RESOLUÇÃO DE AGRUPAMENTO UTILIZANDO SUAVIZAÇÃO HIPERBÓLICA COM ARQUITETURA OPENMP

Marcelo Lins de Souza

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Adilson Elias Xavier

Rio de Janeiro
Setembro de 2013

RESOLUÇÃO DE AGRUPAMENTO UTILIZANDO SUAVIZAÇÃO HIPERBÓLICA COM ARQUITETURA OPENMP

Marcelo Lins de Souza

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

____
Prof. Adilson Elias Xavier, D.Sc.

____
Prof. Sergio Barbosa Villas-Boas, Ph.D.

____
Prof. Argimiro Resende Secchi, D.Sc.

____
Prof. Eduardo Uchoa Barboza, D.Sc.

RIO DE JANEIRO – RJ, BRASIL
SETEMBRO DE 2013

Souza, Marcelo Lins de

Resolução de agrupamento utilizando Suavização Hiperbólica com Arquitetura OpenMP/Marcelo Lins de Souza. – Rio de Janeiro: UFRJ/COPPE, 2013.

XII, 54 p.: il.; $29,7$cm.

Orientador: Adilson Elias Xavier

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2013.

Referências Bibliográficas: p. 53 – 54.

1. Suavização 2. Clusterização 3. Paralelismo I. Xavier,Adilson Elias. II.Universidade Federal do Rio de Janeiro, COPPE , Programa de Engenharia de Sistemas e Computação. III. Título

"Eduque às crianças e não será necessário castigar aos homens."(Pitágoras)

Agradecimentos

Agradeço a minha mãe Cecilia, meu pai Souza, minha irmã Cintya, minha tia Lourdes, minha avó e aos primos e amigos por tudo que fizeram ao longo de todos esses anos, sempre me incentivando a alcançar meus objetivos.

A meu orientador Adilson Elias Xavier, pelos conhecimentos transmitidos, pela generosidade e pela oportunidade de concluir esse trabalho.

Ao co-orientador Sergio Barbosa Villas-Boas, que muito me ajudou e que orientou-me de fato. Seu nome não consta como orientador oficial, mas apenas como co-orientador, meramente por questões burocráticas. Ao Leandro Marzullo que ajudou muito no desenvolvimento deste trabalho.

Aos amigos da Coppetec que sempre me incentivaram a dar o meu melhor nesse processo de aprendizado

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre (M.Sc.)

RESOLUÇÃO DE AGRUPAMENTO UTILIZANDO SUAVIZAÇÃO HIPERBÓLICA COM ARQUITETURA OPENMP

Marcelo Lins de Souza

Setembro/2013

Orientador: Adilson Elias Xavier

Programa: Engenharia de Sistemas e Computação

A formulação matemática para solução do problema de agrupamento está associada a um modelo de mímima soma de quadrados. Este modelo produz um problema do tipo min-max-min, que além da sua natureza intrínsecamente multi-nível, tem a significativa característica de ser não diferenciável e de difícil solução (*NP-Hard*).

Com a finalidade de contornar estas dificuldades, utilizou-se a técnica de suavização hiperbólica, onde a solução final é obtida através da transformação do problema original em uma sequência de subproblemas diferenciáveis que gradualmente aproximam-se do problema original.

Neste trabalho propôs-se uma nova implementação para o método baseada na arquitetura OpenMP para paralelizar trechos do código, aproveitando os recursos de múltiplos processadores que os computadores atuais disponibilizam.

Para a comparação dos resultdos do XCM com o XCM-OpenMP, propôs-se uma "Arquitetura Orientada a Objetos para Comparação de Algoritmos".

Para confirmar experimentalmente o método proposto, resultados do XCM original (sequencial) e do XCM-OpenMP foram comparados. O melhor speed-up conseguido foi de 28 vezes. Os resultados obtidos com XCM-OpenMP não apresentam perda de precisão numérica.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

HYPERBOLIC SMOOTHING CLUSTERING RESOLVE USING OPENMP ARCHITECTURE

Marcelo Lins de Souza

September/2013

Advisor: Adilson Elias Xavier

Department: Systems Engineering and Computer Science

The mathematical formulation for solving the clustering problem is associated with a minimum sum of squares model, producing a problem of type min-max-min. This problem has intrinsic multi-level nature and has the significant characteristic of being non-differentiable and NP-Hard.

In order to overcome these difficulties, the hyperbolic smoothing technique is used; the final solution is obtained by transforming the original problem into a sequence of differentiable subproblems which gradually approach the original problem.

In this work we proposed a new implementation of XCM based on the OpenMP architecture to parallelize parts of the code, using the resources of multiple processors that provide current computers. To compare the results between XCM and XCM-OpenMP, an "Object-Oriented architecture to compare algorithms" was proposed.

To confirm experimentally the proposed method, the results of the original XCM (sequential) and the XCM-OpenMP are compared. The best achieved speed-up was 28. The numerical results obtained with XCM-OpenMP not exhibit loss of accuracy.

Sumário

Lista de Figuras **x**

Lista de Tabelas **xii**

1 Introdução **1**

2 Revisão Bibliográfica **4**

2.1 Penalização Hiperbólica 4

2.2 Análise de Agrupamento 5

2.3 Conceitos de Otimização 7

2.4 API de programação paralela 14

2.4.1 OpenMP 14

2.4.2 MPI 14

2.4.3 TBB 15

2.4.4 Cilk 15

2.4.5 Trebuchet 16

2.5 Tipos de Arquiteturas Paralelas 16

2.6 A Arquitetura OpenMP 18

2.6.1 Histórico 18

2.6.2 Modelo OpenMP 19

2.6.3 Sintaxe OpenMP 20

2.6.4 Exemplo de uso 23

3 O Método XCM **24**

3.1 Análise de Agrupamento como um problema min-sum-min 25

4 O Método XCM-OpenMP **34**

4.1 Conceitos Básicos 34

4.2 O algoritmo do XCM-OpenMP 36

4.3 Arquitetura Orientada a Objetos para Comparação de Algoritmos . . 37

5 Resultados Computacionais **40**

5.1 Problemas de Pequeno Porte 41

5.2 Problemas de Médio e Grande Porte 42

6 Conclusão **51**

6.1 Trabalhos Futuros 52

Referências Bibliográficas **53**

Lista de Figuras

2.1 Exemplo de observações, clusters e centroids no $\mathbb{R}^2$. 6

2.2 Modelo de programação OpenMP 19

3.1 Três primeiras parcelas do somatório da equação 3.9 27

3.2 Três primeiras parcelas do somatório da equação 3.9 com suas equivalentes suavizações 29

3.3 Comportamento da função suavizada 32

4.1 Diagrama de Classes da Arquitetura Orientada a Objetos para Comparação de Algoritmos 38

5.1 Caso de teste simples. 41

5.2 Problema TSPLIB-3038 (q = 5). 42

5.3 Problema TSPLIB-3038 (q = 10). 42

5.4 Problema TSPLIB-3038 (q = 20). 43

5.5 Problema TSPLIB-3038 (q = 30). 43

5.6 Problema TSPLIB-3038 (q = 40). 43

5.7 Problema TSPLIB-3038 (q = 50). 44

5.8 Gráfico dos resultados do problema TSPLIB-3038. 44

5.9 Gráfico de speed-up do problema TSPLIB-3038. 45

5.10 Problema com 15112 observações (q = 5). 45

5.11 Problema com 15112 observações (q = 10). 45

5.12 Problema com 15112 observações (q = 15). 46

5.13 Problema com 15112 observações (q = 20). 46

5.14 Problema com 15112 observações (q = 25). 46

5.15 Problema com 15112 observações (q = 30). 47

5.16 Gráfico dos resultados do problema 15112. 48
5.17 Gráfico de speed-up do problema 15112. 48
5.18 Problema com 85900 observações (q = 5). 48
5.19 Problema com 85900 observações (q = 10). 48
5.20 Problema com 85900 observações (q = 20). 49
5.21 Problema com 85900 observações (q = 25). 49
5.22 Problema com 85900 observações (q = 30). 49
5.23 Gráfico dos resultados do problema 85900. 50
5.24 Gráfico de speed-up do problema 85900. 50

Lista de Tabelas

5.1 Resultados de Tempo e Speed Up para o problema TSPLIB-3038 . . 44
5.2 Resultados de Tempo e Speed Up para o problema 15112 47
5.3 Resultados de Tempo e Speed Up para o problema 85900 49

Capítulo 1

Introdução

Atualmente há grande busca por melhorias no desempenho no processamento computacional a partir do uso de computadores *multicore*, que possibilitam a execução de fluxos de processamento em paralelo. Muita pesquisa vem sendo feita no sentido de adaptar algoritmos para que explorem os múltiplos cores dos computadores atuais. Essa adaptação é feita a nível de código fonte.

Há várias tecnlogias e arquiteturas que possibilitam o desenvolvimento de software para computação paralela. Algumas linguagens de programação usadas nas pesquisas que envolvem computação paralela orientada a desempenho competitivo estão listadas abaixo.

- Fortran
- C
- C++

Dentre as tecnologias que se usam por cima dessas linguagens de programação, pode-se destacar as seguintes:

- OpenMP [2]
- MPI [10]
- Intel®Threading Building Blocks (TBB) [14]
- Intel®Cilk Plus [9]

- Trebuchet [1]

Na seção 2.4, na página 14, é feito um comentário das características das linguagens e tecnologias de paralelização.

A formulação do problema de agrupamento como mínima soma de quadrados produz um problema de Otimização Global não-diferenciável da classe NP-Hard [4]. Foi proposto por [17] o *Hyperbolic Smoothing Clustering Method*, também chamado de *Xavier Clustering Method*, ou XCM para encurtar. Os resultados computacionais do XCM, apresentados em [16] e [18], mostram que o método é extremamente eficiente, robusto e preciso, em alguns casos até mesmo encontrando soluções melhores que as encontradas na literatura. O método XCM baseia-se em minimizações irrestritas calculadas por meio do algoritmo BFGS [13], do tipo *Quasi-Newton*. A principal característica vantajosa desse tipo de método de otimização irrestrita é que o cálculo da matriz hessiana é feito internamente pelo algoritmo, sem necessidade de se escrever explicitamente o valor de segunda derivada da função que se deseja otimizar.

O método XCM requer que seja explicitado o cálculo da função e seu gradiente, para que seja chamado pelo otimizador irrestrito BFGS. Tanto o cálculo da função quanto de suas derivadas contém soma de termos sem dependência, que podem ser feitos em paralelo. Nesse trabalho, é feita a análise desses trechos de código, e também a proposição de uma forma de paralelização baseado em OpenMP.

Foram feitos testes experimentais com o algoritmo proposto e verificou-se significativo speedup em vários casos.

O principal objetivo deste trabalho é propor a versão paralela do algorítmo XCM, usando a arquitetura OpenMP. A opção por essa forma de se implementar o paralelismo é explicado no capítulo 4. Para a comparação dos resultdos do XCM com o XCM-OpenMP, propôs-se uma "Arquitetura Orientada a Objetos para Comparação de Algoritmos", em que em uma classe base IXCM foi implementado o código comum entre XCM e XCM-OpenMP. Na classe derivada XCM foi implementada a versão sequencial do XCM, e na classe derivada XCM-OpenMP foi implementada a versão *Multithread* do XCM.

Os resultados computacionais mostram que, sem perda de precisão, são obtidos ganhos consideráveis de performance. Este trabalho está organizado da seguinte

forma: No capítulo 2 é apresentado um breve resumo do conceito de análise de agrupamento, assim como alguns conceitos sobre métodos de minimização irrestrita. Ainda no capítulo 2 são apresentados os detalhes da arquitetura paralela e OpenMP. No capítulo 3 analiza-se o algoritmo XCM. Essa análise identifica os trechos paralelizáveis do algoritmo. A implementação paralela com OpenMP implica em detalhes específicos de alteração do código para que se permita a adequada paralelização, que são explicados no capítulo 4. No capítulo 5 são apresentados os resultados computacionais, principalmente comparando-se as implementações sequencial e paralela do método e no capítulo 6 tem-se as conclusões sobre o trabalho e trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

Neste capítulo são explicados conceitos necessários para o entendimento deste trabalho. Inicialmente são abordados conceitos de análise de agrupamento. Em seguida são apresentados alguns conceitos sobre algoritmos de minimização. Finalmente, a arquitetura OpenMP é apresentada e os principais conceitos necessários ao seu entendimento são descritos.

2.1 Penalização Hiperbólica

A penalização hiperbólica (*hyperbolic penalty*) é uma técnica de processamento numérico com várias aplicações, tais como clustering [18] [20], covering[19], packing, hub-location e outras. Essa técnica foi originalmente proposta por Adilson Xavier. O mais antigo documento escrito que menciona "penalização hiperbólica" é a tese de mestrado de Adilson Xavier [17], em março de 1982.

A partir da técnica original de penalização hiperbólica, foi proposto pelo mesmo autor Adilson Xavier a técnica de clusterização baseada em *suavização hiperbólica*.

A suavização hiperbólica é um conceito derivado da penalização hiperbólica e, quando aplicado a problemas de agrupamento, tem-se um método de clusterização que será chamado nesse trabalho de XCM (Xavier Clustering Method) [18], definido no capítulo 3.

2.2 Análise de Agrupamento

A Análise de Agrupamento (Cluster Analysis, ou Clustering) é um procedimento de estatística multivariada que engloba técnicas que objetivam organizar objetos em grupos de acordo com a proximidade existente entre eles, ou seja, dado um conjunto de observações, são feitos agrupamentos onde observações pertencentes a um mesmo grupo são mais similares entre si do que observações de outros grupos, segundo um critério específico pré-definido. Os objetos de um mesmo grupo são tão similares quanto possível (coesão interna) e ao mesmo tempo tão dissimilares quanto possível dos objetos dos demais grupos (isolamento externo) [3]. Problemas de análise de agrupamento são encontrados nas mais diversas aplicações, como, por exemplo, processamento gráfico de imagens e até mesmo em outros ramos de ciência como medicina e química.

Pode-se entender como observações um conjunto de pontos definidos em um espaço euclidiano de dimensão p. A primeira parte da figura 2.1 ilustra esse conceito, com 6 pontos definidos no $\mathbb{R}^2$. A segregação destas observações em um número q pré-definido de conjuntos, ou *clusters* é o principal objetivo da análise de agrupamento. Uma possível solução para este conjunto de observações com 3 clusters pode ser visto na segunda parte da Figura 2.1. É comum utilizar pontos como centros de gravidade, ou *centroids*, que definem o posicionamento de um cluster no espaço das observações. Esta ideia está ilustrada na parte 3 da Figura 2.1.

Dois principais objetivos devem ser levados em consideração: Homogeneidade e Separação. O objetivo da homogeneidade dita que observações em um mesmo grupo sejam similares entre si, enquanto que o objetivo da separação dita que observações em grupos distintos sejam o mais diferentes quanto possível. O critério definido *a priori* mais simples e intuitivo que pode ser considerado é o da mínima soma de quadrados.

No entanto, a modelagem matemática de um problema de clusterização com critério de mínima soma de quadrados leva à formulação de um problema NP-Hard de otimização global não diferenciável e não convexo, com um grande número de pontos de mínimos locais.

A ideia principal deste tipo de formulação é muito simples: Deseja-se encontrar o posicionamento de q pontos no espaço de dimensão p das m observações de forma

que o somatório das menores distâncias euclidianas de cada uma das m observações a cada um dos q centroids seja o menor possível.

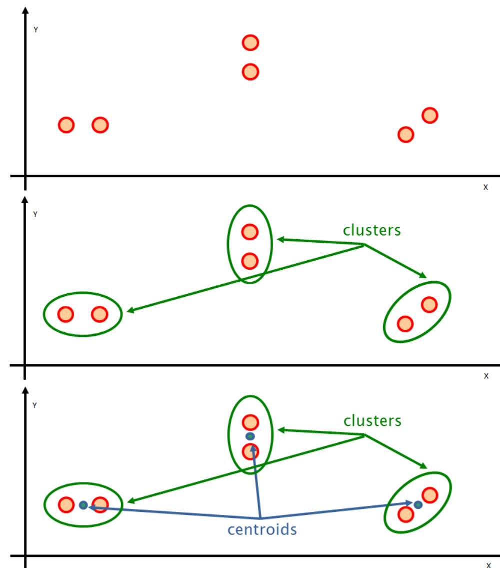

Figura 2.1: Exemplo de observações, clusters e centroids no $\mathbb{R}^2$.

Para se alcançar esse objetivo, um posicionamento inicial para os q centroids é definido segundo um critério qualquer, por exemplo, considerando o ponto médio das m observações e uma dispersão baseada no desvio padrão somada a um número aleatório. Em seguida, para cada observação, é calculado o quadrado da distância euclidiana para todos os q centroids, tomando-se a menor dentre todas as q distâncias calculadas. O somatório dessas m menores distâncias é o valor da função objetivo o qual se deseja minimizar. Na próxima etapa da execução, a posição dos q centroids é modificada (o posicionamento das m observações sempre permanece fixo) e todas as menores distâncias são novamente calculadas. O processo é repetido até que não se consiga mais diminuir o somatório das menores distâncias.

É possível modificar a modelagem matemática do problema original, suavizando-o através de uma técnica conhecida como *Suavização Hiperbólica*. Os detalhes dessa transformação são apresentados no capítulo 3. Essa estratégia de resolução leva à criação de problemas diferenciáveis irrestritos, podendo-se tirar proveito de métodos de minimização irrestritos baseados na primeira derivada.

Existem outras estratégias para resolver problemas de agrupamento, como por exemplo, métodos de partição e métodos hierárquicos. Esses métodos e suas características não serão abordados neste trabalho.

2.3 Conceitos de Otimização

Otimização é uma ferramenta importante na ciência de decisão e na análise de sistemas físicos. Para usá-la, é preciso primeiro identificar algum objetivo, uma medida quantitativa do desempenho do sistema em estudo. O processo de identificação de objetivos, variáveis e restrições para um dado problema é conhecido como modelagem. Quando um modelo for formulado, um algoritmo de otimização pode ser usado para encontrar a sua solução.

Não existe um algoritmo de otimização universal. Pelo contrário, existem inúmeros algoritmos, cada um dos quais é adaptado a um tipo particular de problema de otimização. Muitas vezes, é de responsabilidade do desenvolvedor do modelo escolher um algoritmo que é adequado para a sua aplicação específica.

Após um algoritmo de otimização ser aplicado ao modelo, deve-se ser capaz de reconhecer se obteve-se êxito na tarefa de encontrar uma solução. Em muitos casos, existem elegantes expressões matemáticas conhecidas como *condições de otimalidade* para a verificação se o atual conjunto de variáveis é de fato a solução do problema.

Outra característica usualmente presente nos algoritmos de otimização é o fato de serem algoritmos iterativos. Eles começam com uma estimativa inicial dos valores ideais das variáveis e geram uma sequência de estimativas até chegar a uma solução. A estratégia utilizada para se deslocar de uma iteração para a próxima distingue um algoritmo de outro.

Além dessas características, usualmente os algoritmos de minimização devem possuir as seguintes propriedades:

- Robustez - Eles devem ter um bom desempenho em uma ampla variedade de problemas de sua classe, para escolhas razoáveis das variáveis iniciais.

- Eficiência - Eles não devem exigir muito tempo computacional ou espaço de armazenamento.

- Acurácia - Devem ser capazes de identificar uma solução com acurácia, sem ser excessivamente sensível a erros nos dados ou a erros aritméticos de arredondamento.

Esses objetivos podem ser conflitantes. Por exemplo, um método que converge rapidamente para a solução pode exigir uma capacidade de armazenamento muito

elevada. Por outro lado, um método robusto também pode ser mais lento. Comparações entre a taxa de convergência e requisitos de armazenamento, ou entre robustez e velocidade e assim por diante, são questões centrais na resolução de problemas de otimização numérica.

Dentro do conceito de otimização, muitos outros temas devem ser considerados, tais como: Otimização restrita e irrestrita, otimização global e local e convexidade. Este estudo está focado em Otimização Irrestrita.

Na otimização irrestrita, procura-se minimizar uma função objetivo que depende de variáveis reais, sem restrições nos valores dessas variáveis. Geralmente, o objetivo máximo é encontrar um ponto de mínimo global da função, ou seja, o ponto onde a função atinge seu valor mínimo.

Pode parecer que a única maneira de descobrir se um ponto X^* é um mínimo local é examinar todos os pontos na sua vizinhança imediata, para se certificar de que nenhum deles tem um valor menor. No entanto, quando a função f é suave, existem maneiras muito mais eficientes e práticas de se identificar mínimos locais.

Em particular, se uma função objetivo f é duas vezes continuamente diferenciável, pode-se de dizer que X^* é um ponto de mínimo local, examinando apenas o gradiente $\nabla f(X^*)$ e a Hessiana $\nabla^2 f(X^*)$. A ferramenta matemática usada para estudar mínimos locais de funções suaves é o teorema de Taylor.

Teorema de Taylor: Suponha que $f : \mathbb{R}^n \rightarrow \mathbb{R}$ é continuamente diferenciável e que $p \in \mathbb{R}^n$. Então

$$f(x+p) = f(x) + \nabla f(x+tp)^T p \tag{2.1}$$

para algum $t \in (0,1)$.

Além disso, se f é duas vezes continuamente diferenciável, então

$$f(x+p) = f(x) + \nabla f(x)^T p + \frac{1}{2} p^T \nabla^2 f(x+tp)^T p \tag{2.2}$$

para algum $t \in (0,1)$.

Condições necessárias para otimalidade são obtidas assumindo que X^* é um mínimo local e, em seguida, provando fatos sobre $\nabla f(X^*)$ e $\nabla^2 f(X^*)$.

Condições Necessárias de Primeira Ordem: Se X^* é um ponto de mínimo

local e f é continuamente diferenciável em uma vizinhança aberta de X^*, então $\nabla f(X^*) = 0$.

Condições Necessárias de Segunda Ordem: Se X^* é um ponto de mínimo local de f e $\nabla^2 f$ é contínua em uma vizinhança aberta de X^*, então $\nabla f(X^*) = 0$ e $\nabla^2 f(X^*)$ é semidefinida positiva.

Condições Suficientes de Segunda Ordem: Suponha que $\nabla^2 f$ é contínua em uma vizinhança aberta de X^* e que $\nabla f(X^*) = 0$ e $\nabla^2 f(X^*)$ é positiva definida. Então X^* é um ponto de mínimo local estrito de *f*.

A maioria dos algoritmos de minimização exigem um ponto de partida, que é geralmente denotado por x^0. Começando em x^0, os algoritmos geram uma sequência de iterações x^k que terminam quando algum critério de parada estabelecido é satisfeito. Por exemplo, pode-se assumir um critério de parada quando a diferença entre o valor da iteração x^k e x^{k-1} for menor que um determinado limite.

Ao decidir como passar de uma iteração x^k para a próxima, os algoritmos usam informações sobre o valor da função f em x^k e possivelmente informações de iterações anteriores $x^{k-1}, \ldots, x^1, x^0$. Estas informações são usadas para encontrar uma nova iteração x^{k+1} com um valor de f menor do que o valor de f em x^k.

Há duas estratégias fundamentais para passar do ponto x^k atual para uma nova iteração x^{k+1}. Na estratégia de *busca de linha*, o algoritmo escolhe uma direção p_k e busca ao longo desta direção, a partir do ponto x^k atual, um novo ponto com um valor de função menor. A distância que deve ser percorrida ao longo da direção p_k pode ser encontrada resolvendo-se o seguinte problema de minimização unidimensional para se encontrar uma medida α, definida como *comprimento de passo*:

$$\textit{minimizar}\; f(x_k + \alpha p_k), \alpha > 0 \tag{2.3}$$

Ao encontrar uma solução exata para este problema, o maior benefício a partir da direção p_k terá sido obtido. No entanto, a minimização exata é computacionalmente cara e desnecessária. Em vez disso, algoritmos de busca em linha geram um número limitado de comprimentos de passo até que se encontre uma solução que se aproxima da solução ótima do problema.

No novo ponto, ou seja, na solução exata ou aproximada do problema 2.3, uma

nova direção de pesquisa e um novo comprimento de passo são computados, e o processo é repetido. Resumidamente, a cada iteração de um método de busca em linha, uma direção de pesquisa p_k é calculada e então o algoritmo "decide"o quanto se mover ao longo dessa direção.

O sucesso de um método de busca em linha depende de escolhas efetivas tanto da direção de busca p_k quanto do comprimento de passo α_k. No cálculo do comprimento de passo α_k, existe uma questão de custo-benefício. O ideal é escolher α_k para se obter uma substancial redução no valor da função objetivo mas, ao mesmo tempo, não se deseja gastar muito tempo fazendo essa escolha.

Estratégias práticas para se implementar o método de busca em linha inexata (ou seja, quando a solução ótima do problema 2.3 não é encontrada) para se identificar um comprimento de passo que produza uma redução adequada no valor de f a um custo mínimo são normalmente empregadas. Uma condição de parada, conhecida como *Regra de Armijo* para uma busca em linha inexata usualmente utilizada estipula que α_k deve primeiramente fornecer uma redução suficiente na função objetivo f da seguinte forma:

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^T p_k,\ 0 < c_1 < 1 \tag{2.4}$$

Outra condição de parada para busca de linha inexata são as chamadas *Condições Wolfe*:

$$\begin{aligned} & f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^T p_k \\ & \nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f_k^T p_k \\ & 0 < c_1 < c_2 < 1 \end{aligned} \tag{2.5}$$

Na regra de Wolfe, além da imposição de uma redução no valor da função objetivo, também é imposta uma restrição ao novo valor do gradiente da função no ponto $x_k + \alpha_k p_k$.

Em outra estratégia algorítmica, conhecida como *Região de Confiança*, as informações adquiridas sobre a função objetivo f são usadas para construir um modelo m_k cujo comportamento próximo do ponto atual x_k é semelhante ao comportamento

real de f.

Como o modelo m_k pode não ser uma boa aproximação de f quando $x*$ está longe de x_k, a ideia é restringir a procura por um ponto de mínimo local de m_k para alguma região em torno de x_k. Em outras palavras, encontra-se o passo de minimização candidato p resolvendo-se o subproblema :

$$minimizar\ p\ em\ m_k(x_k + p) \tag{2.6}$$

onde $x_k + p$ está dentro da região de confiança.

Se a solução encontrada não produzir uma redução suficiente no valor da função f, pode-se concluir que a região de confiança é muito grande, reduzi-la e voltar a resolver o problema.

O modelo m_k é geralmente definido como uma função quadrática da forma

$$m_k(x_k + p) = f_k + p^T \nabla f_k + \frac{1}{2} p^T B_k p \tag{2.7}$$

Onde f_k, ∇f_k e B_k são um escalar, um vetor e uma matriz, respectivamente.

A matriz B_k é a Hessiana $\nabla^2 f_k$ ou alguma aproximação dessa matriz.

Em um certo sentido, as abordagens busca em linha e região de confiança diferem na ordem em que são escolhidas a direção e distância a ser percorrida para a próxima iteração. A busca em linha começa pela fixação de uma direção p_k para, em seguida, identificar uma distância adequada a ser percorrida, ou seja, o comprimento do passo α_k.

No método da região de confiança, primeiro é escolhido uma distância máxima (o raio da região de confiança k) e, em seguida, procura-se uma direção e um tamanho de passo que alcancem a maior de redução possível no valor de f com esta restrição de distância. Se essa etapa revelar-se insatisfatória, o raio da região de confiança k é reduzido e uma nova tentativa de se encontrar um ponto que diminua o valor da função objetivo é feita.

Uma importante direção de busca é chamada de *Direção de Newton*. Essa direção é derivada da aproximação da série de Taylor de segunda ordem para $f(x_k + p)$. Métodos que usam a direção de Newton tem uma taxa rápida de convergência local, tipicamente quadrática.

A principal desvantagem da direção de Newton é a necessidade do cálculo da Hessiana $\nabla^2 f(x)$. A computação explícita desta matriz de segundas derivadas é, na maioria das vezes, um processo caro e propenso a erros de arredondamento.

Como alternativa, existem as chamadas direções *Quasi-Newton* de busca, as quais fornecem uma opção atraente na medida em que não requerem computação explícita da matriz hessiana e ainda assim atingem uma taxa de convergência superlinear. No lugar da verdadeira matriz Hessiana $\nabla^2 f(x)$, é utilizada uma aproximação da mesma, usualmente definida como B_k, que é atualizada após cada iteração para se levar em conta o conhecimento adicional sobre a função objetivo e seu gradiente adquiridos durante a iteração atual.

As atualizações na matriz B_k fazem uso do fato de que as mudanças no gradiente fornecem informações sobre a segunda derivada da função objetivo ao longo da direção de busca. A nova aproximação para a Hessiana B_{k+1} é calculada para satisfazer a seguinte condição, conhecida como *equação da secante*:

$$\begin{aligned} & B_{k+1} s_k = y_k \ onde \\ & s_k = x_{k+1} - x_k \ e \\ & y_k = \nabla f_{k+1} - \nabla f_k \end{aligned} \tag{2.8}$$

Tipicamente, requisitos adicionais são impostos em B_{k+1}, tais como simetria (motivada pela simetria da matriz Hessiana exata) e uma restrição onde a diferença entre aproximações sucessivas B_k e B_{k+1} tenha um baixo posto. A aproximação inicial B_0 deve ser escolhida pelo desenvolvedor do modelo.

Uma das fórmulas mais populares para atualizar a aproximação da matriz hessiana B_k é a fórmula BFGS, nomeada a partir das iniciais de seus inventores, Broyden, Fletcher, Goldfarb e Shanno, que é definida por

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k} \tag{2.9}$$

A direção de busca quasi-Newton é dada por B_k substituindo a matriz hessiana exata na fórmula:

$$p_k = -B_k^{-1} \nabla f_k \tag{2.10}$$

Algumas implementações práticas de métodos Quasi-Newton evitam a necessidade de se fatorar B_k a cada iteração atualizando a inversa da matriz B_k em vez da matriz B_k em si.

Após a atualização da matriz b_k, a nova iteração é dada por:

$$x_{k+1} = x_k + \alpha_k p_k \tag{2.11}$$

onde o comprimento do passo α_k é escolhido para satisfazer as condições de Wolfe, por exemplo.

A inversa da matriz B_k, chamada de H_k, é atualizada através da seguinte fórmula:

$$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T \tag{2.12}$$

onde

$$\rho_k = \frac{1}{y_k^T s_k} \tag{2.13}$$

Uma versão simplificada do algoritmo BFGS pode ser definida como:

Algoritmo 1 Algoritmo simplificado do método BFGS

Dado um ponto inicial x_0, uma tolerância $\epsilon > 0$ e uma aproximação para a matriz hessiana H_0;

```
k ← 0
while ‖∇f_k‖ > ε do
   p_k ← −H_k ∇f_k {Calcular a direção de busca p_k}
   x_{k+1} ← x_k + α_k p_k {Onde α_k é calculado a partir de um procedimento de busca
   em linha que satisfaça as condições de Wolfe}
   s_k ← x_{k+1} − x_k
   y_k ← ∇f_{k+1} − ∇f_k
   Calcular H_{k+1} usando (2.12)
   k ← k + 1
end while
```

O método BFGS tem taxa de convergência superlinear.

Os métodos Quasi-Newton não são diretamente aplicáveis a grandes problemas de otimização porque as aproximações a matriz Hessiana ou a sua inversa são geralmente densas. Nestes casos, usam-se métodos Quasi-Newton com memória limitada.

Estes métodos são úteis para resolver problemas de grande porte cuja matriz Hessiana não pode ser computada a um custo razoável ou é demasiadamente densa para ser manipulada facilmente.

Nocedal em [12] propôs um método do tipo Quasi-Newton com memória limitada, o L-BFGS, que mantém aproximações simples de matrizes Hessianas, e ao invés de armazenar plenamente essa matriz de aproximação densa, são armazenados apenas poucos vetores que representam as aproximações de forma implícita.

Para a implementação do XCM, o algoritmo de minimização L-BFGS foi utilizado. Um dos principais pontos fracos do método L-BFGS é que ele muitas vezes converge lentamente, o que geralmente leva a um número relativamente grande de avaliações da função objetivo. Além disso, o método é altamente ineficiente em problemas mal condicionados, especificamente nos casos onde a matriz Hessiana contém uma ampla distribuição de autovalores.

Existem implementações livres do algoritmo BFGS, como por exemplo nas bibliotecas ALGLIB, GSL - GNU Scientific Library e L-BFGS porém a biblioteca L-BFGS apresentou melhores resultados computacionais, sendo a biblioteca escolhida para a implementação final do XCM que serve como base para o desenvolvimento do XCM-OpenMP.

2.4 API de programação paralela

2.4.1 OpenMP

O OpenMP (*Open Multi-Processing*) [5] é uma API para programação paralela em sistemas com memória compartilhada. Ele consiste em um conjunto de diretivas, bibliotecas e variáveis de ambientes para descrever tarefas paralelas. A intenção principal é prover ao programador uma interface simples e flexível para paralelizar aplicações. Detalhes do OpenMP seram abordados na seção 2.6

2.4.2 MPI

O MPI (*Message Passing Interface*) [7] é também uma API que permite a comunicação entre processos através do envio e recebimento de mensagens. O seu uso

é mais adequado em arquiteturas distribuídas, como clusters de computadores. É também um modelo bastante flexível. Porém muitos detalhes são expostos ao programador. No MPI, o programador é responsável por definir a estratégia de paralelização e a hierarquia dos processos para computar paralelamente uma tarefa.

2.4.3 TBB

Threading Building Blocks (TBB) é uma biblioteca para C++ desenvolvida pela Intel para o desenvolvimento de softwares que utilizem a tecnologia de processadores *multicore* [14]. A versao 1.0 foi lançada em 2006 para os primeiros processadores *dualcore* x86, Pentium D. Sua versao comercial suporta os sistemas operacionais Windows(XP ou superior), Mac OS X (versao 10.4.4 ou superior) e Linux. Apresenta suporte para decomposição de dados e controle de granularidade automática, além de balancear a carga ao utilizar um escalonador de tarefas, integrado com a técnica de roubo de tarefas. O TBB permite que o programador especifique tarefas paralelas, o que leva a uma programação de mais alto nível do que escrever diretamente código para gerenciar *threads*. Uma outra funcionalidade do TBB é o uso de *templates* para instanciar mecanismos como *pipelines*. Os *templates*, no entanto, possuem limitações. Por exemplo, apenas *pipelines* lineares podem ser descritos usando *templates*.

2.4.4 Cilk

Cilk é uma linguagem de programação de propósito geral desenvolvida no *Massachusetts Institute of Technology* (MIT) [9]. A interface baseia-se na linguagem C ANSI, possuindo versoes para as plataformas GNU/Linux, MacOS X e Windows. O sistema de execução de Cilk é encarregado de fazer o balanceamento de carga e de escalonar as tarefas criadas para executar em paralelo entre os processadores, garantindo melhor desempenho e eficiência. O escalonamento das tarefas em Cilk pode ser feito tanto pelo compartilhamento de tarefas como pelo roubo de tarefas. No primeiro caso, uma *thread* é escalada para executar em paralelo a cada chamada de função. Isso maximiza o processamento em paralelo, mas é penalizado pelo custo de criação de um nova *thread*. No segundo caso, quando uma *thread* termina sua tarefa ela busca mais trabalho. A vantagem desta abordagem é minimizar o tempo

de criação das *thread* e priorizar a eficiência. Funciona de maneira adequada em ambientes com processadores heterogêneos, mas não foi projetado para ambientes que utilizem memoria distribuída.

2.4.5 Trebuchet

Desenvolvida por alunos da Coppe-UFRJ, a Trebuchet [1] é uma máquina virtual *multithread* baseada em arquitetura *DataFlow*. A aplicação é compilada para o modelo *DataFlow* e suas instruções independentes (segundo o fluxo de dados) são executadas em Elementos de Processamento (EPs) distintos da Trebuchet. Cada EP é mapeado em uma *thread* na máquina hospedeira. O modelo permite a definição de blocos de instruções de diferentes granularidades, que terão disparo guiado pelo fluxo de dados e execução direta na máquina hospedeira, para diminuir os custos de interpretação. Como a sincronização é obtida pelo modelo *DataFlow*, não é necessária a introdução de *locks* ou barreiras nos programas a serem paralelizados.

2.5 Tipos de Arquiteturas Paralelas

Com as limitações impostas pelos materiais utilizados na construção de processadores, viu que não se podeira aumentar indefinidamente a frequência dos processadores para ganhar desempenho [8]. Infelizmente, todo o desenvolvimento de tecnologia dos componentes do computador não se deu de forma uniforme, por exemplo, a diminuição do tempo de acesso à memória não acompanhou o aumento da velocidade de CPU, crescimentos da densidade de armazenamento (memória e disco rígido), entre outros. Como resultado, o balanço de desempenho entre diferentes partes do computador mudou. Esses aspectos forçaram novas arquiteturas de computadores a migrarem para esquemas de hierarquia de memória cada vez mais complexos. Partiu-se para outras abordagens, como a paralelização.

Existem vários modelos de classificação de arquiteturas, porém o modelo proposto em 1966 por Flynn [6] , é ainda atual e apresenta as vantagens de ser simples, fácil de entender e fornecer uma boa aproximação da realidade. A taxonomia de Flynn considera o fluxo de instruções executadas em paralelo e o fluxo de dados tratados em paralelo. A classificação é a seguinte:

- SISD (*Single Instruction stream / Single Data stream*) - Categoria onde se enquadram monoprocessadores. Computadores SISD não são capazes de explorar qualquer tipo de paralelismo, uma vez que uma instrução pode ser processada por ciclo.

- SIMD *(Single Instruction stream / Multiple Data stream)* - Uma mesma instrução é executada por múltiplas unidades de processamento em diferentes dados de uma estrutura. Computadores SIMD exploram paralelismo em nível de dados ao aplicar uma mesma operação sobre múltiplos operandos. Por exemplo um *array* de números inteiros (múltiplos dados), realizar a operação que incrementa o valor contido em cada posição do *array* em 1, de forma simultânea.

- MISD *(Multiple Instruction stream / Single Data stream)* - Múltiplos fluxos de instruções atuando sobre um fluxo de dados. Geralmente nenhuma arquitetura é classificada neste nível. Alguns autores consideram a arquiterura *pipeline* como um exemplo deste tipo de organização.

- MIMD *(Multiple Instruction stream / Multiple Data stream)* - Cada unidade de processamento busca sua própria instrução e opera sobre seus próprios dados. Computadores MMID exploram paralelismo em nível de *threads* e de processos, possibilitando que múltiplas unidades de execução sejam processadas paralelamente.

Pelo fato deste trabalho estar relacionado com o paralelismo em nível de *threads*, nos interessa explorar melhor a categoria MMID. Processadores MMID estão divididos em dois grupos, que são determinados de acordo com a organização da memória de trabalho e a maneira como os diferentes elementos de processamento estão interconectados. Estes grupos são classificados como Arquitetura com Memória Compartilhada e Arquiteturas com Memória Distribuída.

Arquiteturas de memória compartilhada (ou multiprocessadores) caracterizam-se pela existência de uma memória global e única, que é utilizada por todos os processadores (sistema fortemente acoplado), de maneira que através do compartilhamento de posições desta memória ocorre a comunicação entre processos. Para evitar que a memória se torne um gargalo, arquiteturas de memória compartilhada

devem implementar mecanismos de cache, além de garantir a coerência dos dados (através de mecanismos de hardware e software).

Em **arquiteturas de memória distribuída**, cada processador possui sua própria memória local, caracterizando assim um sistema fracamente acoplado. Em virtude de não haver compartilhamento da memória, os processos comunicam-se via troca de mensagens. Nesse tipo de comunicação ocorre a transferência explícita de dados entre os processadores. Este tipo de organização é também conhecida como multicomputador.

2.6 A Arquitetura OpenMP

Com a proliferação das arquiteturas *multicore* e SMP (*Symmetric Multiprocessing*), que possibilitam diversos fluxos de execução simultâneos, faz-se cada vez mais necessário o uso de técnicas de programação que explorem eficientemente o potencial destas arquiteturas, onde os processadores ou núcleos compartilham a mesma memória global, e se comunicam através de escritas e leituras.

Dentre as alternativas para desenvolvimento de aplicações *multithread* , destaca-se o padrão OpenMP (*Open Multi-Processing*) que é uma API para programação paralela em sistemas com memória compartilhada [5]. Ele consiste em um conjunto de diretivas, bibliotecas e variáveis de ambientes para descrever tarefas paralelas. A intenção principal é prover ao programador uma interface simples e flexível para paralelizar aplicações.

2.6.1 Histórico

O padrão OpenMP é desenvolvido e mantido pelo grupo OpenMP Architecture Review Board (ARB), formado pelos maiores fabricantes de software e hardware do mundo, tais como SUN Microsystems, SGI, IBM, Intel, dentre outros, que, no final de 1997, reuniram esforços para criar um padrão de programação paralela para arquiteturas de memória compartilhada. A primeira versão do padrão, OpenMP para FORTRAN 1.0, sendo a versão para C/C++ em 1998. No ano de 2000 surgiu a versão 2.0 para FORTRAN, e em 2002 esta para C/C++. Em maio de 2005 é lançada a versão 2.5, que combina suporte a FORTRAN e C/C++. A versão 3.0 Foi

lançada em maio de 2008. Atualmente a versão é a 3.1 lançada em julho de 2011. A versão 4.0 não possui data de lançamento, mas está definida e sendo implementada pelo comitê de desenvolvimento do OpenMP [2].

2.6.2 Modelo OpenMP

Um programa utilizando a ferramenta OpenMP sofre poucas alterações quando comparado com um programa sequencial. A programação é baseada em diretivas de compilação, que são inseridas no programa sequencial indicando para o compilador as partes que podem ser paralelizadas. Esse compilador tem que possuir suporte a OpenMP, e é ele o responsável pela criação das threads. O sucesso do OpenMP é propagado pelas arquiteturas *multicore* e *multithread* e pode ser atribuído a fatores como a robusta estrutura de programação paralela suportada e a facilidade do seu uso.

O modelo de programação inicia com uma única thread que executa sozinha as instruções até encontrar uma região paralela (identificada pela diretiva), ao encontrar essa região ela cria um grupo de threads (este número é definido pela variável de ambiente `OMP_NUM_THREADS`, ou explicitamente pela diretiva `parallel`) , que juntas executam o código dentro dessa região. Para cada thread, é delegada a execução das intruções contidas na região paralela. Quando as threads completam a execução do código, elas sincronizam-se e somente a thread inicial segue na execução do código até que uma nova região paralela seja encontrada ou que o programador decida encerrar essa thread. Esse modelo de programação é conhecido na literatura como *fork-join* .

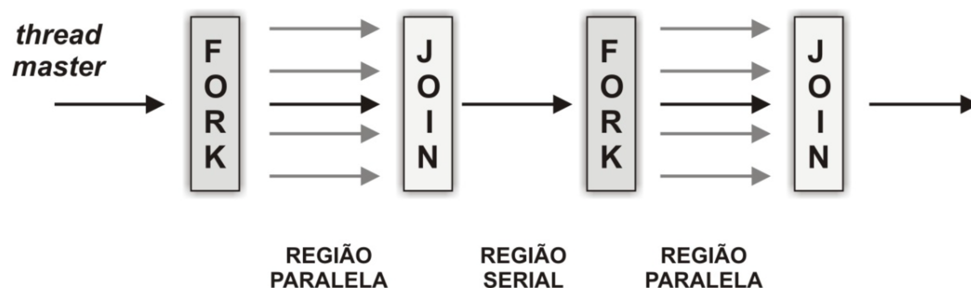

Figura 2.2: Modelo de programação OpenMP

2.6.3 Sintaxe OpenMP

Como citado anteriormente, o OpenMP é constituído de diretivas de compilação, variáveis de ambiente e bibliotecas de serviço.

Diretivas

As diretivas do OpenMP são baseadas na diretiva # `pragma` definida no padrão da linguagem C/C++. Os compiladores que suportam OpenMP em C/C++ possuem uma opção de linha de comando que ativa e permite a interpretação das diretivas do OpenMP.

O formato padrão de uma diretiva OpenMP é mostrado a seguir:

#pragma omp [nome da diretiva] [cláusula, ...] nova linha

Dentre as diretivas existentes podemos citar as sequintes:

- #pragma omp parallel: permite a paralelização do bloco de código que se encontra após essa diretiva. No momento em que esta diretiva é invocada, o bloco é executado paralelamente com as *threads* criadas pela thread principal, incluindo ela mesma.

- #pragma omp for: o laço que se encontra após essa diretiva é executado de forma paralela entre várias threads. Nessa diretiva existe uma sincronização implícita, podendo ser assíncrono através do comando *nowait*.

Cláusulas

As cláusulas definem o comportamento dos construtores aos quais estão associadas e das variáveis envolvidas na execução da região paralela. Além de outras atribuições, as cláusulas definem quais variáveis são compartilhadas entre as threads e quais são privadas. Por padrão, as variáveis são consideradas compartilhadas, exceto as que estão especificadas em uma diretiva `threadprivate` e as que são declaradas dentro da região paralela. A maioria das cláusulas é aplicada às variáveis presentes na lista de variáveis que acompanha a mesma. Uma variável pode estar presente em apenas uma cláusula por diretiva, exceto as cláusulas `firstprivate` e `lastprivate` que podem conter a mesma variável nas suas listas. Nem todas as

cláusulas se aplicam a todas as diretivas. A seguir serão definidas algumas cláusulas do OpenMP, juntamente com uma listagem das diretivas às quais cada cláusula pode estar associada.

- shared: os dados dentro de uma região paralela são compartilhados, o que significa visível e acessível por todas as threads em simultâneo. Por padrão, todas as variáveis da região de compartilhamento são partilhados, exceto o laço (*loop*).
- private: os dados em uma região paralela são individuais para cada thread, com este recurso cada thread terá uma cópia local e será utilizada como uma variável temporária. Uma variável não é inicializada e o valor não é mantido para o uso fora da região paralela.
- default: permite ao programador declarar a extensão do padrão de dados que será compartilhado na região paralela, em C/C++, ou shared, firstprivate.
- firstprivate: como private exceto ao inicializar pelo valor original.
- lasprivate: como private exceto que o valor original é atualizado depois da construção.
- if: Isto fará com que as threads paralelizem a tarefa se a condição for satisfeita. Caso contrário, o bloco de código é executado em série.
- reduction : a variável tem uma copia local em cada thread, mas os valores das copias locais são reduzidos em uma variável global compartilhada.

Funções de ambiente de execução

As bibliotecas do OpenMP incluem um conjunto de rotinas em tempo de execução a nível de usuário: usadas para modificar/checar o número de threads. Detecta se o contexto de execução está em uma região paralela, quantos processadores tem no sistema corrente, setar/desetar locks, funções de timing,etc.

- omp_set_num_threads() : insere um número padrão de threads a serem usadas nas regiões paralelas em que não está definida a cláusula *num_threads*.

- omp_get_num_threads() : retorna o número de threads ativas na região paralela onde a função foi chamada.
- omp_get_num_procs() : retorna o número de processos disponíveis para o programa no momento da chamada da função.

Variáveis de Ambiente

O OpenMP possui algumas variáveis de ambiente cuja função é controlar as variáveis de controle interno que influenciam na execução de aplicações paralelas. Se as variáveis de ambiente forem modificadas ao longo da execução do programa, essas mudanças são ignoradas pelo OpenMP. Por outro lado, as variáveis de controle interno podem ter seus valores modificados durante a execução do programa por meio da utilização de uma cláusula ou da chamada de uma função do OpenMP. As variáveis de ambiente são:

- OMP_SCHEDULE : modifica o comportamento da cláusula `schedule` quando é especificado em uma diretiva `for` ou `parallel for`.
- OMP_NUM_THREADS : é usado para especificar o número de threads que irá executar as instruções da região paralela definida pelo construtor ao qual a cláusula está associada. Pode ser utilizada apenas no construtor paralelo.
- OMP_DYNAMIC : habilita ou desabilita o ajuste dinâmico do número de threads disponíveis para execução da região paralela. Entretanto, o esse ajuste pode ser habilitado ou desabilitado de forma explicita pela chamada da função `omp_set_dynamic()` durante a execução do programa.
- OMP_NESTED : habilita ou desabilita o paralelismo aninhado, a menos que o mesmo seja habilitado ou desabilitado pela chamada da função `omp_set_nested()`. Se a variável de ambiente for definida como `TRUE`, o paralelismo aninhado será habilitado. Se for definida como `FALSE`, será desabilitado. Vale ressaltar que o valor padrão dessa variável é `FALSE`

2.6.4 Exemplo de uso

A seguir foi exemplificado um caso de uso da arquitetura OpenMP. O objetivo é mostrar um pouco da sua sintaxe, diretivas e cláusulas utilizadas.

Exemplo Algoritmo Fibonacci

A sequência Fibonacci é um exemplo clássico utilizados nas aulas de algoritmos e programação para ensinar a recursividade, devido à simplicidade do código necessário para expressar esta equação recorrente, no código abaixo está uma implementação em C de um algoritmo resursivo que calcula o n-ésimo número da sequência, utilizando paralelismo de tarefas OpenMP.

```
int main( int argc, char *argv[] ){
  const int n = atoi(argv[1]);
  #pragma omp parallel{
    #pragma omp single nowait
      fib(n);
  }
}
int fib(int n){
  int x,y;
  if( n < 2 ) return n;
  #pragma omp task untied shared(x)
    x = fib( n-1 );
  #pragma omp task untied shared(y)
    y = fib( n-2 );
  #pragma omp taskwait
  return x + y;
}
```

Capítulo 3

O Método XCM

O método de agrupamento via suavização hiperbólica tradicional, apresentado inicialmente em [16] e definido aqui como XCM, *Xavier Clustering Method*, utiliza a formulação do problema de agrupamento baseado no critério de mínima soma de quadrados como ponto de partida. Este critério corresponde à minimização da soma de quadrados das distâncias das observações para o ponto definido como centro do conjunto, ou *cluster*.

O principal problema associado a essa formulação é a criação de um problema de otimização global não-diferenciável e não-convexo. A ideia principal do XCM é *suavizar* o problema criado a partir dessa formulação. A técnica de suavização aplicada foi desenvolvida a partir de uma adaptação do método de penalização hiperbólica, originalmente apresentado em [17].

Para uma análise da metodologia de resolução, é necessário definir o problema de agrupamento como um problema *min-sum-min*, para em seguida transformar o problema e aplicar a técnica de suavização hiperbólica. As etapas de formulação do problema, até a definição do problema final utilizado no XCM-OpenMP são definidas nas seções a seguir. Como essa formulação serve de base para todo o XCM-OpenMP, ela é apresentada aqui apenas em razão da completude do trabalho. Todo esse desenvolvimento já foi apresentado em [16] e [18], sendo o crédito devidamente destinado aos autores.

3.1 Análise de Agrupamento como um problema min-sum-min

Seja $S = s_1, s_2, ..., s_m$ um conjunto de m pontos, ou observações, em um espaço euclidiano de dimensão n que devem ser agrupados em q conjuntos. O número q é pré-definido, e um dos parâmetros de entrada do algoritmo.

Seja $x_i, i = 1, ..., q$ um conjunto de pontos centrais dos *clusters*, onde cada x_i é chamado de *centroid* e cada $x_i \in \mathbb{R}^n$.

O conjunto dos q centroids é representado por $X \in \mathbb{R}^n q$. Para cada observação s_j, é calculada a distância dela aos q centroids, e a menor distância é definida como

$$z_j = \min_{x_i \in X} \parallel s_j - x_i \parallel_2 \tag{3.1}$$

Considerando uma posição específica dos q centroids, pode-se definir a mínima soma de quadrados das distâncias como:

$$D(X) = \sum_{j=1}^{m} z_j^2 \tag{3.2}$$

Dessa forma, o posicionamento ótimo dos centros dos centroids deve fornecer o melhor resultado para essa medida. Assim, se X^* é definido como o posicionamento ótimo, então o problema pode ser definido como:

$$X^* = \operatorname*{argmin}_{X \in R^{nq}} D(X) \tag{3.3}$$

onde X é o conjunto de pontos dos centros dos q centroids.

Assim, utilizando as equações 3.1, 3.2 e 3.3 acima, chega-se na seguinte definição:

$$X^* = \operatorname*{argmin}_{X \in R^{nq}} \sum_{j=1}^{m} \min_{x_i \in X} \parallel s_j - x_i \parallel_2^2 \tag{3.4}$$

Esse problema pode ser transformado da seguinte forma:

$$
\begin{aligned}
& minimizar \sum_{j=1}^{m} z_j^2 \\
& sujeito\, a: \; z_j = \min_{i=1..q} \parallel s_j - x_i \parallel_2, \; j = 1, \ldots, m
\end{aligned}
\tag{3.5}
$$

Considerando a sua definição em 3.1, cada z_j deve necessariamente satisfazer o conjunto de desigualdades de 3.6,

$$
z_j - \parallel s_j - x_i \parallel_2 \leq 0, i = 1, \ldots, q \tag{3.6}
$$

dado que z_j já é a menor distância entre a observação j e um centroid qualquer.

Substituindo-se as igualdades do problema 3.5 pelas desigualdades de 3.6, obtem-se o problema relaxado

$$
\begin{aligned}
& minimizar \sum_{j=1}^{m} z_j^2 \\
& sujeito\, a: \; z_j - \parallel s_j - x_i \parallel_2 \leq 0, \; j = 1, \ldots, m, \; i = 1, \ldots, q
\end{aligned}
\tag{3.7}
$$

Como as variáveis z_j não são limitadas inferiormente, a solução ótima do problema 3.7 é $z_j = 0, j = 1, \ldots, m$. Assim, o problema 3.7 não é equivalente ao problema 3.5. É necessário, portanto, modificar o problema 3.7 para se obter a equivalência desejada. Para isso, é necessário definir a função φ da seguinte forma:

$$
\varphi(y) = \max\{0, y\} \tag{3.8}
$$

Pode-se observar que, se as desigualdades em 3.6 são válidas, também são igualmente válidas as restrições de 3.9:

$$
\sum_{i=1}^{q} \varphi(z_j - \parallel s_j - x_i \parallel_2) = 0, \; j = 1, \ldots, m \tag{3.9}
$$

Na figura 3.1, extraída de [16], pode-se observar o gráfico das três primeiras parcelas do somatório da equação 3.9 como função de z_j, onde $d_i = \parallel s_j - x_i \parallel_2$ e considerando-se as distâncias d_i ordenadas em ordem crescente segundo os índices.

Com a substituição das desigualdades do problema 3.7 pela equação 3.9, se-

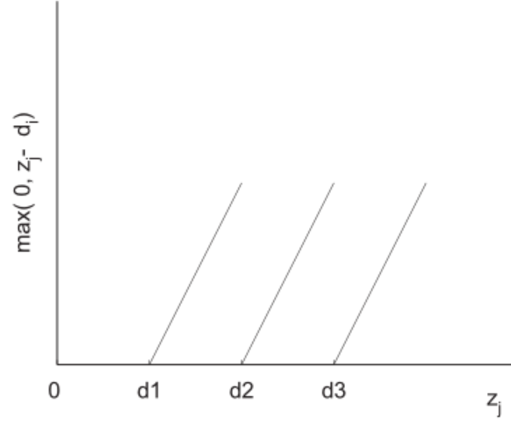

Figura 3.1: Três primeiras parcelas do somatório da equação 3.9

ria obtido um problema equivalente porém com a mesma propriedade indesejável que as variáveis $z_j = 0, j = 1, \ldots, m$ ainda não são limitadas inferiormente. No entanto, como a função objetivo do problema 3.7 forçará os valores das variáveis $z_j = 0, j = 1, \ldots, m$ a receber os menores valores possíveis, pode-se pensar em limitar inferiormente essas variáveis ao se considerar $>$ ao invés de $=$ na equação 3.9. Dessa forma, chega-se ao problema não-canônico 3.10.

$$\begin{aligned} &minimizar \sum_{j=1}^{m} z_j^2 \\ &sujeito\, a: \sum_{i=1}^{q} \varphi(z_j - \| s_j - x_i \|_2) > 0,\ j = 1, \ldots, m \end{aligned} \tag{3.10}$$

Para recuperar a formulação canônica, as desigualdades no problema 3.10 são perturbadas e, dessa forma, obtem-se o seguinte problema modificado 3.11:

$$\begin{aligned} &minimizar \sum_{j=1}^{m} z_j^2 \\ &sujeito\, a: \sum_{i=1}^{q} \varphi(z_j - \| s_j - x_i \|_2) \geq \varepsilon,\ j = 1, \ldots, m \end{aligned} \tag{3.11}$$

para $\varepsilon > 0$. Como o conjunto viável de soluções do problema 3.10 é o limite do conjunto viável de soluções do problema 3.11 quando $\varepsilon \to 0_+$ pode-se, então, resolver o problema 3.10 através da resolução de uma sequência de problemas iguais

a 3.11 para uma sequência de valores decrescentes de ε que se aproximam de 0 pela direita.

A prova que o valor da solução ótima do problema 3.4 está arbitrariamente próximo do valor da solução do problema 3.10 está disponível em [16].

Analisando o problema 3.11, é possível verificar que a definição da função ε impõe a ele uma estrutura não diferenciável de difícil solução. Nesse ponto é aplicado o conceito de suavização, de forma a transformar o problema 3.11 para que sua solução numérica seja simplificada, sem distanciá-lo do problema original.

Assim, utilizando-se do conceito de suavização, deve-se definir uma função ϕ da seguinte forma:

$$\phi(y, \tau) = \frac{(y + \sqrt{y^2 + \tau^2})}{2} \tag{3.12}$$

para $y \in \mathbb{R}$ e $\tau > 0$.

Pode-se notar que a função ϕ possui as seguintes características:

(a) $\phi(y, \tau) > \varphi(y),\ \forall \tau > 0$;

(b) $\lim_{\tau \to 0} \phi(y, \tau) = \varphi(y)$;

(c) $\phi(y, \tau)$ é uma função convexa crescente que pertence à classe de funções C^∞ na variável y.

Assim, a função ϕ consiste em uma aproximação da função φ definida na equação 3.8. Utilizando-se das mesmas convenções definidas na apresentação da figura 3.1, as três primeiras parcelas componentes da equação 3.9 e a correspondente suavização aproximada, obtida a partir da equação 3.12 são mostradas na figura 3.2.

Ao substituir a função ϕ pela função φ no problema 3.11, obtem-se o seguinte problema:

$$\begin{aligned} &minimizar \sum_{j=1}^{m} z_j^2 \\ &sujeito\, a: \sum_{i=1}^{q} \phi(z_j - \| s_j - x_i \|_2, \tau) \geq \varepsilon,\ j = 1, \ldots, m \end{aligned} \tag{3.13}$$

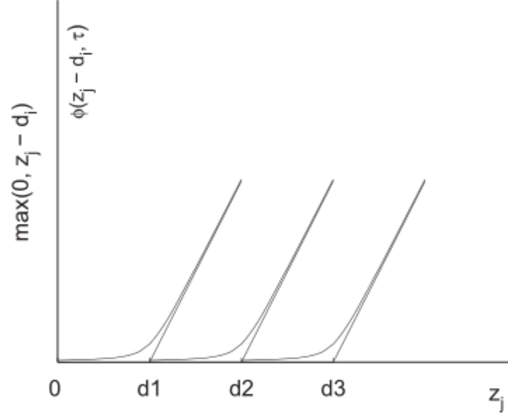

Figura 3.2: Três primeiras parcelas do somatório da equação 3.9 com suas equivalentes suavizações

Para se obter um problema completamente diferenciável, ainda é necessária a suavização da distância euclidiana $\| s_j - x_i \|_2$ do problema 3.13. Dessa forma, define-se a função θ como abaixo:

$$\theta(s_j, x_i, \gamma) = \sqrt{\sum_{l=1}^{n} (s_j^l - x_i^l)^2 + \gamma^2} \tag{3.14}$$

para $\gamma > 0$.

A função θ possui as seguintes propriedades:

(a) $\lim_{\gamma \to 0} \theta(s_j, x_i, \gamma) = \| s_j - x_i \|_2$;

(b) θ é uma função que pertence à classe de funções C^∞.

Ao substituir a distância euclidiana $\| s_j - x_i \|_2$ do problema 3.13 pela função θ, obtém-se o problema diferenciável 3.15, definido da seguinte forma:

$$\begin{aligned} & minimizar \sum_{j=1}^{m} z_j^2 \\ & sujeito\, a: \sum_{i=1}^{q} \phi(z_j - \theta(s_j, x_i, \gamma), \tau) \geq \varepsilon,\; j = 1, \ldots, m \end{aligned} \tag{3.15}$$

As propriedades das funções ϕ e θ permitem buscar uma solução para o problema 3.10 através da resolução de uma sequência de problemas como definidos em 3.15 para valores cada vez menores de ε, τ, e γ, até que uma regra de parada seja

atingida. Os valores de ε, τ, e γ devem tender a zero sempre pela direita, e nunca devem chegar a ser iguais a zero.

Como $z_j \geq 0, j = 1 \ldots, m$ a minimização da função objetivo irá tentar reduzir ao máximo esses valores. Por outro lado, dado qualquer conjunto de centroids $x_i, i = 1 \ldots, q$ e observando-se a propriedade (c) da função de suavização hiperbólica ϕ, as restrições do problema 3.15 são funções monotonicamente crescentes em z_j. Então, essas restrições sempre estarão ativas e o problema 3.15 será equivalente ao problema 3.16 abaixo:

$$\begin{aligned} &minimizar\, f(x) = \sum_{j=1}^{m} z_j^2 \\ &sujeito\, a:\; h_j(z_j, x) = \sum_{i=1}^{q} \phi(z_j - \theta(s_j, x_i, \gamma), \tau) - \varepsilon = 0,\; j = 1, \ldots, m \end{aligned} \tag{3.16}$$

A dimensão do problema 3.16 é $(nq + m)$. Como, em geral, o número de observações m é grande, o problema 3.16 possui um número muito grande de variáveis. No entanto, uma característica interessante desse problema é que ele possui uma estrutura separável, porque cada variável z_j aparece em apenas uma restrição de igualdade. Assim, como a derivada parcial de $h(z_j, x)$ em relação a $z_j, j = 1 \ldots, m$ não é igual a zero, é possível usar o teorema da função implícita para calcular cada componente $z_j, j = 1 \ldots, m$ como função das variáveis dos centroids $x_i, i = 1 \ldots, q$. Dessa forma, o problema irrestrito 3.17 é obtido:

$$minimizar\, f(x) = \sum_{j=1}^{m} z_j(x)^2 \tag{3.17}$$

Cada z_j do problema 3.17 resulta do cálculo da raiz da equação 3.18 abaixo:

$$h_j(z_j, x) = \sum_{i=1}^{q} \phi(z_j - \theta(s_j, x_i, \gamma), \tau) - \varepsilon = 0,\; j = 1, \ldots, m \tag{3.18}$$

Novamente, devido a propriedade (c) da função de suavização hiperbólica, cada termo ϕ de 3.18 é estritamente crescente em relação a variável z_j e, dessa forma,

pode-se concluir que as equações de 3.18 possuem uma única raiz. Além disso, considerando-se novamente o teorema da função implícita, as funções $z_j(x)$ têm todas as derivadas em relação as variáveis $x_i, i = 1 \ldots, q$. Logo, é possível calcular o gradiente da função objetivo do problema 3.17 como:

$$\nabla f(x) = \sum_{j=1}^{m} 2z_j(x)\nabla z_j(x) \tag{3.19}$$

Onde

$$\nabla z_j = -\nabla h_j(z_j, x) / \frac{\partial h_j(z_j, x)}{\partial z_j} \tag{3.20}$$

e $\nabla h_j(z_j, x)$ é obtido das equações 3.12 e 3.14. $\partial h_j(z_j, x)/\partial z_j$ vem da equação 3.18.

Dessa forma, é fácil resolver o problema 3.17 usando métodos de minimização baseados na primeira derivada, como apresentado no capítulo 2. Além desse fato, vale ressaltar que o problema 3.17 está definido no espaço de dimensão (nq). Como o número de centroids, q, é relativamente pequeno em relação ao número de observações e, em geral, é um número pequeno, o problema 3.17 pode ser considerado um problema de dimensão pequena, principalmente quando comparado ao problema 3.16.

Na figura 3.3 pode-se observar o comportamento da aproximação gradual da função suavizada em relação a função original.

O algoritmo do XCM pode ser resumido da seguinte forma:

Assim como em outros métodos iterativos, a solução para o problema de agrupamento é obtida, em teoria, através da resolução de uma sequencia infinita de problemas de otimização. No método XCM, cada problema a ser resolvido é irrestrito e de baixa dimensão. Outro ponto que merece destaque é que o algoritmo faz com que os valores de τ e γ se aproximem de zero. Então, as restrições dos sub-problemas resolvidos tendem àquelas do problema 3.11 e, de forma simultânea, a redução do valor de ε (também tendendo a zero) faz com que o problema 3.11 se aproxime do problema original 3.10.

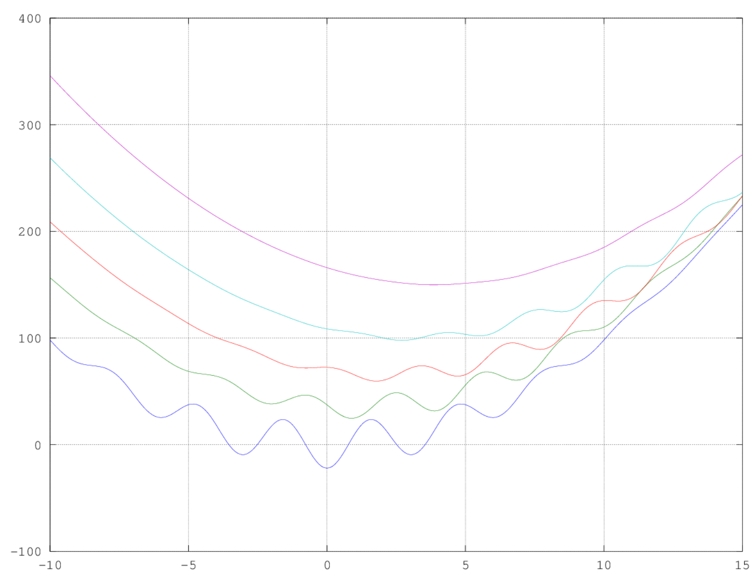

Figura 3.3: Comportamento da função suavizada

Algoritmo 2 Algoritmo XCM simplificado

Inicialização: Defina os valores iniciais de $x^0, \varepsilon^1, \tau^1, \gamma^1$ e os valores $0 < \rho_1 < 1, 0 < \rho_2 < 1, 0 < \rho_3 < 1$;

$k \leftarrow 1$
while Uma regra de parada não for atingida **do**
Resolva o problema 3.17 com $\varepsilon = \varepsilon^k, \tau = \tau^k, \gamma = \gamma^k$ começando no ponto inicial x^{k-1} e seja x^k a solução obtida.
$\varepsilon^{k+1} \leftarrow \rho_1 \varepsilon^k$
$\tau^{k+1} \leftarrow \rho_2 \tau^k$
$\gamma^{k+1} \leftarrow \rho_3 \gamma^k$
$k \leftarrow k + 1$
end while

Diversas regras de parada podem ser empregadas, como, por exemplo, observando-se se os valores da função objetivo não sofreram alteração significativa após um determinado número de iterações. A mesma regra pode ser aplicada aos valores das coordenadas dos pontos dos centroids.

Devido às propriedades de continuidade de todas as funções envolvidas, a sequência $x^1, x^2, \ldots, x^k$ de valores ótimos dos problemas suavizados tendem ao valor ótimo de 3.1. Os resultados computacionais mostraram que, após poucas iterações, o valor mínimo da função objetivo é atingido e uma solução para o posicionamento dos centroids é encontrada.

Capítulo 4

O Método XCM-OpenMP

Para explorar o paralelismo em memória compartilhada escolheu-se trabalhar com programação *multithread*. Existem algumas formas de adicionar *threads* ao programa para ajudar na execução de alguma tarefa, uma delas é através da tecnologia OpenMP [2].

Esta foi a tecnologia escolhida para este trabalho, por ser um padrão flexível e voltado para o alto desempenho, além de permitir o desenvolvimento do código paralelo de forma incremental, favorecendo a implementação das partes do código do XCM que são altamente paralelizáveis. Outro motivo da escolha da tecnologia OpenMP é a facilitação da implantação (*deployment*) do programa que se vai produzir. Com essa tecnologia é suficiente construir (*build*) o programa utilizando as diretivas OpenMP. O programa binário produzido contém dentro de si todo o necessário para rodar executando paralelismo.

4.1 Conceitos Básicos

A análise do método XCM foi a base para as ideias de implementação que utilizam arquiteturas paralelas. Com a transformação do problema 3.16 no problema 3.17, torna-se possível uma grande redução no número de variáveis, permitindo assim, a formulação do mesmo como uma função que é definida por um somatório de parcelas independentes. Esta ideia pode ser observada na equação 4.1 abaixo, onde cada cálculo de $z_j(x)^2$ depende apenas de x, ou seja, da solução da iteração atual do método.

$$minimizar\, f(x) = \sum_{j=1}^{m} z_j(x)^2 = z_1(x)^2 + z_2(x)^2 + z_3(x)^2 + \ldots + z_m(x)^2 \quad (4.1)$$

Pode-se explicitar o gradiente da função $f(x)$ como uma soma de parcelas independentes da mesma forma que a equação 4.1

$$\nabla f(x) = \sum_{j=1}^{m} \nabla z_j(x)^2 = 2z_1(x)\nabla z_1(x) + 2z_2(x)\nabla z_2(x) + \ldots + 2z_m(x)\nabla z_m(x) \quad (4.2)$$

A equação 3.18 representa o cálculo de cada $z_j(x)$. O somatório de $f(x)$ possui m parcelas e, em geral, para aplicações reais, o número m (a quantidade total de observações) é grande. Na resolução de 3.18 necessita-se encontrar a raiz da equação e dessa forma são calculadas m raízes. No entanto, não é necessário encontrar a raiz de z_1 para que o cálculo da raiz de z_2 seja feito, e assim por diante.

Esse é o principal ponto explorado pelo XCM-OpenMP. A base do método é resolver as diversas parcelas referentes ao cálculo de $f(x)$ nas diversas unidades de processamento, utilizando a tecnologia multicore, através da arquitetura OpenMP (definida no capítulo 2).

O mesmo conceito aplica-se ao gradiente da função f. A equação 3.19 define o gradiente da função f também como uma soma de parcelas independentes e, apesar da maior complexidade do cálculo, o mesmo conceito de resolução paralela se aplica.

Excetuando-se os passos de inicialização (discutidos na seção 4.1) necessários à programação em OpenMP, a implementação sequencial e a implementação paralela possuem as mesmas características. As principais diferenças residem na forma do cálculo da função objetivo e seu gradiente onde, na versão paralela, ganhos de performance consideráveis são obtidos.

Uma vez calculado o valor de $f(x)$ e de $\nabla f(x)$, esses valores são repassados à função de minimização irrestrita, que foi implementada da mesma forma tanto na versão sequencial quanto na versão paralela.

Um outro ponto do algoritmo XCM que permite uma implementação paralela é na criação da chamada *lista de associações*. Uma vez calculada a solução final

X^*, é necessário associar as observações a um, e apenas um, cluster seguindo a regra estabelecida para o agrupamento. Considerando a miníma soma-de-quadrados como exemplo, uma observação estará associada ao centroid que estiver mais perto considerando-se a menor distância euclidiana como referência. Como existem q centroids, cada observação deve calcular q distâncias e se associar ao centroid mais próximo. Levando-se em consideração que o posicionamento dos centroids já está definido e é visível para todas as observações, a observação 1 pode descobrir a que centroid está associada de forma independente da observação 2, e assim por diante. No entanto, a implementação paralela da lista de associações não foi feita para que a comparação dos resultados de desempenho sejam puramente focadas na implementação paralela do cálculo da função objetivo e do gradiente da mesma.

4.2 O algoritmo do XCM-OpenMP

Tanto o algoritmo XCM quanto o algoritmo XCM-OpenMP possuem uma etapa de inicialização, onde os valores dos parâmetros são defindos e a estimativa inicial dos centroids é calculada. A diferença de inicialização entre o XCM e o XCM-OpenMP está no método allocate_memory, que leva em consideração o número de threads na alocação de tamanho nos vetores que serão usados durante o processo de cálculo do método.

Uma vez que todas as etapas de inicialização estejam concluídas, o algoritmo XCM-OpenMP é iniciado. Na versão paralela, a chamada da biblioteca de minimização irrestrita L-BFGS para o cálculo da função objetivo e de seu gradiente são substituídas por uma função que recebe todos os parâmetros e faz a divisão do trabalho computacional através das diversas threads da CPU. O cálculo de cada parcela paralela de z_j é feita utilizando a diretiva openMP `omp_get_thread_num` que é responsável por determinar o valor(id) de cada thread que está sendo usada no momento. Dessa forma, não é necessário nenhum controle de concorrência aos dados, pois cada thread acessa somente os endereços de memória a ela destinados.

Quando as threads terminarem sua execução, as diversas parcelas do cálculo da função objetivo f e de seu gradiente estarão "espalhadas"pela memória global da CPU. No caso da função objetivo, apenas um valor deve ser retornado. Este

conceito está associado à ideia de *Redução*.

Em relação a função objetivo $\sum z_j$, apenas uma operação de redução é necessária. No entanto, no caso da função gradiente, diversas reduções (mais precisamente, $p*q$ reduções, onde p é a dimensão do espaço do problema e q é o número de centroids) são necessárias para que esse vetor seja corretamente calculado.

Uma versão resumida do algoritmo XCM-OpenMP está presente no algoritmo 3 abaixo:

Algoritmo 3 Algoritmo XCM-OpenMP simplificado

Inicialização: Defina os valores iniciais de $x^0, \varepsilon^1, \tau^1, \gamma^1$ e os valores $0 < \rho_1 < 1, 0 < \rho_2 < 1, 0 < \rho_3 < 1$; Efetue a inicialização do XCM-OpenMP
  $k \leftarrow 1$
  while Regra de parada não for atingida **do**
    Copie a solução x^{k-1} e os parâmetros $\varepsilon^k, \tau^k, \gamma^k$ para a memória.
    Resolva o problema 4.1 e 4.2 que são as versões paralelas do problema 3.17 e 3.19 nos núcleos de processamento em execução com $\varepsilon = \varepsilon^k, \tau = \tau^k, \gamma = \gamma^k$ utilizando a solução inicial x^{k-1} . Seja x^k a solução obtida.
    $\varepsilon^{k+1} \leftarrow \rho_1 \varepsilon^k$
    $\tau^{k+1} \leftarrow \rho_2 \tau^k$
    $\gamma^{k+1} \leftarrow \rho_3 \gamma^k$
    $k \leftarrow k + 1$
  end while

4.3 Arquitetura Orientada a Objetos para Comparação de Algoritmos

Um dos objetivos centrais desse trabalho é fazer a comparação entre o desempenho dos algoritmos XCM e XCM-OpenMP. Para essa finalidade, foi proposto nesse trabalho uma arquitetura orientada a objetos onde se escreve os códigos comuns do XCM e do XCM-OpenMP uma única vez na classe IXCM (I de interface). Essa arquitetura é chamada de "Arquitetura Orientada a Objetos para Comparação de Algoritmos".

Na figura 4.1 tem-se o diagrama de classe do método proposto.

Na classe base IXCM foi implementado o código comum entre XCM e XCM-OpenMP. Dentre os códigos comuns estão o método run (que executa o cálculo

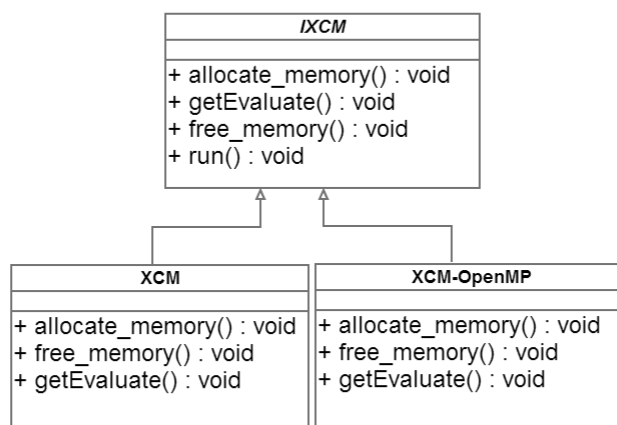

Figura 4.1: Diagrama de Classes da Arquitetura Orientada a Objetos para Comparação de Algoritmos

principal), diversos métodos auxiliares, os casos de teste, os atributos gerais do método XCM e outros.

Na classe derivada XCM foi implementada a versão sequencial do XCM, e na classe derivada XCM-OpenMP foi implementada a versão *multithread* do XCM.

Os métodos virtuais da classe base IXCM estão listados abaixo.

- allocate_memory

  Esse método é usado para alocar memória usada no algoritimo e salvar a referência dessa alocação nos ponteiros, que são membros da classe IXCM. Devido a requisições do tipo de paralelismo usado, pode ser necessário um tipo diferente de alocação de memória. Por isso esse método deve ser virtual.

- initializationAfterAlloc

  Esse método é chamado após a alocação de dados, leitura de disco e preenchimento dos dados de observação.

- free_memory

  Esse método é chamado para se liberar a memória. Como a alocação de dados pode variar dependendo da implementação do algoritmo XCM, a liberação de memória também precisa ser customizada.

- getEvaluate

  Esse método é usado para retornar o ponteiro de função "evaluate" que será usada como argumento da função global L-BFGS. O método não "run" per-

gunta qual o ponteiro da função “evaluate” chamando o método “getEvaluate”, e chama o L-BFGS com esse valor. O resultado é que cada implementação (sequencial ou paralela) do XCM pode implementar como quiser o cálculo da função de custo e de sua derivada. Par isso, basta escrever o código da função estática que realiza esse cálculo e retornar o ponteiro dessa função pelo método getEvaluate. No caso da classe XCM, o retorno é o método estático evaluate_sequencial. No XCM-OpenMP o retorno é o método estático evaluate_paralelo.

Capítulo 5

Resultados Computacionais

A implementação do XCM-OpenMP feita utilizando-se o software livre NetBeans 7.3 que foi executada no servidor Coyote do Labotim que possui 24 núcleos de processamento, no Laboratório de Otimização do PESC/COPPE/UFRJ. A biblioteca L-BFGS, disponível em [13], foi utilizada para calcular as minimizações irrestritas. O sistema operacional utilizado foi o Suse 11.2 para a verão final compilada no servidor, e para testes locais a versão do Ubuntu 13.04. Para a versão paralela foi utilizada a biblioteca OpenMP na sua versão 3.1. A visualização dos resultados foi gerada através do software Octave.

Nos testes operacionais, os pontos iniciais do algoritmo iterativo foram definidos usando-se o ponto médio das observações para definir um ponto base. Em seguida, os outros pontos do conjunto X inicial foram gerados através de uma perturbação nesse ponto base multiplicando-se um número aleatório com distribuição uniforme entre zero e um com o desvio-padrão das observações e um fator de expansão proporcional a ordem de grandeza dos valores das coordenadas das observações.

Vale ressaltar que esse método para a escolha do ponto inicial foi puramente intuitiva e, utilizando-se uma heurística mais desenvolvida é possível obter uma estimativa muito melhor para os pontos iniciais. Mesmo assim, os resultados apresentados são extremamente satisfatórios.

Todos os fatores ρ do algoritmo foram inicializados com valor igual a 0.5 e os parâmetros $\gamma > 0$, $\varepsilon > 0$ e $\tau > 0$. Os parâmetros γ, ε e τ se aproximam de zero conforme o valor da função objetivo se aproxima do ótimo.

5.1 Problemas de Pequeno Porte

Para verificar a correta implementação do algoritmo XCM-OpenMP (e da própria implementação base do XCM), um pequeno caso de teste foram utilizados. Foi feito um exemplo extremamente simples, capaz de ser resolvido rapidamente sem o uso de um computador. Este teste consiste na definição de 4 pontos no $\mathbb{R}^2$, localizados nas coordenadas $(0,1), (1,3), (6,0)$ e $(8,0)$ que devem ser agrupados em dois conjuntos distintos.

Esse caso de teste pode ser visto na figura 5.1. Dessa forma, temos m, o número de observações, igual a 4, n, a dimensão do espaço das observações, igual a 2 e q, o número de centroids, igual a 2.

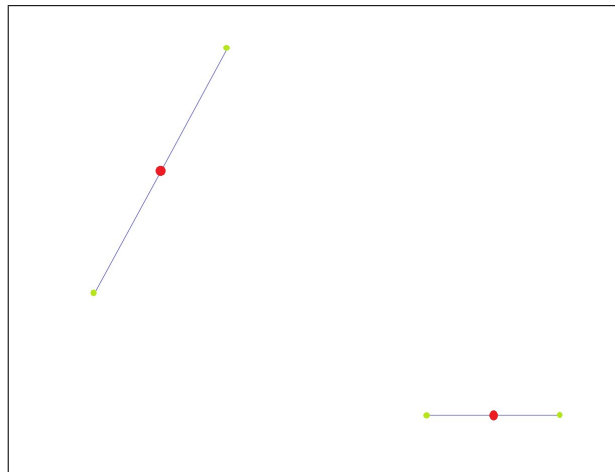

Figura 5.1: Caso de teste simples.

O esquema de cores definido nas figuras resultantes dos casos de teste é definido da seguinte forma:

- Pontos em verde marcam o posicionamento das observações.
- Pontos em vermelho marcam o posicionamento calculado dos centroids.
- As linhas em azul fazem as ligações entre as observações e os centroids a que estão associados.

Todo o desenvolvimento do XCM-OpenMP foi feito utilizando esse caso de teste como verificação da implementação, levando em consideração a acurácia do resultado. Em questão de performance, dado que o problema é extremamente simples, a versão sequencial XCM leva vantagem sobre o XCM-OpenMP. O tempo do cálculo

sequencial é de 0.16 segundos, enquanto que a versão paralela levou 0.52 segundos para ser calculada utilizando-se 4 threads.

5.2 Problemas de Médio e Grande Porte

Após a validação dos resultados dos problemas de pequeno porte, foram considerados os problemas de tamanho médio, com 3038 e 15112 observações e de grande porte, com 85900 observações. O problema de 3038 observações utiliza-se os dados relacionado ao problema clássico do caixeiro viajante da literatura, disponibilizado na biblioteca TSPLIB [15]. As imagens resultantes do caso de teste com 3038 observações podem ser vistas a seguir, nas figuras 5.2, 5.3, 5.4, 5.5, 5.6, 5.7.

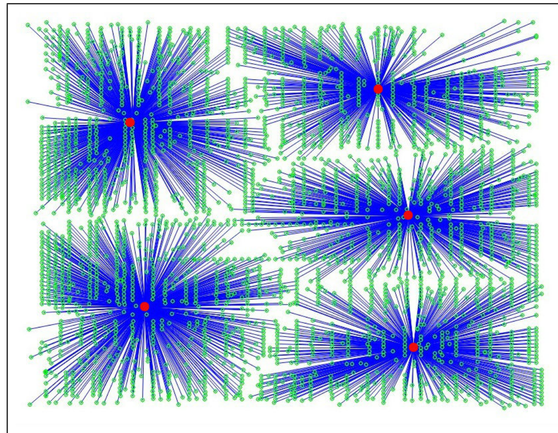

Figura 5.2: Problema TSPLIB-3038 (q = 5).

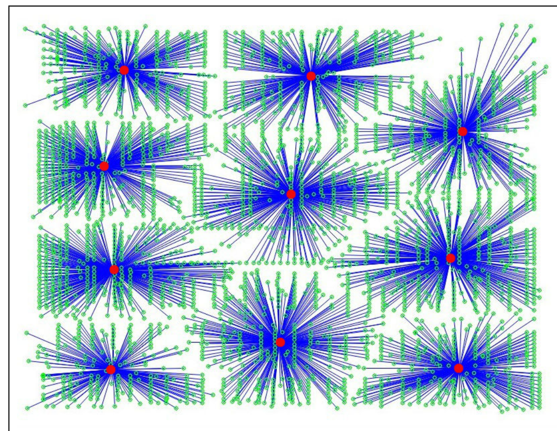

Figura 5.3: Problema TSPLIB-3038 (q = 10).

Os resultados das comparações de tempo de execução entre as implementações sequencial e paralela podem ser vistas na tabela 5.1. O tempo da implementação

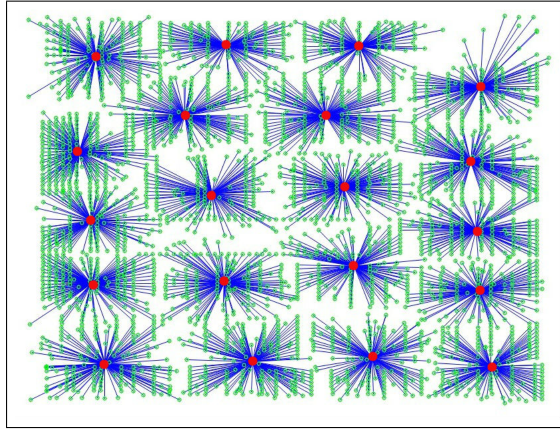

Figura 5.4: Problema TSPLIB-3038 (q = 20).

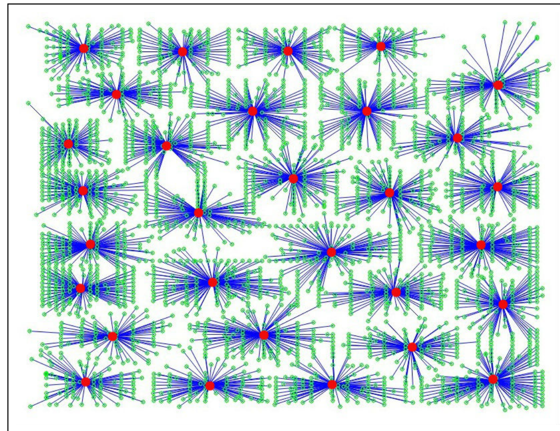

Figura 5.5: Problema TSPLIB-3038 (q = 30).

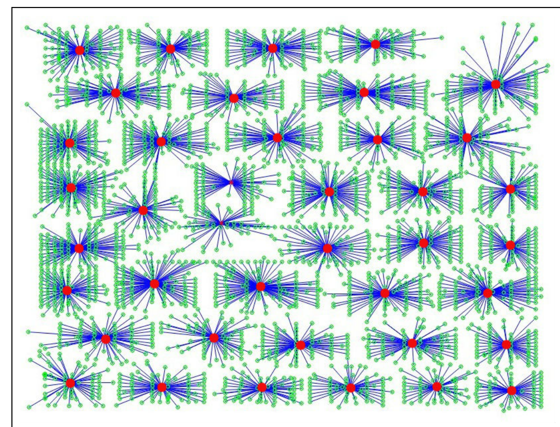

Figura 5.6: Problema TSPLIB-3038 (q = 40).

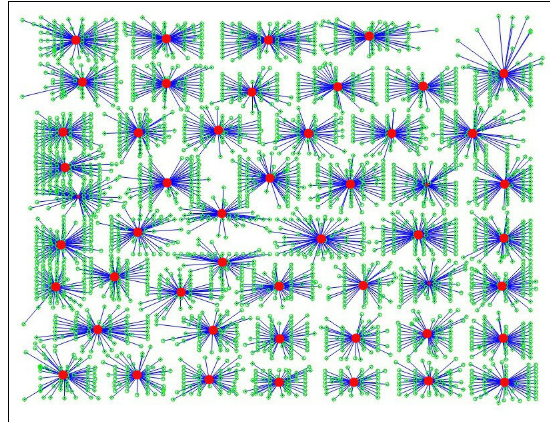

Figura 5.7: Problema TSPLIB-3038 (q = 50).

paralela foi calculado usando-se 24 threads em todos os casos de teste. Na figura 5.8 pode-se observar um gráfico comparando os tempos de execução das implementações sequencial e paralela.

Número de Centroids	Tempo Seq.	Tempo Par.	Speed Up
5	3.204	0.302	10.609
10	8.861	0.589	14.044
20	31.701	1.112	28.508
30	82.946	3.949	21.004
40	122.795	5.655	21.714
50	242.395	14.989	16.171

Tabela 5.1: Resultados de Tempo e Speed Up para o problema TSPLIB-3038

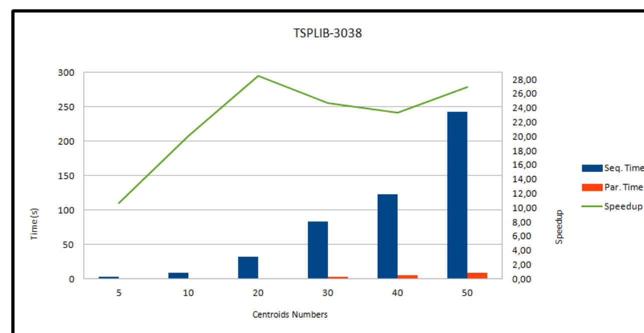

Figura 5.8: Gráfico dos resultados do problema TSPLIB-3038.

Na figura 5.9 é apresentado o gráfico para a curva de speed-up do problema 3038 com $q = 50$ e variando o número de processadores trabalhando em paralelo

Os resultados do caso de teste para o problema com 15112 observações podem ser vistos nas figuras 5.10, 5.11, 5.12, 5.13, 5.14, 5.15, também considerando 5, 10, 15, 20, 25 e 30 centroids, respectivamente.

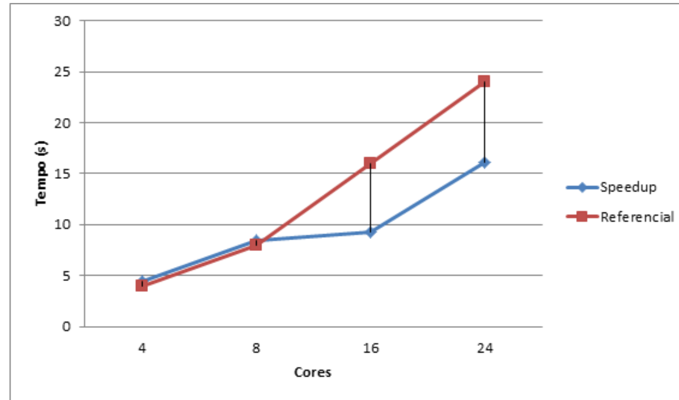

Figura 5.9: Gráfico de speed-up do problema TSPLIB-3038.

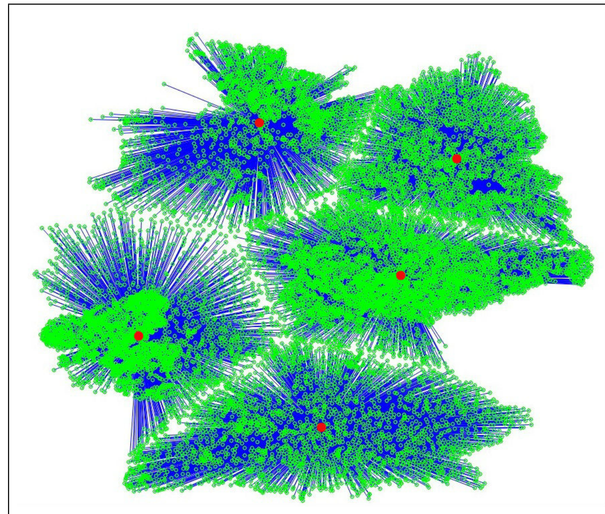

Figura 5.10: Problema com 15112 observações (q = 5).

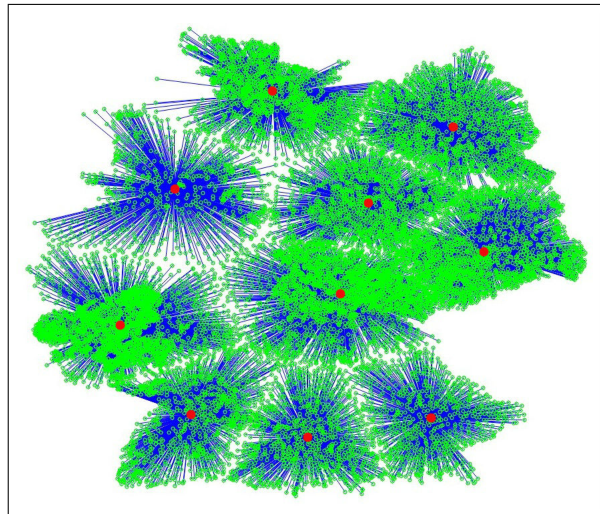

Figura 5.11: Problema com 15112 observações (q = 10).

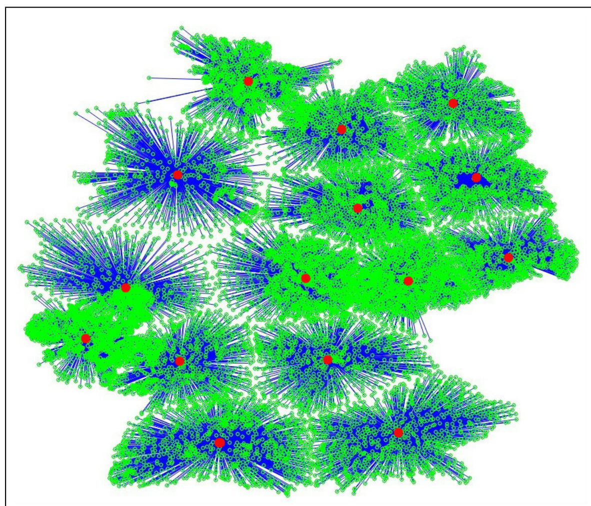

Figura 5.12: Problema com 15112 observações (q = 15).

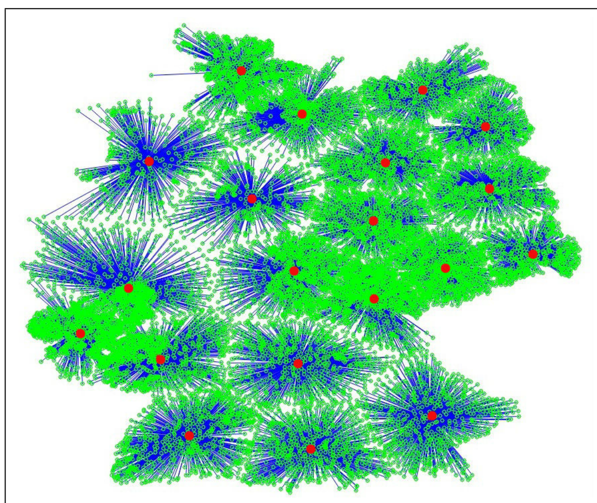

Figura 5.13: Problema com 15112 observações (q = 20).

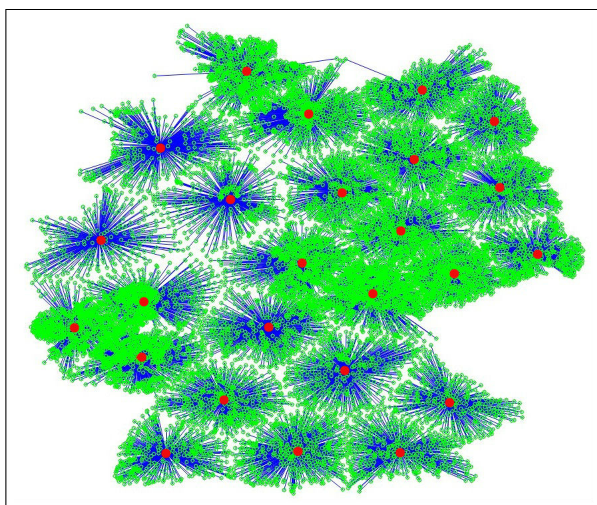

Figura 5.14: Problema com 15112 observações (q = 25).

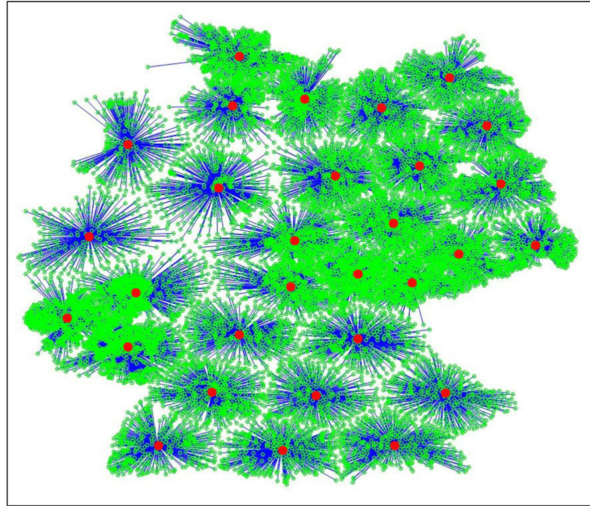

Figura 5.15: Problema com 15112 observações (q = 30).

Os resultados das comparações de tempo de execução entre as implementações sequencial e paralela podem ser vistos na tabela 5.2. Como no caso anterior, as mesmas 24 threads foram usadas na medição de tempo. Na figura 5.16 é possível observar o gráfico com os resultados de tempo.

Número de Centroids	Tempo Seq.	Tempo Par.	Speed Up
5	18.140	1.419	12.783
10	54.562	3.369	16.195
15	100.120	5.994	16.718
20	197.397	10.312	19.142
25	255.437	23.264	10.979
30	349.601	17.504	19.972

Tabela 5.2: Resultados de Tempo e Speed Up para o problema 15112

Na figura 5.17 é apresentado o gráfico para a curva de speed-up do problema 15112 com $q = 30$ e variando o número de processadores trabalhando em paralelo

Para o problema com 85900 observações, foram medidos os tempos como nos casos de teste anteriores, como pode ser visto nas figuras 5.18, 5.19, 5.20, 5.21 e 5.22. Os resultados das medições de tempo encontram-se na tabela 5.3. O gráfico com a comparação dos tempos de execução para esse problema é visto na figura 5.23.

Na figura 5.24 é apresentado o gráfico para a curva de speed-up do problema 85900 com $q = 30$ e variando o número de processadores trabalhando em paralelo

Analisando os gráficos 5.9, 5.17 e 5.24 observa-se que a curva de speed-up se aproxima do referencial teórico de speed-up (quando o speed-up é igual ao número

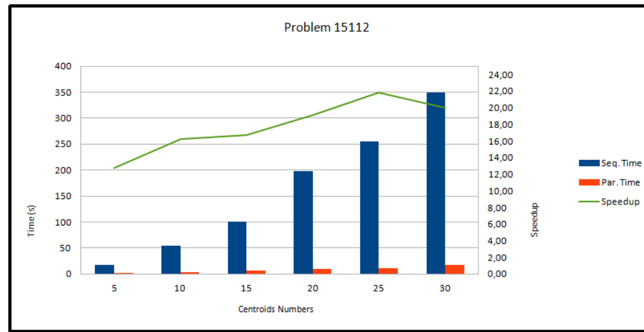

Figura 5.16: Gráfico dos resultados do problema 15112.

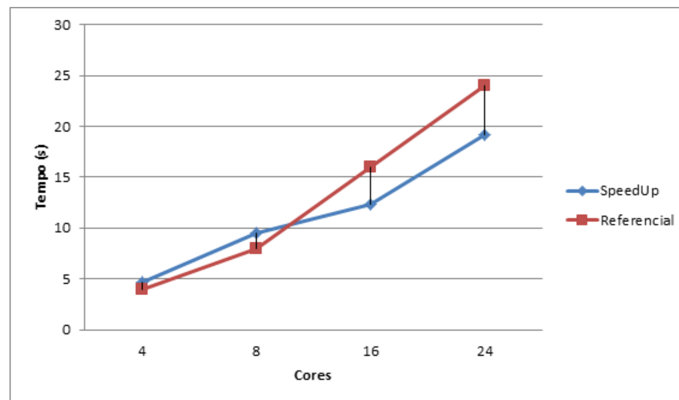

Figura 5.17: Gráfico de speed-up do problema 15112.

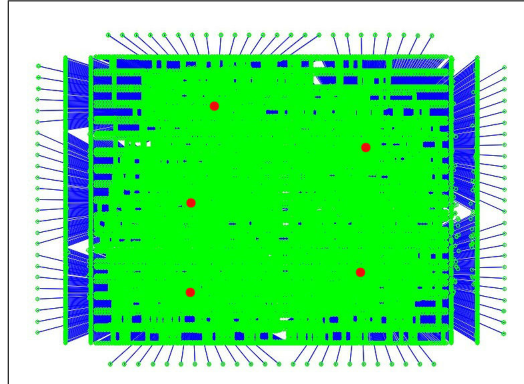

Figura 5.18: Problema com 85900 observações (q = 5).

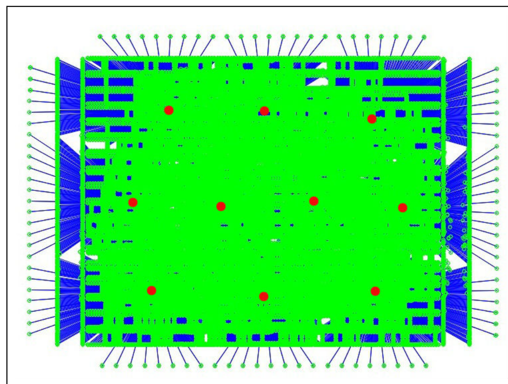

Figura 5.19: Problema com 85900 observações (q = 10).

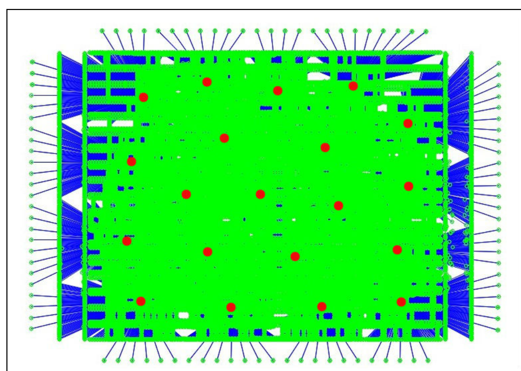

Figura 5.20: Problema com 85900 observações (q = 20).

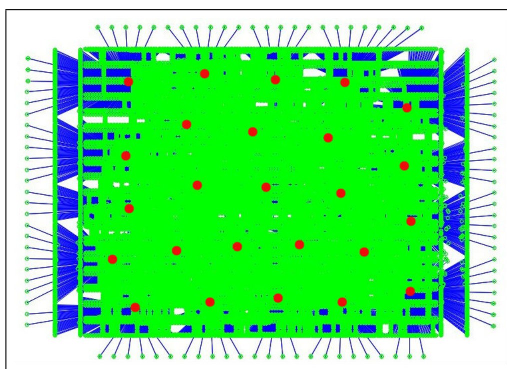

Figura 5.21: Problema com 85900 observações (q = 25).

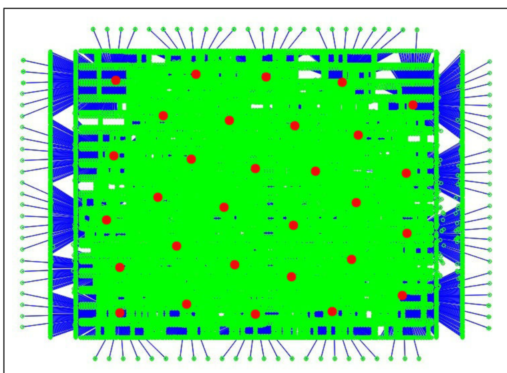

Figura 5.22: Problema com 85900 observações (q = 30).

Número de Centroids	Tempo Seq.	Tempo Par.	Speed Up
5	103.594	9.137	11.337
10	326.780	20.863	15.663
15	593.960	33.285	17.844
20	1059.451	55.963	19.023
25	1471.468	70.779	20.790
30	2253.773	117.727	19.144

Tabela 5.3: Resultados de Tempo e Speed Up para o problema 85900

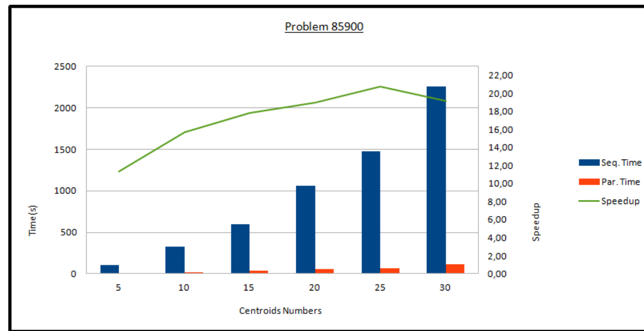

Figura 5.23: Gráfico dos resultados do problema 85900.

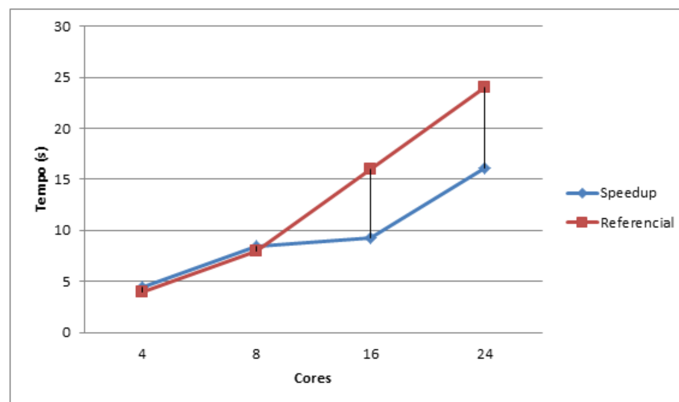

Figura 5.24: Gráfico de speed-up do problema 85900.

de cores) quando os problemas são executados com um número menor de cores. Verificou-se também que com os exemplos estudados não há correlação no aumento da dimensão do problema e a taxa de paralelização (relação entre speed-up obtido e o referencial teórico) do algoritmo proposto. No problema 3038, com 24 cores, a taxa de paralelização foi de 67%. No problema 15112, com 24 cores, a taxa de paralelização foi de 79%. No problema 85900, com 24 cores, a taxa de paralelização foi de 67%.

Capítulo 6

Conclusão

O problema de agrupamento (*clustering*), formulado segundo o critério de mínima de soma de quadrados, é um problema numérico muito importante, útil e de difícil solução (NP-Hard).

O método de clusterização proposto por Xavier, conhecido por XCM (Xavier Clustering Method), introduz um melhora significativa na resolução do problema de agrupamento, com excelentes resultados em relação aos métodos tradicionais.

As partes paralelizáveis do XCM foram identificadas após uma análise detalhada. Nesta tese optou-se por explorar o OpenMP como a arquitetura de hardware para a computação paralela dessas partes do algoritmo XCM. Dessa forma, foi proposto o método que chamamos de XCM-OpenMP.

Para a comparação dos resultdos do XCM com o XCM-OpenMP, propôs-se uma "Arquitetura Orientada a Objetos para Comparação de Algoritmos", em que em uma classe base IXCM foi implementado o código comum entre XCM e XCM-OpenMP. Na classe derivada XCM foi implementada a versão sequencial do XCM, e na classe derivada XCM-OpenMP foi implementada a versão *Multithread* do XCM.

Os resultados experimentais, como esperado, mostraram grande speedup da versão XCM-OpenMP em relação a versão XCM. Nos testes realizados, o maior speedup foi maior que 28 vezes.

Como esperado o speed-up do método proposto aumenta com o aumento do número de processadores utilizados. O referencial teórico de speed-up (speed-up igual ao número de processadores, com taxa de paralelização 100%) foi atingido nos casos em que há poucos processadores (até 8), e gradual queda da taxa de

paralelização com o aumento do número de processadores.

6.1 Trabalhos Futuros

O XCM-OpenMP permite desenvolvimentos futuros, buscando-se ainda mais melhora de performance. Como um modelo híbrido utilizando-se das arquiteruras OpenMP, MPI, Trebuchet para buscar maior eficiência na resolução dos problemas de agrupamento.

A biblioteca L-BFGS utilizada por ser uma rotina externa não foi alterada, apesar de ser um código aberto. Constatou-se após alguns testes que há pontos em que a rotina não converge e volta ao ponto inicial. Com isso é possível uma alteração no código para o reaproveitamento de pontos que já tenham convergido anteriormente.

Seria possível armazenar a informação da relação entre observação e centroid no momento do cálculo do z_j. Uma alteração no código nesse sentido pode ser feita para reaproveitar o valor que já foi calculado.

Referências Bibliográficas

[1] TA Alves, Leandro AJ Marzulo, Felipe MG Franca, and VS Costa. Trebuchet: Explorando tlp com virtualizaçao dataflow. *WSCAD-SSC?09*, pages 60–67.

[2] OpenMP ARB. http://openmp.org/wp/openmp-specifications/.

[3] M. Leese B. S. Everitt, S. Landau and D. Stahl. *Cluster Analysis*. John Wiley & Sons, 2011. ISBN-13: 9780387975061.

[4] P. Brucker. On the complexity of clustering problems. *Optimization and Operations Research*, 1978.

[5] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. The MIT Press, 2008.

[6] M. J. Flynn. Very high-speed computing systems. proceeding of the ieee. v.54 , n.12:1901 – 1909, 1966.

[7] The MPI Forum. Mpi: a message passing interface. In *Proceedings of the Conference on High Performance Networking and Computing*, pages 878–883, 1993.

[8] T. J. Fountain. *Parallel Computing: Principles and Practice*. Cambridge University Press, 2006. ISBN-13: 0521451310.

[9] Matteo Frigo. Multithreaded programming in cilk. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 13–14. ACM, 2007.

[10] William D Gropp, Ewing L Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. the MIT Press, 1999.

[11] M. W. Mendonça. O método l-bfgs com fatoração incompleta para a resolução de problemas de minimização. 2005.

[12] J. Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35 (151)(151):773–782, 1980.

[13] Jorge Nocedal and Naoaki Okazaki. http://www.chokkan.org/software/liblbfgs/.

[14] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[15] G. Reinelt. Tsplib:a traveling salesman library. *ORSA Journal of Computing*, pages 376–384, 1991.

[16] L.C.F. Sousa. Desempenho computacional do método de agrupamento via suavização hiperbólica. *M.Sc. Thesis, COPPE/UFRJ, Rio de Janeiro, RJ*, 2005.

[17] A. E. Xavier. Penalização hiperbólica: Um novo método para resolução de problemas de otimização. *M.Sc. Thesis, COPPE/UFRJ, Rio de Janeiro, RJ*, 1982.

[18] A. E. Xavier. The hyperbolic smoothing clustering method. *Pattern Recognition*, 43 (3):731–737, 2010.

[19] A. E. Xavier and A. A. F. Oliveira. Optimal covering of plane domains by circles via hyperbolic smoothing. *Journal of Global Optimization*, 31 (3)(3):493–504, 2005.

[20] A. E. Xavier and V. L. Xavier. Solving the minimum sum-of-squares clustering problem by hyperbolic smoothing and partition into boundary and gravitational regions. *Pattern Recognition*, 44 (1):70–77, 2011.