precisão da medida, uma vez que o tempo é muito pequeno,

da ordem de 10^-5 segundos. Nessa faixa de valores, o tempo pode ser dominado por operações da CPU e interrupções que não estão associadas ao processo principal. Isso poderia ser evitado utilizando vetores maiores, que exigiriam mais operações. No entanto, a máquina virtual utilizada possui pouca memória, impossibilitando essa abordagem.
Para N=10^6 o tempo gasto na operação varia muito pouco até que foram utilizadas 256 threads, quando o tempo caiu bruscamente, caindo para dois terços do tempo medido em K=128.

Segundo programa – producerCunsumer.cpp

O segundo programa propôs a implementação *multithreaded* de produtores e consumidores, onde o primeiro escreveria o produto numa memória compartilhada limitada e o segundo faria a leitura. A sincronização e coordenação do acesso à memória compartilhada deveria ser feito por meio de semáforos.

As primeiras funções implementadas são auxiliares à atividade principal do programa, sendo utilizadas pela função *main* e suas threads:

- Int **isPrime**(int number): Tenta dividir um número especificado em seu argumento por todos os números do intervalo 2 até o número que antecede ele mesmo. Se alguma divisão proceder, o número não é primo e a função retorna o valor 0. Será primo caso contrário, retornando o valor 1.
- int **getRandomNumber**(int min, int max): Faz uso das funções *srand* e *rand* da biblioteca *stdlib* para obter um número aleatório. Para que o número aleatório esteja entre a faixa especificada nos argumentos da função, ele é dividido pelo valor da faixa e depois o módulo da divisão é somado ao valor mínimo, formando o número a ser retornado.
- int **getFirstFreePosition**(vector<int> memory): Itera todas as posições de um vetor até encontrar a primeira livre, aqui representada pelo valor 0. Retorna o número de tal posição e -1 caso não haja nenhuma posição livre.
- int **getFirstFullPosition**(vector<int> memory): Itera todas as posições de um vetor até encontrar a primeira cheia, aqui representada por um valor diferente de 0. Retorna o número de tal posição e -1 caso não haja nenhuma posição cheia.
- bool **isTotallyFree**(vector<int> memory): Faz uso da função getFirstFullPosition e retorna se todas as posições do vetor estão livres ou não.
- bool **isTotallyFull**(vector<int> memory): Faz uso da função getFirstFreePosition e retorna se todas as posições do vetor estão cheias ou não.

A função ***main*** do programa recebe como parâmetros Np e Nc, que indicam o número de threads do produtor e do consumidor, respectivamente. O número total de threads é a soma desses dois. Fazendo uso da biblioteca *thread*, primeiramente foi criado um container para todas as *threads*:

```
thread allThreads[totalNumberOfThreads];
```

E depois, para cada posição do container, foi definida qual função implementaria o comportamento das *threads* (**consumer** ou **producer**) através da função *thread*:

```
for (int i = 0; i < Np; ++i)
{
   allThreads[i] = thread(producer);
}

for (int i = Np; i < Np+Nc; ++i)
{
   allThreads[i] = thread(consumer);
}
```

Ainda usando a biblioteca *thread,* todas as *threads* foram incorporadas à função *main,* através da função *join*:

```
for (int i = 0; i < Np+Nc; ++i)
{
   allThreads[i].join();
}
```

A função que implementa a *thread* produtora começa com um *while* que limita a produção de produtos para 10000. O produto é produzido fazendo uma chamada a função auxiliar getRandomNumber fora da região crítica, pois nenhuma variável é compartilhada com outras *threads* e, portanto, não há problemas caso ocorra troca de contexto. O semáforo foi implementado fazendo uso das bibliotecas *mutex* e *condition_variable*:

```
unique_lock<mutex> lock(semaphoreMutex);
if(semaphoreEmpty.wait_for(lock, chrono::milliseconds(200), [] {return
!isTotallyFull(sharedMemory);})){
        //Escreve produto na memória compartilhada de tamanho N
        semaphoreFull.notify_all();
}
```

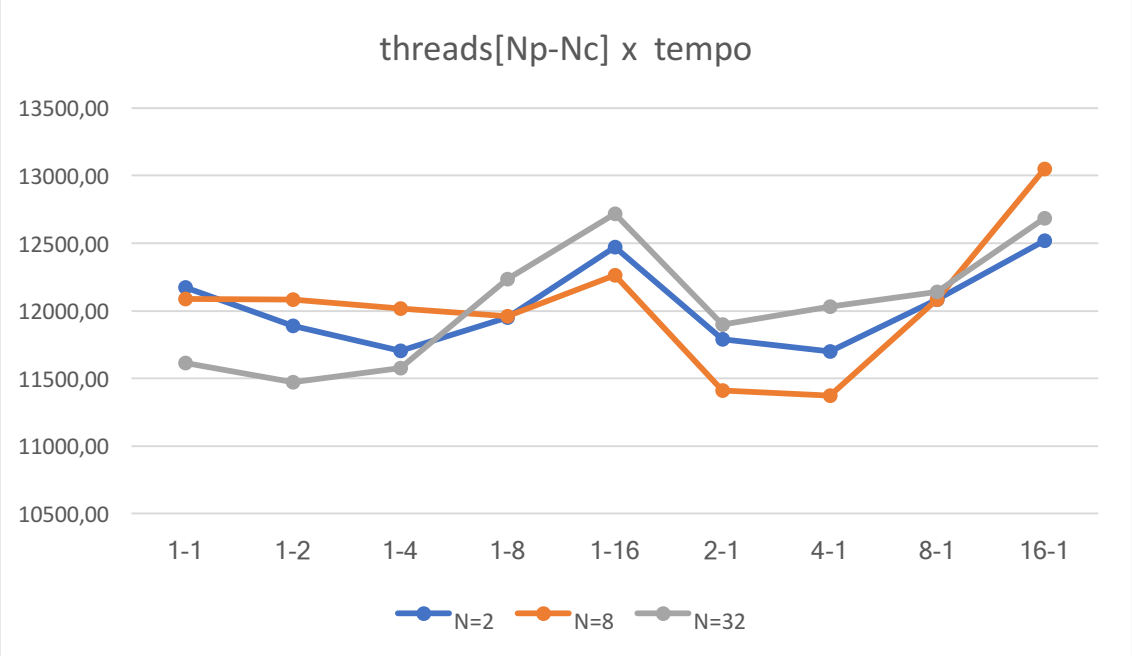
Quando a *thread* produtora em execução lê na função *wait_for* ela checa se o vetor tem posições livres, fazendo uso da função auxiliar isTotallyFull. Se ele não tiver, a thread é bloqueada e colocada na fila de *threads* em espera. Ela é colocada em estado *ready* para continuar sua execução caso o semáforo contador *empty* seja incrementado/notificado pela função *notify_all* ou então quando passar o tempo definido como argumento da função *wait_for*. Quando a thread voltar a ser executada e a condição for satisfeita, o semáforo mutex será trancado, garantindo que a região crítica não será interrompida, e destrancado depois que o produto for escrito na memória compartilhada. Por fim, os semáforos *full* são incrementados/notificados.

A função que implementa a *thread* consumidora funciona de forma semelhante, com a diferença que usa o semáforo *full* (usando a função auxiliar isTotallyEmpty como condicional) e incrementa/notifica o semáforo *empty*:

```
unique_lock<mutex> lock(semaphoreMutex);
if(semaphoreFull.wait_for(lock, chrono::milliseconds(200), [] {return
!isTotallyFree(sharedMemory);})){
   //Lê produto da memória compartilhada, libera a posição e escreve produto na
memória local
   semaphoreEmpty.notify_all();
}
int isPrimeNumber = isPrime(consumerProduct);
```

A verificação se o número é primo ou não acontece fora da região crítica, fazendo uso da função auxiliar isPrime.

Na medição do tempo para diferentes números de threads consumidoras (Nc) e produtoras (Np) e também diferentes tamanhos de memória compartilhada (N), foram tomados os seguintes cuidados: Foi utilizada a biblioteca *chrono* cujas funções permitiram a obtenção do "tempo de relógio" entre o início e fim da execução de todas as threads; Não foram incluídas na medição do tempo a criação de threads, vetores, ou quaisquer outras estruturas de dados, restringindo a medição de tempo a atividade principal do programa. Os resultados podem ser observados no gráfico a seguir:



De maneira geral, em todas as variações de memória, pode ser observado que um pequeno aumento do número de *threads* produtoras ou consumidoras provocou diminuição no tempo de execução, porém o grande aumento acarretou seu aumento. Isso acontece porque o aumento do número de threads é somente para umas das

duas entidades (produtor ou consumidor). Pegando como exemplo 1 thread consumidora e 16 produtoras, muito provavelmente a thread consumidora não dará conta de processar os produtos na velocidade em que são gerados e muito provavelmente a memória ficará com todas as suas posições cheias, fazendo com que as threads produtoras tenham que aguardar uma posição vazia para escrever. Desta forma, o aumento do número de threads consome recursos e não ajuda na melhoria da performance do processo.
Vale ressaltar, também, que durante a medição havia outros programas em execução na máquina que podem ter afetado o tempo de execução deste programa.

Código:
https://github.com/gabrielalucidi/sistemasdistribuidos2018.1

Fontes:
https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial/
https://stackoverflow.com/questions/26583433/c11-implementation-of-spinlock-using-
atomic?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa
https://austingwalters.com/multithreading-semaphores/
http://pt.cppreference.com/w/cpp/thread/condition_variable
http://en.cppreference.com/w/cpp/thread/condition_variable/notify_all
http://en.cppreference.com/w/cpp/thread/condition_variable/wait_for
http://pt.cppreference.com/w/cpp/atomic/atomic
http://www.cplusplus.com/reference/thread/this_thread/yield/