Projeto de Sistemas Operativos 2018-19
`CircuitRouter-ParSolver`
Enunciado do Exercício 2

LEIC-A / LEIC-T / LETI
IST

Antes de lerem este guia, os alunos devem ler primeiro o documento de visão geral do projeto, assim como os enunciados dos exercícios anteriores do projeto.

1 Introdução

Na primeira parte do projeto foi usada uma abordagem multi-processo para paralelizar a execução de instâncias diferentes do algoritmo de Lee. Isto é, calcular em paralelo o roteamento de circuitos *distintos*.

O objetivo do Exercício 2 é desenvolver uma versão paralela do algoritmo de Lee, chamada `CircuitRouter-ParSolver`, tendo como ponto de partida o `CircuitRouter-SeqSolver` desenvolvido anteriormente[1]. No `CircuitRouter-ParSolver`, múltiplas tarefas (*threads*) do mesmo processo coordenam-se e cooperam para calcular o roteamento das várias interconexões do *mesmo* circuito.

Enquanto que, no Exercício 1, os problemas resolvidos em paralelo são independentes, no Exercício 2 as tarefas do mesmo processo partilham o mesmo estado, i.e., a grelha do mesmo circuito, e competem entre si para atribuir posições da grelha a interconexões diferentes. Portanto, para garantir a correção da solução, torna-se incontornável o problema de regular a concorrência entre tarefas diferentes do mesmo processo, de forma a evitar que estas possam aceder de forma incompatível às mesmas posições da grelha.

O desafio principal desta parte do projeto consiste em desenhar (e implementar, claramente!) um mecanismo de sincronização entre tarefas que seja não só *correto*, mas também *eficiente*, ou seja, que maximize o paralelismo efetivamente atingível pelo programa.

Para concretizar este objetivo os alunos deverão estender a implementação sequencial do algoritmo de Lee, adicionando mecanismos de sincronização baseados nas técnicas lecionadas durante as aulas teóricas (p.e., *mutexes*, trincos de leitura-escrita, semáforos, etc).

2 Paralelização do algoritmo de Lee

A implementação sequencial do algoritmo de Lee que é implementada no `CircuitRouter-SeqSolver` presta-se de forma bastante simples a ser paralelizada usando múltiplas tarefas.

Para tal, no início do programa do `CircuitRouter-ParSolver` deverá ser criado um conjunto de nova tarefas *trabalhadoras*. O número de tarefas trabalhadoras é definido por um argumento adicional obrigatório que deve ser passado ao `CircuitRouter-ParSolver`, na forma `-t NUMTAREFAS`, onde `NUMTAREFAS` é um inteiro maior que zero.

[1] Para este exercício não será necessário usar a `CircuitRouter-Shell` desenvolvida no 1º exercício. Voltaremos a integrar a `CircuitRouter-Shell` com o `CircuitRouter-ParSolver` na 3ª parte do projeto.

A maneira mais natural de paralelizar a implementação consiste em permitir que as tarefas trabalhadoras processem as diferentes interconexões do mesmo circuito. O código sequencial fornecido aos alunos coloca as várias interligações a processar numa fila (ver `workQueuePtr` na função `router_solve` em *router.c*), antes de as processar sequencialmente. Cada vez que uma interligação é processada, o percurso resultante no circuito é guardado numa estrutura de dados chamada `myPathVector`, que é alocada no principio da função `router_solve`. No final, os percursos calculados e guardados na função `myPathVector` são copiados para uma estrutura de dados global, apontada pela variável `pathVectorListPtr`.

Mais detalhadamente, ao executar em paralelo as várias tarefas deverão seguir a seguinte estratégia:

1. "Competir" para extrair (i.e., *pop*) a próxima interconexão da fila de input;
2. Calcular a rota (obtida no passo 1) no circuito, usando as primitivas de sincronização necessárias para garantir a correção da solução;
3. Tal como na implementação fornecida, o resultado do processamento de cada interconexão será inicialmente guardado numa lista local a cada tarefa (i.e., `myPathVector`). Os percursos calculados por uma tarefa só deverão ser copiados para a lista global (i.e., `pathVectorListPtr`) quando a tarefa deteta que já não há mais interligações para processar.

Recomenda-se que os alunos sigam os seguintes passos graduais até completarem a solução:

1. Começar por criar as tarefas trabalhadoras.
2. Implementar a estratégia descrita acima usando um *mutex* global por cada estrutura concorrente acedida em cada um dos passos (1-3) da estratégia apresentada acima. Esta solução será, naturalmente, limitada por permitir pouco paralelismo (em particular, só pode estar uma tarefa de cada vez a calcular uma rota).
3. Desenhar e implementar uma solução que substitua o trinco global usado para o passo 2 (para sincronizar o acesso ao circuito) por sincronização de granularidade fina. Esta solução deverá, finalmente, permitir níveis de paralelismo elevados.

O último passo é, claramente, o mais desafiante deste exercício. Sobre este passo, damos as seguintes sugestões:

- Há diferentes soluções possíveis, e nenhuma é trivial. Por isso recomendamos fortemente que o grupo valide primeiro a solução com um docente antes de avançar para a implementação.
- Como discutido nas aulas teóricas, o uso de trincos finos levanta frequentemente problemas de interblocagem ou míngua. Os alunos devem estudar cuidadosamente se a sua solução sofre desses problemas e incorporar mecanismos para os prevenir.
- Como explicado no enunciado geral, o cálculo da rota é feito em 3 fases: leitura do estado atual do circuito (para cópia privada da tarefa), expansão e *traceback*. Para permitir que diferentes tarefas executem estes passos em simultâneo, uma tarefa que acabou de ler o estado atual do circuito (para a sua cópia privada) não deve manter os trincos respetivos trancados, de forma a permitir que entretanto seja possível a outras tarefas modificarem o circuito com novos caminhos. Isto traz uma consequência importante: a cópia privada sobre a qual uma tarefa calcula o seu próximo caminho pode não estar consistente com o circuito atual; logo, o caminho escolhido pela tarefa pode, afinal, não ser possível no circuito atual. As soluções dos alunos devem ter este aspeto em conta na fase de expansão e/ou *traceback*.

3 Avaliação da eficiência da solução

Para medir a eficiência da solução, será utilizada uma métrica tipicamente utilizada para avaliar programas paralelos: *speed-up* ("aceleração") face à versão sequencial. O *speed-up* (1), neste caso, é dado pelo rácio entre o tempo de execução da versão sequencial e o tempo de execução da versão paralela:

$$speed\_up = \frac{t_{sequencial}}{t_{paralela}} \tag{1}$$

Quanto maior o *speed-up* obtido, melhor é o desempenho e eficiência da solução paralela. Por exemplo, um $speed\_up = 10$ corresponde a reduzir em 10 vezes o tempo de execução da implementação paralela, relativamente à versão sequencial.

Para medir o desempenho da sua solução, os alunos deverão desenvolver um *shell script*, chamado *doTest.sh*, que permita produzir, de forma automática, um ficheiro que reporte o *speed-up* ao variar o número de tarefas usadas pelo programa. O *script* deverá reportar o tempo de execução (que pode ser obtido no ficheiro *.res* correspondente à execução usando, p.e., os comandos *grep* e *cut*) e calcular o *speed-up*:

- Usando apenas 1 tarefa, sem nenhum mecanismo de sincronização, sendo este o tempo de referência ($t_{sequencial}$) para o cálculo do *speed-up* (i.e., o *speed-up* é por definição igual a 1 neste caso). Para este efeito, deverá ser usada a implementação sequencial, `CircuitRouter-SeqSolver`, desenvolvida durante o primeiro exercício (os alunos podem optar entre usar a solução fornecida pelos docentes ou a que desenvolveram ao resolver o Ex.1);
- Usando apenas 1 tarefa, mas usando os mesmos mecanismos de sincronização previstos para o caso de utilização de múltiplas tarefas. Este teste permitirá portanto avaliar o custo introduzido pelos mecanismos de sincronização implementados pelos alunos, num contexto em que o programa não tira nenhum partido deles;
- Usando 2, 3, ..., N tarefas, onde N é um parâmetro de entrada do *script*.

O *shell script* deverá receber dois argumentos posicionais obrigatórios: *i)* o número máximo de tarefas (N) e *ii)* o caminho do ficheiro com o problema a analisar. Deverá produzir um ficheiro de texto com os tempos de execução e *speed-ups* cujo nome será composto pelo nome do ficheiro de input, acrescentando o sufixo "`.speedups.csv`". Caso o ficheiro já exista, o seu conteúdo deverá ser truncado. O formato a usar para a produção do ficheiro deverá seguir as seguintes regras:

- Cada execução corresponde a uma linha;
- Para cada execução é apresentado o número de tarefas, o tempo de execução em segundos e o *speed-up* correspondente, usando 6 casas decimais;
- Para separar as colunas deve ser usado o caractere "`,`" (vírgula);
- Na execução de referência, que usa apenas **1 tarefa sem nenhum mecanismo de sincronização**, deve ser apresentado "`1S`" na coluna referente ao número de tarefas;

Por exemplo, ao executar o *script* com os seguintes parâmetros:

```
./doTest.sh 4 inputs/random-x128-y128-z5-n128.txt
```

deverá ser produzido um ficheiro com nome `inputs/random-x128-y128-z5-n128.txt.speedups.csv` com o seguinte conteúdo:

```
#threads,exec_time,speedup
1S,0.490712,1
1,0.511002,.960293
2,0.324582,1.511827
3,0.294138,1.668305
4,0.275881,1.778708
```

Sugestão: para calcular o *speed-up* no *script* é possível usar o comando *bc*, tal como ilustrado neste exemplo:

```
#!/bin/bash
seqTime=10
parTime=5
speedup=$(echo "scale=6; ${seqTime}/${parTime}" | bc)
echo Speedup = ${speedup}
```

4 Compilação e Submissão

A compilação e geração do executável `CircuitRouter-ParSolver` deve ser automatizada com recurso a um Makefile. Deve ser possível gerar o executável correndo apenas o comando `make` (sem argumentos).

Os alunos deverão submeter no Fénix um arquivo zip com os seguintes conteúdos:

- o código da solução e do *shell script*
- uma pasta chamada *results*, onde deverão ser inseridos os ficheiros com a análise do *speed-up* para todos os circuitos presentes na pasta inputs com tamanho até 256x256 (ou seja, todos os circuitos com excepção dos de tamanho 512x512, cuja resolução é bastante demorada). O número máximo de tarefas a usar para estes testes será igual a $2 \times num\_cores$, onde num_cores é o numero de *cores* disponíveis na máquina usada para executar os testes. Para estes testes poderá ser usada qualquer tipo de maquina *multi-core*, p.e., as máquinas disponíveis nos laboratórios ou um *laptop*.
- um ficheiro "README.txt" onde deverá ser descrita: i) a estrutura das directorias existentes no arquivo, ii) os passos para compilar e executar o projecto por linha de comando ou pelo *doTest.sh*, iii) uma descrição das caraterísticas do processador e sistema operativo usado pelo testes (i.e., número de *cores*, *clock rate* e modelo), que poderão ser obtidas, respectivamente, usando os comandos i) *cat /proc/cpuinfo* e ii) *uname -a*.

Tal como indicado no enunciado geral, o arquivo submetido não deve incluir outros ficheiros tais como binários, e o exercício deve **obrigatoriamente** compilar e executar nos computadores dos laboratórios.