Trabalho 2

T2G1, CLE, MEI, 2020

Por Rafael Direito (84921)

Professor: António Borges

universidade de aveiro

deti
departamento de eletrónica, telecomunicações e informática

Exercício 1

O primeiro problema consiste na leitura e processamento de ficheiros de texto, em paralelo, e determinar o número de vogais presentes numa palavra, assim como o comprimento associado a cada uma dessas palavras.

Continuando o trabalho desenvolvido no primeiro projeto, iremos migrar a solução de um paradigma multi-thread, para uma solução que utiliza MPI como interface de comunicação. Esta nova implementação levou a algumas modificações:

- Existe um *dispatcher* que envia porções de dados aos workers, em vez de serem os *workers* a pedir estes dados a um *master*;
- Deixa de existir uma memória partilhada entre *master* e *workers*;
- Deixa de existir a imposição de monitores nas zonas críticas, que desaparecem.
- Melhoria da implementação da solução. Na primeira entrega não tinham sido utilizadas *Structs*, para comunicação entre entidades, algo que foi implementado neste segundo projeto.

Nesta apresentação, não serão descritos os processos de cálculo/lógica necessários para calcular a frequência dos tamanhos das palavras / número de vogais, uma vez que esta lógica já foi apresentada na entrega anterior. Focar-me-ei na utilização de MPI.

Assim:

- Relativamente à troca de mensagens, apenas recorremos aos métodos *MPI_Recv* e *MPI_Send*, uma vez que são suficientes para a implementação desejada;
- Relativamente ao trabalho a ser enviado, o *dispatcher* obtém N pedaços de dados (N = número de *workers*), caso existam, e envia-os aos, em simultâneo, para os *workers*, de forma a possam processar os dados. Posteriormente, bloqueia à espera que os *workers* lhe enviem os resultados.
- Com a utilização da estrutura ControlInfo, para enviar trabalho, o *dispatcher* apenas necessita de uma mensagem para enviar dados para os clientes. A estrutura é convertida em *bytes* e enviada para o *worker.*

Exercício 1

Inicialmente, o *dispatcher* lê e armazena o nome dos ficheiros a serem processados. Após isto, verifica, repetidamente, se há informação a ser processada.

Sempre que existir informação a ser processada:

- O *dispatcher* irá enviar, em simultâneo e para todos os *workers*, uma mensagem inicial, um *booleano,* a indicar que existem dados a ser processados. Com esta informação, cada *worker* bloqueia à espera dos ditos dados. O *dispatcher* envia, então, uma instância da estrutura ControlInfo, que contém todos os dados necessários para os cálculos a serem efetuados. Bloqueando à espera que os *wrokers* envie, os resultados. Por outro lado, cada *worker,* assim que recebe a mensagem do *dispatcher*, irá invocar o método *process_data(controlInfo)*, que calcula os resultados esperados e os armazena na estrutura ControlInfo, que enviará ao *dispatcher.* Este irá guardar estes resultados na sua *stack* de memória.

Assim que não houver mais informação a processar:

- O *dispatcher i*rá enviar uma mensagem *booleana* a todos os *workers,* informando-os que podem terminar o seu ciclo de vida, e termina, também, o seu ciclo de vida, escrevendo, para a consola, os resultados do processamento efetuado.

Relativamente aos dados enviados através do MPI, são enviados dados do tipo *MPI_C_BOOL* e *MPI_BYTE,* para enviar, respetivamente, mensagens indicativas da existência, ou não, de dados a serem processados, e a estrutura *ControlInfo*, com os dados a serem processados e valores necessários para o seu processamento. Esta última estrutura também poderia ter sido enviada com recurso a uma mensagem do tipo *MPI_PACKED.*

Exercício 2

Relativamente ao exercício 2, foram menos as alterações necessárias, uma vez que já tinha sido implementada a utilização de *Structs* para a comunicação entre *master* e *workers*. As alterações necessários para este exercício são aquelas já descritas na análise do exercício 1.

No que toca à escolha do *dispatcher* e dos *workers,* esta é feita com base no *rank* de cada processo MPI. Este *rank* é atribuído aquando a inicialização dos processos. O processo com *rank=0* irá ser o *dispatcher* e os restantes processos serão *workers*. Mais uma vez, o *dispatcher* irá enviar, paralelamente, dados para todos workers.

Relativamente aos ciclo de vida do *dispatcher* e do *workers*, numa fase inicial, o *dispatcher* irá ler os nomes dos ficheiros a serem processados, do *argv*, sendo que, posteriormente:

<u>Sempre que existir informação a ser processada:</u>

- O *dispatcher* irá enviar, em simultâneo e para todos os *workers*, uma mensagem inicial, um *booleano,* a indicar que existem dados a ser processados. Com esta informação, cada *worker* bloqueia à espera dos ditos dados. O *dispatcher* envia, então, uma instância da estrutura ControlInfo, que contém o número de valores do sinal a ser processado, o índice do ficheiro ao qual são relativos e um conjunto de N valores de *t* que devem ser processados. Após isto, será necessário enviar, separadamente, os sinais *x* e *y*. Isto acontece, pois a estrutura controlInfo guarda um ponteiro para estes e, uma vez que os *workers* e o *dispatcher* têm *stacks* de memória independentes, caso os *workers* recebem este ponteiro, não seriam capazes de obter os valores dos sinais. Assim, estes valores têm de ser inseridos num *array* de *doubles*, no *dispatcher,* e enviados individualmente noutra mensagem. Seguidamente, o *dispatcher* bloqueia, então, à espera que os *workers* enviem os resultados. Assim que um *worker* recebe dados a processar vindos do *dispatcher*, irá invocar o método *process_data(controlInfo)*, que calcula os resultados esperados e os armazena na estrutura ControlInfo, que enviará ao *dispatcher.* Este irá guardar os resultados na sua *stack* de memória.

Exercício 2 - Estrutura

Assim que não houver mais informação a processar:

- O *dispatcher i*rá enviar uma mensagem *booleana* a todos os *workers,* informando-os que podem terminar o seu ciclo de vida, e termina, também, o seu ciclo de vida, escrevendo, para a consola, os resultados do processamento efetuado.

```
//The structure containers pointers to the arrays x and y. The MPI can't pass this pointers, so
// we will have to pass them as arrays
double x_signal[controlInfo.nSignals];
double y_signal[controlInfo.nSignals];

for (int i = 0; i < controlInfo.nSignals; i++) {
    x_signal[i] = controlInfo.x[i];
    y_signal[i] = controlInfo.y[i];
}

// send the signals
MPI_Send(x_signal, controlInfo.nSignals, MPI_DOUBLE, workerAssigned, 0, MPI_COMM_WORLD);
MPI_Send(y_signal, controlInfo.nSignals, MPI_DOUBLE, workerAssigned, 0, MPI_COMM_WORLD);
```

Carregamento dos valores dos sinais *x* e *y* para *arrays* de *doubles*, que possam ser enviados, usando MPI.

Resultados do exercício 1:

Cada bloco de dados a processar contém 5 tokens

	1 worker	2 workers	4 workers
Tempo de execução	0.1170 *s*	0.0991 *s*	0.0763 *s*

Cada bloco de dados a processar contém 10 tokens

	1 worker	2 workers	4 workers
Tempo de execução	0.0981 *s*	0.0959 *s*	0.0895 *s*

Cada bloco de dados a processar contém 100 tokens

	1 worker	2 workers	4 workers
Tempo de execução	0.0934 *s*	0.0739 *s*	0.0801 *s*

Cada bloco de dados a processar contém 1000 tokens

	1 worker	2 workers	4 workers
Tempo de execução	0.0486 *s*	0.0744 *s*	0.0786 *s*

Resultados do exercício 2:

Cada bloco tem 5 valores de *t* a serem calculados

	1 worker	2 workers	4 workers
Tempo de execução	34.976 *s*	21.779 *s*	13.536 *s*

Cada bloco tem 10 valores de *t* a serem calculados

	1 worker	2 workers	4 workers
Tempo de execução	24.871 *s*	16.627 *s*	9.777 *s*

Cada bloco tem 20 valores de *t* a serem calculados

	1 worker	2 workers	4 workers
Tempo de execução	28.718 *s*	10.669 *s*	6.3310 *s*

Cada bloco tem 40 valores de *t* a serem calculados

	1 worker	2 workers	4 workers
Tempo de execução	18.892 *s*	7.1115 *s*	4.6331 *s*

Conclusões

Relativamente ao resultados atingidos podemos fazer uma análise diferenciada para o problema 1 e para o problema 2, isto, também, porque o problema 1 não é, de todo, computacionalmente exigente, enquanto que o problema 2 já o é.

No que toca ao problema 1, são poucas as conclusões que podemos tirar, uma vez que os resultados acabam por ser todos bastante semelhantes, pelo que os outros processos a correr na máquina utilizada para a obtenção de resultados, podem afetar as conclusões tiradas. Isto já não acontece no problema 2, onde existe uma elevada diferença entre os resultados obtidos.

No problema 1, observamos que, quando aumentamos o número de tokens a processar por cada worker, de facto, obtemos melhores resultados. Quando aumentamos o número de workers, para o mesmo número de tokens a processar, os resultados, podem manter-se, melhorar ou piorar. Isto acontece pois o problema 1 é um problema pouco existente de um ponto de vista computacional. Note-se que nesta implementação, com MPI, os resultados não são necessariamente piores para um maior número de workers, o que é devido à não existência de condições de corrida.
s

Relativamente ao problema 2, aqui podemos retirar melhores conclusões face ao uso de MPI. Observámos que, quando aumentamos a quantidade de dados a ser processados, por *batch* recebida, o desempenho é melhor. Isto também se verifica quando aumentamos o número de *workers*.

Podemos associar este fenómeno, novamente, ao facto de não existir uma zona crítica, pela o acesso à qual os *workers* competem entre si. Na implementação anterior, esta condição e corrida, levava a que o *workers* ficassem bastante tempo à espera de acesso à região crítica, enquanto que, com MPI, é o *dispatcher* que acede aos dados e os envia para os *workers*, evitando que estes compitam entre si.

De uma forma geral, os resultados com 1 *worker* apenas foram melhores num paradigma *multithreading*, contudo, quando começamos a adicionar mais workers, a implementação com MPI vence claramente. Mais um vez, podemos relacionar este acontecimento com as condições de corrida. Na implementação *multithreading* com apenas 1 *worker*, não se davam condições de corrida, daí obtermos melhores resultados nesse cenário.