# Computação de Alto Desempenho - Trabalho Prático 1

João Ferreira & José Ribeiro
Departamento de Engenharia Informática
Universidade de Coimbra
jpbat@student.dei.uc.pt | jbaia@student.dei.uc.pt
2009113274 | 2008112181

Abril 2013

# Conteúdo

# 1 Introdução

A tecnologia chegou a um ponto em que se torna cada vez mais difícil conseguir aumentar a velocidade de um processador, por razões como a dissipação do calor por este gerado, ou pela barreira da velocidade da luz, uma vez que são cada vez necessários mais ciclos de relógio para aceder a um mesmo sítio da memória. Com isto torna-se cada vez mais importante ser capaz de explorar conceitos como *multi-core* e *HyperThreading*, tecnologias que permitem que se executem tarefas em paralelo.

Neste projecto foi-nos apresentado um problema da vida real, em que o objectivo era conseguir processar um grande número de transacções bancárias, atribuindo a cada uma delas uma ou várias classificações, caso estas fizessem *match* a uma ou mais regras. Como é claro, para conseguirmos maximizar o *throughput* foi necessário recorrer à paralelização.

Para se efectuar uma paralelização eficiente é necessário ter um bom algoritmo base, sendo que o primeiro passo passou por tentar escrever o melhor algoritmo possível, passando depois esse algoritmo para a sua versão paralela.

Durante este documento vão ser abordadas as metodologias utilizadas, explicando o desenvolvimento do algoritmo a cada passo, bem como as alterações que foram implementadas.

# 2 Metodologia

## 2.1 Algoritmo

Começou por se usar o algoritmo mais básico de complexidade $O(m*n)$ com m sendo o número de regras e n o número de transacções, sendo que por discussão entre os membros do grupo se chegou à conclusão que utilizar um algoritmo com uma pesquisa semelhante à binária poderia apresentar bons resultados.

Foi primeiro implementada a versão sequencial deste algoritmo sendo que, quando esta se mostrou correctamente implementada, procedeu-se à sua paralelização.

## 2.2 Linguagem e Paralelização

Optámos por utilizar *C*, uma vez que o grupo se sentia a vontade com a linguagem, e também porque, segundo o professor, seria uma linguagem com que não iríamos ter bastantes problemas no segundo trabalho. Além disso, outro grande factor a favor desta linguagem foi o facto de permitir o uso de `OpenMP`, que começou por ser o método escolhido para implementar o paralelismo, passando posteriormente a usar `pthreads`, como irá ser explicado mais a frente.

## 2.3 Ambiente de Testes

Os tempos apresentados neste relatório foram recolhidos no computador de um dos elementos do grupo, sendo que este seria o mais semelhante ao que o docente havia referido no enunciado (Core2-Duo @ 2.5GHz).

**Processador** Intel Core i5 M 460 @ 2.53GHz, que possui 2 cores com *HyperThreading* e 3MB de L2 Cache.

**Memória RAM** 4GB DDR3 @ 1067MHz.

**Sistema Operativo** Ubuntu Linux 12.04 64bits.

# 3 Algoritmos

## 3.1 Naïve

Tal como já foi dito começámos por implementar o algoritmo mais simples, de complexidade $O(m * n)$ com m sendo o número de regras e n o número de transacções.

### 3.1.1 Descrição

Começa-se por se iterar sobre a lista das transacções, depois pela lista das regras. Em cada uma destas iterações interiores, fazem-se ainda N[1] iterações para comparar cada um dos elementos da transacção com o respectivo elemento da regra e ver se estes produzem um match. Quando se chega ao fim desta iteração sem que este ciclo seja interrompido, então foi encontrado um match.

### 3.1.2 Pseudo Código

```
for transaction in transactions do
  for rule in rules do
    for i in len(transaction) do
      if transaction[i] != rule[i] and rule[i] != 0 then
        break
      end if
    end for
    if i == len(transaction) then
      print match
    end if
  end for
end for
```

## 3.2 Bounded Search

Esta foi a abordagem escolhida, sendo que apresenta uma complexidade de $O(n \log m)$, com n sendo o número de transacções e m o número de regras existentes.

### 3.2.1 Descrição

Para esta solução é mandatório que se comece por ordenar as regras. Depois, itera-se sobre a lista das transacções e dentro de cada uma destas começa-se por se iterar sobre cada um dos N elementos dessa transacção. Tendo em conta esse valor e o índice do elemento em que estamos, vai-se procurar na lista de regras o primeiro e último 0 (*wildcard*) que existem nessa posição e guardar esses valores, fazendo de seguida o mesmo mas para o valor real da transacção na posição. Por fim é efectuada uma chamada recursiva com os novos

[1] número de elementos de uma transacção.

valores de ínicio e fim da pesquisa, de modo a que este crie uma árvore de pesquisa recursiva de complexidade logaritmica. Falta referir apenas que antes de cada chamada recursiva é verificado se foram ou não encontrados *bounds* como modo de *trimming* da árvore de pesquisa.

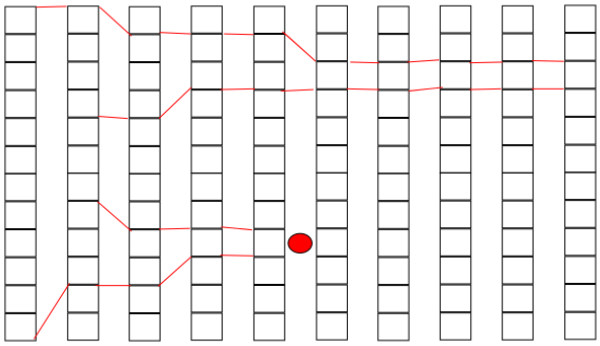

Figura 1: A vermelho os *bounds* ao longo do algoritmo

### 3.2.2 Pseudo Código

```
for transaction in transactions do
  for index in transaction do
    if index == len(transaction) then
      print match
    end if
    first_zero ⇐ find_first_zero(0, len(rules))
    last_zero ⇐ find_last_zero(first_zero, len(rules))
    first_value ⇐ find_first_value(last_zero + 1, len(rules), transaction[index])
    last_value ⇐ find_first_zero(first_value, len(rules), transaction[index])
    if last_zero not NOT_FOUND then
      find_match(transaction, first_zero, last_zero)
    end if
    if last_value not NOT_FOUND then
      find_match(transaction, first_value, last_value)
    end if
  end for
end for
```

# 4 Evolução

Começaram por se usar as directivas de `OpenMP` para utilizar paralelismo.

Na parte de leitura de ficheiros foram criadas `sections` para que tanto o ficheiro das transacções como o ficheiro das regras pudessem ser carregados de modo paralelo. Nessa função começou por se fazer a leitura recorrendo ao `fscanf()`, algo que se demonstrou ser bastante lento, razão pela qual acabou por se recorrer à função `fgets()` que, em conjunto com um parser desenvolvido pelo grupo, é capaz de ler o ficheiro cerca de 10 vezes mais rápido.

Numa segunda parte foi adicionada a directiva `#omp pragma parallel for` ao primeiro ciclo do algoritmo de *bounded search*, sendo que esta alteração obrigou a que fosse implementada uma regra de acesso em exclusão mútua à função de *output*.

Nesta fase começou por se implementar um `flockfile`, sendo que esta solução se tornou algo inviável uma vez que como cada *thread* necessitava de imprimir para poder continuar a trabalhar, estas nunca utilizavam o processador a 100%.

Quase institivamente surgiu a ideia de optar por um esquema de produtor/consumidor, em que as *threads* estariam responsáveis por adicionar os *matches* encontrados a uma *queue*, de onde outra *thread* iria concorrentemente imprimir os *matches* que já se encontravam nessa *queue*. Tal esquema foi implementado utilizando `pthread_mutex`s e `pthread`s (para a `thread` de *output*).

Depois de uma análise cuidada de todas as chamadas de funções, estudou-se a possibilidade de passar por referência (ao invés de proceder à cópia de valores), de forma a obter maior performace. Tais modificações aplicaram-se essencialmente nas funções de `enqueue` e `dequeue`. Com esta modificação obteve-se um aumento considerável do número de transacções por segundo.

Depois de observarmos as melhorias de performance obtidas com a utilização de *pthreads*, optou-se pela substituição completa do `OpenMP`. Para isso foi implementada uma `queue` de trabalho, de onde as `threads` poderiam obter as transacções a processar. Este esquema demonstrou-se bem sucedido, uma vez que a complexidade deste mecanismo é menor que a do funcionamento interno do `OpenMP`.

Uma vez que o processamento continuava a ser prejudicado pelo `lock` efectuado na `queue` de `output` (e, consequentemente, ao ficheiro), decidimos abdicar da `thread` dedicada a *output* para, recorrendo às primitivas de sincronização da biblioteca `pthreads`, tornar todas as `threads` produtoras. Para isto, decidimos utilizar um `buffer` por `thread`, onde estas faziam o seu *output* à medida que procediam ao processamento. A escrita para este *buffer* era feita de forma a substituir as chamadas à função de `fprintf`, uma vez que esta é bastante lenta. Cada vez que o *buffer* individual de uma dada `thread` ficava cheio, esta fazia um `lock` ao ficheiro de *output* e, utilizando `fwrite` (uma chamada mais rápida para *output* para ficheiros) este era escrito e limpo. O valor do tamanho do *buffer* interno de cada `thread` foi `fine-tuned` e relacionado directamente com o tamanho do *work batch*. *Work batch* consiste no `dequeue` de várias transacções em simultâneo, de forma a reduzir o número de chamadas `lock/unlock`.

Em resumo segue-se uma tabela com as várias versões do programa e as alterações que foram efectuadas entre estas.

| Versão | Descrição |
|---|---|
| **1.0** | Versão sequncial do algoritmo de *binary search* com leitura do ficheiro utilizando *fscanf()*. |
| **2.0** | Leitura feita com recurso a *fgets()* |
| **3.0** | Adição da directiva *#omp pragma parallel for* ao primeiro ciclo do algoritmo de *bounded search* com utilização de *flockfile* para output. |
| **4.0** | Implementação do esquema de *producer / consumer* para tratar do output e da divisão de transacções pelas *threads*. |
| **5.0** | Implementação do *buffer* em cada *thread* para tratar do output. |

Tabela 1: Descrição das diferentes versões do programa.

# 5 Resultados

## 5.1 Medição dos Tempos

Tal como referido os tempos foram medidos no servidor indicado pelo docente, sendo que cada um dos valores apresentados foi alvo de tratamento estatístico. Isto é, juntamente com cada um dos valores é apresentado o desvio padrão, que foi obtido através da análise estatistica de 30 medições.

| Versão | Tempo Médio | Tempo Max | Tempo Min | Desvio Padrão | TPS |
|---|---|---|---|---|---|
| **1.0** | 24.106 $s$ | 24.909 $s$ | 23.292 $s$ | 0.324 $s$ | 41483 |
| **2.0** | 22.383 $s$ | 22.756 $s$ | 21.620 $s$ | 0.237 $s$ | 44676 |
| **3.0** | 33.600 $s$ | 35.110 $s$ | 31.643 $s$ | 0.854 $s$ | 29761 |
| **4.0** | 10.650 $s$ | 11.155 $s$ | 10.650 $s$ | 0.189 $s$ | 93896 |
| **5.0** | 8.0433 $s$ | 8.310 $s$ | 7.592 $s$ | 0.176 $s$ | 124327 |

Tabela 2: Medições estatísticas dos tempos de execução das várias versões do programa.

Para obtermos uma noção do ganho obtido com a paralelização, fez-se variar o número de `cores` em utilização (1 - 4) para a última versão. Apresenta-se em seguida o gráfico obtido:

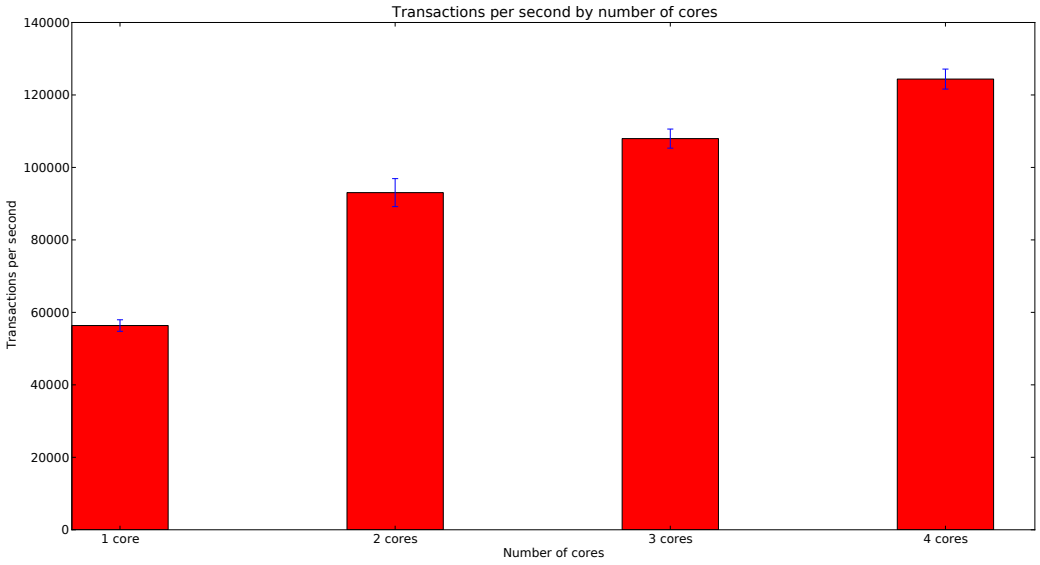


Figura 2: Número de transacções por segundo pelo número de `cores` (versão 5).

Os valores de tempo e speedup obtidos são de seguida apresentados:

| | 1 core | 2 cores | 3 cores | 4 cores |
|---|---|---|---|---|
| **Tempo (em ms)** | 17753.8 | 10765.13 | 9268.2 | 8043.3 |
| **Speedup** | 1.0 | 1.649 | 1.916 | 2.207 |

Tabela 3: Tempos e `speedups` obtidos segundo o número de `cores` (versão 5).

## 5.2 Análise de desempenho

Tal como podemos observar na tabela acima, foi conseguido um *speedup* de apenas 2.207 quando executado num processador de 4 núcleos (e de 1.6 quando utilizados 2 `threads`).

Estes resultados fazem-nos considerar (e a análise de consumo de CPU ao longo da execução o comprova) que o programa continua a ser largamente *IO bounded*. Apesar disso, existem ainda algumas melhorias que poderiam ser implementadas (não implementadas por falta de tempo).

**Paralelização do *sort*** Fazer *sort* ao *array* das regras é uma das fases mais pesadas, visto que demora cerca de 1 $s$. Uma versão paralela deste algoritmo poderia implicar descer, no caso da utilização de quatro *cores*, 750 $ms$ o que representaria um ganho substancial no número de transacções processadas por segundo.

**Transposição das regras** A transposição da matriz de regras poderia trazer ganhos substanciais por usufruir de `cache`. Isto porque o acesso à memória causado pelas funções de pesquisa da `lowerbound` e `upperbound` o faz por cada `slot` da regra (e não linearmente). Pensamos que a implementação desta melhoria poderia introduzir grandes melhorias já que, em experiências anteriores, vimos que um correcto uso desta memória pode melhorar os tempos de execução.

# 6 Conclusão

Ao longo deste projecto, fomos iterando sobre um algoritmo, o qual submetemos a diferentes processos de modo a melhorar o seu desempenho. Uma vez mais ficou provado que normalmente, a solução mais simples é aquela que, mesmo demorando menos na fase de implementação e depuração de error, consome mais tempo na fase de execução.

O primeiro ponto foi repensar o algoritmo em si. Em vez de efectuar pesquisar lineares optámos por implementar pesquisas binárias, o que só por si traz inúmeras vantagens.

Contudo, rapidamente se chega a um momento onde as optimizações se tornam cada vez mais rebuscadas e menos significativas, o que, torna apenas o código mais ilegível sem que tenha efeitos práticos na sua *performance*. É neste momento em que a paralelização aparece para dar um novo fôlego ao processo.

Denotou-se que apesar de trazer benefícios em grande partes dos casos, a paralelização tem de ser vista como um possível *bottleneck* se for aplicada erradamente. A criação de demasiadas *threads* e a sincronização entre elas pode sobrecarregar demasiado o sistema, tal como observámos quando tentámos fazer o *print* dos *matches* com recurso ao *flockfile*.

Contudo, paralelizar a iteração sobre todas as entradas, assim como as leituras a partir de ficheiros diferentes, revelou-se uma mais valia que trouxe melhores resultados a seguir ao melhoramento da base do algoritmo. Isto prova que a paralelização, quando bem aplicada, potencia o uso eficiente dos recursos disponíveis.

O manuseamento de ficheiros acaba mesmo por ser o ponto fraco da maioria das aplicações, uma vez que as leituras e escritas têm de ser na maioria dos casos sequenciais. Para contornar estes aspectos, usámos *threads* para correr leituras simultâneamente de modo a diluir o tempo de escrita no restante tempo do algoritmo.

Para além deste aspecto, as próprias leituras podem ser melhoradas, uma vez que se podem ler grandes quantidades de dados numa única acção em vez de ler pequenos pedaços de muitas vezes.

Ter dispensado algum tempo a estudar e implementar os detalhes de implementação foi algo que se tornou bastante proveitoso. A versão naïve nunca foi totalmente corrida para os testes maiores, já que o tempo de execução era simplesmente demasiado longo.

Estes foram os pontos sobre os quais nos focámos, diferentes abordagens e perspectivas de optimização que em conjunto, proporcionam o objectivo de qualquer programador, criar um trabalho de computação de alto desempenho para o ambiente de execução das suas aplicações. Quanto mais rapidamente uma tarefa for executada, maior *throughput* iremos em priñcípio alcançar e maior sucesso o *software* irá certamente alcançar.