Computação Paralela - Trabalho Prático nº3

Diogo Manuel Brito Pires
Mestrado em Engenharia Informática - 1ºano
Universidade do Minho - Braga, Portugal
PG50334

Miguel Ângelo Machado Martins
Mestrado em Engenharia Informática - 1ºano
Universidade do Minho - Braga, Portugal
PG50655

***Abstract*—3ª e última etapa do trabalho prático. Desenvolver um programa que explore de forma eficiente o paralelismo para obtenção de melhores tempos de execução, podendo-se, eventualmente, recorrer a ambientes de programação paralela alternativos, como o MPI ou o CUDA.**

I. Descrição de diminutivos utilizados

Para facilitar a compreensão do relatório, é importante notificar previamente que durante o mesmo iremos utilizar os seguintes diminutivos:

- N - Número de amostras;
- K - Número de clusters;
- P - Número de processos utilizados para correr o programa;
- T - Número de threads que cada processo tem.

II. Correção/otimização do algoritmo

Antes de experimentarmos novas formas de paralelismo, decidimos corrigir um *data race* que tínhamos na versão anterior, na função *attribution*. Nesse *data race*, várias *threads* podiam escrever na variável *change*, o que poderia causar problemas. No nosso entender, a melhor maneira de o resolver seria com um *reduce*, que calcula o valor máximo dessa variável entre todas as *threads*. Assim, caso uma das *threads* altere esse valor para 1, equivalente a verdadeiro, a função retorna que pelo menos uma amostra trocou de *cluster*.

Para além disso, também retiramos algumas otimizações implementadas em fases anteriores, como alinhar os dados em memória, que deixa de fazer sentido ao implementarmos alocação dinâmica de memória.

A. Exploração de outras formas de paralelismo

Para implementar outras formas de paralelismo no nosso programa, foi necessário previamente analisar cada uma das suas componentes e verificar possíveis problemas que surgissem caso fossem executadas ao mesmo tempo:

- **Inicialização** - Existe um *array* de destino partilhado, em que cada *thread* tem de evocar atomicamente a função *random* duas vezes. Apesar de as amostras poderem estar guardadas em posições diferentes, as suas componentes têm de ser calculadas uma a seguir à outra, porque a função *random* é determinista. Não existe partilha de dados entre *threads*.
- **Atribuição** - A operação mais custosa é o cálculo das distâncias aos centros geométricos, e é feita de forma independente entre as várias *threads*, não existindo partilha de dados entre elas. Pelas análises das fases anteriores, esta função é a mais custosa das 3.
- **Cálculo do centro geométrico** - É necessário percorrer o *array* que associa as amostras aos *clusters*, e guardar a soma dos seus valores. Sendo assim, existe aqui partilha de dados entre as várias *threads*, pois têm de guardar as somas das distâncias e associá-las ao respetivo *cluster*. Neste caso, operações de *reduce* são ideais.

Após estudarmos algumas alternativas de implementação de paralelismo, decidimos que seria mais interessante implementarmos uma versão do nosso programa com MPI, e testá-la no *cluster*, pelas seguintes razões:

- É mais utilizado para arquiteturas distribuídas - O que torna interessante utilizarmos no *cluster*, que possui várias máquinas, permitindo maior distribuição da carga computacional;
- Permite implementar operações do tipo *reduce* com os vários processos;
- Não implica muitas alterações no código, permitindo experimentar facilmente uma versão sem MPI para comparações de performance.

III. Implementação

Para escolhermos onde implementar MPI, decidimos verificar quais as operações mais custosas, e que podiam ser feitas em paralelo. Para além disso, e após garantir que a versão MPI funcionava corretamente, decidimos implementar concorrência com *threads*, em cada processo, tirando proveito do trabalho realizado na fase anterior. Seguidamente, é descrita a implementação do nosso programa:

- A inicialização não foi alterada, e é feita em todos os processos, de forma sequencial;
- A função **attribution** é feita em paralelo, com cada processo a calcular $\frac{N}{P}$ elementos. Dentro de cada processo, utilizamos concorrência para que o cálculo das atribuições sejam feitas em paralelo. Também utilizamos a resolução do *data race* descrita no início do documento. Por fim, é feito um *MPI_Allreduce* para que todos os processos saibam se houve alteração nalguma amostra, e assim saber se o programa acabou. Desta forma, quando todos os processos comunicam a conclusão da

execução, o programa acaba. Nesta função mantivemos as instruções de *openMP* implementadas na fase anterior;
- A função **geometricCenter** também é executada em paralelo, sendo que cada processo calcula $\frac{N}{P}$ amostras. Assim, cada processo tem a soma das coordenadas dos *clusters* de $\frac{N}{P}$ elementos. Também utilizamos paralelismo dentro de cada processo, da mesma forma que no trabalho anterior, evitando *data races*. No fim, é preciso usar as funções *MPI_Allreduce* para que todos os processos saibam a soma de todas as amostras, e quantas amostras tem cada *cluster*.

Ao implementarmos este algoritmo fomos implementando algumas otimizações, como por exemplo:

- Tentamos transmitir o mínimo de dados, como se vê na função de atribuição. Para calcular os centros geométricos, cada processo só precisa das informações de $\frac{N}{P}$ amostras, que foram calculadas na atribuição, e não precisa de conhecer os valores de atribuições calculadas noutros processos.
- Quando usamos funções do comunicações de dados do MPI, preferimos minimizar o número de instruções no código, e por isso enviámos o array, e não cada posição individualmente. Isto porque, inicialmente, iterávamos por cada posição do *array* e usávamos uma função do MPI para cada valor. Contudo, depois melhoramos para usamos só uma função do MPI para todo o *array*.

IV. Testes a realizar

De forma a analisarmos a *performance* do nosso programa, decidimos efetuar os testes no *cluster*, estudando como o *hardware* afeta o desempenho da execução do programa. Antes de pensar nos testes, pensamos em como os parâmetros do problema afetam a execução em termos de *performance*, visto que seria uma abordagem interessante e de maior valor científico, efetuar uma análise teórica *a priori* e só postumamente comparar com os resultados práticos.

*1) **Análise do problema**:* Para analisar a escalabilidade do nosso programa, estudamos a influência do número de amostras e do número de *clusters* no desempenho:

- A memória necessária para resolver o problema aumenta da mesma forma aumentando em uma unidade N ou K, sendo um inteiro (4 *bytes*), e 2 *floats* (8 *bytes*);
- Aumentar o número de amostras implica mais iterações em todas as funções, enquanto que aumentar o número de *clusters* origina maior complexidade nas iterações dos ciclos (nas funções *attribution* e *geometricCenter*).
- Aumentar o número de *clusters* não afeta a função de inicialização.
- Quantos mais dados forem necessários para a execução do programa, mais acessos à memória teremos, piorando os tempos de execução do programa.

*2) **Impacto do número de processos na execução do programa**:* O primeiro conjunto de testes realizados analisa como o número de processos afeta o *speedup* da execução do programa. Para estes testes, decidimos escolher alguns valores de N, e variar o número de processos utilizados por execução (análise de escalabilidade forte). Nesta fase, ignoramos o paralelismo dentro de cada processo (não utilizando *openMP*).

Antes de executar, tentámos prever quais os resultados que iríamos obter, analisando a complexidade de cada uma das funções. Na análise descrita a seguir, decidimos ignorar ciclos de baixa complexidade, com K iterações, porque o seu tempo de execução é muito baixo quando comparado com os ciclos de N iterações. Caso os testes fossem realizados com mais *clusters*, teríamos de ter em consideração esses ciclos. Abaixo temos listada a complexidade de cada uma das funções utilizadas com base na análise teórica:

- **MPI_AllReduce** - Esta função tem a complexidade de $logP(\alpha + n\beta)$, sendo α uma constante bastante mais elevada que β. Assim, a quantidade de dados transferidos só afeta o desempenho da função quando é bastante elevado. Como não encontramos os valores precisos de α e β, e para facilitar as contas seguintes, decidimos considerar o custo desta função como $logP$, um valor arredondado por baixo. Quando testamos para valores de N elevados deveríamos ter em consideração o custo variável desta função.
- **Inicialização** - O desempenho desta função não é afetada pelo número de processos, pois cada processo inicializa sempre todas as amostras, apesar de só utilizar uma parte delas, logo o custo é N. Uma possível melhoria desta função é descrita na análise do trabalho futuro;
- **geometricCenter** - O número de elementos alterados por cada processo é dado por $\frac{N}{P}$, e com 3 *MPI_Allreduce* temos como custo total: $\frac{N}{P} \times 3logP$. Utilizando *openMP*, cada thread analisa $\frac{N}{P \times T}$ amostras, e o custo total por thread será: $\frac{N}{P \times T} \times 3logP$.
- **attribution** - Nesta função analisamos cada amostra K vezes, e mais uma vez, cada processo só analisa $\frac{N}{P}$ amostras. Como temos apenas uma função *MPI_Allreduce*, o custo total de execução desta função é: $\frac{N \times K}{P} \times logP$. Utilizando *openMP*, o custo de execução por *thread* é dado por: $\frac{N}{P \times T} \times logP$.

Sendo assim, o custo teórico de execução do nosso programa é dado pela soma das várias componentes, sendo:

$$\alpha N + \beta \frac{N}{P} \times 3logP + \omega \frac{N \times K}{P} \times logP$$

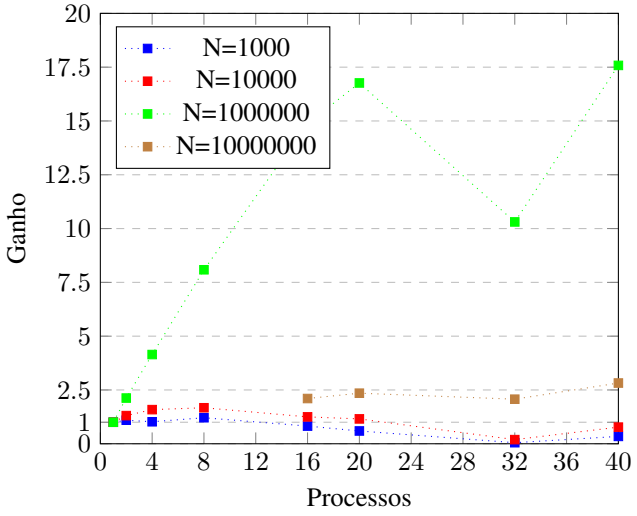
N é constante para cada conjunto de testes, e as constantes representam os custos das funções realizadas por elemento:

- α - Custo da função *random*;
- β - 3 operações de somas de inteiros;
- ω - 4 subtrações e 2 produtos sobre inteiros;

*3) **Influência do hardware** :* Também consideramos importante fazer considerações sobre como o hardware pode influenciar o desempenho do programa. Nos testes, decidimos atribuir preferencialmente os processos à mesma máquina, para minimizar os custos de trocas de mensagem. Isto porque, a comunicação de processos entre duas máquinas tem um custo superior por comparação a troca de mensagens dentro da mesma máquina. Para o conseguirmos, utilizamos a flag

–map-by core. Por isso, esperamos que a partir dos 20 processos, o ganho de *performance* não seja tão elevado, e pode até piorar, por causa dos elevados tempos de troca de informações entre máquinas diferentes.

*4) **Resultados do primeiro conjunto de testes**:* Antes de comentar os resultados obtidos, é importante referir que nos anexos podem ser consultadas tabelas com os tempos de execução e ganhos obtidos para os diferentes testes. Todas os *scripts* destes testes (que utilizam *perf* para o *profiling*) estão ainda disponíveis no repositório do projeto, sendo que dado o grande número de testes realizado, de forma a automatizar a criação dos diferentes *scripts*, o grupo recorreu a código *Python*, também ele disponível no repositório. De seguida apresentámos um gráfico que apresenta a variação do *speedup* (ganho) em função do número de processos, para diferentes tamanhos do problema. Em termos teóricos, previámos que o *speedup* fosse quase linear, mas isso só aconteceu para N=1000000.



Como podemos analisar pelo gráfico, para Ns pequenos (menores que 10000) só existe ganho até 8 processos, porque depois começa a não ter ganhos de performance. Esta perda pode ser explicada pelo elevado custo da comunicação entre processos relativamente aos benefícios de paralelizar. Cada processo analisa poucas amostras, mas perde muito tempo a sincronizar e trocar informações. Para além disso, e apesar de nestes casos a informação que cada processo trata poder ser guardada na *cache* L1, havendo por isso menos acessos à memória, o custo da troca de mensagens não compensa.

Para mais amostras, já compensa usar mais processos, como o gráfico demonstra. De salientar que não conseguimos testar com 10000000 amostras, porque era lançado um erro no *cluster* por excesso de tempo a executar o programa, logo podemos concluir que o ganho é real, e com menos processos demora bastante.

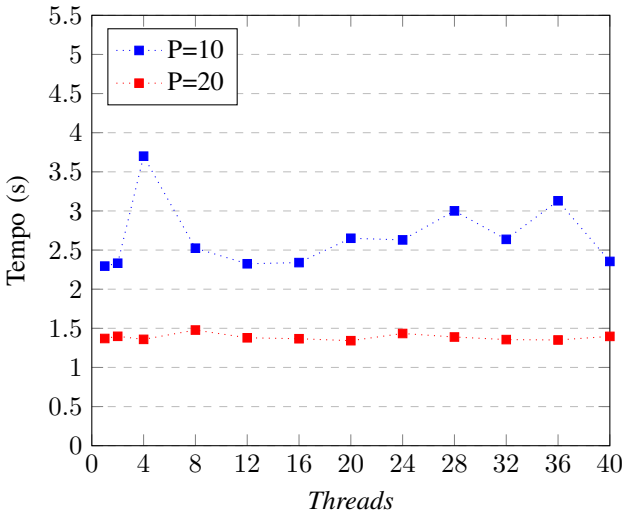
Logo, podemos concluir os seguintes aspetos:

- Pelos resultados obtidos a partir de Ns pequenos, descobrimos que o mínimo que cada processo deve analisar é 100 ou mais amostras;
- Nota-se que a partir de 20 começa a haver uma perda na performance, que só é recuperada com 40 processos, aproximadamente. Podemos, por isso prever, que entre 20 e 32 processos haja mais perdas no *speedup*;
- Como não temos mais resultados para o maior N, não sabemos como o número de processos influencia o seu *speedup*, mas com 1000000 (também ele um N considerável) podemos ver que até 20 processos a nossa previsão é aproximada, pois quantos mais processos são usados melhor é a *perfomance*.

Por fim, consideramos relevante analisar ainda as variações do *output* do programa nas várias vezes que corremos. Ao analisarmos o output do programa, notámos que havia mais variações (tanto no número de iterações, como nos centros geométricos) nos resultados com menos amostras, e isso pode ser explicado pela ordem de somas no *reduce*. Com menores N, o impacto dos *reduces* entre os vários processos é maior.

*5) **Análise de implementação híbrida openMP e MPI**:* Para analisarmos a *performance* desta implementação híbrida, decidimos manter o tamanho do problema, e fazer variar o número de *threads* (análise de escabilidade forte), para um mesmo número de processos. Assim, conseguiremos analisar o impacto de usar *threads* com processos.

Uma caraterística que decidimos analisar foi a possibilidade de cada *thread* conseguir armazenar toda a informação do problema que vai tratar na memória que lhe é dedicada, nomeadamente L1 e L2, minimizando os acessos à memória. A melhor forma de estimar isso é dividindo o tamanho do problema pelo número de processos e *threads*, e comparar com os tamanho de de L1 e L2. No entanto, e após analisar os resultados, concluímos que existem outros fatores mais relevantes que decidimos estudar. Assim, apresentámos os resultados obtidos:



Pela análise do gráfico, concluímos que, com a nossa implementação, não existem vantagens na combinação de *openMP* e MPI, podendo até prejudicar o tempo de execução. Logo, cada processo não consegue ter mais que uma *thread* a correr, e isso deve ter explicações no *hardware* onde corre o programa. Cada processo já deve estar a usar ao máximo os recursos disponíveis, e por isso não consegue ter mais

paralelismo. Para além disso, o resultado obtido com processos e *openMP* é pior que sem *openMP*.

V. Trabalho futuro

Por concluir, ficamos satisfeitos com os resultados obtidos, e achamos que, para melhorar o trabalho realizado, podia ter sido implementado o seguinte:

- Melhorar a implementação híbrida com *openMP* e MPI, para que fosse utilizado mais paralelismo dentro de cada processo, visto que as comunicações dentro do mesmo processo são mais rápidas que entre processos;
- Diminuir o custo da função de inicialização, visto que cada processo só necessita de $\frac{N}{P}$ amostras, e com isto diminuíamos o número de instruções realizadas sequencialmente em todos os processos, e consequentemente o tempo de execução global. No entanto, a lógica de inicialização das variáveis teria de ser alterada;
- Analisar onde o nosso programa demora mais a executar, com a implementação em MPI;
- Analisar e compreender como o MPI influencia o número de ciclos, e o CPI, para estudar o impacto do paralelismo no tempo de execução.

VI. Anexos

	Amostras (N)			
	1000		10000	
	Tempo	Ganho	Tempo	Ganho
1 Processo	0.139 s	—	0.276 s	—
2 Processos	0.127 s	1.094	0.211 s	1.308
4 Processos	0.136 s	1.022	0.174 s	1.586
8 Processos	0.115 s	1.209	0.165 s	1.673
16 Processos	0.169 s	0.822	0.221 s	1.249
20 Processos	0.234 s	0.594	0.239 s	1.155
32 Processos	2.852 s	0.049	1.440 s	0.192
40 Processos	0.406 s	0.342	0.357 s	0.773

TABLE I
Tabela que relaciona o número de processos com os tempos de execução, e respetivos ganhos, para 1,000 e 10,000 amostras : Na versão sequencial

	Amostras (N)			
	1000000		10000000	
	Tempo	Ganho	Tempo	Ganho
1 Processo	23.472 s	—	—	—
2 Processos	11.053 s	2.124	—	—
4 Processos	5.662 s	4.146	—	—
8 Processos	2.903 s	8.085	40.087 s	—
16 Processos	1.578 s	14.875	19.073 s	2.102
20 Processos	1.400 s	16.766	17.057 s	2.350
32 Processos	2.277 s	10.308	19.361 s	2.071
40 Processos	1.335 s	17.582	14.207 s	2.822

TABLE II
Tabela que relaciona o número de processos com os tempos de execução, e respetivos ganhos, para 1,000,000 e 10,000,000 amostras : Na versão sequencial

	Processos (P)			
	10		20	
	Tempo	Ganho	Tempo	Ganho
1 *Thread*	2.295 s	—	1.370 s	—
2 *Threads*	2.332 s	0.984	1.398 s	0.980
4 *Threads*	3.700 s	0.620	1.359 s	1.008
8 *Threads*	2.524 s	0.909	1.478 s	0.927
12 *Threads*	2.325 s	0.987	1.379 s	0.993
16 *Threads*	2.340 s	0.981	1.367 s	1.002
20 *Threads*	2.651 s	0.866	1.342 s	1.021
24 *Threads*	2.630 s	0.873	1.434 s	0.955
28 *Threads*	3.001 s	0.765	1.389 s	0.986
32 *Threads*	2.637 s	0.870	1.356 s	1.010
36 *Threads*	3.130 s	0.733	1.351 s	1.014
40 *Threads*	2.355 s	0.975	1.397 s	0.981

TABLE III
Tabela que relaciona o número de processos com os tempos de execução, e respetivos ganhos, para 1,000,000 amostras: Na versão paralela (com várias threads)

References

[1] : *MPI Reduce and Allreduce*
https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/
[2] : *MPI Broadcast and Collective Communication*
https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/
[3] : *Cost of some MPI routines*
https://who.rocq.inria.fr/Laura.Grigori/TeachingDocs/UPMC_Master2/Slides_HPC_MN_DA/costMPIRoutines.pdf