

Parallel Digital Image Processing and Analysis - SISMD 2023/2024

- Bruno Santos - 1230170
- David Magalhães - 1201237
- Pedro Pacheco - 1181034
- Vera Pinto - 1180730

Implementação

Brightness

1. Sequential -

O método BrighterFilter aumenta o brilho de cada pixel da imagem. Este algoritmo percorre cada pixel da imagem, aumenta os valores de vermelho (red), verde (green) e azul (blue) por um valor definido, mas sem exceder o máximo de 255 para cada cor. Se o aumento proposto para qualquer cor ultrapassar 255, essa cor é definida como 255. Depois de ajustar as cores, o pixel é atualizado na imagem temporária.

2. Multi-threaded -

O método BrighterFilterMultiThread aplica um filtro para tornar uma imagem mais brilhante utilizando múltiplas threads para processar diferentes partes da imagem simultaneamente, melhorando a eficiência do processamento. A imagem é dividida horizontalmente em partes iguais, com cada thread responsável por aumentar o brilho de uma parte específica. As threads são iniciadas para processar suas respectivas seções e, após todas as threads terem concluído o processamento (usando join para sincronização).

3. Thread-Pool

1. Executor -

O método BrighterFilterThreadPool aplica um filtro para aumentar o brilho de uma imagem usando uma pool de threads para processamento paralelo, o que melhora a eficiência, especialmente em imagens grandes. Este método divide a imagem em faixas horizontais, onde cada thread em um pool fixo processa uma faixa, aumentando o brilho de cada pixel na faixa designada.

A divisão é feita de modo que todas as threads tenham aproximadamente a mesma quantidade de trabalho, distribuindo as linhas restantes pelas primeiras threads caso a divisão não seja exata. Após submeter todas as tarefas ao executor, o método espera que todas as threads terminem usando *awaitTermination*.

2. Fork Join Pool

Esse algoritmo executa de maneira recursiva.

É criado um ForkJoinPool com um número específico de threads para processar a imagem de forma paralela.

O método invoke é utilizado para executar uma tarefa ForkJoinPool que processa o filtro de brilho em toda a imagem.

A classe BrightnessFilterForkJoinPoolTask implementa a interface RecursiveAction e é responsável por realizar o processamento do filtro de brilho em uma parte da imagem.

Ela divide o trabalho em duas tarefas menores até que esse tamanho seja uma linha da imagem.

3. Completable Futures

Nessa solução o trabalho é dividido em seções (chunks) verticais, cada uma atribuída a uma thread separada.

Cada thread processa suas seções de forma independente, aplicando o filtro de brilho aos pixels da imagem.

O uso de CompletableFuture permite que as tarefas sejam executadas em paralelo, melhorando o desempenho do processo.

O método `allOfFuture.get()` é chamado para aguardar o fim da execução de todas as threads.

Grayscale

1. Sequential -

O filtro de escala de cinza, implementado no método GrayScaleFilter, converte uma imagem colorida em tons de cinza. Isso é feito calculando a média dos valores de vermelho, verde e azul para cada pixel e, em seguida, definindo os componentes de cor do pixel para esse valor médio.

2. Multi-threaded -

Determina-se o número de linhas que cada thread deveria processar. Isso foi feito dividindo a altura da imagem pelo número de threads e armazena-se o resultado na variável `rowsPerThread`. As linhas restantes foram distribuídas entre as threads.

Em seguida, cria-se um array de threads e inicia-se cada thread para processar uma seção específica da imagem. Cada thread foi criada com uma instância da classe `GrayFilterThread`, que foi passada a imagem original, as linhas de início e fim que a thread deveria processar. Depois que todas as threads foram iniciadas, utiliza-se o método `join` para esperar que todas as threads terminassem de processar as suas respectivas seções da imagem.

No fim, escreve-se a imagem processada em um arquivo usando a função `Utils.writeValueImage`.

3. Thread-Pool

1. Executor -

Determina-se o número de linhas que cada tarefa deveria processar. Isso foi feito dividindo a altura da imagem pelo número de threads e armazenando o resultado na variável `rowsPerTask`. As linhas restantes foram distribuídas entre as tarefas.

Em seguida, cria-se um `ExecutorService` com um número fixo de threads.

Depois, para cada thread, submete-se uma nova tarefa ao `ExecutorService`. Cada tarefa foi criada com uma instância da classe `GrayFilterTask`, que foi passada a imagem original, as linhas de início e fim que a tarefa deveria processar.

Na classe `GrayFilterTask`, o método `run()` aplica o filtro de escala de cinza a cada pixel da imagem à seção de linhas que a tarefa deveria processar.

Após todas as tarefas terem sido submetidas, desliga-se o `ExecutorService` e aguarda-se a conclusão de todas as tarefas.

No fim, escreve-se a imagem processada em um arquivo usando a função `Utils.writeValueImage`.

2. Fork Join Pool -

Cria-se uma nova instância de `ForkJoinPool` com um número específico de threads.

Em seguida, você cria-se uma nova tarefa `GrayFilterForkJoinPoolTask`, que foi passada a imagem original, a imagem de destino, e as linhas de início e fim que a tarefa deveria processar.

Como a classe GrayFilterForkJoinPoolTask estende RecursiveAction, a tarefa é dividida em sub-tarefas menores até que o tamanho da tarefa seja menor que um valor específico.

Depois, submete-se a tarefa ao ForkJoinPool usando o método invoke.

Finalmente, escreve-se a imagem processada em um arquivo usando a função Utils.writeImage.

3. Completable Futures -

Cria-se um ExecutorService com um número fixo de threads.

Em seguida, para cada thread, submete-se uma nova tarefa ao ExecutorService. Cada tarefa foi criada com uma instância da classe GrayCompletableFutureTask, que foi passada a imagem original, as linhas de início e fim que a tarefa deveria processar.

Na classe GrayCompletableFutureTask, o método call() aplica o filtro cinza à seção da imagem que foi atribuída à tarefa e retorna um CompletableFuture contendo a imagem processada.

Depois que todas as tarefas foram submetidas, aguarda-se a conclusão de todas as tarefas usando o método get de cada CompletableFuture.

Finalmente, escreve-se a imagem processada em um arquivo usando a função Utils.writeImage.

Swirl

1. Sequential

O código começa determinando as dimensões da imagem, ou seja, sua altura e largura.

Em seguida, são calculadas as coordenadas do centro da imagem, que serão usadas como base para o filtro swirl.

Além disso, é definido o ângulo máximo de rotação, representado por "maxAngle" que será aplicado com base na distância do pixel ao centro da imagem.

Em seguida, é criada uma matriz para armazenar a imagem filtrada, com as mesmas dimensões da imagem original.

O código itera sobre cada pixel da imagem. Para cada pixel, calcula-se a distância entre o pixel e o centro da imagem usando a fórmula da distância.

Com base nessa distância, calcula-se o ângulo de rotação multiplicando a distância pelo ângulo máximo

Usando o ângulo calculado, são calculadas as novas coordenadas do pixel após a aplicação do filtro swirl.

Isso é feito usando as fórmulas de transformação mencionadas na documentação do projeto.

2. Multi-threaded

O código cria várias threads para processar a imagem em paralelo.

Cada thread é responsável por uma parte da imagem, dividida igualmente com base no número de threads especificado.

Caso não seja possível dividir igualitariamente, a última thread assume a porção excedente da imagem.

Para cada parte da imagem atribuída a uma thread, o código itera sobre cada pixel e aplica a transformação descrita anteriormente.

3. Thread-Pool

1. Executor

Muito semelhante aos anteriores onde cada thread é responsável por uma parte específica da imagem, e o filtro de distorção é aplicado.

O que diferencia essa implementação é que o código utiliza um ExecutorService para gerenciar o pool de threads e aguarda a conclusão do processamento antes de continuar.

Nesse caso, não sendo necessário criar e dar start nas threads e nem fazer o join do trabalho.

2. Fork Join Pool

Nesse caso, também existe divisão das tarefas.

Cada thread é responsável por processar uma parte específica da imagem.

A classe `SwirlFilterForkJoinPoolTask` representa uma tarefa recursiva que aplica o filtro de distorção a uma parte da imagem.

Essa parte foi definida como sendo a linha de pixels da imagem.

Enquanto for possível dividir a imagem, novos empilhamentos de execução são criados e invocados recursivamente.

3. Completable Futures

Esse algoritmo por sua vez executa de forma assíncrona usando CompletableFuture.

Ele começa definindo as dimensões da imagem e os parâmetros do filtro como todos os outros.

Para cada parte da imagem atribuída a uma tarefa, é criado um CompletableFuture que executa de maneira assíncrona para processar cada pixel (a mesma divisão aqui foi explicada no tópico de Multithreads).

Após a criação de todas as tarefas, a conclusão de todas elas é esperada através do método allOf().

Uma vez que todas as tarefas estejam concluídas, a imagem filtrada é escrita em um arquivo.

Glass

1. Sequential -

Cria-se uma cópia da imagem original para armazenar a imagem processada.

Em seguida, percorre-se cada pixel da imagem original. Para cada pixel, calcula-se um deslocamento aleatório dentro de um raio especificado (5 pixels).

Substitui-se o pixel original pelo pixel deslocado na imagem copiada.

Finalmente, escreve-se a imagem processada num arquivo.

2. Multi-threaded -

Cria-se uma cópia da imagem original para armazenar a imagem processada.

Em seguida, divide-se a imagem em várias seções, cada uma processada por uma thread separada.

Para cada thread, percorre cada pixel na seção atribuída a essa thread. Para cada pixel, calcula um deslocamento aleatório dentro de um raio especificado (5 pixels).

Substitui-se o pixel original pelo pixel deslocado na imagem copiada.

Finalmente, espera-se que todas as threads terminem e escreve a imagem processada num arquivo.

3. Thread-Pool

1. Executor -

Cria um ExecutorService com um número fixo de threads igual ao número de núcleos de

CPU disponíveis.

Divide a imagem em várias seções, cada uma processada por uma tarefa separada.

Cada tarefa foi responsável por aplicar o filtro de vidro a uma seção específica da imagem.

Submete todas as tarefas ao ExecutorService. Cada tarefa foi executada por uma thread do pool de threads.

Usa o método shutdown do ExecutorService para iniciar um desligamento ordenado, no qual as tarefas previamente submetidas são executadas, mas não são aceites novas tarefas. Em seguida, chama o método awaitTermination para bloquear até que todas as tarefas tenham concluído a execução após um pedido de desligar.

Escreve a imagem processada em um arquivo.

2. Fork Join Pool -

Cria um ForkJoinPool com um número fixo de threads igual ao número de núcleos de CPU disponíveis.

Cria uma tarefa GlassFilterForkJoinPoolTask que processa toda a imagem. Se a imagem era grande demais para ser processada eficientemente por uma única tarefa, divide a tarefa em duas tarefas menores para processaram metade da imagem cada uma.

Submete a tarefa ao ForkJoinPool. A tarefa foi executada por uma thread do pool de threads.

Usa o método invoke do ForkJoinPool para iniciar a execução da tarefa e esperar até que ela fosse concluída.

Por fim, escreve a imagem processada em um arquivo.

3. Completable Futures -

Cria um ExecutorService com um número fixo de threads igual ao número de núcleos de CPU disponíveis.

Divide a imagem em várias seções, cada uma processada por uma tarefa separada.

Cada tarefa foi responsável por aplicar o filtro de vidro a uma seção específica da imagem.

Submete todas as tarefas ao ExecutorService. Cada tarefa foi executada por uma thread do pool de threads.

Usa o método allOf do CompletableFuture para criar um CompletableFuture que é concluído quando todas as tarefas são concluídas. Em seguida, o método join é chamado para bloquear até que todas as tarefas tenham concluído a execução.

Por fim, escreve a imagem processada em um arquivo.

Blur

1. Sequential -

Para cada pixel da imagem, ele calcula uma média dos valores de vermelho, verde e azul dos pixels vizinhos, incluindo o próprio pixel, numa submatriz de tamanho definido por *matrixSize*. Este tamanho determina quanto longe de cada pixel a média deve considerar, criando um efeito de desfoque ao "misturar" os valores de cores dos pixels adjacentes.

O método percorre todos os pixels da imagem, utilizando um deslocamento (offset) para definir a área da submatriz centrada em cada pixel. Os novos valores médios de cor calculados substituem os originais, resultando numa imagem desfocada.

2. Multi-threaded -

O método BlurFilterMultiThread aplica um filtro numa imagem utilizando múltiplas *threads* para processar de forma paralela e aumentar a eficiência. Cada thread é responsável por desfocar uma parte específica. A imagem é dividida em segmentos horizontais, e cada segmento é processado por uma *thread* diferente. O número de linhas que cada thread processa é calculado para distribuir as linhas da imagem de maneira igualitária entre as *threads*. Após iniciar todas as *threads*, o método espera que todas terminem sua execução usando o método *join()*.

3. Thread-Pool

1. Executor -

O método BlurFilterThreadPool usa um pool de threads para aplicar um filtro de desfoque a uma imagem, melhorando a eficiência para imagens grandes ou para sistemas com múltiplos processadores. A imagem é dividida em várias partes, cada uma sendo processada em paralelo por diferentes threads. A quantidade de tarefas é determinada pelo tamanho da imagem e pelo número de threads, ajustando para que cada tarefa tenha um trabalho significativo mas não muito pequeno (mínimo de 10000 pixels por tarefa).

Cada tarefa é responsável por aplicar o desfoque em uma faixa horizontal específica da imagem, calculada com base na altura total da imagem e no número de tarefas. Depois de iniciar todas as tarefas, o método espera que todas terminem usando *awaitTermination*.

2. Fork Join Pool -

O método BlurFilterForkJoinPool utiliza a estrutura ForkJoinPool para aplicar um filtro de desfoque a uma imagem de maneira eficiente e paralela. A imagem é processada dividindo-a em sub-regiões menores, que são então atribuídas a diferentes threads gerenciadas pelo ForkJoinPool. Cada thread trabalha em uma seção da imagem, aplicando o filtro de desfoque, e esse processo é feito de maneira recursiva até que as seções atinjam um tamanho de limite (THRESHOLD), momento em que o filtro é aplicado diretamente.

A classe BlurFilterForkJoinPoolTask, que estende RecursiveAction, é responsável por essa divisão e pelo processamento do filtro. Se a área a ser processada é pequena o suficiente (menor que o THRESHOLD), o filtro é aplicado diretamente. Caso contrário, a tarefa é dividida em quatro sub-tarefas, processando cada quadrante da área de forma recursiva.

3. Completable Futures -

O método BlurFilterCompletableFuture utiliza o CompletableFuture para aplicar um filtro de desfoque (blur) em uma imagem de forma assíncrona e paralela. A imagem é dividida verticalmente em segmentos com base no número de threads especificado. Cada CompletableFuture é responsável por desfocar uma coluna específica da imagem, desde o início até o fim da altura, utilizando uma largura de segmento definida.

Este processo é realizado de forma assíncrona, permitindo que várias partes da imagem sejam processadas simultaneamente. Após iniciar todas as tarefas assíncronas, o método aguarda a conclusão de todas elas com *CompletableFuture.allOf(futures).join()*, garantindo que todas as partes tenham sido processadas antes de prosseguir.

Conditional Blur

1. Sequential

São percorridos todos os pixels da imagem, verificando se o pixel tem uma certa quantidade de vermelho (50) e aplicando o blur quando esta condição se verifica.

2. Multithreaded

A imagem é dividida em várias partes, cada uma processada por uma thread separada. Espera-se que todas as threads terminem usando uma countdown latch.

3. Thread-Pool

1. Executor

Semelhante ao multithreaded, mas usando um executor service para gerir as threads e o método awaitTermination para esperar que todas as threads terminem

2. Fork Join Pool

A imagem é dividida recursivamente em 4 quadrantes, cada um processado por uma thread separada. O processamento ocorre quando a parte restante da imagem é pequena o suficiente (50000 pixels).

3. CompletableFuture

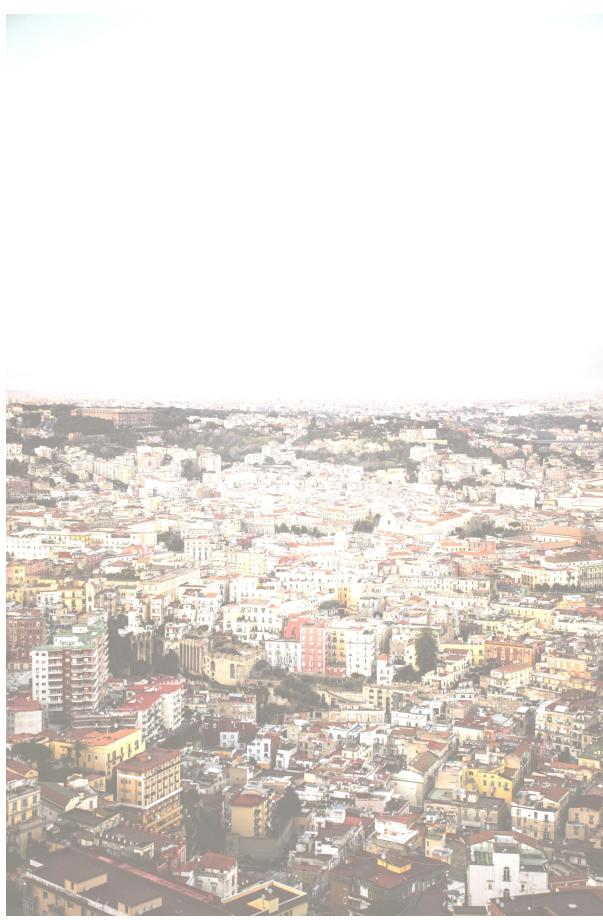
A imagem é dividida por uma quantidade de colunas dependente da quantidade de threads. Para cada conjunto é criado um CompletableFuture para o processar. O método CompletableFuture.allOf é usado para combinar todos os CompletableFutures num só e o método join() é usado para esperar que este termine.

Resultados

City.jpg

Brightness

O processador utilizado para testar esta implementação é um Apple M1 Pro 10-core.



1. Sequential

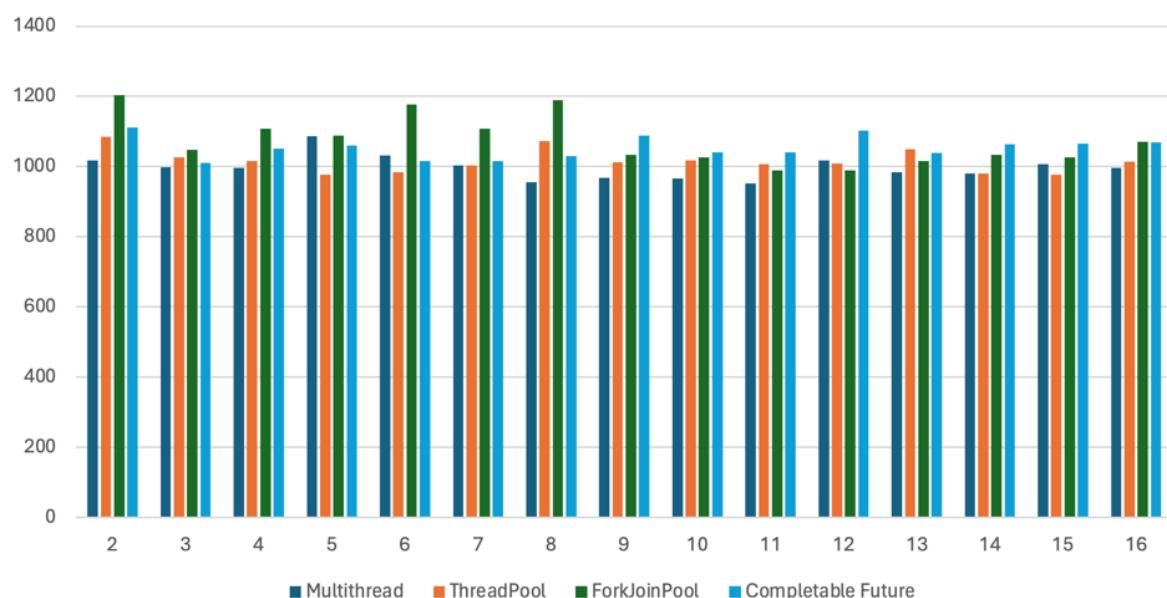
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
City	1147	1169	1150	1155

2. Multithreaded and Thread-Pool

Todos os tempos são resultantes de uma média de três execuções.

Número de thread	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	1017	998	996	1086	1032	1004	955	968	966	952	1018	984	980	1007	996
ThreadPool	1085	1026	1016	977	984	1003	1073	1013	1018	1007	1009	1049	980	977	1014
ForkJoinPool	1204	1048	1107	1089	1177	1107	1189	1033	1026	989	989	1016	1033	1026	1070
CompletableFuture	1111	1011	1051	1060	1016	1016	1029	1089	1040	1041	1103	1038	1064	1066	1069

City.jpg - Bright Filter



Grayscale



1. Sequencial -

Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
City	1955	1881	1982	1939,33

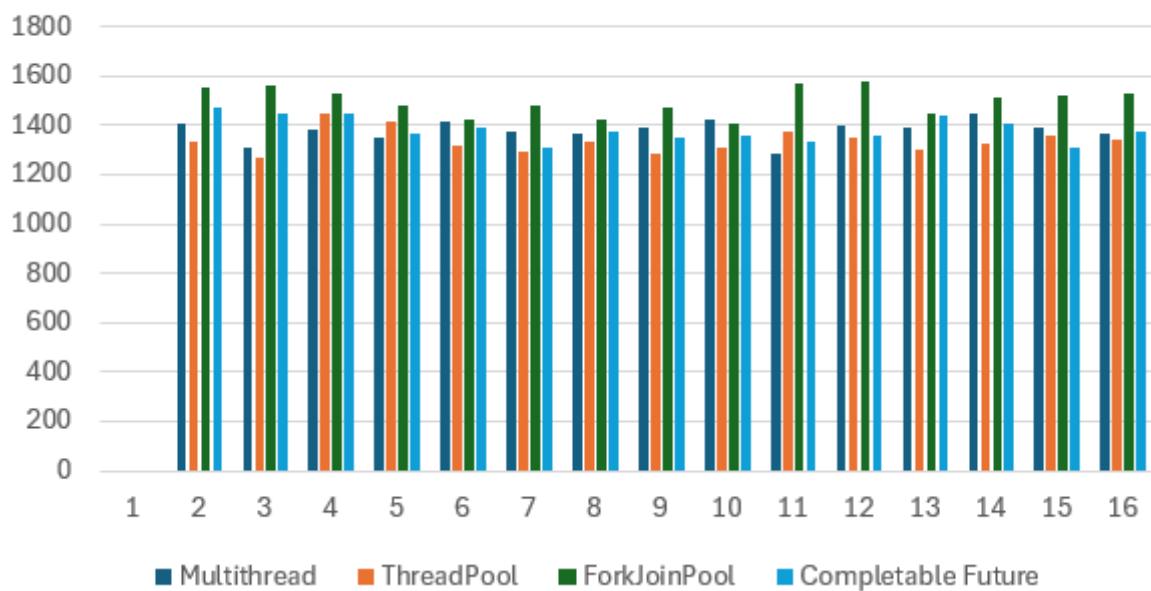
2. Multithreaded and Thread-Pool -

O processador utilizado para testar esta implementação é AMD Ryzen 7 5800H with Radeon Graphics com 8 cores e 16 threads.

Todos os tempos são resultantes de uma média de três execuções.

Numero de thread	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	1403	1308,33	1383	1347	1411	1377,66	1364,33	1391,33	1425,33	1284,66	1397,66	1394	1451	1388	1365,66
ThreadPool	1332	1270,33	1450,66	1418,33	1314,66	1289	1335,66	1287	1311,66	1375,66	1351,33	1300,66	1328	1359,66	1338
ForkJoinPool	1552	1564	1524,33	1479,33	1419	1482	1424,66	1473,33	1407,66	1566,33	1574	1443,33	1512	1517,33	1527
Completable Future	1474,33	1447	1448,33	1363,66	1393	1309,33	1373,66	1348	1354,33	1331,66	1361	1442,33	1405,66	1307,66	1373

Título do Gráfico



Swirl

O processador utilizado para testar esta implementação é um Apple M1 Pro 10-core.



1. Sequential

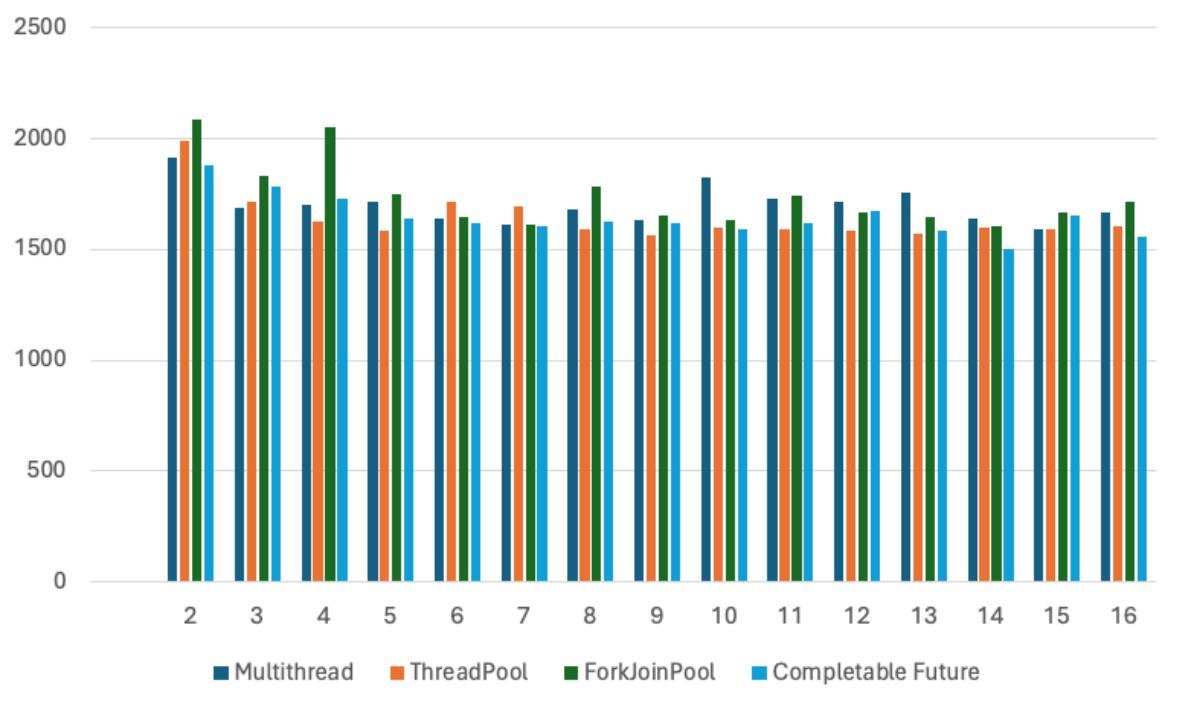
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
City	2269	2278	2328	2291

2. Multithreaded and Thread-Pool

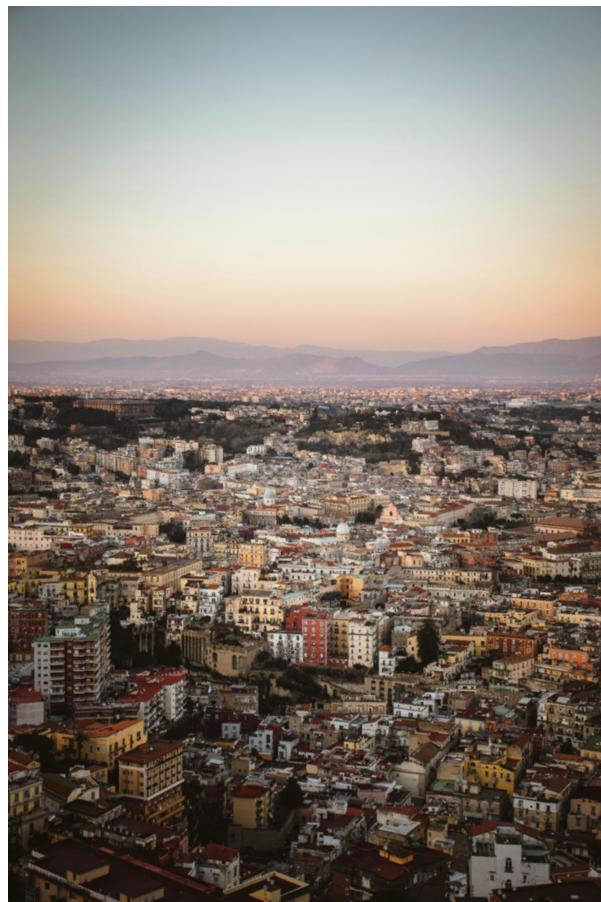
Todos os tempos são resultantes de uma média de três execuções.

Numero de thread	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	1911	1685	1701	1714	1637	1614	1680	1635	1822	1730	1717	1754	1636	1592	1669
ThreadPool	1990	1712	1623	1584	1712	1693	1594	1561	1600	1592	1585	1570	1600	1592	1606
ForkJoinPool	2082	1828	2051	1747	1644	1608	1783	1652	1632	1740	1667	1648	1602	1667	1715
Completable Future	1878	1783	1726	1636	1617	1601	1627	1616	1594	1618	1672	1582	1499	1653	1559

Swirl Filter - city.jpg



Glass



1. Sequencial -

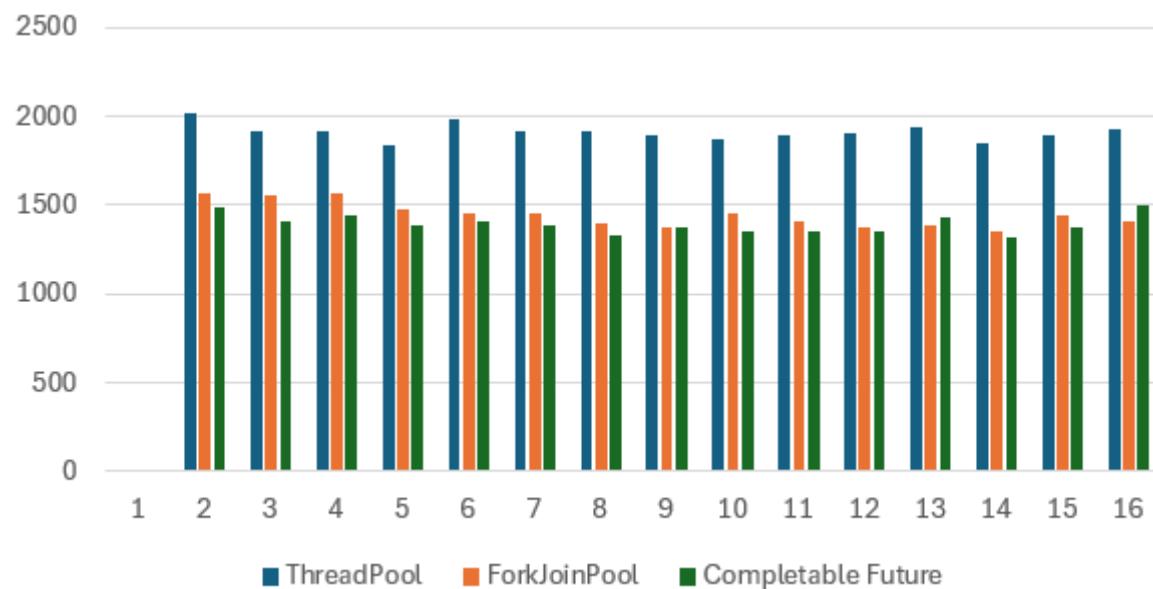
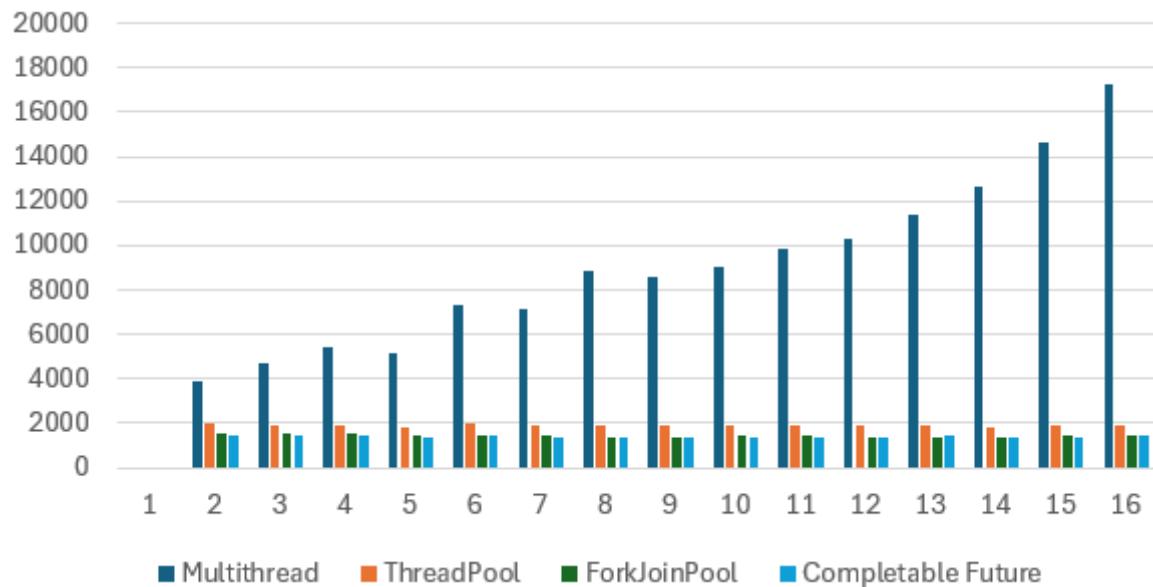
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
City	1955	1881	1982	1939,33

2. Multithreaded and Thread-Pool -

O processador utilizado para testar esta implementação é AMD Ryzen 7 5800H with Radeon Graphics com 8 cores e 16 threads.

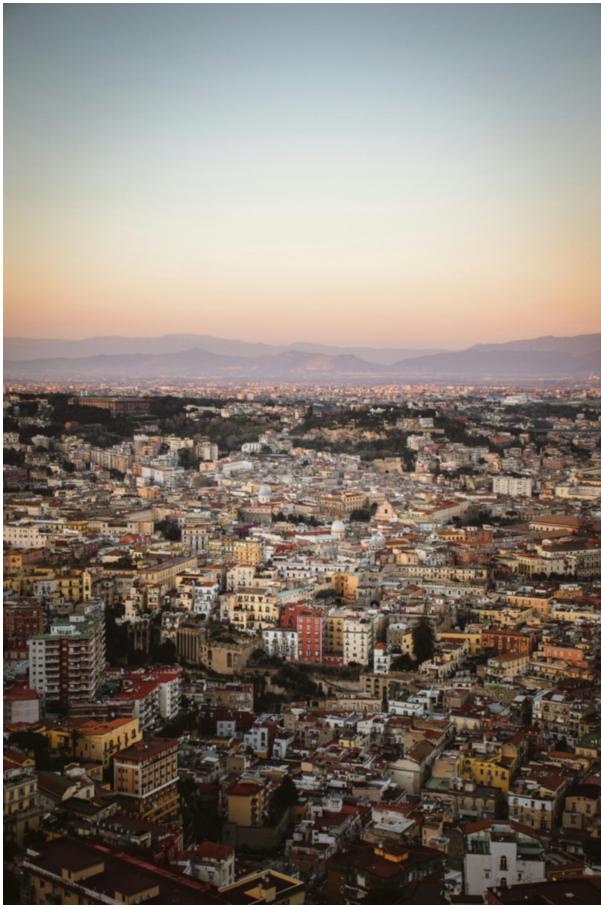
Todos os tempos são resultantes de uma média de três execuções.

Numero de thread	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	3863	4693,33	5412,66	5189	7307,66	7125,33	8891	8569,33	9077	9819,66	10261,33	11367	12611	14678,33	17259
ThreadPool	2022,66	1919	1915,66	1833	1979,33	1916	1921,33	1891,33	1867,66	1890,33	1905	1942,33	1851,66	1897,66	1930,66
ForkJoinPool	1571,66	1550,66	1562	1475,66	1459,33	1453,33	1399,33	1372,33	1454,66	1410,66	1372,33	1383,33	1357,33	1443	1409
CompletableFuture	1493	1411,66	1438,33	1385,33	1411	1391	1328,33	1373	1348,33	1355,33	1355,33	1431,33	1322,33	1374,66	1495,66



Blur

O processador utilizado para testar esta implementação é um 10th Generation Intel® Core™ Core i7-10750H Processor com 6 cores e 12 threads.



1. Sequential

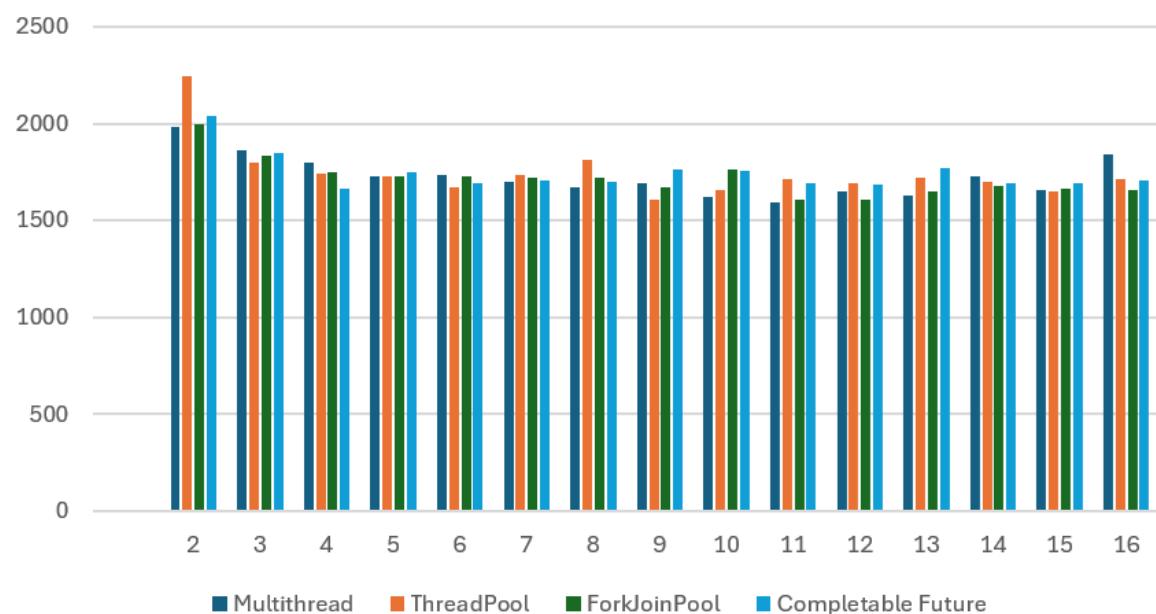
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
City	2697	2705	2750	2717

2. Multithreaded and Thread-Pool

Todos os tempos são resultantes de uma média de três execuções.

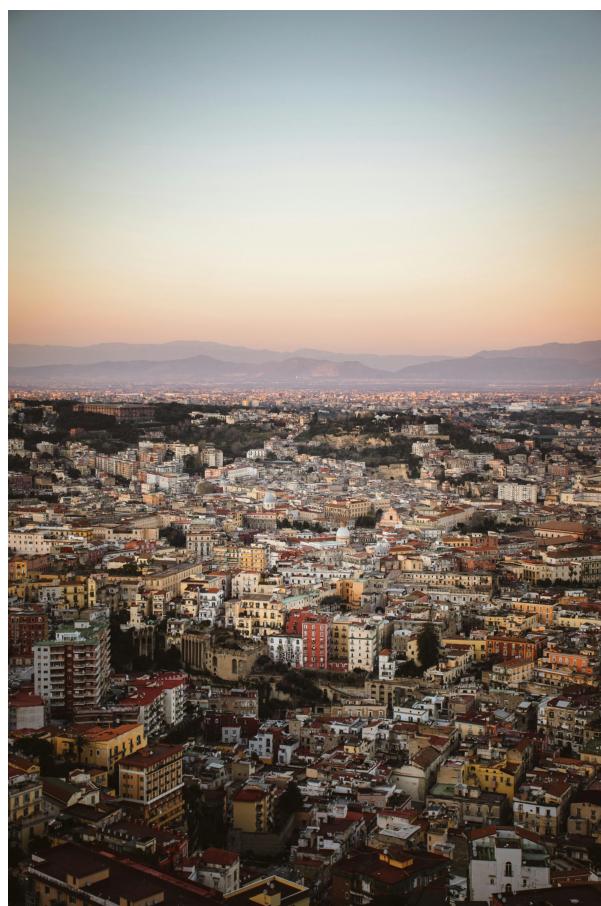
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	1980	1863	1799	1728	1738	1699	1673	1694	1624	1593	1647	1628	1729	1655	1841
ThreadPool	2244	1796	1745	1726	1671	1735	1811	1605	1656	1714	1694	1723	1698	1649	1714
ForkJoinPool	1999	1834	1749	1730	1726	1722	1718	1672	1763	1607	1606	1652	1678	1661	1658
Completable Future	2042	1849	1661	1747	1692	1708	1698	1764	1754	1693	1682	1768	1693	1694	1704

Blur Filter - city.jpg



Conditional Blur

Processor: AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx, 2.30 GHz.



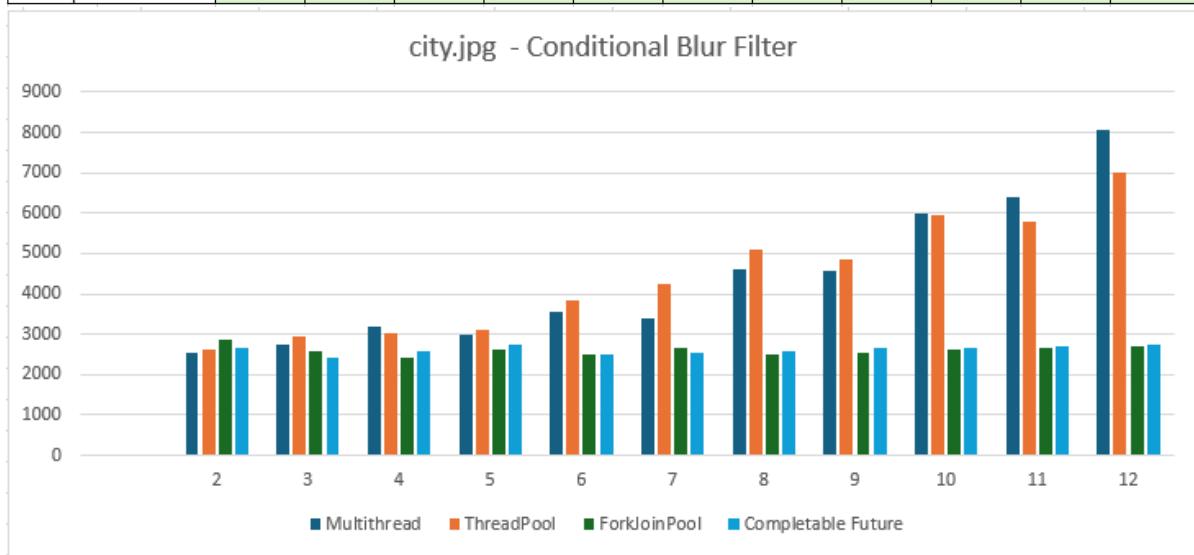
1. Sequential

Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
City	3606	2986	2947	3179

2. Multithreaded and Thread-Pool

Todos os tempos são resultantes de uma média de três execuções.

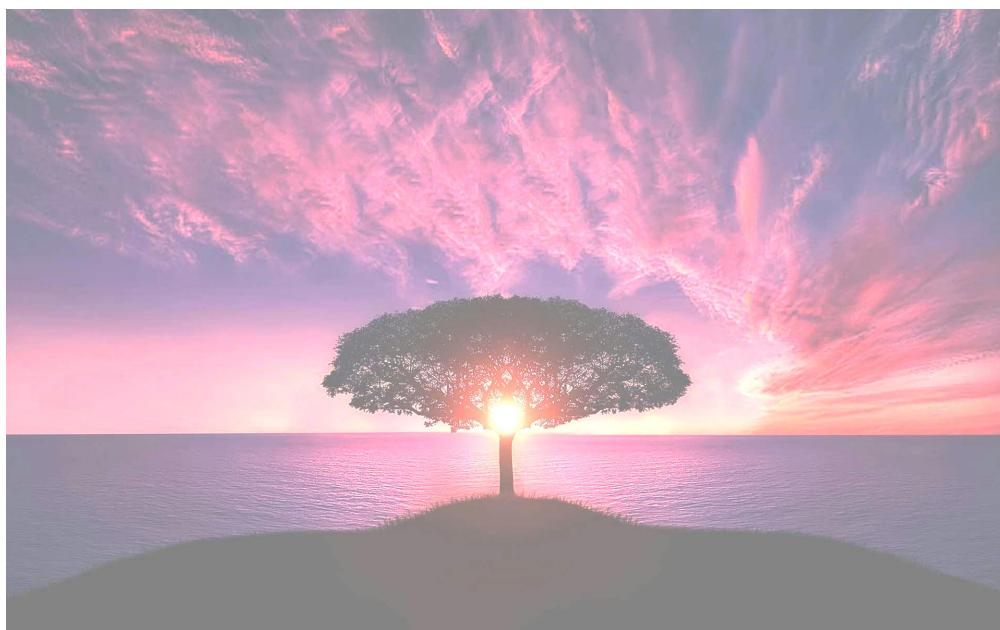
Número de thread	2	3	4	5	6	7	8	9	10	11	12
Multithread	2524	2760	3195	2972	3547	3414	4631	4584	5988	6393	8059
ThreadPool	2628	2932	3033	3092	3834	4248	5083	4860	5933	5781	7000
ForkJoinPool	2851	2578	2401	2616	2493	2645	2501	2556	2640	2675	2717
Completable Future	2679	2433	2576	2731	2502	2547	2564	2646	2683	2696	2745



Tree.jpg

Brightness

O processador utilizado para testar esta implementação é um Apple M1 Pro 10-core.



1. Sequential

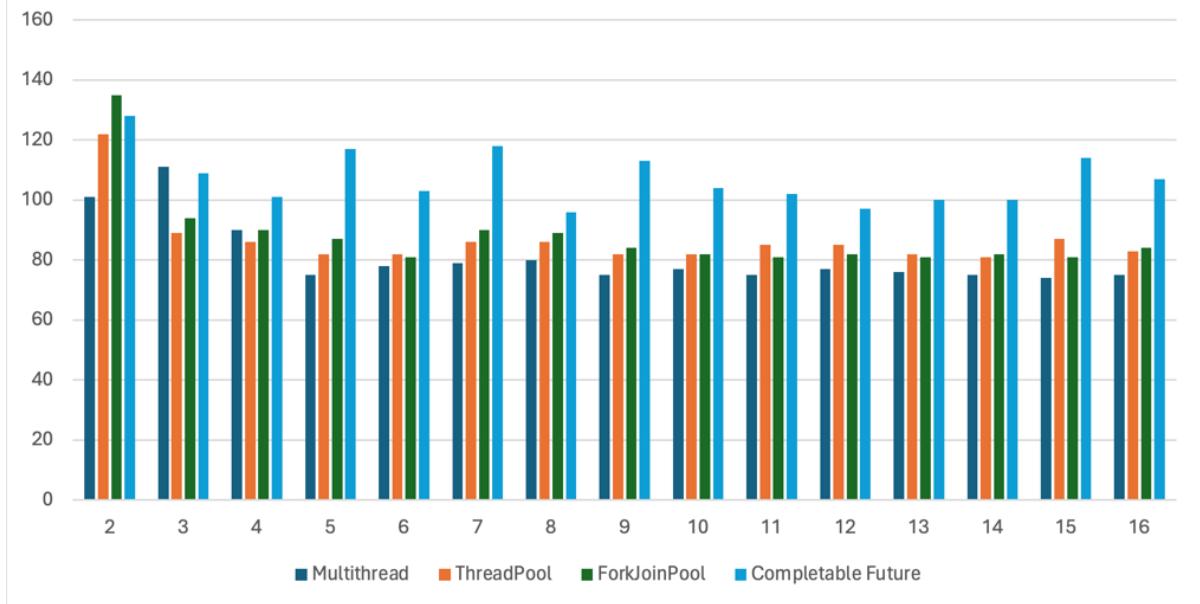
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
Tree	187	193	190	190

2. Multithreaded and Thread-Pool

Todos os tempos são resultantes de uma média de três execuções.

Número de thread	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	101	111	90	75	78	79	80	75	77	75	77	76	75	74	75
ThreadPool	122	89	86	82	82	86	86	82	82	85	85	82	81	87	83
ForkJoinPool	135	94	90	87	81	90	89	84	82	81	82	81	82	81	84
Completable Future	128	109	101	117	103	118	96	113	104	102	97	100	100	114	107

Tree.jpg - Bright Filter



Grayscale



1. Sequencial -

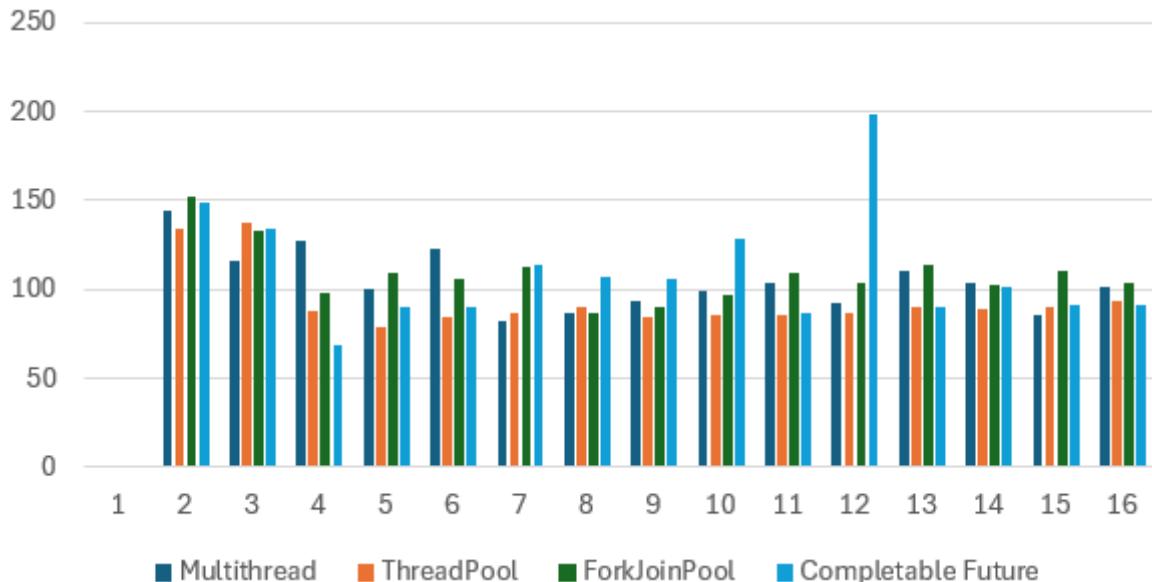
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
Tree	185	184	223	197,33

2. Multithreaded and Thread-Pool -

O processador utilizado para testar esta implementação é AMD Ryzen 7 5800H with Radeon Graphics com 8 cores e 16 threads.

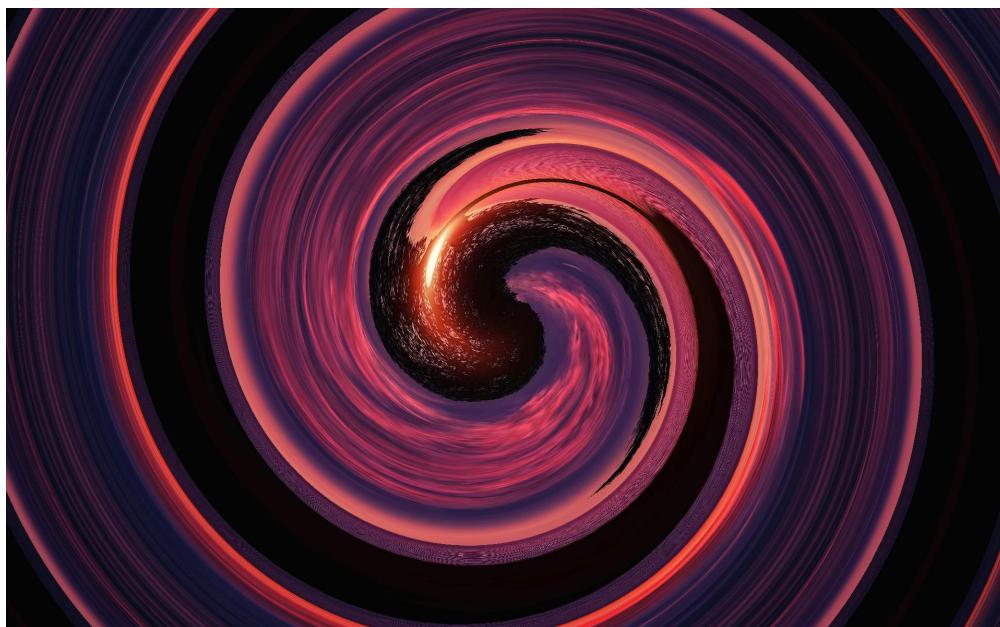
Todos os tempos são resultantes de uma média de três execuções.

Numero de thread	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	144	116,33	127,33	100	123,33	82,66	86,66	93,66	98,66	103,66	92,66	110,33	103,33	85,33	101,66
ThreadPool	134,66	137	87,33	78,66	85	86,66	89,66	84,66	85,33	85,66	86,66	90,66	88,66	90,66	93
ForkJoinPool	151,66	133	98	109	105,66	113	86,33	90,33	97,33	109	103,66	114,33	102,33	110,66	103,66
CompletableFuture	149,33	134	68,66	90	90	114,33	106,66	106	129	86,66	198,33	90,66	101,33	91	91,33



Swirl

O processador utilizado para testar esta implementação é um Apple M1 Pro 10-core.



1. Sequential

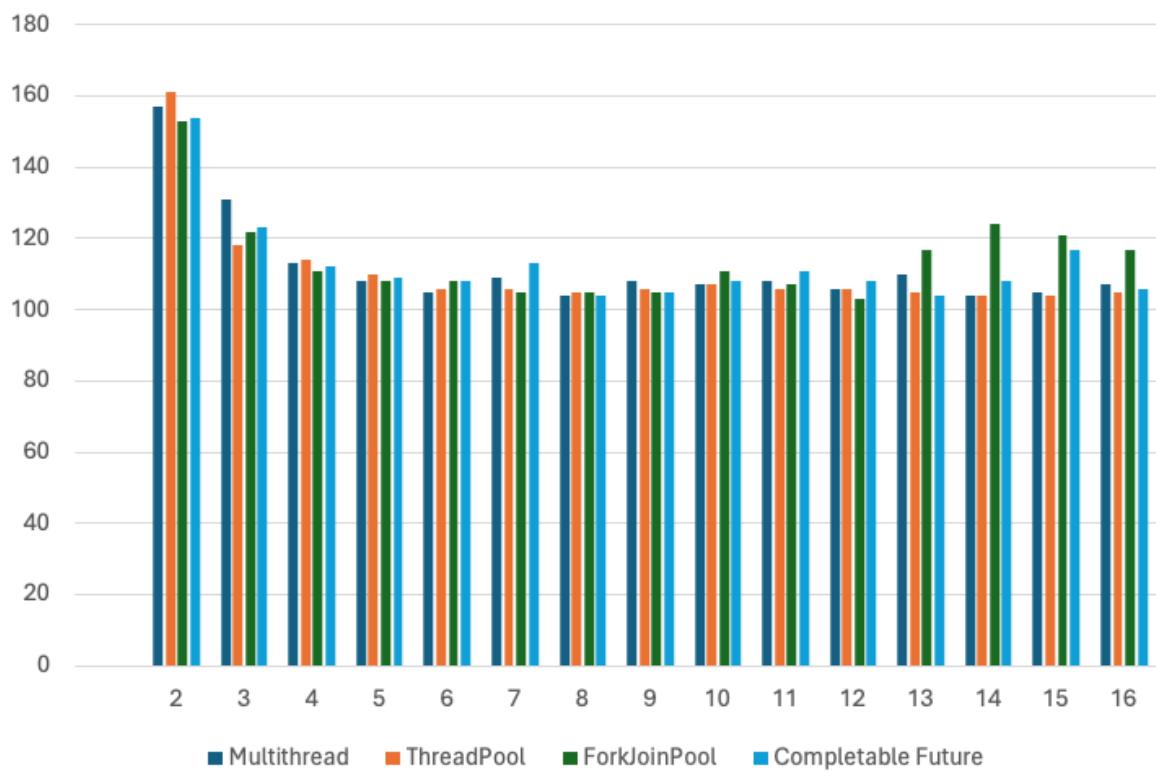
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
Tree	215	211	211	212

2. Multithreaded and Thread-Pool

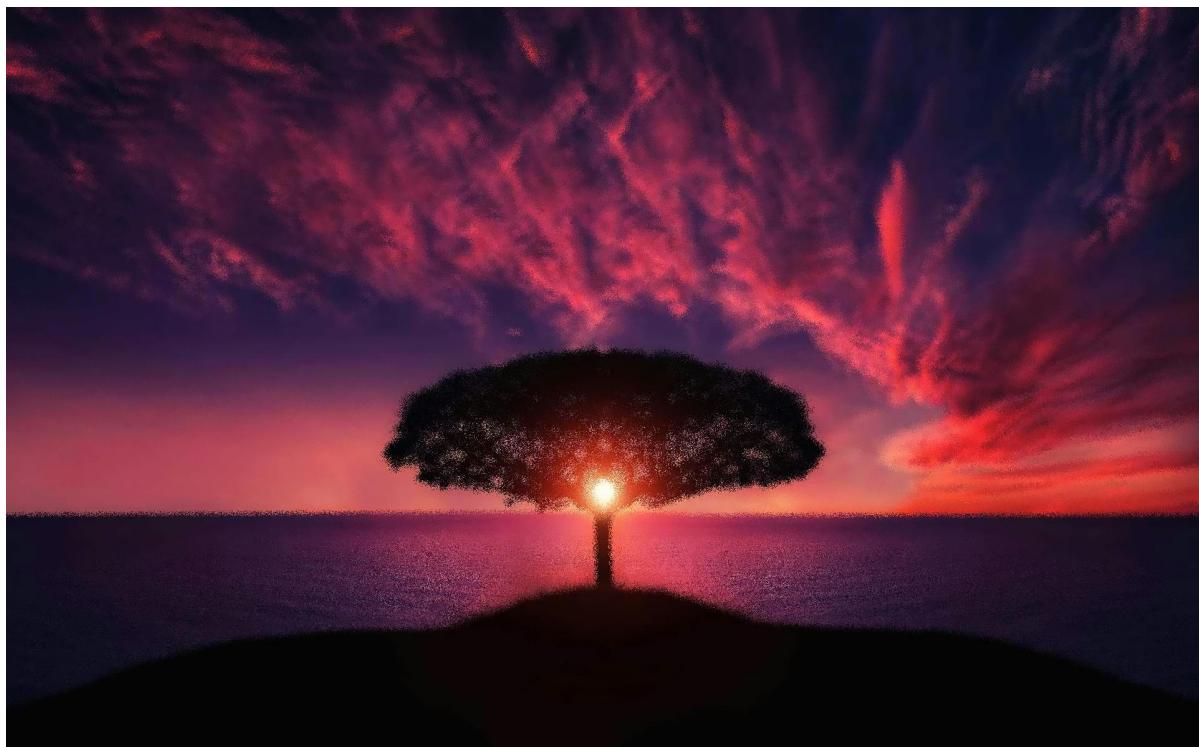
Todos os tempos são resultantes de uma média de três execuções.

Numero de thread	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	157	131	113	108	105	109	104	108	107	108	106	110	104	105	107
ThreadPool	161	118	114	110	106	106	105	106	107	106	106	105	104	104	105
ForkJoinPool	153	122	111	108	108	105	105	105	111	107	103	117	124	121	117
Completable Future	154	123	112	109	108	113	104	105	108	111	108	104	108	117	106

Tree.jpg - Swirl Filter



Glass



1. Sequencial -

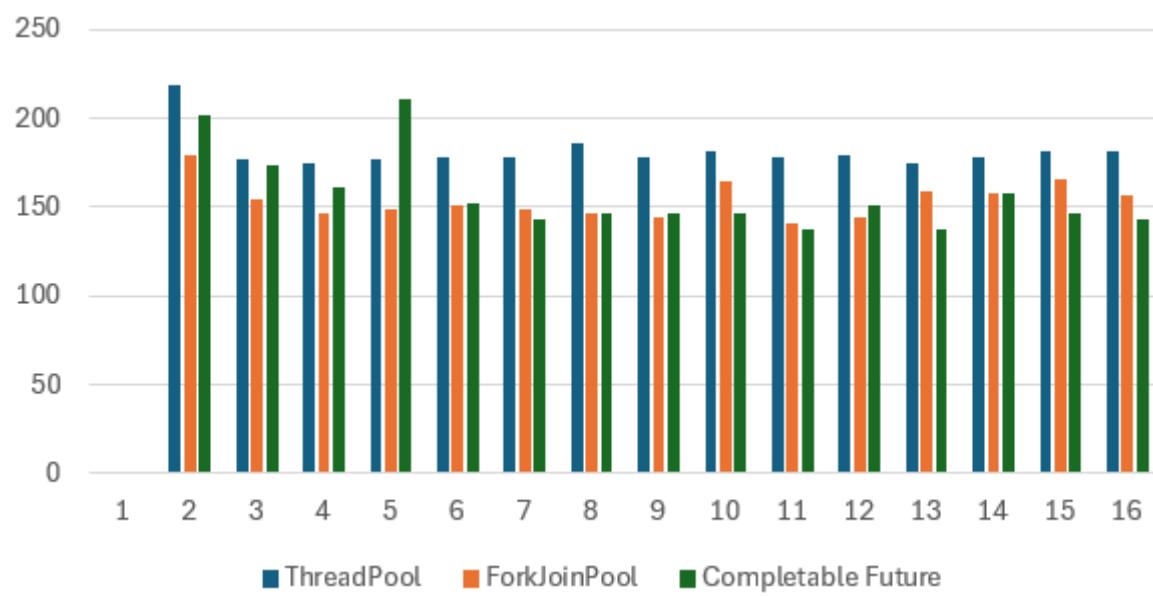
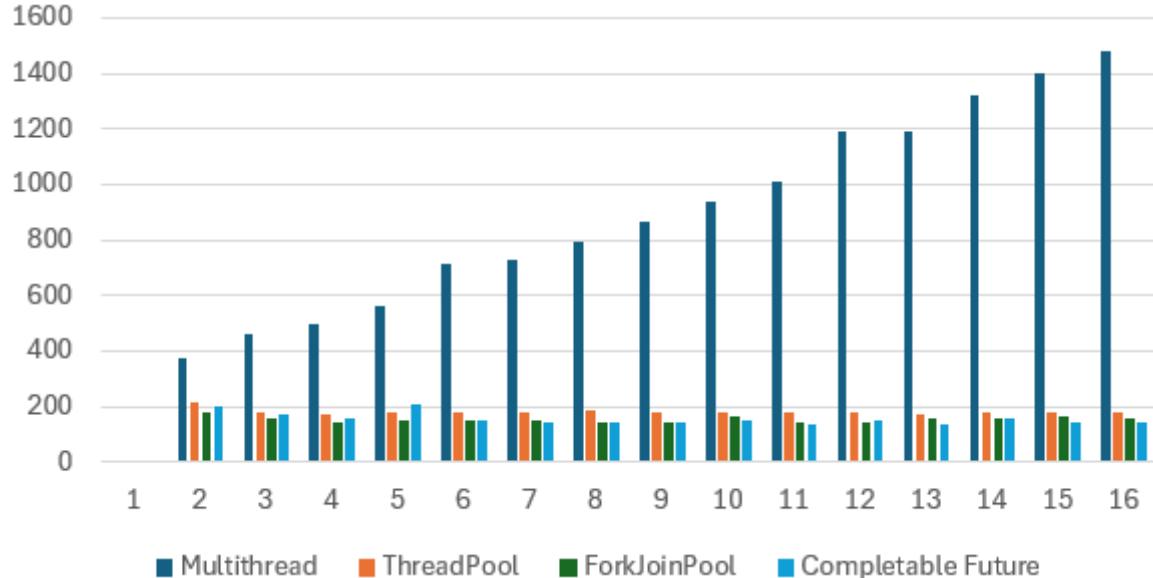
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
Tree	260	236	250	248,66

2. Multithreaded and Thread-Pool -

O processador utilizado para testar esta implementação é AMD Ryzen 7 5800H with Radeon Graphics com 8 cores e 16 threads.

Todos os tempos são resultantes de uma média de três execuções.

Numero de thread	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	371	460,66	500,33	558,66	711	730,33	792,33	866,33	934,66	1009,33	1193,33	1193,33	1324	1402,33	1482,33
ThreadPool	218,33	177,33	175	177	178	178,33	185,66	178,33	181,66	178,66	179	174,66	177,66	182	181
ForkJoinPool	179,66	154,44	146,33	149	150,66	148,66	146	144	164,44	141,33	144,33	159,33	157,66	166	156,33
Completable Future	201,66	173,66	160,66	211	151,66	143,66	146	146	147	137	150,66	137,66	158	146,66	143



Blur

O processador utilizado para testar esta implementação é um 10th Generation Intel® Core™ Core i7-10750H Processor com 6 cores e 12 threads.



1. Sequential

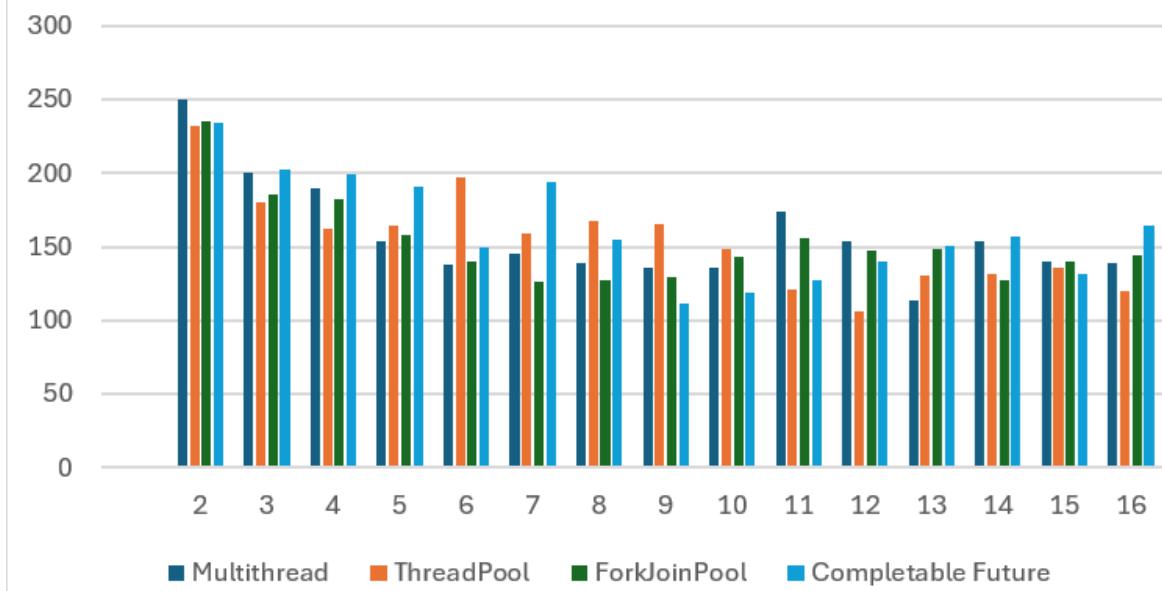
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
Tree	339	344	370	351

2. Multithreaded and Thread-Pool

Todos os tempos são resultantes de uma média de três execuções.

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	250,67	200,67	190	153,67	138	145,67	139	135,67	135,67	173,67	153,67	114	154	139,67	139,33
ThreadPool	232	180	162	164	197	159	168	166	149	121	106	131	132	136	120
ForkJoinPool	235	186	183	158	140	126	127	130	143	156	148	149	127	140	144
CompletableFuture	234	203	199	191	150	194	155	112	119	128	140	151	157	132	165

Tree.jpg - Blur Filter



Conditional Blur

Processor: AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx, 2.30 GHz.



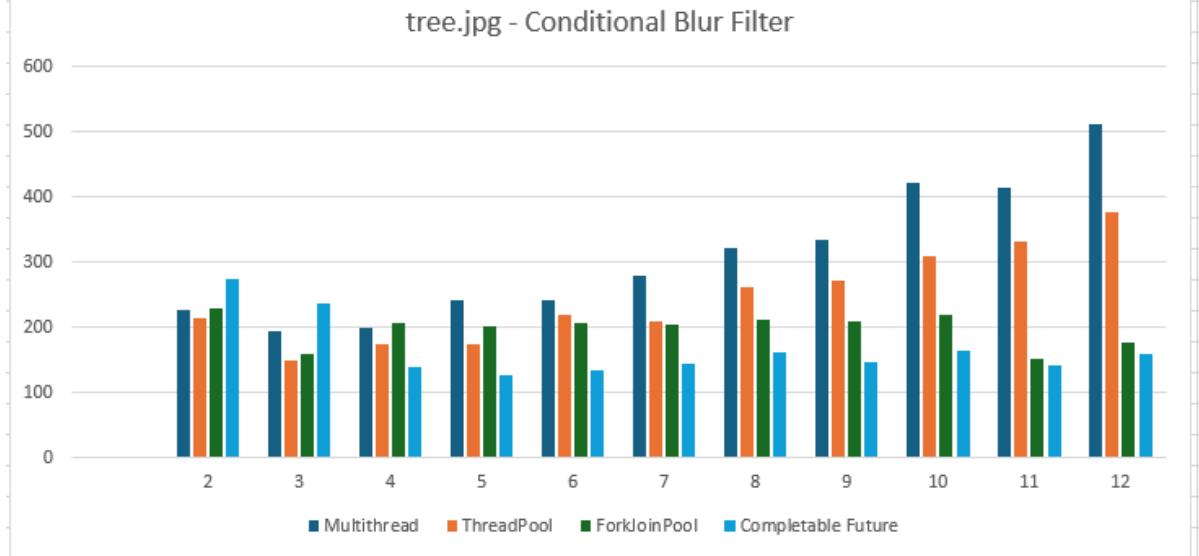
1. Sequential

Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
Tree	361	315	222	299

2. Multithreaded and Thread-Pool

Todos os tempos são resultantes de uma média de três execuções.

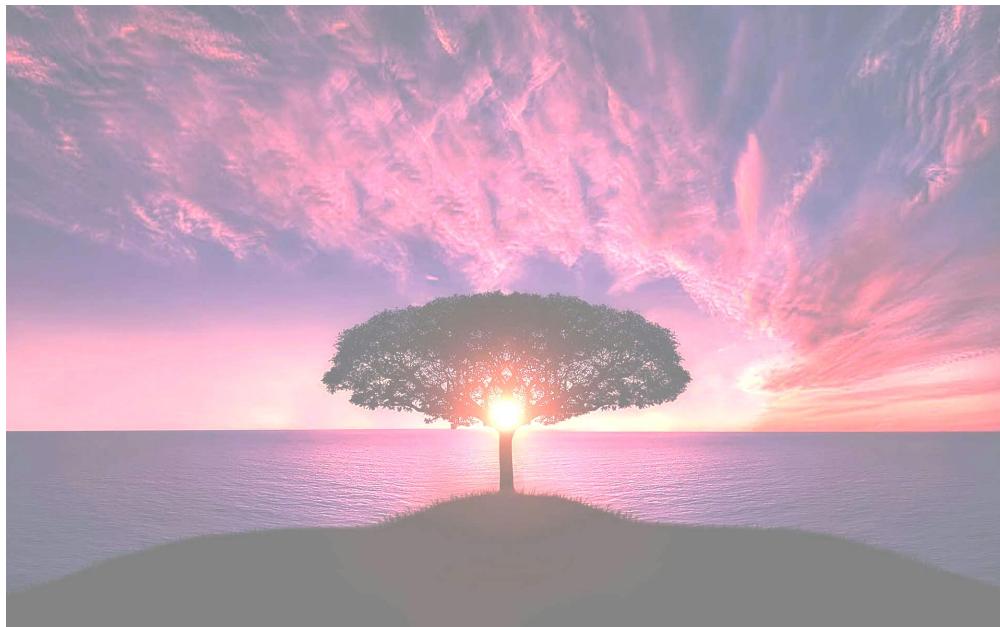
Numero de thread	2	3	4	5	6	7	8	9	10	11	12
Multithread	228	195	199	242	242	279	322	335	421	415	512
ThreadPool	215	150	174	175	219	209	261	273	310	333	378
ForkJoinPool	230	158	208	201	208	204	213	210	220	152	176
Completable Future	274	237	138	126	134	144	162	146	163	141	160



Tree.jpg

Brightness

O processador utilizado para testar esta implementação é um Apple M1 Pro 10-core.



1. Sequential

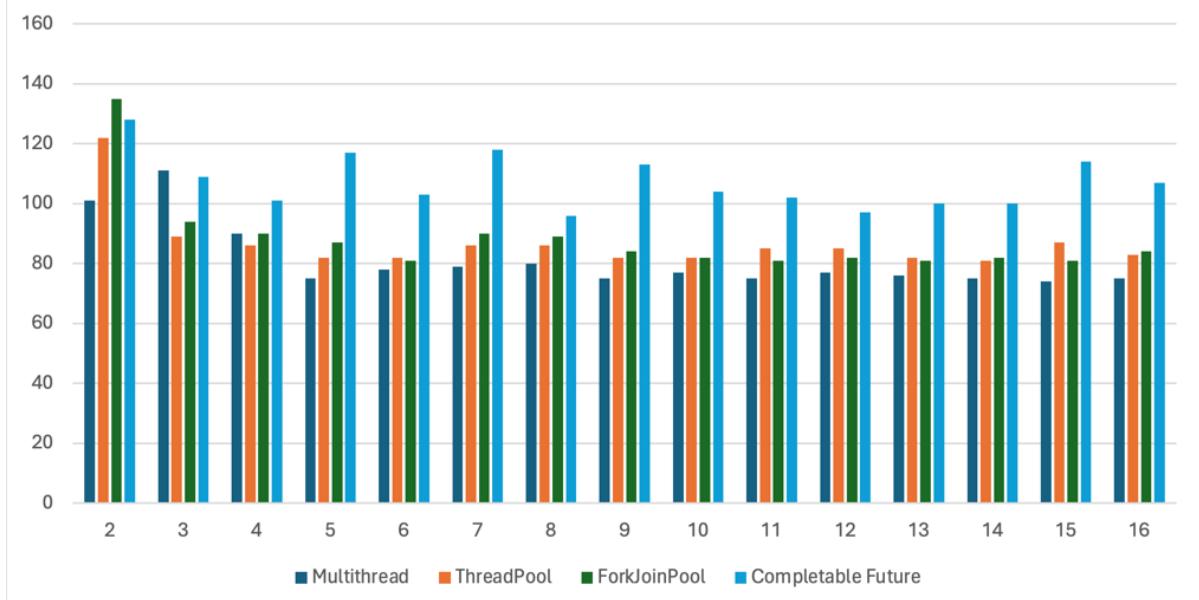
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
Tree	187	193	190	190

2. Multithreaded and Thread-Pool

Todos os tempos são resultantes de uma média de três execuções.

Numero de thread	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	101	111	90	75	78	79	80	75	77	75	77	76	75	74	75
ThreadPool	122	89	86	82	82	86	86	82	82	85	85	82	81	87	83
ForkJoinPool	135	94	90	87	81	90	89	84	82	81	82	81	82	81	84
Completable Future	128	109	101	117	103	118	96	113	104	102	97	100	100	114	107

Tree.jpg - Bright Filter



Turtle.jpg

Brightness

O processador utilizado para testar esta implementação é um Apple M1 Pro 10-core.



1. Sequential

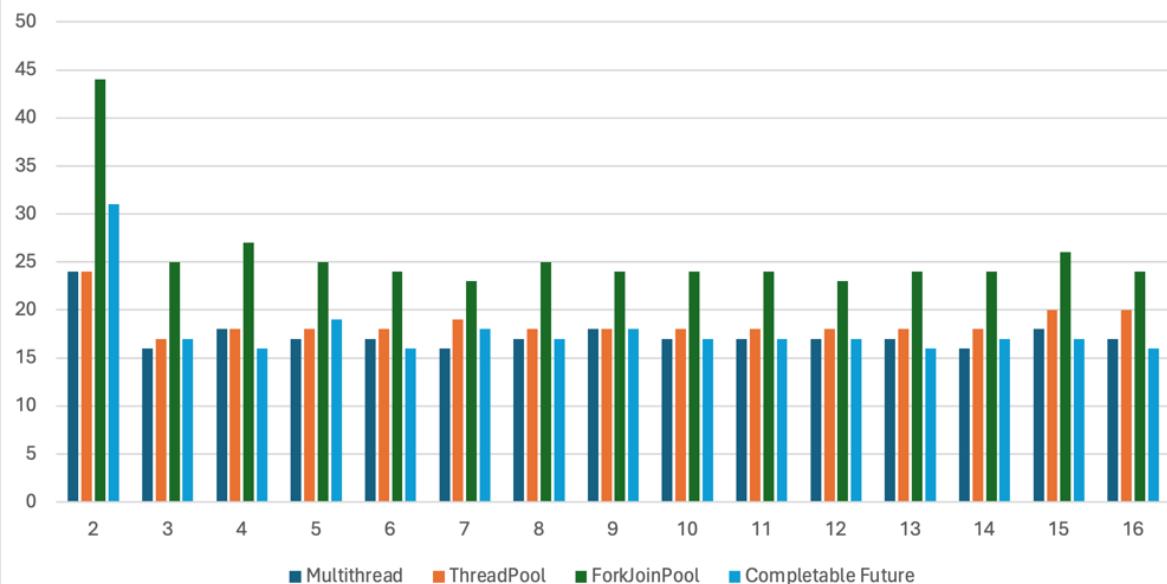
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
Turtle	50	51	51	50

2. Multithreaded and Thread-Pool

Todos os tempos são resultantes de uma média de três execuções.

Número de thread	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	24	16	18	17	17	16	17	18	17	17	17	17	16	18	17
ThreadPool	24	17	18	18	18	19	18	18	18	18	18	18	18	20	20
ForkJoinPool	44	25	27	25	24	23	25	24	24	24	23	24	24	26	24
Completable Future	31	17	16	19	16	18	17	18	17	17	17	16	17	17	16

Turtle.jpg - Bright Filter



Grayscale



1. Sequential -

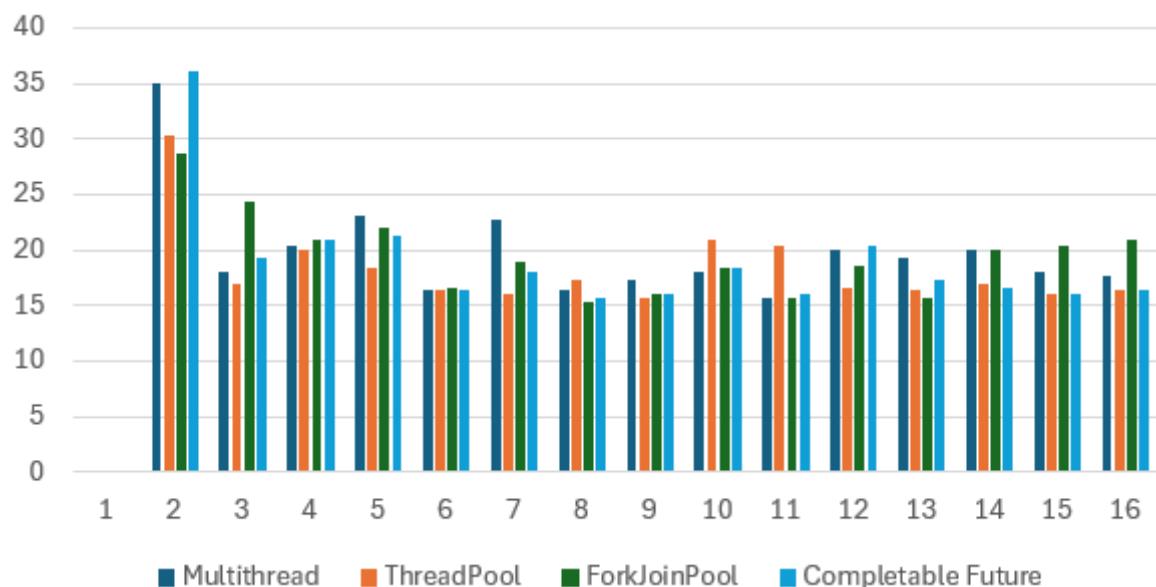
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
Turtle	57	62	68	62,33

2. Multithreaded and Thread-Pool -

O processador utilizado para testar esta implementação é AMD Ryzen 7 5800H with Radeon Graphics com 8 cores e 16 threads.

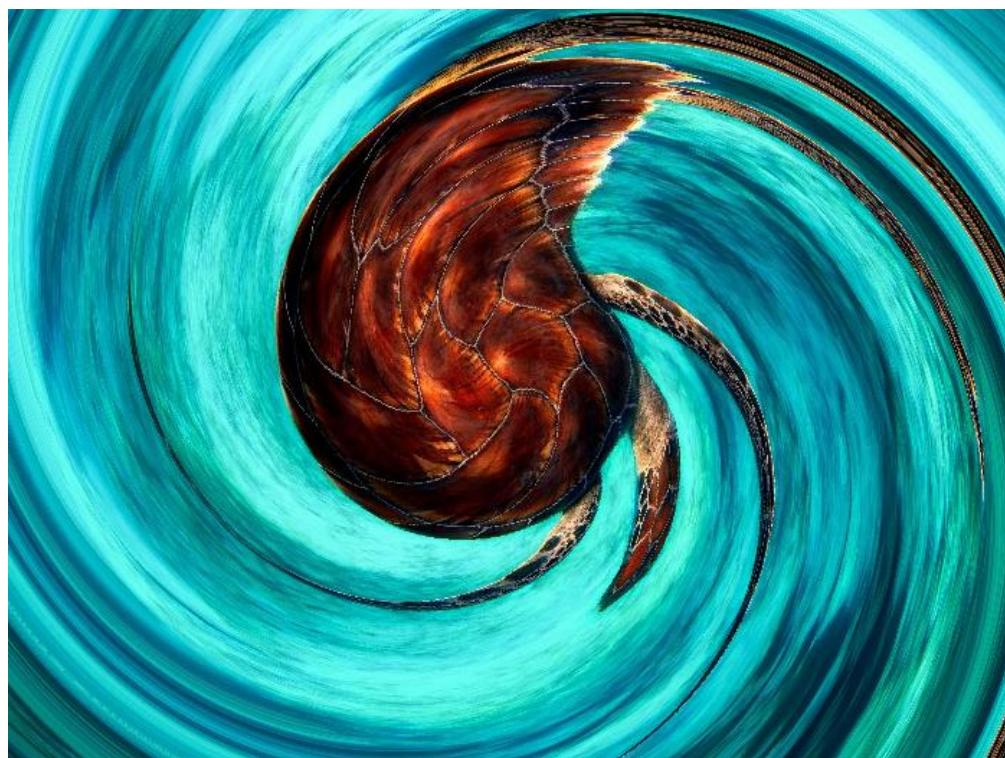
Todos os tempos são resultantes de uma média de três execuções.

Numero de thread	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	35	18	20,33	23	16,33	22,66	16,33	17,33	18	15,66	20	19,33	20	18	17,66
ThreadPool	30,33	17	20	18,33	16,33	16	17,33	15,66	21	20,33	16,66	16,33	17	16	16,33
ForkJoinPool	28,66	24,33	21	22	16,66	19	15,33	16	18,33	15,66	18,66	15,66	20	20,33	21
Completable Future	36	19,33	21	21,33	16,33	18	15,66	16	18,33	16	20,33	17,33	16,66	16	16,33



Swirl

O processador utilizado para testar esta implementação é um Apple M1 Pro 10-core.



1. Sequential

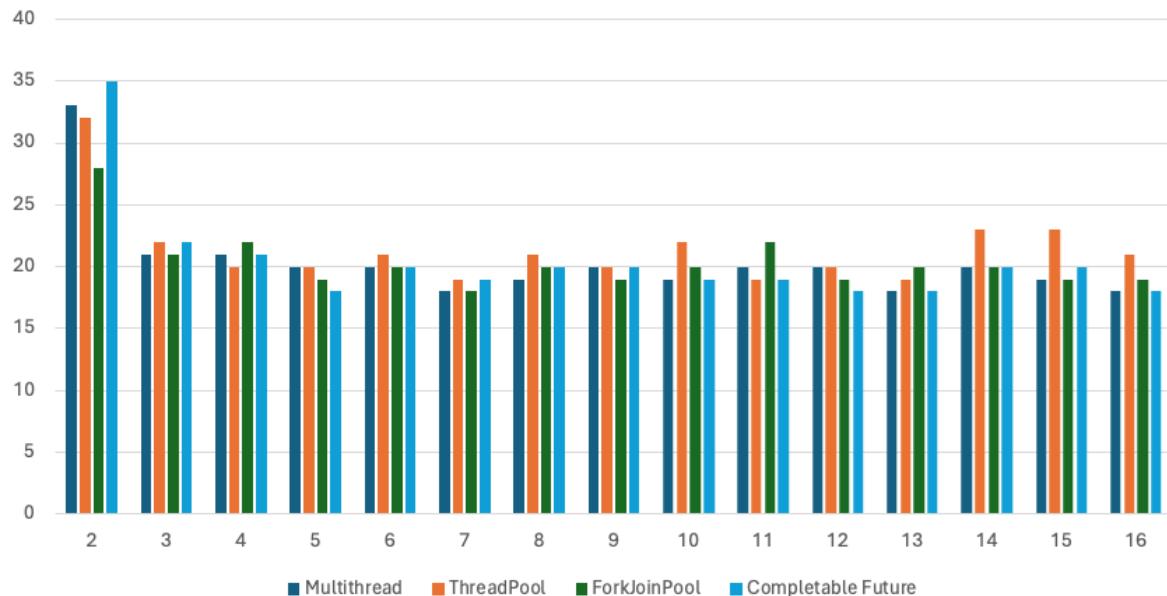
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
Turtle	77	77	91	245

2. Multithreaded and Thread-Pool

Todos os tempos são resultantes de uma média de três execuções.

Numero de thread	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	33	21	21	20	20	18	19	20	19	20	20	18	20	19	18
ThreadPool	32	22	20	20	21	19	21	20	22	19	20	19	23	23	21
ForkJoinPool	28	21	22	19	20	18	20	19	20	22	19	20	20	19	19
CompletableFuture	35	22	21	18	20	19	20	20	19	19	18	18	20	20	18

Turtle.jpg - Swirl Filter



Glass



1. Sequencial -

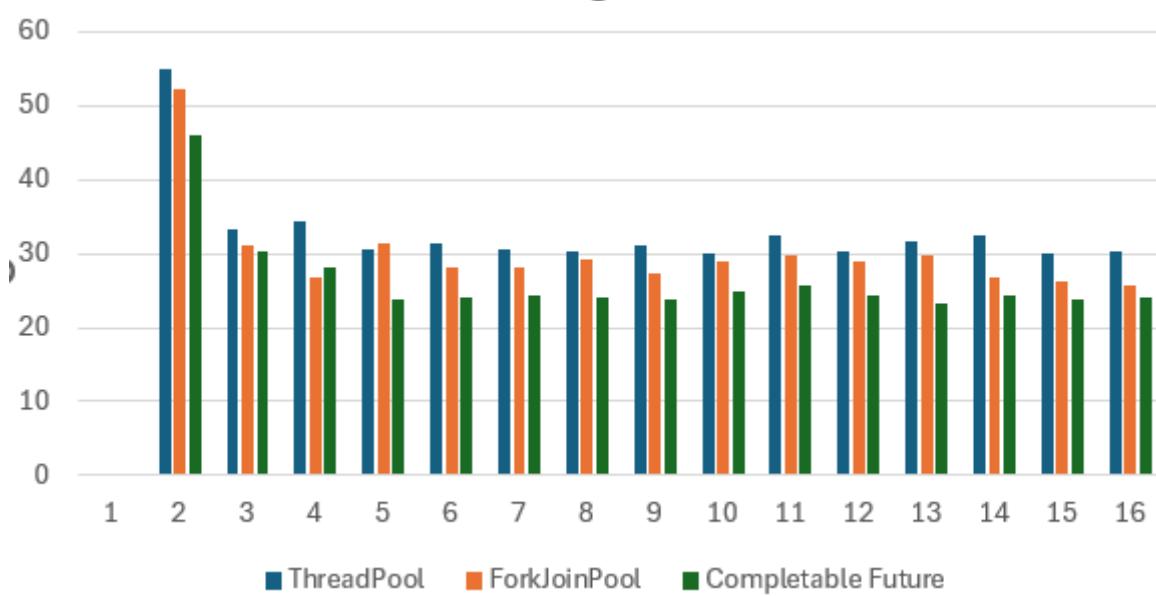
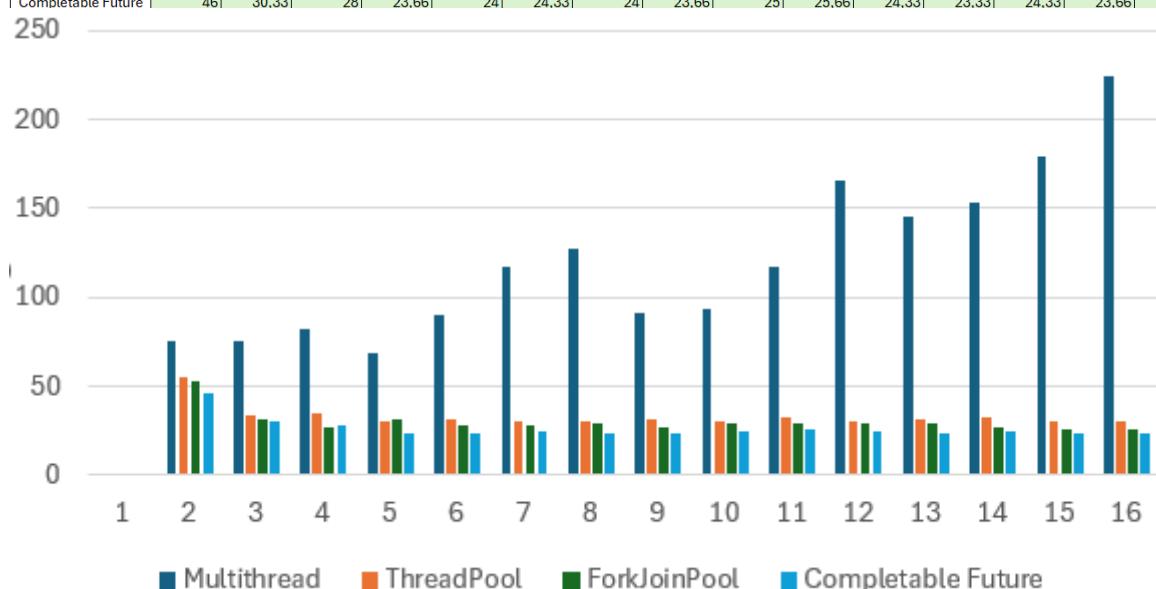
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
Turtle	70	81	83	78

2. Multithreaded and Thread-Pool -

O processador utilizado para testar esta implementação é AMD Ryzen 7 5800H with Radeon Graphics com 8 cores e 16 threads.

Todos os tempos são resultantes de uma média de três execuções.

Numero de thread	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	75,33	75,66	82,66	68,66	90,33	117,66	127	91,33	93,33	116,66	166	145,33	153,66	179	224
ThreadPool	55	33,33	34,33	30,66	31,33	30,66	30,33	31	30	32,33	30,33	31,66	32,33	30	30,33
ForkJoinPool	52,33	31	26,66	31,33	28	28	29,33	27,33	29	29,66	29	29,66	26,66	26,33	25,66
CompletableFuture	46	30,33	28	23,66	24	24,33	24	23,66	25	25,66	24,33	23,33	24,33	23,66	24



Blur

O processador utilizado para testar esta implementação é um 10th Generation Intel® Core™ i7-10750H Processor com 6 cores e 12 threads.



1. Sequential

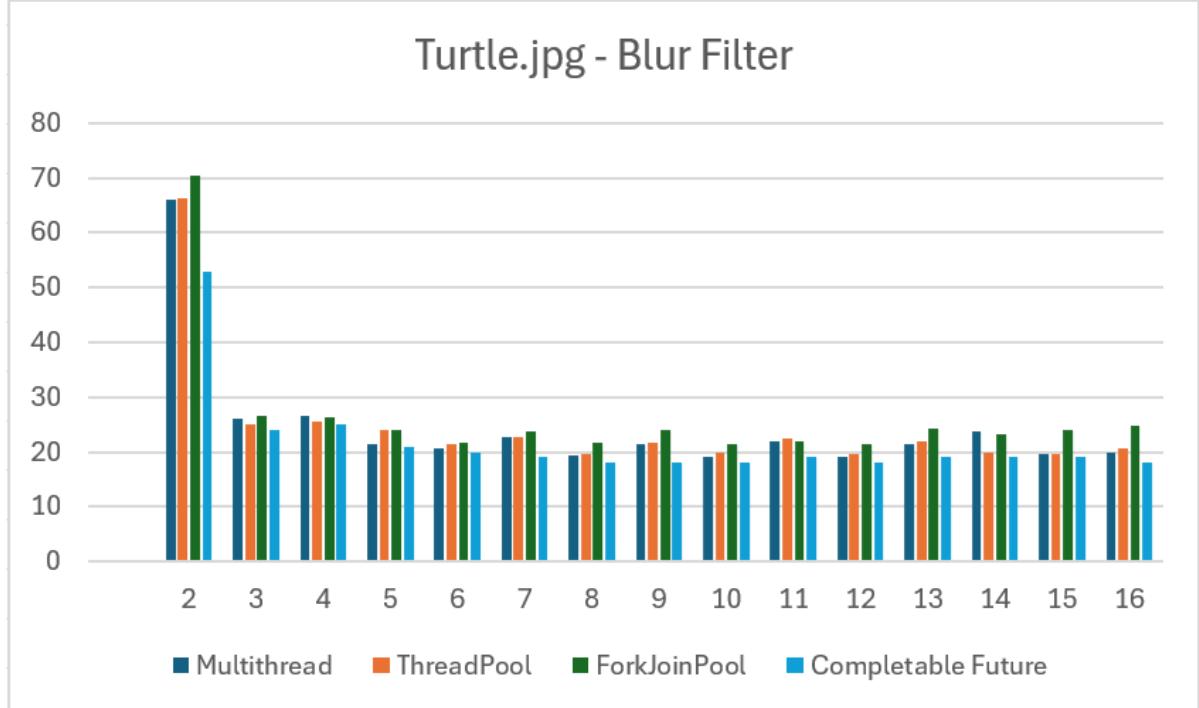
Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
Turtle	121	111	135	123

2. Multithreaded and Thread-Pool

Todos os tempos são resultantes de uma média de três execuções.

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Multithread	66	26	26,67	21,33	20,67	22,67	19,33	21,33	19	22	19	21,33	23,67	19,66	20
ThreadPool	66,33	25	25,67	24	21,33	22,67	19,67	21,67	20	22,33	19,67	22	20	19,67	20,67
ForkJoinPool	70,33	26,67	26,33	24	21,67	23,67	21,67	24	21,33	22	21,33	24,33	23,33	24	24,67
Completable Future	53	24	25	21	20	19	18	18	18	19	18	19	19	19	18

Turtle.jpg - Blur Filter



Conditional Blur

Processor: AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx, 2.30 GHz.



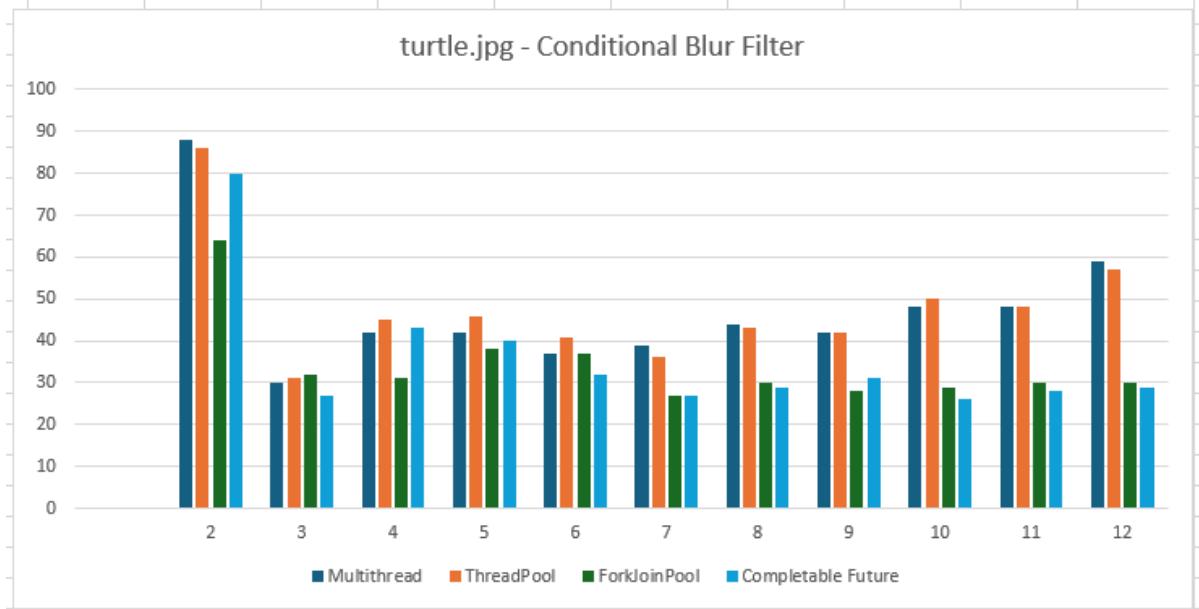
1. Sequential

Resultados	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
Turtle	123	48	36	69

2. Multithreaded and Thread-Pool

Todos os tempos são resultantes de uma média de três execuções.

Numero de thread	2	3	4	5	6	7	8	9	10	11	12
Multithread	88	30	42	42	37	39	44	42	48	48	59
ThreadPool	86	31	45	46	41	36	43	42	50	48	57
ForkJoinPool	64	32	31	38	37	27	30	28	29	30	30
Completable Future	80	27	43	40	32	27	29	31	26	28	29



Conclusões das implementações

Após a análise das diversas abordagens para a aplicação de filtros de imagens, foi possível constatar várias diferenças significativas em termos de desempenho entre os métodos sequenciais e os métodos que utilizam múltiplas threads.

Ficou evidente que o processamento sequencial dos filtros de imagens é consistentemente mais lento quando comparado com as abordagens que empregam múltiplas threads ou utilizam um pool de threads. A utilização de um único fluxo de processamento não aproveita as capacidades modernas dos processadores multi-core, resultando em uma utilização subótima dos recursos disponíveis e, consequentemente, em um tempo de execução prolongado.

Dentro das implementações que utilizam threads, o esperado seria que o método multi-threaded tradicional, embora mais rápido do que o processamento sequencial, apresentasse uma queda de desempenho em comparação com os métodos baseados em thread-pool. Isto deveria ser atingido devido à melhor gestão de recursos e à eficiência na distribuição de tarefas que os pools de threads oferecem, minimizando o tempo ocioso das threads e otimizando a carga de trabalho distribuída, como observamos no filtro glass em qualquer imagem. Apesar disso, não se verificou em todos os filtros, como por exemplo o multithreaded do filtro glass em todos os casos.

A quantidade de threads utilizada tem um impacto direto na performance, especialmente em relação à arquitetura do processador em uso. Esperava verificar que os desempenhos mais elevados seriam alcançados quando o número de threads utilizadas aproxima-se do número de núcleos do processador (n ou $n+1$ ou $n+2$). Isso deve-se ao fato de que cada thread pode ser executada simultaneamente em seu próprio núcleo, maximizando assim a utilização do processador sem causar sobrecarga significativa devido ao contexto de troca ou à competição por recursos. Apesar de tudo, verificou-se um pico de melhoria de desempenho de 2 para 3 threads, mas daí em diante os tempos poderiam manter-se os mesmos por maior parte dos filtros.

Esses resultados reforçam a importância de considerar a arquitetura do hardware ao desenvolver ou otimizar aplicações para processamento de imagens. A escolha entre sequencial e paralelo, e entre diferentes formas de paralelismo, deve ser informada não apenas pelas características intrínsecas do algoritmo, mas também pelo ambiente de hardware em que a aplicação será executada. Além disso, para obter o máximo desempenho, os desenvolvedores devem aspirar a alinhar o número de threads com o número de núcleos disponíveis no processador, ajustando conforme necessário para evitar tanto a subutilização quanto a sobrecarga do sistema.

Garbage Collector

In order to improve performance, we looked into garbage collection tuning.
We analyzed the following garbage collectors: Serial, Parallel, G1, Shenandoah and Z.
The following criteria was used to compare the different garbage collectors:

- Throughput: Average time spent running code vs running GC;
- Latency: amount of time code pauses for GC to run;
- Memory Usage: the size of the heap;

For this project, we are particularly interested in throughput and latency, as image processing is a CPU-bound task, but we are also concerned with the time the user has to wait for the response with the resulting image.

Memory, however, isn't a big concern here since this is a small application.

Serial Garbage Collector

The Serial Garbage Collector works on a single thread.
As such it is best suited for single-processor machines.
It has the advantage of requiring a small amount of memory.

Parallel Garbage Collector

The Parallel Garbage Collector runs on multiple threads.
As such, it is recommended for multicore systems.
It is designed to reduce CPU time spent on garbage collection, being ideal for long-running background tasks.

G1 Garbage Collector

The G1 (or Garbage First) Garbage Collector also uses multiple threads, but differs from the Parallel GB in that some work is done concurrently with the application. The collector tries to achieve high throughput along with short pause times. It is particularly useful for applications that require predictable garbage collection pause times.

Shenandoah Garbage Collector

The Shenandoah Garbage Collector is a low-pause-time garbage collector, having low latency. It also works concurrently with the application, and attempts to avoid stop-the-world pauses. It tries to keep pause times constant, regardless of heap size.

Z Garbage Collector

The Z Garbage Collector is a scalable garbage collector, designed for large heaps. It is also low-latency, aiming not to exceed a pause time of 10ms, and works concurrently with the application.

Results

The following tests were conducted with the following conditions:

- processor - AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx, 2.30 GHz.
- image size - 1920x1195 pixels.
- number of threads - 9.
- filter - Glass with fork join pool implementation.

Garbage Collector	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
Serial	361	322	329	337.33
Parallel	358	321	291	323.33
G1	313	331	321	321.66
Z	437	412	425	424.66

The Shenandoah Garbage Collector was not tested as it is not supported by the used JDK distribution (Oracle).

As we can see, the G1 Garbage Collector had the best average time, closely followed by the Parallel Garbage Collector.

This was expected, as the G1 Garbage Collector has a good balance of throughput and latency, which is adequate for this application.

Tuning

The G1 Garbage Collector has a few possible tuning options that can be used to improve performance.

The two main ones are the following:

- XX:MaxGCPauseMillis=n - sets the maximum pause time goal. The default value is 200ms.

- `-XX:InitiatingHeapOccupancyPercent=n` - sets the percentage of the heap occupancy that triggers a marking cycle. The default value is 45.

We experimented with different values, and found best results with the following configuration

- `-XX:MaxGCPauseMillis=215`, to have a less demanding pause time goal, as the default value was a little low and caused the application to run slower;
- `-XX:InitiatingHeapOccupancyPercent=65`, to trigger a marking cycle at a higher heap occupancy percentage, as we found the default value to be unnecessarily low for this application.

So the command to run the application with these options would be:

```
java -XX:+UseG1GC -XX:MaxGCPauseMillis=215 -XX:InitiatingHeapOccupancyPercent=65 -
jar SISMD_Project.jar
```

With these values, we obtained the following results for the same conditions as before:

Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Average Time (ms)
289	289	285	287.66

As we can see, the average execution time was improved by 44ms, which brings the value below 300ms.