MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA MECÂNICA

UMA PLATAFORMA PARA MANUSEIO, ANÁLISE E AVALIAÇÃO DE DADOS DE OBSERVAÇÃO E CONTROLE DE SIMULAÇÕES

por

Pedro Niederhageböck Sidou

Dissertação para obtenção do Título de
Mestre em Engenharia

Porto Alegre, Agosto de 2017

UMA PLATAFORMA PARA MANUSEIO, ANÁLISE E AVALIAÇÃO DE DADOS DE OBSERVAÇÃO E CONTROLE DE SIMULAÇÕES

por

Pedro Niederhageböck Sidou
Engenheiro Químico

Dissertação submetida ao Corpo Docente do Programa de Pós-Graduação em Engenharia Mecânica, PROMEC, da Escola de Engenharia da Universidade Federal do Rio Grande do Sul, como parte dos requisitos necessários para a obtenção do Título de

Mestre em Engenharia

Área de Concentração: Fenômenos de Transporte

Orientador: Prof. Dr. Bardo Ernst Josef Bodmann

Aprovada por:

Prof. Dr. Volnei Borges .... PROMEC / UFRGS

Prof. Dr. Luiz Alberto Oliveira Rocha .... PROMEC / UFRGS

Prof. Dr. Rubem Mário Figueiró Vargas .... PUCRS

Prof. Dr. Jakson Manfredini Vassoler
Coordenador do PROMEC

Porto Alegre, 23 de Agosto de 2017

AGRADECIMENTOS

Agradeço,

Aos meus familiares, especialmente aos meus pais, pelo suporte prestado ao longo de todos os anos de estudo, e ao meu primo Eduardo Sidou Canha, por seus conselhos.

Ao professor Bardo Bodmann, pela orientação, e à todos os colegas e profissionais que tive a oportunidade de trabalhar lado a lado no GENUC.

Ao PROMEC, pela oportunidade e pelo excelente corpo de funcionários e docentes disponibilizado.

Ao grupo ENEVA, à Linhares Geração S.A. e à Fundação Luiz Englert pela estrutura oferecida e apoio financeiro.

RESUMO

Atualmente, com a melhoria das técnicas de paralelização, cada vez mais vem sendo empregado o uso de *clusteres* de computadores para programas exigentes de alta performance. Porém, para um programa rodar adequadamente em paralelo, o mesmo deve ser escrito de maneira paralelizável. A proposta deste trabalho é o desenvolvimento de uma plataforma computacional que facilite a escrita de programas orientados à análise de dados climáticos e simulações de fenômenos atmosféricos escritos em paralelo. A plataforma foi pensada de maneira a ser o mais flexível possível, permitindo que novas funcionalidades sejam adicionadas através de *plugins*. Em um futuro próximo, pretende-se disponibilizá-la para uso público com licença *GNU General Public License* (GNU GPL) e código aberto. Para demonstrar seu potencial de uso, foi realizado um estudo exploratório com dados provenientes do projeto de reanálise *Twentieth Century Reanalysis version 2* (20CRv2). Tal estudo visa obter informações sobre que parâmetros e escalas de tempo são importantes para a descrição do fenômeno da zona de convergência do Atlântico sul (ZCAS).

Palavras-chave: Simulação; Computação paralela; Climatologia; Zcas; Dispersão de poluentes.

ABSTRACT

Since parallel computing technologies are improving very fast, computer clusteres are getting more and more employed for highly demanding computational tasks. Nevertheless, for a software to run in a cluster, it needs to be written in a parallel way, which is not necessarily a simple task. Therefore, the aim of this work is to develop a computer platform capable of manipulating, analysing and evaluating climate observational data and controlling simulation in a parallel manner. Features can be added to the platform through plugins, making it very flexible and extensible for a very large range of tasks. A simple application is purposed to demonstrate the platform's potential uses. The application consists of a exploratory study of data from the 20th century reanalysis project concerning the *South American Convergence Zone* (SACZ), a very important phenomena in the south hemisphere.

Keywords: Parallel computing; Simulation; Climate; SACZ.

ÍNDICE

1 Introdução . . . 1

1.1 Contexto . . . 1

1.2 Objetivos . . . 3

2 Revisão Bibliográfica . . . 4

2.1 Paralelização de softwares . . . 4

2.2 Formato de dados climáticos . . . 10

2.3 Softwares para tratamento de dados climáticos . . . 11

3 Descrição do Programa . . . 12

3.1 Escopo . . . 12

3.2 Linguagem de programação . . . 13

3.3 Estrutura do programa . . . 14

3.4 Núcleo . . . 16

3.5 Interface de usuário . . . 18

3.6 Gerenciador de dados . . . 20

3.7 *Parser* . . . 23

3.8 Gerenciador de *plugins* . . . 25

3.9 Gerenciador de tarefas . . . 26

4 Uma aplicação do programa . . . 29

4.1 Zona de convergência do Atlântico Sul . . . 30

4.2 Conjunto de dados *Twentieth Century Reanalysis version 2* (20CRv2) . . . 32

4.3 Região de interesse . . . 34

4.4 Análises estatísticas . . . 36

5 Resultados e discussões . . . 40

5.1 Principais aspectos da implementação de um *plugin* . . . 40

5.2 Dados gerados com a aplicação . . . 52

5.3 Escalabilidade 57

6 CONCLUSÕES **59**

REFERÊNCIAS BIBLIOGRÁFICAS **59**

LISTA DE FIGURAS

Figura 2.1 Ganho em velocidade previsto pela lei de Amdahl. 5
Figura 2.2 Ganho em velocidade previsto pela lei de Gustafson. 5
Figura 2.3 Modelos de memória compartilhada 7
Figura 2.4 Modelo de memória distribuída 8

Figura 3.1 Diagrama de caso de uso do programa proposto. 13
Figura 3.2 Estrutura do núcleo. 15
Figura 3.3 Módulo núcleo. 16
Figura 3.4 Método *executar()* do módulo núcleo. 17
Figura 3.5 Módulo interface de usuário. 19
Figura 3.6 TUI 20
Figura 3.7 Tipos de variáveis 21
Figura 3.8 *Containers* para armazenamento de variáveis. 22
Figura 3.9 Parser 24
Figura 3.10 Módulo gerenciador de *plugins*. 26
Figura 3.11 Gerenciador de tarefas. 27
Figura 3.12 Diagrama de sequência da execução paralela de tarefas. 28

Figura 4.1 Imagem de satélite da ZCAS 31
Figura 4.2 Localização da UTE-LORM 35
Figura 4.3 Distribuições assimétricas 38
Figura 4.4 Dependência da forma da distribuição ao variar-se κ. 39

Figura 5.1 Utilização de um *plugin* criado. 51
Figura 5.2 Comparação dos dados de pressão da estação meteorológica de Vitória-ES com dados do conjunto 20CRv2. 52
Figura 5.3 Comparação dos dados de temperatura da estação meteorológica de Vitória-ES com dados do conjunto 20CRv2. 53
Figura 5.4 Comparação dos dados de umidade relativa da estação meteorológica de Vitória-ES com dados do conjunto 20CRv2. 54
Figura 5.5 Influência da variação de escala na média da pressão. 55
Figura 5.6 Influência da variação de escala no desvio padrão da pressão. . . . 55

Figura 5.7 Influência da variação de escala na assimetria da pressão. 56
Figura 5.8 Influência da variação de escala na curtose da pressão. 56
Figura 5.9 Ganho em velocidade obtido ao paralelizar-se o cálculo dos momentos estatísticos da pressão. 58

LISTA DE TABELAS

Tabela 4.1 Parâmetros disponibilizados pelo conjunto de dados 33

LISTA DE SIGLAS E ABREVIATURAS

20CRv2	*Twentieth Century Reanalysis version 2*
ABI	*application binary interface*
ACRE	*Atmospheric Circulation Reconstructions over the Earth*
API	*application programming interface*
ASCII	*American Standard Code for Information Interchange*
BDMEP	Banco de Dados Meteorológicos para Ensino e Pesquisa
CCEE	Câmera de Comercialização de Energia Elétrica
CF	*Climate and Forecast Convention*
CIRES	*Cooperative Institute for Research in Environmental Sciences*
COARDS	*Cooperative Ocean/Atmosphere Research Data Service*
CPU	*central processing unit*
ECMWF	*European Centre for Medium -Range Weather Forecasts*
ESRL	*Earth System Research Laboratory*
FIP Brasil Energia	Fundo de Investimento em Participações Brasil Energia

GCOS	*Global Climate Observing System*
GNU GPL	*GNU General Public License*
GPU	*graphics processing unit*
HDF	*Hierarchial Data Format*
IEEE	*Institute of Electrical and Electronics Engineers*
ISO	*International Organization for Standardization*
ISPD	*International Surface Pressure Databank*
LGSA	Linhares Geração S.A.
MIMD	*multiple-instruction multiple-data*
MISD	*multiple-instruction single-data*
MME	Ministério de Minas e Energia
MPI	*message passing interface*
NCAR	*National Center for Atmospheric Research*
NCEP	*National Centers for Environment Prediction*
NCL	*NCAR command language*
NOAA	*National Oceanic and Atmospheric Administration*
NSF	*National Science Foundation*
NUMA	*non-uniform memory access*
PRONAR	Programa Nacional de Controle de Poluição do Ar
SACZ	*South American Convergence Zone*

SIMD	*single-instruction multiple-data*
SISD	*single-instruction single-data*
SLURM	*Simple Linux Utility Resource Management*
SSH	*Secure Shell*
TUI	*terminal user interface*
UCAR	*University Corporation for Atmospheric Research*
UFRGS	Universidade Federal do Rio Grande do Sul
UMA	*uniform memory access*
UML	*Unified Modeling Language*
UTE-LORM	Usina Termelétrica Luiz Oscar Rodrigues de Melo
UV-CDAT	*The Ultra-scale Visualization Climate Data Analysis Tools*
WMO	*World Meteorological Organization*
WRCP	*World Climate Research Program*
ZCAS	zona de convergência do Atlântico sul
ZCIT	zona de convergência intertropical
ZCPS	*zona de convergência do Pacífico Sul*
ZFB	*zona frontal do Baiu*
ZPS	*zonas de precipitação subtropicais*

LISTA DE SÍMBOLOS

Símbolos Latinos

X	variável aleatória
Z	variável aleatória
TB	*terabyte*

Símbolos Gregos

γ	*Skewness* - assimetria -
κ	Curtose
σ	Desvio padrão
μ	Média
μ_n	N-ésimo momento estatístico

1 Introdução

1.1 Contexto

O presente trabalho está inserido em um projeto em colaboração com a Universidade Federal do Rio Grande do Sul (UFRGS), com o grupo ENEVA e com a sociedade anônima de capital fechado Linhares Geração S.A. (LGSA). O projeto contempla estudos de simulação de dispersão de poluentes e determinação de suas concentrações na atmosfera [Gisch, 2014; Loeck, 2014; Schramm, 2016]. Outra frente do projeto é voltada para a análise de fenômenos climáticos, como a ZCAS, a partir de séries temporais de dados climatológicos.

O grupo ENEVA é uma empresa brasileira com negócios complementares em geração, exploração e produção de hidrocarbonetos. Tem uma capacidade de geração de 2,2GW de energia elétrica, sendo 1.4GW provenientes do gás natural e 725 MW de carvão mineral. Na parte de óleo e gás, é a maior operadora de gás natural do Brasil, produzindo 8,4 milhões de metros cúbicos por dia [ENEVA, 2017].

A LGSA tem como propósito específico gerar energia elétrica controlada pelo Fundo de Investimento em Participações Brasil Energia (FIP Brasil Energia), que por sua vez é gerido pelo banco BTG Pactual. A companhia possui autorização do Ministério de Minas e Energia (MME) para atuar como produtor independente de energia elétrica até 2044, além de possuir contratos de comercialização de energia elétrica no ambiente regulado da Câmera de Comercialização de Energia Elétrica (CCEE) com 35 concessionárias de distribuição de energia elétrica até o ano de 2025 [LGSA, 2017]. A empresa administra a Usina Termelétrica Luiz Oscar Rodrigues de Melo (UTE-LORM), localizada no município de Linhares-ES.

A determinação da concentração de poluentes na atmosfera vem se tornando cada vez mais importante devido a forte presença de indústrias próximas a grandes centros urbanos. No Brasil, a legislação começou a contemplar políticas de monitoramento de poluição atmosférica com a criação do Programa Nacional de Controle de Poluição do Ar (PRONAR) [CONAMA, 1989]. As resoluções CONAMA n°386 de 2006 [CONAMA, 2006] e n°436 de 2011 [CONAMA, 2011] estabelecem limites máximos para a emissão de poluentes atmosféricos provindos de fontes fixas. No Espírito Santo, o decreto n°3463-R [Casagrande, 2013] estabelece novos padrões de qualidade do ar regional bem como um

plano de ação para melhorá-la. Neste decreto ficou determinado que, além do monitoramento em tempo real por meio de sensores, a concentração dos poluentes atmosféricos deverá ser acompanhada por modelos matemáticos, sendo que tais modelos devem ser aprovados pela comunidade científica.

Já a ZCAS é um fenômeno climático de grande importância na América do Sul. É responsável pelo aporte de grandes quantidades de água para a região do Sudeste brasileiro, muitas vezes causando prejuízos. Este fenômeno afeta fortemente a operação da UTE-LORM, pois o mesmo influencia diretamente nos níveis dos reservatórios das usinas hidrelétricas da região. Em situações extremas de chuva, os arredores da UTE-LORM ficam completamente alagados, dificultando o acesso ao local.

Todos os trabalhos relacionados a este projeto envolvem a utilização de muito recurso computacional. Assim, o projeto tem a disposição um *cluster*[1] com 3 nós, utilizando o gerenciador de recursos *Simple Linux Utility Resource Management* (SLURM)2003 e capacidade teórica de processamento de aproximadamente 2000GFLOPS/s[2]. Além disto, dispõe de um enorme banco de dados, com aproximadamente 6TB de informações sobre importantes variáveis climatológicas em formato binário *GRIB*. Os dados compreendem um período que vai da metade do século XIX até os dias atuais e são provenientes do projeto de reanálise 20CRv2 [Compo et al., 2015].

Para aproveitar por completo o potencial computacional oferecido pelo *cluster* e para conseguir processar grandes volumes de dados em tempo hábil, torna-se essencial que os programas que executam as simulações e análises sejam escritos de maneira paralela. Escrever programas paralelizados não é necessariamente uma tarefa fácil. São diversas as tecnologias disponíveis e a maioria delas requer bastante investimento de tempo em seu aprendizado, exigindo alguma experiência do programador para administrar os recursos compartilhados por todas unidades de processamento.

Outra tarefa comum a todos os trabalhos do projeto é a leitura e escrita de dados. Dados climatológicos são armazenados em formatos binários como *GRIB* e *NetCDF* e, portanto, para sua leitura é necessário o auxílio de uma biblioteca que seja capaz de fazê-lo. Quanto as simulações de dispersão de poluentes, os dados são geralmente salvos em formato de texto, que, além de ocupar muito mais espaço do que formato binário e ter

[1]Em computação, chama-se *cluster* um grupo de computadores ligados para formar um sistema computacional distribuído.

[2]FLOPS. Operações de ponto flutuante por segundo - *floating-point operations per second.*

um tempo de leitura/escrita muito maior, sua formatação e *layout* estão sujeitos ao estilo do programador.

1.2 Objetivos

Considerando tal contexto, a proposta do presente trabalho é desenvolver uma plataforma computacional capaz de facilitar em algum nível a escrita de programas paralelizados, bem como padronizar o sistema de leitura/escrita de dados. A plataforma, que foi escrita em *C++*, possui uma interface de usuário para que o mesmo solicite funções a serem executadas sobre os dados. A função por sua vez pode ser da biblioteca padrão, ou uma escrita por um usuário e adicionada ao programa por um sistema de *plugins*. Dados podem ser lidos/escritos por funções específicas e carregados em variáveis para posterior manipulação. Uma outra ideia por trás de tal ferramenta é a de possibilitar que os *plugins* escritos pelos diferentes desenvolvedores possam interagir. Assim, ao invés de diversos programadores criarem cada um seu programa independente, eles poderão criar programas capazes de aproveitar dados gerados por um outro, em um formato comum.

O desenvolvedor de uma determinada função o fará através de uma *application programming interface* (API). Nela, ele terá acesso a diversos serviços oferecidos para interagir com a plataforma. Pensada desta forma, a plataforma torna-se bastante flexível para a solução dos mais diversos problemas, tendo potencial para se tornar uma importante ferramenta para o desenvolvimento de programas orientados a dados climáticos.

O autor do trabalho está ciente que existem ferramentas similares, porém, grande parte delas não tem código aberto, são complexas e exigem a leitura de grandes manuais para seu aprendizado. A detenção da tecnologia empregada torna-se muito importante para casos em que se torna importante fazer adaptações à plataforma. No futuro, pretende-se disponibilizar o programa para acesso público com a licença GNU GPL.

A estrutura da dissertação será dividida em 6 capítulos. O capítulo 2 trata de uma revisão bibliográfica abordando os principais aspectos do estado da arte da computação paralela. No capítulo 3 serão explicados os detalhes envolvendo as decisões tomadas na criação da plataforma. No capítulo 4 é proposta uma simples aplicação da plataforma para ilustrar seu potencial, cujos resultados serão discutidos no capítulo 5. Finalmente, na última parte pode-se encontrar a conclusão.

2 Revisão Bibliográfica

Neste capítulo será realizada uma breve revisão bibliográfica sobre as tecnologias de paralelização de *softwares* existentes, sobre os dados com os quais se pretende trabalhar e sobre os programas similares já existentes no mercado.

2.1 Paralelização de softwares

Nas últimas décadas, a capacidade de processamento dos computadores vem aumentando continuamente ao passo que suas dimensões vêm diminuindo. Contudo, espera-se que esta crescente melhora dos processadores seja restringida pela limitação física de dissipação de grandes quantidades de energia em áreas tão diminutas. Desta forma, a tendência é ver-se cada vez mais programas feitos para rodar em múltiplos processadores ou em *clusters* de computadores, ou seja, programas paralelizados. A programação em paralelo é um conceito que já existe há muitos anos e, apesar de ser inerentemente mais complexa do que a programação serial, passou a ganhar uma importância maior recentemente, com a melhoria das tecnologias empregadas.

É importante salientar que a paralelização de um *software* é bastante dependente da natureza do problema a ser resolvido. Por exemplo, a montagem de um carro pode ser divida em várias tarefas, algumas destas devem ser executadas em uma determinada sequência, mas outras podem ser distribuídas entre diversas pessoas/máquinas, se estas estiverem à disposição. O limite teórico de melhora do desempenho de um algoritmo ao ser paralelizado é dado pela lei de Amdahl [Amdahl, 1967], mostrada na Equação 2.1. Nela, $S_{latencia}$ representa o ganho em velocidade, p, a fração de tempo de execução do programa passível de paralelização e N, o número de processadores disponíveis. Em problemas reais, p e N normalmente são dependentes, ao contrário do que supõe Amdahl em sua análise. Nestes cenários, a lei de Gustafson (Equação 2.2) se mostra uma aproximação mais realista e, podemos ver pelas Figuras 2.1 e 2.2, muito mais otimista do que a primeira, que acaba subestimando o ganho para programas mesmo quando estes apresentam uma pequeníssima fração serial [Gustafson, 1988].

$$S_{\text{latência}}(s) = \frac{1}{1 - p + \frac{p}{N}} \tag{2.1}$$

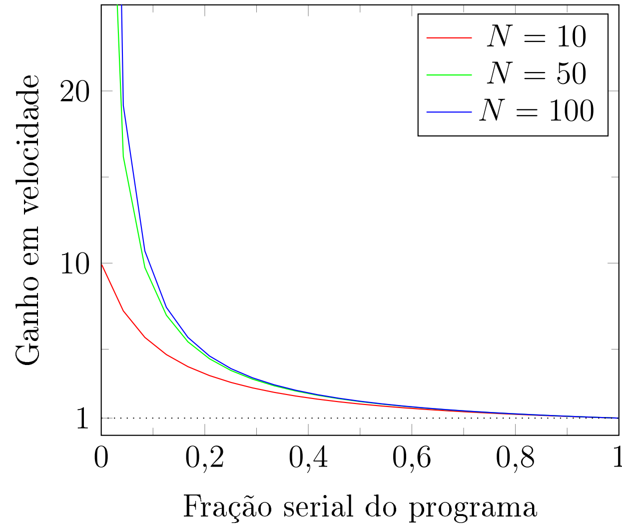


Figura 2.1 – Ganho em velocidade previsto pela lei de Amdahl.

$$S_{\text{latência}}(s) = 1 - p + Np \tag{2.2}$$

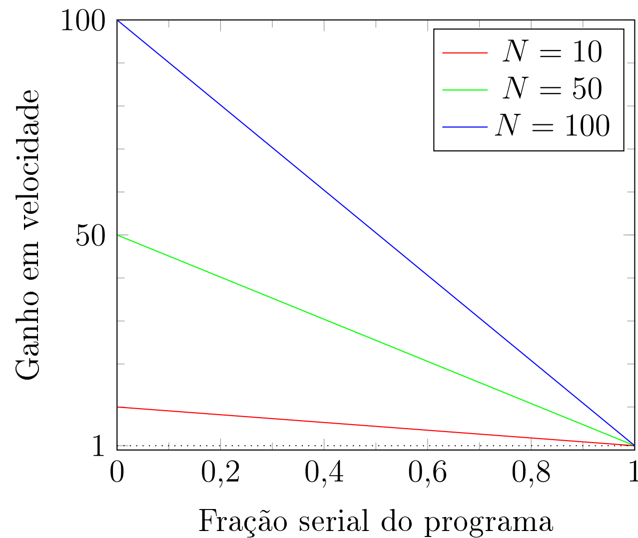


Figura 2.2 – Ganho em velocidade previsto pela lei de Gustafson.

Quanto ao *hardware*, são diversas as arquiteturas que suportam paralelismo. Um

computador, ou nó, pode possuir diversas *central processing units* CPU's (em português, unidades central de processamento), as quais podem possuir diversas unidades de processamento (*cores*, ou núcleos), sendo chamados então de *multicores*. *Clusteres* são compostos por vários nós (*desktop*, *workstation*, servidor) que são independentes e autossuficientes. Sistemas de *clusters* possuem boa escalabilidade[1] e por isso são muito adequados quando se necessitam grandes quantidades de memória e velocidade de processamento [Mishra e Sagnika, 2016]. *Graphics processing units* GPU's (unidades de processamento gráfico) também suportam paralelismo e algumas fornecem API's para escrita de programas que tirem proveito desta característica.

Flynn propõe em seu artigo uma outra forma de classificação, quanto as correntes de instruções e de dados, dividindo o modo de operação de computadores em 4 categorias [Flynn, 1972]. Estes podem operar tendo um fluxo único de instruções para um único conjunto de dados, em inglês, *single-instruction single-data* (SISD), sendo este o caso dos computadores seriais comuns. Podem ter um fluxo único de instruções para múltiplos conjuntos de dados, *single-instruction multiple-data* (SIMD), encontrado em processadores vetoriais e placas gráficas. Podem ter fluxo múltiplo de instruções em um único conjunto de dados, *multiple-instruction single-data* (MISD), sendo este último modo de operação pouco usual. E finalmente, podem ter um fluxo múltiplo de instruções sobre múltiplos conjuntos de dados, *multiple-instruction multiple-data* (MIMD), sendo este o caso dos sistemas distribuídos e o mais utilizado quando se trata de programas paralelos.

Outro aspecto importante a se discutir em computação paralela é o modelo de memória utilizado, que basicamente define como e quando a memória vai ser acessada e visualizada pelas diferentes partes do sistema [Kessler e Keller, 2007]. Podemos classificar tais modelos em dois grandes grupos: modelos de memória compartilhado e modelos de memória distribuídos. No primeiro caso (Figura 2.3), a memória do sistema é compartilhada por todos os processadores e é utilizada para escrever/ler dados de maneira assíncrona. Neste esquema, mecanismos de controle de acesso são necessários para evitar que os processadores acessem o mesmo endereço de memória ao mesmo tempo. Sistemas que utilizam memória compartilhada podem fazê-lo de maneira uniforme - *uniform memory access* (UMA) -, ou seja, todos os processadores tem os mesmos privilégios e compartilham o mesmo espaço físico de memória, ou de maneira não uniforme - *non-*

[1]Escalabilidade é a capacidade de um sistema, rede ou processo de lidar com crescentes quantidades de trabalho. Também pode ser considerada como o seu potencial de aumento para lidar com este crescimento.

uniform memory access (NUMA) -, onde cada processador tem sua própria memória que é mapeada para um endereço global, ficando disponível para os outros[2] [Mishra e Sagnika, 2016]. Nos sistemas de memória distribuídos (Figura 2.4), por outro lado, tem-se vários computadores (nós) conectados por uma rede. Os processos se comunicam trocando mensagens com operações especiais de recepção e envio. O acesso a elementos remotos deve ser feito por comunicação explícita, requerendo que variáveis temporárias sejam alocadas e as mensagens sejam construídas para transmissão [Grotendorst, 2002].

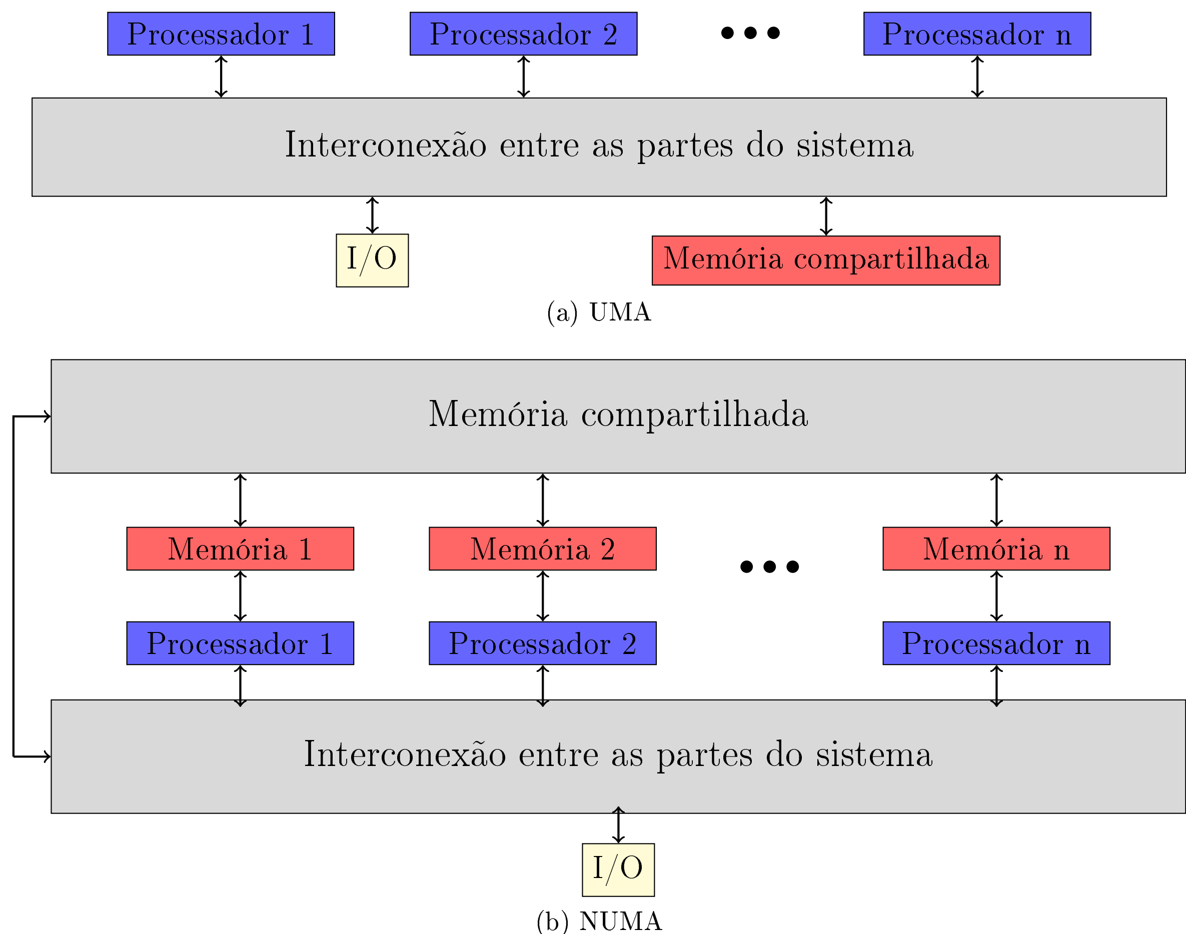


Figura 2.3 – Modelos de memória compartilhada

[2]No modelo NUMA o tempo de acesso a um determinado local na memória depende de sua localização.

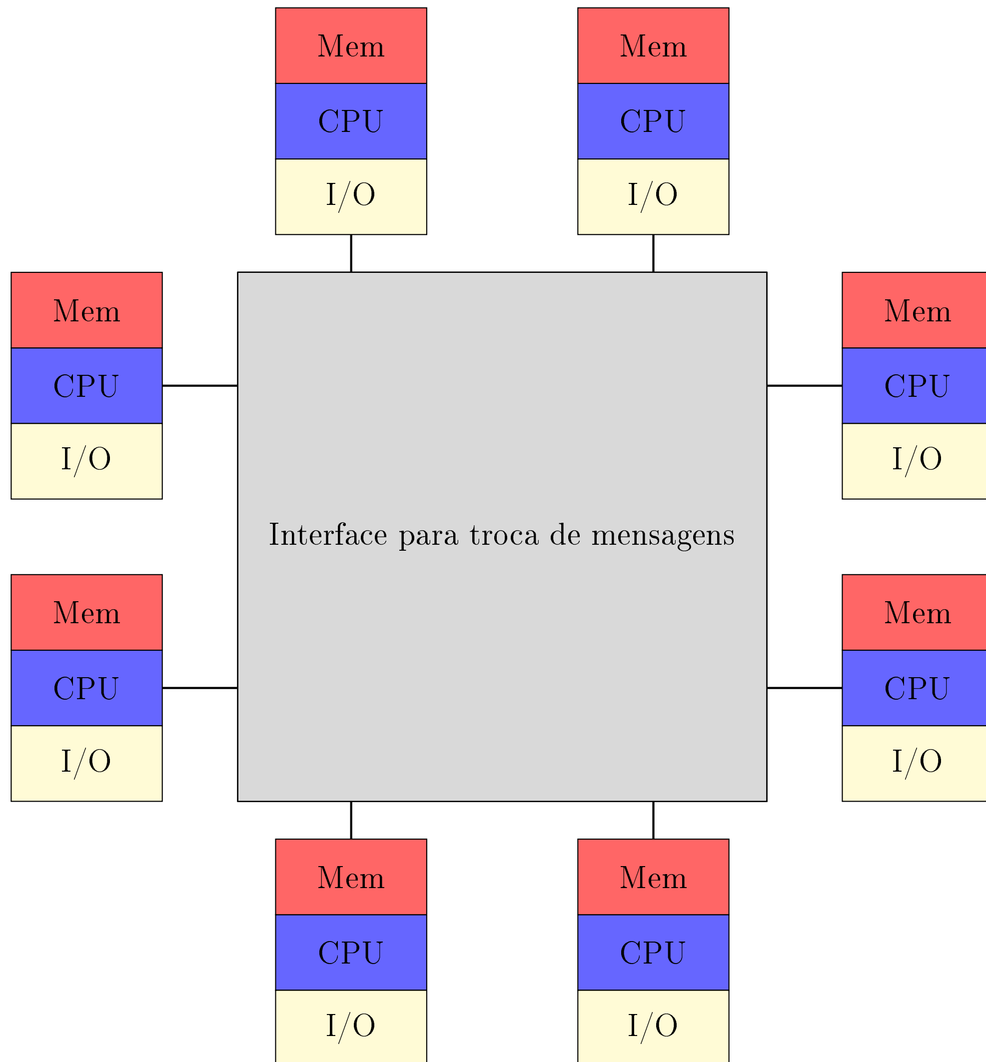


Figura 2.4 – Modelo de memória distribuída

Discutidos os conceitos básicos da computação paralela, pode-se começar a falar sobre os modelos de paralelização e as respectivas implementações que estão à disposição no mercado. Modelos de paralelização abstraem em algum nível detalhes referentes ao *hardware* e arquitetura de memória, facilitando a portabilidade de algoritmos para diferentes linguagens de programação e sistemas [Blaise, 2017]. Existem muitas propostas nesse sentido, destacam-se das demais o modelo de *threads* [3] e o de passagem de mensagens.

O modelo de *threads* tornou-se o modelo dominante para computadores paralelos de pequena escala. Nele um processo inicial inicia vários subprocessos (*thread*) que tem acesso a uma memória compartilhada. Cada *thread* possui uma memória local, bem como acesso aos endereços de memória globais. O acesso a memória deve ser controlado pelo

[3] Em programação paralela, *thread* é o nome dado a um subprocesso de um processo pai.

programador por um sistema de sinalização. POSIX *threads*, também conhecida como *pthreads*, é a implementação mais conhecida deste modelo, definida pelo norma *Institute of Electrical and Electronics Engineers* (IEEE) POSIX 1003.1c [IEEE, 1995], é utilizada pelos sistemas operacionais baseados em unix/linux. Entretanto, *pthreads* é implementado como extensão de linguagens sequenciais como C e FORTRAN (cujos compiladores tem conhecimento de um único processador) e em alguns casos sua eficiência pode ser muito dependente do *hardware* [Boehm, 2005]. Já *Cilk* [Blumofe et al., 1996] é uma liguagem de programação pensada para a criação de algoritmos *multithread*. Nela, o programador especifica a independência entre tarefas criadas dinamicamente que podem ser executadas localmente, no mesmo processador, ou em um outro.

OpenMP [Dagum e Menon, 1998] é um padrão da indústria criado em conjunto por um grupo de fabricantes/vendedores de *hardwares* e *softwares*, por desenvolvedores e por outras organizações. O mesmo vem crescendo em popularidade desde a criação dos processadores *multicore* e pode vir a eventualmente substituir completamente *pthreads*. Consiste em um conjunto de diretivas para linguagens *FORTRAN 77*, *FORTRAN 90* ou seus respectivos conjuntos de *pragmas* para linguagens como *C* e *C++*. Diretivas são comentários especiais que são interpretados pelo compilador usados para delimitar o começo e o fim das regiões paralelizáveis. Uma das vantagens do *OpenMP* é que o código escrito ainda pode ser interpretado como um algoritmo sequencial, desta maneira não se faz necessário a manutenção de duas versões do programa, uma paralela e uma serial.

O *message passing interface* (MPI) é um sistema padronizado e portável para passagem de mensagens entre os nós de uma rede de processadores com memória distribuída. Desenvolvido entre 1993 e 1997, esta API inclui rotinas para comunicação ponto a ponto, comunicação coletiva, comunicação unilateral e para paralelização de entrada e saída. Sua grande vantagem sobre o sistema de *threads* é que este suporta paralelização sobre mais de um nó de um *cluster*. Atualmente é suportado por diferentes linguagens de programação como *C*, *C++* e *Java*, além de existir uma grande disponibilidade de bibliotecas implementando este protocolo, sendo *OpenMPI* uma das mais conhecidas a possuir código fonte aberto [Snir et al., 1996; Kessler e Keller, 2007; Grotendorst, 2002].

Pode-se observar que tanto o modelo de *threads* como o MPI oferecem vantagens e desvantagens. Enquanto o primeiro é mais intuitivo e permite o acesso global à memória, o último é potencialmente mais poderoso, uma vez que permite a paralelização sobre todos

os nós de um *cluster* de computadores. Uma abordagem natural seria portanto misturar os dois ou mais modelos em sistemas híbridos, que se aproveitem dos melhores aspectos de cada um [Blaise, 2017].

2.2 Formato de dados climáticos

Dados computacionais podem ser armazenados basicamente em duas formas: em formato alfanumérico, ou seja, na forma de números e letras (*e.g.*, o padrão *American Standard Code for Information Interchange* (ASCII)), de maneira que pessoas sejam capazes de lê-los; em formato binário, *i.e.*, linguagem de máquina. Formatos alfanuméricos são práticos, pois exigem menos experiência do programador. Formatos binários são mais adequados em situações onde se necessita economizar memória ou velocidade de leitura, pois os mesmos não precisam ser traduzidos para linguagem de máquina antes de serem processados.

Na pesquisa climática, os arquivos binários comumente empregados derivam de 3 tipos: *GRIB*, *Hierarchial Data Format* (HDF) e *NetCDF*[4] [Rew et al., 1989]. Todos os 3 são portáveis, *i.e.*, independentes de máquina, e se auto descrevem. Arquivos com a capacidade de auto descrição podem ser lidos por um programa desde que o usuário conheça a estrutura dos dados. Metadado é o nome atribuído à informação usada para descrever tal estrutura. Metadados podem conter informações para descrever o arquivo em si e as variáveis que o mesmo contém (*e.g.*, unidades, nome do autor, nome e tamanho das variáveis, etc.) [Shea, 2013].

O formato *GRIB* foi desenvolvido em 1985 pela Organização Mundial de Meteorologia - *World Meteorological Organization* (WMO) - para a troca de grandes volumes de dados com centros e estações usando protocolos de telecomunicação de alta velocidade. Nele, os dados são arquivadas na forma de mensagens. Cada mensagem contém informações sobre duas dimensões espaciais e um determinado ponto no tempo [WMO, 2017].

Já os formatos HDF e *netCDF* possuem a interessante capacidade de compressão e paralelização sobre operações de leitura e escrita dos dados. Na verdade, a versão *netCDF-4* não passa de um envólucro da versão *HDF5* com a interface mais amigável do

[4]Os 3 formatos de arquivo, *GRIB*, *HDF* e *NetCDF* foram evoluindo ao longo dos tempos e atualmente possuem várias versões. Infelizmente para os usuários, as versões não são necessariamente compatíveis com suas respectivas predecessoras, o que pode se tornar bastante confuso.

netCDF. Diversas convenções são empregadas para garantir que os arquivos sejam criados e manipulados de uma maneira padrão, facilitando assim o desenvolvimento de *softwares*. As convenções mais utilizadas pelas comunidades de usuários de *netCDF* são *Cooperative Ocean/Atmosphere Research Data Service* (COARDS) e *Climate and Forecast Convention* (CF) [Eaton et al., 2017], sendo a última uma extensão da primeira. As comunidades de HDF aparentemente não seguem convenções [Shea, 2013].

2.3 Softwares para tratamento de dados climáticos

O tratamento de dados climáticos normalmente envolve 3 etapas principais: leitura/escrita dos dados armazenados em um determinado formato, manipulações matemáticas e visualização. Geralmente, o processamento de dados climáticos utiliza uma cadeia de *softwares*, um gerando informação para o próximo. O processamento pode ser feito através de uma linguagem compilada (*i.e.*, *FORTRAN*, *C*, *C++*), de uma linguagem interpretada (*R*, *Phyton*) ou através de linha de comando (*CDO*, *NCO*). Programas compilados são mais rápidos e eficientes, mas são muitas vezes tediosos a escrever. Um compilado das aplicações disponíveis é encontrado no *website* do programa *UNIDATA*[5], discutir-se-á brevemente as principais disponíveis gratuitamente.

NCO [Zender, 2017] e CDO são programas que oferecem uma série de operadores para manipulações sobre arquivos com dados climáticos. Tais operadores incluem funções para extração de informações do arquivo, operadores matemáticos como soma, multiplicação, adição, funções estatísticas, etc. Com o mesmo propósito a *ncar* desenvolveu a linguagem interpretada conhecida por *NCAR command language* (NCL) [Brown et al., 2012]. Estas ferramentas são muito úteis para realização de análises simples, que não exigem alta performance.

A ferramenta que mais se aproxima a proposta deste trabalho é o *software Paraview* [Ayachit et al., 2012], cuja a proposta é a visualização e análise de grandes conjuntos de dados utilizando supercomputadores. Outro programa nessa linha é o *The Ultrascale Visualization Climate Data Analysis Tools* (UV-CDAT) [Santos et al., 2013], que na verdade é construído em cima das tecnologias CDAT e *Paraview*.

[5]O programa *UNIDATA* foi criado em 1983 pela *National Science Foundation* (NSF). O programa consiste um uma diversa comunidade de universidade e centros de pesquisa com o objetivo de compartilhar dados geoclimáticos e as ferramentas necessárias para visualizá-los e processá-los. Atualmente o mesmo é gerido pela *University Corporation for Atmospheric Research* (UCAR).

3 Descrição do Programa

Neste capítulo serão abordados os principais aspectos envolvidos nas escolhas empregadas e no desenvolvimento do programa proposto.

3.1 Escopo

A tarefa de produzir algoritmos para tratamento de dados climáticos pode por vezes ser bastante trabalhosa. Dados climáticos estão disponíveis nos mais diversos formatos e sua leitura requer o aprendizado de API's e convenções que não estão no escopo do trabalho a ser desenvolvido. Problemas deste gênero envolvem grandes quantidades de informação, exigindo bastante memória e capacidade de processamento das máquinas, o que induz a utilização de tecnologias de paralelização dos recursos computacionais.

Assim, a proposta deste trabalho é criar uma plataforma com o intuito de facilitar o desenvolvimento de *softwares* para simulação e análise de clima. Tal plataforma deverá atender as seguintes proposições:

- Possuir uma interface leve, amigável e de fácil utilização;
- Abstrair a leitura e escrita de dados das mãos do programador. Dados serão carregados no programa e, uma vez nele, poderão ser manipulados por funções;
- Ser extensível. Além de possuir uma biblioteca padrão com funções básicas, outras funcionalidades poderão ser adicionadas na forma de *plugins*. O programa terá disponível uma API para que terceiros possam desenvolver suas próprias ferramentas;
- Facilitar a paralelização de algoritmos. As ferramentas para paralelização de *software*, apesar de excelentes ferramentas, requerem uma longa curva de aprendizado. A plataforma proposta espera, na medida do possível, retirar a responsabilidade do desenvolvedor de paralelizar seu código;
- Resultados gerados por uma função, poderão ser diretamente utilizados como entrada para outras.

Pode-se observar na Figura 3.1 as funcionalidades que se esperam que o programa possua. Nas próximas seções, discutir-se-ão suas principais características.

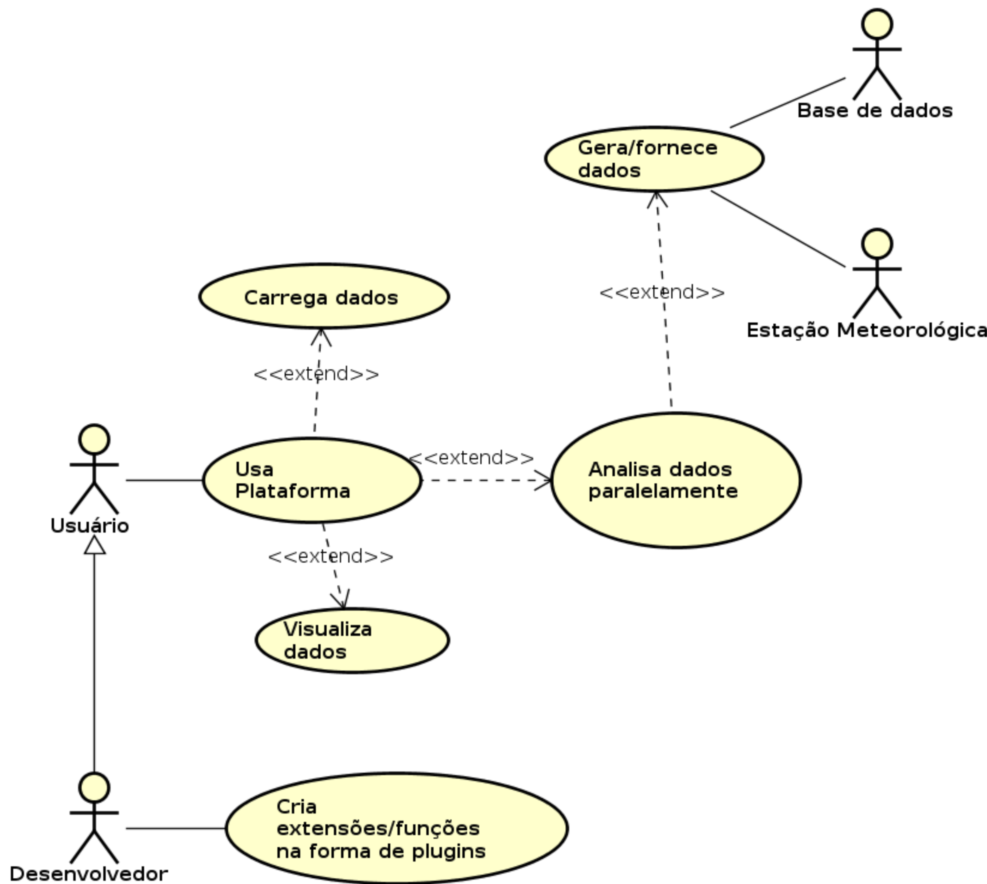


Figura 3.1 – Diagrama de caso de uso do programa proposto.

3.2 Linguagem de programação

Quando se deseja criar um *software* de alta performance, a linguagem de programação empregada se mostra de fundamental importância. Neste sentido, *C++* aparece como uma escolha muito atraente. *C++* é uma linguagem desenvolvida por Bjarne Stroustrup como uma extensão da linguagem *C*, padronizada pela *International Organization for Standardization* (ISO) [ISO, 2011]. Ela se propõe a ser performante e flexível, com algumas características adicionais de alto nível para a organização de programas.

É uma linguagem compilada, portanto, o código escrito é lido diretamente pelo computador, sem a necessidade de um interpretador para traduzi-lo para linguagem de máquina. Por ser orientada a objetos[1], ou seja, suportar encapsulamento, herança e

[1]suporta diversos paradigmas de programação: procedural, genérico, orientação a objetos, etc.

polimorfismo, *C++* torna a escrita de programas complexos mais clara e eficiente. Outra característica da língua é a de escrita de código genérico através de *templates*. Esta possibilita, por exemplo, escrever um algoritmo para uma classe vetor que poderá ser usado para diferentes tipos de variáveis (*int, double, string*, etc), evitando assim a redundância de código. [Stroustrup, 2013].

Além disto, é amplamente utilizada, com comunidade de usuários bastante ativa e dispõe de bibliotecas padrão para as mais diversas aplicações. Oferece suporte para praticamente todas as tecnologias de paralelização e possui compiladores para a grande maioria das plataformas existentes.

Obviamente, como é o caso de outras linguagens, *C++* também possui seus pontos negativos. Uma das maiores críticas à língua é justamente seu ponto forte. Por oferecer tantos recursos de alto e baixo nível, programas escritos nela são usualmente mais extensos e complexos, além de requerer muita atenção do programador para evitar vazamentos de memória e problemas deste gênero.

3.3 Estrutura do programa

Nesta seção pretende-se explicar de maneira objetiva as escolhas que foram feitas e o funcionamento do programa objeto de estudo. Para alguns dos diagramas mostrados ao longo do texto, foi utilizada a *Unified Modeling Language* (UML), uma linguagem para modelagem de sistemas padronizada, que facilita suas respectivas representações e compreensão [ISO, 2011].

Basicamente, a plataforma que se pretende desenvolver consiste de um módulo núcleo que faz a gestão de outros 5 módulos independentes (um módulo não enxerga o outro): interface de usuário, gerenciador de *plugins* [2], gerenciador de tarefas, um gerenciador de dados e um *parser*. Além de gerenciar a relação intermodular, o núcleo também oferece serviços para os demais módulos. Desta forma, os módulos podem solicitar informações e/ou tarefas aos outros, mesmo sendo reciprocamente invisíveis. Como se está trabalhando com uma linguagem orientada a objetos, faz bastante sentido modelar cada uma destas partes do programa como uma classe que defina bem seu comportamento, conforme a Figura 3.2.

[2] *Plugins* são extensões de programas com o intuito de aumentar a funcionalidade do mesmo, sem comprometer seu funcionamento. Por exemplo, navegadores de internet que permitem que o usuário instale *plugins* para adicionar características que não vêm com a versão padrão [CH, 2017].

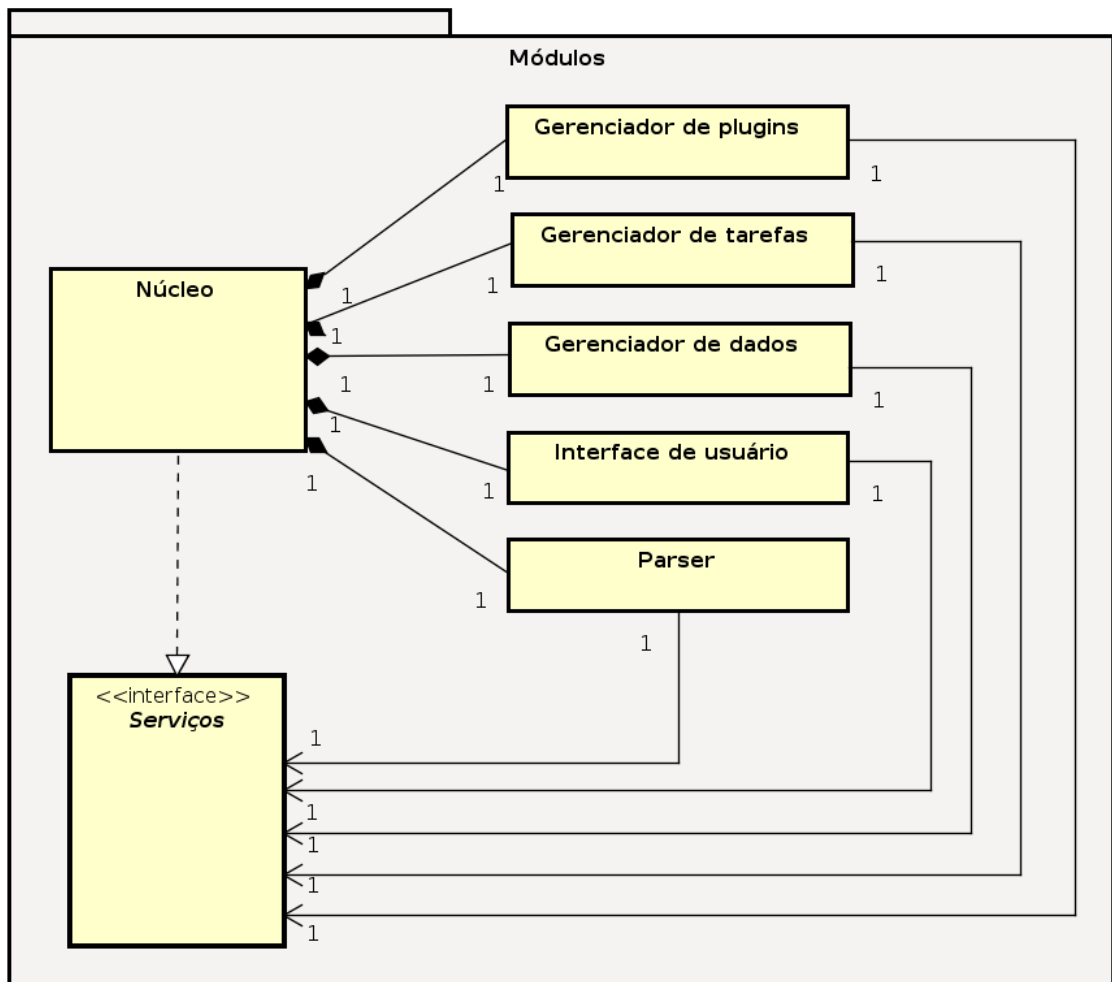


Figura 3.2 – Diagrama de classe representado a estrutura simplificada da relação do núcleo com seus módulos.

Ao ser inicializado, o programa carrega os módulos, inicializando a interface. Um *prompt* de comando pergunta ao usuário que função ele deseja chamar. As funções podem ou não ter argumentos e podem ou não gerar um resultado. O programa aceita variáveis de tipo inteiro, ponto flutuante, *string*, ponto e região, as quais serão explicadas com mais detalhes posteriormente. Dados climatológicos serão carregados por uma respectiva função que os armazena em uma variável de região.

Funções que usam regiões como argumento, tentam paralelizar a tarefa de maneira

orientada aos pontos que a compõem. O usuário que desejar implementar sua própria função, poderá escrever um *plugin*, também em *C++*, estendendo facilmente a aplicabilidade do programa.

3.4 Núcleo

A função do núcleo é basicamente administrar os recursos fornecidos pelos outros módulos. Podemos visualizar na Figura 3.3 uma representação do mesmo em um diagrama de classes.

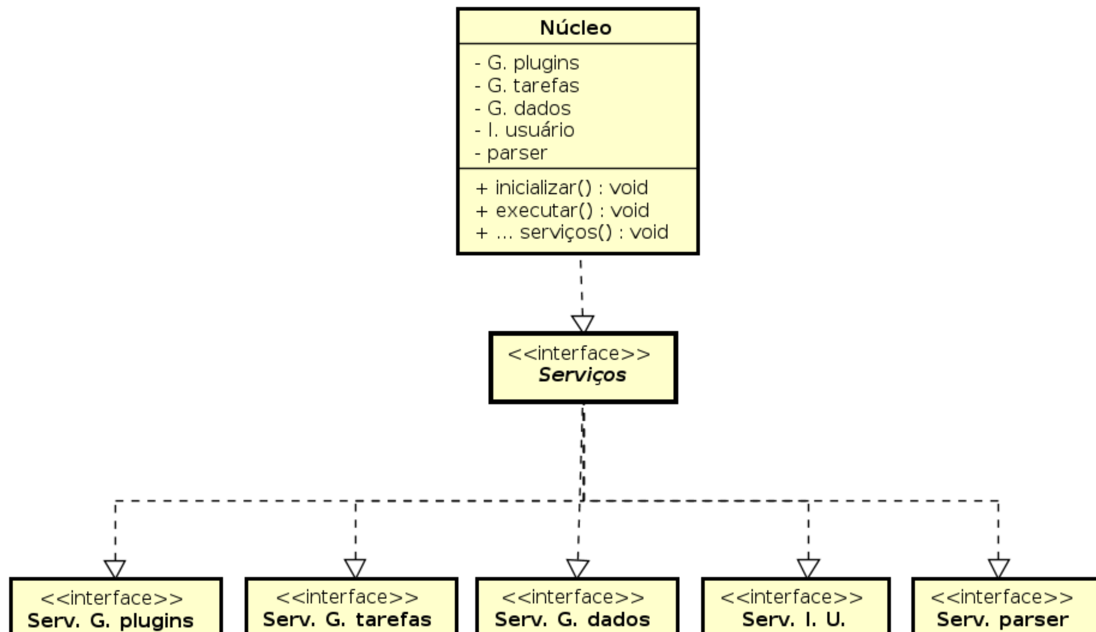


Figura 3.3 – Módulo núcleo.

Para troca de informações e tarefas entre os módulos, o núcleo implementa a interface[3] de serviços. Como o núcleo é o único módulo que visualiza os outros, é o único que tem acesso a todas as funcionalidades oferecidas. Assim, ele encapsula os métodos na interface de serviço que é passada para os outros módulos. Pode-se notar que a interface

[3] Em programação orientada a objetos, uma interface é um meio comum para objetos não relacionados se comunicarem. Um objeto que implementa a interface se "compromete" em disponibilizar alguns métodos para outros objetos através da mesma. Em *C++*, consiste de uma classe puramente abstrata sem propriedades. Classes que herdam a interface devem implementar seus métodos.

de serviços é por sua vez "dividida" em subinterfaces de serviços, uma para cada módulo. Esta arquitetura garante que os módulos tenham acesso somente ao serviços que lhe concernam. Garante também que cada módulo seja independente do outro, o que facilita o desenvolvimento e depuramento do *software*.

A classe núcleo possui dois métodos públicos. O primeiro, *inicializar()*, é responsável pela instanciação dos submódulos, registro de sinais e outras configurações. Já o método *executar()* fica encarregado justamente da execução do programa. Trata-se de um *loop* infinito onde, em cada laço, o núcleo espera um comando do usuário gerado no módulo de interface, interpreta o mesmo no *parser*, solicita ao gerenciador de *plugins* pela tarefa e, se o caso for, manda o administrador de tarefas adicionar o pedido à sua lista de execução (Figura 3.4). O *loop* repete-se até que o usuário solicite o término do programa.

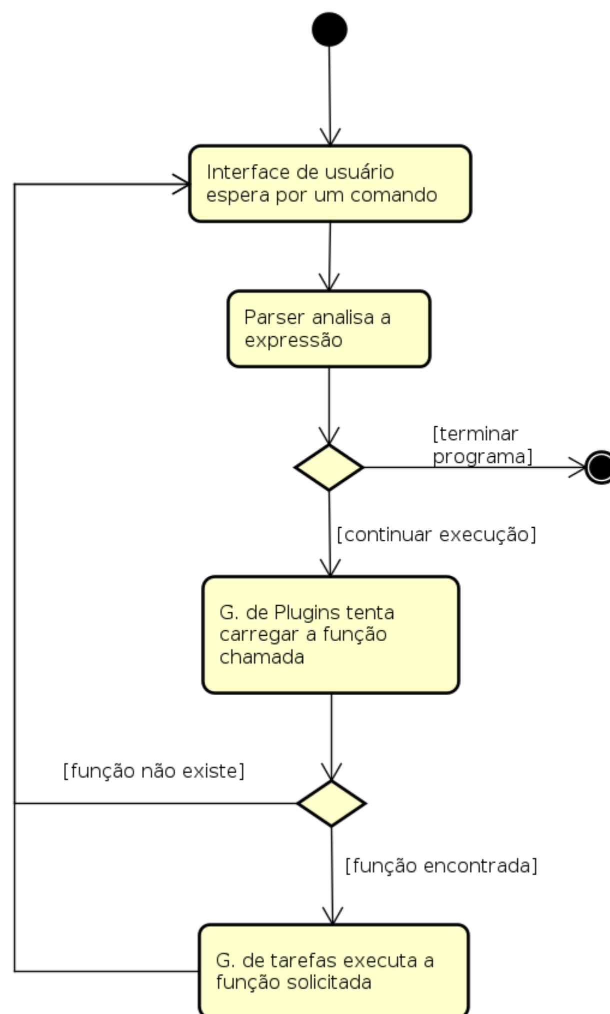


Figura 3.4 – Método *executar()* do módulo núcleo.

3.5 Interface de usuário

Para uma boa aceitação de um programa por seus usuários, é essencial que este seja fácil de utilizar e rápido a aprender. Não é preciso dizer que a interface do usuário tem um papel fundamental neste sentido e seu *design* precisa ser muito bem pensado.

Como o programa é feito para ser utilizado em *clusters*, muitas vezes situados a longas distâncias do usuário, propõe-se uma interface de terminal, em inglês *terminal user interface* (TUI). TUI's têm a vantagem de ser muito leve e poderem ser utilizadas via *Secure Shell* (SSH) [Ylonen e Lonvick, 2006], um protocolo criptografado, modelo cliente-servidor para acesso remoto a computadores. Para o desenvolvimento da interface foi utilizada a biblioteca *ncurses*, a qual fornece uma API para escrita de TUI's portáveis para diversos terminais [Padala, 2005].

Abaixo, estão resumidas as especificações propostas para a TUI, pensadas com o objetivo de proporcionar uma boa experiência de uso ao utilizador final do programa.

- Ser leve. O programa é feito para ser utilizado em *clusters*, muitas vezes localizados a uma longa distância do usuário. Uma interface leve é importante, uma vez que está será utilizada pela rede.

- Possibilitar a manipulação de mais de uma janela na tela do terminal. Desta forma, o usuário pode trabalhar com por exemplo, uma janela para entrada de comandos, outra para mostrar os resultados e outra com a área de trabalho e as variáveis declaradas no programa.

- Quando a janela principal for redimensionada, as outras janelas devem atualizar seu tamanho para manter uma aparência agradável.

- Um histórico de comandos é mantido para que o usuário possa reutilizá-los. Comandos antigos poderão ser acessados com as teclas cima/baixo.

- O usuário deve poder requisitar uma lista com as funções disponíveis para utilização.

Na Figura 3.5 pode-se ver uma versão simplificada de como foi modelizado o módulo de interface de usuário.

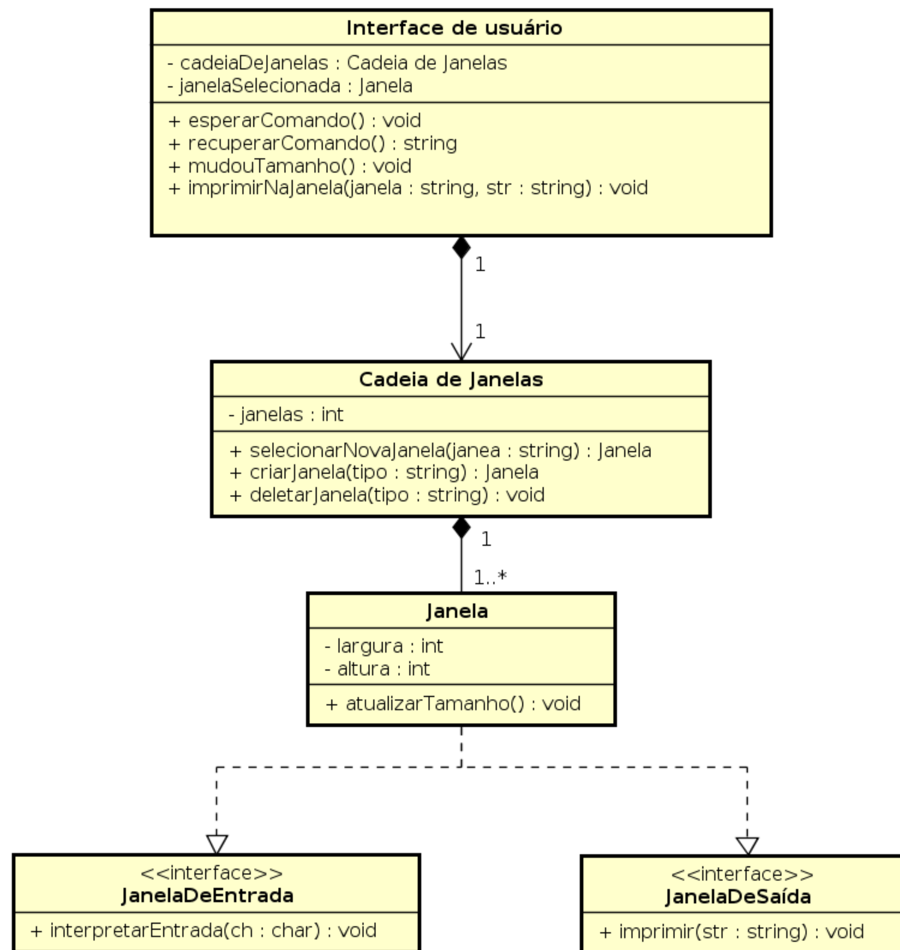


Figura 3.5 – Módulo interface de usuário.

Pode-se observar no diagrama de classe que o módulo é composto por uma “cadeia de janelas”. Cada uma ocupa um espaço na tela do terminal e possui um respectivo conteúdo. As janelas podem ser de saída de texto, como é o caso da janela de resultados, de entrada de comandos, ou ambos tipos. Uma cadeia nada mais é do que é uma sequência destas janelas, quando a tela do terminal muda de tamanho, a cadeia de janelas que é responsável por atualizar cada uma de suas unidades.

Na Figura 3.6 pode-se ver a interface do programa. Neste exemplo, foi criada uma variável *string*, com o nome de *hello_world*. Em seguida seu conteúdo foi impresso na tela de resultados à direita, ao chamar a função *print()*.

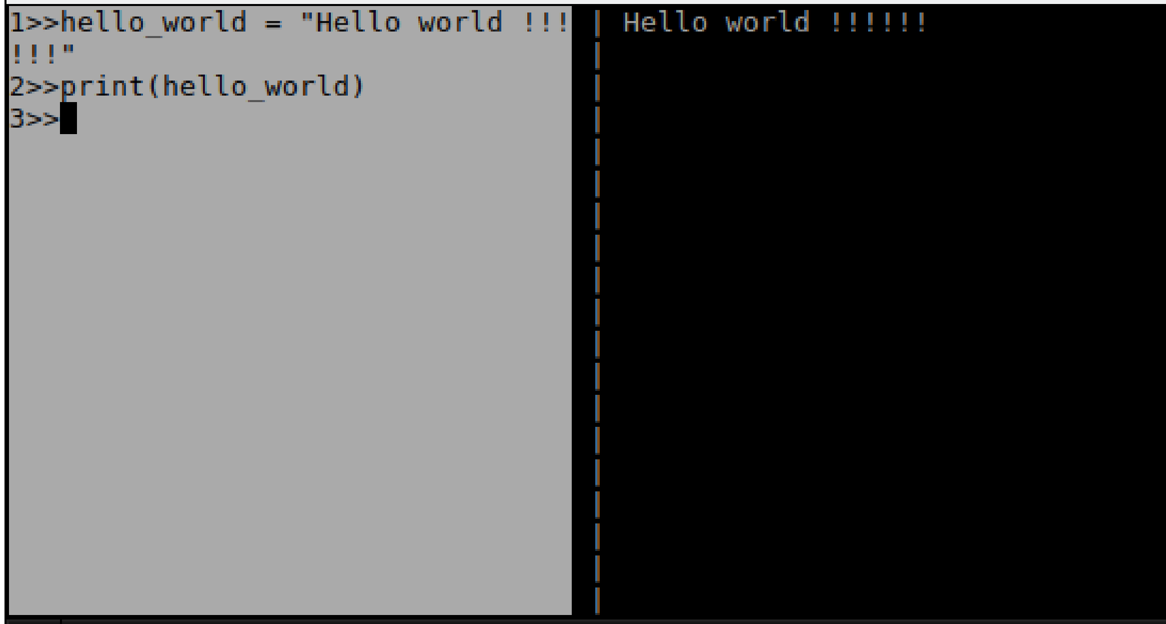


Figura 3.6 – TUI

3.6 Gerenciador de dados

Este módulo é responsável pela criação e manipulação de variáveis do programa. Como mencionado anteriormente, as variáveis podem assumir valores de tipo inteiro, ponto flutuante, *string*, ponto e região, conforme a Figura 3.7.

Todos os tipos de variáveis são derivados de um tipo base, que implementa suas características e métodos comuns. O tipo base também implementa os métodos *bloquear()* e *desbloquear()* da interface *mutex*[4]. Desta forma, quando se deseja utilizar uma variável, deve-se bloquear seu acesso para que outros processos concorrentes não tentem modificá-la concomitantemente.

As variáveis do tipo inteiro e ponto flutuante podem armazenar mais de um valor, como em um vetor, bastando para isso modificar seu tamanho. Os valores são disponibilizados para o desenvolvedor através de um ponteiro para seus respectivos tipos. Variáveis de tipo inteiro e ponto flutuante também podem ter uma unidade e uma escala de tempo associada, se este for o caso.

[4] Em programação paralela, exclusão mútua, *i.e.*, *mutex - mutual exclusion* -, é uma técnica para evitar que dois *threads* acessem algum determinado recurso ao mesmo tempo. Quando um processo bloqueia o acesso ao recurso, os outros têm que esperar seu desbloqueio para utilizá-lo.

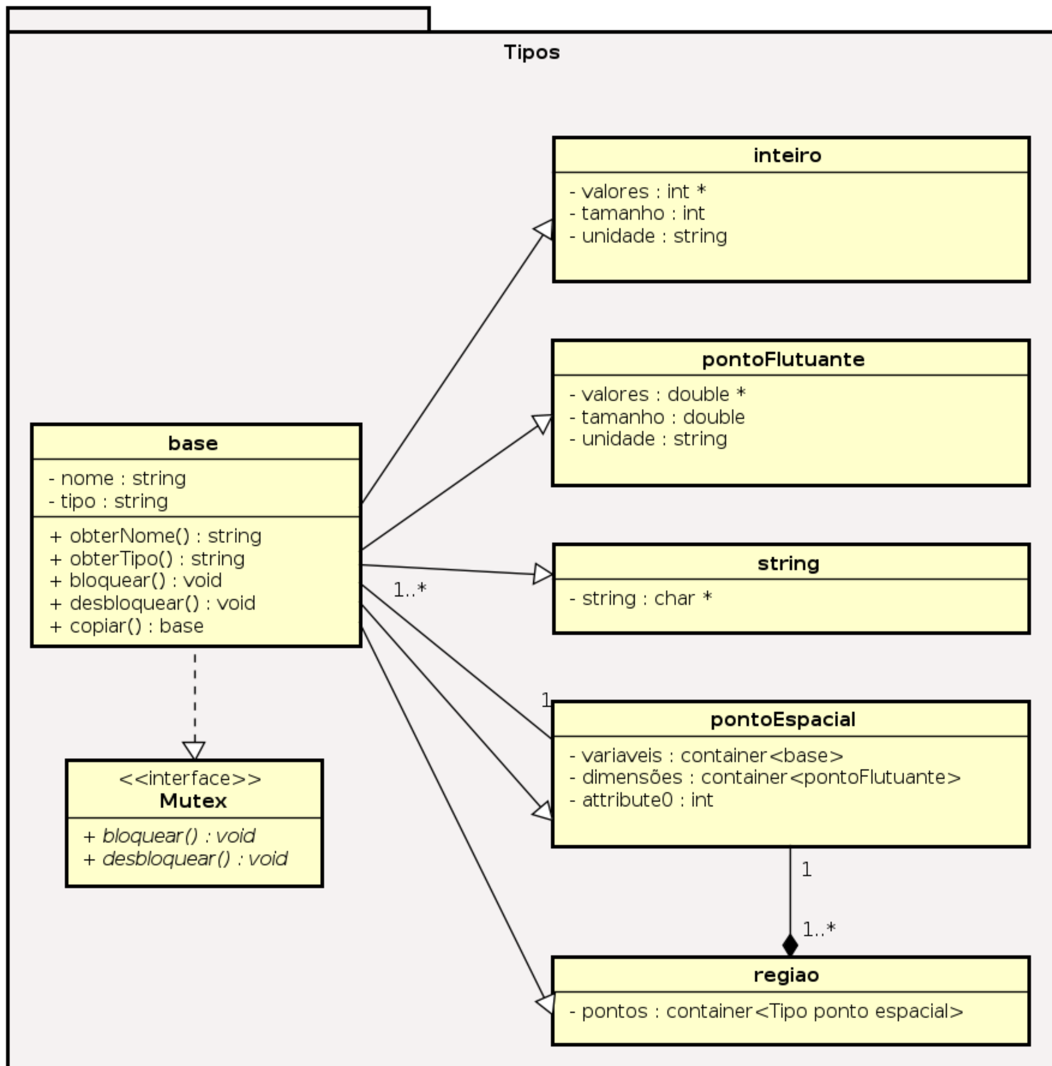


Figura 3.7 – Tipos de variáveis

Dados climáticos carregados a partir de um arquivo são armazenados em variáveis do tipo região. Cada região é formada por pontos espaciais, que carregam a informação referente à sua localização no espaço (*e.g.*, latitude, longitude, nível, etc...) [5]. Um ponto pode conter varias variáveis (*e.g.*, temperatura, pressão, umidade) associadas, cada qual é armazenada na forma de uma série temporal referente ao período analisado.

Cada tipo de variável tem também um *container* associado. Em programação, um *container* é uma estrutura de dados, cuja a função é armazenar outros objetos de um determinado tipo. São úteis quando, por exemplo, se deseja passar um grupo de

[5] A localização espacial de um ponto pode ser definida por uma, duas ou mais dimensões.

parâmetros para uma determinada função. Cada *container* também possui um respectivo iterador, para que se possa iterar entre seus elementos. O módulo de gerenciamento de dados não é nada mais do que um *container* genérico, que pode armazenar qualquer tipo de variável. Ver Figura 3.8.

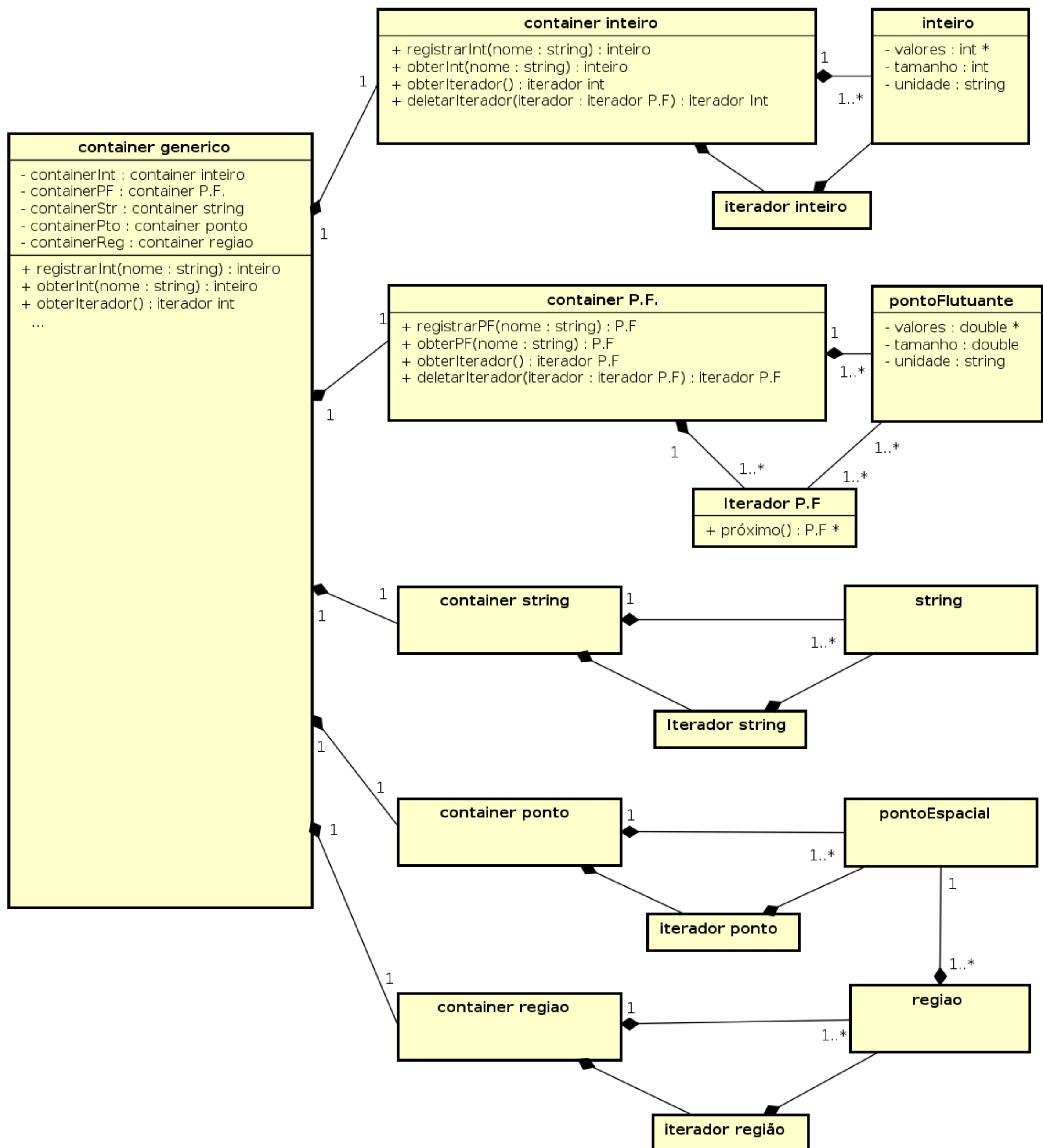


Figura 3.8 – *Containers* para armazenamento de variáveis.

3.7 *Parser*

O processo de *parsing* consiste em analisar um determinado segmento de texto e interpretá-lo de acordo com alguma gramática. Em ciências da computação, refere-se ao processo de separar uma *string* em pedaços menores de informação, resultando em uma estrutura de dados, chamada de *parse tree* - "árvore" -, que relaciona estes pedaços sintaticamente. *Parsers* são usados em compiladores e interpretadores para traduzir textos de uma linguagem de computação de alto nível para linguagem de máquina.

Frequentemente, os *parsers* são constituídos por duas partes: um analisador léxico e um analisador sintático. O primeiro é responsável por receber o texto de entrada e separá-lo em pedaços menores de informação que contenham algum significado, também chamados de *tokens*. Já o analisador sintático é encarregado de ordenar estes *tokens* em uma *parse tree*. Por exemplo, ao analisar-se a sentença matemática representada na Equação 4.1.

$$A = 5 + 3.2 - 8/2 \tag{3.1}$$

Num primeiro momento o analisador léxico deverá separar a frase matemática em *tokens*, como mencionado. Matematicamente, estes fragmentos com significado são: A, "=", 5, "+", 3.2, "-",8, "/" e 2. Posteriormente, o analisador léxico deve interpretar os símbolos, "+","-" e "/" como operadores matemáticos, os símbolos 5, 8 e 2 como números inteiros, o símbolo 3.2 como um número em ponto flutuante, a letra A como uma variável e o "=" como o símbolo de atribuição.

Terminada a etapa de "*tokenização*", entra em ação o analisador sintático. Uma de suas atribuições é garantir que a frase matemática seja livre de erros, como seria o caso de dois sinais "+" seguidos. Como todo sabemos, expressões matemáticas devem ser analisadas em uma certa ordem, operações de divisão e multiplicação tem precedência sobre subtração e adição, sentenças dentro de parenteses devem ser analisadas primeiro e assim por diante. A outra função do analisador sintático é justamente organizar esta estrutura em uma *parse tree*, garantindo que as operações sejam executadas na ordem certa. Por exemplo, a Equação 4.1 deve ser analisada na seguinte ordem:

$$1 \rightarrow 8/2 = 4 \tag{3.2}$$

$$2 \rightarrow 3.2 - 4 = -0.8 \tag{3.3}$$

$$3 \rightarrow 5 + (-0.8) = 4.2 \tag{3.4}$$

$$4 \rightarrow A = 4.2 \tag{3.5}$$

Parsers de compiladores, como por exemplo um compilador para a linguagem *C* ou *C++*, podem atingir níveis de complexidade bastante elevados. Como no programa proposto as expressões a serem analisadas são relativamente simples, fez-se a opção por um desenho de *parser* recursivo de fácil implementação, inspirado naquele de Jong [Jong, 2006].

Desta forma, o processo de *parsing* proposto começa por identificar cada *token*. Se os *tokens* forem algum tipo de variável, o mesmo tem que classificá-lo como um dos 5 tipos suportados (inteiro, ponto flutuante, *string*, ponto ou região) e alocar a memória necessária. Caso sejam o nome de uma função, o *parser* usa um serviço para perguntar ao gerenciador de *plugins* se aquela função existe. Caso seja uma variável, ele deve verificar com o gerenciador de dados se a variável já existe ou se ela deve ser criada e memória deve ser alocada.

Para montar a *parse tree*, o *parser* conta com o auxílio da classe Função (ver Figura 3.9). Se outras funções tiverem preferência, o *parser* pode adicioná-las como subfunções. Assim, uma ordem bem clara de prioridade de execução é estabelecida.

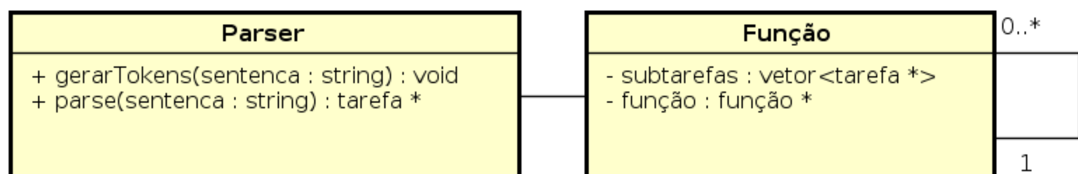


Figura 3.9 – Parser

3.8 Gerenciador de *plugins*

O gerenciador de *plugins* é o responsável por carregar as funções disponíveis em bibliotecas dinâmicas e disponibilizá-las para os outros módulos. *Plugins* são a maneira mais segura de estender e evoluir um sistema, permitindo a terceiros adicionar funcionalidades ao mesmo sem desestabilizá-lo. Porém, pelo fato de *C++* ser uma linguagem extremamente dependente de plataforma e compilador, torna-se bastante difícil implementar uma interface de *plugins* portável. O padrão *C++* não define uma interface binária de aplicações - *application binary interface* (ABI) -, o que implica que bibliotecas criadas a partir de diferentes compiladores, ou mesmo de diferentes versões de um mesmo compilador, não sejam necessariamente compatíveis. Além disso, a língua não define o conceito de carregamento dinâmico de bibliotecas, deixando sua implementação a encargo das diferentes plataformas. Atkinson *et al.* [Atkinson et al., 2010] listam alguns dos principais aspectos a se pensar em termos de ABI quando se desenvolve um *plugin* em *C++*;

Um dos maiores problemas relacionados ABI refere-se aos algoritmos de *name mangling*[6] implementados pelos diferentes compiladores. Uma solução que contorna parcialmente esse problema é somente acessar os objetos através de métodos virtuais (acessá-los através de uma interface), que não são afetados pelo *name mangling.* Entretanto, ainda teria-se outro problema: no sistema operacional em *linux*, não existe maneira direta para criar ou instanciar objetos a partir de uma biblioteca carregada dinamicamente. Este último pode ser resolvido utilizando uma função *linkada* na linguagem *C* (compátivel com *C++* e não sujeita a *name mangling*) para a "fabricação" dos objetos.

No fim, a solução mais eficaz para se montar um sistema de *plugins* em *C++* é começar por uma interface escrita puramente em *C*, que é bem padronizada e compatível entre compiladores, e depois encapsulá-la em *C++*. O algoritmo utilizado para o gerenciador de *plugins* deste trabalho é baseado no desenvolvido por Sayfan [Sayfan, 2007]. Na Figura 3.10 pode-se ver o diagrama do mesmo.

[6] Para diferenciar as propriedades e métodos de cada classe, os nomes de funções, estruturas e tipos de dados são codificados antes de serem *linkados*. Como *C++* não define um padrão de ABI, cada compilador pode ter seu próprio algoritmo de codificação.

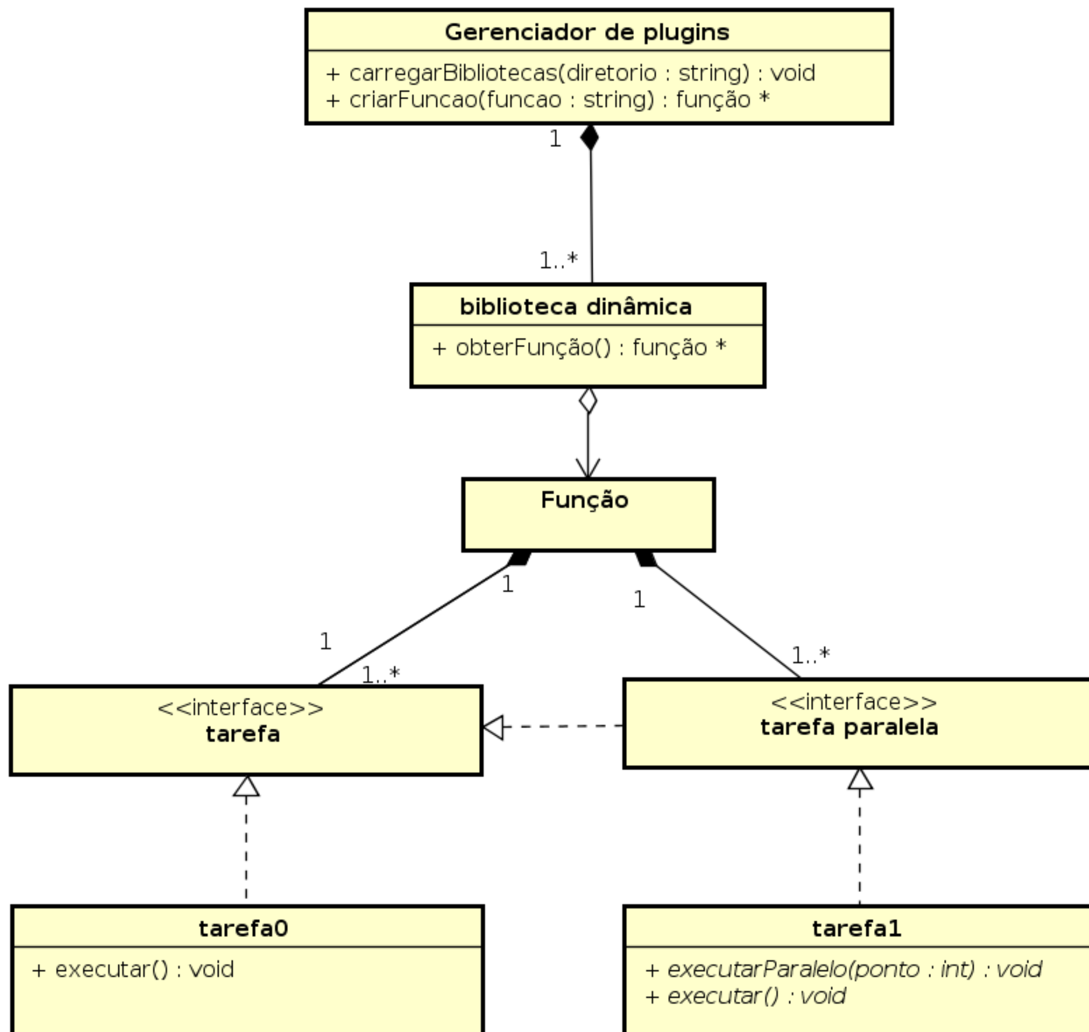


Figura 3.10 – Módulo gerenciador de *plugins*.

Em sua inicialização, o núcleo chama o método *carregarBibliotecas*, com o caminho do diretório configurado para armazenar os *plugins*. Cada biblioteca é armazenada num objeto do tipo biblioteca dinâmica, e pode guardar um ou mais objetos função. Estas, por sua vez, são compostas de objetos tarefas, paralelas ou seriais. Algumas funções podem agir como paralela ou serial dependendo dos argumentos que forem passados, bastando para isso que a mesma chame a tarefa adequada para a situação. Argumentos são passados para as funções através de um objeto *container* genérico.

3.9 Gerenciador de tarefas

Depois do usuário solicitar um comando, o *parser* ser bem sucedido em interpretá-lo e o gerenciador de *plugins* encontrar todas as funções solicitadas, o gerenciador de

tarefas assume o controle do programa. Sua função, como o nome sugere, é garantir que as tarefas sejam executadas na ordem correta. Por exemplo, quando se deseja extrair a média temporal de uma variável composta P/T (P - pressão - e T - temperatura -) em uma determinada região do planeta. Primeiramente, o gerenciador de tarefas vai garantir que a operação P/T seja efetuada sobre cada ponto daquela região, para em um segundo momento aplicar a média temporal sobre a variável composta. As operações P/T e média são executadas paralelamente, ou seja, as informações referentes a cada ponto da região são enviadas para diferentes *threads*, os quais são encarregados de executá-las.

Em sua primeira versão, o gerenciador de tarefas suportará somente paralelização por *threads*. Isso implica que em um *cluster* de computadores por exemplo, o programa só conseguirá utilizar os processadores de um dos nós para execução concorrente do algoritmo. No futuro, o objetivo é tornar o sistema híbrido, suportando *threads* dentro de cada nó e MPI para sincronização externa.

Na Figura 3.11, pode-se ver um diagrama de como este módulo interage com outras partes do sistema.

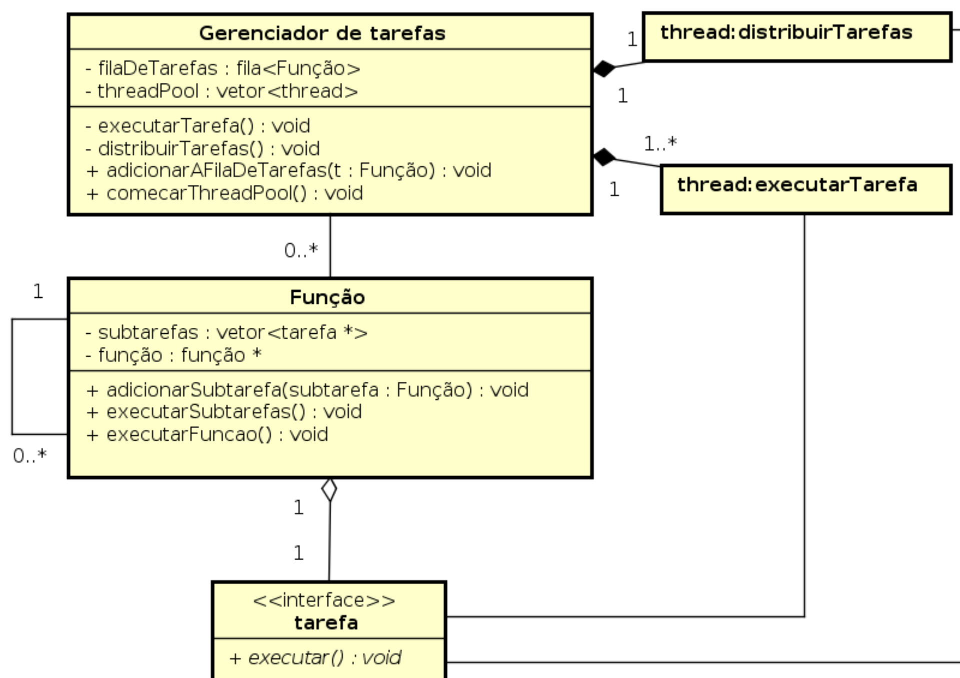


Figura 3.11 – Gerenciador de tarefas.

Ao inicializar, o núcleo chama o método *comecarThreadPool()* para lançar os *th-*

reads e atribuir a cada um a sua rotina de execução. Uma *thread pool* é um padrão de *design* do paradigma de programação paralela que consiste em criar vários *threads* no início do programa e deixá-los em modo de espera até que alguma tarefa seja solicitada. As tarefas, por sua vez, têm uma fila de execução e vão sendo chamadas conforme os processadores vão se liberando. Mantendo uma *thread pool* aumenta-se a performance do modelo de paralelização, diminuindo a latência de execução ocasionada pela constante criação e destruição de novos *threads* [Ling et al., 2000].

No referido programa, existem 3 tipos de rotinas de execução operando paralelamente, duas destas possuem somente uma instância, enquanto a terceira possui múltiplas. A rotina principal roda o método *executar()* do módulo núcleo (Figura 3.4) e, portanto, é responsável por toda a interação com o usuário gestão dos módulos. A rotina *distribuirTarefas()* (Figura 3.11) fica aguardando por uma função solicitada pelo usuário. Quando esta chega, a rotina é encarregada de distribuir suas subtarefas e respectivas informações para cada *thread* de execução disponível. Estes últimos operam segundo a rotina *executarTarefa()* e são encarregados de rodar as tarefas da função solicitada. Quando a execução da função é terminada, ele é responsável por fazer a limpeza da memória e enviar um sinal para o a rotina de distribuição, avisando que está pronto para trabalhar novamente. Na Figura 3.12, o diagrama de sequência mostra mais claramente o processo.

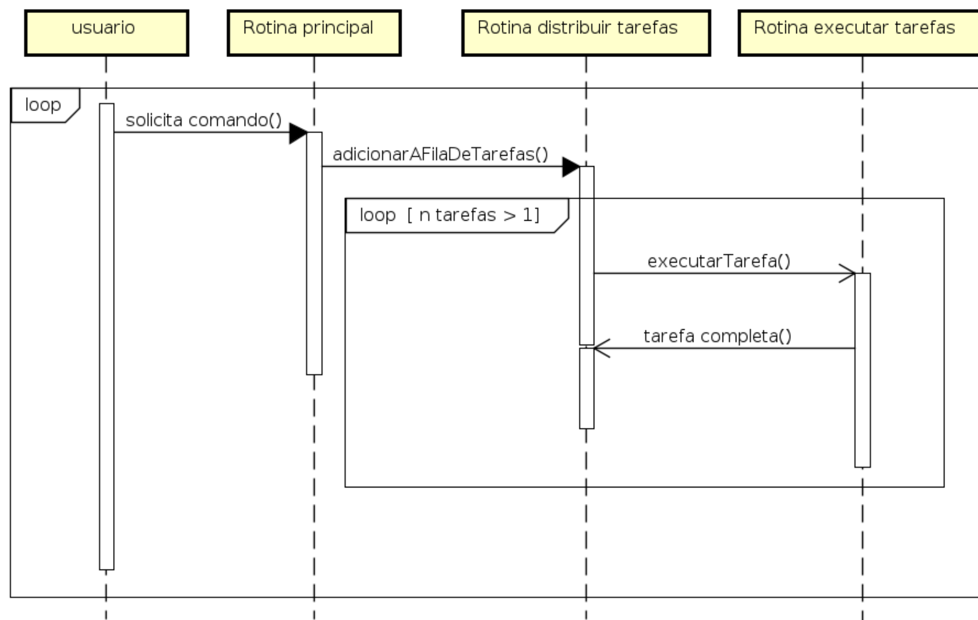


Figura 3.12 – Diagrama de sequência da execução paralela de tarefas.

4 Uma aplicação do programa

Para ilustrar o funcionamento do programa e demonstrar seu futuro potencial, foi desenvolvida uma aplicação simples para a análise de dados históricos referentes a um fenômeno bastante conhecido na climatologia, a ZCAS. Este é o principal fenômeno de grande escala responsável pelo regime de chuvas das regiões sul e sudeste do Brasil, aportando grandes quantidades de água da Amazônia e dos oceanos nos meses de primavera e verão.

Escolheu-se estudar a ZCAS, pois a mesma está diretamente relacionada com o enchimento ou escassez dos reservatórios de água de hidrelétricas da região. Assim, tal fenômeno influencia diretamente no preço da energia elétrica provindo de outras fontes. O foco do presente trabalho será direcionado à realização da análise da ZCAS no estado do Espírito Santo, nas proximidades da termoelétrica de Linhares-ES.

O objetivo é utilizar um banco de dados climáticos provindos do projeto 20CRv2 (Seção 4.2) e aplicar os 4 primeiros momentos estatísticos às séries temporais de algumas variáveis selecionadas em diferentes intervalos de tempos. Esta análise é importante para futuros trabalhos que pretendam utilizar tal banco de dados, permitindo monitorar o comportamento das variáveis nas diversas escalas temporais e na medida do possível reduzir o conjunto de dados ao mínimo necessário à captura dos fenômenos que se deseja observar. Como os dados do projeto 20CRv2 são advindos de modelos de reanálise, faz-se interessante num primeiro momento compará-los com dados reais. Por este motivo, pretende-se comparar de forma qualitativa a validade dos mesmos com dados provenientes do Banco de Dados Meteorológicos para Ensino e Pesquisa (BDMEP) (estação meteorológica 83648, situada em Vitória - ES), disponíveis para o período de 1990 à 2017 [INMET, 2017].

4.1 Zona de convergência do Atlântico Sul

A ZCAS é um sistema meteorológico típico de verão que, junto com a zona de convergência intertropical (ZCIT), é responsável por grande parte das chuvas do Norte, Nordeste, Centro-Oeste e Sudeste brasileiros. Enquanto a ZCAS carrega umidade para o Centro-Oeste e Sudeste, a ZCIT é associada aos maiores volumes de chuva anual no Norte e Nordeste [Pregorim, 2016]. O sistema ZCAS é caracterizado por uma grande faixa de nebulosidade na direção noroeste - sudeste do Brasil, tendo por área de atuação o centro-sul da Amazônia, centro-sul da Bahia, norte do estado do Paraná, se estendendo até o Oceano Atlântico sudoeste (ver Figura 4.1).

Apesar do grande impacto que este fenômeno causa no clima e na economia da América do Sul, o mesmo ainda não é completamente conhecido e suas origens não estão totalmente explicadas. Em seu trabalho, Kodama 1992 associou características comuns entre a ZCAS, a *zona de convergência do Pacífico Sul* (ZCPS) e a *zona frontal do Baiu* (ZFB), também conhecidas como *zonas de precipitação subtropicais* (ZPS). Tais características podem ser resumidas em:

- Apesar das variações intra-sazonais serem consideráveis, as ZPS's produzem aproximadamente um quantidade de $400mm/ms$ de chuva quando ativas.
- Formam-se ao longo de jatos subtropicais em altos níveis e a leste de cavados semi-estacionários.
- Estendem-se para leste nos subtrópicos, a partir de regiões tropicais específicas com intensa atividade convectiva, apesar de apresentarem diferentes características topográficas em seus arredores.
- São caracterizadas por serem zonas de convergências com espessas camadas de umidade em seu interior e regiões baroclínicas[1] com um jato subtropical em seu limite superior.
- Com relação à porção oceânica das ZPS, todas apresentam taxa de evaporação muito superiores à taxa de precipitação. O transporte de umidade em larga escala através

[1] Em mecânica de fluídos, baroclinia é um estado do fluido em que as linhas isóbaras e as linhas isotermas não coincidem. Em um estado baroclínico, a densidade do fluido depende tanto de sua temperatura quanto de sua pressão. Em contraste, para um fluido barotrópico, sua densidade é função somente da pressão.

de ventos de baixo nível é muito importante para a manutenção de grandes volumes de chuva na região das ZPS's.

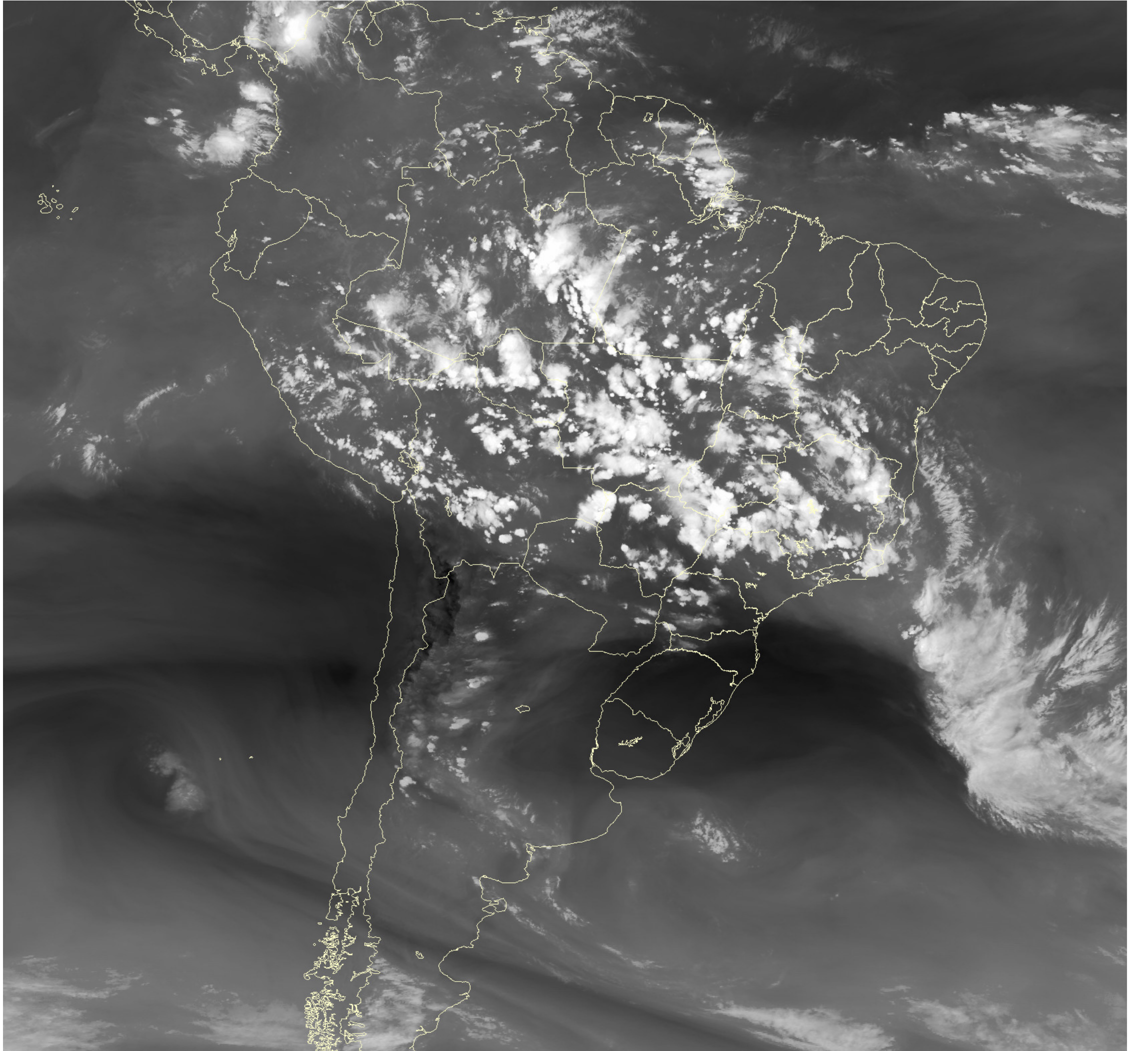


Figura 4.1 – Nesta imagem de satélite é possível observar claramente a formação da ZCAS. Pode-se identificá-la como a faixa de nebulosidade cruzando a superfície do território brasileiro no sentido noroeste-sudeste. Imagens do satélite GOES-12 do dia 27/12/2010, às 18:00 [CPTEC, 2010].

Foi demonstrado que para a formação da ZCAS duas condições principais devem ser satisfeitas. Jatos subtropicais fluindo em altos níveis em latitudes subtropicais e escoamento de ar quente e úmido em direção a altas latitudes [Kodama, 1993]. Outros mecanismos sugerem a influencia de uma interação oceano-atmosfera na região de confluência da corrente das Malvinas com a corrente do Brasil [Nobre, 1988].

Quanto à topografia, alguns estudos sugerem que simulações desconsiderando a mesma conseguem aproximar em algum nível o fenômeno [Gandu e Geisler, 1991]. Entretanto Figueroa 1995 *et al.* demonstraram que a cordilheira dos Andes tem um papel fundamental em seu posicionamento e caracterização. Desta forma, apesar da cordilheira não ter um papel principal na formação da ZCAS, aparentemente sua presença intensifica escoamentos em baixos níveis, auxiliando assim na alimentação da umidade proveniente da bacia amazônica.

4.2 Conjunto de dados *Twentieth Century Reanalysis version 2* (20CRv2)

Produzido pela departamento de ciências físicas do *Earth System Research Laboratory* (ESRL), pertencente a *National Oceanic and Atmospheric Administration* (NOAA), e pelo *Cooperative Institute for Research in Environmental Sciences* (CIRES), da Universidade do Colorado, o 20CRv2 é um esforço entre várias partes para elaboração de um conjunto de dados de reanálise[2] contemplando todo o período dos séculos 20 e 21, bem como boa parte do século 19 (de 1851 até os tempos atuais). Os dados estão disponíveis para grades uniformes com 2° de espaçamento em latitude e longitude e com amostragens temporais de 6 horas, e para grades gaussianas T62 com períodos de amostragem de 3 horas, ambos em formato *grib*. Os parâmetros disponibilizados neste conjunto de dados estão resumidos na Tabela 4.1. [Compo et al., 2015]

Outros conjuntos de dados de reanálise, como por exemplo, o NNRP da *National Centers for Environment Prediction* (NCEP)/*National Center for Atmospheric Research* (NCAR) 1994, o ERA-40 do *European Centre for Medium -Range Weather Forecasts* (ECMWF) 2004, e o ERA-interim também do ECMWF 2009, não se estendem para períodos anteriores à 1948. Tais conjuntos de dados foram criados assimilando-se todas as variáveis observacionais disponíveis. A assimilação de todos os parâmetros observacionais não se mostra adequada para reanálises sobre longos períodos de tempo. Por outro lado, vários estudos demonstram ser possível construir reanálises de períodos mais remotos utilizando somente assimilação de alguns parâmetros de superfície e técnicas avançadas de assimilação. Tais estudos também sugerem que a pressão de superfície seria o parâmetro mais adequado para fazê-lo [Anderson et al., 2005; Compo et al., 2006; Whitaker

[2]Reanálise consiste em uma técnica utilizada na produção de conjunto de dados para análise e pesquisa climática. Nela, dados são gerados por um modelo, o qual assimila dados observacionais em determinados períodos de tempo para torná-lo consistente com a realidade.

et al., 2008]. Por esse motivo, o conjunto 20CRv2 foi construído assimilando-se somente observações de superfície de pressão sinótica [Compo et al., 2011].

Parâmetro	Unidades
Albedo	%
Altura geopotencial	gpm
Altura da camada limite planetária	m
Água precipitável	kg/m^2
Cobertura de neve	%
Cobertura vegetal	%
Convecção	J/kg
Elevação do terreno	m
Espessura de gelo	m
Evaporação	W/m^2
Equivalente de água em neve	kg/m^2
Fluxo de calor	W/m^2
Onda de gravidade	N/m^2
Ozônio	$Dobson$
Pressão ao nível do mar	Pa
Pressão na base de nuvem	Pa
Pressão na superfície	Pa
Pressão no topo de nuvem	Pa
Profundidade de neve	m
Radiação de onda curta	W/m^2
Radiação de onda longa	W/m^2
Quantidade de água na nuvem	kg/m^2
Sublimação	W/m^2
Taxa de precipitação	$kg/m^2.s$
Temperatura da superfície marítima	K
Temperatura do ar	K
Temperatura do ar em alto nível	K
Temperatura máxima	K
Temperatura mínima	K
Temperatura potencial	K
Umidade	%
Umidade do solo	kg/m^2
Uso de terra	%
Velocidade vertical do vento	Pa/s
Ventos de alto nível	m/s
Ventos de superfície	m/s

Tabela 4.1 – Parâmetros disponibilizados pelo conjunto de dados

O modelo utilizado para geração dos dados de previsão foi a versão de abril de 2008 do NCEP *Global Forecast System*, um modelo acoplado de atmosfera-terra. Condições de contorno necessárias para rodar o modelo são especificadas utilizando o conjunto de dados *Met Office HadISST1.1* [Rayner et al., 2003].

O sistema de assimilação de dados utilizado é uma técnica conhecida como filtro de Kalman para conjuntos [Evensen, 1994]. Esta técnica consiste em um método rigoroso para combinar de forma ótima, em termos de mínimos quadrados, observações com estimativas imperfeitas previstas por modelos.

Os dados de pressão assimilados são fornecidos pelo Banco de Dados Internacional de Pressão Superficial - *International Surface Pressure Databank* (ISPD) -, o qual incorpora arquivos internacionais de variáveis meteorológicas. Este banco de dados foi estabelecido a partir de um esforço colaborativo entre programas como *Atmospheric Circulation Reconstructions over the Earth* (ACRE), *Global Climate Observing System* (GCOS) e *World Climate Research Program* (WRCP).

Somando dados de previsão e de análise, o 20CRv2 disponibiliza uma quantidade total de aproximadamente 6TB de dados referentes à atmosfera terrestre.

4.3 Região de interesse

Num primeiro momento, por simplicidade, será estudado o comportamento de variáveis do conjunto 20CRv2 somente em um ponto nas proximidades da UTE-LORM, colaboradora do projeto situada no estado do Espírito Santo. A UTE-LORM se encontra a 4km do Oceano Atlântico, 500m da lagoa do Piába, 2,2km do Rio Ibiriba e 4km do Rio Doce. Esta usina é fortemente afetada pela ZCAS, tendo muitas vezes seus arredores alagados pelas cheias dos cursos de água que a cercam. Outro fator que afeta indiretamente a operação da mesma é o nível dos reservatórios de água das hidrelétricas, que tem prioridade de venda de energia para a rede elétrica.

Devido ao espaçamento da grade espacial, escolheu-se um ponto de latitude $20S$ e longitude $40O$ do conjunto 20CRv2. Este ponto também apresenta o conveniente de se situar próximo à estação meteorológica 83648 - *18°99'S 40°99'O* -[INMET, 2017] de Vitória-ES, conforme pode-se observar na Figura 4.2. Assim, sera possível fazer a validação dos dados de reanálise 20CRv2 com os dados da estação.

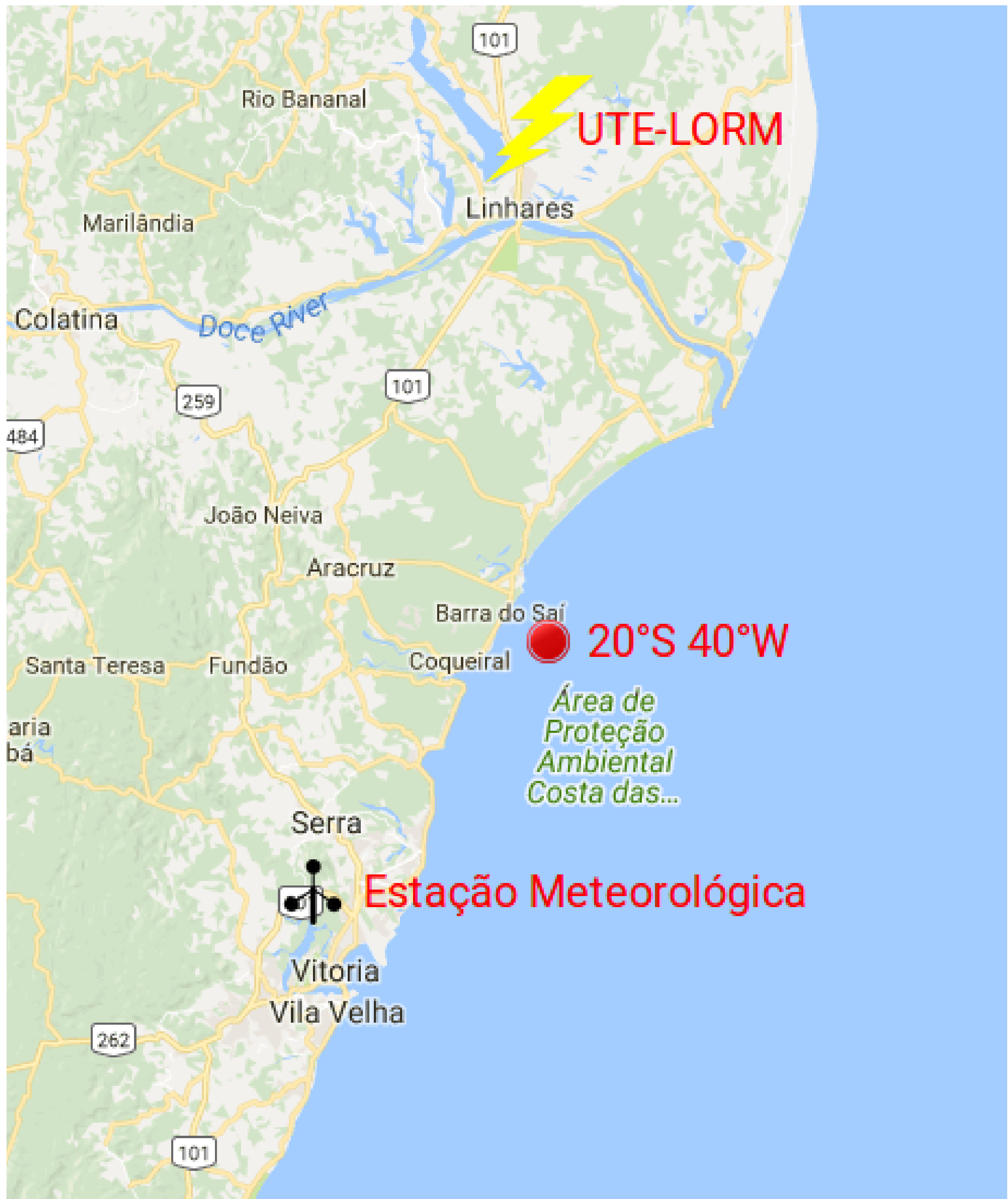


Figura 4.2 – Localização da UTE-LORM e dos pontos de análise. Imagem retirada de *Google Maps* 2017

4.4 Análises estatísticas

Para uma primeira tentativa de redução do conjunto de dados, se pretende fazer uma descrição estatística de variáveis tidas como essenciais à descrição do fenômeno da ZCAS na localidade descrita na Seção 4.3. Para tal, serão aplicado os 4 primeiros momentos estatísticos aos parâmetros escolhidos em diferentes escalas de tempo. Desta forma, pode-se averiguar as informações que podem ser extraídas e àquelas que são perdidas ao se variar o período de análise.

Em matemática, o momento de uma medida específica é uma maneira de caracterizar estatisticamente a forma de um determinado conjunto de pontos. Para uma variável aleatória discreta X, o n-ésimo momento estatístico μ_n é calculado aplicando-se o operador esperança $E[X]$ à potência n da variável, conforme a Equação 4.1.

$$\mu_n = E[X^n] = \frac{1}{N}\sum_{k=1}^{N} x^n \tag{4.1}$$

Já o n-ésimo momento central da variável é calculado por:

$$\mu_n = E[(X - E[X])^n] = \frac{1}{N}\sum_{k=1}^{N}(x - E[X])^n \tag{4.2}$$

O primeiro momento e segundo momento central de uma determinada variável aleatória são bastante conhecidos pelos seus nomes populares, média e variância (Equação 4.3 e Equação 4.4 respectivamente), e sua interpretação é bastante simples. Enquanto a média passa uma ideia da tendência central da distribuição, a variância está relacionada com sua dispersão. A raiz da variância, mais comumente utilizada, é chamada de desvio padrão $\sigma = \sqrt{\sigma^2}$.

$$\mu = \frac{1}{N}\sum_{k=1}^{N} x \tag{4.3}$$

$$\sigma^2 = \frac{1}{N}\sum_{k=1}^{N}(x - \mu)^2 \tag{4.4}$$

O terceiro momento central, quando normalizado pelo desvio padrão, é conhecido como *skewness* (γ) - em português, obliquidade - e sua fórmula é dada pela Equação 4.5. Seu valor está relacionado com a assimetria, distribuições com valores positivos de *skewness* apresentam uma "cauda" mais comprida à direita, enquanto distribuições com valores negativos de *skewness* apresentam uma "cauda" que se estende à esquerda, conforme pode ser visto na Figura 4.3.

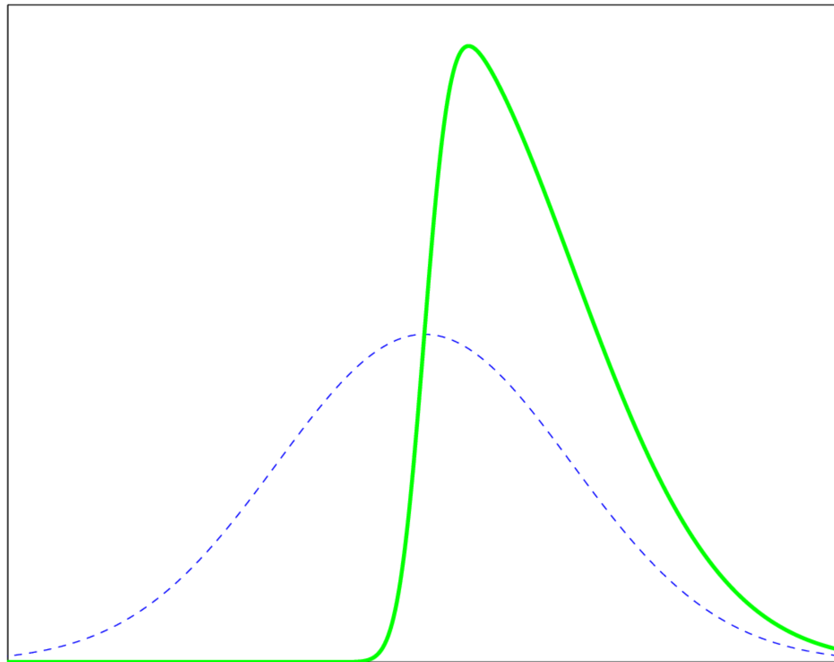
$$\gamma = E[(\frac{X-\mu}{\sigma})^3] = \frac{\mu_3}{\sigma^3} = \frac{\frac{1}{N}\sum_{k=1}^{N}(x-\mu)^3}{\sigma^3} \tag{4.5}$$

Por fim, o quarto momento central normalizado, também chamado de curtose κ, é dado pela Equação 4.6. A interpretação de κ está ligada à quão "extrema" é a cauda da distribuição, ou seja, à capacidade da distribuição de produzir valores extremos. Não é muito difícil de visualizar esta forma de interpretação se considerarmos que a curtose é o valor esperado dos dados normalizados na 4ª potência. Assim, valores normalizados menores do que 1 contribuem com praticamente nada para κ.

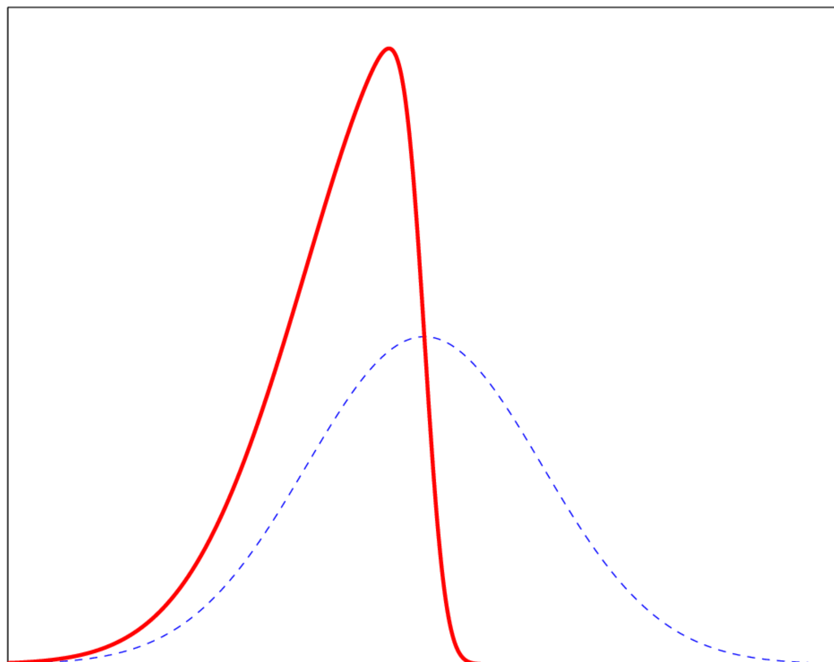
$$\kappa = E[(\frac{X-\mu}{\sigma})^4] = \frac{\mu_4}{\sigma^4} = \frac{\frac{1}{N}\sum_{k=1}^{N}(x-\mu)^4}{\sigma^4} \tag{4.6}$$

Uma interpretação alternativa da curtose é dada por Moor 1986. Considerando que uma variável aleatória Z, dada pela normalização da variável X, conforme Equação 4.7.

$$Z = \frac{X-\mu}{\sigma} \tag{4.7}$$



(a) *skewness* > 0



(b) *skewness* < 0

Figura 4.3 – Distribuições com $\gamma > 0$ e com $\gamma < 0$ respectivamente. Em ambas, a curva em azul pontilhado representa o valor de $\gamma = 0$

Então:

$$\kappa = E[Z^4] = var(Z^2) + E[Z^2]^2 = var(Z^2) + 1 \quad (4.8)$$

Assim, a curtose (κ) pode ser vista como a medida de variância de Z^2 ($var(Z^2)$)em torno de sua esperança.

Para a distribuição normal, $\kappa = 3$. Para os casos em que $\kappa = 3$, chama-se distribuição de mesocúrtica. Distribuições com $\kappa > 3$ são chamadas de leptocúrticas, ou seja, tem caudas mais achatadas do que a distribuição normal. Quando $\kappa < 3$, a distribuição tem caudas mais espessas do que a normal, sendo chamada de platicúrticas. Na Figura 4.4 pode-se ver o efeito da variação de κ em uma família de funções leptocúrticas.

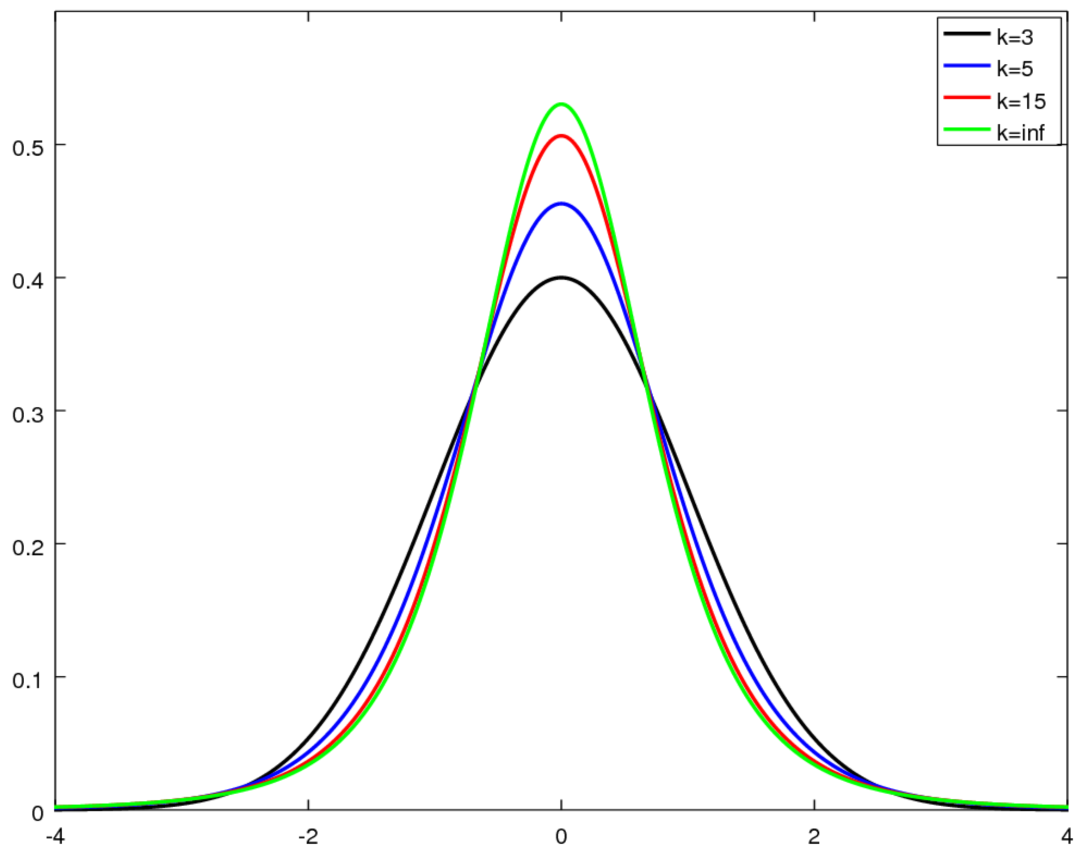


Figura 4.4 – Dependência da forma da distribuição ao variar-se κ.

5 Resultados e discussões

Neste capítulo serão apresentados os aspectos principais do algoritmo usado para implementar a aplicação discutida no Capítulo 4, ilustrando o uso da mesma por um usuário que pretenda desenvolver seu próprio *plugin*. Também serão debatidos os resultados que tal aplicação gera. Os trechos de código que serão apresentados estão em inglês (língua "universal" para a escrita de programas) com comentários em português[1].

5.1 Principais aspectos da implementação de um *plugin*

Para escrever-se um *plugin* personalizado, deve-se criar uma biblioteca dinâmica[2] com as definições das funções que se deseja disponibilizar para o usuário da plataforma. Depois de compilada, a biblioteca deve ser salva no diretório apropriado para poder ser carregada pelo gerenciador de *plugins* (Seção 3.8). Para a aplicação proposta, criou-se a seguinte biblioteca.

```
#include "api/pluginAPI.h"
#include "plugins/StatisticMoments.hpp"

CDA_int CDA_exit_plugin(){
    return 0;
}

PLUGIN_API CDA_exit_function
CDA_init_plugin(const CDA_platform_services * parameters){

    /*
    Registro da função statisticMoments no gerenciador de plugins
    */
    cda::api::FunctionTemplate<cda::plugins::StatisticMoments>
```

[1] Em *C++*, comentários são partes do código dentro dos marcadores /* e */. Trechos comentados servem somente como informação para o leitor e são descartados pelo compilador

[2] Bibliotecas dinâmicas, diferente de bibliotecas estáticas, que são incorporadas ao programa em tempo de compilação, são uma coleção de funções que podem ser utilizadas por vários programas simultaneamente, que as carregam em tempo de execução. Em *C++*, utilizando o compilador *GNU compiler*, bibliotecas dinâmicas devem ser compiladas com a *flag -fPIC* e *linkadas* com *-shared*.

```
            ::registerObject("statisticMoments",parameters);

    return &CDA_exit_plugin;
}
```

Os métodos *CDA_init_plugin* e *CDA_exit_plugin* estão definidos no arquivo cabeçalho *"api/pluginAPI.h"*. O primeiro é responsável pelo registro das funções criadas para a biblioteca e é chamado durante a inicialização do programa. Neste caso, a única função registrada é a *statisticMoments*, definida no arquivo cabeçalho *"plugins/StatisticMoments.hpp"*. O segundo é chamado quando o programa é fechado e deseja-se encerrar a biblioteca. É utilizado para fazer a limpeza de variáveis alocadas dinamicamente.

Na criação de uma função, o usuário/desenvolvedor deve declarar uma classe que herde a classe *template*[3] *FunctionTemplate<>* (declarada no arquivo cabeçalho "api/-FunctionTemplate.hpp") a qual definirá a interface com os métodos que devem ser implementados. A classe *FunctionTemplate<>* também define métodos como *registerObject*, utilizado para fazer o registro da função no gerenciador de *plugins*. Pode-se observar a declaração da função *statisticMoments* no trecho de código abaixo:

```
#ifndef PLUGINS_STATISTICMOMENTS_HPP
#define PLUGINS_STATISTICMOMENTS_HPP

#include "api/FunctionTemplate.hpp"
#include <mutex>

namespace cda{
 namespace plugins{

   class StatisticMoments:
    public api::FunctionTemplate<StatisticMoments>
   {
   public:
```

[3]Em *C++*, um *template* de uma classe/método refere-se à um trecho de código que pode operar com tipos genéricos. Assim, torna-se possível, por exemplo, definir um método soma que pode aceitar tanto tipo integral como tipo ponto flutuante como argumento, evitando a escrita de código redundante. O tipo para qual se deseja especializar determinado método ao chamá-lo é passado como argumento do template, através da sintaxe "*valorDeRetorno metodoTemplate<ArgumentoTemplate>(ArgumentoMetodo)*".

```
/*
Método chamado pela plataforma antes do início da execução
das tarefas da função para seu registro e configurações.
*/
void initialize();
/*
Método chamado depois da execução da função.
Implementação opcional.
*/
void terminate();
/*
Tarefa a ser executada em paralelo.
*/
class statisticMomentsParallelTask:
 public ParallelTaskTemplate
{
 public:
  /*
  Método chamado pela plataforma para execução da tarefa.
  /*
  void run();

 private:
  /*
  Declaração das variáveis utilizadas na execução da tarefa.
  */
  CDA_char           variableName[128];
  types::Inumeric * timeDimension1     = nullptr;
  CDA_numeric     * timeDim1Values     = nullptr;
  types::Inumeric * timeDimension2     = nullptr;
  CDA_numeric     * timeDim2Values     = nullptr;
  types::Inumeric * iVariable          = nullptr;
```

```
      CDA_numeric        * variableValues    = nullptr;
      types::Inumeric   * iMean             = nullptr;
      CDA_numeric        * meanValues        = nullptr;
      types::Inumeric   * iStdDev           = nullptr;
      CDA_numeric        * stdDevValues      = nullptr;
      types::Inumeric   * iSkew             = nullptr;
      CDA_numeric        * skewValues        = nullptr;
      types::Inumeric   * iKurt             = nullptr;
      CDA_numeric        * kurtValues        = nullptr;
      CDA_int              timeDim2length    = 1;
      CDA_int              timeStep          = 0;
      CDA_numeric          lat               = 0;
      CDA_numeric          lon               = 0;
      CDA_numeric          timeMean          = 0;
      CDA_numeric          mean              = 0;
      /*
       Métodos auxiliares utilizados para execução da tarefa.
      */
      void          initializeParameters();
      void          calculateMean();
      void          calculateStdDev();
      void          calculateSkewness();
      void          calculateKurtosis();
      void          saveToFile();
     };
   };
  }
}
#endif
```

Cada função é composta por uma ou mais tarefas, podendo estas serem de execução serial ou dinâmica. Tarefas são definidas como subclasses de uma dada função, sendo que as seriais devem herdar a classe *template TaskTemplate*<> e as dinâmicas devem herdar

a classe *template ParallelTaskTemplate<>*. No exemplo da função acima, temos somente uma tarefa paralela declarada: *statisticMomentsParallelTask*. O desenvolvedor da função deve implementar obrigatoriamente o método *run*, que é chamado pelo gerenciador de tarefas quando chega o momento de sua execução.

O método *initialize* da função é chamada logo antes da execução de suas tarefas. Nele, são registrados as tarefas que a compõe, na ordem em que devem ser executados. Para fazer o registro, o desenvolvedor deve utilizar o método *template addTask*, usando a classe tarefa desejada como argumento para o *template* (trecho de código abaixo). O método *terminate* tem implementação opcional e é chamado após a execução da função. O mesmo pode ser utilizado para impressão de mensagens na tela com informações sobre o resultado da execução.

```
void StatisticMoments::initialize(){
    /*
    Registro da tarefa paralela
    statisticMomentsParallelTask
    */
    addTask<statisticMomentsParallelTask>();
}
```

Finalmente, a parte de execução da tarefa é definida a seguir:

```
void StatisticMoments::TimeParallel::run(){
    /*
    Inicialização dos parâmetros passados para a
    função quando a mesma é chamada pelo usuário
    */
    initializeParameters();
    /*
    Cálculo dos momentos estatísticos
    */
    calculateMean();
    calculateStdDev();
    calculateSkewness();
    calculateKurtosis();
```

```
            /*
            Salva os resultados em um arquivo
            /*
            saveToFile();
    }
```

O método *initializeparameters*, como mencionado antes, é responsável pela inicialização dos parâmetros. Como se pode observar no código abaixo, a obtenção dos parâmetros da função passados pelo usuário é realizada através do método *template getParameter<>*. O argumento de *template* passado é uma interface para o tipo de variável que se deseja obter (*Istring*, *Iinteger*, *Inumeric*, *Ipoint*,*Iregion*). Já o índice usado como argumento para o método refere-se a ordem do parâmetro passado. No exemplo utilizado, o primeiro parâmetro é a identificação da variável do ponto sobre a qual serão calculados os momentos estatísticos. O segundo parâmetro é a escala de tempo, em dias, que será utilizada para o cálculo das médias.

Por ser uma tarefa paralela, o desenvolvedor tem acesso ao membro protegido *_point* (definido na classe *template ParallelTaskTemplate*), através do qual ele pode acessar os dados do ponto sobre o qual a tarefa está sendo executada. As variáveis *timeDimension1* e *timeDimension2* referem-se à escala de tempo da variável de origem e à escala de tempo nas quais serão calculados os momentos estatísticos. As variáveis *iMean, istdDev, iSkew e iKurt* referem-se aos momentos estatísticos que serão calculados e estão associadas à escala de tempo *timeDimension2*.

```
void StatisticMoments::ParallelTaskTemplate
                          ::initializeParameters(){
 /*
 Obtendo parâmetro 1. Variável do ponto espacial sobre
 a qual vão ser aplicados os momentos estatísticos
 */
 if(getParameter<types::Istring*>(1))
  strcpy(variableName,
          getParameter<types::Istring*>(1)->getValue());
 else
  throw EXCEPTION();
```

```
iVariable=_point->getVariablesContainer()
                    ->getVariable(variableName);
if(iVariable==nullptr)
 throw EXCEPTION("variable doesn't exist");


/*
Obtendo parâmetro 2. Escala de tempo na qual serão aplicados
os momentos estatísticos
*/
if(getParameter<types::Istring*>(2)
 timeStep=4**getParameter<types::Iinteger*>(1)->getValue();
else
 timeStep=1;
/*
Obtendo dimensão temporal da variável de origem
*/
timeDimension1=_point->getTimeDimensionsContainer()
                          ->getVariable("4/day");
/*
Alocando espaço para a nova dimensão temporal
*/
timeDim2name   =std::to_string(timeStep)
                   +"["+timeDimension1->getName()+"]";
timeDimension2=_point->getTimeDimensionsContainer()
                          ->registerVariable(timeDim2name.c_str());
timeDimension2->setLength(timeDim2length);
/*
Alocando espaço para as novas variáveis associadas aos
momentos estatísticos
*/
statisticName=variableName+"_"+std::string{"mean_"}+"_"
```

```
                   +timeDim2name.c_str();
 iMean            =_point->getVariablesContainer()
                           ->registerVariable(statisticName.c_str());
 iMean->setTimeDim(timeDimension2);
 statisticName=variableName+"_"+std::string{"stdDev"}+"_"
                   +timeDim2name.c_str();
 StdDev         =_point->getVariablesContainer()
                        ->registerVariable(statisticName.c_str());
 iStdDev->setTimeDim(timeDimension2);
 statisticName=variableName+"_"+std::string{"skewness"}+"_"
                   +timeDim2name.c_str();
 iSkew          =_point->getVariablesContainer()
                        ->registerVariable(statisticName.c_str());
 iSkew->setTimeDim(timeDimension2);
 statisticName=variableName+"_"+std::string{"Kurtosis"}+"_"
                   +timeDim2name.c_str();
 iKurt          =_point->getVariablesContainer()
                        ->registerVariable(statisticName.c_str());
 iKurt->setTimeDim(timeDimension2);
}
```

Para o cálculo da média temporal, foi utilizado o método auxiliar *calculateMean*, conforme abaixo:

```
void StatisticMoments::TimeParallel::calculateMean(){
 /*
 Reiniciando todas as variáveis para o primeiro valor
 associado de tempo
 */
 i  =0;
 meanPtr      =iMean->getValue();
 variablePtr=iVariable->getValue();
 timeDim1ptr=timeDimension1->getValue();
 timeDim2ptr=timeDimension2->getValue();
```

```
/*
Iterando entre todos os valores da primeira dimensão de tempo
*/
while(i++<timeDimension1->getLength()){
 timeMean+=*timeDim1ptr++;
 mean     +=*variablePtr++;
 /*
 Quando o contador for igual a escala temporal escolhida,
 incrementa ponteiros associados a segunda dimensão
 temporal (associada ao cálculo dos momentos estatísticos)
 */
 if(i%timeStep==0){
  /*
  Cálculo da media
  */
 *timeDim2ptr++=timeMean/timeStep;
 *meanPtr++     =mean/timeStep;
  timeMean       =0;
  mean           =0;
 }
}
}
```

O cálculo é bastante simples. Trata-se basicamente de uma iteração entre todos os pontos da primeira dimensão temporal. A variável auxiliar *mean* é utilizada para acumular os valores da variável da qual se está calculando a média. Quando o contador *i* atinge um múltiplo da escala de tempo escolhida, divide-se a variável *mean* pelo número de pontos iterados para obter-se a média. Assim, os pontos da nova variável média criada vão sendo calculados um a um.

Similarmente, os 3 outros momentos estatísticos são calculados através dos seguintes métodos:

```
/*
Cálculo do desvio padrão
*/
void StatisticMoments::TimeParallel::calculateStdDev(){
 mean         =0;
 meanPtr      =iMean->getValue();
 stdDevPtr    =iStdDev->getValue();
 variablePtr=iVariable->getValue();
 while( i++<timeDimension1->getLength()){
  mean       +=pow(*variablePtr++ - *meanPtr,2);
   if(i%timeStep==0){
    meanPtr++;
    *stdDevPtr++=pow(mean/timeStep,0.5);
    mean         =0;
   }
 }
}

/*
Cálculo da assimetria
*/
void StatisticMoments::TimeParallel::calculateSkewness(){
 i=0;
 mean         =0;
 meanPtr      =iMean->getValue();
 stdDevPtr    =iStdDev->getValue();
 variablePtr=iVariable->getValue();
 skewPtr      =iSkew->getValue();
 timeDim1ptr=timeDimension1->getValue();
 timeDim2ptr=timeDimension2->getValue();
 while(i++<timeDimension1->getLength()){
  mean+=pow(*variablePtr++-*meanPtr,3);
```

```
    if(i%timeStep==0){
      meanPtr++;
      *skewPtr++=mean/timeStep/pow(*stdDevPtr++,3);
      mean          =0;
    }
  }
}

/*
Cálculo da curtose
*/
void StatisticMoments::TimeParallel::calculateKurtosis(){
  i=0;
  mean          =0;
  meanPtr       =iMean->getValue();
  stdDevPtr     =iStdDev->getValue();
  variablePtr=iVariable->getValue();
  kurtPtr       =iKurt->getValue();
  timeDim1ptr=timeDimension1->getValue();
  timeDim2ptr=timeDimension2->getValue();
  while(i++<timeDimension1->getLength()){
   mean+=pow(*variablePtr++-*meanPtr,4);
    if(i%timeStep==0){
      meanPtr++;
      *kurtPtr++=mean/timeStep/pow(*stdDevPtr++,4);
      mean          =0;
    }
  }
}
```

Uma vez implementado o *plugin*, a biblioteca pode ser compilada com o seguinte comando (para o compilador *GNU compiler*, usando sistema operacional da família Linux):

```
g++ -std=c++11 -pthread -fPIC -shared -o sampleLibrary.so \\
                  sampleLibrary.cpp statisticMoments.cpp
```

Assim, tem-se um *plugin* pronto para ser utilizado por um usuário terceiro. Na Figura 5.1 pode-se observar uma imagem da função *statisticMoments* sendo chamada em uma situação de uso da plataforma. No lado esquerdo da imagem, fica a janela de comandos, onde o usuário chama as funções desejadas. Ao lado direito, a janela de resultados que, como o nome sugere, mostra os resultados das funções na ordem em que foram chamadas.

```
1>>regiao=load("pgrbanl_mean_*_PRES_sfc.grib",2008
,2009)
2>>print(regiao)
3>>statisticMoments("SurfacePressure",5)[regiao]
4>>print(regiao)
5>>
```

```
Loading files:
        -> pgrbanl_mean_2008_PRES_sfc.grib
        -> pgrbanl_mean_2009_PRES_sfc.grib

 regiao
 |
 -->temporal dimensions
 |       -4/day  [2924]
 -->spatial dimensions
 |       -lat  [180]
 |       -lon  [91]
 -->variables
         -SurfacePressure  [4/day]

Statistic moments
        -variable   = SurfacePressure
        -time scale = 5

 regiao
 |
 -->temporal dimensions
 |       -4/day  [2924]
 |       -5day(s)  [146]
 -->spatial dimensions
 |       -lat  [180]
 |       -lon  [91]
 -->variables
         -SurfacePressure  [4/day]
         -SurfacePressure_kurtosis_5day(s)  [5day(s)]
         -SurfacePressure_mean_5day(s)  [5day(s)]
         -SurfacePressure_skewness_5day(s)  [5day(s)]
         -SurfacePressure_stdDev_5day(s)  [5day(s)]
```

Figura 5.1 – Utilização de um *plugin* criado.

Primeiramente, é necessário carregar um arquivo com dados climáticos através da função *load* (função pertencente à biblioteca padrão). No exemplo, carregou-se dois arquivos, com dados referentes aos anos de 2008 e 2009, na variável "*regiao*". Ao imprimir

a variável "*regiao*" na tela, podemos ver que ela possui uma sub-variável "SurfacePressure" (referente à pressão superficial), duas dimensões espaciais, "lat" e "lon", e uma dimensão temporal.

Após aplicar-se a função "*statisticMoments*" para uma escala temporal de 5 dias, podemos notar que ao imprimir a variável na tela, aparecem mais 1 dimensão temporal e 4 sub-variáveis referentes aos momentos estatísticos calculados.

5.2 Dados gerados com a aplicação

Para validar os cálculos realizados pelo *plugin* desenvolvido, os dados do projeto 20CRv2, lidos com auxílio da plataforma, serão comparados com dados da estação meteorológica de vitória, conforme proposto na Seção 4.3. Na Figura 5.2, são comparados dados de média diária de pressão entre os dias 01/12/2008 até 31/01/2009, período de intensa atividade de ZCAS.

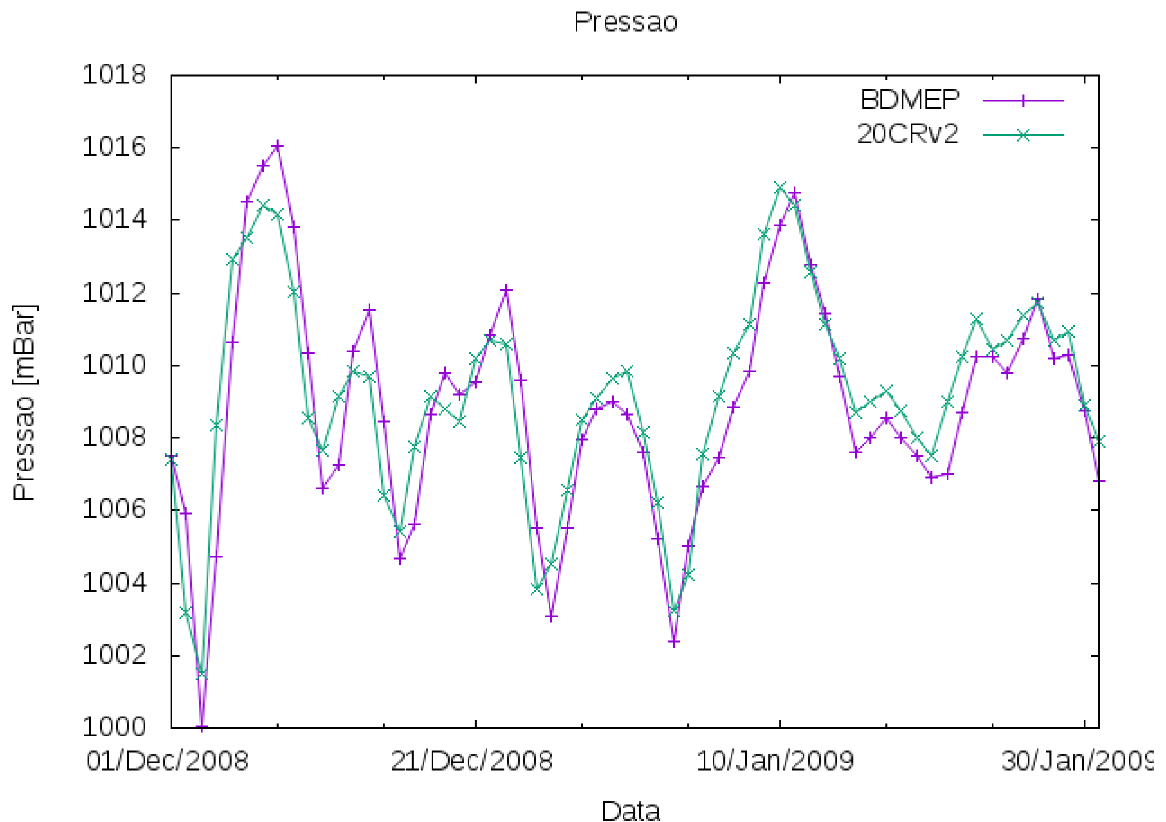


Figura 5.2 – Comparação dos dados de pressão da estação meteorológica de Vitória-ES com dados do conjunto 20CRv2.

Pode-se observar que a pressão média diária calculada pelo modelo segue quase que perfeitamente a mesma pressão lida na estação meteorológica. Assim, conclui-se que, a plataforma utilizada consegue ler os dados adequadamente, o *plugin* desenvolvido calcula a média diária corretamente e, por último, os dados de pressão do conjunto de reanálise do projeto 20CRv2 são bastante precisos nesta região. Esta última afirmação não é surpreendente, uma vez que tais dados assimilam justamente dados de pressão. Para a temperatura, já não é possível tirar a mesma conclusão. Pode-se ver na Figura 5.3 que a temperatura, apesar de seguir mais ou menos a mesma tendência, apresenta desvios consideráveis quando comparados com os dados da estação.

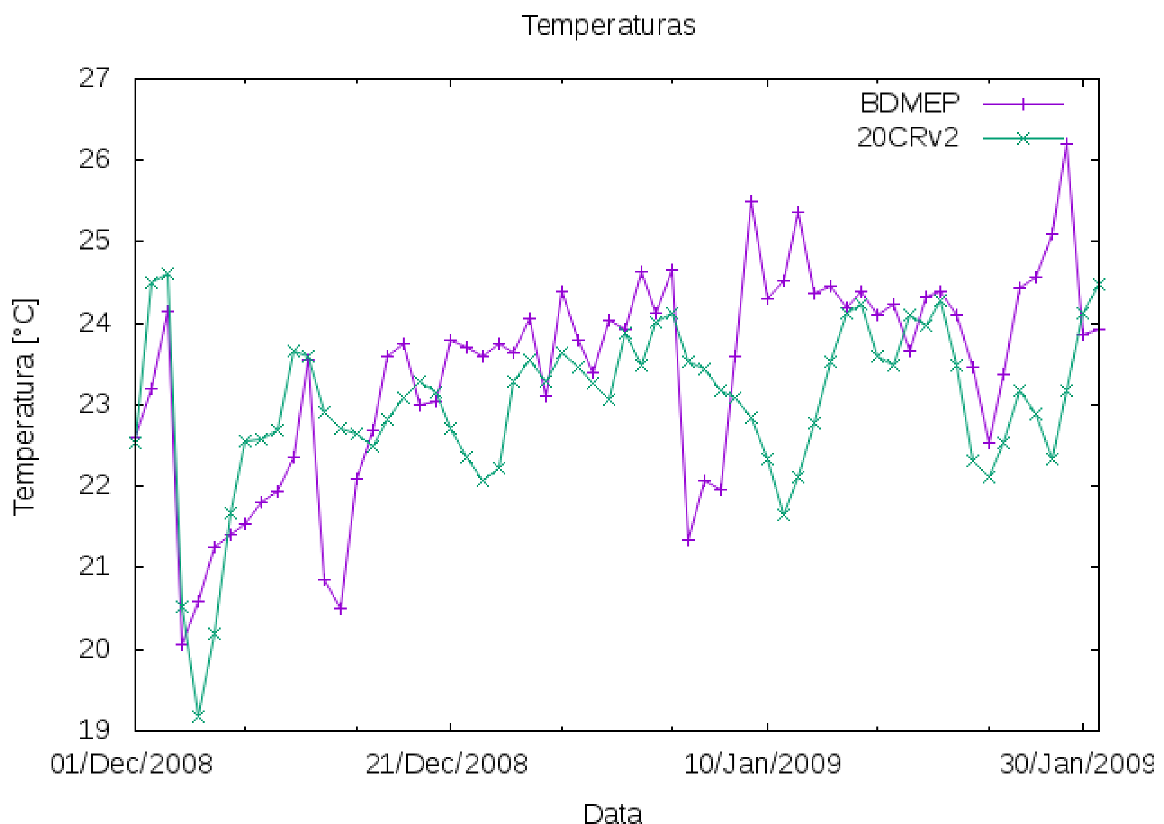


Figura 5.3 – Comparação dos dados de temperatura da estação meteorológica de Vitória-ES com dados do conjunto 20CRv2.

Finalmente, na Figura 5.4, a comparação dos dados de umidade relativa. Estes sim, são bastante distorcidos. A explicação para tal esta no fato desta variável não ser sido utilizada na assimilação do conjunto de dados e os modelos existentes não conseguirem predizê-la com confiabilidade. Assim, apesar da umidade relativa ser um parâmetro muito

importante para descrever a ZCAS, o conjunto de dados 20CRv2 não apresenta valores condizentes com a realidade para torná-la objeto do estudo, sugerindo a utilização de outras variáveis para caraterização do fenômeno.

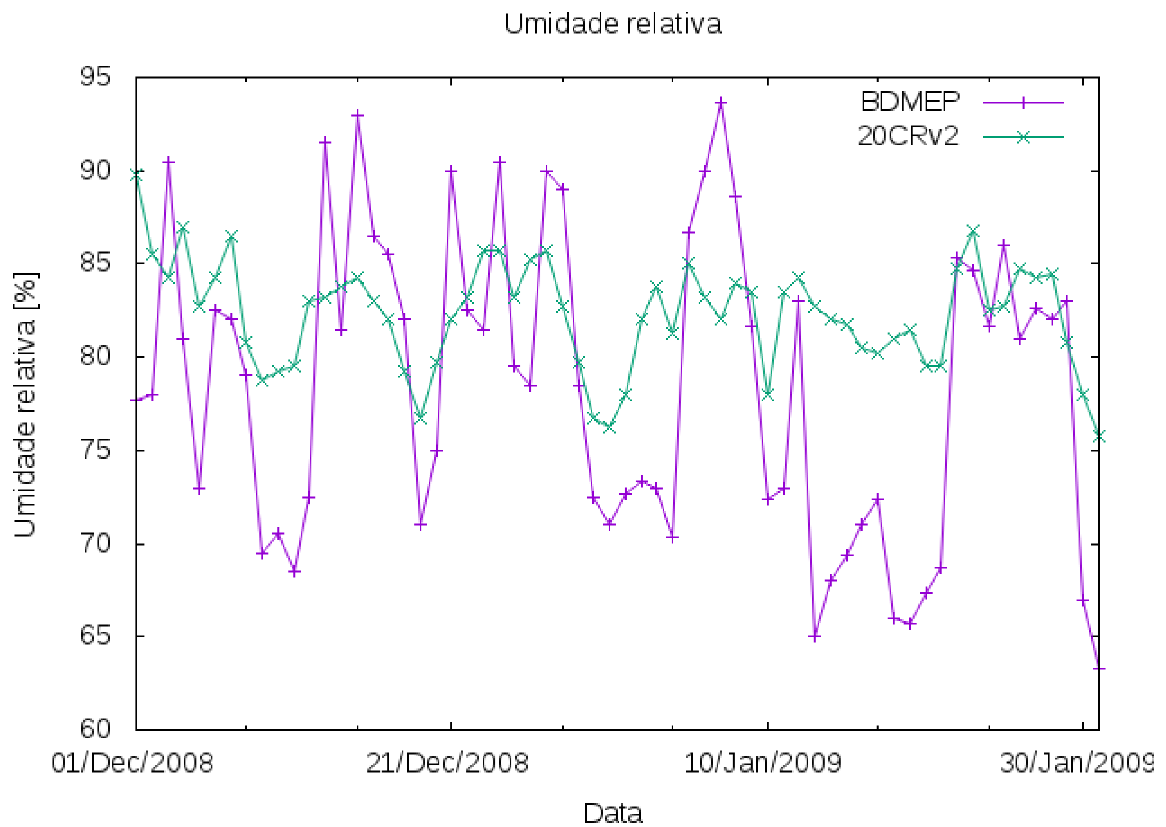


Figura 5.4 – Comparação dos dados de umidade relativa da estação meteorológica de Vitória-ES com dados do conjunto 20CRv2.

Nas Figuras 5.5-5.8 pode-se observar como os momentos estatísticos são influenciados pela variação da escala temporal no período de 01/10/2008-28/02/2009. Podemos observar que a média tem uma clara tendência de se achatar conforme se aumenta a escala de tempo, como era de se esperar.

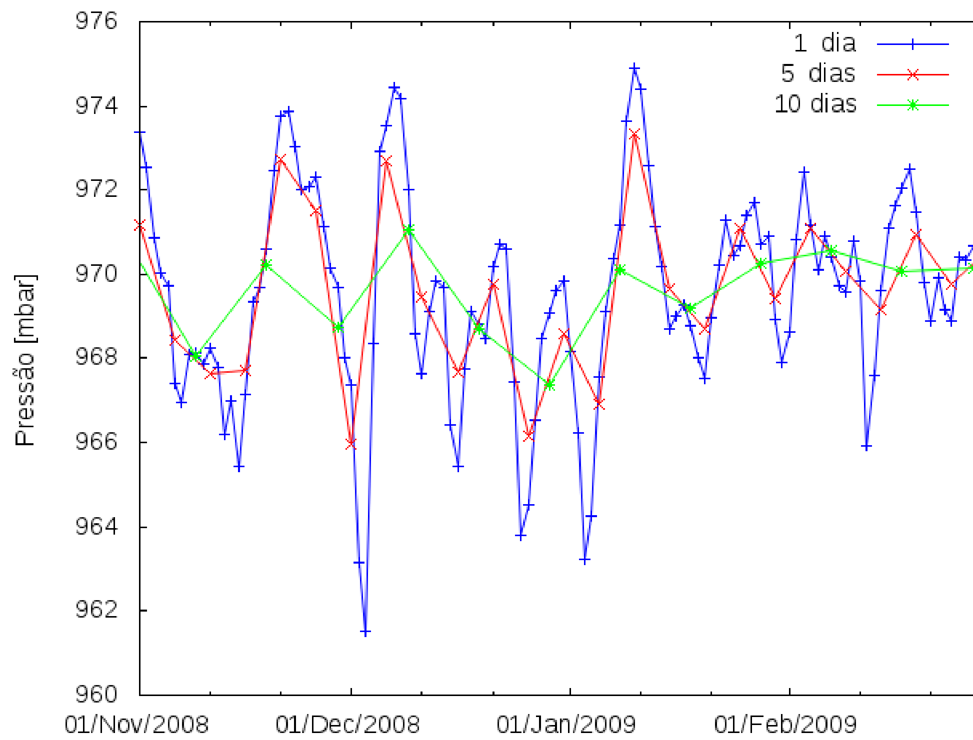


Figura 5.5 – Influência da variação de escala na média da pressão.

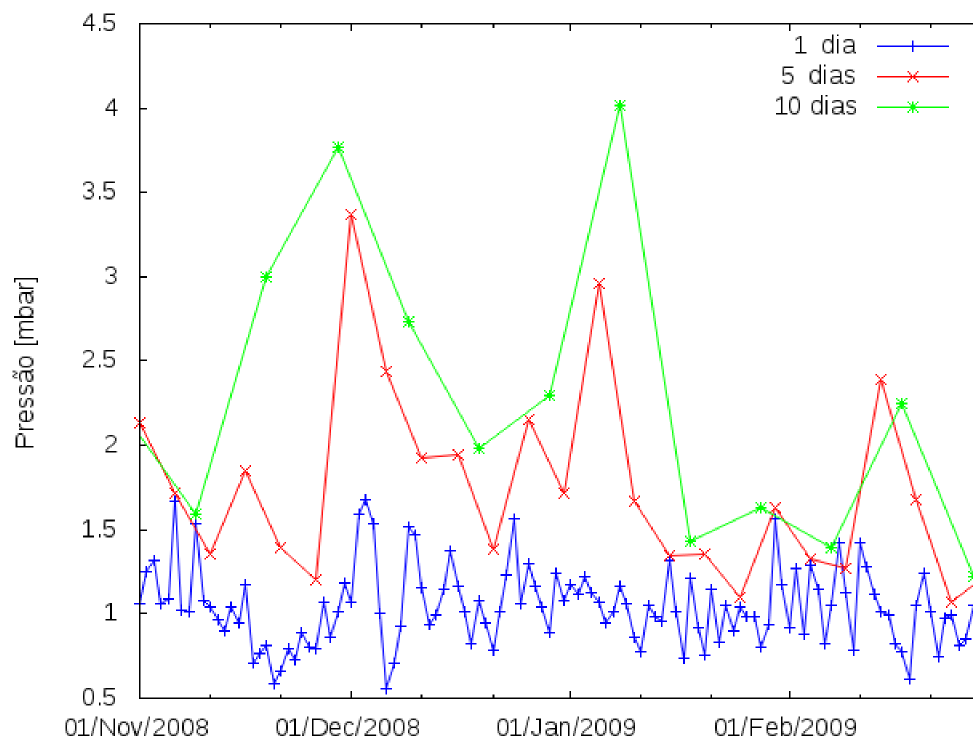


Figura 5.6 – Influência da variação de escala no desvio padrão da pressão.

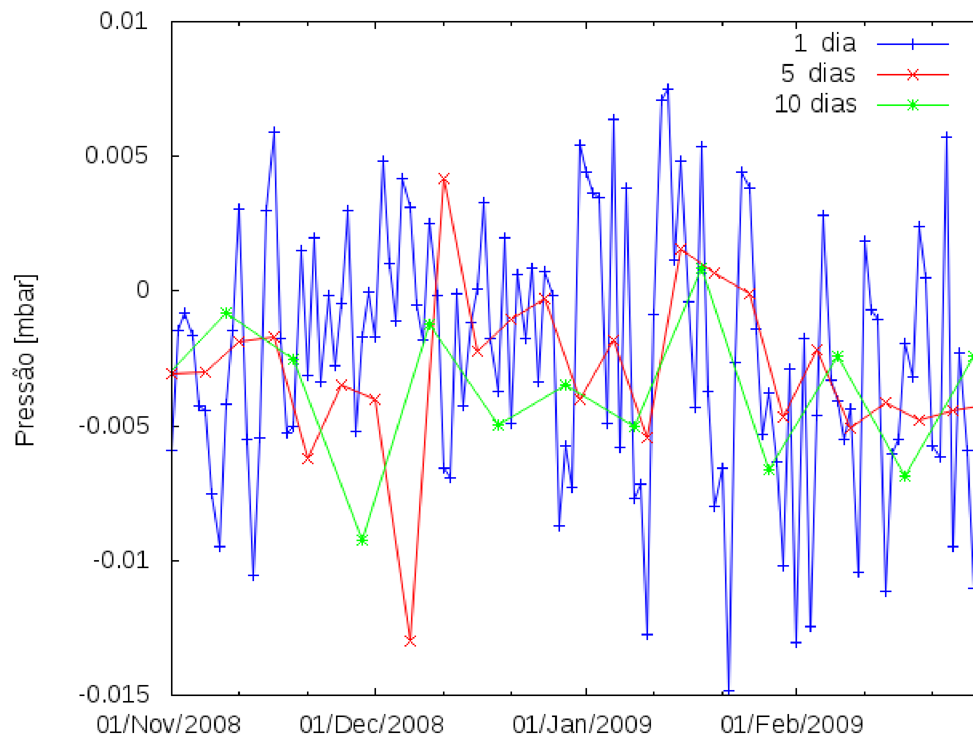


Figura 5.7 – Influência da variação de escala na assimetria da pressão.

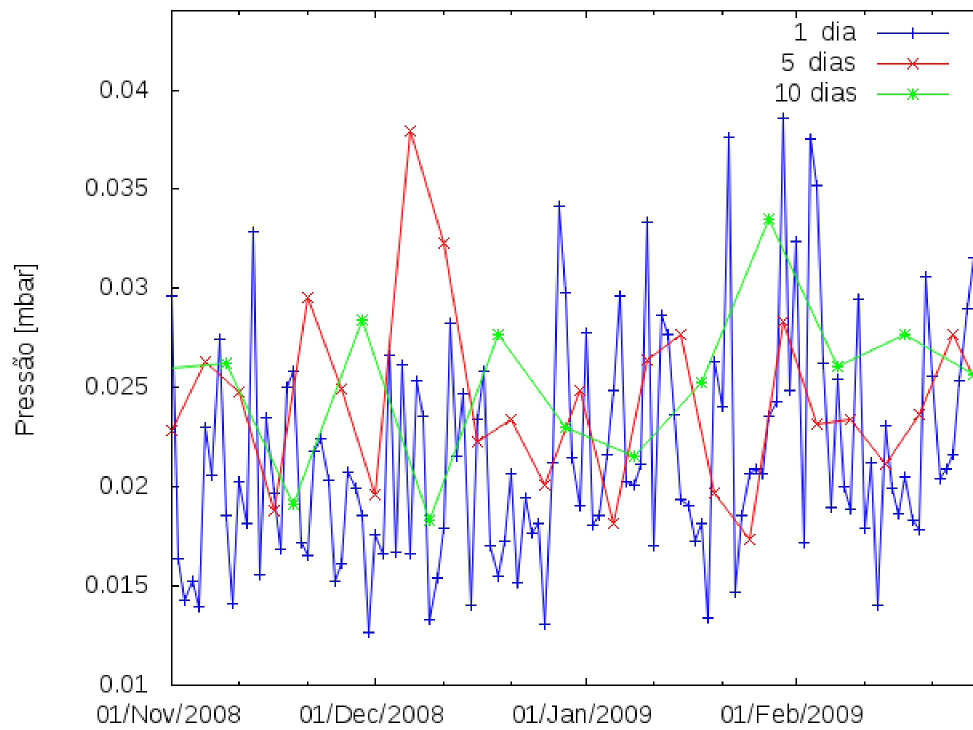


Figura 5.8 – Influência da variação de escala na curtose da pressão.

Esta análise, apesar de não ser suficiente, é um primeiro passo na busca da caracterização da ZCAS. A descrição estatística de um conjunto de dados é muito importante para conhecer os parâmetros de maior relevância e, na medida do possível reduzi-lo, poupando recursos computacionais. Para ter alguma relevância, tal análise deve ser estendida para toda a região de interesse e para todas as variáveis disponíveis no conjunto de dados.

5.3 Escalabilidade

Quando se desenvolve um programa paralelizado, deseja-se reduzir o tempo gasto na execução de um determinado problema ao distribuí-lo entre várias unidades de processamento. Muitas vezes, apesar da lei de Amdahl [Amdahl, 1967] prever que o ganho em tempo é proporcional a fração paralelizável do problema e ao número de máquinas disponíveis, o programa não apresenta a melhora esperada devido a um código mal escrito que cria gargalos em seu andamento. A escalabilidade de um programa é a forma que o mesmo se comporta conforme alteramos o número de processadores disponíveis para a sua execução. Uma maneira simples de medi-la, é comparando o tempo que determinado problema é executado de maneira serial com o tempo gasto quando o mesmo é executado de maneira paralela.

Para fazer a análise de escalabilidade da plataforma objeto de estudo deste trabalho, foram comparados os tempos gastos para o cálculo dos momentos estatísticos da pressão com um, 5, 10, 15 e 20 processadores. Na Figura 5.9, estão resumidos os ganhos de velocidade obtidos com o aumento do número de processadores.

Este resultado mostra claramente que, apesar da plataforma não escalabilizar perfeitamente o problema, os ganhos em velocidade devido a paralelização do programa são notáveis. Com 20 processadores, o problema em questão leva aproximadamente 14 vezes menos tempo do que quando tratado de forma serial, o que é uma melhora satisfatória. É importante salientar que esta versão da plataforma ainda esta em fase de testes e melhorias em seu algoritmo são possíveis para melhorar sua escalabilidade.

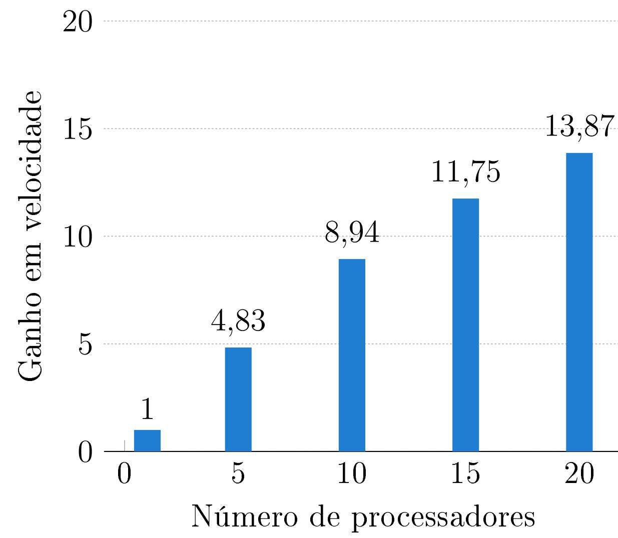


Figura 5.9 – Ganho em velocidade obtido ao paralelizar-se o cálculo dos momentos estatísticos da pressão.

6 CONCLUSÕES

Na presente dissertação, foi apresentada uma proposta de plataforma para controle de simulações paralelas e leitura/ escrita de dados observacionais climatológicos. A plataforma foi pensada para ser altamente flexível, permitindo a extensão de seus recursos através de *plugins*, tornando-a potencialmente aproveitável por uma vasta gama de utilizadores.

O autor deste trabalho está ciente de que existem alternativas no mercado, porém as mesmas possuem uma grande curva de aprendizado, e não necessariamente tem código aberto. O programa objeto de estudo desta dissertação será disponibilizado como *software* livre e de código aberto.

A plataforma foi testada com sucesso em uma simples aplicação para caracterização estatística do fenômeno da ZCAS. O conjunto de dados 20CRv2 se mostrou bastante coerente com a realidade no que concerne a variável pressão de superfície, mas apresentou muitas distorções para a variável umidade relativa, sugerindo a utilização de outros parâmetros para descrição do fenômeno.

A escalabilidade da plataforma se mostrou bastante satisfatória. O problema do cálculo dos momentos estatísticos da pressão foi tratado diversas vezes com diferentes números de processadores dedicados a sua resolução. Quando são utilizados 20 processadores, o problema apresenta um ganho em velocidade de aproximadamente 14 vezes em relação ao serial.

As oportunidades para trabalhos futuros são muitas, desde melhorias na plataforma em si, até extensão de sua biblioteca de funções pela adição de *plugins*. Em curto prazo, pretende-se adicionar outras tecnologias de paralelização, como por exemplo MPI, para trabalhar em conjunto com a paralelização por *threads* (única tecnologia suportada atualmente), tornando-a muito mais poderosa. Também projeta-se em um futuro breve estender a interface oferecida para outras formas de paralelização além da paralelização orientada a pontos espaciais.

REFERÊNCIAS BIBLIOGRÁFICAS

Amdahl, G. M. **Validity of the single processor approach to achieving large scale computing capabilities**, 1967.

Anderson, J. L., Wyman, B., Zhang, S., and Hoar, T. Assimilation of Surface Pressure Observations Using an Ensemble Filter in an Idealized Global Atmospheric Prediction System, **Journal of the Atmospheric Sciences**, vol. 62(8), p. 2925–2938, 2005.

Atkinson, K., Flatt, M., and Lindstrom, G. **ABI compatibility through a customizable language**. page 147. ACM Press, 2010.

Ayachit, U., Geveci, B., Moreland, K., Patchett, J., and Ahrens, J. **High Performance Visualization - The ParaView Visualization Application**. volume 20124456. Chapman and Hall/CRC, 2012.

Blaise, B. **Introduction to Parallel Computing**, 2017.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system, **Journal of parallel and distributed computing**, vol. 37(1), p. 55–69, 1996.

Boehm, H.-J. **Threads cannot be implemented as a library**, 2005.

Brown, D., Brownrigg, R., Haley, M., and Huang, W. **NCAR Command Language (NCL)**, 2012.

Casagrande, J. R. **Decreto Nº 3463-R DE 16/12/2013**. Espírito Santo', 2013.

CH. **What is a Plugin?** `https://www.computerhope.com/jargon/p/plugin.htm`, 2017, Acesso em: 29-06-2017.

Compo, G. P., Whitaker, J. S., and Sardeshmukh, P. D. Feasibility of a 100-Year Reanalysis Using Only Surface Pressure Data, **Bulletin of the American Meteorological Society**, vol. 87(2), p. 175–190, 2006.

Compo, G. P., Whitaker, J. S., Sardeshmukh, P. D., Allan, R. J., McColl, C., Yin, X., Giese, B. S., Vose, R. S., Matsui, N., Ashcroft, L., Auchmann, R., Benoy, M., Bessemoulin, P., Brandsma, T., Brohan, P., Brunet, M., Comeaux, J., Cram, T., Crouthamel, R., Groisman, P. Y., Hersbach, H., Jones, P. D., Jonsson, T., Jourdain, S., Kelly, G., Knapp, K. R., Kruger, A., Kubota, H., Lentini, G., Lorrey, A., Lott, N., Lubker, S. J., Luterbacher, J., Marshall, G. J., Maugeri, M., Mock, C. J., Mok, H. Y., Nordli, O., Przybylak, R., Rodwell, M. J., Ross, T. F., Schuster, D., Srnec, L., Valente, M. A., Vizi, Z., Wang, X. L., Westcott, N., Woollen, J. S., and Worley, S. J. **NOAA/CIRES Twentieth Century Global Reanalysis Version 2c**, 2015.

Compo, G. P., Whitaker, J. S., Sardeshmukh, P. D., Matsui, N., Allan, R. J., Yin, X., Gleason, B. E., Vose, R. S., Rutledge, G., Bessemoulin, P., Brönnimann, S., Brunet, M., Crouthamel, R. I., Grant, A. N., Groisman, P. Y., Jones, P. D., Kruk, M. C., Kruger, A. C., Marshall, G. J., Maugeri, M., Mok, H. Y., Nordli, O., Ross, T. F., Trigo, R. M.,

Wang, X. L., Woodruff, S. D., and Worley, S. J. The Twentieth Century Reanalysis Project, **Quarterly Journal of the Royal Meteorological Society**, vol. 137(654), p. 1–28, 2011.

CONAMA. **Resolução CONAMA n. 5 de 15 de junho de 1989**. Brasília, 1989.

CONAMA. **Resolução CONAMA n. 386 de 26 de dezembro de 2006**. Brasília, 2006.

CONAMA. **Resolução CONAMA n. 436 de 22 de dezembro de 2011**. Brasília, 2011.

CPTEC. **Zona de Convergência do Atlântico Sul (ZCAS) atua em parte do Sudeste e do Centro-Oeste**, 2010.

Dagum, L. and Menon, R. OpenMP: an industry standard API for shared-memory programming, **IEEE Computational Science and Engineering**, vol. 5(1), p. 46–55, 1998.

Eaton, B., Gregory, J., Drach, B., Taylor, C., Hankin, S., Blower, J., Caron, J., Signell, R., Bentley, P., Rappa, G., Höck, H., Pamment, A., Juckes, M., and Raspaud, M. **NetCDF Climate and Forecast (CF) Metadata Conventions**, 2017.

ECMWF. **ERA-40 Monthly Means of Isentropic Level Analysis Data**. Research Data Archive at the National Center for Atmospheric Research, Computational and Information Systems Laboratory, Boulder CO, 2004.

ECMWF. **ERA-Interim Project**. Research Data Archive at the National Center for Atmospheric Research, Computational and Information Systems Laboratory, Boulder CO, 2009.

ENEVA. **Quem Somos – Eneva**. `http://www.eneva.com.br/quem-somos/`, 2017, Acesso em: 07-08-2017.

Evensen, G. Sequential data assimilation with a nonlinear quasi-geostrophic model using Monte Carlo methods to forecast error statistics, **Journal of Geophysical Research: Oceans**, vol. 99(C5), p. 10143–10162, 1994.

Figueroa, S. N., Satyamurty, P., and Da Silva Dias, P. L. Simulations of the Summer Circulation over the South American Region with an Eta Coordinate Model, **Journal of the Atmospheric Sciences**, vol. 52(10), p. 1573–1584, 1995.

Flynn, M. J. Some Computer Organizations and Their Effectiveness, **IEEE Transactions on Computers**, vol. C-21(9), p. 948–960, 1972.

Gandu, A. W. and Geisler, J. E. A Primitive Equations Model Study of the Effect of Topography on the Summer Circulation over Tropical South America, **Journal of the Atmospheric Sciences**, vol. 48(16), p. 1822–1836, 1991.

Gisch, D. L. **Uma equação constituinte para a dispersão não-linear de poluentes na camada limite atmosférica turbulenta : fechamento fickiano modificado e a presença de fase**. PhD thesis, 2014.

Grotendorst, J., editor. **Quantum simulations of complex many-body systems: from theory to algorithms: winter school, 25 February - 1 March 2002, Rolduc Conference Centre, Kerkrade, the Netherlands ; lecture notes**. Number 10 in NIC series. NIC-Secretariat, Jülich, 2002.

Gustafson, J. L. Reevaluating Amdahl's Law, **Commun. ACM**, vol. 31(5), p. 532–533, 1988.

IEEE. IEEE Standard for Information Technology - POSIX-Based Supercomputing Application Environment Profile, **IEEE Std 1003.10-1995**, vol. -, p. 0_1–, 1995.

INMET. **INMET - Instituto Nacional de Meteorologia - BDMEP**, 2017.

ISO. **ISO/IEC 14882:2011**, 2011.

Jong, J. d. **Tutorial - How to create an expression parser**, 2006.

Kessler, C. and Keller, J. **Models for Parallel Computing: Review and Perspectives**, 2007.

Kodama, Y. Large-Scale Common Features of Subtropical Precipitation Zones (the Baiu Frontal Zone, the SPCZ, and the SACZ) Part I: Characteristics of Subtropical Frontal Zones, **Journal of the Meteorological Society of Japan. Ser. II**, vol. 70(4), p. 813–836, 1992.

Kodama, Y.-M. Large-Scale Common Features of Sub-Tropical Convergence Zones (the Baiu Frontal Zone, the SPCZ, and the SACZ) Part II : Conditions of the Circulations for Generating the STCZs, **Journal of the Meteorological Society of Japan. Ser. II**, vol. 71(5), p. 581–610, 1993.

LGSA. **A Empresa - LGSA**. `http://www.lgsa.com.br/a-empresa/`, 2017, Acesso em: 07-08-2017.

Ling, Y., Mullen, T., and Lin, X. Analysis of optimal thread pool size, **Operating Systems Review (ACM)**, vol. 34(2), p. 42–55, 2000.

Loeck, J. F. **Efeitos estocásticos em modelos determinísticos para dispersão de poluentes na camada limite atmosférica**. PhD thesis, 2014.

Maps, G. **Google Maps**, 2017.

Mishra, B. S. P. and Sagnika, S. **Techniques and Environments for Big Data Analysis. Parallel, Cloud, and Grid Computing**. Springer International Publishing, Cham, 2016.

Moors, J. J. A. The Meaning of Kurtosis: Darlington Reexamined, **The American Statistician**, vol. 40(4), p. 283–284, 1986.

NCEP and NCAR. **NCEP/NCAR Global Reanalysis Products, 1948-continuing**. Research Data Archive at the National Center for Atmospheric Research, Computational and Information Systems Laboratory, Boulder CO, 1994.

Nobre, C. Ainda sobre a Zona de Convergência do Atlântico Sul: A importância do Oceano Atlântico, **Climanalise**, vol. 3(4), 1988.

Padala, P. **NCURSES Programming HOWTO**, 2005.

Pregorim, J. **ZCAS se forma sobre o Brasil**, 2016.

Rayner, N. A., Parker, D. E., Horton, E. B., Folland, C. K., Alexander, L. V., Rowell, D. P., Kent, E. C., and Kaplan, A. Global analyses of sea surface temperature, sea ice, and night marine air temperature since the late nineteenth century, **Journal of Geophysical Research: Atmospheres**, vol. 108(D14), p. 4407, 2003.

Rew, R., Davis, G., Emmerson, S., Cormack, C., Caron, J., Pincus, R., Hartnett, E., Heimbigner, D., Lynton Appel, and Fisher, W. **Unidata NetCDF**, 1989.

Santos, E., Poco, J., Wei, Y., Liu, S., Cook, B., Williams, D. N., and Silva, C. T. UV-CDAT: Analyzing Climate Datasets from a User's Perspective, **Computing in Science Engineering**, vol. 15(1), p. 94–103, 2013.

Sayfan, G. **A Cross-Platform Plugin Framework for C/C++**, 2007.

Schramm, J. **Estudo da dispersão de poluentes em uma usina termelétrica localizada em linhares utilizando o modelo Calpuff**. PhD thesis, 2016.

Shea, D. **Common Climate Data Formats: Overview | NCAR - Climate Data Guide**, 2013.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. **MPI - The Complete Reference**, 1996.

Stroustrup, B. **The C++ programming language**. Addison-Wesley, Upper Saddle River, NJ, fourth edition edition, 2013.

Whitaker, J. S., Hamill, T. M., Wei, X., Song, Y., and Toth, Z. Ensemble Data Assimilation with the NCEP Global Forecast System, **Monthly Weather Review**, vol. 136(2), p. 463–482, 2008.

WMO. **Guide to GRIB**. `https://www.wmo.int/pages/prog/www/WDM/Guides/Guide-binary-2.html`, 2017, Acesso em 27-06-2017.

Ylonen, T. and Lonvick, C. **The Secure Shell (SSH) Protocol Architecture**, 2006.

Yoo, A. B., Jette, M. A., and Grondona, M. **SLURM: Simple Linux Utility for Resource Management**. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

Zender, C. **NCO User Guide**, 2017.