Paralelização do sistema de filtragem de identificadores inválidos.

Aluno: Lucas Starling de Paula Salles

Matricula: 2016006697

Introdução

O trabalho de paralelização de um sistema de filtragem de identificadores inválidos foi proposto como o primeiro exercício de programação da matéria de Fundamentos de Sistemas Paralelos e Distribuídos. Consistindo da adaptação de um código sequencial disponibilizado, que resolve o problema da filtragem de identificadores inválidos para duas versões distribuídas, uma usando paralelismo de dados e outra usando paralelismo funcional.

Proposta

Uma organização decidiu usar identificadores numéricos para seus membros, mas alguém (um chefe, com certeza) decidiu que certos números não poderiam ser usados. Ele criou algumas regras que determinariam se um número pode ser usado ou não:

- não podem ser palíndromos,
- não podem ser uma sequência menor dobrada (p.ex., 123123),
- não podem ter a soma dos algarismos igual à soma dos algarismos em ordem (p.ex., 012345),
- não podem ter um mesmo algarismo repetido três vezes em sequência,
- nem pode ter um algarismo repetido quatro vezes em qualquer posição do identificador.

Uma versão sequencial do programa já foi desenvolvida e o código está disponível no moodle. Nas máquinas do laboratório de graduação, essa versão leva por volta de 90 segundos para testar todos os números de 9 dígitos (até 999999999).

Objetivo

Você deve alterar o arquivo principal da versão sequencial (numcheckseq.c) para produzir duas versões paralelas, uma usando paralelismo de dados, outra usando paralelismo de funções. Os arquivos esperados, numcheckdatapar.c e numcheckfuncpar.c devem ser desenvolvidos a partir do código fornecido.

Metodologia

Para o melhor desenvolvimento das versões paralelizadas o primeiro passo foi executar diversas vezes o código original, juntamente com a análise do mesmo para averiguar suas funcionalidades e identificar melhores opções para realizar a paralelização. Tendo sido feito isso foi decidido que a melhor estratégia seria manter as versões paralelas o mais similares com a sequência o possível.

A estratégia de paralelização para dados foi bastante simples de identificar: dividir os números que precisavam ser avaliados pelo número de threads disponíveis para o programa. O paralelismo funcional apresentava um problema maior, como computar os diversos resultados necessários para cada número sendo que esse precisavam ser combinados em seguida para

coletar mais informações para o usuário, contudo fazer a separação por funções de teste individualmente se mostrou possível com um pouco de modelagem.

Paralelismo de Domínio

Lançar threads para uma porção dos dados é razoavelmente simples, gerando apenas uma seção crítica: combinação dos resultados para atualização das informações globais.

O primeiro passo necessário para realizar essa distribuição foi dividir os números em partes iguais, uma para cada thread. Para isso o input do usuário foi dividido pelo número de threads, estabelecido como sendo igual a oito pela descrição fornecida no material do trabalho. Uma vez divididos em cotas, os números foram passados para uma thread executando uma método intermediario para a função `check_number`, `check_range` desenvolvido para receber um apontador para endereço de memória do tipo void, com o qual recupera uma struct contendo o range de números para a avaliação assim como o número de dígitos máximos do problema e chama `check_number` para todos os numeros daquele range .

```
 long quota = maxnum/thread_count;
    for(thread = 0; thread < thread_count-1; ++thread){
        struct range *param;
        param = malloc(sizeof(param));
        param->from = thread*quota;
        param->to = thread*quota + quota-1;
        param->ndigits = ndigits;
        pthread_create(&thread_handles[thread], NULL, check_range, (void*)
param);
    }
    struct range *param;
    param = malloc(sizeof(param));
    param->from = thread*quota;
    param->to = maxnum;
    param->ndigits = ndigits;
    pthread_create(&thread_handles[thread], NULL, check_range, (void*) param);
    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);
```

```
void *check_range(void *data){
    long i;
    struct range *param = (struct range *)data;
    for (i=param->from;i<=param->to;++i) {
        check_num(i,param->ndigits);
    }
    return NULL;
}
```

Seção crítica

Nessa opção de paralelismo o problema era atualizar os valores globais, visto que todas as threads poderiam tentar fazê-lo ao mesmo momento o que corrompe os dados. Para evitar isso foi usado um mutex, que somente permite que uma thread por vez manipule as variáveis globais ou execute a função update_max.

// Para processar número de condições satisfeitas

```
all = pal + rep + sum + dou + fou;
 pthread_mutex_lock(&mutex);
 if (all>0) {
   match_some_test += 1;
}
 update_max( orign, all );

 // Atualiza os contadores por condição
 palindromes += pal;
 repeated_seqs += rep;
 sums_are_ap += sum;
 have_tripled_digits += dou;
 have_four_repetitions += fou;
 pthread_mutex_unlock(&mutex);
```

Paralelismo Funcional

Implementar o programa com paralelismo funcional demandou mais tempo e mais testes mal sucedidos. Era claro que o melhor caminho seria lançar threads para cada uma das funções que avaliam uma das possíveis condições de um numero: `is_palindrome`, `has_repeated_seq`, `sum_is_ap`, `has_tripled_digits`, `has_four_repetitions`. Contudo o desafio foi combinar os resultados individuais dessas threads de forma a não estender a porção sequencial do código. Para isso foi preciso criar uma estrutura capaz de armazenar esses resultados intermediários e, simultaneamente, indicar quando todos os testes já haviam sido executados para cada número individualmente. Isso porque não basta ter uma thread combinando os resultados parciais das avaliações individuais para os números se aquele número ainda não foi avaliado por todas as demais threads.

Para isso foi usado um array de bytes, e um conjunto de operações sobre eles que permitiram, em unico byte, armazenar tanto a contagem de avaliações que haviam sido positivas, que é a informação que de fato importa para o programa, quanto a quantidade de testes que já haviam sido concluídos para cada número. Além disso foi armazenda o maior número que já passou por todas as avaliações parciais.

Modelagem

Dados armazenados:

```
int current = -1;
...
unsigned char *bytes;
```

Um filtro foi implementado para indicar quando cada numero ja havia sido avaliado por todas as 5 threads.

```
unsigned char filter = 5 << 5; //mascara para validar que o numero passou pelos 5 testes.
```

Controladores:

```
const int mutex_count = 10;
pthread_mutex_t mutex_check[10];
pthread_mutex_t mutex_join;
pthread_cond_t calc_max;
```

Operações:

- `count_up_checks`: responsável por aumentar a contagem de avaliações realizadas para um número específico, também usada para somar o resultado obtido pela avaliação. A variável `checked` pode assumir os valores 0 e 1, visto que ela é o resultado das funções de avaliação, e essas retornam um inteiro igual a 0 ou 1. Ao final de sua execução esse método invoca `all_checks_done`.
- `all_checks_done`: avalia para um numero específico, se todos os checks necessários já foram concluídos, caso sim atualiza o contador que indica qual o maior numero ja avaliado totalmente e sinaliza/acorda a thread que combina os resultados parciais.
- `get_checked_amount`: informa quantas avaliações foram afirmativas para um numero específico.

```
void all_checks_done(long n)
{
    if(bytes[n] >= filter){
        pthread_mutex_lock(&mutex_join);
        current = n;
        pthread_mutex_unlock(&mutex_join);
        pthread_cond_signal(&calc_max);
    }
}

void count_up_checks(long n, int checked)
{
    pthread_mutex_lock(&mutex_check[n % mutex_count]);
    bytes[n] += 1 << 5;
    bytes[n] += checked;
    pthread_mutex_unlock(&mutex_check[n % mutex_count]);
    all_checks_done(n);
}

int get_checked_amount(long n)
{
    return bytes[n] - filter;
}
```

Seção crítica

Nas funções acima pode se observar o uso de dois tipos de mutex, um para cada seção crítica da versão funcional do programa.

As threads que executam testes individualmente para todos os números, precisam atualizar uma estrutura de dados comum, o array `bytes`. Contúdo um arranjo não precisa ser todo bloqueado para ser feita atualização apenas de um de seus registros, bastaria bloquear a atualização daquele registro. Como não é viavel declarar um mutex para cada byte do arranjo a escolha foi por criar um conjunto de tamanho fixo de mutex's e usar uma operação de modulo para determinar qual numero avaliado se refere a qual mutex. Com o uso de 10 mutex para controlar o acesso ao arranjo de bytes o codigo teve uma melhoria de desenpenho de cerca de 25%! Demais aumentos nessa quantidade não escalaram em performance.

A thread que combina os resultados individuais das demais tem uma seção crítica relacionada ao acesso à variavel global `current` , alterada pelas demais threads na chamada à `all_checks_done`. Apesar dessa thread não atualizar a variavél em questão é vantajoso que esse fluxo seja dependente de um mutex para que ela fique bloqueada enquanto ela não pode executar, liberando um nucleo para os demais processamentos.

Para iniciar essas threads foi preciso separar o método `check_number` em seis métodos diferentes, cinco deles responsáveis por realizar as avaliações individualmente para todos os números e um para combinar os resultados.

```
void *check_palindrome(void *data)
{
    struct problem param = *(struct problem *)data;
    int pal,i;
    digit_t num;
    for (i=0;i<=param.n;++i) {
        // Transforma número (n) em vetor de dígitos (num)
        break_into_digits(i, num, param.ndigits);

        // Aplica os diversos testes a um dado número
        pal = is_palindrome( num, param.ndigits );

        // Atualiza os contador de palindromos
        palindromes += pal; //nao preciso de mutex aqui pois so essa thread mexe
nessa global
        count_up_checks(i, pal);
    }
    return NULL;
}

void *check_repeated_seq(void *data)
{
    struct problem param = *(struct problem *)data; //nao preciso de mutex aqui
pois threads so fazem leitura
    int rep,i;
    digit_t num;
    for (i=0;i<=param.n;++i) {
        // Transforma número (n) em vetor de dígitos (num)
        break_into_digits(i, num, param.ndigits);

        // Aplica os diversos testes a um dado número
        rep = has_repeated_seq( num, param.ndigits);
```

```
        // Atualiza os contador de palindromos
        repeated_seqs += rep; //nao preciso de mutex aqui pois so essa thread
mexe nessa global
        count_up_checks(i, rep);
    }
    return NULL;
}

void *check_sum_is_pa(void *data)
{
    struct problem param = *(struct problem *)data; //nao preciso de mutex aqui
pois threads so fazem leitura
    int sum,i;
    digit_t num;
    for (i=0;i<=param.n;++i) {
        // Transforma número (n) em vetor de dígitos (num)
        break_into_digits(i, num, param.ndigits);

        // Aplica os diversos testes a um dado número
        sum = sum_is_ap( num, param.ndigits );

        // Atualiza os contador de palindromos
        sums_are_ap += sum; //nao preciso de mutex aqui pois so essa thread mexe
nessa global
        count_up_checks(i, sum);
    }
    return NULL;
}

void *check_has_triple_digits(void *data)
{
    struct problem param = *(struct problem *)data; //nao preciso de mutex aqui
pois threads so fazem leitura
    int dou,i;
    digit_t num;
    for (i=0;i<=param.n;++i) {
        // Transforma número (n) em vetor de dígitos (num)
        break_into_digits(i, num, param.ndigits);

        // Aplica os diversos testes a um dado número
        dou = has_tripled_digits( num, param.ndigits );

        // Atualiza os contador de palindromos
        have_tripled_digits += dou; //nao preciso de mutex aqui pois so essa
thread mexe nessa global
        count_up_checks(i, dou);
    }
    return NULL;
}

void *check_has_four_reps(void *data)
{
    struct problem param = *(struct problem *)data; //nao preciso de mutex aqui
pois threads so fazem leitura
    int fou,i;
    digit_t num;
    for (i=0;i<=param.n;++i) {
```

```
        // Transforma número (n) em vetor de dígitos (num)
        break_into_digits(i, num, param.ndigits);

        // Aplica os diversos testes a um dado número
        fou = has_four_repetitions( num, param.ndigits );

        // Atualiza os contador de palindromos
        have_four_repetitions += fou; //nao preciso de mutex aqui pois so essa
thread mexe nessa global
        count_up_checks(i, fou);
    }
    return NULL;
}

void *update_maxes(void *data)
{
    struct problem param = *(struct problem *)data; //nao preciso de mutex aqui
pois threads so fazem leitura
    int i,all;

    for (i=0;i<=param.n;++i) {

        pthread_mutex_lock(&mutex_join);
        while ( i > current){
             pthread_cond_wait(&calc_max, &mutex_join);
        }
        pthread_mutex_unlock(&mutex_join);
        all = get_checked_amount(i);
        if (all > 0) {
            match_some_test += 1; // nao preciso de mutex aqui pois so essa
thread mexe nessa global
        }
        update_max( i, all ); // nao preciso de mutex aqui pois so essa thread
mexe nessa global
    }
    return NULL;
}
```

O programa é iniciado lançando essas funções em paralelo:

```
    pthread_create(&thread_handles[0], NULL, check_palindrome, (void*) param);
    pthread_create(&thread_handles[1], NULL, check_repeated_seq, (void*) param);
    pthread_create(&thread_handles[2], NULL, check_sum_is_pa, (void*) param);
    pthread_create(&thread_handles[3], NULL, check_has_triple_digits, (void*)
param);
    pthread_create(&thread_handles[4], NULL, check_has_four_reps, (void*) param);
    pthread_create(&thread_handles[5], NULL, update_maxes, (void*) param);
    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);
```

Execução

Para compilar

Do diretório raiz do projeto:

- Versão sequencial: `make` . Será gerado um executável chamado *seq* na pasta *bin* do projeto.
- Versão com paralelismo de domínio: `make data` . Será gerado um executável chamado *data* na pasta *bin* do projeto.
- Versão com paralelismo funcional: `make func` . Será gerado um executável chamado *func* na pasta *bin* do projeto.

O nome do executável original foi alterado para facilitar o uso e manter padronizado com as demais versões.

Para executar

Do diretório raiz do projeto:

- `./bin/seq <numero>`
- `./bin/data <numero>`
- `./bin/func <numero>`

Avaliação

Avaliação executada em uma maquina com intel *i5*, quad core, de terceira geração, sem hyperthreading.

Sequencial

Avaliação por media de 100 execuções:

> **0.9756419999999997 segundos para caso médio de 9999999**

```
for i in $(seq 100); do

./seq 9999999 | grep "tempo" | awk '{ print $2 }' >> tests/seq/aval.txt
done
```

```
cat tests/seq/aval.txt  | python3 -c "import sys; print(sum(float(l) for l in
sys.stdin)/100)"
    0.9756419999999997
```

Dados

Avaliação por média de 100 execuções:

> **1.4968351900000008 segundos para caso médio de 9999999**

```
for i in $(seq 100); do
./bin/data 9999999 | grep "tempo" | awk '{ print $2 }' >> tests/data/aval.txt
done
```

```
cat tests/data/aval.txt  | python3 -c "import sys; print(sum(float(l) for l in
sys.stdin)/100)"
    1.4968351900000008
```

Funcional

Avaliação por média de 100 execuções:

> **4.518436680000001 segundos para caso médio de 9999999**

```
for i in $(seq 100); do

./bin/func 9999999 | grep "tempo" | awk '{ print $2 }' >> tests/func/aval.txt
done
```

```
cat tests/func/aval.txt  | python3 -c "import sys; print(sum(float(l) for l in
sys.stdin)/100)"
    4.518436680000001
```

Resultados

Como havia sido avisado pelo professor as implementações paralelas acabaram não gerando melhoria no processamento do programa, pelo contrário. A versão com paralelismo de dados se mostrou em média 50% mais lenta que a sequencial, o paralelismo funcional desempenhou pior ainda, levando em média 450% do tempo da sequencial.