UNIVERSIDADE FEDERAL DE PELOTAS
Centro de Desenvolvimento Tecnológico
Programa de Pós-Graduação em Computação

Dissertação

Kanga: Uma Interface Genérica Baseada em Esqueletos para Programação Paralela em Anahy3

Deives Mesquita Kist

Pelotas, 2014

Deives Mesquita Kist

Kanga: Uma Interface Genérica Baseada em Esqueletos para Programação Paralela em Anahy3

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Mestre em Ciência da Computação

Orientador: Prof. Dr. Gerson Geraldo H. Cavalheiro
Co-orientador: Prof. Dr. André Rauber Du Bois

Pelotas, 2014

Dados Internacionais de Catalogação na Publicação (CIP)

K61s Kist, Deives Mesquita
Kanga: uma interface genérica baseada em esqueletos para programação paralela em Anahy3 / Deives Mesquita Kist; Gerson Geraldo Homrich Cavalheiro, orientador; André Rauber Du Bois, coorientador. - Pelotas, 2014.
87 f.: il.

Dissertação (Mestrado) – Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2014.

1. Interface de programação. 2. Multithread. 3. Esqueletos paralelos. I. Cavalheiro, Gerson Geraldo Homrich, orient. II. Du Bois, André Rauber, co-orient. III. Título.

CDD: 005

Catalogação na Fonte: Aline Herbstrith Batista CRB 10/ 1737
Biblioteca Campus Porto - UFPel

Dedico esta conquista a minha família e a minha namorada que me incentivaram e apoiaram durante todo o tempo que estive realizando este trabalho.

AGRADECIMENTOS

Em primeiro lugar agradeço a Deus pelas minhas conquistas e pela minha família, amigos e por me dar força nos momentos em que preciso. Também, quero agradecer aos meus pais, Inácio e Elisábethe, e a minha irmã Ingrid por me incentivar, apoiar e por estarem presentes em todos os momentos marcantes da minha vida.

Agradeço especialmente a minha namorada, Nouara, por ter tido paciência comigo durante todo o mestrado e por ter sido uma companheira na horas mais difíceis. Além disso, agradeço por ter sido a revisora dos meus textos.

Também agradeço aos meus orientadores, Prof. Gerson e Prof. André, que me deram a oportunidade de realizar este trabalho e por me orientar durante a execução deste trabalho.

Agradeço o apoio dos meus amigos do laboratório LUPS e também aos demais amigos pelo o apoio e pelo o incentivo durante o trabalho de mestrado.

RESUMO

KIST, Deives Mesquita. **Kanga: Uma Interface Genérica Baseada em Esqueletos para Programação Paralela em Anahy3**. 2014. 87 f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Universidade Federal de Pelotas, Pelotas, 2014.

Os programadores têm grandes dificuldades com a programação paralela, principalmente, porque devem dividir a aplicação em partes que possam ser executadas simultaneamente e mapeá-las sobre os núcleos disponíveis no processador de forma que a aplicação tenha bom desempenho. Para realizar essas tarefas são utilizadas ferramentas de programação paralela, neste trabalho é utilizada a ferramenta de programação *multithread* Anahy3 que tem uma interface baseada no padrão *POSIX Threads*. O Anahy3 oferece recursos para descrição e controle da concorrência e também mecanismos de escalonamento. Estes mecanismos são executados de forma automática pelo ambiente de execução deixando o programador livre da responsabilidade de mapear as atividades entre os núcleos do processador.

O objetivo deste trabalho é estender a interface de programação de Anahy3 por meio de recursos de programação paralela de mais alto nível. Para implementar estes novos recursos encontramos na literatura o conceito de algoritmos de esqueletos paralelos. Os esqueletos são modelos de programação paralela de alto nível que encapsulam padrões de algoritmos recorrentes de programação paralela.

Neste trabalho foi desenvolvida uma interface constituída por esqueletos implementados na linguagem C++ por meio de classes *templates*. Na implementação realizada é possível que o programador defina os tipos de dados a serem manipulados pelos esqueletos, assim eliminando o uso de ponteiros do tipo void* característicos de Pthreads. Os esqueletos que compõem a interface são: Map, Reduce, Zip, Scan, Fork, MapReduce, Farm e Divisão e Conquista.

Palavras-chave: Interface de programação, Multithread, Esqueletos Paralelos.

ABSTRACT

KIST, Deives Mesquita. **Kanga: An Generic Interface Based on Skeletons to Parallel Programming on Anahy3**. 2014. 87 f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Universidade Federal de Pelotas, Pelotas, 2014.

Programmers have great difficulties on parallel programming, mainly, because they must split the application in parts which can be executed simultaneously and mapping them on the processor cores that the application have a performace good. To perform this tasks are used parallel programming tools, this work is used the programming tool mutithread Anahy3 which have an interface based on standard POSIX Threads. Anahy offers resources to description and control of competition and also scheduling mechanisms. These mechanisms are automatically executed by execution environment so the programmer does not have responsibility of distribute the activities between the processor cores.

The target this work is to extend the programming interface of Anahy3 using parallel programming resources of high level. To implement these new resources we found in literature the Skeletons. The Skeletons are a high-level parallel programming model that encapsulate standards algorithms of parallel programming.

In this work was developed an interface constituted by skeletons implemented on language C++ using templates classes. In the implementation realized is possible that the programer define the types of data manipulated by skeletons, so eliminating the use of pointers of type void* feature of Pthreads. The skeletons which belong the interface are: Map, Reduce, Zip, Scan, Fork, MapReduce, Farm e Divide and Conquer.

Keywords: Programming Interface, Multithread, Parallel Skeletons.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
BB	*Branch and Bound*
DatTel	*Data parallel Template Library*
DC	Divisão e Conquista
eSkel	*Edinburgh Skeleton Library*
eDM	*eSkel Data Model*
GPU	*Graphics Processing Unit*
MIMD	*Multiple Instruction Multiple Data*
MPI	*Message Passing Interface*
SIMD	*Single Instruction Multiple Data*
SPMD	*Single Program Multiple Data*
STL	*Standard Template Library*
SkeTo	*Skeletons in Tokyo*

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de *software* paralelo tem um alto grau de dificuldade, uma vez que o programador deve não apenas codificar seu algoritmo, mas também decompô-lo em termos de atividades concorrentes e gerir o bom uso dos recursos de *hardware* disponíveis. Nota-se, no entanto, um esforço da comunidade em disponibilizar ferramentas para a programação paralela com níveis mais elevados de abstração, absorvendo, pelo menos parcialmente, a responsabilidade pela decomposição do problema e/ou da exploração dos recursos de processamento.

As ferramentas de programação paralela fornecem ao programador mecanismos para superar o desafio de converter um programa sequencial em paralelo e distribuir as partes paralelas entre os núcleos de processamento. Além disso, as ferramentas auxiliam o desenvolvedor a tratar as formas de acesso aos dados pelos processos, pois se o acesso não for controlado poderá gerar uma inconsistência de dados. Este controle é realizado por meio de mecanismos de comunicação e sincronização entre os fluxos de execução, permitindo que o programador indique claramente a dependência entre estes, para que o ambiente de execução seja capaz de coordenar o acesso concorrente dos dados.

Dentre as ferramentas mais tradicionais de programação paralela citamos: OpenMP (CHANDRA et al., 2001), Pthreads (DAGUM; MENON, 1998) e MPI (AOYAMA; NAKANO, 1999). Outras ferramentas como Cilk+ (BLUMOFE et al., 1996), Kaapi (GAUTIER; BESSERON; PIGEON, 2007) e Anahy (CAVALHEIRO et al., 2006) oferecem, além dos recursos para descrição e controle da concorrência, facilidades para o desenvolvimento de aplicativos paralelos por meio de mecanismos de escalonamento. Estes mecanismos são executados de forma transparente pelo ambiente de execução, deixando o programador livre da responsabilidade de mapear as atividades entre os núcleos do processador. Contudo, estas ferramentas oferecem poucos mecanismos para auxiliar o programador na descrição da concorrência de sua aplicação. Neste trabalho iremos considerar o caso de Anahy3 (CAVALHEIRO et al., 2006) que tem uma interface

baseada no padrão *POSIX Threads*.

A interface de programação do Anahy propõe o uso de duas primitivas, *fork* e *join*, para criar e sincronizar *threads*, respectivamente. Com uso deste limitado número de recursos, o programador deve descrever toda manifestação de concorrência no seu algoritmo. Embora seja possível descrever uma grande variedade de estruturas concorrentes com estas primitivas, o nível de abstração oferecido ao programador é bastante baixo, uma vez que construtores paralelos mais abstratos não estão disponíveis. Como consequência, o tempo de desenvolvimento de programas é longo, bem como sua robustez sujeita a exaustivos testes de verificação. A ideia deste trabalho é estender a interface de programação de Anahy com recursos de programação paralela de mais alto nível.

Para implementar este novo conjunto de recursos de programação, encontramos na literatura o conceito de *Skeletons* (esqueletos) (VÉLEZ; LEYTON, 2010) como uma alternativa para especificação de construtores paralelos de alto nível. Os esqueletos são abstrações para construtores paralelos que encapsulam padrões de algoritmos recorrentes de programação paralela. Estes esqueletos implementam padrões simples que podem ainda ser combinados para a construção de padrões mais complexos. Os esqueletos capturam, organizam e mascaram do programador os detalhes envolvidos na estrutura da computação paralela.

Este trabalho propõe a construção de uma API (*Application Programming Interface*) genérica que oferece a implementação de esqueletos, tendo a sua interface e implementação na linguagem C++. Além disso, esta API possui suporte do ambiente Anahy para o paralelismo de tarefas.

1.1 Motivação

A popularização dos processadores *multicore* tem aumentado o desenvolvimento de aplicações que se beneficiam do maior poder de processamento destes processadores. Contudo, a programação paralela agrega mais dificuldades ao processo de desenvolvimento de *software*. Logo, a adição de esqueletos à Anahy facilitará a programação paralela, pois a sua capacidade de expressão será aumentada pelo uso de uma interface de programação de mais alto nível. Os esqueletos simplificarão os projetos de aplicações paralelas ao oferecerem soluções padrões de problemas recorrentes e poderão ser combinados para construir aplicações mais robustas. Além disto, as aplicações desenvolvidas utilizando estes esqueletos se beneficiarão dos mecanismos de escalonamento disponíveis no Anahy e terão seus níveis de robustez mais elevados, tendo em vista a prévia validação da implementação dos esqueletos.

Esqueletos utilizam padrões de programação comuns para ocultar a complexidade de aplicações paralelas e distribuídas. A partir de um conjunto básico de esqueletos pode-se construir padrões mais complexos combinando eles. Assim, a programação é basicamente reduzida à habilidade de capturar e compor corretamente o conjunto de esqueletos para a aplicação. Somado a isso, os esqueletos realizam todas as etapas de criação e destruição de processos, além de realizar a sincronização deles. Devido a isto, o programador não precisará se preocupar com problemas de *deadlock* e condição de corrida, pois serão tratados internamente pelos esqueletos. Isto significa que os problemas de baixo nível como concorrência, comunicação, sincronização e o balanceamento de carga são atendidos pela implementação do esqueleto, ou seja, o esqueleto libera o programador de resolver estes problemas.

1.2 Objetivos

O principal objetivo deste trabalho é o desenvolvimento de uma interface de programação (API) composta de padrões recorrentes na programação paralela por meio dos esqueletos. A API será utilizada para estender a interface de programação de Anahy3, fornecendo recursos de programação paralela de mais alto nível.

Na Figura 1 é apresentado um esquema da organização do modelo de programação e execução de Anahy, sendo destacado o módulo que foi desenvolvido neste trabalho. Esta figura destaca que as atuais primitivas desta ferramenta de tratamento de *threads* continuarão a serem fornecidas aos programadores mas novas primitivas, sob a forma de esqueletos, também serão oferecidas, sendo estas construídas utilizando os recursos básicos oferecidos pelas primitivas *fork*/*join*.

A Figura 1 também apresenta os Jobs, que são definidos como recursos do sistema e são alocados aos Processadores Virtuais (VPs) por meio do gerenciador, identificado como AnahyVM. AnahyVM é a máquina virtual do ambiente responsável por gerenciar os VPs e realizar o escalonamento de primeiro nível. Neste primeiro nível *threads* são criados na camada de aplicação e distribuídos entre os VPs da arquitetura. O segundo nível de escalonamento é interno a cada VP e diz respeito ao gerenciamento que a estrutura local de cada VP realiza. O princípio desta distribuição é de que o próprio VP pode implementar a sua estratégia de escalonamento, permitindo que além uma menor contenção dos VPs para a execução dos Jobs do programa, uma vez que os VPs trabalham sobre seu próprio conjunto de Jobs, diferentes algoritmos de escalonamento podem ser utilizados por cada VP dependendo do tipo e/ou custo de cada Job.

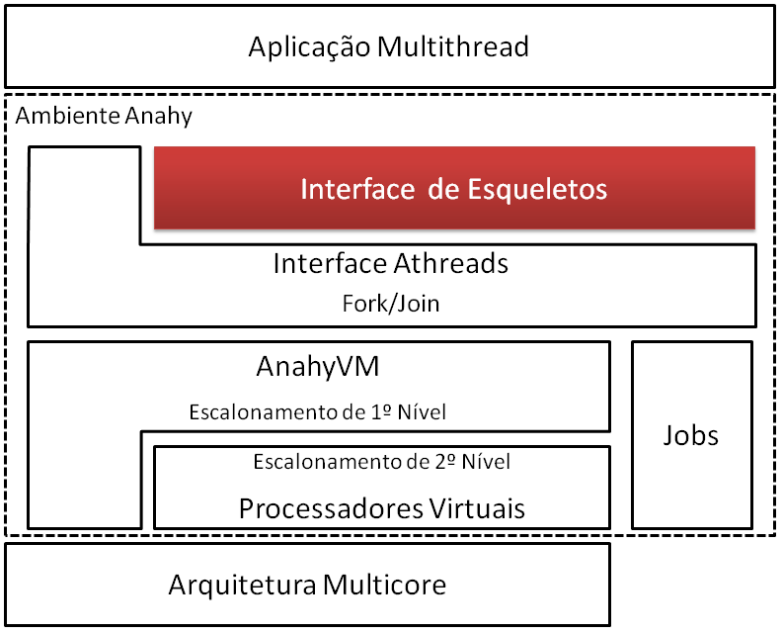


Figura 1: Ilustração da API de esqueletos em conjunto com a organização ferramenta de programação paralela Anahy.

1.3 Resultados Alcançados

Neste trabalho foi desenvolvida uma interface constituída por esqueletos implementados na linguagem C++ por meio de classes *templates*. Na implementação realizada é possível que o programador defina os tipos de dados a serem manipulados pelos esqueletos, eliminando, assim, também o uso de ponteiros do tipo void * característicos de Pthreads. Os esqueletos que compõem a interface são: Map, Reduce, Scan, Zip, Fork, MapReduce, Divisão e Conquista (DC) e Farm.

A interface desenvolvida tem uma estrutura de hierarquia de classes, onde a base é uma classe abstrata que contém a declaração dos métodos. Os métodos são implementados pelos esqueletos e por meio deles o programador manipula os objetos das classes de esqueletos. O padrão de projeto *Composite* (GAMMA et al., 2000) foi usado na especificação dos esqueletos para permitir que os mesmos pudessem ser compostos ou aninhados. Desta forma, cada classe tem um ponteiro do tipo da classe base, assim, um objeto pode executar qualquer outro objeto invocando os métodos definidos na classe base.

1.4 Organização do Texto

Neste primeiro Capítulo foi introduzido o presente trabalho, bem como a motivação e os objetivos deste trabalho.

No Capítulo 2 encontram-se descritos os conceitos relacionados aos esque-

letos, como a sua origem nas linguagens funcionais e a implementação de esqueletos em outras linguagens de programação. Além disso, são apresentados *Frameworks* de esqueletos encontrados na bibliografia.

No Capítulo 3 são descritos oito modelos de esqueletos para a programação paralela. Estes esqueletos foram implementados na solução proposta. Para cada esqueleto apresentado, também é discutida o modelo de execução paralela adotado.

No Capítulo 4 é descrita a API que foi desenvolvida neste trabalho. Neste Capítulo é feita uma descrição detalhada da implementação dos esqueletos que integram esta API e também a forma como os programadores irão utilizar esta API.

As ferramentas de programação paralela utilizadas para o desenvolvimento deste trabalho são apresentadas no Capítulo 5. As ferramentas são: Pthreads e Anahy. Os seus mecanismos de escalonamento também serão descritos.

Os testes realizados neste trabalho para validar a interface desenvolvida são apresentados no Capítulo 6.

O Capítulo 7 apresenta as considerações finais sobre este trabalho e indica os trabalhos futuros.

2 ESTADO DA ARTE

Neste capítulo são apresentados o conceito de esqueleto e os princípios defendidos por Murray Cole (COLE, 2004) na implementação dos esqueletos. Além disso, são listados alguns esqueletos presentes na literatura e *frameworks* de esqueletos. Este capítulo termina apresentando em maiores detalhes de duas ferramentas que utilizam o conceito de esqueletos para a programação paralela. Estas ferramentas são consideradas para avaliação da ferramenta construída no presente trabalho.

2.1 Esqueletos

Um esqueleto é uma função de alta ordem que encapsula um padrão recorrente de paralelismo. Este tipo de função é denominado alta ordem porque pode receber funções como parâmetros, esta característica é originada de funções matemáticas e implementada nas linguagens funcionais (SEBESTA, 2011). Os padrões recorrentes são soluções para problemas comuns no desenvolvimento de software, com a popularização da computação paralela os esqueletos têm sido implementados para serem executados paralelamente (MCCOOL; ROBISON; REINDERS, 2012).

Na programação paralela os esqueletos capturam, organizam e mascaram do programador todos os detalhes envolvidos na estrutura dessa computação que não são relevantes para o programador. Eles são utilizados para resolver problemas rotineiramente encontrados quando se desenvolve programas para execução paralela.

Os esqueletos favorecem a programabilidade, pois são implementados para serem utilizados como funções que recebem parâmetros assim o programador poderá utilizá-los para obter a concorrência da sua aplicação como uma função sequencial. Isto permite a construção de uma aplicação paralela utilizando esqueletos de forma análoga ao modo como se constrói um programa sequencial.

Outra característica dos esqueletos é facilitar a portabilidade entre as diferen-

tes plataformas de programação paralela, pois os esqueletos agem como uma interface entre estas e as aplicações paralelas. Logo é possível que exista uma implementação diferente para cada plataforma com a mesma interface. Isso permite que um aplicativo seja escrito usando esqueletos e que possa ser portável entre as plataformas. Esta característica é considerada um princípio fundamental de muitas estruturas de esqueletos modernos de programação.

Apesar de ter uma interface independente de plataforma, uma implementação de esqueleto pode ser otimizada internamente para explorar os recursos de sincronização, comunicação e paralelismo de uma arquitetura. O esqueleto pode ser facilmente expandido ou ligado a uma implementação eficiente para uma plataforma que engloba todos os detalhes específicos relacionados com paralelismo, balanceamento de carga e comunicação.

A implementação de esqueletos propõe que padrões recorrentes na programação paralela sejam abstraídos e fornecidos aos desenvolvedores em forma de funções, com especificações que transcendem as variações de arquiteturas, e que permitem o aumento do desempenho (COLE, 2004). (COLE, 2004) afirma que esqueletos podem solucionar problemas de desenvolvimento de software paralelo, da seguinte forma:

- simplificando a programação paralela por meio do aumento do nível de abstração;
- aumentando a portabilidade e o reuso de código;
- melhorando o desempenho por meio de implementações otimizadas de esqueletos para arquitetura específicas;
- oferecendo implementações, testadas e bem documentadas, de padrões frequentemente utilizados na programação paralela.

Na seção seguinte são descritos alguns conceitos que devem ser considerados na implementação dos esqueletos.

2.2 Princípios Importantes

De acordo com (COLE, 2004) quatro princípios devem ser considerados no projeto e na implementação de esqueletos paralelas. Os quatro são descritos a seguir:

1. **Oferecer o esqueleto de forma simples:** no desenvolvimento de esqueletos deve-se manter a simplicidade e não inserir sintaxes que não façam parte da linguagem de programação utilizada.

2. **Integrar os esqueleto com o paralelismo *ad-hoc*:** algumas aplicações utilizam menos ou mais estruturas e primitivas, ou seja, cada aplicação expressa o paralelismo de uma maneira diferente. Logo, os esqueletos devem ser desenvolvidos de forma que possam se adequar a estas aplicações.

3. **Permitir a diversidade:** as pesquisas da área de esqueletos têm testemunhado o aparecimento de esqueletos simples e uma variedades de esqueletos mais complexos. As especificações precisas causam variações na semânticas que refletem na maneira que os esqueletos são aplicados em algoritmos reais. Essas especificações acarretam que alguns algoritmos não podem ser expressados em certos sistemas. Logo, deve-se ter o equilíbrio entre a abstração e a flexibilidade quando da concepção e implementação de um esqueleto.

4. **Mostrar o *pay-back*:** para que uma nova tecnologia tenha uma boa aceitação deve-se demonstrar que ela oferece alguma vantagem. Isso é realizado por meio de experimentos e resultados, que devem ser catalogados. Assim, precisa-se demonstrar que um aplicativo implementado com esqueletos supera a sua implementação convencional.

Além destes princípios, (BENOIT; COLE, 2006) definem mais dois conceitos relacionados à composição de esqueletos e à forma de interação com os esqueletos. Para (BENOIT; COLE, 2006) os esqueletos podem ser aninhados ou compostos de forma transitória ou de forma persistente. Além disso, o aninhamento desses esqueletos pode conter vários níveis, onde existem esqueletos mais internos e outros mais externos.

Em um aninhamento transitório um esqueleto executa outro esqueleto durante um determinado período, a fim de processar alguns dados disponíveis localmente, ou para realizar alguma outra computação independente, após isto o esqueleto pode chamar outros esqueletos. No caso de um aninhamento persistente a composição é pré-definida, ou seja, um esqueleto sempre executa o mesmo esqueleto.

As interações com esqueletos podem ser explícitas ou implícitas, as interações explícitas ocorrem no momento da invocação de um esqueleto e as interações implícitas ocorrem no momento de retorno do esqueleto. Contudo, as formas de interação com esqueletos podem variar para cada esqueleto, devido cada um ter características próprias.

2.3 Classificação dos Esqueletos

O *survey* (VÉLEZ; LEYTON, 2010) sugere que os esqueletos podem ser classificados da seguinte forma: esqueletos de paralelismo de dados, esqueletos de paralelismo de tarefas e esqueletos para resolução de problemas. A Tabela 1 resume a classificação dos esqueletos, mostrando o escopo de atuação dos esqueletos pertencentes a cada classificação. Além disso, são citados alguns exemplos de esqueletos que pertencem a cada classificação.

Os esqueletos que realizam o paralelismo de dados operam sobre estruturas de dados, essas estruturas são utilizadas para armazenar, padronizar e representar os dados de forma a tornar possível a execução de operações paralelas sobre um conjunto de dados. Estes esqueletos se comportam estabelecendo relações de correspondências entre os dados e suas estruturas regulares. Os esqueletos de paralelismo de dados tem uma carga de trabalho com granularidade fina.

Os esqueletos que trabalham com tarefas se comportam de acordo com a interação existente entre elas. Estes esqueletos tem uma granularidade variável definida por parâmetro.

Esqueletos para resolução de problemas esboçam métodos algorítmicos para tratar uma determinada família de problemas. Seu comportamento reflete a natureza da solução para uma família de problemas, e a sua estrutura pode envolver diferentes cálculos e primitivas de controle.

Tabela 1: Classificação dos esqueletos. Fonte: (VÉLEZ; LEYTON, 2010)

Classificação	Escopo dos esqueletos	Exemplos de esqueletos
Paralelismo de dados	Estruturas de dados	Map, Reduce, MapReduce, Fork, Scan, Zip
Paralelismo de tarefas	Tarefas	Seq, Farm, Pipe, If, For, While
Resolução de problemas	Famílias de problemas	Divisão e Conquista, Programação dinâmica, *Branch and Bound*

2.4 Frameworks de Esqueletos Paralelos

Existem três principais abordagens para implementação de esqueletos (FALCOU et al., 2006). Em uma delas os esqueletos podem ser utilizados na construção de novas linguagens. Esta forma de usar os esqueletos apresenta

bons índices de desempenho mas que exige, em contrapartida, que o programador aprenda esta nova linguagem, o que pode resultar em um empecilho para a popularização de uma solução com esta abordagem. Outra forma de utilizar os esqueletos é por meio de um compilador para um determinada linguagem. O compilador identifica estruturas em um código fonte sequencial e utiliza uma implementação paralela para esta estrutura. Esta solução é adotada em algumas linguagens modernas, como em OpenMP no seu *for* paralelo que oferece uma alternativa ao esqueleto Map, mas torna a ferramenta pouco flexível. A terceira forma, é implementar os esqueletos como uma biblioteca de programação. Esta solução permite um reuso de código e é de grande aceitação entre os programadores. O presente trabalho segue esta última alternativa de implementação dos esqueletos.

Os esqueletos oferecidos aos programadores por meio de *frameworks*, bibliotecas ou APIs, podem ser construídos utilizando linguagens de programação do tipo de coordenação, funcionais, orientadas a objetos e imperativas. Também, podem ser implementados para diferentes plataformas de programação. Desta forma, (VÉLEZ; LEYTON, 2010) descreve características que podem ser utilizadas pelos programadores para escolher o *framework* mais adequado as suas necessidades. As características são:

- **Linguagem de programação:** é por meio de uma linguagem de programação que os esqueletos são utilizados. Existem diversas linguagens como: funcionais, de coordenação, de marcação, imperativas, orientadas a objeto e com interface gráficas.

- **Linguagem de execução:** é a linguagem para a qual os esqueletos são compilados, podendo ser diferente da linguagem de programação que os esqueletos são implementados, especialmente quando se trata com linguagens funcionais. Um mecanismo de tradução é utilizado para converter a aplicação do esqueleto, definida na linguagem de programação, em uma aplicação equivalente na linguagem de execução. Diferentes processos de tradução e geração de código de esqueletos em um nível mais baixo interagem com uma biblioteca da linguagem de execução. A tradução da aplicação pode as vezes introduzir código para uma arquitetura específica transformando-a numa aplicação paralela.

- **Bibliotecas distribuídas:** são as ferramentas utilizadas para desenvolver aplicações paralelas/distribuídas. Por exemplo: Pthreads, MPI, OpenMP e Anahy.

- **Detecção de tipos:** Refere-se a capacidade de detectar os erros de tipo

de dados na programação dos esqueletos. Os primeiros *frameworks* foram desenvolvidos nas linguagens funcionais como *Haskell*, assim a detecção de tipo é herdada da linguagem. No entanto, linguagens personalizadas e não funcionais precisam de compiladores que realizem detecção de tipos.

- **Aninhamento de esqueletos:** é a capacidade de compor hierarquicamente os esqueletos. Esqueletos aninhados são considerados uma importante característica, pois permitem criar padrões mais complexos. No entanto, o aninhamento de padrões apresenta dificuldades de programação e de verificação de tipos.

- **Acesso a arquivos:** é a capacidade de acessar e manipular arquivos de uma aplicação. Assim, é importante prover mecanismos de manipulação de dados e arquivos para o desenvolvimento de esqueletos.

- **Conjunto de esqueletos:** é a lista de esqueletos suportados. O conjunto de esqueletos varia para cada *framework* e muitos esqueletos com o mesmo nome possuem diferentes semânticas.

A Tabela 2 relaciona como os itens descritos anteriormente podem ser utilizados para implementar os quatro princípios de (COLE, 2004). De acordo com (LEYTON, 2008) utilizando uma linguagem de programação é possível desenvolver uma biblioteca de esqueletos que mantenha o princípio de implementação simples e que também permita o paralelismo do tipo *ad-hoc*. Uma linguagem de execução pode mostrar o *pay-back* desde que seja suportada por diferentes sistemas operacionais e diferentes *hardware*. As bibliotecas distribuídas ou paralelas permitem o paralelismo do tipo *ad-hoc* por meio das suas funcionalidades e para mostra o *pay-back* estas devem ser suportadas por diferentes sistemas operacionais e diferentes *hardwares*. Um *framework* deve oferecer um sistema de detecção de tipos de dados transparente ao programador, para aumentar a eficiência dos programadores, o que facilita mostrar *pay-back*. O aninhamento ou a composição de esqueletos permite que exista uma diversidade de esqueletos. Um *framework*, também, deve ter um sistema de arquivos e tratamento de dados otimizados e transparentes ao programador para que os esqueletos suportem grande quantidades de dados, o que permitirá mostrar o *pay-back* com bons índices de desempenho. Um *framework* deve oferecer um conjunto de esqueletos, cada um implementado de acordo com sua definição, e que possam ser testados para mostrar o *pay-back*.

Tabela 2: Características importantes para os quatro princípios. Fonte: (LEYTON, 2008)

Características	Implementação simples	Integrar paralelismo *ad-hoc*	Permitir a diversidade	Mostrar o *pay-back*
Ling. de programação	x	x		
Ling. de execução				x
Bibliotecas distribuídas		x		x
Detecção de tipos	x			x
Aninhamento de esqueletos			x	
Acesso a arquivos	x			x
Conjunto de Esqueletos	x			x

2.5 Frameworks de Esqueletos

Os *frameworks* de esqueletos são implementados utilizando diferentes linguagens de programação, sobre diferentes plataformas de programação paralela e podem ser destinados para processadores *multicore*, *clusters* e GPUs. Os esqueletos têm origem nas linguagens funcionais, sendo os *frameworks* Eden (LOOGEN; MALLEN; MARí, 2005) e HDC (HERRMANN; LENGAUER; ROBERT, 2009) exemplos de implementações na linguagem funcional Haskell. Contudo, devido a popularidade e ao bom desempenho de aplicações paralelas utilizando linguagens procedurais e orientadas a objetos como C/C++ e Java existe um grande número de exemplos de *frameworks* implementados nestas linguagens.

O eSkel (*Edinburgh Skeleton Library*) é um *framework* desenvolvido para fins acadêmicos (ESKEL, 2013), por Murray Cole, para implementar os esqueletos que ele projetou. Além de implementar os princípios que ele defende no desenvolvimento de esqueletos paralelos. O eSkel foi desenvolvido utilizando linguagem C sobre a plataforma MPI e oferece os seguintes esqueletos: Pipe, Farm e o Divisão e Conquista. Este *framework* se encontra na versão 0.2.

O SkeTo (*Skeletons in Tokyo*) (EMOTO; MATSUZAKI, 2013) implementa diferentes estruturas de dados como listas, árvores e matrizes para que sejam utilizadas pelos esqueletos de paralelismo de dados, nos esqueletos de paralelismo de tarefas não são oferecidos. Sketo foi desenvolvido sobre a plataforma MPI na linguagem C++ para *clusters* com processadores *multicores* e está na versão 1.5.

O JaSkel (FERREIRA; SOBRAL; PROENCA, 2006) é uma biblioteca Java que oferece os esqueletos Farm, Pipe e Heartbeat. Além desse, existe o Skandium (LEYTON; PIQUER, 2010) que oferece esqueletos para paralelismo de dados e tarefas. Em ambos, os programadores podem utilizar os seus esqueletos

por meio da herança das classes que os implementam.

Dentre os *frameworks* listados neste capítulo, o SkePU e o Muesli são objetos de uma discussão mais aprofundada em função das semelhanças que possuem com a presente proposta. Estes *frameworks* foram implementados na linguagem C++, usam *templates* e os conjuntos dos esqueletos oferecidos por cada um é semelhante aos implementados neste trabalho, diferindo nas plataformas de execução.

2.5.1 Muesli

O Muesli (CIECHANOWICZ; KUCHEN, 2010) é uma biblioteca de esqueletos de paralelismo de dados e tarefas. Esta biblioteca é implementada em C++ e utiliza *templates*. O Muesli está na versão 2.2 e usa o MPI e OpenMP para realizar a execução paralela.

Muesli oferece esqueletos de paralelismo de dados e de paralelismo de tarefas. Os esqueletos de paralelismo de tarefas são implementados por classes separadas, tais como: Pipe, Farm, Divisão e Conquista e *Branch and Bound*. Os esqueletos de paralelismo de dados são implementados como métodos de classes de estruturas de dados utilizados pelo Muesli, são eles: Map, MapReduce, Scan, Zip e Fold (Reduce) e algumas variações destes (CIECHANOWICZ; KUCHEN, 2010).

O Muesli fornece estruturas de dados para serem usadas pelos esqueletos, como vetores e matrizes. Estas estruturas são implementadas como uma estratégia para distribuir os dados nas arquiteturas distribuídas. Os tipos de dados utilizados por estas estruturas são definidos pelos seus *templates*. Além disso, este *framework* implementa o suporte para funções parciais e *Curring* (CIECHANOWICZ; KUCHEN, 2010).

Para utilizar os esqueletos do Muesli o programador pode utilizar funções da própria linguagem, contudo os esqueletos de paralelismo de dados devem ser usados por meio de objetos das estruturas de dados disponíveis na biblioteca. Assim, para utilizar, por exemplo o esqueleto Map o programador deve criar um vetor por meio de um objeto da classe *DistributedArray*, provida pelo *framework*, e invocar o método map, passando a função que será aplicada em cada elemento do vetor. As classes das estruturas de dados são as seguintes (CIECHANOWICZ; POLDNER; KUCHEN, 2012):

```
1 template<class E> class DistributedArray
2 template<class E> class DistributedMatrix
3 template<class E, class S, class D> class DistributedSparseMatrix
```

Um exemplo de uma construção de um objeto da classe DistributedArray com um vetor de 10 elementos, pode ser visto a seguir:

```
1 int vet[10];
2 DistributedArray<int> v0(10, vet);
```

Esta estrutura é inicializada com um vetor da própria linguagem C++ e o tamanho do vetor é passado por parâmetro no construtor. No momento da execução a quantidade de elementos deste vetor que cada unidade de execução paralela recebe é calculada dividindo o número de elementos do vetor de entrada pelo número de processadores disponíveis na máquina em que a aplicação está sendo executada.

Para compilar os códigos que utilizam a biblioteca Muesli devem ter instalados a biblioteca MPI e OpenMP. O compilador mpiCC (mpicxx, mpic++) deve ser utilizado.

2.5.2 SkePU

SkePU é uma biblioteca de esqueletos implementada na linguagem C++ para o desenvolvimento de aplicações que utilizam CPUs *multicores* e GPUs (*Graphics Processing Unit*) (DASTGEER; LI; KESSLER, 2013). Os esqueletos para CPUs são implementados utilizando a plataforma OpenMP e para as 9GPUs usam CUDA e OpenCL. O programador pode controlar qual plataforma que a aplicação irá utilizar por meio das macros: SKEPU_CUDA, SKEPU_OPENCL ou SKEPU_OPENMP no código fonte. Outras modificações no código não são necessárias, pois a interface é a mesma para todas as plataformas (ENMYREN, 2010). Este *framework* está na versão 1.0.

O SkePU tem vetores e matrizes baseados na STL, que são utilizados pelos esqueletos para esconderem o gerenciamento de memória da GPU. Além disso, utilizam a técnica de *lazy memory* que copia dados da memória que realmente são necessários, evitando assim operações de memória desnecessárias. Um exemplo de vetor e de matriz:

```
1 skepu:: Vector<double> vetor(100, 10);
2 skepu:: Matrix<double> matriz(50, 50, 10);
```

Para o vetor é alocado um espaço de memória de 100 elementos, todos inicializados com valor 10 e para a matriz é alocado um espaço de memória de 50 linhas por 50 colunas, sendo todos as posições da matriz inicializadas com o número 10.

SkePU implementa os esqueletos Map, Reduce, MapReduce, MapOverlap, MapArray e Scan. Todos eles são para o paralelismo de dados e são implementados para as plataformas OpenMP, OpenCL e CUDA.

Os programadores devem definir suas funções para os esqueletos utilizando macros do SkePU mostradas na Figura 2, estas macros variam no número de parâmetros como UNARY_FUNC, BINARY_FUNC e TERNARY_FUNC. Na Fi-

gura 3 esta a declaração da macro BINARY_FUNC que tem como parâmetros o *name* para o nome da função do usuário, o *type1* para o tipo de dado que a função deve tratar, os dois parâmetros da função são passados por *param1* e *param2* e o *func* é código que a função irá executar. Os parâmetros da macro são utilizados para implementar uma *struct* que contém funções para serem executadas em cada tipo de plataforma. Para as plataformas CUDA e OpenMP são simples funções, porém para OpenCL devem ser definidas por meio de strings como é visto na Figura 3.

```
1 UNARY_FUNC( name, type1, param1, func)
2 UNARY_FUNC_CONSTANT( name, type1, param1, const1, func)
3 BINARY_FUNC( name, type1, param1, param2, func)
4 BINARY_FUNC_CONSTANT( name, type1 , param1 , param2 , const1, func)
5 TERNARY_FUNC( name, type1 , param1 , param2 , param3 , func)
6 TERNARY_FUNC_CONSTANT( name, type1, param1, param2, param3, const1, func)
7 OVERLAP_FUNC( name, type1, over, param1, func)
8 ARRAY_FUNC( name, type1, param1, param2, func)
```

Figura 2: Lista de macros disponíveis no *framework* SkePU (ENMYREN, 2010). Macros que permitem uma variação nos números de parâmeros das funções que cada esqueleto recebe.

Os esqueletos do SkePU implementados em C++ são representados como objetos. Quando o operador é sobrecarregado eles podem se comportar como funções comuns. Todos os esqueletos contêm funções membro representando as diferentes implementações de CUDA, OpenCL e OpenMP. As funções membro são chamadas de CU, CL e OMP, respectivamente. Assim, quando o esqueleto é chamado com o operador, a biblioteca decide qual função membro irá utilizar, dependendo de qual estiver disponível. No caso do OpenCL, o objeto esqueleto também contém o código necessário para geração e compilação de procedimentos. Quando um esqueleto é instanciado o SkePU cria um ambiente que disponibiliza o OpenCL ou CUDA, esse ambiente é compartilhado entre todos os esqueletos.

Para compilar os códigos que utilizam a biblioteca SkePU para o OpenMP deve usar a seguinte linha de comando:

g++ <filename>.cpp -I<path-to-skepu-include-folder>-o <filename>-DSKEPU_OPENMP -fopenmp

Para compilar para a plataforma OpenCL, deve-se alterar a diretivas para -DSKEPU_OPENCL e -lOpenCL e incluir os arquivos com os códigos fonte destinados para OpenCL da SkePU. De forma semelhante a CUDA que deve utilizar as diretivas -DSKEPU_CUDA -Xcompiler.

```
1  #define BINARY_FUNC(name, type1, param1, param2, func)
2
3  struct name{
4   typedef type1 TYPE;
5   bool isConst;
6   skepu::FuncType funcType;
7   name(){
8    funcType = skepu::BINARY;
9    isConst = false;
10  }
11  type1 dummy;
12  type1 getConstant(){ return dummy; }
13  type1 CPU(type1 param1, type1 param2){
14    return CPU(param1, param2, dummy);
15  }
16  type1 CPU(type1 param1, type1 param2, type1 dummy){
17   func
18  }
19  __device__ type1 CU(type1 param1, type1 param2){
20   func
21  }
22  __device__ type1 CU(type1 param1, type1 param2, type1 dummy){
23   func
24  }
25 };
```

Figura 3: Ilustração da implementação da macro BINARY_FUNC do SkePU. Fonte: (ENMYREN, 2010). Nesta ilustração é demonstrado como as funções são implementadas para cada plataforma de execução.

2.6 Conclusão

O conceito de algoritmos de esqueletos foi definido por Murray Cole em 1989 e desde então novos esqueletos vem sendo criados e implementados. Nos últimos anos este autor desenvolveu conceitos que devem ser considerados para que os esqueletos sejam desenvolvidos de maneira eficiente.

Os esqueletos têm sua origem nas linguagens funcionais mas eles estão sendo implementados em outros tipos de linguagens, portanto eles devem ser implementados de forma que mantenham as suas principais características.

Neste capítulo foram ainda apresentadas características das duas ferramentas que oferecem esqueletos para paralelismo: Muesli e SkePU. Estas ferramentas possuem características semelhantes àquelas incorporadas na solução proposta: possuem como linguagem base C++ e se apoiam em *templates* para implementação dos esqueletos. No próximo capítulo são descritos os esqueletos que foram selecionados para integrar a API desenvolvida neste trabalho.

3 DESCRIÇÃO DOS ESQUELETOS IMPLEMENTADOS

Os esqueletos descritos neste capítulo foram divididos de acordo com a classificação: Paralelismo de Dados, Paralelismo de Tarefas e Resolução de Problemas. São oito os esqueletos considerados, sendo que seis deles pertencem à categoria de Paralelismo de Dados. Estes esqueletos foram observados como os mais recorrentes nas ferramentas estudadas no capítulo anterior e foram, desta forma, selecionados para compor a solução proposta de uma API de esqueletos paralelos para Anahy.

Os esqueletos são descritos na sequência em função de suas características operacionais, sendo a discussão de cada um deles complementada pela modelagem de seu comportamento de execução paralela.

3.1 Esqueletos de Paralelismo de Dados

Os esqueletos que descrevem o paralelismo de dados são voltados para problemas da camada de aplicação, nos quais a concorrência se encontra na aplicação de uma mesma operação a um determinado conjunto de dados. Os esqueletos desta classificação são: Map, Fork, Zip, Reduce, Scan e MapReduce. A forma como a operação é aplicada caracteriza cada um dos esqueletos.

3.1.1 Esqueleto Map

O esqueleto Map segue a mesma semântica da função polimórfica map presente nas linguagens de programação funcionais. Seus parâmetros de entrada são uma função F(a), um vetor de entrada e[] e uma referência para o vetor de saída s[], o qual possui o mesmo tamanho do vetor de entrada. A função F recebe como parâmetro somente um valor, do mesmo tipo dos elementos do vetor de entrada, e retorna um elemento do tipo do vetor de saída. O protótipo da função map é ilustrado a seguir:

Map(F(a), e[], s[])

A versão sequencial do map percorre todo o vetor de entrada passando um elemento por vez como parâmetro para função F, esta função é computada e retorna um valor para a mesma posição no vetor de saída.

O esqueleto Map pode ser paralelizado de modo que a aplicação da função F sobre cada elemento do vetor de entrada será executada simultaneamente por um conjunto de *threads*, assim o número de *threads* criados será igual ao número de elementos do vetor de entrada. A versão paralela da função map pode ser observada na Figura 4 que demonstra a função F sendo aplicada simultaneamente a cada posição do vetor de entrada e[] por meio de um conjunto de *threads* composto por 4 *threads*. Após o término da execução de todos os *threads* os valores de retorno são armazenados no vetor de saída s[]. Logo o esqueleto Map é classificado como um paralelismo do tipo SIMD (*Single Instruction Multiple Data*) (VÉLEZ; LEYTON, 2010).

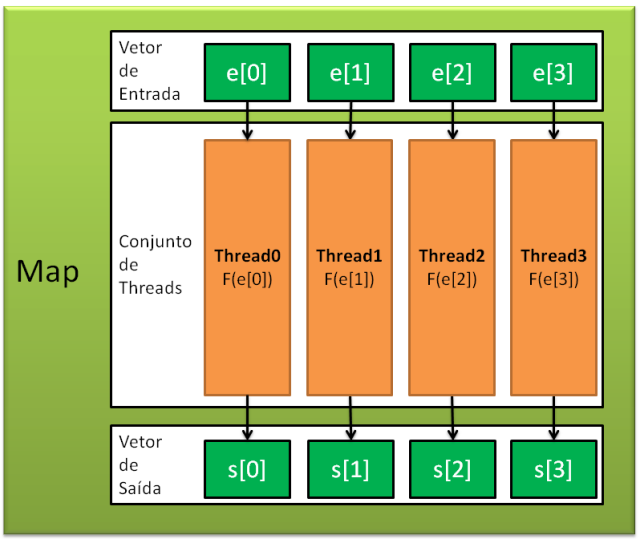


Figura 4: Esquema de execução do esqueleto Map paralelo.

3.1.2 Esqueleto Fork

O esqueleto Fork opera de forma semelhante ao esqueleto Map, mas ao invés de aplicar a mesma função em todos os elementos do vetor de entrada, existe uma função diferente para cada elemento do vetor de entrada. O protótipo do esqueleto Fork pode ser exemplificado da seguinte forma:

Fork(F[], e[], s[])

O esqueleto Fork tem como parâmetros um vetor de funções F[], todas as funções devem possuir somente um parâmetro, e um vetor de dados de entrada e[], sendo que estes dois vetores tem o mesmo tamanho. O Fork também recebe

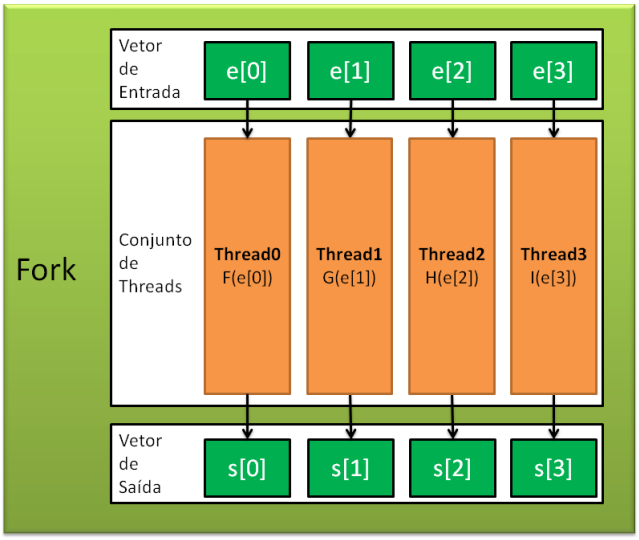


Figura 5: Esquema de execução do esqueleto Fork paralelo.

um ponteiro para o vetor s[] de saída, este vetor deve ter o mesmo tamanho dos vetores de entrada.

Sua implementação paralela consiste que cada um dos *threads* execute uma função do vetor de funções sobre um elemento do vetor de entrada, o número de *threads* será igual ao tamanho dos vetores de entrada. A Figura 5 mostra que cada *thread* executa uma função diferente sobre uma posição do vetor de entrada. Após a execução dos *threads* os valores retornados são armazenados no vetor de saída.

O esqueleto Fork implementa um paralelismo do tipo MIMD (*Multiple Instruction Multiple Data*) (VÉLEZ; LEYTON, 2010).

3.1.3 Esqueleto Zip

O esqueleto Zip é outra variação do esqueleto Map, pois recebe como parâmetro uma função e dois vetores e retorna um vetor, isto é, o Zip aplica a função informada sobre os vetores de entrada para realizar uma operação de união dos elementos destes vetores (MCCOOL; ROBISON; REINDERS, 2012). O protótipo da função deste esqueleto é ilustrado a seguir:

Zip(F(a,b), e0[], e1[], s[])

Zip tem como parâmetro uma função F(a,b) que recebe dois elementos a e b do mesmo tipo de dado dos vetores, dois vetores de dados de entrada e0[] e e1[], ambos são do mesmo tipo, e um ponteiro para o vetor de saída. Os vetores de entrada e saída devem ter o mesmo número de elementos. Quando ocorre a invocação da função F, os parâmetros a e b representarão, respectivamente,

elementos na mesma posição relativa dos vetores de entrada. O retorno de F é o resultado da operação Zip sobre estes dois valores.

A Figura 6 ilustra a execução paralela do Zip que cria um conjunto de *threads* que executam cada aplicação da função de entrada em cada par de elementos, desta forma o número de *threads* criados é igual a número de elementos dos vetores de entrada. Os valores retornados por cada *thread* são armazenados no vetor de saída.

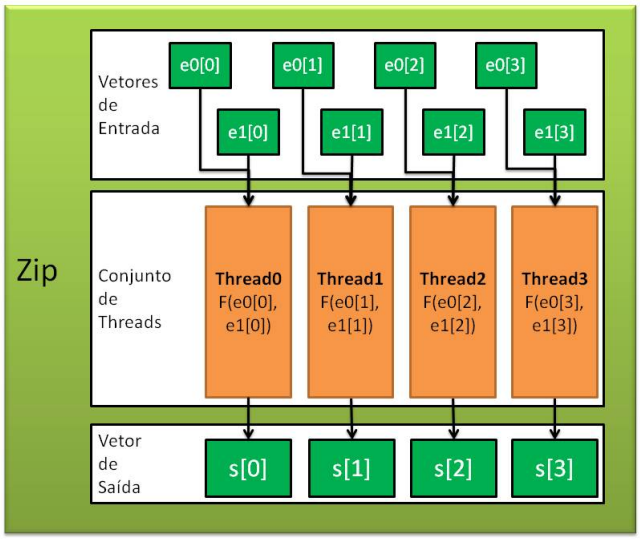


Figura 6: Esquema de execução do esqueleto Zip paralelo.

3.1.4 Esqueleto Reduce

O esqueleto Reduce também tem sua origem nas linguagens funcionais e seus parâmetros são uma função F(a,b) com dois parâmetros e um vetor de entrada e retorna somente um valor de saída. O protótipo do esqueleto Reduce é ilustrado a seguir:

Reduce(F(a,b), e[], s)

O Reduce consiste em aplicar recursivamente a função F aos pares de elementos do vetor de entrada até que o vetor seja reduzido a somente um elemento (MCCOOL, 2010). Na implementação paralela, cada chamada recursiva da função F é realizada por um *thread*, assim são gerados níveis de *threads* que retornam valores intermediários até que um *thread* execute a última chamada da função F. A Figura 7 ilustra este procedimento, no qual *Thread0* executa a função F sobre os dois primeiros elementos do vetor de entrada e o *Thread1* executa a função F sobre os outros dois elementos do vetor de entrada. Estes dois *threads* retornaram dois valores intermediários, nesta ilustração são denominados de a

e b. Logo, para reduzir a e b em um único valor é necessário criar o *Thread2* para aplicar a função F sobre eles. Por fim, após o término deste último *thread*, o valor de saída é retornado.

A operação de redução deve ter a semântica do operador +=, ou seja, a ordem de avaliação não deve influenciar no resultado.

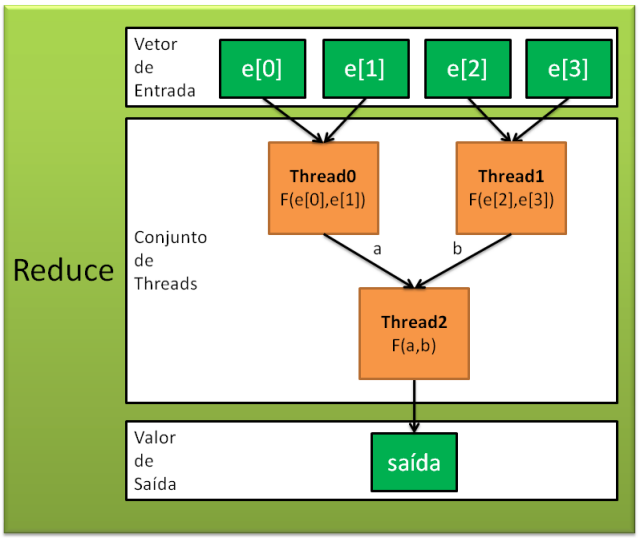


Figura 7: Esquema de execução do esqueleto Reduce paralelo.

3.1.5 Esqueleto Scan

O Scan é semelhante ao Reduce mas em vez de produzir um resultado escalar ele produz um vetor de saída do mesmo tamanho do vetor entrada (MCCOOL, 2010). O Scan possui os seguintes parâmetros: uma função F(a,b) com dois parâmetros, o vetor de entrada e[] e um ponteiro para o vetor de saída s[], como é ilustrado a seguir:

Scan(F(a,b), e[], s[])

Na versão sequencial do esqueleto Scan o primeiro elemento do vetor de entrada é copiado para a mesma posição no vetor de saída. Depois disto, este esqueleto começa a percorrer os vetores de entrada e saída aplicando a função F sobre o elemento na posição n-1 do vetor de saída com o elemento n do vetor de entrada, o retorno da função F é armazenado na posição n do vetor de saída.

O primeiro elemento do vetor de entrada também é copiado para a primeira posição do vetor de saída na versão paralela do Scan. Na primeira etapa da execução do Scan cada *thread* irá computar a função F sobre dois elementos do vetor de entrada que estão nas posições n e n+1. Porém, somente será possível obter os dois primeiros elementos do vetor de saída como é observado

na Figura 8. Ao final da primeira etapa, um vetor com valores intermediários será encontrado e para finalizar a execução do Scan será necessário aplicar novamente a função F nas posições n e n+2 neste vetor intermediário por meio de duas *threads*. No final da execução do Scan os valores retornados serão armazenados nas duas últimas posições do vetor de saída.

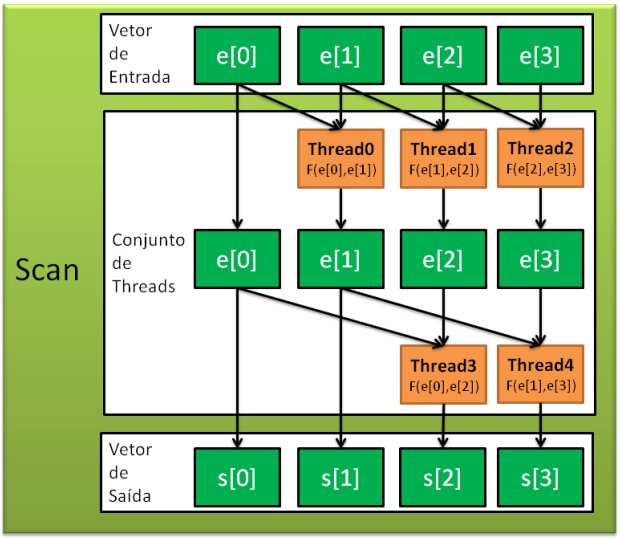


Figura 8: Esquema de execução do esqueleto Scan paralelo.

3.1.6 Esqueleto MapReduce

O esqueleto MapReduce consiste na união dos esqueletos Map e Reduce, ou seja, o MapReduce é dividido em duas etapas. Inicialmente, o esqueleto Map é executado para aplicar sua função a cada elemento do vetor de entrada. A segunda etapa o Reduce recebe o vetor de saída do Map e o reduz a um valor escalar. A execução do MapReduce esta ilustrado na Figura 9.

O protótipo do MapReduce é o seguinte:

MapReduce(FM(), FR(), e[], s)

O parâmetro FM() é a função que o map aplica sobre o vetor de entrada, a função FR() é a função que o Reduce utiliza para reduzir o vetor de saída do Map ao um valor escalar e o parâmetro s é o valor de saída do MapReduce.

3.2 Esqueletos de Resolução de Problemas

A denominação de esqueletos de resolução de problemas reflete a natureza dos esqueletos, a qual descrevem inteiramente um comportamento algorítmico de determinadas classes de problemas. Nesta seção é apresentado o esqueleto Divisão e Conquista.

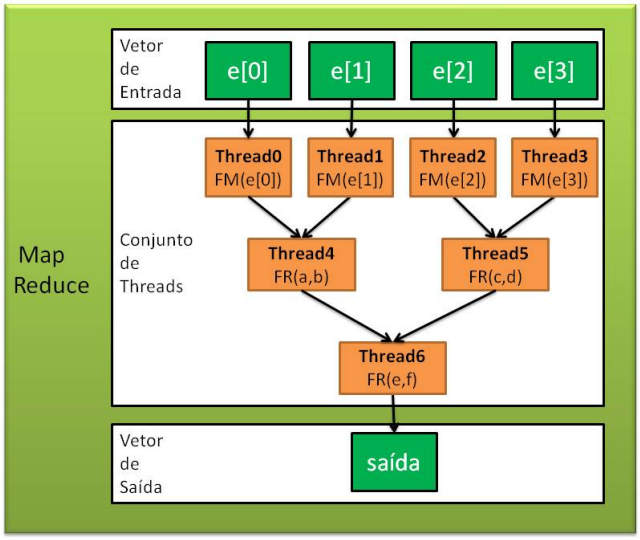


Figura 9: Esquema de execução do esqueleto MapReduce paralelo.

3.2.1 Divisão e Conquista (DC)

O esqueleto DC implementa a técnica de Divisão e Conquista que consiste em dividir um grande problema com solução complexa em vários pequenos problemas de soluções mais simples ou fatoradas. O DC testa se o problema é divisível, caso seja, divide recursivamente o problema em subproblemas. Estes subproblemas são divididos até que se tornem indivisíveis, ou até que, segundo alguma heurística de custo, novas divisões não compensem, sendo então solucionados sequencialmente, gerando soluções parciais. As soluções parciais são combinadas recursivamente, em ordem inversa a divisão do trabalho, até formarem a solução do problema de entrada.

O DC tem como parâmetro a função divisão() que divide o problema em subproblemas, uma função conquista() que encontra a solução dos subproblemas, a função combina() que realiza a união das subsoluções, uma função base() que testa se o problema é indivisível e uma referência para a solução encontrada do problema de entrada. O protótipo do DC pode ser visto a seguir:

DC(problema, divide(), conquista(), combina(), base(), solucao)

Na Figura 10 é ilustrado a execução do DC paralelo. O problema a ser solucionado é testado se é divisível, caso seja, dois *threads* são criados e cada um receberá um dos subproblemas. Cada *thread* executa a função base para testar se os subproblemas são divisíveis, como neste caso são divisíveis, eles são divididos para gerar quatro subproblemas e estes dois *thteads* geram outros quatro *threads* no nível de recursão posterior. Novamente, a função base é executada,

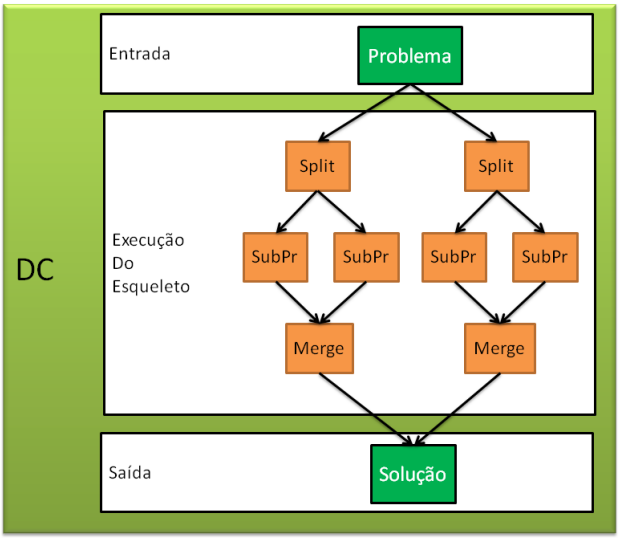


Figura 10: Esquema de execução do esqueleto DC paralelo.

verificando assim que os quatro subproblemas não são divisíveis, logo cada *thread* compondo um ponto final da recursão invoca a função de conquista para gerar quatro soluções parciais. Estas soluções parciais são retornadas para os *threads* no nível anterior da recursão. Neste ponto, cada *thread* invoca a função combinar para gerar duas soluções parciais que combinam as soluções parciais recebidas em retorno dos *threads* já terminados. O ciclo se repete até retornar a raiz da recursão e a combinação de resultados parciais formar a solução completa do problema.

3.3 Esqueletos de Paralelismo de Tarefas

Outro conjunto de esqueletos, denominado de Paralelismo de Tarefas, permite identificar uma determinada ordem de execução de operações em função de uma sequência ou condição pré-determinada, podendo também identificar iterações condicionais. Deste conjunto o esqueleto Farm foi escolhido para ser implementado.

3.3.1 Esqueleto Farm

O esqueleto Farm implementa a estratégia *master-slave*/*worker* ou *bag-of-tasks*. Ele incorpora a capacidade de agendar tarefas independentes nos núcleos do processador. Em termos de *threads*, uma sequência de dados que estão sobre o fluxo de entrada de um Farm são submetidos a um conjunto de *threads*. Cada *thread* aplica a mesma função nos itens de dados recebidos e o resultado é enviado para o fluxo de saída.

O protótipo do esqueleto Farm pode ser exemplificado da seguinte forma:

Farm(F(), entrada, saída)

O esqueleto Farm tem como parâmetro uma função F, os dados de entrada e uma referência para os dados de saída. Na Figura 11 é ilustrada um esquema de execução do esqueleto Farm que utiliza um *thread* denominado de *farmer* para controlar os três *threads* denominados de *workers* que aplicam a mesma função nos dados de entrada. Quando os *workers* terminam a execução das suas funções eles retornam os resultados para o processo *farmer* que envia os dados para o fluxo de saída.

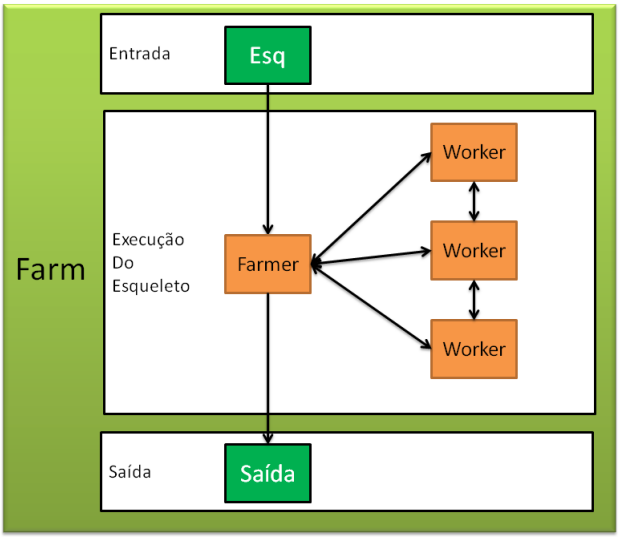


Figura 11: Esquema de execução do esqueleto Farm.

3.4 Conclusão

Neste capítulo foram descritos os esqueletos selecionados para integrarem a API desenvolvida neste trabalho. Para cada um destes esqueletos foi descrita um esboço do comportamento previsto para sua execução paralela. Estes esqueletos foram descritos como funções de alta ordem, porém como este trabalho utiliza a linguagem orientada a objetos C++ os esqueletos são implementados utilizando classes. Assim, no próximo capítulo está descrita a implementação da API.

4 INTERFACE PROPOSTA

A API desenvolvida neste trabalho é descrita neste capítulo juntamente com as classes que a compõem. Esta API foi implementada seguindo o modelo de programação orientada a objetos utilizando a linguagem de programação C++. A ferramenta construída foi denominada Kanga[1].

O capítulo inicia apresentando o modelo do objeto que representa uma instância de um esqueleto. Na sequência é descrito o diagrama de classes das classes implementadas, sendo seguido de um detalhamento de cada uma destas classes. O capítulo termina discutindo a composição de esqueletos na ferramenta proposta.

4.1 Modelo de Execução

O modelo de execução do trabalho proposto define que os esqueletos são objetos, descritos por uma classe. Cada um possui um construtor, que recebe os parâmetros de inicialização, como a função a ser aplicada e os dados de entrada.

Todos os esqueletos possuem um método de ativação, o *start*, responsável por ativar o paralelismo. A implementação deste método é proposta na API, decompondo o problema conforme discutido no Capítulo 3 utilizando os recursos de descrição de concorrência disponibilizados pelas plataformas de execução adotada, Anahy e Pthreads por exemplo, que serviram para validar a implementação. Além do método *start*, tem o método *join* que garante a sincronização com o término de todas atividades geradas pelo esqueleto.

4.2 Diagrama de Classes

O diagrama de classes da Figura 12 representa como as classes dos esqueletos estão estruturadas. Essa figura mostra que todas as classes herdam a definição dos métodos da classe abstrata Kanga, esta classe define os métodos virtuais (ou abstratos): start, join, setDataIn e getDataOut, logo estes métodos

[1] Kanga, em tupi-guarani, significa ”esqueleto”.

devem ser implementados por todos os esqueletos. A classe Kanga age como uma interface entre aplicação do programador e os esqueletos, ou seja, todos os esqueletos implementam os mesmos métodos, o que facilita a comunicação e permite a composição entre os esqueletos.

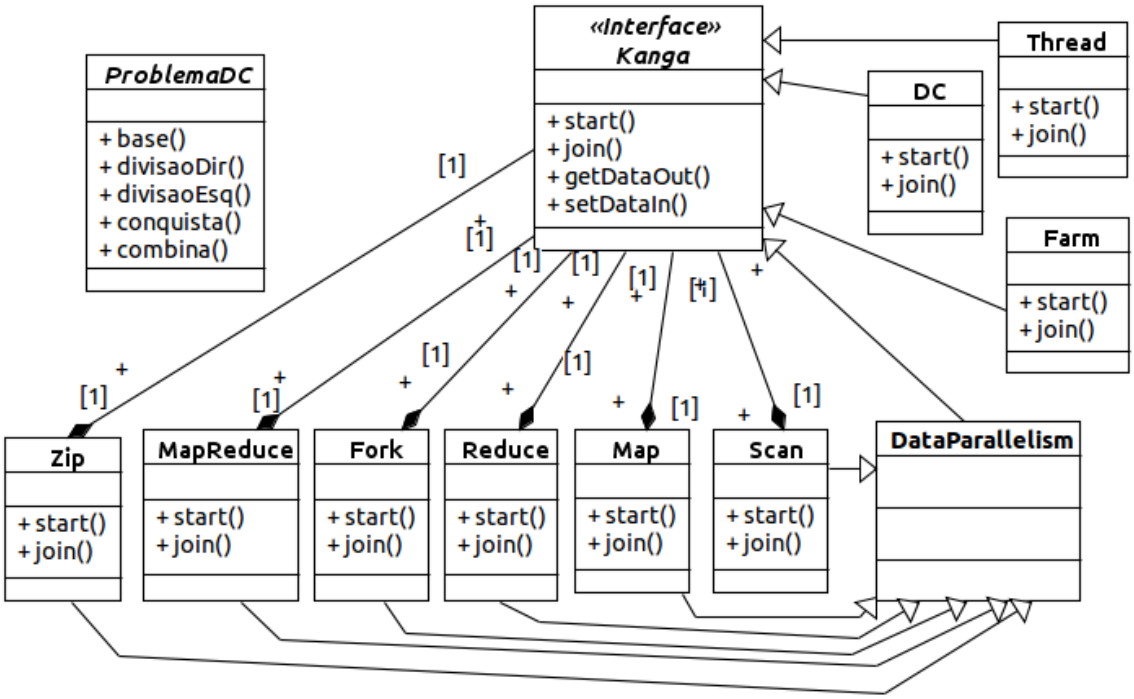


Figura 12: Diagrama de classes da interface desenvolvida para o *framework* de esqueletos Kanga.

A maioria dos esqueletos implementados são destinados para o paralelismo de dados, logo a classe DataParallelism foi modelada para armazenar as variáveis comuns a todos os esqueletos de paralelismo de dados. Desta forma, foi possível desenvolver códigos enxutos e de fácil compreensão nas classes que concretizam cada esqueleto. A classe DataParallelism também herda os métodos de Kanga, porém somente implementa os métodos setDataIn e getDataOut. De acordo com o diagrama da Figura 12 os esqueletos Map, Reduce, Scan, Fork, Zip e MapReduce herdam as variáveis da classe DataParallelism e implementam os métodos start e join.

O projeto desta interface considerou a modelagem de uma classe que abstrai o conceito de *thread*. Assim, somente a classe Thread terá alguma interação com os ambientes de execução, descritos na Capítulo 5, o que permite facilitar a portabilidade entre estes ambientes. A classe Thread também herda os métodos da classe abstrata Kanga de acordo com o diagrama de classe da Figura 12.

4.3 Utilizando a Interface

Nesta seção é apresentado um exemplo de código fonte que utiliza o esqueleto Map. Este exemplo consiste em aplicar paralelamente a função func da linha 4 da Figura 13 sobre o vetor vetIn de 10 elementos. O objeto do esqueleto Map é criado chamando o seu construtor na linha 10. Como parâmetro de construção é informado o ponteiro para a função func, para o vetor de entrada vetIn, o ponteiro para o vetor de saída vetOut e o número de elementos do vetor de entrada e de saída. O início da execução do Map inicia a partir da chamada do método start(), este método cria um conjunto de *threads*, um para cada elemento do vetor de entrada, e passa para cada *thread* o ponteiro para a função func e um elemento do vetor de entrada. A sincronização com os *threads* do Map será realizada após a chamada do método join(), este método executa a operação join em cada *thread* e salva cada valor de retorno no vetor de saída, o espaço de memória do vetor de saída é alocada pelo Map.

```
1  #define ANAHY
2  #include "Map.h"
3
4  int* func(int* n){ *n=*n+2; return n; }
5
6  int main(int argc, char **argv){
7   AnahyVM::init(argc, argv);
8   int vetIn[10], *vetOut;
9   for(int i=0;i <10;i++) vetIn[i]=i+1;
10  kanga::Map<int,int> *my = new kanga::Map<int,int>(func,vetIn,vetOut,10);
11  my->start();
12  my->join();
13  AnahyVM::terminate();
14  return 0;
15 }
```

Figura 13: Exemplo de código utilizando o esqueleto Map.

Para compilar e executar deve utilizar este código as seguintes linhas de comando no terminal Linux:

g++ <filename>.cpp -o <filename><path-to-Anahy-include-folder>-lpthread

4.4 Funções do Programador

O código definido pelo programador nas funções dos esqueletos deve se restringir a manipular os dados passados por parâmetro, pois a função do programador não deve acessar dados compartilhados e nem executar operações bloqueantes durante a sua execução. Isto se deve a implementação da interface não prever o uso de operações bloqueantes como *mutex* e semáforos pelas funções do usuário. Além disso, o ambiente Anahy3 não oferece estas

operações, caso alguma destas operações sejam usadas a estratégia de escalonamento de Anahy3 será afetada, podendo ocorrer *deadlocks*.

As funções são nativas da linguagem C/C++, como pode ser visto a seguir:

```
1 int* func(int* n){
2  int *aux = new int;
3  *aux=*n;
4  return aux;
5 }
```

Os tipos dos parâmetros e de retorno das funções são definidos pelo *template* do esqueleto a qual pertence esta função. Contudo, estes tipos podem ser diferentes caso o *template* tenha dois parâmetros, o que permite que o programador retorne um dado com tipo diferente daquele recebido pelo parâmetro. Os dados resultantes das instruções contidas na função devem ser retornados utilizando a instrução *return*.

4.5 Implementação da Interface

A implementação da API contou com importantes mecanismos da linguagem C++. Entre eles o espaço de nomes ou *namespace*, que foi utilizado para evitar conflitos dos nomes desta API com nomes de outras bibliotecas. Por exemplo, evitará conflitos do nome da classe Thread com o mesmo identificador de *threads* utilizados por outras ferramentas que poderão ser utilizadas em conjunto com a API apresentada. O *namespace* desta API é denominado de kanga.

Para que a API seja independente de tipos de dados, ou seja, uma interface genérica, foram utilizadas classes *templates*. Uma classe *template* implementa o conceito de uma classe genérica, ou seja, uma classe que pode ser instanciada para mais de um tipo de dado, mas que mantém a mesma estrutura de execução. No momento da criação do objeto o tipo de dado deve ser passado por meio dos parâmetros do *template*.

Os dados de entrada das classes dos esqueletos e da classe Thread são passados pelos métodos construtores. O objetivo dos métodos construtores é inicializar os atributos dos objetos, tendo sido previstos os seguintes construtores para os esqueletos implementados:

- Construtor *default* ou vazio: é um método sem parâmetros. Este construtor pode ser utilizado para inicializar os atributos da classe no momento de criação do objeto. Caso o programador não defina um construtor *default* o compilador C++ criará um, porém se existirem atributos do tipo *const* e referências, o compilador não criará um construtor vazio, logo o programador é obrigado a defini-lo. Neste trabalho foram implementados construtores

default para inicializar referências de vetores de dados, de objetos e de lista de *threads*.

- Construtor de cópia: é utilizado para criar uma cópia de um objeto existente. O construtor de cópia recebe como parâmetro um objeto da própria classe. O compilador gera este método caso o programador não tenho definido. Este construtor é invocado nas seguintes situações:

  – Quando um objeto é passado para um método por valor e/ou retornado por valor;

  – Quando um objeto é inicializado utilizando o operador de atribuição (=);

  – Quando um objeto é lançado no tratamento de exceções pelo *throw* e *catch*.

  A implementação realizada faz a cópia profunda, o programador deve estar atento para evitar cópias superficiais. O novo objeto passa a ter uma estrutura idêntica ao objeto original, mas trata-se de um novo objeto que não compartilham dados.

- Construtor com parâmetros: é por meio deste tipo de construtor que os atributos são inicializados. Este construtor pode ser sobrecarregado, ou seja, pode existir na classe vários métodos construtores com assinaturas diferentes. Neste trabalho os métodos construtores deste tipo são utilizados para inicializar os atributos dos esqueletos.

Além dos métodos construtores foram utilizados os destrutores, estes métodos tem o objetivo de finalizar e liberar a memória alocada no construtor e destruir objetos dinâmicos. Esta observação é relevante quando se considera que determinadas ferramentas de programação paralela possuem primitivas de abandono de execução de *thread* como é o caso da primitiva pthread_exit em Pthreads, que não destrói propriamente o registro da pilha da função executada por um *thread*.

A API desenvolvida utiliza como plataforma de suporte as ferramentas Anahy e Pthreads, assim foram utilizadas diretivas de pré-compilação para que o programador possa compilar o seu código para somente uma dessas plataformas. Com isso, o programador deve definir a diretiva *#define ANAHY* para utilizar o Anahy3 ou *#define PTHREADS* para usar Pthreads. Nos dois casos estas diretivas devem estar antes de incluir as classes de Kanga.

4.5.1 Classe Kanga

A classe Kanga é a classe abstrata que contêm a declaração dos métodos utilizados por todas as classes da API desenvolvida. Nesta classe todos os métodos são abstratos e nenhum tem alguma declaração, ou seja, devem ser implementados pelas classes derivadas. Na Figura 14 está o código da classe Kanga com os métodos virtuais.

```
1  namespace kanga{
2  template<class IN, class OUT>
3  class Kanga{
4   virtual void start();
5   virtual void join();
6   virtual void setDataIn(IN *data);
7   virtual void setDataIn(IN* dataIn, IN * dataIn1);
8   virtual void setDataOut(OUT*& data);
9   virtual OUT *getDataOut();
10  virtual ~Kanga();
11 };
12 };
```

Figura 14: Declaração da classe Kanga.

4.5.2 Classe Thread

Um objeto da classe Thread da Figura 15, cria um *thread* conforme definido pelo ambiente de programação paralela utilizado. O método start() inicializa e dispara um *thread*, isso ocorre com a utilização do Anahy por meio da diretiva fork() ou pelo Pthreads por meio da diretiva pthread_create(). O método join() sincroniza o *thread* utilizando join() e pthread_join() do Anahy e Pthreads, respectivamente.

Os construtores de classe da Figura 15 têm como parâmetros funções com o protótipo OUT*(*f)(IN*) ou com o protótipo OUT*(*f)(IN*,IN*). Esta variação de funções se deve que alguns esqueletos utilizam funções com um parâmetro e outros utilizam com dois parâmetros.

A classe Thread tem um *template* com os parâmetros IN para os tipos do parâmetros de entrada e OUT para os tipos de saída. Além disso, implementa os métodos para de setar os parâmetros de entrada e obter os dados de saída, tais como: setFunction, setDataIn e getDataOut.

4.5.3 Classe DataParallelism

A classe DataParallelism é a classe que herda os métodos da classe Kanga mas somente implementa setDataIn() e getDataOut(). A Figura 16 monstra a classe DataParallelism, onde estão os construtores que são utilizados para inici-

```
1  template<class IN, class OUT>
2  class Thread : public Kanga<IN,OUT>{
3   Thread();
4   Thread(const Thread<IN,OUT>& e);
5   Thread(OUT*(*f)(IN*));
6   Thread(OUT*(*f)(IN*,IN*));
7   Thread(OUT*(*f)(IN*), IN *dataIn, OUT *&dataOut);
8   Thread(OUT*(*f)(IN*,IN*), IN *dataIn1, IN *dataIn2, OUT *&dataOut);
9   void start();
10  void join();
11  void setFunction(OUT*(*f)(IN*));
12  void setFunction(OUT*(*f)(IN*,IN*));
13  void setDataIn(IN* data);
14  void setDataIn(IN* data1,IN* data2);
15  OUT * getDataOut();
16  virtual ~Thread();
17 };
```

Figura 15: Declaração da classe Thread.

alizar as variáveis utilizadas por todos os esqueletos de paralelismo de dados.

O construtor que recebe como parâmetro o ponteiro para o vetor de entrada, o ponteiro para o vetor de saída e o número de elementos desses vetores são utilizados por todos os esqueletos de paralelismo de dados. Mesmo os esqueletos Reduce e MapReduce utilizam este construtor, pois recebem um ponteiro para o vetor de entrada e um ponteiro para o dado de retorno.

```
1  template< class IN, class OUT>
2  class DataParallelism : public Kanga<IN,OUT> {
3   DataParallelism();
4   DataParallelism(const DataParallelism<IN,OUT> &e);
5   DataParallelism(IN * listIn, OUT *& listOut, int length);
6   DataParallelism(IN ** matIn, OUT *& listOut, int length);
7   DataParallelism(IN ** matIn, OUT **& matOut, int length);
8   DataParallelism(int length);
9   void setDataIn(IN *listIn);
10  OUT * geDataOut();
11  virtual ~DataParallelism();
12 };
```

Figura 16: Declaração da classe DataParallelism.

4.5.4 Classe Map

O esqueleto Map é implementado pela classe da Figura 17. Nesta figura esta ilustrado o *template* que o Map utiliza com dois parâmetros: o primeiro IN representa o tipo dos dados de entrada e o segundo OUT representa o tipo dos dados de saída. Desta forma, a função do programador que o Map recebe como parâmetro tem o seguinte protótipo: OUT* (*f)(IN*).

```
1  template<class IN, class OUT>
2  class Map : public DataParallelism<IN,OUT>{
3   Map(): DataParallelism<IN,OUT>();
4   Map(const Map &e): DataParallelism<IN,OUT>(e);
5   Map(OUT*(*f)(IN*), IN *listIn, OUT *&listOut, int length): DataParallelism<IN
        ,OUT>(listIn,listOut,length);
6   Map(OUT*(*f)(IN*), int length): DataParallelism<IN,OUT>(length);
7   Map(Kanga<IN,OUT> *obj, IN **matIn, OUT **&matOut, int length):
        DataParallelism<IN,OUT>(matIn,matOut,length);
8   void start();
9   void join();
10  void setDataIn(IN* listIn);
11  OUT* getDataOut();
12  void setFunction( OUT*(*f)(IN*) );
13  virtual ~Map();
14 };
```

Figura 17: Declaração da classe Map.

A execução do Map inicia por meio da invocação do método start(), este método cria um objeto da classe Thread para cada elemento do vetor de entrada, isto é, uma lista de *threads*. Cada *thread* desta lista recebe como parâmetro o ponteiro da função do Map e um elemento da vetor de entrada. Após isto, o método start() de cada *thread* é invocado, iniciando assim a execução do esqueleto Map.

Quando o método join() do Map for chamado a sincronização da lista de *threads* será realizada por meio da chamada do método join() de cada objeto *thread*. Após a chamada deste método o programador poderá acessar os dados na lista de saída.

4.5.5 Classe Reduce

O esqueleto Reduce aplica recursivamente a função do programador sobre os dados de entrada até que sejam reduzidos a um valor escalar. Devido a isto, o seu *template* contém apenas um parâmetro, sendo que o tipo do dado de retorno é igual ao do parâmetro de entrada. A função de entrada possui o seguinte protótipo: T*(*f)(T*).

O construtor do Reduce tem como parâmetro um ponteiro para indicar o vetor de entrada e um ponteiro para indicar o valor de retorno. Este valor também pode ser obtido pelo método getDataOut() que retorna um ponteiro pra este valor. Contudo, este método deve ser chamado depois do join() do Reduce.

O método start() do Reduce cria um único *thread* que executa o método privado run(). O run() realizar a aplicação da função do usuário sobre o vetor de entrada. Quando o join() do Reduce é invocado, o objeto *thread* que executa o run() é sincronizado por meio do seu join().

```
1  template<class T>
2  class Reduce : public Kanga<T,T>{
3   Reduce(): DataParallelism<T,T>();
4   Reduce(const Reduce &e): DataParallelism<T,T>(e);
5   Reduce(T*(*f)(T*,T*), T *listIn, T *&dataOut, int length): DataParallelism<T,
        T>(listIn,dataOut,length);
6   Reduce(T*(*f)(T*,T*), int length): DataParallelism<T,T>(length);
7   Reduce(Kanga<T,T> *obj, T **matIn,T *&listOut, int length): DataParallelism<T
        ,T>(matIn,listOut,length);
8   void start();
9   void join();
10  void setDataIn(T *listIn);
11  void setFunction(T*(*f)(T*,T*));
12  T *getDataOut();
13  virtual ~Reduce();
14 };
```

Figura 18: Declaração da classe Reduce.

4.5.6 Classe Fork

O esqueleto Fork é semelhante ao esqueleto Map. A diferença é que o Fork recebe uma lista de funções, sendo uma para cada elemento do vetor de entrada. A Figura 19 mostra a classe Fork que tem o *template* com dois parâmetros, o tipo IN para os dados de entrada e o OUT para os dados de saída.

```
1  template<class IN, class OUT>
2  class Fork : public DataParallelism<IN,OUT>{
3   Fork(): DataParallelism<IN,OUT>();
4   Fork(const Fork &e): DataParallelism<IN,OUT>(e);
5   Fork(OUT* (*(*f))(IN*), IN *listIn, OUT *&listOut, int length):
        DataParallelism<IN,OUT>(listIn,listOut,length);
6   Fork(OUT* (*(*f))(IN*), int length): DataParallelism<IN,OUT>(length);
7   Fork(Kanga<IN,OUT> *obj, IN **matIn, OUT **&matOut, int length):
        DataParallelism<IN,OUT>(matIn,matOut,length);
8   void start();
9   void join();
10  void setDataIn(IN* listIn);
11  void setFunction(OUT* (*(*f))(IN*));
12  OUT *getDataOut();
13  virtual ~Fork();
14 };
```

Figura 19: Declaração da classe Fork.

O Fork recebe um ponteiro para um vetor de funções com o formato OUT*(*(*f))(IN*), um ponteiro para um vetor de dados do tipo IN, uma referência para o vetor de saída do tipo OUT e o tamanho destes vetores.

A execução do Fork é iniciada com a chamada do seu método start() que cria um vetor de objetos *threads* da classe Thread. Cada um desses objetos é criado passando uma função do vetor de funções com índice i e um elemento do vetor

de dados de entrada com o mesmo índice. Após a construção de cada objeto *thread* o seu método start() é invocado. Assim, cada função é executada com o elemento sendo passado como parâmetro e o retorno sendo armazenado.

O método join() do Fork realiza a sincronização dos *threads* por meio da chamada do método join() de cada *thread*, logo após isto, os dados de saída poderão ser acessados.

4.5.7 Classe MapReduce

O esqueleto MapReduce consiste na união dos esqueletos Map e Reduce, desta forma, ele deve receber como parâmetro uma função para operação de mapeamento e uma função para a operação de redução. Porém, a função de redução deve receber e retornar dados do mesmo tipo do retorno da função de mapeamento como pode ser visto na Figura 20.

```
1  template<class IN,class OUT>
2  class MapReduce : public DataParallelism<IN,OUT>{
3   MapReduce(): DataParallelism<IN,OUT>();
4   MapReduce(const MapReduce &e): DataParallelism<IN,OUT>(e);
5   MapReduce(OUT*(*fM)(IN*), OUT*(*fR)(OUT*OUT*), IN *listIn, OUT *&dataOut, int
        length): DataParallelism<IN,OUT>(listIn,dataOut,length);
6   MapReduce(OUT*(*fM)(IN*), OUT*(*fR)(OUT*,OUT*), int length): DataParallelism<
       IN,OUT>(length);
7   MapReduce(Kanga<IN,OUT> *objMap, Kanga<IN,OUT> *objRed, IN **matIn, OUT *&
       listOut, int length): DataParallelism<IN,OUT>(matIn,listOut,length);
8   void start();
9   void join();
10  void setDataIn(IN *listIn);
11  void setFunction(OUT*(*fM)(IN*), OUT*(*fR)(OUT*OUT*));
12  OUT *getDataOut();
13  virtual ~MapReduce();
14 };
```

Figura 20: Declaração da classe MapReduce.

A execução do MapReduce é realizada em duas etapas, na primeira é aplicada a função de mapeamento sobre o vetor de entrada, seguindo o mesmo algoritmo do esqueleto Map descrito nas seções anteriores. A segunda etapa a função de redução é aplicada com os valores resultantes da etapa anterior. A operação de redução segue o mesmo algoritmo do esqueleto Reduce. No final da execução um valor escalar será retornado.

4.5.8 Classe Zip

O Esqueleto Zip é implementado pela classe da Figura 21. Seguindo a sua definição, o Zip é uma variação do esqueleto Map, logo a sua implementação é semelhante a do Map. Além disso, o *template* do Zip segue a mesma semântica

do *template* do Map.

Os construtores do Zip recebem como parâmetro um ponteiro para uma função com o protótipo OUT*(*f)(IN*,IN*) que recebe dois parâmetro do tipo IN e retorna um valor do tipo OUT, dois ponteiros para vetores de entrada do tipo IN, uma referência para o vetor resultante e o número de elementos dos vetores de entrada e do vetor de saída.

```
1  template<class IN, class OUT>
2  class Zip : public DataParallelism<IN,OUT>{
3   Zip(): DataParallelism<IN,OUT>();
4   Zip(const Zip &e): DataParallelism<IN,OUT>(e);
5   Zip(OUT*(*f)(IN*, IN*), IN *listIn, OUT *&listOut, int length):
        DataParallelism<IN,OUT>(listIn,listOut,length);
6   Zip(OUT*(*f)(IN*, IN*), int length): DataParallelism<IN,OUT>(length);
7   Zip(Kanga<IN,OUT> *obj, IN **matIn, OUT **&matOut, int length):
        DataParallelism<IN,OUT>(matIn,matOut,length);
8   void start();
9   void join();
10  void setDataIn(IN* listIn);
11  void setFunction(OUT*(*f)(IN*, IN*));
12  OUT* getDataOut();
13  virtual ~Zip();
14 };
```

Figura 21: Declaração da classe Zip.

O método start() do Zip dispara um conjunto de *threads* com o mesmo número de elementos dos vetores de entrada. Cada *thread* recebe a função do programador e um elemento de cada vetor de entrada. A execução de cada *thread* consiste em aplicar a função nos dois elementos. Para realizar a sincronização dos *threads* o método do join() do Zip é invocado.

4.5.9 Classe Scan

O esqueleto Scan esta ilustrado na Figura 22 onde é possível perceber que os seus parâmetros são iguais aos demais esqueletos da classe de paralelismo de dados. Contudo, *template* tem apenas um parâmetro, uma vez que a sua função de entrada será aplicada recursivamente no vetor de entrada, assim o seu dado de retorno deve ser do mesmo tipo do dado passado pelo parâmetro. Além disso, a função de entrada deve ter o seguinte protótipo: T*(*f)(T*,T*).

A implementação do Scan utiliza um objeto *thread* para executá-lo, isto é, este *thread* executa um método *private* denominado run() que aplica a função de entrada recursivamente sobre o vetor de dados. Cada chamada recursiva do método run() são criados os *threads* necessários para a operação de Scan sobre os dados.

```
1  template<class T>
2  class Scan : public DataParallelism<T,T>{
3   Scan(): DataParallelism<T,T>();
4   Scan(const Scan &e): DataParallelism<T,T>(e);
5   Scan(T*(*f)(T*,T*), T *listIn, T *&listOut, int length): DataParallelism<T,T
        >(listIn,listOut,length);
6   Scan(T*(*f)(T*,T*), int length): DataParallelism<T,T>(length);
7   Scan(Kanga<T,T> *obj, T **matIn, T **&matOut, int length): DataParallelism<T,
        T>(matIn,matOut,length);
8   void start();
9   void join();
10  void setDataIn(T* listIn);
11  void setFunction(T*(*f)(T*,T*));
12  T getDataOut();
13  virtual ~Scan()
14 };
```

Figura 22: Declaração da classe Scan.

4.5.10 Classe DC

A classe DC implementa o esqueleto de Divisão e Conquista, este esqueleto se propõe a solucionar problemas usando a estratégia de divisão e conquista. Desta forma, foi implementada uma classe para que o programador realize a descrição do problema a ser resolvido pelo DC por meio da herança dos métodos da classe abstrata da Figura 23.

```
1  class ProblemaDC{
2   virtual int base();
3   virtual ProblemaDC* divisaoDir();
4   virtual ProblemaDC* divisaoEsq();
5   virtual void conquista();
6   virtual void combina(ProblemaDC* esq, ProblemaDC* dir);
7   virtual ~ProblemaDC();
8  };
```

Figura 23: Declaração da classe abstrata ProblemaDC.

Os métodos da classe da Figura 23 foram definidos para representar a estratégia de divisão e conquista, que de acordo com (CORMEN et al., 2009) consiste em três fases: a primeira é de divisão do problema até que seja indivisível, a segunda é a fase de conquista que soluciona os subproblemas e a terceira é a que combina as subsoluções encontradas.

A execução do DC se dá recursivamente, assim inicialmente o método base() do objeto do tipo ProblemaDC é invocado para verificar se o problema é indivisível, caso seja, o método conquista é chamado() e a solução do problema é retornada, caso contrário, o problema segue para a etapa de divisão. O programador deve implementar o método base para retornar 1, caso o problema não

seja divisível e zero se for divisível. Para realizar a divisão devem ser utilizador dois métodos: divisaoDir() e divisaoEsq() que retornam objetos da classe ProblemaDC. O primeiro objeto retornado será executado sobre um novo *thread*, enquanto o segundo será executado sobre o *thread* corrente.

O construtor da classe DC ilustrada na Figura 24 têm como parâmetros um ponteiro para um objeto do tipo ProblemaDC que descreve o problema de entrada e um ponteiro do mesmo tipo mas apontando para a solução do problema.

```
1  template<class T>
2  class DC : public Kanga<T,T> {
3    DC();
4    DC(const DC<T> &e);
5    DC(T *proIn, T *&proOut);
6    void start();
7    void join();
8    void setDataIn(T * pro);
9    T * getDataOut();
10   virtual ~DC();
11 };
```

Figura 24: Declaração da classe DC.

Os método start() dispara um *thread* que realiza a execução do método private run(), e o método join() realiza a sincronização deste *thread*. Além destes, foram implementados os métodos getDataOut() para retornar um ponteiro para a solução do problema e o método setDataIn(T * pro) para setar o problema de entrada.

O *template* da classe DC tem somente um parâmetro, o tipo passado por este parâmetro deve ser do tipo do ProblemaDC.

4.5.11 Classe Farm

O esqueleto Farm é implementado pela classe da Figura 25. O *template* desta classe possui somente um parâmetro, pois a função de entrada só possui parâmetro e não retorna valores. Desta forma o construtor de classe do Farm recebe uma função do tipo int*(*f)(T*), o ponteiro para o dado de entrada e o número de *threads* que serão criadas. Cada uma destas *threads* executará uma cópia desta função passando o valor entrada por parâmetro.

A implementação do esqueleto Farm utiliza uma função int*(*f)(T*) que retorna um valor inteiro para determinar o final da execução da função do programador.

O método start() do Farm dispara os *threads* passando a referência para a função de entrada. O método join() realiza a sincronização desses *threads*.

```
1  template<class T>
2  class Farm : public Kanga<T,int>{
3   Farm();
4   Farm(const Farm<T> &e);
5   Farm(int*(*f)(T*), T* in, int nThreads);
6   Farm(Farm<T> *farm,T** in, int nThreads);
7   void setFunc(int*f(T*));
8   void setDataIn(T* in);
9   void start();
10  void join();
11  virtual ~Farm();
12 };
```

Figura 25: Declaração da classe Farm.

4.6 Composições de Esqueletos

A forma que a composição foi implementada neste trabalho é baseada no padrão de projeto *Composite*. O padrão de projeto ou *Design Pattern Composite* tem o objetivo de agrupar objetos que fazem parte de uma relação parte-todo de forma a tratá-los sem distinção (GAMMA et al., 2000). Logo o padrão *Composite* descreve como usar a composição recursiva sem que o programador trate de maneira diferente objetos primitivos dos objetos recipientes (objetos que recebem e executam os objetos primitivos) (GAMMA et al., 2000). O *Composite* define hierarquias de classes contendo uma classe abstrata que descreve tantos os primitivos como recipientes. Os objetos primitivos podem compor objetos mais complexos, os quais, por sua vez podem compor outros objetos, assim por diante, recursivamente.

O padrão *Composite* define que objetos compostos e objetos primitivos sejam tratados de maneira uniforme, devido serem objetos de classes derivadas da mesma classe base. Este padrão de projeto permite que seja mais fácil adicionar novas subclasses primitivas e compostas, pois a estrutura de composição já está definida, apesar disso composição torna-se excessivamente genérica, o que pode tornar difícil restringir alguns componentes de realizarem a composição.

Na composição implementada na API deste trabalho, cada objeto de classe derivada da Kanga pode se comportar como um objeto recipiente ou como objeto primitivo.

O código da Figura 26 ilustra um exemplo de composição do objeto da classe Map com um objeto da classe DC. O objetivo deste código é criar a sequência de Fibonacci, assim o vetor de entrada vetIn do objeto map é um vetor de objetos do tipo Fib. A classe Fib representa um elemento da sequência de Fibonacci, o seu valor é armazenado no atributo inteiro n. Os objetos do vetIn são inicializados com valores do índices da sequência de Fibonacci. O vetOut após o término da

```
1  #define ANAHY
2  #include "Map.h"
3  #include "DC.h"
4
5  class Fib : public ProblemaDC{
6   private: int n;
7   public:
8    Fib(){ this->n=0; }
9    Fib(const Fib &e ){ this->n=e.n;}
10   Fib(int n){ this->n=n; }
11   ~Fib(){}
12   int base(){
13     if( this->n < 3) return 1;
14     return 0;
15   }
16   Fib* divisaoEsq(){ return new Fib(this->n-1); }
17   Fib* divisaoDir(){ return new Fib(this->n-2); }
18   void conquista(){ if(this->n==1 || this->n==2) this->n=1; }
19   void combina(Fib* x, Fib* y){ this->n=x->n+y->n;}
20 };
21
22 int main(int argc, char **argv){
23   AnahyVM::init(argc, argv);
24   int n=10, i;
25   Fib **vetIn = new Fib*[n];
26   Fib **vetOut;
27   DC<Fib> *dcFib = new DC<Fib>();
28   for(i=0; i< n; i++) vetIn[i]=new Fib(i+1);
29   kanga::Map<Fib,Fib> map(dcFib,vetIn,vetOut,n);
30   map.start();
31   map.join();
32   AnahyVM::terminate();
33   return 0;
34 }
```

Figura 26: Código da composição do esqueleto Map com o DC.

execução do map apontará para uma sequência de objetos do tipo Fib, cada um contendo um valor da sequência. Na linha 27 um objeto da classe DC é criado, denominado de dcFib, o parâmetro do seu *template* é do tipo da classe Fib. Da mesma forma os parâmetros do *template* do objeto map são do tipo Fib.

4.7 Conclusão

Este capítulo descreveu a interface desenvolvida neste trabalho com as classes dos esqueletos e as estruturas auxiliares. A Tabela 3 faz um resumo destas classes com a forma de criação dos objetos e o protótipo das funções utilizadas pelos esqueletos.

A interface proposta foi implementada em C++ utilizando diversos recursos de programação que esta linguagem oferece, em particular mecanismos baseados

Tabela 3: Classes implementadas e seus objetos.

Classes	Criação de Um Objeto	Protótipo da Função
Thread	Thread<IN,OUT>th(f, IN* , OUT*) Thread<IN,OUT>th(f, IN*, IN*, OUT*)	OUT* (*f)(IN*) OUT* (*f)(IN*, IN*)
Map	Map<IN,OUT>map(f, IN*, OUT*,int n)	OUT* (*f)(IN*)
Reduce	Reduce<T>red(f, IN*, OUT*,int n)	T* (*f)(T*, T*)
MapReduce	MapReduce<IN,OUT>mp(fM ,fR , IN*, OUT*, int n)	OUT* (*fM)(IN*) , OUT* (*fR)(OUT*, OUT*)
Zip	Zip<IN,OUT>zip(f,IN*, IN*, OUT*, int n)	OUT* (*f)(IN*, IN*)
Scan	Scan<T>scan(f,IN*, OUT*, int n)	OUT* (*f)(IN*, IN*)
Fork	Fork<IN,OUT>fork(f*,IN*, OUT*, int n)	OUT* (*(*f))(IN*)
DC	DC<T>dc(T*, T*);	
Farm	Farm<T>farm(f, int n)	int*(*f)(T*)

no uso de *template* e de herança. O programador com conhecimentos básicos de programação orientada a objetos em C++ não deverá encontrar dificuldades em utilizar o conjunto de classes oferecido.

Ao utilizar o recurso proposto para descrição do paralelismo um cuidado deve ser tomado. A interface proposta propõe um modelo para descrição da concorrência em termos de um conjunto de esqueletos, sendo delegado a estes esqueletos coordenarem a execução das atividades concorrentes geradas. A coordenação desta execução apoia-se no uso de ferramentas de mais baixo nível para criação de paralelismo efetivo e gestão do uso dos recursos do *hardware*, ou seja, da realização do escalonamento destas atividades sobre os recursos de *hardware*. Assim, o programador deve se limitar a utilizar a interface tal como oferecida, não introduzindo mecanismos estrangeiros de sincronização, como uso de *mutex*, ou realizar acesso a dados compartilhados, uma vez que nenhuma garantia de ordem de execução entre os *threads* ou de consistência no acesso aos dados é oferecida. No próximo capítulo são descritas as plataformas Anahy e Pthreads que foram utilizadas no desenvolvimento deste trabalho.

5 SUPORTE DE EXECUÇÃO

Neste capítulo são descritas as ferramentas de programação utilizadas para o desenvolvimento deste trabalho como suporte de execução *multithread*. As ferramentas são Pthreads e Anahy3.

A programação com estas ferramentas segue o conceito de que um processo é uma aplicação em execução que pode ser descrita em termos de vários fluxos de execução (*threads*) capazes de executar instruções de forma concorrente, logo uma aplicação *multithreaded* pode conter vários *threads*, sendo que cada *thread* contém seus dados privados, mas também compartilha recursos do mesmo processo. A concorrência nesse caso não está somente pelos recursos de processamento mas também pelo acesso aos dados na memória.

Pthreads e Anahy3 implementam o padrão de programação fork/join o que tornou a fácil portabilidade de código entre estas ferramentas. Outra característica comum a essas duas ferramentas é que elas possuem *threads* identificáveis individualmente. Assim as operações join podem identificar qual *thread* deve ser sincronizada, não limitando a forma de descrição da concorrência a modelos de execução do tipo fork/join aninhados como em Cilk e OpenMP.

5.1 Pthreads

Pthreads é uma ferramenta de programação que segue o Padrão POSIX (*Portable Operating System Interface*) da IEEE de 1995. Essa ferramenta fornece ao programador funções para a criação e sincronização de *threads* e mecanismos de sincronização *mutexs* e variáveis condicionais.

A ferramenta Pthreads implementa o padrão de programação *fork* e *join* para criação e sincronização de *threads*. Desta forma, o seu modelo de programação consiste em um conjunto de *threads* que executam diferentes tarefas da aplicação dentro de um mesmo processo. Esses *threads* compartilham os mesmos recursos, mas são escalonados separadamente pelo sistema operacional.

5.1.1 Recursos de Programação

Para criar um *thread* utilizando Pthreads o programador deve instanciar uma variável do tipo pthread_t, esta variável consiste no identificador do *thread*, ou seja, cada *thread* deve ter o seu identificador. O corpo do *thread* é definido como uma função ordinária C/C++, como o exemplo a seguir:

```
1 void* start_routine(void* arg){
2  \*Código executado pelo thread.*\
3  return out;
4 }
```

O parâmetro arg é do tipo void* que corresponde ao endereço de memória compartilhada onde estão localizados os dados utilizados pela função do *thread*. A variável out também é do tipo void*, é o retorno do resultado da execução sendo armazenado na memória compartilhada da arquitetura.

A ferramenta Pthreads oferece ao programador primitivas que permitem criar e sincronizar de forma explícitas os *threads* da aplicação, estas primitivas são listadas na Figura 27. A chamada da primitiva pthread_create() por um *thread* resulta na criação de um novo fluxo de execução.

```
1 int pthread_create(pthread_t *threadId, const pthread_attr_t *restrict_attr,
      void*(*start_routine)(void*), void *restrict_arg);
2 int pthread_join(pthread_t threadId, void **value_ptr);
3 int pthread_exit( void *value_ptr );
```

Figura 27: Primitivas oferecidas por Pthreads.

O parâmetro threadId da função pthread_create() é o identificador do *thread*, assim deve existir um identificador único para cada *thread*. O parâmetro start_routime é o ponteiro para a função que contém as instruções para serem executadas em paralelo. Como pode ser visto essa função recebe como parâmetro um ponteiro do tipo void, este ponteiro aponta para o endereço de memória onde estão os parâmetros de entrada da função. O retorno é um ponteiro do tipo void* para a posição de memória, onde devem ser escritos os resultados produzidos pela função. O último parâmetro arg é um ponteiro do tipo void* que aponta para a posição de memória onde estão os dados de entrada da função.

A primitiva que permite que um *thread* realize sincronização com outro *thread* é a pthread_join, isto é, esta primitiva garante que um *thread* continuará sua execução somente após o término da execução do *thread* apontado pelo identificador threadId. Este mecanismo de sincronização pode ser realizado somente uma vez para cada *thread*. O parâmetro value_ptr é atualizado com o endereço de memória que contém os dados retornados pelo *thread* sincronizado quando

este terminar. Caso o *thread* sincronizado já tenha terminado, ocorre simplesmente a recuperação do retorno. Caso contrário, o *thread* que necessita de sincronização permanecerá bloqueado até que o *thread* sincronizado termine a sua execução.

Um *thread* termina a sua execução de duas maneiras: quando a sua função executa a instrução return da linguagem C/C++ ou utilizando a primitiva pthread_exit. Observe que valor retornado é um ponteiro para uma área de dados que contêm o resultado do processamento efetuado pelo *thread*. É importante observar que, embora estes dois modos de terminar um *thread* sejam semelhantes, a semântica de suas execuções difere completamente. No caso do return ser invocado, a função executada pelo *thread* termina sua execução e a pilha de dados é destruída. Neste momento, eventuais objetos (instâncias de classes C++) que estejam aplicados nesta pilha são também destruídos, sendo invocado os destrutores correspondentes. Na invocação da primitiva exit, o *thread* é abandonado e o registro da pilha descartado, sem, portanto, destruir eventuais objetos. Esta diferença é significativa devido a implementação proposta neste trabalho ser baseada no uso de classes C++, e chamadas à primitiva exit pode ser uma fonte potencial de erros no programa.

5.1.2 Escalonamento

O escalonamento dos *threads*, criados por meio de Pthreads, é realizado pelo sistema operacional, sendo assim baseados em prioridades. O programador pode, no momento em que cria um novo *thread*, informar a prioridade de execução desejada para o novo *thread*. Existe quatro tipos de escalonamento definidos pelo serviço setschedpolicy:

- SCHED_RR: implementa uma política do tipo *Round-Robin*.
- SCHED_FIFO: implementa uma política do tipo *First-in First-out*.
- SCHED_OTHER: algoritmo de escalonamento especificado pelo programador.
- SCHED_BATCH: implementa a estratégia de escalonamento em *Batch*.

Os escalonamentos SCHED_RR e SCHED_FIFO estão disponíveis apenas quando o processo em execução tiver privilégios de superusuário. Esses tipos de escalonamentos são definidos para aplicações em tempo real e sua prioridade é superior à de processos com política SCHED_OTHER. No entanto deve ser observado que o padrão Pthreads define apenas a interface de serviços, não a implementação desses. Sendo assim, poucas premissas relacionadas ao

escalonamento podem ser de fato tecidas sem considerar o mecanismo de escalonamento realizado pelo sistema operacional. No entanto, pode-se afirmar que a estrutura de criação dos *threads* não é considerada nas tomadas de decisões do escalonador, e que todos os *threads* criados já são instanciados como fluxos de execução no momento em que são criados.

5.1.3 Acesso a Dados Compartilhados

Pthreds oferece ao programador mecanismos como *mutexs* e variáveis de condição para realizar a sincronização dos *threads* no acesso a dados compartilhados. Nesta seção são descritas as estruturas e funções utilizadas para manipular estes dois mecanismos.

5.1.3.1 Mutex

Mutex é uma abreviação de *mutual exclusion* (exclusão mútua) e consiste em um recurso para sincronização entre *threads* em regime de exclusão mútua. Em Pthreads, *mutex* é oferecido pelo tipo de dado pthread_mutex_t, este recurso oferece funções, listadas na Figura 28, que em conjunto com *mutex* são usadas dois ou mais *threads* no acesso a regiões críticas (trechos de código que acessam dados compartilhados). Assim, o *mutex* previne a inconsistências de dados provocadas por acesso concorrente à memória e condições de corrida.

```
1 int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *
      attr);
2 int pthread_mutex_destroy(pthread_mutex_t *mutex);
3 int pthread_mutex_lock(pthread_mutex_t *mutex);
4 int pthread_mutex_trylock(pthread_mutex_t *mutex);
5 int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Figura 28: Funções para manipulação do *mutex* de Pthreads.

As variáveis *mutex* devem ser inicializadas antes de serem utilizadas. *Mutexs* podem ser inicializadas estaticamente utilizando pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER ou dinamicamente por meio da função pthread_mutex_init().

O parâmetro *mutex* da função pthread_mutex_init() é uma instância do tipo pthread_mutex_t que vai ser inicializada e attr serve para definir atributos para a instância *mutex*. Para definir os atributos default é utilizado o valor NULL. A pthread_mutex_destroy() é usado para liberar a variável *mutex*.

A pthread_mutex_lock() é a função utilizada para adquirir um *lock* de uma variável *mutex* especificada. A chamada dessa função indica que o *thread* entrou em uma seção crítica. Se o *mutex* já encontra-se em poder de outro *thread*, a chamada bloqueia automaticamente o *thread* requisitante até que

o *mutex* em questão seja liberado. Para complementar esta função existe a função pthread_mutex_unlock() que indica que o *thread* chegou ao fim da seção crítica. Essa função retorna um código de erro se o *mutex* já tiver sido liberado ou se o mesmo *mutex* está sendo usado por outro *thread*. Por fim, pthread_mutex_trylock() é utilizada para realizar uma tentativa de executar *lock* em um *mutex*, entretanto se o *mutex* já encontra-se em uso essa função retorna imediatamente um código de erro.

5.1.3.2 Variável Condicional

Os *mutexs* permitem prevenir acessos simultâneos a variáveis compartilhadas. No entanto, por vezes o uso de *mutex* pode ser ineficiente, pois é necessário consultar sucessivamente a variável até que seja liberada. Assim, as variáveis condicionais permitem suspender a execução dos *threads* (liberando o processador) até que alguma condição seja verdadeira.

Variáveis condicionais em Pthreads tem o tipo pthread_cond_t. Uma variável condicional pode ser inicializada estaticamente da seguinte forma: pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER ou dinamicamente utilizando a função pthread_cond_init().

A função pthread_cond_init() tem como parâmetro a variável condicional cond e o attr que define atributos da cond. A função pthread_cond_destroy desaloca a variável condicional.

```
1 int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
2 int pthread_cond_destroy(pthread_cond_t *cond);
3 int pthread_cond_signal(pthread_cond_t *cond);
4 int pthread_cond_broadcast(pthread_cond_t *cond);
5 int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const
       struct timespec *abstime);
6 int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Figura 29: Funções para manipulação de variáveis condicionais de Pthreads.

A função pthread_cond_signal() é usada para sinalizar ou acordar um dos *threads* bloqueados na variável cond. Caso existam vários *threads* bloqueados, apenas um é acordado (não é especificado qual). pthread_cond_broadcast() acorda todos os *threads* que possam estar bloqueados na variável cond. Se nenhum *thread* estiver bloqueado na variável especificada, nada acontece em ambas as funções.

Uma variável condicional está sempre associado com um *mutex*. *Mutex* deve ser usado para prevenir que um *thread* execute pthread_cond_signal() antes que outro execute pthread_cond_wait(). Desta forma, a função pthread_cond_wait() bloqueia o *thread* na variável condicional cond, esta função de modo atômico

libera o *mutex* e espera por cond seja sinalizada. O *thread* em execução é suspenso e não consome tempo de CPU até que a variável condicional seja sinalizada. Isso requer que se obtenha o *lock* sobre o *mutex* antes de invocar a função. Quando a variável é sinalizada e o *thread* é acordado, pthread_cond_wait() readquire o *lock* no *mutex* antes de retornar à execução. De forma semelhante a função pthread_cond_timedwait() atomicamente libera o *mutex* e espera a cond, mas está operação possui um tempo limite de duração especificado pelo parâmetro abstime. Caso a cond não tenha sido sinalizada antes do tempo abstime, o mutex é readquirido e a pthread_cond_timedwait() retorna um erro.

5.2 Anahy3

O Anahy3 implementa o modelo de ambiente de execução *multithread* do Anahy na linguagem C++. Este ambiente utiliza algoritmos de lista no escalonamento de *threads* em seu núcleo, e os seus processadores virtuais (PVs) empregam uma política *Help-First* sem migração de *threads*. Estes PVs oferecem suporte a execução dos *threads* do usuário no modelo NxM, onde N *threads* geradas pelo programa são escalonados, em nível aplicativo, sobre M PVs os quais consistem em unidades de escalonamento para o sistema operacional, responsável pelo mapeamento destes sobre os processadores físicos da máquina.

O Anahy3 tem uma interface minimalista baseada no padrão *fork*/*join*, ou seja, toda a criação e sincronização de *threads* é realizada pela invocação dos métodos *fork* e *join*, respectivamente. A interface utiliza as primitivas *init* para inicializar e *terminate* para encerrar a execução do ambiente.

A arquitetura de Anahy3 é baseada em PVs com listas individuais, sem uma lista global, implicando na necessidade de roubo de trabalho. Estes PVs são implementados como *threads* POSIX, criadas no nível de sistema operacional. O ambiente de execução se encarrega de, em tempo de execução, realizar o balanceamento de carga da aplicação sobre as listas dos PVs criados.

5.2.1 Recursos de Programação

A interface de programação de Anahy3 permite que os *threads* sejam criados por meio de objetos. Estes objetos pertencem à classe AnahyJob e são denominados de *jobs*, sendo por meio deles que o ambiente Anahy3 trabalha. Um *job* consiste num descritor de um *thread* no nível de aplicação. Basicamente, a estrutura de dados de um *job* encapsula o endereço estático da função de início do *thread*, um ponteiro para os argumentos que essa função deve receber e um

ponteiro para a área de memória na qual os resultados da função devem ser escritos. O corpo de um *job* é definido como uma função convencional C/C++, sendo da mesma forma que Pthreads, como pode ser vista na função a seguir:

```
1  void* start_routine(void* arg){
2   \*Código executado pelo thread.*\
3   return out;
4  }
```

Para determinar que o *job* seja executado concorrentemente com os demais *jobs* no momento da execução da aplicação, deve-se chamar a primitiva *fork*. A sincronização deste *job* é realizada pela primitiva *join*. O *fork* e *join*, Figura 30, são implementadas como métodos da máquina virtual do Anahy3, denominada de AnahyVM.

A Figura 30 mostra as primitivas *init* e *terminate* que servem para inicializar e terminar, respectivamente, o ambiente de execução. O método AnahyVM::init recebe os mesmos parâmetros das a função *main*, devido os parâmetros da maquina virtual serem passados por linha de comando no momento da execução. Esses parâmetros definem certos comportamentos do ambiente de execução, como o número de PVs a serem utilizados e atributos de afinidade de *threads* do sistema com processadores, esta afinidade determina que um PV seja escalonado pelo sistema operacional sempre sobre o mesmo processador físico. Enquanto AnahyVM::init cria e inicializa os PVs e os *threads* de sistema que suportam a execução paralela do PVs. O AnahyVM::terminate sincroniza os *threads* de sistema e libera a memória ocupada pelos objetos do ambiente de execução.

```
1  static void AnahyVM::init(init argc, char **argv);
2  static void AnahyVM::terminate();
3  static void AnahyVM::fork(AnahyJob* job);
4  static void AnahyVM::join(AnahyJob* job, void** result);
```

Figura 30: Subconjunto básico da API da máquina virtual de Anahy3.

Existem três situação que o programador pode instanciar um objeto da classe AnahyJob, eles são exemplificadas na Figura 31. A primeira o objeto é criado na pilha da função, como representado na linha 11, nesta situação um custo menor de execução devido o *jobAutoReleased* ser criado na pilha de execução do PV. Esta forma de criação de objetos deve ser utilizada quando o *job* não precisa ser usado após o final da execução da função. O *join* para esse *job* é realizado na linha 20.

A segunda maneira de criar um *job* é por meio da alocação dinâmica usando o *new* do C++, como esta na linha 14. Contudo, os objetos criados desta maneira,

```
1  void *bar(void *a){return NULL;}
2
3  void *baz(void * input){
4   AnahyJob * job = (AnahyJob*)input;
5   void * result;
6   AnahyVM::join(job,&result);
7   delete job;
8  }
9
10 void * foo(void*){
11  AnahyJob jobAutoReleased(bar,NULL);
12  AnahyVM::fork(&jobAutoReleased);
13
14  AnahyJob* jobAllocated = new AnahyJob(far,NULL);
15  AnahyVM::fork(jobAllocated);
16
17  AnahyJob* smartJob = AnahyJob::new_smart_job(baz, (void*)jobAllocated);
18  smartJob->set_join_counter(1);
19
20  AnahyVM::join(jobAutoReleased,NULL);
21  return (void*) smartJob;
22 }
23
24 int main(int argc, char** argv){
25  AnahyVM::init(argc, argv);
26
27  AnahyJob job(foo,NULL);
28  AnahyVM::fork(&job);
29
30  AnahyVM::join(job,NULL);
31  AnahyVM::terminate();
32  return 0;
33 }
```

Figura 31: Exemplo de programa irregular em Anahy3.

explicitamente alocados pelo programador, devem ser desalocados pelo próprio programador, como é demonstrado na linha 7. O objeto *jobAllocated* é criado no *heap* e continuará válido mesmo após o final da função foo, porém este tipo de criação gera mais sobre custos, visto que o *heap* é uma área de memória compartilhada protegida por um *mutex* implícito.

Por fim, a terceira forma é exemplifica na linha 17 usando o método *AnahyJob::new_smart_job*. Os objetos obtidos desta maneira são alocados no *heap* e destruídos automaticamente pelo ambiente de execução do Anahy3. O ambiente sabe que deve destruir um *smart job* quando este *job* já recebeu um número específico de *joins*. Este número deve ser determinado pelo programador, e caso ele não o faça, o ambiente destrói o *job* após o primeiro *join*. A linha 18 mostra como o número de operações *join* que um *job* deve receber pode ser especificado (neste caso, o exemplo é redundante pois 1 é o valor padrão). A utilização dos *smart jobs* gera sobre custos ligeiramente maiores do que a criação de *jobs*

por meio de chamadas do *new*, porém facilita a programação de aplicações concorrentes complexas em Anahy, uma vez que o gerenciamento da memória ocupada pelos descritores é feito pelo ambiente de execução.

5.2.2 Escalonamento

O escalonamento no ambiente Anahy3 é realizado por meio dos processadores virtuais (PVs), pois são eles que determinam a execução do trabalho descrito pelos *jobs*. Os PVs utilizam a política de escalonamento *Help-First* sem migração. Cada PV executa o algoritmo de escalonamento em paralelo com outros PVs da seguinte forma: um PV ocioso busca o *job* com maior co-nível em sua lista; caso não obtenha nenhum, o PV ocioso escolhe um outro PV aleatoriamente, e tenta roubar o *job* com maior co-nível de sua lista. Quando um PV termina a execução de um *job* (ou não obtém sucesso nas buscas por *jobs* prontos) e possui um *thread* no topo de sua pilha, o PV tenta continuar a execução deste *job*.

5.2.3 Modelo de Fluxo de Dados

A ferramente Anahy3 não disponibiliza *mutex* e nem semáforos, pois o lançamento dos *threads* não é imediato. Isso ocorre devido existirem processadores virtuais que limitam a execução de um *thread* da aplicação por vez, de forma que estes *threads* tornam-se fluxos de execução de fato sobre os PVs, por uma decisão do escalonador. Desta forma, por exemplo, o uso de *mutex* poderia implicar em situações de *deadlock*, que é quando um *thread* depende de uma sincronização que será satisfeita por um *thread* que ainda não está em execução.

5.3 Conclusão

A API desenvolvida neste trabalho é destinada para o ambiente Anahy. Porém, a implementação dos testes realizados foi estendida para ter também suporte de Pthreads, o que permite demonstrar que API é independente de plataforma de suporte e poderá ser estendida para outras plataformas de programação paralela. No próximo capítulo são descritas as avaliações realizadas com a API Kanga.

6 AVALIAÇÃO

Este capítulo apresenta avaliações e análises da API desenvolvida neste trabalho. Em um primeiro momento é apresentada uma avaliação de desempenho, onde é dado destaque para aferição do *overhead* introduzido pelo o uso de Kanga, sendo também apresentado os tempos de execução de um caso de estudo completo.

No restante do capítulo a Kanga é discutida em relação às composições de esqueletos que ela permite realizar e é comparada com ferramentas similares.

6.1 Análise de Desempenho

O foco do presente trabalho não foi o processamento de alto desempenho, mas sim uma interface de programação paralela. O entendimento deste trabalho é que o desempenho deve ser provido pelo suporte de execução utilizado, ou seja, Pthreads e Anahy. No entanto, tendo em vista que em grande parte dos problemas onde o paralelismo é utilizado como parte do desempenho, este aspecto não pode ser inteiramente negligenciado. Assim, é importante analisar o *overhead* introduzido pelo uso da Kanga.

Para determinar o *overhead* causado sobre a atual interface do Anahy3 e sobre Pthreads, foram realizados dois testes com aplicações implementadas, com e sem a interface Kanga. As aplicações selecionadas foram o Fibonacci e o algoritmo de ordenação MergeSort.

O Fibonacci foi selecionado para realizar testes com o esqueleto Map, assim por meio do Map um vetor será percorrido, e para cada elemento deste vetor será realizado o cálculo da posição da série identificada pelo valor anotado na respectiva posição. As seguintes versões deste problema foram implementadas:

- MapKanAnahy: uso do esqueleto Map com suporte de execução oferecido por Anahy3;
- MapKanPthread: uso do esqueleto Map com suporte de execução oferecido por Pthread.

- MapAnahy: estratégia equivalente ao esquema de execução do esqueleto Map implementada em Anahy;
- MapPthread: estratégia equivalente ao esquema de execução do esqueleto Map implementada em Pthread;

O algorítimo de ordenação MergeSort foi selecionado para serem realizadas análises com o esqueleto DC, devido utilizar a estratégia de divisão e conquista. As seguintes versões do problema foram implementadas:

- DCAnahy: implementação do MergeSort com suporte de execução oferecido por Anahy3;
- DCKanAnahy: uso do esqueleto DC para implementar MergeSort com suporte de execução oferecido por Anahy3;
- DCPthread: implementação do MergeSort com suporte de execução oferecido por Pthread;
- DCKanPthread: uso do esqueleto DC para implementar MergeSort com suporte de execução oferecido por Pthreads.

As versões DCAnahy e DCPthread são implementações que só utilizam funções. A implementação destas duas versões são praticamente iguais, mas tendo como diferença as primitivas de criação e sincronização oferecidas por Pthreads e Anahy3. O algoritmo MergeSort ordena um vetor dividindo ele, recursivamente, até que resulte em um subvetor de tamanho igual a 1, ou seja, quando o MergeSort alcança a sua base de recurção, e assim os subvetores são ordenados. Nas versões DCAnahy e DCPthread a cada divisão do vetor é criada um *thread* para ordenar um subvetor e o outro subvetor será ordenado sobre o *thread* corrente.

Todas as aplicações foram executadas 30 vezes numa máquina com as seguintes características:

- Número de núcleos do processador: 4;
- Memória RAM: 4 GB;
- Sistema Operacional: Ubuntu;
- Compilador: g++ (4.6).

Nas subseções seguintes são apresentados os resultados dos testes realizados com as aplicações descritas anteriormente sobre o *hardware* já descrito.

6.1.1 *Overhead* de Kanga com Anahy3

Os resultados dos testes realizados para analisar o quanto de *overhead* que a interface gera em Anahy3 são apresentados nesta seção. A Tabela 4 mostra as médias e desvios padrões dos tempos das trinta execuções realizadas com MapAnahy e MapKanAnahy. Além disso, cada aplicação foi executada utilizando vetores com 1, 2, 4 e 8 geradas pelo programa em execução.

Tabela 4: Tempos de execução em Anahy com o esqueleto Map e sem ele.

Tamanho do Vetor	Número de *threads*	MapAnahy		MapKanAnahy	
		Média	Desvio Padrão	Média	Desvio Padrão
1	1	2,30	0,54	2,26	0,56
2	2	2,26	0,57	2,25	0,55
4	4	2,26	0,56	2,27	0,57
8	8	3,60	0,73	3,60	0,73

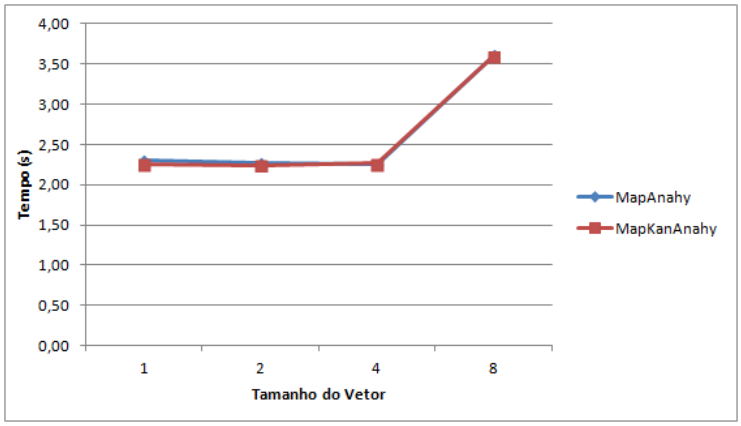


Figura 32: Comparação dos tempos de MapKanAnahy com MapAnahy, variando o número de elementos de um vetor de inteiros.

Na Tabela 5 estão as médias e os desvios padrões das execuções do algoritmo MergeSort utilizando somente a ferramenta Anahy3 e utilizando o esqueleto DC com o suporte de Anahy. As duas aplicações foram executadas com os vetores de diferentes tamanhos e no pior caso de ordenação, ou seja, o MergeSort teve que ordenar em ordem crescente um vetor com elementos em ordem decrescente. Além disso, foram criados, praticamente, um *thread* para cada elemento do vetor.

A versão do MergeSort utilizando o esqueleto DC apresentou um *overhead* em relação a versão implementada somente com Anahy. Como pode ser visto no gráfico da Figura 33 que demonstra que DCKanAnahy teve um aumento no tempo de execução em relação a DCAnahy. A principal causa do *overhead* causado pela implementação do DC é o tempo gasto com a utilização da instrução

Tabela 5: Tempos de execução em Anahy com o esqueleto DC e sem ele.

Tamanho do Vetor	Número de *threads*	DCAnahy		DCKanAnahy	
		Média	Desvio Padrão	Média	Desvio Padrão
40000	40000	0,175	0,344	0,615	0,723
45000	45000	0,126	0,003	0,693	0,745
50000	50000	0,200	0,339	0,696	0,711
55000	55000	0,153	0,002	0,802	0,718

new, que serve para criar os objetos das classes ProblemaDC e Thread, pois a cada nível da recursão são criados dois objetos da classe ProblemaDC e um da Thread.

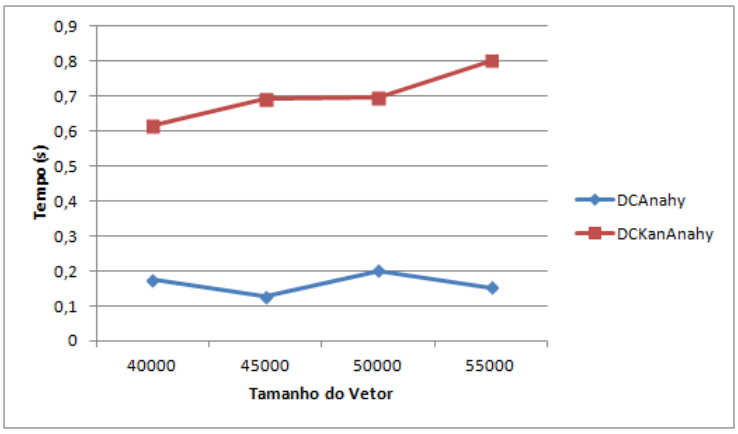


Figura 33: Comparação dos tempos de execução de MapKanAnahy com MapAnahy, variando o número de elementos de um vetor de inteiros.

6.1.2 *Overhead* de Kanga com Pthreads

Nesta seção estão os resultados dos testes para determinar o *overhead* de Kanga sobre Pthreads. Na Tabela 6 estão as médias e os desvios padrões das 30 execuções realizadas com MapPthreads e MapKanPthreads. Nessa tabela é possível perceber que as médias dos tempos de execução da MapPthreads são praticamente os mesmos da MapKanPthreads, baseado no fato de estarem dentro dos desvios padrões.

A Figura 34 apresenta o gráfico que representa uma comparação das médias das execuções da Tabela 6. Neste gráfico percebe-se que o esqueleto Map com o suporte de execução de Pthreads não é apresenta *overhead* em relação a versão sem o Map.

A Tabela 7 apresenta as médias e os desvios padrões das trinta execuções das aplicações DCPthreads e DCKanPthreads. As médias dos tempos de execução mostram que o esqueleto DC apresenta um *overhead* com suporte

Tabela 6: Tempos de execução em Pthreads com o esqueleto Map e sem ele.

Tamanho do Vetor	Número de *threads*	MapPthreads		MapKanPthreads	
		Média	Desvio Padrão	Média	Desvio Padrão
1	1	2,27	0,57	2,22	0,57
2	2	2,30	0,56	2,25	0,55
4	4	2,26	0,56	2,28	0,54
8	8	3,56	0,70	3,62	0,71

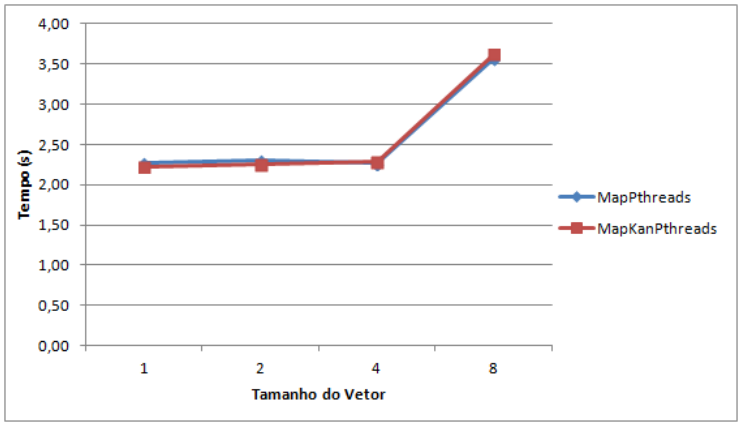


Figura 34: Comparação dos tempos de execução de MapKanPthreads com MapPthreads.

de Pthreads. A Figura 35 ilustra os dados da Tabela 7 na forma de um gráfico. Nesta figura o *overhead* causado se deve ao tempo gasto para alocar memória para os objetos das classes ProblemaDC e Thread.

Tabela 7: Tempos de execução em Pthreads com o esqueleto DC e sem ele.

Tamanho do Vetor	Número de *threads*	DCPthreads		DCKanPthreads	
		Média	Desvio Padrão	Média	Desvio Padrão
40000	40000	1,45	0,68	1,61	0,68
45000	45000	1,75	0,72	1,94	0,68
50000	50000	1,99	0,71	2,21	0,61
55000	55000	2,21	0,60	2,32	0,49

6.1.3 Comparação do Desempenho de Kanga com Anahy3 e Pthreads

A Figura 36 contêm os gráficos que representam os testes realizados para calcular o Fibonacci de 40, utilizando o esqueleto Map e sem ele. Os gráficos são apresentados na forma de diagrama de caixa, onde são destacados os valores máximos e mínimos observados, a mediana e os quartis inferiores e supe-

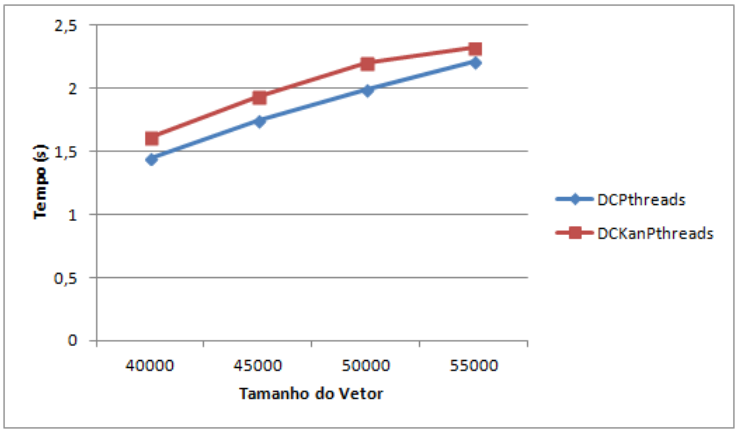


Figura 35: Comparação dos tempos de execução de DCKanPthreads com DCPthreads, variando o número de elementos de um vetor de inteiros.

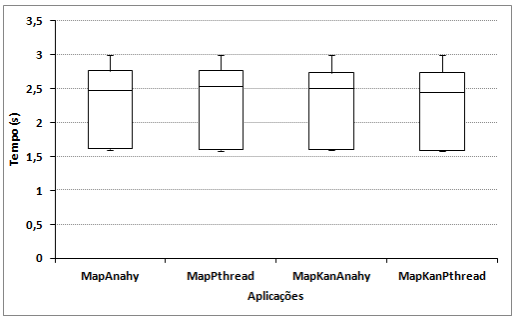
riores à mediana. Cada diagrama de caixa é resultado de 30 execuções de cada caso, com a máquina dedicada ao experimento. A máquina virtual de Anahy foi configurada sempre com 4 processadores virtuais, sobre os quais são mapeados os *threads* gerados pelo programa em execução. Manteve-se constante o número de processadores durante os casos de estudo para que esta variável não influenciasse nos resultados obtidos.

Os gráficos 36(a), 36(b) e 36(c) mostram que os tempos das execuções de todas as versões do problema são muitos semelhantes. Contudo, o gráfico 36(d) mostra que MapKanAnahy apresenta tempos de execução menores do MapKanPthread. Este ganha de desempenho entende-se não ser propriamente obtido pela ferramenta apresentada, mas sim pelos benefícios dos recursos de escalonamento de Anahy. Prova deste fato é que a versão da aplicação implementada diretamente sobre Anahy teve melhor desempenho do que a que utiliza a interface Kanga.

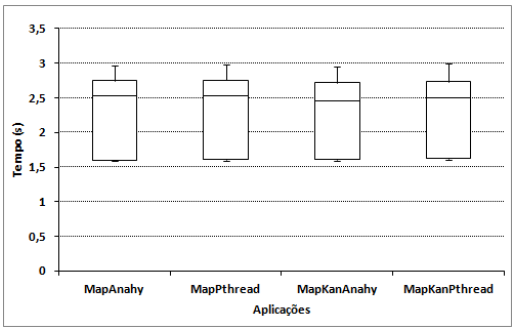
A Figura 37 apresenta os gráficos na forma de diagrama de caixa dos testes realizados com o algoritmo MergeSort. Neste caso o número de processadores virtuais também foi mantido constante em 4. Estes gráficos mostram que a versão DCKanAnahy é mais eficiente do que a versão DCKanPthread. Mais uma vez, nesta aplicação, conclui-se que o mecanismo de escalonamento de Anahy oferece benefícios para a execução em termos de desempenho.

6.2 Testes de Composições

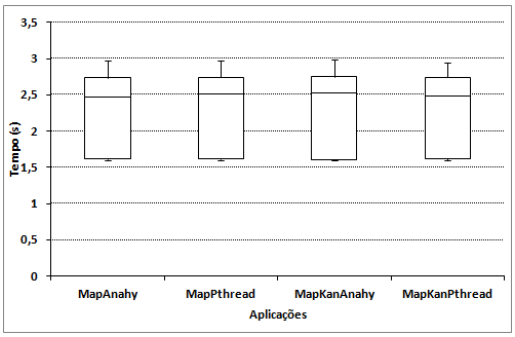
A Tabela 8 descreve a relação das composições entre os esqueletos implementados neste trabalho. Cada célula marcada com X representa que o esqueleto da coluna correspondente pode ser composto com o da linha de onde a marca se encontra.



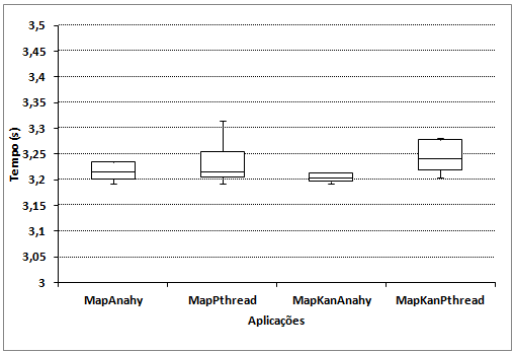
(a) Vetor com 1 elemento e 1 *threads*.



(b) Vetor com 2 elementos e 2 *threads*.

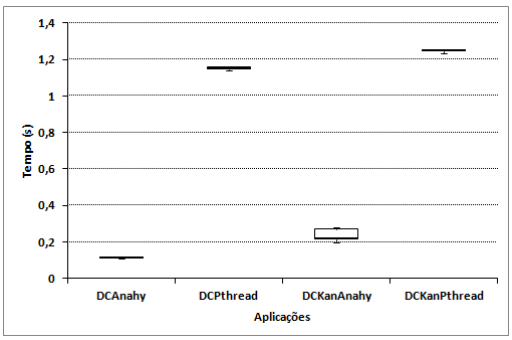


(c) Vetor com 4 elementos e 4 *threads*.

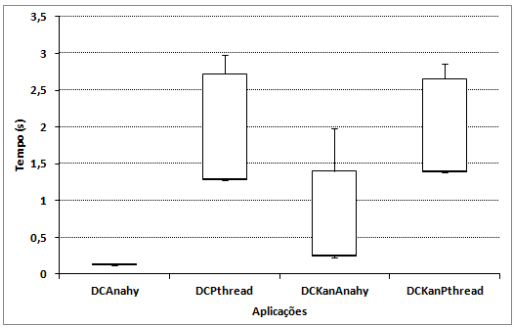


(d) Vetor com 8 elementos e 8 *threads*.

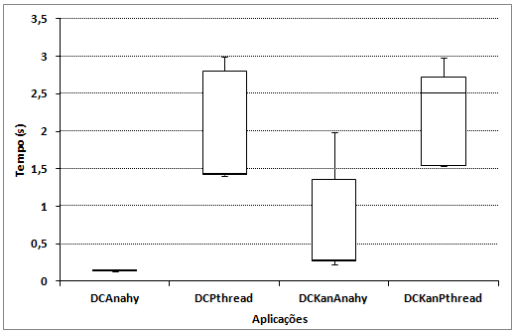
Figura 36: Comparação dos tempos de execução do esqueleto Map.



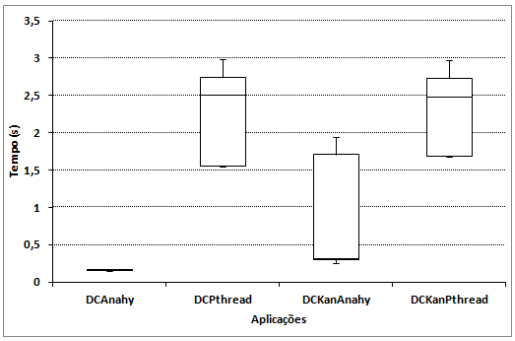
(a) Vetor com 10000 elementos.



(b) Vetor com 15000 elementos.



(c) Vetor com 20000 elementos.



(d) Vetor com 25000 elementos.

Figura 37: Comparação dos tempos de execução com o esqueleto DC. Cada gráfico representa o tempo do mergesort para ordenar o vetor de entrada.

Tabela 8: Relação de composições de esqueletos permitidas em Kanga.

		Map	Fork	Scan	Reduce	Zip	Farm	MapReduce	DC
Map		x	x	x	x		x	x	x
Fork		x	x	x	x		x	x	
Scan						x			
Reduce						x			
Zip						x			
Farm							x		
MapReduce	Map	x	x	x					
	Reduce					x			

Considerando as possibilidades de composição, o Map pode receber como parâmetro todos os esqueletos implementados, exceto o Zip, assim da mesma forma que aplica a função a cada elemento do vetor de entrada, o Map mapeia o esqueleto de entrada sobre cada elemento de um vetor de ponteiros e retorna um vetor do mesmo tipo.

A composição do esqueleto Fork com os outros implementados neste trabalho é realizada por meio do vetor de ponteiros para objetos de esqueletos, ou seja, não é feita por uma lista de funções. Nesse caso, irá existir um esqueleto para cada elemento do vetor de ponteiro de dados. Ambos vetores são passados por referência no construtor do Fork. Assim, o esqueleto Fork executa os métodos start() e join() de todos os esqueletos e armazena os seus respectivos retornos em um vetor de saída. Esta composição tem a limitação de que todos esqueletos da lista de entrada devem tratar o mesmo tipo de dado.

A implementação da composição do Scan e do Reduce se limitou a permitir que eles somente recebessem um objeto do esqueleto Zip. Isso se deve a implementação da composição deste trabalho ser baseada no fato de que o esqueleto é passado por parâmetro para outro esqueleto. Portanto, esse o primeiro deve agir da mesma forma que a função de entrada, ou seja, o Scan e o Reduce utilizam funções do tipo que recebem dois parâmetros de entrada e o Zip recebe duas listas como entrada. Desta forma, o Reduce utiliza um objeto do Zip para reduzir um vetor de ponteiros a um vetor, e o Scan aplica o Zip para realizar a operação de prefixo em um vetor de ponteiros.

O esqueleto Zip somente recebe um objeto dele mesmo, assim ao receber dois vetores de ponteiros será retornado um vetor do mesmo tipo e com mesmo número de elementos dos de entrada. O esqueleto Farm também recebe somente um objeto dele mesmo, assim o Farm executa o objeto recebido o número de vezes que é passado pelo seu construtor.

O esqueleto MapReduce difere dos demais por utilizar duas funções, uma

para etapa do Map e outra para a do Reduce. Os esqueletos que podem ser compostos na etapa do Map são: Map, Fork e Scan pois eles recebem um vetor e retornam um vetor tal como é requerido nesta etapa. Na etapa do Reduce o Zip é o único esqueleto que pode ser composto, isto se deve à ele receber dois vetores e retornar um, como previsto na etapa Reduce. A composição destes esqueletos com o MapReduce é realizada aplicando eles sobre um vetor de ponteiros, desta forma o MapReduce receberá, como entrada, um vetor de ponteiros e retornará um ponteiro.

6.3 Comparação com Trabalhos Relacionados

Nesta seção é realizada uma comparação dos *frameworks* SkePU e Muesli com o desenvolvido neste trabalho. São analisados os conjuntos de esqueletos que cada um oferece, as linguagens nas quais os esqueletos são implementados e as plataformas de programação paralela e distribuída que são utilizadas. Além disso, quatro estudos de caso foram realizados com estas interfaces com o intuito de obter dados para uma comparação qualitativa dos códigos fontes produzidos.

6.3.1 Comparações entre Frameworks

A Tabela 9 contém a relação do conjunto de esqueletos dos *Frameworks* SkePU, Muesli e Kanga. O SkePU possui somente esqueletos destinados ao paralelismo de dados, diferente do Kanga e Muesli que oferecem esqueletos tanto de paralelismo de dados como de tarefas. O Muesli oferece o maior número de esqueletos entre os três, sendo que o diferencial são os esqueletos BB (*Branch and Bound*) e Pipe, contudo o Fork somente é fornecido pelo Kanga.

Tabela 9: Comparação do conjunto de esqueletos implementado.

Framework	Map	Reduce	Scan	MapReduce	Zip	DC	Farm	BB	Pipe	Fork
SkePU	x	x	x	x	x					
Muesli	x	x	x	x	x	x	x	x	x	
Kanga	x	x	x	x	x	x	x			x

A relação das características dos *Frameworks* SkePU, Muesli e Kanga estão na Tabela 10. Os três *frameworks* têm em comum as linguagens de implementação C/C++, fator que influenciou na escolha destas ferramentas para a presente comparação. No entanto, os suportes de paralelismo de execução por elas empregado são diferentes. A composição de esqueletos é oferecida por Kanga e Muesli sendo que em ambas ferramentas a composição é limitada, ou seja, em ambos *frameworks* a composição é possível somente entre alguns es-

queletos. No entanto, o não oferecimento de composição em SkePU reduz suas possibilidades de utilização.

Tabela 10: Comparação dos *Frameworks*.

Framework	*Linguagem*	*Implementação*	*Composição*	*Suporte de Execução*
SkePU	*C++*	*C++*	*Não*	*OpenCL, OpenMP, CUDA*
Muesli	*C++*	*C++*	*Sim*	*MPI, OpenMP*
Kanga	*C++*	*C++*	*Sim*	*ANAHY, Pthreads*

Na Tabela 10 observamos que Muesli e SkePU possuem um leque maior de suportes de execução. Enquanto, em hora atual, Kanga possui suporte apenas de ferramentas *multithread*, as demais incluem, além do suporte de ferramentas *multithread*, suporte de ferramentas baseadas em troca de mensagens (MPI em Muesli) e para exploração de GPUs (OpenCL e CUDA em SkePU).

Deve-se notar que SkePU é uma ferramenta com foco de aplicação bem definido, uma vez que o conjunto de esqueletos oferecidos exploram o paralelismo de dados e por possuir suporte a execução em GPU. No Muesli e na Kanga um perfil tão definido não é observado, mas é ofertado um espectro maior de aplicações. Já a introdução dos esqueletos BB e Pipe em Muesli pode ser compreendida pelo fato desta ferramenta contar, também, com suporte de execução dado por MPI. Tais esqueletos permitem sobrepor, pelo menos parcialmente, custos de comunicação por cálculo efetivo.

6.3.2 Estudos de Casos

As aplicações *DotProduct* (MCCOOL; ROBISON; REINDERS, 2012), SaxPY (MCCOOL; ROBISON; REINDERS, 2012), Média Aritmética e *Prefix Sum* (BLELLOCH, 1990) foram selecionadas para serem implementadas nos *frameworks*: SkePU, Muesli e Kanga. Estas aplicações são operações matemáticas que podem ser implementadas paralelamente utilizando os esqueletos.

6.3.2.1 Dot Product

O *DotProduct*, ou produto escalar, é a soma do produto dos elementos dos vetores $\vec{a}$ e $\vec{b}$. Assim, seja dois vetores $\vec{a} = (a_1, a_2, \cdots, a_n)$ e $\vec{b} = (b_1, b_2, \cdots, b_n)$ o *DotProduct* é calculado da seguinte forma:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Na figura 36 é apresentado o código para *DotProduct* em SkePU 38(a), Muesli 38(b) e Kanga 38(c).

O SkePU oferece um conjunto de abstrações para que o usuário defina o código a ser executado pelo esqueleto. A solução proposta pelo o SkePU é restringir a função para que todos os parâmetros recebidos e o seu retorno sejam do mesmo tipo. Na Figura 38(a), plus_f e mult_f recebem dois inteiros (a,b) e retorna um int, linhas 4 e 5. Para implementar o *DotProduct* o esqueleto MapReduce é utilizado, pois a etapa do Map aplica a estrutura plus_f sobre os dois vetores de entrada e a Reduce utiliza a mult_f para reduzir o vetor resultante da etapa do Map. A mult_f recebe como parâmetro um elemento de cada vetor e retorna o produto deles e a plus_f realiza a soma de todos os produtos. Na linha 12 é realizada a execução do MapReduce passando os vetores de entrada e a variável r recebe o valor *DotProduct*.

O *DotProduct* no Muesli utiliza os esqueletos Zip e o Reduce nas linhas 13 e 14, respectivamente na Figura 38(b). Cada um destes esqueletos recebe funções ordinárias C++ que tem o número de parâmetros determinados pela implementação dos esqueletos no Muesli. Os tipos dos dados dos parâmetros destas funções devem ser do mesmo tipo dos dados de retorno. Para utilizar os esqueletos do Muesli deve-se utilizar as suas estruturas de dados, assim nas linhas 11 e 12 são criados dois vetores, inicializados por meio dos vetores ordinários do C++. Na linha 13 o esqueleto Zip é aplicado sobre os vetores v0 e v1 utilizando a função mult_f. O método v0.zipWith retorna um vetor do tipo DistributedArray. O Reduce é aplicado sobre o vetor v3 na linha 14 por meio do método v3.fold utilizando a função plus_f. A variável r armazena o valor do *DotProduct*.

Em Kanga são utilizadas funções ordinárias C/C++ para definir o código a ser aplicado pelos esqueletos, como está na Figura 38(c). O programador deve observar que estas funções são definidas com o número de parâmetros requeridos pelo esqueleto, caso um número maior de parâmetros seja necessário o programador pode utilizar como alternativa estruturas e classes. A função que do esqueleto Zip recebe pode ter os tipos dos parâmetros diferentes do tipo de retorno, definindo os respectivos tipos por meio *template*, mas no Reduce os parâmetros devem ter o mesmo tipo do retorno, uma vez que o *template* possui somente um parâmetro. O cálculo do *DotProduct* é realizando primeiramente pelo objeto do esqueleto Zip que recebe como parâmetros a função mult_f, os dois vetores de entrada vetIn0 e vetIn1 e o ponteiro vetOut, na linha 12. A soma dos produtos é realizado pelo objeto do Reduce, linha 15, que utiliza o vetOut como vetor de entrada e retorna o valor do *DotProduct* por meio do ponteiro r.

6.3.2.2 SaxPY

A operação SaxPY consiste em $\vec{x} = (x_1, x_2, \cdots, x_n)$, $\vec{y} = (y_1, y_2, \cdots, y_n)$ e $\vec{r} = (r_1, r_2, \cdots, r_n)$ vetores e α uma constante numérica, assim:

```
1  #include "vector.h"
2  #include "mapreduce.h"
3
4  BINARY_FUNC(plus_f, int, a, b, return a+b;)
5  BINARY_FUNC(mult_f, int, a, b, return a*b;)
6
7  int main(int argc, char** argv){
8   int i;
9   skepu::Vector<int> vetIn0(10), vetIn1(10);
10  for(i = 0; i < 10; i++){ vetIn0[i]=i+1; vetIn1[i]=i+2;}
11  skepu::MapReduce<mult_f,plus_f> dotProduct(new mult_f, new plus_f);
12  int r = dotProduct(vetIn0,vetIn1);
13  return 0;
14 }
```

(a)

```
1  #include "DistributedArray.h"
2  #include "Muesli.h"
3
4  int mult_f(int a,int b) { return a*b;}
5  int plus_f( int a, int b) { return a + b;}
6
7  int main(int argc, char** argv){
8   msl::InitSkeletons(argc, argv);
9   int i,r, vetIn0[10], vetIn1[10];
10  for(i = 0; i < 10 ; i++){ vetIn0[i]=i+1; vetIn1[i]=i+2; }
11  msl::DistributedArray<int> v0(10, vetIn0);
12  msl::DistributedArray<int> v1(10, vetIn1);
13  msl::DistributedArray<int> v3=v0.zipWith(v1,&mult_f);
14  r=v3.fold(&plus_f);
15  msl::TerminateSkeletons();
16  return 0;
17 }
```

(b)

```
1  #define ANAHY
2  #include "Zip.h"
3  #include "Reduce.h"
4
5  int *mult_f(int *a,int *b){ *a=(*a) * (*b); return a; }
6  int *plus_f(int *a,int *b){ *a=(*a) + (*b); return a; }
7
8  int main(int argc, char** argv){
9   AnahyVM::init(argc, argv);
10  int vetIn0[10], vetIn1[10], *vetOut,*out, i;
11  for(i = 0; i < 10 ; i++){ vetIn0[i]=i+1; vetIn1[i]=i+2;}
12  kanga::Zip<int,int>zip= new kanga::Zip<int,int>(mult_f,vetIn0,vetIn1,vetOut,10);
13  zip->start();
14  zip->join();
15  kanga::Reduce<int>red = new kanga::Reduce<int>(plus_f,vetOut,out,10);
16  red->start();
17  red->join();
18  AnahyVM::terminate();
19  return 0;
20 }
```

(c)

Figura 38: Código da aplicação *DotProduct*: (a) SkePU, (b) Muesli e (c) Kanga.

$$r = \alpha * x + y$$

Esta operação é implementa paralelamente pelos códigos fontes da Figura 39. Todas as implementações utilizam dois vetores e constante igual $\alpha = 10$.

Para aplicar a operação SaxPY aos dois vetores no SkePU o código da Figura 39(a) utiliza a macro BINARY_FUNC na linha 4. Esta macro tem o nome de f e recebe dois parâmetros, um para cada vetor. Os dois vetores utilizados são declarados com o tipo de dados do SkePU, na linha 7. Para realizar a operação SaxPY é instanciado um objeto do esqueleto Map do SkePU na linha 10 com parâmetro o objeto de f. Na linha 11 o Map é executado passando por parâmetro os vetores de entrada e saída. A macro BINARY_FUNC permite que o esqueleto Map funcione como Zip no SkePU.

A implementação da SaxPY no Muesli, Figura 39(b), utiliza dois vetores vetX e vetY da própria linguagem C++ para armazenar os valores, mas as estruturas de dados do Muesli devem ser usadas como a representação na linha 10. O esqueleto utilizado para aplicar o SaxPY é o Zip, a semântica do Zip do Muesli consiste em um método da classe DistributedArray que recebe como parâmetro a função e outro objeto desta classe. O método v0.zipWith aplica a função saxpy sobre os vetores v0 e v1 e retorna o vetor resultante para v3 na linha 11.

O esqueleto utilizado para a implementação do SaxPY no Kanga é o Zip, este esqueleto recebe como parâmetro um ponteiro para uma função, dois vetores, uma referência para o vetor de saída e o número de elementos deste vetores. Na linha 10 um objeto de Zip é instanciado passando o tipo int por meio do seu *template*. Os vetores de entrada são o vetX e vetY, eles são inicializados na linha 9. A execução do Zip inicia-se na chamada do método start(), linha 11, ele dispara a execução dos *threads*, e a sincronização é realizada na linha 12 por meio do método join().

6.3.2.3 Prefix Sum

O *Prefix Sum* (BLELLOCH, 1990) é a soma de prefixo dos valores de um vetor. Assim, seja um vetor de entrada $\vec{x} = (x_1, x_2, \cdots, x_n)$ e um vetor de saída $\vec{y} = (y_1, y_2, \cdots, y_n)$, a soma de prefixo é calculada da seguinte forma:

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_0 + x_1 \\ y_2 &= x_0 + x_1 + x_2 \\ &\vdots \\ y_n &= x_0 + x_1 + \cdots + x_n \end{aligned}$$

O esqueleto indicado para implementar o *Prefix Sum* é o Scan. Este esqueleto realiza operações de prefixo, logo, ele recebe como parâmetro basicamente

```
1  #include "vector.h"
2  #include "map.h"
3
4  BINARY_FUNC(f, int,  a, b,return a*10+b;)
5
6  int main(){
7   skepu::Vector<int> vetX(10), vetY(10),vetOut(10);
8   int i;
9   for(i=0; i< 10; i++){ vetX[i]=i+1;     vetY[i]=i*2;}
10  skepu::Map<f> Saxpy(new f);
11  Saxpy(vetX,vetY,vetOut);
12  return 0;
13 }
```

(a)

```
1  #include "DistributedArray.h"
2  #include "Muesli.h"
3
4  int saxpy(int x, int y){ return (x*10) + y;}
5
6  int main(int argc, char** argv) {
7   msl::InitSkeletons(argc, argv);
8   int i, vetX[10], vetY[10];
9   for(i=0; i< 10 ;i++){ vetX[i]=i+1;     vetY[i]=i*2;}
10  msl::DistributedArray<int> v0(10, vetX), v1(10, vetY);
11  msl::DistributedArray<int> v3=v0.zipWith(v1,&saxpy);
12  TerminateSkeletons();
13  return 0;
14 }
```

(b)

```
1  #define ANAHY
2  #include "Zip.h"
3
4  int* saxpy(int* x, int* y){ *x= (*x)*10 + (*y); return x;}
5
6  int main(int argc, char **argv){
7   AnahyVM::init(argc, argv);
8   int i, vetX[10], vetY[10],*vetOut;
9   for(i=0; i<10 ;i++){ vetX[i]=i+1; vetY[i]=i*2;}
10  kanga::Zip<int,int> *zip= new kanga::Zip<int,int>(saxpy,vetX,vetY,vetOut,10);
11  zip->start();
12  zip->join();
13  AnahyVM::terminate();
14  return 0;
15 }
```

(c)

Figura 39: Código da aplicação SaxPY: (a) SkePU, (b) Muesli e (c) Kanga.

um vetor e uma função. Os três códigos fontes da Figura 40 utilizam o esqueleto Scan, contudo a do SkePU é o único que precisa utilizar uma estrutura para definir a função de entrada. Além disso, o programador deve recorrer aos vetores da STL para utilizar o seu esqueleto Scan. O Muesli e a Kanga utilizam funções ordinárias C++, mas o Muesli exige o uso da sua estrutura de dados e o seu Scan é um método desta estrutura. A Kanga exige somente o uso de vetores da linguagem C++, além disso, difere-se na execução do esqueleto Scan, que ocorre somente no momento que o método start() é invocado e a sincronização ocorre quando o join() é executado.

6.3.2.4 Média Aritmética

Esta aplicação consiste na média aritmética dos valores de um vetor que foi selecionada para demonstrar a utilização do esqueleto Reduce nos três *frameworks*. Os códigos fontes da Figura 41 implementam a média aritmética de um vetor de 10 elementos utilizando o Reduce. Esse esqueleto recebe como parâmetro um vetor e uma função. A função retorna a soma de dois elementos e é aplicada sobre o vetor até que seja reduzido a um valor escalar.

Na Figura 41(a) a BINARY_FUNC recebe como parâmetro dois valores e retorna a soma deles na linha 4. Esta estrutura é utilizada pelo Reduce do SkePU para realizar a soma dos elementos do vetor v0. Na linha 10, o objeto do Reduce realiza a execução e retorna a soma de todos os elementos. Este valor é dividido pelo número de elementos do vetor de entrada.

O Reduce do Muesli é executado pelo método *v0.fold*, Figura 41(b), que recebe a função que realiza a soma dos elementos do vetor por parâmetro. Este método retorna a soma de todos os elementos que será divididos por 10. O objeto v0 armazena os dados do vetor vetIn, inicializados na linha 9.

A implementação da média aritmética em Kanga é realizada por meio de um objeto do esqueleto Reduce que recebe como parâmetros uma função ordinária do C++, o vetor de entrada, um ponteiro para o valor resultante e o número de elementos do vetor de entrada. Nas linhas 11 e 12 da Figura 41(c) é realizada a execução do objeto red por meio dos métodos start() e join(), após a finalização do método do join, o valor de retorno do red é utilizado para obter a média na linha 13.

6.4 Implementação dos Princípios Importantes

A Kanga foi desenvolvida para ser uma biblioteca de esqueletos, foi utilizado na sua implementação os princípios defendidos por Muray Cole descritos no Capítulo 2. O primeiro princípio diz que o esqueleto deve ser oferecido de forma

```
 1 #include "vector.h"
 2 #include "scan.h"
 3
 4 BINARY_FUNC(plus_f, int, x, y,return x+y;)
 5
 6 int main(){
 7  skepu::Vector<int> v0(10), r;
 8  for(int i = 0; i < 10; ++i) v0[i]=i+1;
 9  skepu::Scan<plus> prefixSum(new plus);
10  prefixSum(v0, r, skepu::INCLUSIVE);
11  return 0;
12 }
```

(a)

```
 1 #include "DistributedArray.h"
 2 #include "Muesli.h"
 3
 4 int plus_f( int x, int y){ return x+y;}
 5
 6 int main(int argc, char** argv) {
 7  msl::InitSkeletons(argc, argv);
 8  int i, v0[10];
 9  for(i = 0; i < 10 ; i++) v0[i]=i+1;
10  msl::DistributedArray<int> r(10, v0);
11  r.scan(&plus_f);
12  msl::TerminateSkeletons();
13  return 0;
14 }
```

(b)

```
 1 #define ANAHY
 2 #include "Scan.h"
 3
 4 int* sum(int* x, int* y){ *x = *x+*y; return x;}
 5
 6 int main(int argc, char **argv){
 7  AnahyVM::init(argc, argv);
 8   int v0[10], *r, i;
 9   for(i=0; i< 10;i++) v0[i]=i+1;
10   kanga::Scan<int> *scan= new kanga::Scan<int>(sum,v0,r,10);
11   scan->start();
12   scan->join();
13   AnahyVM::terminate();
14   return 0;
15 }
```

(c)

Figura 40: Código da aplicação *Prefix Sum*: (a) SkePU, (b) Muesli e (c) Kanga.

```
 1 #include "skepu/vector.h"
 2 #include "skepu/reduce.h"
 3
 4 BINARY_FUNC(plus, int, x, y,return x+y;)
 5
 6 int main(){
 7  skepu::Reduce<plus> red(new plus);
 8  skepu::Vector<int> v0(10);
 9  for(int i = 0; i < 10; ++i) v0[i]=i+1;
10  double average = red(v0)/10;
11  return 0;
12 }
```

(a)

```
 1 #include "DistributedArray.h"
 2 #include "Muesli.h"
 3
 4 int plusM(int x, int y){ return x + y; }
 5
 6 int main(int argc, char** argv) {
 7  msl::InitSkeletons(argc, argv);
 8  int i, vetIn[10];
 9  for(i=0; i<10;i++)vetIn[i]=i+1;
10  msl::DistributedArray<int> v0(10, vetIn);
11  double average = v0.fold(&plusM)/10;
12  msl::TerminateSkeletons();
13  return 0;
14 }
```

(b)

```
 1 #define ANAHY
 2 #include "Reduce.h"
 3
 4 int* func(int* x, int* y){ *x=*x+*y; return x; }
 5
 6 int main(int argc, char** argv){
 7  AnahyVM::init(argc, argv);
 8  int v0[10],*dataOut, i;
 9  for(i=0; i< 10;i++) v0[i]=i+1;
10  kanga::Reduce<int> *red = new kanga::Reduce<int>(func,v0, dataOut,10);
11  red->start();
12  red->join();
13  double average = *dataOut/10;
14  AnahyVM::terminate();
15  return 0;
16 }
```

(c)

Figura 41: Código da aplicação Média Aritmética: (a) SkePU, (b) Muesli e (c) Kanga.

simples e sem adicionar novas sintaxes à linguagem de programação. Esse princípio foi seguido porque a implementação dos esqueletos somente utilizou mecanismos da linguagem C++. Além disso, os esqueletos são utilizados de forma simples por meio de objetos, os dados de entrada são passados pelos seus construtores e a execução é realizada pelos métodos start() e join().

O segundo princípio consiste em facilitar a integração dos esqueletos com o paralelismo específico de cada aplicação. A API desenvolvida tem o objetivo de estender a funcionalidades de Anahy3, assim as primitivas *fork* e *join* continuam sendo oferecidas. Além disso, a API oferece a classe Thread que funciona com a mesma interface dos esqueletos. Portanto, o programador poderá expressar o paralelismo da sua aplicação utilizando estes mecanismos em conjunto com os esqueletos.

Os esqueletos foram implementados para serem genéricos, assim o programador poderá utilizar os esqueletos com seus tipos de dados e determinar como devem ser manipulados, esta manipulação é feita por meio da funções que cada esqueleto recebe por parâmetro. Devido a isso o princípio de permitir a diversidade é mantido. Para mostrar o *pay-back* foram realizados testes nas seções 6.1 e 6.3 com o Kanga com o objetivo demostrar o seu poder de expressão e seu desempenho.

6.5 Conclusão

Os testes mostraram que a interface desenvolvida introduz um *overhead*, como era de se esperar, porém oferece um nível de abstração mais alto, sendo a principal vantagem desta interface em relação as ferramentas de mais baixo nível que oferecem um desempenho melhor. Mas em sistemas altamente paralelos podem se tornar complexas para o programador. Além disso, a API desenvolvida se beneficia dos mecanismos de escalonamento de Anahy3, o que poderá ser aferido de forma mais conclusiva em trabalhos futuros. Uma abordagem que pode ser adotada é introduzir no núcleo de escalonamento de Anahy heurísticas capazes de explorar informações sobre a estrutura do paralelismo gerado em cada esqueleto para auxiliar nas tomadas de decisões.

SkePU e Muesli definem estruturas para manipulação dos dados e a Kanga utiliza vetores convencionais da linguagem C++. A opção por usar vetores convencionais em Kanga reduz a necessidade do programador conhecer uma grande diversidade de tipos de dados, uma vez que entende-se que ele deve adicionar ao seu conhecimento as classes que descrevem os diferentes esqueletos oferecidos. No entanto, pode-se perceber que o uso de estruturas de dados específicas oferece maior poder de expressão e maior confiabilidade do progra-

mador.

Os esqueletos de Kanga precisam de uma série de parâmetros para descrever a entrada, em SkePU estas informações estão implícitas na abstração vector. Por outro lado, Kanga oferece uma maior liberdade ao programador para descrever sua própria estrutura de dados.

Outra diferença é que SkePU e Muesli disparam a execução concorrente na criação do objeto, e na Kanga o programador tem a possibilidade de aumentar a concorrência de sua aplicação devido a função do disparo (start) da execução concorrente ser separada da criação do objeto. Desta forma o programador pode criar vários esqueletos e dispará-los quando seu algoritmo, ou uma fase deste algoritmo, for internamente descrita.

7 CONSIDERAÇÕES FINAIS

Este trabalho apresentou os algoritmos de esqueletos paralelos e a utilização deles para o desenvolvimento da API Kanga. Foram estudados os esqueletos, que possuem como principais características: facilitar a portabilidade de código, ser independente de plataforma e facilitar o desenvolvimento de aplicações paralelas. No Capítulo 2 foram descritos os conceitos e os principais esqueletos encontrado na literatura. Os esqueletos selecionados para integrar a API implementada neste trabalho encontram-se descritos no Capítulo 3. O Capítulo 4 descreve a implementação da interface proposta, neste capítulo todas as classes implementadas são descritas e como elas estão organizadas em uma estrutura de hierarquia de classes.

As ferramentas de programação paralela que servem como suporte de execução para a API desenvolvida são descritas no Capítulo 5. Nesse capítulo, as características de Anahy3 e Pthreads são descritas juntamente com suas interfaces e suas estratégias de escalonamento.

Os testes realizados no Capítulo 6 fornecem resultados que servem para avaliar o poder de expressão e os custos causados pelo uso da API desenvolvida neste trabalho. Observamos, que o esqueleto Map não causa um grande aumento no tempo de execução mas o esqueleto DC pode causar um *overhead* considerável dependendo da aplicação. Foi demonstrado que os esqueletos podem ser compostos uns com os outros utilizando o padrão de projeto *Composite*. Ainda no Capítulo 6 foram implementadas aplicações utilizando os esqueletos da Kanga e dos *frameworks* SkePU e Muesli com objetivo de demonstrar que os esqueletos oferecidos por Kanga possuem poder de expressão equivalente aos esqueletos do SkePU e Muesli. Em particular, cita-se como diferencial da Kanga a possibilidade de aumentar o nível de paralelismo a ser explorado quando separa-se a ativação da concorrência de um esqueleto da sincronização com seu resultado.

Os resultados obtidos comprovam que a utilização de classes *templates* permitem a construção de uma API genérica orientada a objetos. Assim, a adição

de esqueletos ao Anahy3 facilitará a programação paralela, pois eles serão utilizados na forma de objetos, ou seja, os atributos e a implementação serão encapsulados e o programador somente precisará criar objetos por meio dos construtores. Os esqueletos simplificarão os projetos de aplicações paralelas porque oferecem soluções padrões de problemas recorrentes e, assim poderão ser combinados para construir aplicações mais robustas.

A API desenvolvida abstrai do programador as questões de programação paralela tais como criação e sincronização de *threads*. Mesmo assim, as aplicações desenvolvidas que utilizarão esta interface se beneficiarão dos mecanismos de escalonamento disponíveis no ambiente Anahy. Portanto, esta interface de programação oferece maior poder de expressão, permitindo ao programador obter todos os benefícios do Anahy para desenvolver um programa robusto, em particular o escalonamento de tarefas. Consequentemente o programador terá um aumento de produtividade e um aumento da robustez das aplicações que utilizam a ferramenta de programação paralela Anahy3.

7.1 Trabalhos Futuros

No futuro espera-se implementar esqueletos que considerem o funcionamento da estratégia de escalonamento de Anahy3 para que o desempenho desta ferramenta seja mantido mesmo utilizando os esqueletos. Além disso, pretende-se implementar os esqueletos Pipe e BB para aumentar o conjunto de esqueletos oferecidos.

Para permitir que o programador possa utilizar outras estruturas de dados como listas encadeadas com os esqueletos serão adicionados os iteradores da STL. A STL será integrada a Kanga devido ser uma biblioteca de estruturas padrão da linguagem C++. Logo, não será um empecilho para os programadores entenderem e utilizarem as estruturas de STL com a Kanga.

Como outro trabalho futuro pretende-se expandir os esqueletos de Kanga para outras plataformas, como OpenCL e MPI. Isto permitirá que o programador tenha a possibilidade de desenvolver uma aplicação que possa ser executada em diferentes plataformas de execução paralela.

REFERÊNCIAS

AOYAMA, Y.; NAKANO, J. **RS/6000 SP**: Practical MPI Programming. New York: International Technical Support Organization, 1999.

BENOIT, A.; COLE, M. Two Fundamental Concepts in Skeletal Parallel Programming. In: SUNDERAM, V. S.; ALBADA, G. D. van; SLOOT, P. M. A.; DONGARRA, J. (Ed.). **Computational Science – (5th ICCS'05, Part II)**. Reading, UK: Springer-Verlag (New York), 2006. p.764–771. (Lecture Notes in Computer Science (LNCS), v.3515).

BLELLOCH, G. **Prefix sums and their applications**. [S.l.]: Carnegie Mellon University, 1990. School of Computer Science Technical Report. (CMU-CS-90-190).

BLUMOFE, R. D.; JOERG, C. F.; KUSZMAUL, B. C.; LEISERSON, C. E.; RANDALL, K. H.; ZHOU, Y. **Cilk**: An Efficient Multithreaded Runtime System. Cambridge, MA, USA: [s.n.], 1996.

CAVALHEIRO, G. G. H.; GASPARY, L. P.; CARDOZO, M. A.; CORDEIRO, O. C. Anahy: A Programming Environment for Cluster Computing. In: VECPAR, 2006. **Anais...** [S.l.: s.n.], 2006. p.198–211.

CHANDRA, R.; DAGUM, L.; KOHR, D.; MAYDAN, D.; MCDONALD, J.; MENON, R. **Parallel programming in OpenMP**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

CIECHANOWICZ, P.; KUCHEN, H. Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures. In: HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS (HPCC), 2010 12TH IEEE INTERNATIONAL CONFERENCE ON, 2010. **Anais...** [S.l.: s.n.], 2010. p.108–113.

CIECHANOWICZ, P.; POLDNER, M.; KUCHEN, H. **The Münster Skeleton Library Muesli - A Comprehensive Overview**.

COLE, M. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. **Parallel Computing**, [S.l.], v.30, n.3, p.389–406, 2004.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms (3. ed.)**. [S.l.]: MIT Press, 2009. I–XIX, 1–1292p.

DAGUM, L.; MENON, R. OpenMP: an industry standard API for shared-memory programming. **Computational Science Engineering, IEEE**, [S.l.], v.5, n.1, p.46 –55, jan-mar 1998.

DASTGEER, U.; LI, L.; KESSLER, C. W. Adaptive Implementation Selection in the SkePU Skeleton Programming Library. In: ADVANCED PARALLEL PROCESSING TECHNOLOGIES - 10TH INTERNATIONAL SYMPOSIUM, APPT 2013, STOCKHOLM, SWEDEN, AUGUST 27-28, 2013, REVISED SELECTED PAPERS, 2013. **Anais...** Springer, 2013. p.170–183. (Lecture Notes in Computer Science, v.8299).

EMOTO, K.; MATSUZAKI, K. An Automatic Fusion Mechanism for Variable-Length List Skeletons in SkeTo. **International Journal of Parallel Programming**, [S.l.], p.1–18, 2013.

ENMYREN, J. **A Skeleton Programming Library for Multicore CPU and Multi-GPU Systems**. 103p.

ESKEL. **The Edinburgh Skeleton Library**. acessado em janeiro de 2013, http://homepages.inf.ed.ac.uk/mic/eSkel/.

FALCOU, J.; SÉROT, J.; CHATEAU, T.; LAPRESTÉ, J. T. Quaff: efficient C++ design for parallel skeletons. **Parallel Computing**, [S.l.], v.32, n.7–8, p.604–615, Sept. 2006.

FERREIRA, J. a. F.; SOBRAL, J. a. L.; PROENCA, A. J. JaSkel: A Java Skeleton-Based Framework for Structured Cluster and Grid Computing. In: CCGRID '06: PROCEEDINGS OF THE SIXTH IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID (CCGRID'06), 2006, Washington, DC, USA. **Anais...** IEEE Computer Society, 2006. p.301–304.

GAMMA; HELM; JOHNSON; VLISSIDES. **Design Patterns Elements of Reusable Object-Oriented Software**. Massachusetts: Addison-Wesley, 2000.

GAUTIER, T.; BESSERON, X.; PIGEON, L. KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-processors. In: INTERNATIONAL WORKSHOP ON PARALLEL SYMBOLIC COMPUTATION, 2007., 2007, New York, NY, USA. **Proceedings...** ACM, 2007. p.15–23. (PASCO '07).

HERRMANN, C. A.; LENGAUER, C.; ROBERT, C. Y. **HDC**: A Higher-Order Language for Divide-and-Conquer.

LEYTON, M. **Advanced Features for Algorithmic Skeleton Programming**. 2008. Tese (Doutorado em Ciência da Computação) — Universite de Nice - Sophia Antipolis – UFR Sciences.

LEYTON, M.; PIQUER, J. M. Skandium: Multi-core Programming with Algorithmic Skeletons. In: EUROMICRO CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING, 2010., 2010, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2010. p.289–296. (PDP '10).

LOOGEN, R.; MALLEN, Y. O.; MARí, R. P. Parallel Functional Programming in Eden. **Journal of Functional Programming**, [S.l.], v.15, n.3, p.431–475, 2005.

MCCOOL, M. D. Structured Parallel Programming with Deterministic Patterns. In: HOTPAR '10 (2ND USENIX WORKSHIP ON HOT TOPICS IN PARALLELISM), USB DATA STICK, 2010, Berkeley, CA. **Proceedings...** Usenix Assoc., 2010. Intel.

MCCOOL, M.; ROBISON, A. D.; REINDERS, J. **Structured Parallel Programming Patterns for Efficient Computation**. Wyman Street, Waltham, MA 02451, USA: Elsevier, 2012. 792p.

SEBESTA, R. W. **Conceitos de Linguagens de Programação**. [S.l.]: Bookman, 2011. 792p.

VÉLEZ, H. G.; LEYTON, M. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. **Softw, Pract. Exper**, [S.l.], v.40, n.12, p.1135–1160, 2010.