# Sobre um Mecanismo de Controlo de Concorrência baseado em Grupos de Recursos*

Nuno Delgado and Hervé Paulino

CITI / Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
2829-516 Caparica, Portugal

**Abstract.** A consolidação do modelo de memória partilhada na programação de arquiteturas *multi-core* tem centrado atenções no conhecido problema da gestão de estado partilhado. Nessa medida este artigo propõe um mecanismo de controlo de concorrência que permite a aquisição atómica, e mutualmente exclusiva, de conjuntos de recursos. A abordagem distingui-se relativamente a outras na área por permitir composicionalidade livre de *deadlocks* tanto em cenários estáticos como dinâmicos.

## 1 Introducão

A crescente consolidação do modelo de memória partilhada como standard *de-facto* para a programação de arquiteturas *multi-core* tem, cada vez mais, exposto as limitações das soluções de controlo de concorrência existentes. Se por um lado os mecanismos de sincronização baseados em *locks* são suscetíveis a situações de interbloqueio (*deadlocks*) e não são escaláveis, por outro, os sistemas baseados em Memória Transacional de Software (STM) [1, 2] requerem um suporte que concorre com a execução do programa propriamente dito, o que impõe uma penalização não negligenciável no desempenho total.

Nesta medida, este artigo apresenta uma alternativa centrada nas noções de *recurso* e de *grupo de recursos* (GR). Estes podem ser adquiridos (e libertos) de forma atómica e mutuamente exclusiva, por forma a controlar a concorrência tanto em cenários estáticos, em que os recursos necessários são conhecidos de antemão, como em cenários dinâmicos, em que os recursos vão sendo adquiridos à medida que a zona crítica é executada. A ocorrência de *deadlocks* é evitada através de um particionamento do estado da aplicação, que coloca na mesma partição os GRs interdependentes. O nosso trabalho partilha afinidades com os *Atomic Sets* [3]. No entanto, estes utilizam uma abordagem diferente que garante a ausência de *deadlocks* apenas em cenários estáticos. Assim sendo, as contribuições deste artigo são um mecanismo de controlo de concorrência centrado na noção de recurso que oferece propriedades de segurança, nomeadamente ausência de *deadlocks* e composicionalidade, e uma implementação protótipo em Java, que permite comparar a nossa abordagem com outras na área.

* Este trabalho foi parcialmente financiado pela FCT-MCTES através do PEst-OE/EEI/UI0527/2011 - Centro de Informática e Tecnologias da Informação (CITI/FCT/UNL) - 2011-2012.

## 2 Controlo de Concorrência com Grupos de Recursos

Os GRs permitem que um *thread* possa adquirir acesso exclusivo aos recursos de que necessita para executar uma região crítica com segurança. Para tal, esses recursos têm de ser agregados em grupos, os quais expõem uma interface de operações para a gestão do seu acesso mutuamente exclusivo.

Sempre que possível impomos uma semântica *tudo-ou-nada* no processo de aquisição de GRs, i.e. este é realizado a partir de uma operação atómica que garante acesso exclusivo a todos os seus elementos e, uma vez adquirido, um GR não pode crescer. Designamos estes grupos por *grupos estáticos*. A sua construção é realizada de forma incremental com recurso à operação add e a sua aquisição, e libertação, é efetuada através das operações staticAcquire e release, respectivamente. A listagem 1 ilustra a aplicação do procedimento ao conhecido exemplo da transferência de dinheiro entre duas contas bancárias.

```
1 boolean transfer(Account src, Account dest,
      int amount) {
2  ResourceGroup g = new ResourceGroup();
3  g.add(src);  g.add(dst);
4  g.staticAcquire();
5  boolean canTransfer = src.withdraw(amount);
6  if (canTransfer) dst.deposit(amount);
7  g.release();
8  return canTransfer;
9 }
```

Listagem 1: Ex. de grupos estáticos

```
1 boolean transfer2(Account src, Account src2,
      Account dest, int amount) {
2  ResourceGroup g = new ResourceGroup();
3  g.add(dest);       g.acquire();
4  boolean canTransfer = transfer(src, dest);
5  if (canTransfer == false)
6    canTransfer = transfer(src2, dest);
7  g.release();
8  return canTransfer;
9 }
```

Listagem 2: Ex. de grupos dinâmicos

Pode-se facilmente provar que esta aproximação é livre de deadlocks, mas a sua aplicação é restrita aos casos em que os recursos necessários são conhecidos em antemão. Por essa razão, levantamos a restrição referida sobre o crescimento dos grupos, permitindo o que passamos a designar por *grupos dinâmicos*. Contudo, isto não é alcançado sem um custo. Impomos restrições sobre o que pode executar em paralelo, uma vez que a aquisição dinâmica expõe o sistema a possíveis situações de *deadlock*. Para evitar tais padrões particionamos o estado da aplicação de modo a que recursos interdependentes sejam colocados numa mesma partição. Posteriormente, associamos os grupos que contêm esses recursos à respectiva partição, e forçamos o processo de aquisição a obter também acesso exclusivo à partição. Desta forma a condição de espera circular que poderia originar *deadlocks* é quebrada.

De modo a atenuar os impactos no desempenho causados pelas restrições à concorrência, identificamos dois tipos de partição. Recursos que excedam o escopo de um objecto são colocados em partições globais, cujo alcance é a aplicação. Recursos locais a um dado objeto só precisam de ser acedidos em exclusão mútua por threads operando dentro desse mesmo objeto e, portanto, são colocados em partições privadas.

O processo de particionamento não pode ser efetuado em tempo de execução devido à natureza dinâmica do grupos em questão, sendo, portanto, remetido para os passos de compilação e/ou ligação. Todavia, uma análise estática não consegue inferir que recursos serão utilizados ao longo da aplicação. Por conseguinte decidimos utilizar o *tipo* dos recursos como a nossa unidade de partição.

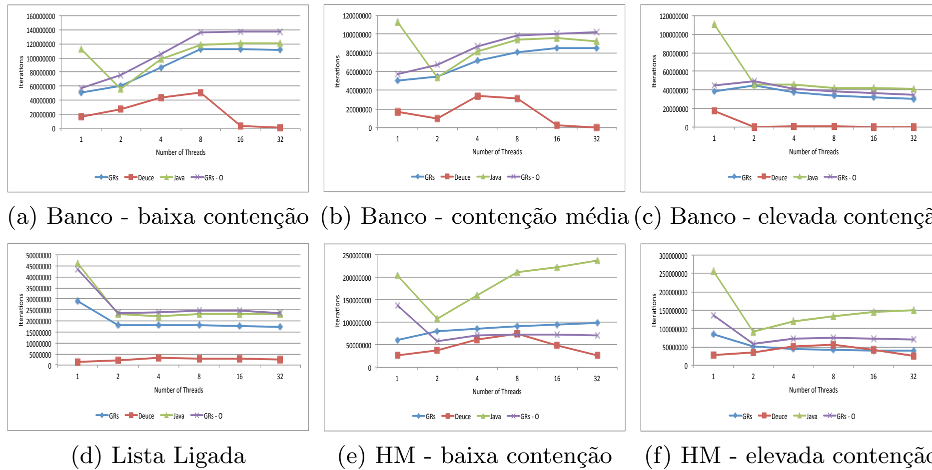
As restrições de concorrência impostas são transparentes para o programador, tendo este apenas de indicar a semântica de aquisição: *estática* ou *dinâmica* - a última é representada pela operação acquire. A listagem 2 baseia-se no exemplo anterior para ilustrar o conceito de grupo dinâmico. Para além das contas src e dest, o exemplo introduz a conta src2 que será utilizada como fonte para a transferência, no caso da original não ter fundos suficientes. A operação acquire (linha 3) adquire dinamicamente o grupo e a partição a ele associada. Esta partição inclui o tipo Account, dado que a análise dos métodos transfer2 e transfer produz, respectivamente, as cadeias de dependências: Account e Account → Account. Assumindo que as contas são recursos públicos, a partição é classificada como *global*. Consequentemente, apenas um grupo contendo recursos globalmente acessíveis do tipo Account pode estar adquirido num dado instante. Desta forma é garantida a ausência de deadlocks, a qual juntamente com a propriedade de reentrância na aquisição de GRs permite a composição segura.

## 3 Experiências

Esta secção apresenta alguns exemplos simples de programação e a avaliação do seu desempenho, comparando-os com o mecanismo de sincronização nativo do Java e com o Deuce [4], um dos mais conhecidos sistemas de STM para Java. As medições foram efetuadas num computador equipado com um CPU Intel CPU i7-2670QM (com 4 cores *hyperthreaded*) a 2.2GHz e 6 GB de memória RAM.

**Grupos estáticos:** O benchmark contabiliza o número de transferências realizadas entre diversas contas existentes num dado banco, uma generalização do problema apresentado na listagem 1. Foram avaliados três cenários como graus de contenção diferentes: *elevada* - 5 contas, *média* - 50 contas e *baixa* - 500 contas. A performance dos GR excede claramente o Deuce nos três cenários, o qual é claramente penalizado pelo número de conflitos de escrita (gráficos (a) a (c) da Figura 1). O problema em causa é puramente estático, pelo que a sobrecarga associada à gestão de grupos dinâmicos pode ser eliminada - linha GRs-O nos gráficos. O desempenho global aumenta cerca de 20%, superando os blocos `synchronized` nos cenários de baixa e média contenção.

**Grupos dinâmicos:** A avaliação de grupos dinâmicos foi realizada com recurso a duas aplicações: lista ligada e tabela de dispersão (HM). O exemplo HM avalia os GRs em cenários de baixa e alta contenção - 5 e 50 listas de colisões, respetivamente. Os gráficos (e) e (f) mostram que perante uma baixa contenção os GR têm um desempenho superior ao do Deuce e que, com o aumento da contenção, a dispersão existente beneficia claramente a STM. A linha GRs-O refere-se a uma nova otimização, aplicável apenas a contextos puramente dinâmicos. Nestes, a aquisição da partição é suficiente para garantir a não interferência entre threads, pelo que não é necessário adquirir os recursos em causa. Na realidade troca-se algum do paralelismo no acesso às estruturas pela redução do overhead das operações de aquisição. O desempenho revela-se melhor que o Deuce mas ainda assim perde contra os blocos synchronized. Os ganhos obtidos no cenário de alta contenção justificam-se pelo facto do problema, já por si, não permitir um grande grau de paralelismo.




(a) Banco - baixa contenção (b) Banco - contenção média (c) Banco - elevada contenção

(d) Lista Ligada (e) HM - baixa contenção (f) HM - elevada contenção

Fig. 1: Gráficos de desempenho

## 4 Conclusões

Este artigo apresenta um mecanismo de controlo de concorrência orientado a recursos que, por um lado, fornece propriedades de segurança, tais como a ausência de deadlocks e a composicionalidade, e por outro, apresenta bons níveis de desempenho. Os GRs podem ser utilizados com segurança tanto em cenários estáticos como dinâmicos. Uma análise estática, e consequente instrumentalização do código, garante o particionamento dos recursos agregados de tal forma que elimina a possível ocorrência de deadlocks.

Avaliámos os GRs do ponto de vista de desempenho, comparando-os com o mecanismo de sincronização nativo do Java e com um sistema de STM conhecido (Deuce). Na maioria dos casos os GRs obtiveram um desempenho próximo dos blocos sincronizados do Java e, portanto, claramente superior ao Deuce. Existe contudo espaço para melhorias no tratamento dos cenários dinâmicos, particularmente na presença de conflitos de leitura/escrita, trabalho que está em curso.

## References

1. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (PODC '95), New York, NY, USA, ACM (1995) 204–213
2. Dice, D., Shavit, N.: Understanding tradeoffs in software transactional memory. In: Fifth International Symposium on Code Generation and Optimization (CGO 2007), IEEE Computer Society (2007) 21–33
3. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, ACM (2006) 334–345
4. Korland's, G.: Deuce STM - Java Software Transactional Memory. `http://sites.google.com/site/deucestm/`