Faculdade de Ciências da Universidade do Porto

Servidor para Cálculo de Polinómios

Programação Concorrente (CC3040)- 2021/2022

Mateus Almeida 201805265

Introdução

Originalmente uma linguagem proprietária da Ericsson, Erlang é uma linguagem de programação de alvo o suporte de aplicações distribuídas e a tolerância a falhas a ser executadas num ambiente de tempo real e ininterrupto. Foi popularizada pelo seu uso no desenvolvimento de aplicações como o *WhatsApp* e o *AdRoll*.

Este relatório foi realizado no âmbito do primeiro trabalho prático da unidade curricular de Programação Concorrente. O objetivo deste projeto foi criar um servidor que calcule operações sobre polinómios a pedido dos clientes.

Desenvolvimento

Módulos

Os módulos utilizados para a execução do programa são *polynomial.erl,* onde estão todas as funções utilizadas para o cálculo das operações; *poly_server.erl,* onde está implementado o modelo cliente-servidor e *print.erl*, onde estão implementadas algumas funções para imprimir os polinómios num formato standard. O ficheiro *polynomial_test.erl* contém casos de teste que podem ser corridos pela função *test/1*.

Para a execução distribuída com sockets, os módulos usados são da biblioteca lib_chan do livro *Programming Erlang: Software for a Concurrent World,* tendo-se implementado apenas o serviço requerido por esse Middle Man, *middle_server.erl*.

Server

O servidor é inicializado com a função start/0, que gera , com a função spawn/1, um novo processo que executa a função loop/0. Este fica registado no nome “polynomial”, com register/2.

A função loop/0 fica à espera de receber pedidos do cliente do formato {Op, Pid, Polynomial1, Polynomial2}, retornando num tuplo {ok, Result}. O resultado é calculado como a operação nos dois polinómios depois de estes serem ordenados com polynomial:sort/1.

Cliente

O cliente dispõe de três operações que pode realizar, sendo as funções nomeadas de acordo com o pedido (add/2, sub/2, mult/2). Ao receber o resultado do pedido, são imprimidos tanto o tuplo resultante como a sua versão simplificada à leitura, com o auxílio da função print:printPol/1. A função stop/0 termina o processo.

Funções auxiliares

Função sort/1

A função sort/1 implementa o algoritmo quicksort para ordenar os polinómios por ordem decrescente dos seus expoentes de modo a facilitar o cálculo das operações. Os polinómios com múltiplas variáveis são organizados de acordo com o seu maior expoente. Retorna uma nova lista ordenada e é sempre chamado antes do cálculo de qualquer das operações.

Função printPol/1

A função printPol/1 imprime no stdout a lista de tuplos que representa um polinómio no formato standard (coeficiente)(variável)^(expoente). Foi implementada de forma a facilitar a interpretação e compreensão dos resultados retornados pelas funções que calculam operações de polinómios.

Funções coef/1, exp/1, variable/1

Funções implementadas de forma a facilitar a legibilidade do código que são utilizadas apenas pontualmente, aquando não é utilizado *pattern matching*. Para um polinómio {variable, coef, exp}, retornam o valor do polinómio respetivo ao seu nome. Para um polinómio com múltiplas variáveis, exp/1 retorna o maior expoente, sendo esta função apenas utilizada no polynomial:sort/1.

Função exp/2

A função exp/2 atualiza o valor do expoente de uma variável existente num polinómio com várias variáveis. É utilizada no cálculo da multiplicação de polinómios.

Função canOperate/2

A função canOperate/2 determina se dois polinómios podem ser somados/subtraídos de forma a resultar num novo polinómio diferente, isto é, se os seus expoente e variáveis são iguais.

Função add_p/1, add_list/1, add_s/1

A função add_p/1 recebe de input a lista ordenada composta pelos dois polinómios concatenados e calcula a sua soma, iterando por cada elemento da lista e somando com o próximo elemento da lista que tiver o mesmo expoente e variável. Se o polinómio tiver mais que uma variável, é chamada a função auxiliar add_list/1 e se tiver apenas uma, add_s/1. A operação de soma é realizada no valor positivo da função canOperate/2. Retorna a lista de tuplos que representa o polinómio resultante da soma.

Função sub_p/1, sub_list/1, sub_s/1

Análogo à função anterior, a função sub_p/1 recebe de input a lista ordenada composta pelos dois polinómios concatenados e calcula a sua subtração. A operação de subtração é realizada caso os expoentes e variável dos elementos nos quais a operação se realiza forem iguais, tal é confirmado pela função canOperate/2. Retorna a lista de tuplos que representa o polinómio resultante da subtração.

Funções mult_p/2 e auxiliares

Estas funções foram implementadas com o objetivo de calcular o produto de dois polinómios dados como input. Como o produto de polinómios segue a propriedade distributiva da multiplicação, todas as operações são calculáveis sem a necessidade de uma condição como a de canOperate/1.

A função mult_r/2 itera recursivamente a primeira lista para realizar a distribuição dos polinómios pela segunda. Cada operação é avaliada por mult_aval/2, sendo as diferentes operações divididas em quatro condições: ambos os polinómios têm várias variáveis, logo, têm listas dentro dos tuplos - mult_var_list/2; apenas um dos polinómios tem várias variáveis - mult_var_check/2 e ambos têm uma única variável - mult_list/2. Nos dois últimos, caso um dos operandos seja uma constante, a função checkForConst/2 certifica-se que o identificador de constante, var(const), não é adicionado a nenhuma lista.

Testes

Para correr o programa num ambiente não distribuído, basta correr o ficheiro Makefile com o argumento server:

```
$ make server
```

Os pedidos são feitos pelo cliente, recorrendo ao módulo poly_server. Alguns exemplos e resultados retornados:

```
>poly_server:add([{[x,y],1,[2,2]},{x,2,4},{y,10,1},{const,-10,0}],[{[x,y],2,[2,2]},{[x,z],10,[3,2]},{y,-5,1},{const,10,0}]).
2x^4 + 10(x^3)(z^2) + 3(x^2)(y^2) + 5y^1 + 0

>poly_server:sub([{[x,y],1,[2,2]},{x,2,4},{y,10,1},{const,-10,0}],[{[x,y],2,[2,2]},{[x,z],10,[3,2]},{y,-5,1},{const,10,0}]).
2x^4 + 10(x^3)(z^2) + -1(x^2)(y^2) + 15y^1 + -20

>poly_server:mult([{[x,y],1,[2,2]},{x,2,4},{y,10,1},{const,-10,0}],[{[x,z],10,[3,2]},{y,-5,1},{const,10,0}]).
20(x^7)(z^2) + 10(y^2)(x^5)(z^2) + -10(x^4)(y^1) + 20x^4 + -5(x^2)(y^3) +
100(y^1)(x^3)(z^2) + -100(x^3)(z^2) + 10(x^2)(y^2) + -50y^2 + 150y^1 + -100
```

Para correr o programa em dois nós diferentes, recorre-se à flag *-sname* do erl. Na primeira janela:

```
$make node
```

, onde estará a correr o servidor.

Numa segunda janela o cliente faz os pedidos recorrendo ao módulo rpc. Um exemplo de chamada:

```
>rpc:call(pc2122@sionideapad, poly_server, mult, [[{x, -2, 1},{x, -1, 1},{x, 4, 5},{x, -3, 5}],[{x, 10, 3}, {x,2,4}]] ).
```

Para correr o programa num ambiente distríbuido, corre-se o Makefile com o argumento:

```
$ make distributed
```

, ficando o ficheiro .config definido para uso futuro, sendo também inicializado o servidor:

```
>poly_server:start().
```

Numa segunda janela do terminal, corre-se o cliente que utilizará o módulo *lib_chan* para servir de Middle Man:

```
$ make client
> {ok, Pid} = lib_chan:connect("localhost",1234,middleServer,"PC20212022","").
```

Os pedidos são feitos recorrendo à função lib_chan:rpc/2 com argumentos no formato {Pid, {fun, [Args]}. Um exemplo de chamada:

```
> lib_chan:rpc(Pid, {add, [{x, -2, 1},{x, -1, 1},{x, 4, 5}],[{x, -3, 5},{x, 10, 3}, {x,2,4}]}).
```

Os casos podem também ser corridos automaticamente com os teste pré-definidos no módulo polynomial_test.erl, com os argumentos:

```
$make test - para casos de teste simples
$make test_server - para inicializar o servidor lib_chan e o servidor para cálculos
$make test_cliente - que envia testes para o servidor
```

Conclusão

A linguagem Erlang demonstrou-se muito simples de entender e levou à maior compreensão da implementação de um servidor, para além de exemplificar a execução de processos distríbuidos.

Os problemas encontrados durante a elaboração do projeto foram a implementação das funções para o cálculo de multiplicação de polinómios, que se revelou confusa devido à estrutura por mim escolhida para representar os polinómios com múltiplas variáveis. Tivesse sido escolhida uma estrutura que mapeia a variável ao seu respetivo expoente, possivelmente tornar-se-ia mais simples.

A execução de casos de teste automáticos também revelam falhas devido ao tuplo retornado não ser disposto no terminal. Porém, apesar destas falhas, penso que os objetivos do trabalho foram atingidos.

Bibliografia

Armstrong, Joe. *Programming Erlang: Software for a Concurrent World*, 2nd ed., The Pragmatic Programmers, 2013, pp. 318-342