Trabalho Prático 1: Aceleração N-Body com PThreads

Jecé Xavier Pereira Neto - 52552,

Matheus Rigoni Galvão - 54185

Supervisors:

Profº. Dr. José Rufino

Bragança

2022-2023

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Neste trabalho são descritas duas abordagens para a otimização de um algoritmo de *nbody*. Aqui está relatado o processo de divisão de trabalho para cada uma das abordagens, focando na diminuição de comunicação entre as threads e nodes. Este trabalho apresenta também os testes realizados para cada estratégia de implementação paralela adotada, e os seus resultados.

1.1 Objetivos

Acelerar o algoritmo de *nbody* escrito na linguagem c, atualizando o código mantendo o modelo de programação em serial e atualizando o mesmo código para usar o modelo de programação paralela usando PThreads.

1.2 Estrutura do Documento

Este documento esta estruturado em capitulos. O capitulo 1 se refere a introdução, os objetivos deste trabalho, e a estrutra do documento. Capitulo 3 ira conter a metodologia utilizada para solucionar o problema. Capitulo 4 será descrito como foi testado a solução do problema. Capitulo 6 está as conclusões.

Capítulo 2

Metodologia

O Algoritmo de simulação inicia verificando as entradas dos parâmetros para definir a quantidade de corpos presentes e a quantidade de interações que serão realizadas. Após é realizada a chamada da função *initSystemFromRandom* (Figura 2.1) para fazer a inicialização dos corpo antes da simulação.

```
54  void initSystemFromRandom()
55  {
56    int i;
57
58    GLOBAL_masses = (double *)malloc(GLOBAL_numBodies * sizeof(double));
59    GLOBAL_positions = (vector *)malloc(GLOBAL_numBodies * sizeof(vector));
60    GLOBAL_velocities = (vector *)malloc(GLOBAL_numBodies * sizeof(vector));
61    GLOBAL_accelerations = (vector *)malloc(GLOBAL_numBodies * sizeof(vector));
62
63    for (i = 0; i < GLOBAL_numBodies; i++)
64    {
65      GLOBAL_masses[i] = rand() % GLOBAL_numBodies;
66      GLOBAL_positions[i].x = rand() % GLOBAL_windowWidth;
67      GLOBAL_positions[i].y = rand() % GLOBAL_windowHeight;
68      GLOBAL_positions[i].z = 0;
69      GLOBAL_velocities[i].x = GLOBAL_velocities[i].y = GLOBAL_velocities[i].z = 0;
70    }
71  }
```

Figura 2.1: Função initSystemFromRandom

Com os dados prontos a função responsável por fazer os cálculos é chamada dentro de um loop com a quantidade de interações definidas. A função *simulate* (Figura 2.2)

é dividida em cinco funções, sendo cada uma delas encarregada de fazer uma parte dos cálculos.

```
160  void simulate()
161  {
162      computeAccelerations();
163      computePositions();
164      computeVelocities();
165      resolveCollisions();
166      validateSystem();
167  }
```

Figura 2.2: Função simulate

A primeira função executada é computeAccelerations, a qual é responsável pelo cálculo da aceleração de cada corpo em relação aos outros corpos utilizando seus vetores. A segunda função computePositions calcula a nova posição de cada corpo a partir da posição atual, velocidade e aceleração. A terceira função computeVelocities é responsável pelo cálculo da velocidade utilizando a velocidade atual e aceleração. A quarta função resolveCollisions é executada para

2.1 Otimização em Serial

A primeira estratégia para a fazer a otimização do algorítmico em modelo serial foi aproveitar que todos os cálculos estão dentro de laços de repetições. O intuito desta atualização foi diminuir a quantidade de laços que há no programa, porém, após alguns testes foi percebido que essa estratégia não executaria como o código original, devido a dependências que um *loop* com o outro, ocasionando assim divergências no *logs* da aplicação.

Como não foi possível diminuir a quantidade de *loops* da aplicação foi pensando em diminuir a quantidade de acesso a memória que o programa faz, ou seja, diminuir a quantidade de vezes que alguma função busca uma variável global. Aplicamos essa estratégia no principal *hotspost*, a função *computeAccelerations* otimizada está apresentada na figura (Figura 2.3). Nas linha 109, 101 e 111 da figura 2.3 são responsaveis por fazer as copias

das variais de massa, posição e aceleração respectivamete para o contexto da função.

```
105  void computeAccelerations()
106  {
107    int i, j;
108
109    double *masses = GLOBAL_masses;
110    vector *positions = GLOBAL_positions;
111    vector *accelerations = GLOBAL_accelerations;
112
113    for (i = 0; i < GLOBAL_numBodies; i++)
114    {
115      accelerations[i].x = 0;
116      accelerations[i].y = 0;
117      accelerations[i].z = 0;
118      for (j = 0; j < GLOBAL_numBodies; j++)
119      {
120        if (i ≠ j)
121        {
122          accelerations[i] = addVectors(accelerations[i], scaleVector
           (CONST_GravConstant * masses[j] / pow(mod(subtractVectors
           (positions[i], positions[j])), 3), subtractVectors(positions[j],
           positions[i])));
123        }
124      }
125    }
126
127    GLOBAL_accelerations = accelerations;
128  }
```

Figura 2.3: Função computeAccelerations otimizada

2.2 Profiling com Valgrind

A realização do profiling do algoritmo foi realizado usando a ferramenta callgrind, através do seguinte comando para a versão serial otimizada.

valgrind –tool=callgrind ./nbody-serial-opt.exe

Os resultados vizualizados no programa KCachegrind para determinar os hotspots podem ser vizualidos na Figura 2.4. A fração paralelizável do programa foi calculada somando a fração de cada função considerada paralelizável, sendo elas computeAccelerations, computePositions, computeVelocities e resolveCollisions. O resultado foi de que

98.76% do programa é paralelizável, sendo o maior principal hotspot a função compute-Accelerations, a qual representa uma fração de 91.76%. O call graph da função main é apresentado na Figura 2.5.

Incl.	Self	Called	Function
100.00	0.00	(0)	0x0000000000020…
99.99	0.00	1	(below main)
99.99	0.00	1	__libc_start_main@…
99.99	0.00	1	(below main)
99.99	0.01	1	main
99.98	0.02	30 000	simulate
91.76	29.86	30 000	computeAccelerati…
30.48	0.47	6 300 000	0x0000000000109…
30.01	4.49	6 300 000	pow@@GLIBC_2.29
25.52	0.47	6 300 000	0x0000000004864…
25.05	25.05	6 300 000	__ieee754_pow_fma
12.29	12.29	12 600 000	subtractVectors
7.46	7.46	7 650 000	addVectors
7.32	4.96	6 300 000	mod
6.08	6.08	6 750 000	scaleVector
3.28	3.28	30 000	resolveCollisions
2.52	1.24	30 000	computePositions
2.36	0.47	6 300 000	0x0000000000109…
1.89	1.18	6 300 000	sqrt
1.20	0.77	30 000	computeVelocities
1.20	1.20	30 000	validateSystem
0.71	0.71	6 300 000	__sqrt_finite@GLIB…
0.01	0.00	1	showSystem
0.01	0.00	15	0x0000000000109…
0.01	0.00	15	fprintf
0.01	0.00	15	__vfprintf_internal
0.01	0.00	15	buffered_vfprintf
0.01	0.00	1	_dl_start
0.01	0.00	15	__vfprintf_internal'2
0.01	0.00	1	_dl_sysdep_start
0.01	0.00	90	__printf_fp
0.01	0.00	90	__printf_fp_l
0.01	0.00	1	dl_main
0.00	0.00	5	_dl_relocate_object
0.00	0.00	112	_dl_lookup_symbol_x

Figura 2.4: Tabela de principais hotspots

5

6

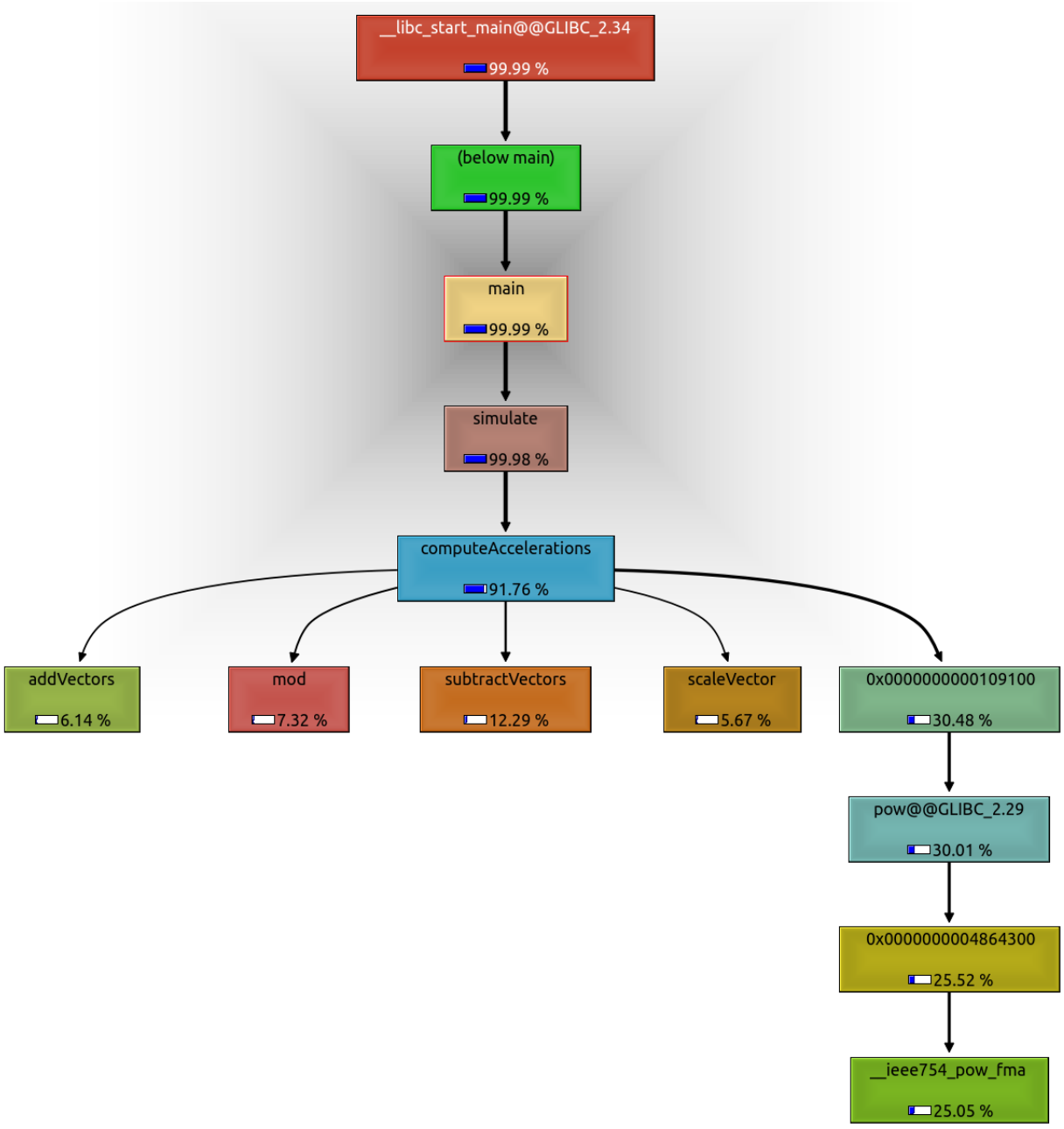


Figura 2.5: Call graph main

2.3 Cálculo de Speedup e Eficiência teóricos

Considerando a fração paralelizável do algoritmo como 98.76%, é possível calcular o Speedup téorico utilizando a Lei de Amdhal para um número de threads N=1...8 como apresentado na tabela 2.1.

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
St	1.00	1.98	2.93	3.86	4.76	5.65	6.52	7.36

Tabela 2.1: Speedups calculados pela Lei de Amdhal

Após o cálculo do speedup para cada quantidade de threads, a eficiência é calculada pela divisão do speedup por seu respectivo número de threads. Os resultados de tais cálculos são apresentados na tabela 2.2.

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Et(%)	100	99	98	96	95	94	93	92

Tabela 2.2: Eficiências calculadas usando o Speedup téorico

Por fim, o limite teórico de speedup é calculado com base na divisão de 1 pela fração não paralelizável do algoritmo que é 1.24%. O resultado para o limite teórico foi de 80.64.

2.4 Otimização em Paralelo

Para a solução em paralelo foi proposto a utilização de *posix threads (pthreads)*. Desta forma foi separado em cada *thread* uma parte do cálculo, isso sendo possível devido que cada etapa da função não tem dependência dela mesma. A primeira atualização do código foi na função *simulate*, como cada etapa depende da anteria além de fazer a criação das *thread* é necessário esperar que todas finalizem, para isso, utilizou-se a *pthreads join* (Figura 2.6).

Na figura (Figura 2.7) é mostrado a função *computeAccelerations* convertida para ser executada como uma *thread*. Deve-se resaltar que a principal alteração desta função para sua versão original foi o modo que o lanço de repetição *(for)* (Linha 103) é definido. O

```
185 void parallel_simulate()
186 {
187   pthread_t threads[NUM_THREADS];
188   int t;
189
190   // computeAccelerations
191   for (t = 0; t < NUM_THREADS; t++)
192     pthread_create(&threads[t], NULL, parallel_computeAccelerations, (void *)t);
193   for (t = 0; t < NUM_THREADS; t++)
194     pthread_join(threads[t], NULL);
195
196   // computePositions
197   for (t = 0; t < NUM_THREADS; t++)
198     pthread_create(&threads[t], NULL, parallel_computePositions, (void *)t);
199   for (t = 0; t < NUM_THREADS; t++)
200     pthread_join(threads[t], NULL);
201
202   // computeVelocities
203   for (t = 0; t < NUM_THREADS; t++)
204     pthread_create(&threads[t], NULL, parallel_computeVelocities, (void *)t);
205   for (t = 0; t < NUM_THREADS; t++)
206     pthread_join(threads[t], NULL);
207
208   // resolveCollisions
209   for (t = 0; t < NUM_THREADS; t++)
210     pthread_create(&threads[t], NULL, parallel_resolveCollisions, (void *)t);
211   for (t = 0; t < NUM_THREADS; t++)
212     pthread_join(threads[t], NULL);
213
214   validateSystem();
215 }
```

Figura 2.6: Função simulate versão em paralelo

inicializador está relacionado com o ”id”que foi passado durante a inicialização da *thread* pela linha 192 da figura 2.6, após isso o incremendo é feito pela quantidade de *threads* definidas. Está estratégia aplicada ajuda e minimizar sobrecarga em um *node*.

```
98  void *parallel_computeAccelerations(void *arg)
99  {
100   long tid = (long)arg;
101   int i, j;
102
103   for (i = tid; i < GLOBAL_numBodies; i += NUM_THREADS)
104   {
105     GLOBAL_accelerations[i].x = 0;
106     GLOBAL_accelerations[i].y = 0;
107     GLOBAL_accelerations[i].z = 0;
108     for (j = 0; j < GLOBAL_numBodies; j++)
109     {
110       if (i != j)
111       {
112         GLOBAL_accelerations[i] = addVectors(GLOBAL_accelerations[i],
          scaleVector(CONST_GravConstant * GLOBAL_masses[j] / pow(mod
          (subtractVectors(GLOBAL_positions[i], GLOBAL_positions[j])), 3),
          subtractVectors(GLOBAL_positions[j], GLOBAL_positions[i])));
113       }
114     }
115   }
116
117   pthread_exit(NULL);
118 }
```

Figura 2.7: Função computeAccelerations vesão em paralelo

O código completo com a solução pode ser acessado no repositório do Github.

Capítulo 3

Testes

Os testes foram feitos em um ambiente local usando a máquina pessoal com configurações conforme a tabela 3.1, os softwares em execução durantes os teste estão descritos na tabela 3.2. Para fazer a automação dos testes foi criado dois *scripts*, um em *shellscript* (figura 3.1), responsável por compilar e executar o projeto, além de fazer salvar os *logs* de tempo e do programa nos arquivos, na linha 91 e 106 é chamada a função destinada a comparar as sáidas do algoritmo original com o algoritmo otimizado usando o comando *diff* (Figura 3.2) (Linha 25).

O segundo *script* é feito em *python* com o objetivo de fazer a leitura *logs* de tempo e gerar a médica de execução. A compilação dos algoritmos foram feitas usando GCC com o auxilio de arquivo *Makefile* (Figura 3.3).

Marca e modelo	Notebook Dell Latitude 3400
Processador	Intel Core i7-8565U de 8ª geração (cache de 8 MB, contagem de 4 núcleos/ 8 threads, 1,8 GHz a 4,6 GHz, 15 W TDP)
Memória	16 GB (2 x 8 GB) DDR4 SDRAM (memória não ECC) 2400 MHz
Armazena-mento	M.2 SSD 2280 240GB
Sistema Operacional	Linux Ubuntu 22.04

Tabela 3.1: Especificações do computador usado para os testes.

Nome	Descrição
Hyper	Uma interface de terminal desenvolvida utilizando o Electron, framework javascript para criação de aplicações desktop, ou seja, toda sua interface é criada utilizando tecnologias web (html, css e javascript).
OhMyZSH	Um framework open-source mantido pela comunidade para gerenciar a configuração do ZSH, um interpretador de comandos UNIX (shell) que pode ser utilizado como processador de comandos de script.
Monitor	O System Monitor é um programa utilitário e está em um grupo com outros utilitários do Ubuntu para verificar o uso de recursos do sistema. Foi utilizando para fazer o tirar *print* do gráficos.
Google Chrome	Navegador de internet desenvolvido pela Google. Durantes os permaneceu aberto, sem interação.
VS CODE	Editor de código da Mycrosoft. Durantes os permaneceu aberto, sem interação.

Tabela 3.2: Sowftwares em execução durantes os testes.

```
82  if [ "$algorithm" == 'serial-opt' ]; then
83
84    echo ""
85    echo "5. INICIANDO TESTES CODIGO >>>>>>>> SERIAL OTMIZADO <<<<<<<< (Rodará $qtdTests vezes):"
86    for numTest in $(eval echo "{1..$qtdTests}") ; do
87
88      echo " Rodando o teste de número $numTest ... "
89      { time ./nbody-$algorithm.exe $nbody $nsteps 2> $result_folder/
      nbody-$algorithm-$nbody_string-$nsteps_string.txt ; } 2>> $result_folder/
      time_$algorithm-$nbody_string-$nsteps_string.txt ;
90
91      comporate_logs $numTest
92
93    done
94
95  else
96
97    echo ""
98    echo "5. INICIANDO TESTES CODIGO >>>>>>>> THREAD OTMIZADO <<<<<<<< (Rodará $qtdTests vezes em
      cada thread):"
99    for thread in {1..8}; do
100     echo " Iniciando testes com $thread thread (s)"
101       for numTest in $(eval echo "{1..$qtdTests}") ; do
102
103         echo "  Rodando o teste de número $numTest ... "
104         { time ./nbody-$algorithm.exe $nbody $nsteps $thread 2> $result_folder/
          nbody-$thread\_$algorithm-$nbody_string-$nsteps_string.txt ; } 2>> $result_folder/
          time_$thread\_$algorithm-$nbody_string-$nsteps_string.txt ;
105
106         comporate_logs $numTest $thread
107
108       done
109   done
110
111 fi
```

Figura 3.1: Função para executar dos teste

```
function comporate_logs
{
  [[ $2 = "" ]] && nameFile="$algorithm" || nameFile="$2_$algorithm";

  DIFF=$(diff $result_folder/nbody-serial-$nbody_string-$nsteps_string.txt
  $result_folder/nbody-$nameFile-$nbody_string-$nsteps_string.txt)
  if [ "$DIFF" == "" ]; then
    echo "   Analisando Logs: NÃO HOUVE DIFERENÇA"
  else
    echo $DIFF >  $result_folder/diff_$1-$nameFile.txt ;
    echo "   Analisando Logs: HOUVE DIFERENÇA, veja o arquivo: $result_folder/
    diff_$1-$nameFile.txt"
  fi
}
```

Figura 3.2: Função para executar dos teste

```
CC=gcc
MFLAGS=-lm
TFLAGS=-pthread

#uncomment only if SDL bgi is installed
GFLAGS=-lSDL_bgi -lSDL2

#use the 1st line in alternative to the 2nd (1st is for valgrind, 2nd is for
benchmarking)
#CFLAGS=-Wall -O0 -g
CFLAGS=-Wall -O2

#use the 1st line in alternative to the 2nd (use 2nd only if SDL bgi installed)
#all: nbody-serial.exe nbody-threads.exe
all: nbody-serial.exe nbody-serial-opt.exe nbody-threads.exe  nbody-serial-gui.
exe nbody-serial-opt-gui.exe

nbody-serial.exe: nbody-serial.c
  $(CC) $(CFLAGS) nbody-serial.c $(MFLAGS) -o nbody-serial.exe

nbody-serial-opt.exe: nbody-serial-opt.c
  $(CC) $(CFLAGS) nbody-serial-opt.c $(MFLAGS) -o nbody-serial-opt.exe

nbody-threads.exe: nbody-threads.c
  $(CC) $(CFLAGS) nbody-threads.c $(MFLAGS) $(TFLAGS) -o nbody-threads.exe

clean:
  rm -f *.exe *.o a.out
```

Figura 3.3: Makefile

3.1 Otimização em Serial

Para fazer os teste foi executados o algorimito original três vezes e posteriormente fez a execução os algorimimo optimizado. Nos resultados obtidos, foi observado que o desempenho em média melhorou um segundo (Figura 3.4).

```
20:23:09 => jece in project1-ca/ac-nbody-resources on  create-parallel
❯ bash ./compare.sh serial-opt 3 300 30000

1. Compilando com MAKEFILE
make: Nothing to be done for 'all'.

2. Lipando pasta de LOGS
rm: cannot remove './resultados/*.txt': No such file or directory

3. Definido dados de teste
 Algoritmo testado: serial-opt
 Quantidade de repetições: 3
 Quantidade de corpos: 300
 Quantidade de interações da simulação: 30000

4. INICIANDO TESTES CODIGO >>>>>>>> ORIGINAL <<<<<<<< (Rodará 3 vezes):
 Rodando o teste de número 1...
 Rodando o teste de número 2...
 Rodando o teste de número 3...

5. INICIANDO TESTES CODIGO >>>>>>>> SERIAL OTMIZADO <<<<<<<< (Rodará 3 vezes):
 Rodando o teste de número 1...
   Analisando Logs: NÃO HOUVE DIFERENÇA
 Rodando o teste de número 2...
   Analisando Logs: NÃO HOUVE DIFERENÇA
 Rodando o teste de número 3...
   Analisando Logs: NÃO HOUVE DIFERENÇA

6. CALCULANDO MÉDIAS DOS TEMPOS
time_serial-opt-300-30000.txt
{'real': 237.17533333333333, 'sys': 0.009333333333333334, 'user': 237.072}
============================================================
time_original-300-30000.txt
{'real': 238.77366666666668, 'sys': 0.005666666666666667, 'user': 238.64566666666667}
============================================================
```

Figura 3.4: Resultados da execução do algoritmo serial otimizado

3.2 Otimização em Paralelo

Foi utlizada a mesma estratégia anterior de teste anterior, executar três vezes o algoritmo original e os algoritmos otimizados, vale reslatar que cada threads teve três execução para a geração da média do tempo.

Foi utlizado o *System Monitor* do SO para verifizar o comportamento dos núcleos da

CPU durantes as simulações. Na figura 3.5 apresenta o gráfico no periodo de execução do algoritmo original, vale resaltar que nenhum núcleo é sobrecarregado, devido, o balanceamento que o próprio SO faz. A difenreça de execução entre cada núcleo é diminuida conforme a quantidade de *threads* aumenta (Figura 3.6), conseguentimente deixando núcleos trabalhando mais próximo da capacidade total.

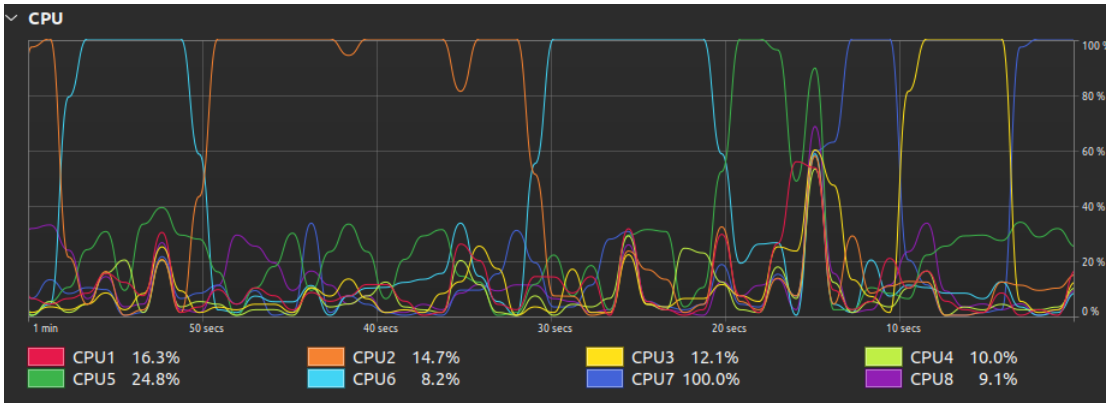


Figura 3.5: *System Monitor* - Algoritmo Original

Os resultados obtidos dos testes utilizando *PThreads* podem ser observados na Tabela 3.3, foi observado que o desempenho melhorou com até 4 *threads* (Figura 3.7). Porém conforme aumentamos a quantidade de threads o desempenho não melhorou. Na Figura 3.8 é mostrado a média de tempo utilizando para execução em cada *thread*.

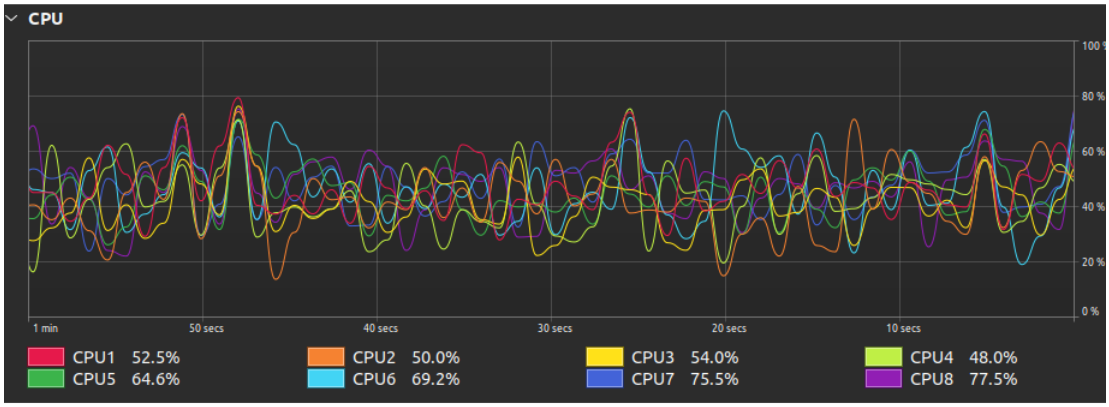
N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$T_R(s)$	260	167	126	104	110	113	112	122

Tabela 3.3: Tempo real em paralelo

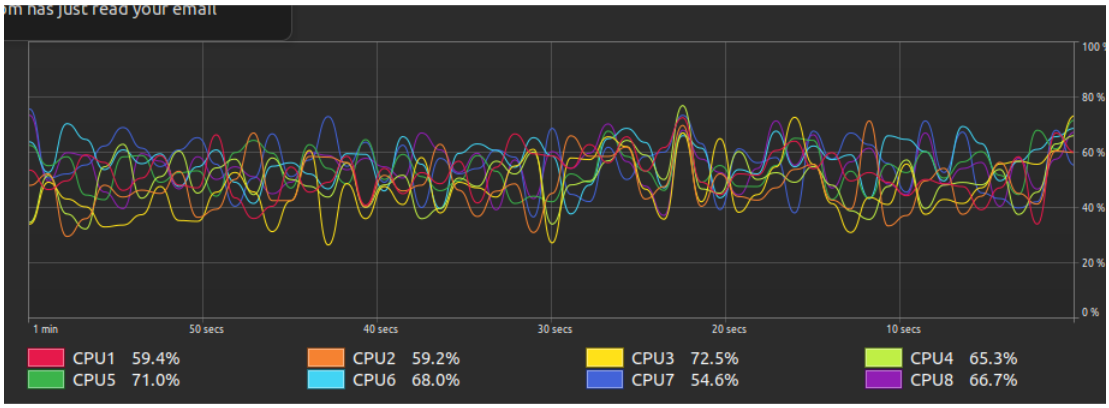
Nas tabelas 3.4, 3.5 e nas figuras 3.9, 3.10 é mostrado os resultados de speedup e eficiência, do algoritmo com *PThreads*. Foi observado que, de modo geral, a paralelização foi eficiente com 4 *threads*, depois disso o algoritmo apresentou um desempenho inferior, e em outros casos até pior que o algoritmo sequencial.

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
S_R	0.89	1.39	1.84	2.23	2.11	2.05	2.07	1.90

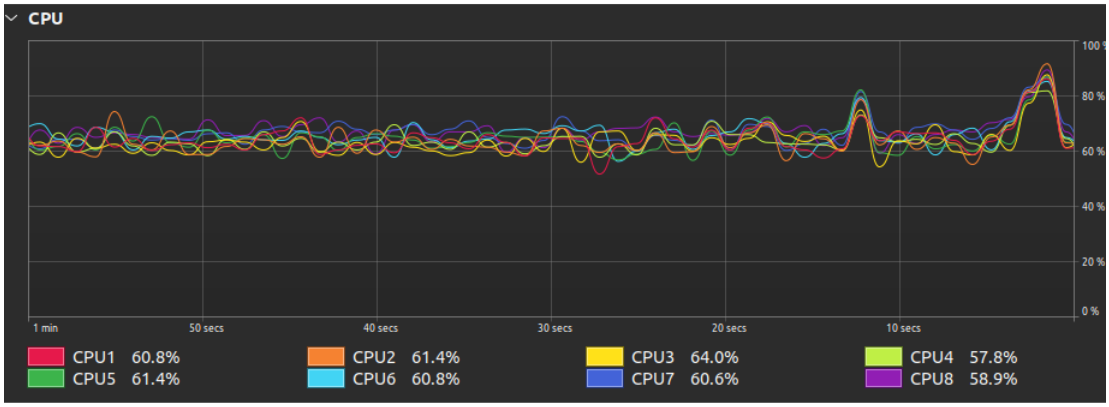
Tabela 3.4: Speedups Real



4 Threads



5 Threads



6 Threads

Figura 3.6: *System Monitor* - Algoritmo em Threads

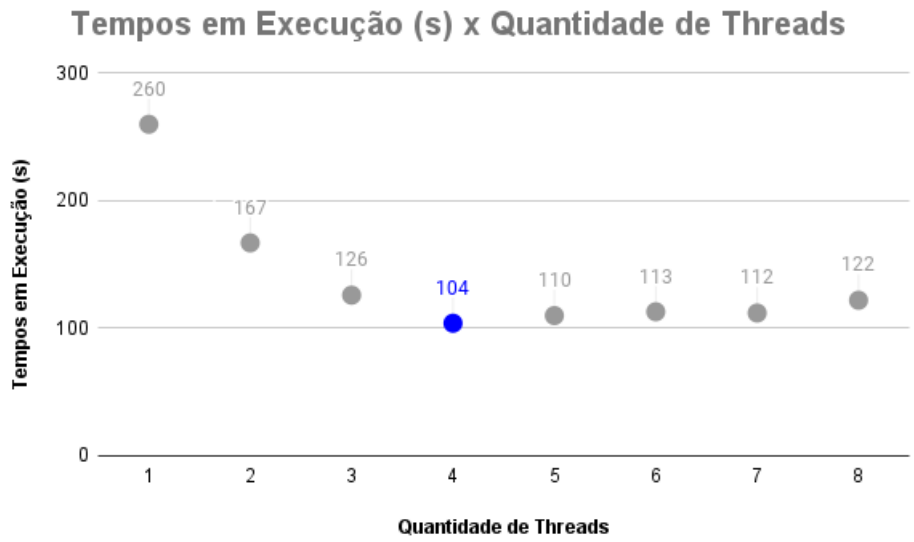


Figura 3.7: Gráfico do tempo de execução em *threads*

```
6. CALCULANDO MÉDIAS DOS TEMPOS
time_1_threads-300-30000.txt
{'real': 260.07099999999997, 'sys': 11.409999999999998, 'user': 244.37300000000002}
============================================================
time_4_threads-300-30000.txt
{'real': 104.29133333333334, 'sys': 23.921000000000003, 'user': 308.606}
============================================================
time_original-300-30000.txt
{'real': 232.69066666666666, 'sys': 0.018666666666666665, 'user': 232.58133333333333}
============================================================
time_3_threads-300-30000.txt
{'real': 126.42966666666666, 'sys': 16.340333333333334, 'user': 309.46099999999996}
============================================================
time_5_threads-300-30000.txt
{'real': 110.78933333333335, 'sys': 35.294333333333334, 'user': 353.8046666666666}
============================================================
time_2_threads-300-30000.txt
{'real': 167.078, 'sys': 14.345, 'user': 300.56800000000004}
============================================================
time_8_threads-300-30000.txt
{'real': 122.66966666666667, 'sys': 65.18533333333333, 'user': 469.01866666666666}
============================================================
time_6_threads-300-30000.txt
{'real': 113.931, 'sys': 44.559666666666665, 'user': 424.667}
============================================================
time_7_threads-300-30000.txt
{'real': 112.88100000000001, 'sys': 58.522, 'user': 440.6776666666667}
============================================================
```

Figura 3.8: Resultados da execução do algoritmo com threads

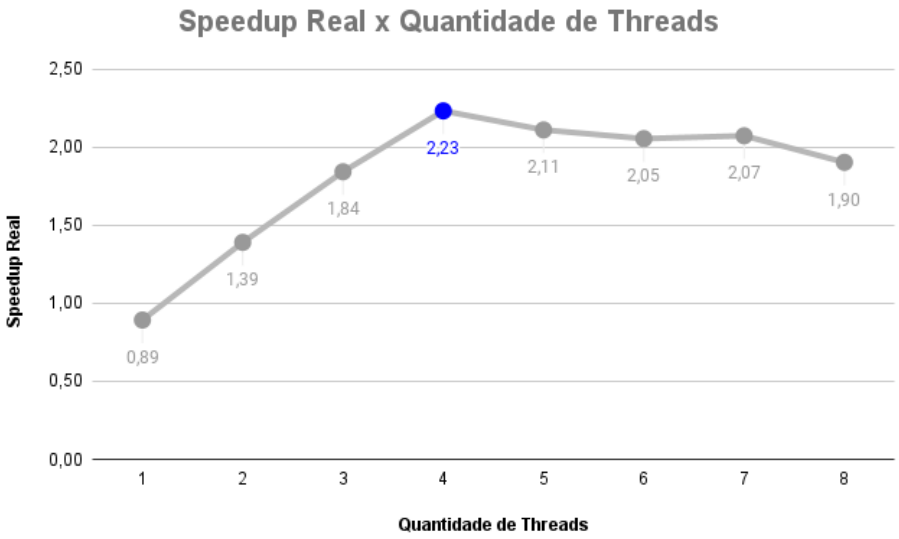


Figura 3.9: Speedups Real

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$E_R(\%)$	89.23%	69.46%	61.38%	55.77%	42.18%	34.22%	29.59%	23.77%
$E_R/E_t(\%)$	89.23%	70.16%	62.63%	58.09%	44.40%	36.40%	31.82%	25.84%

Tabela 3.5: Eficiência Real

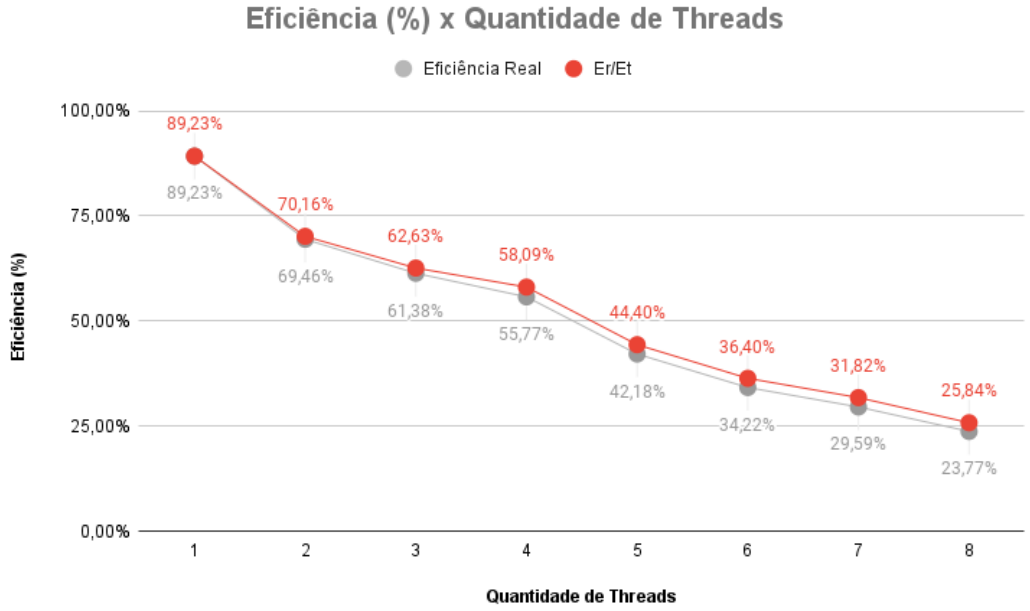


Figura 3.10: Eficiência Real

Capítulo 4

Conclusão e Discussão

Em vista dos resultados obtidos através da implementação do algoritmo utilizando *threads* e avaliado em uma série de testes, é evidente uma melhora significante de tempo em relação ao algoritmo serial, sendo o melhor desempenho observado quando 4 *threads* são utilizadas. Quantidades superiores a 4 *threads* apresentam tempos de execução similares entre si e poucos segundos acima dos testes feitos com 4 *threads*. Uma explicação plausível para tais resultados é a quantidade extra de *overhead* gerada por mais *threads* em função de administra-las, para o problema resolvido pelo algoritmo, quantidades de *threads* superiores a 4 aparentemente geram uma carga de *overhead* que compensa negativamente os benefícios da paralelização obtida por tais *threads* adicionais.

Os cálculos teóricos de speedup feitos aplicando a Lei de Amdhal apresentaram resultados significativamente diferentes do speedup real posteriormente calculado, tal diferença tem como razão pelo fato de a Lei de Amdhal não considerar o *overhead* gerado pelo uso de mais *threads*, assim teoricamente quanto mais *threads*, mais rápido seria a execução do programa, o que não é o caso na realidade. Uma das possíveis melhoras seria aproveitar valores de vetores já calculados, evitando recalcular tais valores. No presente trabalho foi realizada a paralelização de 4 funções, entre elas o principal *hotspot*, é possível que a paralelização das outras 3 funções tenham atrasado a execução, assim sendo necessario uma avaliação do melhor conjunto de funções paralelizadas.