EA876 - Trabalho 2

- Álvaro Marques Macêdo - RA 212466
- Victor Toon de Araújo - RA 225231

Lógica Utilizada

Primeiramente, a ideia que tivemos foi de utilizar os processos/threads pra resolver cada um dos canais de cor separadamente, assim não temos problemas de conflito entre atividades paralelas. Nós interpretamos o vetor de cada canal como uma matriz de tamanho `[Altura da Imagem em Pixels]x[Largura da Imagem em Pixels]`. Mais sobre a implementação pode ser vista nos comentários dos arquivos principais: `linear.c`, `process.c` e `thread.c`, localizados no diretório `src`.

Instruções para a execução dos testes descritos

Comandos:

- make/make all: compila os arquivos de teste com as flags corretas e a imagem selecionada
- make test: roda os testes dos 3 arquivos, e chama a função em python para fazer os gráficos
- make clean: Limpa os arquivos criados na compilação e o executável gerado

Executando:

- Para executar os testes, primeiro use o comando `make` para compilar e buildas os arquivos e depois o `make test` para executar os testes. Os logs dos arquivos serão salvos na pasta logs. No arquivo Makefille, é possível mudar o valor de N do blur e a imagem a ser utilizada.

Executando testes individuais:

- Para a execução de testes individuais, deve ser feito o seguinte:
  `Arquivo-Binário [N] [Caminho para a Imagem de Entrada] [(opcional) Caminho para a Imagem de Saída]`
- Por exemplo: `./build/linear.o 5 ./data/cachorro.jpg ./saida.jpg`
  Nesse caso, buildamos somente o teste linear, com N = 5, com a imagem do cachorro e a saída será saida.jpg (a saída é opcional)

Casos específicos

- Temos o arquivo run-tests.sh também, que serve caso vc queira executar um teste específico várias vezes, que funciona da segunda forma: `./run-test.sh [Número de Iterações Para`

`cada Variação [N] [Imagem de Entrada] [Tipo de Uso]`

- Por exemplo:
  Exemplo para um canal de cor: `./run-test.sh 50 5 ./data/cachorro.jpg -1p`
  Exemplo para três canais: `./run-test.sh 50 5 ./data/cachorro.jpg`
- Tipo de uso: São 3 tipos de uso disponíveis:
  → 1 canal: `-1p` ;
  → 2 canais: `-2p` ;
  → 3 canais: Vazio;

Testes

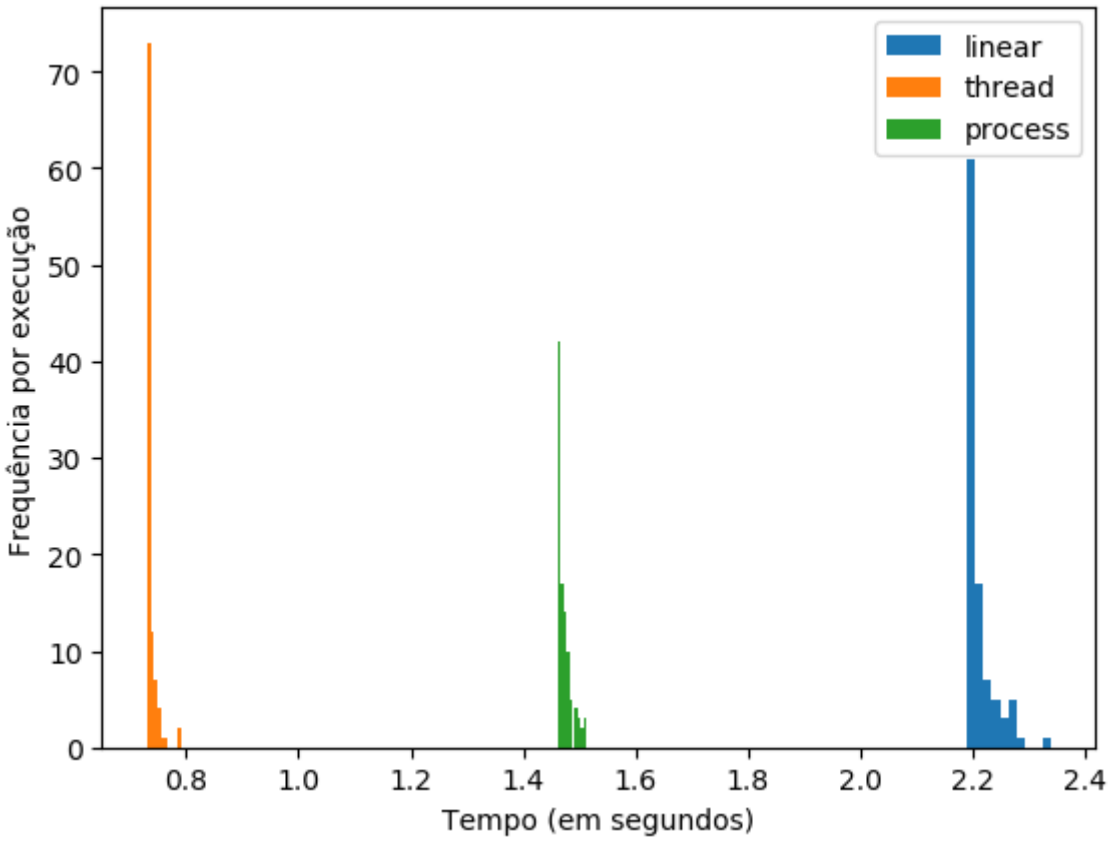
Setup Utilizado

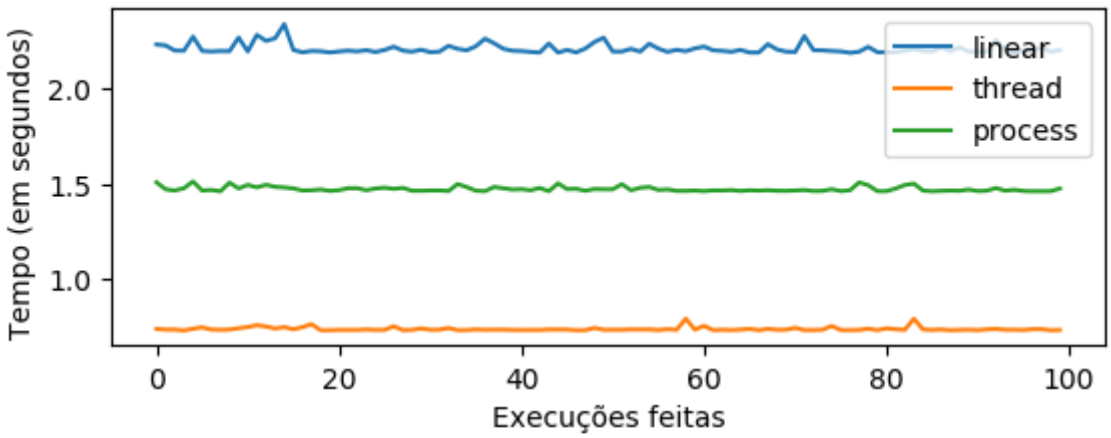
Todos os testes foram conduzidos usando o seguinte setup:

Embora estejamos usando o OSX, comparamos os resultados que obtivemos com testes mais rápidos feitos em outras máquinas com Linux e não vimos uma diferença entre o resultados delas e os que iremos apresentar.

Testes Iniciais

Como pedido, em cada teste executamos cada variação 100 vezes, decidimos considerar no nosso teste incial o N = 5 e a imagem sendo a encontrada em `./data/cachorro.jpg (1920x1080)` :

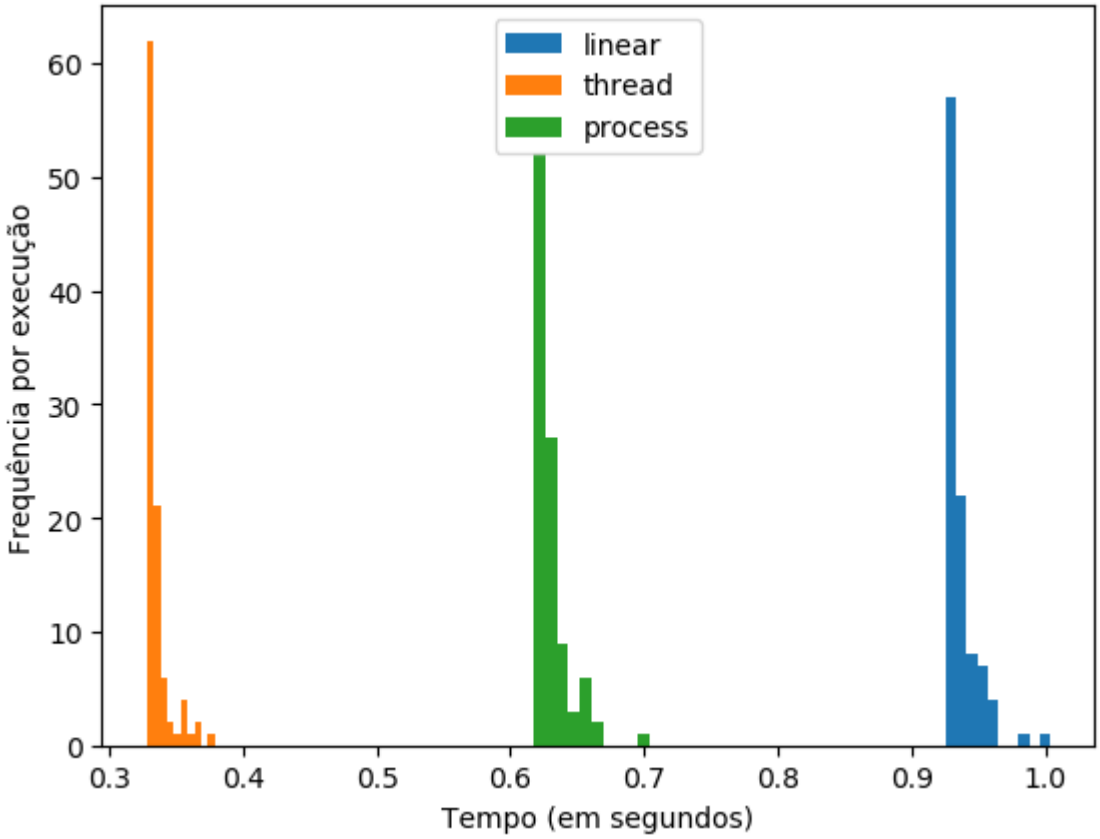


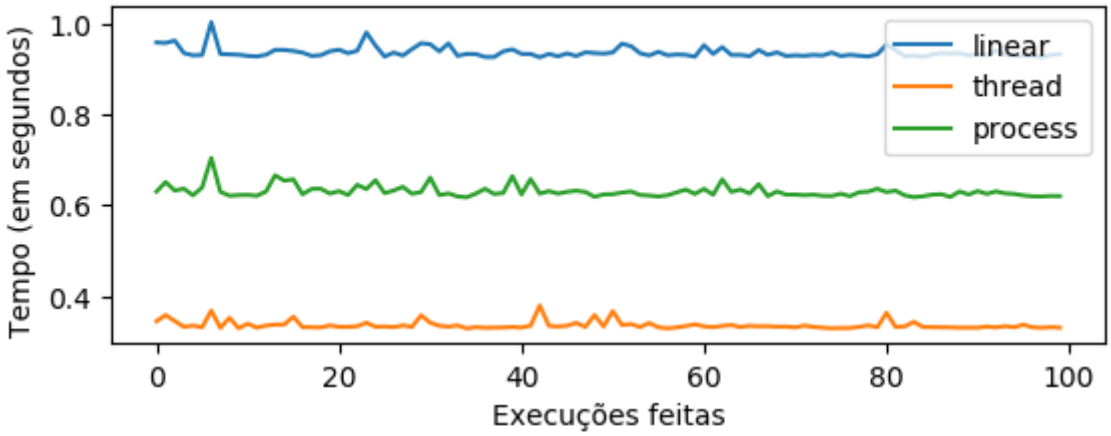


	Média (em segundos)	Desvio Padrão (em segundos)
thread	0.738	0.010
process	1.473	0.012
linear	2.210	0.026

O resultado encontrado não foi exatamente o esperado, visto que já que tanto a variação de multiprocessamento quando a de multithread fazem os três canais paralelamente, mas a variação que utiliza o multiprocessamento foi consideravelmente mais lenta que a de multithread. Alternamos então o N para ver se era um caso isolado:

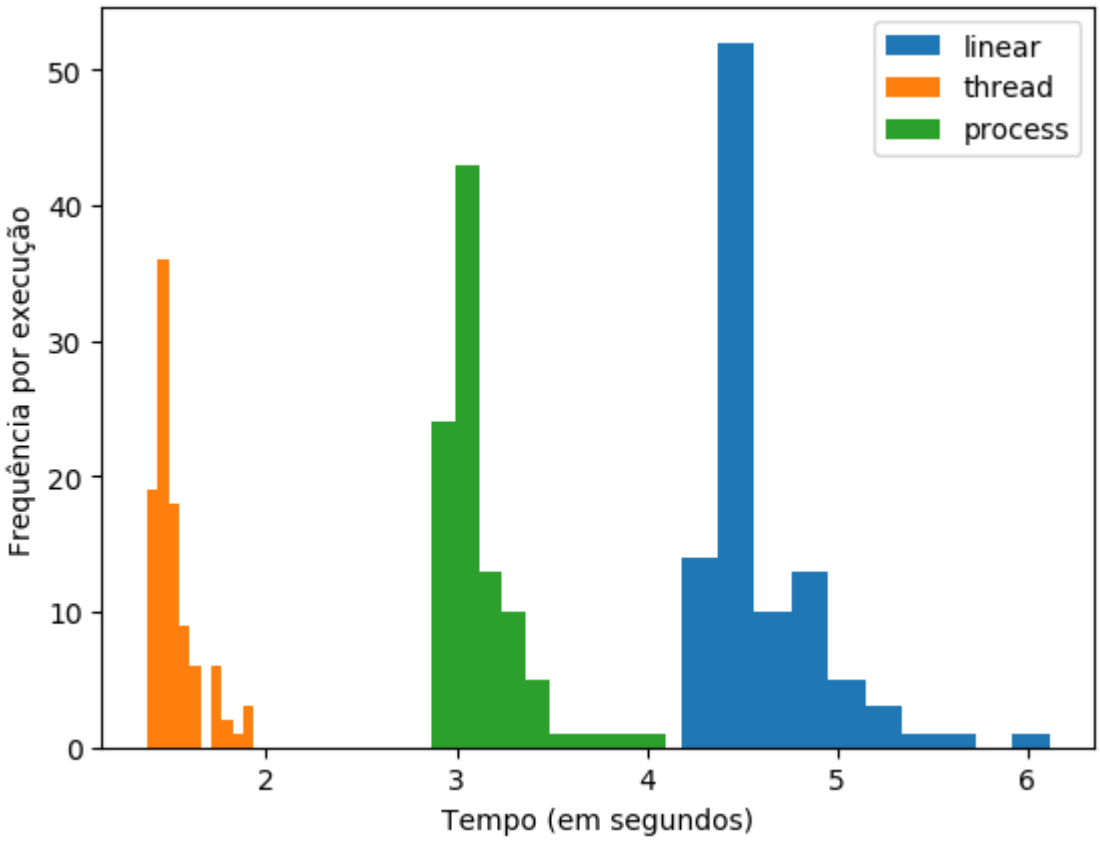
N = 3:

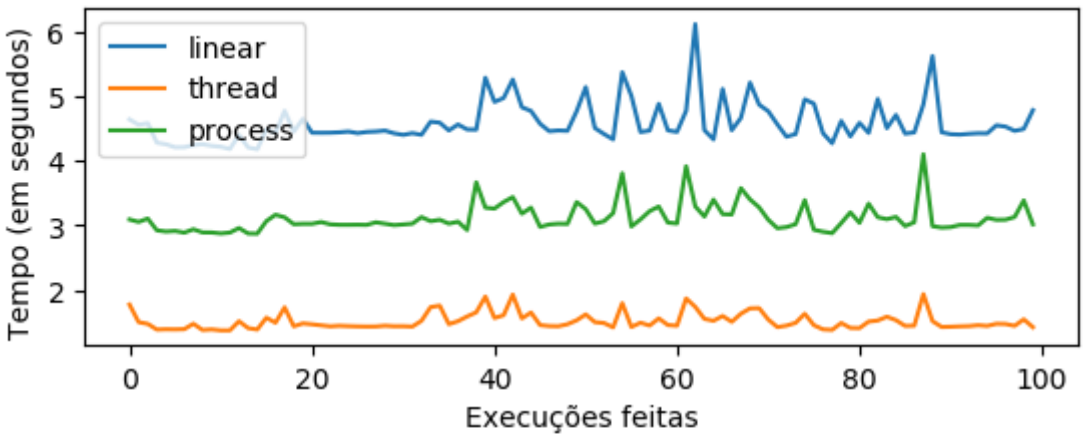




	Média (em segundos)	Desvio Padrão (em segundos)
thread	0.335	0.009
process	0.630	0.013
linear	0.937	0.012

N = 7:

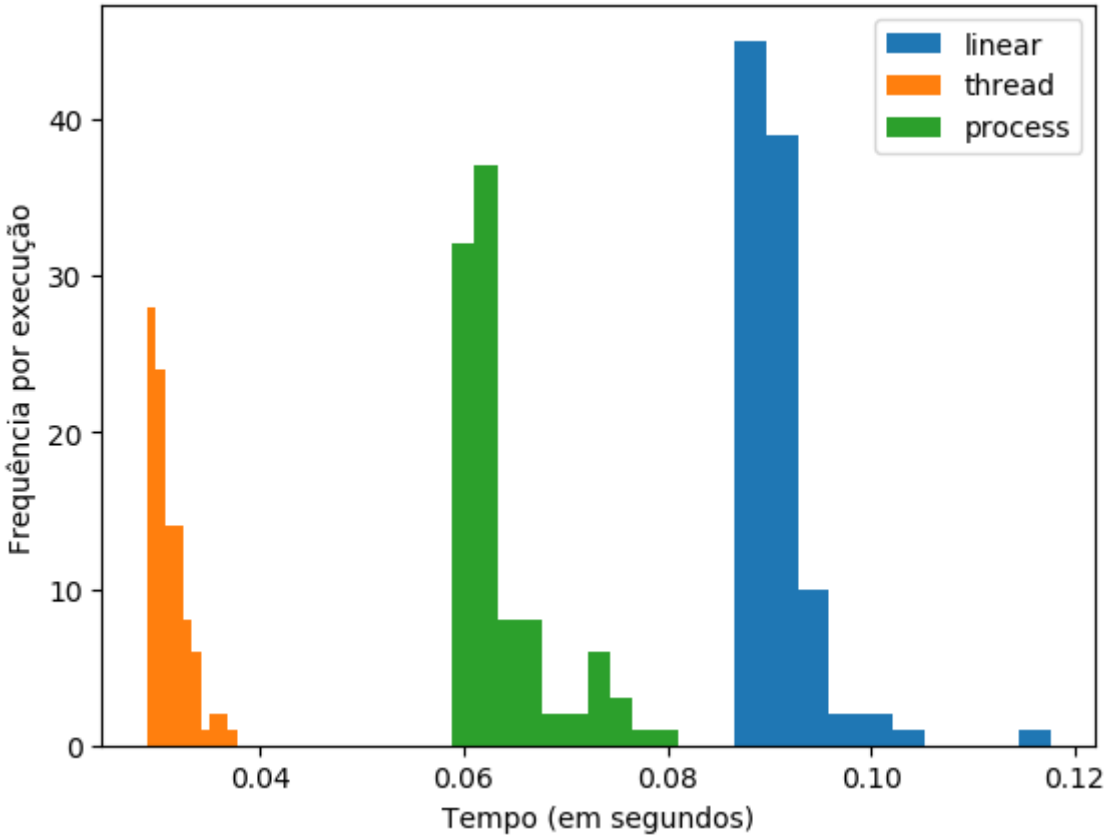


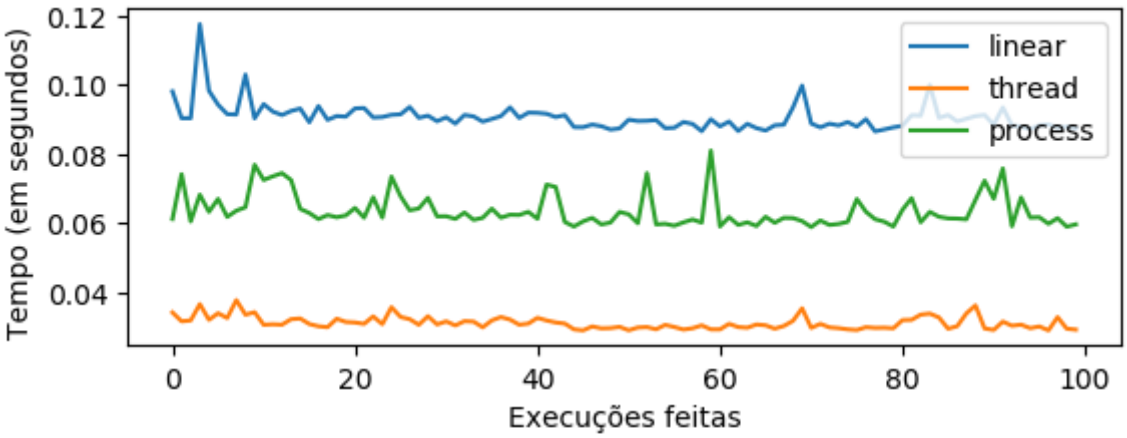


	Média (em segundos)	Desvio Padrão (em segundos)
thread	1.515	0.126
process	3.113	0.214
linear	4.582	0.316

Mas os resultados também foram semelhantes. Utilizamos também imagens de resoluções diferentes:

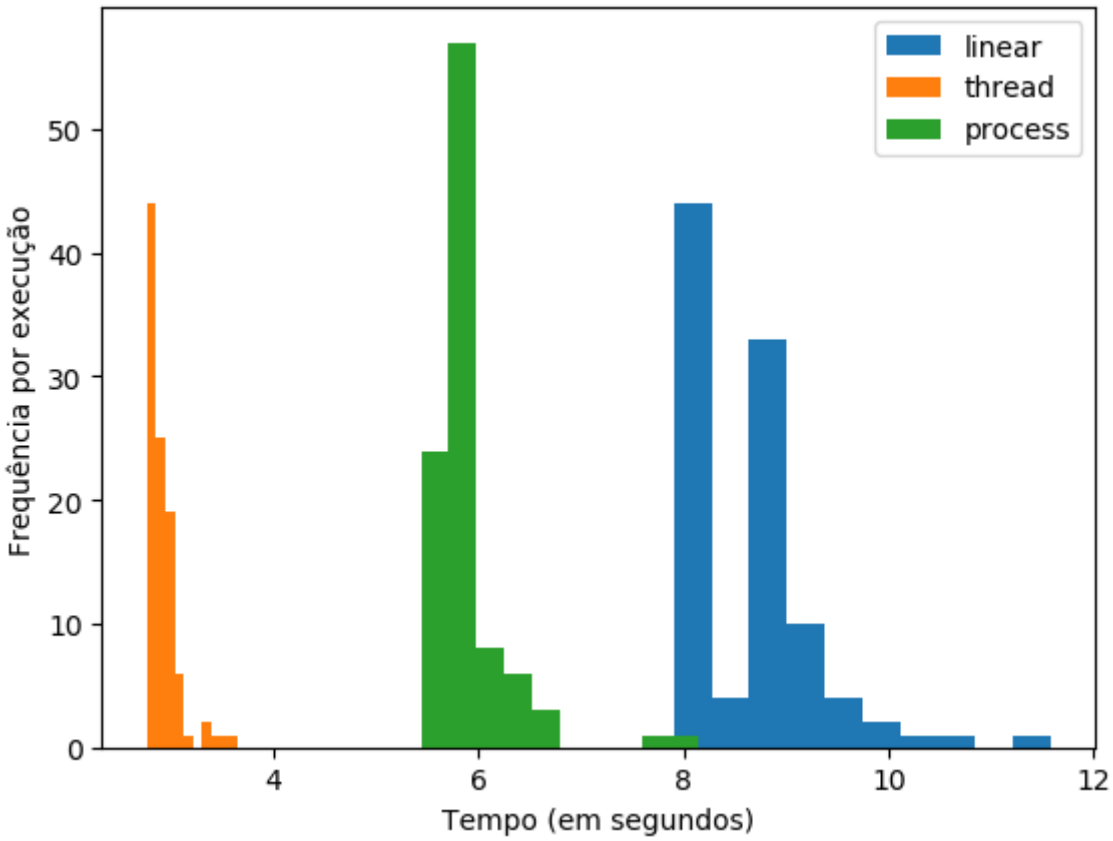
`'./data/soundfood.jpeg' (300x300):`

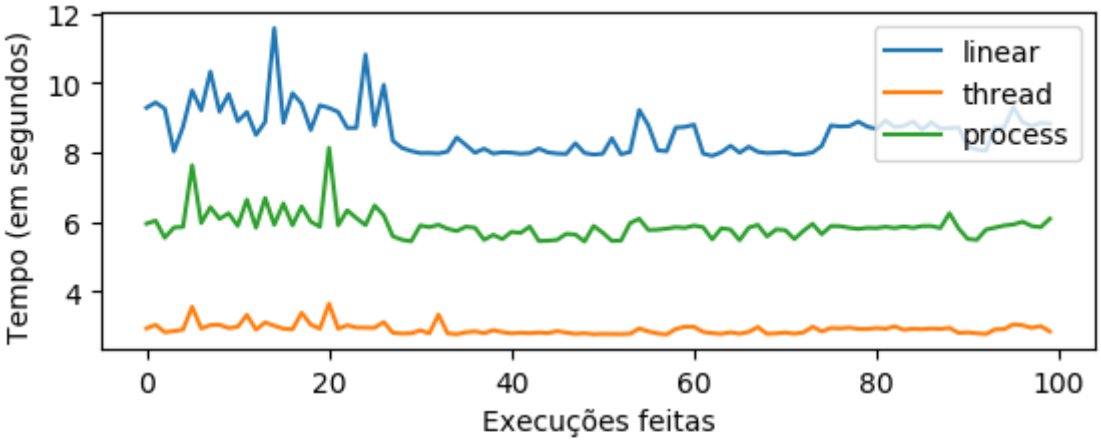




	Média (em segundos)	Desvio Padrão (em segundos)
thread	0.031	0.002
process	0.064	0.005
linear	0.091	0.004

'./data/onePiece.png' (3840x2160) :



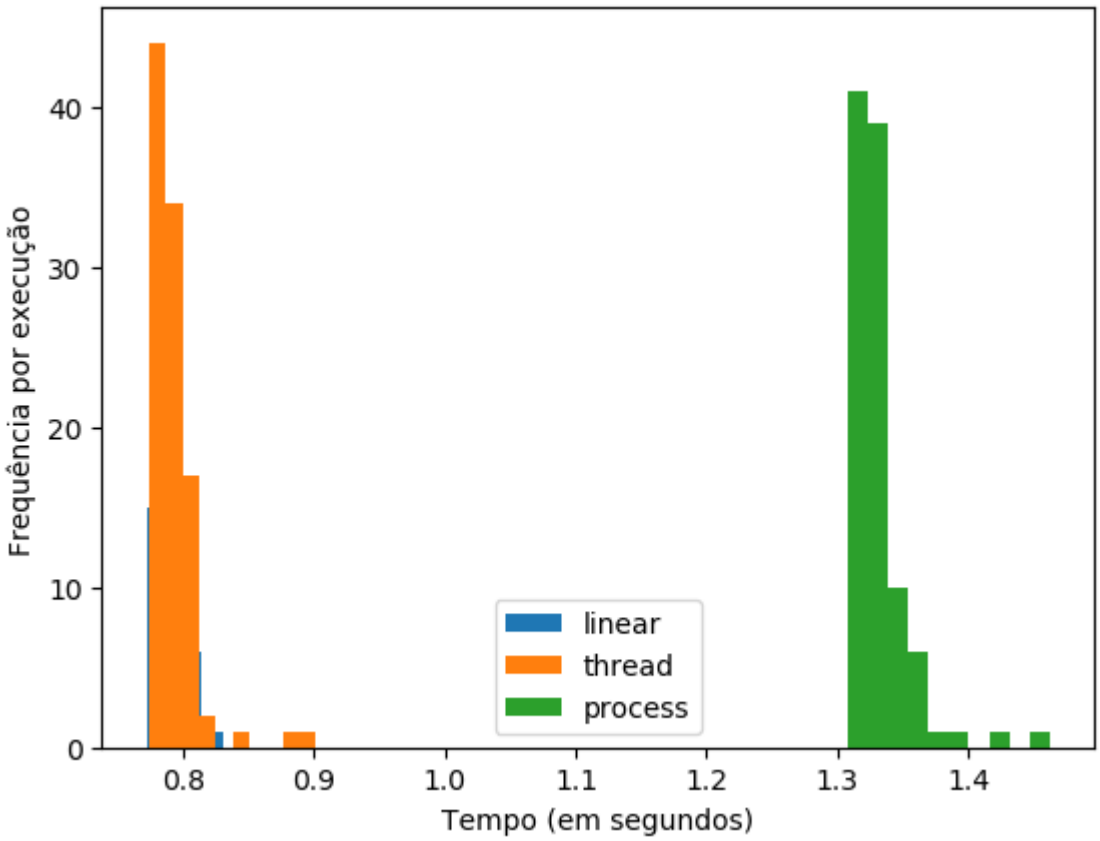


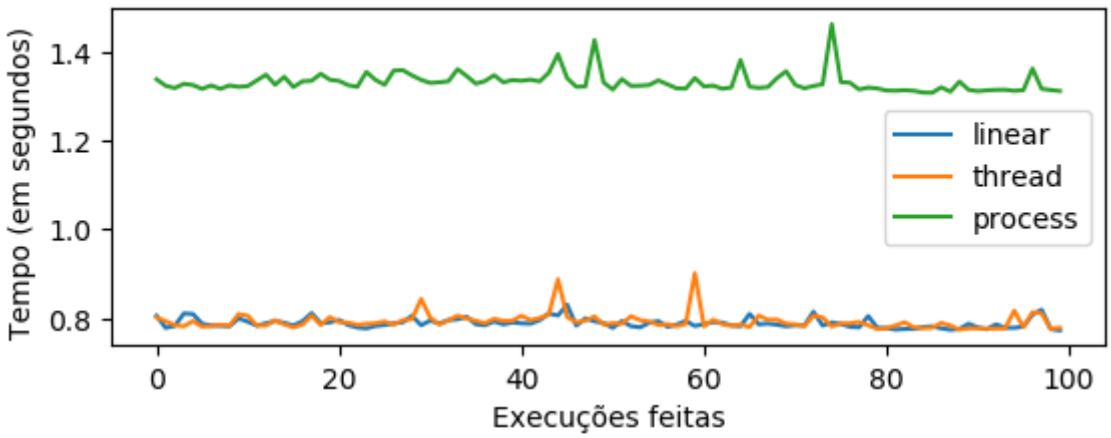
	Média (em segundos)	Desvio Padrão (em segundos)
thread	2.920	0.153
process	5.894	0.385
linear	8.601	0.662

Vimos então que provavelmente não era uma questão de quais são as entradas (tanto de imagem quando tamanho do N), a proporção entre o tempo do processamento da variação de multiprocessos com a da variação de multithreads permanece relativamente alta. Com isso em mente, resolvemos ir atrás dos possíveis motivos disso estar acontecendo, entre as hipóteses sobre o porquê disso estar acontecendo, uma delas era que sabendo que o tempo de troca de contexto (*context switching*) do multiprocessos é mais lento que o de threads essa diferença poderia estar causando a diferença no tempo total do processamento, para averiguá-la, testamos o processamento utilizando apenas um canal e depois usando dois canais (scripts para os testes de

um canal: `src/linear-1p.c` , `src/process-1p.c` , `src/thread-1p.c` ; scripts para os testes de dois canais: `src/linear-2p.c` , `src/process-2p.c` , `src/thread-2p.c`). Para os dois testes usamos `N = 5` e a imagem como `./data/cachorro.jpg` :

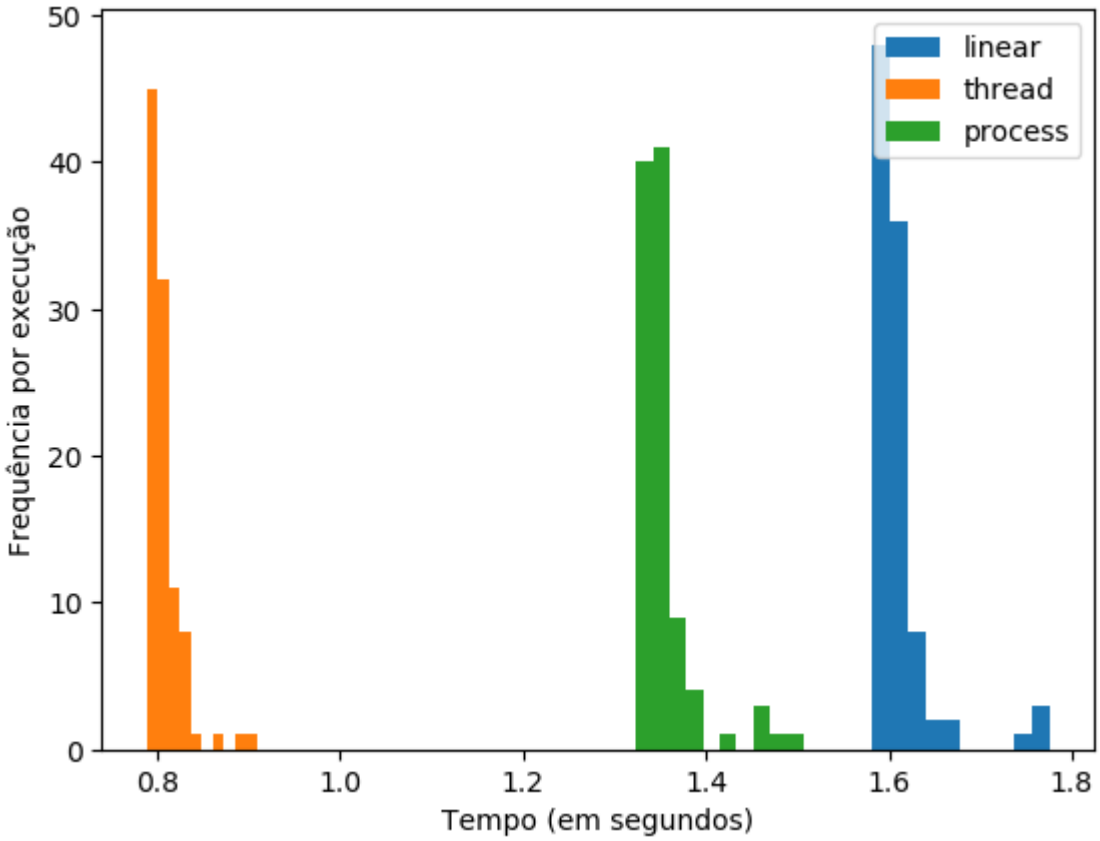
Um canal de Cor:

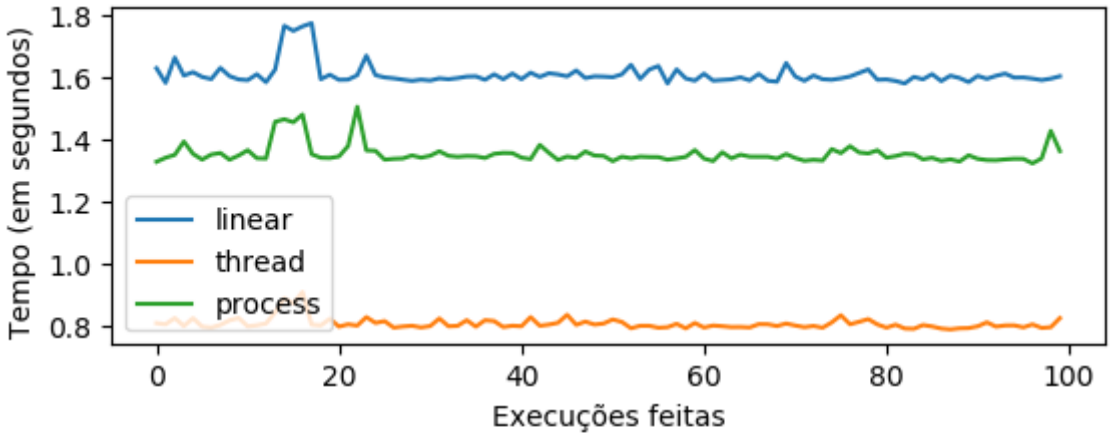




	Média (em segundos)	Desvio Padrão (em segundos)
thread	0.793	0.018
process	1.331	0.022
linear	0.789	0.011

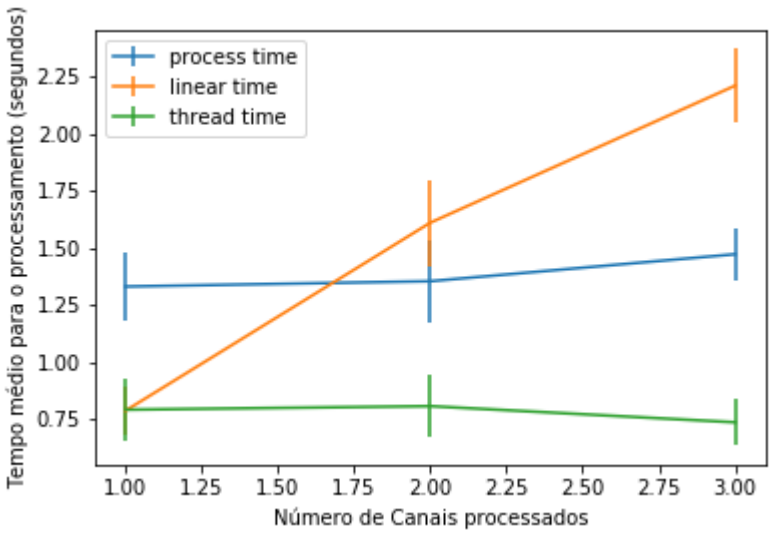
Dois canais de Cor:





	Média (em segundos)	Desvio Padrão (em segundos)
thread	0.808	0.018
process	1.355	0.031
linear	1.611	0.035

Fazendo a comparação entre o tempo médio do processamento de 1, 2 e 3 canais e considerando mostrando os erros em linhas verticais (considerando `erro = √(desvio padrão do tempo)`) temos:



Como pudemos observar, o tempo de execução não está mudando de forma significativa se mudarmos o números de processos/threads feitos ao mesmo tempo. Com isso em mente, vemos que o *context switch* não é o maior influenciador dessa alta diferença de tempo. Uma das susposicões que tivemos após pesquisas que fizemos sobre o desempenho dos processos relacionado ao desempenho das threads é que da forma que o código foi estruturado o tempo de criação e finalização dos processos está significativamente maior do que o das threads.