Programação Concorrente
Report

Nuno Neto
up201703898

June 2020

1 Introdução

Neste trabalho queríamos implementar várias estruturas de dados de uma forma concorrente utilizando várias estratégias. As estratégias que foram utilizadas foram as seguintes:

- Monitores e locks.
- Lock free, utilizando variáveis atómicas e quartos para controlar o acesso.
- STM, utilizando memória transacional para transformar um conjunto de instruções num bloco atómico.

As estruturas de dados que pretendíamos implementar são as seguintes:

- Bounded queues, em que operações de add(E e), remove() bloqueiam quando a lista está cheia ou vazia, respetivamente.
- Unbounded queues, em que apenas operações de remove() bloqueiam quando a lista está vazia.
- Double ended unbounded queues, em que apenas operações de removeFirst(), removeLast() bloqueiam, quando a lista está vazia.

2 Implementações

2.1 MBQueue

Esta implementação utiliza locks e monitores para controlar o acesso à memória partilhada. Para implementar foi utilizado o **synchronized**, wait() e notifyAll() . As implementações foram feitas da seguinte maneira:

2.1.1 Variáveis utilizadas:

```
protected E[] array;
protected int head, size;
```

Para implementar esta queue não utilizamos variáveis especiais pois esta implementação garante-nos que não iremos ter mais que uma thread a aceder à zona crítica concorrentemente, logo não necessitamos de garantir atomicidade nas operações.

2.1.2 Size

O método size foi implementado da seguinte forma:

```
public synchronized int size() { return size; }
```

O uso de synchronized permite a que apenas uma thread tenha acesso à lock do Object ao mesmo tempo, permitindo apenas uma operação concurrentemente.

2.1.3 Add

O método add foi implementado da seguinte forma:

```
public synchronized void add(E e) {
    while (size == array.length) {
    // queue is full
        try {
          wait();
        }
        catch (InterruptedException e) {
          throw new UnexpectedException(e);
        }
  }
  array[(head + size) % array.length] = elem;
  size++;
  notifyAll();
}
```

Este método usa o synchronized da mesma maneira que o size, contudo o add usa também o wait(), notifyAll() da seguinte maneira:

- O método remove() fica em wait() quando a queue está vazia, à espera que algum add(E) ponha algum elemento disponível para ele remover. Quando é efetuado um add(E), este chama o notifyAll() para acordar as threads que estão à espera e as notificar que um elemento está disponível para ser eliminado, a primeira thread a ser executada faz as suas alterações e o resto volta a entrar em estado de espera.

- O método add(E) fica em wait() quando a queue está cheia, à espera que algum remove() remova algum elemento que faça com que haja um local para adicionar o novo elemento. Quando o remove() é efetuado, este chama o notifyAll() para acordar as threads em espera para tentarem adicionar o seu elemento. A primeira thread a ser executada faz as suas alterações e o resto volta a entrar em estado de espera.

2.1.3.1 Bugs resolvidos:

Duplo Synchronized:

Ao ter dois synchronized separados (Um para a parte essencial do código, outro para sincronizar o acesso ao size) o que acontecia é que outra thread poderia ter acesso à zona crítica antes de a lock ser dada ao synchronize que pretendia modificar o size ou seja uma thread iria ter acesso a um estado em que o elemento já foi adicionado ao array, contudo ainda não foi incrementado o size, isto levaria a que elementos fossem *overwritten* e dados fossem perdidos pois após ter dado *overwrite* aos dados, o size iria ser incrementado como se tivessem sido adicionados 2 elementos à queue, contudo no array apenas se pode encontrar um deles. Uso de notify() em detrimento do notifyAll():

Com o uso de notify() que é um método não determinístico, não podemos saber qual das threads que está à espera vai ser acordada, logo existe a possibilidade de alguma thread ficar em espera permanentemente pois nunca é acordada. Com o notifyAll() temos que todas as threads são acordadas e apenas a primeira a adquirir o lock consegue fazer a sua ação, contudo a possibilidade de alguma das threads ficar permanentemente em espera é removida pois todas as threads têm a sua hipótese de realizar as suas operações.

2.1.4 Remove

O método remove foi implementado da seguinte forma:

```
public synchronized E remove() {
  E elem = null;
    while (size == 0) {
      // queue is empty
      try {
        wait();
      }
      catch (InterruptedException e) {
        throw new UnexpectedException(e);
```

```
            }
        }
        elem = array[head];
        array[head] = null;
        head = (head + 1) % array.length;
        size--;
        notifyAll();
        return elem;
    }
```

Este método usa o synchronized da mesma maneira que o size e o add, para limitar o acesso à memória partilhada. Tal como o add, este método usa também o wait(), notifyAll() da seguinte forma:

- Quando o método add(E) fica em wait() à espera que a queue tenha um local disponível para ser adicionado. Quando é efetuado um remove(), fica uma posição disponível e portanto é chamado o notifyAll() para acordar as threads que estão à espera de uma chance de adicionar o elemento a primeira thread a executar consegue adicionar, o resto das threads volta a entrar no estado de espera.

- Quando o método remove() fica em wait() à espera que a queue tenha algum membro disponível para ser removido. Quando é efetuado um add(E), fica um elemento disponível para ser removido, logo é chamado notifyAll() para acordar as threads que estão à espera de ter um elemento disponível para remover. A primeira thread a ser executada faz as suas alterações e todas as outras voltam a entrar em estado de espera.

2.1.4.1 Bugs resolvidos:

Duplo Synchronized:

O duplo synchronized estava a causar o mesmo tipo de problemas que no add(E), contudo em vez de dar *overwrite* em alguns elementos iria retornar certos elementos repetidamente, pois outra thread teria acesso à zona crítica antes que a thread original tivesse tempo de retificar a remoção no size.

Uso de notify() em vez de notifyAll():

De novo, da mesma maneira que no add(E) o que poderia acontecer seria que como o método notify() não é determinístico, poderíamos ver threads que ficariam bloqueadas permanentemente à espera do sinal de notify(), o que não acontece com o notifyAll() pois todas as thread são despertadas.

2.1.5 MBQueueU

Para criar a MBQueue unbounded, não foi necessário a criação de novas variáveis nem de qualquer alteração às variáveis existentes nem foi necessária a alteração aos métodos remove() e size().

O método add(E) foi feito da seguinte forma:

```
public synchronized void add(E elem) {
    if (size == array.length) {
        E[] newArray = (E[]) new Object[array.length * 2];
        int head = this.head;
        for (int i = 0; i < size; head = (head + 1) % array.length) {
            newArray[i++] = array[head];
        }
        this.head = 0;
        array = newArray;
    }
    array[(head + size) % array.length] = elem;
    size++;
    notifyAll();
}
```

Como podemos ver, em vez de termos o caso de espera quando o array se encontra cheio, temos a criação de um novo array com o dobro do tamanho do array atual. Após a criação, todos os elementos do antigo array são movidos (Por ordem) para a posição 0 do novo array, a head é relocalizada, o elemento é adicionado e prosseguimos como normal.

2.2 STMBQueue

Esta implementação utiliza Software Transactional Memory (STM) para transformar operações que não são atómicas por natureza em operações atómicas que podem ser executadas concorrentemente com segurança.

STM permite-nos criar blocos de leituras e escritas sobre objectos atómicos (chamados casos base) que mesmo após compostos, mantêm as suas propriedades atómicas e são designados por **Transações**.

A implementação foi feita da seguinte forma:

2.2.1 Variáveis utilizadas:

```
private final Ref.View<Integer> size;
private final Ref.View<Integer> head;
private final TArray.View<E> array;
```

Vemos então dois inteiros atómicos *size* e *head* que funcionam de maneira muito semelhante ao encontrado em LFBQueue, contudo em vez de termos um *head* e um *tail* que incrementam os dois indefinitivamente, temos apenas um *head* que é incrementado e para obtermos a posição *tail*, apenas somámos o *size* ao *head*.

Temos também um array que garante acessos e escritas atómicas, *array*.

2.2.2 Size

```
public int size() {
    return size.get();
}
```

Como armazenamos o tamanho numa variável atómica, podemos simplesmente retornar o valor armazenado.

2.2.3 Add

O método add(E e) foi implementado da seguinte forma:

```
public void add(E elem) {
  STM.atomic(() -> {
     if (size.get() == array.length()) {
       STM.retry();
     }
     array.update((head.get() + size.get()) %
                  array.length(), elem);
     STM.increment(size, 1);
   });
}
```

Utilizamos o STM.atomic() para criar uma transação que garante que a operação será feita atomicamente tornando-se assim *thread-safe*. Quando o tamanho atual da queue é igual à capacidade da mesma, temos que a operação não se pode realizar e deve entrar em estado bloqueante. Para fazer isto utilizamos o STM.retry() que tal como o nome indica tenta fazer a operação novamente. Quando o tamanho é inferior obtemos a posição atual da tail da queue, com head.get() + size.get() \% array.length() e modificamos o elemento localizado nessa posição. Após modificar, incrementamos o tamanho da queue e está terminada a operação.

2.2.3.1 Bugs resolvidos

Dois blocos STM.atomic() separados:

Ter dois blocos separados de STM.atomic() garante que cada um desses blocos são executados atomicamente contudo não garante absolutamente nada sobre a atomicidade dos dois blocos seguidos. O que isto gera é um problema semelhante ao problema encontrado em MBQueue (Secção 2.1.3.1), em que o elemento é colocado na array, contudo como o incremento do size está localizado noutro bloco, pode ser executada outra operação antes da alteração ao size ser feita, o que pode causar a que algum elemento seja *overwritten* ou seja retornado mais que uma vez.

2.2.4 Remove

O método remove() foi implementado da seguinte forma:

```
public E remove() {
```

```
    return STM.atomic(() -> {
      if (size.get() == 0)
        STM.retry();
      E elem = array.apply(head.get());
      head.set((head.get() + 1) % array.length());
      STM.increment(size, -1);
      return elem;
    });
  }
```

Para realizar o remove(), temos primeiro que assegurar que temos elementos disponíveis na queue para serem removidos, caso este não seja o caso, utilizamos o STM.retry() para entrar num loop bloqueante até existir um elemento para remover. Quando existe um elemento para remover, obtemos o elemento que está na *head* da queue, incrementamos a *head*, decrementamos o size da queue e retornamos o elemento.

2.2.4.1 Bugs resolvidos

Blocos separados de STM.atomic():

Tal como no método add(E) pudemos observar que havia 2 blocos distintos de STM.atomic() um que verificava se havia elementos para remover e bloqueava até haver e outro bloco que fazia as alterações à array e retornava o resultado. Tal como no add(E) sabemos que 2 blocos atómicos seguidos não são garantidos de ser atómicos, portanto o que poderia acontecer seria que entre passar do bloco que verifica se existem elementos para ser removidos e a remoção em si, poderia haver outra thread que remova o elemento da queue, causando com que a operação de remoção que se sucede retorne um elemento não esperado, pois está a retornar com a queue vazia.

2.2.5 STMBQueueU

Para implementar uma Queue unbounded utilizando STM, foram efetuadas as seguintes alterações:

2.2.5.1 Variáveis alteradas

```
private final TArray.View<E> array; ->

private final Ref.View<TArray.View<E>> arrayRef;
```

Foi necessário alterar a maneira de armazenar o array pois não podemos mais manter uma referência a um array pois estes não podem ser redimensionados, em vez disso guardámos uma referência que pode ser alterada para um array, dando-nos a possibilidade de criar um novo array com um novo tamanho e colocar a referência a apontar para este.

2.2.5.2 Add

O método de add foi feito da seguinte forma:

```
public void add(E elem) {
  STM.atomic(() -> {
    TArray.View<E> arrays = arrayRef.get();
    if (size.get() == arrays.length()) {
      TArray.View<E> newArray =
              STM.newTArray(arrays.length() * 2);
      int head = this.head.get();
      for (int i = 0; i < size.get();
              head = (head + 1) % arrays.length()) {
        newArray.update(i, arrays.apply(head));

        i++;
      }
      this.head.set(0);
      this.arrayRef.set(newArray);
      arrays = newArray;
    }
    arrays.update((head.get() + size.get()) %
                  arrays.length(), elem);
    STM.increment(size, 1);
  });
}
```

Podemos ver que as alterações realizadas foram principalmente no que acontece quando atingimos a capacidade máxima da Queue. Neste caso, em vez de entrar num estado bloqueante à espera que algum elemento seja removido, criamos uma nova array, movemos os elementos em ordem para este novo array e modificamos todas as referências que é necessário modificar.

2.2.5.3 Remove

As únicas alterações feitas no método remove foi a adição de uma linha:

```
TArray.View<E> array = arrayRef.get();
```

De modo a suportar a nossa alteração às variáveis, pois agora guardamos a referência para o array em vez de guardar o array em si.

2.3 LFQueue

Esta implementação utiliza variáveis atómicas e Rooms para controlar o acesso às zonas críticas: Para implementar foram utilizados as seguintes estruturas: AtomicInteger, Rooms e um array normal.

A classe Rooms funciona da seguinte forma:

Quando entramos no quarto x, podem entrar n threads nesse mesmo quarto. Contudo, quando entramos no quarto x, não podemos entrar no quarto y ao mesmo tempo. Após sairem todas as threads do quarto x, já podem entrar noutro quarto y.

2.3.1 Bugs encontrados:

No código inicial pudemos observar que se tentarmos executar varios tipos de ações ao mesmo tempo (Por exemplo executar 2 ou mais das 3 operações possiveis (Size, Add e Remove)). O que acontecia para isto não poder ser feito? Por exemplo no caso de executar um Add concurrentemente a um remove com uma queue com 0 elementos, uma linearização possível seria a seguinte: O Add incrementa a tail de modo a reservar um lugar no array, após isto, mas antes da posição do array ser preenchida com o elemento, um remove obtém a head que é agora uma posição válida pois o add já incrementou a tail, retorna a posição que está armazenada no índice da array que seria NULL, pois o método Add ainda não preencheu a posição que reservou. Esta linearização é apenas um dos casos possíveis que daria origem a problemas de execução. Por este motivo é que precisamos de utilizar a classe Rooms para controlar estes acessos.

As implementações foram feitas da seguinte maneira:

2.3.1.1 Variáveis utilizadas:

```
private E[] array;
private final AtomicInteger head, tail;
private final Rooms rooms;
private final boolean useBackoff;
```

Temos então um array normal, junto com dois inteiros atómicos que guardam a posição atual da *head* e da *tail*. Estes dois arrays apenas crescem e para obter o local no array fazemos então a operação head \% array.length.

O array não precisa de ter nenhum tipo de processo de sincronização pois nunca iremos aceder ao mesmo local em threads diferentes, como vamos ver mais à frente.

Temos também uma instância da class Rooms que serve como controlo de acesso em certas áreas críticas, como especificado anteriormente.

2.3.1.2 Size

O size é feito da seguinte forma:

```
public int size() {
    rooms.enter(SIZE_ROOM);
    int size = tail.get() - head.get();
    rooms.leave(SIZE_ROOM);
    return size;
}
```

Primeiro entramos no quarto correspondente ao método size(), calculamos o tamanho, saímos do quarto e retornamos o tamanho calculado.

2.3.1.3 Add

O add é feito da seguinte forma:

```
public void add(E e) {
  while (true) {
    rooms.enter(ADD_ROOM);
    int p = tail.getAndIncrement();
    if (p - head.get() < array.length) {
      array[p % array.length] = elem;
      break;
    } else {
      tail.getAndDecrement();
      rooms.leave(ADD_ROOM);
      if (useBackoff)
        Backoff.delay();
    }
  }
  rooms.leave(ADD_ROOM);
  if (useBackoff)
    Backoff.reset();
}
```

Utilizamos um loop até conseguirmos inserir com sucesso no array pois o add é feito às "tentativas", dentro deste loop é executada a lógica em si. Primeiro entramos no quarto destinado ao add, o ADD_ROOM, após isto obtemos o valor atual da tail e incrementamos, pois após este add, vamos ter mais um elemento no array. Após incrementar vemos se a array tem espaço para ter o elemento armazenado. Se não houver posições disponíveis, revertemos a alteração do tail e saimos do quarto, para dar hipótese às outras threads de remover elementos para dar espaço para adicionar.

Como podemos ter várias threads a adicionar?

Quando fazemos tail .getAndIncrement() que é uma operação atómica fica a posição retornada reservada apenas à thread que fez o getAndIncrement(), pois quaisquer threads que a sucedam se fizerem getAndIncrement(), vai retornar um número superior. Quando a operação de inserção falha, a tail é decrementada e o lugar previamente reservado à thread, fica disponível para as outras threads se sucederem. Estas "reservas" são possíveis pois estamos a utilizar inteiros atómicos o que implica que é impossível uma thread obter o mesmo resultado da operação getAndIncrement(), mesmo que seja executado em várias threads ao mesmo tempo.

Uso de backoff:

Utilizamos o backoff nas situações em que a thread iria ficar bloqueada num loop infinito, à espera que alguma outra thread remova um elemento. Ao utilizar

backoff, em vez de ter a thread a ocupar CPU a executar iterações do loop, a thread entra num estado de *sleep*, poupando assim tempo de CPU e permitindo-lhe executar outras threads que podem desbloquear o nosso estado. Ao acabar a nossa operação, voltamos a resetar o tempo do Backoff ao seu valor inicial.

2.3.1.4 Remove

O remove é feito da seguinte forma:

```
public E remove() {
    E elem = null;
    while(true) {
      rooms.enter(REMOVE_ROOM);
      int p = head.getAndIncrement();
      if (p < tail.get()) {
        int pos = p % array.length;
        elem = array[pos];
        array[pos] = null;
        break;
      } else {
        // "undo"
        head.getAndDecrement();
        //We must give the chance to other threads
        //To fill the array or we'll be stuck here forever
        rooms.leave(REMOVE_ROOM);

        if (useBackoff)
          Backoff.delay();
      }
    }
    rooms.leave(REMOVE_ROOM);
    if (useBackoff)
      Backoff.reset();
    return elem;
}
```

Da mesma maneira que no método 2.3.1.3 (Add), corremos um loop infinito em que por cada iteração, entramos no quarto da operação *REMOVE* para certificar que podemos efetuar a operação, reservámos o index que queremos extrair, com a operação head.getAndIncrement(). A seguir verificamos que não estamos a ultrapassar a tail (Significaria que já tinhamos chegado ao fim da queue, logo não há mais nenhum elemento para retirar. Quando acontece isto, voltamos a reduzir a head, efetivamente libertando a nossa reserva daquele index, após isto saímos do quarto respetivo para dar a oportunidade à operação *ADD* de adicionar novos elementos e libertar então o deadlock. Quando a operação de remoção é feita com sucesso, também necessitamos de sair do quarto, pois caso contrário, nenhuma thread teria acesso a adicionar, ou verificar o tamanho da

queue.

Como podemos ter várias threads a remover?

O remove mantém as propriedades do add que permitem a execução concorrente deste método nomeadamente a "reserva" atómica do indíce que pretendemos remover da Queue, impossibilitando outras threads que estejam a correr ao mesmo tempo de remover este mesmo elemento.

Uso de backoff:

O backoff também é utilizado de maneira muito semelhante ao add, mas em vez de estarmos à espera que alguma thread remova um elemento, esperamos que alguma thread adicione um elemento para poder ser removido. Da mesma maneira utilizar o Backoff faz com que a thread entre num estado de *sleep*, poupando assim tempo de CPU. Após ter concluído a operação com sucesso, repomos o tempo de Backoff ao seu valor original e retornámos o resultado.

2.3.2 LFBQueueU

Para implementar a unbounded queue, foram efetuadas as seguintes alterações:

2.3.2.1 Variáveis alteradas:

Para implementar esta queue, necessitamos de criar uma nova variável:

```
private final AtomicBoolean addElementFlag;
```

Esta variável serve para controlar a entrada na zona crítica por parte do add. Como vamos ter que dar resize ao array, não podemos ter mais que uma thread a efetuar um add ao mesmo tempo pois poderia-se dar origens a situações como haver a criação no novo array com tamanho incrementado mais que um vez, causando uma perda de informação pois a thread que está a criar um array não sabe das alterações que as outras threads estão a executar. Para evitar isto limitamos o acesso à zona critica a 1 thread de cada vez.

2.3.2.2 Size

Não foi necessária qualquer atualização ao método size.

2.3.2.3 Add

O método add foi feito da seguinte forma:

```
public void add(E elem) {
        while(true) {
          //Try to aquire the lock with and atomic compare and set
          if (addElementFlag.compareAndSet(false, true)) {
            rooms.enter(ADD_ROOM);
            int p = tail.getAndIncrement();
            if (p - head.get() < array.length) {
              array[p % array.length] = elem;
```

```
                rooms.leave(ADD_ROOM);
                addElementFlag.set(false);
                break;
              } else {
                E[] newArray = (E[]) new Object[this.array.length * 2];
                for (int i = head.get(); i < p; i++) {
                    newArray[i % newArray.length] =
                      this.array[i % this.array.length];
                }
                newArray[p % newArray.length] = elem;
                this.array = newArray;
                //We must leave the room to give
                //chance to other threads
                //Of emptying the array or
                //we'll be stuck here forever
                rooms.leave(ADD_ROOM);
                addElementFlag.set(false);
                break;
              }
            } else {
              if (useBackoff)
                Backoff.delay();
            }
          }
          if (useBackoff)
              Backoff.reset();
      }
```

Como é possível observar, para podermos entrar na zona crítica, precisamos que a variável addElementFlag seja false, o que significa que de momento não está nenhum thread na zona crítica. O método compareAndSet(v1, v2) compara v1 com o valor atual da variável e caso sejam iguais, substitui-o por v2. Este processo é atómico logo não há necessidade de implementar qualquer tipo de lock para regular o acesso à zona critica. Quando conseguimos acesso à zona crítica, verificamos se existe espaço no array para guardar o elemento caso exista o processo é semelhante à bounded queue com o acrescento de libertar a flag sobre a zona crítica mas no caso de não existir espaço, vamos expandir o array. Copiamos os valores na ordem que se encontram, armazenamos o nosso novo valor, mudamos a referência do array, saímos do quarto de ADD e libertá-mos a flag sobre a zona crítica.

Uso de backoff: O uso de backoff foi alterado ligeiramente pois não é necessário ter backoff quando não há espaço na array pois quando não há espaço simplesmente expandimos a array, criando mais espaço. Contudo, com esta nova implementação, obtemos um novo local de bloqueamento, nomeadamente a tentar obter acesso à área crítica, com o uso do compareAndSet(). Após efetuar a operação com sucesso, resetamos o nosso timer de Backoff, com Backoff.reset().

2.3.2.4 Remove

Como no remove não é feito nenhum resize da array, não foi necessário efetuar qualquer alteração em relação ao método remove da LFQueue.

2.4 Análise de execução linearizável

2.4.1 Análise de execução para a primeira parte:

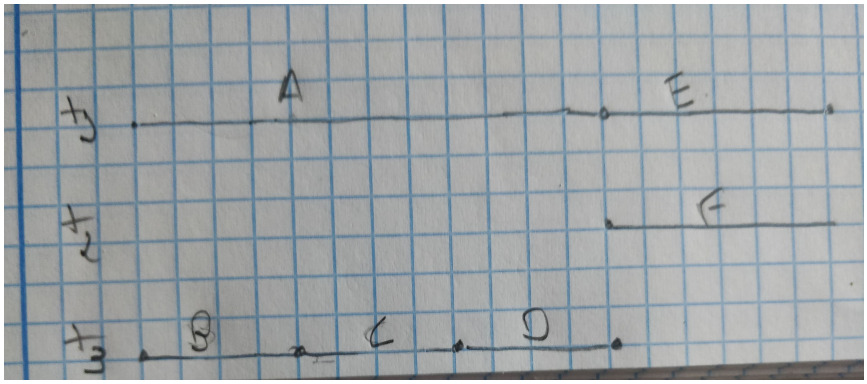

Figure 1: História não linear da primeira parte

Como sabemos que as queues são uma estrutura FIFO (First In First Out), sabemos que o que importa é a ordem em que adicionamos e a ordem em que retiramos portanto podemos assumir que os remove vêm sempre após todos os adds terem ser feitos, pois se viessem entre os add o resultado seria exatamente o mesmo. Também sabemos que um remove tem que ser sempre precedido por um add(), pois o remove() bloqueia caso a queue estiver vazia.

As letras e as suas ações correspondentes são as seguintes:

$A : add(3)$, $B : n = size()$, $C : add(n)$, $D : add(n+1)$, $E : a = remove()$, $F : b = remove()$.

As precedências são as seguintes: $A \rightarrow E$,$A \rightarrow F$, $B \rightarrow C \rightarrow D$, $D \rightarrow F$.

Logo temos as seguintes concorrências:

$A||B$, $A||C$, $A||D$, $E||F$.

Ficamos então com as seguintes possibilidades: No início temos que $q = \emptyset$

- [A][B][C][D] $(q = [3,1,2])$ [E][F] $\rightarrow q = [2], a = 3, b = 1$
- [A][B][C][D] $(q = [3,1,2])$ [F][E] $\rightarrow q = [2], b = 3, a = 1$
- [B][A][C][D] $(q = [3,0,1])$ [E][F] $\rightarrow q = [1], a = 3, b = 0$
- [B][A][C][D] $(q = [3,0,1])$ [F][E] $\rightarrow q = [1], b = 3, a = 0$
- [B][C][A][D] $(q = [0,3,1])$ [E][F] $\rightarrow q = [1], a = 0, b = 3$

- [B][C][A][D] $(q = [0, 3, 1])$ [F][E] $\rightarrow q = [1], b = 0, a = 3$
- [B][C][D][A] $(q = [0, 1, 3])$ [E][F] $\rightarrow q = [3], a = 0, b = 1$
- [B][C][D][A] $(q = [0, 1, 3])$ [F][E] $\rightarrow q = [3], b = 0, a = 1$

2.4.2 Análise de execução para a segunda parte:

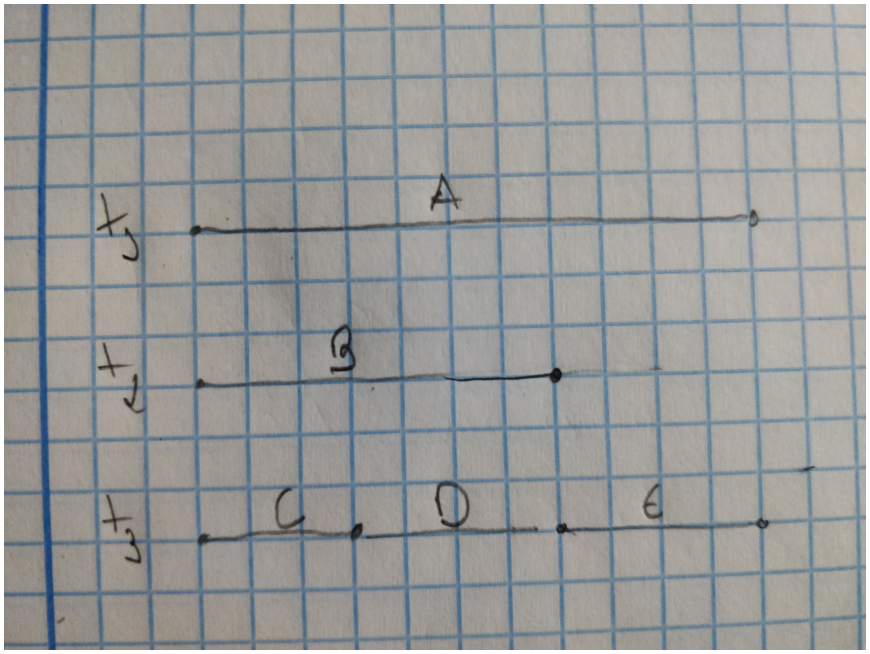

Figure 2: História não linear da segunda parte

As letras e as suas correspondências são:
$A : c = remove()$, $B : add(a)$, $C : n = size()$,$D : add(n)$, $E : d = remove()$.
As precedências são as seguintes: $B \rightarrow E$, $C \rightarrow D \rightarrow E$.
Logo, temos as seguintes concorrências:
$A||B$, $A||C$, $A||D$, $A||E$, $B||C$, $B||D$.
Ficamos então com as seguintes possibilidades:
No início temos que $q = [Q1]$, e a é o valor de a que resultou do teste anterior.

- [A] $(q = \emptyset)$ [B] $(q = [a])$ [C] $(n = 1)$ [D] $(q = [a, 1])$ [E] $\rightarrow c = Q1, d = a, q = [1]$
- [A] $(q = \emptyset)$ [C] $(n = 0)$ [B] $(q = [a])$ [D] $(q = [a, 0])$ [E] $\rightarrow c = Q1, d = a, q = [0]$
- [A] $(q = \emptyset)$ [C] $(n = 0)$ [D] $(q = [0])$ [B] $(q = [0, a])$ [E] $\rightarrow c = Q1, d = 0, q = [a]$

- [B] $(q = [Q1, a])$ [A] $(q = [a])$ [C] $(n = 1)$ [D] $(q = [a, 1]$ [E] $\rightarrow c = Q1, d = a, q = [1]$
- [B] $(q = [Q1, a])$ [C] $(n = 2)$ [A] $(q = [a])$ [D] $(q = [a, 2]$ [E] $\rightarrow c = Q1, d = a, q = [2]$
- [B] $(q = [Q1, a])$ [C] $(n = 2)$ [D] $(q = [Q1, a, 2])$ [A] $(q = [a, 2])$ [E] $\rightarrow c = Q1, d = a, q = [2]$
- [B] $(q = [Q1, a])$ [C] $(n = 2)$ [D] $(q = [Q1, a, 2])$ [E] $(q = [a, 2])$ [A] $\rightarrow c = a, d = Q1, q = [2]$
- [C] $(n = 1)$ [A] $(q = \emptyset)$ [B] $(q = [a])$ [D] $(q = [a, 1])$ [E] $\rightarrow c = Q1, d = a, q = [1]$
- [C] $(n = 1)$ [A] $(q = \emptyset)$ [D] $(q = [1])$ [B] $(q = [1, a])$ [E] $\rightarrow c = Q1, d = 1, q = [a]$
- [C] $(n = 1)$ [B] $(q = [Q1, a])$ [A] $(q = [a])$ [D] $(q = [a, 1])$ [E] $\rightarrow c = Q1, d = a, q = [1]$
- [C] $(n = 1)$ [B] $(q = [Q1, a])$ [D] $(q = [Q1, a, 1])$ [A] $(q = [a, 1])$ [E] $\rightarrow c = Q1, d = a, q = [1]$
- [C] $(n = 1)$ [B] $(q = [Q1, a])$ [D] $(q = [Q1, a, 1]$ [E] $(q = [a, 1]$ [A] $\rightarrow c = a, d = Q1, q = [1]$
- [C] $(n = 1)$ [D] $(q = [Q1, 1])$ [A] $(q = [1])$ [B] $(q = [1, a]$ [E] $\rightarrow c = Q1, d = 1, q = [a]$
- [C] $(n = 1)$ [D] $(q = [Q1, 1])$ [B] $(q = [Q1, 1, a])$ [A] $(q = [1, a])$ [E] $\rightarrow c = Q1, d = 1, q = [a]$
- [C] $(n = 1)$ [D] $(q = [Q1, 1])$ [B] $(q = [Q1, 1, a])$ [E] $(q = [1, a])$ [A] $\rightarrow c = 1, d = Q1, q = [a]$

Ficamos então com as seguintes possibilidades para c e d distintas:

- $c = Q1, d = a$
- $c = Q1, d = 0$
- $c = Q1, d = 1$
- $c = a, d = Q1$
- $c = 1, d = Q1$

Temos as seguintes possibilidades de q e a à entrada e os seus respetivos resultados:

- $q = [2], a = 3 \rightarrow c = 2, d = 3 || c = 2, d = 0 || c = 2, d = 1 || c = 3, d = 2 || c = 1, d = 2$
- $q = [2], a = 1 \rightarrow c = 2, d = 1 || c = 2, d = 0 || c = 2, d = 1 || c = 1, d = 2$
- $q = [1], a = 3 \rightarrow c = 1, d = 3 || c = 1, d = 0 || c = 1, d = 1 || c = 3, d = 1$
- $q = [1], a = 0 \rightarrow c = 1, d = 0 || c = 1, d = 1 || c = 0, d = 1$
- $q = [3], a = 0 \rightarrow c = 3, d = 0 || c = 3, d = 1 || c = 0, d = 3 || c = 1, d = 3$
- $q = [3], a = 1 \rightarrow c = 3, d = 1 || c = 3, d = 0 || c = 1, d = 1$

2.5 Análise de Desempenho

2.5.1 Sistema utilizado nos testes:

OS: Ubuntu 20.04
CPU: 8 Cores 16 Threads a 4.2GHz
RAM: 32GB 3600Mhz

2.5.2 Resultados:

Os resultados aqui apresentados são a médias das 5 runs separadas. Todos os resultados estão em milhares de operações por segundo por thread, sem ser o total que se encontra em operações por segundo.

Para calcular a variância foi utilizada a fórmula: $V = \sqrt{\sum_{i=0}^{n}(X - AVERAGE)^2 * \frac{1}{N}}$.

2.5.2.1 MBQueueU

Os resultados para a MBQueue unbounded podem ser encontrados na Tabela 1.

Threads	Implementação	Média	Variância	Total
2	MBQueueU	2314	162.51	4628
4	MBQueueU	926	60.52	3704
8	MBQueueU	386	17.14	3088
16	MBQueueU	184	8.21	2944
32	MBQueueU	96	9.22	3072

Table 1: Resultados da MBQueueU

Pode-se então observar pelos valores obtidos que a performance deste tipo de implementação tende a piorar significativamente quanto mais threads são colocadas a executar, devido não só ao facto de apenas poder estar 1 thread ao mesmo tempo a executar aquele código mas também porque a JVM tem que atualizar a memória global com as alterações que foram efetuadas.

2.5.2.2 LFBQueueU com Backoff

Os resultados para a LFBQueue Unbounded com backoff **ativo** podem ser encontrados na Tabela 2.

Threads	Implementação	Média	Variância	Total
2	LFBQueueU	8724	724.60	17448
4	LFBQueueU	5713	621.01	22852
8	LFBQueueU	3430	75.75	27440
16	LFBQueueU	1422	40.15	22752
32	LFBQueueU	624	5.93	19968

Table 2: Resultados da LFBQueueU com Backoff ativo

Em geral, como pode ser observado nos valores, esta implementação é bem mais eficiente sendo bastante mais rápida por thread mas também tenha tido um melhor desempenho quando aumentamos as threads até 8. A partir desse momento, alguma performance foi perdida mas continua a ser significativamente superior à implementação baseada em locks. O uso de backoff (Como pode ser visto na Secção 2.5.2.3) aumenta a performance em cerca de 10x, provando que iria ser perdido muito tempo de CPU se estivessem threads constantemente a tentar aceder à zona critica

2.5.2.3 LFBQueueU sem Backoff

Os resultados para a LFBQueue Unbounded sem backoff podem ser encontrados na Tabela 3.

Threads	Implementação	Média	Variância	Total
2	LFBQueueU	1416	102.48	2832
4	LFBQueueU	610	87.63	2440
8	LFBQueueU	284	14.51	2272
16	LFBQueueU	139	7.00	2224
32	LFBQueueU	72.6	2.24	2323

Table 3: Resultados da LFBQueueU sem Backoff

Estes resultados demonstram que o uso de backoff é extremamente importante, pois esta implementação tem performance ainda inferior à implementação que utiliza locks. Contudo, ao contrário do que se observa com a implementação de locks, os resultados mantêm-se mesmo com o aumento do número de threads.

2.5.2.4 STMBQueueU

Os resultados para a STMBQueue Unbounded podem ser na Tabela 4.

Como podemos ver esta implementação foi a mais lenta o que seria de esperar pois partilha semelhanças com a implementação baseada em locks (Apenas é

Threads	Implementação	Média	Variância	Total
2	STMBQueueU	701	160	1402
4	STMBQueueU	164	15.70	656
8	STMBQueueU	47	1.90	376
16	STMBQueueU	13	0.75	208
32	STMBQueueU	5	0.59	160

Table 4: Resultados da STMBQueueU

permitido uma thread de cada vez com acesso à zona crítica e após o final da operação toda a memória tem que ser atualizada. Partilha também semelhanças com a implementação com uso de variáveis atómicas sem backoff pois para atingir o estado de bloqueante o programa tem que entrar num loop infinito até passar na condição de paragem, enquanto que na implementação baseada em locks estas threads ficaram em *sleep* até receberem uma notificação a informar que alguma alteração foi efetuada, poupando assim tempo de CPU e permitindo uma execução mais eficiente (Na implementação LFBQueueU isto é conseguido com o uso de backoff).

2.6 Desafio Extra

2.6.1 LFDeque

O LFDeque foi implementado como uma subclasse de LFBQueueU da seguinte forma:

2.6.1.1 Variáveis Usadas:

Não foram utilizadas quaisquer variáveis adicionais, sem foi efetuada nenhuma alteração em relação às variáveis, utilizando apenas as variáveis da classe pai.

2.6.1.2 AddLast

O método addLast(E) não sofreu nenhuma alteração em relação à implementação do add(E), pois estes métodos fazem o mesmo.

2.6.1.3 AddFirst

O método addFirst(E) foi implementado da seguinte maneira:

```
public void addFirst(E elem) {
    while (true) {
      if (addElementFlag.compareAndSet(false, true)) {
        rooms.enter(ADD_ROOM);
        int p = head.updateAndGet(i -> {
          if (i <= 0) {
              tail.getAndAccumulate(array.length, Integer::sum);
```

```
            return array.length - 1;
          }
          return i - 1;
        });
        if (tail.get() - p < array.length) {
          array[p % array.length] = elem;
          rooms.leave(ADD_ROOM);
          addElementFlag.set(false);
          break;
        } else {
          //Resize array to have space in the back part
          E[] newArray = (E[]) new Object[this.array.length * 2];
          for (int i = p + 1; i <= tail.get(); i++) {
            newArray[i % newArray.length] =
                this.array[i % this.array.length];
          }
          newArray[p % newArray.length] = elem;
          this.array = newArray;
          addElementFlag.set(false);
          rooms.leave(ADD_ROOM);
          break;
        }
      } else {
        if (useBackoff)
          Backoff.delay();
      }
    }
    if (useBackoff)
      Backoff.reset();

  }
```

Nesta implementação podemos ver uma grande diferença na maneira que calculamos o p, pois como não estamos a adicionar no final não modificamos a tail mas sim a head. Contudo quando vamos adicionar algo à head e esta está em 0, haveria um problema pois esta iria ser negativa e iria indicar um índice negativo que é impossível. Para resolver este problema, quando a head chega a 0, deslocamos a head para o final da queue, dando o tamanho todo da queue para preencher contudo ao deslocar a head, também temos que deslocar a tail para esta se manter à distância adequada a representar o tamanho verdadeiro. A tail mantém-se a apontar para o mesmo local pois é deslocada pelo tamanho do array e como é modulada pelo tamanho do array, dá o mesmo resultado.

2.6.1.4 RemoveFirst

O método removeFirst() não foi alterado em relação à implementação do método remove() do LFBQueueU, pois efetuam a mesma coisa.

2.6.1.5 RemoveLast

O método removeLast() foi implementado da seguinte maneira:

```
public E removeLast() {
  E elem = null;
  while (true) {
    rooms.enter(REMOVE_ROOM);
    int p = tail.decrementAndGet();
    if (p >= head.get()) {
      int pos = p % array.length;
      elem = array[pos];
      array[pos] = null;
      rooms.leave(REMOVE_ROOM);
      break;
    } else {
      tail.getAndIncrement();
      rooms.leave(REMOVE_ROOM);
      if (useBackoff)
          Backoff.delay();
    }
  }
  if (useBackoff)
    Backoff.reset();

  return elem;
}
```

Esta implementação é muito semelhante à implementação do removeFirst(), apenas alterando o objecto que é decrementado para a tail.

2.6.1.6 Uso de Backoff

O uso de backoff manteve a mesma estrutura que na LFBQueue pois a lógica dos métodos mantém-se extremamente semelhante, empregando o backoff quando necessitamos de entrar num estado bloqueante à espera de ação por parte de outras threads.

2.6.1.7 Benchmarks - Com Backoff

Podemos encontrar os resultados do LFDeque com o backoff **ativo** na Tabela 5.

Como é possivel observar, os resultados foram bastante semelhantes ao que podemos encontrar na LFBQueueU com backoff, o que é de esperar pois estas duas estruturas são bastante semelhantes.

Threads	Implementação	Média	Variância	Total
2	LFDeque	8953	307.48	17906
4	LFDeque	6197	298.74	24788
8	LFDeque	3786	57.26	30288
16	LFDeque	1571	20.39	25136
32	LFDeque	681	6.71	21792

Table 5: Resultados para o LFDeque com Backoff

2.6.1.8 Benchmarks - Sem Backoff

Podemos encontrar os resultados do LFDeque sem backoff na Tabela 6.

Threads	Implementação	Média	Variância	Total
2	LFDeque	1082	105.33	2164
4	LFDeque	597	100.93	2388
8	LFDeque	249	25.15	1992
16	LFDeque	130	9.52	2080
32	LFDeque	71	1.55	2272

Table 6: Resultados para o LFDeque sem Backoff

Novamente podemos observar que a performance foi bastante semelhante ao que podemos encontrar na LFBQueueU sem backoff, pois as duas estruturas são bastante semelhantes.

2.7 STMDeque

A implementação do STMDeque foi baseada na lógica de acesso ao array do LFDeque mas com a lógica de controlo de acesso às zonas críticas baseado em STM.

2.7.0.1 Variáveis Utilizadas

As variáveis utilizadas foram as seguintes:

```
private final Ref.View<Integer> head;
private final Ref.View<Integer> tail;
private final Ref.View<TArray.View<E>> arrayRef;
```

Temos um conjunto de variáveis bastante semelhante ao que podemos encontrar em STMBQueueU, com a pequena alteração de em vez de armazenar o tamanho, armazenamos sim a posição da tail da queue, como no LFDeque.

2.7.0.2 Size

O size teve que ter ligeiramente alterado pois já não armazenamos o tamanho diretamente. Passou então para o seguinte:

```
public int size() {
  return STM.atomic(() ->
    tail.get() - head.get()
  );
}
```

Temos que calcular o tamanho subtraindo a head à tail. Para isto criamos um bloco atómico para que não possa haver data races nestas variáveis.

2.7.0.3 AddFirst

O método addFirst(E) foi implementado da seguinte forma:

```
public void addFirst(E elem) {
    STM.atomic(() -> {
      int p = this.head.transformAndGet(i -> {
          if (i <= 0) {
            this.tail.transform(t -> t + this.arrayRef.get().length());

            return this.arrayRef.get().length() - 1;
          }

          return i - 1;
        });
        if (tail.get() - p < this.arrayRef.get().length()) {
          this.arrayRef.get().update(p % this.arrayRef.get().length(), elem);
        } else {

          TArray.View<E> newArray = STM.newTArray(this.arrayRef.get().length() *

          for (int i = p + 1; i < tail.get(); i++) {
            newArray.update(i % newArray.length(), this.arrayRef.get().apply(i %
          }

          newArray.update(p % newArray.length(), elem);

          this.arrayRef.set(newArray);
        }
    });
}
```

Como podemos ver, o código é derivado de STMBQueueU e de LFDeque. Mantemos a maneira de criar um novo array e modificar as referências para este novo array da STMBQueueU, contudo mudamos a maneira de calcular a posição da head para suportar a deslocação da head para permitir que elementos sejam adicionados à cabeça mesmo quando esta esteja a apontar para a posição 0.

2.7.0.4 AddLast

O método addLast(E) manteve a sua implementação da classe STMBQueueU, apenas alterando o facto que em STMBQueueU era necessário calcular a posição da tail, com a conta **int** p = head.get() + size.get() contudo nesta implementação isto já não é necessário pois armazenamos a localização da tail diretamente.

2.7.0.5 RemoveFirst

O método removeFirst() manteve a sua implementação da classe STMBQueueU, apenas alterando que como já não armazenamos o size, já não é necessário decrementar este, apenas sendo necessário incrementar a head da queue.

2.7.0.6 RemoveLast

Este método é implementado de uma maneira muito semelhante a removeFirst(), apenas utilizando a tail em vez da head.

2.7.0.7 Benchmarks

Podemos encontrar os resultados do STMDeque na Tabela 7.

Threads	Implementação	Média	Variância	Total
2	STMDeque	371	34.43	742
4	STMDeque	305	153.21	1220
8	STMDeque	65	3.03	520
16	STMDeque	19	0.49	304
32	STMDeque	8.8	0.40	281

Table 7: Resultados para o STMDeque

Como esperado, os resultados desta estrutura são extremamente semelhantes aos resultados da estrutura STMBQueueU pois a lógica utilizada não foi alterada significativamente.

2.7.1 Testes Feitos

Os testes que foram implementados tentavam testar não só o correto funcionamento em situações multithreaded, mas também assegurando o funcionamento lógico da Deque sequencial. Com estes testes foi possivel obter um yield point coverage de 83%.

3 Web Crawler

3.1 Implementação

A implementação do crawler concurrente foi feita da seguinte maneira:

3.1.1 Variáveis utilizadas:

Foram utilizadas as seguintes variáveis:

```
private final Set<String> visited = ConcurrentHashMap.newKeySet();
private final ConcurrentLinkedQueue<Future<Void>> futures =
                    new ConcurrentLinkedQueue<>();
private final AtomicInteger ridCounter = new AtomicInteger(0);
```

Estas variáveis fazem o seguinte: A variável visited guarda todos os visitados e é um Set baseado num ConcurrentHashMap, que garante atomicidade nas suas operações, o futures é uma queue com operações thread-safe que permite acesso seguro às suas operações de várias threads e o ridCounter é um contador atómico para contar a quantidade de ficheiros que foram transferidos.

Estas variáveis também não precisam de ser **static** pois como a classe TransferTask não é estática, todas as instâncias de TransferTask guardam uma referência da ConcurrentCrawler que lhe deu origem, logo todas mantêm uma referência aos mesmos objetos.

3.1.2 Crawl

As alterações feitas crawl foram as seguintes:

```
public void crawl(String root) {
  long t = System.currentTimeMillis();
  log("Starting at %s", root);
  visited.add(root);
  pool.invoke(new TransferTask(ridCounter.getAndIncrement(), root));

  Future<Void> f = null;
  while ((f = futures.poll()) != null) {
    try {
      f.get();
    } catch (Exception e ){
      throw new UnexpectedException(e);
    }
  }
  t = System.currentTimeMillis() - t;
  log("Done %d transfers in %d ms", ridCounter.get(), t);
}
```

O loop que inserimos que executa até a queue futures estar vazia implica que o programa tem que esperar que todas as tarefas terminem antes de poder terminar ele próprio e como vamos ver, a maneira que temos implementado a adição de Futures à queue, esta nunca vai ficar vazia com tarefas ainda por fazer, pois as novas tarefas são colocadas na queue antes de dar por terminada a tarefa e retornar. Podemos também ver que iniciamos a TransferTask com o ridCounter.getAndIncrement() pois este contador pode ser utilizado em todas as threads com segurança.

3.1.3 Compute

O compute foi implementado da seguinte forma:

```
    @Override
    protected Void compute() {
      try {
        URL url = new URL(path);
        List<String> links = performTransfer(rid, url);
          links.forEach((link) -> {
            try {
              String newURL = new URL(url,
                            new URL(url, link).getPath()).toString();
              if (visited.add(newURL)) {
                Future<Void> future = pool.submit(
                    new TransferTask(ridCounter.getAndIncrement(),
                                     newURL));
                futures.add(future);
              }
            } catch (MalformedURLException e) {
              e.printStackTrace();
            }
          });
      } catch (Exception e) {
        throw new UnexpectedException(e);
      }
      return null;
    }
  }
```

Nesta porção de código foram feitas as seguintes modificações para suportar a execução concorrente.

- O uso do método add(E) do Set concurrente que é atómico e retorna true caso adicione com sucesso e false caso falha. Como um Set apenas permite um elemento igual dentro dele, se tentarmos adicionar um elemento que já está contido o add() retorna falso e quando não está contido o add() retorna true e adiciona o elemento atomicamente.

- Utilizamos o método pool.submit(...) para submeter uma nova tarefa à pool para ser executada e adicionamos esta tarefa à lista de tarefas pendentes futures. Podemos ver que esta adição é feita logo quando a tarefa é criada e não quando a tarefa é executada pois se tivessemos utilizado a segunda opção, poderia acontecer alguma situação em que a Queue estaria vazia com tarefas ainda por realizar.

- Incrementamos o contador atómico com ridCounter.getAndIncrement() como também fizemos no método crawl.

3.2 Benchmarks

Todos os tempos apresentados são em millisegundos.

Os resultados foram obtidos utilizando o caso com um grande número de páginas HTML que o professor disponibilizou.

3.2.1 ConcurrentCrawler sem WORK_STEALING_POOL

Os resultados das benchmarks do crawler com a flag WORK_STEALING_POOL **não ativa** podem ser encontrados na Tabela 8.

Threads Cliente	Threads Servidor	Média	Variância
1	1	55189	4748.92
2	1	26129	335.47
2	2	27269	946
4	2	14168	322.85
4	4	14561	597.29
8	4	8797	275.09
8	8	8708	79.20
16	4	5577	184.67
16	8	5465	172.53
16	16	5622	188.28

Table 8: Resultados da execução do WebCrawler Concurrente sem WORK_STEALING_POOL

Como podemos ver o tempo que demora a fazer o download normalmente é relacionado mais proximamente do número de threads de um cliente, pois não temos uma subida notável de performance ao aumentar o número de threads do servidor.

3.2.2 ConcurrentCrawler com WORK_STEALING_POOL ativo no servidor

Os resultados das benchmarks do crawler com a flag WORK_STEALING_POOL **ativa** podem ser encontrados na Tabela 9.

Novamente, a quantidade de threads que foram entregues ao servidor não parece fazer muita diferença. Também não vemos uma grande mudança por utilizar-mos a WorkStealingPool cujo objetivo é quando alguma thread acaba as suas tarefas mais rápido que as outras esta "rouba" tarefas a outras threads que ainda estão a executar e executa-as ela própria, para não desperdiçar tempo de CPU. Esta melhoria não parece estar a surtir grande efeito neste caso pois as tarefas são pequenas e rápidas de executar e todas as threads costumam estar ocupadas a servir pedidos para poderem "roubar" pedidos às outras.

Threads Cliente	Threads Servidor	Média	Variância
1	1	60259	106.18
2	1	26309	191.50
2	2	28384	1769.93
4	2	14267	167.72
4	4	14138	901.57
8	4	8940	226.22
8	8	8602	232.09
16	4	5585	150.54
16	8	5411	229.66
16	16	5475	188.27

Table 9: Resultados da execução do WebCrawler Concurrente sem WORK_STEALING_POOL