Trabalho Prático 2

Computação paralela

1st Francisco Reis Izquierdo
Universidade do Minho
Mestrado em Engenharia Informática
Aveiro, Portugal
pg50384@alunos.uminho.pt

1st Francisco Alberto do Fundo Novo
Universidade do Minho
Mestrado em Engenharia Informática
Braga, Portugal
pg50374@alunos.uminho.pt

***Abstract*—O presente relatório tem como objetivo ilustrar todo o trabalho feito pelo grupo no qual, ao longo do mesmo, estarão detalhadas toda a implementação bem como técnicas de paralelismo em programas e correspondente justificação.**

***Index Terms*—OpenMP, paralelismo, tempos de execução, resultados.**

I. Introdução

No âmbito da unidade curricular de Computação Paralela foi desenvolvido o segundo trabalho prático no qual são abordados conceitos de paralelismo de programas em que o objetivo principal é o aumento do desempenho do algoritmo implementado no primeiro trabalho prático através da exploração da interface **OpenMP**.

II. Implementação

A implementação foi dividida em várias componentes que juntamente compactuam na medida de atingir o objetivo supramencionado. Assim, de forma sequencial, foram abordados os seguintes tópicos:

- Identificação de regiões computacionalmente exigentes;
- Programação paralela;
- Regiões críticas;
- Balanceamento de carga;
- Medição de perfis de execução;
- Retrospeção dos perfis de execução;
- Análise de escalabilidade;

Além disto, primeiramente foram feitas retificações ao código desenvolvido, nomeadamente a eliminação das estruturas de dados e *unroll looping* para o uso de *arrays* dinâmicos, na medida em que esta estratégia era computacionalmente mais eficiente.

III. Desenvolvimento

A. *Identificação de regiões computacionalmente exigentes*

Através de uma breve análise, foi fácil identificar dois blocos de código que tinham maior impacto no programa desenvolvido:

- Bloco de código responsável pela geração de todos os pontos 2D da amostra;
- Bloco de código responsável por, para cada ponto, calcular as distâncias aos centroides de cada *cluster* e identificar qual o cluster a que o mesmo pertence;

O último bloco mencionado é mais exigente computacionalmente, sendo que foi neste segmento onde o grupo de trabalho teve maior foco de forma a atingir uma melhor performance.

B. *Programação paralela*

De modo a combater o que fora supramencionado, recorremos ao uso de paralelismo através da interface **Openmp**, apenas no último bloco de código identificado. Com isto, foram exploradas as seguintes primitivas da interface no qual algumas se revelaram ineficazes dadas o objetivo proposto:

- ***pragma omp parallel num_threads(T) for***: Esta primitiva, permitiu o uso de um número específico *T* de fios de execução, no qual a mesma mostrou ser bastante eficaz, uma vez que foi atingido um melhor desempenho principalmente ao nível de tempo de execução. No entanto, apenas se mostrou ser eficaz nos segmentos responsáveis por lidar com o espaço **N** da amostra e não com o número **K** de *clusters*;
- ***Loop Scheduling***: Foram exploradas diversas primitivas tais como agendamento **estático**, **dinâmico** e **guiado** de forma a haver uma distribuição uniforme da carga computacional entre os diversos fios de execução. No entanto, mostrou ser ineficiente uma vez que qualquer fio realiza, para cada ponto, a mesma carga, estando já implementada uma distribuição uniforme;

Nota: Para o bloco responsável pela geração dos pontos 2D da amostra, não foi aplicado o paralelismo, uma vez que a função que gera os números aleatórios (*rand()*) não é considerada segura para vários fios de execução, devido à semente usada pela função, não poder ser partilhada entre os fios, algo que fora testado e comprovado, originando diferentes valores para as componentes *x* e *y* dos pontos da amostra sempre que se realizavam execuções do programa.

C. *Regiões críticas*

A única região crítica identificada diz respeito ao segmento responsável por atualizar/escrever a informação do cluster, nomeadamente o somatório da componente do *x* e *y* de todos os pontos do *cluster* e o número de elementos, por forma no

fim de uma iteração recalcular o novo centroide. Deste modo, foram exploradas diversas primitivas por forma a assegurar a consistência do resultado esperado, no qual as mesmas se revelaram ineficazes:

- ***pragma omp critical***: Esta primitiva permitiu um acesso concorrente controlado, no qual era assegurado o correto resultado. No entanto, dado o objetivo proposto, não era temporalmente eficiente, uma vez que os vários fios de execução aguardavam sequencialmente para entrar na secção crítica.
- ***pragam omp single***: Esta primitiva permitiu que a carga computacional da região fosse efetuada por um dos fios de execução disponíveis, garantindo o correto resultado. No entanto, esta primitiva em termos temporais mostrou ser ineficiente, uma vez que os restantes fios teriam de aguardar pelo mesmo.
- ***pragam omp master***: Esta primitiva permitiu que a carga computacional fosse efetuada pelo fio de execução principal, mas compactuou com a primitiva anterior;

Solução: A região crítica foi deslocada para o exterior do ambiente paralelo, por forma a deixar de ser região crítica. Deste modo, o fio de execução principal ficou responsável por toda a carga computacional, sem que os restantes fios ficassem à espera, dado que já não há paralelismo, no qual o mesmo é responsável por, para cada *cluster* atualizar toda a informação acima descrita.

*D. **Balanceamento de carga***

No que concerne ao balanceamento de carga, no bloco de código onde foram realizadas as diversas etapas, a carga é uniformemente distribuida pelos vários fios de execução, no entanto toda a carga da região crítica indicada é realizada pelo fio de execução principal, bem como outros processos dedicados ás informações de cada *cluster*. Desta forma, na secção computacionalmente mais exigente do programa, existe um balanceamento da carga entre os diversos fios de execução, que permite atingir uma melhor performance.

*E. **Previsão do perfil de execução***

Uma vez resolvida a questão referente ao uso de paralelismo e regiões críticas, a seguinte etapa passou por perceber como tirar o máximo proveito do uso de paralelismo. Com isto, foram feitas previsões do que era expectável ocorrer:

- A versão paralela trará melhores resultados nas diferentes métricas em relação à versão sequencial;
- Irá haver uma diminuição gradual nos tempos de execução para os vários perfins de execução das versões paralelas com o aumento do número de fios de execução;
- Haverá um número de fios de execução ideal, isto é, aumentar o número de fios de execução tem um limite;

A seguinte tabela, procura mostrar o impacto mencionado tendo em vista diversas métricas indicadas, podendo por isso, comparar-se o impacto de diferentes números de fios de execução.

*F. **Medição de perfis de execução***

Versão	Threads	Clusters	CPI	#I (mil milhões)	Ciclos (mil milhões)	Tempo (seg)
Sequencial	1	4	0,4	17,321	6,6	2,235
Sequencial	1	32	0,5	87,926	41,318	13,494
Paralelo	4	4	0,4	17,580	7,396	1,216
Paralelo	4	32	0,5	88,155	41,260	5,404
Paralelo	8	4	0,4	17,835	7,856	1,197
Paralelo	8	32	0,5	88,389	41,864	3,208
Paralelo	12	4	0,5	18,129	8,701	0,890
Paralelo	12	32	0,5	89,668	42,557	1,541
Paralelo	16	4	0,5	18,388	9,262	0.987
Paralelo	16	32	0,5	88,969	43,265	2,174
Paralelo	32	4	0,9	19,550	17,537	1,003
Paralelo	32	32	0,9	90,110	73,884	2,761

*G. **Retrospeção dos perfis de execução***

Tendo em vista a tabela supramencionada, percebemos que:

- A versão paralela do algoritmo implementado é mais eficiente comparativamente com a versão sequencial, no que concerne à métrica do tempo de execução;
- Ao aumentar o número de fios de execução até 12, há uma melhoria progressiva no tempo de execução enquanto que o número de ciclos não aumenta significativamente;
- Ao continuarmos a aumentar o número de fios de execução, começamos a ter um declínio no tempo de execução, como por exemplo com 16 fios e até mesmo uma sobrecarga relativamente ao número de ciclos gerados, algo notório com 32 fios;

Concluimos pela retrospeção dos perfis de execução, que um aumento do número de fios de execução tem um limite, sendo que o melhor número encontrado foi **12** fios de execução.

*H. **Escalabilidade***

A versão paralela comparativamente à versão sequencial é mais escalável do ponto de vista de performance, algo evidenciado no tópico de perfis de execução. Duma prespetiva de análise de escalabilidade, percebe-se portanto que com o aumento do número de *clusters* haverá um aumento da exigência computacional, nomeadamente no que diz respeito ao tempo de execução. Relativamente à versão sequencial, teremos um aumento do tempo de execução linear na ordem de $N * K$ enquanto que na versão paralela teremos um aumento do tempo de execução também linear, no entanto menor, na ordem de $(N * K)/T$, com **N** o tamanho da amostra, K o número de *clusters* e **T** o número de fios de execução.

Nota: Existem outros blocos de código que para um número de *clusters* bastante elevado podem tornar-se computacionalmente mais exigentes, como o segmento responsável por recalcular o novo centroide para cada *cluster* e/ou "limpar" a informação de cada *cluster* da iteração anterior. De forma a tornar a solução escalável, poderíamos alargar o conceito de paralelismo a estes blocos, porém a solução desenvolvida teve em vista 4 e 32 *clusters*.

IV. Conclusão

A realização deste projeto permitiu consolidar noções acerca de correção e eficiência de programas paralelos com o auxílio da interface **OpenMP**, tornando claro que as escolhas tomadas na implementação determinaram o desempenho do programa.