Projeto de Sistemas Operativos 2017-18
`heatSim`
Exercício 1

LEIC-A / LEIC-T / LETI
IST

Resumo

Este documento constitui um guia para a realização do primeiro exercício do projeto da disciplina de Sistemas Operativos. Consiste do desenvolvimento de uma versão paralela do problema recorrendo ao paradigma da troca de mensagens, recorrendo a uma condição simples de paragem. Para a resolução deste exercício, os alunos usarão uma biblioteca que concretiza a troca de mensagens, fornecida pelo docentes.

1 Introdução

Os alunos devem ler primeiro o documento que faz uma introdução ao projeto, assim como os guias do exercício 0, antes de lerem este guia.

Neste primeiro exercício, o objetivo é construir uma versão paralela do `heatSim` baseada em troca de mensagens.

2 Estratégia de Paralelização

A simulação será executada por uma tarefa mestre (que executa o corpo principal do programa) e por um conjunto de tarefas trabalhadoras que se executam em paralelo. A tarefa mestre cria as tarefas trabalhadoras, indica a cada uma destas tarefas qual o trabalho que deve realizar, envia a cada trabalhadora os dados iniciais e posteriormente recolhe os dados finais das várias trabalhadoras. Durante a execução da simulação, as trabalhadoras trocam entre si a informação necessária para poderem realizar o seu trabalho.

A tarefa mestre deve usar a interface das *pthreads* para criar as tarefas trabalhadoras. Esta interface permite que seja passada uma estrutura de dados para a tarefa que é criada. Esta estrutura pode ser usada pelo mestre para passar informação de controlo às trabalhadoras, tal como o número de trabalhadoras que existe no total, qual o trabalho que cabe a cada uma, etc.

Após a criação das tarefas trabalhadoras, toda a comunicação e sincronização entre o mestre e as trabalhadoras deve ser realizada por troca de mensagens. Quando as trabalhadoras necessitam de trocar informação ou de se sincronizar entre si, devem também fazê-lo por troca de mensagens.

A estratégia de paralelização consiste em dividir a superfície em fatias horizontais. O número de fatias deve ser igual ao número de trabalhadoras. A cada iteração, cada trabalhadora calcula os novos valores da fatia que lhe é atribuída. Desta forma, os valores de diferentes fatias podem ser calculados em paralelo por diferentes *cores* do processador. Como será explicado de seguida, antes de realizar uma nova iteração, cada trabalhadora necessita de saber alguns dos valores que foram calculados pelos seus vizinhos na iteração anterior. Por isso, no final de cada iteração as trabalhadoras devem trocar informação.

3 Biblioteca de Troca de Mensagens

Para realizarem este projeto, os alunos devem usar uma biblioteca de troca de mensagens que é fornecida. Esta biblioteca exporta apenas 4 funções: uma função para iniciar as estruturas de dados necessárias para o seu funcionamento; uma função para libertar estas estruturas quando a biblioteca já não é necessária; uma função para enviar mensagens e outra para receber mensagens.

A biblioteca concretiza canais de comunicação unidirecionais com capacidade limitada. A capacidade do canal refere-se ao número de mensagem que este pode memorizar, não existindo limite ao tamanho de cada mensagem. Cada canal é definido por uma tarefa de origem e uma tarefa de destino. Assim, se existirem t tarefas (tipicamente 1 mestre e várias trabalhadoras), existirão t^2 canais, todos com a mesma capacidade.

De seguida as funções para enviar e receber mensagens são descritas sucintamente:

```
int enviarMensagem(int tarefaOrig, int tarefaDest, void *msg, int tamanho);
```

Quando uma tarefa quer enviar uma mensagem, indica qual o seu identificador (processo de origem), o identificador do processo de destino, fornece um ponteiro para os dados a transferir e indica o tamanho da mensagem. Se o canal não estiver cheio, os dados são copiados para um *tampão* temporário gerido pela biblioteca e a função retorna o tamanho da mensagem. Se o canal estiver cheio o emissor é bloqueado até esta condição deixe de se verificar (isto é, quando a tarefa de destino ler mensagens do canal).

```
int receberMensagem(int tarefaOrig, int tarefaDest, void *buffer, int tamanho);
```

Quando uma tarefa quer receber uma mensagem, indica qual o identificador do processo de origem, o seu identificador (processo de destino), fornece um ponteiro para a zona de memória onde os dados devem ser colocados e indica o tamanho máximo da mensagem que pode receber. Se o canal não estiver vazio, uma mensagem será removida do canal e copiada do *tampão* mantido pela biblioteca para a zona indicada pelo recetor; nesse caso, a função retorna o tamanho (em bytes) da mensagem recebida. Se o canal estiver vazio, o recetor é bloqueado até esta condição deixe de se verificar (isto é, quando a tarefa de origem enviar mensagens para canal).

A biblioteca suporta também canais com capacidade 0. Isto significa que os dados do emissor nunca são copiados para um tampão temporário nem mantidos pelo canal até que o recetor esteja pronto para os receber. Pelo contrário, o emissor fica bloqueado até que o recetor leia do canal e, nesse momento, os dados são copiados diretamente da memória do emissor para a memória do recetor, sem recorrer a um *tampão* temporário (poupando uma cópia). Este modo de funcionamento é conhecido por "*rendez-vous*".

4 Parâmetros do Programa

O `heatSim` deve ser lançado numa linha de comandos com os seguintes argumentos:

```
heatSim N tEsq tSup tDir tInf iter trab csz
```

Todos os argumentos acima são obrigatórios, sendo o seu significado:

- `N` corresponde ao número de linhas/colunas contendo pontos interiores. A estas linhas e colunas devem ser adicionadas as arestas, logo cada lado da matriz inicial terá um comprimento efetivo de $N + 2$ pontos. Este argumento é do tipo inteiro e deve ser igual ou superior a 1.
- `tEsq`, `tSup`, `tDir` e `tInf` indicam as temperaturas medidas em cada aresta da superfície quadrada (arestas esquerda, superior, direita e inferior, respetivamente). São do tipo *double*, tendo valor igual ou superior a zero.
- `iter` é um inteiro positivo que define ao fim de quantas iterações o `heatSim` conclui a simulação e imprime a matriz calculada.

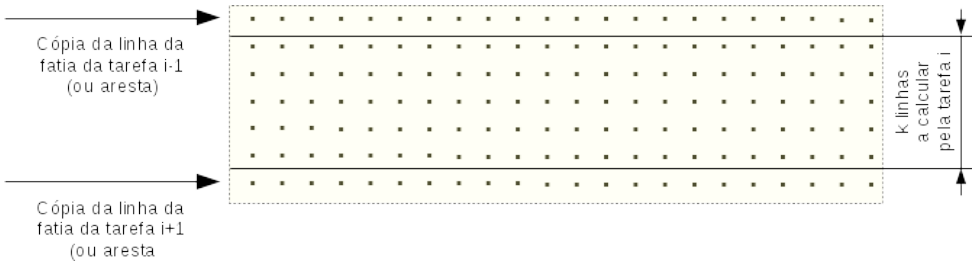

Figura 1: Fatia atribuída à tarefa i. A fatia contém $k+2$ linhas. As k linhas interiores (fatia interior) são aquelas cujos valores serão calculados pela tarefa i. As restantes duas linhas (superior e inferior) são cópias auxiliares, que contêm os valores recebidos por mensagem relativos às fatias adjacentes (ou das arestas da matriz).

- `trab` é um inteiro positivo que define quantas tarefas trabalhadoras devem executar a simulação. O seu valor deve ser tal que `N` seja um múltiplo de `trab`. O caso em que `trab=1` corresponde a uma simulação sequencial.
- `csz` é um inteiro positivo que define quantas mensagens cada canal de comunicação pode armazenar.

5 Pormenores sobre a Paralelização

Cada fatia é composta por um conjunto de linhas consecutivas:

- Um conjunto de linhas (mais precisamente, $k = N/trab$ linhas) contendo pontos interiores que serão calculados pela tarefa que possui esta fatia. A este conjunto de k linhas chamamos a *fatia interior*.
- Adicionalmente, cada fatia é complementada pelas 2 linhas imediatamente adjacentes (linha inferior e linha superior). Estas linhas ou correspondem a uma aresta horizontal da matriz original ou a uma linha adjacente pertencente a uma fatia de outra tarefa. Como será explicado de seguida, estas 2 linhas são cópias auxiliares, cujos valores não são calculados pela tarefa que possui esta fatia.

A Figuras 1 e 2 ilustram como as fatias são decompostas e atribuídas a cada tarefa trabalhadora.

Como exemplo, a fatia da tarefa 0 inclui as linhas 0 a $k+1$. Entre estas, as linhas 1 a k constituem a fatia interior, cujas temperaturas serão calculadas pela tarefa 0. Nos dois extremos da fatia, a linha 0 é uma aresta da matriz original e a linha $k+1$ é uma cópia da linha da fatia da tarefa 1.

Mais genericamente, podemos definir que a fatia da tarefa i corresponde às linhas $(k \times i)$ a $(k \times (i+1)) + 1$. Entre estas, as linhas entre $(k \times i) + 1$ a $(k \times (i+1))$ constituem a fatia interior da tarefa i; por outro lado, as duas linhas limite da fatia (linhas $(k \times i)$ e $(k \times (i+1)) + 1$) correspondem a uma aresta (no caso da primeira e última fatia) ou a uma linha adjacente sobreposta com outra fatia.

Para que o resultado final seja igual àquele que seria obtido com uma execução sequencial, as tarefas trabalhadoras precisam de se coordenar:

- Em cada iteração, cada tarefa calcula as temperaturas de cada ponto interior da sua fatia interior, recorrendo ao algoritmo sequencial descrito no documento com a panorâmica do projeto.
- Após terminar uma iteração sobre a sua fatia interior, cada tarefa trabalhadora comunica às tarefas adjacentes os novos valores da primeira e última linhas da sua fatia interior. Ou seja, a tarefa i envia uma mensagem à tarefa $i-1$ (caso exista) com o conteúdo da linha $k \times i + 1$; e uma mensagem à tarefa $i+1$ (caso exista) com os valores da linha $k \times (i+1)$.

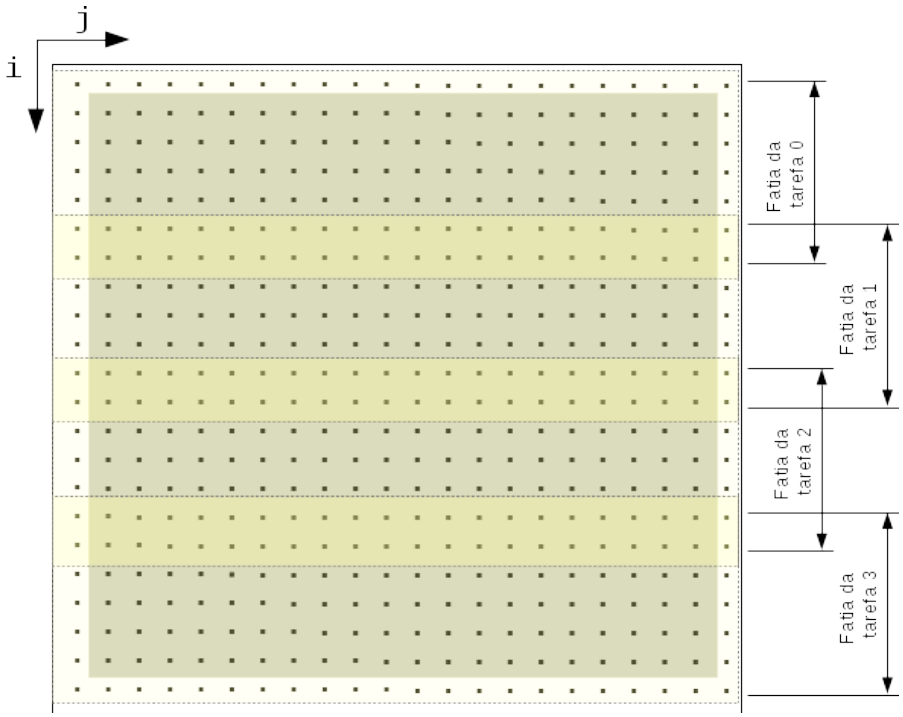

Figura 2: Divisão da matriz em fatias, em exemplo com 4 tarefas. Entre cada par de fatias adjacentes, há 2 linhas que são incluídas em ambas as fatias.

- A tarefa i espera até receber uma mensagem equivalente das tarefas adjacentes (tarefa $i-1$ e tarefa $i+1$, caso existam). Assim que receber cada mensagem, a tarefa i copia o seu conteúdo para a linha (auxiliar) correspondente na sua fatia.
- Finalmente, caso não tenha sido excedido o limite de iterações, a tarefa prossegue para a próxima iteração.

A troca de mensagens deve ser implementada recorrendo à biblioteca `mplib3`, fornecida no site do projeto.

6 Experimente

Para uma matriz com $N = 20$, experimente correr com 10 iterações e com 100 iterações. Um maior número de iterações contribui para uma maior precisão?

Experimente resolver a mesma matriz com diferentes números de tarefas trabalhadoras, mas igual número de iterações. O resultado final é idêntico? *Sugestão: em cada execução, redirecione o* stdout *para um ficheiro diferente usando o símbolo* > *na linha de comandos; no final, compare os diferentes ficheiros usando o comando* diff *e confirme que não há diferenças.*

Com uma matriz relativamente grande (por exemplo, 1024 por 1024), compare os tempos gastos com execução sequencial vs. tantas tarefas quanto o número de núcleos hardware da sua máquina. *Sugestão: use o comando* time *para medir o tempo de execução do programa na linha de comandos.*

Experimente usar canais com capacidade 0 (zero), ou seja em modo *rendez-vous*. Confirme que a sua solução continua correta.

7 Entrega e avaliação

Os alunos devem submeter um ficheiro no formato zip com o código fonte e o ficheiro *Makefile*. O exercício deve obrigatoriamente compilar e executar nos computadores dos laboratórios.

A data limite para a entrega do primeiro exercício é 13/10/2017 até às 23h59m. A submissão é feita através do Fénix.

Após a entrega, o corpo docente disponibilizará a codificação da respetiva solução, que pode ser usada pelos alunos para desenvolverem os exercícios seguintes.