Pontifícia Universidade Católica
do Rio de Janeiro

Andre Luis Cavalcanti Bueno

Relaxamento Adaptativo da Sincronização Através do Uso de Métodos de Aprendizagem Supervisionada

Tese de Doutorado

Tese apresentada como requisito parcial para obtenção do grau de Doutor pelo Programa de Pós–graduação em Informática da PUC-Rio.

Orientadora : Profª. Noemi de La Rocque Rodriguez
Co-orientadora: Profª. Elisa Dominguez Sotelino

Rio de Janeiro
Março de 2018

Pontifícia Universidade Católica
do Rio de Janeiro

Andre Luis Cavalcanti Bueno

Relaxamento Adaptativo da Sincronização Através do Uso de Métodos de Aprendizagem Supervisionada

Tese apresentada como requisito parcial para obtenção do grau de Doutor pelo Programa de Pós–graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Profª. Noemi de La Rocque Rodriguez
Orientadora
Departamento de Informática – PUC-Rio

Profª. Elisa Dominguez Sotelino
Co-orientadora
Departamento de Engenharia Civil – PUC-Rio

Profª. Ana Lúcia de Moura
Departamento de Informática – PUC-Rio

Prof. Hélio Côrtes Vieira Lopes
Departamento de Informática – PUC-Rio

Profª. Marley Maria Bernardes Rebuzzi Vellasco
Departamento de Engenharia Elétrica – PUC-Rio

Profª. Lucia Maria de Assumpção Drummond
Departamento de Informática – UFF

Profª. Silvana Rossetto
Departamento de Informática – UFRJ

Prof. Marcio da Silveira Carvalho
Coordenador Setorial do Centro Técnico Científico – PUC-Rio

Rio de Janeiro, 7 de Março de 2018

Andre Luis Cavalcanti Bueno

Graduou-se em Engenharia de Computação pela PUC-Rio em 2010. Concluiu o Mestrado em Informática pela PUC-Rio em 2013. Em 2007 fez iniciação científica em métodos de resolução de equações diferenciais. De 2008 a 2011 trabalhou no Laboratório de Inteligência Computacional Aplicada (ICA) pertencente ao departamento de Engenharia elétrica da PUC-Rio. Trabalha desde 2011 no Instituto Tecgraf na PUC-Rio.

Ficha Catalográfica

Cavalcanti Bueno, Andre Luis

Relaxamento Adaptativo da Sincronização Através do Uso de Métodos de Aprendizagem Supervisionada / Andre Luis Cavalcanti Bueno; orientador: Noemi de La Rocque Rodriguez; co-orientador: Elisa Dominguez Sotelino. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2018.

v., 80 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. Relaxamento de Sincronização;. 3. Computação de Alto Desempenho;. 4. Computação Aproximada;. 5. Computação Paralela;. 6. Métodos de Aprendizagem Supervisionada.. I. Noemi de La Rocque Rodriguez, Noemi. II. Dominguez Sotelino, Elisa. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 620.11

Dedicado ao meu querido avô João Cavalcanti (*in memoriam*).

Agradecimentos

Primeiramente, agradeço às minhas orientadoras, Profª. Noemi Rodriguez e Profª. Elisa Sotelino. Nesses 7 anos de trabalho em equipe pude adquirir, em nossas incontáveis reuniões, um aprendizado inestimável. Muito obrigado por todo carinho, respeito, paciência e amizade.

Agradeço ao Instituto Tecgraf, em especial ao grupo MGEO coordenado pelo Dr. Márcio Santi. Fazer parte desse grupo, que é quase uma família para mim, foi de extrema importância para a plena conclusão deste trabalho.

Agradeço à CAPES e PUC-Rio, pelos auxílios concedidos para que este trabalho fosse plenamente realizado.

Agradeço à minha mãe por todo o carinho, incentivo, por sempre acreditar em mim e nunca me deixar desanimar.

Agradeço aos colegas, professores e funcionários do Departamento de Informática da PUC-Rio, por me acompanharem em mais esta jornada.

Resumo

Cavalcanti Bueno, Andre Luis; Noemi de La Rocque Rodriguez, Noemi; Dominguez Sotelino, Elisa. **Relaxamento Adaptativo da Sincronização Através do Uso de Métodos de Aprendizagem Supervisionada**. Rio de Janeiro, 2018. 80p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Sistemas de computação paralelos vêm se tornando pervasivos, sendo usados para interagir com o mundo físico e processar uma grande quantidade de dados de várias fontes. É essencial, portanto, a melhora contínua do desempenho computacional para acompanhar o ritmo crescente da quantidade de informações que precisam ser processadas. Algumas dessas aplicações admitem uma menor qualidade no resultado final em troca do aumento do desempenho de execução. Este trabalho tem por objetivo avaliar a viabilidade de usar métodos de aprendizagem supervisionada para garantir que a técnica de Sincronização Relaxada, utilizada para o aumento do desempenho de execução, forneça resultados dentro de limites aceitáveis de erro. Para isso, criamos uma metodologia que utiliza alguns dados de entrada para montar casos de testes que, ao serem executados, irão fornecer valores representativos de entrada para o treinamento de métodos de aprendizagem supervisionada. Dessa forma, quando o usuário utilizar a sua aplicação (no mesmo ambiente de treinamento) com uma nova entrada, o algoritmo de classificação treinado irá sugerir o fator de relaxamento de sincronização mais adequado à tripla aplicação/entrada/ambiente de execução. Utilizamos essa metodologia em algumas aplicações paralelas bem conhecidas e mostramos que, aliando a Sincronização Relaxada a métodos de aprendizagem supervisionada, foi possível manter a taxa de erro máximo acordada. Além disso, avaliamos o ganho de desempenho obtido com essa técnica para alguns cenários em cada aplicação.

Palavras-chave

Relaxamento de Sincronização; Computação de Alto Desempenho; Computação Aproximada; Computação Paralela; Métodos de Aprendizagem Supervisionada.

Abstract

Cavalcanti Bueno, Andre Luis; Noemi de La Rocque Rodriguez, Noemi (Advisor); Dominguez Sotelino, Elisa (Co-Advisor). **Adaptive Relaxed Synchronization Through the Use of Supervised Learning Methods**. Rio de Janeiro, 2018. 80p. Tese de doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Parallel computing systems have become pervasive, being used to interact with the physical world and process a large amount of data from various sources. It is essential, therefore, the continuous improvement of computational performance to keep up with the increasing rate of the amount of information that needs to be processed. Some of these applications admit lower quality in the final result in exchange for increased execution performance. This work aims to evaluate the feasibility of using supervised learning methods to ensure that the Relaxed Synchronization technique, used to increase execution performance, provides results within acceptable limits of error. To do so, we have created a methodology that uses some input data to assemble test cases that, when executed, will provide input values for the training of supervised learning methods. This way, when the user uses his/her application (in the same training environment) with a new input, the trained classification algorithm will suggest the relax synchronization factor that is best suited to the triple application/input/execution environment. We used this methodology in some well-known parallel applications and showed that, by combining Relaxed Synchronization with supervised learning methods, it was possible to maintain the maximum established error rate. In addition, we evaluated the performance gain obtained with this technique for a number of scenarios in each application.

Keywords

Relaxed Synchronization; High Performance Computing; Approximate Computing; Parallel Computing; Supervised Learning Methods.

Sumário

Lista de figuras

Lista de tabelas

Man can indeed do what he wants, but he cannot will what he wants.

Arthur Schopenhauer, *The World as Will and Representation.*

1
Introdução

Sistemas de computação paralelos vêm se tornando pervasivos, sendo usados para interagir com o mundo físico e processar uma grande quantidade de dados de várias fontes. É essencial, portanto, a melhora contínua do desempenho computacional para acompanhar o ritmo crescente da quantidade de informações que precisam ser processadas. Afortunadamente, algumas dessas aplicações possuem a propriedade intrínseca da resiliência ao erro [1]. Muitas vezes, elas processam dados ruidosos e redundantes de entrada (dados oriundos por exemplo de sensores) e, como consequência, seus algoritmos associados são normalmente de natureza imprecisa. Como não requerem a computação de um único resultado específico, aceitam uma gama de aproximações. Por exemplo, no processamento multimídia (imagem, som e vídeo) devido à limitada percepção humana, erros como queda de determinado *frame* ou uma perda pequena da qualidade de imagem raramente afetam a satisfação final do usuário. Como outro exemplo, em análise de dados, um mesmo classificador, implementado de mais de uma forma, pode produzir resultados diferentes de classificação em um conjunto de objetos. Porém, em grande parte das vezes, é muito difícil, senão impossível, dizer qual classificação é melhor.

Para obter uma execução de alto desempenho em programas paralelos é necessário, além de boas práticas de programação [2], reduzir o uso excessivo de primitivas de sincronização como, por exemplo, as usadas para ordenar os acessos à memória compartilhada. A área da *Computação Aproximada* [3] surgiu para lidar com a habilidade de muitos sistemas e aplicações tolerarem alguma perda de qualidade no resultado da computação em troca de benefícios

no desempenho.

Neste trabalho, estamos interessados especificamente na sub-área da Computação Aproximada conhecida como *Sincronização Relaxada* [4]. Ela age reduzindo o *overhead* causado pela sincronização através da minimização ou até mesmo completa remoção dos pontos de sincronização. Essa técnica pode ser usada em aplicações onde o resultado produzido pelo programa original (com os pontos de sincronização inalterados) admita uma margem de tolerância ao erro nos resultados. Naturalmente, a aplicação dessa técnica promove o surgimento de condições de corrida que, por possuirem um comportamento indeterminado, são recriminadas por alguns autores. Em contrapartida, outros pesquisadores as toleram como uma forma eficiente de aumento de desempenho [4, 5, 6, 7].

1.1 Motivação e Objetivo

Após o estudo de trabalhos [4, 5, 6, 7] que utilizam a Sincronização Relaxada, observamos que um desafio fundamental de qualquer sistema que a aplica é garantir que o programa relaxado irá produzir resultados que estejam dentro dos limites de acurácia desejados para diferentes entradas e ambientes/condições de execução. Todos os trabalhos procuram garantir grande probabilidade estatística da produção de resultados dentro destes limites baseando-se em um conjunto de entradas representativas, sendo essa garantia válida apenas para o ambiente de execução em que foi testado.

Este trabalho tem por objetivo avaliar a viabilidade de usar métodos de aprendizagem supervisionada [8] para garantir que a técnica de Sincronização Relaxada forneça resultados dentro de limites aceitáveis de erro. Para isso, criamos uma metodologia que utiliza alguns dados de entrada para montar casos de testes que, ao serem executados, irão fornecer valores representativos de entrada para o treinamento de métodos de aprendizagem supervisionada. Dessa forma, quando o usuário utilizar a sua aplicação (no mesmo ambiente de treinamento) com uma nova entrada, o algoritmo de

classificação treinado irá sugerir o fator de relaxamento mais adequado à tripla aplicação/entrada/ambiente de execução.

Analisamos os resultados através da qualidade das previsões obtidas e também nos ganhos de tempo em relação à versão paralela e 100% sincronizada.

1.2 Escopo

Como cenários de exemplo da aplicação de nosso trabalho, utilizamos duas aplicações provenientes de benchmarks bem conhecidos: *Labyrinth* [9] e *Graph 500*, e também o algoritmo de *K-Means* [10] aplicado à segmentação de cores em vídeos.

Utilizamos, para todos os exemplos estudados, o *software* Matlab [11] como ferramenta na criação e treinamento dos métodos de aprendizagem supervisionada. Como o nosso objetivo principal é a comprovação da viabilidade de aliar a técnica de Sincronização Relaxada com métodos de aprendizagem supervisionada, não nos preocupamos com a otimização de parâmetros intrínsecos de cada um dos métodos de aprendizagem, deixando todos com os valores sugeridos pelo Matlab.

Esse trabalho analisou somente aplicações que possuem somente um ponto de sincronização. Dessa forma, os experimentos de relaxamento de sincronização foram feitos para somente um ponto de sincronização por aplicação.

1.3 Organização da Tese

O restante da tese está organizado da seguinte forma: O Capítulo 2 descreve os conceitos básicos para o entendimento do nosso trabalho. O Capítulo 3 apresenta em detalhes a metodologia criada. O Capítulo 4 apresenta a aplicação da metodologia em três exemplos e seus resultados. Por fim, as conclusões são descritas no Capítulo 5.

2
Conceitos Básicos

Neste capítulo apresentamos os conceitos básicos relacionados a esta tese. Primeiro, o conceito de Sincronização Relaxada é abordado com uma breve discussão sobre os efeitos de condições de corrida e a seguir introduzimos o tema da aprendizagem supervisionada.

2.1
Computação Aproximada

A área da Computação Aproximada lida com aplicações que admitem uma margem de *trade-off* entre precisão e custo. Nela tenta-se obter uma melhoria de desempenho através da degradação na qualidade dos resultados. A Figura 2.1 apresenta uma representação gráfica deste espaço de *trade-off*. Como mostrado, existe um conjunto de pontos que servem como soluções aceitáveis para um determinado problema.

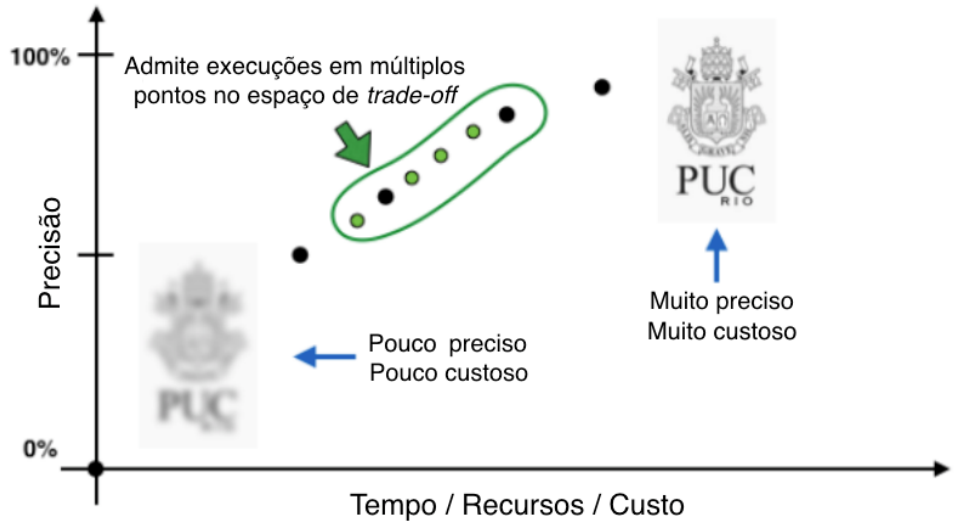

Figura 2.1: Espaço de *trade-off* da Computação Aproximada.

As principais aplicações da Computação Aproximada são: busca (procura de respostas imprecisas para consultas imprecisas), entrega proativa de

informações (propagandas, ofertas, notícias, alertas), mídia (renderização aproximada de áudio, vídeo ou imagens) e processamento de fluxo de dados (decisões rápidas baseadas em dados possivelmente incompletos). Estes tipos de aplicações permitem a troca da precisão dos resultados por um tempo de resposta mais curto. Neste cenário, tipicamente existe uma solução altamente precisa e cara, mas há também uma vasta gama de soluções menos precisas e às vezes muito baratas, que são bastante úteis. A Computação Aproximada busca o conjunto – modelo computacional e *hardware* – que permita a obtenção de resultados computacionais em algum intervalo aceitável no espaço de *trade-off*.

Existem várias abordagens na literatura para transformar um programa convencional em sua versão relaxada (localizada em pontos arbitrários no espaço de *trade-off*): *loop perforation* [12] [13], *dynamic knobs* [14], *approximate memoization* [15] [16], *tile approximation* [16], descarte de computações com alto *overhead* [17] [18] e Sincronização Relaxada [19].

A Sincronização Relaxada é o foco de nosso interesse, pois acreditamos que ela representa um passo importante para tornar a Computação Aproximada prática e acessível aos desenvolvedores de *software* em geral.

2.2 Sincronização Relaxada

A Sincronização Relaxada é uma sub-área da Computação Aproximada que procura reduzir a sobrecarga de sincronização minimizando ou mesmo eliminando completamente os pontos de sincronização. Ela pode ser usada em aplicações onde o resultado produzido pelo programa correto original (com pontos de sincronização inalterados) não é necessariamente único. O resultado gerado é apenas um no *pool* de resultados caracterizado por uma métrica de qualidade. O problema de encontrar uma versão adequada pode ser caracterizado como uma busca no espaço de *trade-off* representado na Figura 2.1.

Como descrito por Renganarayana et al. [4], os principais usos da

sincronização são:

1. Para garantir que todas as *threads* vejam valores consistentes de variáveis compartilhadas, uma vez que uma determinada *thread* possa estar esperando para ler um valor compartilhado atualizado por outra para prosseguir – este é o melhor candidato para a Sincronização Relaxada.

2. Para garantir que as *threads* alcancem vários pontos em sua execução de forma previsível (por exemplo, barreira) – este pode ser relaxado dependendo do contexto.

3. Para garantir a consistência na atualização paralela de estruturas de dados (por exemplo, listas encadeadas), uma vez que a estrutura poderia ser quebrada devido a manipulações simultâneas por diferentes *threads* – difícil de relaxar, pode levar a um erro fatal do programa.

2.3 Trabalhos relacionados

Nesta seção, citamos algumas das pesquisas relevantes sobre sincronização relaxada existentes. Selecionamos aqueles que abordam as principais idéias do campo.

Renganarayana et. al [4] escreveram um trabalho abrangente, que introduz muito bem o tema. Em seguida, Misailovic et. al [5] criaram um sistema chamado Dubstep. Neste trabalho, os autores controlam a qualidade dos resultados formalizando os limites de precisão estatística para a saída do programa relaxado. Rinard [6] apresenta técnicas para obter cálculos paralelos não sincronizados aceitáveis que preservam as principais restrições de consistência da estrutura de dados ao produzir um resultado suficientemente preciso. Finalmente, Carbin e Rinard [7] exploram como o raciocínio relacional pode ajudar os desenvolvedores a verificar propriedades relativas de programas relaxados. Tal raciocínio relacional permite que os desenvolvedores transfiram seu raciocínio sobre o programa original para o relaxado.

2.4 Lidando com condições de corrida

De acordo com o padrão C++ [20], uma condição de corrida ocorre quando duas ou mais *threads* podem acessar um dado compartilhado e tentam alterá-lo ao mesmo tempo. Como o algoritmo de escalonamento de *threads* pode trocar a ordem de execução das *threads* a qualquer momento, não é possível prever a ordem em que elas tentarão acessar o dado compartilhado. Além disso, compiladores podem reordenar as instruções de execução. Portanto, o resultado final das alterações concorrentes no dado depende do algoritmo de escalonamento do sistema. O problema geralmente ocorre quando uma *thread* faz um procedimento chamado *"check-then-act"* (por exemplo, "verifique" se o valor do dado for x, então "aja" para fazer algo que dependa que o valor seja x) e outra *thread* faz algo utilizando o valor entre o *"check"* e o *"act"*.

A condição de corrida é um dos tipos de problemas mais comuns e difíceis de depurar na programação de sistemas concorrentes. Em muitos casos, o comportamento de um programa contendo condições de corrida é indefinido. Isso significa que, em teoria, uma condição de corrida pode levar a qualquer comportamento em tempo de execução. Alguns autores, como Boehm [21], possuem uma visão totalmente restritiva sobre condições de corrida, recriminando seu uso em qualquer tipo de aplicação e objetivo. Em contrapartida, como visto, diversos autores [4, 5, 6, 7] as toleram como uma forma eficiente de aumento de desempenho.

Na prática, durante nossos testes (milhares de execuções em variados ambientes de execução) não observamos nenhum tipo de comportamento grave, como execuções abortadas, apenas a já esperada degradação na qualidade final dos resultados devido a, por exemplo, perda de contagens parciais feitas em regiões sujeitas a condições de corrida. Porém, para aplicações críticas, altamente portáveis e que necessitem da confiabilidade dos resultados, as condições de corrida devem ser removidas e evitadas a todo custo.

2.5 Aprendizagem de Máquina

A aprendizagem de máquina cria modelos capazes de "aprender" com a experiência. Os algoritmos de aprendizagem de máquina usam métodos computacionais para aprender informações diretamente dos dados sem depender de uma equação predeterminada, melhorando adaptativamente seu desempenho à medida que aumenta o número de amostras disponíveis para aprendizagem. A aprendizagem de máquina, como mostrado na Figura 2.2, utiliza dois tipos de técnicas: a aprendizagem supervisionada, que treina um modelo a partir de dados conhecidos de entrada e saída para que ele possa prever resultados futuros, e a aprendizagem não supervisionada, que encontra padrões ocultos ou estruturas intrínsecas a partir de dados de entrada.

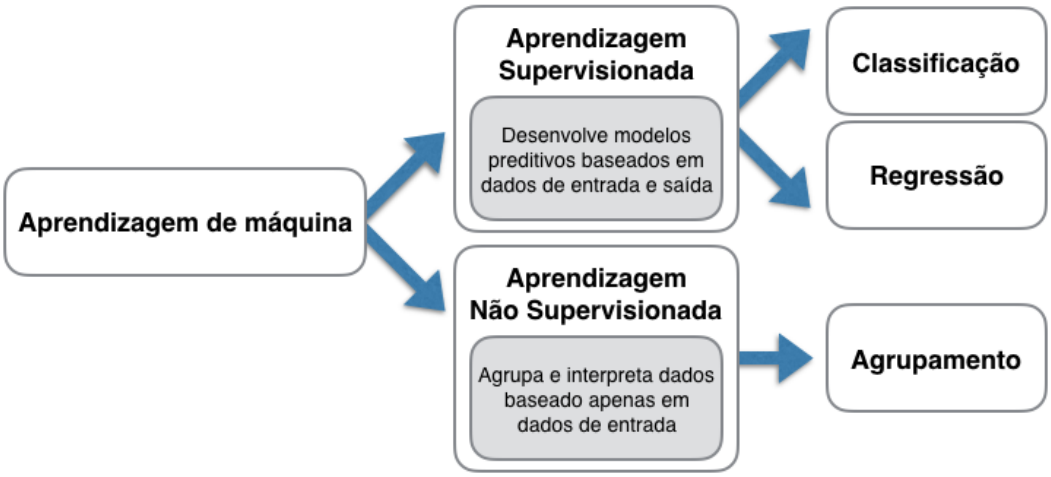

Figura 2.2: Técnicas de aprendizagem de máquina

O objetivo da aprendizagem supervisionada é construir um modelo que faça previsões baseadas em evidências na presença de incerteza. Usando essa técnica, um programa de computador pode "aprender" a partir de observações. Quando exposto a mais observações, o programa melhora seu desempenho preditivo. Especificamente, um algoritmo de aprendizagem supervisionada recebe um conjunto conhecido de dados de entrada e um conjunto conhecido de respostas aos dados de entrada e treina um modelo para gerar previsões

razoáveis para novos dados de entrada, como mostrado na Figura 2.3.

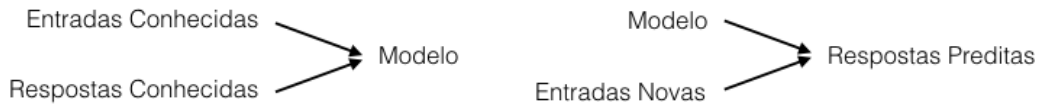

Figura 2.3: Modelo geral da Aprendizagem Supervisionada.

A aprendizagem supervisionada se divide em duas categorias: classificação e regressão. Na classificação, o objetivo é atribuir classes (ou rótulos) a um conjunto finito de dados. Ou seja, as respostas são variáveis categóricas. Suas aplicações incluem filtros de *spam*, sistemas de recomendação de propagandas e reconhecimento de imagens e fala. Prever se um indivíduo terá ou não certa doença dentro de um ano é um problema de classificação, com possíveis classes verdadeiro ou falso. Os algoritmos de classificação usualmente se aplicam a valores de resposta nominais. Na regressão, o objetivo é prever medidas contínuas de uma observação. Ou seja, as variáveis de resposta são números reais. Suas aplicações incluem previsão de preços de ações, consumo de energia ou incidência de eventos.

A aprendizagem não supervisionada encontra padrões escondidos ou estruturas intrínsecas nos dados. Ela é usada para extrair inferências de conjuntos de dados sem respostas rotuladas. A agrupamento é a técnica de aprendizagem não supervisionada mais comum. Ela é usada para análise exploratória de dados, buscando padrões ocultos ou agrupamentos nos dados. As suas aplicações incluem análise de sequência de genes, pesquisa de mercado e reconhecimento de objetos.

Como o nosso interesse neste trabalho é descobrir o melhor fator de relaxamento dentre valores pré-estabelecidos, utilizamos, para todos os exemplos estudados, a coleção de métodos de aprendizagem supervisionada de classificação oferecida pelo *software* Matlab.

Uma importante ferramenta utilizada para avaliar os resultados da previsão de métodos de aprendizagem supervisionada de classificação é a

chamada matriz de confusão. Com ela é possível consultar rapidamente com que frequência as previsões estão sendo feitas com precisão, como veremos a seguir.

Matriz de confusão

A matriz de confusão, também conhecida como matriz de erro, é um tipo específico de tabela que permite a visualização do desempenho do algoritmo de aprendizagem supervisionada de classificação. O nome "matriz de confusão" se deve ao fato de que através da matriz é possível ver se e como o sistema está confundindo as classes (ou seja, se está prevendo classes erradas).

A Figura 2.4 apresenta, como exemplo, uma matriz de confusão. As colunas representam as classes preditas pelo algoritmo de classificação treinado quando apresentada uma entrada nova (nunca vista durante o treinamento). No nosso caso, as classes são os fatores de relaxamento que variam de 100% até 0%. Já as linhas representam as classes que deveriam ser preditas pelo algoritmo de classificação treinado.

Dessa forma, se o algoritmo de classificação treinado obtiver uma acurácia de 100% de acerto a matriz de confusão estará populada somente em sua diagonal principal, ou seja, a classe predita foi sempre a classe real (que deveria ser predita).

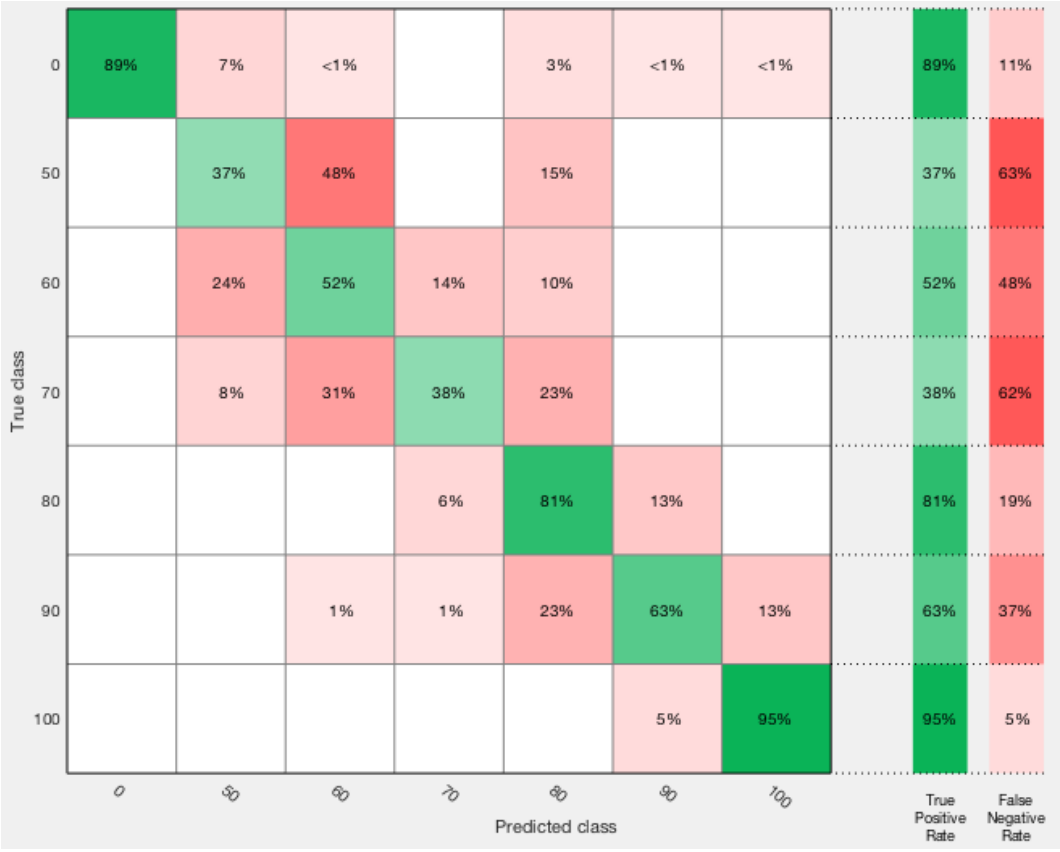

Figura 2.4: Matriz de confusão gerada a partir dos dados obtidos durante um de nossos testes (4 *threads*).

3
Metodologia Utilizada

Este trabalho tem por objetivo avaliar a viabilidade de usar métodos de aprendizagem supervisionada para garantir que a técnica de Sincronização Relaxada forneça resultados dentro de limites aceitáveis de erro para a tripla aplicação/entrada/ambiente de execução.

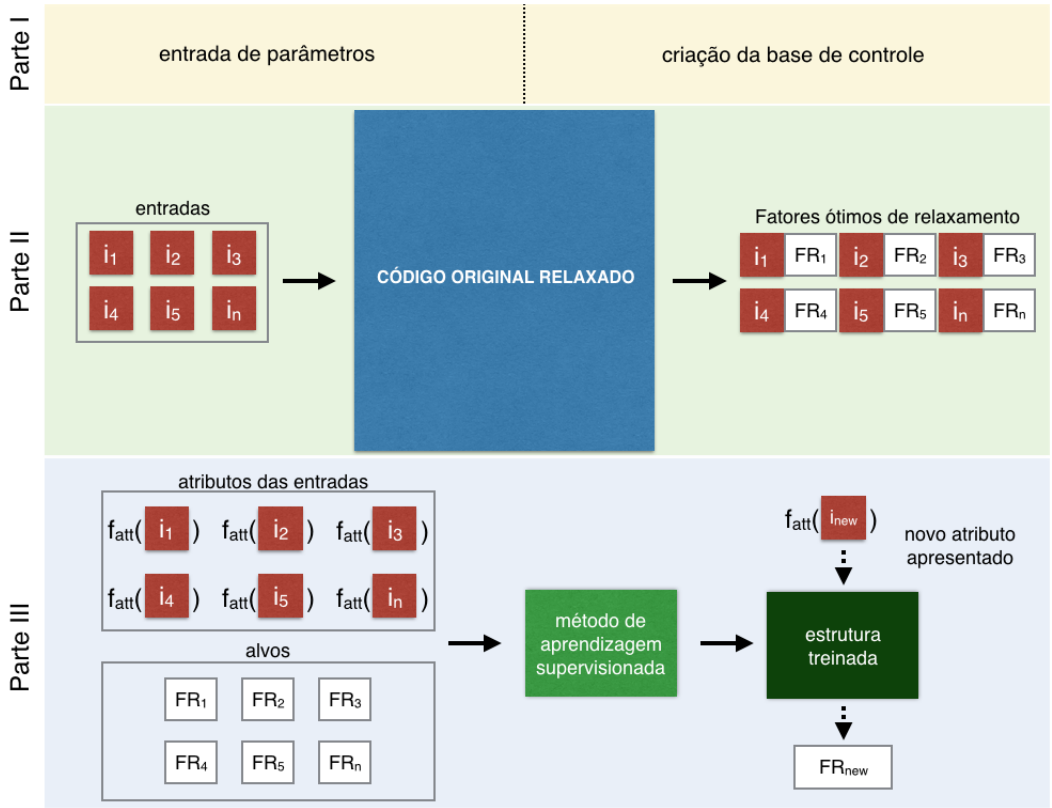

Figura 3.1: Esquemático da metodologia criada.

Para isso, criamos uma metodologia que utiliza alguns dados de entrada para montar casos de testes que, ao serem executados, irão fornecer valores representativos de entrada para a criação e treinamento de métodos de aprendizagem supervisionada. Dessa forma, quando o usuário utilizar a sua

aplicação (no mesmo ambiente de treinamento) com uma nova entrada, o algoritmo de classificação treinado irá sugerir o fator de relaxamento mais adequado. A Figura 3.1 ilustra essa metodologia, que é composta de 3 partes fundamentais que são descritas em detalhe nas seções subsequentes.

3.1 Detalhamento da Metodologia

Parte I - Parâmetros de entrada e criação de base de controle

A Parte I possui duas funções principais. A primeira é receber os parâmetros de entrada necessários para nossa metodologia (estes parâmetros são explicados a seguir). A segunda é criar a chamada base de controle: para cada entrada fornecida, a aplicação original (sem relaxamento) é executada a fim de gerar a base de controle que será utilizada posteriormente para o cálculo do erro. A geração da base de controle se dá a partir de instâncias de treinamento. Essas instâncias podem ser obtidas de duas formas: através de um *script* de geração de instâncias ou pelo fornecimento direto das mesmas.

Os parâmetros de entrada são:

(a) $app_{mod} + entradas \rightarrow$ O código fonte do programa (*app*) modificado. O programa deverá receber como argumento o fator de relaxamento (*RELAX_FACTOR*) e, na porção paralela do código que se desejar relaxar, é preciso criar uma função que retorna se a execução corrente será feita de forma relaxada ou sincronizada. Essa função é exemplificada na Figura 3.2. Também é necessário adicionar a expressão condicional que utiliza essa função, também exemplificada na Figura 3.2. Essa função irá gerar um número aleatório entre 1 e 10 que será utilizado para dizer se a execução da *thread* corrente será ou não relaxada.

É preciso fornecer também as instâncias de entrada a serem utilizadas durante os testes ou um *script* de geração das instâncias de entrada (com os devidos parâmetros).

```
int isRelaxedRun()
{
    int result = rand() % 10 + 1;

    if (result > RELAX_FACTOR)
        return 1;

    return 0;
}
```

```
if (isRelaxedRun())
   // execução sem sincronização
else
   // execução com sincronização
```

Figura 3.2: Adições no código fonte do programa que se deseja relaxar.

(b) f_{att} → Função de classificação da instância de entrada, exemplificada pela Figura 3.3. Essa função (f_{att}) recebe cada um dos atributos de entrada e retorna suas características. Por exemplo, a função de classificação pode receber uma imagem e retornar o número de cores que ela possui. Os atributos de classificação serão utilizados como entrada para a fase de treinamento dos métodos de aprendizagem supervisionada.

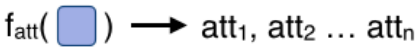

Figura 3.3: Função de classificação de entradas.

(c) f_{error} → A função de cálculo de erro, que recebe como parâmetro a saída do programa e retorna o respectivo erro associado. Essa função tem que ser escolhida para cada aplicação. Para descobrir o erro associado utilizamos a base de controle gerada inicialmente através da comparação de cada instância da base de controle (saída da execução sincronizada) com a respectiva saída proveniente da execução relaxada.

(d) $e_{máx}$ → o erro máximo tolerado.

(e) $FR_{máx}$ *e* FR_{min} → Os valores máximo e mínimo do fator de relaxamento também devem ser fornecidos. Estes valores são utilizados como pontos

de partida e parada do fator de relaxamento máximo. Podem ser por exemplo, $FR_{máx}$ 100% e FR_{min} 10%.

Parte II - Geração dos fatores máximos de relaxamento

O objetivo da Parte II é descobrir o fator máximo de relaxamento admitido pelo programa para cada uma das entradas fornecidas (i_k). Esta parte é detalhada na Figura 3.4. Cada par <entrada, fator de relaxamento corrente> é passado como parâmetro na execução da aplicação modificada (app_{mod}). Toda vez que uma execução termina com uma taxa de erro (e_{out}) menor ou igual à taxa de erro máximo ($e_{máx}$) fornecida, ela é contabilizada como uma execução satisfatória, caso contrário o fator de relaxamento (FR) é diminuído em 10% e o par é executado novamente. A taxa de erro é calculada através da função de erro (f_{error}) que recebe como parâmetro a saída da aplicação modificada e a respectiva saída da aplicação original. Ao final da fase de execução de testes, i.e. todos os pares <entrada, fator máximo de relaxamento> terem sido encontrados, passamos para a Parte III, a seleção de treinamento do método de aprendizagem supervisionada.

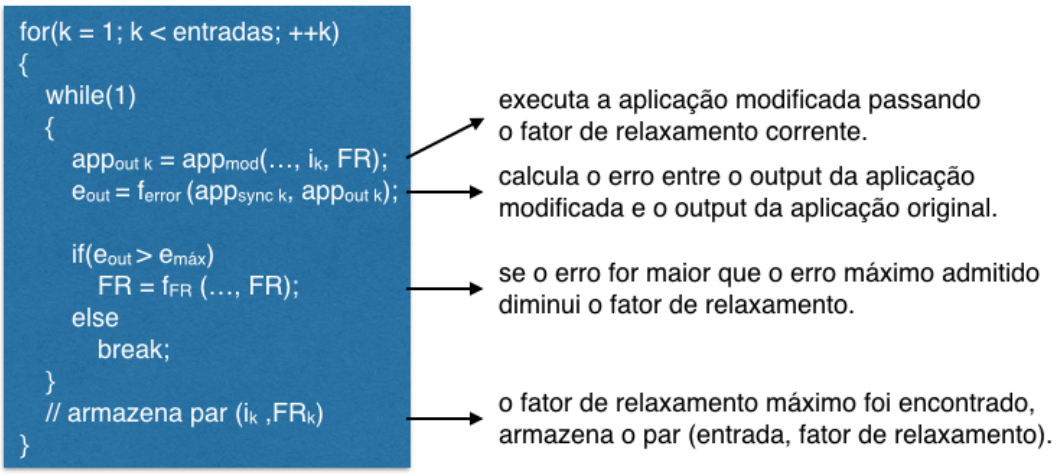

Figura 3.4: Geração dos pares (entrada, fator de relaxamento).

Parte III - Seleção e treinamento do método de aprendizagem supervisionada

Escolher o método de aprendizagem ideal é um processo trabalhoso. Existem dezenas de métodos supervisionados e não supervisionados, e cada um tem uma abordagem diferente para a aprendizagem. Não há melhor método ou um que sirva para todos os problemas. O processo de encontrá-lo se baseia em tentativa e erro. Métodos que utilizam modelos altamente flexíveis tendem ao *overfit* dos dados (levam em conta até pequenas variações que podem ser somente ruídos). Modelos mais simples são mais fáceis de interpretar, mas podem ter menor precisão. Portanto, escolher o método correto requer a ponderação de um benefício em relação a outro, incluindo velocidade, precisão e complexidade do modelo. O teste e o erro são o cerne da aprendizagem de máquina, se uma abordagem ou algoritmo não funcionar, tente outro. Optamos pela utilização do *software* Matlab, que fornece ferramentas para que possamos experimentar uma variedade grande de métodos de aprendizagem de máquina e escolher o melhor.

Neste trabalho utilizamos somente os métodos de classificação (queremos atribuir classes a um conjunto finito de dados), pertencentes a técnica de aprendizagem supervisionada (já que nosso algoritmo de classificação será treinado utilizando dados conhecidos de entrada e de saída). O fluxo de seleção do algoritmo de classificação ideal, usando o Matlab, é mostrado na Figura 3.5.

A extração dos atributos de entrada é feita pela função de classificação da instância de entrada (f_{att}), exemplificada pela Figura 3.3. Essa função recebe cada uma das instâncias de entrada e retorna seus atributos. Por exemplo, a função de classificação pode receber uma imagem e retornar o número de cores que ela possui. Os atributos de saída são os fatores de relaxamento máximo encontrados para cada instância de entrada. A fase seguinte consiste no treinamento de métodos utilizando 22 algoritmos de aprendizagem supervisionada disponíveis no Matlab. O algoritmo de

classificação que exibir a maior taxa de acurácia durante o treinamento é o algoritmo selecionado.

Figura 3.5: Fluxo de seleção do método ideal.

Considerações

Encontramos dificuldades inerentes ao processo de generalização dos passos de nossa metodologia. Dessa forma, a aplicação da metodologia precisou ser feita de forma personalizada e praticamente artesanal para cada tipo de aplicação estudada.

A maior dificuldade foi encontrar os atributos pertinentes a instância de entrada e, portanto, a função de classificação. Essa pode ser uma tarefa árdua, principalmente se existirem muitos atributos disponíveis como, na busca de atributos pertinentes no caso da instância de entrada ser uma imagem. Imagens possuem, tipicamente, diversos atributos e, dependendo do algoritmo, uns podem ser mais ou menos pertinentes durante o treinamento dos métodos de aprendizagem supervisionada). Em casos como este é preciso realizar testes, com grupos de atributos, a fim de verificar quais deles apresentam o melhor

resultado durante o treinamento e validação dos métodos de aprendizagem supervisionada.

Outros exemplo de dificuldades foram encontrar a função de erro, que quantifica o quão perto o resultado relaxado se encontra do resultado original e modificar a aplicação original para introduzir a fração relaxada.

4
Avaliação das Aplicações

Para testar a viabilidade de nossa proposta, precisamos de aplicações que satisfaçam os seguintes requisitos:

1. Os pontos de sincronização a serem relaxados não pertençam à classe de garantia de consistência de estruturas de dados (como visto na seção 2.1).

2. A qualidade de seus resultados possa ser de alguma forma quantificada.

3. A aplicação deve ser tolerante a erros.

Procuramos então, em conjuntos de *benchmarks* disponíveis, aplicações que possuam essas características. *Benchmarks* são, tipicamente, programas considerados representativos para testes de diversos propósitos. O uso de aplicações retiradas de *benchmarks* conhecidos nos ajuda a garantir que as aplicações estudadas abrangem um espectro amplo e diminui a chance de nossos resultados serem válidos apenas para um conjunto muito específico de programas.

Como descrito por Renganarayana et al. [4], alguns *benchmarks* importantes pertencentes a conjuntos de aplicações como *STAMP* (*Stanford Transactional Applications for Multi-Processing*) [22] e *Graph500* [1], pertencem a classe de aplicação que possui tais características.

Testamos primeiramente os *benchmarks* do conjunto de aplicações *STAMP*, onde apenas a aplicação *labyrinth* possuía as características procuradas. Nas aplicações *bayes*, *genome*, *intruder* e *vacation*, os pontos de sincronização fazem parte da classe de garantia de consistência de

[1] *The Graph 500 benchmark*, http://www.graph500.org.

estrutura de dados, e por isso não podem ser retirados. Na aplicação *yada*, a maioria dos pontos de sincronização servem para proteger a alocação e desalocação de memória. Consequentemente, não pertencem a classe de pontos de sincronização possíveis de serem removidos. Na aplicação *ssca2*, cada *thread* adiciona nós na estrutura de dados de um grafo de forma paralela e utiliza transações para proteger o acesso concorrente a *arrays*. Estes acessos ocorrem de forma infrequente e o tempo perdido nessas operações é relativamente pequeno. Dessa forma, os benefícios de relaxar a sincronização só aparecem a partir de 4 *threads*. Apesar disso, a saída deste *benchmark* não admite uma quantificação da qualidade do resultado final (a saída é binária, conseguiu ou não gerar uma representação final de um grafo). Logo, não satisfaz os requisitos. Finalmente, a aplicação *labyrinth* é um excelente candidato para a nossa metodologia, já que possui as três características principais.

Em seguida, realizamos testes utilizando o algoritmo *Breadth-First Search* presente no *benchmark Graph500*. Por fim analisamos em detalhes a aplicação de nossa metodologia no algoritmo de *K-Means* aplicado à segmentação de cores em vídeos.

Variação do fator de relaxamento por aplicação

Cada aplicação que utilizamos para avaliar a possibilidade de aliar métodos de aprendizagem supervisionada com o relaxamento de sincronização, apresenta formas particulares de variação do fator de relaxamento. Este fato é sumarizado na Figura 4.1. De forma geral fixamos o fator de relaxamento para cada entrada do algoritmo que estamos relaxando. A entrada do algoritmo *Labyrinth* é o *grid*, já o algoritmo *Breadth-First Search* presente no *benchmark Graph500* apresenta como entrada o grafo. Por fim, apesar de querermos aplicar o algoritmo de K-Means em vídeos, o algoritmo recebe como entrada apenas um *frame*. Como um vídeo é composto de diversos *frames* temos por consequência a variação do fator de relaxamento para cada *frame* do vídeo que

estamos aplicando o algoritmo.

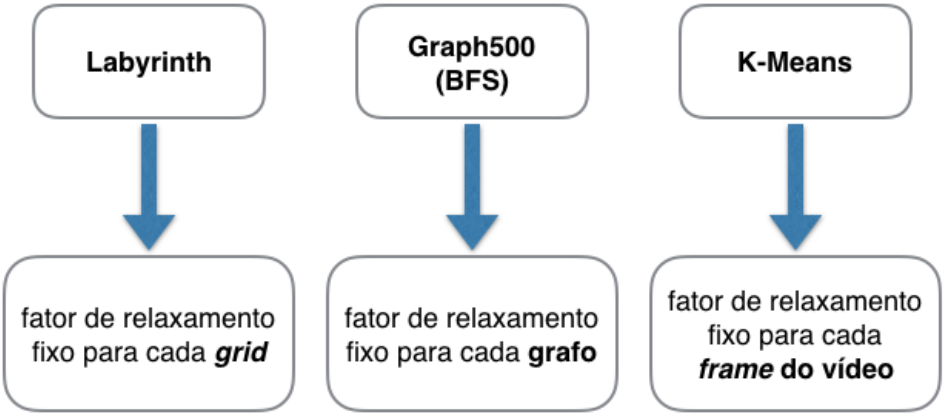

Figura 4.1: Variação do fator de relaxamento por aplicação.

Aquisição de parâmetros

Precisamos obter os atributos pertinentes a instância de entrada da aplicação em questão (tanto no momento de encontrar a função que irá gerá-los como na hora de executar a função). Para as aplicações Labyrinth e Graph500 (BFS) foi intuitivo encontrar os atributos para a classificação das instâncias. Já para a aplicação K-Means precisamos, através de experimentação, testar a utilização de diversos atributos diferentes até descobrir o que fornecia o melhor resultado final.

Custos computacionais

A nossa metodologia apresenta custos computacionais devido ao treinamento dos métodos de aprendizagem supervisionada e a utilização do método que obteve o melhor desempenho. Neste caso, para todas as aplicações o custo foi o mesmo, consistindo no tempo de treinamento de todos os métodos de aprendizagem supervisionada disponíveis no *software* Matlab (alguns segundos). Já o custo da utilização do método de aprendizagem treinado que obteve o melhor desempenho pode ser desprezado já que consiste

apenas na aplicação dos atributos encontrados pela função de classificação na equação do método de aprendizagem.

Definição de ganho

Definimos o ganho obtido pela nossa metodologia como sendo o tempo de execução da aplicação em sua versão paralela e original dividido pelo tempo de execução da aplicação que utiliza o Relaxamento de Sincronização. Como visto na Figura 4.1, cada aplicação que utilizamos, apresenta formas particulares de variação do fator de relaxamento e, dessa forma, a forma de cálculo do ganho também foi particular, como mostrado na Figura 4.2.

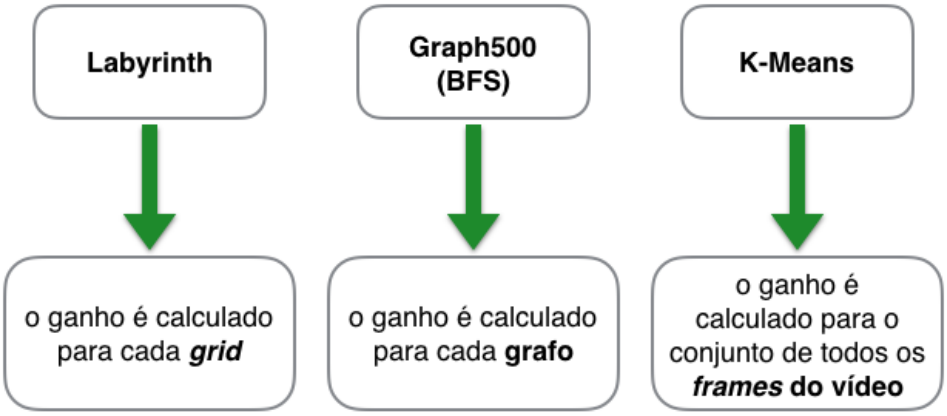

Figura 4.2: Cálculo do ganho por aplicação.

Ambiente de execução

Executamos todos os nossos testes em 4 CPUs Intel Xeon E5-2640 v4 2.40GHz, totalizando 40 núcleos. Variamos o número de *threads* em 2, 4, 8, 16 e 32.

Como o número de *threads* foi uma variável em nossos resultados, o utilizamos como um atributo no treinamento dos métodos de aprendizagem supervisionada. Caso optássemos por usar somente um valor fixo de número de *threads* esse atributo poderia ser suprimido na hora do treinamento.

4.1 Labyrinth

O *benchmark Labyrinth* apresenta uma implementação do algoritmo de Lee [9]. Este algoritmo é usualmente aplicado em roteamento de circuitos, que é o processo automatizado de produção da interconexão de componentes eletrônicos. Estes componentes podem ser transistores em um circuito integrado, elementos lógicos em uma FPGA ou até mesmo os que compõem a placa mãe de um PC.

Em sua forma mais simples, o problema de roteamento de circuitos pode ser reduzido a conectar pontos em um *grid* bidimensional, o qual representa a arquitetura do circuito. O algoritmo de Lee garante encontrar a interconexão mais curta entre dois pontos utilizando a técnica explicada a seguir, ilustrada na Figura 4.3.

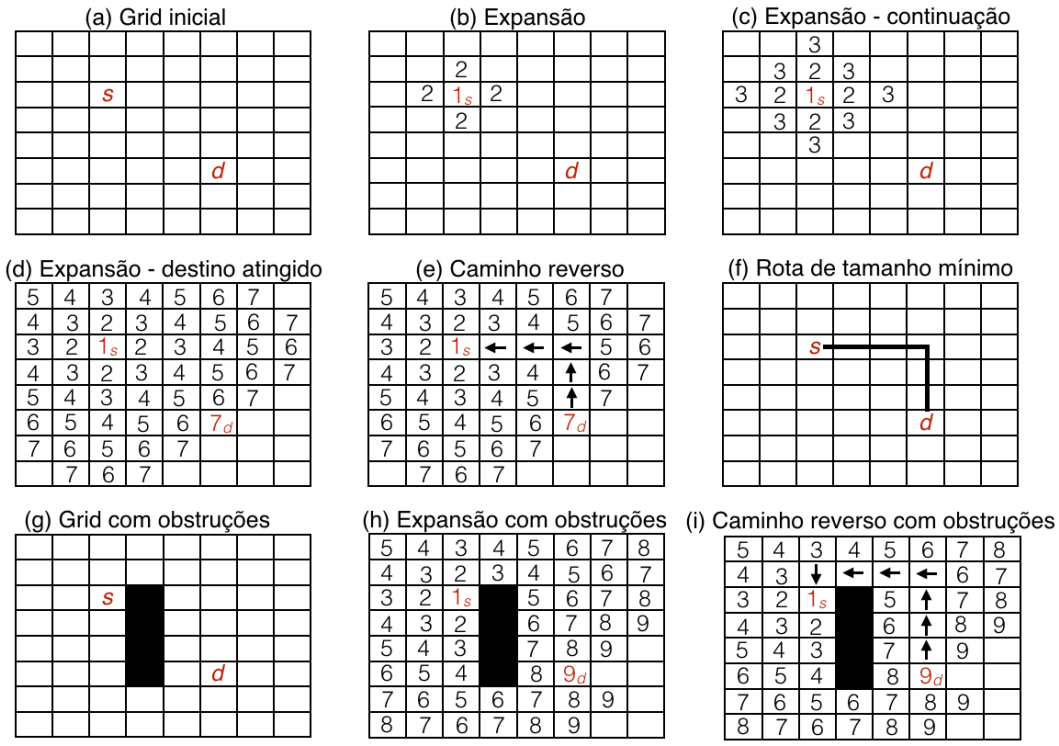

Figura 4.3: Fases do algoritmo de Lee.

Começando a partir do ponto inicial *s*, os pontos do *grid* vão sendo enumerados através da expansão nas direções vertical e horizontal até que o

ponto *d* seja atingido (veja a Figura 4.3(a)-(d)). Em cada estágio da expansão todo ponto do *grid* que irá se expandir marca seu vizinho não enumerado com o incremento de seu valor. Uma vez que o ponto de destino *d* tenha sido atingido, um caminho reverso é traçado até o ponto de início *s*, seguindo qualquer sequência decrescente de pontos do *grid* (veja a Figura 4.3(e)). O processo de achar o caminho de volta pode ser implementado de forma particular dependendo da estratégia do programador. Por exemplo, pode ser desejável minimizar mudanças na direção do caminho.

A partir do momento em que uma rota for determinada, os pontos do *grid* pertencentes a ela estarão ocupados e não poderão ser utilizados por outras rotas. As expansões devem portanto seguir apenas em pontos livres do *grid*. A Figura 4.3(g)-(i) mostra um exemplo disso e como o processo de traçar o caminho reverso pode achar, ainda assim, o caminho de tamanho mínimo entre os pontos *s* e *d*.

Utilizamos em nosso exemplo uma variação do algoritmo de Lee, similar ao implementado por Watson et al. [23], denominado *labyrinth*, um *benchmark* pertencente ao conjunto de aplicações *STAMP*. A estrutura de dados principal é um *grid* n-dimensional que representa um labirinto. Em sua versão paralela, cada *thread* possui um ponto inicial e um final que devem ser conectados através de um caminho de pontos adjacentes no *grid*. O cálculo da rota e sua adição no *grid* global são protegidos por uma transação única. Um conflito ocorre quando duas *threads* possuem caminhos que se interceptam. Para reduzir a chance de conflitos, a técnica de privatização descrita por Watson et al. [23] é utilizada. Nela, cada *thread* possui um *grid* local utilizado para o cálculo de sua rota. Quando uma *thread* vai adicionar sua rota ao *grid* global, ela a valida através da leitura de todos os pontos do *grid* global que pertencem a sua rota. Se a validação falhar, a transação é abortada e o processo se repete, começando com uma nova cópia do *grid* global.

De modo geral, as transações do algoritmo são muito demoradas, pois

realizam operações de leitura/escrita em conjuntos grandes de dados. A Sincronização Relaxada pode, portanto, reduzir o tempo de execução total do algoritmo, caso a aplicação do usuário tolere um percentual de perda de rotas encontradas. A seguir exemplificaremos todos os passos da aplicação de nossa metodologia para o algoritmo de Lee.

4.1.1 Parâmetros de entrada – Labyrinth

A seguir listamos os parâmetros que foram fornecidos como entrada.

(a) *Script* de geração das instâncias de entrada. Para essa aplicação utilizamos um *script* em python [Apêndice A] para a geração das instâncias de entrada. Os parâmetros utilizados pelo nossa metodologia, em conjunto com o *script*, são apresentados na Tabela 4.1. Começamos com um *grid* de dimensões (16, 16, 2), somando a cada iteração 8 nas dimensões x e y, ex: (24, 24, 2), (32, 32, 2) ... (256, 256, 2). Cada um destes *grids* irá tentar encontrar um número de caminhos múltiplos de 8, ex: 8, 16, 24 ... 256. Dessa forma terminamos com 992 instâncias de entrada.

START_DIMENSION	16
DIMENSION_SUM	8
END_DIMENSION	256
Z_DIMENSION	2
START_NUM_PATHS	8
PATHS_SUM	8
MAX_PATHS	256

Tabela 4.1: Parâmetros utilizados pelo *script* de geração das instâncias de entrada.

(b) Atributos que classifiquem cada instância de entrada. Os atributos utilizados são: dimensão do *grid* (int) e o número de caminhos (int).

(c) Código fonte do programa com as devidas adições de chamadas (apresentado na Figura 4.4).

```
int isRelaxedRun()
{
    int result = rand() % 10 + 1;

    if (result > RELAX_FACTOR)
        return 1;

    return 0;
}
```

```
void TMgrid_addPath (TM_ARGDECL  grid_t* gridPtr, vector_t* pointVectorPtr)
{
    long n = vector_getSize(pointVectorPtr);

    for (long i = 1; i < (n-1); ++i)
    {
        long* gridPointPtr = (long*)vector_at(pointVectorPtr, i);
        long value = (long)TM_SHARED_READ(*gridPointPtr);

        if (value != GRID_POINT_EMPTY)
            TM_RESTART();

        if(isRelaxedRun())
            *gridPointPtr = GRID_POINT_FULL;
        else
            TM_SHARED_WRITE(*gridPointPtr, GRID_POINT_FULL);
    }
}
```

Figura 4.4: Trecho do código fonte do algoritmo de Lee com o relaxamento dinâmico adicionado.

A função *isRelaxedRun* retorna 1 caso gere um número aleatório maior que RELAX_FACTOR e 0 caso contrário. A função gera números aleatórios entre [1 ... 10] e o RELAX_FACTOR varia entre [0 ... 5], 0 significa 100% relaxado e 5 significa 50% relaxado.

O ponto de sincronização está na fase de verificação, que valida os caminhos encontrados pelas *threads*, para garantir que a condição de aceitabilidade (caminhos não podem se sobrepor) seja alcançada. A função *TMgrid_addPath* recebe como argumento o *grid* global (*gridPtr*) e o caminho a ser adicionado (*pointVectorPtr*). A função itera sobre o vetor que contém o caminho a ser adicionado e verifica se cada posição

do vetor (posição do caminho a ser adicionado) está ou não ocupada. Em caso negativo escreve no *grid* global que a posição está ocupada.

No final, o *benchmark* possui uma fase de verificação para validar os caminhos escritos no *grid* global, para assegurar que o critério de aceitabilidade (caminhos não se cruzarem ou caminhos sem buracos), foi alcançado.

(d) Função de cálculo de erro que recebe como parâmetro a saída do programa e retorna o respectivo erro associado e o erro máximo tolerado. Para essa aplicação o erro é calculado através da equação:

$$\%erro = \frac{|num\ max\ rotas - num\ rotas\ encontradas|}{num\ max\ rotas} \times 100 \qquad (4\text{-}1)$$

Para o erro máximo tolerado escolhemos 10%.

4.1.2 Montagem dos casos de teste – Labyrinth

Através dos parâmetros mostrados na Tabela 4.1 foram montados diversos casos de teste partindo de *grids* de tamanho (16, 16, 2) até (256, 256, 2). Para cada tamanho de *grid* variamos o número de caminhos a serem encontrados (entre 8 e 256). Também utilizamos como parâmetro o número de *threads* utilizado em cada execução.

4.1.3 Geração dos fatores máximos de relaxamento – Labyrinth

Executamos testes para cada conjunto <dimensão, número de caminhos, número de threads, fator de relaxamento>. Toda vez que uma execução termina tendo encontrado um número de caminhos maior que uma dada porcentagem do número de caminhos totais, ela é contabilizada como uma execução satisfatória. Caso haja um número maior que um limiar de execuções

não satisfatórias em uma tripla, o fator de relaxamento é diminuído e os testes recomeçam, caso contrário o fator de relaxamento máximo foi encontrado.

As Figuras 4.5 e 4.6 mostram os fatores máximos de relaxamento admitidos, encontrados para um subgrupo de exemplificação (64 amostras) utilizando 2 e 4 *threads*, respectivamente. Para cada célula, vemos o fator de relaxamento máximo admitido para o par <dimensão do grid, número de caminhos>. Células cinzas representam a impossibilidade de relaxamento para determinado par <dimensão do grid, número de caminhos>.

Conforme esperado, quando o erro máximo admitido é aumentado maior se torna a possibilidade de relaxamento. Além disso, quando utilizamos 4 *threads* ao invés de 2 a possibilidade de relaxamento diminui. Isso se deve ao fato de que com mais *threads*, maior a chance de ocorrerem condições de corrida perante a remoção dos pontos de sincronização.

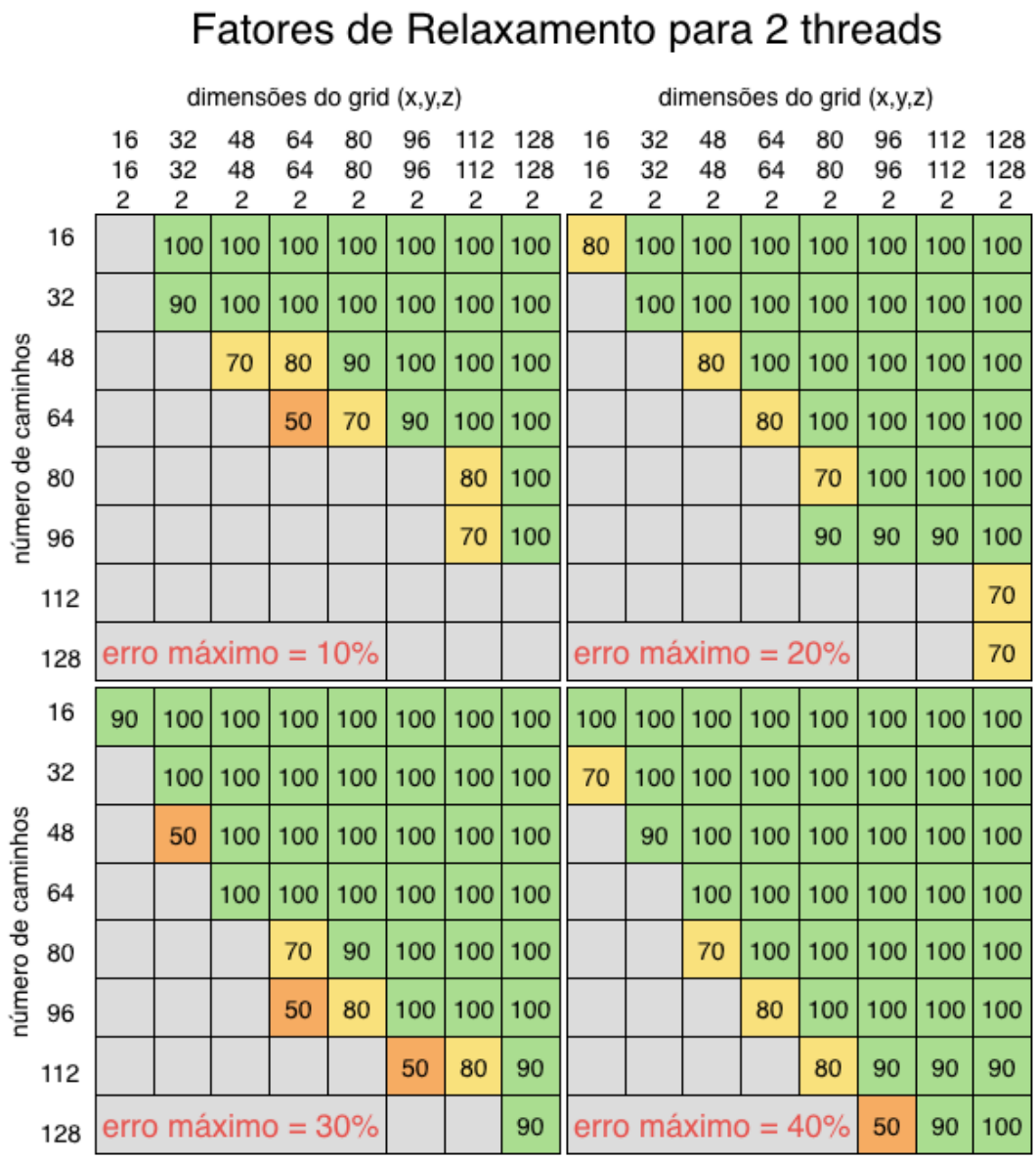

Figura 4.5: Fatores máximos de relaxamento, utilizando 2 *threads*, para o algoritmo de Lee em cada dupla <dimensões do *grid*, # de caminhos> para erros máximos de 10%, 20%, 30% e 40%.

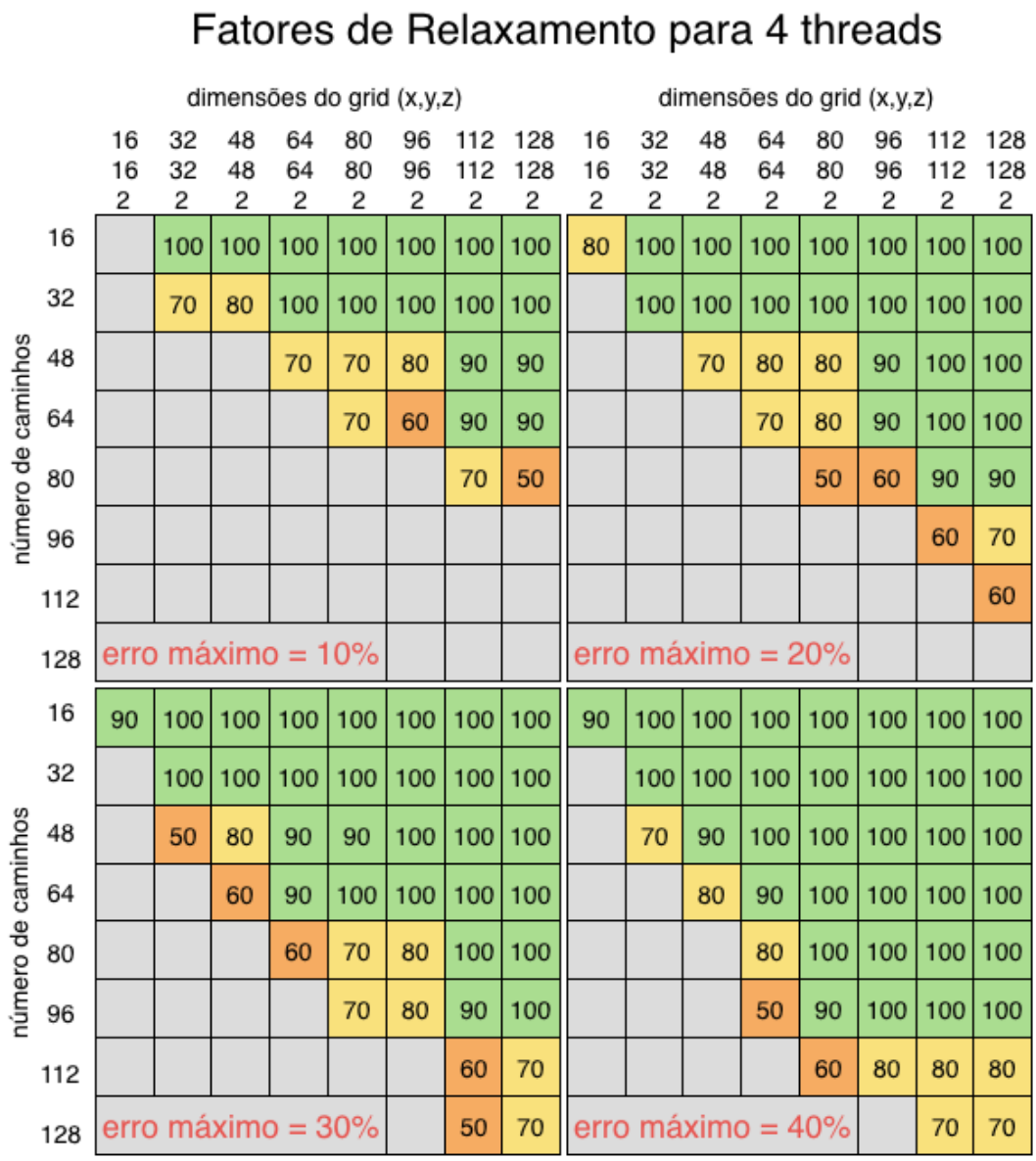

Figura 4.6: Fatores máximos de relaxamento, utilizando 4 *threads*, para o algoritmo de Lee em cada dupla <dimensões do *grid*, # de caminhos> para erros máximos de 10%, 20%, 30% e 40%.

A Figura 4.7 apresenta os fatores máximos de relaxamento para todas as 992 amostras geradas. Como é possível observar, existe uma grande quantidade de amostras que não permitem que o relaxamento de sincronização seja aplicado (para manter o erro máximo estipulado). Essa barreira vai sendo quebrada gradativamente na medida em que a dimensão do *grid* aumenta,

como mostrado nas Figuras 4.5 e 4.6.

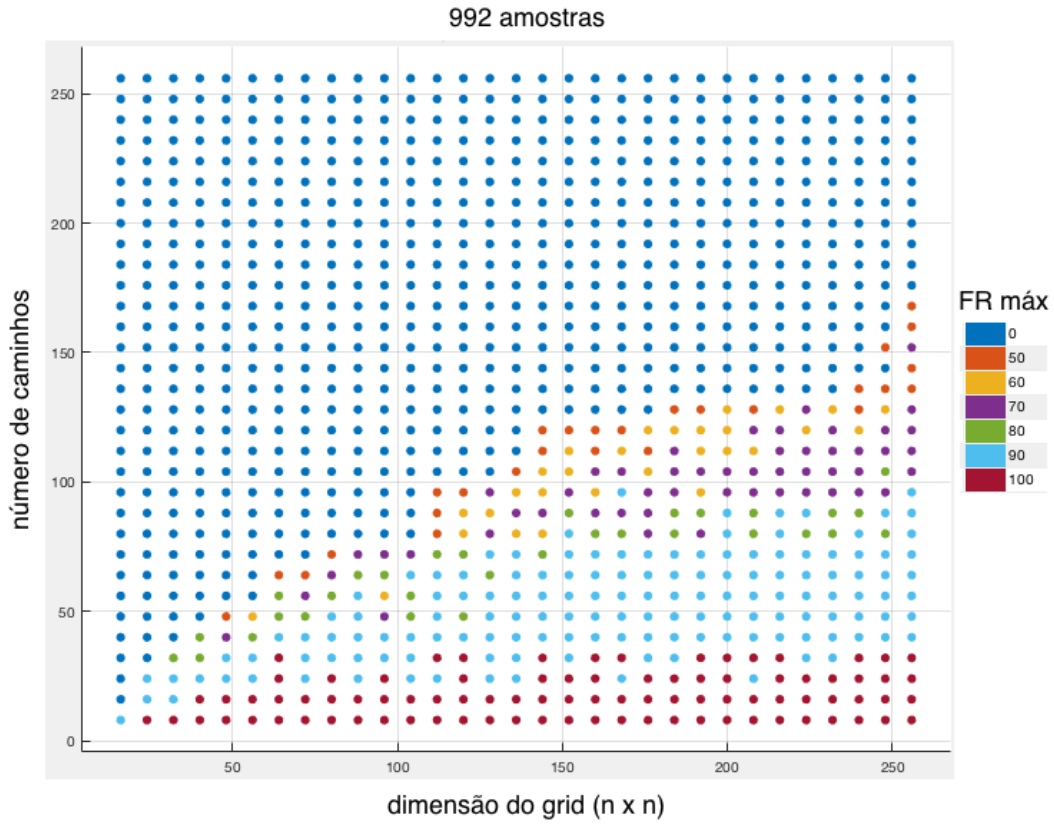

Figura 4.7: As 992 amostras (dimensão, número de caminhos, fator de relaxamento máximo) usadas como entrada no treinamento dos métodos de aprendizagem supervisionada (8 *threads*).

O próximo passo, após termos adquiridos os fatores máximos de relaxamento, é a seleção e treinamento do método de aprendizagem supervisionada mais adequado para a nossa aplicação. Cada algoritmo de classificação treinado representa uma generalização para um conjunto de triplas <dimensão, número de caminhos, fator de relaxamento> gerado a partir de um valor fixo de erro máximo.

4.1.4 Seleção e treinamento do método de aprendizagem supervisionada – Labyrinth

Primeiramente selecionamos os atributos de entrada que serão utilizados durante o treinamento dos algoritmos de classificação (aprendizagem supervisionada). No caso do *Labyrinth*, utilizamos como entrada a dimensão do *grid* (d), o número de caminhos a serem encontrados ($\#$) e o número de *threads*. Não tivemos dificuldade de selecionar os atributos pertinentes da instância de entrada dessa aplicação já que o grid só apresenta estes dois atributos. Como saída utilizamos o fator de relaxamento máximo (fr), encontrado no passo anterior de nossa metodologia. Após a execução de todos os métodos de classificação obtemos os resultados mostrados na Figura 4.8.

1.1 Tree Last change: Complex Tree	Accuracy: 85.7% 2/2 features	1.12 KNN Last change: Fine KNN	Accuracy: 81.5% 2/2 features
1.2 Tree Last change: Medium Tree	Accuracy: 84.6% 2/2 features	1.13 KNN Last change: Medium KNN	Accuracy: 86.5% 2/2 features
1.3 Tree Last change: Simple Tree	Accuracy: 80.2% 2/2 features	1.14 KNN Last change: Coarse KNN	Accuracy: 77.1% 2/2 features
1.4 Linear Discriminant Last change: Linear Discriminant	Accuracy: 78.1% 2/2 features	1.15 KNN Last change: Cosine KNN	Accuracy: 73.6% 2/2 features
1.5 Quadratic Discrimi... Last change: Quadratic Discriminant	Accuracy: 86.3% 2/2 features	1.16 KNN Last change: Cubic KNN	Accuracy: 86.4% 2/2 features
1.6 SVM Last change: Linear SVM	Accuracy: 87.0% 2/2 features	1.17 KNN Last change: Weighted KNN	Accuracy: 86.7% 2/2 features
1.7 SVM Last change: Quadratic SVM	Accuracy: **88.5%** 2/2 features	1.18 Ensemble Last change: Boosted Trees	Accuracy: 85.6% 2/2 features
1.8 SVM Last change: Cubic SVM	Accuracy: 86.9% 2/2 features	1.19 Ensemble Last change: Bagged Trees	Accuracy: 86.3% 2/2 features
1.9 SVM Last change: Fine Gaussian SVM	Accuracy: 86.4% 2/2 features	1.20 Ensemble Last change: Subspace Discriminant	Accuracy: 69.1% 2/2 features
1.10 SVM Last change: Medium Gaussian SVM	Accuracy: 87.4% 2/2 features	1.21 Ensemble Last change: Subspace KNN	Accuracy: 63.3% 2/2 features
1.11 SVM Last change: Coarse Gaussian SVM	Accuracy: 86.5% 2/2 features	1.22 Ensemble Last change: RUSBoosted Trees	Accuracy: 80.4% 2/2 features

Figura 4.8: Resultados obtidos após o treinamento de todos os métodos de classificação disponíveis no Matlab (4 *threads*).

Como é possível observar o método que obteve o melhor treinamento para a nossa aplicação foi o *SVM* que utiliza o kernel *Quadratic SVM*. A Figura

4.9 mostra a matriz de confusão obtida após o seu treinamento. A diagonal principal contém a porcentagem de amostras que foram corretamente preditas. Observe pela Figura 4.9 que a maior parte dos elementos foram preditos ou na diagonal principal ou na diagonal imediatamente superior e inferior da matriz. Dessa forma, o modelo gerado se mostrou robusto durante o processo de generalização.

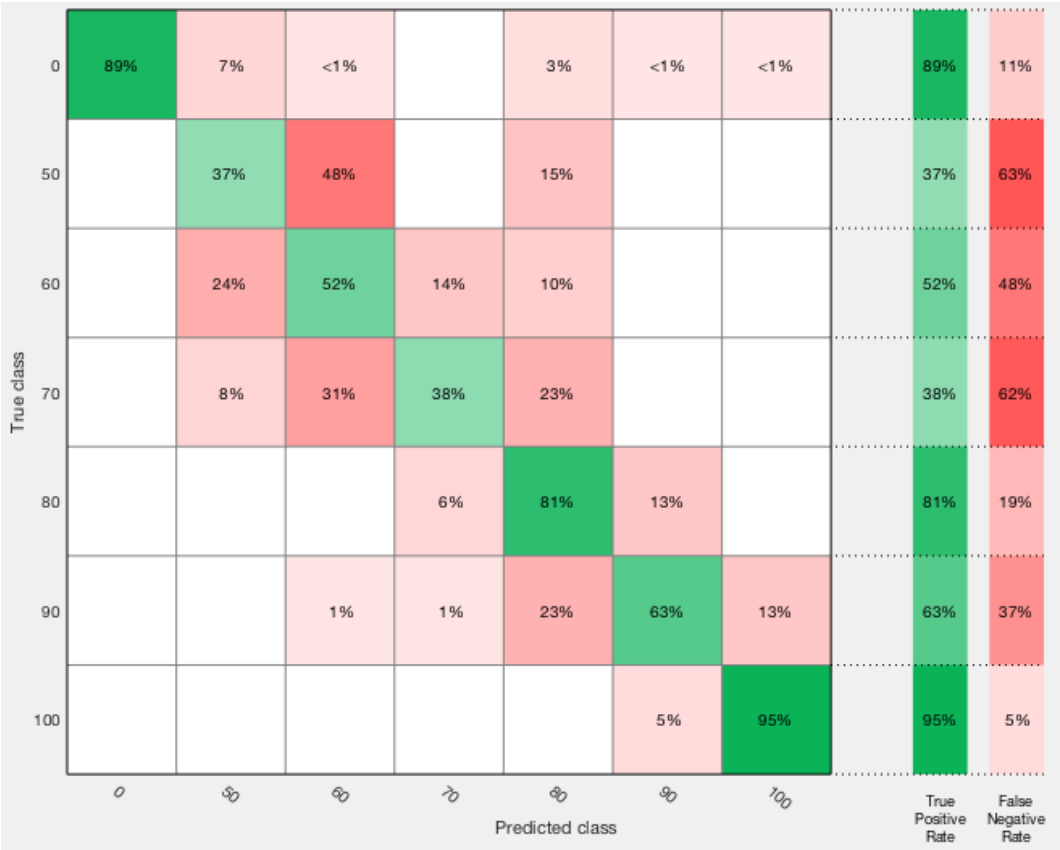

Figura 4.9: Matriz de confusão gerada a partir dos dados obtidos durante os testes com o kernel *Quadratic SVM* (4 *threads*).

4.1.5 Resultados – Labyrinth

Para exemplificar os resultados que podem ser obtidos com a aplicação de nossa metodologia no *benchmark Labyrinth*, apresentamos na Figura 4.10 os valores dos tempos de execução e os consequentes ganhos obtidos para

diferentes fatores de relaxamento utilizando 2, 4, 8, 16 e 32 *threads* em um *grid* de tamanho (256, 256, 2) onde desejamos encontrar 128 caminhos. Em todas as execuções utilizamos 4 x CPU: Intel Xeon E5-2640 v4 2.40GHz (40 núcleos).

Como podemos observar, o ganho máximo para este exemplo foi entre 1,5x e 1,6x para 100% de relaxamento. Para outros fatores de relaxamento, obtivemos um ganho de até 1,3x.

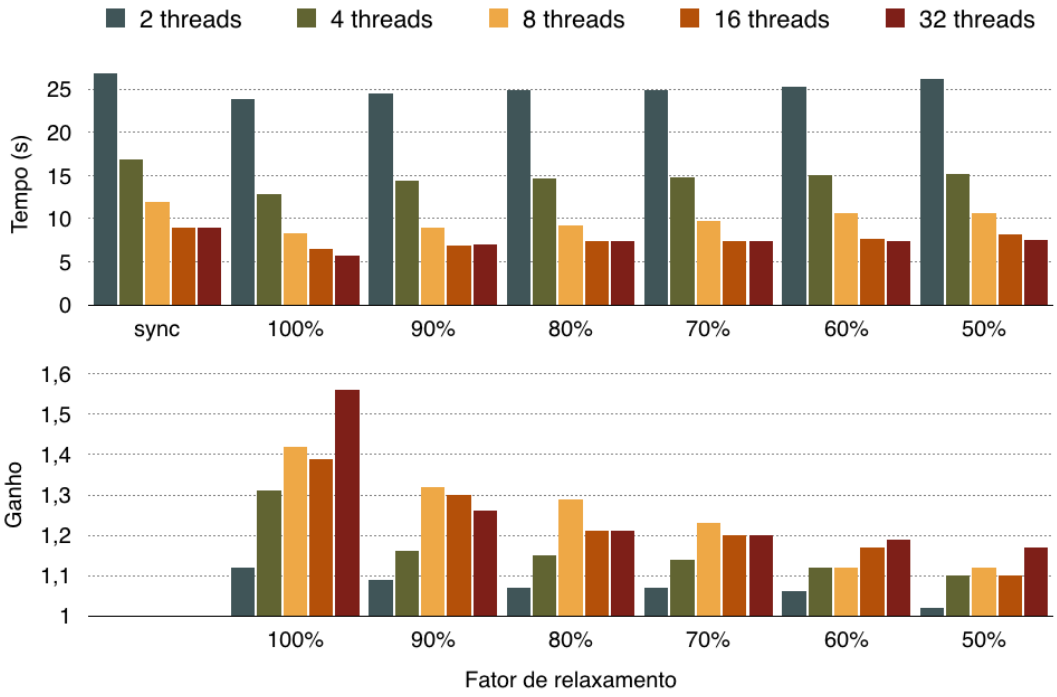

Figura 4.10: Tempo de execução e ganho (em relação a versão paralela e sincronizada) em diferentes fatores de relaxamento e números de *threads*. 4 x CPU: Intel Xeon E5-2640 v4 2.40GHz (40 núcleos).

4.2 Graph500 - Breadth-First Search

O *Graph500* estabelece um *benchmark* de larga escala para aplicações que utilizam um grande volume de dados e são processadas em super computadores. Uma grande parte dessa classe de aplicações utiliza grafos como a estrutura de dados principal.

Para aplicar a nossa metodologia utilizamos a implementação paralela (OpenMP) do algoritmo *Breadth-First Search* (BFS), contido no *benchmark Graph500*, que opera sobre grafos do tipo *scale-free*. Nestes grafos, ilustrados na Figura 4.11, a maioria dos nós (vértices) tem poucas ligações, enquanto alguns poucos nós apresentam um elevado número de ligações.

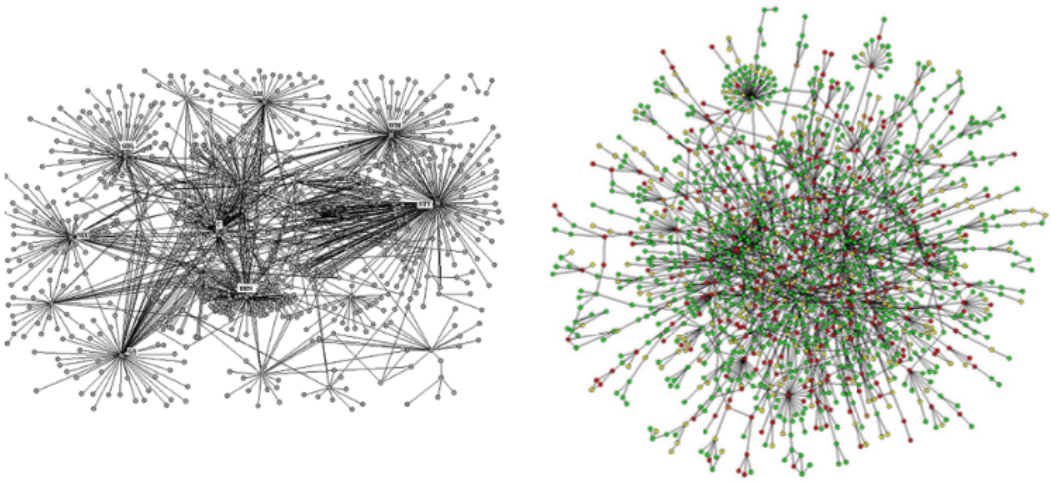

Figura 4.11: Exemplos de grafos *scale-free*.

A ideia central do algoritmo BFS é visitar, a partir de um nó inicial, todos os nós imediatamente vizinhos a ele antes de prosseguir para o próximo nível de nós vizinhos. Assim é possível obter o grupo de nós relacionado ao nó de início. Uma possível aplicação deste algoritmo é a seleção de uma lista de endereços eletrônicos relacionados, a partir de um nó (endereço) origem, para oferecer um determinado produto a potenciais compradores.

Em nosso estudo utilizamos como critério de qualidade a razão entre o número de nós que conseguimos alcançar a partir do nó inicial (tamanho da lista de endereços gerada) e o número alcançado pela versão 100% sincronizada do algoritmo.

A seguir apresentamos em detalhes todos os passos de nossa metodologia para o BFS/Graph500.

4.2.1 Parâmetros de entrada – Graph500

Listamos aqui os parâmetros que foram fornecidos como entrada.

(a) Instâncias de entrada. Utilizamos a variação de dois parâmetros do algoritmo BFS, a escala do grafo e o seu fator de aresta:

- Escala (E): $log_2(N)$, onde N é o número de nós. E = 10, 11, 12 ... 22.
- Fator de aresta (FA): $\frac{numero\ de\ arestas}{numero\ de\ nos}$.
- Número de threads (T). T = 2, 4, 8, 16, 32.

(b) Atributos que classifiquem cada instância de entrada. Os atributos utilizados são exatamente a escala (int) e o fator de aresta (int).

(c) Código fonte do programa com as devidas adições de chamadas, apresentado na Figura 4.12. A função *isRelaxedRun* retorna 1 caso gere um número aleatório maior que RELAX_FACTOR e 0 caso contrário. A função gera números aleatórios entre [1 ... 10] e o RELAX_FACTOR varia no intervalo [0 ... 5], onde 0 significa 100% relaxado e 5, 50%.

Para cada execução, a aplicação começa com um nó origem aleatório e busca todos os nós alcançáveis a partir dele. A implementação paralela atualiza a lista de nós alcançados atomicamente. Esta atualização atômica foi substituída por uma atualização não-atômica na versão

relaxada. Isso pode fazer com que a lista final tenha menos vértices visitados em comparação com a versão 100% sincronizada do algoritmo, devido a possibilidade de existência de condições de corrida.

```
int isRelaxedRun()
{
    int result = rand() % 10 + 1;

    if (result > RELAX_FACTOR)
        return 1;

    return 0;
}
```

```
if (isRelaxedRun())
    output = int64_cas_RELAX (&bfs_tree[j], -1, v);
else
    output = int64_cas (&bfs_tree[j], -1, v);
```

```
int int64_cas(int64_t* p, int64_t oldval, int64_t newval)
{
    int out = 0;
    OMP("omp critical (CAS)") {
        int64_t v = *p;
        if (v == oldval) {
            *p = newval;
            out = 1;
        }
    }
    OMP("omp flush (p)");
    return out;
}

int int64_cas_RELAX(int64_t* p, int64_t oldval, int64_t newval)
{
    int64_t v = *p;
    int out = 0;
    if (v == oldval) {
        *p = newval;
        out = 1;
    }
    return out;
}
```

Figura 4.12: Trecho do código fonte do algoritmo BFS exibindo a adição do relaxamento dinâmico.

(d) Função de cálculo de erro que recebe como parâmetro a saída do programa e retorna o respectivo erro associado e o erro máximo tolerado. Para essa aplicação o erro é calculado através da equação 4-2. Para o erro máximo tolerado escolhemos 10%.

$$\%erro = \frac{|num\ max\ nos - num\ nos\ visitados|}{num\ max\ nos} \times 100 \qquad (4\text{-}2)$$

4.2.2
Geração dos fatores máximos de relaxamento – Graph500

Executamos testes para cada tripla <escala, fator de aresta, fator de relaxamento>. Toda vez que uma execução termina tendo visitado um número de nós maior que uma dada porcentagem do número máximo de nós (obtido através da execução do algoritmo 100% sincronizado), ela é contabilizada como uma execução satisfatória. Caso haja um número maior que um limiar de execuções não satisfatórias em uma tripla, o fator de relaxamento é diminuído e os testes recomeçam. Caso contrário o fator de relaxamento máximo foi encontrado.

4.2.3
Seleção e treinamento do método de aprendizagem supervisionada – Graph500

Primeiramente selecionamos os atributos de entrada que serão utilizados durante o treinamento dos algoritmos de classificação (aprendizagem supervisionada). Como entrada utilizamos a escala e o fator de aresta. Não tivemos dificuldade de selecionar os atributos pertinentes da instância de entrada dessa aplicação já que o grafo só apresenta estes dois atributos. Como saída utilizamos o fator de relaxamento máximo, encontrado no passo anterior de nossa metodologia. Após a execução de todos os métodos de classificação obtemos os resultados mostrados na Figura 4.13.

Figura 4.13: Resultados obtidos após o treinamento de todos os métodos de classificação disponíveis no Matlab.

Como é possível observar, dois método obtiveram o melhor treinamento para a nossa aplicação: o *KNN* que utiliza o kernel *Fine KNN* e o *KNN* que utiliza o kernel *Weighted KNN*. Após analisarmos as duas matrizes de confusão optamos pelo kernel *Fine KNN*. A Figura 4.14 mostra a matriz de confusão obtida após este treinamento. A diagonal principal contém a porcentagem de amostras que foram corretamente preditas. Observe pela figura que a maior parte dos elementos foram preditos ou na diagonal principal ou na diagonal imediatamente superior e inferior da matriz. Dessa forma, o modelo gerado se mostrou robusto durante o processo de generalização.

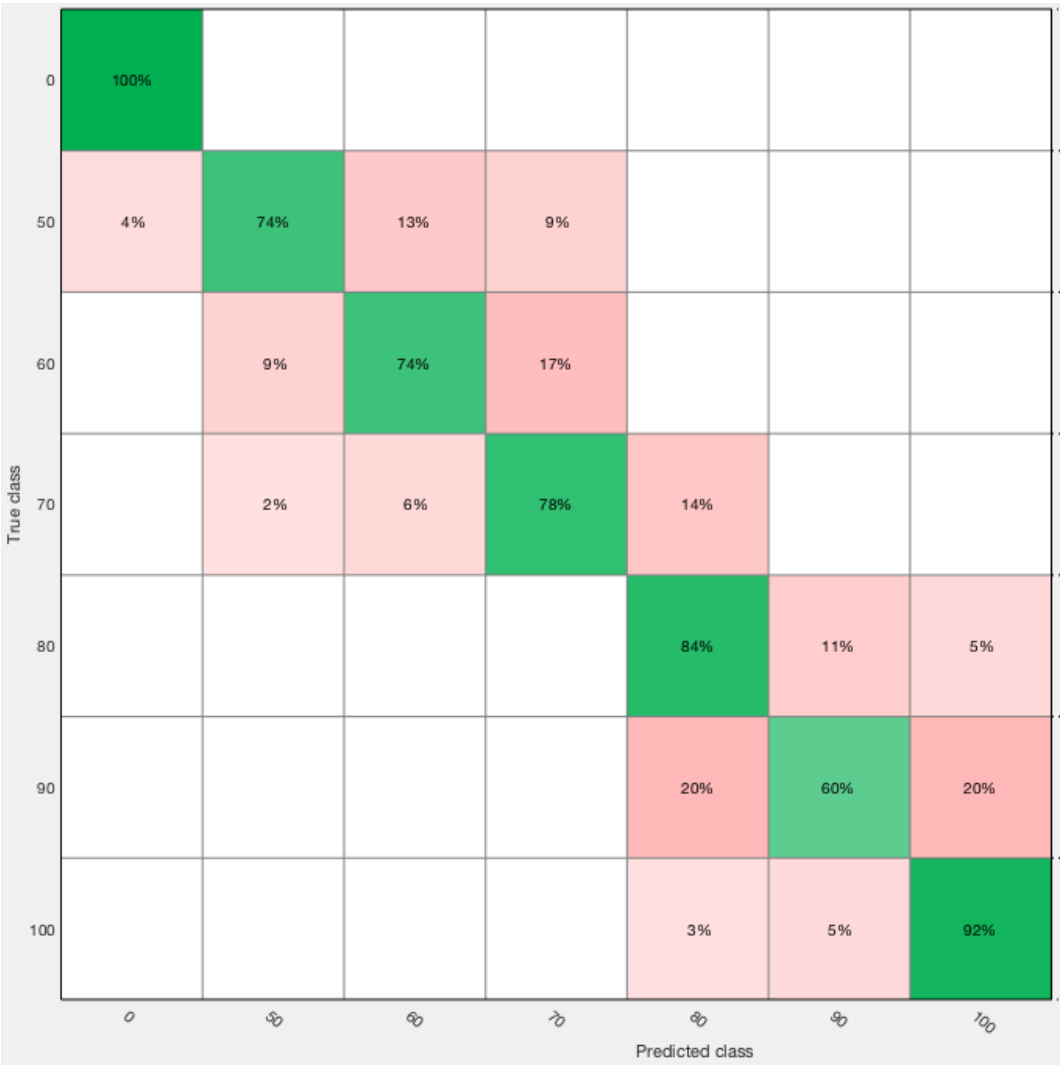

Figura 4.14: Matriz de confusão gerada a partir dos dados obtidos durante os testes com o algoritmo de classificação treinado.

4.2.4 Resultados – Graph500

Para exemplificar os ganhos que podem ser obtidos com a aplicação de nossa metodologia no algoritmo BFS, contido no *benchmark Graph500*, para determinados fatores de relaxamento, fixamos o fator de aresta em 16 (número de arestas é 16 vezes o número de nós) e variamos a escala do grafo entre 10 e 22 (2^{10} e 2^{22} nós). Os resultados são apresentados na Figura 4.15.

Adicionalmente, apresentamos na Figura 4.16 os resultados ao fixarmos a taxa de erro máximo em 10, 20, 30 e 40%.

A Figura 4.15 mostra os valores máximos de ganho obtidos para valores fixos de relaxamento (em cada gráfico) de forma livre, isto é, sem a utilização da nossa metodologia. Já a Figura 4.16 apresenta os valores de relaxamento máximos para cada grafo e número de threads (com erros fixados) utilizando a nossa metodologia. Para obter o ganho alcançado pelos valores dos gráficos da Figura 4.16 temos que relacionar o fator de relaxamento indicado nessa figura com o respectivo gráfico da Figura 4.15. Por exemplo, ao observar o gráfico de erro máximo = 30% da Figura 4.16, podemos notar que o grafo de escala 21 que utiliza 8 *threads* só pode relaxar nó máximo 90%. Voltando a Figura 4.15 observamos pelo gráfico de 90% de relaxamento que um grafo de escala 21 obtém no máximo um ganho de cerca de 5 vezes.

Em todas as execuções utilizamos 4 x CPU: Intel Xeon E5-2640 v4 2.40GHz (40 núcleos) variando o número de *threads* em 2, 4, 8, 16 e 32.

Na Figura 4.15 observamos um comportamento interessante. Tanto para 2, 4, 8, 16 ou 32 *threads* o ganho aumenta até uma determinada escala do grafo e então começa a diminuir. Em outras palavras, o algoritmo atinge uma saturação no ganho obtido pelo relaxamento de sincronização em um determinado número de nós e a partir daí tem o seu ganho diminuído com o aumento do número de nós. Observamos ainda que obtivemos ganhos máximos com 8 *threads*, em relação à versão paralela e sincronizada que utiliza 8 *threads*, para todos os fatores de relaxamento utilizados.

Na Figura 4.16 é possível observar um comportamento semelhante ao observado no *benchmark Labyrinth*. Quando o erro máximo admitido é aumentado maior se torna a possibilidade de relaxamento. Além disso, quanto maior o número de *threads*, menor a possibilidade de relaxamento. Isso se deve ao fato de que com mais *threads*, mais condições de corrida ocorrem perante a remoção dos pontos de sincronização.

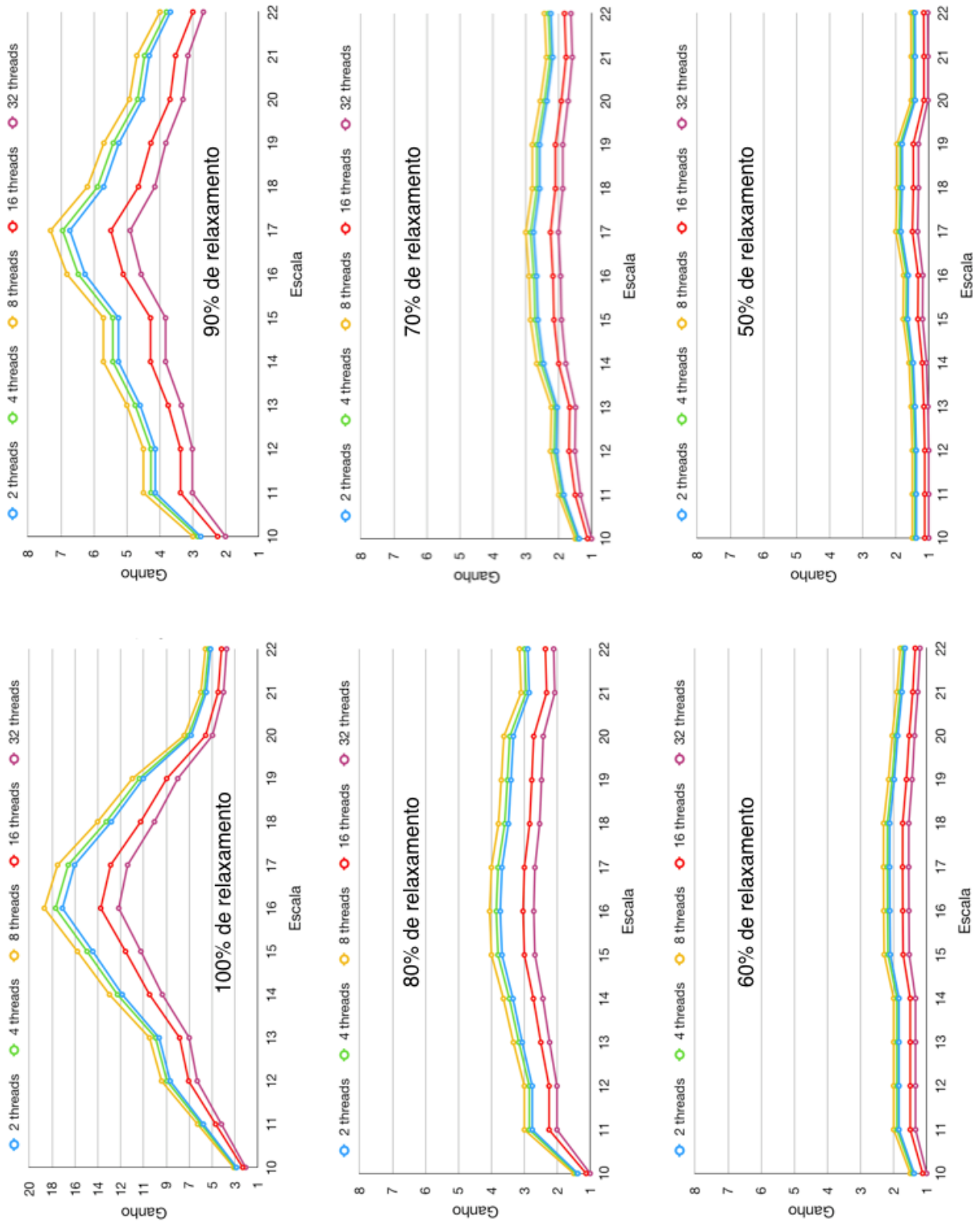

Figura 4.15: Ganho (em relação à versão paralela e sincronizada) para diferentes fatores de relaxamento e diferentes números de *threads*.

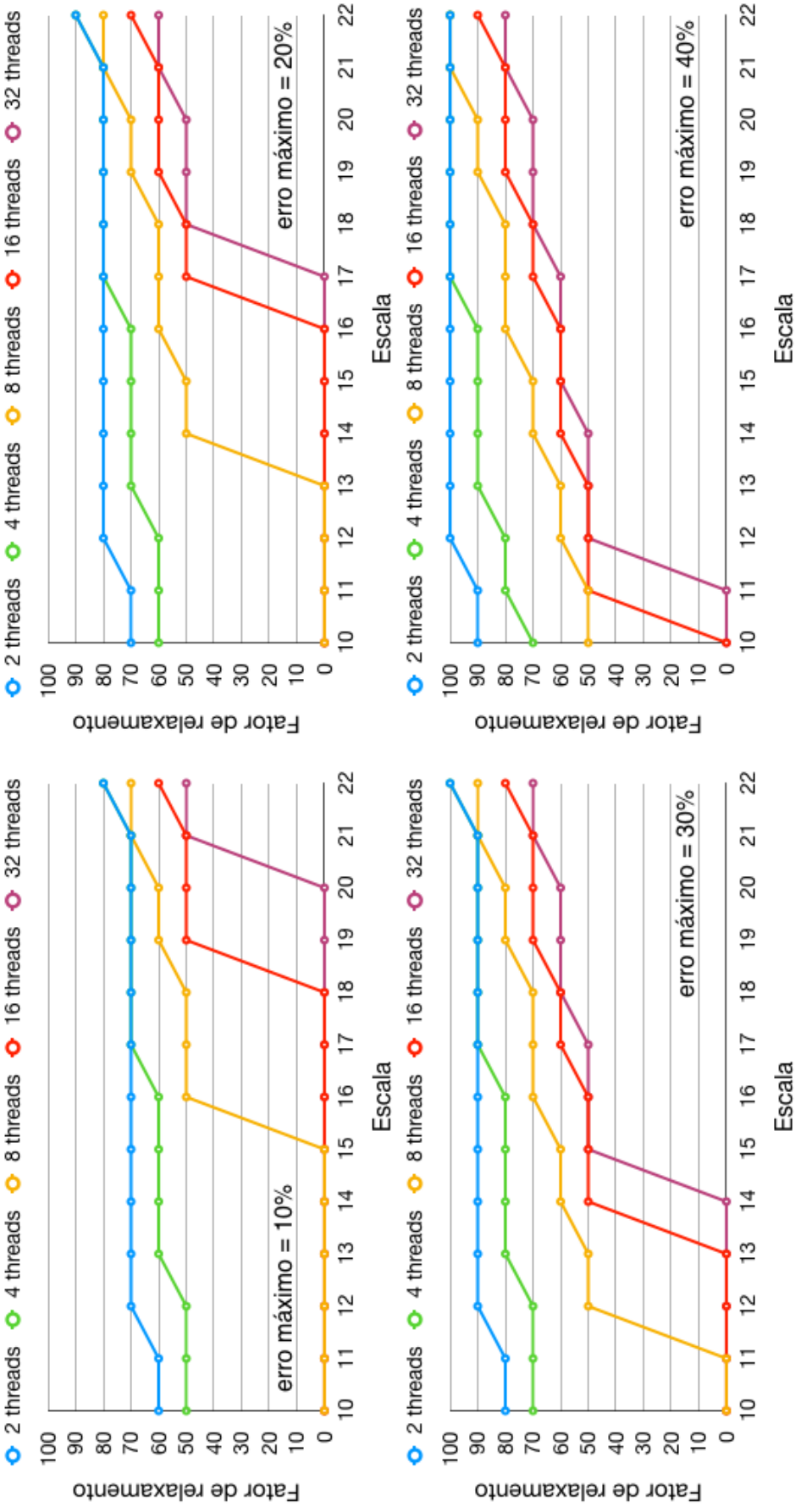

Figura 4.16: Fator de relaxamento obtido através de nossa metodologia para diferentes escalas de grafos e diferentes números de *threads*.

4.3 K-means

O *K-means* é um dos algoritmos iterativos mais simples da classe de algoritmos de aprendizagem não supervisionada, que resolve o problema de agrupamento de dados. O procedimento adotado pelo algoritmo classifica o conjunto de dados apresentado, em um certo número de clusters (k clusters) que é fixado a priori. A ideia principal do algoritmo é definir k centroides, um para cada cluster. Estes centroides devem ser inicializados preferencialmente distantes um dos outros, de forma uniforme. Inicialmente, cada ponto do conjunto de dados é associado ao centroide mais próximo. Em seguida, o algoritmo recalcula o posicionamento de todos os centroides como sendo o baricentro de cada cluster resultante do passo anterior. Após este reposicionamento uma nova associação é feita entre os pontos do conjunto de dados e os centroides. A cada iteração os k centroides mudam suas localizações até que nenhuma mudança aconteça, isto é, até que os centroides permaneçam com seus posicionamentos inalterados. Estes passos são mostrados na Figura 4.17.

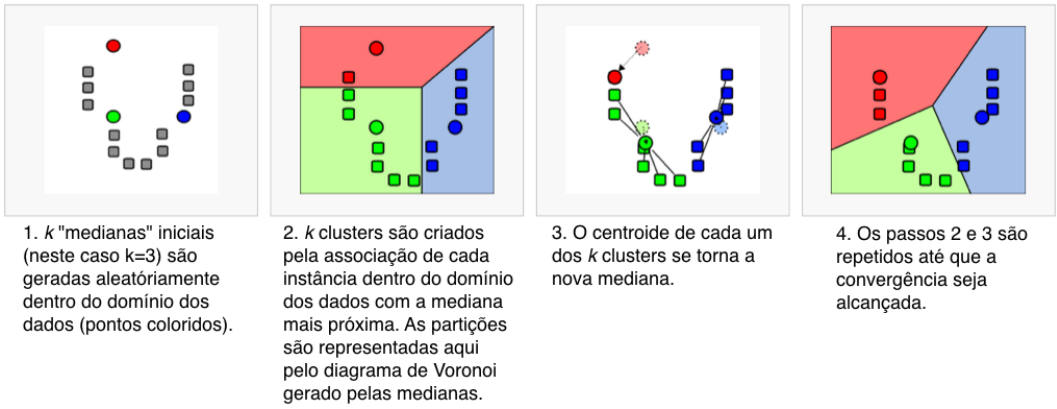

Figura 4.17: Passos do algoritmo de *K-means*.

Como aplicação do *K-means*, vamos utilizá-lo na redução de cores em vídeos.

Sabemos que nos *softwares* e *hardwares* de hoje imagens de alta qualidade

podem ser exibidas sem muito esforço. No entanto, essas imagens podem conter uma grande quantidade de informações detalhadas, implicando em tamanhos grandes e em tempos de processamento e transferência altos. Para evitar estes problemas, as informações desnecessárias podem ser eliminadas das imagens. Isso pode ser feito através de métodos de pré-processamento antes da transmissão. Por exemplo, para a construção de modelos de avaliação digital (DEM) de mapas topográficos, informações de cores desnecessárias podem ser removidas [24]. O método de quantificação de cores funciona como uma aplicação de pré-processamento usada para reduzir o número de cores em imagens com um mínimo de distorção, de modo que a imagem reproduzida seja muito próxima, visualmente, da imagem original. Em geral, a quantificação de cores é realizada em duas etapas. O primeiro passo é escolher o número de cores da palheta (geralmente entre 8 e 256). O segundo passo é mapear *pixels*, isto é, substituir a cor de cada *pixel* pela cor respectiva da palheta. A quantificação de cores desempenha um papel crítico em muitas aplicações como: segmentação, compressão, análise de textura de cores, localização/detecção de texto, renderização não-fotorealística e recuperação baseada em conteúdo [25].

A seguir, descrevemos o uso do algoritmo de *K-means* na redução de cores em um vídeo e nossa avaliação do uso combinado da Sincronização Relaxada com métodos de aprendizagem supervisionada nessa aplicação realística.

Utilizamos o código do algoritmo de K-means paralelizado obtido na página do professor Wei-keng Liao [26] da *Northwestern University*. Executamos o algoritmo separadamente para cada frame. Dessa forma nosso conjunto de dados passa a ser o conjunto de *pixels* de cada *frame* a ser modificado. Os *pixels* residem no espaço tridimensional de cores RGB. K representa aqui o número de cores RGB a que se deseja reduzir as cores do *frame* original.

A Figura 4.18 apresenta em linhas gerais os passos da aplicação de nossa

metodologia para o algoritmo de *K-means*: a partir de um vídeo inicial com um certo número de *frames*, selecionamos alguns destes em um certo intervalo de tempo. Para cada um dos *frames* selecionados geramos um arquivo texto contendo as cores RGB de todos os *pixels* da imagem (sem cores repetidas).

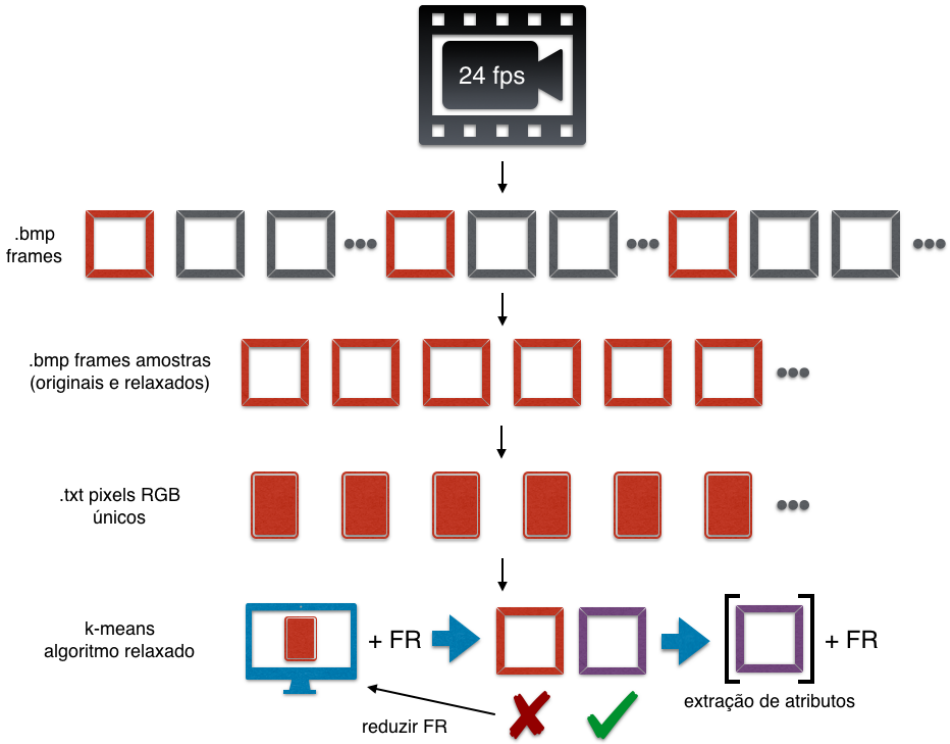

Figura 4.18: Aplicação da nossa metodologia no algoritmo de *K-means* para segmentação de cores de imagens.

Cada arquivo será apresentado como entrada para o algoritmo de *K-means*. Primeiramente executamos o algoritmo em sua versão sincronizada para todos os *frames* selecionados, já que precisaremos comparar cada saída relaxada do algoritmo com sua respectiva saída sincronizada. Em seguida o executamos com fatores regressivos de relaxamento, isto é, começamos com um fator de 100% e efetuamos a comparação de similidaridade da saída sincronizada com a saída relaxada. Se a comparação respeitar o erro máximo admitido, já achamos o fator de relaxamento máximo, caso contrário, o diminuímos em 10% e recomeçamos os testes. Após encontrar os fatores

máximos de relaxamento para todos os *frames* selecionados, iniciamos então a busca pelo método de aprendizagem supervisionada mais eficiente.

A seguir apresentaremos em detalhes todos os passos de nossa metodologia para o primeiro minuto do vídeo de animação *Big Buck Bunny* [2].

4.3.1 Parâmetros de entrada – K-means

A seguir listamos os parâmetros que foram fornecidos como entrada (para melhor visualização neste documento utilizamos de forma ilustrativa 60 *frames*, porém em nosso treinamento real utilizamos 120 *frames*):

(a) 60 *frames* selecionados, exibidos na Figura 4.19, que terão suas cores reduzidas. Os *frames* pertencem ao primeiro minuto do vídeo de animação *Big Buck Bunny*. O vídeo possui uma taxa de 24 *frames* por segundo (fps). Com um total de 1440 *frames*. Selecionamos como amostra 1 *frame* por segundo.

(b) Função de classificação de cada instância de entrada.

Experimentamos alguns atributos para a classificação dos *frames* antes de finalmente obter bons resultados através de uma técnica de visão computacional, muito usada para processamento de imagens, chamada extração de característica do ponto (*feature point extraction*). A característica extraída pode se referir a uma estrutura específica na imagem, por exemplo, uma área escura, brilhante ou uma borda. Essa técnica pode ainda ser usada para avaliar o resultado de alguma operação de vizinhança, por exemplo, calcular a direção local do campo de gradiente.

Em nossa aplicação utilizamos o algoritmo *Edward Rosten's FAST* [27] para detectar os cantos/esquinas mais fortes de cada *frame* através

[2] Big Buck Bunny. https://peach.blender.org, 2008.

da extração de características de pontos. Este algoritmo recebe como parâmetro uma imagem em preto e branco (escala de cinza).

Figura 4.19: Os 60 *frames* utilizados para a realização dos testes.

Para entender a ideia básica deste método observe a Figura 4.20. O parâmetro do algoritmo é um dado t, limiar de intensidade. Para cada ponto p da imagem de entrada a intensidade dos 16 *pixels* vizinhos é examinada. Existem 3 casos definidos para cada comparação (C):

$$C = \begin{cases} |I_p - I_n| < t & normal \\ I_n - I_p > t & claro \\ I_p - I_n > t & escuro \end{cases} \quad (4\text{-}3)$$

I_p denota a intensidade do ponto central, I_n é a intensidade do n-ésimo *pixel* vizinho. O ponto p é marcado como um canto/esquina, se 9-12 *pixels* contíguos são mais claros ou mais escuros que ele.

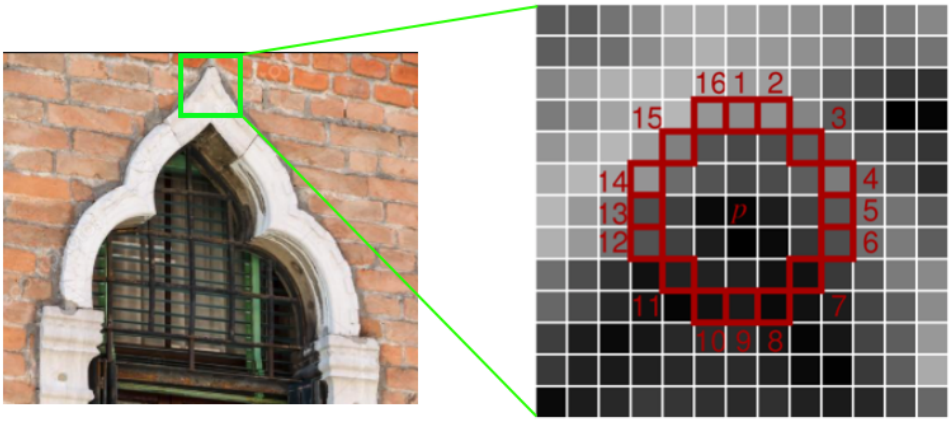

Figura 4.20: Algoritmo *Edward Rosten's FAST*.

Aplicamos o algoritmo *FAST* utilizando o Matlab para cada um dos 60 *frames* de entrada. Como o algoritmo só trabalha com imagens em preto e branco, utilizamos a função *rgb2gray* do Matlab para realizar essa transformação. A Figura 4.21 apresenta em verde os 10 cantos/esquinas mais fortes para cada um dos *frames*. Adicionalmente mostramos à direita o *frame* 18 ampliado com as respectivas 10 coordenadas $[x, y]$ (vídeo contendo os cantos/esquinas mais fortes para todos os *frames* disponível em https://goo.gl/BkCMxb).

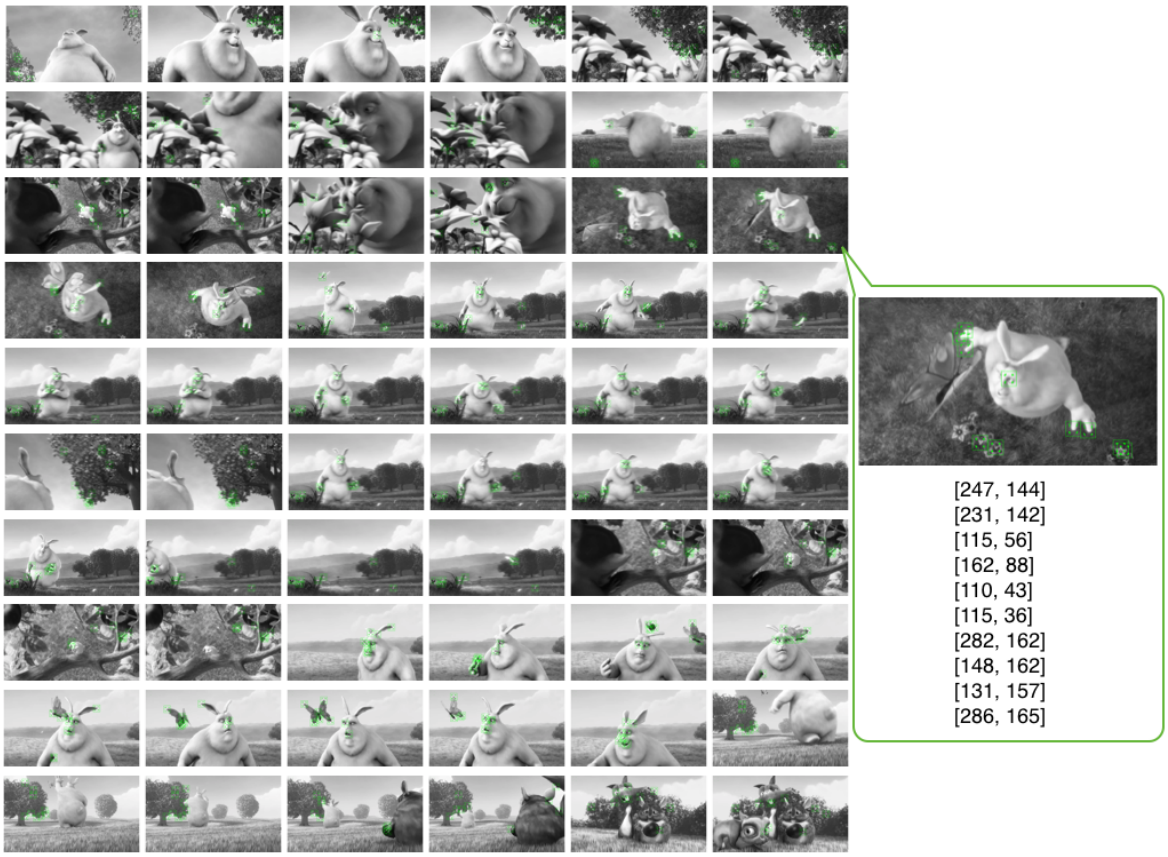

Figura 4.21: Algoritmo *Edward Rosten's FAST* aplicado para os 60 *frames* de entrada.

(c) Código fonte do programa com as devidas adições de chamadas apresentado na Figura 4.22. A função *isRelaxedRun* retorna 1 caso gere um número aleatório maior que RELAX_FACTOR e 0 caso contrário. A função gera números aleatórios entre [1 ... 10] e o parâmetro RELAX_FACTOR varia entre [0 ... 10], 0 significa 100% relaxado e 10, 0%.

(d) Função de cálculo de erro que recebe como parâmetro a saída do programa e retorna o respectivo erro associado.

Apesar da escolha final da qualidade de uma imagem ser definida pela visão humana, existem medidas automáticas definidas por algoritmos.

Usamos como função de cálculo de erro o chamado *Índice de Similaridade de Estrutura* (*SSIM*) [28] que funciona medindo a similaridade estrutural de duas imagens comparando padrões locais de intensidades de *pixel* que foram normalizadas pela luminância e contraste. Essa métrica de qualidade é baseada no princípio de que o sistema visual humano se baseia na estrutura da imagem para extrair informações.

```
int isRelaxedRun()
{
    int result = rand() % 10 + 1;

    if (result > RELAX_FACTOR)
        return 1;

    return 0;
}
```

```
#pragma omp parallel for private(i,j,index) \
    firstprivate(numObjs,numClusters,numCoords) \
    shared(objects,clusters,membership,newClusters) \
    schedule(static)
for (i=0; i < numObjs; ++i)
{
    // find the array index of nestest cluster center
    index = find_nearest_cluster(numClusters, numCoords,
    objects[i], clusters);

    // assign the membership to object i
    membership[i] = index;

    // update new cluster centers:
    if (!isRelaxedRun())
    {
        for (j=0; j < numCoords; ++j)
          #pragma omp atomic
          newClusters[index][j] += objects[i][j];
    }
    else
    {
        for (j=0; j < numCoords; ++j)
          newClusters[index][j] += objects[i][j];
    }
}
```

Figura 4.22: Trecho do código fonte do algoritmo de *K-means* exibindo a adição do relaxamento dinâmico.

4.3.2 Geração dos fatores máximos de relaxamento – K-means

Neste passo, precisamos achar para cada *frame* o seu fator máximo de relaxamento. Toda vez que uma execução termina com um SSIM maior ou igual

ao SSIM mínimo definido, ela é contabilizada como uma execução satisfatória. Caso contrário, o fator de relaxamento é diminuído e os testes recomeçam. A Figura 4.23 apresenta os fatores máximos de relaxamento obtidos para cada um dos 60 *frames* exibidos na Figura 4.19, para SSIMs de 90-95%, 80-85% e 70-75%.

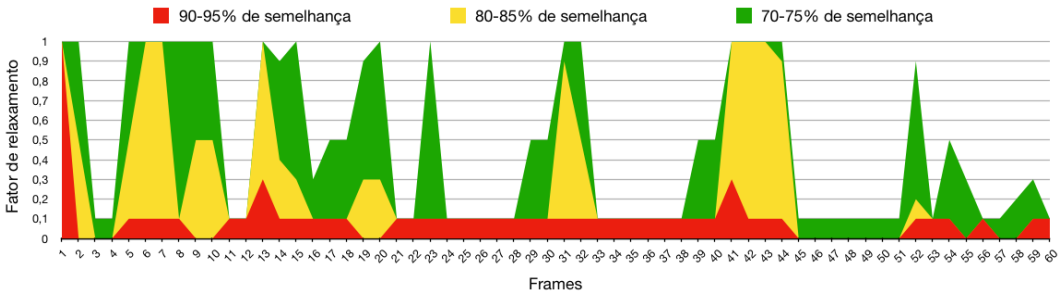

Figura 4.23: Fatores máximos de relaxamento, utilizando 4 *threads*, para diferentes fatores de similaridade.

4.3.3 Seleção e treinamento do método de aprendizagem supervisionada – K-means

Primeiramente selecionamos os atributos de entrada que serão utilizados durante o treinamento dos algoritmos de classificação. Para essa aplicação encontramos dificuldades ao selecionar os atributos pertinentes da instância de entrada, no caso cada frame do vídeo. Testamos diversos atributos como por exemplo o número de cores e parâmetros do histograma de cada frame que, ao serem utilizados, não resultaram em bons resultados de treinamento. Após extensa pesquisa, encontramos o algoritmo *FAST* através da ferramenta Matlab e assim utilizamos como entrada as 5 coordenadas $[x, y]$ mais fortes geradas por este algoritmo, para cada frame. Como saída utilizamos o fatores de relaxamento máximos dos *frames*, encontrados no passo anterior de nossa metodologia. Após a execução de todos os métodos de classificação obtivemos os resultados mostrados na Figura 4.18. Como é possível observar, o método

que obteve o melhor treinamento para a nossa aplicação foi o *Ensemble* que utiliza o kernel *Bagged Trees*. A Figura 4.25 mostra a matriz de confusão obtida após o seu treinamento. A diagonal principal contém a porcentagem de amostras que foram corretamente preditas.

1.1 ☆ Tree Last change: Complex Tree	Accuracy: 86.1% 10/10 features	1.12 ☆ KNN Last change: Fine KNN	Accuracy: 84.7% 10/10 features
1.2 ☆ Tree Last change: Medium Tree	Accuracy: 84.5% 10/10 features	1.13 ☆ KNN Last change: Medium KNN	Accuracy: 84.2% 10/10 features
1.3 ☆ Tree Last change: Simple Tree	Accuracy: 82.2% 10/10 features	1.14 ☆ KNN Last change: Coarse KNN	Accuracy: 76.5% 10/10 features
1.4 ☆ Linear Discriminant Last change: Linear Discriminant	Accuracy: 74.3% 10/10 features	1.15 ☆ KNN Last change: Cosine KNN	Accuracy: 83.1% 10/10 features
1.5 ☆ Quadratic Discriminant Last change: Quadratic Discriminant	Failed 10/10 features	1.16 ☆ KNN Last change: Cubic KNN	Accuracy: 84.1% 10/10 features
1.6 ☆ SVM Last change: Linear SVM	Accuracy: 75.4% 10/10 features	1.17 ☆ KNN Last change: Weighted KNN	Accuracy: 85.8% 10/10 features
1.7 ☆ SVM Last change: Quadratic SVM	Accuracy: 83.5% 10/10 features	1.18 ☆ Ensemble Last change: Boosted Trees	Accuracy: 85.2% 10/10 features
1.8 ☆ SVM Last change: Cubic SVM	Accuracy: 83.5% 10/10 features	1.19 ☆ Ensemble Last change: Bagged Trees	**Accuracy: 87.5%** 10/10 features
1.9 ☆ SVM Last change: Fine Gaussian SVM	Accuracy: 84.5% 10/10 features	1.20 ☆ Ensemble Last change: Subspace Discriminant	Accuracy: 73.8% 10/10 features
1.10 ☆ SVM Last change: Medium Gaussian SVM	Accuracy: 84.8% 10/10 features	1.21 ☆ Ensemble Last change: Subspace KNN	Accuracy: 85.8% 10/10 features
1.11 ☆ SVM Last change: Coarse Gaussian SVM	Accuracy: 74.2% 10/10 features	1.22 ☆ Ensemble Last change: RUSBoosted Trees	Accuracy: 74.2% 10/10 features

Figura 4.24: Resultados obtidos após o treinamento de todos os métodos de classificação disponíveis no Matlab (para 120 *frames* selecionados) - fator de similaridade de 90-95% (4 *threads*).

Como é possível observar na Figura 4.25, obtivemos nessa matriz de confusão resultados não muito precisos de previsão, porém o método treinado está errando para valores mais conservadores de relaxamento, por exemplo, 22% das amostras foram previstas corretamente com 50% de relaxamento enquanto o 78% foram previstas erroneamente entre 30% e 0% (fatores menores de relaxamento). Sendo assim não houve perda de qualidade e sim uma certa perda no desempenho adicional que poderia ser obtido.

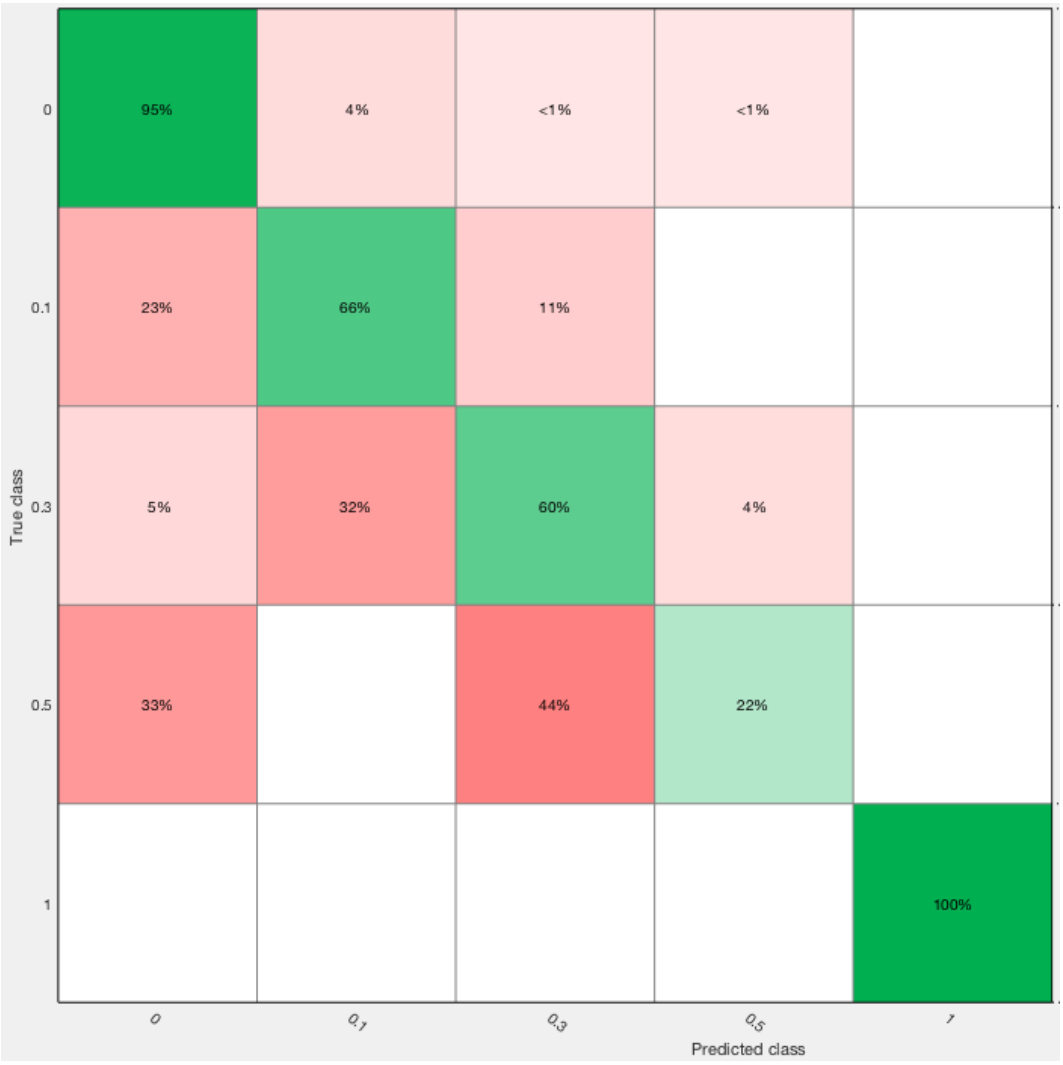

Figura 4.25: Matriz de confusão gerada a partir dos dados obtidos durante os testes com o algoritmo de classificação treinado (para 120 *frames* selecionados) - fator de similaridade de 90-95% (4 *threads*).

4.3.4 Resultados – K-means

Por fim, apresentamos através da Figura 4.26, a variação do fator de relaxamento obtido através da apresentação de todos os *frames* do vídeo para o algoritmo de classificação treinado, utilizando três diferentes fatores de similaridade. Note que as barras no eixo x aparecem na seguinte ordem: vermelho por cima do amarelo e amarelo por cima do verde.

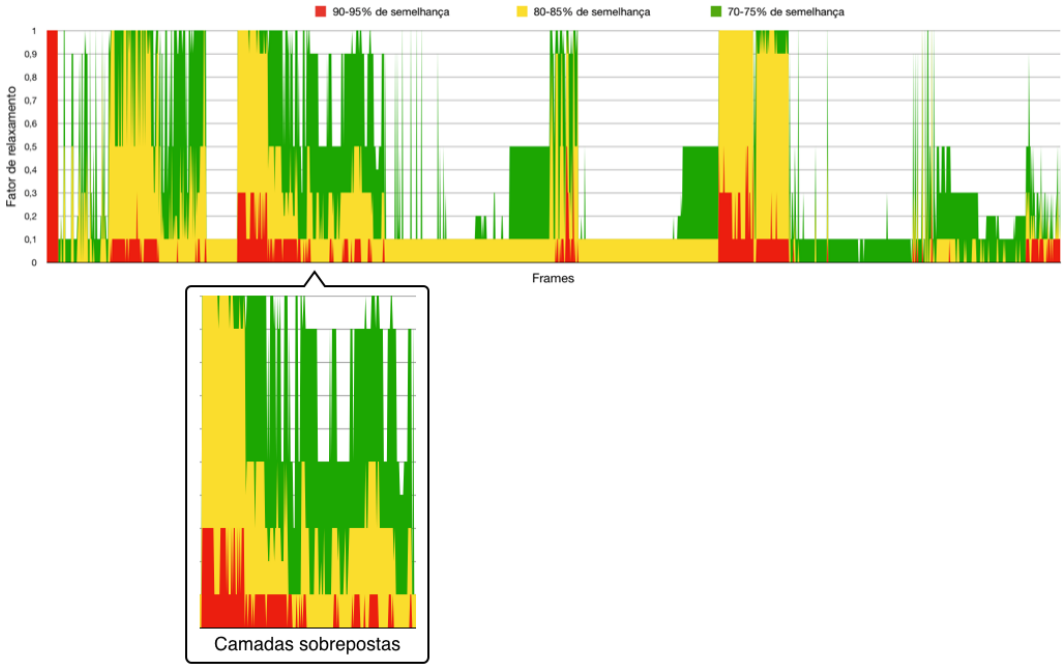

Figura 4.26: Fatores máximos de relaxamento, utilizando 4 *threads*, para diferentes fatores de similaridade.

O tempo total da metodologia é composto das seguintes partes:

(a) Geração das entradas para o algoritmo de *K-means* - essa parte é responsável por transformar o arquivo de vídeo em um arquivo texto (para cada frame) contendo as cores RGB de cada *pixel* (sem cores repetidas).

(b) Treinamento - essa parte é responsável por encontrar para cada *frame* selecionado o seu fator máximo de relaxamento.

(c) Execução - essa parte é responsável por apresentar cada um dos *frames* do vídeo original (já transformados em arquivo texto) para o método de aprendizagem supervisionada que obteve melhor treinamento, a fim de descobrir o seu fator de relaxamento e assim executar o algoritmo de *K-means* utilizando este fator de relaxamento próprio de cada frame.

A Tabela 4.3 apresenta os tempos de execução (com o fator de relaxamento já descoberto para cada frame) e o tempo de treinamento (o tempo de treinamento foi estimado para 8, 16 e 32 *threads* já que não possuíamos a licença Linux do software Matlab) para diferentes números de *threads* e fatores de semelhança. Adicionalmente, a Figura 4.27 estes tempos de execução através de um gráfico e a Figura 4.28 o ganho obtido em relação a versão paralela e sincronizada.

	Sinc	**95-90% (exec/treino)**		**85-80% (exec/treino)**		**75-70% (exec/treino)**		**100% relax**
4 *threads*	2049	1207	411	1067	373	1030	352	872
8 *threads*	1140	642	341	561	223	533	210	444
16 *threads*	814	366	227	304	134	280	126	230
32 *threads*	521	228	151	169	81	149	76	121

Tabela 4.2: Tempos (s) de execução do treinamento e posterior execução dos *frames* no algoritmo de classificação treinado, para 4, 8, 16 e 32 *threads*, em diversos níveis de relaxamento.

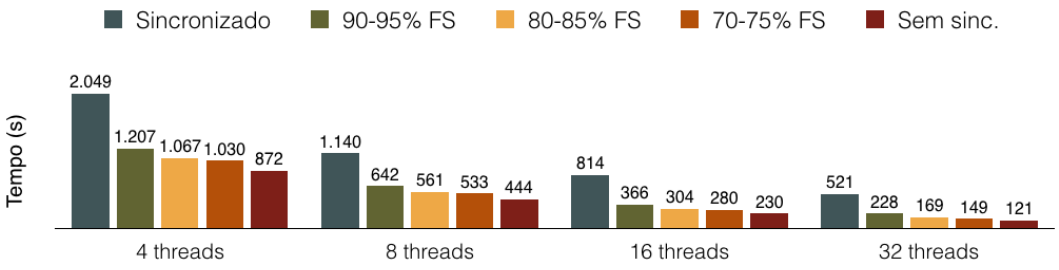

Figura 4.27: Tempo de execução em diferentes fatores de semelhança e números de *threads*. 4 x CPU: Intel Xeon E5-2640 v4 2.40GHz (40 núcleos).

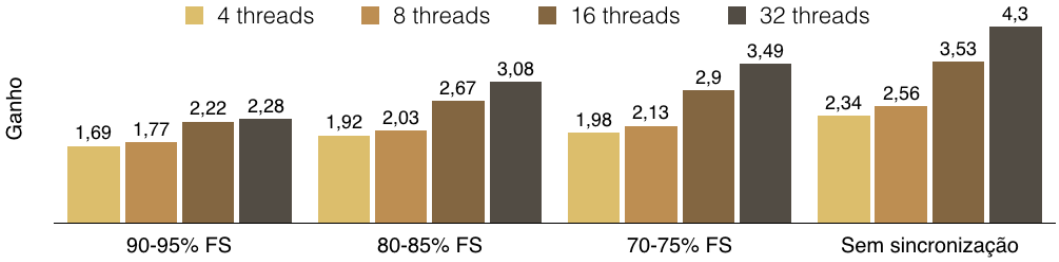

Figura 4.28: Ganho (em relação a versão paralela e sincronizada) em diferentes fatores de semelhança e números de *threads*. 4 x CPU: Intel Xeon E5-2640 v4 2.40GHz (40 núcleos).

Disponibilizamos o vídeo com os resultados finais obtidos em https://goo.gl/nDrzvW.

4.3.5 Câmera de trânsito estática

Adicionalmente quisemos testar o comportamento da geração de fatores de relaxamento máximo para uma câmera de trânsito estática. Diferente do vídeo analisado anteriormente, os *frames* aqui mudam muito pouco durante a transmissão, esperamos portanto pouca mudança do fator de relaxamento durante a passagem dos *frames*. As Figuras 4.29 e 4.30 apresentam na parte

superior o *frame* do vídeo original gerado pela câmera e na parte inferior o mesmo *frame* processado pelo algoritmo de *K-means* para K=2 (duas cores).

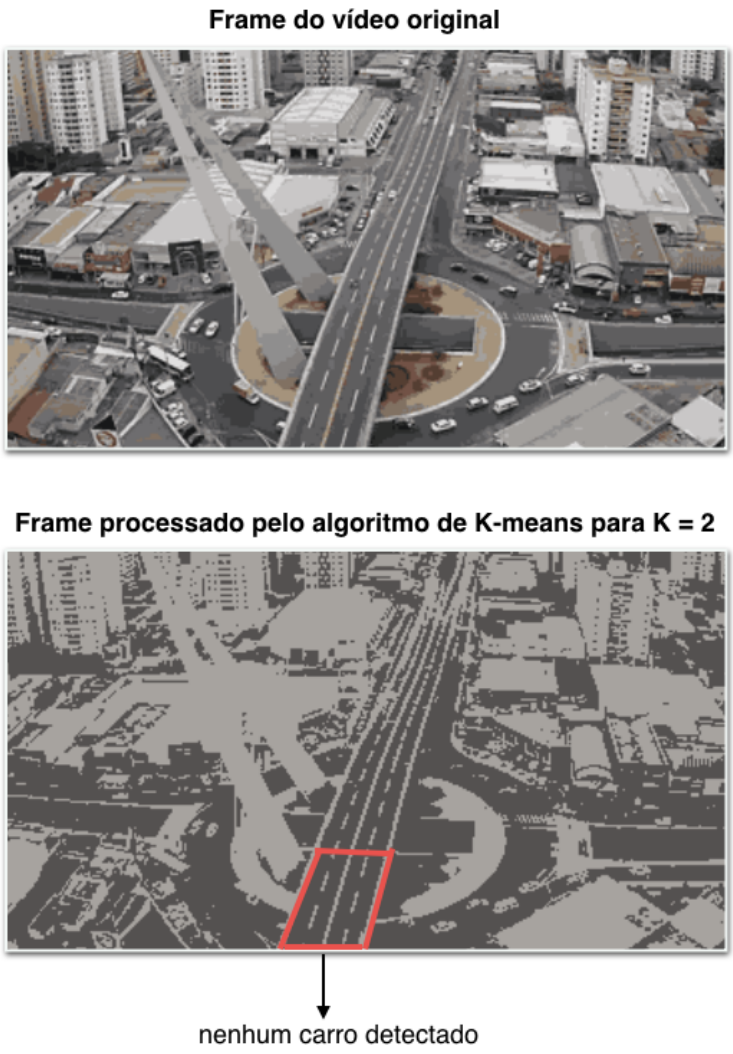

Figura 4.29: Exemplo da não detecção de carros na aplicação do algoritmo de *K-means* em vídeos em uma câmera de trânsito estática.

Disponibilizamos o vídeo do algoritmo de *K-means* aplicado a este vídeo em https://goo.gl/Vy2gUo.

A detecção do número de carros que cruza a ponte em um determinado tempo poderia ser feita, por exemplo, através de um código que "observa" se determinado grupo de *pixels* (representado nas Figuras 4.29 e 4.30 pelo quadrilátero vermelho) sofrerá modificações significativas com o passar dos *frames*.

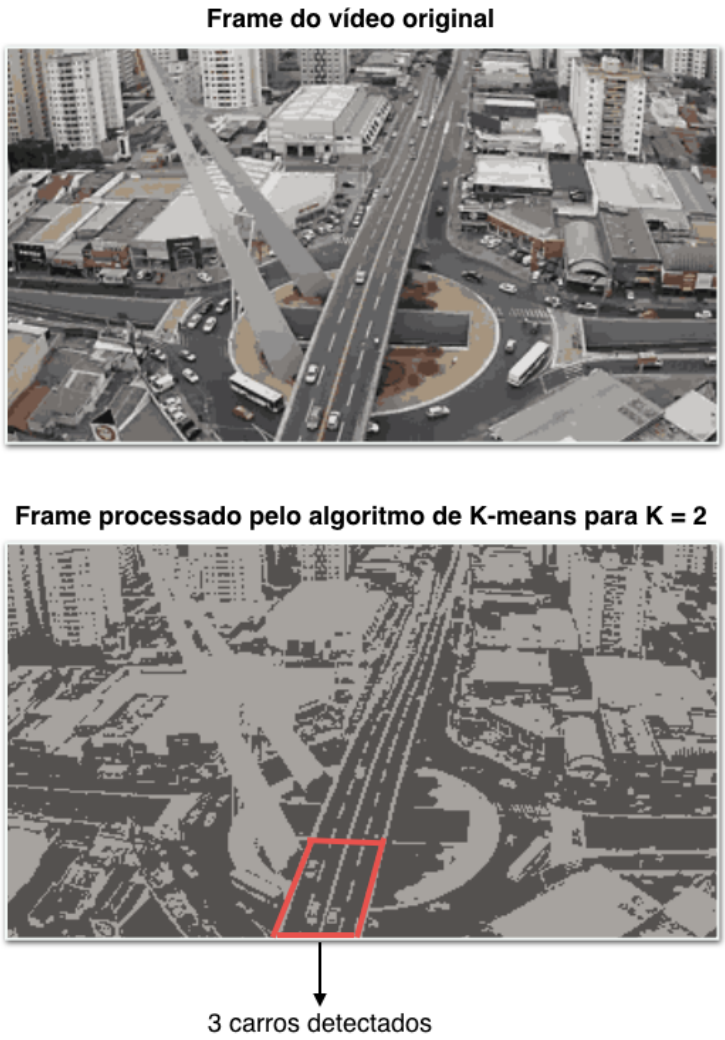

Figura 4.30: Exemplo da detecção de carros na aplicação do algoritmo de *K-means* em vídeos em uma câmera de trânsito estática.

Nossos testes mostraram que para qualquer fator de semelhança fixado a taxa de relaxamento máximo se manteve fixa. Isto é, se quisermos um vídeo relaxado com um fator de semelhança entre 70-75% poderemos relaxar a sincronização com uma taxa fixa de 90%. Já para um fator de semelhança entre 80-85%, relaxar com uma taxa fixa de 80%. Finalmente para um fator de semelhança entre 90-95%, relaxar com uma taxa fixa de 70%. O algoritmo de *K-means* portanto, reage de forma particular ao relaxamento dependendo do conjunto de entrada apresentado (neste caso, o conjunto de *frames*).

Dessa forma, para essa aplicação em particular não se faz necessário o

treinamento de métodos de aprendizagem supervisionada visto que, durante a fase de geração dos fatores máximos de relaxamento, a taxa permaneceu fixa. Apesar disso, foi relevante descobrir que, mesmo para uma câmera estática, o vínculo entre o fator de semelhança e a taxa de relaxamento permanecem.

Considerações finais

Testamos a viabilidade de nossa proposta em três aplicações, duas provenientes de benchmarks: *Labyrinth - STAMP* e *BFS - Graph 500*, e também o algoritmo de *K-means* aplicado à segmentação de cores em vídeos.

Na aplicação *Labyrinth* obtivemos um ganho máximo entre 1,5x e 1,6x para 100% de relaxamento. Para outros fatores de relaxamento, obtivemos um ganho de até 1,3x. Na aplicação *BFS* obtivemos um ganho máximo entre 18x e 19x para 8 *threads* em grafos de escala 21 e 22. Na aplicação *K-means* obtivemos um ganho máximo de 3,5x para um fator de similaridade entre 70-75% e 32 *threads*.

Mostramos que, para todos os casos estudados, métodos de aprendizagem supervisionada são uma excelente ferramenta para dar estabilidade e certa segurança para o programador ao aplicar a Sincronização Relaxada em sua aplicação, já que mantiveram resultados suficientemente bons gerando maior desempenho aos algoritmos.

Atestamos também que obter a função de classificação e a função de erro pode ser uma tarefa árdua de experimentação dependendo das características do algoritmo a ser relaxado.

5
Conclusão

Todos os trabalhos na área da Sincronização Relaxada que encontramos na literatura se limitam a remover totalmente os pontos de sincronização do algoritmo a ser relaxado e verificar se, para um pequeno conjunto de instâncias de entrada, o resultado final permanece aceitável. Logo, tivemos por objetivo neste trabalho avaliar a viabilidade de usar métodos de aprendizagem supervisionada para garantir que a técnica de Sincronização Relaxada forneça resultados dentro de limites aceitáveis de erro para qualquer tripla aplicação/entrada/ambiente de execução. Para isso, criamos uma metodologia que utiliza alguns dados de entrada para montar casos de testes que, ao serem executados, possam fornecer valores representativos para a criação e treinamento de métodos de aprendizagem supervisionada. Ao apresentar uma nova entrada, o algoritmo de classificação treinado sugere o fator de relaxamento mais adequado à tripla aplicação/entrada/ambiente de execução. Analisamos os resultados através da qualidade das previsões obtidas e também nos ganhos de tempo em relação à versão paralela e 100% sincronizada.

O *overhead* gerado pela sincronização em aplicações paralelas é um importante fator limitante de desempenho. Apresentamos uma nova metodologia capaz de viabilizar melhorias significativas de desempenho, para aplicações que toleram alguma perda de qualidade no resultado da computação, através da minimização ou até mesmo completa remoção dos pontos de sincronização. A nossa técnica permite que os programadores relaxem a sincronização de suas aplicações, de forma dinâmica, com a garantia probabilística de obtenção de resultados finais com uma taxa de similaridade acordada inicialmente.

Nossos testes mostraram que, ao utilizar a Sincronização Relaxada de forma dinâmica e com o apoio de métodos de aprendizagem supervisionada, pudemos obter ganhos significativos e controlados de desempenho. Por exemplo, obtivemos um ganho de 3,5x para o algoritmo de *K-means* e entre 18 e 19x para o algoritmo BFS em relação a versão paralela e totalmente sincronizada, mantendo a taxa de similaridade desejada. Os resultados obtidos neste trabalho são uma forte indicação do potencial da área geral da Computação Aproximada, onde o determinismo e a precisão no processo de obtenção de resultados é sacrificado em prol de um melhor desempenho, consumo de energia reduzido e um menor custo do sistema.

Apesar de demonstrarmos a eficácia de nossa metodologia em conseguir acelerar a velocidade de execução das aplicações testadas, atestamos também a dificuldade durante o processo de generalização de seus passos para que se adequasse qualquer tipo de aplicação. Dessa forma, a aplicação da metodologia precisou ser feita de forma personalizada e praticamente artesanal para cada tipo de aplicação estudada. Os maiores desafios que nós encontramos e que poderão ser abordados em trabalhos futuros foram: encontrar os atributos pertinentes a instância de entrada e, portanto, a função de classificação, encontrar a função de erro, que quantifica o quão perto o resultado relaxado se encontra do resultado original, efetuar a melhor modificação da aplicação original para introduzir a fração relaxada e incluir o número de threads como parte do treinamento.

Referências bibliográficas

[1] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *Proceedings of the 50th Annual Design Automation Conference*, p. 113, ACM, 2013.

[2] M. D. McCool, A. D. Robison, and J. Reinders, *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[3] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *2013 18th IEEE European Test Symposium (ETS)*, pp. 1–6, IEEE, 2013.

[4] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener, "Programming with relaxed synchronization," in *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, pp. 41–50, ACM, 2012.

[5] S. Misailovic, S. Sidiroglou, and M. C. Rinard, "Dancing with uncertainty," in *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, pp. 51–60, ACM, 2012.

[6] M. C. Rinard, "Unsynchronized techniques for approximate parallel computing," in *RACES-SPLASH (Systems, Programming, Languages and Applications: Software for Humanity) Workshop*, 2012.

[7] M. C. Rinard, "(relative) safety properties for relaxed approximate programs," in *RACES-SPLASH (Systems, Programming, Languages and Applications: Software for Humanity) Workshop*, 2012.

[8] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, "Supervised machine learning: A review of classification techniques," in *Informatica*, pp. 249–268, 2007.

[9] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Transactions on Electronic Computers*, no. EC-10(3), pp. 346–365, 1961.

[10] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[11] MATLAB, *version 9.2.0.538062 (R2017a)*. Natick, Massachusetts: The MathWorks Inc., 2017.

[12] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 124–134, ACM, 2011.

[13] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using code perforation to improve performance, reduce energy consumption, and respond to failures," in *Technical report*, MIT, 2009.

[14] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *ACM SIGPLAN Notices*, vol. 46, pp. 199–212, ACM, 2011.

[15] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, "Proving programs robust," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European conference on Foundations of Software Engineering*, pp. 102–112, ACM, 2011.

[16] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 35–50, ACM, 2014.

[17] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 13–24, ACM, 2013.

[18] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications," *Multimedia, IEEE Transactions on*, vol. 15, no. 2, pp. 279–290, 2013.

[19] M. Rinard, "Parallel synchronization-free approximate data structure construction.," in *5th USENIX Workshop on Hot Topics in Parallelism*, pp. 1–8, 2013.

[20] H.-J. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," in *ACM SIGPLAN Notices*, vol. 43, pp. 68–78, ACM, 2008.

[21] H.-J. Boehm, "Position paper: nondeterminism is unavoidable, but data races are pure evil," in *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, pp. 9–14, ACM, 2012.

[22] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pp. 35–46, IEEE, 2008.

[23] I. Watson, C. Kirkham, and M. Luján, "A study of a transactional parallel routing algorithm," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pp. 388–398, IEEE Computer Society, 2007.

[24] R. Samet and E. Hancer, "A new approach to the reconstruction of contour lines extracted from topographic maps," *Journal of Visual Communication and Image Representation*, vol. 23, no. 4, pp. 642–647, 2012.

[25] M. E. Celebi, Q. Wen, and J. Chen, "Color quantization using c-means clustering algorithms," in *Image Processing (ICIP), 2011 18th IEEE International Conference on*, pp. 1729–1732, IEEE, 2011.

[26] Wei-keng Liao. `http://www.ece.northwestern.edu/~wkliao/Kmeans/index.html`, 2017.

[27] E. Rosten, R. Porter, and T. Drummond, "Faster and better: A machine learning approach to corner detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 1, pp. 105–119, 2010.

[28] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

A
Script de geração das instâncias de entrada para o algoritmo de Lee

```
import random
import sys

if len(sys.argv) != 5:
    print "Usage: ./generate.py x y z n"
    sys.exit(1)

x = int(sys.argv[1])
y = int(sys.argv[2])
z = int(sys.argv[3])
numPath = int(sys.argv[4])

random.seed(0)

print "# Dimensions (x, y, z)"
print "d  %i %i %i" % (x, y, z)
print ""
print "# Paths: Sources (x, y, z) -> Destinations (x, y, z)"
for i in range(numPath):
    src = (random.randint(0, x-1),
           random.randint(0, y-1),
           random.randint(0, z-1))
    while 1:
        dst = (random.randint(0, x-1),
               random.randint(0, y-1),
               random.randint(0, z-1))
        if dst != src:
            break;
    print "p %3i %3i %1i %3i %3i %1i" % (src[0], src[1], src[2],
                                         dst[0], dst[1], dst[2])
```