Trabalho Prático 1

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 com o intuito de realizar o trabalho prático proposto pela equipa docente. Ao longo do mesmo, estarão detalhadas toda a implementação bem como técnicas de otimização de código e correspondente justificação.**

***Index Terms*—Otimizações, estruturas de dados, análise, *kmeans*, resultados.**

I. Introdução

No âmbito da Unidade Curricular de Computação Paralela foi desenvolvido o primeiro trabalho prático proposto pelos docentes. Neste projeto são abordados conceitos acerca de técnicas de otimização de código, nomeadamente, foi desenvolvida em C uma versão sequencial otimizada de um algoritmo k-means simples, baseado no algoritmo de Lloyd. O foco passa por minimizar o tempo total de execução do algoritmo, fazendo as alterações necessárias ao algoritmo fornecido pelos docentes, mas mantendo a sua filosofia.

II. Desenvolvimento

A. *Descrição algoritmo kmeans (Lloyd)*

1) Criar as amostras e iniciar os clusters.
   - Iniciar um vetor com valores aleatórios (N amostras no espaço (x,y)
   - Iniciar os K clusters com as coordenadas das primeiras K amostras.
   - Atribuir cada amostra ao cluster mais próximo usando a distância euclidiana.
2) Calcular o centroide de cada cluster (centro geométrico).
3) Atribuir cada amostra ao cluster mais próximo usando a distância euclidiana
4) Repetir os passos 2 e 3 até não existirem pontos que mudem de cluster.

B. *Implementação*

Para a implementação do algoritmo supramencionado foi necessário criar duas estruturas de dados.

- **Ponto**: Esta estrutura de dados alberga as coordenadas de duas dimensões que cada ponto irá ter. Assim, teremos 2 variáveis do tipo *float*, que indicam o valor de *x* e *y*. Além disso, tem também uma variável do tipo *inteiro* que indica qual o *cluster* a que o ponto pertence.
- ***Cluster***: Esta estrutura de dados é responsável por conter toda a informação de cada *cluster*, nomeadamente as coordenadas do centroide, que é do tipo *Ponto*, bem como o somatório das coordenadas *x* e *y* dos vários pontos que são associados ao mesmo ao longo de cada iteração, sendo guardados em variáveis do tipo *float*. Contém também o número total de pontos pertencentes ao *cluster*.

É de salientar que estas estruturas de dados foram contidas em *arrays* dinamicamente alocados, no qual existe um *array* responsável por guardar todos os pontos da amostra de dados e outro por guardar os diversos *clusters*.

Tal como é exigido pelo enunciado a primeira etapa é definida pela inicialização dos pontos e dos clusters de forma aleatória, representada pela função $initialize$. Com os centroides e as amostras definidas em cada cluster, a cada nova iteração é calculado a métrica da distância euclidiana para cada ponto para perceber a qual cluster este pertence e caso seja notificado um cluster diferente ao qual o ponto estava associado anteriormente é dada a indicação de que o algoritmo ainda não convergiu. Assim, no final de percorrer toda a amostra e caso o algoritmo não tenha convergido são recalculados os centroides respetivos a cada cluster, na função $recalculateCentroids$, com base na informação contida na estrutuda de dados já mencionada. No caso em que o algoritmo converge, isto é, não há mais nenhum ponto que altere de cluster é ativada uma flag definindo a última iteração do programa e é feita a impressão do resultado obtido, onde se destaca qual o centroide de cada cluster, o número de elementos destes e o número de iterações efetuadas. A função $kmeans$ trata de todo este processo.

C. *Análise*

Por forma a obter melhores resultados, o grupo de trabalho realizou diversas análises ao programa que estava a ser desenvolvido, como a leitura do código em *assembly* e/ou as métricas computacionais relativas ao código desenvolvido obtidas através da ferramenta *perf*. Exemplos destas métricas são:

- CPI.
- Número de instruções.
- Número de ciclos.

- Número de *misses* ao nível 1 de cache.
- Tempo de execução.

Com isto, conseguimos redirecionar os esforços no âmbito de procurar a melhor solução e basear os mesmos nos conceitos abordados nas aulas práticas da disciplina. Assim, conseguimos primeiramente implementar e extender a solução a conceitos relativos a localidade espacial e temporal, no qual foi possível devido a análise do código *assembly* da solução que fora desenvolvida. Posto isto, reparámos também através da vertente mencionada, que o uso de funções matemáticas implementadas em bibliotecas importadas eram computacionalmente ”pesadas” e que este era um fator a ter em conta. Já com a ferramenta *perf* conseguimos identificar que havia um número elevado de *misses* ao nível 1 de cache e que este era também um aspeto a ter em conta na otimização da solução. Por fim, decidimos extender a análise da solução implementada relativamente ao uso de *flags* que poderiam ou não ter um impacto positivo na mesma.

D. *Otimização*

Após efetuadas as várias análises às soluções o grupo realizou diversas otimizações com o intuito de melhorar a performance do programa no qual podemos separar em duas vertentes, nomeadamente no âmbito das *flags* que auxiliam o compilador e o código desenvolvido, estando intrinsicamente ligadas uma à outra.

- **Reestruturação de estruturas de dados**: a existências de ciclos aninhados produziam um maior número de instruções desnecessárias devido à ausência total de localidade temporal, uma vez que a estrutura de dados ***Cluster*** inicialmente não conter informação necessária para tal, isto é, o somatório das coordenadas x e y dos vários pontos que são associados ao mesmo ao longo de cada iteração.
- **Eliminação de funções matemáticas**: O uso das funções *pow* e *sqrt* pré-definidas de bibliotecas importadas, no cálculo da distância euclidiana, aumentava significativamente o número de instruções. A solução passou pela eliminação de ambas, sendo realizada a função *pow* manualmente.
- **Retificação do código**: Uma otimização importante foi a retificação do código, nomeadamente a eliminação de instruções desnecessárias e rearranjo de instruções computacionalmente mais eficientes.
- ***Loop unrolling* manual**: Uma das otimizações mais eficientes foi a implementação de *loop unrolling* manualmente, no qual após vários testes, percebemos que ao ”desenrolar” cada iteração do ciclo 4 vezes, era extremamente eficiente. Apesar de produzir-se código repetitivo, os efeitos trazidos eram demasiados significativos para não ser implementado.
- **Uso de *flags***: O uso de *flags* para auxílio do compilador foram extremamente úteis, mas apenas em algumas circusntâncias quando compactuadas com um código apropriado às mesmas. Assim, após as otimizações acima mencionadas, foram usadas as *flags*:
  - *-O2*: que aumenta o tempo de compilação e o desempenho do código gerado,
  - *-funroll-loops*: que, jutamente com o código desenvolvido, ”desenrola” ciclos cujo número de iterações pode ser determinado em tempo de compilação ou na entrada no ciclo.
  - *-ftree-vectorize -msse4*: que realiza a vetorização do código, sendo que a segunda *flag* gera instruções de vetor SSE4 que suporta operações vetoriais de 128 bits.

E. *Resultados*

Com o objetivo de obter a melhor interpretação dos resultados, o projeto foi testado usando o cluster computacional SeARCH disponibilizado pelos docentes com diferentes níveis de otimização a fim de analisar para cada caso os tempos de execução. Para uma melhor compreensão da tabela abaixo explicamos que em cada nova linha no sentido descendente é aplicada a otimização da linha anterior juntamente com a da própria linha, que irá dar uma ideia do impacto que cada nova otimização tem nas diferentes métricas computacionais do programa.

Versões	Tempo	CPI	I(inst_retired.any)	L1_DMiss	Ciclos
-O0	1m52.296s	0,6	294554361424	89243321	176540725694
-O2	0m36.665s	1,1	44470841545	82134811	47256245591
Sem pow e sqrt	0m12.333s	0,8	28034332666	80478089	23629151617
Loop Unrolling (flag e manual)	0m6.833s	0,9	22834732747	80254664	19806876424
Vetorização (flag)	0m3.842s	0,5	23579133039	79627402	11095681967

III. Conclusão

A realização deste projeto permitiu consolidar noções acerca de otimização de desempenho, hierarquia de memória, entre outros. É de sublinhar que os conceitos de localidade temporal bem como localidade espacial foram bastante decisivos na resolução deste projeto. Além disso, percebemos que com a análise de código assembly conseguimos manipular de uma melhor forma qual a ordem é que o código deve ser executado obtendo um melhor desempenho do programa. Realça-se que este trabalho permitiu entender as diferenças dos vários níveis de otimização e perceber que dependendo do código desenvolvido, quais as otimizações é que apresentam melhores resultados em relação a outras. Destacam-se destas otimizações o *loop unrolling* e a vetorização.

Por fim, como todos os pontos foram abordados de acordo com o propósito da unidade curricular, concluímos que podemos retirar um balanço positivo deste projeto já que foi essencial para o nosso conhecimento pois adquirimos boas bases para trabalhos futuros.

References

[1] Referenciar