Comparação de Desempenho Computacional com Vários Algoritmos Implementados Paralelamente em C

Tiago M. Rohde, Edilaine R. Ferrari, Luciano A. Destefani, Leonardo B. Motyczka, Rogério Martins

Departamento de Ciências Exatas e Engenharias - Universidade Regional do Noroeste do Estado do Rio Grande do Sul (UNIJUI)
Rua do Comércio, 3000, Bairro Universitário CEP 98700-000 - Ijuí – RS - Brasil

{tiago.mri, edinharevers, lucianodestefani, leomoty}@hotmail.com, {rogerio}@unijui.edu.br

Resumo: *Visando obter o maior aproveitamento possível do processador este artigo tem por objetivo comparar o desempenho computacional da execução de diferentes blocos de códigos em C, baseado no sistema operacional Linux. Com o intuído de não deixar nenhuma parte do processador ociosa, os algoritmos foram desenvolvidos utilizando a tecnologia de threads.*

Abstract: *In order to obtain the best possible use of the processor this article have the objective to compare the computational performance of the execution of different blocks of codes in C based on the operational system Linux. With intuited to leave no part of the processor idle, the algorithms were developed using the threads tecnology.*

Introdução

Um algoritmo nada mais é do que uma sequência de instruções, ou seja, comandos de repetições e decisão, ou até mesmo uma chamada de funções dentro ou fora do escopo de código principal, com o uso ou não dos threads, sendo que até 1980 não havia um conceito concreto sobre a utilização de threads, sendo que os primeiros a utilizar, de forma clara, foram os alunos da universidade de Carnegie Mellon (Machado; Maia, pg. 83).

O número de núcleos em um processador vem crescendo a cada dia, possibilitando o multiprocessamento, sendo assim o mesmo pode processar cada vez mais rápido as informações, com isso podemos encontrar processadores com dois, quatro e até mesmo de oito núcleos ou mais, possibilitando assim, a criação de threads, ganhando poder de processamento, conforme afirmam Machado; Maia:

> "A partir do conceito de múltiplos threads (multithread) é possível projetar e implementar aplicações concorrentes de forma eficiente, pois um processo pode ter partes diferentes do seu código sendo executados em paralelo, com um menor overhead do que utilizando múltiplos processos" (pg. 83).

Porém deve-se tomar cuidado com a criação de threads, pois além do gerenciamento da troca de informações também se tem o limite de memória e a capacidade máxima de gerenciamento do processador. Quando é criado um número muito grande de threads acaba-se tendo um "estouro" de pilha, parando bruscamente a execução. Outro problema também é a criação de threads para

executar tarefas muito pequenas, perde-se muito tempo na criação das mesmas (Maia, 1998).

1. PORQUE USAR THREADS

Uma das soluções encontradas para aumentar a velocidade de resposta para o usuário é a execução de várias funções ao mesmo tempo, tendo como maior beneficio a utilização de threads, se temos apenas um processo com um único thread sendo executado, perdemos o poder de processamento deixando os outros núcleos ociosos. Porém, um dos problemas é o gerenciamento dos threads, pois quando se cria variáveis globais e elas são acessadas dentro de algum bloco de código crítico e é feita uma tentativa de modificação ao mesmo tempo por duas partes concorrentes não podemos ter um resultado confiável, por isso utilizou-se a variável tipo *pthread_mutex_t* que cria uma espécie de cadeado que enquanto um thread utiliza os outros esperam a liberação da mesma.

Os algoritmos desenvolvidos para a execução em apenas um núcleo do processador, não aproveitam o poder de processamento disponível, devido a este fato utilizou-se a técnica de divisão das tarefas para serem executadas de forma concorrente, sendo assim, parte do código é executado em um núcleo do processador enquanto as outras partes são executadas nos demais núcleos, otimizando assim o algoritmo (Machado;Maia).

2. SPEEDUP - CÁLCULO PARA AUMENTO DE DESEMPENHO

A formula Speedup é dada por T1/Tp, onde o Speedup é o ganho de poder de processamento que temos, o T1 é o tempo gasto para executá-lo de forma monothread e o Tp é o tempo gasto para executá-lo com "n" threads, conforme afirma Blume: "o tempo dispensado para executar um algoritmo em um único processador (T1) e o tempo gasto para executá-lo em 'p' processadores (Tp). Speedup = T1/Tp**". (BLUME 2002, p. 60).**

3. FIBONACCI

Os algoritmos a seguir foram desenvolvidos para calcular a sequência de Fibonacci, esse cálculo é dado da seguinte maneira: inicia-se com os elementos 0 e 1 e a partir do terceiro é o valor da soma dos dois anteriores. Ele foi utilizado pelo fato de podermos abri-lo em forma de árvore possibilitando a divisão do processo em duas partes concorrentes, observe a notação do calculo na figura01 e na figura02 a demonstração do processo que "abre" em forma de árvore.

$$\mathbf{F(n)} = \begin{cases} \textbf{se } \mathbf{n<=1, n} \\ \textbf{senão } \mathbf{F(n-1)+ f(n-2)} \end{cases}$$

Figura01

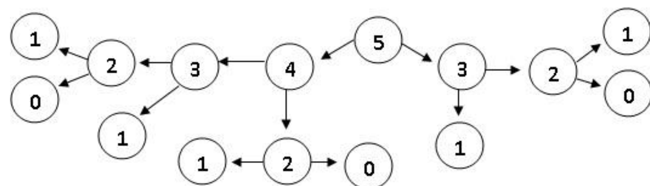


Figura02

4. DEFINIÇÕES

Em algumas estruturas do código foi necessário a utilização da variável *param* que é do tipo *struct*. Com essa variável podemos passar por parâmetro mais de uma variável ao

mesmo tempo, pois ela “(...) é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, colocadas sob um único nome.” (MIZRAHI, pg. 65). No algoritmo abaixo pode-se observar a implementação da variável *param* do tipo *struct*:

```
typedef struct {
        unsigned int n;  // É o número que passa.
        unsigned int r;  // É o número de retorno.
        unsigned int d;  // É o controlador da quantidade de threads na função do Fibonacci com
um número determinado de threads.
} param;
```

Algoritmo 02

A parte inicial e final de algumas implementações são iguais, mudando apenas o seu interior, nos Algoritmo 3 e 4 são demonstrados tais blocos de códigos.

```
void *fib(void *arg)
 {
        if((*(param*)arg).n > 1)
        {
```

Algoritmo 03

```
        }        else        {
        (*(param*)arg).r = (*(param*)arg).n;
        }        return (void*)0;
}
```

Algoritmo 04

5. IMPLEMENTAÇÃO

Para podermos analisar o ganho de tempo na execução deste cálculo utilizou-se algumas implementações que realizam a mesma tarefa, porém de modo diferente, com implementações distintas, mudando assim, o seu desempenho, sendo que os algoritmos foram divididos em recursivo, paralelizado a cada nodo da árvore, paralelo com número de threads pré-definido e paralelizado com número de threads dinâmicos.

5.1.RECURSIVO

De acordo com MIZRAHI, “uma função é dita recursiva se é definida em termos de si mesma. Isto é, uma função é recursiva quando dentro dela está presente uma chamada a ela própria” (MIZRAHI, pg.126).

No algoritmo recursivo testa-se o número que vem por parâmetro para a função se é menor que dois, é retornado o próprio valor, caso contrário a função chama a si mesma, obtendo o seguinte escopo de código:

```
unsigned int fib(unsigned int n)  {return n < 2 ? n : fib(n-1)+fib(n-2); }
```

Algoritmo 01

Para a passagem do resultado utilizou-se o comando *return* o qual “(...) fará com que a execução do programa volte para o código de chamada assim que o computador encontrar este comando (...)” (MIZRAHI, pg. 112).

5.2.PARALELIZADO EM CADA NODO DA ÁRVORE

Nesta implementação se obteve um estouro de pilha do sistema que ocorreu devido ao fato de não se ter espaço de armazenamento e condições de gerenciar um número alto de threads, “(...) um fator importante em aplicações multithread é o numero total de threads e a forma como são criados e eliminados. Se uma aplicação cria um número

excessivo de threads, poderá ocorrer um overhead no sistema (...)” (MACHADO;MAIA, pg. 94).

Porém, utilizando cálculos pequenos, como o número 10, temos o seu processo concluído, mesmo não obtendo um desempenho satisfatório, que ocorre pelo fato de não compensar o tempo de criação de um thread para executar pouca demanda. Na execução do Algoritmo 05 é criado dois threads, um para n-1 e outro para n-2:

```
Idem Algoritmo 03
        param p1 = {(*(param*)arg).n-1, 0}, p2 = {(*(param*)arg).n-2, 0};
        pthread_t t1, t2;
        pthread_create(&t1, NULL, fib, (void*)&p1);
        pthread_create(&t2, NULL, fib, (void*)&p2);
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
Idem Algoritmo 04
```

Algoritmo 05

5.3.PARALELIZADO COM NÚMERO DE THREADS DINÂMICOS UTILIZANDO A FUNÇÃO RECURSIVA

Tem-se de analisar vários fatores antes de invocar um novo thread, um deles é o tempo de criação e também a possibilidade do overhead, sendo que nesse algoritmo será criado CPU threads para serem executadas paralelamente, onde CPU é o número de núcleos que o processador possui, o qual foi obtido através da função *getCPUCount().*

Para ter controle sobre a criação foi utilizado a variável *d*, na *struct param*, sendo assim, quando um thread termina seu trabalho, outro é invocado fazendo com que haja uma perda de tempo nas várias criações. Em algumas execuções não compensa criar um thread que fara o calculo de pouca demanda computacional.

Para esse algoritmo será desprezado os Algoritmos 03 e 04, onde temos os seguintes procedimentos:

```
unsigned int mfib(unsigned int n)
{
        if(number_threads >= cpu_count)
        {       return n < 2 ? n : mfib(n-1)+mfib(n-2);
        }       else    {
                return scheduler(n);
        }
}
```

Algoritmo 06

No Algoritmo 06, faz-se um comparativo, verificando se o número de threads existentes é maior ou igual ao número de núcleos do processador, caso seja, a função recursiva é executada, caso contrário executa-se o Algoritmo 07, o qual analisa se o número que retorna por parâmetro deve ser calculado, se necessário é executada a função pthread_mutex_lock(&mutex).

Após a análise do algoritmo 07, é criado dois threads executando o Algoritmo 08, este por sua vez, encerra o ciclo executando o Algoritmo 06. Sendo assim, cada vez que um procedimento é encerrado, o resultado calculado é passado para a função através do comando *return*.

```
unsigned int scheduler(unsigned int n)  { if(n >= 2) { control thc = {PTHREAD_MUTEX_INITIALIZER, false};
                    param p1 = {n-1, 0, &thc}, p2 = {n-2, 0, &thc};
                    pthread_mutex_lock(&mutex);
                    if(number_threads < cpu_count) {number_threads++;
                              pthread_mutex_unlock(&mutex);
                              pthread_t t1, t2;
                              pthread_create(&t1, NULL, thread, (void*)&p1);
                              pthread_create(&t2, NULL, thread, (void*)&p2);
                              pthread_join(t1, NULL);
                              pthread_join(t2, NULL);
                    } else {pthread_mutex_unlock(&mutex);
                              p1.res = mfib(p1.n);
                              p2.res = mfib(p2.n);
                    } return p1.res+p2.res;
          }         else {
                    return n;           }
}
```

Algoritmo 07

```
void *thread(void *arg)
{
          ((ptr_param)arg)->res = mfib(((ptr_param)arg)->n);
          pthread_mutex_lock(&(((ptr_param)arg)->thc->mutex));
          if(!((ptr_param)arg)->thc->decremented)     {
                    ((ptr_param)arg)->thc->decremented = true;
                    pthread_mutex_lock(&mutex);
                    number_threads--;
                    pthread_mutex_unlock(&mutex);          }
          pthread_mutex_unlock(&((ptr_param)arg)->thc->mutex);
          return (void*)0; }
```

Algoritmo 08

5.4.PARALELIZADO COM UM NÚMERO PRÉ-DETERMINADO DE THREADS UTILIZANDO A FUNÇÃO RECURSIVA

O algoritmo a seguir tem o objetivo de sanar problemas como a criação de várias partes concorrentes que executam pequenos cálculos. Neste algoritmo pode-se observar que os threads são criados de acordo com o número de núcleos do processador.

```
Idem Algoritmo 03
param p1 = {(*(param*)arg).n-1, 0, (*(param*)arg).d/2}, p2 = {(*(param*)arg).n-2, 0, (param*)arg).d/2};
                    if((*(param*)arg).d > 1) {
                              pthread_t t1, t2;
                              pthread_create(&t1, NULL, fib, (void*)&p1);
                              pthread_create(&t2, NULL, fib, (void*)&p2);
                              pthread_join(t1, NULL);
                              pthread_join(t2, NULL);
                    }                   else  {
                              p1.r = fib(p1.n);
                              p2.r = fib(p2.n);}
Idem Algoritmo 04
```

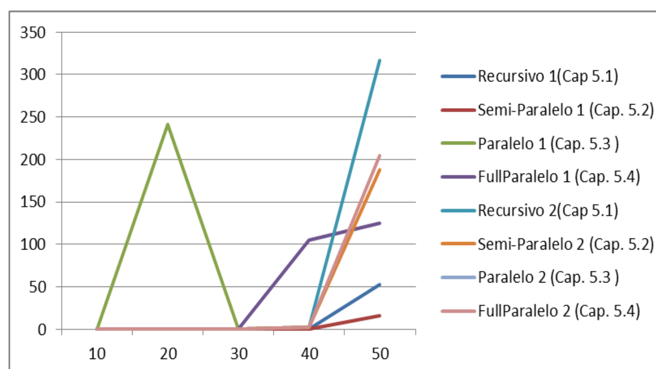
Algoritmo 09

Com isso temos um algoritmo com maior desempenho, dentre os outros, porém ainda temos o problema da criação de threads com tarefas desiguais, onde uns threads terão cálculos maiores do que outros. Seriar possível resolver esse problema com a implementação de algoritmo que separa as tarefas de forma homogênea, sendo que este algoritmo deveria pré-agendar a criação de threads.

6. ANÁLISE DE RESULTADOS

Os algoritmos foram executados em dois computadores sendo que os resultados foram dados em segundos, então para o 1 temos um Core i7 2630QM sendo 4 núcleos com 8 Threads e para o 2 temos um Intel Core 2 duo t5870 2.0 GHZ sendo dois núcleos com 2 threads, veja a tabela de tempo e o gráfico abaixo.

	Número calculado				
Alg./proc.	10	20	30	40	50
Sequencial 1	0,001	0,003	0,009	0,42	52,2
Semi-Paralelo 1	0,002	0,003	0,009	0,145	16,25
Paralelo 1	0,02	241	Falha de segmentação		
FullParalelo 1	0,01	0,014	0,104	105	125
Sequencial 2	0,005	0,046	0,03	2,58	316,6
Semi-Paralelo 2	0,007	0,106	0,427	1,79	187,83
Paralelo 2	0,03	Falha de segmentação			
FullParalelo 2	0,005	0,105	0,448	1,78	204,2



7. CONCLUSÃO

Nos dias de hoje, a utilização do método sequencial se tornou obsoleto, em certos casos, devido ao fato de se ter processadores com vários núcleos, abrindo assim espaço para a execução de sistemas que operam em paralelo, porém a utilização dessas rotinas ainda é importante para executar tarefas de pequeno porte. Também se deve ter cuidado na manipulação de threads por causa do "estouro" de pilha ou até mesmo pela perda de poder computacional causado pelo tempo de criação versus tempo de execução.

8. REFERÊNCIAS BIBLIOGRÁFICAS

BLUME, Evandro. "Algoritmo de Organização Paralelo: Um modelo proposto e implementado". Dissertação (Mestrado em Ciência da Computação) Universidade Federal de Santa Catarina, Florianópolis/SC, 2007.

MACHADO, Francis Berger; MAIA, Luis Paulo. "Arquitetura de Sistemas Operacionais". 3ª Edição, LCT, 2002.

MAIA, Luis Paulo. Monografia: "Multithread" - 18/05/1998. Instituto de Matemática da UFRJ/NCE.

SILBERSCHATZ, Abrahan; PETER, Galvin; GREG, Gagne. "Sistemas Operacionais: Conceitos e Aplicações". Elsevier Editora Ltda, 2001.

MIZRAHI, Victorine Viviane. "Treinamento em Linguagem C". Módulo 1. Editora: Pearson Makron Books, 1990.

MIZRAHI, Victorine Viviane. "Treinamento em Linguagem C". Módulo 2. Editora: Pearson Makron Books, 2001.