Universidade do Minho
Departamento de Informática

Mestrado em Engenharia Informática

Técnicas de ponto de controlo e adaptação em grelhas computacionais

Bruno Silvestre Medeiros

Dissertação para obtenção do
Grau de Mestre em Engenharia Informática

Trabalho realizado sob a orientação do:
Professor Doutor João Luís Sobral

Braga, 31 de Agosto de 2011

"O trabalho afasta de nós três grandes males: o tédio, o vício e a necessidade."

Voltaire

Resumo

A recente popularidade dos ambientes de grelhas introduziu a necessidade de suportar a execução robusta de aplicações numa gama alargada de recursos computacionais. Em contextos de grelhas computacionais, onde a fiabilidade e disponibilidade dos recursos não é garantida, as aplicações deverão ser capazes de suportar dois requisitos fundamentais: 1) tolerância a faltas; 2) adaptação aos recursos disponíveis. As técnicas tradicionais utilizam uma abordagem "caixa-negra", onde a camada intermédia de *software* (mediador) é a única responsável por assegurar estes dois requisitos. Estes tipos de abordagens possibilitam o suporte a estes serviços com uma intervenção mínima do programador, mas limitam a utilização de conhecimento sobre as características da aplicação, visando a optimização destes serviços. Nesta tese são apresentadas abordagens orientadas aos aspectos para suportar tolerância a faltas e adaptação dinâmica aos recursos em grelhas computacionais. Nas abordagens propostas, as aplicações são aprimoradas com capacidades de tolerância a faltas e de adaptação dinâmica através da activação de módulos adicionais. A abordagem de tolerância a faltas utiliza a estratégia de ponto de controlo e restauro, enquanto a adaptação dinâmica utiliza uma variação da técnica de sobre-decomposição. Ambas são portáveis entre sistemas operativos e restringem a quantidade de alterações necessárias no código base das aplicações. Além disso, as aplicações poderão adaptar de uma execução sequencial para uma configuração multi-cluster. A adaptação pode ser realizada efectuando o ponto de controlo da aplicação e restaurando-a em diferentes máquinas, ou então, realizada em plena execução da aplicação.

Palavras-chave: ponto de controlo e restauro; adaptação dinâmica aos recursos; grelhas computacionais; programação orientada aos aspectos.

Abstract

Grids' recent popularity introduced the necessity of supporting robust execution of applications on a wide range of computing resources. In computational grids' context, where reliability and availability are not granted, applications must support two fundamental requirements, namely, fault tolerance and adaptation to available resources. Traditional techniques use a "black-box"approach, where middleware is the only sponsor for those requirements. These kind of approaches enable this services' support with a minimum programmer's intervention, but limits knowledge utilization of application's features in order to optimize services. This thesis presents aspect-oriented approaches to support fault tolerance and dynamic adaptation to resources in computational grids. In the proposed approaches, applications are enhanced with the ability of fault tolerance and dynamic adaptation through additional modules activation. Fault tolerance approach uses a check point and restore strategy while dynamic adaptation uses a variation of the over-decomposition technique. Both are portable between operating systems and minimize alterations to base code of applications. Moreover, applications can adapt from a sequential execution to a multi-cluster configuration. Adaption can be performed by checkpointing the application and restarting on a different mode or can be performed during run-time.

Key Words: checkpointing and restart; run-time adaptation; grid computing; aspect oriented programming.

Agradecimentos

Antes de mais, e sobretudo, ao meu orientador Sr. Professor Doutor João Luís Sobral, por ter me orientado durante este projecto, pela sua disposição e pela realização quase semanal de reuniões e apresentações de grupo. Nunca poderei exprimir plenamente a minha gratidão para com a minha excepcional família, pelo seu suporte quer a nível financeiro como emocional. Não podia deixar de agradecer o suporte financeiro dos projectos AspectGrid (GRID/GRI/81880/2006) e GAsPar (PTDC/EIA-EIA/108937/2008), que me proporcionaram a possibilidade de realizar um trabalho estimulante, alicerce da minha tese. E por fim, um agradecimento especial à minha namorada Rafaela, pelo seu apoio incondicional.

Conteúdo

Lista de figuras

Lista de tabelas

Abreviaturas

CL	**C**handy- **L**amport
FIFO	**F**irst **I**n **F**irst **O**ut
FPC	**F**icheiros de **P**onto de **C**ontrolo
GPC	**G**ravação do **P**onto de **C**ontrolo
LE	**L**inha de **E**xecução
LEM	**L**inha de **E**xecução **M**estre
PARC	**P**alo **A**lto **R**esearch **C**enter
PCL	**P**onto de **C**ontrolo **L**ocal
PCNA	**P**onto de **C**ontrolo ao **N**ível **A**plicacional
PCNSO	**P**onto de **C**ontrolo ao **N**ível do **S**istema **O**perativo
PCR	**P**onto de **C**ontrolo e **R**estauro
POA	**P**rogramação **O**rientada aos **A**spectos
SMD	**S**istema de **M**emória **D**istribuída
SMPD	**S**istema de **M**emória **P**artilhada **D**istribuída
SPM	**S**istema de **P**assagem de **M**ensagens

Capítulo 1

Introdução

O conceito de grelhas é relativamente recente, tendo sido introduzido primeiramente por Foster e Kesselman em 1998 [IC98]. Foster, Kesselman e Tuecke definiram, em 2001, grelhas como partilha de recursos, utilizados para a resolução de problemas em organizações virtuais multi-institucionais [FKT01]. De uma forma mais geral, as grelhas são serviços de partilha de recursos, que incluem poder computacional, armazenamento, sensores e redes, através da internet.

As grelhas podem ser classificadas em vários tipos, consoante a sua finalidade, nomeadamente: - grelhas computacionais [BFH03]; - grelhas de dados [CFK+99]; - grelhas de colaboração [McQ04]; - grelhas de rede (também conhecidas como grelhas de entrega)[LZZ04]. O trabalho descrito ao longo desta tese recai sobre grelhas computacionais, as quais recebem especial atenção.

O surgimento das grelhas computacionais remonta aos anos 90, com o objectivo de auxiliar actividades de pesquisa e desenvolvimento científico. A premissa subjacente foi a maximização da utilização dos recursos computacionais existentes, através da partilha de poder computacional, entre organizações distintas. A ideologia das grelhas computacionais foi inspirada na metáfora de uma rede eléctrica, na qual o seu utilizador obtém acesso à energia eléctrica simplesmente ligando a tomada à corrente, sem a necessidade de saber de onde esta energia é proveniente e/ou as particularidades das malhas de transmissão e distribuição por detrás desta. Seguindo esta analogia, um indivíduo que se conecta para obter recursos computacionais na grelha terá acesso ao poder computacional, sem a necessidade de conhecer de onde este provém e qual a rede de ligação aos recursos. As grelhas computacionais consistem assim na agregação de recursos computacionais derivados de diversos domínios administrativos, que podem estar dispersos geograficamente, utilizados na resolução de uma tarefa em comum, usualmente relacionada com um problema de carácter científico, técnico ou de negócio.

O que diferencia as grelhas das demais plataformas distribuídas é o facto de estas serem mais diversas e complexas, devido ao seu nível de heterogeneidade; à sua alta dispersão geográfica (as grelhas podem ter uma escala global, agregando recursos provenientes de várias partes do planeta); aos múltiplos domínios administrativos (agregam recursos de várias instituições) e ao controlo distribuído sobre as grelhas (tipicamente, não existe uma única entidade que detenha todo o controlo sobre ela) [CSN05].

Em contextos de grelhas computacionais, a fiabilidade das máquinas remotas não pode ser tomada como garantida, assim como a disponibilidade dos recursos, que poderá variar durante a execução de uma aplicação. Sob tal panorama, a tolerância a faltas [Nel90] e a capacidade de adaptar dinamicamente as aplicações aos recursos disponíveis [FPS06], tornam-se dois requisitos fundamentais em ambientes de grelhas computacionais.

O tipo de máquinas partilhadas pela grelha e a sua própria natureza fazem com que uma parte significativa do trabalho intensivo das aplicações executadas sobre a grelha possa ser realizada em paralelo, nomeadamente, com programas baseados na passagem de mensagens. Obter tolerância a faltas em aplicações executadas em paralelo ou distribuídas por vários recursos, envolve um conjunto adicional de preocupações, tais como a portabilidade, consistência e optimização dos dados do ponto de controlo [DPMG08].

1.1 Motivação

As grelhas computacionais emergiram como solução para diversos projectos científicos, que necessitam de elevado poder computacional e manuseamento de dados [SS08]. As grelhas, permitem a resolução de problemas de maior complexidade e dimensão num menor tempo, facilitam a colaboração entre organizações e retiram melhor partido dos recursos partilhados. A adaptação de aplicações distribuídas/paralelas às grelhas computacionais, permitirá executar estas mesmas aplicações, em mais e melhores recursos, obtendo consequentemente melhores desempenhos.

À medida que o número de máquinas de um sistema distribuído aumenta, também aumenta a probabilidade de falha no sistema. O problema de perda de toda a computação feita até um dado momento, por falha de uma máquina, torna-se mais preocupante e inaceitável quando nos referimos a sistemas críticos ou a aplicações que possuem longos tempos de execução, como é o caso de inúmeras aplicações científicas que correm sobre as grelhas computacionais. Uma forma de resolver este problema consiste na implementação de um mecanismo de tolerância a faltas.

Uma das técnicas tradicionais de tolerância a faltas consiste em efectuar periodicamente uma cópia do estado aplicacional, que pode ser utilizada posteriormente para

restaurar o estado da aplicação. Esta técnica é designada por ponto de controlo/restauro (PCR).

Existem duas grandes abordagens quando se fala em PCR, nomeadamente, introdução de ponto de controlo ao nível do sistema operativo e ao nível aplicacional. Na primeira, o ponto de controlo fica a cargo exclusivamente do sistema operativo (SO), que efectua a recolha e o armazenamento em disco de todo o estado aplicacional, sendo geralmente implementada ao nível do *middleware*. Por sua vez, na segunda, fica a cargo do programador a introdução, no código fonte, das directivas responsáveis pelos processos de ponto de controlo e de restauro. A primeira abordagem tem como principal vantagem a obtenção de um mecanismo de tolerância a faltas, sem qualquer esforço por parte do programador, tendo como principais desvantagens a falta de portabilidade e o armazenamento de dados desnecessários. No que se refere à segunda abordagem, ainda que permita a obtenção de pontos de controlo portáveis e optimizados (o programador restringe a quantidade de dados a armazenar apenas à estritamente necessária), poderá revelar-se complexa e trabalhosa, obrigando a um grande esforço por parte do programador.

Em ambientes heterogéneos e de grandes dimensões, como é o caso das grelhas computacionais, portabilidade e optimização dos dados a armazenar são dois requisitos fundamentais, que devem ser garantidos pelo mecanismo de tolerância a faltas. Nas grelhas computacionais, uma aplicação deverá também lidar eficazmente com a volatilidade de recursos, dado que, durante a sua execução, os recursos alocados para a uma dada aplicação poderão diminuir/aumentar. Em alguns casos, a aplicação poderá mesmo ser obrigada a reiniciar num conjunto diferente de recursos. Uma forma de contornar este obstáculo, garantido simultaneamente portabilidade e optimização, é optar por uma abordagem do PCR ao nível aplicacional, adicionando também a capacidade de adaptação dinâmica da aplicação aos recursos disponíveis.

A utilização da programação orientada aos aspectos (POA) [GJA+97] permite aos programadores separar e organizar o código segundo funcionalidades secundárias, sem necessidade de alterações no código fonte. Em [GS09, SCM07, PRS10] utilizaram-se técnicas baseadas em POA, para o desenvolvimento de aplicações paralelas. Estas técnicas utilizam módulos amovíveis, que contêm as rotinas necessárias para a *paralelização* da aplicação, separando assim as rotinas do código base. Conceptualmente, a informação presente nesses módulos será suficiente para efectuar o PCR e a auto-adaptação ao nível da aplicação. Estas técnicas promovem intrinsecamente a separação das funcionalidades da aplicação das questões relacionadas com a tolerância a faltas e adaptação, facilitando a sua análise e evolução.

1.2 Objectivos

Esta tese de mestrado tem como objectivo desenvolver serviços de tolerância a faltas e de adaptação a recursos, para aplicações científicas que utilizam os recursos disponíveis nas grelhas computacionais.

Pretende-se a criação de uma nova forma de implementar serviços de ponto de controlo ao nível da aplicação, minimizando as alterações ao código base, mas ao mesmo tempo, tirando o máximo partido da semântica da aplicação.

Estes serviços deverão ser funcionais em múltiplos ambientes, tais como os existentes nas grelhas computacionais, garantindo também portabilidade, eficiência e transparência em relação à introdução do ponto de controlo.

Para atingir estes objectivos serão exploradas as potencialidades da POA. A utilização da POA permitirá a introdução de directivas (relacionadas com o PCR), implementadas sobre a forma de aspectos externos ao código base, sem necessidade de o alterar, permitindo assim uma maior modularização, estruturação e legibilidade do código.

A tese tem também como objectivos comparar o desempenho e o esforço de programação necessários nesta implementação com os das técnicas tradicionais, assim como documentar as dificuldades encontradas e possíveis entraves.

1.3 Estrutura da tese

Grande parte desta tese recai sobre o estudo e desenvolvimento de um mecanismo de PCR para sistemas de grelhas computacionais. No capítulo 2 são apresentadas a definição de mecanismo de PCR e as abordagens de implementação do mesmo, assim como os algoritmos e os problemas da sua implementação em diversos ambientes de programação. Por fim, são detalhadas técnicas de optimização dos ficheiros de ponto de controlo. No capítulo 3 referem-se as características da POA e casos descritos na literatura que tiram partido deste paradigma de programação. Por sua vez, no capítulo 4, apresentam-se os requisitos, as soluções e a implementação dos mecanismos de PCR e de adaptação dinâmica. No capítulo 5 são realizadas várias experiências e testes para diversos casos de estudo, de forma a validar os mecanismos de PCR e de adaptação dinâmica. Finalmente, no capítulo 6 são retratadas as ilações da utilização da POA na implementação dos mecanismos propostos, assim como as perspectivas futuras em relação ao trabalho realizado.

Capítulo 2

Ponto de controlo/restauro

Neste capítulo explora-se o estado da arte no que se refere a mecanismos de tolerância a faltas e de adaptação dinâmica. Na secção 2.1 apresenta-se o mecanismo de tolerância a faltas baseado em PCR, descrevendo de forma geral o seu funcionamento, as suas vantagens/desvantagens e as suas funcionalidades. Na secção 2.2 introduzem-se as abordagens ao mecanismo de PCR. Nas secções 2.3, 2.4 e 2.5 são detalhados os mecanismos de PCR descritos na literatura, para aplicações de memória distribuída, memória partilhada e memória partilhada distribuída, respectivamente. Na secção 2.6 discute-se a aplicação de mecanismos de PCR em grelhas computacionais. Por fim, na secção 2.7, apresentam-se algumas abordagens utilizadas para optimização dos mecanismos de PCR.

2.1 Introdução

O PCR é um mecanismo de tolerância a faltas que permite gravar o estado aplicacional, em intervalos regulares, para que este possa ser carregado em caso de interrupções forçadas, evitando assim repetir toda a execução da aplicação. Este mecanismo divide-se em dois processos distintos, o processo de ponto de controlo e o processo de restauro. O primeiro processo monitoriza a aplicação, efectuando periodicamente o armazenamento do estado aplicacional. Neste processo, denomina-se por ponto de controlo o ponto de execução na aplicação onde se procede à gravação dos dados em ficheiro. O segundo processo consiste no recomeço da execução a partir do último estado aplicacional correctamente gravado nos ficheiros de ponto de controlo (FPC).

O mecanismo de PCR, para além de proporcionar tolerância a faltas nos sistemas computacionais, também pode ser utilizado na gestão de recursos (permite parar trabalhos de longa duração e libertar CPU e memória), manutenção de *hardware*, depuração

[Lab, Kho] (restaurar a aplicação de um FPC e anexar, por exemplo, o gdb [Deb]) e migração [CLG05] (permite suspender a aplicação numa máquina e restaurar a execução da aplicação a partir dos FPC, em novas máquinas).

No que se refere à implementação de um mecanismo de PCR existem duas abordagens, o ponto de controlo a nível aplicacional (PCNA) e o ponto de controlo a nível do sistema operativo (PCNSO). Na primeira abordagem, a aplicação é responsável por armazenar os dados durante o ponto de controlo, enquanto na segunda o SO realiza uma cópia integral em disco do estado aplicacional. Existe, na literatura, quem faça a subdivisão do PCNSO em nível do *kernel* (quando existem modificações ao *kernel* [ZN01]) ou em nível do utilizador (sem modificações do *kernel* [TLM97]) [DLJ99]. Independentemente da abordagem utilizada, um mecanismo de PCR deverá dar resposta às seguintes problemáticas:

1. Qual o ponto de execução na aplicação onde deverá ser efectuado o ponto de controlo;
2. Que dados deverão ser gravados/recuperados;
3. Como gravar/restaurar a aplicação de uma forma globalmente consistente.

A forma como são solucionadas as problemáticas (1 a 3) irá depender da abordagem utilizada e do tipo de aplicação (sequencial, paralela baseada em memória partilhada, memória distribuída ou memória partilhada distribuída) onde é inserido o mecanismo de PCR.

A escolha do local do ponto de controlo irá não só influenciar a eficiência e utilidade do mecanismo de PCR, como também condicionar os dados que irão ser armazenados em disco. Enquanto no PCNSO, o ponto de controlo poderá ser realizado em qualquer local da aplicação, no PCNA o ponto de controlo só poderá ser realizado em pontos de execução específicos da aplicação. Idealmente, o ponto de controlo deverá ser colocado nos locais com maior rácio entre o custo adicional do processo de ponto de controlo e ganho com o processo de restauro.

Em relação ao conteúdo dos FPC, estes deverão armazenar o contexto dos processos, compostos pelos registos do processador, conteúdos de memória e contexto do SO. A memória é dividida, normalmente, em código, pilha (contendo as variáveis locais das funções/métodos ainda activos, e a ordem com que estas funções/métodos foram chamadas) e *heap* (que representa as estruturas de dados criadas dinamicamente). O código pode ser omitido no armazenamento, podendo ser recuperado do arquivo de origem. Em relação ao contexto SO, o seu armazenamento depende da abordagem utilizada pelo ponto de

controlo. Em programas com passagem de mensagens poderá ser necessário armazenar também o estado do canal de comunicação. Em sistemas de memória partilhada (SMP) deve-se armazenar o estado privado e partilhado de cada uma das linhas de execução (LE).

No PCNSO, dado que a aplicação pode ser interrompida em qualquer momento, para se proceder à GPC, é necessário guardar os registos do processador. Estes mesmos registos não podem ser armazenados na abordagem PCNA, uma vez que todo o procedimento é realizado pelas instruções existentes no código fonte. Uma forma de armazenar o contador do programa no PCNA passa pela utilização de uma variável que indica o ponto de execução onde se realizou a GPC. Durante o restauro, a aplicação retorna a sua execução a partir do valor guardado na variável, sendo os valores dos registos automaticamente criados durante a execução da aplicação. No PCNSO, o conteúdo dos FPC será uma cópia integral do estado aplicacional, durante o ponto de controlo, incluindo o estado dos canais de comunicação em programas de passagem de mensagens. Em contraste, no PCNA é possível reduzir o conteúdo dos FPC ao estritamente necessário para a correcta recuperação da aplicação.

Na abordagem PCNSO, a GPC é normalmente realizada por um processo ou uma LE independente. Para abordagens PCNA, existem vários algoritmos, descritos na literatura [KR00] e detalhados nas secções 2.3.1, 2.4.1 e 2.5 deste documento, para os diversos tipos de ambientes onde se inserem os mecanismos de PCR. A GPC em sistemas de memória distribuída (SMD) inclui a produção de um ponto de controlo para cada um dos processos a decorrer, que se denomina de ponto de controlo local. Para além dos pontos de controlo inerentes a cada processo, será necessário também capturar o estado de comunicação do programa. Dado que qualquer comunicação tem uma latência, algumas mensagens poderão estar a ser processadas pelo sistema no momento em que o estado de um processo é guardado. Em SMP o algoritmo de GPC deverá ser capaz de coordenar a gravação do estado privado e global das diferentes LE.

No PCNSO, o restauro aplicacional é deixado complemente a cargo do SO, enquanto no PCNA fica a cargo do programador o desenvolvimento de rotinas responsáveis por tal tarefa. Perante aplicações paralelas/distribuídas deverá ser garantido que todo o processo é restaurado num estado global consistente. Como tal, o programador deverá ter em conta que em SMP e SMD, podem-se considerar três tipos de estados a serem recriados durante o restauro, nomeadamente, aplicacional, escondido e de sincronização. Estado aplicacional refere-se às variáveis globais e locais, objectos da *heap* e pilha. O estado escondido está, entre outros, relacionado com as LE e com os processos no que diz respeito à criação, identificadores, pilha e conteúdo dos(as) mesmos(as). Basicamente, é todo o estado presente nas interfaces de programação paralela/distribuída, durante a

execução da aplicação. Por fim, o estado de sincronização é o estado dos mecanismos de sincronização activos durante o momento em que se dá o ponto de controlo, como, por exemplo, barreiras, fechos e regiões críticas.

Estes três tipos de estados podem ser recriados através de três formas distintas: -leitura do conteúdo dos pontos de controlo, como é o caso do estado aplicacional; -recriação, durante o processo de restauro, do estado que não pode ser acedido através da aplicação (executando as operações que criaram este estado na primeira instância), como é o caso dos fechos do OpenMP[arb05]; -recriação do mecanismo, no caso do estado que não poderá ser recriado nem por restauro, nem pela leitura do conteúdo do ponto de controlo, por exemplo, alguns tipos de objectos considerados estado escondido. Todo o estado que não pode ser recriado denomina-se de restrito e não é suportado pelo mecanismo de PCR ao nível aplicacional [BPS06].

2.2 Técnicas existentes

O mecanismo de PCR pode ser providenciado das seguintes formas:

1. A cargo do programador da aplicação [MT90];
2. Providenciado por uma biblioteca [CF05, PBKL95, DLJ99, PL94b];
3. Através do compilador [CF90, cJLkCFH91, RMG+10];
4. A cargo do SO [ZN01, TLM97].

A implementação de um mecanismo não está restrita à utilização de uma única das quatro técnicas a cima enumeradas. Na abordagem PCNA, o programador pode tirar partido de um compilador para determinar o local e o conteúdo do ponto de controlo [SS98a] (problemáticas 1 e 2 da subsecção 2.1), e o algoritmo de GPC poderá ser implementado na própria aplicação ou providenciado dentro de uma biblioteca. Relativamente ao restauro aplicacional, o programador poderá utilizar uma biblioteca que automatiza este processo. Na abordagem PCNSO é comum a utilização de bibliotecas que tiram partido do SO [PBKL95, ZN01, TLM97] para efectuar parte do PCR, nomeadamente, tirando partido do mecanismo de protecção de páginas para optimização dos FPC ou do controlador de sinais do SO para proceder à GPC, por exemplo.

2.2.1 A cargo do programador

Se optarmos por uma implementação do mecanismo de PCR utilizando a abordagem PCNA, deixando-o completamente a cargo do programador da aplicação, este fica

responsável por inserir sobre o código base as directivas necessárias para que a própria aplicação efectue o PCR. É uma técnica flexível que permite obter uma elevada eficiência, uma vez que o programador pode tirar partido da semântica da aplicação e reduzir ao máximo o custo da execução do PCR. No entanto, esta técnica tem como desvantagens a demora e a complexidade da sua implementação, assim como o facto da aplicação poder introduzir fortes restrições tanto no local onde poderá ser colocado o ponto de controlo, como na frequência com que este ocorre. Dado que muitas vezes o ponto de controlo só pode ser tomado em pontos específicos na aplicação, se o tempo entre os diferentes pontos de controlo for muito elevado, podemos estar a restringir a utilidade do PCR. Esta tarefa requer que o programador detenha um bom conhecimento sobre a aplicação e que realize alterações ao nível do código base.

2.2.2 Providenciado por uma biblioteca

Esta técnica consiste na utilização de uma biblioteca, de modo a auxiliar o programador, automatizando parte do PCR. O programador necessitará de realizar, no máximo, três chamadas à biblioteca: -indicação do local do ponto de controlo; -activação do processo de restauro; -especificação do conteúdo do ponto de controlo. Com recurso a uma biblioteca para PCR, ganha-se flexibilidade (permite ao programador um maior controlo sobre a frequência e colocação do GPC) e a possibilidade de obtenção de FPC portáveis (restauro da aplicação em máquinas com diferentes arquitecturas) e optimizados (o programador pode reduzir os dados do ponto de controlo ao estritamente necessário). No entanto, apresenta como desvantagens a possibilidade de intrusividade e o facto de ficar à responsabilidade do programador a análise do melhor local para colocação do ponto de controlo e do conteúdo do mesmo, obrigando-o a possuir um bom conhecimento do funcionamento interno da aplicação.

Ickp [PL94b], segundo James S. Plank, foi a primeira biblioteca a permitir PCR num sistema paralelo de memória distribuída, implementando o mecanismo de PCR utilizando a abordagem PCNSO sem modificações do *kernel*. Esta biblioteca permite reduzir o espaço ocupado pelos FPC, através da utilização de um algoritmo de compressão. Assim como a biblioteca Ickp, CPPC [RMG+10] também utiliza compressão dos FPC, no entanto ao contrário da Ickp, CPPC permite a obtenção de FPC portáveis. Tal é possível, porque CPPC implementa o mecanismo de PCR ao nível aplicacional.

Tal como Ickp, Libckpt [PBKL95] é uma biblioteca que implementa PCNSO sem modificações do *kernel*. Libckpt utiliza técnicas de optimização dos FPC, nomeadamente, a implementação de ponto de controlo cópia na escrita e ponto de controlo incremental (descritas na secção 2.7). A Libckpt implementa o ponto de controlo incremental ao

nível do SO, tirando partido do mecanismo de protecção de páginas do mesmo para identificação de porções de dados inalteradas entre pontos de controlo, armazenando em disco apenas os dados alterados. Para além destas técnicas, a Libckpt permite ao programador efectuar exclusão/inclusão de memória [PCL+99] e definir os locais no código onde serão colocados os pontos de controlo. Ao contrário da Libckpt, que não providencia suporte para programas com múltiplas LE, a ferramenta desenvolvida em [CF05] (expansão da biblioteca *Ckpt* [Zan]) suporta PCR em aplicações MPI [SOHL+96] paralelas.

2.2.3 Utilizando um compilador

Nesta abordagem o compilador ajuda a obter transparência, dispensando o programador de tarefas de análise de comunicações e fluxo de dados. A ideia desta técnica é explorar o conhecimento do compilador de forma a colocar o ponto de controlo no melhor local, especificar e optimizar os dados a serem armazenados. O compilador permite: - detectar as secções de código com maiores tempos de execução, para potenciais locais de colocação do ponto de controlo, maximizando o ganho do processo de restauro; - excluir as áreas de memória irrelevantes, reduzindo assim o tamanho dos FPC e tornando o PCR mais eficiente; - a utilização da técnica de exclusão de memória para optimização dos FPC [PBK95]. Uma das desvantagens desta técnica é o facto de ser de difícil utilização em aplicações paralelas/distribuídas, que comunicam através de passagem de mensagens, dada a dificuldade do compilador em determinar o estado dos canais de comunicação no ponto de controlo [SS98a].

Porch [Str98] é um exemplo de um compilador com suporte para PCR, que transforma programas em C para programas em C com suporte para pontos de controlo portáveis, com a desvantagem de não suportar aplicações paralelas. Outro exemplo de uma ferramenta que utiliza um compilador para auxiliar no PCR é o CPPC [RMG+10] que, ao contrário de Porch, utiliza em conjunto com o compilador uma biblioteca, permitindo assim suporte para aplicações paralelas. Nesta ferramenta, o compilador é utilizado para analisar e efectuar as alterações necessárias ao nível do código. O CPPC obriga a que os pontos de controlo sejam realizados em locais onde não existam mensagens nos canais de comunicação, locais estes designados por pontos seguros. A identificação dos pontos seguros fica a cargo do compilador, que procede a uma análise estática do estado das comunicações. O compilador tem também a tarefa de inserir rotinas na aplicação para a obtenção de pontos de controlo. Para a inserção destas rotinas, o compilador analisa todas as secções de código com elevados tempos de execução, atribuindo um valor a cada uma das secções. Em seguida, o compilador lista todas as secções, e sobre estas secções

é aplicada uma heurística [RMG+10] e é verificada a existência de pontos seguros, nos quais serão colocadas as rotinas de GPC.

2.2.4 A cargo do sistema operativo

PCNSO consiste na obtenção de recuperação automática utilizando o SO, sem qualquer esforço por parte do programador. A maior parte das ferramentas como *CRAK* [ZN01], que utilizam o PCNSO, procedem à cópia integral do estado aplicacional durante o processo de ponto de controlo. Esta abordagem não tira partido da semântica da aplicação, ocorrendo assim desperdício de espaço, uma vez que o SO irá armazenar todo o estado aplicacional, incluindo dados que poderiam ser omitidos sem afectar a correcta recuperação da aplicação, tornando o PCR menos eficiente. O desperdício de espaço é mais acentuado em sistemas de passagem de mensagens, onde o estado de todas as mensagens e o conteúdo de todos os processos em execução são armazenados na GPC, podendo ocorrer replicação de informação. Por exemplo, para as aplicações de desdobramento proteico na máquina *IBM Blue Gene* [Gen], utilizando PCNA o tamanho dos FPC era de alguns *Megabytes*, enquanto utilizando PCNSO era de alguns *Terabytes* [BMP+04]. A falta de portabilidade é outra desvantagem, uma vez que os FPC ficam condicionados à arquitectura que os originou, impedindo o restauro da aplicação utilizando estes FPC, em máquinas com diferentes arquitecturas. A falta de portabilidade e o tamanho que os FPC gerados podem atingir, tornam esta abordagem impraticável em sistemas heterogéneos de grandes dimensões, como é o caso das grelhas computacionais. Para as aplicações na maior parte das plataformas, tais como *IBM Blue Gene* e máquinas ASCI o mecanismo de PCR é por definição ao nível aplicacional [BMP+04].

CRAK [ZN01] é uma ferramenta que implementa PCR ao nível do *kernel* do linux, sem a necessidade de qualquer modificação ao código fonte da aplicação. Esta ferramenta suporta processos paralelos e a migração de aplicações entre ambientes linux, utilizando como pressuposto que o ambiente onde a ferramenta está inserida é homogéneo.

Condor [TLM97] também implementa PCNSO, mas contrariamente a CRAK esta implementação não é ao nível do *kernel* do linux. Em vez disto, Condor providencia uma biblioteca que contém um controlador de sinais para *SIGTSTP* (sinal em UNIX que indica a uma aplicação para parar a sua execução temporariamente). Sempre que for o momento de efectuar a GPC, Condor envia um sinal ao processo ligado à biblioteca e o controlador de sinais procede à gravação do estado aplicacional do processo em ficheiro. Uma das grandes vantagens do Condor é o facto de não obrigar a alterações ao nível do código fonte, requerendo apenas que a biblioteca seja acoplada à aplicação, sem a necessidade de recompilação. Por outro lado, esta mesma vantagem restringe a

utilização do Condor a aplicações onde os ficheiros objectos estejam disponíveis. Para além disto, Condor apresenta a desvantagem comum às ferramentas implementadas ao nível do SO, nomeadamente, falta de portabilidade.

2.2.5 Comparação entre abordagens

Analisando as duas grandes abordagens de implementação de PCR (PCNA e PCNSO), chega-se à conclusão que apesar de mais trabalhosa e complexa, a implementação de PCNA apresenta FPC de menores dimensões e um mecanismo de PCR mais eficiente. Segundo James Kasdorf, director de projectos no centro de super-computação em São Petersburgo, *"Ponto de controlo ao nível do SO é útil mas muito custoso, dado que o SO não tem conhecimento dos dados que a aplicação realmente necessita para o correcto restauro, portanto, grava tudo cegamente. Se imaginarmos uma máquina com os mesmos dados replicados por 512 processadores, SO não tendo conhecimento disto, grava tudo, e no fim acabas com centenas de cópias desnecessárias de dados, código do programa e bibliotecas do sistema."*.

Uma das grandes desvantagens da PCNA, para além do esforço requerido por parte do programador, são as restrições na colocação e frequência do ponto de controlo. Nesta abordagem, ao contrário do que acontece com PCNSO, os pontos de controlo só podem ser realizados em pontos específicos da aplicação, afectando consequentemente a frequência da GPC. No caso do PCNSO, a GPC poderá ser realizada em qualquer momento, ocorrendo mediante uma frequência baseada numa medida temporal, permitindo portanto realizar a GPC em intervalos regulares de tempo. Outra desvantagem do PCNA é a necessidade do código fonte da aplicação estar disponível, o que nem sempre é possível.

Quando se implementa PCNA, não é estritamente obrigatório optar apenas por uma das técnicas (programador, biblioteca ou compilador), pode-se (e é aconselhável) utilizar uma implementação que tire partido de mais do que uma das técnicas, como é o caso da ferramenta CPPC. O mecanismo de PCR a implementar deverá tirar partido do compilador para a identificação do local e conteúdo do ponto de controlo, e da biblioteca, que para além de automatizar os processos de ponto de controlo e restauro, permite a criação de FPC portáveis e optimizados.

A tabela 2.1 apresenta, de forma resumida, algumas vantagens e desvantagens de cada uma das técnicas de implementação de PCR, anteriormente descritas.

TABELA 2.1: Tabela de vantagens/desvantagens das diferentes abordagens de PCR.

Abordagens	Vantagens	Desvantagens
Programador	- Elevada eficiência; - Optimização dos FPC; - Portabilidade dos FPC; - Flexibilidade.	- Complexidade da implementação; - Maior esforço do programador; - Exige conhecimento sobre a aplicação.
Biblioteca	- Menor esforço do programador; - Portabilidade dos FPC; - Optimização dos FPC.	- Possibilidade de intrusividade; - Identificação do local e conteúdo do ponto de controlo.
Compilador	- Menor esforço do programador; - Optimização dos FPC.	- Difícil utilização em aplicações paralelas/distribuídas.
SO	- Mecanismo de PCR totalmente automatizado; - A GPC poderá ser realizada a qualquer momento.	- Armazenamento desnecessário de dados; - FPC não portáveis; - Aumento do custo adicional do PCR.

2.3 Memória distribuída

Um sistema de memória distribuída é composto por múltiplos processadores, que possuem a sua própria memória local. A memória local de um processador não está directamente acessível a outro processador, portanto, não existe memória partilhada entre todos os processadores. Uma vez que cada processador possui a sua própria memória local, estes operam de forma independente entre si e portanto alterações na sua memória local não têm qualquer efeito sobre a memória dos outros.

Nas aplicações em sistemas de memória distribuída, sempre que um processo pretende aceder a dados presentes noutro processo necessita de comunicar com este, ficando a cargo do programador explicitar quando e como ocorrem estas comunicações entre processos. Uma das técnicas mais utilizadas para troca de dados entre processos, é através de passagem de mensagens, sendo a MPI uma das bibliotecas de passagem de mensagens mais utilizadas [WPH03].

2.3.1 Algoritmos

Segundo a literatura, algoritmos de ponto de controlo para programas de passagem de mensagens podem ser:

- Coordenados [CS98, CL85], também designados por síncronos [JLM08];
- Descoordenados [PL05], também conhecidos por assíncronos [JLM08] ou independentes [KT88, KR00];

- Quase-síncronos [MS99, MJYS08], também qualificados como induzidos à comunicação [TKW98].

Na figura 2.1 são apresentadas, de forma detalhada, as subclasses destes três tipos principais de algoritmos.

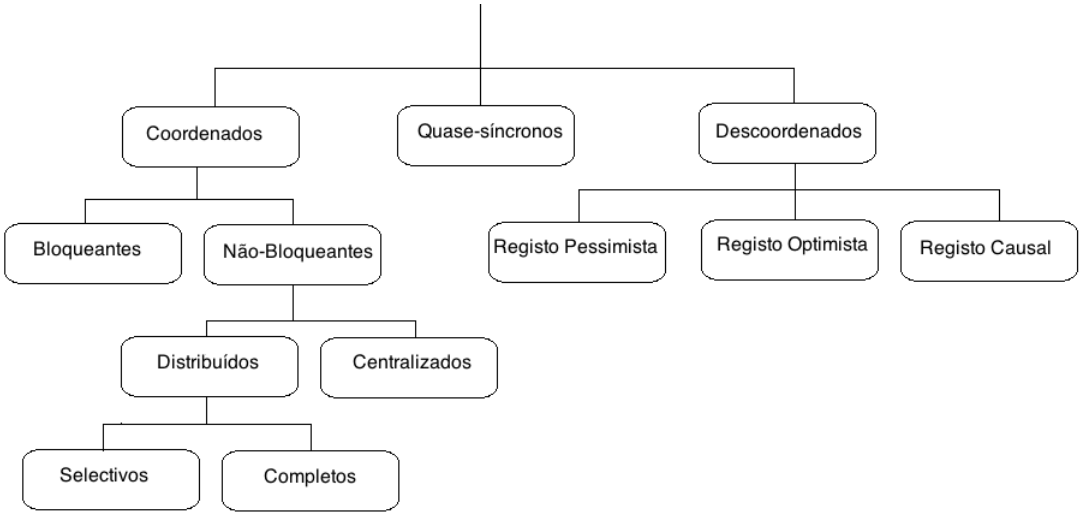

FIGURA 2.1: Classificação dos algoritmos de ponto de controlo em programas de passagem de mensagens.

2.3.1.1 Descoordenados

Nos algoritmos descoordenados, o processo toma o seu ponto de controlo de forma autónoma, não necessitando de esperar pelos restantes processos. Consequentemente, forma-se, durante o processo de restauro, uma linha de recuperação válida a partir dos pontos de controlo locais criados independentemente, o que torna o processo de recuperação mais complicado. A necessidade em estabelecer um estado consistente, durante o processo de restauro, poderá forçar a que os processos retrocedam para pontos de controlo antigos, o que pode fazer com que outros processos, por sua vez, retrocedam para pontos de controlo ainda mais antigos. No caso mais extremo o estado consistente encontrado é o estado inicial (efeito dominó).

Em aplicações determinísticas, o efeito dominó pode ser evitado combinando o algoritmo descoordenado com o registo de mensagens, limitando o número de processos que retrocedem em caso de falha [AM98], com a desvantagem de ser necessário efectuar o registo de todas as mensagens. No entanto, para aplicações com a propriedade de envio-determinista, comum a muitas aplicações MPI de alta performance [CGS10], foi proposto recentemente em [GRB+11] um novo protocolo descoordenado que apenas requer o registo de um pequeno subconjunto de mensagens, para garantir a eliminação do efeito

dominó. Uma aplicação possui a propriedade de envio-determinista quando dado um conjunto de parâmetros de entrada, a sequência de emissão de mensagens, para qualquer processo, é a mesma em todas as execuções correctas [GRB+11].

Os algoritmos descoordenados têm como principal vantagem a eliminação do custo acrescido de coordenação dos processos, durante o processo de ponto de controlo. Em vez disto, os processos serão coordenados apenas durante o processo de restauro, de forma a obter-se uma linha de recuperação consistente. Uma desvantagem é a obrigatoriedade da utilização de múltiplos FPC em disco, ocupando assim mais espaço.

O PCR, em conjunto com registos de mensagens, providencia um mecanismo de tolerância a faltas eficiente em ambientes de passagem de mensagens. Os processos efectuam o ponto de controlo de forma esporádica, enquanto todas as mensagens recebidas pelos processos são gravadas em disco. O registo de mensagens permite dispensar qualquer tipo de processo de coordenação entre os pontos de controlo locais a cada processo. Alvisi e Marzullo [AM95] classificam os protocolos de registo de mensagens em três tipos, designadamente, registo pessimista [JZ87], registo optimista [DDGG95] e registo causal [AM98]. No primeiro, o registo de mensagens é realizado de forma síncrona à medida que são recebidas as mensagens, enquanto no segundo é realizado de forma assíncrona. O processo de sincronização no primeiro protocolo pode ser efectuado bloqueando o destinatário da mensagem até que esta seja registada, ou bloqueando o destinatário caso este tente enviar uma nova mensagem, antes da mensagem recebida estar registada. No protocolo assíncrono, o registo das mensagens é efectuado mais tarde, permitindo ao destinatário continuar normalmente a sua execução. O registo de mensagens neste protocolo pode ser realizado numa única operação que agrupa as várias mensagens e grava-as em disco, diminuindo o custo adicional de múltiplos acessos ao disco. Em caso de falha, no primeiro protocolo, um processo é restaurado a partir do seu último ponto de controlo e das mensagens originalmente recebidas até ao carregamento deste ponto de controlo. As mensagens são processadas na mesma ordem pela qual foram recebidas, antes da ocorrência de falha. No entanto, as mensagens enviadas, por este processo, são ignoradas durante o restauro da aplicação, para impedir a ocorrência de mensagens duplicadas. O processo de restauro no segundo protocolo é mais complicado do que no primeiro, uma vez que é necessário encontrar uma linha de recuperação válida a partir da combinação dos estados dos processos, de forma a que nenhum destes estados gere mensagens órfãs (ver secção 2.3.3). O processo de restauro pode não ser bem sucedido devido à falta de linhas de recuperação válidas (efeito dominó).

O protocolo de registo pessimista providencia uma recuperação rápida, tendo como desvantagem o custo acrescido do processo de sincronização. Já o protocolo optimista, comparativamente ao pessimista, diminui o custo acrescido do mecanismo de PCR, ao

eliminar o processo de sincronização, mas apresenta como desvantagem um processo de recuperação mais lento e complicado, passível da ocorrência do efeito dominó.

O protocolo causal é um híbrido das abordagens pessimista e optimista, possuindo portanto as vantagens de ambas, permitindo assim diminuir o custo adicional do registo de mensagens, obtendo-se na mesma uma rápida recuperação do estado aplicacional. Durante o processo de ponto de controlo, cada processo envia informação adicional, juntamente com cada mensagem a ser enviada a outro processo, de forma a evitar estados de processos inconsistentes durante o processo de restauro. A informação adicional inclui o identificador do processo emissor e receptor da mensagem, os números das sequências de envio e recepção atribuídos à mensagem, por parte do emissor e receptor, respectivamente. Desta forma, este protocolo restringe a recuperação de qualquer processo ao mais recente ponto de controlo armazenado, reduzindo o número de pontos de controlo locais mantidos em disco [AMHY02]. Algumas implementações param a execução dos processos activos (que não sofreram falha) durante o processo de restauro, até que a recuperação aplicacional esteja terminada.

Em [AMHY02] é apresentado um sistema que combina registo de mensagens casual com ponto de controlo assíncrono. Este sistema não obriga à imobilização dos processos activos durante o restauro, e necessita apenas de manter o último ponto de controlo de cada processo, dispensando o uso de um mecanismo colector de lixo. Permite também que cada processo seja responsável pelo seu próprio restauro.

2.3.1.2 Coordenados

Nos algoritmos coordenados, os processos cooperam para guardar um conjunto de pontos de controlo de forma coordenada. Em caso de falha, os processos são restaurados a partir do conjunto mais recente de pontos de controlo, composto pelo último ponto de controlo de cada processo. Para recuperar de uma falha, o sistema reiniciará a sua execução a partir de um estado global consistente e sem erros, gerado pelos pontos de controlo dos processos envolvidos. Estes algoritmos têm como vantagens:

- não ser necessário recorrer a registo de mensagens;
- recuperação de falha mais simples do que nos descoordenados;
- não sofrer o efeito dominó;
- não são mantidos, em disco, múltiplos pontos de controlo;
- de implementação mais simples que os descoordenados.

Uma das desvantagens deste tipo de algoritmo passa pela existência de sobrecarga intrínseca ao processo de coordenação. Quando se faz ponto de controlo de um processo, são verificados todos os processos relevantes em relação a este, para se fazer ponto de controlo dos mesmos. A consequência disto é a elevada sobrecarga associada ao processo de ponto de controlo [CS98]. Outra desvantagem desta abordagem é o facto de quando ocorrida uma falha de sistema, ser necessário que todos os processos reiniciem a sua execução.

Com o objectivo de minimizar o número de mensagens de controlo e o número de pontos de controlo durante o processo de ponto de controlo, surgiram os algoritmos bloqueantes [KT86]. Os algoritmos bloqueantes sincronizam todos os processos durante a GPC, até que a linha de recuperação esteja concluída. Uma consequência negativa é a degradação da performance do sistema [BLL90, EJZ92], causada pelo elevado custo adicional do bloqueio dos processos. Em [LM10], com o objectivo de reduzir o custo do bloqueio dos processos, introduziu-se uma abordagem para algoritmos coordenados bloqueantes utilizando dois níveis de ponto de controlo. O primeiro nível trata do ponto de controlo local, ou seja, cada processo grava o ponto de controlo num disco local. Por sua vez, o segundo nível é referente ao ponto de controlo global, em que cada um dos processos envia o seu ponto de controlo para um disco partilhado. No caso de uma falha transitória (falha ao nível do *software*), o nó de computação continua acessível, e portanto o processo é restaurado utilizando o ponto de controlo local. Por outro lado, em caso de falha permanente (falha ao nível do *hardware*), o processo é restaurado num novo nó de computação, utilizando o ponto de controlo gravado no disco partilhado. Em sistemas onde a probabilidade de falha transitória é superior à de falha permanente, os pontos de controlo globais são realizados menos frequentemente do que os locais, sendo que os pontos de controlo locais são realizados periodicamente, em intervalos óptimos baseados na taxa de falha. A grande vantagem desta abordagem é a utilização de pontos de controlo locais, que não obrigam ao bloqueio e sincronização de processos, em detrimento dos pontos de controlo globais. Os resultados [LM10] demostram que esta abordagem reduz significativamente o tempo de execução quando comparada com o algoritmo tradicional de ponto de controlo coordenado bloqueante.

Os algoritmos de ponto de controlo coordenados não bloqueantes [GR05], como por exemplo *Chandy-Lamport* [CL85] (CL), são preferencialmente utilizados em detrimento da abordagem bloqueante [PL05] pelo facto de terem uma baixa sobrecarga. Nestes algoritmos não existe a necessidade de bloquear a execução dos processos, durante o ponto de controlo, permitindo ao processo retomar a sua execução logo após ter efectuado o seu ponto de controlo local. Segundo S. Kalaiselvi e R. Rajaraman [KR00], pode-se subdividir os algoritmos coordenados não bloqueantes em centralizados e distribuídos. Nos

algoritmos centralizados, como é o caso do CL, fica a cargo de um processo central a responsabilidade de dar início ao processo de GPC e a coordenação dos restantes processos. A desvantagem destes algoritmos é que os restantes processos ficam sempre dependentes do processo central para efectuar a GPC. No caso dos algoritmos distribuídos, os processos podem realizar a GPC por si mesmo, sem que seja necessária a activação por parte de um processo central.

Os algoritmos coordenados não bloqueantes distribuídos podem ser completos [LY87] ou selectivos [KT86]. Na primeira vertente enunciada, todos os processos têm que participar em todas as GPC. Já no que concerne à segunda, apenas um grupo de processos dependentes entre si é que participa no processo de GPC, realizando para tal, durante a execução aplicacional, a monitorização dinâmica das dependências entre processos. Durante o processo de restauro, o grupo de processos dependentes entre si é restaurado. A grande vantagem dos selectivos é o facto de não ser necessário, em caso de erro da aplicação, restaurar todos os processos, tal como acontece nos completos, onde inclusive os processos desnecessários, à correcta recuperação do estado consistente da aplicação, são restaurados. Por outro lado, a desvantagem dos selectivos é a dificuldade do processo de coordenação dos processos que deverão realizar a GPC, porque para tal é necessário determinar as dependências entre processos e resolver conflitos quando múltiplos pedidos de GPC chegam a um processo[KR00].

2.3.1.3 Quase-síncronos

Em programas de passagem de mensagens, a principal diferença entre algoritmos está relacionada com o processo de coordenação, que pode ocorrer durante a execução da aplicação ou durante o processo de recuperação. Os algoritmos coordenados coordenam os processos e formam uma linha de recuperação em plena execução, enquanto os descoordenados fazem-no durante a recuperação da aplicação. Uma abordagem mais recente é a realização do processo de coordenação de forma parcial, designada como quase-síncrona.

Pode-se reduzir o número de pontos de controlo inúteis nos algoritmos descoordenados, fazendo-se para tal pontos de controlo induzidos à comunicação, ao invés de pontos de controlo descoordenados. Este tipo de algoritmo é designado na literatura de quase-síncrono ou ponto de controlo induzido à comunicação. Por pontos de controlo inúteis, entendem-se todos os pontos de controlo locais que não poderão fazer parte do ponto de controlo global consistente.

Nos algoritmos quase-síncronos existem dois tipos de pontos de controlo: - pontos de controlo básicos, realizados de forma independente pelos processos. - pontos de controlo forçados, induzidos à comunicação. Nos algoritmos descoordenados podem existir alguns

pontos de controlo inúteis, em casos mais extremos onde todos os pontos de controlo são inúteis, ocorre o efeito dominó. A ideia principal dos algoritmos quase-síncronos é reduzir estes pontos de controlo, utilizando para tal pontos de controlo forçados, que são realizados mediante alguns padrões de mensagens, ou mediante conhecimento anteriormente adquirido sobre as dependências entre pontos de controlo dos processos. Nesta abordagem é enviada informação de controlo, juntamente com as mensagens da aplicação. Assim, quando um processo recebe uma mensagem a informação de controlo é analisada e o algoritmo de PCR decide a realização, ou não, de uma GPC forçada. A GPC forçada é realizada mediante alguns padrões de ponto de controlo e mensagens, deduzidos por Manivanna e Singhal, baseando-se nos conceitos de caminhos de ziguezague e ciclos-Z [NX95]. Manivanna e Singhal dividiram os padrões de ponto de controlo e mensagens, e respectivos protocolos, em quatro classes distintas: *Partially Z-Cycle Free, Z-Cycle Free, Z-Path Free* e *Strictly Z-Path Free*. Em [MS99] são detalhadas em pormenor estas quatro classes.

Estudos realizados em [AER+99] consideram que os algoritmos quase-síncronos podem ter na prática comportamentos imprevisíveis. O espaço necessário em disco, o custo adicional na performance e o número de pontos de controlo forçados dependem fortemente do número de processos e dos padrões de comunicação da aplicação.

2.3.1.4 Chandy e Lamport

Chandy-Lamport [CL85] (CL) é um algoritmo frequentemente utilizado para implementação de tolerância a faltas em sistemas de passagem de mensagens [PL05]. A maior parte dos algoritmos propostos para sistemas de passagem de mensagens utilizam como base o algoritmo CL, derivando de modificações em algumas etapas ou alterações de alguns pressupostos impostos pelo algoritmo [KR00]. O algoritmo CL baseia-se nos seguintes pressupostos:

- Todas as mensagens chegam intactas e uma única vez;
- O canal de comunicação é unidirecional e ordenado como uma fila (o primeiro a entrar é o primeiro a sair);
- Existe um caminho de comunicação entre dois processos;
- O estado global do sistema inclui o estado local dos processos e o estado dos canais de comunicação.

No algoritmo CL, o estado global é obtido através da coordenação de todos os processos, gravando-se o estado dos canais de comunicação durante a GPC. A GPC é iniciada por um único processo, considerado central.

Sejam *Saida_p* e *Entrada_p* os números dos canais de comunicação de envio e recepção de mensagens, respectivamente, do processo *p*. Então, cada processo *p* [PL05]:

1. Procede à GPC local;
2. Envia um marcador, ao longo do canal de saída i, de i = 1 até *Saida_p*;
3. Continua a execução;
4. Grava as mensagens do canal de recepção i até à recepção de um marcador, ao longo deste canal, de i = 1 até *Entrada_p*.

Quando todos os processos tiverem terminado os passos a cima, está formada a linha de recuperação. No passo 2, o envio do marcador tem como objectivo avisar os restantes processos que deverão iniciar também a GPC. Já no que se refere ao passo 4, o marcador irá funcionar como separador entre mensagens, desde a GPC anterior até à actual.

Existem várias ferramentas de PCR para MPI que utilizam, sem modificações, o algoritmo CL ao nível do sistema operativo [Ste96, AF99]. O facto do algoritmo CL pressupor que a topologia de comunicação é fixa, que os canais de comunicações são do tipo FIFO e que a GPC poderá ocorrer em qualquer altura, faz com que não seja válido para ponto de controlo ao nível aplicacional para programas MPI [BMPS03].

2.3.2 Problemas

Quando se pretende implementar um sistema de tolerância a faltas para MPI, utilizando uma abordagem ao nível aplicacional, não é suficiente a realização de ponto de controlo para cada um dos processos individuais. É, então, necessário ter em consideração as seguintes problemáticas:

- O estado do sistema de passagem de mensagens;
- Canal de comunicação não-FIFO;
- Funções de comunicação colectiva;
- Estado escondido.

2.3.2.1 Estado do sistema de passagem de mensagens

Existem dois tipos de mensagens que tornam a obtenção de PCR mais complexa, as mensagens em-trânsito e órfãs, ilustradas na figura 2.2 [PL05].

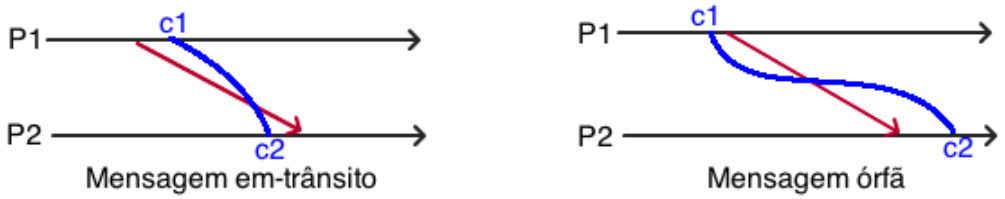

FIGURA 2.2: Mensagens em-trânsito e órfãs.

As linhas P1 e P2 representam os processos 1 e 2, respectivamente, e C1 e C2 os seus pontos de controlo locais. A linha azul representa a linha de recuperação composta por todos os pontos de controlo locais, enquanto a seta vermelha representa o envio e recepção da mensagem entre os processos.

Na figura 2.2, onde se representam as mensagens em-trânsito (lado esquerdo), P1 realiza a GPC depois de ter efectuado o envio de uma mensagem, mensagem esta que é recebida pelo P2, após ter realizado a sua GPC. Em caso de falha do sistema, o processo de restauro é activado e é carregada a linha de recuperação composta pelos pontos de controlo C1 e C2, iniciando a execução da aplicação deste mesmo local. Assim, como não é gravado o estado de comunicação e como C1 ocorre após o envio da mensagem, P1 irá assumir que já enviou a mensagem para P2, não repetindo o envio. Por outro lado, o ponto local C2 é feito antes do processo P2 ter recebido a mensagem, assim P2 irá bloquear enquanto espera a mensagem que nunca será reenviada [PL05].

Relativamente às mensagens órfãs, P1 envia a mensagem depois do seu ponto de controlo local C1, enquanto P2 recebe a mensagem antes do ponto de controlo local C2. Ao reiniciar a aplicação devido a falha, P1 irá enviar novamente a mensagem para P2 (mensagem esta que já foi recebida e guardada por P2). A duplicação da mensagem poderá quebrar a semântica da comunicação [PL05].

Para se criar uma linha de recuperação válida, esta deve ser recuperável (livre de mensagens em-trânsito) e consistente (livre de mensagens órfãs).

Uma solução, para o problema de mensagens em-trânsito e órfãs, passa por deixar a cargo do programador a sincronização dos pontos de controlo nos diferentes processos, utilizando, por exemplo, uma barreira. O programador teria de garantir que tais pontos eram realizados em locais onde não existem mensagens em aberto entre processos. Este

tipo de solução é utilizada pelos algoritmos coordenados bloqueantes. Este tipo de algoritmo é suficiente para várias aplicações paralelas baseadas no modelo *Bulk Synchronous Parallel* (BSP) [HMS+98]. No entanto, existem aplicações paralelas onde são poucos, ou até mesmo nenhuns, os pontos na aplicação onde existem garantias da não existência de mensagens abertas entre processos, como, por exemplo, aplicações que utilizam uma abordagem mais assíncrona e de auto-sincronização para comunicação, em detrimento do modelo BSP. Para este tipo de aplicações paralelas é necessária a utilização de algoritmos de ponto de controlo coordenados não bloqueantes [BFM+06].

2.3.2.2 Canal de comunicação não-FIFO

Tal como CL, existem vários algoritmos que assumem que a comunicação entre processos respeita a propriedade FIFO, no entanto, tal propriedade não é garantida numa abordagem PCNA que utiliza a biblioteca por defeito da MPI[BMPS03]. Como tal, um sistema de PCR ao nível aplicacional deverá suportar aplicações com envio de mensagens não-FIFO . O algoritmo CL assume a propriedade FIFO, para que, o marcador funcione como um separador entre as mensagens à volta do ponto de controlo. Vejamos a figura 2.3:

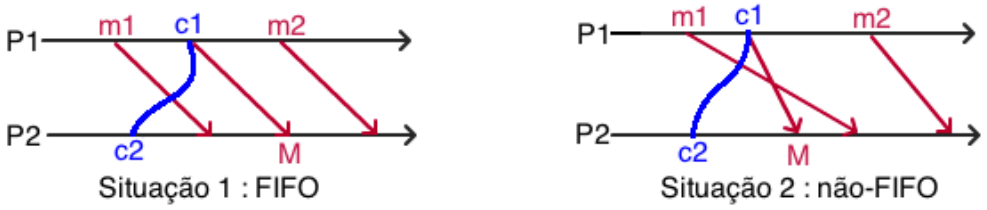

FIGURA 2.3: Canais de comunicação.

A notação utilizada na figura 2.3 é a mesma da figura 2.2, com o acréscimo de m1, m2 e M que representam as mensagens 1, 2 e o marcador, respectivamente.

Na situação 1, o canal de comunicação respeita a propriedade FIFO, e portanto a ordem de recepção das mensagens é a mesma de envio, enquanto na situação 2 o canal de comunicação é do tipo não-FIFO. Na primeira situação, m1 é recebida logo após c2, e como tal CL regista m1, como sendo uma mensagem em-trânsito. Na segunda situação, a recepção do marcador, por parte de P2, acontece antes da recepção m1, fazendo com que CL não registe m1, tornando, assim, inválida a linha de recuperação.

Abordagens como etiquetagem de mensagens [BMPS03, PBK95] ou registo de eventos [PL05] permitem implementar variações do algoritmo CL, que suportam comunicações não-FIFO. A primeira abordagem substitui a utilização do marcador pelo envio de um cabeçalho juntamente com as mensagens, contendo o número do intervalo do ponto de

controlo de onde as mensagens são originadas. Na segunda abordagem, cada processo guarda um registo dos eventos de envio e recepção de mensagens, para que no momento da realização da GPC, os processos troquem os seus registos com os processos que enviaram ou receberam mensagens. Estes registos são combinados entre si, de forma a determinar a existência de mensagens em-trânsito ou órfãs.

2.3.2.3 Funções de comunicação colectiva

MPI suporta um grupo de funções colectivas, algumas das quais envolvem transferência de dados entre múltiplos processos. O problema de tais funções para o PCR acontece quando alguns processos realizam uma chamada a uma comunicação colectiva antes de efectuarem a GPC, enquanto outros processos realizam esta chamada depois da GPC. É necessário garantir que no processo de restauro, os processos que re-executam as chamadas não entrem nem em interbloqueio, nem recebem informação incorrecta. Para além disto, é preciso preservar, no restauro, a semântica de sincronização fornecida pela MPI [BMPS03, SBF+04]. Vejamos o seguinte exemplo: os processos 1 e 2 executam uma chamada à função *MPI_ Allreduce* antes do ponto de controlo e o processo 0 executa a chamada depois de efectuar o ponto de controlo. Dado que a linha de recuperação é formada pelos pontos de controlo locais de cada um dos processos (0,1 e 2), durante o processo de restauro, os processos 1 e 2 irão re-executar a função *MPI_ Allreduce*, mas o processo 0 não. Isto resulta numa situação de interbloqueio dos processos 1 e 2 em relação ao processo 0. Uma solução, proposta em [BMPS03], é que durante o ponto de controlo o resultado da chamada à função *MPI_ Allreduce*, por parte dos processos 1 e 2, seja armazenado. Durante o restauro, em vez dos processos (1 e 2) re-executarem a chamada à função *MPI_ Allreduce*, irão ler o resultado anteriormente armazenado, sendo que o processo 0 não re-executa a função *MPI_ Allreduce*. É necessário, portanto, definir quando é que os resultados das funções de comunicação colectiva deverão ser gravados.

No que se refere às barreiras da MPI não é possível aplicar a mesma solução, uma vez que um processo não termina a chamada à barreira enquanto todos os processos não tiverem chamado uma barreira. Neste caso uma solução passaria por tentar garantir que as barreiras são chamadas pelos processos, no mesmo ponto de execução da aplicação. Em [BMPS03] é proposta a seguinte implementação: antes de um processo chamar a barreira executa a função *MPI_ Allreduce*, para determinar se todos os processos se encontram no mesmo ponto de execução. Se não, todos os processos que não tiverem efectuado o ponto de controlo, fazem-no. Em [SBF+04] são descritas, em detalhe, estratégias para lidar com problemas referentes às funções específicas de comunicação colectiva.

2.3.2.4 Estado escondido

Em MPI, estado escondido entende-se como o estado que não é exposto à aplicação, ou seja, não se encontra directamente acessível. Temos como exemplos *buffers* de mensagens e objectos solicitados. *Buffer* é um espaço de endereçamento da aplicação, composto pelos dados que deverão ser enviados ou recebidos, que será utilizado para guardar os dados transferidos entre processos. Vejamos o seguinte caso: um processo 0 pretende enviar uma mensagem ao processo 1, e para tal recorre à chamada de uma comunicação não-bloqueante de envio. O processo 0 recorre à utilização do sistema de *buffer*, como um local seguro, para armazenar a mensagem até que esta seja recebida pelo processo 1. Numa comunicação não-bloqueante, o processo que realiza a sua chamada retoma a computação logo após ter efectuado a chamada, sem ter que esperar pelo término da mesma. Se a aplicação efectuar o ponto de controlo dos processos 0 e 1, entre a chamada realizada pelo processo 0 e a recepção dos dados, por parte do processo 1, é necessário que durante o ponto de controlo seja também gravada informação referente ao *buffer* e à chamada de envio pendente, de forma a permitir a correcta recepção da mensagem, após o processo de restauro. Caso contrário, ocorrão mensagens consideradas em-trânsito.

Uma abordagem, utilizada no passado para resolver este problema, passava por inserir directamente sobre a biblioteca MPI as rotinas do PCR, para que durante a GPC fosse gravado o estado escondido da aplicação. Por exemplo, o sistema CoCheck [Ste96], integra o sistema de PCR Condor (referenciado na subsecção 2.2) com a biblioteca MPICH [MPI]. A desvantagem de tais abordagens é a falta de portabilidade e o facto do código fonte de algumas versões MPI poder não estar acessível.

Uma abordagem mais recente descrita em [BFM+06], utiliza uma camada extra situada entre a aplicação e a implementação nativa da MPI. Para gravar o estado escondido, esta camada extra intercepta chamadas à biblioteca MPI, por parte da aplicação, e grava informação referente ao estado escondido, que é armazenada juntamente com o ponto de controlo em disco. No processo de restauro, a camada extra utiliza a informação guardada para recriar o estado escondido. Esta abordagem permite ao sistema C3 (referenciado na subsecção 2.2) funcionar conjuntamente com qualquer implementação nativa da MPI [BFM+06], obtendo, assim, uma solução portável.

2.4 Memória partilhada

Os sistemas de memória partilhada contextualmente recaem sobre ambientes com múltiplos processadores (e/ou múltiplos núcleos). A memória partilhada refere-se a vários processadores que acedem a um ou mais módulos de memória partilhada. Existe uma

variedade de estratégias para ligar fisicamente os processadores aos módulos de memória, mas em termos lógicos cada processador está ligado a todos os módulos.

Em contexto de múltiplos processadores, memória partilhada refere-se a um endereço global de memória partilhada por todos os processadores. A memória *cache* permite reduzir a latência da memória, obrigando simultaneamente à utilização de protocolos de coerência da *cache* [CKH02], de forma a manter a memória consistente. A necessidade de utilizar protocolos de coerência advém do facto de que quando uma *cache* é actualizada com dados que poderão ser utilizados por outros processadores, esta actualização precisa de ser visíveis aos outros processadores, pois caso contrário os processadores trabalhariam sobre dados inconsistentes.

Em ambientes com múltiplos processadores, a memória partilhada pode ser utilizada (ao nível do *software*) como método de comunicação entre processadores [Str10]. Um modelo de programação normalmente utilizado é o modelo de linhas de execução, onde a memória é partilhada atráves da partilha de variáveis, entre as diversas LE. Num modelo de LE em programação paralela, um único processo poderá criar múltiplas LE que irão ser executadas em paralelo. Cada LE possui a sua própria pilha, variáveis locais e identificador, considerados estado privado. Para todas as LE existe uma única *heap*, partilhada entre estas, que é, portanto, considerada memória partilhada.

As LE, normalmente, são implementadas através de sub-rotinas fornecidas por uma biblioteca, tais como POSIX Threads [But97] e OpenMP [Qui03], e introduzidas sobre o código que se pretende executar em paralelo, ou através de um conjunto de directivas fornecidas por um compilador e embutidas sobre o código fonte. Seja qual for o método de implementação das LE, fica a cargo do programador especificar todo o paralelismo.

Em SMP é necessário garantir a consistência do estado partilhado lido por uma LE, visto que poderão surgir inconsistências nos dados quando uma LE altera o estado que está a ser lido por outra. Nesta situação, a LE pode estar a ler um estado desactualizado dos dados. Para prevenir este tipo de situações, é necessário garantir a exclusão mútua no acesso às variáveis partilhadas modificáveis.

2.4.1 Algoritmos

O ponto de controlo em SMP não foi tão extensivamente estudado na literatura, uma vez que estes sistemas são limitados no tamanho, não sendo por isso, sistemas onde os mecanismos de tolerância sejam uma preocupação primordial [BPS06]. Grande parte das abordagens existentes para PCR em SMP estão restritas ao PCNSO e centradas numa

implementação particular de SMP. Porém, foram propostas abordagens tanto a nível de *hardware* [SMHW02, PZT02] como de *software* [DLJ99, BMP+04].

Em SMP, ao nível de *hardware*, alguns algoritmos de ponto de controlo são incorporados como parte dos próprios protocolos de coerência [KR00], iniciando o seu ponto de controlo segundo as modificações das linhas da *cache* (algoritmos baseados em *cache*). Nestes algoritmos, geralmente, no momento de efectuar a GPC, é forçada a consistência da memória utilizando os protocolos de coerência da cache. O contexto dos processos é gravado em memória, enquanto o estado partilhado é gravado em disco. O contexto dos processos não é gravado em disco porque é assumido que a memória é segura e que o estado do processo pode ser restaurado a partir desta. Estes tipos de algoritmos são utilizados apenas para recuperação em caso de erros de *software* [AFM90]. O principal problema desta abordagem é que se a memória falhar é necessário restaurar a aplicação do início, tornando inútil o trabalho realizado pelo mecanismo de PCR.

Em SMP ao nível de *hardware* o ponto de controlo poderá ser global (coordenado bloqueante) [MGBK96, PL94a], local (coordenado não-bloqueante) [AFM90, BGJ+96] ou descoordenado [SFG99]. Grande parte dos algoritmos baseados na abordagem ao nível do *hardware*, para memória partilhada coerente, utilizam ponto de controlo global, no qual todos os processos participam na GPC, utilizando uma barreira na maior parte dos casos, assim como no restauro da aplicação [Aga11]. Uma alternativa é a utilização de algoritmos de ponto de controlo local, nos quais apenas os processos dependentes entre si realizam o ponto de controlo. No caso dos descoordenados, cada processo realiza o ponto de controlo independentemente.

Muito recentemente, em [Aga11], foi apresentado o primeiro ponto de controlo coordenado local, ao nível do *hardware*, para sistemas de memória partilhada, baseado em coerência da *cache*. Nesta abordagem, intitulada de *Rebound* pelos seus autores, o PCR é realizado por um grupo de processos dependentes entre si, e não por todos os processos. Sempre que é realizado o ponto de controlo, para um determinado grupo de processos, são gravadas em memória todas as "linhas sujas"nas *caches* dos processadores correspondentes, retendo cópias limpas e o estado de registo dos processadores nas *caches*. Em cada linha "escrita de volta", o controlador de memória lê da memória o antigo valor das linhas, armazenando-o num *software* de registo na memória. Em caso de falha de um processo, todos os processos que lhe são dependentes são obrigatoriamente restaurados. Restaurar nesta abordagem envolve invalidar as *caches* dos processadores correspondentes, copiar do registo para a memória, em ordem inversa, qualquer entrada registada destes processadores até à realização do ponto de controlo, e restaurar o estado do registo do processador [Aga11].

SafetyNet [SMHW02] é outro exemplo de uma abordagem ao nível do *hardware*, onde são registadas as alterações na memória local do processador. Para tal, são realizadas alterações ao nível do *hardware*, o que faz com que esta abordagem sofra de falta de portabilidade. Em SafetyNet, todos os componentes (processadores/*cache* e controladores de memória) coordenam o seu ponto de controlo local, de forma a que o conjunto dos pontos de controlo locais representem uma linha de recuperação global válida. Durante o processo de restauro, os processadores restauram os seus registos dos pontos de controlo e tanto a *cache* como a memória carregam os seus registos locais para restaurar o sistema para um estado global consistente [SMHW02].

Ao contrário de SafetyNet, o sistema C3 [BPS06] que utiliza uma abordagem ao nível do *software* para SMP, utilizando PCNA, permite a obtenção de FPC portáveis. Neste sistema, composto por um pré-compilador e uma biblioteca, o programador tem a função de escolher o local no código fonte onde deverá ser feito o ponto de controlo, sendo esta a única modificação ao código fonte. O pré-compilador analisa o código fonte da aplicação C/OpenMP e instrumentaliza-o de forma a que seja possível gravar a nível aplicacional, tanto o estado partilhado, como o estado privado das LE. O resultado do pré-compilador é compilado pelo compilador nativo existente na máquina, onde é compilada a aplicação, e ligado com a biblioteca permitindo a implementação de um protocolo coordenado, sem a necessidade de alterar as implementações nativas do OpenMP [BMP+04]. Durante a execução da aplicação, a biblioteca vai guardando todas as alocações e libertações de memória dinâmica, assim, aquando da GPC, serão armazenadas as porções da *heap* que estão correntemente alocadas.

A GPC em C3 é realizada utilizando um protocolo coordenado bloqueante, onde após cada LE gravar o seu estado local fica a cargo de uma única LE, considerada mestre (LEM), a gravação dos dados partilhados. Antes e depois da GPC, existe uma barreira global. A primeira serve para garantir que todas as LE já terminaram a escrita/leitura dos dados partilhados, antes de se iniciar a GPC, e a segunda tem como funcionalidade garantir que durante a gravação do estado aplicacional não existem LE a trabalhar sobre os dados partilhados. Durante o processo de restauro, as LE irão carregar o conteúdo dos seus respectivos pontos de controlo e restaurar o seu estado privado, enquanto a LEM, por sua vez, carrega todo o estado partilhado gravado em ficheiro e restaura-o. Em seguida, cada LE chama uma barreira, para garantir que todo o estado aplicacional foi correctamente restaurado, antes de autorizar o acesso a este, por parte das LE. Quando todas as LE tiverem chamado uma barreira procedem à execução normal da aplicação.

2.4.2 Problemas

A criação de PCR com suporte para SMP utilizando uma abordagem ao nível aplicacional, um dos objectivos desta tese, necessita de ter em consideração as seguintes problemáticas:

- Mecanismos de sincronização, nomeadamente fechos e barreiras;
- Estado escondido.

2.4.2.1 Fechos

Um dos cuidados a ter aquando a utilização de um protocolo bloqueante é garantir que este não provocará interbloqueio, uma situação onde duas ou mais LE estão bloqueadas para sempre, à espera uma(s) da(s) outra(s). Vejamos, na figura 2.4, uma situação onde ocorreria interbloqueio.

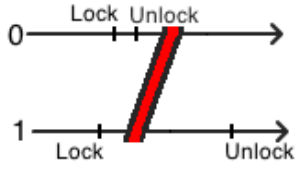

FIGURA 2.4: Situação de interbloqueio.

A região a vermelho, na figura 2.4, representa o momento de GPC e as rectas que a delimitam representam as barreiras colocadas antes e depois de realizada a GPC. O problema nesta situação é que a LE 0 pretende adquirir um fecho anteriormente adquirido pela LE 1. Enquanto a LE 0 esperava pela libertação do fecho por parte da LE 1, a LE 1 chamou a primeira barreira (antes da GPC) e encontra-se à espera que a LE 0 faça o mesmo, assim sendo, ficam ambas as LE à espera uma da outra, ocorrendo portanto uma situação de interbloqueio.

Para resolver a problemática, ilustrada na figura 2.4, pode-se recorrer à seguinte estratégia: quando as LE atingirem a primeira barreira do ponto de controlo (barreira antes da GPC), gravarão os fechos actualmente adquiridos e em seguida libertarão estes mesmos fechos. Esta libertação irá desbloquear as LE que se encontravam à espera destes fechos. Esta informação será utilizada durante a recuperação para readquirir os fechos. Depois da GPC e antes de retomar a execução normal da aplicação, o estado dos fechos de cada uma das LE irá retornar ao estado existente antes da libertação forçada dos fechos.

2.4.2.2 Barreiras

A utilização de barreiras por parte da própria aplicação poderá originar inconsistências no conteúdo dos pontos de controlo. Vejamos a figura 2.5, onde se ilustra este problema:

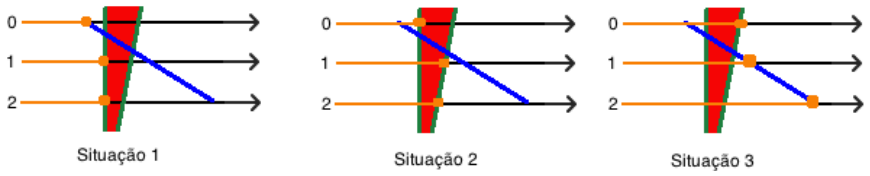

FIGURA 2.5: Situação de possível inconsistência do ponto de controlo.

A recta azul simboliza uma barreira utilizada pela própria aplicação, a intercepção da recta azul ou da recta verde com a recta preta (linha de execução) significa que a LE chamou a barreira e encontra-se à espera que as restantes LE façam o mesmo. A recta laranja indica o estado de execução da LE até ao momento. As restantes notações são as mesmas que as utilizadas na figura 2.4. Em OpenMP, qualquer chamada a uma barreira, por parte de uma LE, espera pela chamada de barreiras pelas restantes LE, mesmo que esta chamada não corresponda a barreiras na mesma localização no código fonte. O que acontece na situação 1 é que a LE 0 chama primeiro a barreira da aplicação, enquanto nos casos das LE 1 e 2 a barreira existente antes da GPC é chamada primeiro, ou seja, as LE 1 e 2 já estão a processar a GPC, enquanto a LE 0 espera na barreira, antes desta gravação (situação 2). Ao mesmo tempo que a LE 0 grava o conteúdo do ponto de controlo, as LE 1 e 2 executam a aplicação (situação 3), havendo a possibilidade destas estarem a alterar o estado partilhado que irá ser gravado pela LE 0, podendo ocorrer, assim, inconsistências nos dados dos FPC. Uma das formas de resolver tanto os problemas de interbloqueio, como os problemas de inconsistência do FPC, é garantir que o ponto de controlo ocorre num local antes ou depois das chamadas a barreiras, e que não existem fechos por libertar.

2.4.2.3 Estado escondido

Entende-se por estado escondido, em SMP, todo o estado presente na interface de programação de memória partilhada, durante a execução da aplicação, incluindo, entre outras, as informações referentes às LE e construtores de partilha de trabalhos.

Deverá ser gravada, durante o ponto de controlo, informação suficiente sobre as LE, de forma a que seja viável recuperar o seu conteúdo, durante o processo de restauro. Deverá ser possível recriar o mesmo número de LE, identificadores e respectivas pilhas, presentes durante a GPC. Para recriação do número de LE basta, durante a execução aplicacional, interceptar a chamada ao método que cria as LE e gravar o número de

LE que este irá criar. Em relação aos identificadores das LE, a solução passa por gerar identificadores de forma determinista, durante o processo de ponto de controlo, para que durante o processo de restauro sejam gerados da mesma forma. É necessário garantir o correcto mapeamento entre as LE e as respectivas pilhas, pois é possível que existam na aplicação apontadores para objectos presentes nas pilhas. Através de repetição e de recriação garante-se que a cada LE se encontra associada a respectiva pilha. Por exemplo, através da recriação do método que devolve o número de LE activas, é exequível mapear as regiões da pilha para os respectivos identificadores das LE [BPS06]. O conteúdo das pilhas pode ser restaurado, através da execução das chamadas às funções que estavam presentes nas pilhas (técnica de repetição) no momento da GPC.

Existem construtores cujo o estado é escondido, nas interfaces de programação paralela, e que geram execuções não-deterministas. O facto de serem não-deterministas, faz com que não possam ser recriadas utilizando repetição. Um exemplo deste tipo de construtores é o *single* da biblioteca OpenMP. A directiva *single* faz com que o bloco de código, que lhe é associado, seja executado por uma única LE. O bloco de código será executado pela primeira LE a chegar ao construtor *single* e as restantes ficam à espera numa barreira implícita no fim do construtor *single*, a não ser que seja especificada a cláusula *nowait*. Através de repetição não é possível garantir que o bloco de código executado por uma dada LE seja, durante o processo de restauro, executado pela mesma LE. A solução será portanto recriar os construtores de trabalho partilhado, tal como em [BPS06], onde se verifica uma possível recriação para alguns construtores presentes no OpenMP, utilizando um compilador.

2.5 Memória partilhada distribuída

Sistemas de memória partilhada distribuída (SMPD) implementam uma abstracção de memória partilhada em cima de sistemas de passagem de mensagens. Esta abstracção cria a ilusão de existir memória física partilhada, permitindo a utilização de paradigmas de programação de memória partilhada [NL91]. Nos SMPD não existe na realidade uma memória única partilhada entre os processos, o que realmente existe é espaço de endereçamento virtual partilhado entre os processos.

Tal como nos SMP, nos SMPD é utilizada *cache* para minimizar a latência de acesso aos dados e é necessário garantir a consistência da memória durante o ponto de controlo. Todavia, ao contrário dos SMP, nos SMPD a memória está separada pelos processos, portanto, quando um processo falha, parte da memória global fica inacessível e como tal, não é possível realizar o ponto de controlo idêntico ao realizado em SMP. Para realizar o ponto de controlo deverá ser mantida, à semelhança dos SMP, a coerência

da memória lógica global e como a memória física é distribuída, é necessário gravar em disco o contexto dos processos, ao contrário do que acontecia nos algoritmos baseados em *cache*, em SMP.

Segundo S. Kalaiselvi e V. Rajaraman, em [KR00] as abordagens de PCR em SMPD são extensões dos algoritmos baseados em *cache*, uma vez que as dependências criadas entre processos acontecem através da partilha de memória. Kalaiselvi e Rajaraman designam tais algoritmos como algoritmos baseados em memória, uma vez que são incorporados como base de protocolos de coerência de memória. Este tipo de algoritmo utiliza técnicas de sobreposição entre o armazenamento do contexto do processo e a computação, como forma de diminuir o impacto de gravação em disco.

A forma como é realizado o ponto de controlo em SMDP pode ser classificada em duas categorias já citadas: coordenado [KTT00, WF90] ou descoordenado[wLyK00, IS93, SJ95]. Vejamos alguns exemplos existentes na literatura de algoritmos de ponto de controlo coordenados e descoordenados.

Em [KTT00] é apresentado um algoritmo de ponto de controlo coordenado baseado em coerência da memória para providenciar PCR em SMPD. Este algoritmo utiliza a informação referente à coerência de memória da aplicação para seleccionar os dados realmente necessários para uma correcta recuperação, diminuindo assim, o tamanho dos FPC. Em [WF90] também é utilizado ponto de controlo coordenado, onde cada processo realiza o ponto de controlo sempre que uma das suas páginas modificadas é lida por outro processo. Em caso de falha de um processo, este pode ser independentemente restaurado a partir dos FPC mais recentes, sem a necessidade de envolvimento dos outros processos. Para que tal seja possível, é gravado nos FPC o estado dos processos dependentes ao processo que realiza a GPC.

Em [IS93] é apresentada uma abordagem que utiliza ponto de controlo descoordenado, baseado em registos, para providenciar tolerância a faltas em SMPD. Nesta abordagem, sempre que um processo lê uma página partilhada regista os dados lidos na sua memória volátil. O registo existente na memória volátil de um processo só é transferido para o seu disco rígido quando uma das páginas partilhadas, pertencentes a este processo, for lida por outro processo. Em caso de falha num processo, os registos existentes na sua memória volátil são perdidos. Se tal acontecer, e se apenas for este o processo que falhou, este processo poderá restaurar os seus registos perdidos a partir dos outros processos. Esta abordagem só permite restaurar o sistema quando há apenas um processo que falha. De forma a colmatar esta desvantagem foi desenvolvido em [SJ95] um melhoramento a esta abordagem, que permite restaurar sistemas onde múltiplos processos falham e reduzir o custo acrescido dos registos[wLyK00].

2.6 Grelhas computacionais

Os ambientes de computação em grande escala, como é o caso das grelhas computacionais *CERN LCG* [Tea04], *NorduGrid* [EKS+03], *TeraGrid* [TTP] e *Grid'5000* [CCD+05], entre outras, colocam à disposição centenas de recursos para a resolução de problemas de elevada complexidade/dimensão. Uma das principais vantagens das grelhas computacionais é o facto de as redes envolvidas poderem ser distribuídas entre entidades ou empresas, que concordaram trabalhar no mesmo projecto ou problema. Na verdade, o que realmente distingue as grelhas computacionais dos sistemas de *clusters* mais tradicionais é que estas tendem a ser geograficamente mais dispersas. Uma das principais desvantagens é o facto dos resultados de todos os processos terem de ser enviados de um local para o outro e avaliados de forma colaborativa, para que o resultado final possa ser deduzido.

A heterogeneidade das grelhas computacionais, a existência de arquitecturas paralelas e a sua complexidade inerente, representam um desafio para os mecanismos de PCR. Esta heterogeneidade faz com que seja impossível a aplicação das tradicionais técnicas de armazenamento do estado, que utilizam estratégias não portáveis, requerendo, assim, ferramentas e representações de pontos de controlo portáveis. Os FPC deverão permitir o restauro da aplicação, em máquinas com arquitecturas diferentes daquelas onde foram criados. O mecanismo de PCR, para grelhas computacionais, deverá suportar aplicações paralelas com protocolos de comunicações independentes. A ferramenta de ponto de controlo não deverá fazer suposições em relação à interface ou à implementação que está a ser utilizada [RMG+10]. As grelhas computacionais incluem máquinas pertencentes a diferentes entidades que não podem ser obrigadas a fornecer determinadas versões da interface MPI. Uma vez que nas grelhas computacionais são executadas aplicações paralelas/distribuídas, deverá ser preservada a consistência global de uma aplicação após o seu restauro. Devemos ter em conta que em aplicações paralelas as dependências criadas por comunicações entre os processos devem ser preservadas durante a recuperação.

Nas grelhas computacionais existem elementos de armazenamento remotos, onde os dados são armazenados [SAR]. Os elementos de armazenamento podem ser utilizados para guardar o resultado de um trabalho, enquanto o utilizador e a máquina cliente estão desconectados da grelha. Quando comparados com os ambientes *cluster* tradicionais, os elementos de armazenamento aumentam a latência de gravação/leitura de dados, portanto, a optimização de dados torna-se mais fulcral nas grelhas do que nos *clusters* tradicionais.

A ampla distribuição geográfica das grelhas computacionais, a sua dimensão e as restrições colocadas pelos responsáveis dos recursos partilhados levantam problemas à disponibilidade dos recursos das grelhas [IJSE07], portanto, não é de estranhar, que os recursos disponibilizados para uma aplicação variem durante a execução. Esta variabilidade nos recursos das grelhas, pode ocorrer pelos seguintes motivos:

- Sobrecarga computacional nos nós;
- Indisponibilidade dos nós devido a falhas;
- Manutenção;
- Disponibilidade de novos recursos;
- Libertação dos recursos alocados para serem utilizados por trabalhos com uma maior prioridade.

Muitos dos recursos das grelhas estão agrupados em *clusters*, disponibilizados por períodos limitados de tempo, pelos seus responsáveis. Alguns destes recursos podem mesmo ser removidos do sistema pelos seus responsáveis, para serem utilizados noutras tarefas [IJSE07]. É, então, fundamental, que as aplicações em execução nas grelhas possuam um mecanismo de adaptação à variabilidade de recursos, de forma a obter uma melhor performance e a retirar um melhor partido dos recursos disponíveis em cada instante. O próprio mecanismo de tolerância a faltas pode ser utilizado para adaptação a novos recursos, sendo para tal necessário efectuar o ponto de controlo, terminar a execução da aplicação e reiniciá-la nos novos recursos disponíveis. No entanto, o ideal é um mecanismo de adaptação dinâmico (portável), que permita adaptar a aplicação aos recursos disponíveis, sem necessidade de reiniciar a aplicação.

Um problema importante em grelhas computacionais consiste na selecção de um conjunto de recursos iniciais que permita à aplicação obter boas performances. A determinação do conjunto de recursos óptimo para uma dada aplicação (o conjunto de recursos com o menor tempo de execução) é considerado um problema NP-completo [VD05]. Outro problema também relevante é a adaptação dinâmica de recursos durante a execução aplicacional, que se refere à performance e disponibilidade dos recursos das grelhas ao longo da execução de uma aplicação. Sumariamente, o primeiro problema refere-se à escolha inicial dos recursos para uma dada aplicação, enquanto o segundo refere-se à adaptação dos recursos de uma dada aplicação durante a sua execução.

Algumas estratégias de selecção dos recursos iniciais, como as utilizadas em [AAB+05, VD05], utilizam um modelo de performance que permite prever os tempos de execução

da aplicação para diferentes conjuntos de recursos. Na fase de selecção dos recursos iniciais, o modelo selecciona, de um leque de recursos, o conjunto que possibilita o tempo de execução previsto mais baixo. Esta fase é novamente repetida se durante a execução da aplicação ocorrer degradação da sua performance. Na abordagem descrita em [WMB07] inicia-se a aplicação num conjunto qualquer de recursos e durante a execução ajustam-se os recursos computacionais (retirando ou adicionando nós/*clusters*), consoante estimativas baseadas na colecta periódica de dados da performance.

Actualmente, abordagens de adaptação dinâmica a recursos, como por exemplo CP3 [FPS06], AMPI [HLK03] e TMPI [TY01] baseiam-se em sobre-decomposição. A ideia da sobre-decomposição consiste na utilização de um número lógico de 'n' processos, superior ao número de processadores físicos, sendo 'n' (idealmente) o número de processadores a partir do qual a aplicação paralela não escala, repartindo assim os 'n' processos pelos processadores físicos. No caso do CP3, os processos lógicos mapeados para um processador físico, são aglomerados num único processo com múltiplas LE, em que cada linha corresponde a um processo lógico [FPS06]. O objectivo é providenciar uma implementação de baixo custo com excesso de tarefas paralelas, em que se procede à redistribuição de trabalho pelos recursos alocados. À medida que os recursos alocados para a aplicação variam durante a sua execução, é efectuado um re-mapeamento dos processos pelos recursos actualmente disponíveis.

2.7 Optimizações

Podemos optimizar o PCR se optimizarmos o conteúdo dos FPC e a frequência com que são gravados. Apresentam-se de seguida algumas técnicas propostas na literatura para optimização dos FPC e para obtenção da frequência de GPC óptima.

2.7.1 Ficheiros de ponto de controlo

Nos tempos que correm é comum encontrar aplicações científicas que utilizam *Megabytes* de espaço de memória, resultando muitas vezes em FPC de grandes dimensões. Sob tais condições é fundamental optimizar o tempo de gravação e o espaço ocupado pelos FPC. Apresenta-se duas categorias de optimizações de FPC: dissimulação da latência e exclusão da memória.

2.7.1.1 Dissimulação da latência

A dissimulação da latência baseia-se em reduzir ou dissimular o custo de GPC em disco. Duas técnicas de dissimulação de latência são ponto de controlo cópia na escrita [PL88, LNP94] e ponto de controlo sem disco [PL94a, SS98b].

Na técnica ponto de controlo cópia na escrita ocorre uma bifurcação (recorrendo à função *fork()* [NW] do SO) do processo principal num processo filho, que realiza eficientemente uma cópia do ponto de controlo em memória. O processo original continua a execução enquanto o processo filho escreve em disco, de forma assíncrona, o conteúdo da referida cópia. O ponto de controlo cópia na escrita permite ao processo que o escreve aceder à memória original do programa. Uma página de memória é copiada para o processo filho apenas se for modificada antes de ser escrita em disco. Esta optimização permite ao ponto do controlo ser gravado de forma assíncrona, sem o custo adicional de fazer cópia da memória, podendo ser implementada quase sem esforço em sistemas que utilizam cópia na escrita na função *fork()*[KBP96].

O ponto de controlo é na maior parte dos mecanismos de PCR gravado num disco, mas nas aplicações que necessitam, por alguma razão, de realizar com mais frequência a GPC, estes poderão causar um grande impacto na sua performance. Por este motivo, foi proposto por Plank a técnica de ponto de controlo sem disco [PLP98]. Em vez de gravar o ponto de controlo em disco, processadores extra são utilizados para a gravação de informação redundante, de forma a que se algum processador falhar o sistema possa continuar, na mesma, em execução [KBP96]. Esta técnica elimina, assim, o custo adicional de leitura/escrita do ponto de controlo no disco. Esta técnica tem, no entanto, uma menor cobertura de falhas ser do que a do ponto de controlo baseado em disco, uma vez que não cobre os casos de falha do processador onde está gravado o conteúdo do ponto de controlo. Além disto, o ponto de controlo sem disco inclui o custo adicional da memória, do processador e de rede que são inexistentes nos esquemas baseados em disco [SRNSK10].

2.7.1.2 Exclusão de memória

A exclusão de memória optimiza o tamanho dos FPC filtrando os dados a serem gravados. Esta técnica classifica as área de memória em três categorias: limpa, morta e suja. Durante o ponto de controlo, as técnicas de exclusão de memória apenas gravam as áreas de memória consideradas sujas. As áreas consideras limpas ou mortas, são excluídas dos FPC. Áreas de memória suja são aquelas que foram modificadas desde o ponto de controlo anterior.

```
01: private void copiarJogador(Jogador a, Jogador b, int factor){
02: int i,j;
03: int size = a.tab.length;
04:
05: b.copias = b.copias+1;
06: // Gravar ponto de controlo
07:
08: for(i = 0; i < size; i++)
09:   for(j = 0; j < size; j++)
10:     a.tab[i][j] = b.tab[i][j] * factor;
11:
12: // Gravar ponto de controlo
13:}
```

FIGURA 2.6: Ilustração de memória limpa e memória morta.

A figura 2.6 ilustra as definições de memória morta e memória limpa. Denomina-se memória morta aos dados que após a GPC terão um novo valor escrito neles, antes do seu valor actual ser lido (valor antes da GPC), ou seja, o valor actual destes dados nunca será lido, e, portanto, estes dados podem ser omitidos durante a GPC. Na figura 2.6, o objecto 'a', durante a GPC na linha 06, é um exemplo de memória morta, dado que o valor do objecto 'a' durante a GPC (linha 06) será alterado na linha 10, sem que antes o seu valor (na linha 06) tenha sido lido. Memória limpa corresponde aos dados que não foram alterados na GPC anterior. Na figura 2.6, durante a GPC na linha 12, o objecto 'b' é considerado memória limpa, uma vez que o seu conteúdo durante a gravação na linha 12 é o mesmo que na gravação na linha 06, portanto, não necessita de ser gravado novamente em disco. Para tal, é necessário manter em disco os FPC anteriores, ocupando, assim, mais espaço em disco. Exemplos de técnicas de exclusão de memória são: ponto de controlo incremental [FB88, SPJ+04, GSJP05] e ponto de controlo direccionado ao utilizador [KBP96], entre outros.

O ponto de controlo incremental permite a obtenção de FPC com um tamanho mínimo, próximo do tamanho óptimo, à custa de procedimentos adicionais. Estes procedimentos adicionais poderão ser efectuados através de um compilador [BMP+08] ou de uma biblioteca [Jey04], que monotoriza as leituras e escritas do programa [NW94, LLMM99], de forma a que seja possível a exclusão de memória limpa e morta dos FPC. Esta computação adicional também poderá ser realizada com a utilização do mecanismo de protecção de páginas do sistema operativo (como é o caso da ferramenta libckpt [PBKL95]), para que registe as páginas que foram alteradas desde o último ponto de controlo. Em [BMP+08] mostrou-se que para algumas aplicações, o ponto de controlo incremental reduziu o tamanho dos FPC até 80%. Esta técnica apresenta boas performances em programas que exibem boa localidade [Den05], no entanto, em programas com grandes quantidades de áreas de memória suja, o custo de motorizar estas páginas pode aumentar o custo do ponto de controlo [KBP96].

Ao contrário da técnica a cima descrita, que funciona de forma automatizada, no ponto de controlo direccionado ao utilizador, fica a cargo do programador especificar os locais de memória que deverão ser excluídos de cada FPC (locais de memória morta ou limpa), utilizando, para tal, as rotinas providenciadas. O programador também possui a liberdade de escolher os locais de memória que deverão ser incluídos, assim como escolher o local onde poderá ser tomado o ponto de controlo. Esta técnica permite deste modo que o programador coloque o ponto de controlo em locais onde a memória morta é mais acentuada (como, por exemplo, fim de ciclos), minimizando o tamanho dos FPC. Uma das vantagens desta técnica, comparativamente ao ponto de controlo incremental, é o facto de o programador poder reduzir o custo adicional da motorização das leituras e escritas, ficando responsável pela pré-análise destas mesmas leituras e escritas, colocando assim os pontos de controlo em locais que gerem FPC mais reduzidos [KBP96].

2.7.2 Frequência de ponto de controlo

A implementação de um mecanismo de PCR sobre uma aplicação aumenta o seu tempo médio de execução. Quando maior for a frequência de GPC em disco maior será o impacto na performance. O ideal será obter uma frequência de GPC optimizada, de forma a minimizar o impacto causado na performance da aplicação, pelo processo de ponto de controlo. Em [VSL09, LML01] foram deduzidas equações para o cálculo do número óptimo de GPC para uma dada probabilidade de falhas. Em ambos os estudos realizados concluiu-se que a frequência óptima depende da taxa de falha do sistema.

Um dos problemas relacionados com o cálculo do número óptimo de GPC é a obtenção da probabilidade de falha. Nunca se poderá saber antes da execução de uma aplicação, num dado sistema, qual a verdadeira probabilidade de falha do mesmo. Para além disto, a probabilidade de falha poderá mudar durante a execução da própria aplicação. Este valor poderá ser maior durante a primeira hora de execução do que na última hora de execução, por exemplo.

Em [DUVE10] foram propostas duas abordagens para ajustamento dinâmico da frequência de GPC, durante a execução da aplicação, uma periódica e outra não periódica. Ambas adaptam a probabilidade de falha baseando-se em estatísticas de falha de anteriores execuções. Estas estatísticas foram calculadas através de uma unidade de armazenamento que monotoriza o número de sucessos e insucessos durante a execução de uma dada aplicação. A diferença entre estas duas abordagens está no intervalo de tempo em que é recalculada a probabilidade de falha. Na abordagem periódica a probabilidade de falha é calculada em intervalos regulares de tempo, enquanto na abordagem não periódica o intervalo de tempo é não regular. Na abordagem não periódica os intervalos

de tempo variam consoante a probabilidade de falha. Se a probabilidade de falha for diminuindo, isto implica que menos falhas estão a ocorrer, portanto deveremos diminuir a frequência de GPC e aumentar o intervalo de tempo para o próximo recálculo. Por outro lado, se a probabilidade aumentar deverá-se-à aumentar a frequência de GPC e diminuir o intervalo de tempo para o próximo recálculo.

Capítulo 3

Programação orientada aos aspectos

Neste capítulo é apresentado o paradigma de programação orientado aos aspectos (POA). Na secção 3.1 apresenta-se uma visão geral sobre este paradigma. Enquanto na secção 3.2 e 3.3, descrevem-se a extensão da POA para JAVA e exemplos de aplicações onde se utilizou a POA.

3.1 Visão geral

A maior parte das aplicações são constituídas por módulos que trabalham em conjunto para concretizar um leque de requisitos. Devido à natureza das ferramentas e das linguagens orientadas aos objectos, o ideal de programação modular é dificilmente realizável em sistemas de grande complexidade. Em vez disto, os programadores são forçados a criar módulos que contêm diferentes objectivos, por exemplo, uma simples função de registo poderá estar distribuída por vários módulos na aplicação [GL03]. A POA foi criada em 1997 nos laboratórios da Xerox [Cen], com o objectivo de resolver problemas e requisitos que não são bem solucionados pelas metodologias tradicionais de programação. Por exemplo, o problema do reforço de segurança de uma aplicação é transversal à modularidade aplicacional, sendo a sua resolução de forma disciplinada difícil em linguagens de programação tradicionais [PP09]. Em linguagens orientadas aos objectos, requisitos transversais como os de reforço de segurança não são facilmente transformados em classes, uma vez que tais requisitos lhe são transversais, e como tal não podem ser reutilizáveis, refinados ou herdados. Estes requisitos estão espalhados pelo código fonte da aplicação de forma insubordinada, o que torna difícil a sua modularidade.

A POA permite a criação de aspectos, que são módulos que servem para localizar funcionalidades distribuídas por diversos pontos de execução da aplicação. A POA é

uma abordagem de modularização de requisitos transversais, assim como a programação orientada a objectos é uma abordagem para modularização de requisitos comuns. Apesar dos mecanismos de modularização das linguagens de programação orientada aos objectos serem muito úteis, não são capazes de modularizar todos requisitos de sistemas complexos, pois haverão requisitos transversais à modularização da aplicação [KHH+01]. Várias pesquisas [BH08, Ml07] demostraram que a POA beneficia o desenho e qualidade do software. Os benefícios da POA incluem :

- Aumento da legibilidade do código;
- Redução de código;
- Maior facilidade de manutenção e evolução da aplicação;
- Aplicações passíveis de serem depuradas mais facilmente;
- Aumento da reutilização do código.

Com o objectivo de adoptar os conceitos e metodologias da POA, várias linguagens possuem extensões para a POA, nomeadamente, a *AspectJ* [KHH+01], *AspectC++* [Spi05], *AspectH* [Meu97], entre outras, que funcionam juntamente com a linguagem.

3.2 AspectJ

AspectJ[KHH+01] consiste numa extensão orientada aos aspectos criada pela Xerox para a linguagem de programação orientada aos objectos JAVA [JAV], tendo como principais componentes pontos de junção, *pointcuts*, *advices* e aspectos. Ponto de junção é um ponto de execução bem definido numa aplicação. Os *pointcuts* são construtores que servem para interceptar um dado conjunto de pontos de junção. O *advice* serve para especificar as acções atribuídas a um dado *pointcut*. Finalmente, os aspectos servem para encapsular pontos de junção, *pointcuts* e *advices*, sendo criados da mesma forma que as classes, o que permite o encapsulamento da implementação de determinados requisitos.

AspectJ possui assinaturas personalizadas para diferentes tipos de pontos de junção, nomeadamente, assinaturas para chamada de métodos (por exemplo *call(public static void main(..))*), construtores (*initalization (private ClasseX.new ())*) e tratamento de excepções (*handle(ClasseXException)*), entre outros. *AspectJ* permite a detecção de padrões através do uso do caracter '*'. Por exemplo, imaginemos que queríamos todos os métodos privados, sem argumentos, que retornem inteiros e que pertençam a uma dada classe chamada *ClasseX*. Para isso definiríamos a assinatura *call(private int ClasseX.*())*.

Se, por outro lado, pretendermos todos os métodos de todas as classes a assinatura seria *call(* *.*())*. Depois de definidos os pontos de junção é necessário especificá-los utilizando *pointcuts*. *Pointcuts* são construtores utilizados em *AspectJ* para identificar pontos de junção e obter o conteúdo circundante a este. Vejamos, na figura 3.1, vários exemplos de *pointcuts*:

```
01: ...
02: pointcut getSaldoP(): call(public int getSaldo());
03:
04: pointcut getMainP() : call(public static void main (..));
05:
06: pointcut getAllsP() : call(public int getSaldo()) &&
07:                       initalization (new (..));
08: ...
```

FIGURA 3.1: Exemplos de *pointcuts*.

O primeiro e o segundo *pointcut* interceptam as chamadas aos métodos *getSaldo()* e *main(..)*, respectivamente, enquanto o último *pointcut* (linha 06) intercepta a chamada ao método *getSaldo()* e aos construtores existentes em toda a aplicação.

```
01: ...
02: before() : getMainP(){
03:     System.out.println("A aplicacao vai ser executada");
04: }
05:
06: after() : getMainP(){
07:     System.out.println("A aplicacao terminou");
08: }
09:
10: int around(): getSaldoP(){
11:      System.out.println("Vamos calcular o saldo!");
12:      int saldo = proceed();
13:      System.out.println("Saldo disponivel :"+saldo);
14:      saldo *= 2;
15:    return saldo;
16: }
17: ...
```

FIGURA 3.2: Exemplos de *advices*.

Um *advice* é constituído pelo código que deverá ser executado quando a aplicação atinge um dado ponto de junção. Os tipos de *advices* são distinguidos mediante o momento em que entram em acção, relativamente aos pontos de junção. Existem três tipos de *advices*, nomeadamente, *before*, *after* e *around*. O código existente no *advice* poderá ser executado antes, depois ou em vez do ponto de junção, respectivamente. No caso do

tipo *around*, o código que lhe pertence é executado em vez do ponto de junção, sendo, no entanto, possível invocar o próprio ponto de junção recorrendo à função *proceed()*.

Os *advices* da linha 02 e da linha 06 (figura 3.2) atribuem acções ao *poincut* da linha 04 da figura 3.1, enquanto o *advice* da linha 10 atribui acções ao *pointcut* da linha 02, da mesma figura. O primeiro e segundo *advice* imprimem uma mensagem antes e depois da execução do método *main*, respectivamente. O último *advice* da figura 3.2 intercepta a chamada do método *getSaldo()*, imprime uma mensagem antes e depois da sua execução (linha 11 e 13, respectivamente) e guarda o valor devolvido por este método numa variável (linha 12), acabando por a retornar (linha 15) depois do seu valor ser manipulado (linha 14).

Os aspectos servem para modularizar os requisitos transversais, encapsulando os pontos de junção, *pointcuts* e *advices*. Quando dois ou mais aspectos possuem *pointcuts* que interceptam pontos de junção comuns, *AspectJ* permite definir precedências em relação à ordem de execução destes aspectos.

3.3 Aplicações

A POA, para além de tema central em diversas publicações [Fil05], é utilizada por grupos de investigadores como parte do seu trabalho [GS09, SCM07, PRS10, SM11, MMJ10]. Descrevem-se, de seguida, alguns dos projectos presentes na literatura que utilizam a POA, a *AspectGrid* e *JEColi*, dois projectos desenvolvidos na Universidade do Minho. Por fim, dão-se a conhecer dois projectos recentes, envolvendo a POA, cujo último relaciona-se com os objectivos desta tese.

O projecto *AspectGrid* [Asp] consiste na criação de módulos que permitem adaptar aplicações científicas a ambientes de grelhas computacionais. A implementação de tais módulos utiliza as técnicas da POA de forma a disponibilizar serviços de execução concorrente e distribuída, mecanismo de tolerância a faltas [SM11], adaptação dinâmica a recursos, entre outros [SM11].

Foi desenvolvida uma versão paralela da plataforma *JECoLi* [EMR09], que permite a execução de problemas complexos em plataformas de alto desempenho. Para resolver os problemas de paralelismo utilizaram-se técnicas de POA (*AspectJ*), o que permitiu o desenvolvimento de paralelismo de uma forma não invasiva, localizada e injectável [PRS10]. Através destas técnicas foram construídos modelos paralelos, que podem ser anexados à plataforma, para ir de encontro aos requisitos do utilizador ou requisitos computacionais. Os módulos podem ser, ou não, seleccionados através de uma GUI, pelo utilizador final, para ir de encontro às suas necessidades.

No projecto realizado em [SLB02] utilizaram-se técnicas de POA, mais especificamente *AspectJ*, para a implementação de requisitos relacionados com distribuição e persistência no sistema de informação *Health Watcher* [Wat]. Os aspectos relacionados com a distribuição implementam um sistema de acesso remoto utilizando Java RMI [Ora], enquanto os aspectos que dizem respeito à persistência implementam funcionalidades de persistência utilizando uma base de dados relacionais. Os autores deste projecto concluíram que a utilização de *AspectJ* permite uma implementação mais robusta, quando comparada com a implementação utilizando apenas JAVA. Os mesmos aconselham a utilização dos construtores do *AspectJ* com precaução, para que não se obtenham efeitos colaterais indesejados. Para além disto, alertam para o facto de a reutilização dos *pointcuts* ficar comprometida, uma vez que estes identificam pontos específicos de um dado sistema(s).

Muito recentemente, foi detalhado em [ABM11] um trabalho que envolve o desenvolvimento de PCR não instrusivo para aplicações sequenciais e paralelas. Neste trabalho, as rotinas relacionas com o PCR encontram-se separadas do código base da aplicação, permitindo ao programador activá-las ou desactivá-las sempre que achar necessário. Nesta abordagem são deixadas a cargo do programador as tarefas de identificação do local, conteúdo e frequência do ponto de controlo, assim como a de garantir a consistência dos dados que irão ser gravados. Os detalhes relacionados com o ponto de controlo são especificados através do preenchimento dos próprios *pointcuts*, para lidar com estes mesmos detalhes.

Capítulo 4

Trabalho realizado

Neste capítulo descreve-se o trabalho de implementação e de desenho das ferramentas de PCR, assim como de adaptação dinâmica a recursos. Na secção 4.1 dão-se a conhecer os requisitos a serem suportados pela ferramenta de PCR, de modo a que seja funcional em grelhas computacionais. Na secção 4.2 apresentam-se as técnicas utilizadas para cumprir os requisitos propostos. Por sua vez, na secção 4.3 é descrita, em detalhe, a estrutura da ferramenta de PCR e o seu funcionamento em diferentes ambientes, sendo apresentadas, na secção 4.4, a sua implementação em POA e as respectivas vantagens/-desvantagens. Por fim, são ilustrados o funcionamento da ferramenta de detecção do local do ponto de controlo e a estratégia de adaptação dinâmica a recursos, nas secções 4.5 e 4.6, respectivamente.

4.1 Requisitos

Antes de iniciar a implementação da ferramenta de PCR, especificaram-se os seus requisitos funcionais. Tendo em conta que um dos objectivos desta tese é a construção de um mecanismo de PCR para grelhas computacionais, identificaram-se os seguintes requisitos:

1. Portável entre diferentes sistemas operativos;
2. Minimizar a informação a guardar;
3. Funcional em múltiplos ambientes;
4. Minimizar alterações ao código fonte das aplicações.

Os requisitos 1,2 e 3 estão directamente relacionados com as características das grelhas computacionais. Como referido anteriormente, grelhas computacionais são ambientes heterogéneos (requisito 1) e de grandes dimensões (requisito 2), onde são executadas aplicações paralelas/distribuídas (requisito 3). Quando ocorrem falhas numa aplicação e se procede ao respectivo processo de restauro, não é garantido que esta aplicação seja restaurada em máquinas com igual arquitectura. Portanto, o mecanismo de PCR terá que criar FPC portáveis, para que seja possível o restauro dos mesmos em máquinas com diferentes arquitecturas. Advindo do facto das grelhas computacionais utilizarem elementos de armazenamento remotos, e de sobre estas serem executadas aplicações de grandes dimensões é necessário optimizar os FPC. Reduzindo o tamanho dos FPC, o tempo de GPC também é reduzido, diminuindo consequentemente o custo adicional causado pelo mecanismo de PCR .

A disponibilidade nas grelhas computacionais de máquinas em paralelo, e a sua própria natureza distribuída, fazem com que a maior parte das aplicações executadas sobre estas sejam paralelas/distribuídas [DPMG08]. Como tal, é fundamental que o mecanismo de PCR para grelhas computacionais suporte sistemas sequenciais, SMP e SMD.

O requisito 4 está relacionado com questões de estruturação, legibilidade de código e seguimento de uma linha ideológica, anteriormente utilizada em [GS09, SCM07, PRS10]. Pretende-se implementar o mecanismo de PCR, de modo a separar as suas rotinas do código base da aplicação, rotinas estas que deverão ser implementadas em módulos separados, para que sejam adicionados, removidos e analisados de forma acessível e rápida, sempre que desejado.

4.2 Visão geral da abordagem

Antes de apresentar a abordagem, introduziremos algumas definições:

Definição 1 Local do ponto de controlo é o local no código da aplicação onde é realizada a GPC.

Definição 2 Define-se como ponto seguro o ponto de execução no código fonte no qual não existam nem mecanismos de sincronização activos, nem mensagens em-trânsito ou órfãs durante a GPC.

Definição 3 Denomina-se por linha de recuperação o último estado consistente da aplicação gravado em disco.

Definição 4 Os métodos cuja execução é desnecessária para o correcto restauro da aplicação denominam-se de métodos ignoráveis.

Definição 5 Denominam-se de sistemas híbridos, os sistemas compostos por múltiplos processos com múltiplas LE.

Tendo em conta os requisitos funcionais, especificados na secção 4.1, optámos pelo desenvolvimento da ferramenta de PCR completamente ao nível aplicacional. A utilização de uma abordagem ao nível aplicacional, para além de evitar alterações à camada de *software* intermédia das grelhas computacionais, permite a criação de FPC portáveis e optimizados. A portabilidade é também estendida a aplicações paralelas com paralelismo baseado tanto em linhas de execução JAVA, como em processos MPI.

A ferramenta foi concebida para ser utilizada como uma biblioteca. A biblioteca automatiza os processos de ponto de controlo e restauro, deixando a cargo do programador as tarefas iniciais de especificação do local, conteúdo e frequência do ponto de controlo e de identificação dos métodos ignoráveis. Todo o restante código adicional, necessário para a realização do processo de ponto de controlo e do processo de restauro, é providenciado pela nossa biblioteca. Com o objectivo de auxiliar o programador na especificação dos itens a cima referidos, foi também criada uma ferramenta de análise dinâmica que funciona como pré-compilador, cuja utilização é de carácter opcional.

Conceptualmente, para restaurar a aplicação no mesmo ponto de execução é necessário guardar todas as chamadas a métodos e respectivos parâmetros. De forma a evitar este custo adicional, a nossa abordagem baseia-se em pontos seguros e métodos ignoráveis. Durante o processo de ponto de controlo, é contabilizado o número de pontos seguros já interceptados e realizada a GPC periodicamente. A periodicidade da GPC é definida pelo programador e expressa em número de pontos seguros interceptados por execução. Com a utilização de pontos seguros apenas é necessário acompanhar a execução dos métodos que estão presentes na pilha, quando os pontos seguros são interceptados. Por esta razão, é fornecido o módulo de métodos ignoráveis que permite ao programador especificar os métodos que podem ser ignorados durante o processo de restauro. Durante a GPC, para além do conteúdo do ponto de controlo, é gravado também o número de pontos seguros interceptados até então. Durante o processo de restauro, a ferramenta para reconstruir a pilha executa apenas os métodos que não podem ser ignorados, até que o número de pontos seguros gravados seja alcançado. A especificação dos métodos ignoráveis permite minimizar o tempo de restauro aplicacional.

Em SMP, a GPC é realizada utilizando um algoritmo coordenado bloqueante, onde todas as LE chamam uma barreira antes e depois da GPC. O restauro aplicacional é realizado sequencialmente, executando na mesma os construtores de criação das LE, de forma a reconstituir o número de LE e suas respectivas pilhas. Cada LE chama uma barreira logo após o carregamento do estado aplicacional.

Em SMD, a GPC é realizada utilizando um algoritmo coordenado não bloqueante centralizado. Durante a GPC, cada processo envia os seus dados locais, a serem gravados, ao processo mestre (algoritmo coordenado) e enquanto este procede à gravação dos dados colectados, os processos escravos retomam a execução aplicacional (estratégia não bloqueante centralizada). Se os processos possuírem múltiplas LE, tanto o processo de ponto de controlo como o processo de restauro locais são realizados de forma idêntica à realizada em SMP, caso contrário serão realizados de forma sequencial.

A nossa abordagem usufrui das potencialidades da POA, sendo desenvolvida utilizando *AspectJ* [KHH+01] para obter o mínimo de alterações possíveis ao código base, permitindo encapsular o código relacionado com o PCR em diferentes aspectos, obtendo-se assim uma maior legibilidade e transparência com menor intrusividade. Para além disso, o código gerado é portável. A nossa abordagem apresenta dois importantes benefícios: - o código base da aplicação mantém-se praticamente inalterado; - providencia mecanismos para realizar PCR em sistemas sequenciais, SMP, SMD e híbridos.

4.2.1 Local de colocação do ponto de controlo

Comecemos, esta secção, por especificar e diferenciar entre local do ponto de controlo, ponto de controlo e ponto seguro.

```
01: ...
02: x = 10;
03: while(x > 0){
04:    x--;
05:    exec_algorit();
06: }
07: ...
```

FIGURA 4.1: Exemplo de código antes de definido o local do ponto de controlo.

Na nossa abordagem, local de ponto de controlo entende-se como sendo ou a execução de um método, ou a execução de um determinado bloco de instruções, onde se poderá realizar a GPC, logo após o término da sua execução. No caso do exemplo da figura 4.1, o local de ponto de controlo poderia ser apenas a instrução da linha 04, a da linha 05 ou todo o bloco de instruções do ciclo *while*. Quando falamos em pontos de controlo, presentes no local de ponto de controlo, referimo-nos aos pontos imediatamente antes ao local do ponto de controlo, onde se poderá realizar a GPC. Se estes pontos de controlo respeitarem a definição 2, descrita no início da secção 4.2, serão designados de pontos seguros.

Vejamos o seguinte exemplo: se escolhermos como local do ponto de controlo o método da linha 05, estamos, então, interessados nos pontos de controlo presentes imediatamente antes da linha 05 e imediatamente depois da linha 04.

Quando o programador pretender definir o local onde deverá ser realizada a GPC deverá tomar em consideração os seguintes critérios:

1. Ganho temporal;
2. Frequência de GPC;
3. Dispersão dos pontos seguros;
4. Probabilidade de falha.

O primeiro critério refere-se ao tempo poupado pela aplicação ao realizar o processo de restauro, utilizando o dito local para GPC, ao invés de re-executar a aplicação do início. O segundo critério refere-se à frequência de pontos seguros neste local, onde quanto maior a frequência maior o número de oportunidades de realizar a GPC, durante a execução da aplicação. No caso do código da figura 4.1, se o local de ponto de controlo for o método da linha 05, então, temos 10 pontos seguros neste local. E por fim, o terceiro critério refere-se à dispersão dos pontos seguros em relação a toda a execução aplicacional. O ideal é que os pontos seguros estejam distribuídos uniformemente ao longo de toda a execução. Vejamos o seguinte exemplo: uma aplicação demora dez horas em execução e existe um local candidato que possui os seus pontos seguros na primeira hora de execução. Este candidato não será um bom local para a colocação do ponto de controlo, uma vez que todos os seus pontos seguros se situam no início da aplicação. A tarefa de selecção do local indicado para a colocação do ponto de controlo pode-se revelar demorada e gerar ambiguidades. Por estas razões, definimos na ferramenta de análise dinâmica métricas (explicadas na secção 4.5) para encontrar o melhor local, entre um leque de locais candidatos.

Apesar da dificuldade, ou até mesmo impossibilidade em alguns casos, o ideal seria possuir dados estatísticos sobre a probabilidade de erro em diferentes intervalos de tempo de execução da aplicação. Quando ocorre aumento da probabilidade de erro, deverá-se, consequentemente, aumentar a frequência de GPC.

Mediante a escolha do local do ponto de controlo poderá ser necessário efectuar modificações ao código fonte. Se o próprio local do ponto de controlo for um método da aplicação, então, não será necessária nenhuma alteração ao código fonte. Por outro lado, se o local do ponto de controlo for um bloco de instruções, cabe ao programador criar um novo método chamado *safe_point(..)*, composto pelo bloco de instruções. Por exemplo,

se no código da figura 4.1 o local do ponto de controlo fosse o corpo do ciclo *while*, então, o programador teria que alterar o código desta figura para o código da figura 4.2.

```
00: ...
01: x = 10;
02: while(x > 0){
03:     safe_point(x);
04:}
05: ...
```

FIGURA 4.2: Exemplo de código depois de definido o local do ponto de controlo.

Na secção 4.3.1 é indicado, passo a passo, como o programador deverá proceder às alterações no código base.

4.2.2 Conteúdo do ponto de controlo

Como se pode verificar na subsecção anterior, os pontos seguros são sempre pontos no ínicio da execução de métodos. Durante a GPC deverão ser gravadas todas as variáveis alteradas entre pontos seguros, com excepção às locais ao ponto seguro. Para além destas, deverão ser gravadas todas as variáveis que durante a execução da aplicação guardem valores lidos de entrada. Pode-se gravar também (apesar de dispensável) o estado relacionado com temporizadores, desde que seja possível interceptar o valor do temporizador durante a GPC, e ter acesso ao seu construtor durante o restauro da aplicação. As variáveis de entrada serão automaticamente gravadas pela ferramenta, e o programador ficará apenas responsável por declarar as variáveis que deverão ser gravadas no ponto de controlo. Esta declaração é realizada num módulo próprio fornecido pela ferramenta. Em relação ao estado aplicacional é detalhado na secção 4.3 como a ferramenta de PCR lida com esta questão.

4.2.3 Frequência de ponto de controlo

A frequência do ponto de controlo deverá ter em conta o tempo de execução da aplicação, o tempo de GPC, a probabilidade de ocorrência de falha e a importância do processo. Vejamos o seguinte exemplo: Um dado programa com uma probabilidade de falha de 1%, que demora 20 minutos a ser executado e onde o custo adicional do PCR com uma única GPC é de 5 minutos permite obter um ganho de 10 minutos, em caso de falha, utilizando o PCR. Neste caso, não compensa realizar qualquer GPC, pois se executarmos 100 vezes o programa, realizando apenas uma GPC por execução, teremos

no total um custo adicional de 500 minutos, porém, o ganho global do uso do PCR foi apenas de 10 minutos.

Com o intuito de ajudar o programador, definimos uma fórmula simples para calcular a frequência máxima que deverá ser utilizada, sem comprometer a performance da aplicação.

Seja $Cpc(f) = Cp + (Cgpc * f)$ a função que nos dá o custo adicional da realização do ponto de controlo, sendo Cp o custo do ponto de controlo per si (sem GPC), $Cgpc$ o custo de GPC e f a frequência de gravação do mesmo. Então, compensa fazer ponto de controlo para uma frequência f se:

$$Cpc(f) \leq Pe * Gr$$

onde Pe é a probabilidade de ocorrer falha e Gr o ganho de tempo obtido quando restaurada a aplicação.

Para termos uma fórmula mais completa seria necessário inserir um factor de ponderação, que quantificaria o quão fulcral é o sistema não retornar do início a execução, em caso de erro.

4.2.4 Métodos a ignorar durante o restauro

Cabe ao programador analisar e especificar, num módulo próprio fornecido pela ferramenta, os métodos ignoráveis durante o processo de restauro. Poderão ser ignorados:

- Todos os métodos que apenas modificam variáveis que estão presentes nos FPC, porque no processo de restauro será carregado dos FPC um estado mais recente destas variáveis;
- Todos os métodos de saída ou de apenas leitura de dados, pois estes não modificam o estado aplicacional, sendo desnecessária a sua execução;
- Todo o conteúdo dos pontos seguros onde se realiza a GPC. A própria ferramenta de PCR ignora os pontos seguros durante o restauro, sem necessidade de especificação por parte do programador. São ignorados todos os pontos seguros até chegar ao ponto seguro que corresponde à posição da linha de recuperação gravada em ficheiro. Nos FPC está gravado o conteúdo relacionado com o ponto seguro, portanto não é necessário reconstruir o estado produzido por este, pois o estado anteriormente produzido é reconstruído, carregando-o do disco.

4.3 Descrição da abordagem

Optou-se por uma estratégia de concepção da ferramenta de baixo para cima (*bottom-up*), começando por implementar a ferramenta para um sistema sequencial, evoluindo-a e acrescentando-lhe novos módulos para SMP e SMD. Decidiu-se dividir a ferramenta em três pacotes, um por cada sistema, obtendo assim uma melhor estruturação do código. O pacote sequencial representa a estrutura base da ferramenta, podendo-se acrescentar sobre este outros módulos, de forma a adaptar a ferramenta às condições impostas pelo tipo de sistema em que a aplicação está a ser executada.

4.3.1 Modificações ao código base

Um dos objectivos da nossa abordagem é ser o mínimo intrusiva possível. Infelizmente, existem situações em que o programador necessitará de efectuar alterações no código base da aplicação. Estas situações surgem de restrições impostas pela POA, sentidas na especificação do local e conteúdo do ponto de controlo. Seguidamente, detalham-se as ditas situações.

4.3.1.1 Local do ponto de controlo

No local de ponto de controlo é obrigatória a presença de pontos seguros. Se este local for um método da aplicação, apenas será necessário especificar no módulo pontos seguros a assinatura deste método, não sendo necessária nenhuma alteração ao código base. Por outro lado, se este local for um bloco de instruções, cabe ao programador criar um novo método chamado *safe_point(..)*, composto pelo bloco de instruções, para que seja possível a identificação dos pontos seguros, por parte da ferramenta. Nesta situação deverão ser realizadas as seguintes modificações no código base:

Seja :

- *Var = (T,N,V)* uma dada variável em que T,N e V representam, respectivamente, o seu tipo, nome e valor;
- *Instr* a representação de uma dada instrução;
- *Bloco* = $\{Instr_0, \ldots, Instr_n\}$ a representação de um bloco de código composto por n instruções, com $n \in \mathbf{N}$;
- *Cond* a representação de uma condição;

- *f(..)* o local de colocação do ponto de controlo, *argsF* os argumentos do método f e *argsSP* os argumentos do método *safe_point(..)*.

Com:

f(argsF){
$Bloco_1$
Repetir $Bloco_2$ enquanto $Cond_1$
$Bloco_3$
}

Os blocos de código *Bloco1*, *Bloco2* e *Bloco3* representam, respectivamente, o bloco de código antes, durante e depois de um ciclo, enquanto *Cond1* a condição do mesmo. Inicialmente, são feitas as seguintes duas transformações :

1. f(argsF){
   $Bloco_1$
   Repetir safe_point(argSP) enquanto $Cond_1$
   $Bloco_3$
   }
   com argSP= $\{N|(Var = (T, N, V) \in Bloco_2)\}$
2. safe_point(argSP){
   $Bloco_2$
   }
   com argSP= $\{(T, N)|(Var = (T, N, V) \in Bloco_2)\}$

As figuras 4.3 e 4.4 ilustram um exemplo de código base, antes e depois das transformações, respectivamente. Posteriormente, poderá ser necessário realizar mais transformações ao código base, caso se verifiquem as seguintes condições:

Condição 1 Existência de pelo menos uma variável local primitiva presente no bloco de código, posterior ao ciclo do método f ($Bloco_3$), cujo valor tenha sido alterado no método *safe_point*.

Condição 2 Existência de pelo menos uma variável local primitiva, passada como argumento do método *safe_point*, cujo o valor altera a cada iteração do ciclo, devido às instruções existentes no $Bloco_2$.

Antes das transformações realizadas (figura 4.3), após terminado o ciclo (linha 03) as variáveis x e y tomavam ambas o valor 10, mas após as transformações realizadas (figura 4.4), as variáveis x e y após terminado o ciclo (linha 03) são ambas iguais a 0.

```
01: public void f(){
02:    int x = 0, y = 0, z = 10;
03:    while(--z != 0){
04:       if(x == 1) y = 10;
05:       x++;
06:     }
07:    System.out.println(y);
08: }
```

FIGURA 4.3: Antes das transformações ao código base.

```
01: public void f(){
02:    int x = 0, y = 0, z = 10;
03:    while(--z != 0){
04:       safe_point(x,y);
05:     }
06:    System.out.println(y);
07: }
08: public void safe_point(int x,int y){
09:       if(x == 1) y = 10;
10:       x++;
11:  }
```

FIGURA 4.4: Após as transformações ao código base.

No caso de se verificarem as condição 1 e 2, então, as seguintes alterações ao método *f(..)* deverão ocorrer :

1. Criação de um novo objecto composto pelas variáveis que obedecem às Condição 1 e 2, que por motivos de ilustração se intitulam por *Global* e *Var_Z*, respectivamente. Então, temos que $Global = \forall(T, N, V_n) \in Var\_Z$, em que V_n representa o último valor tomado pela variável antes do início do ciclo do método f(..);

2. É introduzida a instrução *Global global = new Global(Var_Z)*;, em que *global* é o nome do novo objecto;

3. Todas as variáveis de *Var_Z* presentes no $Bloco_2$ e $Bloco_3$ são substituídas pela variável correspondente no objecto *Global*, passando a ser intitulados de $Bloco_{2'}$ e $Bloco_{3'}$, respectivamente;

4. Argumentos do método safe_point passam a ser:
   argSP' = (global,$\{N|(Var = (T, N, V) \in Bloco_2) \bigwedge Var \notin Var\_Z\}$*).*

Consequentemente, f será :

f(argsF){
    $Bloco_1$
    Global global = new Global(Var_Z);
    Repetir safe_point(argSP') enquanto $Cond_1$
    $Bloco_{3'}$
}

Tais transformações são necessárias, uma vez que as variáveis em *Var_Z* são locais e do tipo primitivo e como têm o seu valor alterado dentro do novo método (*safe_point*), este valor não seria visível fora do método f, razão pela qual estas variáveis são aglomeradas dentro de um novo objecto. As figuras 4.5 e 4.6 ilustram o resultado da aplicação destas novas transformações, sobre o exemplo apresentado na figura 4.3.

```
01: public classe Global{
02:    public int x;
03:    public int y;
04:
05:    public Global(int x, int y){
06:          this.x = x;
07:          this.y = y;
08: }
09:}
```

FIGURA 4.5: Classe composta pelas variáveis conflituosas.

```
01: public void f(){
02:    int x = 0, y = 0, z = 10;
03:    Global global = new Global (x,y);
04:    while(--z != 0){
05:       safe_point(global);
06:      }
07:    System.out.println(global.y);
08: }
09: public void safe_point(Global global){
10:      if(global.x == 1) global.y = 10;
11:      global.x++;
12:  }
```

FIGURA 4.6: Código base após introdução do ponto de controlo.

A pergunta que surge é: uma vez que poderá ser necessário realizar as alterações descritas, porquê transformar o bloco de código num novo método? A razão desta estratégia prende-se com umas das restrições da POA. Com a POA não é possível aceder às variáveis locais de um método, mas é possível detectar a sua chamada, o valor que

este retorna e seus argumentos, portanto, ao transformar o fluxo de código num método é permitido encapsular este código e obter acesso às suas variáveis, pois estas, como já foi explicado, são passadas como argumentos no novo método.

4.3.1.2 Conteúdo do ponto de controlo

Durante o processo de ponto de controlo, sempre que forem interceptadas alocações de memória de objectos, os quais o programador pretende que sejam gravados durante o ponto de controlo, a ferramenta de PCR guarda os apontadores dos mesmos.

No caso do programador pretender guardar dois ou mais objectos não públicos, do mesmo tipo, pertencentes à mesma classe, e não existirem no código fonte métodos que retornem o seu apontador, será, então, necessário realizar algumas adaptações. O problema do caso descrito é a impossibilidade de intercepção dos objectos, utilizando o construtor *new*, pois não é possível diferenciá-los recorrendo ao seu tipo, ou à classe onde foram criados. A outra opção seria recorrer a um método que retorne o apontador deste objecto, mas uma das premissas do caso descrito é a inexistência de tal método. Por exemplo, este conflito sucederia caso o programador pretende-se guardar os objectos primos e perfeitos apresentados na figura 4.7.

```
01: public class Teste {
02:   private int [] primos;
03:   private int [] perfeitos;
04:   private int ciclos;
05:
06:   public void calculaPrimos(){..}
07:
08:   public void calculaPerfeitos(){..}
09: }
```

FIGURA 4.7: Problema do conflito entre objectos do mesmo tipo.

Uma solução, para este problema, passa pelo programador no código fonte da aplicação, criar um método, para cada um dos objectos visados, que retorne o apontador dos mesmos. No caso de existirem 'N' objectos com o conflito descrito, bastará aplicar a solução proposta sobre 'N-1' objectos, podendo um deles ser interceptado utilizando o construtor *new*. Por exemplo, no caso apresentado para a figura 4.7, bastaria acrescentar na classe *Teste* o método da linha 01 ou o método da linha 02, da figura 4.8.

No caso do programador pretender armazenar variáveis primitivas globais (por exemplo, linha 04 da figura 4.7), como o JAVA não permite aceder ao endereço de memória

```
01:  public int [] getPrimos()     {return this.primos;}
02:  public int [] getPerfeitos() {return this.perfeitos;}
```

FIGURA 4.8: Resolução do conflito entre objectos do mesmo tipo.

das mesmas, é necessário efectuar algumas alterações ao código base. Tira-se de pressuposto, nesta fase, que o programador já especificou o local do ponto de controlo e que já procedeu, caso necessário, às alterações apresentadas na subsecção anterior, assim sendo:

Seja *f(argsF)* o local de colocação do ponto de controlo, *argsF* os argumentos do método f e *Var_Z* o conjunto de variáveis primitivas globais que o programador pretende armazenar. Os argumentos do método *f* passaram a ser *argsF' = (argsF* $\bigcup$ *Var_Z)*. Desta forma, durante o ponto de controlo, quando o método *f(argsF')* for invocado, a ferramenta de PCR irá interceptar os argumentos deste método e proceder à gravação dos mesmos em disco.

4.3.2 Aplicações sequenciais

Durante o processo de ponto de controlo, em aplicações sequenciais, sempre que o mecanismo de PCR detectar o local do ponto de controlo, será contabilizado o número de pontos seguros interceptados. A GPC será realizada de 'x' em 'x' pontos seguros interceptados (sendo 'x' a frequência de GPC definida). Por fim, quando for interceptado o término da execução aplicacional, a indicação do sucesso da aplicação será gravada em ficheiro, terminando assim, o processo de ponto de controlo.

Durante o processo de restauro, em aplicações sequenciais, sempre que forem interceptados métodos definidos como ignoráveis, a sua execução será ignorada, sendo os restantes métodos executados. Quando ocorrer a interceptação do local do ponto de controlo, será ignorada a execução de 'n' pontos seguros, sendo 'n' o número de pontos seguros interceptados (na execução onde a aplicação falhou) até ao momento da última GPC, sendo posteriormente carregada do disco a linha de recuperação gravada. Por fim, termina-se o processo de restauro e inicia-se o processo de ponto de controlo.

Na GPC deverá ser armazenado em ficheiro o estado aplicacional suficiente para a correcta recuperação da aplicação em caso de falha, nomeadamente, *heap*, pilha, variáveis de instância e variáveis locais. Não será obrigatório gravar por completo a *heap* em ficheiro, restringindo o armazenamento dos objectos ao estritamente necessário, optimizando assim os FPC. Será também gravada a informação referente à linha de recuperação, nomeadamente, o número de pontos seguros interceptados, para que em caso de falha,

seja identificado o local exacto da última linha de recuperação correctamente gravada. Durante o processo de restauro, os objectos são carregados do FPC, sendo a *heap* actualizada de seguida. Em relação à pilha, como o ponto de controlo só é realizado nos pontos seguros, apenas é necessário manter o rastreio das chamadas aos métodos que estão presentes na pilha, durante a execução destes pontos. Por esta razão, durante o processo de restauro é efectuada a reconstituição da pilha, através da não execução dos métodos ignoráveis e da execução dos restantes. Em relação às variáveis de instância, serão armazenadas apenas as alteradas entre pontos seguros e as que contêm valores lidos de entrada. No que toca às variáveis locais, serão somente carregadas aquelas que contêm valores lidos de entrada, enquanto as restantes serão recriadas, durante o processo de restauro, pelo código original que as criou.

4.3.3 Aplicações de memória partilhada

Para adaptação do caso base (sequencial) para SMP, foi necessário acrescentar novas restrições/funcionalidades. Tendo em conta a existência de múltiplas LE, de estado escondido (estado relacionado com a API de criação das LE), estado de sincronização (barreiras ou *locks*) e o facto do estado aplicacional poder ser dividido em conteúdo partilhado ou em conteúdo privado, o conteúdo partilhado é visível a todas as LE, enquanto o privado apenas à LE que o gera. A *heap* trata-se de conteúdo partilhado e a pilha, por sua vez, de conteúdo privado (existindo uma pilha por LE). Uma vez que existe uma única *heap* para todas as LE, a ferramenta trata-a da mesma forma que no ambiente sequencial, tal como descrito anteriormente, ficando a cargo da LEM a responsabilidade de guardar os apontadores dos objectos partilhados que irão ser gravados durante o ponto de controlo.

O processo de restauro em SMP é realizado de forma idêntica ao realizado em ambiente sequencial, sendo que os métodos (da aplicação) criadores das LE são executados de forma a reconstruir o número de LE, respectivas pilhas, variáveis locais e estado escondido das mesmas.

A existência de variáveis privadas exigiu a criação de uma estrutura de dados por cada LE, onde serão armazenados os apontadores das variáveis privadas. Desta forma, durante a GPC estas variáveis poderão ser gravadas, recorrendo-se para tal, à sua localização na memória, que se encontra armazenada nas estruturas de dados.

Relativamente ao estado de sincronização, não é tomada qualquer providência por parte da ferramenta, pois como já referido, o ponto de controlo é realizado em locais definidos como pontos seguros, ou seja, durante a GPC os únicos mecanismos de sincronização activos são os da própria ferramenta de PCR.

A GPC é realizada utilizando um protocolo coordenado bloqueante, com o propósito de garantir exclusão mútua dos dados a serem armazenados, mantendo a consistência dos mesmos. Neste protocolo, cada LE chama uma barreira antes e depois da GPC. Durante a GPC, cada LE grava o seu estado privado, sendo que a LEM grava também o estado partilhado.

No restauro, todas as LE carregam dos FPC o seu estado privado, enquanto a LEM carrega o estado partilhado. Após esse carregamento e antes de prosseguirem com a sua execução, cada LE chama uma barreira, de forma a garantir o restauro de todo o estado aplicacional, antes que qualquer LE lhe possa aceder.

4.3.4 Aplicações de memória distribuída

Em SMD, cada processo realiza o PCR, tal como se tratasse de um ambiente sequencial. Durante a GPC é garantido que cada processo realiza o ponto de controlo no mesmo ponto seguro, contabilizando para tal o número de pontos seguros por si interceptados. Como os locais dos pontos seguros são os mesmos em todos os processos, basta realizar o ponto de controlo após a intercepção de 'n' pontos seguros, sendo 'n' o mesmo valor em todos os processos.

A GPC é realizada utilizando um algoritmo coordenado não bloqueante centralizado, no qual cada processo envia os seus dados distribuídos ao processo mestre, para que este proceda à GPC. Durante o restauro da aplicação, o processo mestre carrega o estado aplicacional, distribuindo-o pelos restantes processos, que retornam à execução normal da aplicação, assim que a distribuição de dados estiver concluída. A GPC é realizada em pontos onde é garantida a não existência de mensagens em-trânsito ou órfãs, comunicações colectivas em aberto e onde não existem mensagens presentes no *buffer*. Como tal, não existe a necessidade de a ferramenta se preocupar com o estado escondido, com os canais de comunicações não-FIFO ou com as mensagens pendentes. Tal é possível porque, durante a GPC, os processos são bloqueados em pontos considerados pontos seguros, e são todos obrigados a participar na GPC.

4.3.5 Aplicações híbridas

Neste ambiente cada processo irá realizar PCR, tal como se de um SMP se tratasse. Durante a GPC, o envio dos dados ao processo mestre, por parte dos restantes processos, é realizado pelas LEM de cada um destes processos. O mesmo acontece durante o processo de restauro, no qual o processo mestre irá enviar os dados carregados dos FPC para as

LEM dos restantes processos. Tanto o envio como a recepção dos dados, por parte do processo mestre, será realizado pela sua LEM.

4.3.6 Adaptação entre diferentes configurações

A nossa abordagem permite, no caso de não existir gravação de estado privado por parte das LE ou por parte dos processos, o restauro da aplicação utilizando os FPC em qualquer um dos sistemas suportados (sequencial, SMP, SMD e híbridos). Portanto, é possível restaurar um SMD utilizando os FPC criados num SMP. Tal é exequível, uma vez que o conteúdo do ponto de controlo é o mesmo nos diversos ambientes. Utilizando esta abordagem é possível realizar uma adaptação estática aos recursos, realizando para tal a GPC e restaurando a aplicação numa configuração diferente.

4.4 Implementação com POA

Neste secção, iremos explicar a implementação, utilizando POA, das abordagens descritas anteriormente, para os diferentes ambientes. Serão apresentados exemplos de código em POA de forma a ilustrar os algoritmos utilizados nos diferentes ambientes. Comecemos por visualizar, na figura 4.9, a estrutura da ferramenta:

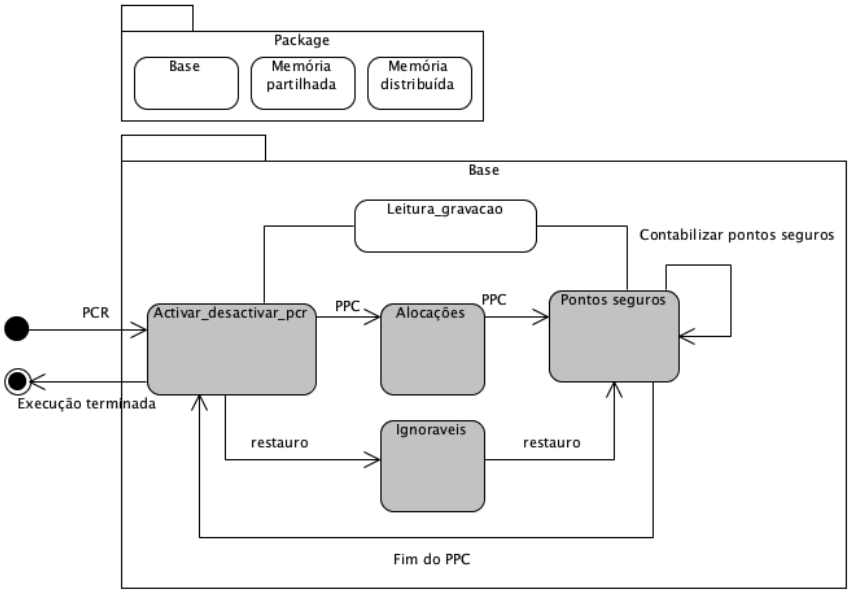

FIGURA 4.9: Base comum aos três ambientes (aspectos/*pointcuts*).

Na figura 4.9, PPC é a abreviatura de processo de ponto de controlo. O separador com a etiqueta *Package* representa o conjunto de pacotes disponíveis na ferramenta. O pacote *Base* é de carácter obrigatório, enquanto os restantes são opcionais. Cabe

ao programador decidir, mediante o tipo de ambiente onde será utilizada a ferramenta de PCR, que pacote(s) necessita de importar, podendo optar por uma das seguintes combinações:

- *Base*, no caso de uma aplicação sequencial;
- *Base + Memória partilhada*, no caso de um SMP;
- *Base + Memória distribuída*, no caso de um SMD;
- *Base + Memória partilhada + Memória distribuída*, no caso de um híbrido.

4.4.1 Módulos base

4.4.1.1 Aspecto de activação/desactivação do PCR

O aspecto *Activar_desactivar_pcr* (figura 4.10), responsável pela activação/desactivação do PCR, é composto por um *pointcut* (linha 02), para o qual existem dois *advice's* distintos (linha 05 e linha 11).

```
01: public aspect Activar_desactivar_pcr  {
02:   pointcut detectarMain() :
03:      execution(public static void *.main(..));
04:
05:  before() : detectarMain(){          // Comeco da aplicacao
06:   if(falha_na_ultima_execucao())
07:      activar_restauro();
08:   else activar_ponto_controlo();
09:  }
10:
11:   after() : detectarMain(){         // Termino da aplicacao
12:     sucesso();
13:   }
14: }
```

FIGURA 4.10: Codificação do aspecto de activar e desactivar o pcr.

O primeiro *advice* intercepta o início da execução do método *main(..)* e verifica se ocorreu alguma falha na última execução (linha 06). Em caso de falha é activado o processo de restauro (linha 07), caso contrário é activado o processo de ponto de controlo (linha 08). Por sua vez, o segundo *advice* intercepta o fim da execução do método *main(..)* e grava em ficheiro um sinal de sucesso aplicacional (linha 12). Sempre que seja iniciada a aplicação, o *advice* da linha 05 lê este ficheiro (linha 06) e activa um dos dois processos possíveis.

4.4.1.2 Aspecto das alocações de objectos

O aspecto *Alocações* (figura 4.11), contém as referências aos dados a serem armazenados durante a GPC, permitindo ao utilizador especificar o conteúdo do ponto de controlo.

```
01: public aspect Alocacoes {
02:  // Objecto que guarda o conteudo do ponto de controlo
03:  public ArrayList<Object> conteudoPC = new ArrayList<Object>();
04:  public ArrayList<Object> input = new ArrayList<Object>();
05:
06:
07:  pointcut guardarApontadores() :              // Especificar
08:           call(dados1) ||                     // os apontadores
09:           call(dados2) ||
10:           ...          ||
11:           call(dadosN);
12:
13:  Object around() : guardarApontadores(){     // Armazenamento
14:         Object o = proceed();                // dos apontadores
15:         conteudoPC.add(o);
16:         return o;
17:      }
18:
19:  pointcut metodosInput() :                    // Detectar Input
20:        call (metodoInput1) ||
21:        call (metodoInput2) ||
22:        ...
23:        call (metodoInputN);
24:
25:  Object around () : metodosInput(){          // Tratar do Input
26:        Object obj;
27:        if(restauro_activo()) obj = carregarInput();
28:        else                  obj = proceed();
29:        input.add(obj);
30:        return obj;
31:      }
32: }
```

FIGURA 4.11: Código relacionado com a fase de alocação de dados do ponto de controlo.

O aspecto *Alocações* pode dividir-se em duas partes:

1. Especificação dos pontos de junção do *pointcut* responsável pela intercepção das alocações dos dados que deverão ser gravados no ponto de controlo (da linha 07 à linha 17);

2. Tratamento de dados de entrada (da linha 19 à linha 31).

O programador fica apenas responsável pela especificação dos pontos de junção do *poincut* da linha 07. O ponto de junção deverá interceptar o seu construtor (*new*), ou

um método que retorne o apontador para o objecto que o programador pretende gravar durante o ponto de controlo.

Ao interceptar métodos de entrada (linha 25), é verificado se o processo de restauro está activo. Se sim, a execução do método é substituída pelo carregamento do estado gerado por este na última execução (linha 27). Caso contrário, depois da execução do método, é guardado o seu valor de retorno (linha 28). Independentemente do processo activo, é guardado o estado gerado pelo método (linha 29) e retornado o seu endereço de memória (linha 30). O seu estado não é carregado juntamente com a linha de recuperação, porque poderá ser necessário para a aplicação antes da chegada ao local do ponto de controlo. No caso das variáveis que guardam valores de entrada, não é necessária nenhuma diferenciação. Uma vez que todos os métodos de entrada são interceptados, basta realizar a gravação das suas variáveis e o carregamento das mesmas, por ordem de chegada.

4.4.1.3 Aspecto dos métodos ignoráveis

O aspecto *Ignoraveis*, codificado na figura 4.12, permite ao programador especificar quais os métodos que deverão ser ignorados durante o processo de restauro.

```
01: public aspect Ignoraveis {
02:
03:  pointcut metodos_ignoraveis() :          // Especificar
04:           call(metodo1) ||                // os metodos
05:           call(metodo2) ||                // ignoraveis
06:           ...           ||
07:           call(metodoN);
08:
09:  void around() : metodos_ignoraveis(){ // Tratamento
10:        if(restauro_activo())            //  dos metodos ignoraveis
11:          {   } // Nao executar
12:         else proceed();
13:      }
14:}
```

FIGURA 4.12: Codificação do aspecto referente aos métodos ignoráveis.

Cabe ao programador preencher o *pointcut metodos_ignoraveis()* (linha 03) com as chamadas aos métodos que poderão ser ignoradas durante o processo de restauro. Da linha 09 à linha 12 está definido o comportamento associado ao *pointcut* da linha 03. Ao ser interceptado um método ignorável, é verificado se o processo de restauro está activo (linha 10). Se sim, o método interceptado não é executado (linha 11), se não, o método é normalmente executado (linha 12). Todos os métodos de saída são, por defeito, considerados ignoráveis.

4.4.1.4 Aspecto dos pontos seguros

O aspecto *Pontos_Seguros* (figura 4.13), composto por um *pointcut* (linha 03) e um *advice* (linha 06), contém as rotinas para o tratamento de pontos seguros, cabendo ao programador definir a frequência da GPC (linha 02) e preencher a etiqueta (linha 04) do *pointcut* da linha 03 com o local que foi identificado como ponto seguro.

```
01: public aspect Pontos_Seguros {
02: int frequenciaGPC = ..;          // a definir pelo programador
03: pointcut pontoSeguro() :
04:          call (..);             //local do ponto seguro
05: ...
06: void around () : pontoSeguro(){ // processo de restauro
07:    pontos_seguros++;
08:    if(restauro_activo()){
09:      if(carregar_FPC()){
10:         carregar_ponto_controlo();
11:         desactivar_restauro();
12:      }
13:    }
14:    else{                        // processo ponto de controlo
15:         if((pontos_seguros % frequenciaGPC) == 0)
16:            gravar_ponto_controlo();
17:         proceed();
18:        }
19: }
20: ...
21:}
```

FIGURA 4.13: Codificação do aspecto relacionado com os pontos seguros.

Durante a execução da aplicação, sempre que é interceptado um ponto seguro é actualizado o seu valor (linha 07) e no caso do processo de restauro estar activo (linha 08) é verificado se está na altura de carregar os dados do FPC (linha 09). Se nos encontramos perante o ponto seguro onde foi realizada a GPC na última execução aplicacional com falhas, será carregado o estado aplicacional (linha 10) e desactivado o processo de restauro (linha 11). É de realçar que da linha 09 à linha 13 não existe o método *proceed()* da POA, portanto, a aplicação não irá executar o conteúdo dos pontos seguros interceptados. No caso do processo de restauro não estar activo (linha 14) será testado se é necessário realizar a GPC (linha 15), e em caso afirmativo procede-se à gravação (linha 16). Por fim, é executado o próprio ponto seguro (linha 17).

4.4.2 Suporte a memória partilhada e memória distribuída

De forma a conseguir adaptar o mecanismo de PCR para SMP e SMD, sem alteração da estrutura base da ferramenta, recorreu-se à criação de aspectos extra designados

Memoria_Partilhada e *Memoria_Distribuida*, respectivamente. Uma vez que em SMP e em SMD, existem, respectivamente, múltiplas LE e múltiplos processos em execução, é necessário adaptar tanto o processo de ponto de controlo como o processo de restauro, de forma a garantir a consistência dos dados guardados/recuperados.

4.4.2.1 Aspecto memória partilhada

Em SMP existem múltiplas LE e cada uma poderá ter estado privado, é necessário, portanto, adaptar o aspecto *Alocações* de forma a permitir guardar apontadores privados a uma dada LE. A GPC em SMP utiliza um algoritmo bloqueante, de forma a garantir a consistência dos dados guardados, e como tal é necessário adaptar também o processo de GPC. É igualmente necessária a adaptação do processo de restauro, de forma a permitir o carregamento dos dados privados, por parte das LE, e a introdução de uma barreira imediatamente a seguir ao carregamento de todo o estado aplicacional.

Por questões de legibilidade subdividimos o aspecto *Memoria_Partilhada* em quatro partes (figura 4.14 à figura 4.17). A figura 4.14 apresenta a declaração dos *pointcuts*, enquanto as restantes expõem as adaptações realizadas nos aspectos base.

```
01: public aspect memoria_partilhada {
02:
03: pointcut actPCR()   : call (public void activar_restauro ())
04:                     || call (public void activar_ponto_controlo());
05: pointcut getAloc()  : call (public void guardar_apontadores());
06: pointcut gpc()      : call (public void gravar_ponto_controlo ());
07: pointcut loadFPC()  : call (public void carregar_ponto_controlo());
08: ...
```

FIGURA 4.14: Declaração dos *pointcuts* do aspecto memória partilhada.

Cada um dos *pointcuts* da figura 4.14 tem como função adicionar rotinas extras ao pacote base (sequencial), de forma a adaptar o mecanismo de PCR a SMP. O primeiro (linha 03 à linha 04) e o segundo (linha 05) adaptam os aspectos *Activar_desactivar_pcr* e *Alocações*, respectivamente, enquanto os últimos dois (linha 06 e linha 07) realizam adaptações ao aspecto *Pontos_Seguros*.

```
09: after () : actPCR{        // Adaptar aspecto Activar_desactivar_pcr
10:     criar_instancias();
11:     definir_mestre();
12:   }
```

FIGURA 4.15: Especificação das adaptações ao aspecto de activar e desactivar o PCR.

O *advice* da figura 4.15 detecta o término da activação de um de dois processos possíveis, por parte do aspecto *Activar_Desactivar_PCR*. Este *advice* intercepta a chamada do método *activar_restauro()* (linha 07 da figura 4.10) ou *activar_ponto_controlo()* (linha 08 da figura 4.10), criando instâncias (linha 10) com o intuito de guardar futuramente informações referentes às LE, nomeadamente, o seu identificador, o número de pontos seguros que esta já interceptou e os apontadores para as variáveis privadas que farão parte do conteúdo do ponto de controlo. De seguida, define-se a LE actualmente em execução como sendo a LEM. (linha 11).

```
13:
14: void around () : getAloc(){ // Adaptar aspecto Alocacoes
15:       if(dados_privados()) apontadores_privados();
16:       if(dados_partilhados()){
17:         if(mestre())
18:          proceed();
19:       }
20:    }
21: ...
```

FIGURA 4.16: Especificação das adaptações realizadas ao aspecto de alocações.

O *advice* da figura 4.16 detecta o armazenamento de apontadores no aspecto *Alocações*, verificando primeiramente se tais apontadores são privados a uma LE em específico, ou se são comuns a todas as LE. No caso de serem privados (linha 15), estes serão guardados na instância da LE a que se referem. Por outro lado, se estivermos perante apontadores partilhados (linha 16), estes serão guardados no próprio aspecto *Alocações* (linha 18) pela LEM (linha 17).

```
22:
23: void around () :  gpc() {     // Adaptar a GPC
24:       chamar_barreira();
25:       guardar_dados_privados();
26:       if(linha_mestre()) proceed();
27:       chamar_barreira();
28:      }
29: ...
30:
31: void around () :  loadFPC() { // Adaptar o carregamento de dados
32:       carregar_dados_privados();
33:       if(linha_mestre()) proceed();
34:       chamar_barreira();
35:  }
```

FIGURA 4.17: Especificação das adaptações realizadas ao aspecto referente aos pontos seguros.

Ambos os *advice's* especificados na figura 4.17 (linha 23 e linha 31) interceptam os métodos do aspecto *Pontos_Seguros*, nomeadamente o método de GPC (linha 16 da figura 4.13) e o método de carregamento da linha de recuperação (linha 10 da figura 4.13). O primeiro *advice* introduz, antes e depois da GPC, uma barreira (linha 24 e linha 27). Antes do método interceptado ser executado pela LEM (linha 26), cada LE grava o seu estado privado (linha 25). No segundo *advice*, cada LE carrega o seu estado privado (linha 32) antes do carregamento de dados pela LEM (linha 33). Após terminado o carregamento de dados cada LE chama uma barreira (linha 34).

4.4.2.2 Aspecto memória distribuída

O aspecto *Memoria_Distribuida* permite adaptar: 1) PCR sequencial em PCR distribuído; 2) PCR memória partilhada em PCR híbrido. Cada processo irá realizar o PCR como se de um sistema sequencial (1) ou de um SMP (2) se trata-se. Para ambos os casos, o aspecto *Memoria_Distribuida* insere rotinas que obrigam os processos a pararem no mesmo ponto seguro para a realização da GPC. No entanto, no último caso são inseridas precedências, porque os aspectos *Memoria_Distribuida* e *Memoria_Partilhada* interceptam os mesmos métodos presentes no aspecto *Pontos_Seguros*. A prioridade será dada ao aspecto *Memoria_Partilhada*. Vejamos a codificação do aspecto *Memoria_Distribuida* ilustrada na figura 4.18.

```
01: public aspect memoria_distribuida {
02:
03:
04: pointcut gpcDistr()  : call (public void gravar_ponto_controlo ());
05: pointcut loadDistr() : call (public void carregar_ponto_controlo());
06:
07:
08: void around () :  gpcDistr() { // Adaptar a GPC
09:       unirDados();
10:       if(processo_mestre()) proceed();
11:       else{}
12:     }
13: ...
14:
15: void around () :  loadDistr()  {   // Adaptar o carregamento de dados.
16:       if(processo_mestre()) proceed();
17:       else {}
18:       distribuirDados();
19:    }
20:}
```

FIGURA 4.18: Código relacionado com a implementação de rotinas para suportar SMD.

Os *pointcut's* da linha 04 e linha 05 interceptam, respectivamente, o método de GPC e o método de carregamento da linha de recuperação, presentes no aspecto *Pontos_Seguros*.

O *advice* da linha 08, antes de permitir a execução do método de GPC, por parte do processo mestre (linha 10), vai unir os dados distribuídos entre os diversos processos (linha 09). Só após o processo mestre receber os dados dos restantes processos é que será possível a execução do método interceptado, por parte do processo mestre. No *advice* da linha 15, o processo mestre carrega os dados (linha 16), distribuindo-os depois pelos restantes processos (linha 18).

```
13: ...
14:   else{ // Ponto de controlo
15:         if((pontos_seguros % frequenciaGPC == 0){
16:           chamar_barreira();
17:           guardar_dados_privados();
18:           if(linha_mestre()){
19:             unirDados();
20:             if(processo_mestre()) gravar_ponto_controlo();
21:             else{}
22:             }
23:           chamar_barreira();
24:         }
25:       proceed();
26:     }
```

FIGURA 4.19: Processo de GPC em sistemas híbridos.

A figura 4.19 mostra o resultado da adaptação da ferramenta de PCR a aplicações híbridas quando ambos os aspectos são incluídos. As linhas a negro correspondem ao código do processo de ponto de controlo em ambiente sequencial (da linha 14 à 18 da figura 4.13), enquanto as linhas vermelhas e as linhas azuis representam adaptações realizadas pelos aspectos *Memoria_ Partilhada* e *Memoria_ Distribuida*, respectivamente. Descrevem-se de seguida os passos que são realizados durante a GPC híbrida: Um *pointcut* presente no aspecto *Pontos_ Seguros* intercepta o ponto seguro e verifica se está na altura de efectuar a GPC (linha 15). De seguida, um *pointcut* presente no aspecto *Memoria_ Partilhada* (linha 06 da figura 4.14) detecta o método de GPC que iria ser executado pelo *advice* do aspecto *Pontos_ Seguros*. Em vez de efectuar já a GPC, o *pointcut* (linha 06 da figura 4.14) obriga a que todas as LE chamem uma barreira. Após a chamada da barreira por parte das LE, cada uma grava o seu estado privado. A LEM executa o método *proceed()* da POA, para que seja possível executar o método de GPC presente no aspecto *Pontos_ Seguros*. As restantes LE chamam a segunda barreira e ficam à espera da LEM. O aspecto *Memoria_ Distribuida* intercepta a chamada ao método de GPC (linha 04 da figura 4.18), por parte da LEM. Em vez de efectuar a GPC, o aspecto (linha 04 da figura 4.18) obriga a que cada processo envie os seus dados distribuídos ao processo mestre. Este envio/recepção é realizado apenas pelas LEM (únicas que acedem ao código do aspecto *Memoria_ Distribuida*) de todos os processos. Quando todos os processos terminarem o envio de dados, o processo mestre irá finalmente executar o método de GPC existente no aspecto *Pontos_ Seguros* (linha 20 da figura 4.19).

4.4.3 Vantagens e desvantagens da abordagem

Um dos objectivos fulcrais desta dissertação era o estudo da viabilidade da POA para implementar mecanismos de PCR. Esta secção discute a utilidade da POA para implementação da abordagem proposta. De uma forma mais geral, as principais vantagens proporcionadas pela sua utilização são:

1. Modularização;
2. Reutilização de aspectos;
3. Generalização da abordagem para diferentes ambientes e aplicações;
4. Flexibilidade.

A utilização da POA permitiu desenvolver módulos separados do código base, constituídos pelas rotinas do PCR, ao invés de colocar estas mesmas rotinas directamente sobre o código base, como aconteceria numa abordagem tradicional. Para além de se obter código mais estruturado e legível, este tipo de desenvolvimento facilita tarefas futuras de depuração. A utilização da POA permitiu também construir módulos que podem ser reutilizáveis em diferentes aplicações de forma semelhante a uma biblioteca. Estes módulos podem ser ligados/desligados da aplicação, possibilitando ao programador executar a aplicação com ou sem mecanismos de tolerância a faltas.

O desenvolvimento com POA possibilitou a criação de uma estrutura base, capaz de introduzir PCR em aplicações sequenciais, que facilitou, consequentemente, a adaptação da ferramenta a SMP, SMD e híbridos. Com o uso de uma estrutura base, apenas é necessária a introdução de um aspecto extra para adaptação da ferramenta de um sistema sequencial para um SMP ou SMD, sendo que quando combinados os aspectos extras dos dois últimos ambientes referidos a ferramenta permite PCR em sistemas híbridos. Conceptualmente, numa abordagem tradicional seria necessária a criação de um conjunto distinto de módulos para cada um dos quatro sistemas.

A utilização da POA na implementação das rotinas do aspecto *Activar_ desactivar_ pcr* permitiu uma adaptação e um correcto funcionamento deste aspecto em qualquer aplicação, sem a necessidade de replicação das rotinas para adaptação a novas aplicações, ou introdução das mesmas directamente sobre o código base. Para além disto, permitiu generalizar o processo de ponto de controlo para qualquer aplicação, desde de que o programador proceda à especificação do local do ponto de controlo. No que se refere ao módulo de ignorar métodos, a principal vantagem é a do programador apenas necessitar de especificar a assinatura dos métodos a ignorar. Já numa abordagem tradicional, o

programador teria de introduzir condições no código base para ignorar métodos durante o restauro. Por exemplo, imaginemos que numa dada aplicação é possível ignorar, durante o processo de restauro, um método fictício chamado *gerar_solucoes()*. Numa técnica tradicional, possivelmente o programador precisaria de fazer algo do género:

```
01: ...
02: if(!restauro){
03:    gerar_solucoes();
04:    }
05: ...
```

FIGURA 4.20: Implementação tradicional de métodos ignoráveis.

Para além de tornar o código menos legível, o programador teria de fazer condições para todos os métodos passíveis de serem ignorados. Na nossa abordagem, todo o processo de ignorar métodos está presente num só módulo e o programador apenas precisa de preencher o *pointcut* presente no aspecto *Ignoraveis* com a assinatura destes mesmos métodos.

Por fim, outra das vantagens da utilização da POA refere-se à possibilidade de executar chamadas a métodos sem executar o seu conteúdo, o que nos permitiu restaurar as chamadas aos métodos presentes na pilha, sem a necessidade de os executar, obtendo-se assim melhores tempos de execução.

Expõem-se, seguidamente, as desvantagens da implementação da POA. Uma das desvantagens relaciona-se com a monitorização dos apontadores dos dados a serem gravados durante o ponto de controlo. Numa abordagem tradicional, guardar os apontadores das variáveis é uma tarefa trivial, pois bastará aceder directamente ao valor das variáveis, sem a necessidade da criação de novos métodos, ou de intercepção de métodos existentes no código base, por parte de módulos exteriores. Na nossa abordagem existem situações em que é necessário criar novos métodos para que seja possível interceptar apontadores de determinadas variáveis (secção 4.3.1.2). Outra desvantagem é a obrigatoriedade de o ponto de controlo se situar num local considerado ponto seguro e de ser transformado num método. No que se refere ao processo de ignorar métodos, comparativamente com uma abordagem tradicional, a nossa abordagem possui uma menor flexibilidade, pois apenas possibilita ignorar métodos, enquanto numa técnica tradicional é possível ignorar qualquer tipo de instruções, desde que inseridas as devidas condições.

4.5 Ferramenta de detecção do local do ponto de controlo

Para auxiliar o programador em tarefas de análise, criou-se uma ferramenta de análise dinâmica que permite detectar o local do ponto de controlo. Esta é executada juntamente com a aplicação, sem o mecanismo de PCR activo. Se o programador pretender encontrar o local do ponto de controlo, deverá indicar as seguintes condições à ferramenta:

1. Número mínimo de pontos onde poderá ocorrer GPC;
2. Frequência de GPC (em segundos).

A ferramenta divide o processo de detecção do local de ponto de controlo em três fases. Na primeira fase, a ferramenta cria uma lista com todos os métodos que respeitam a restrição 1, ou seja, métodos candidatos à colocação do ponto de controlo. Sugerimos que o número mínimo de pontos de controlo deverá ser fixado entre 10 a 100 vezes o número de GPC por execução, de forma a garantir uma maior uniformidade na distribuição dos pontos de controlo pela execução aplicacional.

Seja $Texe$ o tempo de execução da aplicação e Fo a frequência definida pelo programador. Uma vez que a GPC deverá ocorrer idealmente em múltiplos de Fo, podemos definir $Gpc = \{1Fo, 2Fo, \ldots, nFo\}$ com $n \in \mathbf{N} \bigwedge nFo < Texe$ como o conjunto composto por todos os momentos onde deverá ocorrer GPC. Visto que se utilizou uma abordagem para PCR ao nível aplicacional, nem sempre é possível efectuar GPC exactamente em múltiplos de Fo, pois a frequência Fo está expressa em unidade de tempo. Em vez disto, a GPC ocorre em locais de código específicos. Assim, definimos:

$$Igpc = \{[Fo - \frac{Fo}{2}, Fo + \frac{Fo}{2}[, [2(Fo - \frac{Fo}{2}), 2(Fo + \frac{Fo}{2})[, \ldots, [n(Fo - \frac{Fo}{2}), n(Fo + \frac{Fo}{2})[\}$$

com $n \in \mathbf{N} \bigwedge nFo < Texe$, sendo o conjunto composto pelos intervalos de tempo onde poderá ocorrer a GPC. Na segunda fase, a ferramenta irá listar por ordem decrescente os métodos (transitados da fase 1) que possuem o maior número de intervalos de tempo (pertencentes ao conjunto $Igpc$) com pontos onde poderá ser realizada a GPC. No caso da ocorrência de métodos empatados em primeiro lugar, a ferramenta passa à terceira fase, e se tal não se verificar o método em primeiro lugar será o local do ponto de controlo. Na terceira fase, é calculado para cada método a função $Disp() = aux(Fo, Fo)$, sendo aux a seguinte função:

$$aux(Fo, T) = \begin{cases} SpP(T) + aux(Fo, T + Fo) & \text{se } T < Texe \\ 0 & \text{se } T \geq Texe \end{cases}$$

onde T é um múltiplo de Fo e $SpP(T)$ é a função que devolve o tempo (em segundos) do ponto de controlo mais próximo de T que pertença ao método interceptado. A função $Disp()$ devolve para cada método (transitado da fase 2) o somatório dos tempos dos pontos de controlo mais próximos de cada um dos elementos do conjunto Gpc. Será escolhido o método para qual a função $Disp()$ devolve o menor valor. Caso os valores atribuídos pela função $Disp()$ aos métodos da terceira fase apresentem uma diferença insignificante entre si, o programador poderá optar pelo método com o menor número de pontos de controlo. Quanto menos pontos de controlo tiver um método, menos intercepções serão efectuadas pela ferramenta de PCR, portanto, menor será o custo adicional do PCR.

4.6 Adaptação dinâmica

Criou-se um mecanismo de adaptação dinâmica a recursos em SMP, que é executado em simultâneo com a aplicação, e que é capaz de adaptar a aplicação aos recursos disponíveis num dado momento da execução. Se, por alguma razão, os recursos disponíveis num dado sistema variarem durante a execução aplicacional, o mecanismo de adaptação dinâmica deverá ser capaz de adaptar a execução aplicacional de acordo com os novos recursos disponíveis, de forma a optimizar a performance da mesma.

O nosso mecanismo de adaptação dinâmica aos recursos possui duas técnicas de funcionamento diferentes. Nas duas técnicas a aplicação é iniciada com um número de LE maior que o de recursos disponíveis, e a região paralela (região do programa a ser executada por múltiplas LE em paralelo) é transformada num método para que possa ser detectada por parte dos *pointcuts*. As LE são divididas entre activas e adormecidas e a atribuição de identificadores às LE é realizada sequencialmente, por ordem crescente, começando no zero, sendo considerada LEM a linha com identificador zero. A primeira técnica utiliza a abordagem de sobre-decomposição, cuja implementação pode ser parcialmente observada na figura 4.21.

Em ambas as técnicas são LE activas todas as que possuírem identificadores entre zero e *linhas_activas*-1, sendo as restantes consideradas inúteis. Na primeira técnica, designada por nós de técnica das *LE adormecidas*, sempre que é interceptada a região paralela, verifica-se se a LE que a intercepta é activa (linha 08). Se sim, esta irá trabalhar na região paralela (linha 09) e caso contrário, a LE ignora a região paralela, chamando uma barreira (linha 12). Depois de executada a região paralela, as LE activas chamam também uma barreira (linha 12). Quando todas as LE (inúteis e activas) tiverem chamado uma barreira, repetem o processo anterior até não existirem mais intercepções da região paralela. De 'x' em 'x' intercepções a LEM verificará a ocorrência de alterações

```
01: ...
02: pointcut getRegiaoParelela() : call (regiao paralela);
03: private final int max_LE = // definir pelo programador
04: private int linhas_activas = // definir pelo programador
05:
06: // Inteceptacao da regiao parelela
07: void around () :  getRegiaoParelela() {
08:       if(getLE_id() < linhas_activas)
09:          proceed();
10:       if(mestre() && verificar_recursos())
11:          linhas_activas = recursos_disponiveis();
12:       barreira();
13:    }
14: ...
```

FIGURA 4.21: Código com o primeiro módulo de implementação dinâmica.

nos recursos disponíveis (linha 10) e o número de LE activas será actualizado (linha 11), caso isto se verifique.

A segunda técnica, designada por nós de técnica das *LE inactivas*, difere da primeira no comportamento das LE inúteis. Enquanto na primeira técnica as LE inúteis ignoram a região paralela, na segunda estas não serão sequer iniciadas. Na segunda técnica, todas as LE da aplicação são colocadas numa estrutura (pilha). Inicialmente, a aplicação é executada com uma percentagem das LE existentes nestas pilhas, as LE que não participam nesta execução ficam paradas na pilha, podendo ser utilizadas a qualquer altura. Sempre que a ferramenta decidir executar a aplicação com mais LE é utilizado o seguinte algoritmo: são activadas da pilha as 'n' LE do topo (sendo 'n' o novo número de LE pretendido). As LE actualmente em execução (excluíndo as 'n' novas) chamam uma barreira, esperando que as novas LE também o façam. As novas LE irão ignorar o mesmo número de execuções da região paralela que as realizadas pelas LE antigas. Quando as novas LE chegarem ao local das LE antigas chamarão uma barreira. Por fim, as LE antigas e novas trabalham conjuntamente sobre a região paralela.

Em ambas as técnicas sempre que for necessário diminuir o número de LE, a ferramenta irá colocar 'n' LE como inúteis (sendo 'n' a redução pretendida). Estas LE, consideradas inúteis, irão ignorar a região paralela, ou seja, passarão a não executar o seu conteúdo. As restantes LE procedem à execução normal da região paralela.

Capítulo 5

Avaliação e resultados

Na secção 5.1, deste capítulo, apresentam-se os casos de estudo que serviram para validar as ferramentas de PCR, de detecção do local do ponto de controlo e de adaptação dinâmica. Por sua vez, na secção 5.2 descreve-se o ambiente no qual se realizaram os testes. Na secção 5.3 são ilustrados o processo de detecção do local do ponto de controlo, o processo de identificação dos métodos ignoráveis e o processo de determinação do conteúdo do ponto de controlo. Finalmente, nas secções 5.4, 5.5 e 5.6 detalham-se os resultados dos testes referentes ao processo de ponto de controlo, ao processo de restauro e ao mecanismo de adaptação dinâmica.

5.1 Casos de estudo

Para validação das ferramentas desenvolvidas, testaram-se as mesmas nas versões sequencial, memória partilhada e memória distribuída dos casos de estudo SOR e MOLDYN, baseadas nas implementações do JGF [SBO01]. Passaremos, de seguida, a descrever, de forma sucinta, os casos de estudo citados.

"Successive over relaxation"(SOR) é um método interactivo de resolução das equações de *Laplace* sobre uma matriz. A versão escolhida utiliza a estratégia vermelho-preto do algoritmo para permitir paralelismo.

Na versão memória distribuída deste algoritmo a matriz é repartida em partes iguais, ficando cada parte a cargo de um processo. Na versão memória partilhada a matriz é partilhada entre as diversas LE, ficando cada uma delas responsável por um determinado conjunto de linhas da matriz. A actualização dos pontos na matriz é baseada num esquema vermelho/preto, realizando-se para tal, duas fases por iteração: uma para os pontos pretos e outra para os pontos vermelhos. Na versão memória distribuída, durante cada iteração, os processos trocam os limites dos seus blocos de dados com os seus dois vizinhos. No fim de cada iteração, em ambas as versões, é realizada uma sincronização global para que seja realizado um teste de convergência.

MOLDYN é um algoritmo de simulação de dinâmica molecular num potencial de Lennard-Jones [LJ31]. Para cada partícula, é necessário saber a sua posição, velocidade e a força que actua sobre esta. Grande parte do custo computacional do MOLDYN provém do cálculo da força que actua em cada partícula. Este cálculo envolve dois ciclos, um ciclo exterior sobre todas as partículas no sistema e um ciclo interior variando a partir do número da corrente partícula até ao número total de partículas. A cada iteração do ciclo interior é actualizada a força entre pares de partículas presentes num dado raio. No fim do ciclo exterior é actualizada a velocidade de cada partícula em função da força a que está sujeita, assim como as coordenadas de cada partícula, baseando-se na sua posição e velocidade [SBO01].

Nas versões de memória partilhada e de memória distribuída do MOLDYN, as iterações do ciclo exterior são divididas pelas LE ou pelos processos, respectivamente. A segunda versão referida tem como particularidade a utilização de uma operação de comunicação colectiva para, no final de cada iteração, difundir por todos os processos as novas posições e velocidades de cada partícula.

5.2 Ambiente

Os testes foram executados no *Cluster* SeARCH [Sea], recurso computacional da Universidade do Minho, utilizando duas máquinas AMD Opteron 6174, 2.2 GHz, 12MB L3 cache, com 24 cores cada. As aplicações foram compiladas utilizando o compilador JAVA e JVM da SUN JDK 1.5.0_02 e a versão 1.6.10 do AspectJ. A versão distribuída utilizou a openMPI + mpiJava versão 1.5 como protocolo de comunicação. Todos os resultados apresentados nas subsecções que se seguem são a mediana de 25 medições. De notar que em todos os testes realizados com 32 processos a carga de trabalho é dividida em partes iguais pelas duas máquina, ou seja, 16 processos por máquina.

5.3 Identificação dos métodos ignoráveis, do local e do conteúdo do ponto de controlo

Apresenta-se, de seguida, o processo de identificação do local do ponto de controlo por parte da ferramenta auxiliar descrita na secção 4.5, começando por apresentar um excerto do código do SOR (figura 5.1).

```
01: public class Sor {
02: ...
03: static double[][] G = new double[5000][5000];
04: ...
05: private static final void doIterations(int num_iterations) {
06:   for(int p = 0; p < num_iterations; p++) {
07:     iteration(0);     // interaccao dos elementos vermelhos
08:     iteration(1);     // interaccao dos elementos pretos
09:   }
10: }
11:
12: private static final void iteration(int is_red) {
13:   for(int row = 1; row < Mm1; row++)
14:     updateRow(row, (row+is_red)%2+1);
15: }
16:
17: static final void updateRow(int row, int start_elem) { ... }
18:
19: private static void makeMatrix(int M, int N, Equations e)  {
20:     for(int i = 0; i < M; i++)
21:       for(int j = 0; j < N; j++)
22:         G[i][j] = e.getElement(i,j);
23: }
24: ...
```

FIGURA 5.1: Código base do SOR.

Inicialmente, o método *makeMatrix(..)* (linha 19 da figura 5.1) preenche a matriz *G* com os elementos das equações a serem resolvidas. Posteriormente, o método *doIterations* (linha 05) realiza iterativamente a chamada do método *iteration*, tanto nos elementos vermelhos (linha 07) como nos elementos pretos da matriz (linha 08). Por sua vez, o método *iteration* chama o método *updateRow* (linha 14), para cada linha da matriz G, que aplica o estêncil de 5-pontos em todos os elementos da linha.

A primeira tarefa a ser realizada, antes da introdução do mecanismo de PCR, é a identificação do local do ponto de controlo. Para a resolução de tal tarefa utilizou-se a ferramenta de análise dinâmica (secção 4.5), cujos resultados estão ilustrados na figura 5.2.

```
*******Relatorio de dados*******
Tempo de execucao total = 15.226 (s)

Metodo: double Equacoes.getElement(int, int)
[Inicio,Fim]: [0.138,3.385] (s)
Tempo execucao: 3.247 (s)
Total de pontos seguros: 6250000
Intervalos com pontos seguros: 1/4
Variancia: 2.180 E-13 (s)

Metodo: void Sor.iteration(int, int, int, double, double)
[Inicio,Fim]: [3.385,15.205] (s)
Tempo execucao: 11.819 (s)
Total de pontos seguros: 100
Intervalos com pontos seguros: 4/4
Variancia: 0.103 (s)

Metodo: void Sor.updateRow(int, int, int, double, double)
[Inicio,Fim]: [3.386,15.205] (s)
Tempo execucao: 11.820 (s)
Total de pontos seguros: 1998400
Intervalos com pontos seguros: 4/4
Variancia: 0.097 (s)
```

FIGURA 5.2: Determinação do local de ponto de controlo, por parte da ferramenta de análise dinâmica.

Todas as características detalhadas na figura 5.2 encontram-se explicitadas na secção 4.5 deste documento. Antes de activar a ferramenta de análise dinâmica, é necessário definir o número mínimo de pontos seguros aceitáveis, que neste caso foi definido como 10, pois apenas pretendíamos uma única GPC por aplicação. Todos os métodos que respeitam a condição anterior são apresentados na figura 5.2. Destes métodos é inicialmente excluído para colocação do ponto de controlo o método *Equacoes.getElement(int, int)*, pois possui o menor número de intervalos de tempo com pontos seguros ("Intervalos com Pontos Seguros: 1/4"). Entre os dois últimos métodos (ambos com o mesmo número de intervalos com pontos seguros), uma vez que estes apresentam uma variância entre

pontos seguros muito similar (0,103 e 0,097), optou-se pelo que possui um menor número de pontos seguros, dado que quanto menos pontos seguros existirem num método, menos intercepções serão efectuadas pela ferramenta de PCR, e portanto, menor será o custo adicional do PCR. Então, para local de ponto de controlo foi escolhido o método *iteration(int, int, int, double, double)*.

Após a escolha do local do ponto de controlo, é altura de identificar o conteúdo do ponto de controlo e os métodos a serem ignorados. Os dados a serem armazenados serão as variáveis não locais alteradas, entre os pontos seguros presentes no local de ponto de controlo escolhido. Analisando o método *iteration* verificamos que unicamente a matriz *G* deveria fazer parte dos FPC. Por fim, os métodos ignorados poderão ser, entre outros, todos aqueles que apenas alterem os dados do ponto de controlo, como, por exemplo, os métodos *makeMatrix* e *iteration*. O primeiro pode ser ignorado porque durante o ponto de controlo é gravado um estado mais recente da matriz G do que aquele gerado por este método. Já o segundo pode ser ignorado pois o estado gerado por este é gravado periodicamente nos FPC.

Para este caso de estudo foram gerados os *pointcuts* (ilustrados na figura 5.3) *lpc()*, *data()* e *ignore()*, que representam o local do ponto de controlo, o contéudo dos FPC e os métodos ignoráveis, respectivamente.

```
01: pointcut lpc () : call(private static final void iteration(int))
02:                 && within(Sor);
03;
04: pointcut data() : call (double[][] new(..)) && within(Sor);
05:
06: pointcut ignore() :
07:          (call(private static void makeMatrix (int, int, Equations))
08:       ||  call(private static final void iteration(int)))
09:       && within(Sor);
```

FIGURA 5.3: SOR - *Pointcuts* gerados.

5.4 Processo de ponto de controlo

Nesta secção pretende-se analisar o custo adicional do processo de ponto de controlo na nossa abordagem. Para tal efectuaram-se testes de forma a medir os seguintes custos: 1) Custo de utilização de aspectos para introdução das rotinas do PCR; 2) Custo adicional do ponto de controlo, per si (sem GPC); 3) Custo adicional da GPC e sua variação mediante o tipo de versão da aplicação, nomeadamente, sequencial, memória partilhada e memória distribuída. Os resultados destes testes, apresentados nas figuras 5.4, 5.5, 5.6

e 5.7, incluem a execução sequencial, a execução em SMP com 2 até 16 LE (utilizando *threads* Java) e em SMD com 2 até 32 processos MPI no SOR e 2 até 16 processos MPI no MOLDYN, em ambos cada processo é uma instância da JVM. Por fim, é de assinalar que as versões do SOR e MOLDYN possuem, nos testes realizados neste capítulo, 100 e 50 pontos seguros, respectivamente.

De forma a determinar o custo 1), mediram-se os tempos de execução de duas versões distintas de PCR, designadas de não intrusiva e de intrusiva. Em ambas as versões são introduzidas rotinas referentes ao mecanismo de PCR. Estas rotinas na versão não intrusiva são implementadas utilizando a POA (nossa abordagem), enquanto na versão intrusiva são inseridas directamente sobre o código de origem.

Para cada uma das duas versões (intrusiva e não intrusiva) medimos os tempos de execução com e sem GPC. Por sua vez, para medição do custo 2), introduziram-se também, nesta experiência, os tempos de execução dos casos de estudo sem utilizar o PCR (original). Os resultados desta experiência encontram-se detalhados nas tabelas A.1 e A.2 do apêndice A, e estão ilustrados nas figuras 5.4 (SOR) e 5.5 (MOLDYN).

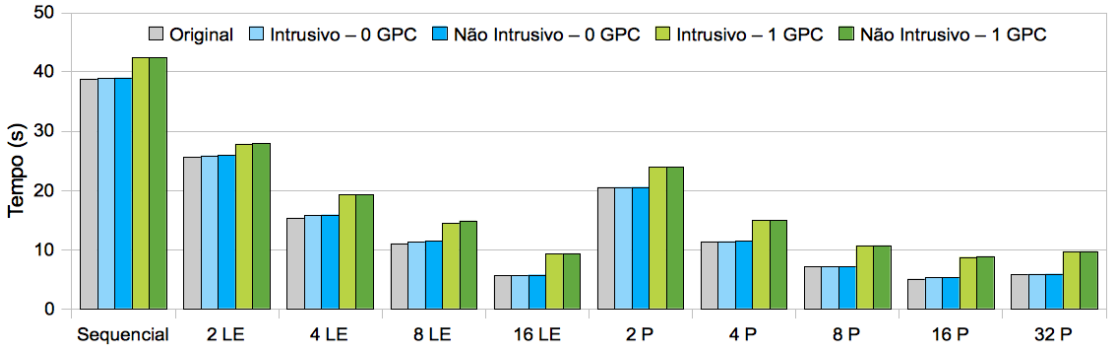

FIGURA 5.4: SOR - Comparação de abordagens.

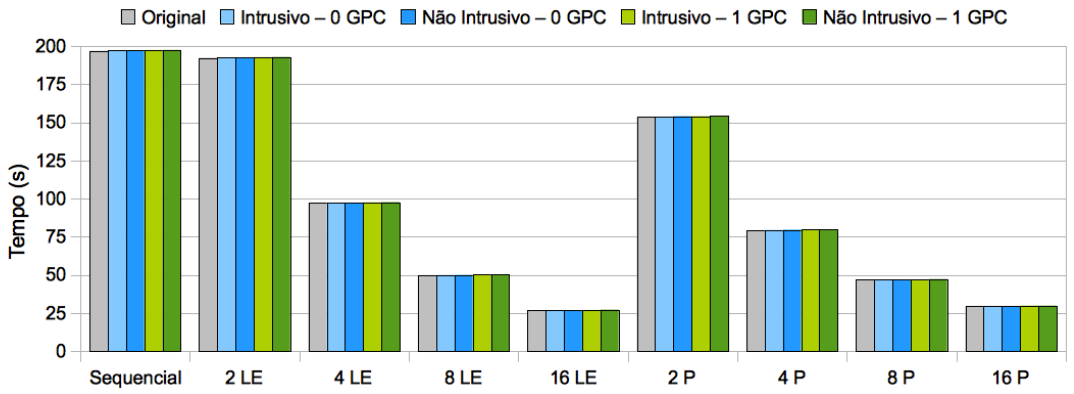

FIGURA 5.5: MOLDYN - Comparação de abordagens.

Para determinar o custo acrescido da utilização dos aspectos, basta comparar os tempos de execução sem GPC das versões intrusiva e não intrusiva. Através desta comparação

é possível concluir que os aspectos introduzem um custo adicional insignificante, pois os acréscimos máximos obtidos foram de 0,13 segundos para o SOR e 0,18 segundos para o MOLDYN.

Relativamente ao custo do ponto de controlo per si da nossa abordagem, o valor mais alto obtido para todas as versões de ambos os casos de estudo foi de apenas 0,5 segundos, valor extraído através da comparação entre os tempos da versão não intrusiva (sem GPC) e os tempos da versão original (sem PCR). Podemos constatar, portanto, que a motorização da *heap* e a contabilização do número de pontos seguros executados tem um custo adicional insignificante.

Na segunda experiência, o tempo de GPC na versão de memória partilhada, de cada caso de estudo, é contabilizado a partir do momento em que as LE dão entrada na primeira barreira até ao momento em que saem da segunda barreira, depois da LEM ter gravado os dados. Na versão distribuída é contabilizado o custo a partir do momento em que os processos escravos enviam os seus dados ao processo mestre, até ao momento em que o processo mestre acaba a gravação dos mesmos em disco. Os resultados desta experiência estão expressos na forma de gráficos nas figuras 5.6 e 5.7.

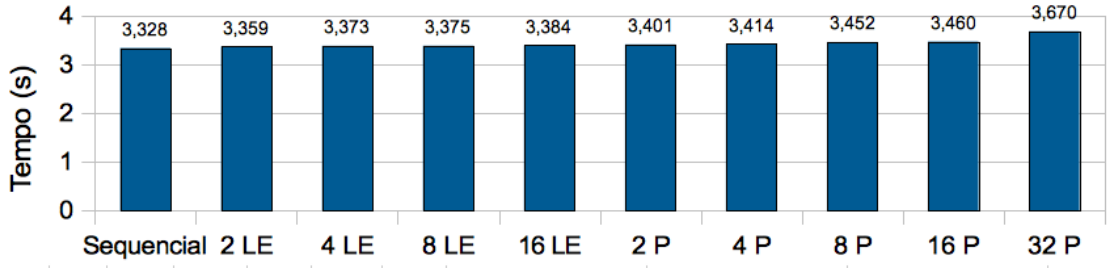

FIGURA 5.6: SOR - Custo da GPC.

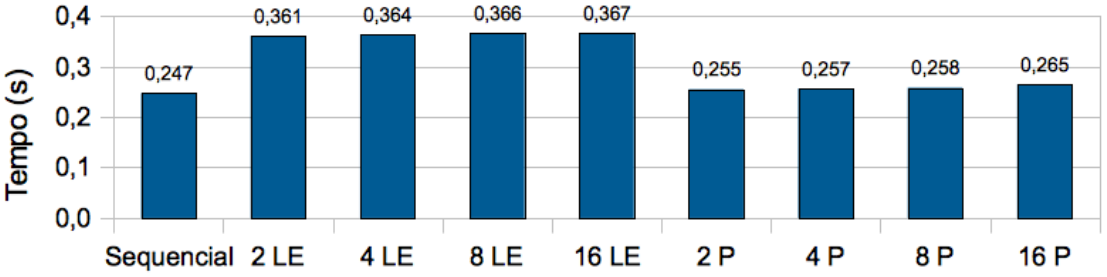

FIGURA 5.7: MOLDYN - Custo da GPC.

Em ambos os casos de estudo, para a versão de memória partilhada, o tempo da GPC cresce ligeiramente com o aumento do número de LE, como resultado da utilização de um algoritmo bloqueante. Quando comparamos os tempos de execução da versão de memória partilhada com o tempo de execução da versão sequencial, verifica-se que o bloqueio das LE introduz um baixo custo acrescido. No SOR, o custo adicional da GPC é

superior na versão distribuída, pelo facto dos processos comunicarem entre si. Este custo também aumenta de forma mais significativa com o aumento do número de processos, devido ao maior custo de comunicação neste tipo de ambiente (SMD). Já no que se refere ao MOLDYN, o custo de GPC na versão distribuída é praticamente o mesmo da versão sequencial. Tal sucede no MOLDYN, porque os processos escravos não enviam os seus dados ao processo mestre, para que este proceda à GPC. Esta troca de dados entre processos, no momento da GPC, foi omitida no MOLDYN, porque nesta aplicação o processo mestre, no momento do ponto de controlo, possui sempre o último estado dos dados a serem armazenados nos FPC. A actualização do estado, do processo mestre, é realizada pela própria aplicação, sem qualquer intervenção do nosso mecanismo de PCR, como tal, aproveitou-se este facto para reduzir o custo da GPC na versão distribuída do MOLDYN.

Na tabela 5.1 é expresso, em percentagem, o custo da GPC nos dois casos de estudo, para cada uma das suas versões, em relação ao tempo total de execução da aplicação com a utilização do mecanismo de PCR. Verifica-se que este custo é muito superior no SOR comparativamente ao MOLDYN, consequência da criação de FPC com maiores dimensões no caso do SOR.

Tabela 5.1: Custo de GPC (em percentagem).

Versão	SOR	MOLDYN
Sequencial	7,86%	0,13%
2 LE	12,06%	0,19%
4 LE	17,51%	0,37%
8 LE	22,81%	0,73%
16 LE	36,47%	1,36%
2 Processos	14,22%	0,17%
4 Processos	22,88%	0,32%
8 Processos	32,51%	0,55%
16 Processos	39,40%	0,89%
32 Processos	38,07%	-

5.5 Processo de restauro

Nesta secção pretende-se analisar a duração do processo de restauro e verificar a fiabilidade da utilização do mecanismo de PCR numa dada aplicação. Para tal, realizaram-se, para todas as versões de cada um dos casos de estudo, duas experiências distintas. Na primeira mediu-se o tempo do processo de restauro, dividindo-o em duas partes: - o tempo correspondente à reconstrução do estado da pilha, no momento da centésima e quinquagésima intercepção do ponto seguro, no SOR e MOLDYN, respectivamente; - o tempo de carregamento da linha de recuperação, do disco. Na segunda experiência,

mediu-se o ganho da realização do processo de restauro e determinou-se qual a probabilidade de falha mínima que viabiliza a utilização do PCR. Os resultados da primeira experiência são apresentados nas figuras 5.8 e 5.9, enquanto os da segunda são apresentados nas figuras 5.10 e 5.11. Ambos os resultados incluem a execução sequencial, a execução de 2 até 16 LE e de 2 até 32 processos MPI no SOR e de 2 até 16 processos MPI no MOLDYN.

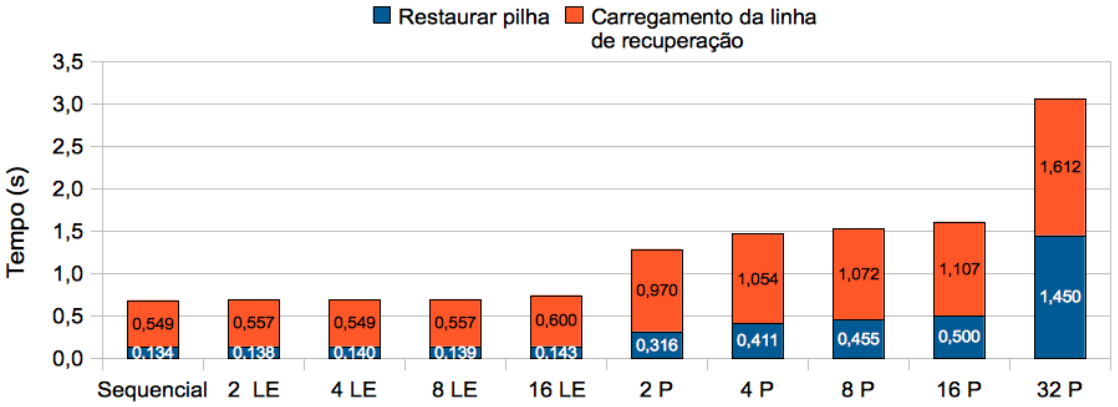

FIGURA 5.8: SOR - Custo do processo de restauro.

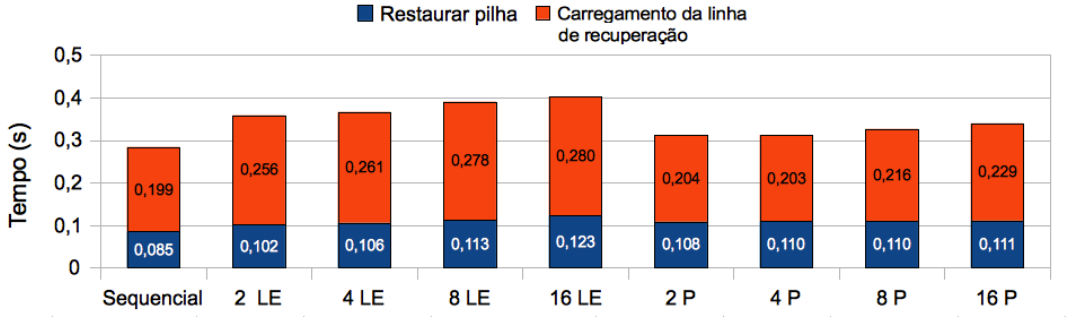

FIGURA 5.9: MOLDYN - Custo do processo de restauro.

O tempo de carregamento de dados é contabilizado, em memória partilhada, a partir do momento em que a LEM carrega os dados até à saída da barreira por parte de todas as LE, enquanto em memória distribuída é contabilizado a partir do momento em que o processo mestre carrega os dados, até ao término da distribuição dos mesmos, pelos restantes processos. No entanto, no caso MOLDYN em ambiente distribuído, uma vez que durante a GPC o processo mestre não colecta os dados, por razões já referidas na secção 5.4, no processo de restauro não existirá distribuição de dados do processo mestre, pois esta é realizada internamente pelo próprio MOLDYN. No que se refere ao tempo de restauro da pilha, este é contabilizado a partir do reinício da aplicação até à finalização da reconstrução do estado da pilha, e imediatamente antes de se iniciar o carregamento da linha de recuperação.

Analisando as figuras 5.8 e 5.9, verifica-se que para todas as versões dos casos de estudo, o custo adicional do processo de restauro advém, na sua grande parte, do carregamento dos dados. Também se verifica que o tempo de carregamento dos dados aumenta com o aumento do número de LE, facto que advém da utilização da barreira, no fim do carregamento dos dados. Em memória distribuída, para o SOR, este custo é superior dado que ocorre distribuição dos dados pelos processos, após o seu carregamento. Já no MOLDYN, o custo de carregamento dos dados em memória distribuída é praticamente o mesmo do ambiente sequencial, pois não existe distribuição dos dados pelos processos, por parte da ferramenta de PCR.

O custo de restaurar a pilha é praticamente o mesmo para todas as versões do MOLDYN. No entanto, no SOR, na versão memória distribuída, este custo duplica para dois processos chegando a triplicar (aproximadamente) para 16 processos, quando comparado com o custo das restantes versões. Tal acontece porque ao contrário do que ocorre na versão sequencial e na versão memória partilhada, em memória distribuída durante o restauro da pilha ocorre a execução de métodos de comunicação entre processos próprios à aplicação e que não podem ser ignorados pela ferramenta de PCR. Por fim, o aumento mais acentuado que se verifica para 32 processos MPI deve-se, em parte, ao facto deste teste ser realizado em duas máquinas, e como tal ocorre transferência de dados entre estas.

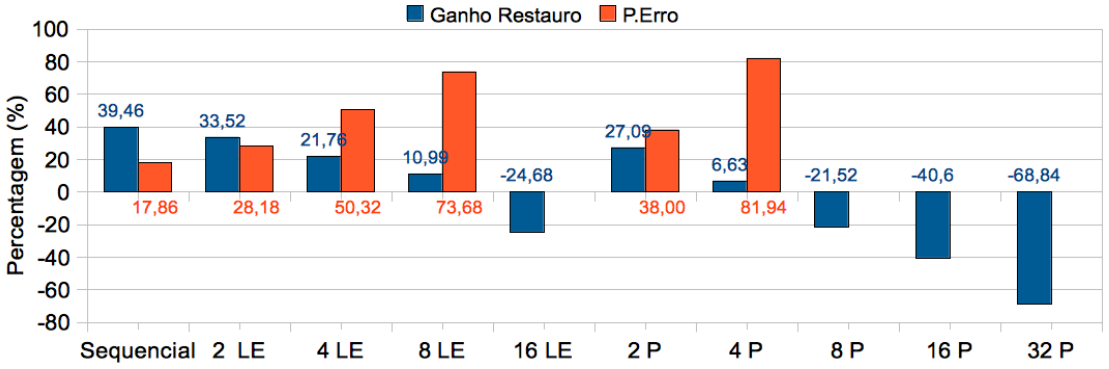

FIGURA 5.10: SOR - Ganho do processo de restauro.

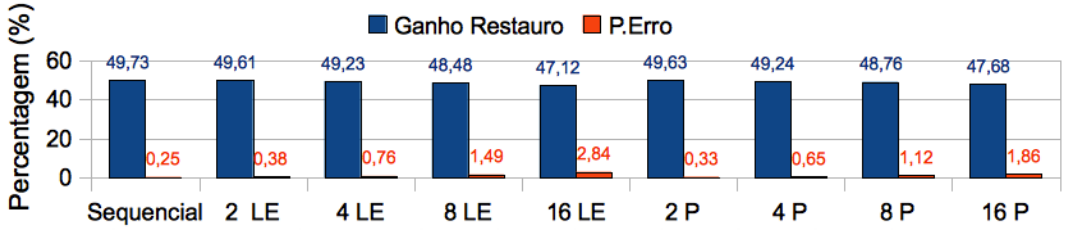

FIGURA 5.11: MOLDYN - Ganho do processo de restauro.

Na segunda experiência, pretendemos determinar a viabilidade do mecanismo de PCR para as versões dos casos de estudo. A percentagem do ganho do restauro (Gr) é calculada

utilizado a seguinte equação:

$$\%Gr = \frac{To - Tr + Tpc}{To}$$

onde To é o tempo de execução da aplicação sem PCR, Tr o tempo de execução da aplicação durante o processo de restauro e Tpc o custo adicional do processo de ponto de controlo com uma GPC. A probabilidade de falha mínima do sistema ($P.Erro$) necessária para viabilizar a utilização do mecanismo de PCR é calculada utilizado a seguinte equação:

$$P.Erro = \frac{Tpc}{Tr}$$

No SOR, como é possível verificar na figura 5.10, existem casos onde $P.Erro$ não é considerada. Para estes casos, o Gr possui valores negativos, o que significa que para esta versão o custo do processo de ponto de controlo é tal, que o tempo poupado a restaurar a aplicação não justifica este custo, e consequentemente a $P.Erro$ não se encontra entre 0 e 100%, não fazendo sentido representá-la. Por exemplo, no SOR com 16 processos o tempo de execução sem PCR é de 5,0 segundos, mas activando o mecanismo de PCR com uma GPC por execução são incrementados a este valor 3,8 segundos e a aplicação demora 3,6 segundos a restaurar. Ora, para que o processo de restauro fosse possível foi necessário realizar pelo menos uma vez o processo de ponto de controlo (com uma GPC), ou seja, é adicionado ao tempo de restauro o tempo do processo de ponto de controlo com uma GPC, perfazendo um total de 7,4 segundos, que é superior ao tempo de execução do SOR sem PCR com 16 processos. Assim, conclui-se facilmente que para a versão memória distribuída com 16 processos do SOR não é justificável a utilização do mecanismo de PCR.

Em contraste com o que acontece no SOR, no MOLYDN os ganhos com o processo de restauro são elevados, próximos dos 50%, e a probabilidade de falha do sistema mínima necessária para viabilizar a utilização do PCR é menor. As diferenças apresentadas entre os dois casos de estudo devem-se, basicamente, ao rácio entre o custo da execução aplicacional e o custo da GPC. O custo do processo de ponto de controlo per si, como possui um valor muito baixo, comparativamente à execução aplicacional, pode ser desprezado. O SOR apresenta tempos de execução muito inferiores ao MOLDYN, mas os seus tempos de GPC são muito superiores. Consequentemente, SOR possui ganhos do processo de restauro inferiores aos de MOLDYN. Desta experiência conclui-se que o ganho do processo de restauro está dependente do tempo de GPC, uma vez que quanto maior o tempo de GPC menor será o ganho obtido com o processo de restauro, portanto é de grande importância a optimização dos FPC.

5.6 Adaptação a recursos

Nesta secção, pretende-se analisar a performance da nossa ferramenta de adaptação dinâmica. Para tal, mediram-se os custos de algumas estratégias de adaptação a recursos, nomeadamente, sobre-decomposição e adaptação estática, utilizando os FPC, para posterior comparação com a nossa abordagem.

Começou-se por medir os custos de sobre-decomposição, utilizando a técnica tradicional (providenciar mais LE/processos que os recursos alocados para a execução aplicacional) descrita na secção 2.6. Para tal, utilizaram-se as versões partilhada e distribuída do SOR e do MOLDYN, com quatro factores de decomposição (Of = 1, Of = 2, Of = 4 e Of = 8), determinados dividindo o número de processos ou de LE em execução pelo número de processadores da máquina, disponibilizados para a aplicação. A execução foi iniciada com 24 LE na versão memória partilhada e 24 processos na versão memória distribuída, utilizando os 24 processadores de uma das máquinas (Of = 1). Posteriormente, foi-se duplicando o número de LE/processos, mantendo o número de processadores, até atingir Of = 8. Nesta experiência modificou-se o número de iteracções do SOR de 100 para 1000, com o objectivo de se obterem tempos de execução mais elevados.

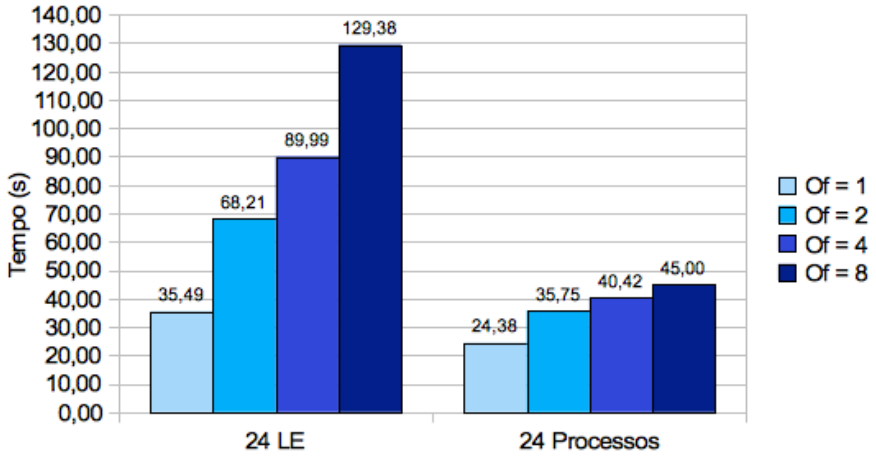

FIGURA 5.12: SOR - Custo adicional da sobre-decomposição.

Os resultados apresentados nas figuras 5.12 e 5.13 mostram que as técnicas de sobre-decomposição tradicionais, podem causar um impacto considerável no tempo de execução. Por exemplo, utilizando 192 LE (Of = 8), o tempo de execução do SOR aumentou, aproximadamente, 264% (de 35,49 para 129,38 segundos). Verifica-se também que o impacto não aumenta de forma proporcional ao aumento do factor de carga e que varia de forma diferente de aplicação para aplicação. Por exemplo, no SOR os aumentos são mais significativos na versão com LE, enquanto no MOLDYN estes aumentos são mais significativos na versão com processos.

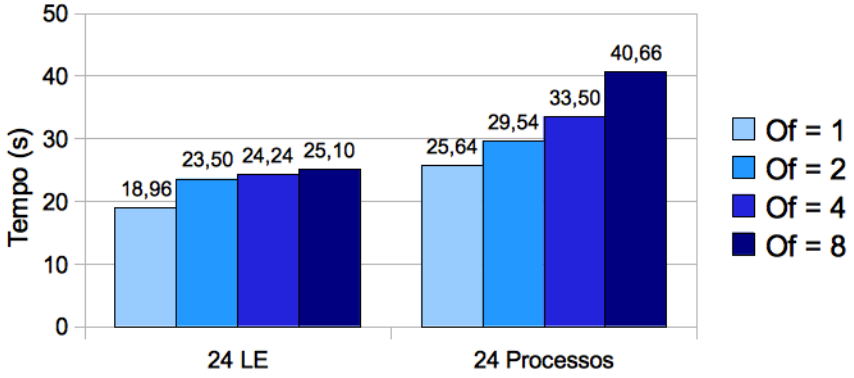

FIGURA 5.13: MOLDYN - Custo adicional da sobre-decomposição.

Seguidamente à finalização dos testes relativos à sobre-decomposição, realizaram-se medições para determinar o custo adicional das duas técnicas de adaptação dinâmica, descritas na secção 4.6, utilizadas pela nossa ferramenta. Esta experiência foi realizada nas mesmas condições da anterior, no entanto, contrariamente aos testes anteriores, as medições foram realizadas apenas sobre a versão memória partilhada do SOR, uma vez que a nossa ferramenta não suporta aplicações com memória distribuída. Os resultados das medições realizadas estão representados na figura 5.14.

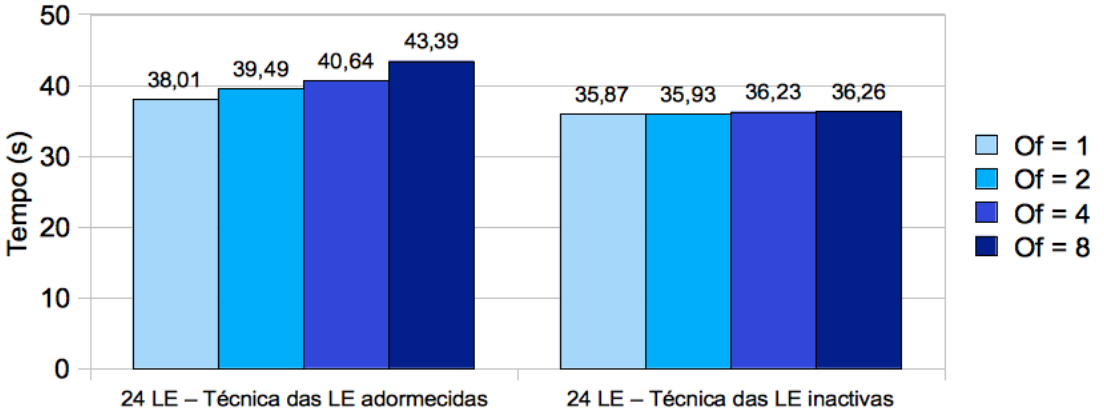

FIGURA 5.14: SOR - Custo adicional da ferramenta de adaptação dinâmica.

Ambas as técnicas utilizadas pela nossa ferramenta apresentam custos adicionais, excepto para Of = 1, muito inferiores aos apresentados na figura 5.12, sendo a técnica das *LE inactivas* a que apresenta os melhores tempos. As nossas técnicas apresentam melhores tempos porque, ao contrário da sobre-decomposição ilustrada na figura 5.12, nem todas as LE participam nas tarefas. Na sobre-decomposição tradicional, para um Of = 8, numa máquina com 24 processadores disponíveis encontram-se 8 LE a realizar tarefas em cada processador. Por sua vez, recorrendo à técnica das *LE adormecidas*, apenas uma LE trabalha, em cada processador, enquanto as restantes 7 LE estão paradas à sua espera numa barreira. Já no que se refere à técnica das *LE inactivas*, apenas está activa 1 LE por processador, e as restantes LE encontram-se retidas numa estrutura de

dados, à espera de serem activadas. Posto isso, não é de estranhar que a técnica das *LE adormecidas* apresente melhores resultados relativamente à técnica tradicional, pois o custo necessário para manter as LE à espera na barreira é inferior ao custo para as obrigar a trabalhar umas sobre as outras, num processador. Por sua vez, menor será o custo se as LE estiverem estagnadas, tal como sucede na técnica das *LE inactivas*, facto que contribui para a obtenção dos melhores resultado recorrendo a esta técnica.

Apresentam-se na tabela 5.2 os ganhos percentuais das nossas técnicas, em relação aos tempos obtidos pela sobre-decomposição tradicional (figura 5.12). A sobre-decomposição tradicional só apresenta melhores resultados para Of = 1, uma vez que existe apenas uma LE por processador, não ocorrendo sobre-carga de recursos nos processadores. Ainda que o mesmo aconteça para as técnicas das *LE adormecidas* e das *LE inactivas*, estas possuem o custo de motorização por parte da ferramenta de adaptação dinâmica, custo inexistente na sobre-decomposição tradicional. Como se pode verificar, pela análise da tabela 5.2, os ganhos percentuais aumentam com o aumento do factor de sobre-decomposição.

Tabela 5.2: Ganhos percentuais das nossas técnicas de adaptação dinâmica.

Of	Técnica das *LE adormecidas*	Técnica das *LE inactivas*
1	-7%	-1%
2	42%	47%
4	55%	60%
8	66%	72%

Após averiguar a melhor técnica para a nossa ferramenta de adaptação, nomeadamente a técnica das *LE inactivas*, comparou-se-a com a adaptação estática, utilizando o mecanismo de PCR. Para tal, realizou-se uma experiência para o SOR (sem alteração do número de iteracções), onde se mediram os tempos de execução sem adaptação, os tempos de execução com adaptação utilizando a nossa ferramenta e os tempos com adaptação utilizando o PCR. Em cada caso, a aplicação inicia a sua execução num conjunto de recursos (2,4 ou 8 LE) e a partir da intercepção do quinquagésimo ponto seguro ficam disponíveis mais recursos, ocorrendo a adaptação da aplicação para 16 LE.

Em todos os casos, a adaptação dinâmica providencia os tempos de execução mais baixos. Já o processo de restauro, na adaptação de 8 para 16 LE, aumenta o tempo de execução aplicacional, ou seja, o tempo de carregamento dos dados, mais o tempo de restauro da pilha, mais o tempo da execução da aplicação após o restauro é superior ao tempo de execução da aplicação sem adaptação. Com adaptação dinâmica obtêm-se melhores tempos de execução do que com o processo de restauro, pois não é necessário o carregamento de dados, nem o restauro da pilha.

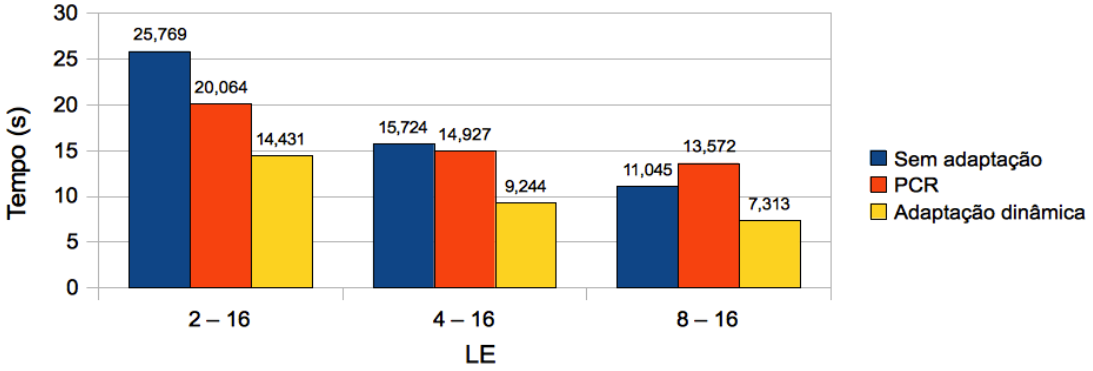

FIGURA 5.15: SOR - Adaptação estática *versus* adaptação dinâmica.

O próximo conjunto de testes realizados mediu a performance da nossa ferramenta de adaptação dinâmica com a técnica das *LE inactivas*. Realizaram-se dois testes diferentes, dos quais o primeiro consistiu na expansão de recursos e o segundo na sobrecarga dos recursos. Os resultados de ambos os testes encontram-se detalhados nas tabelas A.3 e A.4 do apêndice A, e estão ilustrados nas figuras 5.16 e 5.17, respectivamente.

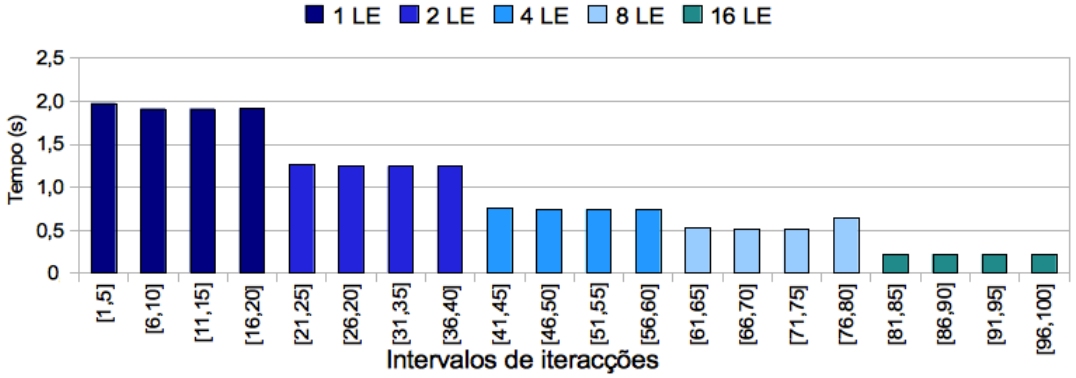

FIGURA 5.16: SOR - Expansão para mais linhas de execução.

A figura 5.16 apresenta o tempo de execução do SOR por intervalo de iterações. Nesta experiência a ferramenta de adaptação dinâmica aumenta o número de LE para o dobro a cada 20 iterações. Inicialmente, a aplicação SOR é executada numa máquina com 16 processadores disponíveis, mas começa a sua execução utilizando apenas uma LE das 128 criadas pelo mecanismo de adaptação dinâmica, terminando a sua execução com 16 LE. Os resultados obtidos mostram que a ferramenta conseguiu melhorar a performance da aplicação SOR, através de adaptação progressiva aos recursos, obtendo-se um ganho no tempo de execução na ordem dos 50%, quando comparado com o tempo de execução das 100 iteracções sem se realizar qualquer adaptação aos recursos.

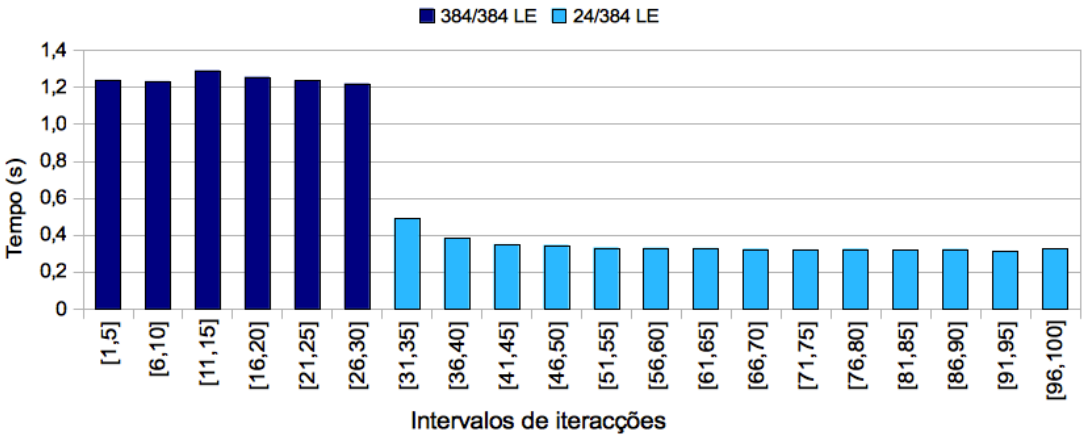

FIGURA 5.17: SOR - Sobrecarga dos recursos.

A figura 5.17 apresenta os resultados da sobrecarga de recursos. Nesta experiência, a aplicação SOR é executada numa máquina com 24 processadores, começando a sua execução com 384 LE activas de 384 LE possíveis (384/384 LE). A partir vigésima quinta iteração a ferramenta de adaptação dinâmica procede à adaptação do número de LE aos recursos disponíveis, reduzindo, portanto, o número de LE de 384 para 24 (24/384 LE). Esta adaptação permitiu obter uma redução de quase 46% no tempo de execução da aplicação.

Os resultados, obtidos neste capítulo, demostraram que a introdução de aspectos na implementação das rotinas do PCR não introduz custos adicionais significativos. Portanto, conclui-se que a utilização de aspectos é uma mais valia, pois o custo adicional que induzem é justificado na plenitude pelas vantagens, já detalhadas, da sua utilização. A ferramenta de adaptação dinâmica demostrou ser uma boa substituta da utilização de sobre-decomposição tradicional, pois o custo adicional que provoca é muito inferior ao causado pela sobre-decomposição tradicional, possibilitando assim, maiores ganhos na performance da aplicação.

Capítulo 6

Conclusões e perspectivas de trabalho futuro

O objectivo desta tese de mestrado foi criar mecanismos de tolerância a faltas e de adaptação dinâmica de recursos para sistemas de grelhas computacionais, com o mínimo de alterações possíveis ao código, utilizado para tal as potencialidade da programação orientada aos aspectos.

Todo o processo de investigação e de análise do trabalho, realizado na literatura, sobre PCR e adaptação dinâmica provou ser fundamental no processo de criação dos nossos próprios mecanismos, não só pelo estudo de algoritmos e abordagens já existentes mas também pela percepção e ganho em relação ao domínio do problema. Através do estudo realizado no capítulo 2 tornou-se notório que não existe muito trabalho desenvolvido no que conserne a algoritmos de PCR de forma não intrusiva, o que tornou atractivo o desenvolvimento de uma abordagem de tolerância a faltas e adaptação dinâmica a recursos.

No capítulo 4 apresentou-se e explicou-se o funcionamento das nossas abordagens. Conseguimos o desenvolvimento de um mecanismo de PCR portável, optimizado e flexível. A nossa abordagem tem com principais vantagens a separação das rotinas do PCR e a adaptabilidade dinâmica do código base das aplicações. Tais rotinas estão presentes em módulos separados que podem ser activados ou desactivados sempre que desejado.

Nesta tese mostrou-se, no capítulo 5, que o custo adicional da utilização de aspectos para o PCR é muito baixo, quando comparado com versões similares onde as rotinas são introduzidas directamente sobre o código fonte. A partir da análise dos resultados do capítulo 5 verifica-se que o mecanismo de PCR, apesar de utilizar um algoritmo bloqueante para SMP e um algoritmo coordenado para SMD, obtém boas performances.

No capítulo 5 ficou também demostrando que a nossa estratégia de sobre-decomposição permite a adaptação da aplicação aos recursos de forma eficiente, sendo mais eficaz que as estratégias tradicionais de sobre-decomposição, que provocam um grande impacto sobre a performance da aplicação.

As restrições impostas pelo mecanismo de PCR na introdução do local de ponto de controlo podem ser vistas como uma desvantagem, pois a ferramenta de PCR ficará limitada a certos tipos de aplicação, mas, por outro lado, estas mesmas restrições permitem reduzir o custo adicional do processo de ponto de controlo, que apenas conta com o custo da contabilização de pontos seguros e o custo de optimização do tamanho dos FPC.

O desenvolvimento do mecanismo de PCR e adaptação dinâmica demostrou ser viável não só pelos testes por nós realizado, mas também pela comunidade científica, ao ter sido aceite a publicação no *Journal of Computing and Informatics* [MS], na conferência *Ibergrid'11* [MS11a] e principalmente pela aceitação e uma nomeação para melhor artigo na prestigiada *40th International Conference on Parallel Processing (ICPP'11)* [MS11b], cuja taxa de aceitação foi inferior a 23%.

Actualmente, no caso do mecanismo de PCR, o programador é responsável por especificar os métodos ignoráveis e os dados a serem gravados no ponto de controlo. Apesar de fornecermos métricas para a determinação de ambos, pretendemos evoluir a ferramenta auxiliar ao PCR (secção 4.6), de forma a dispensar o programador destas tarefas. Esta ferramenta já se encontra em processo de desenvolvimento, faltando optimizar a sua performance de forma a ser considerada fiável. No que se refere à ferramenta de adaptação dinâmica, a implementação actual desta abordagem assenta na utilização de ferramentas externas para a determinação do conjunto de recursos óptimos a ser utilizado pelas aplicações. Uma evolução natural será incorporar mecanismos para encontrar oportunidades para auto-adaptação, para diminuir os tempos de execução através da monitorização dos estados do sistema e da aplicação.

No futuro, pretende-se desenvolver e implementar uma metodologia programacional para ser utilizada no desenvolvimento de aplicações, que visa facilitar, entre outras, a implementação do mecanismo de tolerância a faltas. A metodologia servirá para ajudar a colmatar algumas das restrições impostas pela POA, que foram sentidas na implementação do PCR, nomeadamente, acesso ao fluxo de dados de um determinado método e armazenamento de variáveis do tipo primitivo. Esta metodologia possibilitaria também a introdução de mecanismos exteriores à aplicação, tais como o PCR, adaptação dinâmica, paralelismo e depuração, totalmente não intrusivos, obtendo-se assim código mais legível, estruturado e modular. O principal objectivo desta metodologia é, portanto, guiar o programador, durante o desenvolvimento de uma aplicação, de forma a que esta seja passível à introdução dos mecanismos exteriores já referidos, de maneira facíl, eficiente e

não intrusiva. Parte destas métricas já se encontram especificadas e o objectivo final será especificar todas as métricas necessárias para a metodologia num documento próprio.

Apêndice A

Tabelas com resultados

Tabela A.1: SOR - Resultados da comparação de abordagens (tempos em segundos).

		PCR - Intrusivo		PCR - Não Intrusivo	
Versão	Original (s)	0 GPC (s)	1 GPC (s)	0 GPC (s)	1 GPC (s)
Sequencial	38,79	38,93	42,34	38,95	42,36
2 LE	25,54	25,81	27,77	25,85	27,86
4 LE	15,30	15,74	19,23	15,81	19,26
8 LE	10,97	11,25	14,39	11,38	14,80
16 LE	5,58	5,56	9,24	5,67	9,28
2 P	20,49	20,49	23,89	20,49	23,92
4 P	11,31	11,33	14,86	11,35	14,92
8 P	7,03	7,05	10,61	7,07	10,62
16 P	5,01	5,18	8,68	5,21	8,78
32 P	5,73	5,81	9,63	5,82	9,64

Tabela A.2: MOLDYN - Resultados da comparação de abordagens (tempos em segundos).

		PCR - Intrusivo		PCR - Não Intrusivo	
Versão	Original (s)	0 GPC (s)	1 GPC (s)	0 GPC (s)	1 GPC (s)
Sequencial	196,86	197,19	197,40	197,22	197,47
2 LE	192,32	192,35	192,72	192,48	192,84
4 LE	96,99	97,17	97,44	97,30	97,57
8 LE	49,75	49,83	50,04	49,85	50,13
16 LE	26,68	26,73	26,91	26,78	27,07
2 P	153,55	153,72	154,00	153,90	154,16
4 P	79,24	79,34	79,73	79,49	79,75
8 P	46,70	46,82	46,91	46,88	47,13
16 P	29,36	29,40	29,74	29,47	29,69

Tabela A.3: SOR - Resultados da expansão para mais linhas de execução (tempos em segundos).

Intervalos de iteracções	Tempo (s)
1 - 5	1,972
6 - 10	1,909
11 - 15	1,909
16 - 20	1,910
21 - 25	1,256
26 - 20	1,240
31 - 35	1,239
36 - 40	1,239
41 - 45	0,752
46 - 50	0,736
51 - 55	0,737
56 - 60	0,739
61 - 65	0,521
66 - 70	0,510
71 - 75	0,510
76 - 80	0,636
81 - 85	0,216
86 - 90	0,215
91 - 95	0,216
96 - 100	0,215

Tabela A.4: SOR - Resultados da sobrecarga dos recursos (tempos em segundos).

Intervalos de iteracções	Tempo (s)
1 - 5	1,239
6 - 10	1,232
11 - 15	1,290
16 - 20	1,255
21 - 25	1,236
26 - 30	1,219
31 - 35	0,488
36 - 40	0,382
41 - 45	0,349
46 - 50	0,344
51 - 55	0,330
56 - 60	0,329
61 - 65	0,328
66 - 70	0,323
71 - 75	0,320
76 - 80	0,323
81 - 85	0,319
86 - 90	0,322
91 - 95	0,313
96 - 100	0,324

Bibliografia

[AAB+05] M. Aldinucci, F. Andre, J. Bussion, S. Campa, M. Coppala, M. Danelutto, and C. Zoccolo. Parallel program/component adaptivity management. *Parco 2005*, September 2005.

[ABM11] Rita Arora, Purushotham Bangalore, and Marjan Mernik. A technique for non-invasive application-level checkpointing. *The Journal of Supercomputing*, 57(3):227–255, 2011.

[AER+99] Lorenzo Alvisi, Elmootazbellah Elnozahy, Sriram Rao, Syed Amir Husain, and Asanka De Mel. An analysis of communication-induced checkpointing. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, FTCS '99, pages 242–249, Washington, DC, USA, 1999. IEEE Computer Society.

[AF99] Adnan Agbaria and Roy Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, HPDC '99, pages 167–176, Washington, DC, USA, August 1999. IEEE Computer Society.

[AFM90] R. E. Ahmed, R. C. Frazier, and P. N. Marinos. Cache aided rollback error recovery (carer) algorithms for shared memory multiprocessor systems. *Proc. IEEE 20th Int. Symp. on Fault Tolerant Computing*, pages 82–88, 1990.

[Aga11] Rishi Agarwal. Rebound: Scalable checkpointing for coherent shared memory. Master's thesis, University of Illinois at Urbana-Champaign, USA, 2011.

[AM95] Lorenzo Alvisi and Keith Marzullo. Message logging: pessimistic, optimistic, and causal. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, ICDCS '95, pages 229–236, Washington, DC, USA, 1995. IEEE Computer Society.

[AM98] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[AMHY02] Jinho Ahn, Sung-Gi Min, Chong-Sun Hwang, and Heonchang Yu. Efficient garbage collection schemes for causal message logging with independent checkpointing. *The Journal of Supercomputing*, 22(2):175–196, June 2002.

[arb05] OpenMP architecture review board. Openmp application program interface, version 2.5. www.openmp.org, May 2005.

[Asp] AspectGrid. http://cctc.uminho.pt/projects/AspectGrid.

[BFH03] Fran Berman, Geoffrey Fox, and Anthony J. G. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[BFM+06] Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Recent advances in checkpoint/recovery systems. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 282–282, Washington, DC, USA, 2006. IEEE Computer Society.

[BGJ+96] Michel Banâtre, Alain Gefflaut, Philippe Joubert, Christine Morin, and Peter A. Lee. An architecture for tolerating processor failures in shared-memory multiprocessors. *IEEE Transactions on Computers*, 45(10):1101–1115, October 1996.

[BH08] Marc Bartsch and Rachel Harrison. An exploratory study of the effect of aspect-oriented programming on maintainability. *Software Quality Control*, 16(1):23–44, March 2008.

[BLL90] B. Bhargava, S.R. Lian, and P.J. Leu. Experimenal evaluation of concurrent checkpointing and rollback-recovery algorithms. *Proceedings of the International Conference on Data Engineering*, pages 182–189, March 1990.

[BMP+04] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural*, pages 235–247, New York, NY, USA, 2004. ACM Press.

[BMP+08] Greg Bronevetsky, Daniel J. Marques, Keshav K. Pingali, Radu Rugina Cornell, and Sally A. McKee. Compiler-enhanced incremental checkpointing for openmp applications. *PPoPP '08 Proceedings of the 13th ACM*

SIGPLAN Symposium on Principles and practice of parallel programming, February 2008.

[BMPS03] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. C: A system for automating application-level checkpointing of mpi programs. In *LCPC'03*, pages 357–373, 2003.

[BPS06] Greg Bronevetsky, Keshav Pingali, and Paul Stodghill. Experimental evaluation of application-level checkpointing for openmp programs. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 2–13, New York, NY, USA, 2006. ACM.

[But97] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[CCD+05] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 99–106, Washington, DC, USA, 2005. IEEE Computer Society.

[Cen] Xerox Palo Alto Research Center. www.parc.com/.

[CF90] C.C.J.Li and W.K. Fuchs. Catch - compiler assisted techniques for checkpointing. *Proc. 20th Fault- Tolerant Computing Symposium, FTCS-20*, pages 74–81, 1990.

[CF05] P. Czarnul and M. Fraczak. New user-guided and ckpt-based checkpointing libraries for parallel mpi applications. In *PVM/MPI'05*, pages 351–358, 2005.

[CFK+99] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. In *Journal of network and computer applications*, volume 23, pages 187–200, 1999.

[CGS10] F. Cappello, A. Guermouche, and M. Snir. On communication determinism in parallel hpc applications. *19th International Conference on Computer Communications and Networks (IC- CCN 2010)*, 2010.

[cJLkCFH91] Chung chi Jim Li, Shyh kwei Chen, W. Kent Fuchs, and Wen-Mei W. Hwu. Compiler-assisted multiple instruction retry. Technical report, Coordinated Science Laboratory, University of Illinois, 1991.

[CKH02] Mainak Chaudhuri, Daehyun Kim, and Mark Heinrich. Cache coherence protocol design for active memory systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 83–89, 2002.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CLG05] Jiannong Cao, Yinghao Li, and Minyi Guo. Process migration for mpi applications based on coordinated checkpoint. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems - Volume 01*, ICPADS '05, pages 306–312, Washington, DC, USA, 2005. IEEE Computer Society.

[CS98] Guohong Cao and Mukesh Singhal. On coordinated checkpointing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1213–1225, December 1998.

[CSN05] W. Cirne and E. Santos-Neto. Grids computacionais: Da computação de alto desempenho a serviços sob demanda. *Tutorial presented at the Brazilian Symposium on Computer Networks (SBRC'2005)*, May 2005.

[DDGG95] Om P. Damani, Om P. Damani, Vijay K. Garg, and Vijay K. Garg. How to recover efficiently and asynchronously when optimism fails. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 108–115, 1995.

[Deb] GDB: The GNU Project Debugger. www.gnu.org/software/gdb/.

[Den05] Peter J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, July 2005.

[DLJ99] William R. Dieter and James E. Lumpp Jr. A user-level checkpointing library for posix threads programs. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, FTCS '99, pages 224–227, Washington, DC, USA, 1999. IEEE Computer Society.

[DPMG08] Daniel Díaz, Xoán C. Pardo, María J. Martín, and Patricia González. Application-level fault-tolerance solutions for grid computing. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 554–559, Washington, DC, USA, 2008. IEEE Computer Society.

[DUVE10] Nikolov Dimitar, Ingelsson Urban, Singh Virendra, and Larsson Erik. Online techniques to adjust and optimize checkpointing frequency. *IEEE International Workshop on Realiability Aware System Design and Test (RASDAT 2010)*, pages 29–33, January 2010.

[EJZ92] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. *Proc. 11th Symp. Reliable Distributed Systems, pp. 86-95*, October 1992.

[EKS+03] P. Eerola, B. Kónya, O. Smirnova, T. Ekelöf, M. Ellert, J. R. Hansen, J. L. Nielsen, A. Wäänänen, A. Konstantinov, J. Herrala, M. Tuisku, T. Myklebust, F. Ould-Saada, and B. Vinter. The nordugrid production grid infrastructure, status and plans. In *Proceedings of the 4th International Workshop on Grid Computing*, GRID '03, pages 158–165, Washington, DC, USA, 2003. IEEE Computer Society.

[EMR09] Pedro Evangelista, Paulo Maia, and Miguel Rocha. Implementing metaheuristic optimization algorithms with jecoli. In *Proceedings of the 2009 Ninth International Conference on Intelligent Systems Design and Applications*, ISDA '09, pages 505–510, Washington, DC, USA, 2009. IEEE Computer Society.

[FB88] Stuart I. Feldman and Channing B. Brown. Igor: a system for program debugging via reversible execution. *SIGPLAN Notices*, 24(1):112–123, November 1988.

[Fil05] Robert E. Filman. A bibliography of aspect-oriented software development version 2.0. Technical report, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, California, November 2005.

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, August 2001.

[FPS06] Rohit Fernandes, Keshav Pingali, and Paul Stodghill. Mobile mpi programs in computational grids. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 22–31, New York, NY, USA, 2006. ACM.

[Gen] IBM Blue Gene. www-03.ibm.com/systems/deepcomputing/solutions/bluegene/.

[GJA+97] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J-M, and Irwin J. Aspect-oriented programming. In *ECOOP'97-object-oriented programming, 11th European conference. Lecture notes in computer science*, volume 1241, pages 220–242, Springer, Berlin, October 1997.

[GL03] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java.* John Wiley & Sons, Inc., New York, NY, USA, 2003.

[GR05] B. Gupta and S. Rahimi. A new non-blocking synchronous checkpointing scheme for distributed system. In *Proceedings of the 9th WSEAS International Conference on Systems*, pages 70:1–70:6, Stevens Point, Wisconsin, USA, 2005. World Scientific and Engineering Academy and Society (WSEAS).

[GRB+11] A. Guermouche, T. Ropars, E. Brunet, Snir M, and Franck Cappello. Uncoordinated checkpointing without domino effect for send-deterministic message passing applications. *Accepted to the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2011.

[GS09] R. Gonçalves and J. Sobral. Pluggable parallelization. *High Performance Distributed Computing, (HPDC'09), Munique, ACM Press*, June 2009.

[GSJP05] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, page 9, Washington, DC, USA, 2005. IEEE Computer Society.

[HLK03] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive mpi. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03*, pages 306–322, 2003.

[HMS+98] J. M. D. Hill, B. Mccoll, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. Bsplib: The bsp programming library. *Parallel Computing*, 24(14):1948–1980, 1998.

[IC98] Foster Ian and Kesselman Carl. *The Grid: Blueprint for a New Computing Infrastructure, 1st edition.* Morgan Kaufmann Publishers, San Francisco, USA, November 1998.

[IJSE07] Ru Iosup, Mathieu Jan, Ozan Sonmez, and Dick H. J. Epema. On the dynamic resource availability in grids. In *Proceedings of the 8th IEEE/ACM*

International Conference on Grid Computing, Austin, Texas, USA, September 2007.

[IS93] Golden G. Richard III and Mukesh Singhal. Using logging and asynchronous checkpointing to implement recoverable distributed shared memory. In *Proceedings of the 12 th Symposium on Reliable Distributed Systems*, pages 58–67, 1993.

[JAV] JAVA. java.sun.com/docs/books/jls/download/langspec-3.0.pdf.

[Jey04] Ashwin Raju Jeyakumar. Metamori: A library for incremental file checkpointing. Master's thesis, Virgina Tech, Blacksburg, June 21 2004.

[JLM08] Qiangfeng Jiang, Yi Luo, and D. Manivannan. An optimistic checkpointing and message logging approach for consistent global checkpoint collection in distributed systems. *Journal of Parallel and Distributed Computing*, 68(12):1575–1589, December 2008.

[JZ87] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *Digest of Papers: 17 Annual International Symposium on Fault-Tolerant Computing*, pages 14–19. IEEE Computer Society, 1987.

[KBP96] G. Kingsley, M. Beck, and J. S. Plank. Compiler- assisted checkpoint optimization using suif. In *First SUIF Compiler Workshop*, January 1996.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[Kho] David Khosid. Advanced debugging with gdb. www.haifux.org/lectures/222/GDB_haifux_David_Khosid.pdf.

[KR00] S. Kalaiselvi and V. Rajaraman. A survey of checkpointing algorithms for parallel and distributed computers. *Sadhana*, 25(5):489–510, June 2000.

[KT86] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. In *Proceedings of 1986 ACM Fall joint computer conference*, ACM '86, pages 1150–1158, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[KT88] R. Koo and S. Toueg. Independent checkpointing and concurrent rollback for recorvery - an optimist aproach. *Proc. IEEE Symp. Reliable Distributed System*, pages 3–12, 1988.

[KTT00] Angkul Kongmunvattana, Santipong Tanchatchawal, and Nian-Feng Tzeng. Coherence-based coordinated checkpointing for software distributed shared memory systems. In *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, ICDCS '00, pages 556–563, Washington, DC, USA, 2000. IEEE Computer Society.

[Lab] Open Systems Laboratory. Checkpoint/restart enabled debugging. osl.iu.edu/research/ft/crdebug/examples.php.

[LJ31] J. E. Lennard-Jones. Cohesion. In *Proceedings of the Physical Society*, volume 43, pages 461–482, 1931.

[LLMM99] Julia L. Lawall, Julia L. Lawall, Gilles Muller, and Gilles Muller. Efficient incremental checkpointing of java programs. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 61–70. IEEE, 1999.

[LM10] M. Lotfi and S. A. Motamedi. Adaptive two-level blocking coordinated checkpointing for high performance cluster computing systems. *Journal of information science and engineering*, 26(1):951–966, January 2010.

[LML01] Yibei Ling, Jie Mi, and Xiaola Lin. A variational calculus approach to optimal checkpoint placement. *IEEE Trans. Comput.*, 50(7):699–708, July 2001.

[LNP94] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, August 1994.

[LY87] Ten H. Lai and Tao H. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, May 1987.

[LZZ04] Z. H. Lv, S. Y. Zhang, and Y. P. Zhong. Research on service model of content delivery grid. *The Proc.of the Sixth Asia Pacific Web Conference (APWeb'04), LNCS Series by Springer*, 2004.

[McQ04] William K. McQuay. The collaboration grid: trends for next-generation distributed collaborative environments. In *Enabling Technologies for Simulation Science VIII*, volume 5423, pages 359–366, 2004.

[Meu97] Wolfgang De Meuter. Monads as a theoretical foundation for aop. In *International Workshop on Aspect-Oriented Programming at ECOOP*, page 25. Springer-Verlag, 1997.

[MGBK96] Christine Morin, Alain Gefflaut, Michel Banâtre, and Anne-Marie Kermarrec. Coma: an opportunity for building fault-tolerant scalable shared memory multiprocessors. *SIGARCH Comput. Archit. News*, 24(2):56–65, May 1996.

[MJYS08] D. Manivannan, Q. Jiang, Jianchang Yang, and M. Singhal. A quasi-synchronous checkpointing algorithm that prevents contention for stable storage. *Inf. Sci.*, 178(15):3110–3117, August 2008.

[Ml07] L. Madeyski and L. Sza la. Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study. *IET Software*, 1(5):180–187, 2007.

[MMJ10] Soumaya Marzouk, Afef Jmal Maâlej, and Mohamed Jmaiel. Aspect-oriented checkpointing approach of composed web services. In *Proceedings of the 10th international conference on Current trends in web engineering*, ICWE'10, pages 301–312, Berlin, Heidelberg, 2010. Springer-Verlag.

[MPI] MPICH. http://www.mcs.anl.gov/research/projects/mpi/mpich1-old/.

[MS] Bruno Medeiros and João Sobral. Aspectgrid: Aspect-oriented fault-tolerance in grid platforms. *Accepted for publication in Journal of Computing and Informatics.*

[MS99] D. Manivannan and Mukesh Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703–713, 1999.

[MS11a] Bruno Medeiros and João Sobral. An aspect-oriented approach to fault-tolerance in grid platforms. In *5th Iberian Grid Infrastructure Conference (Ibergrid'11)*, Santander, June 2011.

[MS11b] Bruno Medeiros and João Sobral. Checkpoint and run-time adaptation with pluggable parallelisation. In *40th International Conference on Parallel Processing (ICPP'11)*, Taipei, Taiwan, September 2011.

[MT90] Paula McGrath and Brendan Tangney. Scrabble distributed application with an emphasis on continuity. *Software Engineering Jornal*, 5(3):160–164, July 1990.

[Nel90] Victor P. Nelson. Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25, July 1990.

[NL91] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, August 1991.

[NW] R. H. B Netzer and M. H. Weaver. Ieee std 1003.1, 2004 edition. pubs.opengroup.org/onlinepubs/009695399/functions/fork.html.

[NW94] Robert H. B. Netzer and Mark H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. *SIGPLAN Notices*, 29(6):313–325, June 1994.

[NX95] Robert H. B. Netzer and Jian Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, February 1995.

[Ora] Oracle. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html.

[PBK95] James S. Plank, Micah Beck, and Gerry Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *Ieee Technical committe on operating systems and application environments*, 7(4):10–14, 1995.

[PBKL95] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. *Conference Proceedings, Usenix Winter 1995 Technical Conference, New Orleans, LA*, pages 213–223, January 1995.

[PCL+99] James S. Plank, Yuqun Chen, Kai Li, Micah Beck, and Gerry Kingsley. Memory exclusion: optimizing the performance of checkpointing systems. *Software - Practice and Experience*, 29(2):125–142, February 1999.

[PL88] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution for parallel programs. *SIGPLAN Notices*, 24(1):124–129, November 1988.

[PL94a] J. S. Plank and K. Li. Faster checkpointing with n+1 parity. *24th International Symposium on Fault-Tolerant Computing, pages 288-297*, 1994.

[PL94b] James S. Plank and Kai Li. Ickp: A consistent checkpointer for multicomputers. *IEEE Parallel and Distributed Technology*, 2(2):62–67, June 1994.

[PL05] Zhao Peng and Alexey Lastovetsky. Event logging: Portable and efficient checkpointing in heterogeneous environments with non-fifo communication platforms. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 1 - Volume*

02, IPDPS '05, page 125.2, Washington, DC, USA, 2005. IEEE Computer Society.

[PLP98] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[PP09] S. V. Patel and Kamlendu Pandey. Soa using aop for sensor web architecture. In *Proceedings of the 2009 International Conference on Computer Engineering and Technology - Volume 02*, pages 503–507, Washington, DC, USA, 2009. IEEE Computer Society.

[PRS10] J. Pinho, M. Rocha, and J. Sobral. Pluggable parallelization of evolutionary algorithms applied to the optimization of biological processes. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 395–402, Washington, DC, USA, 2010. IEEE Computer Society.

[PZT02] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. Revive: Costeffective architectural support for rollback recovery in shared-memory multiprocessors. In *ISCA-02*, pages 111–122, 2002.

[Qui03] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[RMG+10] Gabriel Rodríguez, María J. Martín, Patricia González, Juan Touriño, and Ramón Doallo. Cppc: a compiler-assisted tool for portable checkpointing of message-passing applications. *Concurrency and Computation: Practice and Experience*, 22(6):749–766, April 2010.

[SAR] SARA. grid.sara.nl/wiki/index.php/Using_the_Grid/Grid_storage.

[SBF+04] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, page 38, Washington, DC, USA, 2004. IEEE Computer Society.

[SBO01] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, page 8, New York, USA, November 2001. ACM.

[SCM07] João L. Sobral, Carlos A. Cunha, and Miguel P. Monteiro. Aspect oriented pluggable support for parallel computing. In *Proceedings of the 7th international conference on High performance computing for computational science*, VECPAR'06, pages 93–106, Berlin, Heidelberg, 2007. Springer-Verlag.

[Sea] Cluster Search. http://search.di.uminho.pt/wordpress/.

[SFG99] Dwight Sunada, Michael Flynn, and David Glasco. Multiprocessor architecture using an audit trail for fault tolerance. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, FTCS '99, pages 40–47, Washington, DC, USA, 1999. IEEE Computer Society.

[SJ95] G. Suri and B. Jannsens. Reduced overhead logging for rollback recovery in distributed shared memory. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, FTCS '95, pages 279–288, Washington, DC, USA, 1995. IEEE Computer Society.

[SLB02] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with aspectj. *SIGPLAN Notices*, 37(11):174–190, November 2002.

[SM11] J. Sobral and B. Medeiros. An aspect-oriented approach to fault-tolerance in grid platforms. *IBERGRID'2011 - The 5th Iberian Grid Infrastructure Conference Santander*, June 2011.

[SMHW02] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. *SIGARCH Computer Architecture News*, 30(2):123–134, May 2002.

[SOHL+96] M. Sir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. J. Dongarra. Mpi: The complete reference. *MIT Press, Cambridge, MA*, May 1996.

[Spi05] Olaf Spinczyk. Aspectc++ execution model overview. *Friedrich-Alexander University Erlangen-Nuremberg Computer Science 4*, May 2005.

[SPJ+04] José Carlos Sancho, Fabrizio Petrini, Greg Johnson, Juan Fernández, and Eitan Frachtenberg. On the feasibility of incremental checkpointing for scientific computing. *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, April 2004.

[SRNSK10] Dr Ch D V Subba Rao, Dr M M Naidu, and V Sai Krishna. Article: Efficient diskless checkpointing and log based recovery schemes. *International Journal of Computer Applications*, 5(12):29–36, August 2010. Published By Foundation of Computer Science.

[SS98a] L. M. Silva and J. G. Silva. System-level versus user-defined checkpointing. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, SRDS '98, pages 68–74, Washington, DC, USA, 1998. IEEE Computer Society.

[SS98b] Luís M. Silva and João Gabriel Silva. An experimental study about diskless checkpointing. In *Proceedings of the 24th Conference on EUROMICRO - Volume 1*, EUROMICRO '98, pages 395–402, Washington, DC, USA, August 1998. IEEE Computer Society.

[SS08] C. Shah and Ms. S. Shah. Minimizing latency and optimizing resources in a data grid using a dynamic replication strategy. *2nd National Conference on Challenges and Opportunities (COIT 2008), Mandi Gobindgarh*, March 2008.

[Ste96] Georg Stellner. Cocheck: Checkpointing and process migration for mpi. In *Proceedings of the 10th International Parallel Processing Symposium*, IPPS '96, pages 526–531, Washington, DC, USA, 1996. IEEE Computer Society.

[Str98] Volker Strumpen. Portable and fault-tolerant software systems. *IEEE Micro*, 18(5):22–32, September 1998.

[Str10] Martin Streicher. Speaking unix: Interprocess communication with shared memory. http://public.dhe.ibm.com/software/dw/aix/au-spunix_sharedmemory-pdf.pdf, September 2010.

[Tea04] EGEE Team. Lcg. lcg.web.cern.ch, 2004.

[TKW98] Jichiang Tsai, Sy-Yen Kuo, and Yi-Min Wang. Theoretical analysis for communication-induced checkpointing protocols with rollback-dependency trackability. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):963–971, October 1998.

[TLM97] T.Tannenbaum, J.B.M Litzkow, and M.Livny. Checkpoint and migration of unix processes in the condor distributed processing system. *Technical Report Technical Report 1346, University of Wisconsin-Madison*, 1997.

[TTP] NPACI The TeraGrid Project. www.teragrid.org/.

[TY01] Hong Tang and Tao Yang. Optimizing threaded mpi execution on smp clusters. In *Proceedings of the 15th international conference on Supercomputing*, ICS '01, pages 381–392, New York, NY, USA, 2001. ACM.

[VD05] Sathish S. Vadhiyar and Jack J. Dongarra. Self adaptivity in grid computing: Research articles. *Concurrency and Computation: Practice and Experience*, 17(2-4):235–257, February 2005.

[VSL09] Mikael Väyrynen, Virendra Singh, and Erik Larsson. Fault-tolerant average execution time optimization for general-purpose multi-processor system-on-chips. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 484–489, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[Wat] Health Watcher. http://healthwatcher.net/.

[WF90] Kun-Lung Wu and W. Kent Fuchs. Recoverable distributed shared virtual memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

[wLyK00] Jenn wei Lin and Sy yen Kuo. A new log-based approach to independent recovery in distributed shared memory systems. *Journal of Information Science and Engineering*, 16(2):271–290, March 2000.

[WMB07] Gosia Wrzesinska, Jason Maassen, and Henri E. Bal. Self-adaptive applications on the grid. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 121–129, New York, NY, USA, 2007. ACM.

[WPH03] P. Werstein, M. Pethick, and Z. Huang. A performance comparison of dsm, pvm, and mpi. In *Proceedings of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 476–482, SW Jiaotong University, Chengdu, China, 2003.

[Zan] Zandy. Ckpt library. www.cs.wisc.edu/~zandy/ckpt/.

[ZN01] H. Zhong and J. Nieh. Linux checkpoint/restart as a kernel module. *Crak: Technical Report CUCS-014-01, Columbia University*, November 2001.