Universidade Federal Rural de Pernambuco

Departamento de Estatística e Informática

Programa de Pós-Graduação em Informática Aplicada

Extraindo conhecimento de séries temporais com o uso de um Sistema Híbrido Inteligente desenvolvido na plataforma CUDA

Henrique Correia Torres Santos

Recife

Abril de 2014

Henrique Correia Torres Santos

Extraindo conhecimento de séries temporais com o uso de um Sistema Híbrido Inteligente desenvolvido na plataforma CUDA

Orientador: Prof. Dr. Tiago Alessandro Espínola Ferreira

Dissertação de mestrado apresentada ao Curso de Pós-Graduação em Informática Aplicada da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do grau de Mestre em Informática Aplicada.

Recife

Abril de 2014

Henrique Correia Torres Santos

Extraindo conhecimento de séries temporais com o uso de um Sistema Híbrido Inteligente desenvolvido na plataforma CUDA

Dissertação julgada adequada para obtenção do título de Mestre em Informática Aplicada, defendida e aprovada por unanimidade em 20/02/2014 pela Banca Examinadora.

Orientador:

Prof. Dr. Tiago Alessandro Espínola Ferreira

Universidade Federal Rural de Pernambuco

Banca Examinadora:

Prof. Dr. Francisco Madeiro Bernardino Junior

Universidade de Pernambuco

Prof. Dr. Giordano Ribeiro Eulalio Cabral

Universidade Federal Rural de Pernambuco

Prof. Dr. Jones Oliveira de Albuquerque

Universidade Federal Rural de Pernambuco

Recife

Abril de 2014

Dedico este trabalho primeiramente aos meus pais que sempre me apoiaram e também a minha esposa e filha porque sem elas não seria possível ter concluído a realização deste trabalho.

Agradecimentos

Agradecer a todos que me ajudaram, direta ou indiretamente, a construir esta dissertação. Não foi uma tarefa fácil e sem a ajuda destas pessoas este trabalho não seria concluído. Em primeiro lugar gostaria de agradecer aos meus pais, Alcides e Edna, que sempre entenderam as minhas ausências e me tranquilizavam para que eu me concentrasse no meu trabalho. Sem essa tranquilidade não seria possível me manter focado no meu trabalho. Gostaria de agradecer também a minha esposa Kate e a minha filha Bárbara por entenderem que eu nem sempre poderia acompanhá-las nos eventos sociais e familiares. Sei que vocês entendiam que eu tinha prazos para cumprir, mas até mesmo para mim era difícil não estar com vocês em alguns momentos. Fiz muitos amigos nesse curso e gostaria de agradecer a todos pelo companheirismo e ajuda prestada nos momentos de dificuldade. Entre esses amigos estão alunos, João, Kleber, Neilson, Gabriela, Ronaldo, Ricardo, Paulo e professores como Giordano Cabral por ter despertado em mim o interesse pelos sistemas híbridos inteligentes. Em especial gostaria de agradecer ao meu orientador Tiago Alessandro primeiro por me convencer a seguir essa área de pesquisa, que trouxe muitos desafios, e por me dar todo o suporte para que eu encontrasse o caminho para enfrentar e vencer todos esses desafios.

"Não reze por uma vida fácil, reze para ter forças para suportar uma vida difícil." (Bruce Lee)

Resumo

Neste trabalho, um Particle Swarm Optimization (PSO) combinado com uma lista tabu é proposto para extrair os padrões de comportamento de fenômenos dependentes do tempo. Uma série temporal pode ser criada pela observação temporal de um fenômeno e a dinâmica das tendências local do fenômeno pode ser descrito por uma codificação binária da série temporal, onde "1" é atribuído para as tendências positivas e "0" é atribuído para os outros casos. O algoritmo proposto procura os padrões de comportamento incorporados na série histórica, ou regras que descrevem as leis que regem a dinâmica do fenômeno estudado. Portanto, cada partícula do enxame é composta por uma regra de prova com uma janela de reconhecimento e um valor de previsão. Um conjunto de séries temporais artificiais, naturais e financeiras é utilizado para avaliar o método proposto. Para aumentar o desempenho das simulações o algoritmo proposto é desenvolvido em CUDA fazendo uso do poder de processamento paralelos das GPGPUs. Os resultados experimentais mostram que o método proposto é uma abordagem promissora para a previsão de tendência e extração de conhecimento a partir dos dados de séries temporais.

Palavras-chave: Séries Temporais, Previsão de Tendências, Forecasting, Particle Swarm Optimization, PSO, CUDA.

Abstract

In this paper, a Particle Swarm Optimizer combined with a tabu list is proposed to extract the behavior patterns from time dependents phenomena. A time series can be created by the temporal phenomenon observation and the local tendency dynamic of the phenomenon can be described by a binary codification of the time series, where "1"is assigned for positive trends and "0"is assigned for otherwise. The proposed algorithm searches for the behavior patterns embedded in the time series, or rules that describe the laws that govern the dynamics of the studied phenomenon. Therefore, each particle of the swarm consists of a trial rule with a recognition window and a forecast value. A set of artificial, natural and financial time series is used to evaluate the proposed method. To increase the performance of the simulations the proposed algorithm is developed making use of the CUDA parallel processing power of the GPGPUs. The experimental results show that the proposed method is a promising approach for tendency forecasting and extraction of knowledge from the time series data.

Keywords: Time Series, Tendency Forecasting, Particle Swarm Optimization, PSO, CUDA.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Neste capítulo são apresentados uma breve introdução e os objetivos deste trabalho e também trabalhos que estão sendo desenvolvidos na mesma linha de pesquisa e que envolvem o uso de PSO e extração de regras de séries temporais.

1.1 Estado da arte

Usar *Particle Swarm Optimization* (PSO) para resolver problemas envolvendo séries temporais vem sendo estudado e pesquisado por outros trabalhos da literatura [2],[13],[26],[20].

Em 2010 o trabalho apresentado por J. Behnamian, S.M.T. Fatemi Ghomi [2] desenvolveu um PSO híbrido para um novo modelo de regressão para previsões em séries temporais. A ideia do trabalho foi combinar um PSO com um algoritmo de *simulated anealing* (SA) para encontrar o melhor conjunto de parâmetros que ajusta o processo de previsão em séries temporais não lineares.

Já em 2011 Yao-Lin Huang e Shi-Jinn Horng [13] desenvolveram um modelo híbrido para prever as matrículas de alunos na Universidade do Alabama que combinou um PSO com previsão em séries temporais difusas.

Em 2012 um PSO com mecanismo de controle por *feedback* foi apresentado por W.K. Wong, S.Y.S. Leung e Z.X. Guo [26] para previsão de séries temporais. A ideia do projeto é cada partícula do PSO ter um conjunto de parâmetros de pesquisa que é controlado pela qualidade da partícula no PSO.

Em 2013 o trabalho apresentado por Singh e Borah [20] também desenvolveu um metodologia de previsão de séries temporais difusas através do uso de um PSO.

Neste cenário, aqui é proposto à combinação de um PSO com o uso de uma lista tabu, que registra as soluções já encontradas e são usadas para guiar o algoritmo de busca, para o reconhecimento e extração de regras em séries temporais. A metodologia proposta foi desenvolvida em CUDA da NVIDIA.

1.2 Objetivos

O objetivo geral deste projeto de mestrado é apresentar uma proposta de análise de séries temporais que permita auxiliar na previsão de tendências de crescimento ou queda de valores futuros através da extração de conhecimento implícitos nessas mesmas séries. Na composição deste objetivo é possível determinar alguns objetivos específicos onde enumera-se:

- Estudo e implementação do algoritmo inicialmente proposto por Eberhart [14] para o PSO;
- Combinação do PSO a uma lista tabu;
- Estudo da plataforma de computação paralela da NVIDIA (CUDA);
- Desenvolvimento do PSO em CUDA;
- Elaboração de um conjunto de testes para a verificação e ajuste do sistema projetado;
- Analise dos resultados experimentais de forma comparativa com a literatura.

1.3 Descrição do Documento

Esta dissertação e composta por seis capítulos que estão distribuídos da seguinte forma:

- Capítulo 2 - Descreve o Problema de Previsão de Séries Temporais e a extração de regras de séries temporais. Também introduz os conceitos de método de computação inteligente/evolutiva;
- Capítulo 3 - Detalha a metodologia proposta e apresenta o PSO com detalhes da implementação paralela analisando a arquitetura utilizada nas simulações;
- Capítulo 4 - Descreve as séries temporais utilizadas nas simulações;
- Capítulo 5 - Apresenta os resultados experimentais obtidos pela metodologia proposta utilizando as séries apresentadas no Capítulo 4;
- Capítulo 6 - Faz as conclusões sobre os resultados obtidos e também são apresentadas as contribuições e as limitações observadas.

Capítulo 2

Análise do Problema

Neste capítulo é descrito o problema de previsão e extração de regras de séries temporais, assim como introduz os conceitos de método de computação inteligente/evolutiva.

2.1 O problema de previsão e extração de regras

O sucesso ou o fracasso de um agente, que espera obter vantagens, é diretamente relacionado com o seu conhecimento das leis que regem a dinâmica da sociedade em que vive. Com as informações corretas, o agente é capaz de fazer a ação correta.

A decisão sobre o momento presente é um posicionamento para o futuro, quando é desejável maximizar algum objetivo. Por exemplo, um agente quer comprar ou vender uma ação em um mercado financeiro, onde o agente espera obter lucro com esta operação. Neste sentido, a previsão [4],[28],[27] é um elemento-chave na tomada de decisão.

Uma maneira de aprender o comportamento temporal de um determinado fenômeno é realizar um conjunto de observações no tempo e analisar tais dados históricos. Esses dados históricos constituem uma série temporal, comumente estudada pela estatística, matemática, processamento de sinal, econometria e outras ciências [3],[15].

Uma série temporal de valores reais X, pode ser vista como um processo estocástico [3],[24], que realiza um mapeamento,

$$X : T \times \Omega \to \mathbb{R},$$

tal que, para um valor fixo $t \in T$ e $\omega \in \Omega$, $X(t, \omega)$ é uma variável aleatória num espaço de probabilidade, onde T = 1,. . . , N é o conjunto de índice cronológico (ou simplesmente índice de tempo), ω são os eventos elementares e Ω é um espaço amostral.

Dado que uma série temporal pode ser definida como um conjunto de observações de um evento em uma granularidade de tempo, vários fenômenos naturais, ou mesmo artificiais, podem ser representados por uma série temporal. Entre esses fenômenos é possível destacar, por exemplo:

- Número de exportações em um ano de um dado setor industrial;
- Crescimento da população de um país em uma década;
- Número de passageiros por mês em uma companhia aérea;
- Venda diária de um determinado produto;
- etc.

Dependendo do exemplo, a série temporal pode ser decomposta em um conjunto de componentes que facilitarão a sua análise, esses componentes são:

- Tendência: Determina o padrão de comportamento da série temporal que pode ter seus valores ascendente, descendente ou mantendo-se constante ao longo do tempo;
- Ciclo: Representa oscilações de crescimento e queda que acontecem em intervalos irregulares de tempo;
- Sazonalidade: É a identificação de padrões de crescimento e queda que se repetem no tempo e em determinados intervalos regulares de tempo;

- Variação aleatória: São interferências irregulares que afetam a série temporal e que podem ser atribuídas a fatos inesperados como uma guerra, ou mesmo na introdução de novas leis que afetem o mercado financeiro.

Estatisticamente espera-se que uma série temporal contenha as informações necessárias para determinar as leis que regem a dinâmica temporal do fenômeno em observação, com a possível construção de um espaço de fase isomorfo para o fenômeno observado, através da escolha correta do tempo de atraso das séries temporais, como afirma o teorema de Takens [22],[12]. Com a decomposição da série temporal em componentes é possível analisar a série e possivelmente criar um modelo matemático que permita algum grau de previsão.

Neste trabalho é proposta uma metodologia para extrair regras de séries temporais, com particular interesse para o comportamento de padrões temporais que descrevam a tendência futura da série temporal e assim permitir a previsão da repetição destes padrões sem a necessidade de criar um modelo matemático formal que defina a série temporal.

Desta forma, um *Particle Swarm Optimization* (PSO) [14] combinado com uma lista tabu [16],[9] é usado para procurar as relações entre os tempos de atraso de uma dada série temporal e construir os padrões de comportamento, ou regras, do fenômeno gerador da série temporal.

2.2 Método de computação inteligente e evolutiva

Computação evolutiva é o nome dado à categoria de algoritmos baseados no processo de evolução por seleção natural apresentado por Charles Darwin. Em seu trabalho, Darwin apresentou a ideia de que em uma determinada população o indivíduo que compete de forma mais eficiente pelos recursos disponíveis tem maior chance de sobrevivência. Estar apto seria então um dos critérios de evolução deste indivíduo.

Outro critério para a evolução de um indivíduo é a forma com que este indivíduo se relaciona com o ambiente e com outros indivíduos. Dois indivíduos de uma mesma espécie que se desenvolvem em ambientes diferentes podem apresentar características evolutivas

diferentes.

Os indivíduos mais bem adaptados propagam suas características, mesmo que sejam características definidas pelo ambiente, através de seus descendentes e a tendência dos não bem adaptados é desaparecer sem colaborar com a evolução de sua população.

Baseados nessas ideias, cientistas passaram a tentar resolver problemas computacionais através de algoritmos evolutivos onde uma população de indivíduos é criada de forma aleatória, e cada indivíduo desta população representa uma solução tentativa para um determinado problema em análise.

Através de um processo de avaliação, que determina a adaptabilidade deste individuo, e de operações que realizam cruzamentos e mutações é possível que cada indivíduo evolua, ou mesmo desapareça, fazendo com que a população passe a apresentar indivíduos mais aptos e consequentemente chegar a uma solução aceitável para o problema.

Na literatura, os algoritmos que implementam esses conceitos são classificados como parte de uma família de algoritmos evolucionários e são divididos de forma geral em Algoritmos Genéticos, Estratégias Evolucionárias, Programação Evolutiva e Programação Genética [10]. Um *Particle Swarm Optiminization* (PSO) pode ser classificado como um algoritmo evolucionário que mantém vínculos tanto com os algoritmos genéticos quanto com a programação evolucionária [14].

Neste trabalho um PSO será aplicado na extração de conhecimentos de séries temporais e por isso os Algoritmos Genéticos, Estratégias Evolucionárias, Programação Evolutiva e Programação Genética não serão detalhados.

2.2.1 Particle Swarm Optiminization (PSO)

O PSO foi originalmente apresentado em um artigo publicado por Kennedy e Eberhart em 1995 [14]. Neste artigo foi apresentada uma proposta de simulação estocástica baseada no comportamento de um enxame que representa um modelo social simplificado, como um enxame de abelhas, pássaros e cardumes de peixes e mesmo sendo intuitivo e fácil de ser

implementado o PSO pode apresentar resultados mais satisfatórios que outros métodos mais complexos de simulação [14].

O algoritmo do PSO realiza uma busca por soluções em um espaço de busca muito amplo e para ter sucesso implementa uma cooperação entre os elementos do enxame, como em uma revoada de pássaros em busca de alimento onde o conhecimento que um pássaro tem sobre onde está a comida é potencializado pelo conhecimentos dos outros pássaros do enxame [14].

O PSO é formado por um enxame de partículas, partícula é um termo genérico para um componente do enxame, onde cada partícula representa uma possível solução para o problema em análise. Cada partícula tem uma posição definida dentro do espaço de busca e uma velocidade que controla a variação de sua posição, esses valores são representados através de vetores. O enxame sobreviverá durante um determinado número de iterações, que pode ser um valor pré-definido, até encontrar uma solução ou entrar em processo de estagnação.

Na implementação original do PSO as partículas serão atualizadas durante as iterações a partir das informações observadas durante a evolução do enxame. Para avaliar as partículas será definido um fator de qualidade, chamado de $fitness$, que determinará se uma partícula está em uma posição mais favorável que as outras. Cada partícula é avaliada individualmente, através de seu $fitness$, e a melhor posição encontrada por cada partícula até o momento será armazenada, essa informação é chamado de *pbest* e a melhor posição encontrada pelo enxame também será armazenada, essa informação é chamada de *gbest*. Essas informações são importantes na atualização da posição de cada partícula no enxame onde o *pbest* representa os máximos locais da posição de cada partícula e o *gbest* representa o máximo global da posição encontrado pelo enxame.

Suponha que o tamanho do enxame é dado por s. Cada indivíduo ($1 \leq i \leq s$) tem uma posição atual no espaço de busca (x_i), uma velocidade atual (v_i) e uma melhor posição pessoal no espaço de busca (y_i). Assumindo-se que a função de $fitness$ deve ser maximizada, o enxame consiste de s partículas, e a cada iteração, a velocidade de cada partícula do enxame é atualizado por,

$$v_{i,j}(t+1) = wv_{i,j}(t) + c_1 r_1[y_{i,j}(t) - x_{i,j}(t)] + c_2 r_2[\hat{y}_j(t) - x_{i,j}(t)] \ , \tag{2.1}$$

onde $i \in 1, 2, \ldots, s$; $\hat{y}_j(t)$ denota a melhor posição atual no espaço de busca (encontrado pelo enxame como um todo, ou seja, o *gbest*), $y_{i,j}(t)$ representa a melhor posição pessoal no espaço de busca (encontrado por cada partícula do enxame, ou seja, o *pbest*), $v_{i,j}$ é a velocidade da j-ésima dimensão da i-ésima partículas, c_1 e c_2 representam os coeficientes de aceleração, que controlam o quanto uma partícula vai mover-se em uma única iteração, e $r_1 \sim U(0,1)$ and $r_2 \sim U(0,1)$ são os elementos de duas sequências aleatória uniformes no intervalo [0,1]. É importante perceber que tanto c_1, c_2, r_1 e r_1 determinam o grau de influência de máximos locais de cada partícula (*pbest*) e máximo globais do enxame (*gbest*) na nova posição da partícula. O termo w é denominado coeficiente de inércia, em que este valor é tipicamente configurado para variar linearmente desde 1 até quase 0 durante o decurso do processo, ou pode ser definido como um valor constante. Este coeficiente é derivado do ajuste de temperatura encontrado em *SimulatedAnnealingAlgorithms* [14], [23].

Assim, a nova posição da partícula é atualizada pela equação,

$$x_i(t+1) = x_i(t) + v_i(t+1). \tag{2.2}$$

A melhor posição de cada partícula (y_i) e a melhor posição encontrada por qualquer partícula durante todas as iterações anteriores ($\hat{y}_i$) são atualizadas pelas Equações 2.3 e 2.4, respectivamente.

$$y_i(t+1) = \begin{cases} y_i(t) & \text{se } f(x_i(t+1)) \leq f(y_i(t)), \\ x_i(t+1) & \text{caso contrário.} \end{cases} \tag{2.3}$$

$$\hat{y}(t+1) = \text{argmax} f(y_i(t+1)). \tag{2.4}$$

O termo v_i é normalizado no intervalo $[-v_{max}, v_{max}]$ a fim de reduzir a probabilidade das partículas saírem do espaço de busca. Este mecanismo não restringe os valores de x_i no intervalo de v_i, apenas limita a distância máxima que uma partícula irá mover durante cada iteração [14],[23]. A cada geração as posições das partículas são atualizadas de forma que um regime de cooperação ocorra entre elas, diferente do regime de competição existente em outras estratégias evolucionárias, o que torna a implementação de um PSO um processo simples, como sumarizado no gráfico apresentado na Figura 2.1 e que será detalhado no Capítulo 3.

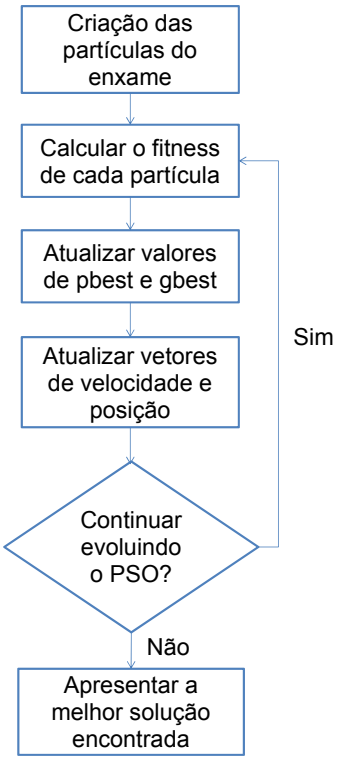

Figura 2.1: Fluxo de execução de um PSO.

2.2.2 PSO e Problemas Multimodais

Na natureza, fonte de inspiração para a criação da computação evolutiva, ou inspirada na natureza, várias espécies podem apresentar mais de um padrão de evolução diferente e isso pode depender de uma grande quantidade de fatores como, por exemplo, a adaptação do indivíduo às condições locais onde vive. Essa perspectiva abre a possibilidade de um algoritmo de computação evolutiva apresentar mais de uma solução para um determinado problema.

Em um PSO, um problema multimodal pode não ter todas as respostas encontradas já que normalmente as partículas do enxame seguem aquela que apresenta o melhor $fitness$ aparente. Diante disso a avaliação da qualidade de um indivíduo, o $fitness$, poderá levar em consideração outros fatores para dessa forma explorar mais o espaço de busca e permitir

que grupos de partículas possam convergir em direções diferentes e com isso diminuir a possibilidade da simulação ficar presa em máximos locais.

Dependendo do tamanho do espaço de busca as outras soluções, do problema multimodal, podem não ser acessadas pelas partículas do enxame e para evitar que isso aconteça o PSO pode ser segmentado em grupos, onde cada grupo é guiado por partículas distintas, ou ter suas partículas reiniciadas com suas posições definidas de forma aleatória após encontrar uma das soluções esperadas para o problema.

Reiniciar as partículas do PSO de forma aleatória pode posicionar as partículas em uma posição próxima de outra solução para o problema multimodal, e para evitar as mesmas soluções encontradas anteriormente o processo de reinicialização das partículas pode evitar locais previamente pesquisados.

Segmentar o PSO em grupos de partículas permite posicionar cada grupo em um local diferente do espaço de busca, onde cada grupo será guiado por uma partícula diferente, e dessa forma ter cada grupo buscando, de forma paralela, uma solução para o problema multimodal. A principal vantagem da segmentação em grupos está na paralelização do enxame, como mostrado na Figura 2.2 que apresenta a quantidade de PSOs sendo executados em relação ao tempo.

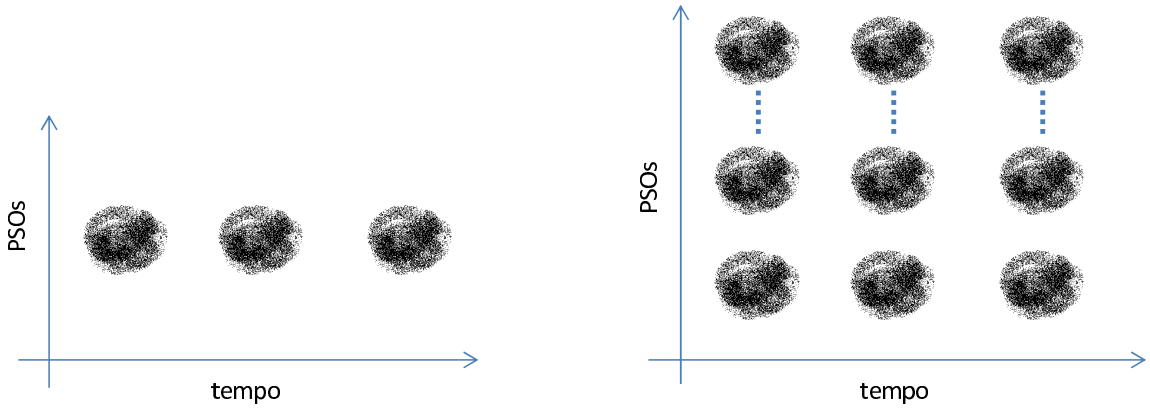

Figura 2.2: Número de PSOs em relação ao tempo.

2.2.3 PSO e Implementação Paralela

Neste trabalho será apresentada uma implementação paralela onde um número determinado de PSOs será inicializado de forma simultânea e cada PSO fornecerá informações sobre as posições de soluções já encontradas para que as partículas de um PSO evitem o espaço de busca ocupado pelas partículas dos outros PSOs. A implementação paralela de um PSO pode ser feita através de diferentes artifícios computacionais e isso dependerá da tecnologia utilizada.

Uma implementação paralela pode ser desenvolvida levando em consideração o processador do computador (CPU) como alvo da execução do programa. Linguagens de programação como C e Java permitem esse tipo de desenvolvimento através de um recurso chamado *Thread*. Uma *Thread* pode ser definida como uma tarefa que deverá ser executada e por mais simples que seja o programa ele por si só já é considerado uma tarefa.

Em ambientes multitarefas várias tarefas podem ser executadas de forma simultânea e para que isso aconteça elas devem ser identificadas, no programa, pelo programador e serem codificadas como uma *Thread*. A Figura 2.3 apresenta a ideia de execução de um programa com uma única *Thread* onde os processos são executados de forma sequencial e de um programa com várias *Threads* onde diferentes processos são executados de forma sequencial, mas ao mesmo tempo outros conjuntos de processos estão sendo executados de forma paralela.

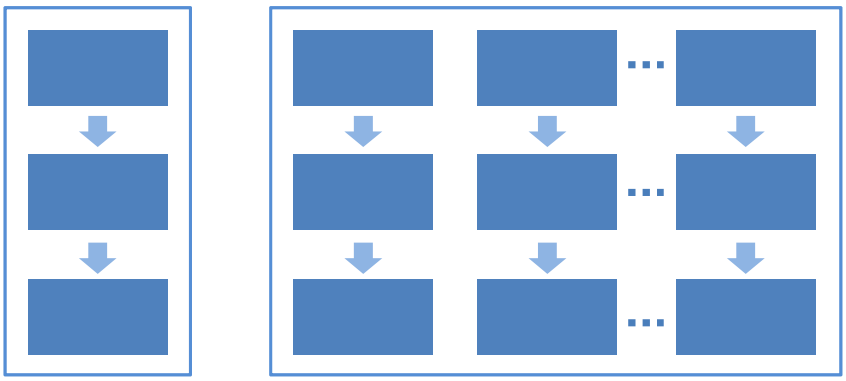

Figura 2.3: Representação de um programa Single Thread e Multi Thread.

Outra forma de implementação paralela é através de um processo inovador que permite utilizar o processador das placas de vídeo como alvo da execução do programa, ous seja, usar a Unidade de Processamento Gráfico de Propósito Geral (GPGPU). Utilizar a GPGPU revolucionou o mercado de computação gráfica/científica por permitir o processamento geral, não apenas gráfico, e vem revolucionando o desenvolvimento de sistemas de simulações computacionais[25].

Existem diversas tecnologias que permitem implementar programação paralela para GPGPUs e com o crescimento da capacidade de *hardware* apresentada pelos novos dispositivos gráficos, além da grande disponibilidade de dispositivos compatíveis, encontrados inclusive em notebooks [18], faz com que esse tipo de desenvolvimento se torne mais acessível. A Figura 2.4 compara o crescimento da velocidade de processamento de CPUS e GPGPUs ao longo do tempo.

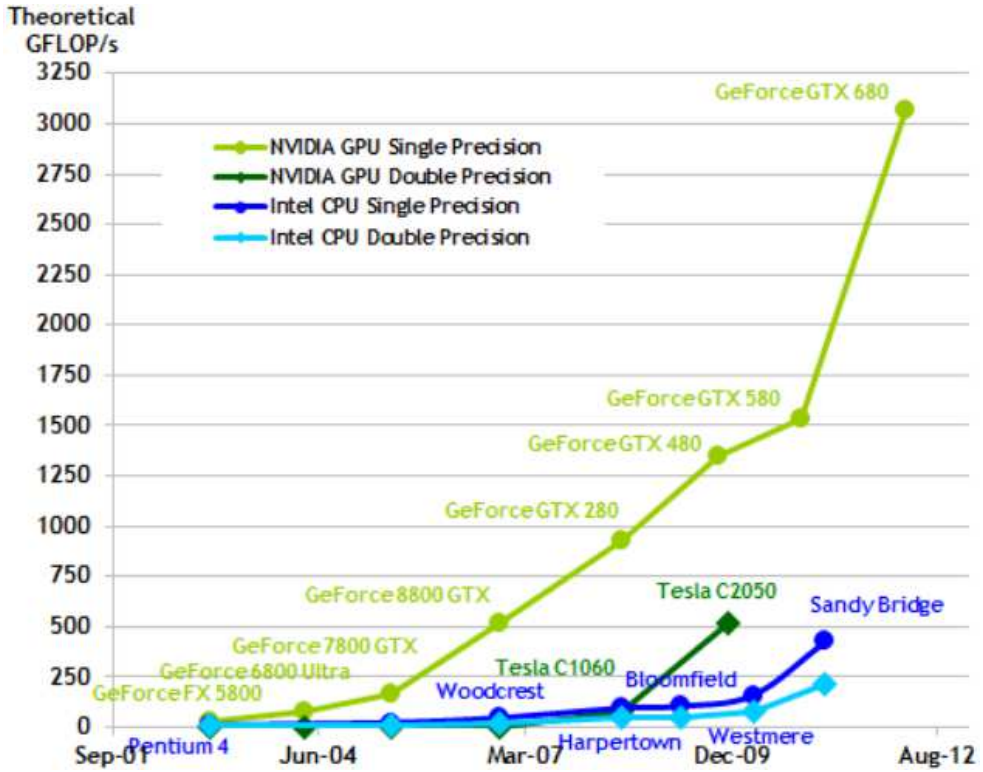

Figura 2.4: Comparação da velocidade de processamento CPU x GPGPU [8].

Com a evolução da tecnologia para GPGPU a vantagem do uso deste tipo de processamento foi percebida e em algumas situações, principalmente onde cálculos envolvendo vetores

precisam ser executados, a GPGPU tem um desempenho muito maior que uma CPU. Apesar de tudo isso, escrever códigos que utilizem o poder de processamento de uma GPGPU não era trivial e o desenvolvedor deveria estar familiarizado com as especificações técnicas de cada dispositivos [18].

Para facilitar e popularizar o uso de GPGPUs como ferramenta de apoio ao desenvolvimento de programas paralelos a NVIDIA criou a *ComputeUnifiedDeviceArchitecture* (CUDA), uma API que funciona como uma extensão a linguagem C e usada em dispositivos compatíveis com a plataforma CUDA, fabricados pela própria NVIDIA [18].

A NVIDIA desenvolveu CUDA como uma plataforma de computação paralela para uso geral o que a torna uma ferramenta mais apta a resolver problemas computacionais complexos e massivos do que uma CPU. A arquitetura apresentada pelas GPGPUs da NVIDIA difere significativamente da arquitetura normalmente encontrada nas CPUs convencionais. A Figura 2.5 apresenta a estrutura interna típica de uma CPU e de uma GPGPU.

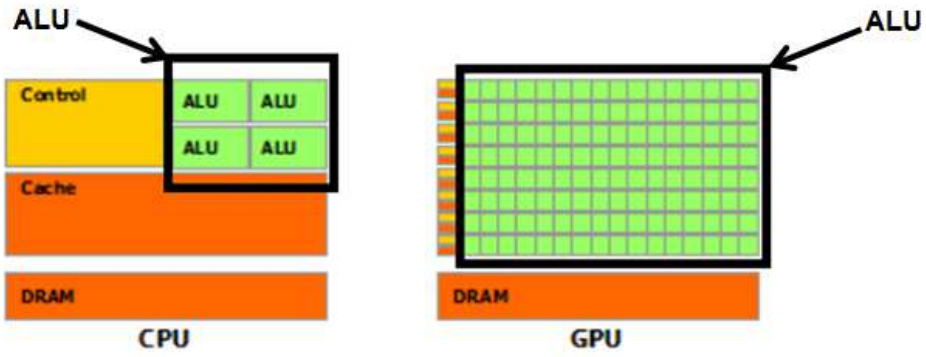

Figura 2.5: Estrutura interna típica de uma CPU e uma GPGPU [8].

A Figura 2.5 mostra que enquanto uma CPU reserva um espaço maior para as memórias e unidades de controle uma GPGPU reserva muito mais espaço para unidades de lógica e aritmética (ALU). E justamente por ter uma quantidade maior de ALUs que torna a GPGPU mais apta a executar programas que necessitem de programação paralela extrema.

Além de facilitar o acesso ao poder de processamento da GPGPU, a NVIDIA desenvolveu também um compilador que deve ser utilizado em arquivos de código fontes da linguagem C que usem a extensão CUDA. Também está disponível para os usuários de Linux um

ambiente de desenvolvimento chamado *Nsight* que é um projeto desenvolvido utilizando o projeto Eclipse como base [7]. A Listagem 2.1 apresenta um programa feito na linguagem C, que faz a soma de dois vetores, fazendo uso de comandos da extensão CUDA da NVIDIA.

Listagem 2.1: Exemplo de um programa C/CUDA para somar 2 vetores

```
1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3  #include <stdio.h>
4  __global__ void addKernel(int *c, const int *a, const int *b) {
5      int i = blockDim.x*blockIdx.x + threadIdx.x;
6      c[i] = a[i] + b[i];
7  }
8  int main() {
9      const int arraySize = 15;
10     const int a[arraySize] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
11     const int b[arraySize] = { 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70 };
12     int c[arraySize] = { 0 };
13     int *dev_a = 0;
14     int *dev_b = 0;
15     int *dev_c = 0;
16     cudaMalloc((void**)&dev_c, arraySize * sizeof(int));
17     cudaMalloc((void**)&dev_a, arraySize * sizeof(int));
18     cudaMalloc((void**)&dev_b, arraySize * sizeof(int));
19     cudaMemcpy(dev_a, a, arraySize * sizeof(int), cudaMemcpyHostToDevice);
20     cudaMemcpy(dev_b, b, arraySize * sizeof(int), cudaMemcpyHostToDevice);
21     addKernel<<<1, arraySize>>>(dev_c, dev_a, dev_b);
22     cudaThreadSynchronize();
23     cudaMemcpy(c, dev_c, arraySize * sizeof(int), cudaMemcpyDeviceToHost);
24     for(int i=0; i<arraySize; i++) {
25         printf("a[%d]=%d + b[%d]=%d = c[%d]=%d\n", i, a[i], i, b[i], i, c[i]);
26     }
27     cudaFree(dev_c);
```

```
28      cudaFree(dev_a);
29      cudaFree(dev_b);
30      return 0;
31  }
```

De forma simplificada o código da Listagem 2.1 pode ser explicado como um conjunto de operações que serão executados na CPU, o método main, e um conjunto de operações que serão executados na GPGPU, através do método *addKernel*, que ao receber o modificador `__global__` passa a ser identificado como um *kernel*[1] CUDA.

O método *main* inicializa variáveis na CPU e em seguida transfere esses valores para a memória da GPU, linhas 19 e 20, para que possam ser acessados pelos processos em execução na GPU, o *kernel addKernel* na linha 4. Uma abordagem mais detalhada do código da Listagem 2.1 e da plataforma CUDA pode ser encontrada no Apêndice A.

[1]Um kernel é uma ou um conjunto de *threads* que serão executadas em paralelo pela GPGPU

Capítulo 3

Descrição da metodologia

Neste capítulo será detalha a metodologia proposta e apresenta o PSO com detalhes da implementação paralela analisando a arquitetura utilizada nas simulações.

3.1 Método Proposto

Uma previsão de tendência de séries temporais tenta identificar o comportamento futuro de crescimento ou queda da série temporal observada e dependendo dos resultados encontrados pode funcionar como uma ferramenta de apoio à tomada de decisão.

Neste trabalho o processo de previsão será baseado na previsão de tendências e por isso no início do processo a série temporal observada é codificada em uma sequência binária, que é uma forma de mapear os comportamentos de crescimento e queda das séries temporais.

No processo de conversão para uma codificação binária o primeiro elemento é fixado em 0 (zero) e o segundo elemento da série será 0 se o seu valor for menor que o valor do primeiro ponto, ou 1 (um) se o seu valor for maior ou igual ao valor do primeiro ponto. Essa metodologia é aplicada para todos os pontos da série temporal, onde o valor de um ponto é definido por uma comparação com o valor do seu predecessor. A Equação 3.1 descreve a

codificação da serie temporal:

$$Z_{t+1} = \begin{cases} 0, & \text{se } X_{t+i} \leq X_t, \\ 1, & \text{se } X_{t+i} \geq X_t \end{cases} \tag{3.1}$$

onde Z é a série *binarizada* e X a série temporal observada.

Neste trabalho vários PSOs serão executados de forma paralela e cada PSO é composto por uma população de partículas, onde cada partícula representará uma janela de observação que varrerá todo o conjunto de treinamento da série temporal, a divisão da série em um conjunto de treinamento e e testes é definida no Capítulo 5. Cada partícula além da janela de reconhecimento também é composta por um valor de previsão. A janela de reconhecimento é uma sequencia numérica binária que apresenta um possível padrão de comportamento persistente da série temporal. Se a partícula encontrar um padrão, o próximo valor após esse padrão será o seu valor de predição. A Figura 3.1 descreve uma partícula, onde ela é representada por um vetor contendo a janela de reconhecimento e o valor de predição.

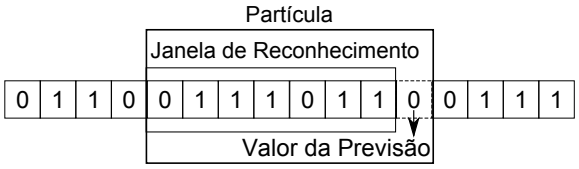

Figura 3.1: Representação de uma partícula.

A qualidade de cada partícula é verificada comparando elemento por elemento da janela de reconhecimento e o conjunto de valores correspondentes da série temporal sendo analisada, já convertida para a codificação em binário ou *binarizada*. A partícula é comparada com todos os valores do conjunto de treinamento da série, como mostrado na Figura 3.2. A comparação é feita usando a distância euclidiana descrita na Equação 3.2.

Todas as partículas são iniciadas com 0 ou 1, mas com a evolução do PSO os elementos podem assumir qualquer valor no intervalo [0,1]. É considerado que é encontrado uma correspondência entre a janela de reconhecimento da partícula e o intervalo analisado da série temporal *binarizada* se a distância euclidiana não ultrapassar 0.30 na comparação elemento a elemento. Este caso acontece quando pelo menos 80% da janela de reconhecimento combinam

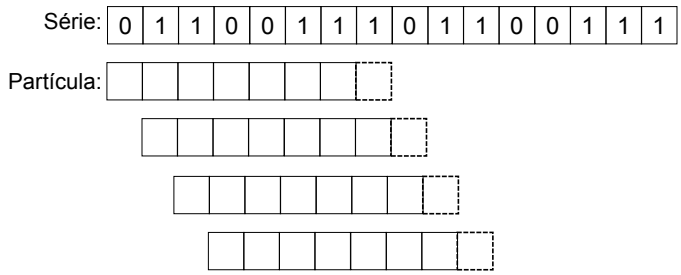

Figura 3.2: Comparação entre a partícula e a série temporal.

com os valores em observação da série. Essas margens de tolerância de 30% na distância euclidiana e 80% de correspondência da janela de reconhecimento foram definidas baseadas em testes realizados com as séries artificiais, que serão apresentadas no Capítulo 4, e para deixar o sistema mais generalista evitando assim o problema de *overfitting* que ocorre quando o modelo gerado se ajusta ao conjunto de dados analisado de forma que não apresente bons resultado quanto testado em conjuntos de dados diferentes.

Ocorrendo a correspondência de 80% da janela de reconhecimento com o intervalo analisado da série temporal *binarizada* uma regra foi encontrada e os valores do intervalo analisado da série temporal e o valor de previsão, também da série temporal, são armazenados na lista tabu. O uso da lista tabu será detalhado no cálculo do $fitness$ da partícula.

A verificação do desempenho de cada partícula é feita a cada iteração do PSO com o valor de predição alcançado por todas as partículas. A distância euclidiana é calculada elemento por elemento de cada partícula através da Equação 3.2,

$$d_i = \sqrt{(p_i - Z_i)^2} \tag{3.2}$$

onde d representa a distância euclidiana entre os elementos, i é o índice do elemento atual, p_i é o elemento i da janela de reconhecimento da partícula e Z_i é o elemento i do intervalo em observação da serie temporal *binarizada*.

Em cada etapa, verifica-se a correspondência entre a partícula e a série temporal. Os atributos do enxame são definidos de acordo com a correspondência dos valores. Cada partícula tem dois atributos: o *suporte* e a *acurácia*. O atributo de suporte é definido pela quantidade de correspondências entre a janela de reconhecimento das partículas e os

dados das séries temporais. O atributo de acurácia é a precisão da previsão das partículas no seu suporte. Portanto, se na iteração do PSO ocorrer uma correspondência, o suporte da partícula é incrementado em uma unidade e a partícula pode ser usada para fazer uma previsão. Se a previsão estiver correta em algum padrão observado da série temporal, a acurácia da partícula é incrementada em uma unidade e, assim, uma regra é encontrada.

Para impedir que PSOs diferentes encontrem o mesmo conjunto de regras uma lista tabu foi criada para registrar as regras já encontradas por todos os PSOs e no final de cada iteração, de cada PSO, as regras encontradas são coletadas e armazenadas nesta lista tabu. A lista será consultada quando o $fitness$ de cada partícula for calculado e modificará o valor deste $fitness$ de forma que as partículas dos PSOs sejam afastadas dos locais onde essas regras foram encontradas.

Seja uma partícula com uma janela de reconhecimento composta pelos valores: 0.2, 0.7, 0.1, 0.0, 0.3, 0.8 e seu valor de previsão igual a 0.7. Seja a janela de dados da série temporal composta pelos valores: 0, 1, 1, 0, 0, 1, 1 e do valor futuro igual a 1, como observado na Figura 3.3.

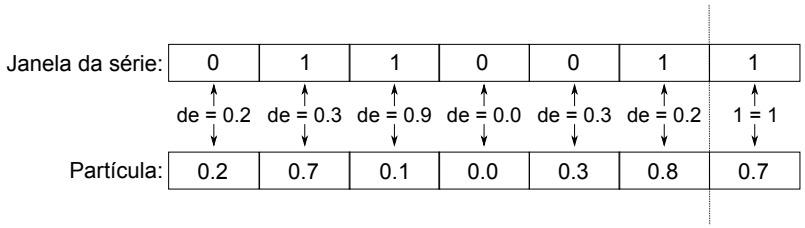

Figura 3.3: Comparação entre a partícula e a janela de observação.

Neste exemplo da Figura 3.3, a soma de correspondências é igual a 5, uma vez que os valores de 0.2, 0.7, 0.0, 0.3 e 0.8 da partícula coincidiram com os elementos da janela de dados da série temporal correspondente, isto é, tiveram a distância Euclidiana (de) menor ou igual a 0.30. Considerando o caso de 80% de correspondência, esta partícula é capaz de representar um padrão observado na série histórica de dados, e nesta situação o atributo de suporte da partícula é incrementado. Se o suporte é incrementado então o valor de predição da partícula será verificado, se o valor de previsão da partícula combinar como o valor futuro da janela de dados da série temporal, o atributo de acurácia da partícula é incrementado.

Após a partícula percorrer toda a série temporal é possível verificar a sua qualidade através de seu $fitness$. A definição de uma função para calcular o $fitness$ de uma partícula pode variar entre problemas distintos. Nesta proposta o $fitness$ da partícula se baseia inicialmente no suporte, na acuraria e no *Mean Squared Error* (MSE), que representa a distância entre a partícula e a janela de observação da regra, que é calculado para cada partícula através da Equação 3.3,

$$MSE_i = \frac{1}{N_{TS}} \sum_{j=1}^{N_{TS}} \sum_{k=0}^{T_K} (p_{i,k} - Z_{j+k})^2 \tag{3.3}$$

onde N_{TS} é o número de pontos da série temporal analisada, MSE_i representa a o Mean Squared Error entre a partícula i e a série temporal *binarizada*, $p_{i,j}$ representa o valor k da janela de reconhecimento da particular i, T_K representa o tamanho da janela de observação da partícula e Z_{j+k} é o valor do elemento do intervalo em observação da serie temporal *binarizada*. Portanto, com base nas medidas do suporte, da acurácia e do MSE é possível definir uma função de $fitness$,

$$fitness_i = \frac{1}{3} \left(\frac{suporte_i}{N_{TS}} + \frac{acuracia_i}{suporte_i} + \frac{1}{1 + MSE_i} \right), \tag{3.4}$$

onde o primeiro termo da equação considera o suporte da partícula normalizado pelo número total de pontos da série temporal analisados. O segundo termo considera valor da acurácia normalizado pelo seu suporte e o terceiro termo considera a distância entre a janela de reconhecimento da partícula e o padrão encontrado na série temporal.

Uma vez calculado o $fitness$ da partícula em execução é verificado a existência de elementos na lista tabu e se existir será calculado a distância entre a partícula em execução e todas as regras armazenadas na lista tabu. A distância é calculada através do *Mean Squared Error* (MSE) entre os elementos da partícula e os elementos de cada regra na lista tabu como mostrado na Equação 3.5,

$$dT_i = \frac{1}{N_T} \sum_{j=1}^{T_J} \sum_{k=1}^{N_K} (p_{i,j} - t_{k,j})^2 \tag{3.5}$$

onde N_K representa a quantidade de regras encontradas na lista tabu, dT é a distância entre a partícula i e todas as regras da lista tabu, j representa a posição de cada valor da partícula

ou regra da lista tabu, T_J representa o tamanho da janela de observação da partícula, $p_{i,j}$ representa o valor j da partícula i, $t_{k,j}$ representa o elemento j da regra k da lista tabu.

Uma vez calculada a distância, entre a partícula e a lista tabu, ela é utilizada como um fator de redução no $fitness$ da partícula. Quanto menor a distância mais próxima a partícula está de um espaço de busca que já foi, ou está sendo, explorado por partículas de outro PSO e a redução em seu $fitness$ será calculado como mostrado na Equação 3.6,

$$fitness_i = fitness_i - \frac{1}{1 + dT_i} \tag{3.6}$$

onde $fitness_i$ é o valor do fitness calculado para a partícula, e i é o índice da partícula sendo avaliada e dT_i é a distância tabu calculada para a partícula i.

Em cada iteração do PSO, as partículas são comparadas com a série e seu fitness é calculado. O sistema evolui o enxame até que um número máximo de iterações, informado pelo usuário no início do processo seja alcançado. No entanto, a população de partículas pode ser submetida a algum processo de estagnação em sua evolução [5]. Devido a esse problema de estagnação a evolução das partículas da população é verificada após um determinado número de iterações pela Equação 3.7,

$$Stagnation = \frac{(\overline{fitness}_u - \overline{fitness}_p)}{iteracao}, \tag{3.7}$$

onde $\overline{fitness}_u$ é a aptidão média da última iteração processada e $\overline{fitness}_p$ é a aptidão média da primeira iteração no intervalo do período observado e $iteracao$ representa a iteração atual. Desta forma, o critério de estagnação é uma condição de parada para o PSO. Após cada iteração, a partícula com o melhor $fitness$ é removida do PSO e as suas respectivas regras são armazenadas na lista de tabu [17]. Para substituir a partícula removida uma nova partícula é inicializada aleatoriamente e inserida no enxame, mantendo a população do PSO com um número constante de partículas. A lista tabu é usada para ter certeza de que o PSO não encontrará sempre as mesmas regras e garantir que as partículas dos outros PSOs em execução não explorem o mesmo espaço de busca.

Após o final de todas as iterações, ou após o PSO ter parado por entrar em estagnação, as regras constantes da lista tabu são testadas usando o conjunto de testes (30% finais dos

dados da série em observação). A verificação é feita comparando a janela de reconhecimento das regras contidas na lista tabu e o conjunto de teste da série temporal, tal como descrito na Figura 3.4.

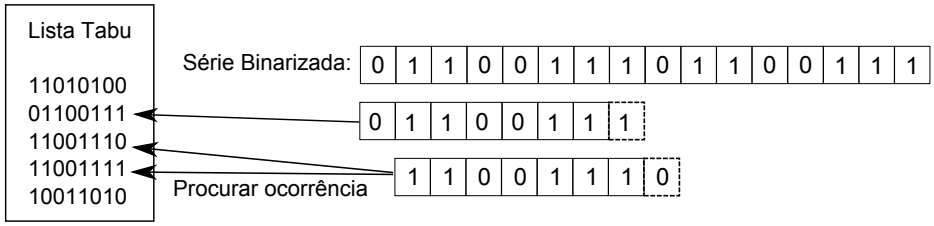

Figura 3.4: Procedimento da lista tabu.

No entanto, as regras foram construídas com uma margem de tolerância para corresponder com os dados de séries temporais, 30% de desvio na distância euclidiana e de apenas 80% do tamanho total da janela de reconhecimento, garantindo um maior poder de generalização em relação a uma correspondência de 100% da janela de reconhecimento da partícula e evitando o problema de $overfitting$.

Assim, a qualidade das regras deve ser verificada seguindo um conjunto de diretrizes, definidas por esse trabalho, que determina que para cada regra de previsão estimada um contador de acerto, erro ou incerteza é atualizado. Se o valor da previsão da regra selecionada é igual ao valor futuro da série temporal (ou seja, uma previsão correta) um contador de acertos é incrementado. Caso contrário, uma predição incorreta foi feita, e um contador de erro é incrementado. Sempre que o sistema não está apto a fazer uma predição, um contador de incerteza é incrementado. Dada uma janela de observação

- Se apenas uma regra da lista tabu corresponde com os dados da série temporal, então esta regra é usada para previsão da série temporal, e o contador de acertos é incrementado, $contAcerto = contAcerto + 1$;
- Se nenhuma regra da lista tabu corresponde com os dados da série temporal, então o sistema não reconhece o padrão mostrado na série temporal e por isso não pode fazer predição, e o contador de incertezas é incrementado, $contIncerteza = contIncerteza + 1$;

- Se mais de uma regra da lista tabu corresponde com os dados da série temporal, então para cada uma das regras é calculado uma medida de qualidade em relação a um modelo de passeio aleatório[21],

$$qualidade = suporte * \frac{(acuracia - 50\%)}{50}, \tag{3.8}$$

onde o *suporte* e a *acuracia* estão em valores percentuais. A regra, que atingiu o maior valor absoluto de qualidade vai ser utilizada para a previsão, isto é, a regra que estiver mais longe de um passeio aleatório. Observe que as medidas de qualidade podem ser positivo, negativo e zero. Se a qualidade é maior do que zero (ou seja, positivo) então a regra selecionada é melhor do que um passeio aleatório e serão empregados diretamente para previsão. Se a qualidade for menor que zero (isto é negativo), então a regra selecionada é pior do que um passeio aleatório. No entanto, como a predição de um passo a frente é uma informação binária, se a qualidade for negativa, então o valor complementar será utilizado para a predição, isto é, se o valor de previsão da regra for 1 será usado 0 e se o valor de previsão for 0 será usado 1. Finalmente, se a qualidade é igual à zero, então a regra selecionada é um passeio aleatório. Neste último caso, não é possível fazer previsões com esta regra e o contador de incerteza também é incrementado em 1.

Capítulo 4

Séries Utilizadas

Neste capítulo são apresentadas as séries temporais utilizadas para avaliar a metodologia proposta. Um conjunto de oito séries temporais, artificiais, composto por duas séries temporais determinísticas e seis estocásticas foi criado. Além das séries artificiais apresentadas também foram utilizadas séries temporais baseadas em dados reais. As séries são: Sunspots, Down Jones, S&P500 e Brilho de Estrela.

4.1 Séries Temporais Artificiais

As simulações com séries temporais artificiais ajudaram a analisar os resultados apresentados pelas simulações realizadas com as séries baseadas em dados reais, porque os resultados esperados pelas simulaçãos com as séries artificiais eram conhecidos antecipadamente. Essas séries também foram importantes na definição das margens de tolerância usadas para definir a correspondência entre as partículas e o intervalo observado da série temporal.

O processo de criação das séries artificiais determinísticas foi definido por esse trabalho e estas séries foram geradas com base nas sequências binárias 10110010 (série A) e 01100111 (série B), como mostrado na Figura 4.1, repetindo cada sequência até ficar com um total de 1000 registros cada uma.

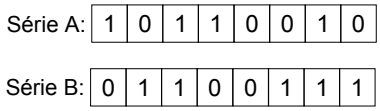

Figura 4.1: Séries artificiais.

Um ruído, representado por um componente aleatório $\sim U(0, 1)$, foi introduzido nestas séries para torná-las estocásticas. Assim, mais seis séries temporais foram criadas com base em um ruído de 1%, 5%, 10%, 15% e 20%. Se o termo do ruído é igual a 1%, por exemplo, então aproximadamente dez valores são invertidos (1% de 1000). Ou seja, o 0 vira 1 e o 1 vira em 0 em 1% dos dados, como exemplificado na Figura 4.2.

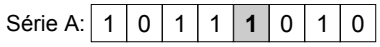

Figura 4.2: Exemplo da série A com alteração de ruído destacado.

Na Figura 4.3 estão representados os gráficos para os 100 pontos iniciais da série A e suas variações.

Algumas estatísticas para a série A são exibidas na Tabela 4.1, e por apresentar valores aproximados as estatísticas das séries A com ruído não serão apresentadas.

Tabela 4.1: Estatísticas da série A

Estatísticas	Valores
Quantidade de Pontos	1000
Valor Máximo	1
Valor Mínimo	0
Valor Médio	0,5
Variância	0,25
Desvio Padrão	0,50

Na Figura 4.4 estão representados os gráficos para os 100 pontos iniciais da série B e suas variações.

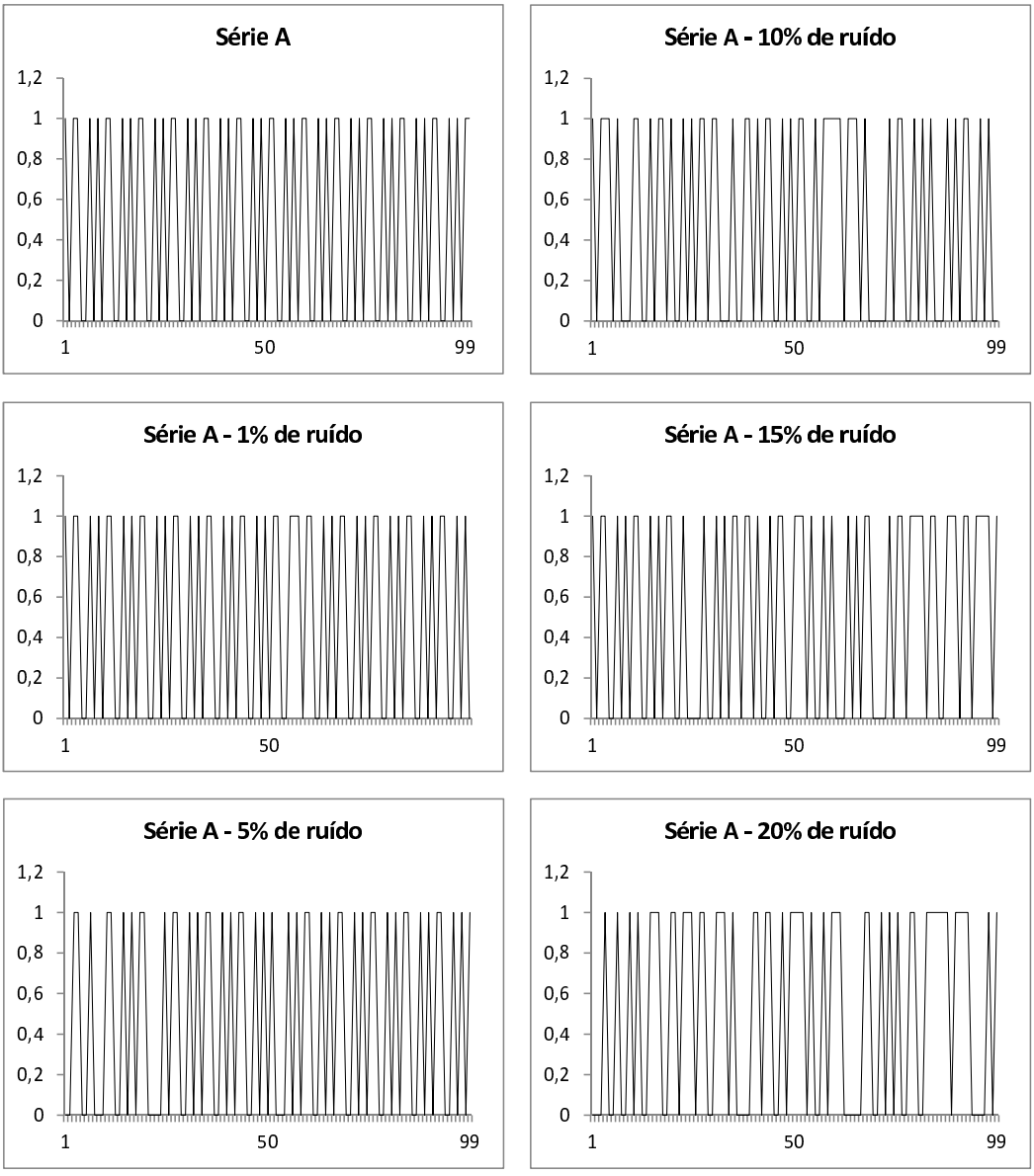

Figura 4.3: Gráficos da série A e suas variações.

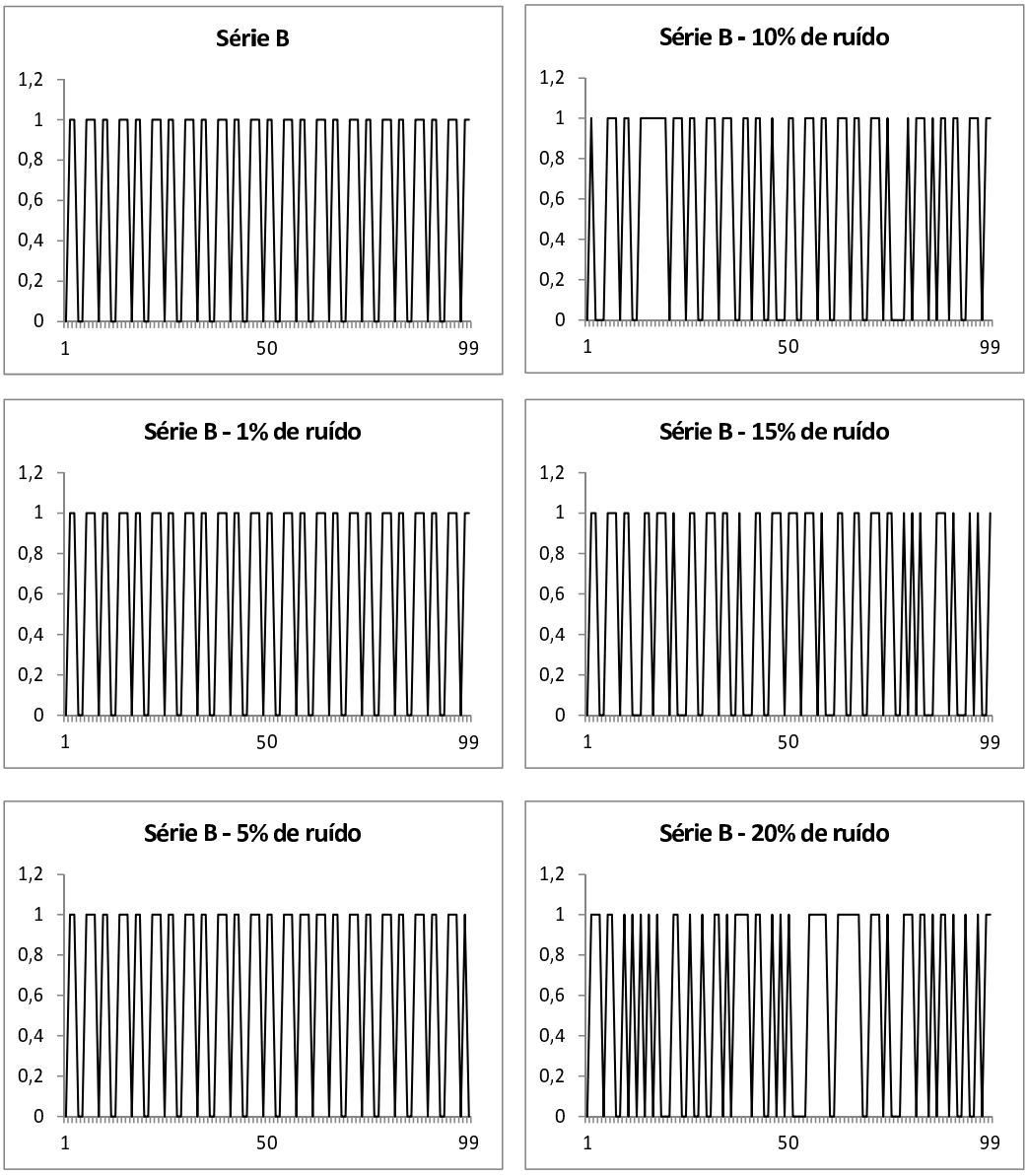

Figura 4.4: Gráficos da série B e suas variações.

Algumas estatísticas para a série B são exibidas na Tabela 4.2 e por apresentar valores aproximados as estatísticas das séries B com ruído não serão apresentadas.

Tabela 4.2: Estatísticas da série B

Estatísticas	Valores
Quantidade de Pontos	1000
Valor Máximo	1
Valor Mínimo	0
Valor Médio	0,63
Variância	0,48
Desvio Padrão	0,23

4.2 Séries temporais baseadas em dados reais

As séries baseadas em dados reais escolhidas para testar a metodologia são séries temporais que foram criadas baseadas nos registros de eventos naturais ou no mercado financeiro. Estas séries são encontradas com em uma série de estudos prévios baseados em seus valores o que torna a análise dos resultados apresentados por esse trabalho mais fácil de ser realizado. A seguir será feito uma breve descrição de cada uma dessas séries.

4.2.1 Down Jones Industrial Average (DJIA)

É uma série que representa o registro de variação do índice Dow Jones que é um dos principais indicadores de variação do mercado americano. O índice é criado com base nas cotações das 30 empresas mais influente do mercado americano e a série utilizada é formada por 1521 pontos que representam observações diárias entre 01/01/2008 e 17/01/2014.

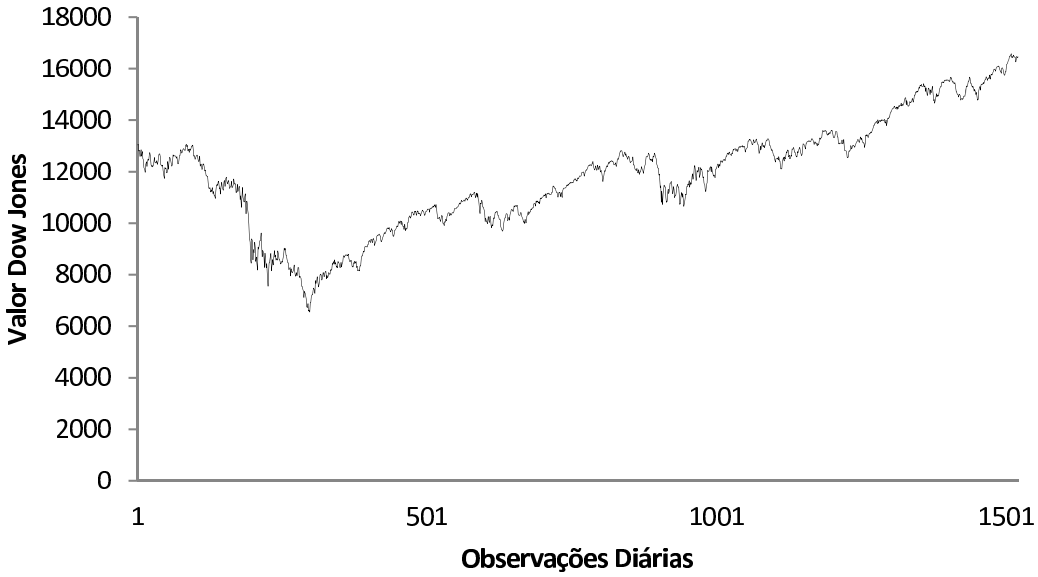

Figura 4.5: Gráfico da série DJIA.

Algumas estatísticas para esta série são exibidas na Tabela 4.3.

Tabela 4.3: Estatísticas da série DJIA

Estatísticas	Valores
Quantidade de Pontos	1521
Valor Máximo	16576,66
Valor Mínimo	6547,05
Valor Médio	11825,75
Variância	4518504,13
Desvio Padrão	2125,68

4.2.2 S&P500

É uma série que representa o registro da variação das 500 ações mais importantes para o mercado americano. O índice é criado baseado em uma proporção entre o preço da ação e o número de ações disponíveis para movimentação e a série utilizada é formada por 1000 pontos que representam observações mensais entre 01/08/1930 e 1/11/2013.

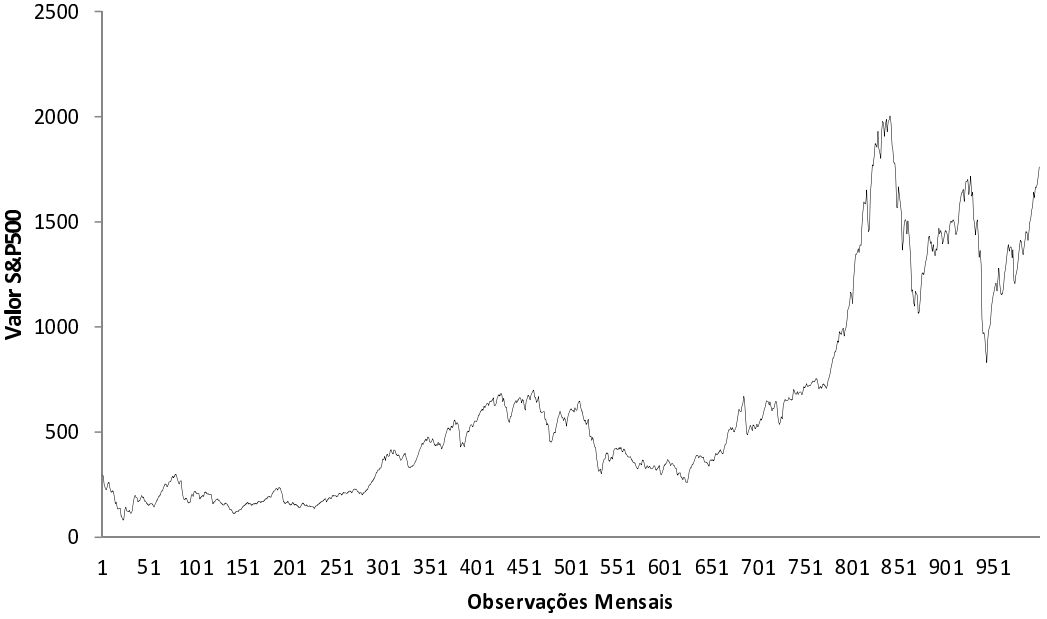

Figura 4.6: Gráfico da série S&P500.

Algumas estatísticas para esta série são exibidas na Tabela 4.4.

Tabela 4.4: Estatísticas da série S&P500

Estatísticas	Valores
Quantidade de Pontos	1000
Valor Máximo	2003,56
Valor Mínimo	81,75
Valor Médio	608,97
Variância	224099
Desvio Padrão	473,39

4.2.3 Brilho de Estrela

É uma série que representa a magnitude da intensidade luminosa de uma estrela com brilho variável registrado durante 600 noites consecutivas sempre no mesmo local e horário.

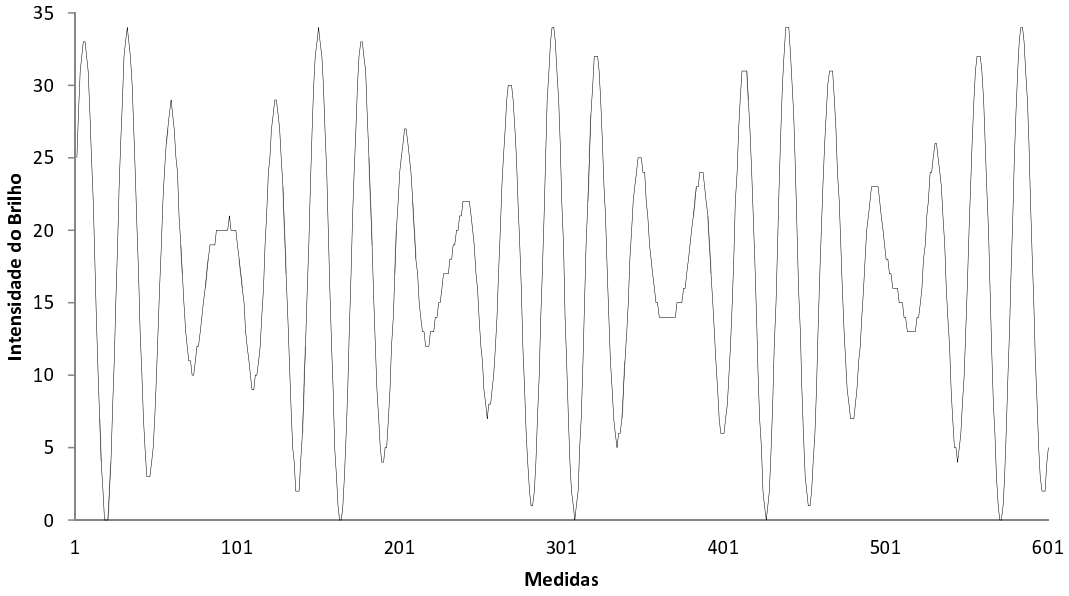

Figura 4.7: Gráfico da série Brilho de uma Estrela.

Algumas estatísticas para esta série são exibidas na Tabela 4.5.

Tabela 4.5: Estatísticas da série Brilho de uma Estrela

Estatísticas	Valores
Quantidade de Pontos	600
Valor Máximo	34,00
Valor Mínimo	0,00
Valor Médio	17,11
Variância	80,67
Desvio Padrão	8,98

4.2.4 Sunspots

É uma série que representa o registro de manchas solares composta por 313 pontos que representam observações anuais no período de 1700 a 2012. Uma mancha solar é um fenômeno que acontece na fotosfera do sol e são identificados como manchas escuras em imagens capturadas do sol. Algumas estatísticas para esta série são exibidas na Tabela 4.6.

Tabela 4.6: Estatísticas da série Sunspots

Estatísticas	Valores
Quantidade de Pontos	313
Valor Máximo	190,20
Valor Mínimo	0,00
Valor Médio	49,48
Variância	1626,74
Desvio Padrão	40,33

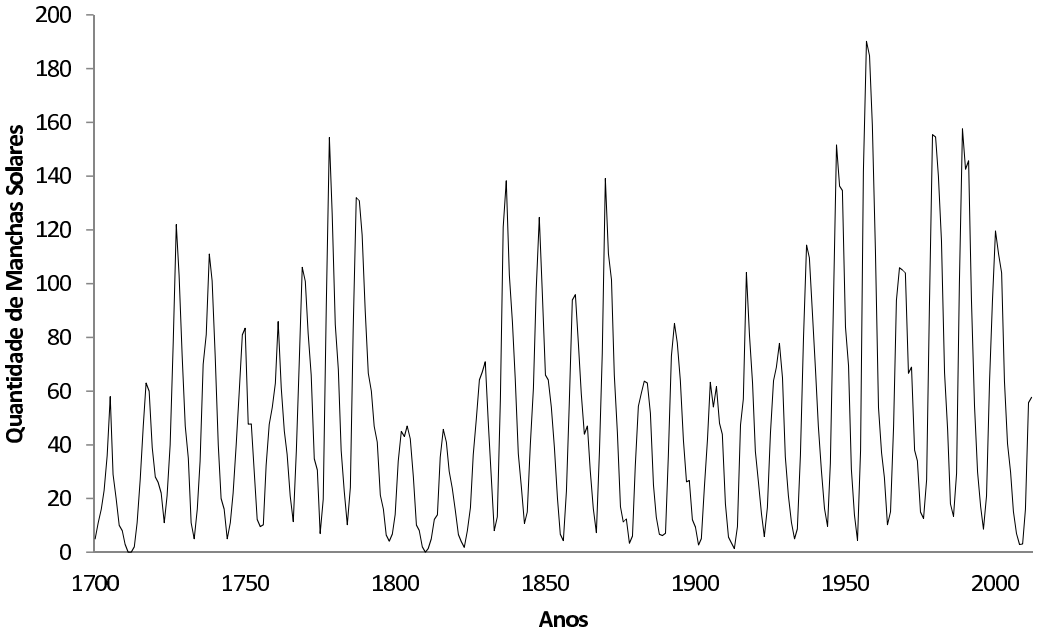

Figura 4.8: Gráfico da série Sunspots.

Capítulo 5

Resultados

Neste capítulo são apresentados os resultados alcançados em cada uma das séries apresentadas. Cada série foi dividida em dois conjuntos de dados, o conjunto de treinamento com 70% dos dados e conjunto de testes com 30% dos dados. Essa proporção de 70% e 30% é normalmente utilizada na literatura. O PSO proposto utiliza o conjunto de treinamento para pesquisar os padrões de comportamento (ou regras) e o conjunto de teste para verificar o desempenho das regras extraídas das partículas (desempenho de previsão).

5.1 Séries Temporais Artificiais

Para a simulação com as séries temporais artificiais o PSO foi parametrizado, através de parâmetros passados por linha de comando, com os seguintes valores:

- Tamanho da janela de reconhecimento da partícula: 7;
- Número máximo de gerações: 10000;
- Probabilidade de correspondência: 80%;
- Coeficiente de inércia w: 0.9;
- Constantes c1 e c2: 1.0.

Para ambas as séries temporais deterministas (séries A e B), sem ruído, o sistema foi capaz de determinar todas as regras necessárias para descrever os padrões de comportamentos apresentados em seus dados. O sistema encontrou um suporte e uma acurácia de 100% no conjunto de teste para as séries determinísticas A e B. Com o aumento do percentual de ruído para ambas as séries A e B, o suporte e a acurácia diminuíram. Para um baixo nível de ruído, 1% do ruído, o suporte ainda atingiu o valor de 100% para a série A, mas para a série B o suporte diminuiu para $98,98\%$. Para os outros valores de ruído analisados, em geral, tanto o suporte quanto a acurácia diminuíram, onde a diminuição da acurácia foi mais intensa em comparação com a diminuição do suporte, como mostrado na Figura 5.1 e na Figura 5.2.

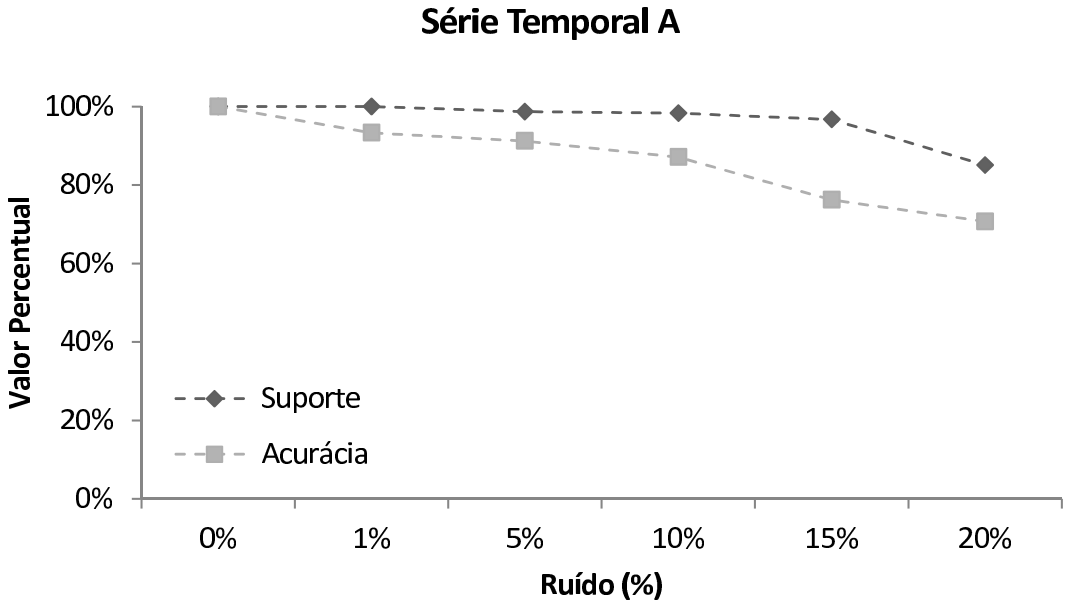

Figura 5.1: Gráfico dos resultados das simulações com a Série A.

Um resumo de todos os resultados experimentais para as séries temporais A com e sem ruídos são apresentados na Tabela 5.1 e para as séries temporais B com e sem ruído estão apresentados na Tabela 5.2.

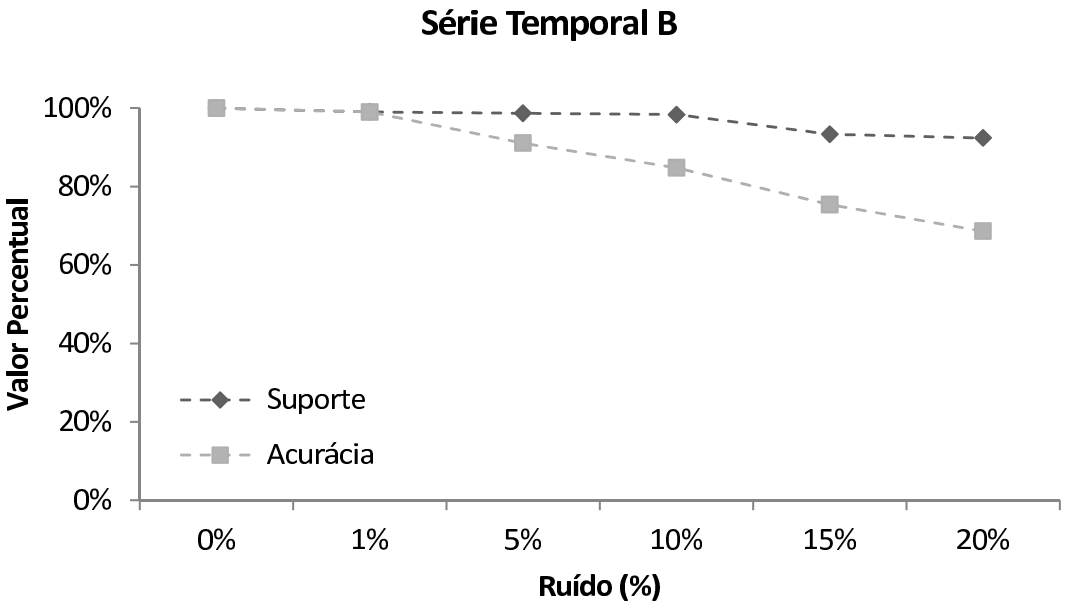

Figura 5.2: Gráfico dos resultados das simulações com a Série B.

5.2 Séries naturais ou financeiras

As séries naturais apresentam um desafio diferente das séries artificiais por não ter a informação do nível do ruído presente na série. Para encontrar um melhor resultado de previsão foram feiras várias simulações variando os parâmetros do PSO. Para cada série analisada o resultado dos valores de suporte e acurácia são melhores que um passeio aleatório[21] e está apresentado na Tabela 5.3, Tabela 5.4, Tabela 5.5 e Tabela 5.6.

5.3 Performance

Os resultados apresentados foram gerados por uma simulação desenvolvida em CUDA. A finalidade é garantir que a simulação seja executada em um tempo menor devido a capacidade de paralelização que as GPGPUs da NVIDIA permitem. Na tabela 5.7 são apresentados os tempos em segundos que cada simulação executada com a série artificial B durou. Foram

Tabela 5.1: Resultados experimentais para a série A

Porcentagem de ruído	Suporte	Acurácia
0%	100,00%	100,00%
1%	100,00%	99,30%
5%	98,67%	91,22%
10%	98,33%	87,12%
15%	96,67%	76,21%
20%	85,05%	70,70%

Tabela 5.2: Resultados experimentais para a série B

Porcentagem de ruído	Suporte	Acurácia
0%	100,00%	100,00%
1%	99,00%	98,99%
5%	96,67%	91,03%
10%	98,33%	84,74%
15%	93,33%	75,35%
20%	92,33%	68,59%

realizadas três baterias de simulação, onde a primeira trabalhou com apenas 1 PSO inicializado com 100 partículas, a segunda bateria com 1 PSO inicializado com 1000 partículas e a terceira bateria com 10 PSOs, rodando em paralelo, sendo cada um inicializado com 100 partículas. Através dos resultados apresentados na Tabela 5.7 a paralelização mostra que mesmo aumentando em 10x a quantidade de partículas processadas o esforço computacional não cresceu da mesma forma que executando de forma linear e se manteve sempre abaixo de 50% do esforço de tempo gastos pela segunda bateria.

Esses resultados mostram que o esforço do aprendizado da tecnologia é recompensado pelo ganho no tempo de execução das simulações e que ainda pode ser menor com o aprimoramento do código no uso das memórias das GPGPUs.

Tabela 5.3: Resultados das simulações da série S&P500

Tamanho da Janela	Gerações	C1	C2	Suporte	Acurácia
1	1000	1,0	1,0	100,00	56,26
2	1000	1,0	1,0	100,00	57,10
3	1000	1,0	1,0	91,36	59,15
3	1000	1,0	1,0	73,82	59,62
4	1000	1,0	1,0	89,97	61,61
5	1000	1,0	1,0	100,00	56,82
7	1000	1,0	1,0	95,82	63,66
7	1000	0,7	0,3	95,82	63,66
7	1000	0,5	0,5	95,82	63,66
7	1000	0,3	0,7	95,82	63,66
7	1000	1,0	1,0	95,82	63,66
7	2000	1,0	1,0	95,82	63,66
7	5000	1,0	1,0	95,82	63,66
8	1000	1,0	1,0	94,99	60,70
9	1000	1,0	1,0	77,44	56,83
10	1000	1,0	1,0	26,46	52,63

5.4 Comparações com o método TAEF

Os resultados apresentados nas simulações com séries temporais artificiais e baseadas em dados reais demonstram que a metodologia apresentada consegue atingir um bom percentual de acerto, mas para verificar esse desempenho é necessário comparar esses resultados com os resultados de outros trabalhos publicados. O método TAEF [11] foi comparado com diferentes técnicas de previsões de séries temporais e seus resultados tiveram um desempenho superior, além disso o resultados do método TAEF também foram comparados com os resultados de outros trabalhos publicados que apresentam diferentes técnicas de previsão de série temporais. Comparar os resultados apresentados por este trabalho com os resulta-

Tabela 5.4: Resultados das simulações da série Sunspot

Tamanho da Janela	Gerações	C1	C2	Suporte	Acurácia
1	1000	1,0	1,0	100,00	75,00
2	1000	1,0	1,0	100,00	75,00
3	1000	1,0	1,0	100,00	77,38
4	1000	1,0	1,0	100,00	80,95
5	1000	1,0	1,0	100,00	78,57
6	1000	1,0	1,0	97,62	84,15
7	1000	1,0	1,0	92,86	87,18
7	1000	0,7	0,3	92,86	87,18
7	1000	0,5	0,5	92,86	87,18
7	1000	0,3	0,7	92,86	87,18
7	1000	1,0	1,0	92,86	87,18
7	2000	1,0	1,0	92,86	87,18
7	5000	1,0	1,0	92,86	87,18
8	1000	1,0	1,0	88,10	86,50
9	1000	1,0	1,0	89,29	79,99
10	1000	1,0	1,0	84,52	85,92

dos apresentados pelo método TAEF poderá indicar se a metodologia proposta é realmente eficiente. A comparação entre esses resultados são apresentados na Tabela 5.8.

5.5 Análises dos resultados

Nas simulações realizadas com as séries artificiais a metodologia proposta conseguiu alcançar um bom desempenho nas previsões, apesar da queda de rendimento com o aumento do ruído, mas isso é um comportamento esperado nesse tipo de situação, onde o ruído interfere na análise das informações.

Tabela 5.5: Resultados das simulações da série Dow Jones

Tamanho da Janela	Gerações	C1	C2	Suporte	Acurácia
1	1000	1,0	1,0	100,00	52,69
2	1000	1,0	1,0	100,00	52,69
3	1000	1,0	1,0	63,94	55,31
4	1000	1,0	1,0	93,44	58,15
5	1000	1,0	1,0	93,44	55,14
6	1000	1,0	1,0	96,25	56,69
7	1000	1,0	1,0	89,23	57,48
7	1000	0,7	0,3	89,23	57,48
7	1000	0,5	0,5	89,23	57,48
7	1000	0,3	0,7	89,23	57,48
7	1000	1,0	1,0	89,23	57,48
7	2000	1,0	1,0	89,23	57,48
7	5000	1,0	1,0	89,23	57,48
8	1000	1,0	1,0	91,80	58,93
9	1000	1,0	1,0	66,28	49,12
10	1000	1,0	1,0	24,82	53,77

Com as séries baseadas em dados reais os resultados demonstraram que o método proposto também conseguiu alcançar um bom desempenho nas previsões, apesar de atingir percentuais mais altos nas séries naturais em comparação com os resultados alcançados com as séries financeiras.

Na comparação com os resultados apresentados pelo método TAEF, para as mesmas séries utilizadas, a metodologia apresentada consegue resultados melhores nas séries naturais. Enquanto o método TAEF apresentou um percentual de 77.34% o trabalho proposto atingiu 100.00% de previsão na série de Brilho de uma Estrela. Entretanto, nas séries financeiras o método TAEF atinge resultados superiores conseguindo uma previsão de 100.00% na série S&P500 enquanto que a metodologia apresentada atingiu 95.82%.

Tabela 5.6: Resultados das simulações da série Brilho de uma Estrela

Tamanho da Janela	Gerações	C1	C2	Suporte	Acurácia
1	1000	1,0	1,0	100,00	90,56
2	1000	1,0	1,0	100,00	90,56
3	1000	1,0	1,0	100,00	91,11
4	1000	1,0	1,0	98,89	91,57
5	1000	1,0	1,0	100,00	85,56
6	1000	1,0	1,0	98,88	85,39
7	1000	0,7	0,3	97,77	85,23
7	1000	0,5	0,5	97,77	85,23
7	1000	0,3	0,7	97,77	85,23
7	1000	1,0	1,0	97,77	85,23
7	2000	1,0	1,0	97,77	85,23
7	5000	1,0	1,0	97,77	85,23
7	1000	1,0	1,0	97,77	85,23
8	1000	1,0	1,0	96,66	85,63
9	1000	1,0	1,0	96,11	88,44
10	1000	1,0	1,0	95,55	84,88

Apesar de apresentar resultados inferiores nas séries financeiras, mas com uma diferença percentual não tão acentuada, os resultados apresentados para as séries naturais apresentaram um desempenho superior e isso indica que trabalho proposto é uma boa opção para a previsão de séries temporais.

5.5.1 Verificação dos resultados com árvores de decisão

Apesar dos resultados das simulações apresentarem um bom desempenho em relação ao método TAEF as regras encontradas, e armazenadas na lista tabu, durante a evolução dos PSOs podem ter seu desempenho verificado através de árvores de decisão[6]. As árvores de

Tabela 5.7: Tempo de execução das simulações

Serie	Bateria 1 1 PSO com 100 Partículas	Bateria 2 1 PSO com 1000 Partículas	Bateria 3 10 PSOs com 100 Partículas
Serie B - Sem ruído	7s	38s	17s
Série B - Ruído 1%	7s	60s	28s
Série B - Ruído 5%	7s	68s	30s
Série B - Ruído 10%	7s	71s	32s
Série B - Ruído 15%	8s	74s	31s
Série B - Ruído 20%	8s	68s	33s

Tabela 5.8: Comparação dos resultados com o método TAEF

Série	Metodologia Proposta	Método TAEF
Brilho de uma estrela	100.00%	77.34%
Sunspot	97.62%	84.06%
S&P500	95.82%	100.00%
Dow Jones	91.80%	97.14%

decisão foram criadas, para cada série analisada, através da biblioteca *rpart* do sistema *R*. A biblioteca *rpart* utiliza o método CART (*Classification and Regression Trees* - Árvores de Classificação e Regressão).

As regras encontradas nas simulações das séries temporais foram carregadas no sistema R e para criar as árevores de decisão cada componente da janela de observação das regras foi chamado de $j1$, $j2$, ..., jn (onde n é o tamanho da janela de reconhecimento e o valor de previsão foi chamado simplesmente de p. A Figura **??** apresenta a nomenclatura utilizada.

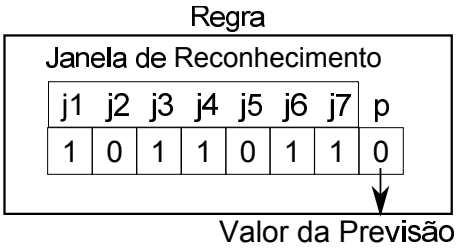

Figura 5.3: Nomenclatura da regra na lista tubu.

Com base nessas informações o sistema R construiu uma árvore de decisão, para cada conjunto de regras, que foi testada com o conjunto de dados de testes, de sua respectiva série temporal. O teste é ralizado comparando cada valor do conjunto de testes com as regras da árvore de decisão e para veriricar a qualidade do teste um percentual de erro e acerto foi calculado.

O contador de erro é incrementado, $contErro = contErro + 1$, em duas situações:

- Se qualquer um dos n elementos não for verificado pela regra da árvore de decisão;
- Se todos os n elementos foram verificados, mas o valor de previsão não correspondeu ao valor esperado.

Não acontecendo nenhumas das duas situações acima o contador de acertos é incrementado, $contAcerto = contAcerto+1$. Ao final da verificação do conjunto de testes os resultados são apresentados. A Figura 5.4 apresenta a árvore de decisão para as regras da Série A sem ruído e a Tabela 5.9 apresenta os resultados de todas as verificações para as árvores de decisão da série A. Os valores em cada ramo da árvore representam a quantidade de itens que foram validados com sucesso pela regra presente no nó anterior ao ramo.

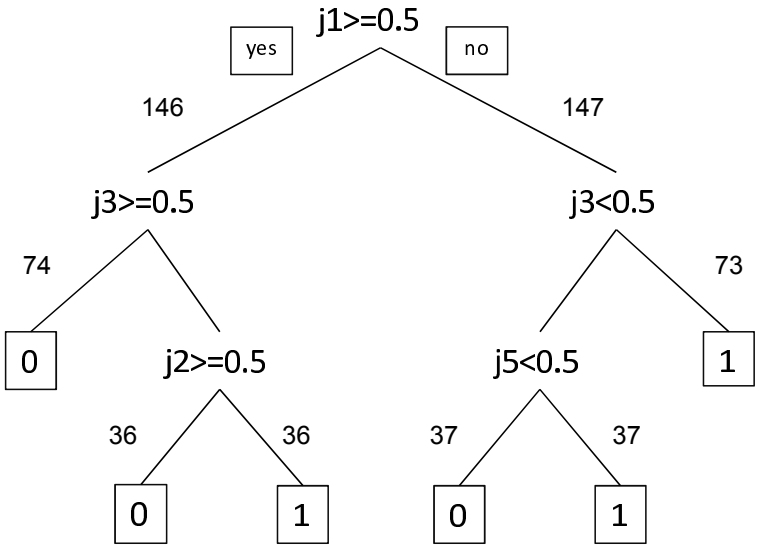

Figura 5.4: Árvore de decisão para as regras da Série A - Sem ruído.

As árvores foram testadas no conjunto de dados de testes e um percentual de erro e acerto foi calculado verificando em cada passo se os elementos da janela respeitavam a estrutura de decisão de cada árvore. Para cada valor de previsão encontrado que coincidia com o comportamento apresentado pela árvore um contador de acertos é incrementado, $contAcerto = contAcerto + 1$ e para cada valor de previsão diferente um contador de erros é incrementado, $contErro = contErro + 1$. Por questões de simplicidade e organização do trabalho as figuras das outras árvores para a série A foram omitidas.

Tabela 5.9: Análise das árvores de decisão da série A

Serie A	Acertos(%)	Erros(%)
Ruído de 0%	100,00%	0,00%
Ruído de 1%	99,32%	0,68%
Ruído de 5%	87,03%	12,97%
Ruído de 10%	76,11%	23,89%
Ruído de 15%	63,36%	36,64%
Ruído de 20%	54,76%	45,24%

Para a série B e as séries naturais e financeiras as árvores também foram omitidas e as tabelas 5.10 e 5.11 apresentam os resultados das análises da série B e das outras séries respectivamente.

Tabela 5.10: Análise das árvores de decisão da série B

Serie A	Acertos(%)	Erros(%)
Ruído de 0%	100,00%	0,00%
Ruído de 1%	87,03%	12,97%
Ruído de 5%	78,84%	21,16%
Ruído de 10%	74,06%	25,94%
Ruído de 15%	63,82%	36,18%
Ruído de 20%	59,39%	40,61%

Tabela 5.11: Análise das árvores de decisão das séries naturais e financeiras

Serie A	Acertos(%)	Erros(%)
Brilho de uma estrela	94,92%	5,08%
Sunspot	79,47%	20,53%
S&P500	56,85%	43,15%

Os resultados demonstram que o total de acertos produzido pelas árvores de decisão é um valor aproximado dos valores calculados para a acurácia nos resultados da simulação. Isso indica que as regras extraídas das séries temporais podem representar comportamentos que as séries temporais venham a repetir em eventos futuros. A série Dow Jones não produziu nenhuma árvore de decisão por suas regras representarem um modelo de passeio aleatório, é possível perceber na Tabela 5.5 que os valores de acúracia calculados por todas as simulações para essa série ficaram com valores abaixo de 60

Capítulo 6

Conclusão

Neste capítulo são apresentadas as conclusões obtidas com os resultados deste trabalho. A metodologia apresentada demonstrou resultados satisfatórios em comparação com o método TAEF, que apresentou desempenho superior a outras técnicas de previsão de séries temporais. É possível perceber que quanto maior o nível de ruído apresentado por uma série temporal mais difícil se tornou extrair conhecimento a partir da mesma, esse é um comportamento esperado e as regras artificiais permitiram mapear esse comportamento por ter o nível de ruído conhecido.

A série Sunspot, por não apresentar um comportamento linear é conhecida por ser uma série de difícil previsão [1] e mesmo nessas condições a metodologia apresentada neste trabalho conseguiu um bom desempenho nas previsões de tendência, diferente do que aconteceu com a série Dow Jones e a S&P500 onde elas apresentam um aumento na perda de correlação entre os pontos e esse comportamento afeta de forma negativa o resultado da simulação.

6.1 Pontos Fortes

O trabalho permitiu a utilização de um PSO, um algoritimo de fácil implementação, para realizar previsões de tendências em séries temporais com resultados superiores nas séries

naturais e aproximados nas séries financeiras que outros modelos da literatura. Outro ponto forte deste trabalho foi a combinação do PSO com uma lista tabu para criar um modelo híbrido e tornar possível o processo de execução paralela dos PSOs, explorando diferentes espaços de busca. É importante ressaltar também que o desenvolvimento do modelo proposto com a tecnologia CUDA da NVIDIA permitiu melhorar o desempenho das simulações permitindo que os resultados sejam gerados de forma mais rápida e dessa forma usar de forma mais eficiente os recursos computacionais disponíveis.

6.2 Pontos Fracos

Os resultados experimentais obtidos pela metodologia apresentada neste trabalho demonstraram que as simulações realizadas com as séries naturais apresentaram um desempenho melhor que as séries financeiras. Essa diferença de comportamento entre os tipos de séries podem indicar possíveis ajustes na metodologia e na forma como as simulações foram parametrizadas. Outro ponto importante é a análise da estagnação do PSO que apesar de descrito na metodologia não foi parametrizado na execução das simulações sendo a quantidade de gerações a única condição de parada utilizada. Permitir que os PSOs evoluam por mais gerações avaliando o processo de estagnação não foi testado efetivamente e seria necessário uma análise mais aprofundada desse comportamento.

6.3 Trabalhos Futuros

Devido ao desempenho mais baixo nas séries financeiras uma análise mais aprofundada no comportamento do PSO em relação a essas séries pode ser realizada e podem ser desenvolvidos filtros em uma operação de pré-processamento para facilitar o processo de previsão dessas séries.

Outra abordagem para a metodologia teria o foco no tamanho da quantidade de bits utilizado para o valor de previsão das partículas e dessa forma não restringir o sistema a fazer apenas previsões de um ponto à frente.

Também pode ser verificado o comportamento do uso de memória da GPU para permitir um modelo em grade e desta forma expandir a exploração do espaço de busca sem o aumento do custo computacional por equipamento utilizado nas simulações.

6.4 Trabalhos Publicados

Com esta dissertação foi gerado o seguinte trabalho:

- A System Based on Swarm Particle Optimization to Extract Knowledge from Times Series Data [19].

Referências Bibliográficas

[1] LA Aguirre, C Letellier, and J Maquet. Forecasting the time series of sunspot numbers. *Solar Physics*, 249(1):103–120, 2008.

[2] J Behnamian and SMT Fatemi Ghomi. Development of a pso–sa hybrid metaheuristic for a new comprehensive regression model to time-series forecasting. *Expert Systems with applications*, 37(2):974–984, 2010.

[3] George EP Box, Gwilym M Jenkins, and Gregory C Reinsel. *Time series analysis: forecasting and control*. John Wiley & Sons, 1998.

[4] Pei-Chann Chang, Di-di Wang, and Chang-le Zhou. A novel model by evolving partially connected neural network for stock price trend forecasting. *Expert Systems with Applications*, 39(1):611–620, 2012.

[5] Maurice Clerc and James Kennedy. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *Evolutionary Computation, IEEE Transactions on*, 6(1):58–73, 2002.

[6] Yann Coadou. Decision trees. In *EPJ Web of Conferences*, volume 4, page 02003. EDP Sciences, 2010.

[7] NVIDIA Corporation. Nvidia nsight. `https://developer.nvidia.com/nsight-eclipse-edition`. Accessed: 2014-02-09.

[8] C Cuda. Programming guide. *NVIDIA Corporation (October 2012)*, 2012.

[9] Paulo SG de M Neto, Gustavo G Petry, RLJ Aranildo, and Tiago AE Ferreira. Combining artificial neural network and particle swarm system for time series forecasting. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, pages 2230–2237. IEEE, 2009.

[10] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. springer, 2003.

[11] Tiago AE Ferreira. *Uma nova metodologia híbrida inteligente para a previsão de séries temporais*. PhD thesis, Thesis (Ph. D.), Centro de Informática UFPE, 2006.

[12] Tiago AE Ferreira, Germano C Vasconcelos, and Paulo JL Adeodato. A new intelligent system methodology for time series forecasting with artificial neural networks. *Neural Processing Letters*, 28(2):113–129, 2008.

[13] Yao-Lin Huang, Shi-Jinn Horng, Mingxing He, Pingzhi Fan, Tzong-Wann Kao, Muhammad Khurram Khan, Jui-Lin Lai, I Kuo, et al. A hybrid forecasting model for enrollments based on aggregated fuzzy time series and particle swarm optimization. *Expert Systems with Applications*, 38(7):8014–8023, 2011.

[14] James Kennedy, Russell Eberhart, et al. Particle swarm optimization. *Proceedings of IEEE international conference on neural networks*, 4(2):1942–1948, 1995.

[15] Terence C Mills. The econometric modelling of financial time series. *Cambridge Books*, 2003.

[16] PSG de M NETO, Ricardo de A Araújo, Gustavo G Petry, Tiago AE Ferreira, and Germano C Vasconcelos. Hybrid swarm system for time series forecasting. *VI Encontro Nacional de Inteligência Artificial (ENIA)*, 2007.

[17] Said Salhi. Defining tabu list size and aspiration criterion within tabu search methods. *Computers & Operations Research*, 29(1):67–86, 2002.

[18] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[19] Henrique CT Santos, Gabriela IL Alves, Neilson F de Lima, Paulo SG de Mattos Neto, and Tiago AE Ferreira. A system based on swarm particle optimization to extract knowledge from times series data. In *Neural Networks (SBRN), 2012 Brazilian Symposium on*, pages 244–249. IEEE, 2012.

[20] Pritpal Singh and Bhogeswar Borah. Forecasting stock index price based on m-factors fuzzy time series and particle swarm optimization. *International Journal of Approximate Reasoning*, 2013.

[21] Renate Sitte and Joaquin Sitte. Neural networks approach to the random walk dilemma of financial time series. *Applied Intelligence*, 16(3):163–171, 2002.

[22] Floris Takens. Detecting strange attractors in turbulence. In *Dynamical systems and turbulence, Warwick 1980*, pages 366–381. Springer, 1981.

[23] Frans Van den Bergh and Andries Petrus Engelbrecht. A cooperative approach to particle swarm optimization. *Evolutionary Computation, IEEE Transactions on*, 8(3):225–239, 2004.

[24] Vladimir Vapnik. Statistical learning theory. 1998, 1998.

[25] Martin Weigel. The gpu revolution at work. *Computing in Science & Engineering*, 13(5):5–6, 2011.

[26] Wai Keung Wong, SYS Leung, and ZX Guo. Feedback controlled particle swarm optimization and its application in time-series prediction. *Expert Systems with Applications*, 39(10):8557–8572, 2012.

[27] Lean Yu, Shouyang Wang, and Kin Keung Lai. Mining stock market tendency using ga-based support vector machines. In *Internet and Network Economics*, pages 336–345. Springer, 2005.

[28] Lean Yu, Shouyang Wang, and KK Lai. A rough-set-refined text mining approach for crude oil market tendency forecasting. *International Journal of Knowledge and Systems Sciences*, 2(1):33–46, 2005.

Apêndice A

Plataforma NVIDIA CUDA

No Capítulo 2 foi apresentado um código desenvolvido na linguagem C e utilizando elementos da API da plataforma CUDA da NVIDIA. Neste anexo o código será novamente apresentado e explicado de forma mais detalhada.

Além de facilitar o acesso ao poder de processamento da GPU, a NVIDIA desenvolveu também um compilador que deve ser utilizado em arquivos de código fontes da linguagem C que usem a extensão CUDA. Também está disponível para os usuários de Linux um ambiente de desenvolvimento chamado Nsight que é um projeto desenvolvido utilizando o projeto Eclipse como base [7]. A Listagem 2.1 apresenta um programa feito na linguagem C, que faz a soma de dois vetores, fazendo uso de comandos da extensão CUDA da NVIDIA que estão destacados em negrito.

Listagem A.1: Exemplo de um programa C/CUDA para somar 2 vetores

```
1 #include "cuda_runtime.h"
2 #include "device_launch_parameters.h"
3 #include <stdio.h>
4 __global__ void addKernel(int *c, const int *a, const int *b) {
5     int i = blockDim.x*blockIdx.x + threadIdx.x;
6     c[i] = a[i] + b[i];
7 }
```

```
8  int main() {
9      const int arraySize = 15;
10     const int a[arraySize] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
11     const int b[arraySize] = { 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70 };
12     int c[arraySize] = { 0 };
13     int *dev_a = 0;
14     int *dev_b = 0;
15     int *dev_c = 0;
16     cudaMalloc((void**)&dev_c, arraySize * sizeof(int));
17     cudaMalloc((void**)&dev_a, arraySize * sizeof(int));
18     cudaMalloc((void**)&dev_b, arraySize * sizeof(int));
19     cudaMemcpy(dev_a, a, arraySize * sizeof(int), cudaMemcpyHostToDevice);
20     cudaMemcpy(dev_b, b, arraySize * sizeof(int), cudaMemcpyHostToDevice);
21     addKernel<<<1, arraySize>>>(dev_c, dev_a, dev_b);
22     cudaThreadSynchronize();
23     cudaMemcpy(c, dev_c, arraySize * sizeof(int), cudaMemcpyDeviceToHost);
24     for(int i=0; i<arraySize; i++) {
25         printf("a[%d]=%d + b[%d]=%d = c[%d]=%d\n", i, a[i], i, b[i], i, c[i]);
26     }
27     cudaFree(dev_c);
28     cudaFree(dev_a);
29     cudaFree(dev_b);
30     return 0;
31 }
```

Para desenvolver programação paralela em CUDA é necessário estar familiarizado com os termos e conceitos que envolvem essa plataforma. Identificar no código o que será de responsabilidade da CPU e o que poderá ser enviado para a GPU é uma atividade que o programador deve realizar para que a performance da execução seja satisfatória. A programação paralela envolve a criação de *threads* que podem ser definidas como linhas de execução paralelas como mostrado na Figura 3 e na Figura 4.

Em CUDA um conjunto de *threads* é definido através de um *kernel* que é implementado de forma similar a uma função, encontrada nas linguagens convencionais, como é mostrado na linha 4 do código apresentado na Listagem A.1.

```
__global__ void addKernel(int *c, const int *a, const int *b)
```

O uso do modificador `__global__` é o que determina que este método seja um *kernel* e que a sua execução será realizada pela GPU. As threads definidas por esse *kernel* executarão o seu código de forma paralela e são definidas no momento de sua chamada, como mostrado na linha 21 do código apresentado na Listagem A.1.

```
addKernel<<<1, arraySize>>>(dev_c, dev_a, dev_b);
```

A estrutura interna de um *kernel* pode ser mapeada para representar um vetor, uma matriz, ou mesmo um cubo e isso dependerá exclusivamente dos valores informados entre os símbolos `<<< e >>>`. A estrutura hierárquica de execução das threads pode ser dividida em:

- Thread: É unidade de execução paralela que executa o código definido no kernel e é identificada por um índice. Esse índice pode ser usado com a identificação de uma posição em um *array*.
- Bloco: É um conjunto de *threads* e assim como elas também é identificado por um índice. As threads de um mesmo bloco podem ser sincronizadas de forma que uma thread pare em determinado ponto de execução e aguarde que todas as outra *threads*, do mesmo bloco, alcancem esse mesmo ponto.
- Grid: É um conjunto de blocos que não podem sincronizar suas *threads* entre si. A estrutura hierárquica de execução das *Threads* pode ser visto na Figura 23.

Os valores apresentados entre `<<< e >>>` determinam a estrutura do *kernel* e no caso do código apresentado na Listagem A.1 representam respectivamente a quantidade de blocos e a quantidade de *threads* em cada bloco criando um modelo unidimensional de execução paralela, como o exemplo mostrado na Figura 24. Outros exemplos de definição da estrutura interna de um *kernel* podem ser vistos na Figura 25 e na Figura 26.

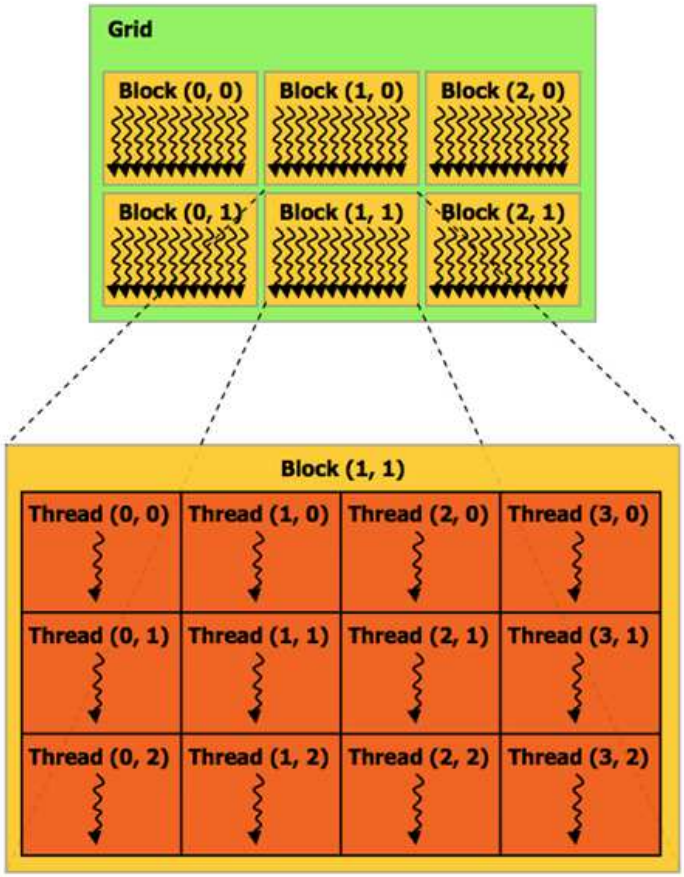

Figura A.1: Estrutura hierárquica de execução das threads.

O acesso a cada *thread* em execução é feito através dos índices de identificação das *threads* e dos blocos e como foi utilizado um modelo unidimensional a construção do identificador de cada *thread* é feita apenas com o identificador da *thread* como mostrado na linha 5 do código apresentado na Listagem A.1.

```
int i = blockDim.x * blockIdx.x + threadIdx.x;
```

onde $threadIdx.x$ é o índice da *thread* na dimensão x do bloco, $blockIdx.x$ é o índice do bloco na dimensão x do *grid* e $blockDim.x$ representa o número de *threads* na dimensão x

Figura A.2: Estrutura paralela com 1 bloco e 5 threads.

Figura A.3: Estrutura paralela com 5 blocos e 1 thread.

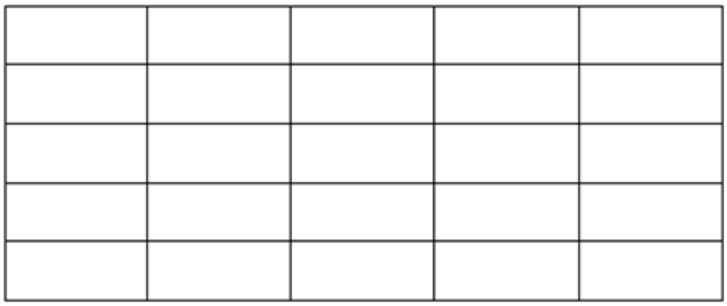

Figura A.4: Estrutura paralela com 5 blocos e 5 thread.

de um bloco. Como o código é executado com um único bloco a construção do índice da *thread* poderia ser definido simplesmente como:

```
int i = threadIdx.x;
```

A modificação na estrutura deste *kernel* pode ser feita através dos parâmetros `<<<3, arraySize/3>>>` que criará uma estrutura bidimensional com 3 blocos e 5 *threads* em cada bloco, que teria a representação interna e identificação de cada *thread* como mostrado na

threadIdx.x					threadIdx.x					threadIdx.x				
0	1	2	3	4	0	1	2	3	4	0	1	2	3	4
blockIdx=0					blockIdx=1					blockIdx=2				

Figura A.5: Estrutura bidimensional de um kernel.

E o índice da *thread* destacada no segundo bloco seria construído, usando a fórmula:

```
int i = blockDim.x * blockIdx.x + threadIdx.x; int i = 5 * 1 + 2 = 7
```

Um dos problemas na implementação de uma estrutura dinâmica, como um PSO, em CUDA é o modelo de memória utilizado por essa tecnologia. Em um sistema com CUDA o programa deve identificar que parte do código será executada pela GPU e que parte será executada pela CPU do computador. Uma vez identificada que parte do código que será paralelizada pela GPU é importante definir que tipo de memória será usado pelas informações do código em execução no *kernel*. O desempenho da execução do programa dependerá do tipo de memória utilizado. Os tipos de memórias que podem ser usadas em CUDA estão apresentados na Figura 28 e a memória principal do computador será chamada de *host* enquanto que a memória da GPU será chamada simplesmente de *device*.

- Texture Memory: É uma memória que não pode ser modificada pelo *kernel* e que mantém os valores armazenados após a execução do *kernel*. É uma memória otimizada para determinadas operações de leitura que nesses casos oferece uma performance superior a memória global que é o tipo de memória normalmente utilizado. É uma memória que utiliza cache para aumentar a sua performance.

- Constant Memory: É uma memória que não pode ser modificada pelo *kernel* e que mantém os valores armazenados após a execução do *kernel*. Dependendo da forma de acesso pode ser uma memória tão rápida quanto os registradores (*registers*). É uma memória que utiliza *cache* para aumentar a sua performance.

- Global Memory: É uma memória que não pode ser modificada pelo *kernel* e que pode ser compartilhada por diferentes blocos em execução. É a memória com maior capacidade disponível e mantém os valores armazenados após a execução do *kernel*. Apesar de ser uma memória versátil é uma memória que apresenta um alto grau de latência e por isso é uma memória lenta. É uma memória que utiliza um *cache*, dependendo do modelo da placa de vídeo, que será invalidado sempre que acontecer uma operação de escrita o que prejudica o seu desempenho.

- Shared Memory: É uma memória que não pode ser modificada pelo *kernel*, mas tem seu escopo reduzido a um bloco. Todas as *threads* do mesmo bloco compartilham as mesmas informações e é uma memória rápida.

- Local Memory: É uma memória que não pode ser modificada pelo *kernel*, mas tem seu escopo reduzido a uma *thread*. A finalidade dessa memória é armazenar as informações de variáveis que excedam a capacidade dos registradores (*registers*). É uma memória tão lenta quanto a *global memory* e é preciso evitar que as threads necessitem de uma quantidade grande de memória para não impactar na performance de execução do programa.

- Registers: É uma memória que não pode ser modificada pelo *kernel* e da mesma forma que a *local memory* tem seu escopo reduzido a uma *thread*, mas o acesso a essa memória é extremamente rápido.

De forma similar aos programas desenvolvidos apenas na linguagem C é necessário alocar memória, para o uso de estruturas como vetores e matrizes, tanto no *host* quando no *device*. As linhas 12, 13 e 14 inicializam os vetores a, b e c no host e através do comando *cudaMalloc* é alocado a memória no *device*, como mostrado nas linhas 16, 17 e 18 do código apresentado na Listagem A.1:

```
    cudaMalloc((void**)&dev_c, arraySize * sizeof(int));
cudaMalloc((void**)&dev_a, arraySize * sizeof(int));
cudaMalloc((void**)&dev_b, arraySize * sizeof(int));
```

O primeiro parâmetro é um ponteiro para o espaço de memória que será alocada no dispositivo e o segundo parâmetro é o tamanho deste espaço que será alocado. O comando *cudaMalloc* alocará um espaço de memória na *global memory*. Uma vez que a memória do dispositivo é alocada ela estará disponível para ser utilizada pelo código que será executado na GPU, mas antes é necessário fazer a transferência dos valores do *host* para o *device* da forma como mostrado nas linhas 19 e 21 do código apresentado na Listagem A.1:

```
cudaMemcpy(dev_a, a, arraySize * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, arraySize * sizeof(int), cudaMemcpyHostToDevice);
```

O comando *cudaMemcpy* permite que os dados armazenados no *host* e referenciadas por *a* e *b* sejam copiados para *device*, que será referenciada por `dev_a` e `dev_b`. A cópia para do host para o device é definida pelo terceiro parâmetro *cudaMemcpyHostToDevice*, já que o

mesmo comando é utilizado para copiar os valores do *device* para o *host*, como na linha 23 do código apresentado na Listagem A.1:

```
cudaMemcpy(c, dev_c, arraySize * sizeof(int), cudaMemcpyDeviceToHost);
```

Novamente o terceiro parâmetro é utilizado para informar a direção da transferência e nesse caso o valor $cudaMemcpyDeviceToHost$ determina que a direção da transferência acontecerá do *device*, onde os dados estão armazenados no espaço de memória referenciado por `dev_c`, para o *host* e serão armazenados na varável *c*. Como neste caso os vetores `dev_a` e `dev_b` não serão modificados no kernel eles poderiam estar armazenados na *constant memory*, mas para isso os vetores deveriam ser declarados com o atributo `__constant__` e serem preenchidos com o comando $cudaMemcpyToSymbo()$. Outra característica importante em CUDA é a possibilidade de executar vários *kernels* em paralelo. A paralelização de *kernels* é realizada através dos *streams* e é necessário especificar na chamada do *kernel* em que *stream* ele será executado. O trecho de código abaixo declara um *stream* e o informa com parâmetro na chamada do kernel.

```
cudaStream_t stream;

addKernel<<<1, arraySize, stream>>>(dev_c, dev_a, dev_b);
```

Não informar o *stream* faz com que os *kernels* sejam executados no *stream* padrão o que faz com que a execução dos kernels fique sincronizada e não aconteça de forma paralela. Com a introdução dos *streams* na metodologia proposta por esse trabalho é possível que vários PSOs diferentes estejam em execução paralela ampliando a exploração do espaço de busca e aumentando a possibilidade de encontrar novas soluções para o problema em análise.

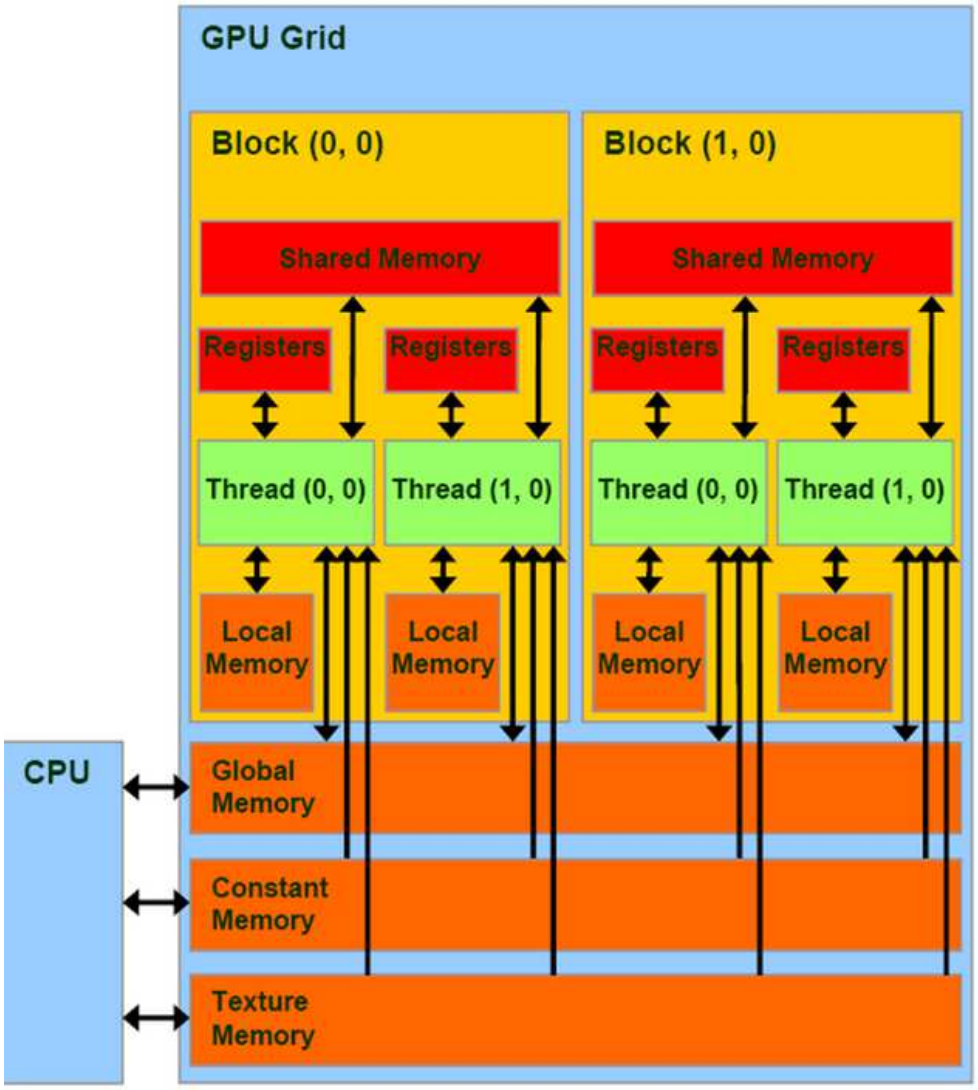

Figura A.6: Modelo de memória - CUDA.