Relatório do 1º Projeto da Disciplina de Concorrência e Paralelismo

Gonçalo Banha (45429) & Miguel Almeida (45526)

14 de Outubro de 2017

Resumo

O problema inerente ao trabalho prático da cadeira de *Concorrência e Paralelismo* consiste na aprendizagem e respetiva implementação de como melhorar um programa funcional do jogo Reversi, na linguagem C. A solução passa por paralelizar o programa sequencial dado, utilizando a biblioteca Cilkplus. É também objetivo do trabalho avaliar a qualidade da solução paralelizada e o impacto da mesma no funcionamento do programa. Como resultado da nossa implementação conseguimos que os recursos da máquina fossem utilizados de uma maneira mais eficiente melhorando também os tempos de execução do problema.

1 Introdução

Este relatório foi elaborado pelo grupo nº16 para o primeiro projeto prático da unidade curricular de *Concorrência e Paralelismo.*

Neste projeto foi-nos pedido que paralelizássemos um programa funcional em C, programa este referente ao jogo *Reversi*, que é um jogo de estratégia, jogado por duas pessoas num tabuleiro. A paralelização utiliza o modelo *fork-join* implementado pela biblioteca *Cilkplus* de modo a reduzir o tempo de execução do algoritmo. Um dos objetivos deste trabalho recai em perceber onde e de que maneira deve ser feita a paralelização das funções do jogo, de modo a ter resultados claros das melhorias da paralelização do programa. Este também é um dos aspetos mais interessante e ao mesmo tempo desafiantes do projeto em si, uma vez que nos obriga a pensar e a perceber quais as melhores decisões a tomar. A nossa abordagem para resolver o problema foi dividir o problema em sub-problemas menores de maneira que o resultado não fosse alterado.

2 Abordagem

De modo a melhorar os tempos de execução do programa decidimos implementar algumas melhorias no que toca ao método *make_move*, que é responsável por analisar todas as jogadas possíveis, calculando a sua heurística, e perceber qual a melhor opção a ser tomada de acordo com a mesma. Por ultimo, este método também tem a função de realizar a jogada se assim for possível.

Analisámos o código fornecido e reparámos que o calculo das heurísticas para cada jogada não depende de nenhum cálculo prévio (não possui dependências), como tal, este cálculo pode ser feito de forma independente. Graças a este facto é possível um desdobramento do trabalho

necessário para computar o tabuleiro de jogo de modo a melhorar a *performance* do programa. Desta forma, em vez de computar o tabuleiro de jogo sequencialmente, decidimos dividir o tabuleiro de jogo em linhas, de modo a que exista uma distribuição equilibrada de trabalho, para que o sistema possa tratar cada uma destas linhas em paralelo. Esta divisão está representada na figura 1.

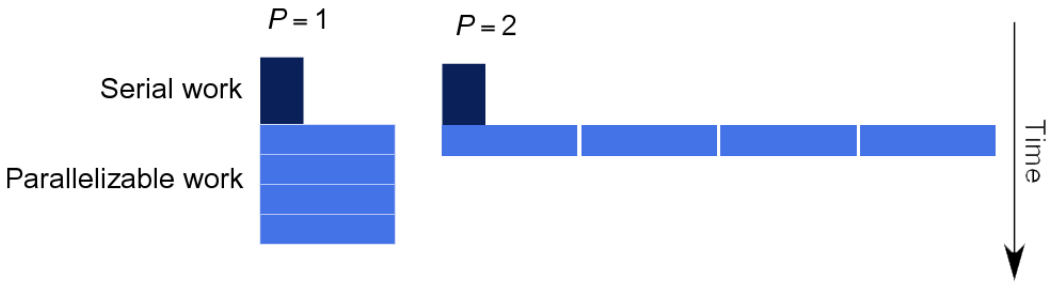

Figura 1: Diferença do tempo de computação entre a versão original e a versão paralelizada

Após percebermos onde a maioria do tempo de execução estava a ser despendida, concluímos que paralelizar a inicialização do tabuleiro de jogo e a libertação de memoria do mesmo não era algo necessário, uma vez que o tempo de execução das duas operações era insignificante comparado com o tempo de execução do resto do programa.

No que toca ao padrão de paralelização empregue neste trabalho prático utilizámos o padrão de *Divide-and-Conquer* uma vez que é um padrão que deve ser aplicado se um problema maior puder ser dividido em sub-problemas menores. Neste problema, e como já referido anteriormente, decidimos dividir o trabalho necessário para computar o tabuleiro de jogo recorrendo à divisão do mesmo por linhas. Desta forma e atendendo às características do problema achámos que este seria a melhor solução para paralelizar o algoritmo.

Os problemas que encontrámos ao longo da implementação da solução deviam-se à forma como eram inicializadas as variáveis do método, tais como: os inteiros responsáveis por percorrerem os ciclos e a variável responsável por assinalar o movimento da jogada em causa, estarem a ser partilhadas por diversas *threads*, o que cria conflitos, uma vez que diversas *threads* em diferentes pontos de execução tentavam aceder ao mesmo valor da variável. Como solução para este problema, criámos um *array* de movimentos, uma vez que deixámos de ter um movimento genérico para o método e passamos a ter um movimento para cada *thread*. A criação deste *array* prende-se com a necessidade de guardar os diferentes movimentos obtidos nas várias linhas, para que no final seja possível analisar todos os movimentos contidos no *array* e selecionar o melhor movimento do mesmo, de forma a garantir que em cada execução do método, é escolhido o melhor movimento possível.

Para resolver o problema da inicialização do inteiros que são utilizados na contagem dos ciclos, resolvemos passar a inicialização destes mesmos inteiros para o corpo do ciclo for, deste modo, cada *thread* possui os seus próprios inteiros e não existem acessos indevidos ou conflitos de acessos por partes das mesmas.

3 Validação

Ao longo do desenvolvimento da nossa solução foram efetuados vários testes, de forma a perceber de que formas estávamos a progredir. Numa primeira fase estes testes eram feitos numa máquina pessoal (quad-core) de forma a perceber se os tempos de execução estavam ou não a ser diminuídos, sendo que numa segunda fase de testes passamos a utilizar uma máquina (8 * dual-core) disponibilizada pelo docente da cadeira.

Foram realizados testes de diferentes perfis. Certos testes tinham como objetivo perceber se as alterações que fazíamos ao nosso projeto estavam, de algum modo, a alterar o *output* do programa e outros testes serviam para perceber até que ponto o programa estava a ser otimizado. Para isto, corremos vários testes em que, para diferentes tamanhos de tabuleiros testávamos a versão original e a versão alterada por nós, para que deste modo fosse possível ver passo a passo o *output* do nosso programa e assim conseguíssemos assegurar que este continuava correto e não havia jogadas inválidas.

De modo a perceber até que ponto o nosso programa estava a melhorar do ponto de vista temporal, efetuámos diversos testes, tendo por base os tempos do código original. Efetuámos por isso testes com diferentes tamanhos de tabuleiro de jogo e diferentes números de *threads*, para que pudéssemos assim abranger o maior número de casos possível.

4 Avaliação

Do ponto de vista da avaliação de resultados obtidos foi necessário efetuar diversos testes de *performance* e de correção. Os testes de *performance* devem-se ao facto de necessitarmos de ter a noção do impacto que a utilização de diferentes números de *threads* e tamanhos de tabuleiros têm no programa. Quanto aos testes de correção, estes são necessários para podermos garantir que não existem erros de execução no que toca ao processamento do programa.

Relativamente aos testes de correção, foram executados vários testes com vários tamanhos de tabuleiros de jogo, tanto na versão original do código, como na versão paralelizada, de maneira a perceber se em algum momento da execução do programa as duas versões do jogo produzem *outputs* diferentes. Estes testes foram também executados com um número diferentes de *threads*, garantido assim que mesmo que o número útil de *threads* se altere, o resultado do problema continue correto e inalterado.

No que toca aos testes de *performance* estes foram executados numa máquina, disponibilizada pelos docentes da unidade curricular, com as seguintes características: Model: Sun Fire X4600 M2 x64 server, Operating System: Ubuntu 16.04.3 LTS, 8 x AMD Opteron Model 8220 processor (2.8GHz-dual-core), Modular processor board with 8-DIMM slots 16x2GB DDR2-667 memory (32GB total)

Os testes feitos foram realizados de modo a averiguar diversos aspetos. De modo a ter uma ideia mais clara de como o fator *thread* e o tamanho do tabuleiro influenciam o resultado final, fizemos vários testes para cada tipo de tabuleiro. Para um dado tamanho de tabuleiro, que varia entre 8, 25, 50, 100, 200, executámos vários testes, alterando o número de *threads* utilizadas para cada um deles, variando as mesmas entre: 1,2,4,8,16,32. Para ter um resultado mais equilibrado efetuamos também estes mesmos testes diversas vezes, de forma a conseguir excluir se algum resultado estava fora do normal e calcular a média de resultados para cada teste.

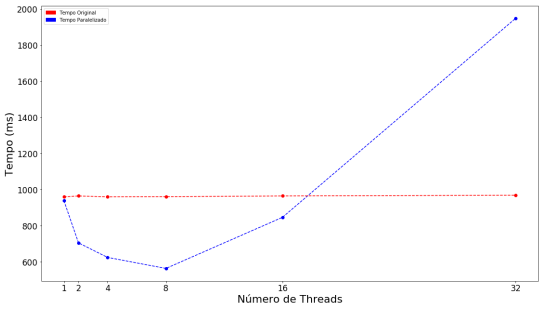

Figura 2: Teste de performance com tabuleiro de 50x50

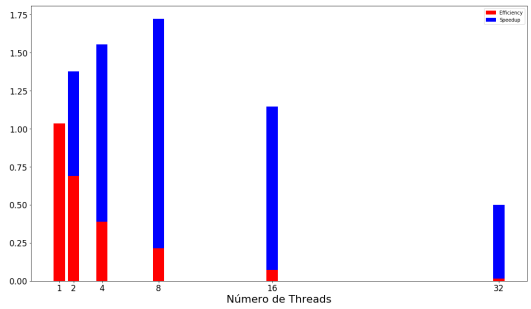

Figura 3: Valores de *Speedup* e *Effiency* com tabuleiro de 50x50

Os valores representados graficamente ao longo do relatório foram traçados recorrendo ao valor médio de cada execução. Todos os gráficos referentes aos testes de *performance* executados têm dois elementos, a linha vermelha, referente ao teste com o programa original e a linha azul, referente ao teste com a versão paralelizada. Verifica-se que na versão original os tempos de execução não variam com o aumento do número de *threads*. Isto deve-se ao facto do programa original não estar paralelizado e por isso mesmo não fazer uso das diferentes *threads* disponíveis. É por isso claro, que tanto no programa original como no programa paralelizado, quando o número de *threads* é igual a 1, o tempo de execução seja o mesmo, porque um programa paralelizado com apenas umas *thread* disponível, é na verdade um programa sequencial. Foram também traçados gráficos de barras que mostram o valor do *speedup* e *effiency* para os diferentes tamanhos de tabuleiros e para os testes com diferentes números de *threads*.

Como podemos ver no quadro da figura 2, o tempo de execução diminui drasticamente, até certo ponto, assim que o número de *threads* utilizadas aumenta de 1 para valores superiores, o que mostra os benefícios da utilização de *threads* na paralelização de programas, contudo, como é visível na figura, a partir das 8 *threads* o tempo de execução do programa volta a aumentar. Isto deve-se ao facto de o problema em causa não ter uma dimensão tão significativa que justifique a utilização de mais do que 8 *threads* em simultâneo, uma vez que cada *thread* tem inerente a si mesma um tempo de criação que não pode ser ignorado nem reduzido. Desta forma, se para um problema relativamente pequeno for criado um número elevado de *threads*, grande parte do tempo é passado a criar e inicializar *threads* e não a resolver o problema em causa. O gráfico acima prova este facto.

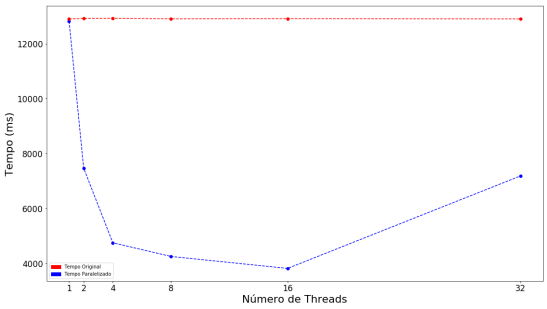

Figura 4: Teste de performance com tabuleiro de 100x100

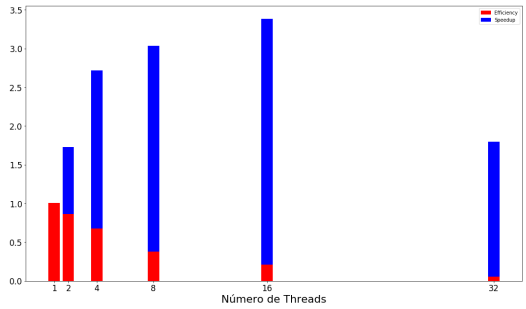

Figura 5: Valores de *Speedup* e *Effiency* com tabuleiro de 100x100

No que toca ao gráfico da figura 4, este tem um comportamento bastante similar ao da figura 2, contudo, e devido ao facto do problema ser de uma ordem maior, o número de *threads* que minimiza o tempo necessário para executar o jogo é atingido com 16 *threads*, em vez das 8. Após a analise deste gráfico é possível inferir, à semelhança da figura 2, que existe um ponto crítico do gráfico em que o aumento do número de *threads* não provoca nenhum beneficio no que toca aos tempos de execução, antes pelo contrário. Como no caso anterior, o problema em questão não tem uma grandeza suficiente que justifique a inicialização de mais *threads* para o resolver. Desta forma, com a utilização de 32 *threads* o tempo de execução volta a aumentar.

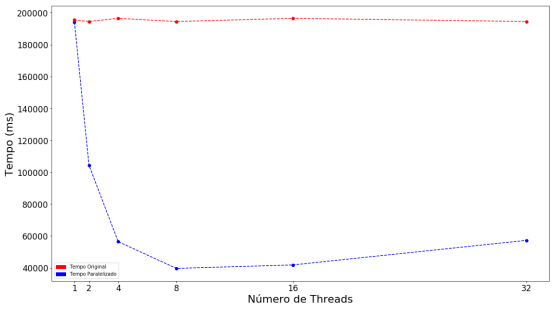

Figura 6: Teste de performance com tabuleiro de 200x200

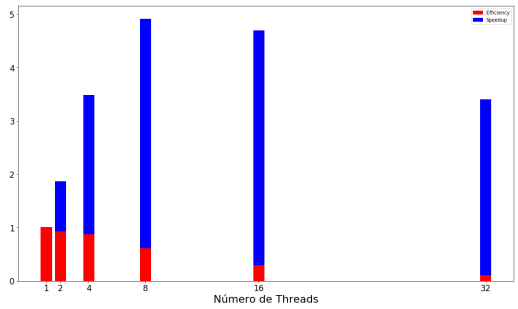

Figura 7: Valores de *Speedup* e *Effiency* com tabuleiro de 200x200

Para o teste com um tabuleiro de 200 por 200, representado na figura 6, existe uma menor diferença de tempos de execução quando se corre o programa com 16 e 32 *threads* uma vez que neste caso o problema em causa tem um complexidade bastante superior, o que faz com a utilização de 32 *threads* seja cada vez menos descabida. Como é visível na figura 6, o tempo de execução com 8 *threads* ficou um pouco abaixo do tempo de execução com 16. Isto deve-se ao facto da máquina utilizada para correr os testes ser uma máquina partilhada com os outros estudantes da cadeira, por isso, ao correr testes quando existem outros grupos a correr os seus testes, resulta em desvios ligeiros nos resultados. Isto, acrescido ao facto de quando se executam testes com 16 *threads* todos os *cores* da máquina estão ocupados, não havendo por isso margem para que mais instruções sejam executadas pela máquina, o que faz com que, se existir algum acesso à máquina por parte de outros grupos os resultados dos testes terão resultados flutuantes.

5 Conclusões

Inicialmente pensávamos que quantas mais *threads* um programa paralelizado tivesse para utilizar, mais rápido seria. Contudo, viemos a descobrir que isto nem sempre é verdade. Como já referido, se um programa tiver uma complexidade pequena, não compensa inicializar um número elevado de *threads*.

Os testes que fizemos vieram validar a nossa teoria, uma vez que é possível ver, que para qualquer um dos testes efetuados, havia sempre um ponto crítico em que o aumento do número de *threads* vinha a piorar o tempo de execução do programa. Este ponto crítico surge num número superior de *threads* à medida que a complexidade do problema aumenta.

6 Bibliografia

McCool M., Arch M., Reinders J., Structured Parallel Programming, Patterns for Efficient Computation, Morgan Kaufmann (2012);