UNIVERSIDADE FEDERAL DO PARANÁ - UFPR
GUSTAVO SILVEIRA FREHSE
VINÍCIUS MAURÍCIO RIBEIRO

RELATÓRIO DE AVALIAÇÃO DE PERFORMANCE E OTIMIZAÇÃO DO MÉTODO DE NEWTON PARA ENCONTRAR PONTOS CRÍTICOS DA FUNÇÃO DE ROSENBROCK

CURITIBA
2022

GUSTAVO SILVEIRA FREHSE
VINÍCIUS MAURÍCIO RIBEIRO

RELATÓRIO DE AVALIAÇÃO DE PERFORMANCE E OTIMIZAÇÃO DO MÉTODO DE NEWTON PARA ENCONTRAR PONTOS CRÍTICOS DA FUNÇÃO DE ROSENBROCK

Trabalho apresentado
à disciplina de Introdução
à Computação Científica
Do Curso Bacharelado em
Ciência da Computação
Da Universidade Federal
do Paraná - UFPR

Prof. Armando Luiz Nicolini Delgado

CURITIBA
2022

SUMÁRIO

INTRODUÇÃO

Este trabalho tem como objetivo avaliar a performance e encontrar otimizações para dois algoritmos que visam encontrar pontos críticos de funções utilizando o método de Newton. Apresenta também experimentos realizados e gráficos com os resultados obtidos.

CONFIGURAÇÃO DO EXPERIMENTO

2.1 MÁQUINA UTILIZADA

A máquina utilizada no experimento faz parte do laboratório do DINF da UFPR. O professor disponibilizou o computador h31 para ser acessado remotamente.

As especificações da máquina, extraídas pela ferramenta LIKWID, estão no trecho abaixo.

```
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
CPU type:       Intel Coffeelake processor
CPU stepping:   9
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                1
Cores per socket:       4
Threads per core:       1
--------------------------------------------------------------------------------
HWThread        Thread          Core            Socket          Available
0               0               0               0               *
1               0               1               0               *
2               0               2               0               *
3               0               3               0               *
--------------------------------------------------------------------------------
Socket 0:               ( 0 1 2 3 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level:                  1
Size:                   32 kB
Type:                   Data cache
Associativity:          8
Number of sets:         64
Cache line size:        64
Cache type:             Non Inclusive
Shared by threads:      1
Cache groups:           ( 0 ) ( 1 ) ( 2 ) ( 3 )
--------------------------------------------------------------------------------
Level:                  2
Size:                   256 kB
Type:                   Unified cache
Associativity:          4
Number of sets:         1024
Cache line size:        64
Cache type:             Non Inclusive
```

```
Shared by threads:      1
Cache groups:           ( 0 ) ( 1 ) ( 2 ) ( 3 )
--------------------------------------------------------------------------------
Level:                  3
Size:                   6 MB
Type:                   Unified cache
Associativity:          12
Number of sets:         8192
Cache line size:        64
Cache type:             Inclusive
Shared by threads:      4
Cache groups:           ( 0 1 2 3 )
--------------------------------------------------------------------------------
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:           1
--------------------------------------------------------------------------------
Domain:                 0
Processors:             ( 0 1 2 3 )
Distances:              10
Free memory:            5230.84 MB
Total memory:           7863.17 MB
--------------------------------------------------------------------------------


********************************************************************************
Graphical Topology
********************************************************************************
Socket 0:
+---------------------------------------------+
| +---------+ +---------+ +---------+ +---------+ |
| |    0    | |    1    | |    2    | |    3    | |
| +---------+ +---------+ +---------+ +---------+ |
| +---------+ +---------+ +---------+ +---------+ |
| |  32 kB  | |  32 kB  | |  32 kB  | |  32 kB  | |
| +---------+ +---------+ +---------+ +---------+ |
| +---------+ +---------+ +---------+ +---------+ |
| | 256 kB  | | 256 kB  | | 256 kB  | | 256 kB  | |
| +---------+ +---------+ +---------+ +---------+ |
| +-----------------------------------------+ |
| |                   6 MB                  | |
| +-----------------------------------------+ |
+---------------------------------------------+
```

2.2 EXECUÇÃO DO EXPERIMENTO

O presente trabalho (Trabalho 2) é baseado no Trabalho 1, onde foram previamente implementados os métodos Newton Padrão, Newton Inexato e Newton Modificado.

Seguindo as orientações do professor, o trabalho inicial foi modificado especificamente para utilizar a função de Rosenbrock. Isso implicou em não utilizar a biblioteca *libmatheval* para o cálculo de funções “genéricas”, mas sim funções especializadas para calcular derivadas parciais de primeira e segunda ordem da função de Rosenbrock. A essa versão modificada do Trabalho 1 deu-se o nome T1 Linha.

O trabalho atual, denominado T2, é resultado de modificações no T1 Linha para torná-lo mais otimizado.

Seguindo essa lógica, a estrutura do projeto utilizado para executar o experimento está representada da seguinte forma:

```
./
├── gera_csv.py
├── gera_rosenbrock.sh
├── gera_rosenbrock_mod.sh
├── grafico.gnuplot
├── Makefile
├── t1/
├── t1_linha/
└── t2/
```

gera_csv.py: gera CSVs dos gráficos utilizados para análise de performance do T2 em relação ao T1 Linha;

gera_rosenbrock.sh: arquivo disponibilizado pelo professor. Gera as entradas para o programa principal.

gera_rosenbrock_mod.sh: Gera as entradas para o programa principal com dimensão específica.

grafico.gnuplot: contém configurações para a geração dos gráficos;

Makefile: gera o executável das diferentes versões dos trabalhos, com opções de *profiling* pelas ferramentas `gprof` e `likwid`.

t1: pasta com os arquivos fonte do Trabalho 1.

t1_linha: pasta com os arquivos fonte do T1 Linha.

t2: pasta com os arquivos fonte do trabalho atual.

Abaixo estão listas opções para compilação e execução das versões dos trabalhos:

- Para rodar o T1 Linha:

```
$ make t1-linha
$ ./gera_rosenbrock.sh | ./newtonPC_t1_linha
```

- Para rodar o Trabalho 2:

```
$ make t2
$ ./gera_rosenbrock.sh | ./newtonPC
```

- Para rodar com *profiling* (gprof):

```
$ make clean
$ echo performance > /sys/devices/system/cpu/policy3/scaling_governor
$ make t1-linha-profiling
$ ./gera_rosenbrock.sh | ./newtonPC_t1_linha
$ gprof newtonPC_t1_linha
$ make t2-profiling
$ ./gera_rosenbrock.sh | ./newtonPC
$ gprof newtonPC_t1_linha
$ echo powersave > /sys/devices/system/cpu/policy3/scaling_governor
```

- Para rodar com o *profiling* (likwid):

```
$ make clean
$ make t1-linha-likwid
$ make t2-likwid
$ echo performance > /sys/devices/system/cpu/policy3/scaling_governor
$ ./gera_rosenbrock.sh | likwid-perfctr -C 3 -g <grupo> -m  ./newtonPC_t1_linha
$ ./gera_rosenbrock.sh | likwid-perfctr -C 3 -g <grupo> -m  ./newtonPC
$ echo powersave > /sys/devices/system/cpu/policy3/scaling_governor
```

- Para gerar os gráficos:

```
$ ./gera_csv.py newtonPC_t1_linha newtonPC
$ gnuplot -p ./graficos.gnupl
```

OTIMIZAÇÕES

3.1. TRABALHO INICIAL

Com ajuda da ferramenta `gprof` foi observado que cerca de 92% do tempo do programa foi utilizado em apenas duas funções: `triangularizar()` e `resolver_sl_gauss_seidel()`:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 57.56      2.45     2.45      459     5.34     5.36  triangularizar
 35.71      3.97     1.52      459     3.31     3.35  resolver_sl_gauss_seidel
  2.11      4.06     0.09      934     0.10     0.10  copiar_matriz_double
  1.17      4.11     0.05 20824444     0.00     0.00  rosenbrock_dxdy
```

Com esse feedback, resolvemos começar a otimizar essas duas funções. Começamos tentando usar as flags de compilação `-ftree-vectorize` e `-fopt-info-vec-all` . Mesmo modificando o código e verificando que realmente houve a vetorização, os resultados não refletiram nosso esforço e assim, partimos para tentar outro modo de otimização.

3.2 TRIDIAGONALIDADE

Notamos, em um momento, que as matrizes hessianas utilizadas eram matrizes tridiagonais. Não sabíamos, porém, se eram realmente todas ou apenas uma coincidência. Para verificarmos fizemos a seguinte prova:

Seja R a função de Rosenbrock:

$$R(\vec{x}) = \sum_{i=1}^{N-1} \left(100\left(x_{i+1} - x_i^2\right)^2 + (1 - x_i)^2\right)$$

Derivada parcial de R(x) em relação à x_a, onde $a \in \{1..N\}$

$$\frac{\partial}{\partial x_a}R(\vec{x}) = \frac{\partial}{\partial x_a}\left(100\left(x_a - x_{a-1}^2\right)^2 + (1-x_{a-1})^2\right) + \frac{\partial}{\partial x_a}(100\left(x_{a+1} - x_a^2\right)^2 + (1-x_a)^2$$
$$+\frac{\partial}{\partial x_a}\sum_{i=1}^{a-2}\left(100\left(x_{i+1} - x_i^2\right)^2 + (1-x_i)^2\right) + \frac{\partial}{\partial x_a}\sum_{i=a+1}^{N-1}\left(100\left(x_{i+1} - x_i^2\right)^2 + (1-x_i)^2\right)$$

$$\iff \frac{\partial}{\partial x_a}\left(100\left(x_a - x_{a-1}^2\right)^2 + (1-x_{a-1})^2\right) + \frac{\partial}{\partial x_a}(100\left(x_{a+1} - x_a^2\right)^2 + (1-x_a)^2$$

$$\iff 200\left(x_a - x_{a-1}^2\right) - 400x_a\left(x_{a+1} - x_a^2\right) - 2(1-x_a)$$

Derivada parcial de segunda ordem em relação à $x_a, x_b;\ a, b\ \in \{1..N\}$

$$\frac{\partial^2}{\partial x_a \partial x_b}R(\vec{x}) = \frac{\partial}{\partial x_b}\left(200\left(x_a - x_{a-1}^2\right) - 400x_a\left(x_{a+1} - x_a^2\right) - 2(1-x_a)\right)$$

Se $x_b = x_a$

$$\frac{\partial^2}{\partial x_a^2}R(\vec{x}) = 1200x_a^2 - 400x_{a+1} + 202$$

Se $x_b = x_{a-1}$

$$\frac{\partial^2}{\partial x_a \partial x_{a-1}}R(\vec{x}) = -400x_{a-1}$$

Se $x_b = x_{a+1}$

$$\frac{\partial^2}{\partial x_a \partial x_{a+1}}R(\vec{x}) = -400x_a$$

Demais x_b

$$\frac{\partial^2}{\partial x_a \partial x_b} R(\vec{x}) = 0$$

Note que a derivada parcial de R(x) depende apenas de x_a, x_(a+1) e x_(a-1). Portanto, ao realizar a derivada de segunda ordem, apenas os casos mostrados na prova acima não tem resultado igual a zero. Isso implica que, dada a matriz hessiana H de dimensões N x N, apenas os índices H[a][a], H[a][a+1] e H[a][a-1] têm algum valor. Como H[a][a] corresponde à diagonal principal e H[a][a+1] e H[a][a-1] correspondem às diagonais adjacentes, conclui-se que a matriz hessiana é tridiagonal.

3.3 PRÁTICAS EFETUADAS

Sabendo que a hessiana da função de Rosenbrock é sempre tridiagonal, podemos fazer diversas otimizações. A primeira é no *layout* de memória usado para guardar as matrizes. Com esse novo *layout* é possível, por sua vez, otimizar tanto o `triangularizar()` quanto o `resolver_sl_gauss_seidel()`.

No novo *layout* de memória, uma matriz consiste de três vetores, um com tamanho N e outros dois com tamanho N-1. Esses vetores representam as diagonais da matriz. Assim o espaço de memória vai de $O(n \times n)$ para $O(n)$.

No algoritmo de triangularização, é possível fazer mudanças que tornam o algoritmo de $O(n \times n \times n)$ para $O(n)$.

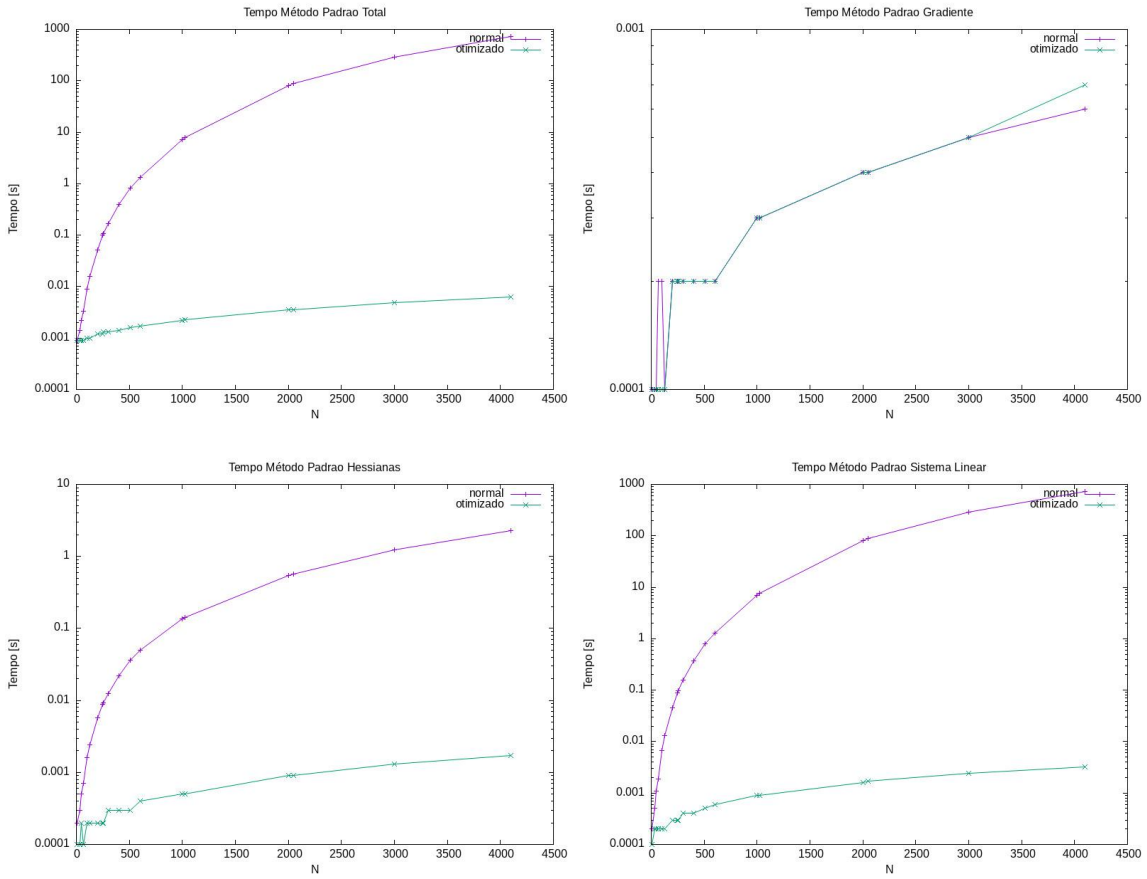
Já no algoritmo de Gauss-Seidel, as mudanças permitem o tempo ir de $O(maxiters \times n \times n)$ para $O(maxiters \times n)$.

Obs: Os algoritmos utilizados para trabalhar com o novo *layout* são os mesmos apresentados pelo professor em aulas anteriores sobre matrizes tridiagonais.

GRÁFICOS

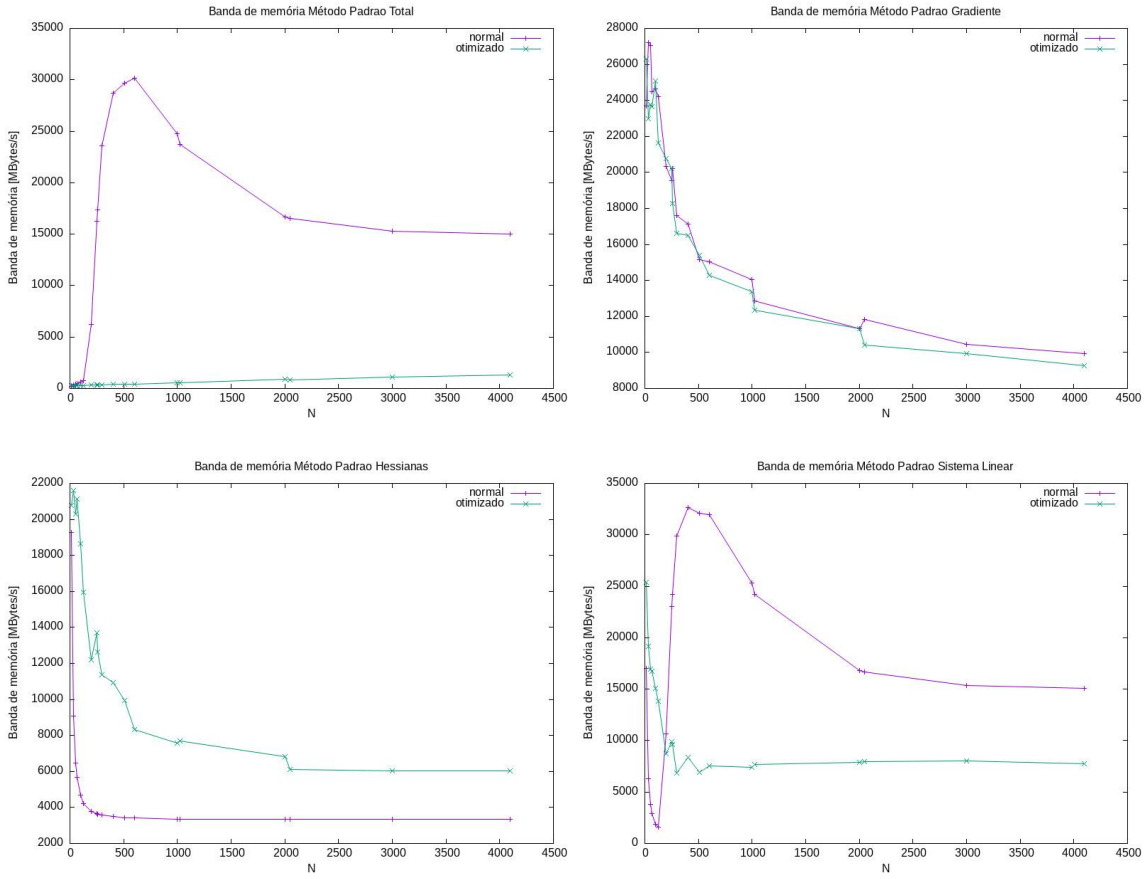
4.1 NEWTON PADRÃO

- Tempo



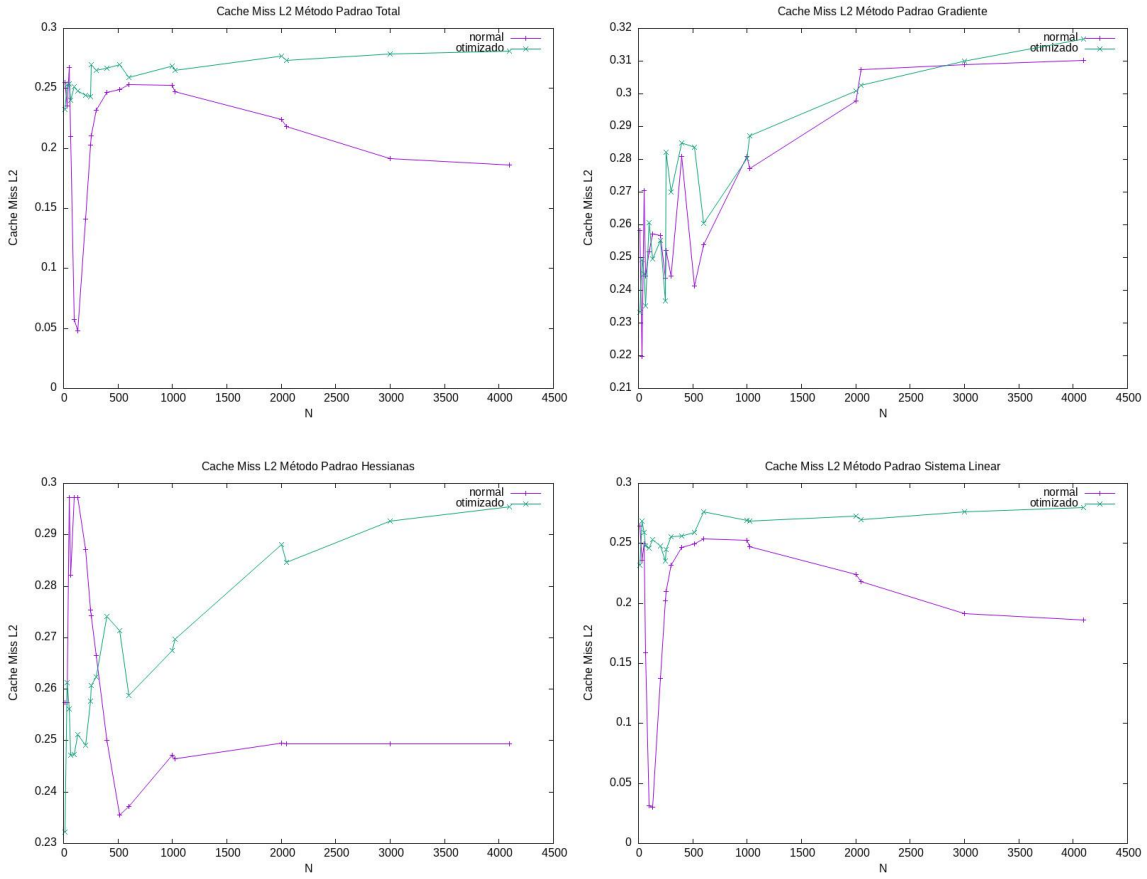
Houve uma melhora no tempo. É possível observar as melhoras tanto no trecho Hessianas quanto no trecho do Sistema Linear. O mesmo não pode ser dito para o cálculo do Gradiente, que não se beneficiou do novo *layout* da matriz.

- Banda da memória



No total, o método antigo tem uma banda de memória significativamente maior que o método otimizado. Nos gráficos é possível ver que o culpado é o trecho do Sistema Linear, que, apesar de ter menos Banda de Memória no começo, explode em aproximadamente 500 dimensões. Isso provavelmente se deve a um *cache trashing* ocorrendo em 512 dimensões.

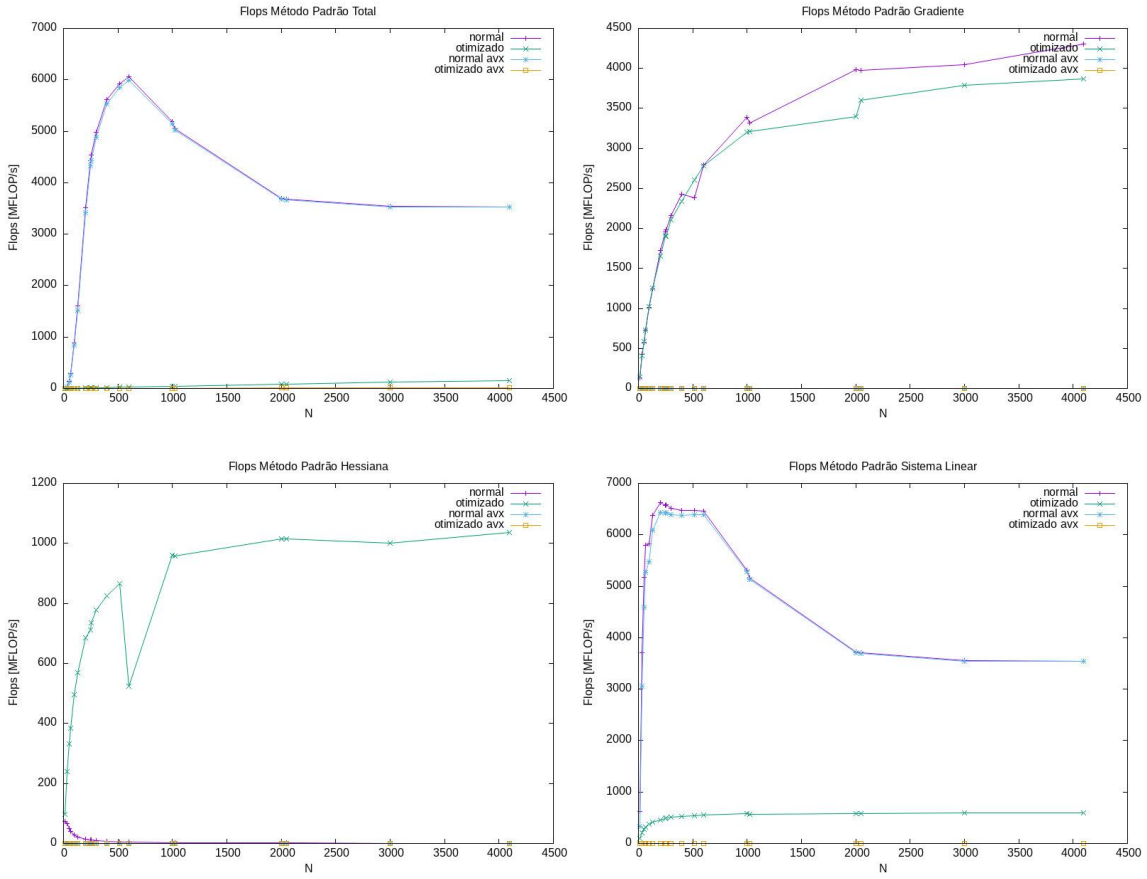
- Cache miss L2



O método otimizado tem mais *cache miss ratio* que o método original. Isso se deve a dois fatores:

1. Existem bem menos acessos à memória na versão otimizada, fazendo com que cada *cache miss* tenha mais impacto na *ratio*.
2. No trecho Hessianas, “caminhamos” pela matriz primeiro pelas linhas e depois pelas colunas. Considerando que cabe apenas um vetor na *cache* de uma vez, no método original trocamos a *cache* apenas quando trocamos de linha, contrastando com o método otimizado que a cada novo acesso terá que trocar a *cache*. O trecho Sistema Linear sofre de um problema parecido.

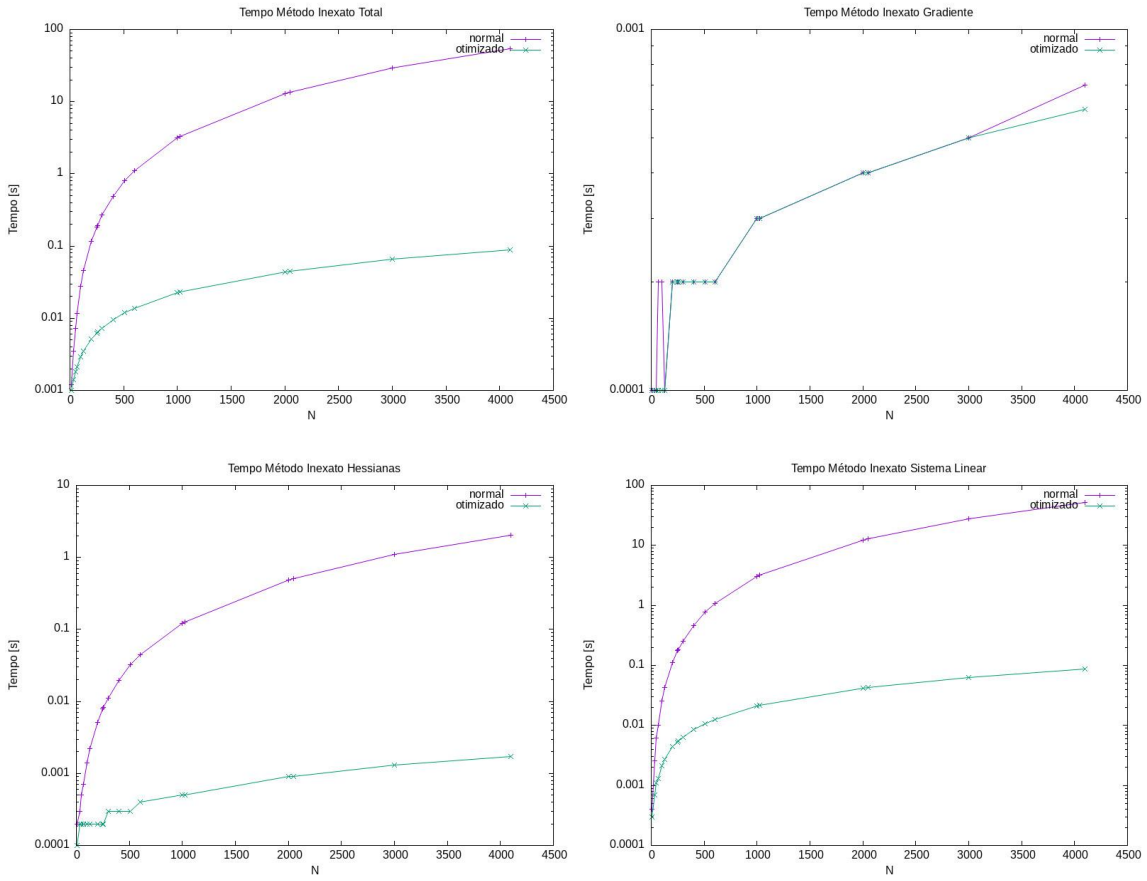
- Operações aritméticas



O método anterior, no trecho Total, produz mais FLOPS que o otimizado. Isso provavelmente se dá pois há muitos mais elementos que precisamos mudar no sistema linear (n² vs. n)

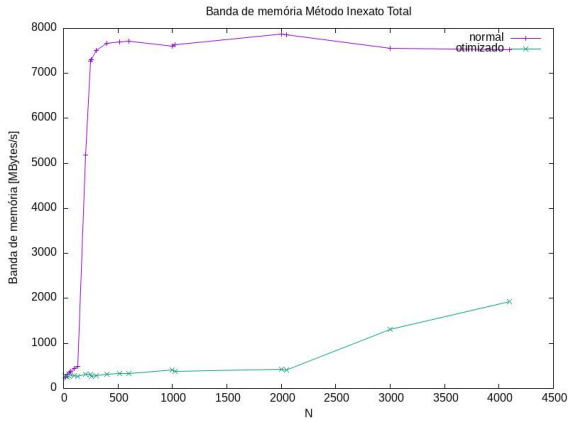
4.2 NEWTON INEXATO

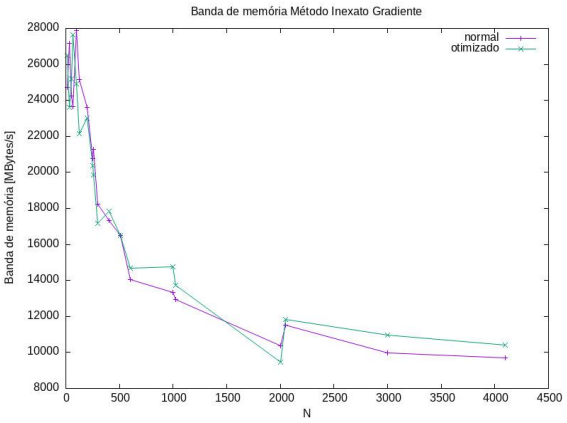
- Tempo

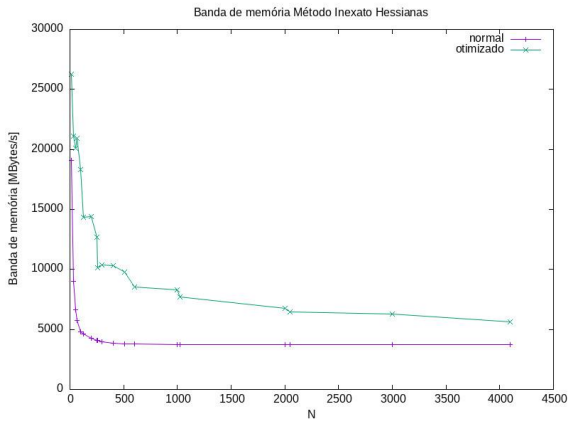


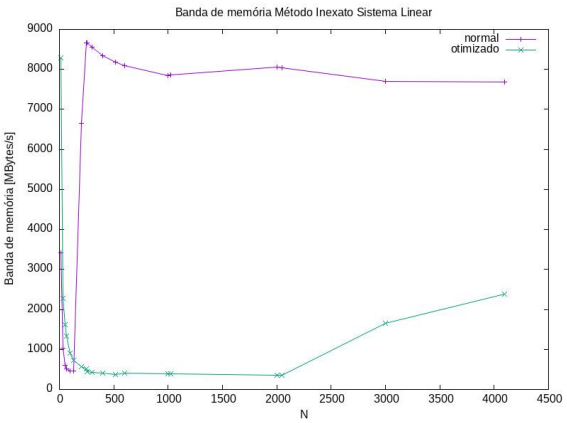
Semelhante ao método padrão.

- Banda de memória



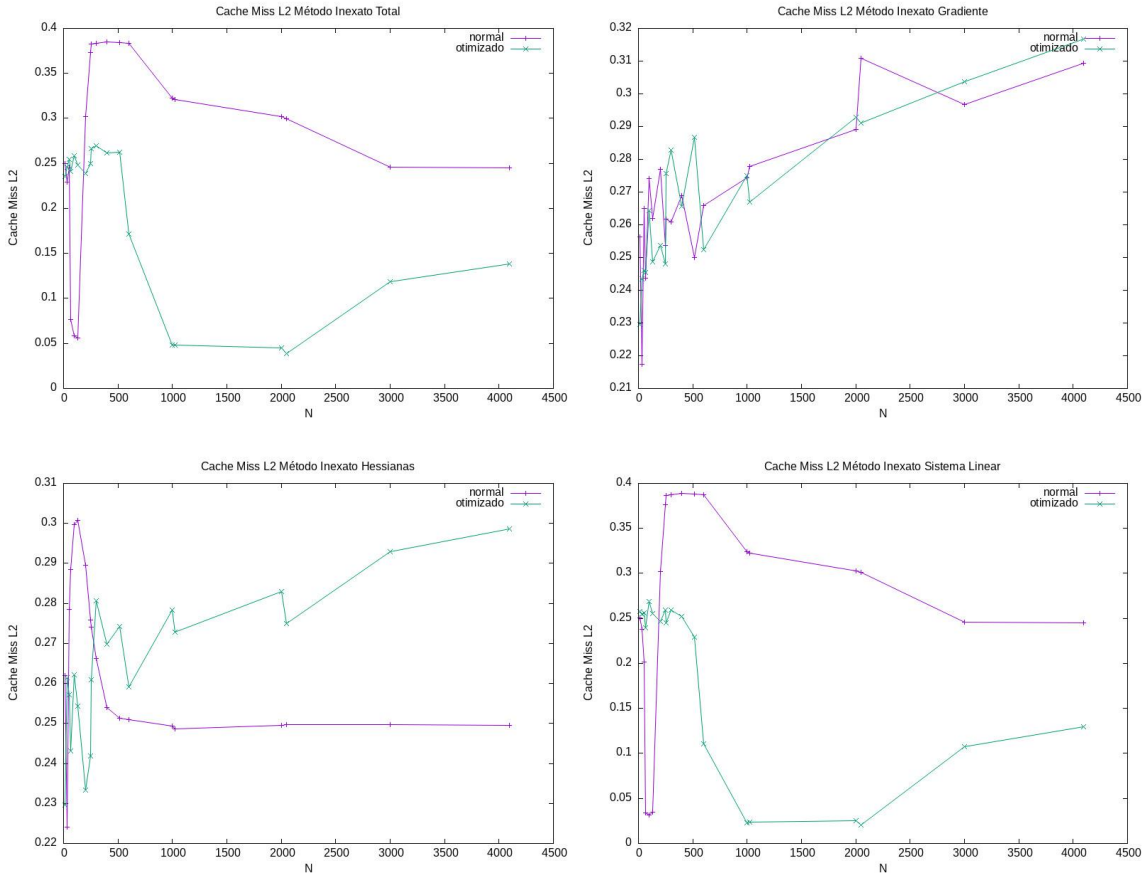






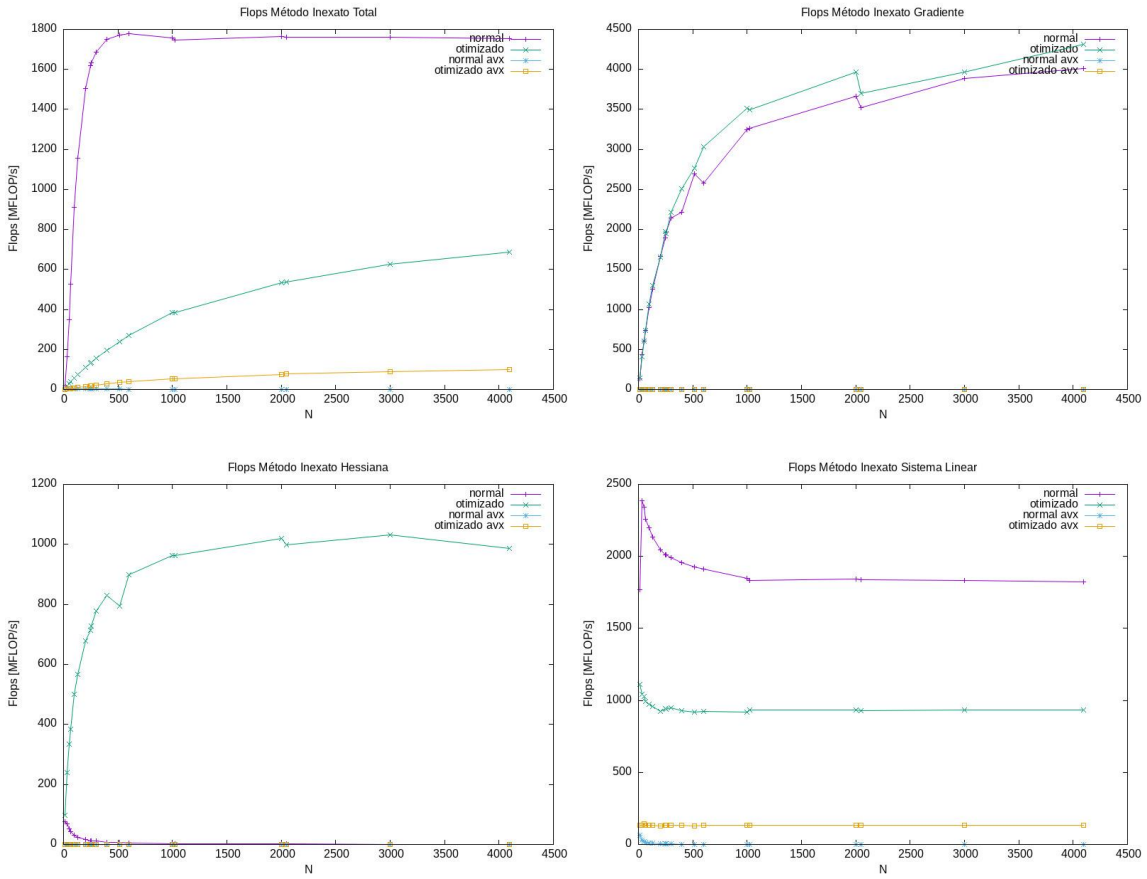
Semelhante ao método padrão

- Cache miss L2



Comparado com o método padrão, a principal diferença se deve no trecho Sistema Linear.

- Operações aritméticas



Semelhante ao padrão, porém no método de Gauss-Seidel foi possível uma pequena otimização de instruções AVX.

CONCLUSÃO

Após as otimizações efetuadas, ao executar novamente a ferramenta `gprof` obtemos:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 59.24      0.29     0.29      969   299.58   393.39  resolver_sl_gauss_se…
 20.43      0.39     0.10    42179     2.37     2.37  norma
  6.13      0.42     0.03  4913544     0.01     0.01  rosenbrock_dxdy
  6.13      0.45     0.03      969    30.99    30.99  retrossubs
  4.09      0.47     0.02      969    20.66    20.66  triangularizar
  2.04      0.48     0.01  1639140     0.01     0.01  rosenbrock_dx
  2.04      0.49     0.01     1938     5.17     5.17  somar_vetor
  0.00      0.49     0.00    15504     0.00     0.00  timestamp
```

Observamos que:

- Houve aumento significativo de desempenho da função `triangularizar()` em relação ao T1 Linha. Agora é responsável por apenas 4% do tempo de execução, contra 57% na versão anterior;
- Pelos gráficos vemos que `resolver_sl_gauss_seidel()` também teve um expressivo aumento de desempenho, mas é ainda responsável por boa parte do tempo de execução;
- A otimização de ambas as funções “permitiu” que outras funções começassem a ocupar os tempos de execução;
- Apesar de termos encontrado fórmulas para o cálculo das derivadas parciais, o que permitiria otimização de `rosenbrock_dxdy()`, nota-se que essa função ocupa no momento apenas 6% do tempo de execução, então julgamos não valer a pena alterar essa função em específico (o mesmo vale para `rosenbrock_dx()`);

Ao compilar o código com a flag `-fopt-info-vec-all`, vimos que houve a vetorização automática da função `retrossubs()`, mas não da

`triangularizar()` nem do loop mais interno da `resolver_sl_gauss_seidel()`:

```
triangularizar():
...
for (int i = 0; i < sl->n - 1; i++) {
    double m = sl->M->a[i] / sl->M->d[i];
    sl->M->a[i] = 0.0;
    sl->M->d[i + 1] -= sl->M->c[i] * m;
    sl->b[i + 1] -= sl->b[i] * m;
}
...

resolver_sl_gauss_seidel():
...
for (int i = 1; i < sl->n - 1; i++) {
    s->X_old[i] = sl->X[i];
    sl->X[i] = \ (sl->b[i] - sl->M->a[i - 1] * sl->X[i - 1] - sl->M->c[i] *
sl->X[i + 1]) / sl->M->d[i];
    s->delta[i] = sl->X[i] - s->X_old[i];
}
...
```

No primeiro caso, existe dependência de dados entre d[i + 1] e d[i], e entre b[i + 1] e b[i] (linhas em vermelho). Mas uma possível otimização seria separar a linha em verde para outro loop. Esse loop separado seria então vetorizado.

Já no segundo caso, infelizmente existe dependência de dados entre X[i], X_old[i], X[i - 1] e X[i + 1], então num primeiro momento não há como vetorizar esse loop em específico.

Uma solução para os casos acima seria então realizar o *loop unrooling*. No trabalho atual não aplicamos esse método.

CONSIDERAÇÕES FINAIS

Após a análise final dos resultados obtidos, nota-se que muitas outras otimizações ainda poderiam ser efetuadas. Entretanto, as modificações resultantes da triangularização foram tão impactantes que chegamos a conclusão de que otimizar ainda mais o trabalho não refletiria em uma melhora tão significativa no tempo de execução. Em um momento, executamos o programa com um milhão de dimensões, e o tempo de execução gasto (utilizando o utilitário `time` do gnu) foi de aproximadamente 24 segundos (real time). Para 4096 dimensões, o tempo gasto foi de 94 ms. Para efeito de comparação, o T1 Linha, executando com 4096, demorou 12 minutos.

Por isso, chegamos à conclusão de que não valeria a pena aplicar a mesma quantidade de trabalho (ou maior) que a já realizada na otimização do código para obter ganhos de performance na casa dos milissegundos. Para as especificações atuais, o programa tem execução praticamente instantânea (inferior a 1 segundo).

REFERÊNCIAS

<https://moodle.c3sl.ufpr.br/mod/assign/view.php?id=33468>

Acesso em: 01/04/22

<https://www.youtube.com/watch?v=x6LTt02hIEg>

Acesso em: 20/04/22

<https://moodle.c3sl.ufpr.br/pluginfile.php/143165/mod_resource/content/16/05-SistemasLineares.pdf>

Acesso em: 20/04/22

<https://en.wikipedia.org/wiki/Rosenbrock_function>

Acesso em: 29/04/22