UNIVERSIDADE FEDERAL DO RIO DE JANEIRO – UFRJ

INSTITUTO DE MATEMÁTICA – IM
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO – DCC

ESTUDO SOBRE THREADS

Disciplina: Sistemas Operacionais
Professor: Thomé

Júlio César Machado Bueno	106033507
Luiza Diniz e Castro	107362705
Roberta Santos Lopes	107362886

Sumário

Estudo de Comandos

pthread_create()

A função *pthread_create()* é uma função que possibilita a criação de uma nova *thread* a partir de parâmetros configurados na inicialização das estruturas *pthread_attr_t* e *pthread_t* utilizando a implementação POSIX *thread* presente no cabeçalho *pthread.h*.

Então, como o primeiro passo para a criação da thread, devemos criar a variável do tipo *pthread_t*, em seguida configurar os parâmetros que definem a criação e execução da *thread* a ser criada. A *thread* criada em C executa uma função definida no código. Dentro deste código, variáveis locais são acessíveis apenas a *thread* com exceção dos recursos compartilhados. Ao contrário da criação de um subprocesso, onde o subprocesso executa todo o código do processo pai posterior a parte em que foi chamada a criação do subprocesso, a *thread* executa somente o código definido pela função especificada.

Para a execução da função *pthread_create* é necessário a passagem dos argumentos:

- Variável do tipo *pthread_t*
- Variável do tipo *pthread_attr_t* (*NULL* se não há execução de *pthread_attr_init()*)
- Função a ser executada como um *thread* distinta
- Ponteiro para a variável passada como argumento para a função

A função *pthread_create()* retorna o valor 0 (zero) se a *thread* foi criada com sucesso. Caso contrário, retorna o número do erro associado a falha da criação da *thread*. A seguir um exemplo de uso da função *pthread_create():*

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

//Inclui o cabeçalho para a criação da thread
#include <pthread.h>

int print(void *arg){
   printf("%s", arg);
}

int main(void){
   //Cria os valores da thread
   pthread_t thread;
   int retorno;

   //Define o argumento a ser passado
   char string[256] = "Mensagem exibida pela Thread criada!";

   //Cria de fato a thread
   retorno = pthread_create(&thread, NULL, print, (void*) &string);

   //Verifica se houve um erro ao criar a thread
   if(retorno != 0){
      printf("Erro número %i ao criar a thread", retorno);
      exit(1);
   }

   exit(0);
}
```

pthread_join()

Ao executar a função *pthread_join()*, uma *thread* fica suspensa enquanto outra *thread*, passada como parâmetro não é terminada ou cancelada. Ou seja, a *thread* "pai" fica aguardando o término ou o cancelamento da *thread* "filho". *pthread_join()* tem então a mesma funcionalidade que *wait()* sendo então responsável pela sincronização entre as *threads*.

pthread_join() necessita dos seguintes de um argumentos:

- *pthread_t* utilizado na função *pthread_create()*.
- Valor de retorno da t*hread* (*thread_return*), Usualmente *NULL*.

A seguir um exemplo da execução de *pthread_join()*:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

//Inclui o cabeçalho para a criação da thread
#include <pthread.h>

int print(void *arg){
   printf("%s", arg);
}

int main(void){
   //Cria os valores da thread
   pthread_t thread;
   int retorno;

   //Define o argumento a ser passado
   char string[256] = "Mensagem exibida pela Thread criada!";

   //Cria de fato a thread
   retorno = pthread_create(&thread, NULL, print, (void*) &string);

   //Verifica se houve um erro ao criar a thread
   if(retorno != 0){
      printf("Erro número %i ao criar a thread", retorno);
      exit(1);
   }

   //Aguarda o retorno da thread criada
   pthread_join(&thread, NULL);

   exit(0);
}
```

pthread_exit()

A função *pthread_exit()* termina a execução da *thread* corrente. Caso a *thread* "pai" tenha executado o comando *pthread_join()* para esta *thread* "filho" em questão, *pthread_exit()* pode retornar um valor que será disponível à *thread* "pai". Como padrão, retorna-se o valor *NULL*.

O valores das variáveis locais são liberados na chamada de *pthread_exit()*. Já recursos compartilhados continuam disponíveis no sistema mesmo que a sua *thread* foi terminada.

A seguir o exemplo do comnado *pthread_exit()*:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

//Inclui o cabeçalho para a criação da thread
#include <pthread.h>

int print(void *arg){
   printf("%s", arg);
   pthread_exit(NULL);
}

int main(void){
   //Cria os valores da thread
   pthread_t thread;
   int retorno;

   //Define o argumento a ser passado
   char string[256] = "Mensagem exibida pela Thread criada!";

   //Cria de fato a thread
   retorno = pthread_create(&thread, NULL, print, (void*) &string);

   //Verifica se houve um erro ao criar a thread
   if(retorno != 0){
      printf("Erro número %i ao criar a thread", retorno);
      exit(1);
   }

   //Aguarda o retorno da thread criada
   pthread_join(&thread, NULL);

   exit(0);
}
```

pthread_attr_init()

Inicializa atributos padrões para uma *thread* que será criada. É importante para a configuração dos atributos de execução da *thread* de forma flexível.

pthread_attr_init() recebe como um parâmtro um endereço de memória para uma variável do tipo *pthread_attr_t*. É possível iniciar uma nova *thread* com *pthread_create* sem inicializar atributos para a mesma. Nesse caso, o SO configura valores padrões para a nova *thread*. A seguir uma tabela que informa os valores padrões do SO para a criação de uma *thread*.

Atributo	Padrão
Detach state	PTHREAD_CREATE_JOINABLE
Scope	PTHREAD_SCOPE_PROCESS
Inherit scheduler	PTHREAD_INHERIT_SCHED
Scheduling policy	SHCED_OHTER
Scheduling order	0
Guard size	4096 bytes
Stack address	#Address
Stak size	#Address size in bytes

No exemplo a seguir, como não há modificação dos atributos, a execução de *pthread_create()* somente com o atributo inicializado como parâmetro é a mesma que execução com o argumento *NULL* como parâmetro:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

//Inclui o cabeçalho para a criação da thread
#include <pthread.h>

int print(void *arg){
   printf("Mensagem exibida pela Thread criada!");
   pthread_exit(NULL);
}

int main(void){
   //Cria os valores da thread
   pthread_t thread;
   pthread_attr_t atributosThread;
   int retorno;

   //Inicializa os atributos da thread
   pthread_attr_init(&atributosThread);

   //Cria de fato a thread
   retorno = pthread_create(&thread, &atributosThreads, print, NULL);

   //Verifica se houve um erro ao criar a thread
   if(retorno != 0){
      printf("Erro número %i ao criar a thread", retorno);
      exit(1);
   }

   //Aguarda o retorno da thread criada
   pthread_join(&thread, NULL);

   exit(0);
}
```

pthread_attr_setscope()

Configura o atributo *SCOPE* da *thread* a ser iniciada. Desta forma configura o nível de execução da nova *thread*, se será no nível *kernel* ou no nível usuário. *pthread_attr_setscope()* faz parte da família de funções *pthread_attr_set*()* que configura os parâmetros da *thread* a ser criada. Ela recebe os seguintes itens como parâmetro.

- Endereço de memória para a variável do tipo *thread_attr_t*
- Constante do SO que definem o nível de execução podendo ser:
  - PTHREAD_SCOPE_PROCESS – execução em nível usuário, padrão do SO.
  - PTHREAD_SCOPE_SYSTEM – execução em nível *kernel*.

Configuração do atributo *SCOPE* através de pthread_attr_setscope*()* para a criação de uma *thread* de nível *kernel*:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

//Inclui o cabeçalho para a criação da thread
#include <pthread.h>

int print(void *arg){
   printf("%s", arg);
   pthread_exit(NULL);
}

int main(void){
   //Cria os valores da thread
   pthread_t thread;
   pthread_attr_t atributosThread;
   int retorno;

   //Define o argumento a ser passado
   char string[256] = "Mensagem exibida pela Thread criada!";

   //Inicializa os atributos da thread
   pthread_attr_init(&atributosThread);

   //Configura a thread como thread do nível kernel
   pthread_attr_setscope&atributosThread, PTHREAD_SCOPE_SYSTEM);

   //Cria de fato a thread
   retorno = pthread_create(&thread, &atributosThreads, print, (void*) &string);

   //Verifica se houve um erro ao criar a thread
   if(retorno != 0){
      printf("Erro número %i ao criar a thread", retorno);
      exit(1);
   }

   //Aguarda o retorno da thread criada
   pthread_join(&thread, NULL);

   exit(0);
}
```

Prog1

Funcionamento

O Prog1 utiliza a criação de *threads* para o cálculo do Produto Interno descrito na especificação do trabalho utilizando os comandos pthread_create*(),* pthread_join*(), pthread_exit()* associados ao uso de:

- Comunicação inter-*threads* (através de *POSIX*).
- Manipulação de sequência de execução inter-*threads* (através do *pthread_join()*).
- Término e retorno de *threads*.

Configuração da máquina utilizada no desenvolvimento e testes

As implementações e testes foram realizados nas máquinas do LCI-UFRJ onde, de acordo com os testes especificados, 1 dos núcleos foi habilitado ou não e que apresentam a seguinte configuração:

Intel Core 2 Duo E7300 2,66GHz (2 núcleos) 2 GB Memória RAM HD 40 GB Sistema Operacional: Linux Gentoo World

Comparação de desempenho entre as versões

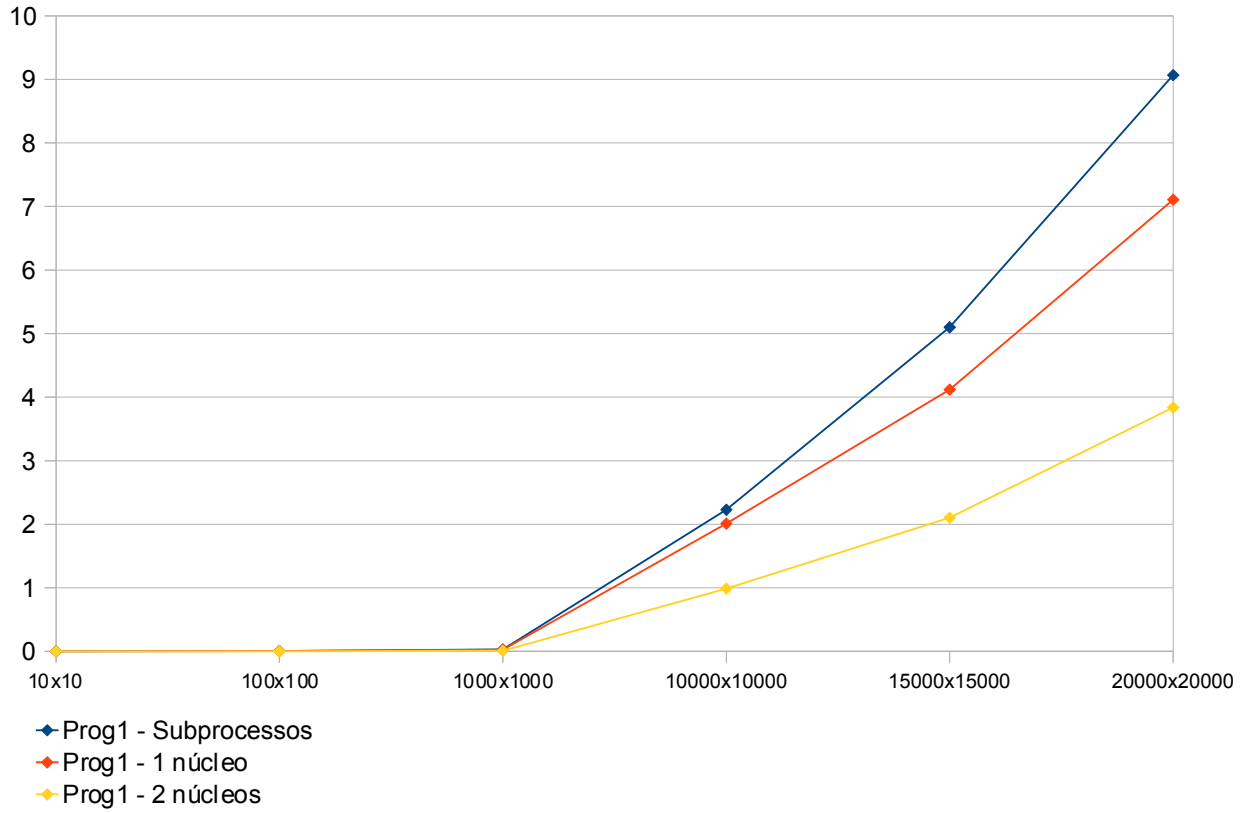
A seguir, temos os valores dos testes de performance realizados comparativamente entre os processos. Para fim de comparações equivalentes, utilizamos valores de *m* e *k* iguais e seus incrementos de forma a deixar claro a natureza de cada versão do programa. É importante perceber que para valores de *m* e *k* menores que 1000 a diferença de performance entre os programas é irrelevante. Foi utilizado a versão 3 do Prog1 do trabalho anterior como base comparativa, dado que ela apresenta a melhor implementação feita que utiliza subprocessos.

Os resultados obtidos através da execução das versões dos programas estão listados na tabela abaixo:

Dimensão da Matriz *m* x *k*	**Versão 1** Subprocessos Tempo (segundos)	**Versão 2** 1 núcleo habilitado Tempo (segundos)	**Versão 3** 2 núcleos habilitados Tempo (segundos)
10 X 10	0,000	0,000	0,000
50 X 50	0,004	0,004	0,000
100 X 100	0,004	0,004	0,001
500 X 500	0,009	0,007	0,003
1000 X 1000	0,028	0,025	0,009
5000 X 5000	0,574	0,414	0,314
10000 X 10000	2,227	2,009	0,987
15000 X 15000	5,099	4,119	2,103
20000 X 20000	9,068	7,108	3,837

Análise dos resultados e conclusões

O gráfico a seguir mostra os mesmo valores da tabela anterior para que possa ser visível a diferença de performance entre cada programa, assim como o seu comportamento diante da variação da entrada de *m* e *k*.



Desta forma podemos então concluir que o uso de *threads* se mostra mais eficiente que o uso de subprocessos em máquinas com 1 núcleo e mais claramente em máquinas que possuem 2 ou mais núcleos. Também podemos observar que a criação de um número muito grande *threads* é prejudicial a dinâmica do sistema, portanto havendo um número ótimo de *threads* a serem iniciadas para que haja um desempenho ótimo da aplicação. Isso comprova o que se é pensado em teoria com relação a criação de *threads* ser menos onerosa que a de subprocessos, gerando menos *overhead* já que a estrutura de PCB não é copiada a cada nova criação. Também fica mais claro que a alternância entre *threads* é mais rápida que entre subprocessos.

Prog2

Especificação geral

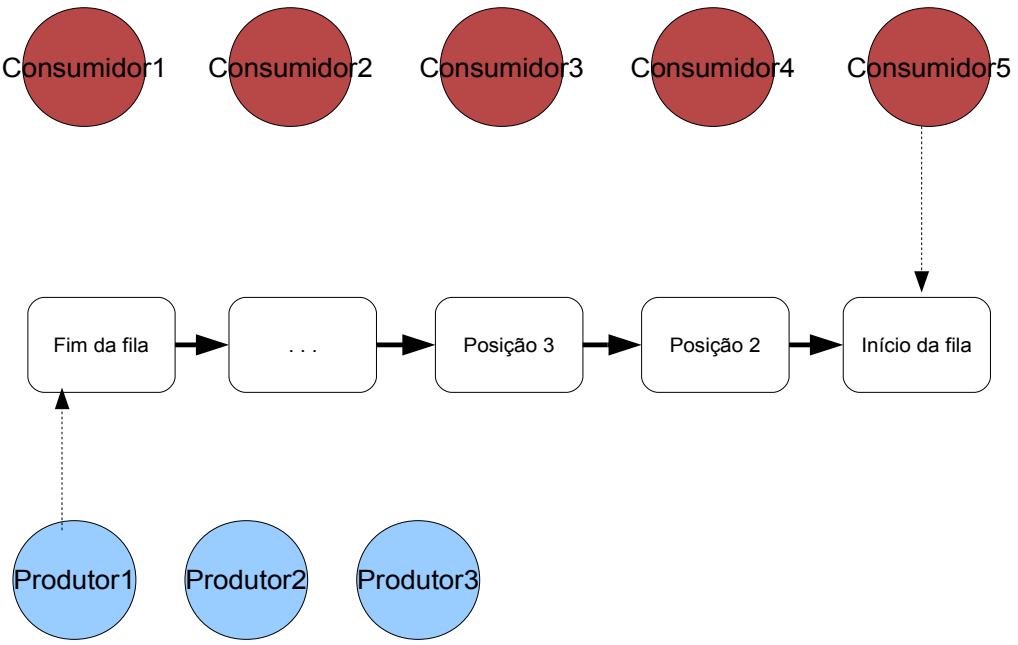
O Prog2 foi criado para ser uma simulação de um problema conhecida na literatura como Sistema Produtor/Consumidor. Neste problema temos disposto *m* entidades chamada de Produtor e *n* entidades chamadas de Consumidores. Também temos no sistema uma terceira entidade chamada de Recurso ou Produto. Nesse sistema, cada Produtor é responsável por produzir de forma independente das demais entidades, uma unidade do Recurso em tempo constante e finito. Da mesma forma, um consumidor deve consumir uma unidade do Recurso também em tempo constante e finito.

A dinâmica deste sistema se dá pela forma em que os Produtores se organizam para produzir, como os Recursos são armazenados e disponibilizados aos Consumidores e como estes se dispõem para consumir os Recursos.

Dado as especificações do Trabalho temos a seguinte dinâmica a ser implementada:

- 3 Produtores e 5 Consumidores.
- Produção de 1000 Recursos no total.
- Recursos dispostos em um fila com 50 posições.
- Produtores inserem no final da fila.
- Consumidores consomem no início da fila.
- Ambiente assíncrono.

Para ilustrar como essa dinâmica se dá podemos esboçar a seguinte figura:

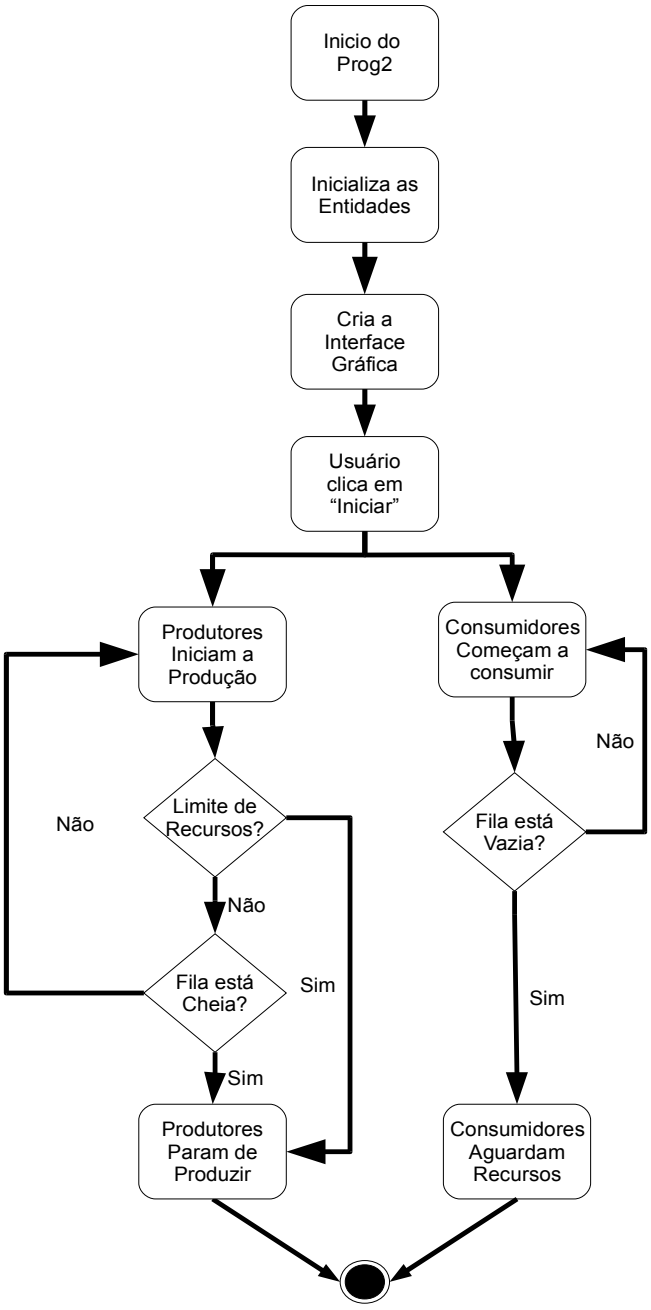


A especificação do Programa2 também requeria que a simulação contasse com uma interface gráfica que exibisse os valores relacionados a implementação. Desta forma, diferente dos demais programas, foi escolhida a linguagem ***Java*** por possuir uma fácil implementação de interface gráfica entre outros fatores.

Versão 1

Estrutura e Análise

A versão 1 do Prog2 determina a execução da simulação através do seguinte fluxo de execução. É importante lembrar que, dado em um ambiente assíncrono, este fluxo tem como objetivo meramente exemplificar uma execução.



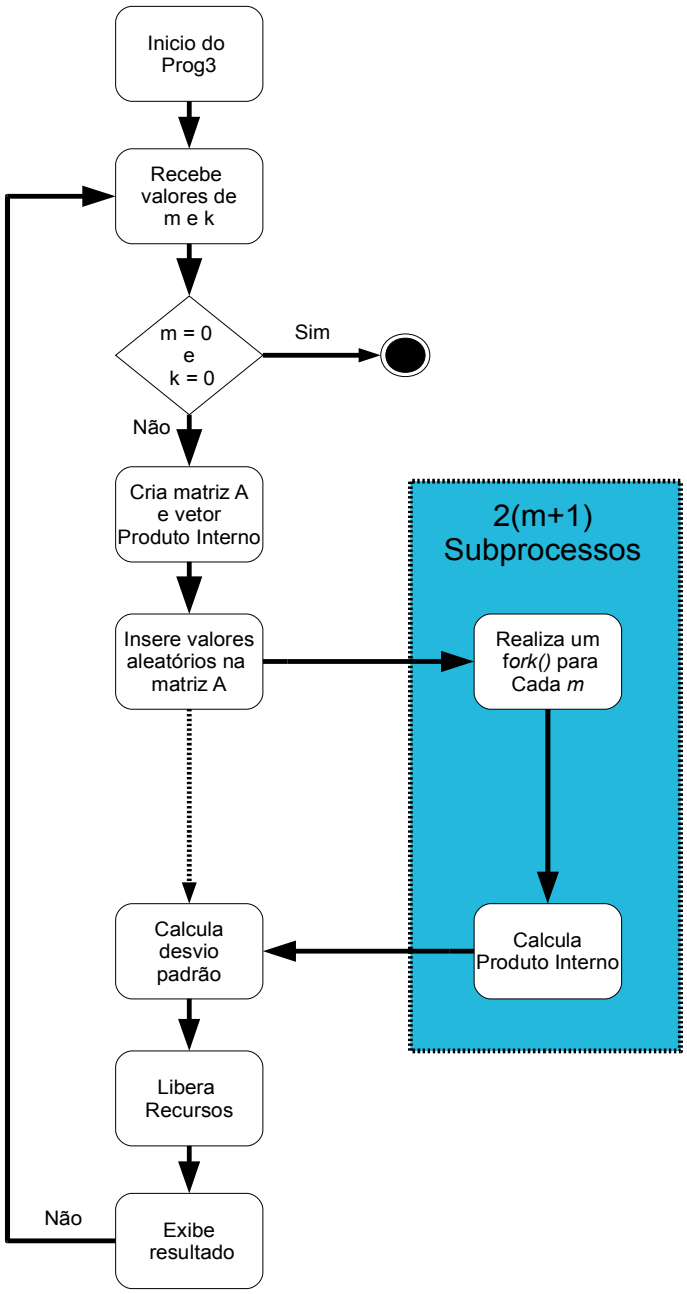
Dado que o Prog3a não possui nenhuma técnica de processamento paralelo, podemos esperar que o seu desempenho seja caracterizado pela sensibilidade aos valores de *m* e *k*. *P*ortanto, a fim de estabelecermos uma notação de comparação, podemos dizer que o algoritmo possui a complexidade de $O(n^2)$.

Versão 2

Estrutura e Análise

A versão 2 do Prog3 (ou Prog3b) tem como base a versão do Prog3a. Nela, é feita uma tentativa de melhoria de performance e aproveitamento de recursos. Para isso, a cada cálculo do Produto Interno foi criado um processo para calcular uma determinada linha *m* da matriz A gerada.

A seguir o fluxograma do programa apresentado:



Devido ao elevado número de subprocessos gerados no Prog3b, observou-se que a tática adotada para melhorar a performance da implementação anterior (Prog3a), na realidade piora os resultados obtidos. Tal fato ocorre devido aos seguintes fatores:

- O Sistema Operacional precisa alocar recursos, como o time-slice, para todos os subprocessos criados. Isso prejudica a performance da tarefa, devido ao elevado overhead do SO. Em outras palavras, o Sistema Operacional gasta mais tempo nas rotinas de controle dos subprocessos criados do que na execução da tarefa.

- Como há muitos processos concorrentes, há também um grande número de trocas de

contexto.

Desta forma podemos concluir que a implementação é altamente sensível a aumentos no valor de *m*. Sendo assim, a versão Prog3b não se mostrou proveitosa, necessitanto de uma outra abordagem, descrita a seguir.

Apêndice

A seguir disponibilizamos os códigos fonte de todos os programas utilizados no estudo.

Prog1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <sys/timeb.h>

#include <pthread.h>

#define INF 0x33333333
#define THREADS 2

int **matriz, *produtoInterno;
int i, j, k, m, menor, maior, menor_i, maior_i, menor_matriz, maior_matriz,
menor_matriz_i, maior_matriz_i, menor_matriz_j, maior_matriz_j;
int parte[THREADS], returnThread[THREADS];
double soma, soma_desvio, desvio_padrao, tempo_execucao;
struct timeb inicio_execucao, fim_execucao;
pthread_t thread[THREADS];
pthread_attr_t atributosThread;

int **aloca_matriz(int m, int k) {
      //ponteiro para a matriz e variável de iteração
      int **v, i;

      //Veririfica os parâmetros
      if(m < 1 || k < 1 ){
            printf ("\n\nErro: Valores de m e k inválidos!\n\n");
            exit(1);
      }

      //Aloca a linha da matriz
      v = (int **) calloc (m, sizeof(int *));
      if(v == NULL){
            printf ("\n\nErro: Memória insuficiente!\n\n");
            exit(1);
      }

      //Aloca as colunas
      for( i = 0; i < m; i++ ){
            v[i] = (int*) calloc (k, sizeof(int));
            if(v[i] == NULL){
                  printf ("\n\nErro: Memória insuficiente!\n\n");
                  exit(1);
            }
      }

      //Retorna a matriz
      return (v);
}
```

```
int **free_matriz(int m, int k, int **v){
      int  i;
      if(v == NULL){
            exit(1);
      }

      //Verifica parâmetros
      if(m < 1 || k < 1){
            printf("\n\nErro: Parâmetro invalido!\n\n");
            return(v);
      }

      //Libera as linhas da matriz
      for(i=0; i<m; i++){
            free(v[i]);
      }

      //Libera a matriz
      free(v);
      return (NULL);
}


int *aloca_vetor(int m){
      //ponteiro do vetor
      int *v;

      //Veririfica o parâmetro
      if(m < 1){
            printf ("\n\nErro: Parâmetro invalido!\n\n");
            exit(1);
      }

      //Aloca a linha da matriz
      v = (int *) calloc (m+1, sizeof(int *));
      if(v == NULL){
            printf ("\n\nErro: Memória insuficiente!\n\n");
            exit(1);
      }

      //Retorna o vetor
      return (v);
}


void *calculaProdutoInterno(void *arg){
      int i, j, *pos, inicio, fim, somatorio;

      pos = (int *) arg;
      inicio = (*pos - 1)*floor(m/THREADS);
      fim = inicio + floor(m/THREADS) ;

      //Detecta se a thread é a última. Nesse caso precisa calcular também o resto
da divisão
      if( m-fim < THREADS ){
            fim = m;
      }

      //Calcula o somatório
      for(i=inicio; i<fim; i++){
            somatorio = 0;

            for(j=0; j<k; j++){
```

```
                //Realiza o produto interno
                somatorio += matriz[i][j] * matriz[i][j];
          }

          //Armazena o PI(i) e soma para cálculo de média
          produtoInterno[i] = somatorio;
          soma += produtoInterno[i];
     }

     printf("\nThread %i terminou de calcular as linhas entre %i e %i", *pos,
inicio+1, fim);

     //Termina a thread
     pthread_exit(NULL);
}


int main(void){

     //Recebe os valores iniciais de m e k
     printf("Defina o número de linhas  -> m = ");
     scanf("%i", &m);
     printf("Defina o número de colunas -> k = ");
     scanf("%i", &k);

     //Inicia o parâmetro da contagem de tempo
     srand((unsigned)time(NULL));

     //Configura os atributos da Thread
     pthread_attr_init(&atributosThread);
     pthread_attr_setdetachstate(&atributosThread, PTHREAD_CREATE_JOINABLE);
     //No caso, tipo Kernel
     pthread_attr_setscope(&atributosThread, PTHREAD_SCOPE_SYSTEM);

     while( m!= 0 && k!=0 ){
          //Inicializa outros valores da iteração
          menor = INF;
          maior = -INF;
          menor_matriz = INF;
          maior_matriz = -INF;
          menor_i = 0;
          maior_i = 0;
          menor_matriz_i = 0;
          maior_matriz_i = 0;
          menor_matriz_j = 0;
          maior_matriz_j = 0;
          soma = 0;
          soma_desvio = 0;
          desvio_padrao = 0;

          //Inicia a contagem do tempo de execução
          //ftime(&inicio_execucao);

          //Aloca a matriz e vetor de Produto Interno
          printf("\nMontando a matriz... ");
          matriz = aloca_matriz(m, k);
          produtoInterno = aloca_vetor(m);
          printf("Concluído!\n\n");

          //Gera a matriz
          printf("Inserindo valores na Matriz... ");
          for(i=0; i<m; i++){
               for(j=0; j<k; j++){
                    //gera o número aleatório e armazena na matriz
```

```
                        matriz[i][j] = (rand()%201)-100;

                        //Detecta o maior e menor na matriz
                        if(matriz[i][j] <= menor_matriz){
                              menor_matriz = matriz[i][j];
                              menor_matriz_i = i+1;
                              menor_matriz_j = j+1;
                        }
                        if(matriz[i][j] >= maior_matriz){
                              maior_matriz = matriz[i][j];
                              maior_matriz_i = i+1;
                              maior_matriz_j = j+1;
                        }
                  }
            }
            printf("Concluído!\n\n");


            printf("Calculando a Produto Interno...");
            //Neste programa estamos calculando o tempo necessário para realizar o
produto interno utilizando threads.
            //pega o tempo inicial da execução
            ftime(&inicio_execucao);

            //Loop que cria as threads
            for(i=0; i<THREADS; i++){

                  //Determina o posicionamento da Thread na matriz
                  parte[i] = i+1;

                  //Inicializa as threads passando o parâmetro da sua posição
                  returnThread[i]  =  pthread_create(&(thread[i]),  &atributosThread,
calculaProdutoInterno, (void*) &(parte[i]));

                  //Verifica se houve um erro ao criar a Thread
                  if(returnThread[i] > 0){
                        printf("Não foi possível criar a Thread para o segmento i=
%i", i);
                        exit(1);
                  }

            }

            //Aguarda o término das threads
            for(i=0; i<THREADS; i++){
                  pthread_join(thread[i], NULL);
            }
            //pega o tempo final da execução
            ftime(&fim_execucao);
            printf("Concluído!\n\n");


            //Calcula o desvio padrão
            for(i=0; i<m; i++){
                  soma_desvio += pow(produtoInterno[i]-(soma/m), 2);

                  //Detecta o maior e menor
                  if(produtoInterno[i] <= menor){
                        menor = produtoInterno[i];
                        menor_i = i+1;
                  }
                  if(produtoInterno[i] >= maior){
                        maior = produtoInterno[i];
                        maior_i = i+1;
```

```
                }
            }
            desvio_padrao = sqrt(soma_desvio/m);


            //Calcula o tempo de execução
            //ftime(&fim_execucao);
            tempo_execucao                  =                  (((fim_execucao.time-
inicio_execucao.time)*1000.0+fim_execucao.millitm)-inicio_execucao.millitm)/1000.0;


            //Libera as linhas da matriz
            free_matriz(m, k, matriz);
            free(produtoInterno);

            //Exibe os valores resultantes
            printf("---------------------------------------Valores
Aferidos---------------------------------------\n");
            printf("Menor valor na matriz = %i (i=%i, j=%i) e Maior valor na matriz
= %i (i=%i, j=%i)\n", menor_matriz, menor_matriz_i, menor_matriz_j, maior_matriz,
maior_matriz_i, maior_matriz_j);
            printf("Menor valor Produto Interno = %i (m=%i) e Maior valor Produto
Interno = %i (m=%i)\n", menor, menor_i, maior, maior_i);
            printf("Desvio Padrão = %f\n", desvio_padrao);
            printf("Tempo de execução = %.3f segundos\n", tempo_execucao);


printf("--------------------------------------------------------------------------------
--------------------\n\n");

            //Recebe os valores de m e k para nova iteração
            printf("Nova iteração\n");
            printf("Defina o número de linhas  -> m = ");
            scanf("%i", &m);
            printf("Defina o número de colunas -> k = ");
            scanf("%i", &k);
      }

      exit(0);
}
```

Prog2.c

```
#include <stdio.h>
```

```
#include <stdlib.h>

#include <math.h>

#include <string.h>

#include <sys/wait.h>

#include <unistd.h>


#define m 1


int i, j, k, id, d1, d2, status;

int nFilhos = 0;

int meuPID = -1;

char pid[16];

char pid_list[512];


int main(void) {


            //inicialize as variáveis d1 e d2 com valores distintos;

            d1 = 0;

            d2 = 1;




            printf("\n");

            //mostre o PID do processo corrente e os valores de d1 e d2 na
tela da console

            printf("PID do processo corrente = %i    |    d1 = %i    |    d2 =
%i\n\n", getpid(), d1, d2);




            /*

==========================================================================
=====================================

            Responda: Quais processos executarão este trecho do código?
```

```
                    Somente o processo pai original ("raiz da árvore de
processos") executará este trecho.

              ------------------------------------------------------------------
-------------------------------------------------

              */



              j = 0;



              for (i = 0; i <= m; i++){

                 //mostre na tela da console, a cada passagem, os seguintes
valores: PID do processo corrente; “i”, “d1”, “d2” e “m”

                 printf("PID do processo corrente = %i     |     d1 = %i     |
d2 = %i    |   m = %i\n\n", getpid(), d1, d2, m,i);



                 /*


==================================================================================
==================================

                 Responda: Quais processos executam este trecho do código?

                 Todos os processos executam este trecho já que neste trecho o
fork() ainda não foi realizado.

                 ------------------------------------------------------------------
-------------------------------------------------

                 */

                 id = fork();



                 //Verifica validade da atualização

                 if(meuPID != getpid()){

                      meuPID = getpid();

                      nFilhos = 0;

                      //Limpa o meu vetor

                      memset(pid_list, 0, sizeof(pid_list));

                 }
```

```
                if (id){

                        //altere os valores de d1 e d2 de diferentes maneiras como
exemplificado abaixo

                        d1 = d1 + i + 1;

                        d2 = d2 + d1 * 3;


                        // mostre na tela da console, a cada passagem, os
seguintes valores:

                        // PID do processo corrente, "i", "d1", "d2", "m" e
informe estar no ramo “then” do “if”

                        printf("PID do processo corrente = %i    |    d1 = %i    |
d2 = %i    |   m = %i  Ramo if\n\n", getpid(), d1, d2, m, i,j);


                        //Contabilizando o número de filhos

                        nFilhos++;


                        //Armazenando os processos filhos criados

                        sprintf(pid, "%i ", id);

                        strcat(pid_list, pid);



                        /*


================================================================================
==================================

                        Responda: Quais processos executam este trecho do código?

                        Todos os processos que possuem algum subprocesso associado
a ele, ou seja, todos os processos pai.

                        ------------------------------------------------------------
-----------------------------------------------------

                        */

                }else{

                        //altere os valores de d1 e d2 de diferentes maneiras e
```

```
também diferente do usado no trecho “then”

                    d1 = d1 - i;

                    d2 = (d2 - d1) * 3;


                    //execute o comando de atualização de “j” abaixo

                    j = i + 1;


                    //mostre na tela da console, a cada passagem, os seguintes
valores: PID do processo corrente; “i”, “d1”, “d2”, “m” e informe
                estar no ramo “else” do “if”

                    printf("PID do processo corrente = %i    |    d1 = %i    |
d2 = %i    |   m = %i  Ramo else\n\n", getpid(), d1, d2, m, i,j);


                    /*


==========================================================================
==================================

                    Responda: Quais processos executam este trecho do código?

                    Este trecho de código é executado por todos os processos
que possuem um pai, ou seja, por todos os processos exceto

                    o processo pai original.

                    ----------------------------------------------------------
----------------------------------------------------

                    */

                }


            }

            /*


==========================================================================
==================================

            Responda: Quais processos executam este trecho do código?

            Todos os processos executam este trecho.

            ----------------------------------------------------------------
--------------------------------------------
```

```
            */



            if (id != 0) {

               //mostre na console o PID do processo corrente e verifique
quais processos executam este trecho do código

               printf("PID do processo corrente = %i\n", getpid());



               for (i = j; i <= m; i++){

                    /*


==========================================================================
==================================
```

Responda: Explique o papel da variável “j”

A variável "j" contabiliza a altura da árvore genealógica a partir dos processos que são filho e pai ao mesmo tempo.

Desta forma, o processo pai original e os procesos folhas (processos filhos que não possuem filhos) não são considerados.

Responda: Verifique se o comando “for” está correto de forma que cada processo pai aguarde pelo término de todos seus

processos filhos

O comando for(i = j; i == m; i++) está incorreto porque ele se refere somente ao último pai.

O correto seria for(i = j; i<=m; i++), garantindo que cada processo pai tenha conhecimento de todos os seus filhos e aguarde

pelo término deles.

```
                    ------------------------------------------------------
-------------------------------------------------------
                    */



                    //mostre na console o PID do processo corrente e o número
de filhos que ele aguardou ou está aguardando

                    printf("PID do processo corrente = %i e número de filhos =
%i", getpid(), nFilhos);
```

```
                    //Mostra os processos que estão sendo esperados no momento

                    printf("\nO processo de PID = %i está esperando os
seguintes filhos: %s\n\n", getpid(), pid_list);



                    wait(&status);


                    if (status == 0){


                         /*


==========================================================================
==================================

                              Responda: o que ocorre quando este trecho é
executado?

                              Os processos filhos referentes ao processo corrente
terminaram a sua execução com sucesso.

                              ----------------------------------------------------
------------------------------------------------------------

                              */

                    }else{

                         /*


==========================================================================
==================================

                              Responda: o que ocorre quando este trecho é
executado?

                              Os processos filhos referentes ao processo corrente
ainda não terminaram sua execução ou terminaram com falhas.

                              ----------------------------------------------------
------------------------------------------------------------

                              */



                    }
```

```
                }
            }


            exit(0);
}
```

Método de execução

Os programas citados no estudo devem ser compilados e executados em um ambiente UNIX compatível através dos seguintes comandos:

1) Prog 1

gcc -o Prog1 Prog1.c **-lrt**

./Prog1

2) Prog 2

./Prog2.jar