Computação em Larga Escala - Trabalho 2

T2 - Grupo 1: Alina Yanchuk - 89093 - Mestrado em Engenharia Informática
Ana Sofia Moniz Fernandes - 88739 - Mestrado em Engenharia Informática

Maio de 2021

Exercício 1

Este exercício consiste na leitura de ficheiros de texto, cujos nomes são passados como argumentos ao programa, e no seu processamento de modo a realizar uma contagem de palavras e o cálculo da frequência da ocorrência de consoantes em cada palavra, para cada um dos ficheiros. É de notar que, no processamento do texto, a vogal “ü” é transformada para “u” e, como tal, é tida em conta na contagem.

Neste trabalho, iremos ter por base a solução previamente desenvolvida em *multi-thread*, realizada no primeiro trabalho prático, e adaptá-la para uma solução que utiliza MPI como interface de comunicação. Assim, existem algumas considerações que devem ser tidas em conta:

- Existe, agora, um ***dispatcher***, cujas responsabilidades são processar os nomes dos ficheiros de texto passados como argumento e distribuir os pedaços de dados pelos diferentes ***workers***. Quando todo o trabalho estiver feito (neste caso, as contagens de palavras e cálculos de ocorrências de consoantes de cada ficheiro), ele informa os *workers* que não existe mais trabalho por realizar. Tal como acontecia no trabalho anterior, para processar um *chunk* é usado um *buffer* de caracteres que formam palavras completas, e para guardar resultados parciais é usada uma estrutura do tipo *struct* (uma para cada ficheiro);
- Deixam de existir monitores e regiões críticas;
- O processamento deixa de ser concorrente e passa a paralelo;
- Para troca de mensagens, usamos apenas os métodos ***MPI_Recv*** e ***MPI_Send***, visto serem suficientes na nossa implementação;
- Os dados que circulam nas mensagens de *MPI* são do tipo
    - *MPI_C_BOOL* para que o *dispatcher* envie aos *workers* um booleano que diz se existe, ou não, dados para processar;
    - *MPI_CHAR* para o *dispatcher* enviar, ao *worker*, o *buffer* que contém caracteres que formam palavras completas, cujas o *worker* deve processar e fazer os cálculos necessários;
    - *MPI_BYTE* para fazer o envio ao *dispatcher* da estrutura *PartFileInfo*, convertida em *bytes*, que transporta os dados processados pelo *worker* para um dado *buffer*.
- De cada vez que o *dispatcher* recebe, de um *worker*, informação já pronta a ser guardada (ou seja, contagens de palavras e cálculos de ocorrências no respetivo *buffer* recebido), vai guardando esses resultados parciais numa *struct* final do mesmo tipo para, quando não existir mais trabalho para os *workers* realizarem, poder finalmente imprimir os resultados finais.

```
/** \brief struct to store data of one file*/
typedef struct {
   int  fileId;    /* file with data */
   int n_words;    /* number words */
   int n_chars; /* number chars */
   int n_consonants;    /* number consonants */
   int in_word;     /* to control the reading of a word */
   int max_chars;   /* max chars found in a word */
   int counting_array[50][51];    /*  to store and process the final countings  -> counting_array[MAX_SIZE_WORD][MAX_SIZE_WORD+1]*/
   bool firstProcessing; /* indicates wether it is the first time processing that file or not */
   bool done;        /* to control the end of processing */
} PartFileInfo;
```

Exercício 1 - Resultados

Ficheiro *text0.txt* - 14 palavras

Nº *workers*	1	2	4
Tempo de Execução (s)	0.004050 0.006474	0.002572 0.007249	0.003514 0.194233

Ficheiro *text1.txt* - 1184 palavras

Nº *workers*	1	2	4
Tempo de Execução (s)	0.034702 0.110930	0.025152 0.092663	0.029405 3.500800

Ficheiro *text2.txt* - 11027 palavras

Nº *workers*	1	2	4
Tempo de Execução (s)	0.305043 0.714531	0.227798 0.620404	0.260162 35.602815

Ficheiro *text3.txt* - 3369 palavras

Nº *workers*	1	2	4
Tempo de Execução (s)	0.094058 0.206675	0.070187 0.181883	0.084540 9.473102

Ficheiro *text4.txt* - 9914 palavras

Nº *workers*	1	2	4
Tempo de Execução (s)	0.285957 0.616815	0.209897 0.832675	0.247734 32.578175

Os 4 ficheiros de texto

Nº *workers*	1	2	4
Tempo de Execução (s)	0.715493 1.579527	0.527496 1.377598	0.605375 86.368328

Nota: os resultados foram obtidos usando dois computadores diferentes

- Linha de cima- Mac Air M1 Silicone, com 8 cpus
- Linha de baixo - Lenovo Yoga 710 intel core i7, com 4 cpus

Exercício 2

O segundo exercício consiste na leitura de ficheiros binários, cujos nomes são passados como argumentos ao programa, e no seu processamento de modo a realizar o cálculo da correlação cruzada circular entre um par de sinais, para cada ficheiro.

Neste trabalho, iremos ter por base a solução previamente desenvolvida em multi-thread, realizada no primeiro trabalho prático, e adaptá-la para uma solução que utiliza MPI como interface de comunicação. Assim, existem algumas considerações que devem ser tidas em conta:

- Existe, agora, um ***dispatcher***, cujas responsabilidades são processar os nomes dos ficheiros binários passados como argumento e distribuir os pedaços de dados pelos diferentes ***workers***. Quando todo o trabalho estiver feito (não existirem mais pontos para processar), ele informa os *workers* que não existe mais trabalho por realizar.
- Deixam de existir monitores e regiões críticas;
- O processamento deixa de ser concorrente e passa a paralelo;
- Para troca de mensagens, usamos apenas os métodos ***MPI_Recv*** e ***MPI_Send****,* visto serem suficientes na nossa implementação;
- Os dados que circulam nas mensagens de *MPI* são do tipo
    - *MPI_C_BOOL* para que o *dispatcher* envie aos *workers* um booleano que diz se existem, ou não, dados para processar;
    - *MPI_INT* para o *dispatcher* enviar, ao *worker*, um *buffer* com dois inteiros que são o ponto de processamento atual e o tamanho dos sinais;
    - *MPI_DOUBLE* para o *dispatcher* enviar, ao *worker*, um *buffer* com o sinal X e outro com o sinal Y; e também para o *worker* enviar, ao *dispatcher*, o sinal XY que processou num dado ponto;
- De cada vez que o *dispatcher* recebe, de um *worker*, informação já pronta a ser guardada (ou seja, o valor XY calculado num dado ponto), vai guardando esses resultados parciais numa *struct* final, para quando não existir mais trabalho para os *workers* realizarem, poder finalmente imprimir os resultados finais.

```
/** \brief struct to store data to process signal of one file*/
typedef struct {
   int  fileId;    /* file with data */
   int signalLength;  /* length of signals */
   double *x;   /* signal X */
   double  *y;   /* signal Y */
   double *xy;   /* will store the calculated signal XY */
   double *xyCorrect;   /* correct signal XY */
   int point;   /* point of processing */
   bool done;   /* to control the end of processing */
} Signal;
```

Exercício 2 - Resultados

Ficheiro *newSigVal01.bin* - 1024 elementos em cada sinal

Nº *workers*	1	2	4
Tempo de Execução (s)	0.010375 0.030418	0.006268 0.012949	0.009439 2.805357

Ficheiro *newSigVal02.bin* - 2048 elementos em cada sinal

Nº *workers*	1	2	4
Tempo de Execução (s)	0.031031 0.047920	0.023633 0.057791	0.024437 5.196485

Nota: os resultados foram obtidos usando dois computadores diferentes

- Linha de cima- Mac Air M1 Silicone, com 8 cpus
- Linha de baixo - Lenovo Yoga 710 intel core i7, com 4 cpus

Ficheiro *newSigVal03.bin* - 4096 elementos em cada sinal

Nº *workers*	1	2	4
Tempo de Execução (s)	0.119872 0.159085	0.087879 0.197767	0.093791 10.586882

Ficheiro *newSigVal04.bin* - 8192 elementos em cada sinal

Nº *workers*	1	2	4
Tempo de Execução (s)	0.468224 0.613726	0.351894 0.578707	0.354778 10.586882

Os 4 ficheiros - 15360 elementos em cada sinal no total

Nº *workers*	1	2	4
Tempo de Execução (s)	0.623815 0.817832	0.469229 0.770023	0.453370 68.550931

Conclusões

Neste trabalho, por forma a conseguirmos alcançar melhores conclusões, comparámos os resultados obtidos em dois computadores:

1. Mac Air M1 silicone, com memória ram de 16 gb. O processador é o M1 da Apple, com 4 núcleos de desempenho e 4 núcleos de eficiência (ou seja, 8 cpus), e o sistema operativo é o macOS Big Sur;
2. Lenovo Yoga 710, com memória ram de 16 gb. O processador é o Intel Core i7-7500U 2.70GHz, com 4 cpus, e o sistema operativo é o Ubuntu 18.04.

Relativamente ao **primeiro exercício**:

1. Ambos os computadores têm melhor desempenho para 2 workers, e pior para 4;
2. O computador Mac Air M1 apresenta melhoria de desempenho de 1 worker para 4 workers em todos os casos;
3. Tal como no trabalho anterior, quanto maior o número de palavras, pior é o desempenho;
4. Para ficheiros de maior tamanho (como é o caso dos ficheiros text2.txt e text4.txt) o tempo de execução entre 2 e 4 workers tem um maior aumento;
5. Enquanto que, para 4 workers e com 4 ficheiros, no computador Mac Air M1 se verifica um aumento (de 2 workers para 4) de apenas cerca de 0.1s, no Lenovo Yoga este aumento é de cerca de 85 segundos.

Os aumentos de tempo com mais processos podem ser explicados devido ao facto de o problema ser bastante simples e pouco exigente em termos computacionais, fazendo com que a maior parte do tempo de execução seja utilizada na troca de mensagens entre os diferentes processos e não no próprio processamento paralelo, que é bastante rápido. Por outro lado, coerência da cache também pode ser um motivo.

Relativamente ao **segundo exercício**:

1. Ambos os computadores apresentam um aumento no tempo de execução de 2 workers para 4 - a exceção a esta regra é o Mac Air M1, quando se executa o programa com os 4 ficheiros binários e 4 workers, pois apresenta uma redução de cerca de 0.01s;
2. Enquanto que, para 4 workers e com 4 ficheiros, no computador Mac Air M1 se verifica uma diminuição (de 2 workers para 4) de cerca de 0.01s, no Lenovo Yoga aumenta cerca de 61 segundos.

Aqui, os grandes aumentos de tempo no computador Lenovo podem ser justificados devido ao menor número de cpus, fazendo com que os processos compitam entre si. Por outro lado, as diminuições de tempo são normais visto que deixamos de ter uma região crítica e partilha de recursos, permitindo aos processos realizar operações computacionalmente exigentes em paralelo, sendo que os mesmos apenas têm de esperar pela recepção das mensagens, que influenciam grande parte do tempo de execução.

Concluímos, mais uma vez, que as características do computador onde o programa corre influenciam bastante a performance.