EP 1 - Programação Concorrente

Alberto Bueno Júnior - Nº USP: 6514202
Daniel Francisco Reverbel - Nº USP: 4512210
Felipe Simionato Solferini - Nº USP: 6431304

Maio de 2011

Contents

1 Introdução ao trabalho

Este é um trabalho feito para a disciplina "MAC0438 - Programação Concorrente", ministrada, em 2011, pelo professor Daniel Macêdo Batista (`http://www.ime.usp.br/~batista`). Esta disciplina é oferecida aos alunos de graduação em Ciência da Computação do Instituto de Matemática e Estatística (IME - `http://www.ime.usp.br`) da Universidade de São Paulo (USP - `http://www.usp.br`).

2 Problema

O programa deste trabalho simula um analisador de pacotes de rede. Este dispositivo recebe pacotes e os copia para um disco rígido interno, para análise posterior. A parte deste programa é, na verdade, lidar com os pacotes que chegam ao dispositivo e simular uma cópia para o disco rígido. A entrada do programa é **M** e **N**. **M** representa o número de pacotes que irão chegar e devem ser copiados. **N** representa o número de processos copiadores que rodam concorrentemente.

3 Código

ep1.c:

```
1  /* Alberto Bueno Junior
2   * Felipe Solferini
3   * Daniel Reverbel
4   *
5   * Compilando:  gcc -Wall -pthread ep1.c -o ep1
6   * Rodando: ./ep1 <m> <n> [s]
7   */
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <pthread.h> /* Para trabalhar com threads */
12 #include <sys/time.h>
13 #include <unistd.h>
14
15 #define COPY_RATE 0.75 /* bytes / ms */
16 #define MIN_CHEGADA 250 /* ms */
17 #define MAX_CHEGADA 1000 /* ms */
18 #define MAX_TAMANHO 1500 /* bytes */
19 #define CONV 10
20
21 struct pacote {
22   int t_chegada;
23   int tamanho;
24 };
25 typedef struct pacote* pacote_pointer;
26
27 struct node {
28   pacote_pointer pacote;
29   struct node* next;
30 };
31 typedef struct node * node_pointer;
32
33 struct dados {
34   int pac_copiados;
35   int b_copiados;
36   double ociosidade_total;
37   double tempo_total;
38 };
39 typedef struct dados *copiador_dados;
40
41 /* Variaveis globais */
42 int senha;
43 int copiados;
44 node_pointer head;
45 node_pointer tail;
```

```
46 copiador_dados *cop_dados;
47 static pthread_mutex_t cs_mutex = PTHREAD_MUTEX_INITIALIZER;
48
49 /* Cria novo no, para depois ser colocado na fila */
50 node_pointer novo_node() {
51   node_pointer novo;
52   novo = (node_pointer) malloc(sizeof(* novo) );
53   if (novo == NULL) {
54     perror("Malloc ERROR no novo");
55     exit(1);
56   }
57   novo->pacote = (pacote_pointer) malloc(sizeof(*novo->pacote) )
       ;
58   novo->pacote->t_chegada = MIN_CHEGADA + (rand() % (MAX_CHEGADA
       - MIN_CHEGADA) );
59   novo->pacote->tamanho = 1 + (rand() % MAX_TAMANHO);
60   novo->next = NULL;
61   return novo;
62 }
63 /* Verifica se a fila esta vazia */
64 int fila_vazia() {
65   pthread_mutex_lock( &cs_mutex );
66   if (head->next == NULL) {
67     pthread_mutex_unlock( &cs_mutex );
68     return 1;
69   }
70   else {
71     pthread_mutex_unlock( &cs_mutex );
72     return 0;
73   }
74 }
75
76 /* Funcao que sera usada nas threads 'copiador' */
77 void * copiador (int *n) {
78   struct timeval t_antes, t_depois, t_final;
79   double delta;
80   node_pointer prox;
81   float espera;
82   while(1) {
83     gettimeofday(&t_antes, NULL);
84     while(fila_vazia()) {
85       usleep(MIN_CHEGADA * 500);
86     }
87     gettimeofday(&t_depois, NULL);
88     delta = (t_depois.tv_sec - t_antes.tv_sec) + (t_depois.
         tv_usec - t_antes.tv_usec) / 1000000; /* s */
89     /* Inicio da secao critica */
90     pthread_mutex_lock( &cs_mutex );
91     prox = head->next;
```

```
92      if (prox == NULL) {
93        pthread_mutex_unlock( &cs_mutex );
94        continue;
95      }
96      head->next = prox->next;
97      if (prox == tail) {
98        tail = head;
99      }
100     pthread_mutex_unlock( &cs_mutex );
101     /* Final da secao critica */
102     espera = prox->pacote->tamanho / (float) COPY_RATE;
103     usleep((espera * 1000) / CONV);
104     cop_dados[*n]->b_copiados += prox->pacote->tamanho;
105     cop_dados[*n]->pac_copiados += 1;
106     cop_dados[*n]->ociosidade_total += delta;
107     gettimeofday(&t_final, NULL);
108     delta = (t_final.tv_sec - t_antes.tv_sec) + (t_final.tv_usec
          - t_antes.tv_usec) / 1000000; /* s */
109     cop_dados[*n]->tempo_total += delta;
110     copiados++;
111   }
112
113   return NULL;
114 }
115
116 /* Funcao que sera usada nas threads 'pacote' */
117 void * pacote (int *n) {
118   node_pointer novo;
119
120   /* Criando pacote */
121   novo = novo_node();
122   /* Espera a sua vez */
123   while (senha != *n) {
124     usleep((MIN_CHEGADA * 100) / CONV);
125   }
126   /* Rodando o processo */
127   usleep((novo->pacote->t_chegada * 1000) / CONV);
128   /* Inicio da secao critica */
129   pthread_mutex_lock( &cs_mutex );
130   novo->next = NULL;
131   tail->next = novo;
132   tail = novo;
133   senha++;
134   pthread_mutex_unlock( &cs_mutex );
135   /* Final da secao critica */
136   return NULL;
137 }
138
139 int main (int argc, char *argv[]) {
```

```
140   /* ====== Declaracoes ====== */
141   int i, m, n, seed, script_out;
142   pthread_t * ids_pacotes; /* Guardam os ids das threads */
143   pthread_t * ids_copiadores;
144   int * arg; /* Vetor com os argumentos passados para cada
          thread */
145   /* m = quantidade total de pacotes a serem lidos */
146   /* n = quantidade de pacotes que podem ser copiados
          simultaneamente */
147
148   /* ====== Inicializacoes ====== */
149   if (argc == 3) {
150     m = atoi(argv[1]);
151     n = atoi(argv[2]);
152     script_out = 0;
153   }
154   else {
155     if (argc == 4) {
156       script_out = 1;
157       m = atoi(argv[1]);
158       n = atoi(argv[2]);
159     }
160     else {
161       script_out = 0;
162       printf("Entrada incorreta!\n");
163       return -1;
164     }
165   }
166   seed = time(NULL);
167   srand(seed);
168   senha = 0;
169   head = (node_pointer) malloc(sizeof(*head) );
170   if (head == NULL) {
171     perror("Malloc ERROR no HEAD");
172     exit(1);
173   }
174   head->pacote = NULL;
175   head->next = NULL;
176   tail = head;
177   cop_dados = (copiador_dados *) malloc(n * sizeof(
          copiador_dados) );
178   if (cop_dados == NULL) {
179     perror("Malloc ERROR no cop_dados");
180     exit(1);
181   }
182   for (i = 0; i < n; i++) {
183     cop_dados[i] = (copiador_dados) malloc(sizeof(struct dados)
            );
184     if (cop_dados[i] == NULL) {
```

```
185         perror("Malloc ERROR no cop_dados[i]");
186         exit(1);
187       }
188       cop_dados[i]->b_copiados = 0;
189       cop_dados[i]->ociosidade_total = 0;
190       cop_dados[i]->tempo_total = 0;
191       cop_dados[i]->pac_copiados = 0;
192     }
193
194     /* ====== Malloca os vetores ====== */
195     ids_pacotes = (pthread_t *) malloc(m * sizeof(pthread_t));
196     if (ids_pacotes == NULL) {
197       perror("Malloc ERROR no ids_pacotes");
198       exit(1);
199     }
200     if (!ids_pacotes) {
201       fprintf(stderr,"Erro no malloc do pthread_t\n");
202       return(1);
203     }
204     ids_copiadores = (pthread_t *) malloc(n * sizeof(pthread_t));
205     if (!ids_copiadores) {
206       fprintf(stderr,"Erro no malloc do pthread_t\n");
207       return(1);
208     }
209     arg = (int *) malloc(m * sizeof(int) );
210     if (!arg) {
211       free(ids_pacotes);
212       free(ids_copiadores);
213       fprintf(stderr,"Erro no malloc do vetor\n");
214       return(1);
215     }
216
217     /* ====== Inicializa os copiadores ====== */
218     for (i = 0; i < n; i++) {
219       arg[i] = i;
220       if (pthread_create(&(ids_copiadores[i]),NULL,(void *)
              copiador,(void *)&(arg[i]))) {
221         fprintf(stderr,"Erro no pthread_create\n");
222         free(ids_pacotes);
223         free(ids_copiadores);
224         free(arg);
225         return(2);
226       }
227     }
228     for (i = 0; i < m; i++) {
229       arg[i] = i;
230       if (pthread_create(&(ids_pacotes[i]),NULL,(void *)pacote,(
              void *)&(arg[i]))) {
231         fprintf(stderr,"Erro no pthread_create\n");
```

```
232         free(ids_pacotes);
233         free(ids_copiadores);
234         free(arg);
235         return(2);
236       }
237     }
238     /* ====== LOOP PRINCIPAL ====== */
239     while(copiados < m) {
240       usleep(1000);
241     }
242
243     for (i = 0; i < n; i++) {
244       if (script_out) {
245         printf("%d %d %d %.2f %.2f\n", i + 1, cop_dados[i]->
              pac_copiados, cop_dados[i]->b_copiados, cop_dados[i]->
              ociosidade_total, cop_dados[i]->tempo_total);
246       }
247       else {
248         printf("C%d: %d pacotes, %d bytes, %.2f s (total = %.2f s)
              \n", i + 1, cop_dados[i]->pac_copiados, cop_dados[i]->
              b_copiados, cop_dados[i]->ociosidade_total, cop_dados[i
              ]->tempo_total);
249       }
250     }
251
252     free(ids_copiadores);
253     free(ids_pacotes);
254     return(0);
255 }
```

4 Sobre os testes

Foram relalizados testes com o seguinte conjunto de entradas:

- $\mathbf{M} = 10$ e $\mathbf{N} = 1$
- $\mathbf{M} = 100$ e $\mathbf{N} = 1$
- $\mathbf{M} = 1000$ e $\mathbf{N} = 1$
- $\mathbf{M} = 1000$ e $\mathbf{N} = 10$
- $\mathbf{M} = 1000$ e $\mathbf{N} = 100$
- $\mathbf{M} = 1000$ e $\mathbf{N} = 1000$
- $\mathbf{M} = 1000$ e $\mathbf{N} = 10000$

Todos os testes foram automatizados com os scrips escritos na lingaguem *Perl*. Além disso, cada teste foi realizado 30 vezes. Com os dados obtidos de cada um deles, montamos três tipos diferentes de gráfico:

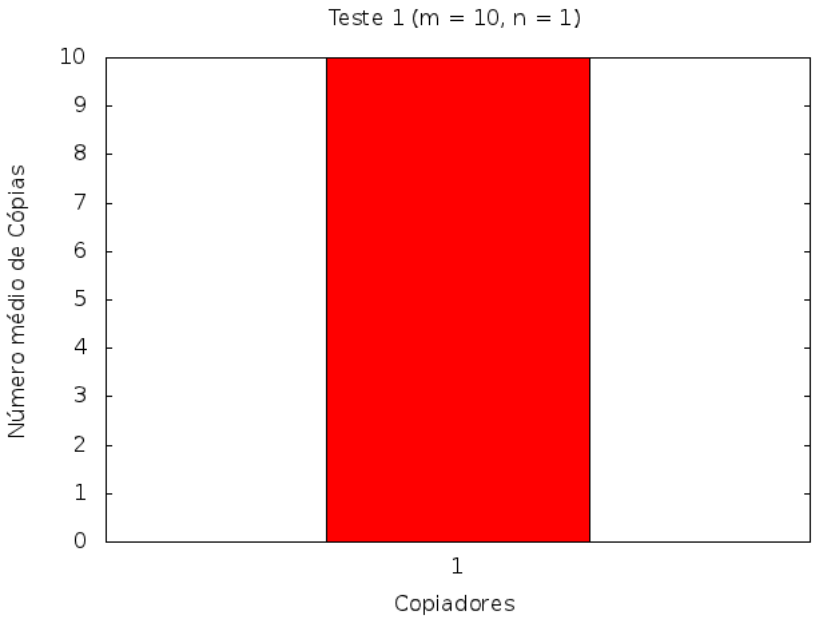
- **Média de cópias por copiador:** No eixo X, os copiadores. No eixo Y, a média de cópias realizada por cada copiador.
- **Média de Bytes copiados:** No eixo X, os copiadores. No eixo Y, a média do número de Bytes copiados por cada copiador.
- **Média de ociosidade:** No eixo X, os copiadores. No eixo Y, a média do tempo de ociosidade de cada copiador (em vermelho) e a média do tempo total de execução da thread representada pelo copiador (em verde).

4.1 Máquina utilizada

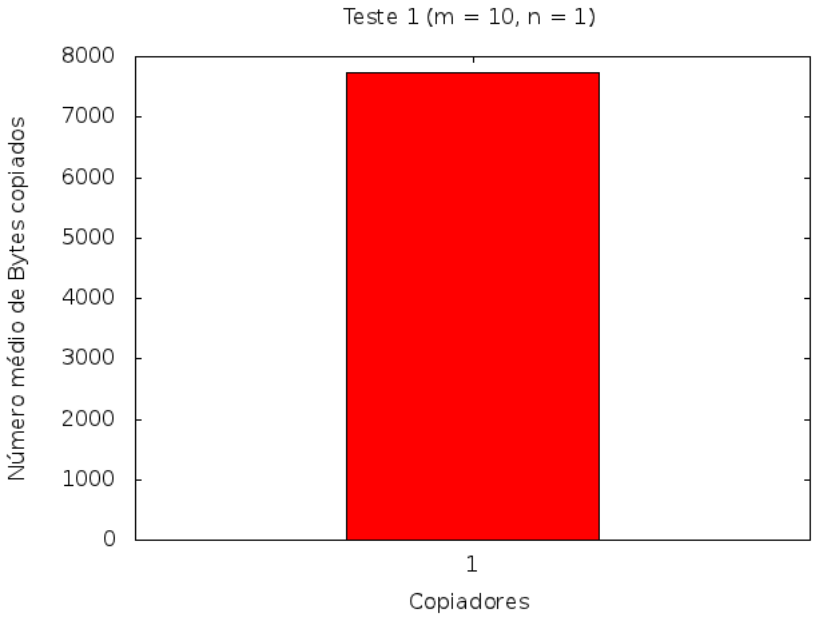
Os testes foram realizados em um Laptop Dell Inspiron - Intel Core 2 Duo; 4 GB de memória.

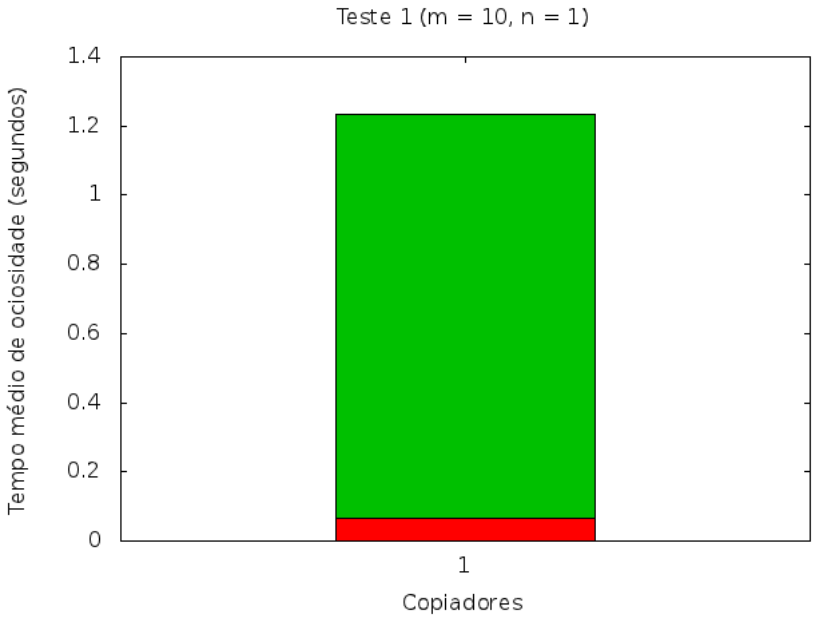
5 Testes

5.1 Teste 1 - M = 10, N = 1

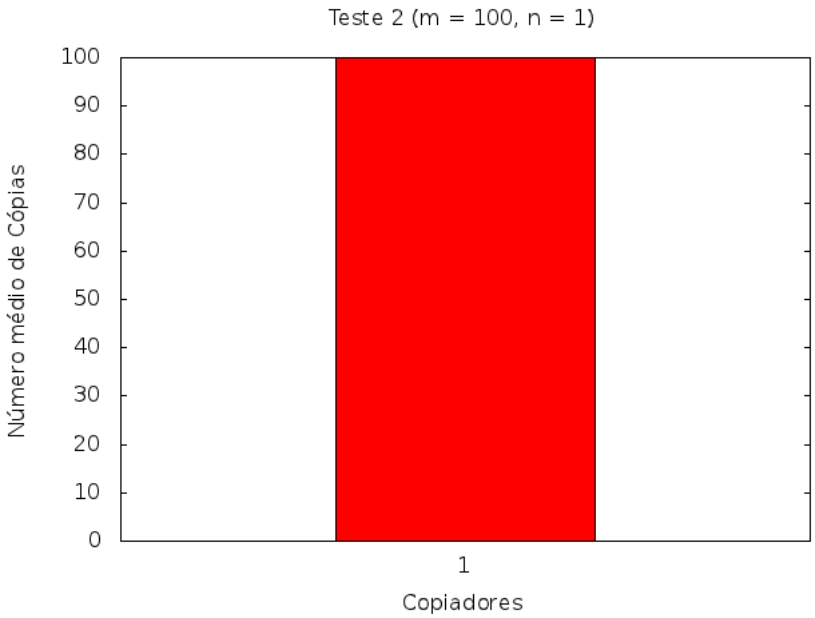


MAX CÓPIAS = 10
MIN CÓPIAS = 0

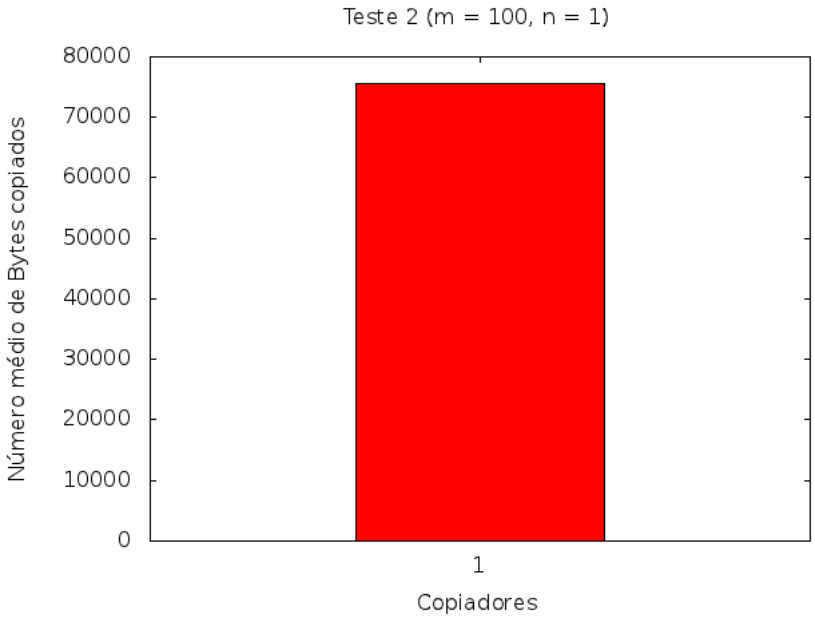

Teste 1 (m = 10, n = 1)
Número médio de Bytes copiados
8000
7000
6000
5000
4000
3000
2000
1000
0
1
Copiadores

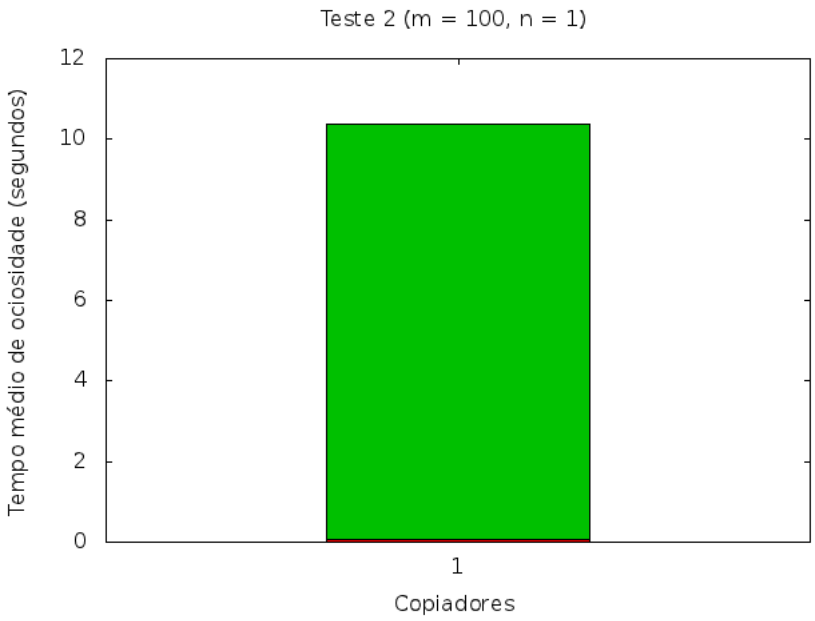

Teste 1 (m = 10, n = 1)
Tempo médio de ociosidade (segundos)
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
Copiadores

5.2 Teste 2 - M = 100, N = 1

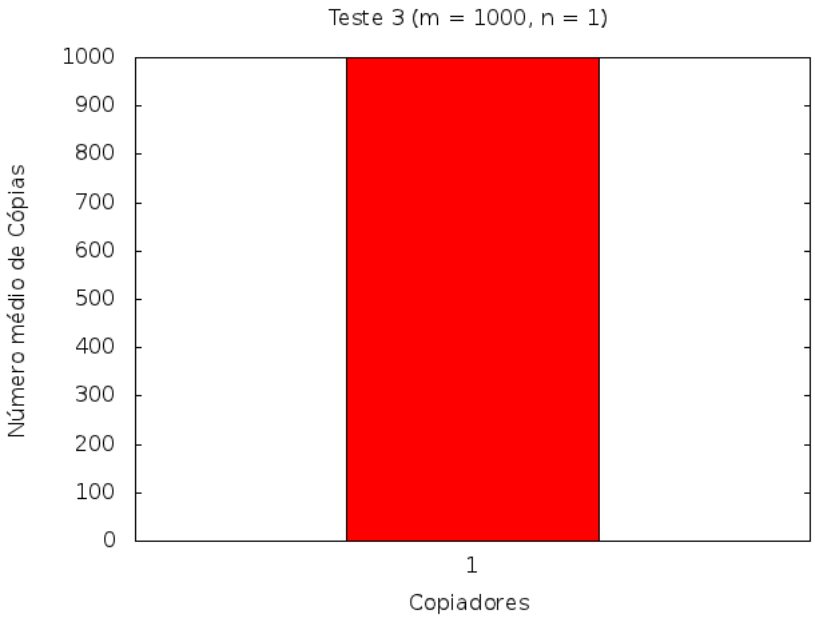


MAX CÓPIAS = 100
MIN CÓPIAS = 0

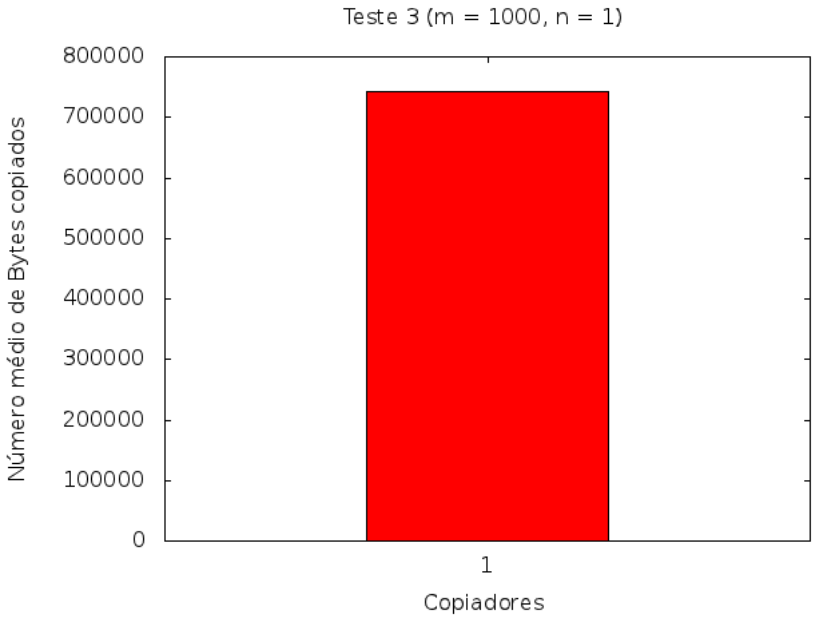


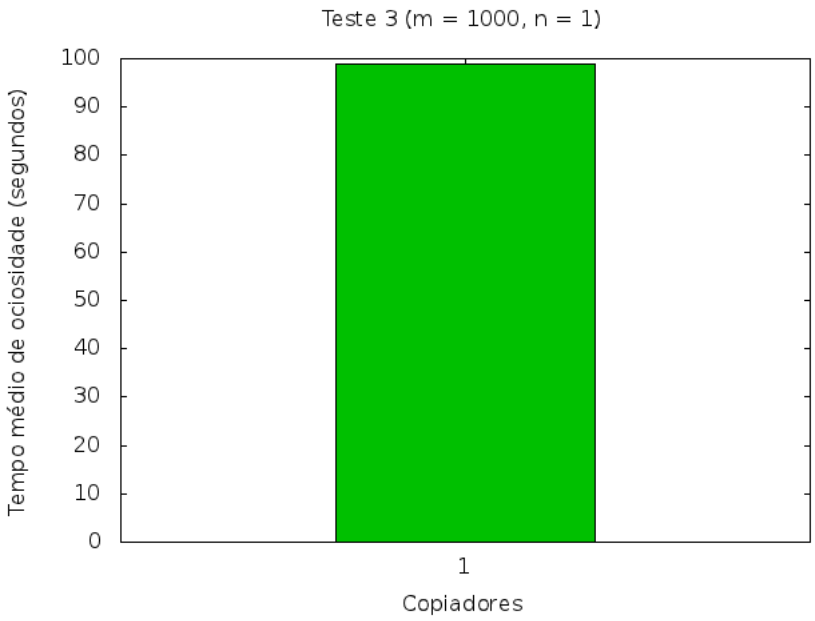

Teste 2 (m = 100, n = 1)
Tempo médio de ociosidade (segundos)
12
10
8
6
4
2
0
1
Copiadores

5.3 Teste 3 - M = 1000, N = 1

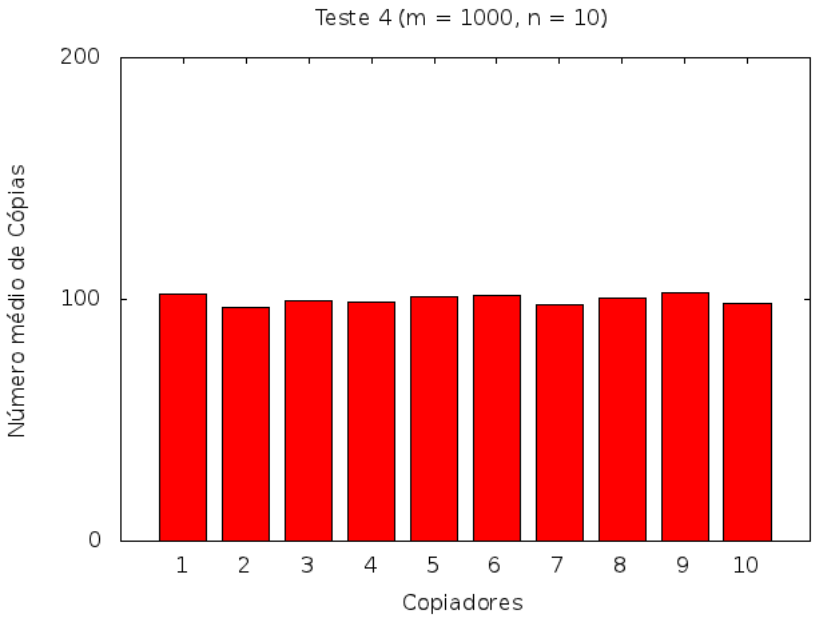


MAX CÓPIAS = 1000
MIN CÓPIAS = 0

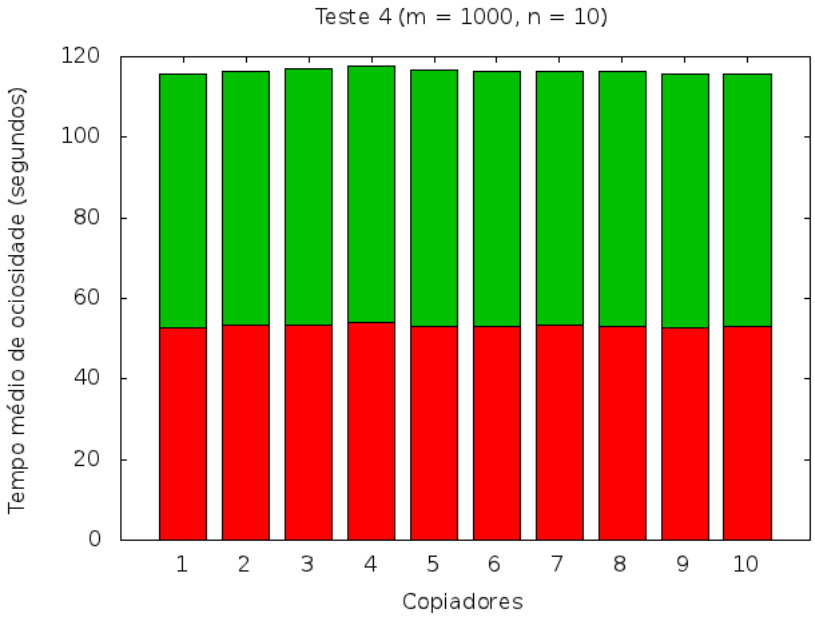



Teste 3 (m = 1000, n = 1)
Tempo médio de ociosidade (segundos)
100
90
80
70
60
50
40
30
20
10
0
1
Copiadores

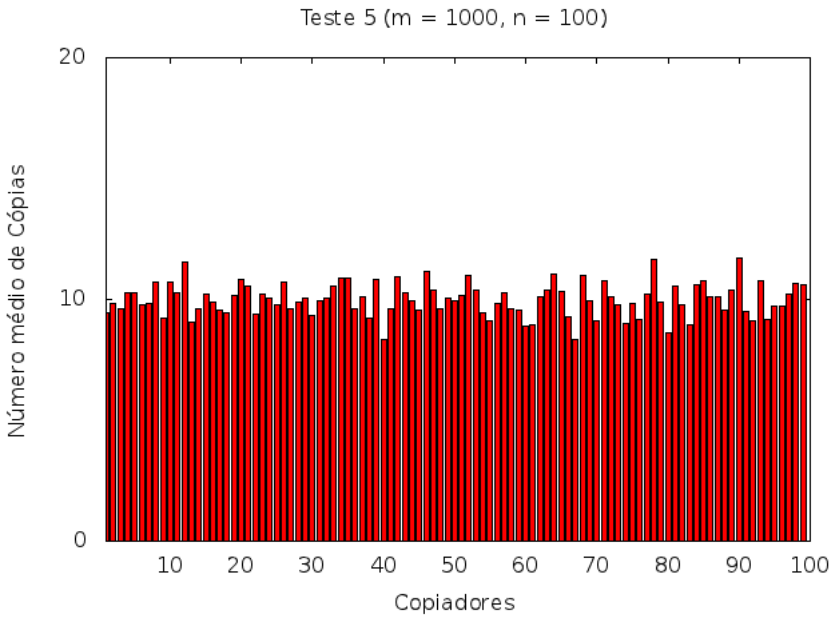
5.4 Teste 4 - M = 1000, N = 10



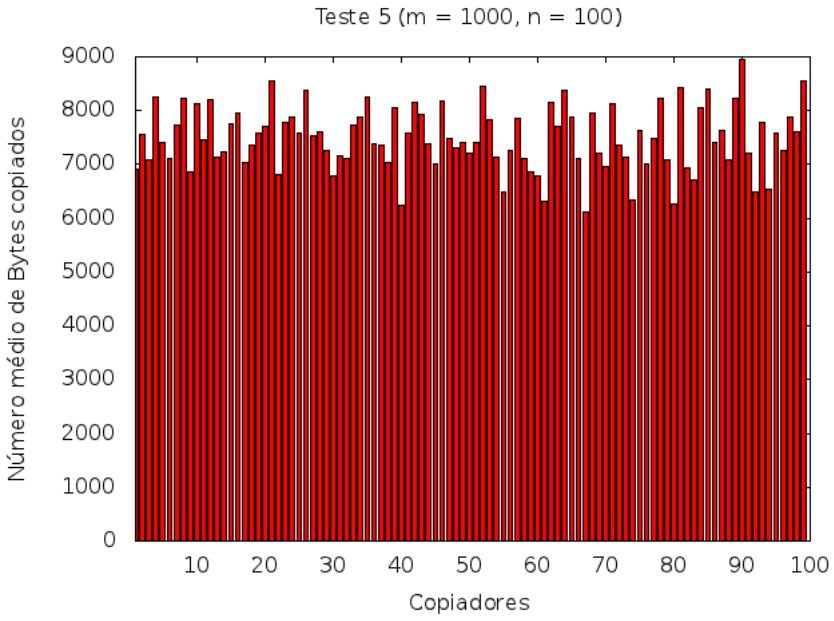
MAX CÓPIAS = 127
MIN CÓPIAS = 0

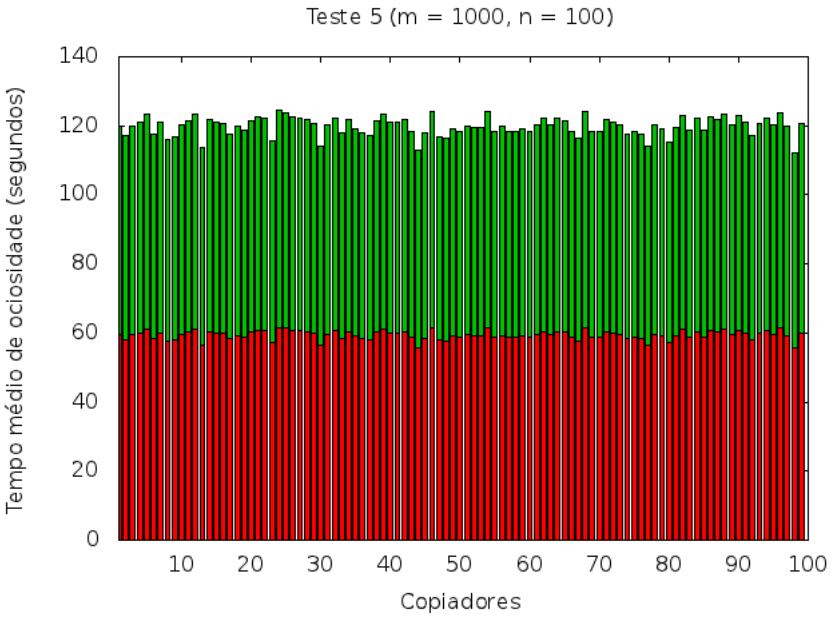

Teste 4 (m = 1000, n = 10)
Tempo médio de ociosidade (segundos)
120
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
10
Copiadores

5.5 Teste 5 - M = 1000, N = 100

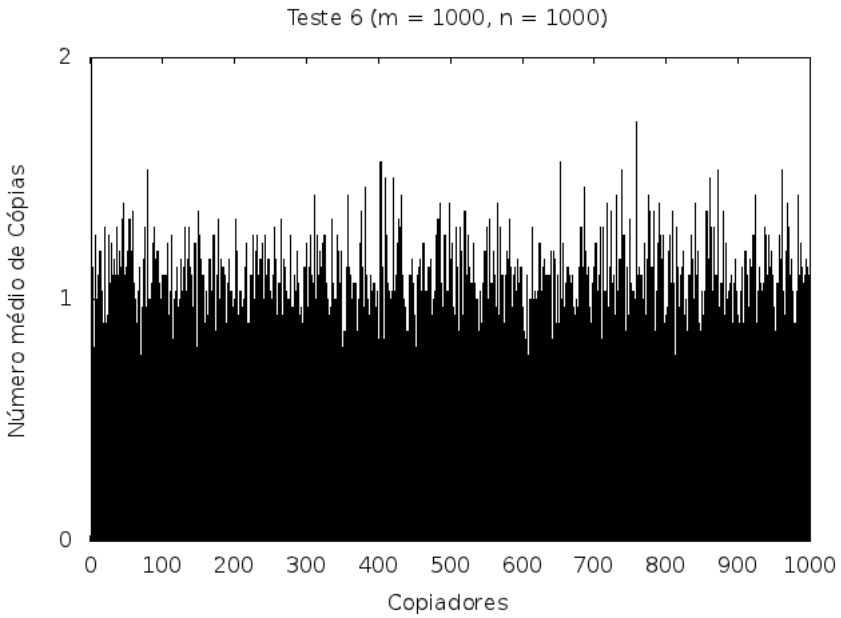


MAX CÓPIAS = 23
MIN CÓPIAS = 0

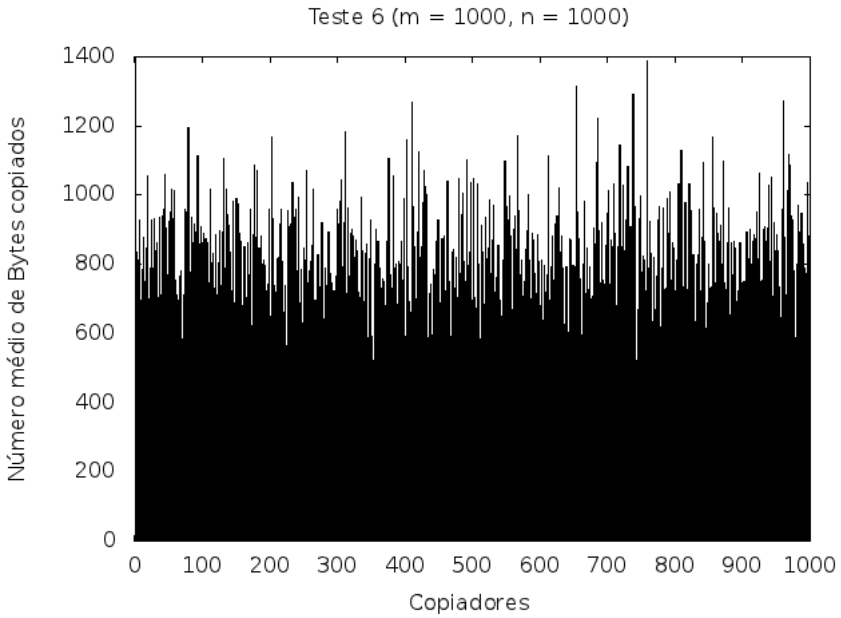


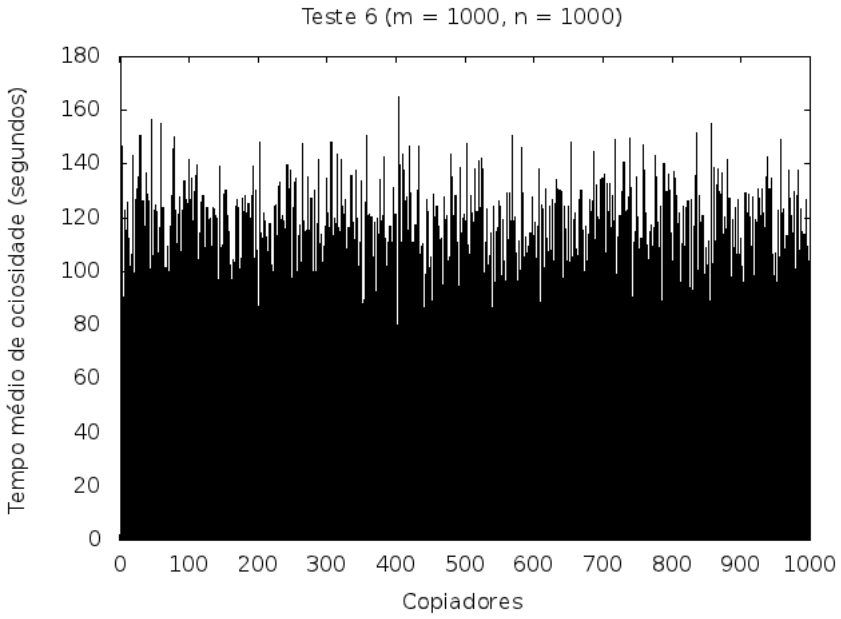

Teste 5 (m = 1000, n = 100)
Tempo médio de ociosidade (segundos)
Copiadores
0
20
40
60
80
100
120
140
10
20
30
40
50
60
70
80
90
100

5.6 Teste 6 - M = 1000, N = 1000

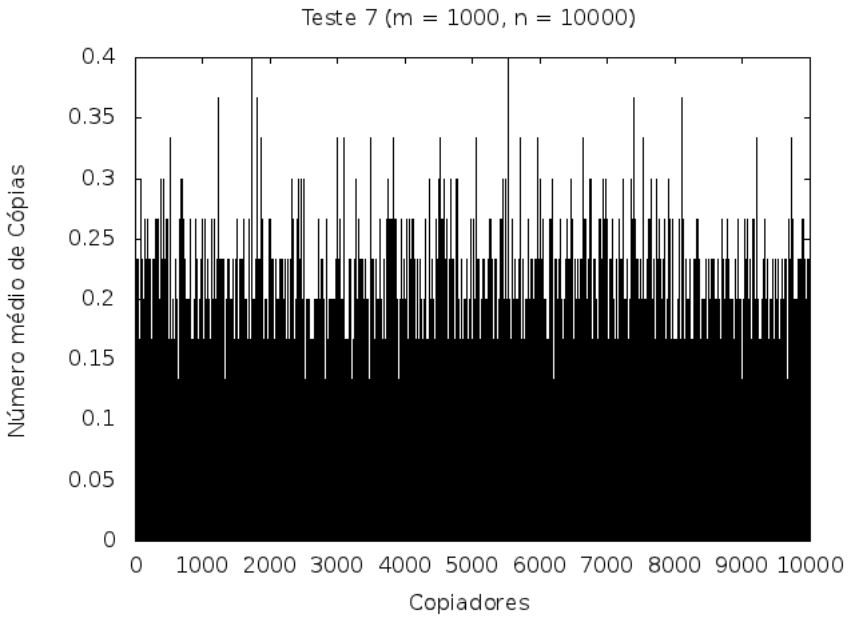


MAX CÓPIAS = 7
MIN CÓPIAS = 0

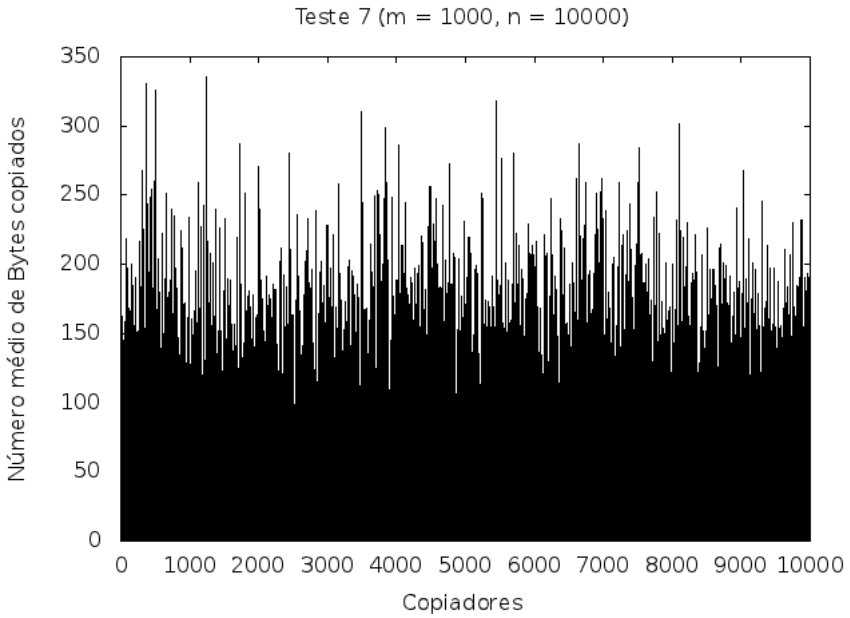


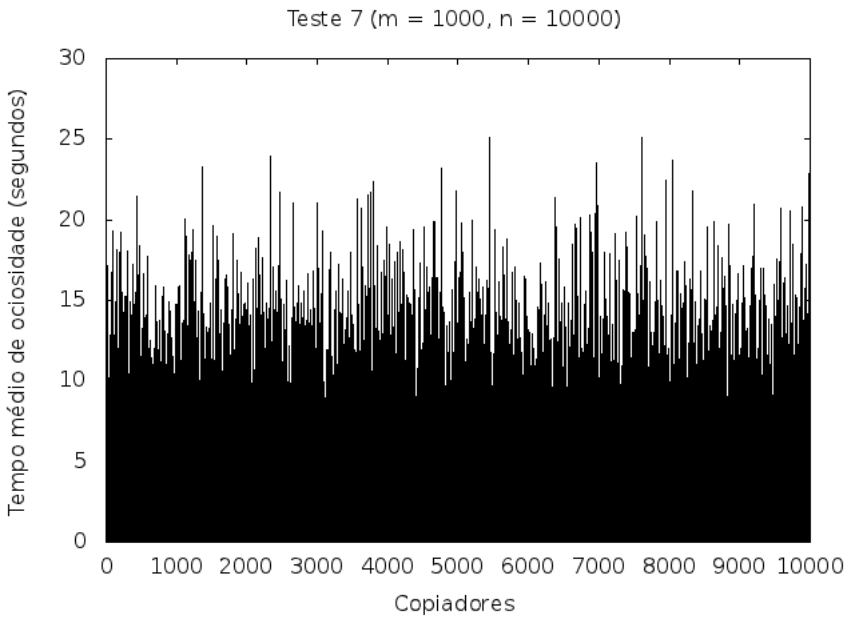

Teste 6 (m = 1000, n = 1000)
Tempo médio de ociosidade (segundos)
180
160
140
120
100
80
60
40
20
0
0
100
200
300
400
500
600
700
800
900
1000
Copiadores

5.7 Teste 7 - M = 1000, N = 10000



MAX CÓPIAS = 5
MIN CÓPIAS = 0




Teste 7 (m = 1000, n = 10000)
Tempo médio de ociosidade (segundos)
30
25
20
15
10
5
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Copiadores

6 Conclusões

Durante a etapa de codificação do programa, percebemos que foi necessário adicionar o comando "usleep" nos processos rodando em paralelo. Isso fazia com que os processos não sobrecarregassem a CPU do computador com condicionais desnecessárias.
Procuramos estabelecer um valor de *sleep* adequado para não prejudicar a eficiência do algoritmo e gerar ociosidade articial e indesejada.
Ao realizar os testes, percebemos também que era necessário limitar o tamanho da pilha de cada thread, caso contrário não era possível alocar memória para todas elas. Para tanto, utilizamos o comando do Linux *ulimit*.