Universidade do Minho
Escola de Engenharia

Rui António Sabino Castiço da Silva

Algoritmos paralelos para simulação de dinâmica molecular

Outubro de 2010

Universidade do Minho
Escola de Engenharia

Rui António Sabino Castiço da Silva

Algoritmos paralelos para simulação de dinâmica molecular

Dissertação de Mestrado
Mestrado em Informática

Trabalho efectuado sob a orientação do
Professor Doutor João Luís Ferreira Sobral

Outubro de 2010

É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA DISSERTAÇÃO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Universidade do Minho, ____/____/_____

Assinatura: _____

Agradecimentos

Agradeço primeiramente ao meu orientador,o professor João Luis Sobral que acreditou no meu trabalho e o tornou possível.
Agradeço, também aos apoios financeiros disponibilizados pelos projectos P-Found e Gaspar, financiados pela FCT (Fundação para a Ciência e Tecnologia).
Queria agradecer também ao professor Nuno Micaelo, pela disponibilidade em tirar as minhas duvidas na área de simulação de dinâmica molecular.
Agradeço ainda, aos administradores do cluster Search, que ajudaram sempre que precisei.
Não posso esquecer a preciosa ajuda dos meus colegas de laboratório, cuja ajuda foi importante na integração de uma equipa de trabalho.
À família as minhas muitas desculpas por me terem aturado nas minhas tarefas e aturado algumas das minhas impaciências.
A todas as pessoas que conviveram comigo, durante este último ano queria agradecer, pois de alguma maneira contribuíram para este trabalho.

Abstract

This document presents the work in the field of molecular dynamics simulation and the field of parallel computing. In molecular dynamics software is essential to use parallel processing techniques, due to the computational power needed. In this paper work several algorithms are discussed as well as some of the techniques commonly used in molecular dynamics packages.

The parallelization of applications is a complex task, therefore, is used a technique based on aspect-oriented programming.

Since the main objective of the work is to validate the techniques of parallelization and optimization, based on aspect oriented programming, most of this document paper emphasizes the use of these techniques. Thus, this work presents a contribution to a better understanding of optimization techniques and parallelization of molecular dynamics simulations. Techniques based on the aspect-oriented paradigm show to be a viable alternative to express these optimizations.

Resumo

Este documento apresenta o trabalho desenvolvido nas áreas da simulação de dinâmica molecular e no campo da computação paralela. No software de dinâmica molecular é essencial o uso de técnicas de processamento paralelo, devido ao poder computacional necessário. Neste documento são discutidos diferentes algoritmos frequentemente utilizados e técnicas utilizadas nos pacotes de dinâmica molecular.

A "paralelização" das aplicações é uma tarefa bastante complexa, por este motivo, é utilizada uma técnica baseada em programação orientada ao aspecto para proceder à "paralelização".

Sendo o principal objectivo do trabalho desenvolvido a validação das técnicas de "paralelização"' e optimização, baseadas na programação orientada ao aspecto, grande parte deste documento dá ênfase à utilização destas técnicas. Assim, esta tese apresenta um contributo para uma melhor compreensão das técnicas de optimização e "paralelização" de simulações de dinâmica molecular. As técnicas baseadas no paradigma orientado ao aspecto mostram ser uma alternativa viável para expressar estas optimizações.

Conteúdo

Lista de Figuras

Lista de Gráficos

Lista de Tabelas

Capítulo 1

Introdução

1.1 Enquadramento

A simulação de dinâmica molecular tem revelado bastante utilidade para a percepção do funcionamento de moléculas. Este tipo de software tem sido muito estudado e, por consequência, bastante optimizado. Mesmo assim, os diferentes pacotes de software existentes são pesados computacionalmente [HKvL08].

Por este motivo, o uso da computação paralela neste tipo de software é uma solução para que em tempo útil possamos estudar sistemas de maior dimensão (exemplo: proteínas, dna) [HKvL08,HG04,KSB+99]. A "paralelização" deste tipo de software é tipicamente escalável [FEV+09], pois o cálculo das novas posições das partículas (átomos ou moléculas) é independente em cada salto no tempo. Assim, é apenas necessária a sincronização dos processos em cada salto no tempo.

Os recentes avanços da tecnologia sugerem que, no futuro, o aumento da performance dos sistemas seja obtida com o aumentando das unidades de processamento (multi-cores e multi-processadores). Este avanço contrasta com o que foi feito no passado, como é o caso do aumento da frequência do relógio dos processadores, a introdução de memória inter-

média não visível ao programador (memória "cache"), a possiblidade de execução encadeada, possibilidade de execução de instruções em paralelo (super escalabilidade), entre outras. O aumento da frequência nos processadores tem, geralmente um impacto directo no desempenho dos programas sem envolvimento do programador. No entanto para sistemas multi-core os programas desenvolvidos podem não retirar partido deste tipo de tecnologia. Assim, para retirar partido dessa tecnologia é necessário uma mudança drástica na maneira de programar, uma vez que o paradigma actual tem por base um modelo sequencial [AR07].

No paradigma actual, o paralelismo existente entre as diferentes instruções não é explicitado, apesar de as instruções poderem ser executadas em paralelo (paralelismo ao nível da instrução). Com o aparecimento dos multi-cores e dos multi-processadores é necessário explicitar o paralelismo existente entre as diferentes tarefas. O problema surge quando necessitamos de explicitar o paralelismo, pois é uma tarefa bastante complexa e de difícil correcção [HG04, DB09]. Para tal, foram desenvolvidas várias técnicas de modo a que mudança do paradigma fosse mais progressiva. Uma das técnicas utilizadas é baseada na programação orientada aos aspectos, que consiste na separação do código base (código que foi ou será escrito nos paradigmas actuais) do código relativo ao aproveitamento da tecnologia [FSP06, HG04].

Por outro lado, os programas desenvolvidos para os supercomputadores já utilizam um paradigma (Passagem de Mensagens) capaz de retirar potencialidades da nova tecnologia.

1.2 Motivação e Objectivos

Nos pacotes de simulação de dinâmica molecular mais populares, a "paralelização" é efectuada de forma invasiva do código base, isto é, o código base é alterado, não sendo possível distinguir o código referente ao algoritmo base do código referente à "paralelização".

As técnicas não invasivas tem sido utilizadas com sucesso. Um dos casos com grande destaque é o *JECoLi* [Pin09], uma "framework" para algoritmos evolucionários, onde foram utilizadas estas técnicas para se proceder às optimizações necessárias. Nesta tese pretende-se estudar a viabilidade destas técnicas de programação, mas num caso de estudo de uma área aplicacional diferente: simulação de dinâmica molecular. Este estudo apresenta uma complexidade adicional, devido ao elevado grau de optimização das técnicas utilizadas. Assim, esta tese, tem também por objectivo a identificação e melhorar a compreensão das técnicas de optimização deste domínio.

As técnicas não invasivas do código base têm como objectivo separar o código do domínio do problema do código referente às optimizações ao algoritmo. Existem vantagens em utilizar este tipo de técnicas entre as quais importa distinguir as seguintes:

- melhorar a legibilidade do algoritmo base;
- possibilitar a reutilização do código das optimizações;
- facilitar a escolha das optimizações específicas ao sistema alvo, possibilitando assim o colocar ou retirar optimizações;
- possibilitar o desenvolvimento em paralelo do código referente ao algoritmo que permite resolver o problema em questão e o código que permite melhorar o desempenho.

1.3 Estrutura do Documento

No próximo capítulo é efectuada uma introdução à simulação de dinâmica molecular, sendo apresentados algoritmos de simulação de dinâmica molecular. Os diferentes algoritmos apresentados distinguem-se no cálculo das interacções, logo o estudo centra-se neste cálculo. Para

finalizar este capítulo, são apresentadas ao leitor algumas optimizações que são feitas nos programas de simulação de dinâmica molecular.

No capítulo três serão apresentadas diferentes técnicas para a optimização do código, dando ênfase às técnicas não invasivas. Com este capítulo pretende-se que o leitor fique familiarizado com as técnicas não invasivas. No fim deste capítulo é apresentado um caso de estudo onde estas técnicas tem sido aplicadas com sucesso.

O trabalho desenvolvido é apresentado no capítulo quatro, capítulo este dividido em duas partes distintas. Na primeira parte é analisado o perfil de execução da *JGF*[1] e analisados alguns dos contadores de desempenho existentes no processador. Este estudo, serviu para uma melhor percepção dos algoritmos e suas características. Ainda neste capítulo são debatidos os algoritmos implementados utilizando Programação Orientada ao Aspecto (POA). No fim, são apresentados os resultados obtidos com a implementação POA.

O documento termina com uma discussão sobre o trabalho desenvolvido e ainda apontando as possíveis futuras linhas de investigação.

[1] A *JGF* é um conjunto de programas para análise de desempenho, um desse programas é o *Moldyn*, que simula deslocamentos de partículas.

Capítulo 2

Dinâmica Molecular

A simulação de dinâmica molecular é uma técnica de simulação por computador, onde um conjunto de partículas (átomos e/ou moléculas) interagem num certo período de tempo. Os movimentos efectuados pelas partículas nesse período de tempo são representativos do que se passaria na realidade [KHAK06].

A simulação de dinâmica molecular é extremamente útil para a percepção do funcionamento das moléculas biológicas (exemplo: proteínas, dna) [HKvL08]. As simulações envolvem tipicamente 10^2 a 10^6 partículas, no entanto este número é insignificante quando comparado com o número de partículas em sistemas macroscópicos [Nak04, KHAK06].

Na dinâmica molecular as partículas interagem entre si. Para cada par de partículas existe uma interacção entre ambas, que pode ser insignificante a partir de uma certa distância, denominada por raio de corte. A força exercida em cada partícula é igual ao somatório das interacções entre a partícula em questão e as restantes partículas do domínio.

A figura 2.1 apresenta os vários passos de uma simulação de dinâmica molecular [Pos07], explicados de seguida:

Posições iniciais das partículas No início da simulação é necessário definir as posições iniciais das partículas;

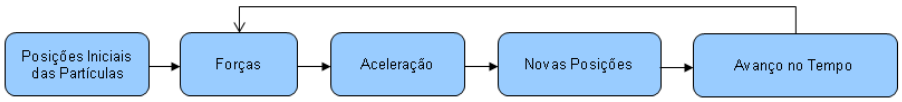


Figura 2.1: Ilustração dos passos do algoritmo de dinâmica molecular

Forças Neste passo são calculadas as interacções aplicadas a todas as partículas. Este cálculo é baseado na segunda lei de Newton.

Aceleração Depois de obtidas as forças é possível calcular a aceleração de cada partícula.

Novas posições Neste passo são calculadas as novas posições das partículas com base na velocidade de cada partícula e no avanço do tempo da simulação.

Avanço no tempo É feito o salto no tempo e o processo é repetido, ou então, a simulação termina.

Dos passos descritos anteriormente o que representa maior esforço computacional é o cálculo das forças [HKvL08, KSB+99]. Essas forças podem ser classificadas nos seguintes tipos [NHG+96]:

Ligantes As interacções das partículas que contenham ligações físicas;

Não ligantes As interacções das partículas que não estejam ligadas fisicamente.

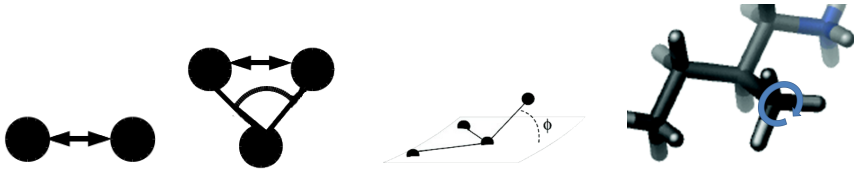
As forças ligantes estão divididas em [BSD95]:

Ligações Interacção entre dois átomos onde, a força é de atracção (a partir de uma certa distância) ou repulsão (quando as partículas se encontram demasiado perto) entre os dois átomos (Figura 2.2a);

Ângulos Interacções entre 3 átomos (Figura 2.2b);

Diedros Impróprios Interacção entre 4 átomos (Figura 2.2c);

Diedros próprios Interacção entre 4 átomos, a diferença da anterior é que é uma força de rotação (Figura 2.2d).



(a) Ligações (b) Ângulos (c) Diedros (d) Diedros Próprios

Figura 2.2: Ilustração de interacções ligantes

Nas forças não ligantes existem duas forças que se destacam [PZKK06]:

Van der Waals Forças atractivas ou repulsivas dependendo da distância, o raio de acção é bastante pequeno;

Electrostáticas Forças atractivas ou repulsivas onde o raio de acção é bastante elevado [SD99], normalmente para além do domínio.

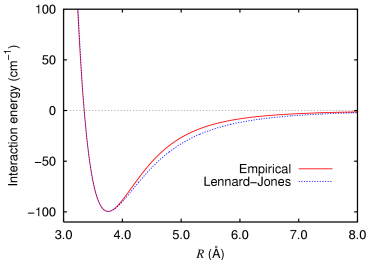


Figura 2.3: Modelo Lennard-Jones

Um dos modelos mais utilizados na dinâmica molecular, para modelar as interacções de *Van der Waals*, é o modelo de *Lennard-Jones* (Figura 2.3) [Kno05, Nak04]. Como é observável na figura 2.3, a partir de uma certa distância a energia da interacção das partículas é aproximadamente zero. Por este motivo, podemos definir um raio de acção (raio de corte) das partículas [Nak04], este raio define quais as partículas

que interagem com a partícula em causa, diminuindo assim o número de cálculos efectuados.

Nas ligações electrostáticas é necessário ter em consideração as partículas existentes fora do domínio representado, para isso existem duas técnicas:

- Particular Mesh Ewald (PME) (Figura 2.4);
- Reaction Field (RF).

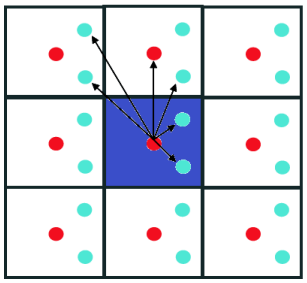


Figura 2.4: Ilustração do método PME

No método denominado por PME o domínio é estendido, como ilustrado na figura 2.4. Na figura o quadrado a azul representa o domínio do problema, os restantes quadrados são réplicas do domínio de forma a calcular as forças electrostáticas. As partículas pertencentes ao sistema interagem com todas as outras partículas incluindo as que se encontram no domínio estendido. Já no que diz respeito às partículas que se encontram no domínio estendido, estas são uma cópia do domínio base. Logo se a partícula i é uma cópia de j, e se j se movimentar, então i efectua o mesmo movimento. Por exemplo, as partículas a vermelho, na figura 2.4, irão todas efectuar o mesmo movimento.

Por outro lado, no método *Reaction Field* o domínio é submetido a um campo electrostático. Depois do campo ser calculado, todas as partículas do sistema são submetidas a esse campo, não sendo necessário a extensão do domínio.

A principal vantagem na utilização do método PME é a qualidade dos resultados obtidos, mas este método é mais pesado em termos de computação. Outro entrave, na utilização de PME, é que esta técnica é mais difícil de optimizar, porque, no seu cálculo, são efectuadas duas transformadas de Fourier tri-dimensionais (FFT), que requer comunicação global na sua implementação paralela [HKvL08].

2.1 Algoritmos de Dinâmica Molecular

Neste capítulo são descritos os principais algoritmos utilizados na dinâmica molecular. Começamos por analisar os algoritmos sequenciais e suas técnicas. Algumas das técnicas utilizadas nos algoritmos sequenciais são, por sua vez, utilizadas nos algoritmos paralelos. As diferenças encontradas nos algoritmos são essencialmente na forma de cálculo das forças, pois o tempo gasto nesta fase do algoritmo é elevado comparativamente com o tempo gasto nas outras fases do algoritmo. De seguida, são apresentados os diferentes algoritmos:

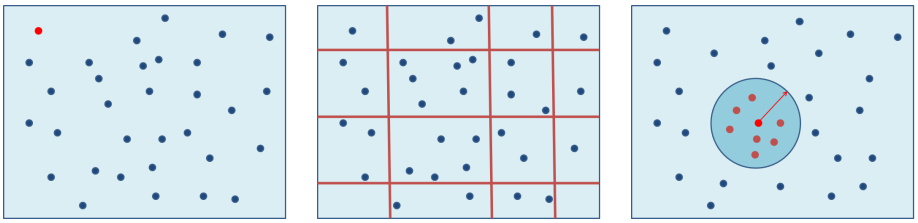
- Sequenciais:
  - Todos os pares;
  - Divisão por células;
  - Lista de vizinhos;
- Paralelos:
  - Decomposição das partículas;
  - Decomposição das forças;
  - Decomposição de células;

2.1.1 Sequenciais

Quando utilizamos o algoritmo *Todos os pares* (figura 2.5a), é efectuado o cálculo das forças entre todas as partículas, o que conduz a uma complexidade de N^2, onde N é o número de partículas. A principal vantagem na utilização deste algoritmo é a sua simplicidade. Um dos problemas encontrados na sua utilização é a necessidade de efectuar cálculos desnecessários (cálculos em duplicado ou cálculo de forças desprezáveis) [Nak04, PH95].

No algoritmo *Divisão por células* (figura 2.5b) as partículas estão organizadas de acordo com a sua posição no domínio e é reduzido o peso computacional para uma complexidade igual a N, melhoria obtida porque os cálculos desprezáveis, não são efectuados. Os cálculos em questão são os das interacções com partículas que se encontrem fora do raio de corte. No caso mais comum o tamanho das células usado é superior ou igual ao raio de corte, pois, assim sendo, só é necessário calcular interacções com partículas que estejam na mesma célula e em células vizinhas. Como as partículas se deslocam ao longo do tempo é necessário actualizar as células, o que obriga a um esforço computacional adicional [Nak04, PH95].

O algoritmo *Lista de vizinhos* (figura 2.5c) cria, para cada partícula, a lista das partículas com que esta interage. As partículas que se encontram na lista são as que se encontram dentro do raio de corte. Em casos em que o movimento das partículas é lento, é possível considerar um raio de corte maior de forma a não ser necessário a actualização da lista em todos os saltos no tempo. Em relação ao algoritmo anterior este algoritmo reduz ainda mais o número de cálculos visto que a organização é à partícula e não à célula [Nak04, PH95].



(a) Todos os Pares (b) Divisão de Células (c) Lista de Vizinhos

Figura 2.5: Algoritmos Sequenciais

2.1.2 Paralelos

Os algoritmos paralelos de dinâmica molecular dividem essencialmente o cálculo das interacções, pois este ocupa grande parte do tempo de execução do algoritmo (80 a 90% [KSB+99]). Nos algoritmos paralelos é necessário que exista comunicação entre as diferentes unidades de processamento (UP). O volume de dados e o que é comunicado depende do algoritmo paralelo utilizado, na tabela 2.1 é mostrada a complexidade de comunicação dos diferentes algoritmos por cada UP.

Algoritmo	Complexidade
Decomposição de Átomos	O(N)
Decomposição de Células	$O(N/P^{\frac{2}{3}})$
Decomposição de Forças	$O(N/\sqrt{P})$

Tabela 2.1: Complexidade de comunicação dos diferentes algoritmos [KSB+99].

O cálculo de cada interacção pode ser realizado em paralelo, porque é baseado nos valores da iteração anterior; e a resultante das interacções numa partícula é igual ao somatório de todas as interacções dessa partícula.

No algoritmo de *Decomposição de partículas* (figura 2.6a), a cada UP é associado um conjunto de partículas. Na figura 2.6a, as cores representam as UPs existentes. Cada UP calcula as interacções das

partículas que lhe forem atribuídas, embora seja necessário que cada UP contenha as posições de todas as partículas, porque uma partícula interage com todas as outras. Assim, é necessário que todas as UPs comuniquem entre si para manter as posições das partículas coerentes [Nak04, PH95].

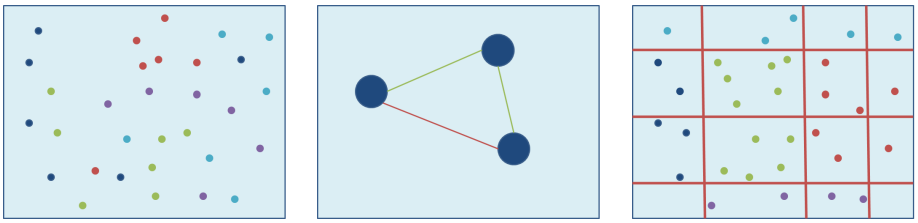
Este algoritmo tem problemas de balanceamento de carga, porque as partículas não interagem com todas as outras devido a dois factores:

Terceira lei de Newton Tipicamente as partículas que são calculadas primeiramente efectuam mais cálculos.

Distribuição desigual das partículas Para partículas que distam entre si uma distância superior ao raio de corte, não é efectuado o cálculo da força, o que implica que existam partículas com diferentes números de interacções.

Na *Decomposição de forças* a divisão é feita repartindo o conjunto das forças pelas várias UPs (figura 2.6b). O problema encontrado com balanceamento de carga, no algoritmo de decomposição de partículas, fica assim resolvido. Por outro lado, existe um problema semelhante, pois para o cálculo de forças diferentes o custo computacional pode ser diferente [PH95, SJ07].

Por fim, o algoritmo de *Decomposição de células* (Decomposição do domínio) divide o domínio físico, isto é, a cada UP é atribuído um subespaço (figura 2.6c). Quando uma partícula muda de célula, é necessário atribuir a partícula à respectiva UP. Neste algoritmo é necessário que cada UP guarde as partículas existentes na célula e as partículas das células vizinhas. O procedimento é necessário por existirem interacções entre partículas da célula em questão e partículas que se encontram nas células vizinhas. O balanceamento de carga é uma aspecto a ter em conta neste algoritmo, porque número de partículas em cada célula é variável, logo o balanceamento de carga é essencial para o bom funcionamento do algoritmo [Nak04, PH95, SJ07].



(a) Decomposição de Partículas (b) Decomposição de Foras (c) Decomposição de Células

Figura 2.6: Algoritmos Paralelos

De forma a diminuir o número de cálculos nos diferentes algoritmos é utilizada a terceira lei de Newton, se a partícula A exerce uma força na partícula B, então B exerce sobre A a força inversa. Assim sendo só é necessário o cálculo de uma das forças, sendo que a utilização desta técnica diminui o número dos cálculos efectuados, mas a comunicação aumenta [Nak04].

2.2 Pacotes de Simulação de Dinâmica Molecular

Nesta secção são apresentadas algumas das técnicas utilizadas em dois pacotes de simulação de dinâmica molecular de ampla divulgação e utilização. Um desse pacotes é o *Gromacs* [HKvL08] que é conhecido por um melhor desempenho sequencial; o outro pacote estudado é o *Namd* [KSB+99] que, como característica principal, tem um bom desempenho na execução num número elevado de UPs.

2.2.1 Evolução

A primeira versão do *Gromacs* paralela surgiu em 1992, utilizando a decomposição das partículas e forças, mas o desempenho obtido não escalava. Em 2000 foi lançada uma nova versão que introduziu o "single

intruction multiple data" (SIMD) e "kernels" para o cálculo das forças levando a que o desempenho global melhorasse, mas diminuindo a escalabilidade do programa na execução num número elevado de UPs. A versão actual do *Gromacs* saiu em 2008, com melhorias significativas [HKvL08].

A última versão do *Gromacs* introduziu partição do domínio em células e os métodos de território neutro (descrito na secção 2.2.3.1). Em comparação, nas versões anteriores do *Gromacs* era utilizada uma topologia de comunicação entre UPs em anel [HKvL08].

O *Namd1* [NHG+96] utiliza a partição de células com a dimensão da célula maior de que o raio de corte o que limita bastante o número de células possíveis. Esta restrição implica que cada célula interaja exclusivamente com os seus vizinhos directos. Como a decomposição das células é baseada no espaço e as partículas não estão distribuídas de uma forma uniforme, pode haver problemas de balanceamento de carga [PZKK06].

Na segunda versão do *Namd* este problema foi resolvido com a combinação da decomposição de forças e decomposição de células [PZKK06], o que permitiu um melhor balanceamento de carga descrito na secção 2.2.3.2.

2.2.2 Optimizações para Arquitectura Sequencial

Em dinâmica molecular é difícil combinar o desempenho obtido numa execução sequencial com o desempenho numa execução paralela. Para ser obtido um desempenho nas duas versões, é necessário, por vezes, mudar o algoritmo utilizado no cálculo da simulação. O *Gromacs* tem uma boa performance a correr sequencialmente e também na versão paralela. A performance obtida na versão sequencial é sobretudo devida à implementação manual utilizando o SSE, SSE2 e ALTIVEC no cálculo das forças [HKvL08].

		número de átomos por UP			
algoritmo	*ordenada*	1705	8525	34100	272800
RF	*sim*	**241**	**48,5**	**11,9**	**1,39**
RF	*não*	238	44	9,6	0,6
PME	*sim*	**102**	**22,5**	**5,4**	**0,61**
PME	*não*	101	20,6	4,8	0,33

Tabela 2.2: Número de saltos dados por segundo.

A capacidade de processamento tem aumentado significativamente. Contudo, a latência e a largura de banda no acesso à memória não tem acompanhado essa evolução. Em compensação tem sido adicionada memória "cache", mas para tirar proveito da "cache", é necessário aceder ao mesmo bloco de dados várias vezes. Por consequência, o acesso aleatório à memória é bastante penalizante para o desempenho.

O *Gromacs* utiliza a procura ordenada de células vizinhas para obter um acesso a memória óptimo, como é reportado na tabela 2.2. Na tabela podemos verificar que os resultados obtidos com ordenação são superiores aos resultados obtidos sem ordenação.

2.2.3 Algoritmos Paralelos

Os algoritmos paralelos utilizados nos dias que correm, como no caso da dinâmica molecular, seguem o modelo "Simple-Program, Multiple-Data" (SPMD). Este modelo é utilizado porque é uma solução óbvia para decompor sistemas com milhares de partículas similares. Contudo, esta abordagem, relativamente ao algoritmo PME, não se mostrou de grande utilidade porque [HKvL08]:

- As interacções de curto alcance podem ser calculadas por os métodos tradicionais não sendo necessário uma extensão ao domínio.

- A escalabilidade de PME é normalmente bastante limitada, pois é necessário comunicação global.

No *Gromacs* o algoritmo paralelo PME foi rescrito e introduziram-se algumas optimizações para suportar a decomposição de domínio. Mas, a falta de escalabilidade do algorimo PME foi lidada com suporte adicional e opcional de "Multiple-Program, Multiple-Data" (MPMD), onde um subconjunto de UPs é dedicado ao algoritmo PME, enquanto que as outras UPs são responsáveis pelas interacções. As UPs dedicados às interacções directas devem ser na razão de 2:1 ou 3:1 em relação às UPs dedicadas ao algoritmo PME. Numa futura versão do *Gromacs* pretende-se automatizar este processo.

2.2.3.1 Decomposição de Domínio

Os métodos de território neutro são utilizados em vários domínios que necessitam de calcular interacções entre pares de partículas. Alguns dos exemplos onde estes métodos tem sido utilizados são: dinâmica molecular, simulações de Monte Carlo; e nas simulações Nbody [BDS05].

Nos métodos tradicionais, como no caso do método denominado por *Half Shell* [HKvL08], as interacções são calculadas onde uma das partículas reside e o total de comunicação necessária é metade do volume cujo raio é igual ao raio de corte. Contudo, nos métodos de território neutro isto pode não ser verdade como ilustrado na figura 2.7, isto é, a interacção entre a partícula i e j não é necessariamente calculada na UP onde a partícula i ou j reside. Pode parecer estranho mas a utilização destes métodos revela-se mais eficiente que os métodos tradicionais, uma vez que reduzem o total da comunicação necessária [HKvL08,BDS05,Sha05].

A figura 2.7 apresenta um domínio dividido em 9 células. Cada célula tem 3 colunas e na primeira estão presentes as interacções entre células calculadas, seguindo um método de território neutro; na segunda encontra-se o número das células necessárias para o cálculo; na terceira

coluna, encontra-se o o número das células necessárias para o cálculo no caso de utilizarmos métodos tradicionais [Sha05].

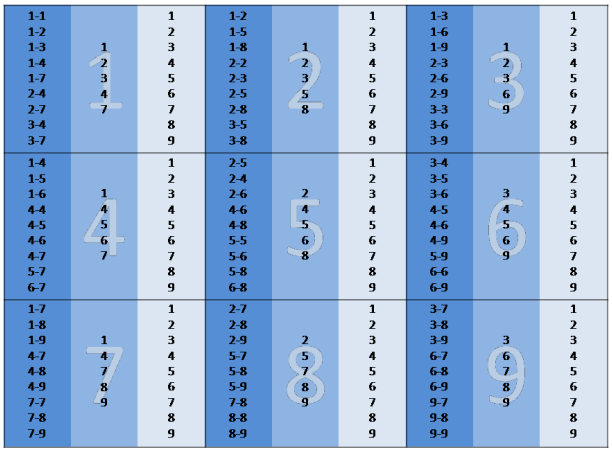


Figura 2.7: Comparação do volume de dados comunicado quando são utilizados os métodos tradicionais ou os métodos de território.

O método *Eighth Shell* é responsável por melhorar o cálculo de interacção entre partículas que residem em diferentes células. O volume de dados comunicado com a utilização deste método é um subconjunto do volume comunicado no método *Half Shell*. Para além do volume comunicado ser menor, a comunicação necessária envolve menos passos o que diminui a latência no acesso aos dados [HKvL08].

O *Gromacs* utiliza uma versão alterada do método *Eighth Shell*, a alteração efectuada permitiu que as células possam interagir com vizinhos que não sejam directos [HKvL08]. Assim, é permitido que cada célula tenha uma dimensão menor que o raio de corte.

O *Namd* utiliza uma decomposição de domínio baseada decomposição de células e na partição de forças. No *Namd* as células são chamadas *Patches* sendo a sua dimensão estática, e um pouco superior ao raio de corte. As interacções são calculadas nos *Compute Objects*, sendo associado a este um tipo de interacção e associado a um ou mais *Patches*. Após o cálculo das interacções os *Patches* são notificados pelos *Compute Objects* sobre o valor das interacções e estes são somado às outras

interacções. As *Patches* são distribuídas pelas várias UPs disponíveis.

As forças entre células vizinhas são calculadas livremente entre as diferentes UPs disponíveis, possibilitando assim um melhor balanceamento de carga. No *Namd* são utilizadas threads para ser efectuada a comunicação diminuindo assim a sobrecarga introduzida [PZKK06].

2.2.3.2 Balanceamento de Carga

O balanceamento de carga é tipicamente necessário, na dinâmica molecular, devido a três razões:

Distribuição desigual das partículas no espaço Implica que as partículas tenham mais ou menos interacções nulas com outras partículas.

Desigualdade das interacções É devido a que nem todos os tipos de interacções tem o mesmo custo computacional.

Flutuação das partículas Implica que a distribuição das partículas pelo espaço varie, obrigando portanto a um novo balanceamento de carga.

O *Gromacs* ajusta automaticamente as dimensões da células durante a simulação de forma a melhorar o balanceamento de carga. O balanceamento é determinado através do tempo despendido nas rotinas responsáveis pelo cálculo das forças em cada UP. O tempo é determinado utilizando o contador de ciclos disponível nas novas arquitecturas de processadores. O volume de cada célula é ajustado na direcção apropriada e para tal, são tomados os seguintes passos:

1. O tempo é acumulado na direcção Z os valores são enviados para as UPs que contém as células com Z=0.

2. As UPs que contêm as células com Z=0 somam os valores e enviam os resultados para as UPs que contém as células Y=0.

3. Por sua vez estas UPs somam os valores e enviam os resultados para a UP que contêm a célula X=0.

4. Este processo consegue agora, deslocar o limite na direcção X e enviar esta informação para as UPs que contenham a célula com Y=0.

5. As UPs que contenham as células com Y=0 determinam a deslocação em Y e enviam os deslocamentos em x e y para as UPs que contenham as células com Z=0.

6. Por fim determinado os limites em Z pelas UPs que contenham as células com Z=0.

Com o procedimento acima descrito é evitada a comunicação global. O ajuste dos limites deve ser efectuado de uma forma conservadora para evitar oscilações e instabilidade, que podem ocorrer devido flutuações em células de menor dimensão.

No *Namd* o balanceamento é feito redistribuindo os *Compute Objects* pelas várias unidades de processamento, isto é possível porque as interacções entre células vizinhas podem ser calculadas em qualquer UP. O balanceamento dos *Compute Objects* é feito nos *Compute Objects* que são responsáveis por interacções não ligantes, pois estes ocupam grande parte do tempo da simulação. Contudo, no início da simulação é necessário distribuir os *Patches* pelas várias UPs disponíveis, para tal é utilizado o algoritmo recursivo da bisectriz. Este procedimento, é tipicamente feito com espaçamentos de tempo na ordem dos minutos [PZKK06].

Capítulo 3

Técnicas de "paralelização" não Invasivas

3.1 Paradigma Orientado ao Aspecto

Na maioria dos paradigmas existe uma mistura dos diferentes conceitos no código, como, por exemplo, segurança, persistência, registo de eventos, etc. Como podemos observar na figura 3.1 existe uma mistura de conceitos, o código relativo a persistência e registo de eventos (RE) encontra-se disperso pelas classes do problema [DB09,FEC04].

Ao utilizar POA é possível separar os diferentes conceitos. Assim, o programador ao utilizar a POA está a identificar onde os diferentes blocos de código serão executados possibilitando assim a separação de conceitos.

Em POA seriam criados aspectos dependendo do número de funcionalidades, sendo que como ilustrado na figura 3.2 seria necessário criar os aspectos de *Persistencia* e de *RE*. Com este método seria mais fácil reutilizar código, pois o código necessário para cada funcionalidade tem pontos em comum independentemente da classe onde serão aplicados.

Com a separação do código em módulos bem definidos e relativos a

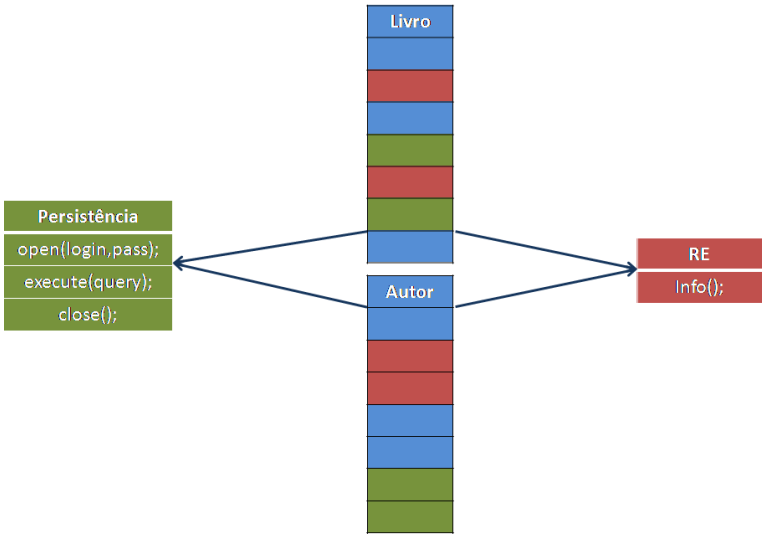


Figura 3.1: Exemplo de mistura de conceitos [Cad09].

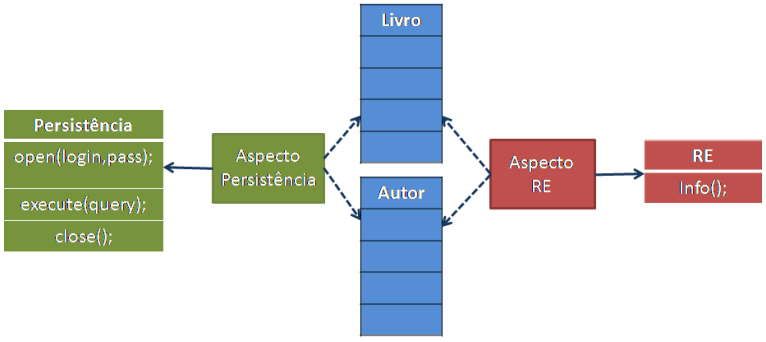


Figura 3.2: Utilização de POA [Cad09].

uma funcionalidade são evitados alguns dos erros. Uma das principais razões para a diminuição dos erros é a reutilização de código. Quando reutilizamos código, diminuímos a replicação de código, podendo com isto evitar o propagar de erros.

Outro dos motivos para melhorar a correcção é a possibilidade de utilizar ou não as diferentes funcionalidades, sendo assim possível ao programador saber se a lacuna se encontra no código do domínio, ou no código das diferentes funcionalidades.

A tabela 3.1 é comparada a utilização de POA com as abordagens tradicionais. Na tabela está uma possível implementação de uma transferência bancária. No caso tradicional o conceito base de uma transferência bancária (retirar valor de uma conta para outra) está misturado com o código de validação da mesma transferência (validar se a conta de origem tem saldo disponível). Em contrapartida, na implementação POA, o código de validação está num módulo separado.

3.2 Linha de Produtos

A linha de produtos de software é considerada uma metodologia para desenvolvimento de conjuntos de programas similares de forma rápida e sistemática [PBVDL05, TBKC07], através de um conjunto de elementos básicos. Ao utilizamos a linha de produtos estamos utilizar uma estratégia de reutilização [CN02].

A principal razão para utilizar a linha de produtos como paradigma é a redução de custo. Como é observável no gráfico 3.1, os custos iniciais de utilização de uma linha de produtos são maiores de que no caso de criação de um sistema normal. Apesar disso existe um ponto em que compensa a utilização da linha de produtos. Este ponto é designado como *Break-Even Point*.

Para além da vantagem dos custos, a linha de produtos aumenta a

Tradicional(JAVA)

```
void transfer(double valor, Conta destino){
  if(valor < this.saldo) {
    return ;
  }
  this.saldo-=valor;
  destino.saldo+=valor;
}
```

POA

```
void transfer(double valor, Conta destino){
  this.saldo-=valor;
  destino.saldo+=valor;
}

void around(double valor, Conta destino, Conta origem) :
  call(public void transfer(double,Conta))
    && args(valor,destino) && target(origem){
  if(valor < origem.saldo)
    proceed(valor,destino);
}
```

Tabela 3.1: Comparação POA com as abordagens tradicionais

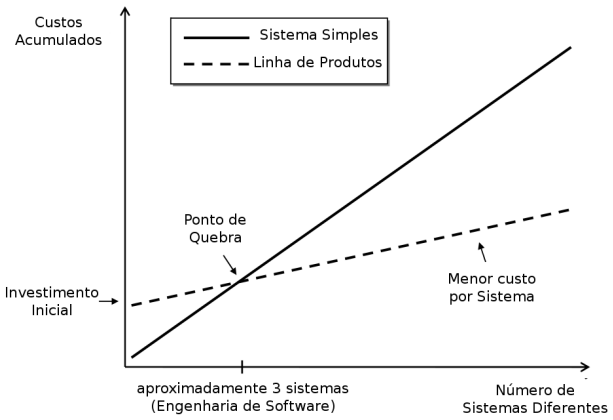


Gráfico 3.1: Comparação dos custo entre a utilização de uma linha de produtos e um sistema simples [PBVDL05].

qualidade de software, pois, como existe reutilização de código este é mais testado e revisto por ser utilizado em vários programas.

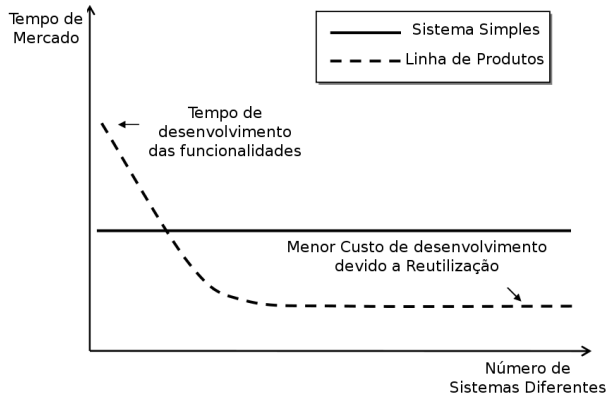


Gráfico 3.2: Comparação dos custo temporais entre a utilização de uma linha de produtos e um sistema simples [PBVDL05].

A reutilização de software também possibilita a diminuição do tempo necessário para a produção do sistema. Esta vantagem é também de bastante importância, pois o tempo é um factor importante no sucesso do sistema. Contudo, os primeiros sistemas criados com o paradigma de linha de produtos tem um custo de tempo maior, como é observável no gráfico 3.2.

Além das vantagens descritas acima são ainda de ressalvar as seguintes:

Custos de manutenção Como existe reutilização de código, quando este precisa de alguma modificação esta modificação é feita num único sítio.

Evolução do sistema A evolução do sistema é feita de uma forma organizada. Com o passar do tempo vão sendo adicionadas novas funcionalidades ao programa [DCB09]. Os modelos das funcionalidades servem para especificar quais as combinações de funcionalidades que podem ser utilizadas.

Lidar com a complexidade A complexidade é reduzida, porque algum do código é reaproveitado.

A linha de produtos apresenta limitações, assim para justificar a utilização da linha de produtos é necessário que os diferentes sistemas tem algo em comum.

Os diagramas de funcionalidades são uma representação gráfica para os modelos das funcionalidades [Bat05], logo são utilizadas para especificar as relações entre as diferentes funcionalidades.

Na figura 3.3 é ilustrada a notação utilizada para especificar estes diagramas. Quando utilizamos o *e* no diagrama todas as funcionalidades especificadas devem ser utilizadas, caso exista funcionalidades especificadas com o símbolo *opcional* significa que estas podem ou não ser utilizas. O símbolo denominado por *alternativo* serve para especificar funcionalidades alternativas, isto é, só uma delas deve ser utilizada. Ao ser utilizado o símbolo *ou* qualquer combinação de funcionalidades é permitida.

Figura 3.3: Diagramas de funcionalidades

Ao ser utilizado como paradigma de desenvolvimento a programação orientada a funcionalidade (POF) é utilizada a linha de produtos como metodologia.

Na POF software complexo é decomposto em funcionalidades. Logo, os produtos desenvolvidos são combinações de funcionalidades [Bat05, ALRS05]. Neste paradigma as funcionalidades são agregadas de uma forma flexível, ao contrário do que acontece nas linguagens orientadas ao objecto [Pre01].

Por outro lado, é possível utilizar a metodologia linha de produtos utilizando o paradigma POA. Ao construir aspectos representativos de funcionalidades, estes podem ser composto de forma a gerar vários software independentes.

3.3 “Paralelização” não Invasiva

As optimizações e transformações necessárias para obtenção de código paralelo, são bastante complexas e obrigam tipicamente à invasão do código base, não sendo possível o retorno ao código original (i.e., sem a ”paralelização”). Para simplificar estas transformações pode ser utilizada a programação orientada ao aspecto, separando assim o código base do código paralelo [GS09, dP+10].

A utilização de técnicas tradicionais para a transformação do código em código paralelo, dificulta (ou mesmo estagna) o desenvolvimento do código específico do domínio enquanto se procede à ”paralelização”.

Esta técnica de ”paralelização” seguem a seguinte filosofia:

Não invasão As modificações ao código base deverão de ser mínimas;

Localização O código relativo à ”paralelização” deve estar localizado em módulos bem definidos;

Incremental Deve possibilitar a reutilização do código base, desenvolvendo o código paralelo de forma progressiva.

A técnica POA tem sido utilizada para a ”paralelização” de alguns códigos existentes. Na próxima secção é apresentado um exemplo da utilização da técnicas POA.

3.3.1 Exemplo

O *Java Evolutionary Computation Library*(JECoLi) é uma "framework" para algoritmos evolucionários desenvolvida na Universidade do Minho nos últimos anos. A biblioteca original não retirava proveito da utilização da tecnologia multi-core nem da computação em *GRID*[1] e *CLUSTER*[2], pois a implementação existente era sequencial.

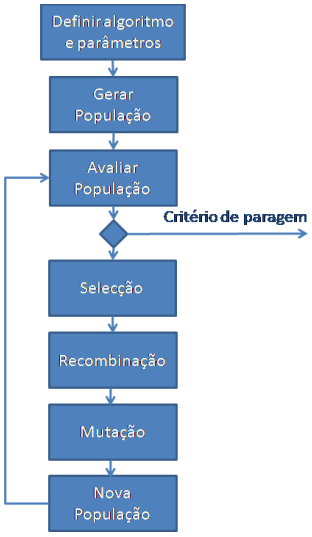


Figura 3.4: Algoritmo evolucionário presente na JECoLi

O algoritmo presente na JECoLi segue os passos presentes na figura 3.4. No primeiro passo do algoritmo são definidos os algoritmos que serão utilizados, bem como os seus parâmetros. Após esse passo, tipicamente é gerada uma população inicial aleatoriamente. A partir deste ponto, são feitas as iterações ao algoritmo, sendo estas divididas nas seguinte fases:

Avaliação Para cada indivíduo é calculado o seu valor de aptidão. Após a avaliação a todos os indivíduos é testada a condição de paragem.

[1] http://pt.wikipedia.org/wiki/Computao_em_grelha
[2] http://pt.wikipedia.org/wiki/Cluster

Selecção Na selecção são escolhidos os indivíduos que farão parte da nova população, onde os indivíduos com mais aptidão têm mais probabilidade de serem seleccionados.

Recombinação Os indivíduos organizam-se de forma a gerarem novos indivíduos, segundo determinados critérios.

Mutação A população existente sofre uma transformação.

Nova população Após a mutação é gerada a nova população e os passos anteriores são repetidos.

A motivação para "paralelização" e optimização do código de algoritmos evolucionários deve-se ao elevado custo computacional dos algoritmos utilizados. Nestes algoritmos, a parte significativa do processamento é ocupada pela função de avaliação. Para proceder a "paralelização" da JECoLi foram utilizados dois métodos:

Avaliação Paralela Neste caso, a função de avaliação é aplicada a vários indivíduos em paralelo.

Modelo de Ilhas Para este modelo a população é dividida em várias subpopulações, designadas por ilhas. Cada ilha, evolui com a sua população de forma independente, contudo é necessário existir migração entre ilhas.

O modelo de *Modelo de Ilhas* pode ser utilizado para versões sequenciais, com a utilização deste modelo existe uma melhor pesquisa das soluções no espaço, evitando que a solução tenda para uma solução local.

As técnicas utilizadas na transformação do JECoLi têm por base POA. Com sua utilização foi possível proceder ao desenvolvimento do código paralelo e do código do domínio em simultâneo. Para além desta vantagem, existe a possibilidade de em tempo de compilação gerar diferentes versões do programa (sequencial ou paralelo).

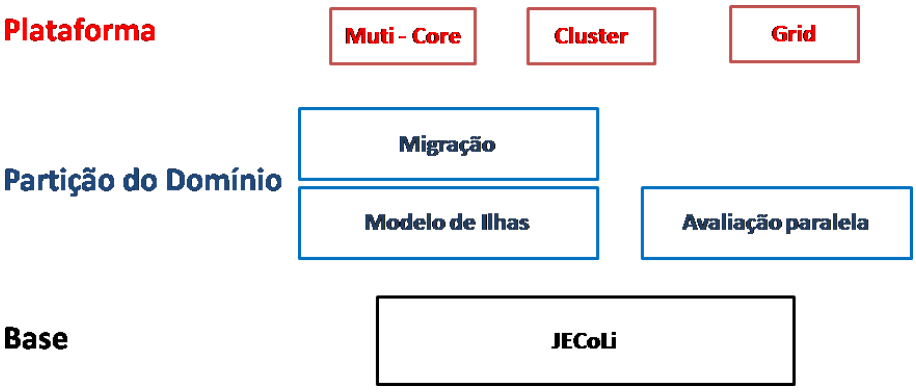


Figura 3.5: Organização dos Aspectos da Jecoli.

Na figura 3.5 é mostrada a arquitectura simplificada da "paralelização" efectuada. Em baixo, é representado o código relativo a *JECoLI*.

Na linha denominada por *Partição do Domínio* encontram-se as opções de "paralelização", onde existem dois modelos: a *Avaliação Paralela* e *Modelo de Ilhas*.

Na *Avaliação Paralela* é utilizado o modelo de *master-slave*, sendo que os *slaves* são responsáveis por avaliar e/ou aplicar os operadores genéticos a cada indivíduo. Quanto ao *master* é responsável por distribuir as tarefas existentes pelos vários *slaves* e também analisar os vários resultados que são enviados pelos *slaves*.

Por outro lado, existe o modelo de *Modelo de Ilhas* onde existem vários algoritmos evolucionários que evoluem de forma independente, existindo uma migração periódica de indivíduos [Pin09].

Por fim, existe a linha denominada por *Plataforma*, onde é especificado o código relativo a cada plataforma. Na plataforma *Multi-core* é utilizado *Java Threads*. Quanto a plataforma *Cluster* é utilizado o *MPIJava* para distribuir os vários processos pelos vários nodos de um cluster. Por fim, é possível utilizar uma versão que corre numa versão *GRID*, nesta versão é utilizado o *storage element* para comunicar entre os diversos *sites* que estão a executar o programa [Pin09].

Os resultados obtidos com este caso de estudo são bastante positivos. Os ganhos obtidos quanto ao desempenho são próximos dos ganhos teóricos e em alguns casos são mais elevados [dP$^+$10].

Capítulo 4

Trabalho desenvolvido

O trabalho desenvolvido consiste na validação das técnicas de optimização não invasivas em algoritmos de dinâmica molecular. Após os estudos dos diferentes algoritmos utilizados na dinâmica molecular, foi escolhido como caso de estudo o Modyn presente na *Java Grand Forum* (JGF) [BSW+00].

Esta escolha deveu-se ao facto de ser um exemplo simples de dinâmica molecular e por existirem várias versões do programa. Uma das versões é sequencial. As outras duas exploram o paralelismo para memória partilhada e para memória distribuída através de técnicas de programação paralela tradicionais. Este ponto é importante pois permitiu comparar os resultados obtidos nas várias versões, sendo assim possível analisar o impacto da utilização de linguagens POA, no tempo de execução.

No âmbito deste trabalho foi necessário um estudo aprofundado sobre os diferentes algoritmos. Para tal, foi utilizado o caso de estudo da JGF. Este estudo consistiu em elaborar um perfil de execução sobre os diferentes algoritmos sequenciais. Devido às ferramentas que foi preciso utilizar não serem compatíveis com *Java*, foi necessário reescrever o JGF na linguagem de programação *C++*.

Este capítulo esta dividido em três partes distintas. Na primeira parte será analisado o algoritmo que se encontra presente na JGF, para tal, são ilustrados os diferentes passos do algoritmo de simulação de dinâmica molecular. Na segunda secção deste capítulo algoritmos descritos na secção 2.1.1 são analisados, em detalhe, ao nível da sua execução. Para finalizar este capítulo, são descritas as diferentes implementações desenvolvidas com o paradigma POA.

4.1 Análise do Moldyn do JGF

O Moldyn tem duas partes distintas. No que se refere à primeira parte são geradas as partículas e colocadas nas respectivas estruturas. Neste estudo a parte das inicializações não tem relevo, uma vez que num caso real as partículas não seriam geradas pelo programa de dinâmica molecular, mas seriam um dado de entrada. Na segunda parte do Moldyn, é efectuado o cálculo propriamente dito, este cálculo está dividido em várias fases com correspondência no algoritmo apresentado na figura 2.1. A figura 4.1 faz corresponder os nomes das funções do Moldyn, aos diferentes passos apresentados na figura 2.1.

No Moldyn é utilizado o algoritmo *Todos os pares* para o cálculo. Neste software é modelado o potencial de *Lennard-Jones*, onde as partículas se encontram num espaço tridimensional com condições de fronteira periódica [BSW+00].

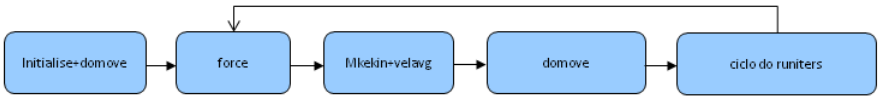


Figura 4.1: Funções Moldyn

Como foi dito anteriormente o cálculo das forças é relativamente pesado em relação aos restantes, na figura 4.2 são mostrados os diferentes

métodos e respectivos tempos de execução. Como podemos observar o método que ocupa a maior parte da execução é denominado por *force*, este método calcula as interacções das partículas em questão. Por ser o método mais pesado a nível de tempo de execução, esta função é tratada com especial atenção. As diferentes optimizações que serão efectuadas ao algoritmo são essencialmente para reduzir o tempo de execução desta função.

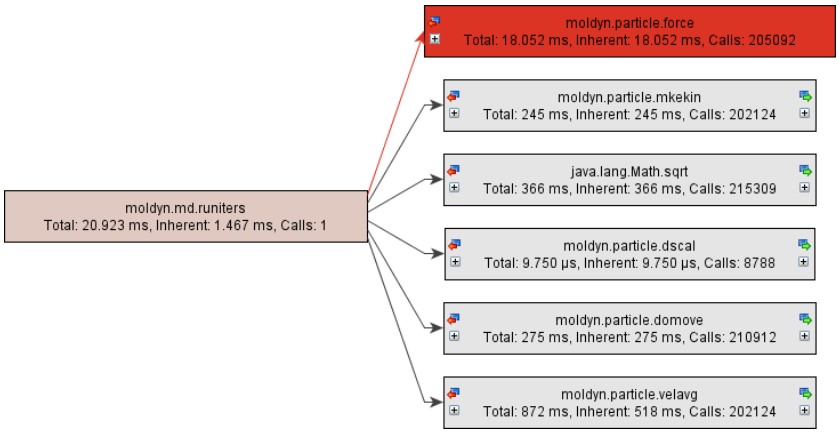


Figura 4.2: Tempo de execução das funções do Moldyn.

4.2 Perfil de Execução dos Algoritmos Sequenciais

Para detectar os estrangulamentos na eficiência do programa, foram utilizadas ferramentas que analisam o perfil de execução dos programas. Devido a essa necessidade optou-se por converter parte do código do *Moldyn* para a linguagem *C++*, evitando também o eventual impacto introduzido pela utilizado pela utilização de uma máquina virtual.

No código em *Java* foi utilizado o *JProfiler* [Tec10] para ver o perfil de execução (figura 4.2), mas a aplicação não se revelou útil na análise

de informação de mais baixo nível (e.g. acessos à memória, contador de instruções).

Neste capítulo pretende-se abordar os estrangulamentos da eficiência de cada algoritmo referido anteriormente. Para identificar as limitações existentes no programa foi decidido analisar os seguintes aspectos: número de instruções, número de ciclos por instrução, perfil de acesso a memória.

As ferramentas utilizadas para a analisar o perfil de execução da aplicação foram o valgrind [Dev10], e *Performance Application Programming Interface* (PAPI) [DLM+01]. O valgrind foi utilizado para medir os acessos a "cache" e também para criar os gráficos de execução do programa. O PAPI serviu para aceder aos contadores do processador entre os quais o número de ciclos gastos pelo processador para executar o programa, o número de instruções, etc.

A utilização do valgrind é bastante simples mas introduz uma grande sobrecarga na execução do programa. Ao medir os acessos a memória com o valgrind a execução do programa aumenta cerca de cinquenta vezes, este aumento é justificado pelo modo de funcionamento do valgrind. O valgrind utiliza a simulação como técnica para obter o perfil de execução [Dev10]. Como o valgrind utiliza a simulação como método os resultados obtidos são determinísticos.

Em relação ao valgrind o PAPI tem uma utilização mais complexa, pois fornece uma API ao utilizador de forma a este ter acesso aos contadores do processador. Para o utilizador medir o que deseja tem que inserir no código várias chamadas à API do PAPI. De seguida mostramos um pequeno exemplo do código PAPI:

```
#include <papi.h>

...

int main(int argc, char **argv) {
```

```
  int retval;
  int EventSet;
  long long values[1];

  //Inicialização o Papi
    if ((retval = PAPI_library_init(PAPI_VER_CURRENT)) !=
        PAPI_VER_CURRENT) {
    printf("Library initialization error! \n");
    return (1);
  }

  //Criação de eventos
  if ((retval = PAPI_create_eventset(&EventSet)) != PAPI_OK)
    ERROR_RETURN(retval);

  //Adicionar um evento
  if ((retval = PAPI_add_event(EventSet, PAPI_TOT_CYC)) !=
      PAPI_OK)
        ERROR_RETURN(retval);

  //Iniciliza os contadores
  if ((retval = PAPI_start(EventSet)) != PAPI_OK)
    ERROR_RETURN(retval);

  funcao_medida();

  //Pausa nos contadores e colocação dos resultados em
      values
  if ((retval = PAPI_stop(EventSet, values)) != PAPI_OK)
    ERROR_RETURN(retval);
  ...
}
```

O PAPI introduz menos sobrecarga na execução do programa relativamente ao valgrind, pois recorre aos contadores presentes no hardware [TJYD10]. Apesar da sobrecarga introduzida ser baixa, é necessário correr o programa várias vezes de modo a ser possível retirar diferentes contadores do programa [DLM+01], sendo que na mesma execução

é possível obter mais de que um contador.

Os valores apresentados nesta secção foram obtidos numa máquina com as seguintes características:

Processador

- Intel Core2 Duo CPU T7500[1];
- Dois cores a 2.20GHz;
- Front Side bus - 800MHz;

Sistema operativo

- Ubuntu 10.4[2];
- Versão do kernel 2.6.32-24;

Memória

- "Cache" de dados L1 - 32 KB
- "Cache" de Intruções L1 - 32 KB
- "Cache" L2 - 4096 KB
- 2GB de Memória

No caso de estudo deste documento foi necessário a execução do programa quatro vezes de modo a obter os seguintes contadores:

PAPI_TOT_INS Conta o total de instruções do programa;

PAPI_TOT_CYC Conta o total de ciclos do programa;

PAPI_L1_DCA Conta os acessos a "cache" *L1* do programa;

PAPI_L1_DCM Conta os acessos a "cache" *L1* que resultaram em faltas;

[1] http://ark.intel.com/Product.aspx?id=29761
[2] http://www.ubuntu.com/

PAPI_L2_DCA Conta os acessos a "cache" *L2* do programa;

PAPI_L2_DCM Conta os acessos a "cache" *L2* que resultaram em faltas;

Devido ao caso em estudo caber na "cache" *L2*, os valores ai recolhidos não são relevantes para o estudo.

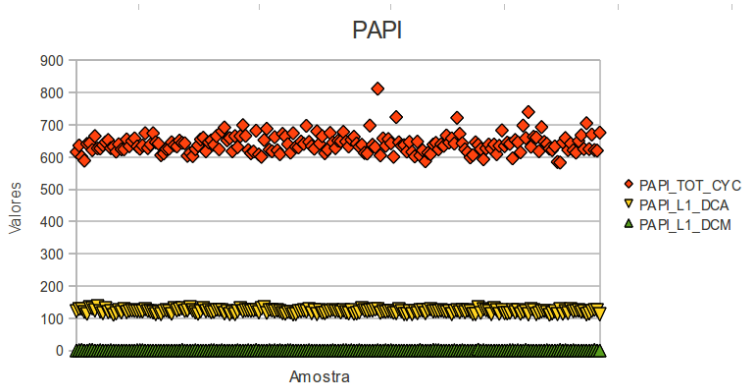


Gráfico 4.1: Sobrecarga introduzida pelo PAPI.

Contrariamente ao valgrind o PAPI não é determinístico, porque a carga do sistema não é sempre a mesma. No gráfico 4.1 é possível verificar as diferenças obtidas em várias execuções e também a sobrecarga introduzida pelo PAPI. Os valores presentes no gráfico foram obtidos, sem qualquer processamento entre a inicialização e paragem dos contadores.

4.2.1 Versão Base

Nesta análise e nas seguintes foi utilizada a versão *C++* reescrita no âmbito deste trabalho. A análise feita a esta versão vai servir como base de comparação. Na tabela 4.1 podemos analisar os resultados obtidos na execução desta versão. Nesta versão base é utilizado um raio de corte de forma a diminuir as interacções a calcular, embora seja necessário aceder a todas as partículas para verificar que estas se encontram dentro do

raio de corte. Quando a interacção é considerada nula (distância entre partículas, maior que o raio de corte), não existe reutilização do valor que foi lido para a "cache", logo a percentagem de faltas no acesso aos dados na *L1* é significativa (tabela 4.1). Na tabela é, ainda possível observar que as faltas ocorridas na *L2* são insignificantes, uma vez que o tamanho do problema é menor que a "cache" *L2*.

	Base
Tempo de execução	35,58s
Faltas na "cache" L1	8.1%
Faltas na "cache" L2	0%
Número de Instruções	1,25 x 10^{11}

Tabela 4.1: Análise da versão base em *C++*.

A figura 4.3 mostra que, tal como na versão *Java*, a função que ocupa a maior parte do tempo de de execução é a função que calcula as forças, e as funções restantes não têm um impacto na execução.

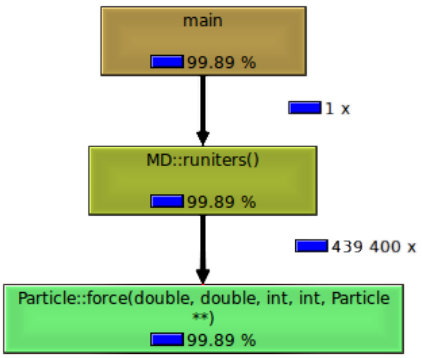


Figura 4.3: Perfil de execução do programa obtido da versão base em *C++*.

4.2.2 Células

A versão de células como foi dito anteriormente reduz significativamente os cálculos desprezáveis e também os acessos a memória, para além disso, a versão de células tem teoricamente uma melhor localidade temporal no acesso aos dados. Isto acontece porque uma vez carregada uma célula para a "cache" esta é utilizada para calcular as interacções de todas as partículas da célula.

A localidade espacial não existe no caso de estudo uma vez que, as partículas são objectos, e quando criamos as partículas não conseguimos controlar o espaço físico na memória onde estas se encontram.

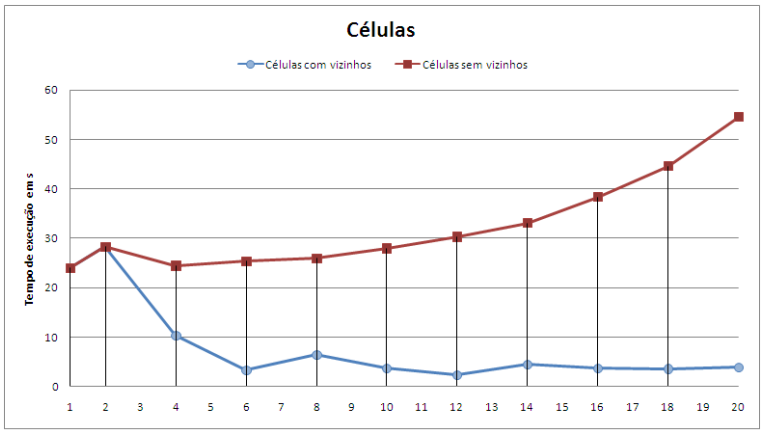


Gráfico 4.2: Comparação entre os tempos de execução das duas versões de células.

Neste caso utilizaremos duas versões diferentes do algoritmo sendo a especificação de cada uma apresentada de seguida:

Células sem vizinhos Esta versão parte o domínio em subdomínios e cada subdomínio tem interacções com os restantes subdomínios. Esta versão permite analisar os acessos a memória, porque não diminui o número de interacções, não diminuindo assim o número de acessos à memória.

Células com vizinhos Neste caso cada célula interage com as células que estão dentro do raio de corte. Sendo assim, diminui o número de interacções e consequentemente o número de acessos à memória.

No gráfico 4.2 é observável as diferenças de tempos obtidos entre as duas versões. O objectivo deste gráfico é mostrar que as diferenças no tempo de execução são significativas. A versão sem vizinhos tem tempos superiores comparativamente à versão com vizinhos. Esta diferença é devida ao cálculo de interacções que não tem impacto no resultado final.

Na versão sem vizinhos existe uma subida nos tempos de execução à medida que é aumentado o número de células. Este aumento é resultado da sobrecarga criada pela versão de células, pois se o número de células aumentar o custo introduzido pela utilização deste algoritmo também aumenta. Os aumentos dos custo de gestão das células são devidos a dois factores:

- Como existem mais células cada uma delas é mais pequena e os movimentos efectuados pelas partículas são os mesmos, as partículas trocam de célula mais frequentemente. Apesar do custo ser pequeno existe um custo associado a esta troca.
- Ao ser utilizado este algoritmo, para cada célula é necessário carregar da memória todas as células com identificador maior que a célula em questão. Quando é aumentado o número de células mais células são necessárias carregar da memória, mesmo que estas não contenham partículas.

Por outro lado, a versão células com vizinhos reduz significativamente o tempo de execução. Essa redução no tempo de execução é devido ao menor número de cálculos redundantes. Contudo a partir de um certo ponto é notada uma ligeira subida nos tempos de execução. Ao aumentarmos o número de células estamos a diminuir o número de partículas por célula, sendo que a partir de certo número de células teremos

células com poucas partículas ou mesmo vazias, e, como foi explicado anteriormente, existe um custo no cálculo destas células.

4.2.2.1 Células com vizinhos

Agora é analisada mais especificamente a versão de células com vizinhos. A grande diferença a nível de código de implementação é no cálculo dos vizinhos. Neste caso os vizinhos são calculados tendo em consideração o raio de corte. Para uma célula A os vizinhos são todas as células com o identificador maior que A (devido à utilização da terceira lei de Newton) e que se encontrem a uma distância menor que o raio de corte.

Como é observável no gráfico 4.3 o tempo de execução é mínimo no caso de termos 6^3 células. Existe uma subida quando existem 2^3 pois como foi explicado anteriormente a versão de células cria uma sobrecarga na execução do programa e não retira partido da "cache" uma vez que a célula não cabe na "cache" *L1*. O tamanho da "cache" *L1* de dados onde os resultados foram obtidos é 32KB. Na tabela 4.2 para cada número de células existentes é mostrado o tamanho ocupado por cada célula.

Como é possível observar na tabela 4.2, quando temos 1^3 e 2^3 células, o tamanho de cada célula é maior que o tamanho da "cache" de dados *L1*, não cabendo portanto uma célula na "cache" *L1*. A partir de 4^3 de células é possível armazenar em "cache" mais de 2 células, diminuindo significativamente o número de faltas na *L1*.

Nesta versão do programa ainda é possível observar que os tempos de execução são mais baixos quando o número de células utilizado é múltiplo de 6^3, a justificação para tal encontra-se na última coluna da tabela 4.2. Na coluna está presente a percentagem do domínio que é considerado no cálculo das interacções. Este valor é relativamente mais baixo quando a dimensão da célula é múltipla do raio de corte. Para tamanhos de células 20^3 este valor também é mínimo contudo o tempo de execução é mais elevado porque o número de partículas por célula é

baixo (aproximadamente 1,1 partícula por célula).

Número de Células	Tamanho de Cada Célula	Domínio considerado em %
1^3	617,9 KB	100
2^3	77,24 KB	100
4^3	**9,655** KB	12,5
6^3	2,861 KB	**3,7**
8^3	1,207 KB	5,3
10^3	0,618 KB	2,7
12^3	0,358 KB	**1,5**
14^3	0,225 KB	2,3
16^3	0,151 KB	1,6
18^3	0,106 KB	**1,1**
20^3	0,077 KB	0,8

Tabela 4.2: Espaço ocupado por cada célula.

No gráfico 4.3 podemos analisar que existe uma melhoria nas faltas de acesso a "cache" *L1*. Por outro lado, também existe uma diminuição no número de instruções; logo a percentagem das faltas no acesso aos dados pode ser influenciada por não se aceder às partículas com interacções consideradas nulas. Para ser mais fácil de analisar a influência da "cache" no algoritmo, é utilizado o algoritmo Células sem Vizinhos para que não exista a diminuição do número de instruções.

4.2.2.2 Células sem Vizinhos

No caso das células sem vizinhos, para a mesma célula A os vizinhos são todos aqueles que tem um identificador maior do que o identificador da célula A. Nesta versão (gráfico 4.4) nota-se claramente a melhoria no acesso aos dados originada pelo uso das células, enquanto o número de

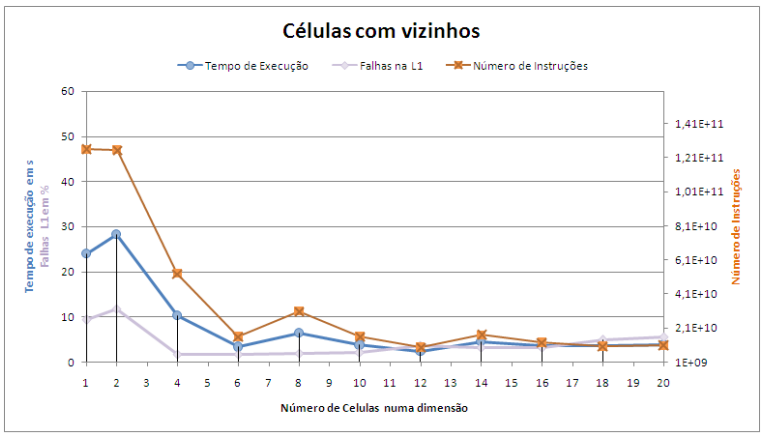


Gráfico 4.3: Perfil de execução da versão células com vizinhos.

instruções tende a aumentar ligeiramente à medida que é aumentado o número de células.

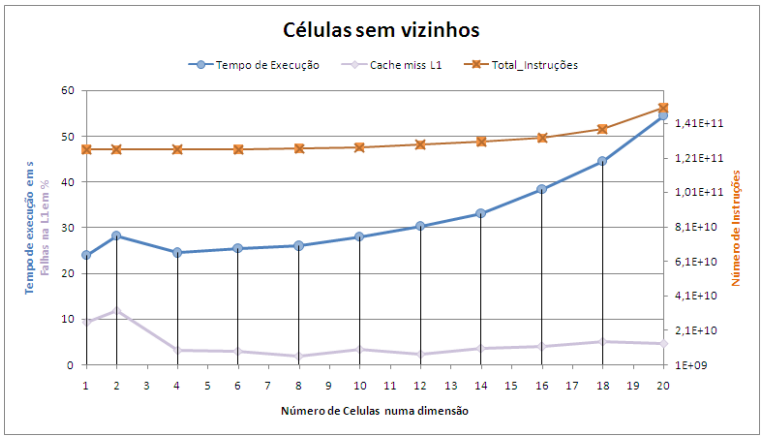


Gráfico 4.4: Perfil de execução da versão células sem vizinhos.

4.2.3 Vizinhos

Relativamente a este algoritmo, é previsível que diminua o número de cálculos em relação aos algoritmos anteriores, mas relativamente às faltas no acesso aos dados é espectável que sejam superiores. Pois, como o algoritmo não tem em consideração as posições que as partículas ocupam, não há localidade temporal.

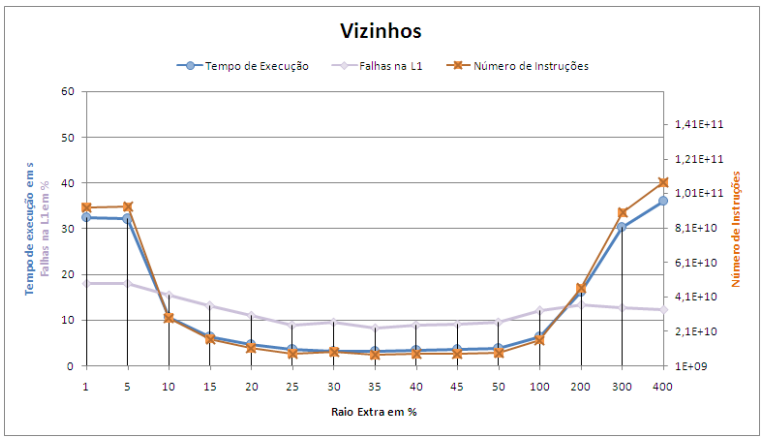


Gráfico 4.5: Analise de perfil de execução da versão vizinhos.

No eixo do x do gráfico 4.5 estão os valores do raio extra que é usado no cálculo da lista de vizinhos. As partículas consideradas vizinhas tem que estar dentro deste raio, sendo este igual ao raio de corte mais o raio extra que é resultado de uma constante multiplicada por o raio de corte.

Os tempos obtidos no gráfico 4.5 comprovam o que foi afirmado anteriormente, pois o tempo de execução é proporcional ao número de instruções. Quanto às faltas no acesso aos dados, se compararmos com resultados obtidos nas outras versões, são bastante elevados aproximando-se dos valores obtidos na versão base quando raio extra é de 35%.

À medida que aumentarmos o raio extra o tempo de execução diminui, passando a aumentar a partir de raios extra maiores que cerca de 35%. Com raios pequenos o tempo é mais elevado devido à actualização da lista de vizinhos em todas as iterações. Para raios grandes, o tempo de execução, também é elevado, porque todas as partículas são vizinhas das outras deixando portanto, de beneficiar da lista de vizinhos.

Os valores das faltas no acesso aos dados, quando o raio extra é baixo, são superiores ao dobro das faltas de acesso aos dados existentes versão base. Nesta situação a lista de vizinhos é actualizada a todas as iterações, logo somos obrigados a percorrer a lista de partículas duas vezes: uma para actualizar a lista e uma segunda para calcular as forças.

Por cada vez que a lista de partículas é percorrida, esta é carregada para a "cache" *L1*, duplicando assim as faltas no acesso aos dados.

4.3 Linha de Produtos com POA

4.3.1 Arquitectura

Na dinâmica molecular podemos utilizar vários algoritmos para a resolução do problema (explicados anteriormente). Neste trabalho são implementados vários algoritmos que foram descritos anteriormente. O código foi desenvolvido utilizando o *AspectJ*, que segue a POA apresentada na secção 3.1 para implementar uma linha de produtos de algoritmos de simulação de dinâmica molecular.

O código base de dinâmica molecular é o *Moldyn* que se encontra no JGF, o algoritmo presente no código do *Moldyn* segue o algoritmo todos os pares, embora contenha algumas optimizações da qual se destaca a utilização da terceira lei de Newton, evitando assim alguns dos cálculos.

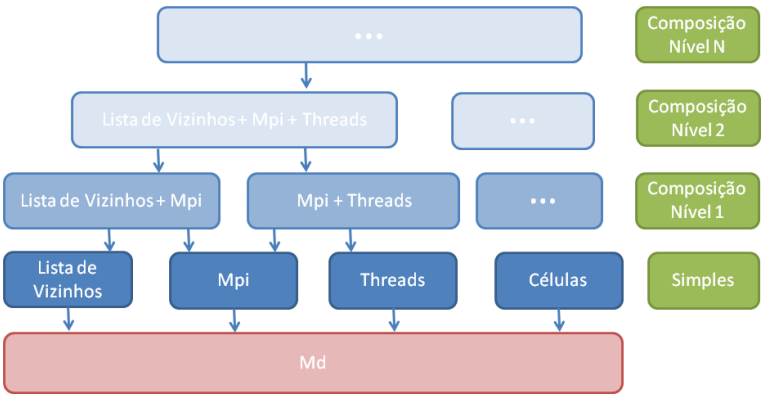


Figura 4.4: Estrutura dos aspectos

Na figura 4.4 são ilustrados os aspectos isto é, "deltas" ou funcionalidades criados para modificar o algoritmo base utilizado no Moldyn, podendo observar-se as combinações possíveis. O quadrado denominado

por *Md* representa o algoritmo base utilizado no Moldyn. Na segunda linha (denominada por *Simples*), encontram-se os aspectos que podem ser utilizados directamente sobre o algoritmo base. Nas restantes linhas encontram-se os aspectos, que representam composições de aspectos anteriores. Existem modificações ao algoritmo base não representadas na figura.

Foi necessário representar as restrições das diferentes combinações possíveis, para tal, recorreu-se a um diagrama de funcionalidades presente na figura 4.5. Na funcionalidade denominada por *Base* encontra-se o algoritmo base da JGF, este pode ser combinado com as funcionalidades presentes na funcionalidade *Par_ Vizinhos* ou em alternativa ser combinado com a funcionalidade Células. Na funcionalidade denominada por *Par_ Vizinhos* pode-se utilizar a funcionalidade *Lista de Vizinhos* e ou a funcionalidade *Partição de partículas*, sendo que neste caso pode ser utilizada a funcionalidade *Threads* e ou *MPI*.

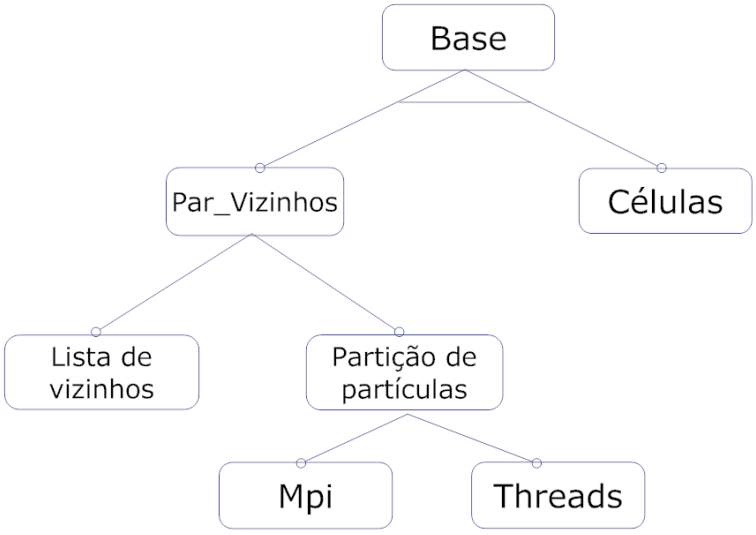


Figura 4.5: Possibilidade de combinação das funcionalidades [Bat05].

Como podemos observar na figura 4.4 são mostradas as diferentes opções que podemos utilizar. Na linha denominada por *Simples* são mostradas as opções que podem ser tomadas. Esta opções estão descri-

tas seguidamente:

Células O algoritmo utilizado passa a ser o da *Decomposição de Células*.

Lista de Vizinhos O algoritmo utilizado para o cálculo é o da *Lista de Vizinhos*.

Mpi O algoritmo utilizado passa a ser o da partição de partículas distribuindo as diferentes partículas por um determinado número de processos, adequado para um sistema alvo de memória distribuída (Clusters).

Threads A utilização deste aspecto é parecida com a do aspecto anterior, mas as partículas são distribuídas por fios de execução em vez de processos, este aspecto é adequado para sistema alvo com memória partilhada (multi-cores).

Por outro lado, podemos utilizar combinações dos aspectos anteriores, estas estão representadas na figura por *Composição Nível 1* e descritas seguidamente:

Mpi + Threads Distribui o cálculo das partículas por processos e fios de execução.

Lista de Vizinhos + Mpi Utiliza a *Lista de Vizinhos*, dividindo o cálculo das forças pelos diferentes processos.

Lista de Vizinhos + Threads Junta a *Lista de Vizinhos* com a utilização do aspecto denominado por *Threads*.

Na linha denominada por *Composição Nível 2* encontra-se a combinação denominada por *Lista de Vizinhos + Mpi + Threads* sendo que esta combinação combina os aspectos *Lista de Vizinhos*, *Mpi* e *Threads*.

4.3.2 Implementação

Nesta secção é pretendido que o utilizador perceba genericamente a implementação dos diferentes aspectos descritos anteriormente.

4.3.2.1 Células

O aspecto *Células* permite adicionar ao algoritmo base a *Divisão por células*. A base deste algoritmo consiste em criar diferentes simulações em cada célula, para isso foi necessário efectuar o seguinte:

- Na chamada do método relativo aos cálculos *runiters*:
  - Inicializa as células:
    - Cria as células, corresponde a criar várias simulações iguais à que está a correr;
    - Define os limites de cada célula;
    - Coloca as partículas nas respectivas células;
  - Define a vizinhança de cada célula;
- Intercepta a função responsável pelos movimentos da partícula:
  - Executa o movimento da partícula;
  - Move as partículas entre células:
    - Remove todas as partículas que se encontrem fora da célula;
    - Coloca as partículas que foram removidas na respectiva célula;
- Intercepta as restantes funções (*cicle_forces*, *cicle_mkekin* , *cicle_velavg*, *cicle_dscal*):
  - Corre a função para cada célula;
  - Reduz o resultado com o respectivo operador;

4.3.2.2 Lista de Vizinhos

O aspecto denominado por *Lista de Vizinhos* possibilita o uso do algoritmo *Lista de Vizinhos*. Este aspecto efectua as seguintes alterações ao algoritmo base:

- Adiciona a cada partícula um array para guardar a lista de vizinhos da partícula;
- Adiciona a cada partícula um array de posições, este contêm a posição ocupada na última actualização da lista de vizinhos;
- Inicia o array com os vizinhos de cada partícula;
- Intercepta a função que calcula as forças para:
  - Efectuar a actualização das partículas **se** existir alguma partícula que se tenha movido mais de que o raio de corte desde a última actualização;
  - Intercepta o acesso ao array de partículas, para trocar este acesso por um acesso à lista de vizinhos;
- Intercepta a função que efectua os movimentos, para calcular a partícula que efectua o maior movimento desde a última actualização da lista de vizinhos, sendo este valor é utilizado para decidir se é necessária actualizar a lista de vizinhos;

De modo a poder saber qual a partícula que se deslocou mais na última iteração é necessário, antes de efectuar o movimento, proceder à inicialização de uma variável a zero; para tal precisamos de interceptar o ciclo abaixo apresentado.

```
public void runiters() {
  move = 0;
  for (move = 0; move < movemx; move++) {
    corpo_do_ciclo
```

```
  }
}
```

Para interceptar este ciclo surgiu uma limitação do *AspectJ*, pois não nos permite interceptar ciclos. Uma primeira abordagem possível seria colocar o corpo do ciclo num método à parte (tabela 4.3). Com esta abordagem foi possível resolver alguns dos problemas encontrados na intercepção de ciclos. Mas nem sempre é viável fazer esta alteração como no caso de termos variáveis locais do método. Neste caso estas devem ser passadas como parâmetro do método. Assim, a tarefa de colocar o corpo do ciclo num método pode ser uma tarefa complexa e obrigar a várias modificações no código base.

Código Base	Código Alterado

`public void metodoA{`
`...`
`  for(int i=MIN; i < MAX; i++){`
`  corpo do ciclo`
`  }`
`...`
`}`
	`public void metodoA{`
	`...`
	`  for(int i=0; i < MAX; i++){`
	`  metodoB(...);`
	`  }`
	`...`
	`}`
	`public void metodoB{`
	`  corpo do ciclo`
	`}`

Tabela 4.3: Solução para interceptar ciclos

Na tabela 4.4 é ilustrada outra possibilidade para resolver esta problemática. Esta solução revelou-se útil na *Lista de Vizinhos*, pois para cada partícula necessitamos de iterar sobre o array que contém a lista de vizinhos e aceder na mesma ao array das partículas. Sendo assim só precisamos de interceptar a função *max*, *principio* e *fim* e alterar o valor de retorno.

Código Base	Código Alterado

`public void metodoA{`
`...`
`for(int i=MIN; i < MAX; i++){`
`corpo do ciclo`
`}`
`...`
`}`
	`public void metodoA{`
	`...`
	`inc = incremento();`
	`min = min();`
	`max = max();`
	`for(int i=min; i < max; i+=inc){`
	`i = pricipio(i);`
	`corpo do ciclo`
	`i = fim(i);`
	`}`
	`...`
	`}`
	`private int incremento(){`
	`return 1;`
	`}`
	`private int min(){ return MIN;}`
	`private int max(){ return MAX;}`
	`private int principio(int i){`
	`return i;`
	`}`
	`private int fim(int i){ return i;}`

Tabela 4.4: Segunda opção para interceptar ciclos

4.3.2.3 Mpi

O aspecto denominado por *Mpi* permite que o programa seja executado em paralelo em sistemas alvo de memória distribuída. O paralelismo é feito através de processos e passagem de mensagens entre processos. A partição do trabalho é feita através da posição que cada partícula ocupa no array de partículas. Ao processo é atribuído um conjunto de partículas intercaladas por processo, logo é utilizado o algoritmo *Partição de partículas* (tabela 4.5).

Este tipo de divisão é mais aconselhada porque as partículas que

são processadas em último lugar têm menos interacções para calcular, isto acontece devida à utilização da terceira lei de Newton, que evita calcular a mesma força duas vezes. Neste aspecto é necessário efectuar as seguintes operações:

- Interceptar a função inicial do programa, para inicializar o Mpi;
- Interceptar todos os métodos que só devam ser executados por um processo (validação de resultados, impressão de tempos...);
- Interceptar o fim da fase de cálculo das forças para sincronizar os dados entre processos;
- Interceptar o cálculo da força de cada partícula, de forma a que esta só seja calculada num processo;

4.3.2.4 Threads

Outra possibilidade para executar o programa em paralelo é a utilização do aspecto *Threads* que como o nome indica executa o cálculo em múltiplas threads, a divisão feita pelas diferentes threads é igual à do aspecto *Mpi* (tabela 4.5). O aspecto efectua as seguinte alterações no algoritmo:

- Criar um array por thread de modo a que cada thread tenha um espaço de escrita independente;
- Interceptar o método relativo à terceira lei de Newton, e escrever a variável na parte reservada a thread;
- Intercepta os métodos de escrita das variáveis *md.interactions, md.epot, md.vir*, e escreve a variável na parte reservada a thread;
- Interceptar o cálculo das forças:

- Distribuir as partículas pelas diferentes threads (neste passo são criadas às threads ou adicionado trabalho às threads que se encontram em execução);
- Reduzir os valores dos array armazenados em cada thread no final do cálculo das forças;

4.3.2.5 Híbridos

Vários algoritmos podem ser obtidos por composição dos anteriores, mas algumas dessas composições requerem aspectos para lidar com particularidades dessa composição, no caso de queremos combinar a *Lista de Vizinhos* com a versão *Threads* não é necessário qualquer aspecto adicional.

O aspecto *Lista de Vizinhos+Mpi* permite que utilizemos a *Lista de Vizinhos* com a versão *Mpi*, este aspecto é necessário para definirmos a ordem pelo qual os aspectos são aplicados. O primeiro aspecto a ser aplicado é o *Mpi* seguido da *Lista de Vizinhos*. A necessidade de definir a ordem deve-se à necessidade de actualizar a lista de vizinhos em todos os processos, caso contrário cada processo só actualizaria a lista das partículas que tem de calcular.

No *Mpi + Threads* é necessário um aspecto adicional para dividir o trabalho entre as threads e os processos. O cálculo é dividido primeiro pelos processos e após a divisão cada thread de cada processo procede a uma divisão semelhante (tabela 4.5).

4.3.3 Resultados Obtidos

4.3.3.1 Ambiente de Teste

As diferentes versões dos algoritmos foram executadas no cluster Search[3]. Nesta secção serão analisados os resultados das diferentes versões

[3]http://search.di.uminho.pt/wordpress/

Posição da partícula	Threads	Mpi	Mpi + Threads
0	0	0	0+0
1	1	1	1+0
2	2	0	0+1
3	0	1	1+1
4	1	0	0+2
5	2	1	1+2
6	0	0	0+0
7	1	1	0+1

Tabela 4.5: Partição do domínio por partículas
Exemplo ilustrativo de como é feita a divisão do domínio. O número de threads é igual a 3 e o número de processos é igual a 2

descritas.

As versões foram corridas em quatro dos nodos denominados por *compute-311-X* que tem as seguintes características:

Processador

- Dois processadores Intel Xeon Processor E5420[4];
- Quatro cores a 2.5 GHz;
- Front Side bus - 1333 MHz;

Sistema operativo

- Rocks 5.2[5];
- CentOS 5.3;
- Versão do kernel 2.6.18;

Memória

[4]http://ark.intel.com/Product.aspx?id=33927
[5]http://www.rocksclusters.org/wordpress/

- "Cache" de dados L1 - 32 KB
- "Cache" de Instruções L1 - 32 KB
- "Cache" L2 - 12MB
- 8GB de Memória

Software

- *Java* - Java Sun 1.6.0_18[6]
- AspectJ - AspectJ 1.6.9[7]

Os valores são obtidos através da média de dez execuções eliminando os extremos (valor min e max).

4.3.3.2 Resultados e Discussão

Na tabela 4.6 são ilustradas as diferenças entre as versões base em *C++* e *Java* os tempos obtidos em cada versão estão na mesma ordem de grandeza. As diferenças nos tempos podem ser justificadas pela utilização de linguagens diferentes e diferentes optimizações utilizadas por cada uma.

Versão	Tempos
C++	26,45s
Java	23,69s

Tabela 4.6: Comparação entre a versão *C++* utilizada no secção 4.2 e a versão *Java* presente na secção 4.3. O algoritmo utilizado é o Base.

O gráfico 4.6 mostra que os tempos de execução na versão de células com a lista de vizinhos em *Java* seguem a mesma tendência da versão em *C++*.

[6]http://www.oracle.com/us/technologies/java/index.html
[7]http://www.eclipse.org/aspectj/

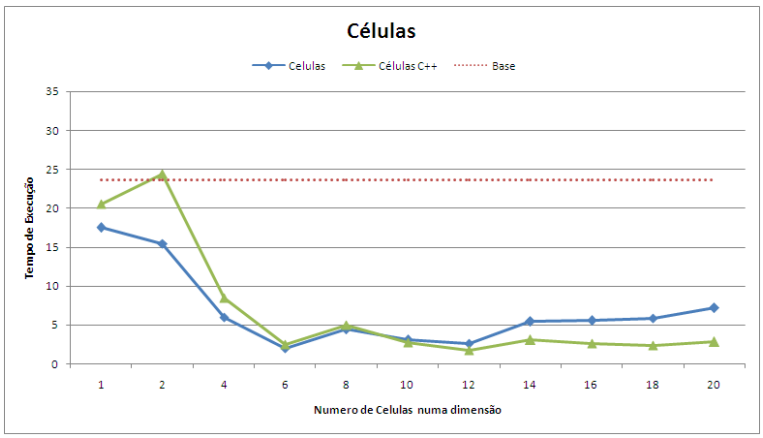


Gráfico 4.6: Tempos obtidos na versão de célula.

No caso das células em *Java* não existe a subida como no caso da versão *C++* quando passamos de uma célula para duas, a justificação para tal é a diferença nas linguagens utilizadas. Quanto comparamos esta versão com a versão base deparamos-nos com uma melhoria significativa em todos os casos de teste.

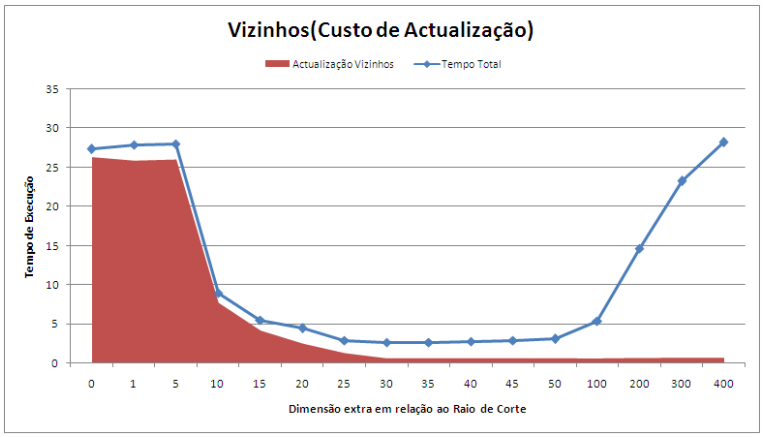


Gráfico 4.7: Análise do tempo para actualizar a lista de vizinhos.

Nos gráficos 4.7 e 4.8 esta representado o algoritmo *Lista de Vizinhos*. Os tempos de execução são equivalentes aos tempos de execução da versão em *C++*, contudo esta versão tem um tempo superior à versão base quando utilizamos raios extra demasiado pequenos ou demasiado

grandes. No gráfico 4.7 é possível observar que o tempo de cálculo para raios pequenos é basicamente o tempo de cálculo da actualização da lista de vizinhos. Esta linha mostra ainda que a partir de um certo ponto o tempo de actualização da lista permanece constante pois a lista só é actualizada uma única vez.

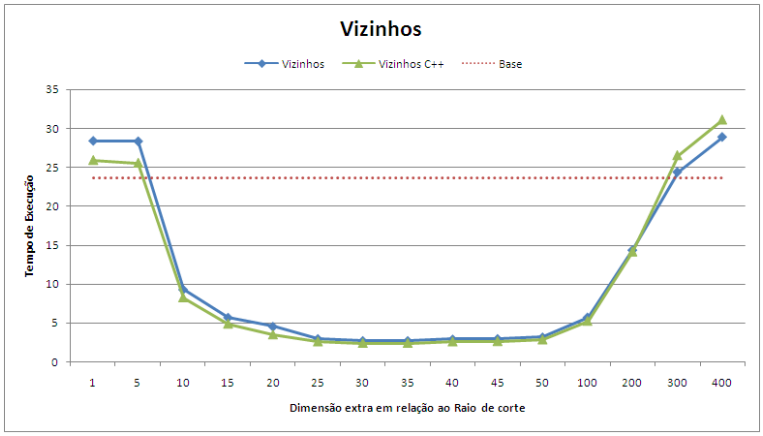


Gráfico 4.8: Tempos obtidos na versão de vizinhos.

Os resultados obtidos no algoritmo *Threads*, estão presentes no gráfico 4.9. Como podemos observar existe um aumento do tempo de execução com uma thread em relação à versão base. O tempo mínimo de execução é quando o número de threads é igual a oito, este número é justificado pelas capacidades da máquina (oito cores) onde foi feita a execução. Contudo os tempos de execução são próximos dos valores teóricos (segundo a lei de Amdahl), tendo em atenção as capacidades da máquina.

A versão desenvolvida com POA mostra-se mais eficiente, do que a versão da JGF, quando o número de threads é maior que quatro. Nesta versão é utilizado um *executor* criado no início da execução do programa e onde as diferentes tarefas (divisão cálculo das forças por partículas) são executadas. Por outro lado, a versão da JGF utiliza o modelo, onde no início da execução do programa são criadas as várias threads e cada uma é responsável por processar uma parte dos dados,

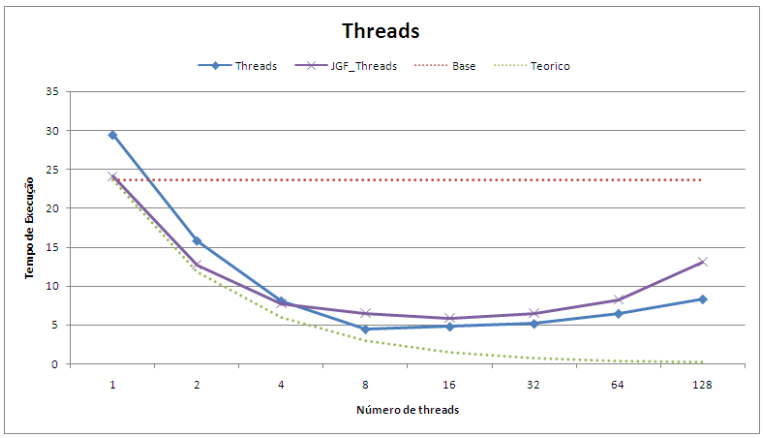


Gráfico 4.9: Tempos obtidos na versão de threads.

embora seja necessário sincronizar o trabalho entre as várias threads, para tal são utilizadas barreiras. Estás são responsáveis pela degradação da performance em comparação com a versão com *executores*. A versão com *executores* introduz uma sobrecarga relativa a criação do executor.

No gráfico 4.10 são representados os tempos de execução da versão *MPI*. Os valores obtidos são próximos dos valores teóricos, sempre melhores ou iguais que a versão da JGF. Existe um afastamento dos valores teóricos quando o número de processos é maior que quatro, porque o tempo gasto no processamento das forças é relativamente mais baixo que o tempo gasto na comunicação. Para cento e vinte e oito processos aumenta o tempo de execução devido a limitação do número das máquinas utilizadas o que implica a utilização de mais de que um processo por core disponível.

As versões *Híbridas* foram executadas com várias combinações de threads e processos, sendo que o número de threads variou entre 1, 2, 4 e 8 e o número de processos variou entre 1, 2, 4, 8. Para cada número de UPs (threads x processos), foi escolhido a combinação que obtinha tempos mais baixos.

Os tempos representados no gráfico 4.11 foram obtidos utilizando a versão *Threads + MPI*. Para valores de número de UPs inferiores a

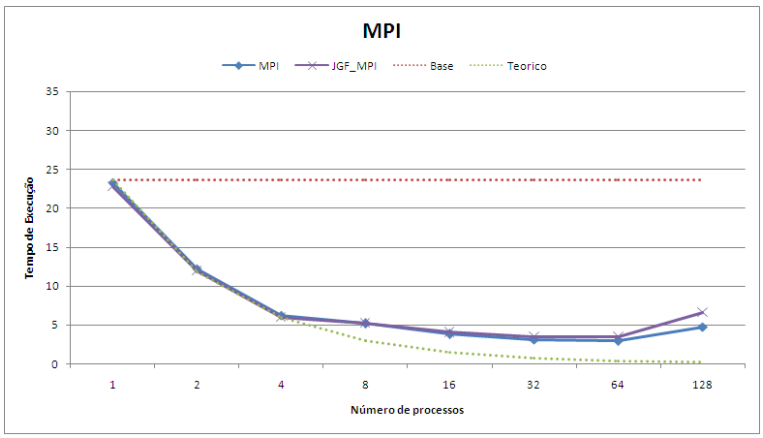


Gráfico 4.10: Tempos obtidos na versão de MPI.

oito, a curva seguida pela versão é igual a curva da versão *Threads*, pois nesta versão existe a mesma sobrecarga que na versão *Threads*. Contudo para números UPs maiores que oito, a versão tem tempos de execução inferiores às outras duas versões, pois neste caso retira partido da memória partilhada e da memória distribuída.

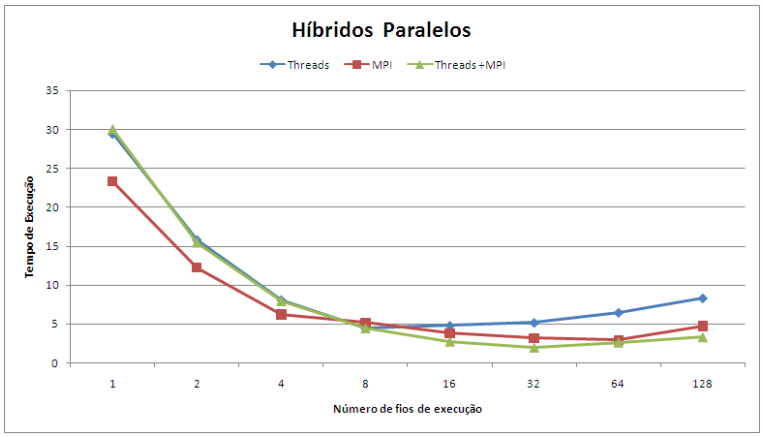


Gráfico 4.11: Tempos obtidos nas versões híbridas.

Por fim foi analisada a versão *Vizinhos*, composta com as versões *Threads*, *MPI* e *Threads + MPI*. Para uma melhor análise foi decidido fixar o raio extra em 40%.

Os resultados obtidos estão presente no gráfico 4.12, onde é possí-

vel observar que a execução com um tempo menor é quando utilizamos a versão *Vizinhos + Threads + MPI*, e são utilizadas dezasseis UPs. Esta versão e a versão *Vizinhos + Threads* tem uma sobrecarga inicial relativamente a versão sequencial base com vizinhos. Como seria previsível a versão *Vizinhos + Threads* tem um aumento no tempo de execução quando são utilizadas mais de oito UPs, pois são mapeadas várias threads por core.

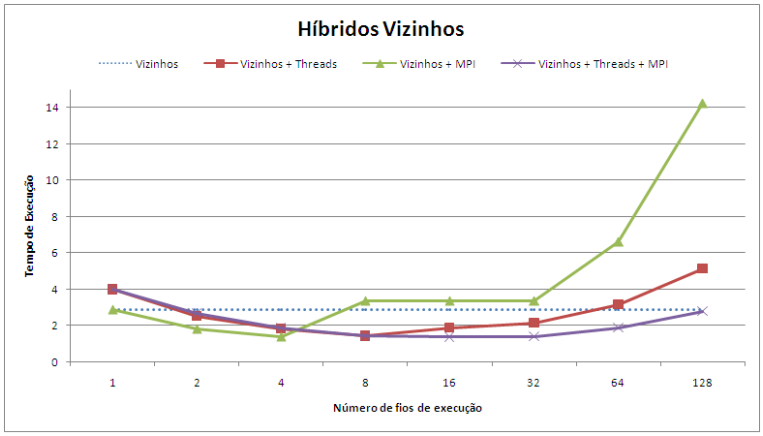


Gráfico 4.12: Tempos obtidos na versão híbrida com vizinhos.

Quanto à versão *Vizinhos + MPI* tem tempos de execução menores de que a versão *Vizinhos* para número de UPs utilizadas inferiores a oito. Para números de UPs maiores existe uma quebra significativa na performance, sendo este valor justificado pela comunicação necessária entre os processos *MPI*.

Após a análise das várias versões criadas com POA, e comparando-as com as versões *C++* e JGF, é possível concluir que os tempos obtidos são comparáveis e que as versões híbridas, resultantes da composição de optimizações, apresentam os menores tempos de execução.

Capítulo 5

Conclusão e Trabalho Futuro

5.1 Conclusão

Este trabalho baseia-se nos algoritmos de dinâmica molecular para o estudo das técnicas de "paralelização" e optimização não invasiva. A dinâmica molecular foi utilizada visto que existe bastante trabalho na parte da optimização e "paralelização". No início do trabalho foram estudados diferentes algoritmos utilizados. Embora exista muito trabalho na área não existe um catálogo de algoritmos, logo fomos obrigados a esforço adicional para perceber algumas das técnicas utilizadas. Pensamos ter contribuído para um melhor conhecimento das técnicas de optimização utilizadas na dinâmica molecular.

Ao longo do trabalho efectuado verificou-se que o caso de estudo é bastante rico para o estudo das técnicas não invasivas e para a exploração de metodologias de desenvolvimento baseadas em linha de produtos, pois existem muitas alternativas do algoritmo (sequenciais e paralelas). Os algoritmos utilizados na dinâmica molecular têm bastante influência no desempenho, como foi analisado neste trabalho. Esta característica torna atractiva a utilização de optimizações não invasivas e linhas de produtos por forma a poder gerar a versão do algoritmo mais adequada

para uma dada simulação e plataforma de execução.

De referir ainda, que este estudo contribui para reforçar a ideia que a "paralelização" e optimização não invasiva pode ser aplicada neste tipo de algoritmos. Foram criadas nove versões diferentes com a utilização desta técnica mantendo o algoritmo base intacto. A criação de nove versões foi possível graças à utilização da técnica, pois estas são resultado da composição de quatro aspectos que são Vizinhos, Células, Threads e Mpi. Contudo foi necessária a alteração de algum código base da JGF, visto que *AspectJ* tem algumas limitações como é o caso da intercepção dos ciclos.

Outra das contribuições deste trabalho foi ao nível das ferramentas de análise de perfil e a importância da sua utilização. No caso do *JProfiler* utilizado na versão *Java*, este revelou-se de pouca utilidade pois só permite obter o perfil de execução do programa. A análise de desempenho pode ter um custo pois cria sobrecarga na execução e no caso do PAPI é necessário a criação de código extra.

A importância da utilização de ferramentas para medir o perfil de execução implicou o desenvolvimento de uma segunda versão do código na linguagem *C++*. A necessidade da criação da versão *C++* prende-se com a existência de melhores ferramentas para à analise da performance nessa linguagem.

Uma das questões ao nível de utilização destas técnicas prende-se com a performance obtida. No caso deste estudo a performance obtida com a utilização das técnicas é semelhante a performance obtida com as técnicas tradicionais. As versões híbridas, obtidas através da composição de optimizações mais simples, superam mesmo as versões já existentes. Logo podemos concluir que a utilização destas técnicas é válida no nosso caso de estudo, tendo como principal vantagem possibilitar a construção de versões mais complexas do algoritmo através da composição de optimizações mais simples.

5.2 Trabalho Futuro

No âmbito deste trabalho surgiram novas perspectivas de estudo uma delas seria a organização dos aspectos por níveis de abstracção. Por exemplo no caso da partição de partículas teríamos um nível que definiria o que são as diferentes tarefas (cada partícula calcula numa tarefa a sua força) e noutro nível existiriam mapeamentos para as diferentes plataformas. Assim existiria uma melhor separação entre o código dependente e independente da plataforma, promovendo ainda mais reutilização de código.

Com a organização dos aspectos por níveis, surge a oportunidade de estudar o mapeamento noutras plataformas não incluídas neste estudo e analisar mudanças mais drásticas no tipo de plataforma, como é o caso da utilização de *GPU* e *GRID*.

De modo a desenvolver versões paralelas mais eficientes, é necessário estender a análise do perfil a estas versões. Para tal existem duas possibilidades em aberto:

- Passagem do código destas versões para *C++*.
- Desenvolvimento de uma ferramenta de análise de código que possibilite um estudo equivalente em *Java*.

Por fim, seria importante aplicar estas técnicas a software utilizado para casos reais, como é o caso de *MOIL* [ERS+95]. Este pacote de simulação de dinâmica molecular é desenvolvido na Universidade do Texas, com a qual existem actualmente várias colaborações. Este pacote está disponível em várias plataformas (*GPU*, *Cluster* e multi-core), mas praticamente não existe reutilização de código entre as várias versões.

Bibliografia

[ALRS05] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In *In Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 125–140. Springer, 2005.

[AR07] S. Akhter and J. Roberts. Multi-Core Programming. *Publishing House of Electronics*, 2007.

[Bat05] D. Batory. Feature models, grammars, and propositional formulas. *Software Product Lines*, pages 7–20, 2005.

[BDS05] K.J. Bowers, R.O. Dror, and D.E. Shaw. Overview of neutral territory methods for the parallel evaluation of pairwise particle interactions. In *Journal of Physics: Conference Series*, volume 16, page 300. IOP Publishing, 2005.

[BSD95] H. J. C. Berendsen, D. Van Der Spoel, and R. Van Drunen. Gromacs: A message-passing parallel molecular dynamics implementation. *Comp. Phys. Comm*, 91:43–56, 1995.

[BSW+00] M. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. Benchmarking java grande applications. In *Proceedings of the Second International Conference on The Practical Applications of Java, Manchester, UK*, pages 63–73, 2000.

[Cad09] Kevin Cador. Aspect Oriented Programming. http://thinkitdifferently.wordpress.com/2009/02/12 /aopwithaspectj, 2009.

[CN02] P. Clements and L. Northrop. Software product lines. *Practices and Patterns Boston: Addison-Wesley*, 2002.

[DB09] C. Dangelmayr and W. Blochinger. Aspect-oriented component assembly-a case study in parallel software design. *Software: Practice and Experience*, 39(9):807–832, 2009.

[DCB09] B. Delaware, W. Cook, and D. Batory. A machine-checked model of safe composition. In *Proceedings of the 2009 workshop on Foundations of aspect-oriented languages*, pages 31–35. ACM, 2009.

[Dev10] V. Developers. Valgrind user manual, 2010.

[DLM+01] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI for hardware performance monitoring on Linux systems. In *Conference on Linux Clusters: The HPC Revolution*, 2001.

[dP+10] J.H.M. de Pinho et al. Pluggable Parallelization of Evolutionary Algorithms Applied to the Optimization of Biological Processes. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 395–402. IEEE, 2010.

[ERS+95] R. Elber, A. Roitberg, C. Simmerling, R. Goldstein, H. Li, G. Verkhivker, C. Keasar, J. Zhang, and A. Ulitsky. MOIL: A program for simulations of macromolecules. *Computer Physics Communications*, 91(1-3):159–189, 1995.

[FEC04] R. Filman, T. Elrad, and S. Clarke. *Aspect-oriented software development*. Addison-Wesley Professional, 2004.

[FEV+09] M.S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A.L. Beberg, D.L. Ensign, C.M. Bruns, and V.S. Pande. Accelerating molecular dynamic simulation on graphics processing units. *Journal of Computational Chemistry*, 30(6):864–872, 2009.

[FSP06] JF Ferreira, JL Sobral, and AJ Proenca. JaSkel: A Java skeleton-based framework for structured cluster and grid computing. In *Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 06*, volume 1, 2006.

[GS09] R.C. Gonçalves and J.L. Sobral. Pluggable parallelisation. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 11–20. ACM New York, NY, USA, 2009.

[HG04] B. Harbulot and J.R. Gurd. Using AspectJ to separate concerns in parallel scientific Java code. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 122–131. ACM, 2004.

[HKvL08] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl. Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *J. Chem. Theory Comput*, 4(3):435–447, 2008.

[KHAK06] S. Kumar, C. Huang, G. Almasi, and L.V. Kalé. Achieving strong scaling with NAMD on Blue Gene/L. In *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, volume 2006, 2006.

[Kno05] Knordlun. Wikipedia Lennard-Jones. http://en.wikipedia.org/wiki/Lennard-Jones_potential, 2005.

[KSB+99] Laxmikant Kalé, Robert Skeel, Milind Bh, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. Namd2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[Nak04] A.S. Nakahara. Análise de Métodos de simulação de Dinâmica Molecular em Arquitecturas Paralelas. Master's thesis, Universidade Estatal Maringá, 2004.

[NHG+96] M.T. Nelson, W. Humphrey, A. Gursoy, A. Dalke, L.V. Kalé, R.D. Skeel, and K. Schulten. NAMD: a parallel, object-oriented molecular dynamics program. *International Journal of High Performance Computing Applications*, 10(4):251, 1996.

[PBVDL05] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag New York Inc, 2005.

[PH95] Steve Plimpton and Bruce Hendrickson. Parallel molecular dynamics algorithms for simulation of molecular systems. In Oxford University Press, editor, *In: T. G. Mattson (ed) Parallel Computing in Computational Chemistry*, pages 114–136, 1995.

[Pin09] Jorge Pinto. Desenvolvimento de Algoritmos Evolucionários para Ambientes Grid com Aplicações à Optimização de Sistemas Biológicos. Master's thesis, Universidade do Minho, 2009.

[Pos07] Poszwa. Wikipedia Dinâmica Molecular. http://en.wikipedia.org/wiki/Molecular_dynamics, 2007.

[Pre01] C. Prehofer. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience*, 13(6):465–501, 2001.

[PZKK06] J.C. Phillips, G. Zheng, S. Kumar, and L.V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Supercomputing, ACM/IEEE 2002 Conference*, page 36. IEEE, 2006.

[SD99] C. Sagui and T.A. Darden. Molecular dynamics simulations of biomolecules: long-range electrostatic effects. *Annual review of biophysics and biomolecular structure*, 28(1):155–179, 1999.

[Sha05] D.E. Shaw. A fast, scalable method for the parallel evaluation of distance-limited pairwise particle interactions. *Journal of computational chemistry*, 26(13):1318–1328, 2005.

[SJ07] G. Sutmann and F. Janoschek. Communication and Load Balancing of Force-Decomposition Algorithms for Parallel Molecular Dynamics. *Parallel Computing: Architectures, Algorithms and Applications*, 2007.

[TBKC07] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering*, page 104. ACM, 2007.

[Tec10] Ej Technologies. Jprofiler user manual. http://resources.ej-technologies.com/ jprofiler/help/doc/help.pdf, 2010.

[TJYD10] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting Performance Data with PAPI-C. *Tools for High Performance Computing 2009*, pages 157–173, 2010.

Apêndice A

Resultados

Versão	Tempos
Java	23,69
C++	26,45

Tabela A.1: Tempos obtidos na versão células

Número de Células Versão	1	2	4	6	8	10	12	14	16	18	20
POA	17,52	15,43	5,98	2,02	4,44	3,04	2,62	5,42	5,54	5,84	7,18
C++	20,51	24,42	8,49	2,53	4,97	2,72	1,74	3,08	2,64	2,41	2,82

Tabela A.2: Tempos obtidos na versão células

Raio Extra Versão	1%	5,00	10%	15%	20%	25%	30%	35%	40%	45%	50%	100%	200%	300%	400%
POA	28,37	28,31	9,26	5,67	4,51	2,93	2,68	2,67	2,87	2,94	3,14	5,70	14,34	24,37	28,87
C++	25,92	25,58	8,25	4,88	3,48	2,58	2,39	2,39	2,58	2,63	2,84	5,23	14,15	26,56	31,14

Tabela A.3: Tempos obtidos na versão vizinhos

Número de UP Versão	1	2	4	8	16	32	64	128
MPI POA	23,28	12,22	6,20	5,17	3,82	3,14	2,94	4,72
MPI JGF	22,76	11,94	5,97	5,24	4,04	3,53	3,50	6,63
Threads POA	29,46	15,83	8,10	4,44	4,77	5,14	6,45	8,31
Threads JGF	24,08	12,70	7,71	6,47	5,86	6,51	8,26	13,08

Tabela A.4: Tempos obtidos nas versões paralelas

Número de UP/ Número de Processos	1	2	4	8	16	32	64	128
1	29,98	16,10	8,33	4,74	5,25			
2		15,46	8,55	4,66	2,87	3,47		
4			7,95	4,70	2,75	1,97	2,68	
8				4,48	2,87	2,11	2,64	3,31

Tabela A.5: Tempos obtidos nas versões híbrida *MPI + Threads*

Número de UP/ Número de Processos	1	2	4	8	16	32	64	128
1	4,02	2,69	2,06	1,89				
2		2,80	2,11	1,74	1,63			
4			1,85	1,61	1,37	1,58		
8				1,42	1,48	1,38	2,00	
16					1,94	1,96	1,86	2,79

Tabela A.6: Tempos obtidos nas versões híbrida *MPI + Threads + Vizinhos*