Rafael Molines Ismael

Igor Montagner

Supercomputação

20 September 2018

Paralelização com simulação 2D

Uma das técnicas mais utilizadas para aumentar o desempenho dos programas que rodam em nossos computador é a paralelização. Graças aos processadores com múltiplos núcleos, conseguimos dividir tarefas entre esses núcleos, melhorando muito o desempenho de nossos programas. Neste relatório, iremos observar em detalhe como se mostram esses ganhos com diferentes usos dos recursos presentes em nossos computadores.

No nosso caso, iremos comparar as estratégias sequencial, SIMD, paralela usando OpenMP e a junção das duas últimas. A estratégia sequencial é simplesmente o código sem usar nenhum dos recursos de ganho de desempenho, assim ele executaria um instrução de cada vez,uma depois da outra. Já a SIMD usa um tipo de memória especial que permite um ganho de desempenho para certos tipos de operações, em especial operações envolvendo vetores. A estratégia de paralelização irá dividir funções dentro do código que podem ser divididas entre as "threads" (pense nelas como se fossem os núcleos do computador) que nosso computador tiver e iremos utilizar a biblioteca OpenMP para C++. Já nossa última estratégia será a combinação das últimas duas, na qual prevemos que teremos o maior ganho de desempenho.

Para medir nosso desempenho, iremos rodar uma simulação 2D de colisões entre bolinhas. Nosso programa irá realizar uma série de cálculos para saber qual o próximo ponto em que cada bolinha estará. Iremos medir, então, o tempo que ele demora para terminar uma iteração dessa simulação, sendo que cada iteração irá calcular a próxima posição de todas as bolinhas presentes. Também iremos medir, por cada execução, o quanto de memória foi utilizada pelo programa.

TESTES

Antes que possamos analisar os desempenhos de cada estratégia, devemos primeiro gerar nossos dados para a análise. Para isso, basta rodar a seguinte linha de comando dentro do diretório deste arquivo:

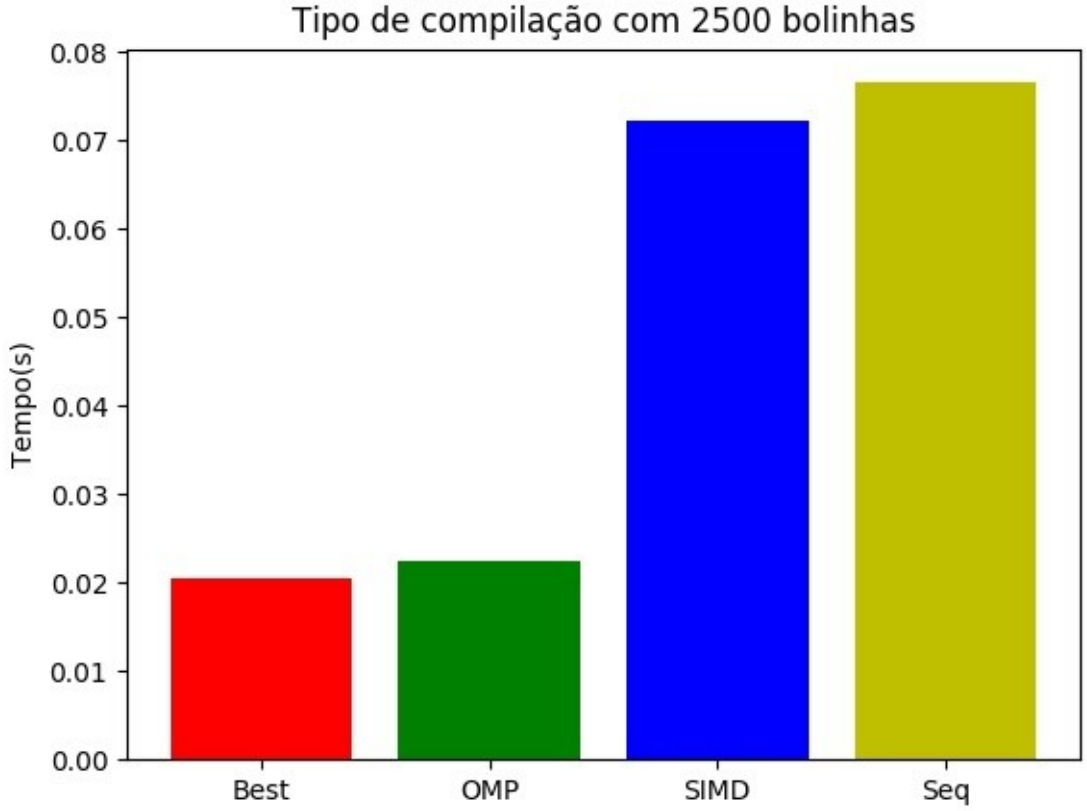
```
python ./visualizador/RODATUDO.PY -gui 2 -s results
```

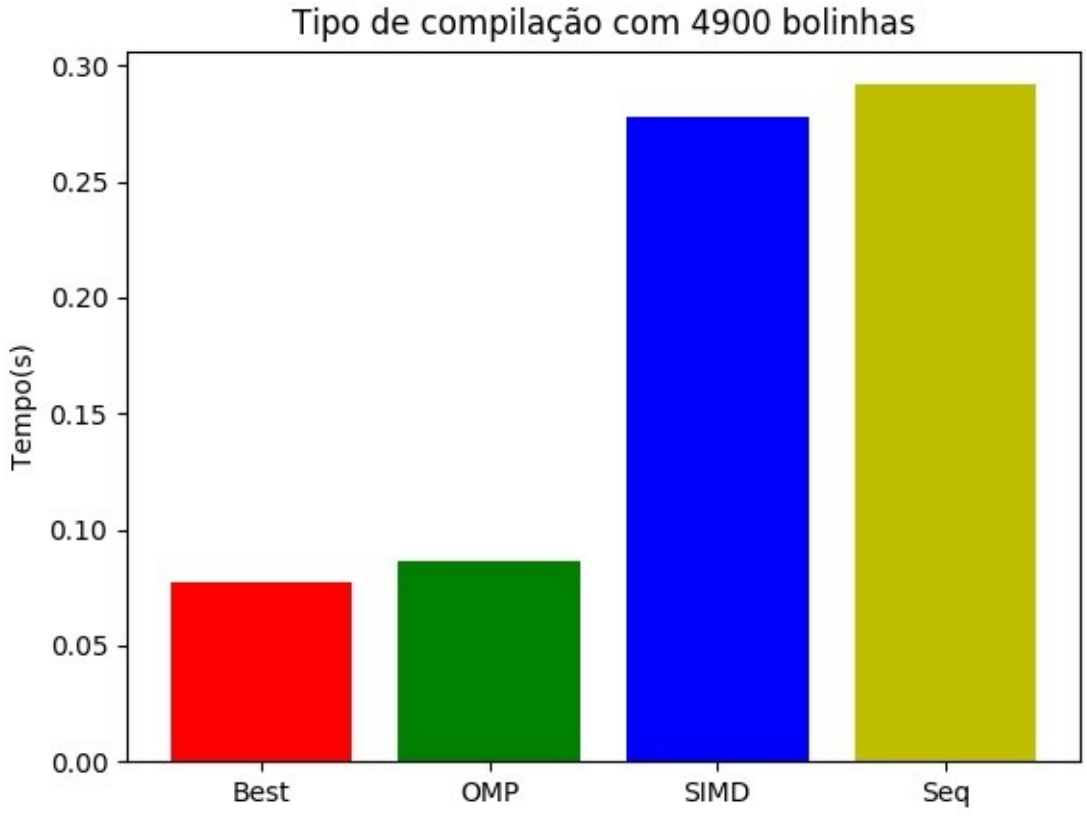
Iremos mais afundo sobre o que quer dizer cada um desses comandos na seção de documentação, mas por enquanto basta saber que esse programa python irá rodar uma série de simulações com as diferentes estratégias e com diferentes quantidades de bolinhas simuladas. Cada simulação irá computar 100 iterações, e irá contar quanto tempo cada uma delas demorou. Ao final, teremos a média de tempo de cada iteração. Como saída, este programa irá gerar uma quatro gráficos para cada tipo de estratégia utilizada, cada gráfico com um número diferente de bolinhas que estarão presentes na pasta "./visualizador/results", mostrando a média de

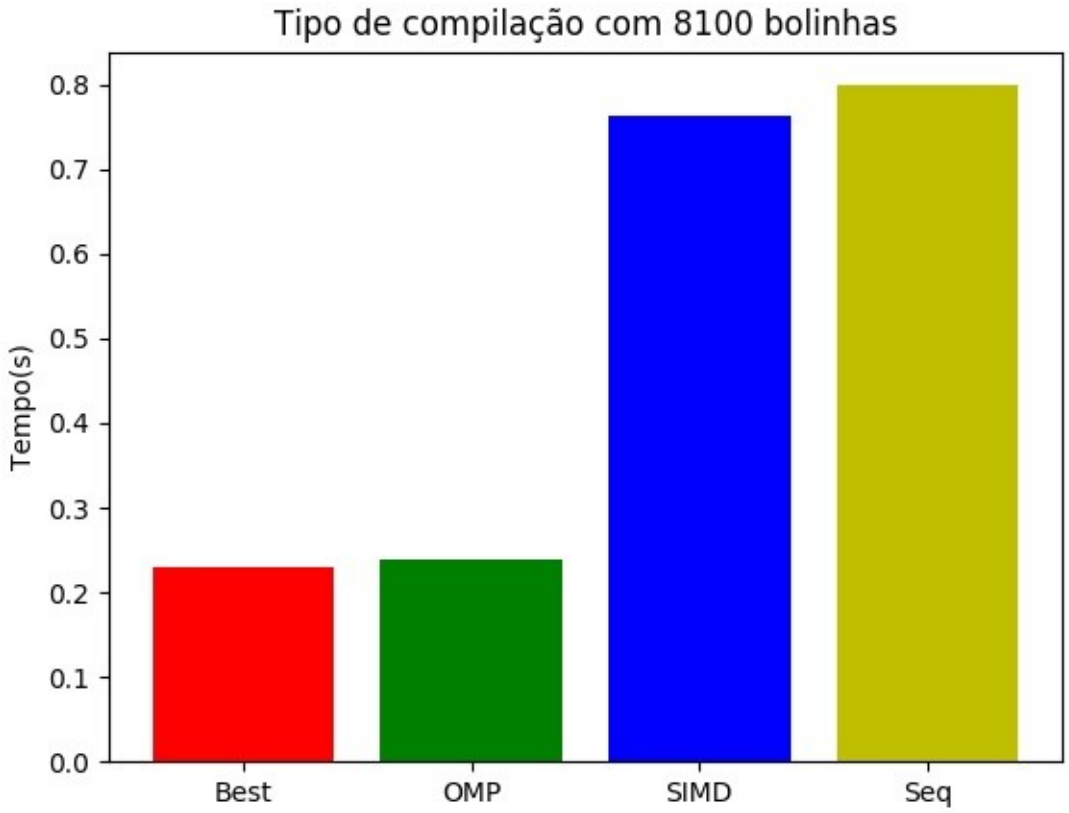
tempo. Ele também irá gerar uma série de gráficos mostrando o quanto de memória RAM foi utilizado por cada estratégia para as 100 iterações. Estes estarão na pasta "./visualizador/results-img".

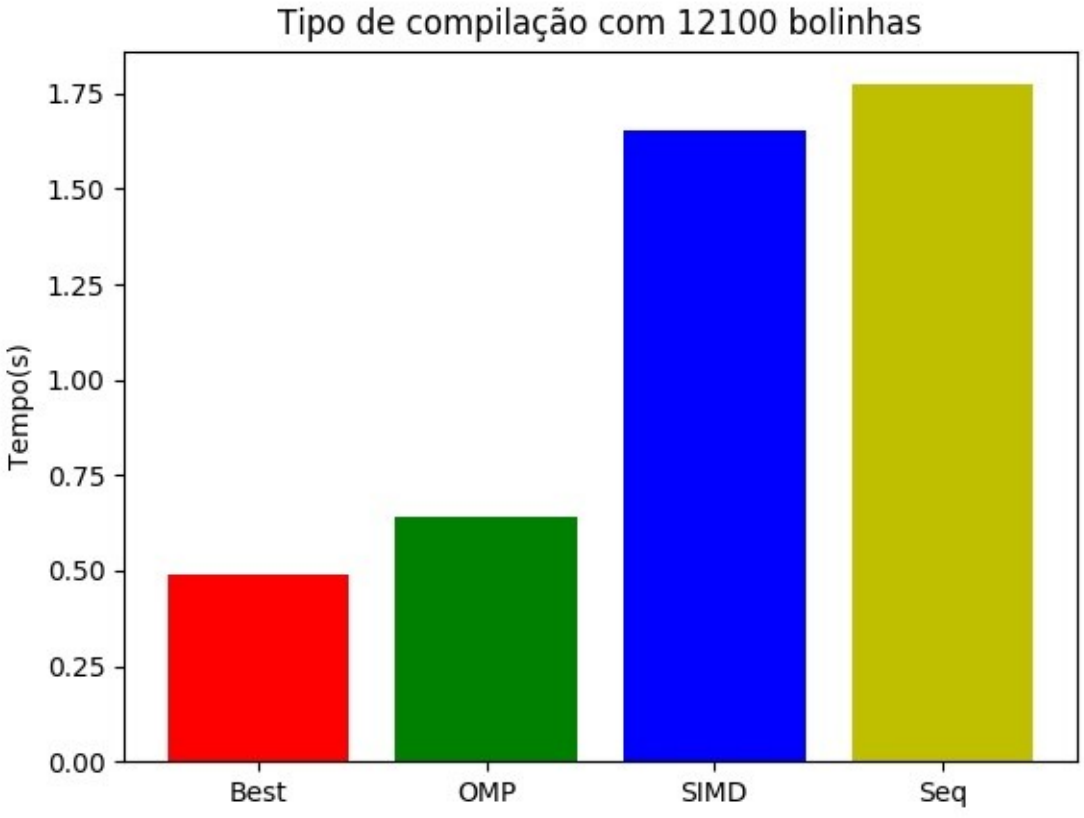
RESULTADOS

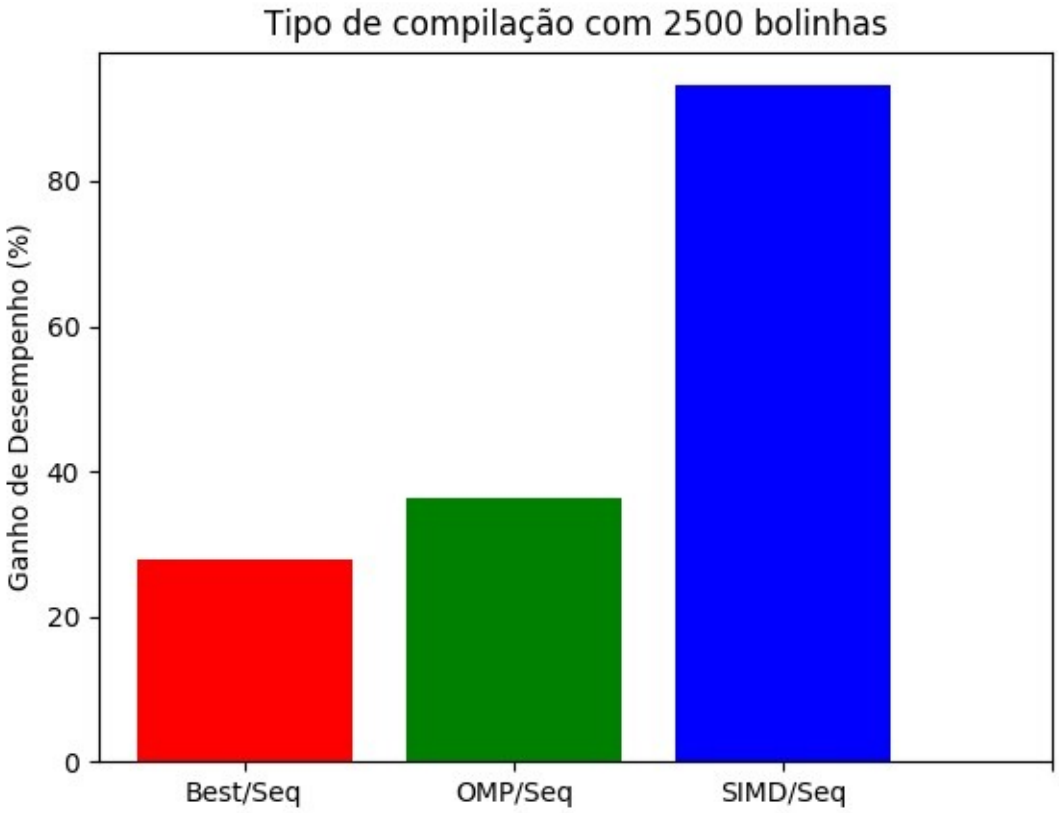
Podemos observar que as melhores estratégias em relação a ganho de desempenho são, na ordem, Best, OMP, SIMD e sequencial, o que era de fato esperado. A estratégia sequencial não havia nenhuma otimização de desempenho, logo foi a pior. A SIMD apenas melhorava algumas operações bem específicas que não eram as que mais consumiam memórias neste programa. A OMP conseguiu dividir a maioria das tarefas mais demoradas entre os 8 threads presentes no computador de teste, o que na teoria demoraria apenas 12,5% do tempo sequencial, muito próximo dos 15% alcançados. Já a Best apesar de ter sido a junção das últimas duas, conseguiu um tempo pouco melhor que do OMP, provavelmente por causa do fato de com a paralelização, não haver muitas possibilidades do uso da memória SIMD. Apesar do ganho de velocidade, não observamos nenhum ganho significativo em termos de consumo de memória. Abaixo, observamos os gráficos gerados pelo script python:

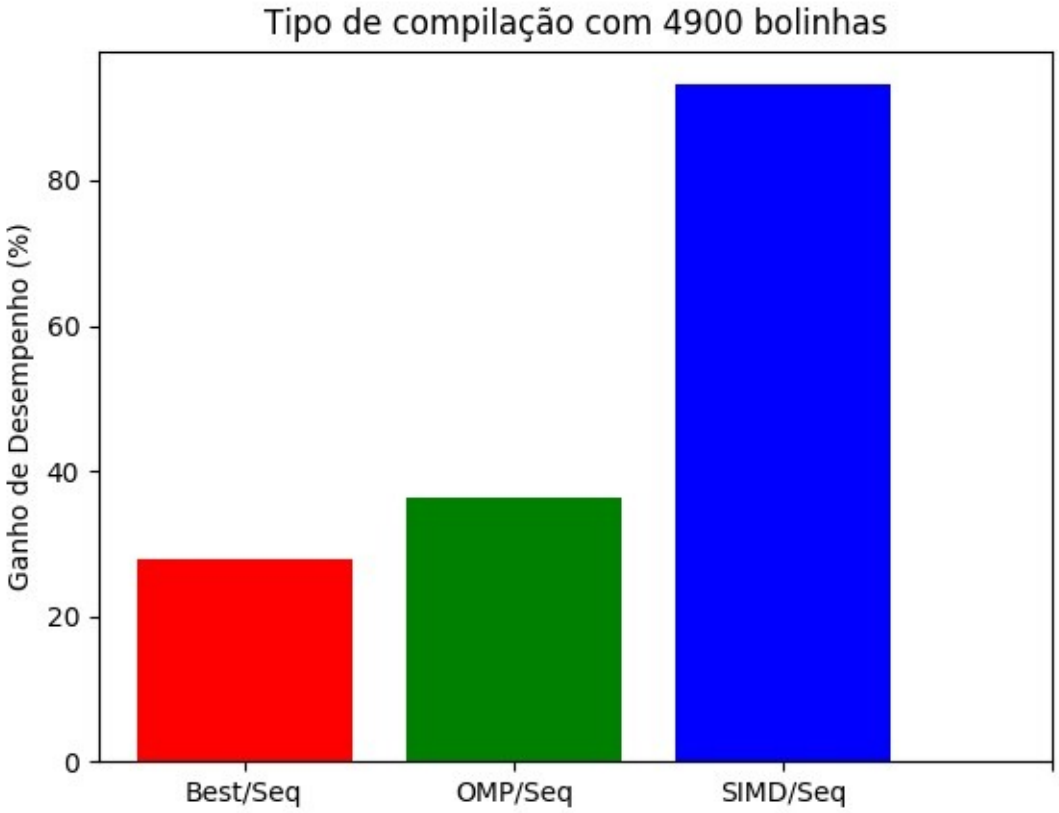

Tipo de compilação com 2500 bolinhas
Tempo(s)
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
Best
OMP
SIMD
Seq

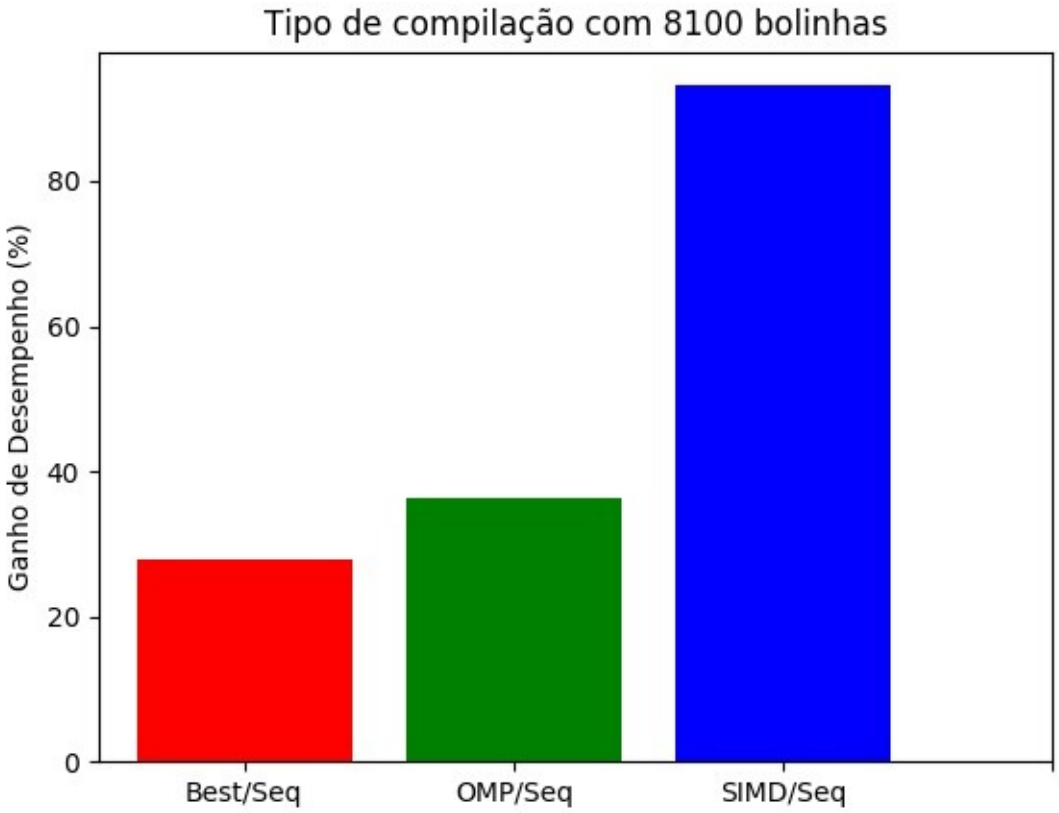

Tipo de compilação com 4900 bolinhas
Tempo(s)
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Best
OMP
SIMD
Seq

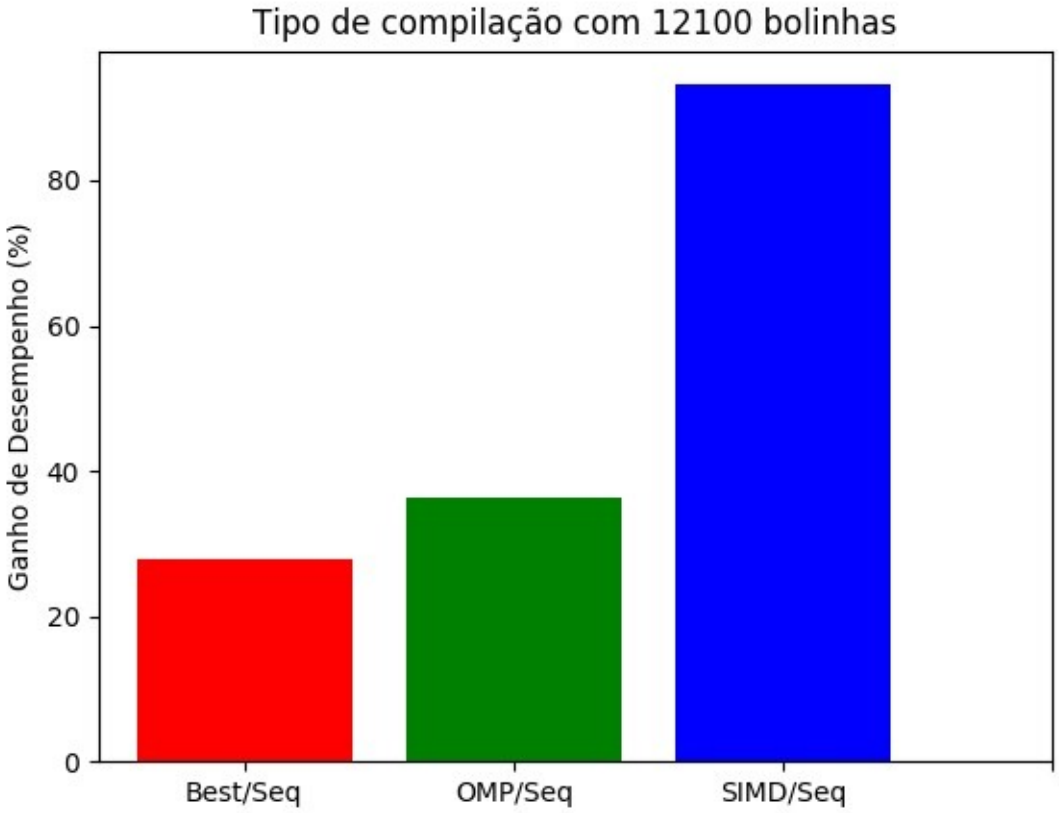

Tipo de compilação com 8100 bolinhas
Tempo(s)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Best
OMP
SIMD
Seq

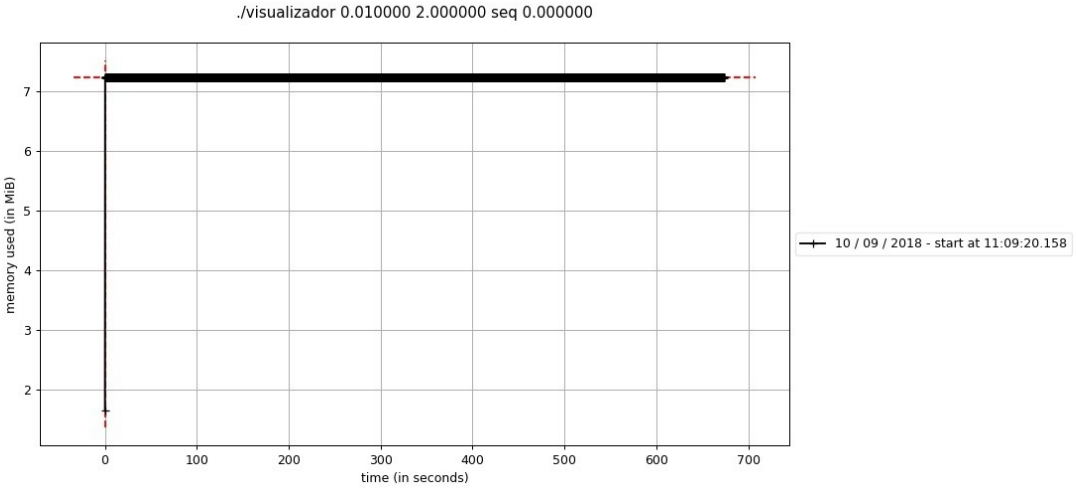

Tipo de compilação com 12100 bolinhas
Tempo(s)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
Best
OMP
SIMD
Seq

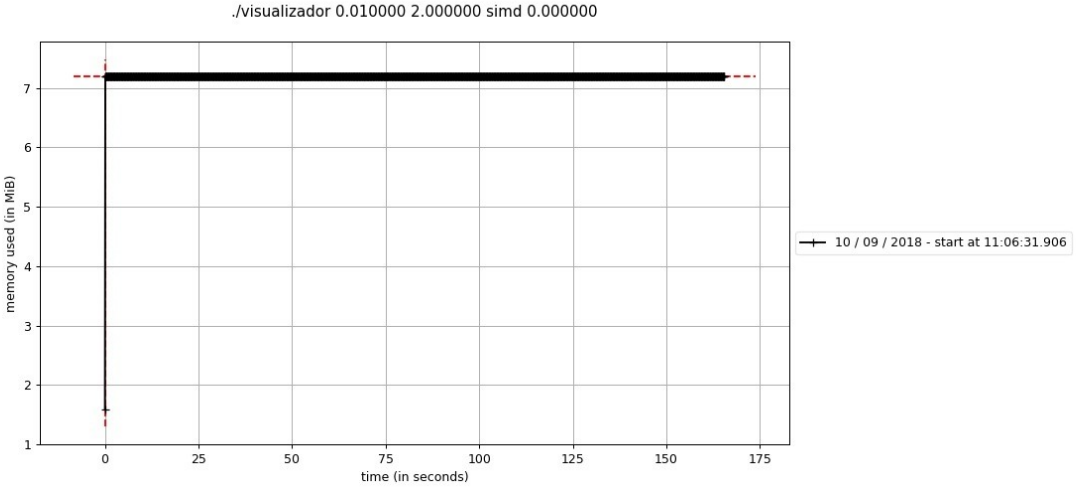

Tipo de compilação com 2500 bolinhas
Ganho de Desempenho (%)
0
20
40
60
80
Best/Seq
OMP/Seq
SIMD/Seq

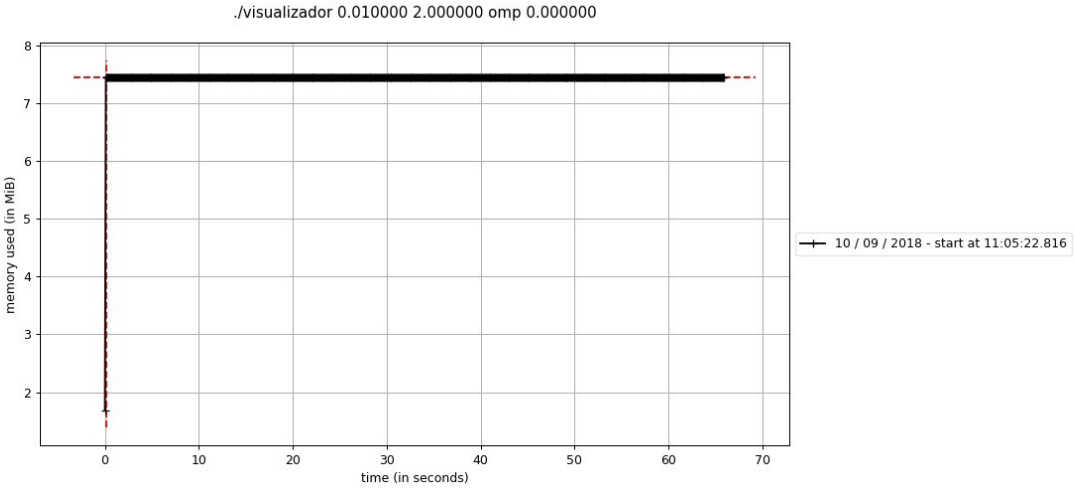

Tipo de compilação com 4900 bolinhas
Ganho de Desempenho (%)
0
20
40
60
80
Best/Seq
OMP/Seq
SIMD/Seq


Tipo de compilação com 8100 bolinhas
Ganho de Desempenho (%)
0
20
40
60
80
Best/Seq
OMP/Seq
SIMD/Seq


Tipo de compilação com 12100 bolinhas
Ganho de Desempenho (%)
0
20
40
60
80
Best/Seq
OMP/Seq
SIMD/Seq


./visualizador 0.010000 2.000000 seq 0.000000
memory used (in MiB)
time (in seconds)
10 / 09 / 2018 - start at 11:09:20.158


./visualizador 0.010000 2.000000 simd 0.000000
memory used (in MiB)
time (in seconds)
10 / 09 / 2018 - start at 11:06:31.906



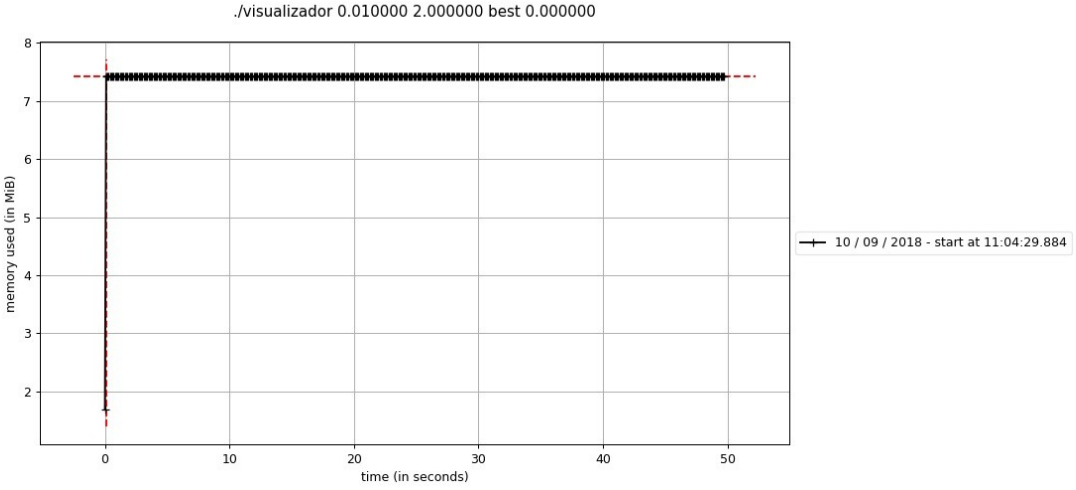


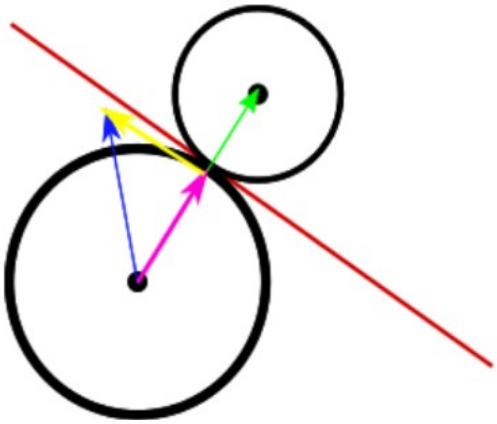
Fig 2. - Os quatro gráficos acima mostram o consumo de memória da

simulação com 12100 bolinhas, sendo a ordem seq, simd, omp e best das estratégias utilizadas em cada gráfico.

IMPLEMENTAÇÃO

O hardware utilizado para a implementação foi um laptop com Ubuntu 18.04LTS nativo, 16GB de memória RAM e um processador Intel 4720HQ, de 8 threads. O código foi escrito em C++ e o script que executa o código em Python.

Para a simulação foram usados dois modelos, o de colisão geométrica e o de colisão elástica. O segundo foi implementado, porém por apresentar muitos bugs decidiu-se por utilizar somente o primeiro para os testes.



No modelo geométrico, considera-se simplesmente que que cada colisão entre as bolinhas fosse como uma bolinha batesse em uma parede paralela à reta tangente à colisão. Assim, não há troca de energia entre as bolinhas.

O primeiro passo da simulação é atualizar as velocidades de todas as bolinhas em relação a desaceleração causada pelo atrito dinâmico. Para isso, subtrai-se o atrito da velocidade, e depois a decompõe em x e y e atualiza estes atributos de cada bolinha. Depois, procura-se por todas as bolinhas que irão colidir na próxima iteração, prevendo tal com a velocidade atual de cada uma. Para cada par de bolinhas

que colidir, calcula-se o novo vetor velocidade após a colisão, e então, mais uma vez, atualiza-se a velocidade de cada bolinha. Após isso, procura-se por colisões entre todas as bolinhas e as paredes do campo. Caso haja tal colisão, se inverte a componente da velocidade perpendicular à parede colidida. O próximo passo é então, com todas as velocidades atualizadas, calcular a próxima posição de cada bolinha. Repete.

O modelo de colisão elástica foi feito a partir das fórmulas desenvolvidas neste link.

Toda a paralelização foi feita apenas em loops for, com a função "#pragma omp for parallel". Devido à natureza do programa, este passo foi bem simples por todos os cálculos poderem ser feitos independente um dos outros.

A estratégia SIMD foi feita apenas utilizando a auto-vetorização como opção de compilação com as seguintes flags "-ffast-math -ftree-vectorize -mavx2 -O2".

DOCUMENTAÇÃO

RODATUDO.PY - script que roda todos os programas necessàrios para compilar a simulação

Através dele é possível passar os seguintes argumentos:

-s (define a estratégia utilizada)

default: best

-gui (define entre mostrar o visualizador das colisões,1, apendar

printar as posições e choques na tela,0)

default: 1

-deltat (define o tempo entre uma iteração e outra)

default: 0.01

-model (define entre o modelo geométrico, 0, ou o modelo elástico, 1)

default: 0

-nballs (define o número de bolinhas para simular, devido à detalhes de implementação, o número real de bolinhas será sempre o quadrado de nballs)

default: 10

Create_input.py - script utilizado por RODATUDO.PY para criar a entrada da simulação

Possui três argumentos:

-test (gera uma entrada de testes padrão)

-n (gera uma entrada padronizada com espaçamento e raio igual para todas as bolinhas

-r (gera uma entrada com bolinhas totalmente aleatorizadas

Para todos os casos, é necessário entrar com um número de bolinhas desejado na simulação.

Também será preciso instalar as bibliotecas python matplotlib e memory_profiler.