Abstract

O problema proposto foi o de paralelizar o jogo Othello, caso isso trouxesse benefícios à performance. A solução foi paralelizar o ciclo que percorre o tabuleiro de forma a encontrar a melhor jogada possível. Os resultado obtidos permitem concluir que apenas a partir de uma determinada dimensão é que há melhorias de performance significativas ao paralelizar, sendo que ainda assim, em contextos reais, tem que se ter em conta a eficiência da solução, ou seja, tem que haver sempre um jogo de *trade-offs* quando se está a decidir se a paralelização é vantajosa, e se o é, quantos processadores atribuir-se-á.

Introdução

Este primeiro projecto da cadeira de Concorrência e Paralelismo consiste na aplicação de padrões de paralelização de modo a melhorar a performance do jogo **Othello** na sua versão não paralelizada. Para tal, é necessário primeiramente analisar o código, em **C**, que está em série de modo a compreender-se quais os possíveis padrões de paralelização a implementar com o auxílio da biblioteca da Intel **Cilk Plus**. Após a implementação destes padrões é necessário analisar a performance do jogo e concluir se, efectivamente, houveram melhorias.

Os principais desafios que se colocam é saber **quais** os padrões de paralelização que se irão aplicar e **onde** serão implementados.

De forma sucinta, para resolver os desafios descritos, anteriormente, foi necessário dividir o problema do othello em sub-problemas e analisá-los individualmente de modo a concluir qual deles tinha um peso maior na performance do jogo. A descrição detalhada da análise do problema está descrita mais à frente.

Abordagem

A solução encontrada reparte o trabalho entre **n** threads ao dividir o tabuleiro de jogo em **n** conjuntos de linhas, isto é, cada thread terá um intervalo de linhas em que calculará a melhor jogada possível para cada linha desse intervalo. Após todas as threads terminarem a sua execução ter-se-á as melhores jogadas de cada linha do tabuleiro. Assim, bastará calcular o máximo entre estas e será obtida a melhor jogada possível tendo em conta todo o tabuleiro.

O padrão de paralelização utilizado para resolver o problema foi o ***map***, que reduz o tempo de execução de um loop mas apenas considerando que todas as iterações do loop são independentes. O ***map*** aplica uma função a todos os elementos de uma colecção em paralelo. Mais geralmente, o ***map*** executa um conjunto de invocações de funções, cada uma das quais acede a segmentos de memória independentes [1].

Considerando o problema do jogo Othello, a colecção será o tabuleiro de jogo e a função a aplicar será calcular o ganho para cada casa do tabuleiro.

Na solução desenvolvida, não se guarda o valor de cada casa mas sim o máximo de cada linha. Este máximo não poderá ser uma variável partilhada entre todas as threads pois isso poderia levar a inconsistências e à redução de performance. Já que existiriam, potencialmente, várias threads a aceder ao mesmo tempo à mesma posição de memória. A solução para este problema é cada linha ter a sua variável com a melhor jogada correspondente, o que torna cada iteração do loop independente. No final, acha-se a melhor jogada entre as de cada linha.

André Lopes nº45617
Nelson Coquenim nº45694
Grupo 23

Validação

Para suportar a decisão de que parte do código traria melhorias de performance ao paralelizar procedeu-se à análise temporal de cada uma das funções utilizando o **gprof**. Desta forma permite verificar se há funções que ocupam uma fatia maioritária do tempo de execução que são as principais candidatas a esta paralelização.

De modo a analisar o desempenho do programa efectuou-se medições do tempo de execução do programa com diferentes valores nos seguintes parâmetros: tamanho do tabuleiro e número de workers cada um com os seguintes valores, respectivamente, {10, 25, 50, 100, 150} e {1, 2, 4, 6, 8, 10, 12, 14, 16}. De realçar que os resultados foram calculados através da média de dez medições.

Analisou-se os dados, utilizando R script para a elaboração de gráficos, sobre três métricas importantes na avaliação da performance e do paralelismo: *speedup*, eficiência e o custo. Adicionalmente elaborou-se um diagrama de barras que representa a percentagem do trabalho em série e em paralelo com o aumento do número de workers.

O ***speedup*** compara o tempo necessário para resolver o problema numa unidade de hardware (*worker*) vs P unidades. [1]

A **eficiência** é o speedup dividido pelo número de *workers*, mede o retorno do investimento de hardware para resolver um determinado problema.[1]

O **custo** reflete a soma dos tempos que cada processador demora a resolver o problema.

$$speedup = Sp = \frac{T1}{Tp} \qquad cost = P \; x \; Tp \qquad efficiency = \frac{Sp}{P}$$

T1 - tempo de execução apenas num processador
Tp - tempo de execução com P workers
P - número de workers

Discussão dos resultados

Nota: Resultados obtidos podem ter algum *noise* já que a máquina em que foi testado poderia estar a ser partilhada por outros utilizadores.

A **Fig.1** esquematiza os resultados obtidos pela ferramenta **gprof** o que permite concluir qual a função a ser paralelizada é **make_move,** mais especificamente, o loop que percorre o tabuleiro por linhas**.**

Em termos de resultados obtidos, é importante fazer a análise do gráfico da **Fig.3** em que é relacionado o Speedup (Sp) pelo número de *workers* com os diferentes tamanhos de tabuleiro. Pode-se inferir existem duas categorias de resultados: as execuções com um declive crescente e decrescente. Observa-se uma reta decrescente com tabuleiros de tamanho 10 e 25, o que permite concluir que a paralelização nestes casos não é benéfica pois o tempo da versão em série é inferior ao da em paralelo, o que resulta num Sp < 1. Por outro lado, com tabuleiros de tamanho 50, 100, 150 já é benéfica a paralelização pelos motivos contrários à análise anterior.

André Lopes nº45617
Nelson Coquenim nº45694
Grupo 23

Adicionalmente, pode-se observar que o Sp máximo para tamanhos de 50, 100, 150 é, respectivamente, ~ 1.5 (valor afetado pelo *noise*), ~2.8, ~5.8 o que significa que não existe aumentos de performance a partir destes pontos, ou seja, atribuir mais do que 6, 8, 12 *workers*, respetivamente, não resultará numa diminuição do tempo de execução.

Adicionalmente na **Fig.2** pode-se observar, num tabuleiro de 100*100 casas como o trabalho em paralelo desce percentualmente em relação ao trabalho em série não paralelizável à medida que se aumenta o número de processadores, o que permite concluir que o Sp está limitado pela porção de trabalho em série não paralelizável. Imaginando que se tem capacidade para 16 threads o Sp máximo não ultrapassará 12.5, pois pela Amdahl's Law, $S(16) \leq \frac{1}{0.02 + (1-99.8) / 160}$, como comprova a **Fig.3.**

De facto, é importante analisar os resultados duma perspectiva de eficiência pois ter-se-á que decidir qual o número de processadores que serão utilizados para resolver o problema com um tabuleiro de dada dimensão, tendo em conta se estão a ser usados de forma eficiente. Posto isto, observa-se pela **Fig.4,** que os tabuleiros com dimensões 10 e 25 a eficiência decai drasticamente aproximando-se de 0, com o aumento do número de *workers*. Isto corrobora os resultados do Sp nestes tabuleiros provando que paralelizar não trará melhorias de performance. Em relação aos tabuleiros de dimensão 50, 100 e 150, apenas o de 150 é que apresenta uma eficiência perto de um modelo linear o que significa que ao aumentar o número *workers* a eficiência decairá linearmente e não de forma abrupta.

Por último, a **Fig.5** revela que para todos os tabuleiros o custo é exponencial, ou seja, com o aumento do número de workers maior será o gasto dos recursos

Conclusões

No final do presente relatório importa retirar conclusões: os resultados obtidos foram os esperados, para problemas de menor dimensão o overhead de criação de threads acaba por causar redução da performance. Por outro lado, quanto maior a dimensão do problema maior são os ganhos de performance provocados pelo aumento do número de processadores. Isto é comprovado pelos resultados apresentados já que há casos em que o aumento de workers prejudicou a performance em problemas de pequena dimensão, mas em tabuleiros de maior dimensão existiram execuções em que a versão paralelizada foi 6 vezes mais rápida.

Por outro lado, foi um obstáculo a decisão de qual das partes do código deveriam ser paralelizadas pois esta decisão não deve ser feita apenas baseada na intuição mas sim com a análise de dados empíricos que o comprovem.

Outro dos dados interessantes a reter é que neste ambiente académico de testes a eficiência não é importante pois assume-se que os recursos (por exemplo, energéticos) são ilimitados. Mas numa aplicação real é importante a análise da eficiência pois utilizar mais processadores (gastar mais energia) sem observar um aumento significativo de performance não será útil, sendo que se terá que analisar estes *trade-offs* para chegar à decisão final dos recursos a ser utilizados.

André Lopes nº45617
Nelson Coquenim nº45694
Grupo 23

Gráficos

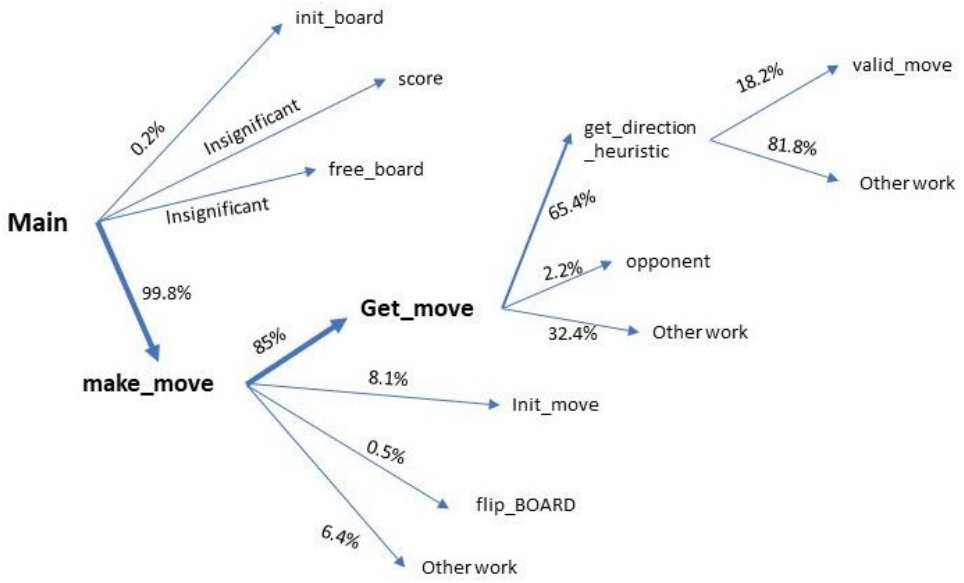

Fig.1 - Esquema da duração (%) correspondente a cada função, sem paralelização, com um tamanho de tabuleiro de 100*100 casas

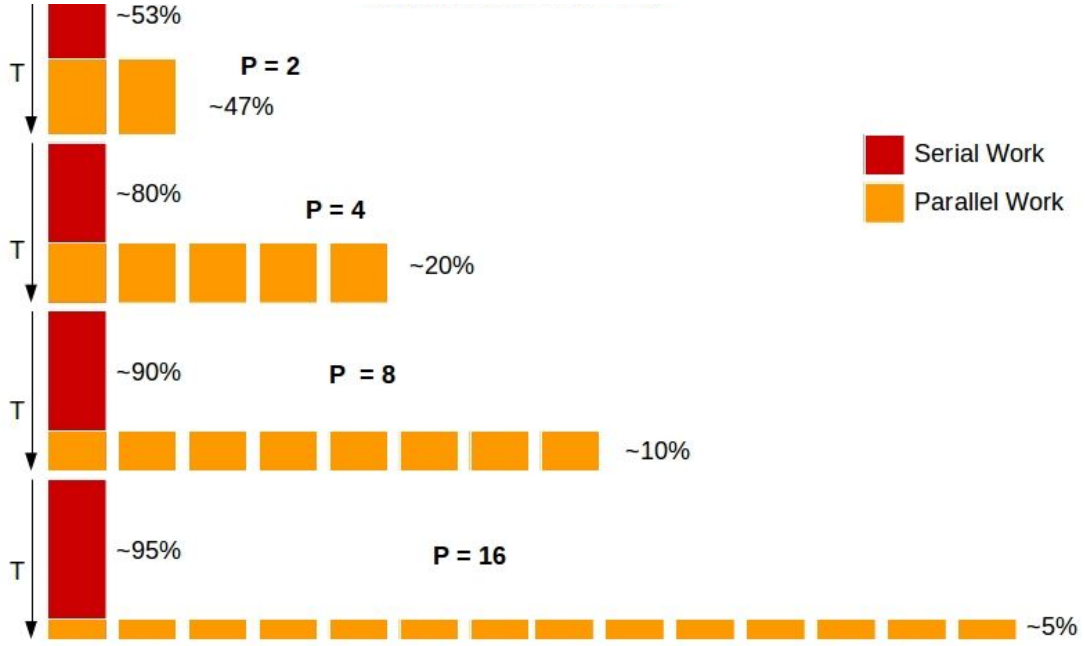

Fig.2 - Esquema do trabalho realizado pela porção do código paralelizado e pela porção em série, com um *board size* de 100, resultados obtidos através da média de 5 medições.

André Lopes nº45617
Nelson Coquenim nº45694
Grupo 23

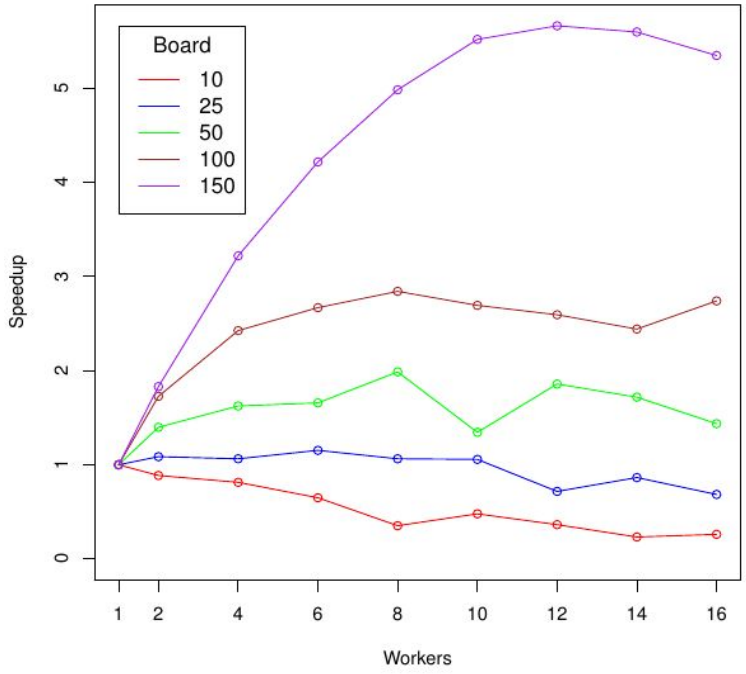

Fig.3 - Gráfico que ilustra o speedup

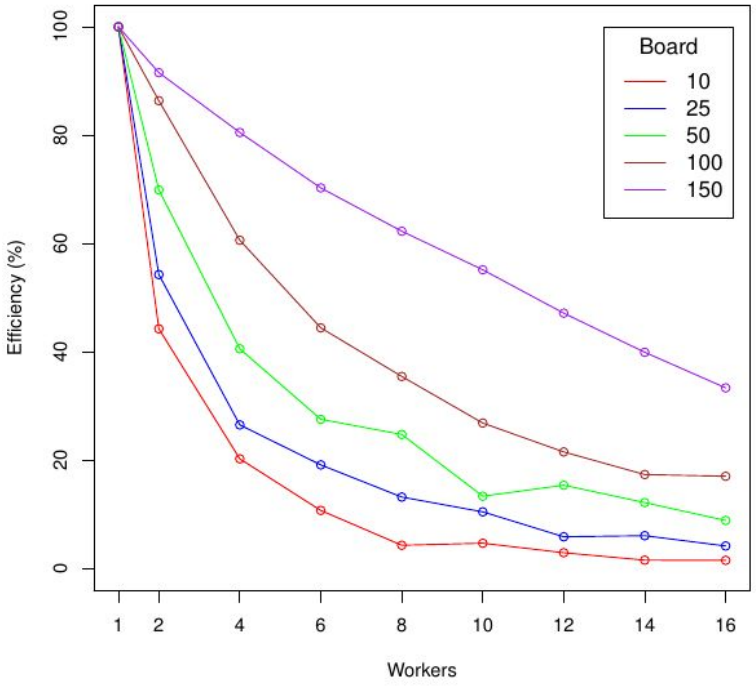

Fig.4 - Gráfico que ilustra a eficiência

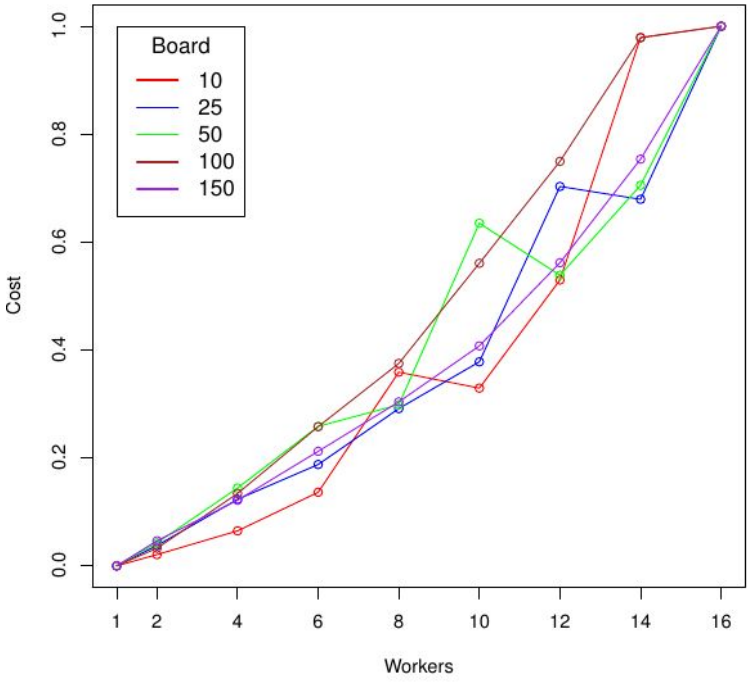

Fig.5 - Gráfico que ilustra o custo

André Lopes nº45617
Nelson Coquenim nº45694
Grupo 23

Agradecimentos

Ambos os elementos do grupo agradecem a disponibilidade dos professores João Lourenço e Bernardo Ferreira em esclarecer dúvidas mas também os colegas que participaram na discussão na plataforma do Piazza em especial ao aluno André Neves do grupo 23 número 46264 por ter disponibilizado uma versão de código R que serviu de base para a análise do problema.

Agradecer também aos autores das seguintes páginas web que nos ajudaram a esclarecer certas dúvidas:

https://www.harding.edu/fmccown/r/

Bibliografia

[1] McCool M., Arch M., Reinders J.; Structured Parallel Programming: Patterns for Efficient Computation; Morgan Kaufmann (2012); ISBN: 978-0-12-415993-8