# Exemplo de Threads e ExecutorService (com vídeo)

(O vídeo tem narração e explica mais detalhes. Se o texto não for suficiente, assista o vídeo com fones de ouvido)

## Código Serial

A implementação serial do produto escalar se resume a soma dos produtos dos elementos correspondentes de dois vetores (no caso duas instâncias `List<Double>`). Se as listas possuem tamanhos diferentes, uma exceção é lançada. Senão o produto escalar é computado usando um `for`.

```
@Override
public double compute(@Nonnull final List<Double> a, @Nonnull final List<Double> b) {
    if (a.size() != b.size())
        throw new IllegalArgumentException("a.size() != b.size()");

    double sum = 0;
    for (int i = 0; i < a.size(); i++) {
        sum += a.get(i) * b.get(i);
    }
    return sum;
}
```

## Solução com Threads

Na solução com threads, a estratégia é criar várias threads e dividir o trabalho (porções contínuas das duas listas) entre as threads de maneira aproximadamente igual. Teremos `numThreads` instâncias de `Thread` e `numThreads` resultados parciais de cada thread. Cada thread irá computar aproximadamente `chunk` produtos escalares parciais:

```
int chunk = a.size()/numThreads;
Thread workers[] = new Thread[numThreads];
final double sums[] = new double[numThreads];
```

De posse dessas informações podemos criar as threads em um for:

```
for (int i = 0; i < numThreads; i++) {
    workers[i] = new Thread(new Runnable() {
        @Override
        public void run() {
            //...
        }
    });
    workers[i].start();
}
```

Duas coisas importantes sobre a criação de threads acima:

1. Em Java, threads precisam ter o método `start()` chamado;
2. O código usa uma classe anônima que implementa a interface `Runnable`.

Usar classes anônimas tem duas vantagens:

1. É mais rápido de programar e legível;

2. O código da classe anônima pode ler variáveis marcadas como **`final`** no escopo da função que contém a definição da classe anônima. Isso permitirá que a classe anônima escreva em `sums[i]` o resultado parcial do chunk atribuído aquela thread.

A `i`-ésima thread precisa computar o produto escalar entre os índices `i*chunk` e `(i+1)*chunk` (que seria o início da próxima thread). No entando, se `a.size() % chunk != 0`, utilizar essa definição fará com que alguns índices no final das listas não sejam processados por nenhuma thread. Por isso, é necessário que, como um caso especial, **a última thread processe até o final do array**. Para cada thread, o intervalo de índices é computado da seguinte forma:

```
for (int i = 0; i < numThreads; i++) {
    final int begin = i*chunk;
    final int end = i == numThreads-1 ? a.size() : (i+1)*chunk;
    //...
}
```

Uma vez computado o `begin` e o `end` de cada thread, o corpo da thread pode ser escrito. Note que foi necessário criar uma cópia da variável i marcada como `final` para que as threads possam acessar. Essa cópia de `i` é usada para enderecar uma posição dentro do array `sums`.

```
for (int i = 0; i < numThreads; i++) {
    final int begin = i*chunk;
    final int end = i == numThreads-1 ? a.size() : (i+1)*chunk;
    final int idx = i;
    workers[i] = new Thread(new Runnable() {
        @Override
        public void run() {
            double sum = 0;
            for (int j = begin; j < end; j++)
                sum += a.get(j) * b.get(j);
            sums[idx] = sum;
        }
    });
    workers[i].start();
}
```

Criadas todas as threads, basta esperar pelo término de todas e agregar os resultados parciais:

```
double result = 0;
for (int i = 0; i < numThreads; i++) {
    try {
        workers[i].join();
        result += sums[i];
    } catch (InterruptedException ignored) {}
}

return result;
```

# Solução com ExecutorService (08:30)

A solução com ExecutorService possui apenas algumas diferenças com a solução anterior usando Threads:

- É utilizado `ExecutorService.submit()` ao invés de `new Thread(Runnable)` + `Thread.start()`;

- São usadas classes anônimas que implementam `Callable<Double>` ao invés de `Runnable`
- Ao invés de escrever os resultados em um array, cada tarefa executada em paralelo retorna um resultado. Esse resultado é acessado usando `Future.get()`.

A criação do `ExecutorService` é trivial:

```
ExecutorService ex = Executors.newFixedThreadPool(numThreads);
```

No lugar do `double sums[]`, é criada uma lista de `Future<Double>`. Para cada uma das `numThreads` sub-tarefas, um `Future<Double>` é adicionado à essa lista. O `Future` é obtido como retorno de ex.submit(), que por sua vez recebe um `Callable<Double>` como classe anônima que retorna o produto escalar parcial referente ao *chunk* da `i`-ésima thread.

```
ArrayList<Future<Double>> futures = new ArrayList<>();
for (int i = 0; i < numThreads; i++) {
    final int begin = i*chunk;
    final int end = i == numThreads-1 ? a.size() : (i+1)*chunk;
    futures.add(ex.submit(new Callable<Double>() {
        @Override
        public Double call() {
            double sum = 0;
            for (int j = begin; j < end; j++) sum += a.get(j) * b.get(j);
            return sum;
        }
    }));
}
```

Como no caso da implementação com `Threads`, o próximo passo é coletar os resultados parciais e os agregar. O método `Future.get()` pode lançar dois tipos de exceção:

1. `InterruptedException`: o get() coloca a thread main em espera até que o `Callable<Double>` que deu origem ao future termine sua execução em alguma thread gerenciada pelo ExecutorService e retorne um resultado. Durante esse momento outra thread poderia interromper a main thread usando o método `Thread.interrupt()`. Se isso acontecesse, a thread cessaria a espera e retornaria do get() imediatamente através dessa exceção.
2. `ExecutionException`: Uma implementação de `Callable<Double>` pode lançar qualquer tipo de exceção no método `call` (o que não é o caso desse exemplo). Se o método call lançasse alguma exceção, ela seria relançada na main thread durante o get(). A ExecutionException conteria a exceção original lançada de dentro do call() como sua causa (`Throwable.getCause()`)

```
double result = 0;
for (Future<Double> future : futures) {
    try {
        result += future.get();
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
```

Ao final desse `for`, basta retornar o resultado na variável `result`.

O código completo com a solução está disponível no moodle.