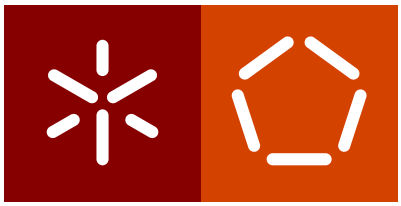
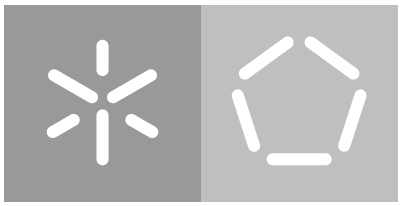


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Vasco Luzio Leitão

Padrões arquitecturais e de desenho para aplicações paralelas

Julho 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Vasco Luzio Leitão

Padrões arquitecturais e de desenho para aplicações paralelas

Dissertação de mestrado
Mestrado Integrado em Engenharia Informática

Dissertação orientada por
João Luís Ferreira Sobral

Julho 2021

AGRADECIMENTOS

A realização desta dissertação não seria possível sem a ajuda de várias pessoas, as quais gostaria de agradecer.

Ao meu orientador, Professor Doutor João Luís Sobral, por me ter guiado e motivado durante todo este processo, e por ter estado disponível sempre que foi necessário.

Por último, dirijo um agradecimento especial a toda a minha família pelo apoio incondicional durante todo o meu percurso académico.

ABSTRACT

Parallel programming is becoming increasingly common, given the evolution of hardware into multicore and manycore architectures. To take advantage of these architectures it is necessary to develop parallel code, since the automatic generation of parallel code from a sequential base code is not a viable option.

Parallel programming tends to be complex and therefore the developed code tends to be difficult to maintain or even difficult to reuse.

This dissertation aims to explore the use of algorithmic skeletons in the development of parallel applications in order to allow faster development as well as the production of higher quality final software. A skeleton framework was developed. The skeletons were developed through a technique called "mixin layers", which makes use of generic classes of C++. A great emphasis wass given to the use of generic classes and other alternatives in the construction of the skeletons.

Finally, the performance of these solutions were evaluated in comparison with solutions already consolidated within parallel computing. For that, the codes of a set of applications were restructured in order to make use of the skeletons provided by the library.

Keywords: software architecture, HPC, algoritmic skeletons

RESUMO

A programação paralela está a tornar-se cada vez mais comum, dada a evolução do hardware para arquiteturas *multicore* e *manycore*. Para tirar partido destas arquitecturas é necessário desenvolver código paralelo, uma vez que a geração automática de código paralelo a partir de um código base sequencial não é uma opção viável.

A programação paralela tende a ser complexa e por isso o código desenvolvido tende a ser de difícil manutenção ou até mesmo de difícil reutilização.

Esta dissertação pretende explorar utilização de esqueletos algorítmicos no desenvolvimento de aplicações paralelas de forma a permitir um desenvolvimento mais célere assim como a produção de um software final de maior qualidade. Foi desenvolvida uma biblioteca de esqueletos. Os esqueletos foram desenvolvidos através de uma técnica denominada "mixin layers", que faz uso de classes genéricas de C++. Foi dada uma grande ênfase à utilização destas classes genéricas e outras alternativas na construção dos esqueletos.

Por fim foi avaliada a performance desta soluções em comparação com soluções já consolidadas dentro da computação paralela. Para isso foram reestruturados os códigos de um conjunto de aplicações de modo a fazerem uso dos esqueletos disponibilizados pela biblioteca.

Palavras-chave: arquitectura de software, HPC, esqueletos algorítmicos

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

DECLARAÇÃO DE INTEGRIDADE

Declaro ter actuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

CONTEÚDO

1 INTRODUÇÃO 1
- 1.1 Contexto 1
- 1.2 Hipótese de investigação 2
- 1.3 Motivação 2
- 1.4 Objectivos 3

2 ESTADO DA ARTE 5
- 2.1 C++ moderno 5
  - 2.1.1 Programação genérica 5
  - 2.1.2 Funções objecto e lambdas 7
  - 2.1.3 Threads 7
  - 2.1.4 Aquisição de Recurso é Inicialização 7
- 2.2 Esqueletos algorítmicos 8
- 2.3 Frameworks e Bibliotecas 11
  - 2.3.1 SkePU2 11
  - 2.3.2 JaSkel 12
- 2.4 Arquitectura 12
- 2.5 Padrões de Desenho 13
  - 2.5.1 Padrão estratégia 13
  - 2.5.2 Padrão adaptador 14
  - 2.5.3 Mixin Layers 15

3 DESENVOLVIMENTO 18
- 3.1 Visão geral 18
  - 3.1.1 SKL1 19
  - 3.1.2 SKL2 20
  - 3.1.3 SKL3 21
- 3.2 Utilização da biblioteca de esqueletos 24
  - 3.2.1 Sintaxe e descrição do esqueletos 25
  - 3.2.2 Exemplos do uso de esqueletos esqueletos 27
  - 3.2.3 Interfaces 31
  - 3.2.4 Compilação e execução 31
- 3.3 Implementação 32
  - 3.3.1 SKL1 32
  - 3.3.2 SKL2 36

3.3.3 SKL3 41
4 RESULTADOS 48
4.1 Casos de estudo 49
4.1.1 AXPY 49
4.1.2 Kmeans 51
4.1.3 Multiplicação de Matrizes 56
5 CONCLUSÃO 59
5.1 Trabalho futuro 60

A TABELA DE RESULTADOS 63
B LISTAGENS 67
C FIGURAS 70

LISTA DE FIGURAS

Figura 1 Exemplo do padrão map 9
Figura 2 Exemplos dos dois tipos de reduções, sequencial à esquerda, em árvore à direita. 10
Figura 3 Exemplos do padrão *MapReduce*, à esquerda versão utilizando colecção temporária, à direita versão eficiente 10
Figura 4 Exemplo em UML do padrão estratégia 14
Figura 5 Exemplo em UML do padrão adaptador através de composição. 15
Figura 6 Exemplo em UML do padrão adaptador através de herança de classe 15
Figura 7 Exemplo de design baseado em colaborações. 16
Figura 8 Arquitectura da biblioteca SLK1. 19
Figura 9 Arquitectura da biblioteca SKL2. 20
Figura 10 Arquitectura da versão 3 da biblioteca. 22
Figura 11 Diagrama de classes da SKL1 versão argumentos. 32
Figura 12 Diagrama de classes da SKL1 versão parâmetros. 33
Figura 13 Diagrama de classes da SKL2, versão argumentos. 37
Figura 14 Diagrama de classes da SKL2, versão parâmetros. 41
Figura 15 Diagrama de classes da biblioteca SKL3. 42
Figura 16 Diagrama de classes da interface *Adapters* e dos adaptadores que a implementam. 43
Figura 17 Diagrama de classes da interface *Skeleton* e dos esqueletos que a implementam. 44
Figura 18 *Speedup* das várias versões do problema *axpy* paralelas em relação à versão sequencial sem esqueletos. 50
Figura 19 *Speedup* do problema *axpy* em relação à versão sequencial sem esqueletos, com tamanhos de problemas diferentes 51
Figura 20 *Speedup* das várias versões do problema *Kmeans* paraleleas em relação à versão sequencial sem esqueletos. 54
Figura 21 *Speedup* do problema *Kmeans* em relação à versão sequencial sem esqueletos, com tamanhos de problemas diferentes 55
Figura 22 *Speedup* das várias versões do problema *Matrix Multiplication* paralelas em relação à versão sequencial sem esqueletos. 57

Figura 23 *Speedup* do problema *Matrix Multiplication* em relação à versão sequencial sem esqueletos, com tamanhos de problemas diferentes 58
Figura 24 Escalabilidade das várias implementações da aplicação *axpy*. 70
Figura 25 Escalabilidade das várias implemntações da aplicação *kmeans*. 71
Figura 26 Escalabilidade das várias implementações da aplicação *matrix multiplication*. 71

LISTA DE TABELAS

Tabela 1 Descrição e sintaxe dos esqueletos/funcionalidades disponibilizados. 25
Tabela 2 Descrição e sintaxe dos esqueletos da versão SKL3. 26
Tabela 3 Descrição e sintaxe dos adaptadores da versão SKL3. 26
Tabela 4 Tempos de execução, em micro segundo, da aplicação *axpy* 64
Tabela 5 Tempos de execução, em micro segundos, da aplicação *kmeans* 65
Tabela 6 Tempos de execução, micro segundos, da aplicação *Matrix Multiplication* 66

LISTA DE LISTAGENS

2.1 Exemplo do uso de *templates* C++ para criar uma rotina genérica. 6
2.2 Exemplo do uso de *templates* C++. 6
2.3 Exemplo de como declarar o tipo da lambda. 7
2.4 Exemplo da utilização do esqueleto *map* na framework SkePU2. 12
2.6 Exemplo de uma *mixin layer*. 17
2.5 Exemplo de uma *mixin class*. 17
3.1 Exemplo de dois *maps*, passagem por parâmetros e por argumento 27
3.2 Exemplo do *map_i* 27
3.3 Exemplo das diferenças do *reduce* e *reduce2* 28
3.4 Exemplo *map_reduce* 29
3.5 Exemplo da fusão de dois *reduces* 30
3.6 Exemplo da fusão do *map* e do *reduce*, utilizando o adaptador *index* 30
3.7 Exemplo da refinação da implementação do *map_impl* por diversos tipos de paralelização. 35
3.8 Exemplo da implemntação rotina *map* disponibilizada pela API da biblioteca. 36
3.9 Implementação da especialização classe *t_range* 38
3.10 Implementação da classe *zip_iterator* 39
3.11 Implementação sequencial *map* passando o morfismo como argumento . . . 40
3.12 Implementação sequencial do Executor 42
3.13 Implementação sequencial do esqueleto *reduce* 45
3.14 Implementação do *mixin* que refina o Executor 46
4.1 Implemntação do algoritmo Kmeans utilizando a biblioteca de esqueletos SKL2 53
4.2 Implemntação do algoritmo Kmeans utilizando a biblioteca de esqueletos SKL3 53
B.1 Exemplo da implementação do algoritmo kmeans utilizando o TBB 67
B.2 Exemplo da implementação do algoritmo kmeans utilizando o OpenMP . . . 68

LISTA DE ABREVIATURAS

API Application Programming Interface. 18

BLAS Basic Linear Algebra Subprograms. 49

CUDA Compute Unified Device Architecture. 11

GPU Graphics Processing Unit. 11

HPC High Performance Computing. 11
HT Hyper-Threading Technology. 48

ILP Instruction-Level Paralelism. 1

MPI Mesage Passing Interface. 11

OpenCL Open Computing Language. 11
OpenMP Open Multi-Processing. xi, 3, 11, 45, 46, 50, 53, 55, 58, 60, 69

RAII Resource acquisition is initialization. 5, 7
RAM Random Access Memory. 48

SFINAE Substitution Failure Is Not An Error. 21

TBB Threading Building Blocks. xi, 3, 53, 60, 68

1

INTRODUÇÃO

1.1 CONTEXTO

A programação paralela está a tornar-se cada vez mais comum, dada a evolução do hardware para arquitecturas *multicore* e *manycore* e também ao facto de actualmente arquitecturas de suporte ao paralelismo se encontrarem presentes não só em supercomputadores mas também em computadores utilizados no dia-a-dia, como por exemplo em computadores pessoais e até em *smartphones*.

A computação paralela pode ser vista em duas perspectivas uma de mais baixo nível, o Instruction-Level Paralelism (ILP) que é explorado extensiva e eficazmente pelos processadores através de técnicas micro-arquitecturais como: *pipeline*, superescalaridade, execução fora de ordem, execução especulativa, entre outras.

Numa outra perspectiva de alto nível não existem soluções que consigam explorar automaticamente o paralelismo, tendo assim de ser o programador a expressar directamente o paralelismo através de *threads* e processos, uma vez que a geração automática de código paralelo através de um código base sequencial não é uma opção viável. Utilizar *threads* e processos na programação faz com que o código tenda a ser complexo, difícil de compreender e a ser propício a erros, como por exemplo *data-races* que costumam ser de difícil identificação e em alguns casos até mesmo de difícil resolução. Isto leva a que os códigos paralelos tendam a ser de difícil manutenção ou até mesmo de difícil reutilização.

Por outro lado a diversidade de arquitecturas de *hardware* paralelo exige formas diferentes de optimizar o código paralelo para atingir a melhor performance possível, que é o principal objectivo da computação paralela. Isto é para se obter a melhor performance possível é necessário que o programador tenha um bom conhecimento da arquitectura para o qual a aplicação se destina e das suas especificidades, o que leva a que as aplicações sejam um amalgamado de código dos algoritmos com código relacionado a um tipo de paralelização especifico de uma determinada arquitectura, o que reduz a portabilidade da aplicação.

Como foi dito anteriormente o suporte ao paralelismo é cada vez mais diversificado, é então necessário que as aplicações sejam capazes de o explorarem. É portanto necessário que qualquer programador seja capaz de produzir aplicações que consigam explorar todo

o potencial de paralelismo presente na máquina que as executa, quer esta tenha apenas dois cores ou milhares deles. Para isto é necessário uma forma de produzir código paralelo que abstraia o programador comum das dificuldades da computação paralela mas que ao mesmo tempo tenha uma boa performance e que seja uma solução escalável e portável.

1.2 HIPÓTESE DE INVESTIGAÇÃO

A comunidade científica identificou um conjunto de padrões que se repetem em códigos paralelos estes padrões denominam-se "esqueletos algorítmicos"[1].

Estes padrões[3] estão presentes numa grande diversidade de algoritmos, assim a paralelização de código torna-se similar para um diverso número de algoritmos, isto é, os algoritmos são desenvolvidos utilizando um conjunto de blocos básicos que permitem uma paralelização eficiente dos códigos. Ao contrario dos padrões de desenho estes esqueletos têm uma semântica bem definida o que permite que estes blocos básicos sejam disponibilizados por uma framework. Desta forma é possível o desenvolvimento de código bem estruturado.

O uso de esqueletos algorítmicos permite assim encapsular o potencial de paralelismo de uma aplicação, o que evita o utilizador de uma biblioteca de esqueletos o uso explicito de programação paralela. Desta forma as aplicações que fazem uso deste tipo *frameworks* baseadas em esqueletos tendem a reduzir a quantidade erros e a expressar o código com um maior nível de abstracção.

As implementações destes esqueletos podem ser feitas de diversas formas desde que respeitem a mesma semântica, isto leva a que seja possível ter diferentes implementações de um esqueleto que executam quer de forma sequencial quer paralela. Desta forma quanto melhor for a implementação dos esqueletos mais eficiente será o código desenvolvido com o uso destes.

1.3 MOTIVAÇÃO

Os esqueletos algorítmicos tem características interessantes em relação quer em relação á estruturação do código quer em relação a reutilização de código paralelo apesar disso tem os seus problemas.

A motivação desta dissertação é por um lado uma melhor estruturação da programação paralela o que leva ao uso dos esqueletos e por outro lado, por já existirem *frameworks* de esqueletos, a uma motivação sobre problemas no uso dos esqueletos. Este problemas serão divididos em duas grandes categorias, uma sobre desempenho e a outra sobre a usabilidade, portabilidade e manutenção da biblioteca.

Um dos problemas de performance no que existem com uso de esqueletos é a integração do código do utilizador. Esta problema deve-se ao facto de os esqueletos necessitarem de

segmentos de código de utilizador (especifico da aplicação/algoritmo). Desta forma os esqueletos recebem como argumentos partes desse código. O problema surge na utilização de apontadores de funções, que fazem com que os esqueletos sofram de problemas de indirecção, isto é, os esqueletos utilizam a função passada como argumento e esta é chamada em tempo de execução pelo nome, ou seja, o conhecido problema de *late binding*, que não possibilitará optimizações como a vetorização por parte do compilador. Outros dos problemas de performance que podem existir são as cópias redundantes de dados ao fazer o uso dos esqueletos e a fusão eficiente destes esqueletos.

Com o uso de esqueletos a aplicação passa a ser dependente da biblioteca para a paralelização do código, e uma vez que existem diversas formas de paralelizar um esqueleto dependendo da arquitectura de *hardware* alvo, a invocação do esqueleto não deve ser invasiva ao ponto de conduzir para um tipo de paralelização, ou sequer da própria paralelização, uma vez que estes esqueletos têm também um implementação sequencial. Assim uso da biblioteca requer a reestruturação do código sequencial, mas esta deve ser o menos invasiva possível, preferencialmente seria apenas necessário invocar os esqueletos.

Uma vez que como foi referido anteriormente existem diferentes tipos de paralelização consoante arquitectura alvo é necessário que a biblioteca consiga explorar todos estes tipos paralelização, ou que pelo menos tenha a capacidade para integrar novos tipo de paralelização. Mas mais do que isso, a biblioteca deve ser capaz de utilizar diferentes combinações dos diferentes tipos de paralelização, ou seja, tipos de paralelização híbridos. Isto apresenta um problema de manutenção pois se a biblioteca depender de uma implementação estática para cada tipo de paralelização e por conseguinte para cada combinação destas, de cada vez que um novo tipo de paralelização for implementado isso implica a implementação de um número exponencial de implementações de cada esqueletos.

1.4 OBJECTIVOS

Esta tese pretende desenvolver uma biblioteca para programação paralela baseada em esqueletos, com as seguintes características: deve ser capaz de fundir os esqueletos com código específico da aplicação da forma mais eficiente possível. Outro objectivo é a avaliação da biblioteca, para isso serão feitas comparações relativamente à forma de utilização do esqueletos; serão também feitas comparações da performance da versão sequencial da biblioteca com código sequencial normal e a comparação entre a versão paralela da biblioteca com outras tecnologias usadas na industria (p.e.,Open Multi-Processing (OpenMP), Threading Building Blocks (TBB)), para um conjunto de aplicações com características diferentes.

Sendo estes os objectivos principais desta dissertação de seguida serão apresentados sub-objectivos referentes à implementação da biblioteca:

1. Comparar duas formas de implementar esqueletos, relativamente à fusão do código do utilizador e do esqueleto, em que uma o código é passado como parâmetro de um *template*, ou seja, em tempo de compilação e a outra será passar o código do utilizador como argumento de uma função, ou seja, em tempo de execução.

2. Explorar uma serie de funcionalidades do standards C++11 e posteriores (inclusive o C++2a).

3. Implementar a biblioteca por camadas. Em caso de sucesso isto permitirá ter os diferente paradigmas de computação paralela em camadas diferentes. Este objectivo tem uma importância notória na portabilidade do código, dado que dependendo do ambiente de execução de uma aplicação, pode não ser possível utilizar algum dos paradigmas, por exemplo uma aplicação não terá possibilidade de utilizar uma paralelização baseada em memoria distribuída e desta forma basta "desactivar"a camada de associada a este tipo de paralelização.

4. Explorar as capacidades dos *templates* de *C++* de forma a utilizar *mixins* (explicado na secção Mixin Layers) na implementação dos esqueletos de forma a facilitar o objectivo 3.

2

ESTADO DA ARTE

Neste capitulo serão apresentados os conceitos fundamentais para o trabalho a desenvolver. Serão também apresentadas as principais funcionalidades relevantes da linguagem C++, em especial dos standards C++11 e posteriores, para a implementação deste trabalho. Por fim será também apresentado algum trabalho relacionado, como *frameworks* e bibliotecas de esqueletos.

2.1 C++ MODERNO

Neste secção serão abordadas as características do C++ moderno com mais relevância para uma implementação eficiente e simples de uma *framework* baseada em esqueletos. Entenda-se C++ moderno como os standards C++11 e posteriores, inclusive o C++2a.

Será então dada ênfase às seguintes funcionalidades:

2.1.1 Programação genérica;

2.1.3 *Threads*;

2.1.2 Funções como objectos;

2.1.4 Resource acquisition is initialization (RAII).

2.1.1 *Programação genérica*

A programação genérica é um estilo de programação em que o código é escrito sem se saber a priori qual o tipo dos dados a que os algoritmos se referem, ou seja utilizando tipos genéricos.

No âmbito da tese será dada mais relevância a classes genéricas, ou segundo [2] "tipos parametrizados que permitem que um tipo possa ser definido sem especificar todos os outros tipos que ele utiliza". Estas classes são fundamentais para a implementação de *Mixin Layers*.

A relevância deve-se ao facto de assim ser possível que uma classe estenda uma outra que é passada como parâmetro da classe genérica, ou seja, é possível ter uma classe derivada

que não dependa de uma classe base específica mas sim de uma serie de comportamentos ou dados que esta disponibiliza, como que se dependendo de uma classe abstracta.

Na Listagem 2.1 pode ver-se um exemplo do uso de *templates* para criar uma função genérica que soma dois objectos de tipos não especificados, neste caso de tipo genérico **T**. Este código é apenas compilado se o tipo **T** instanciado é um tipo aritmético ou um tipo cuja definição faz o *overload* do operador "**+**".

```
1 template <class T>
2 T sum(T x, T y)
3 {
4   return x + y;
5 };
```

Listagem 2.1: Exemplo do uso de *templates* C++ para criar uma rotina genérica.

A introdução de *templates* na linguagem trouxe uma serie de dificuldades. De forma a dar melhor suporte a estes problemas foram adicionadas duas novas palavras reservadas à linguagem:

auto - permite inferir o tipo de uma variável na sua declaração

decltype - fornece o tipo declarado de uma expressão

Um dos principais usos de *decltype* é escrever rotinas genéricas que possam retornar o resultado de outras sub-rotinas. Usando *decltype* é possível declarar o retorno da rotina genérica como sendo do mesmo tipo da sub-rotina. No excerto 2.2 pode ver-se um exemplo deste caso, em que são feitos dois *overloads* da rotina **foo**, em que um recebe um int e retorna a referência para int e a segunda recebe um float e retorna um float por valor, a rotina **bar** é genérica e portanto pode receber tanto ints como floats, retornando assim o mesmo que o *overload* seleccionado. A importância do uso do *decltype*, deve se ao facto de que sem este, isto é, fazendo a rotina genérica retornar um resultado do tipo **T**, o resultado seria sempre passado por valor.

```
1 int& foo(int& i);
2 float foo(float& f);
3
4 template <class T>
5 auto bar(T& t) -> decltype(foo(t)){ return foo(t); }
```

Listagem 2.2: Exemplo do uso de *templates* C++.

Como o argumento desta *keyword* é uma expressão, é possível declarar o tipo de uma função lambda, de notar uma vez que o resultado da *decltype* é um tipo não é possível utilizar em *lambdas* com capturas pois estas capturas são estado de uma instância específica. Na Figura 2.3 pode ver-se como o *decltype* pode ser utilizado para declarar o tipo de uma

lambda. No exemplo temos que **sum** é um tipo e assim pode servir como parâmetro de uma classe genérica.

```
1 using sum = decltype([](auto x, auto y){ x+y });
```

Listagem 2.3: Exemplo de como declarar o tipo da lambda.

2.1.2 *Funções objecto e lambdas*

Funções objecto, ou functores, são objectos que podem ser invocados como funções. Em C++ functores são obtidos fazendo o *overload* do operador "**()**".

Funções *lambda*, ou funções anónimas, são basicamente açúcar sintáctico para estes functores. Ainda assim tem uma característica que pode ser difícil de replicar em functores, que é estas permitirem o uso da *keyword auto* para especificar o tipo dos argumentos da *lambda*, o que as torna numa ferramenta poderosa quando aliada a técnicas de programação genérica. O compilador gera um tipo único para cada lambda.

Os functores, por serem objectos, introduzem várias vantagens relativamente às funções:

1. contêm estado o que é útil, por exemplo, para acumular valores entre chamadas (redução de valores de uma colecção);

2. podem ser herdados por outras estruturas, ou seja, é feito *static binding*, ao contrário do que aconteceria se a estrutura tivesse um apontador para uma função.

2.1.3 *Threads*

As *threads* passaram a ser directamente suportadas pela linguagem a partir standard C++11. As *threads* são fornecidas pelo standard por uma classe std::thread. O uso destas retira flexibilidade ao utilizador em comparação ao uso de por exemplo **pthreads**. O uso de std::therads é mais fácil dada a simplicidade da API disponibilizada de um API, para além disso o uso destas oferece uma maior portabilidade do código uma vez que esta classe abstrai o utilizador da sua implementação.

2.1.4 *Aquisição de Recurso é Inicialização*

RAII é um padrão de desenho, que associa a inicialização e finalização de um recurso ao tempo de vida de um objecto. Este padrão é conseguido através do construtor e destrutor de um objecto, assim um objecto contem um recurso que é inicializado no construtor e finalizado no destrutor. Este técnica é tipicamente utilizada para fazer a gestão de objectos

alocados dinamicamente e para libertar *mutex's* uma vez que o destrutor do objecto associado é chamado mesmo que tenha havido excepções, mas pode ser utilizado para uma variedade vasta de recursos, por exemplo inicialização e finalização de uma biblioteca.

2.2 esqueletos algorítmicos

Os esqueletos partilham várias similaridades com os padrões de desenho. Um das principais diferença é o facto de os esqueletos terem um semântica muito bem definida o que torna possível o desenvolvimento de bibliotecas de esqueletos. Portanto os esqueletos permitem, de facto, reutilizar código, o que não acontece com os padrões em que as soluções é que são reutilizáveis.

Como referido em [6] os programas sequenciais estruturados são baseados em três padrões de controlo de fluxo: o sequencial(1), a selecção(2) e a iteração(3). Estes padrões podem-se compor hierarquicamente, emergindo assim um outro padrão o aninhamento(4).

1. **Sequencial** é o padrão que se refere á ordem pela qual as tarefas são executadas, ou seja, este padrão visiona um programa com sendo um conjunto de operações que são executadas pela mesma ordem que aparece no código, independentemente da dependência de dados.

2. **Selecção** é um padrão condicional, isto é, o programa executará uma de duas tarefas consoante o resultado de uma condição.

3. **Iteração** é um padrão condicional, mas neste caso a condição é executada e no caso de esta ser falsa uma tarefa executada seguido de uma nova avaliação da condição, no caso contrario é apenas executada a tarefa seguinte.

4. **Aninhamento** é a composição hierárquica de outros padrões, assim encontra-se quando as tarefas que derivam são também padrões.

Os esqueletos algorítmicos resultam então da paralelização destes padrões sequenciais, ou no caso de esqueletos algorítmicos mais avançados da paralelização de uma composição destes. Embora seja necessário relaxar estas premissas de forma a que a paralelização seja possível.

Muitos dos esqueletos mais básicos provém da paralelização do padrão iterativo, como é o exemplo dos padrões *map*, *reduction*, *scan*, *recurrence*, *scatter*, *gather* e *pack*. Este padrão dá origem a este vasto conjunto de esqueletos devido às diferentes formas de como as dependências surgem entre iterações.

1. **Map** é o esqueleto mais simples e é também embaraçosamente paralelo, este baseia-se em aplicar um morfismo a todos os elementos de uma colecção. Este esqueleto está

associado ao padrão de controlo de fluxo iterativo e é o caso particular em que não existem dependências entre as diversas iterações do ciclo. A Figura 1 exemplifica a semântica paralela do esqueleto *map*, aplicado a uma colecção representada pelo rectângulo a verde, em que os quadrados azuis representam a transformação através de um morfismo dos elementos de uma colecção representados como quadrados verdes, as tarefas azuis podem assim executar em paralelo uma vez que não existem dependências ente elas.

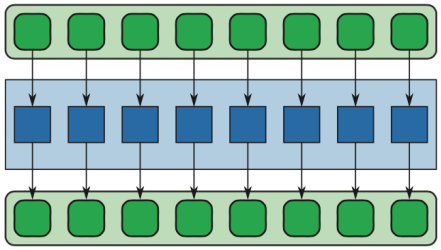


Figura 1: Exemplo do padrão map

2. **Reduce** este padrão combina todos os elementos de uma colecção num só elemento. Existem dois métodos para implementar este padrão:

   em árvore, neste é usada uma operação de redução que recebe dois elementos da colecção, esta operação utilizada para combinar todos os elementos da colecção dois a dois numa nova colecção repetindo este passo até que o resultado seja apenas um elemento;

   serialmente, nesta implementação a operação de redução opera sobre um elemento da colecção e um acumulador, desta forma a colecção é iterada serialmente acumulando todos o elementos, nesta implementação o acumulador pode não ser do mesmo tipo dos elementos da colecção.

   De notar ainda que as propriedades algébricas como comutatividade e associatividade da operação de redução desempenham um papel fundamental na correcção da implementação paralela deste padrão. A Figura 2 mostra os dois tipo de redução, à esquerda a versão acumulativa e à direita a redução em árvore. Como se pode ver para o resultado das duas versões ser o mesmo é necessário as propriedades algébricas de comutatividade, uma vez que na primeira figura a segunda tarefa (a azul) depende dos primeiros três elementos (a verde), enquanto que na redução em árvore a segunda tarefa depende apenas de dois elementos o segundo e o quarto da colecção. Pode observar-se também que na redução em árvore o paralelismo emerge naturalmente uma vez que as dependências não são sequenciais.

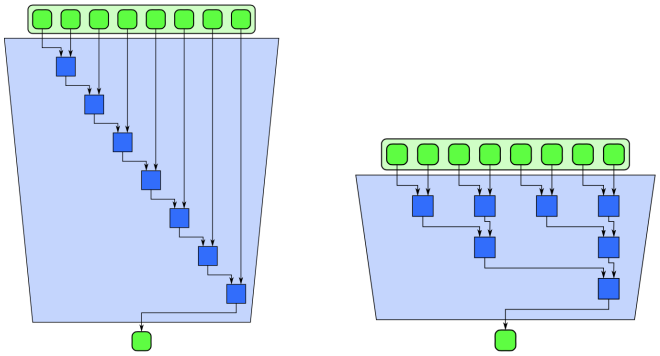


Figura 2: Exemplos dos dois tipos de reduções, sequencial à esquerda, em árvore à direita.

3. **MapReduce** resulta da combinação do *map* e do *reduce*, ou seja, este algoritmo aplica um morfismo a todos os elementos de uma colecção e de seguida combina todos os elementos num único. Uma versão eficiente deste padrão seria aplicar o morfismo a um elemento e combiná-lo no elemento final de imediato, de modo a aumentar a localidade temporal da implementação. Este é um dos padrões mais recorrentes em aplicações reais. Na Figura 3 pode ver-se à esquerda uma implementação ineficiente do *MapReduce* em que é primeiro executado o *map* e de seguida o *reduce*. À direita implementação eficiente do esqueleto em que é feita a fusão dos dois esqueletos, ou seja, não existe necessidade de uma colecção temporária para armazenar os resultados.

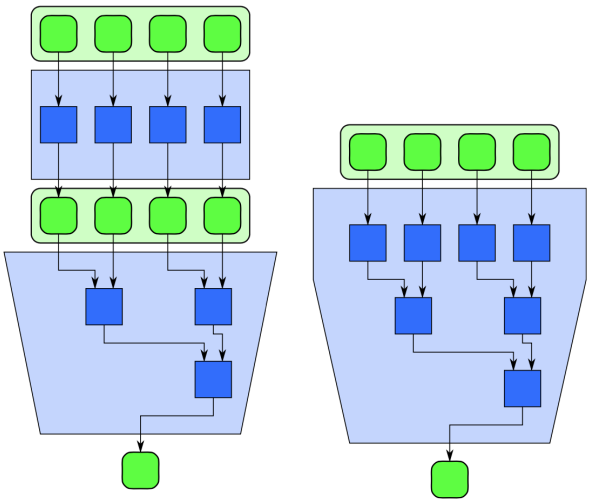


Figura 3: Exemplos do padrão *MapReduce*, à esquerda versão utilizando colecção temporária, à direita versão eficiente

2.3 FRAMEWORKS E BIBLIOTECAS

Os esqueletos começaram por ser desenvolvidos como estrutura de controle de linguagens, ou seja, através da criação de linguagens de domínio específico. Recentemente é mais comum a criação da *framework* e bibliotecas em linguagens orientadas a objectos como C++ e *Java*. Ainda assim existem várias implementações para linguagens imperativas como C ou para linguagens funcionais como *Haskell*. Estas *frameworks* tendem a ser implementadas usando tecnologias maduras no mundo do High Performance Computing (HPC), como é o caso do Mesage Passing Interface (MPI). Sendo que a utilização destas bibliotecas tende a reduzir o número de erros comparando com o caso de se utilizarem directamente esses paradigmas de mais baixo nível utilizados tradicionalmente em HPC.

As *frameworks* tendem a variar no conjunto de esqueletos que disponibilizam, ainda assim é clara a recorrência de algum esqueletos, como *farm*, *pipe*, e *map*. De salientar que um dos aspectos mais importantes do sucesso destas implementações é o facto de permitirem o aninhamento dos esqueletos.

2.3.1 *SkePU2*

O *SkePU2* é uma *framework* de programação de esqueletos para multi-cores e sistemas multi-GPU. A *framework* é implementada como uma biblioteca genérica em C++. De modo a paralelizar o código é necessário uma compilação *source-to-source* do código do *SkePU*, é também necessário o uso de um *Parallel backend*, que de acordo com [4] é capaz de permitir uma execução híbrida e é também *performance-aware*, ou seja, é capaz de fazer escalonamento dinâmico e balancear a carga. Para além de C++ a *framework* utiliza tecnologias como: OpenMP para programação multi-core, Open Computing Language (OpenCL) para aceleradores de *hardware* e Graphics Processing Unit (GPU), assim como Compute Unified Device Architecture (CUDA) para a última arquitectura.

O *SkePU2* implementa sete esqueletos algorítmicos, sendo que seis destes esqueletos referem-se a paralelização sobre dados (*Map*, *Reduce*, *MapReduce*, *Stencil* e *Scan*) e apenas um assente em paralelismo sobre tarefas (*Call*).

A implementação dos esqueletos faz-se com recurso a functores, isto é, classes em que é feito o *overload* do operador "**()**", deste modo uma instância desta classe pode ser chamada como se de uma função se tratasse. Por exemplo o esqueleto *map* é definido como sendo uma classe genérica, em que uma das variáveis da instância é um apontador para uma função.

A instanciação de um objecto desta classe é feita passando como argumento ao construtor o apontador da função. A diferença entre dois *maps* é assim o valor do apontador da função, comportando-se assim como se de uma função virtual se tratasse, ou seja, o compilador só

```
1 float sum(float a, float b) { return a + b; }
2
3 Vector<float> vector_sum(Vector<float> &v1, Vector<float> &v2)
4 {
5   auto vsum = Map<2>(sum);
6   Vector<float> result(v1.size());
7   return vsum(result, v1, v2);
8 }
```

Listagem 2.4: Exemplo da utilização do esqueleto *map* na framework SkePU2.

tem acesso a definição da função em tempo de compilação (*late-binding*). A instanciação de um *map* também necessita da aridade do morfismo ou número de colecções que se pretende iterar, este valor é passado como parâmetro da classe genérica.

Uma instância do esqueleto *map* é executada quando é chamada com os seguintes argumentos:

1. colecção onde serão colocados os resultados,
2. colecções cujos elementos serão aplicados ao functor, o número colecções é a aridade passada à classe genérica *map*.

No código 2.4 pode ver-se a utilização do padrão *map* para computar um vector, cujos elementos são a soma ponto a ponto de dois vectores. Neste caso o functor é a função *sum*. Uma vez que a aridade desta função é um parametrizada com o valor dois na instanciação do *map*(linha 5).

2.3.2 *JaSkel*

JaSkel é outra *framework* de esqueletos mas esta é baseada em *Java*, fomentada com a linguagem *AspectJ*, e tem como foco computação em *clusters* e em *grid*. Esta *framework* disponibiliza os padrões, *farm*, *pipe* e *heartbeat*.

Esta *framework* segue uma abordagem diferente, neste caso os esqueletos são de definidos como classes abstractas, desta forma para utilizar um destes esqueletos é necessário estender a sua classe abstracta e acrescentar código referente á aplicação num conjuntos de métodos.

Esta forma de definir os esqueletos através da extensão de uma classe base torna o código do utilizador verboso. Esta forma obriga assim á criação de novas classes para cada tipo de esqueleto.

2.4 ARQUITECTURA

> *"O principal objectivo da arquitectura software é minimizar os recursos humanos necessários para desenvolver e manter o sistema."*

- Robert Cecil Martin, *Clean architecture[5]*

Os padrões de desenho partem de um conjunto de boas práticas da programação no desenvolvimento de software e assentam principalmente sobre o ideal de reusabilidade de soluções para um dado problema. Os padrões visam também estabelecer um vocabulário comum face a um problema, o que facilita a comunicação, documentação e compreensão de um sistema de software.

Como referido em [6], estes padrões são compostos por duas partes, a semântica, que se refere ao que este pretende desempenhar, e a implementação, que é a forma como se consegue cumprir a primeira. A implementação é tipicamente dependente da arquitectura do hardware, da linguagem de programação e do sistema, em particular nos padrões paralelos, apesar de a semântica não o ser.

A arquitectura de *software* de um sistema baseia-se na definição dos componentes de *software*, isto é das suas propriedades e das suas colaborações. O foco principal da arquitectura é então a definição de como um sistema deve ser organizado, ou seja, a estrutura geral do sistema.

A arquitectura que melhora se adequa ao caso do código paralelo é uma arquitectura organizada por camadas. Esta arquitectura permite separar o código em módulos de modo que estes possam ser reutilizados. Para isto ser possível, é importante que as camadas não dependam da implementação da camada inferior mas sim de uma interface que ambas implementem [5].

Estas camadas, numa linguagem orientada ao objecto, podem ser representadas por um conjunto de classes. Um caso particular da implementação deste padrão baseado em colaborações são os *Mixin Layers*.

2.5 PADRÕES DE DESENHO

Neste capítulo serão apresentadas alguns dos principais padrões de desenho, que permitiram a implementação de uma biblioteca de esqueletos estruturada por camadas. As implementações apresentadas, são baseados na linguagem de programação C++, uma vez que foi esta a linguagem com que a biblioteca de esqueletos foi desenvolvida.

2.5.1 *Padrão estratégia*

O padrão estratégia consiste em agrupar um conjunto de comportamentos, de modo a que estes possam ser trocados entre si eventualmente tempo de execução. A principal vantagem deste padrão é a possibilidade de delegar um comportamento de uma classe para o de um objecto concreto. Isto significa que a invocação de um método por um cliente pode resultar

em comportamentos diferentes (da mesma família) em tempo de execução consoante a instanciação de uma classe concreta da estratégia num determinado momento da execução.

Como se pode ver Figura 4, o método *operation()* da classe StategyClient delega a execução para o método *operation()* de uma instância de uma qualquer das classes ConcreteStrategyX.

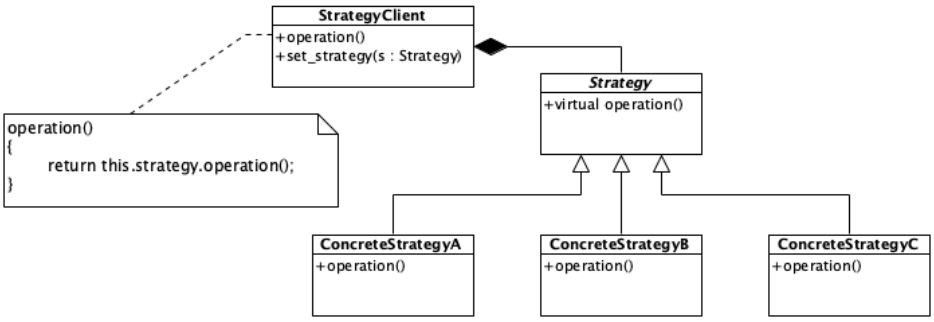


Figura 4: Exemplo em UML do padrão estratégia

2.5.2 *Padrão adaptador*

Este padrão é utilizado quando uma classe cliente necessita de utilizar um método de uma outra classe (adaptado) mas as interfaces não coincidem. Para resolver este problema é criada uma interface alvo com os métodos que o cliente usa, desta forma e possível implementar uma classe intermediária entre o cliente e o adaptado que impelente esta interface, ou seja possa ser utilizada pelo cliente mas em que o seu método redireccione a invocação para o método da classe adaptado.

Mais concretamente, é possível obter este comportamento de duas formas como é ilustrado nas Figuras 5 e 6, respectivamente através de composição e por herança. No primeiro o Adpater recebe como parâmetro um Adaptee e quando o AdapterClient invoca o método *operation()* o Adapter delega este comportamento para o método *specific_operation()* da instância do Adptee. No caso da herança, o Adpater estende directamente o Adaptee assim não necessita de uma instância deste e ao ser invocado o método *operation()* este invoca directamente o método *specific_operation()*, podendo ser feito o *inline* deste método.

A utilização da versão composição oferece uma maior flexibilidade apesar disso possivelmente apresenta uma pior performance.

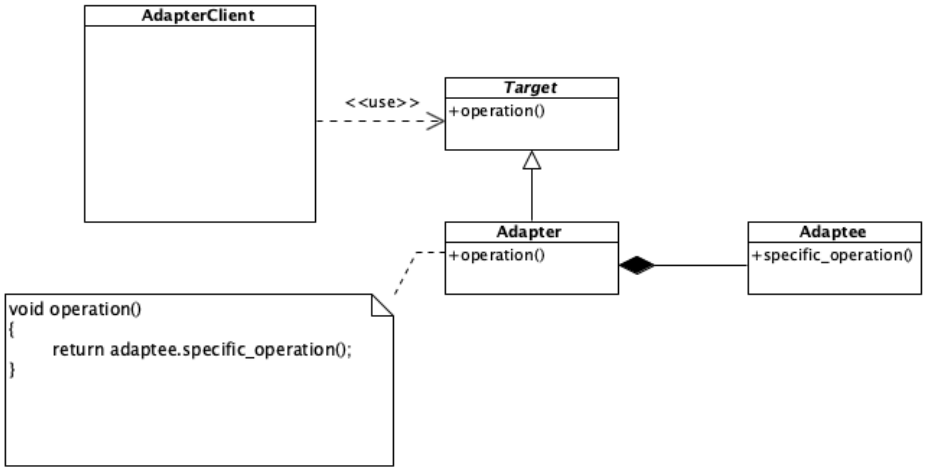


Figura 5: Exemplo em UML do padrão adaptador através de composição.

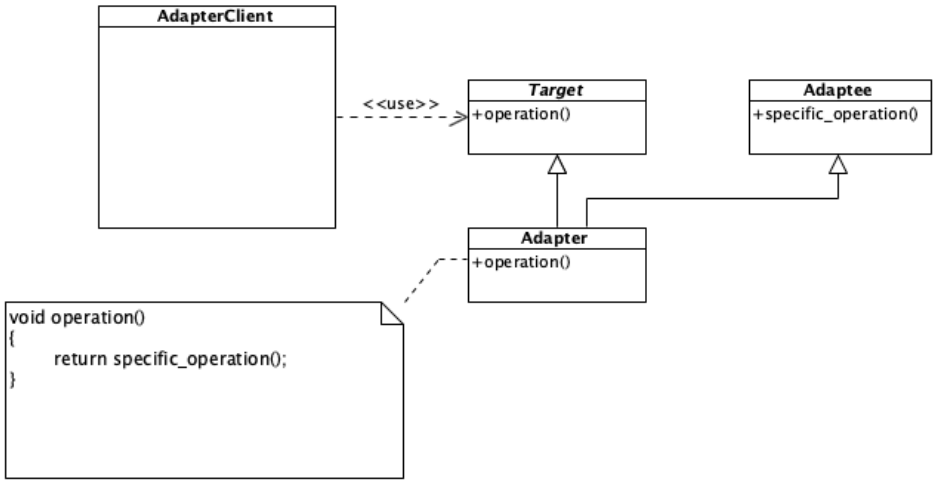


Figura 6: Exemplo em UML do padrão adaptador através de herança de classe

2.5.3 *Mixin Layers*

Os *Mixin Layers* são uma implementação de um design baseado em colaborações. Desta forma, tem-se uma *layer* base que é refinada de modo a que seja acrescentado novos comportamentos a essa *layer*, criando a assim uma nova *layer*, que pode também ser refinada e assim sucessivamente. Um refinamento é uma adição a um projecto de software que pode influir sobre diversas entidades do projecto, por exemplo funções ou classes.

A Figura 7 representa um projecto baseado em colaborações, em que as *layers* são as figura ovais e são definidas por diversas entidades, os *roles* na figura. Estes *roles* são então adições às

entidades da *layer* anterior ou até novas entidades que podem ser utilizadas pelas entidades posteriores. De notar que uma colaboração não necessita de adicionar comportamento a todas as entidades da colaboração anterior, como é o caso da colaboração **C2** em relação à entidade **OC**. Também é possível uma colaboração refinar uma entidade sem que esta esteja directamente presente na colaboração imediatamente inferior, como é o caso da colaboração **C3** em relação à entidade **OC** que não esta presenta na colaboração **C2**.

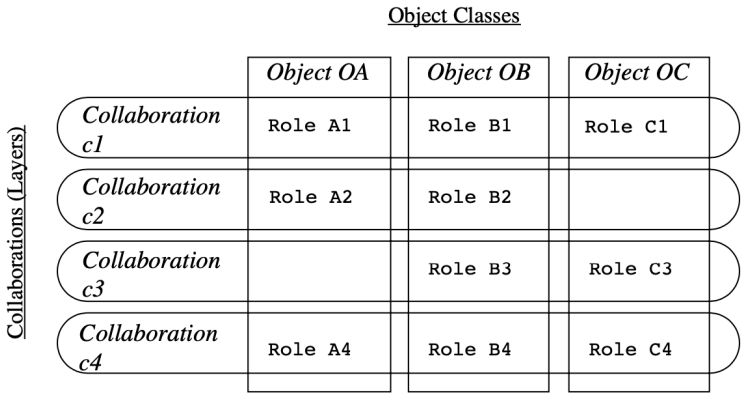


Figura 7: Exemplo de design baseado em colaborações.

Implementação em C++

A ideia principal desta implementação é, em linguagens baseadas no paradigma orientado a objectos, ter uma classe derivada que pode ser definida sem se especificar a classe base. Desta forma uma classe derivada pode servir como extensão de múltiplas classes base diferentes, como acontece normalmente, no caso contrario, em que um classe base serve de base para múltiplas classes.

A implementação de *Mixin Layers* faz uso deste conceito em dois níveis os *Inner Mixins*, também denominados *Mixin Classes* e os *Outer Mixin* que, de facto, são os *Mixin Layers*.

Implementação de Mixin Classes

A implementação de *mixin classes* para a linguagem C++ pode ser obtida, como referido em [7], através da parametrização de uma classe genérica, sendo o parâmetro desta classe a classe base que se pretende estender.

No excerto de código 2.5 pode ver-se a definição de uma *mixin class*, isto é, uma classe genérica denominada **RoleDerived** que é parametrizada com uma classe base **RoleBase** que é estendida na linha dois. Assim no corpo de **RoleDerived** podem ser utilizadas a funções e variáveis declaradas na classe base, ou acrescentar as suas.

```
1 template <class CollabBase>
2 class CollabDerived : public CollabBase
3 {
4   public:
5   class  FirstRole : public CollabBase::FirstRole { ... };
6   class SecondRole : public CollabBase::SecondRole { ... };
7   class  ThirdRole : public CollabBase::ThirdRole { ... };
8   ... // more roles
9 };
```

Listagem 2.6: Exemplo de uma *mixin layer*.

```
1 template <class RoleBase>
2 class RoleDerived : public RoleBase
3 {
4 ... /* add funcionality */
5 /* or */
6 .../* use funcionality from RoleBase*/
7 };
```

Listagem 2.5: Exemplo de uma *mixin class*.

Implementação de Mixin Layers

As *mixin classes* não são suficientemente capazes concretizar este design baseado em colaborações devido a problemas de escalabilidade como referido no artigo [8], em que é proposta uma forma alternativa mais viável, os *mixin layers*.

As *mixin layers*, baseiam-se no mesmo conceito que, ou seja, são também uma classe genérica que é parametrizada com uma classe base. Apenas com a diferença que esta classe é composta por sub classes. Assim a classe derivada para além de adicionar ou utilizar variáveis e funções da classe base, pode também estender os comportamentos dessa classe base, ou seja, estender ou utilizar as subclasses da classe base. Uma *mixin layer* é então uma *mixin class*, com a particularidade de esta ser a composição de um conjunto de *sub mixin classes*.

No excerto 2.6 pode observar-se a declaração de uma *mixin layer*, a **CollabDerived**. Esta colaboração é então uma classe genérica que recebe como parâmetro uma classe base que representa a colaboração que se pretende refinar. A colaboração é um conjunto de vários *roles*, ou seja, uma classe composta por várias sub classes, como se pode ver no excerto a classe **CollabDerived** é composta pelas classes **FirstRole**, **SecondRole** e **ThirdRole**. Estas sub classes são elas próprias *mixin classes*, uma vez que estendem uma classe genérica, mas neste caso não precisam de ser parametrizadas pois estendem uma classe que é uma sub classe da parametrização utilizada no *outter mixin*, ou seja, da classe **CollabBase**.

3

DESENVOLVIMENTO

3.1 VISÃO GERAL

Um dos principais objectivos desta dissertação é o desenvolvimento de uma biblioteca que facilite/agilize a implementação algoritmos paralelos e cujo código seja portável para um conjunto variado de arquitecturas de hardware, mantendo a eficiência. Assim sendo, esta é uma biblioteca de esqueletos algorítmicos, em que a implementação destes é feita por camadas, onde cada camada implementa um conjunto de conceitos, por exemplo, o tipo de paralelização. Estas camadas servem como forma de virtualizar a implementação dos esqueletos (em tempo de compilação), de modo que é possível atingir um elevado grau de portabilidade sem perda de performance.

De seguida são apresentados os principais requisitos desta biblioteca.

1. Suporte aos principais esqueletos:
   a) map;
   b) reduce;
   c) mapreduce (preferencialmente através da fusão dos anteriores).
2. Interface de utilização simples, flexível e extensível:
   a) Application Programming Interface (API) de estilo funcional;
   b) iteração de várias colecções em simultâneo;
   c) utilização de functores capazes de identificar o índice da iteração.
3. Arquitectura de software baseada em camadas:
   a) implementação de uma arquitectura por camadas semelhante aos *mixin layers* capaz de modularizar/separar os diferentes tipos de código (p.e., sequencial, paralelo em memória partilhada, paralelo em memória distribuída).
4. Desempenho:
   a) execução eficiente do código para os pelos esqueletos;

b) fusão eficiente dos esqueletos.

O desenvolvimento da biblioteca de esqueletos passou por várias etapas, das quais resultaram três versões da biblioteca. Esta secção apresenta a arquitectura de cada uma dessas versões, assim como as suas características e os seus principais componentes.

3.1.1 *SKL1*

Nesta primeira versão foram implementados apenas os esqueletos *map* e *reduce*, assim como uma forma simples de os disponibilizar ao utilizador, respeitando a arquitectura por camadas proposta nos requisitos.

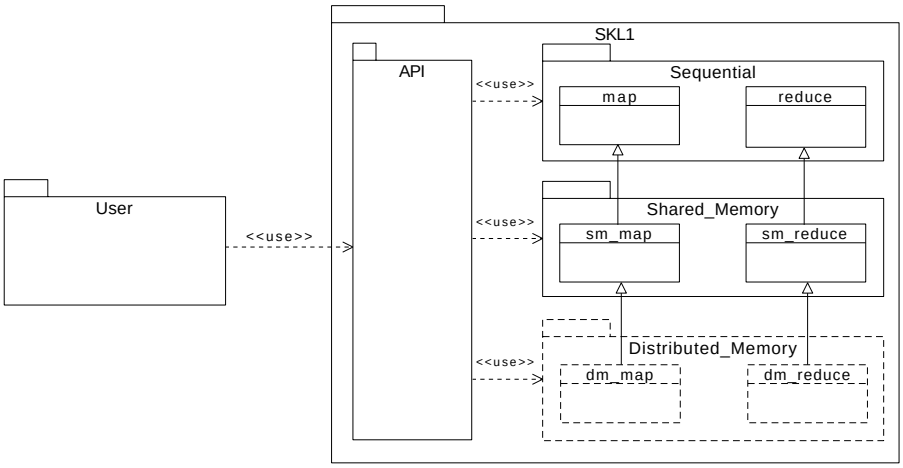


Figura 8: Arquitectura da biblioteca SLK1.

A paralelização destes esqueletos é feita com recurso a memória partilhada fazendo o refinamento através da extensão por herança das classes *map* e *reduce*, respectivamente pelas classes *sm_map* e *sm_reduce*, como ilustra a Figura 8.

Observe-se que a implementação da biblioteca assemelha-se às *Mixin Layers*, isto é, a biblioteca é construída por camadas. Estas camadas são constituídas por classes, as camadas são refinadas através da extensão por herança das suas classes. Ou seja, as classes implementam quer os esqueletos, quer as suas refinações. Estas *inner classes* são *Mixin Classes*, ou seja não são dependentes de uma classe base concreta o que permite remover as camadas intermédias ou trocar a ordem com que o refinamentos são feitos.

Para além destes requisitos suporte dos esqueletos (1a, 1b) e da arquitectura por camadas (3a), foi também implementado o requisito disponibilizar os esqueletos com um estilo funcional 2a. Esta é a componente *API* da figura, esta componente apenas simplifica/reduz a quantidade de código na utilização dos esqueletos.

Note-se que na figura a camada de memória distribuída está a tracejado dado que não está implementada, estando apenas representada na figura para se poder observar como seria a sua integração nesta arquitectura.

3.1.2 *SKL2*

A versão SKL1 apesar de cumprir um dos requisitos fundamentais, a separação do código por camadas, apresenta uma fraca usabilidade. Os esqueletos só podem ser aplicados a uma colecção de cada vez, motivo pelo qual estes esqueletos não são capazes de implementar os algoritmos apresentados no capitulo 4 sem que se tenha de adaptar as estruturas utilizadas, ao invés de ser o esqueleto adaptado.

Esta segunda versão é um melhoramento da SKL1, é então simplesmente um melhoramento, quer a nível da estruturação interna da biblioteca, quer das suas funcionalidades. Os requisitos adicionais implementados nesta versão foram os seguintes: iteração com índices 2c, iteração de várias colecções em simultâneo 2b e implementação do esqueleto *mapreduce* 1c(hardcoded).

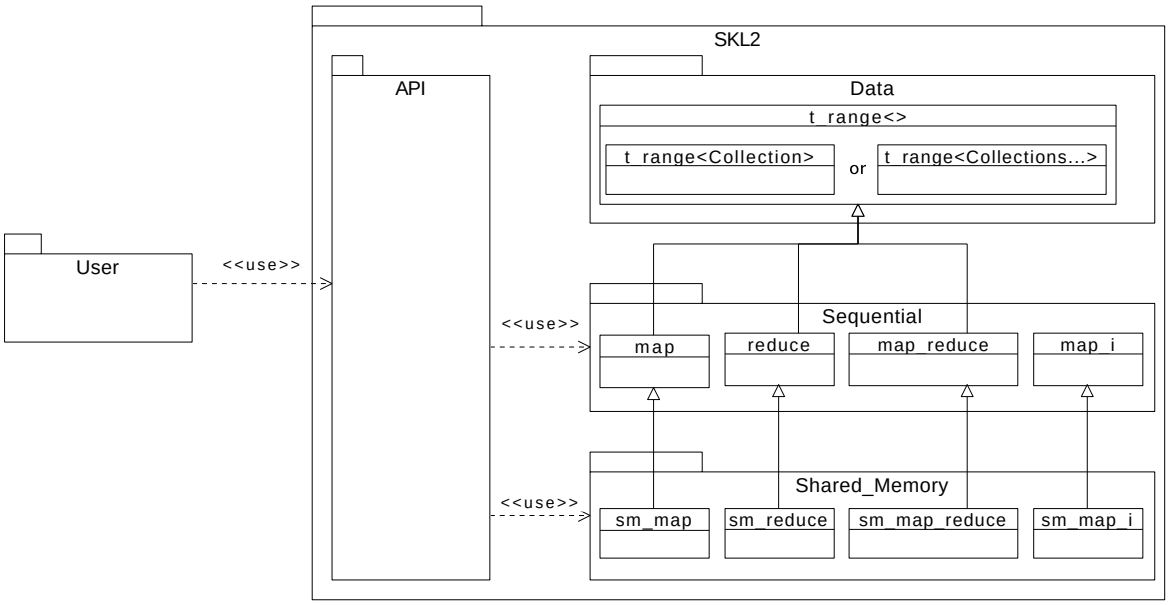


Figura 9: Arquitectura da biblioteca SKL2.

A iteração através de índices foi implementada como um novo esqueleto, ou seja, foi implementada um a variante do *map*, o *map_i*. Para que os outros esqueletos também poderem pudessem ter esta variante, teriam de ser implementas variações de todos os esqueletos. Note-se que a implementação do *map_i* é semelhante à do *map*, isto é, existe uma duplicação do código do esqueleto original.

Nesta versão, a iteração de múltiplas colecções em simultâneo é um comportamento comum a todos os esqueletos. De forma a evitar a replicação deste comportamento em novas implementações de variantes do esqueleto, este comportamento foi implementado numa nova classe genérica o *t_range*, que é estendida pelos esqueletos, dando assim origem a uma nova camada *Data*, que fica responsável pela iteração dos dados. O comportamento foi implementado com recurso a uma estrutura que agrega os vários iteradores das respectiva colecções, esta estrutura funciona como um compósito, isto é, tem a mesma interface dos iteradores que o constituem e quando os métodos são invocados esta invoca-os, respectivamente, a todos os seu iteradores. A utilização desta estrutura poderia ter uma perda de performance no caso de apenas se iterar uma colecção. Para evitar esta possível perda de performance a classe genérica foi implementada com duas especializações, *t_range<Collection>* e *t_range<Collections...>* respectivamente para a iteração de uma única colecção e para a iteração de múltiplas colecções. Estas especializações são automaticamente seleccionadas pelo compilador através da regra Substitution Failure Is Not An Error (SFINAE), inferida sobre os tipos dos argumentos com que o esqueletos é instanciado, funcionando assim como *plugable adapters* implícitos. Em suma, estes *pluggable adapters* são automaticamente estendidos pelo esqueleto de forma a adaptá-los às necessidades do utilizador.

Nesta versão foi implementado também o *mapreduce*, foi implementado por um esqueleto novo, ou seja, por uma nova classe. Este esqueleto é uma amalgama dos códigos dos esqueletos *map* e *reduce*. Apesar disso é uma forma eficiente de implementar a fusão destes esqueletos a nível de performance.

3.1.3 *SKL3*

A versão anterior apresenta dois problemas fundamentais/arquitecturais. O primeiro é a utilização de índices. Como pode ser observado na Figura 9, apenas o *map_i* suporta esta funcionalidade, o que se deve ao facto de essa funcionalidade estar directamente acoplada ao esqueleto ao invés de ser conectável. Ou seja, para que outros esqueletos suportem esta funcionalidade, com a arquitectura apresentada anteriormente, seria necessário implementar uma cópia dos esqueletos existentes com, apenas, algumas diferenças mínimas na forma de como a iteração implementada.

O outro problema é a falta de flexibilidade que a implementação do *mapreduce* apresenta. Esta falta de flexibilidade deve-se ao facto de o *mapreduce* apenas aceitar dois functores, um para aplicar. o *map* e o outro para aplicar o *reduce*, o que significa, por exemplo, que no caso de o utilizador ter vários functores que pretende aplicar aos elementos de uma colecção, este tem de implementar um novo functor que faz a composição desses. Pelo contrario a fusão de esqueletos permite ao utilizador fundir um número arbitrário de *maps* com um número arbitrário de *reduces*. A fusão de esqueletos, já implementados, permite

uma melhor estruturação do código uma vez que permite usar functores mais pequenos e menos complexos, o que os torna mais reutilizáveis uma vez que ficam menos específicos da aplicação. Para além disso aumenta a abstracção do uso dos esqueletos e melhora expressividade do código.

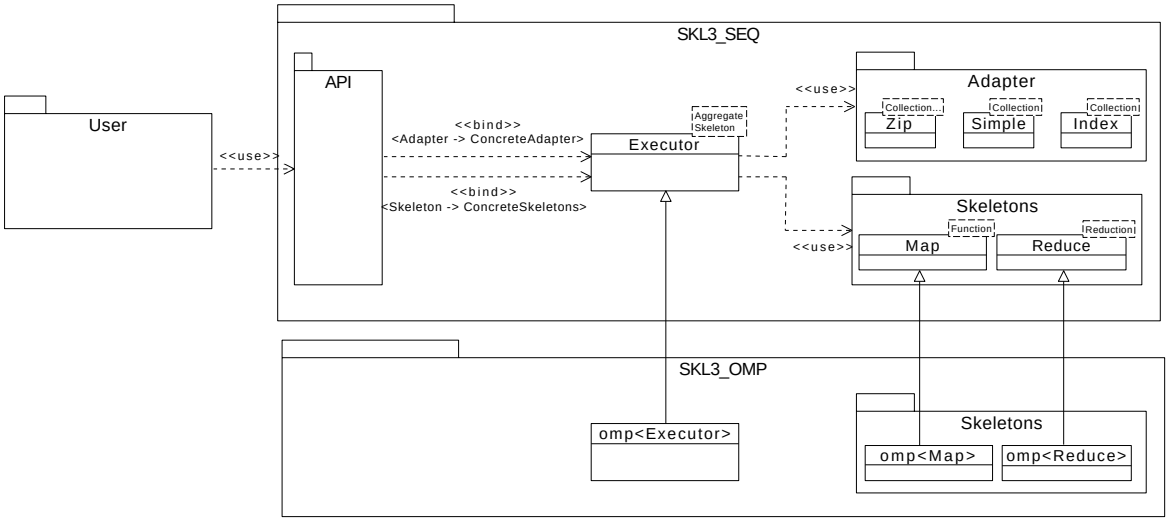


Figura 10: Arquitectura da versão 3 da biblioteca.

A solução para estes problemas acabou por implicar pequenas diferenças arquitecturais assim como na API de utilização dos esqueletos. A principal diferença, como se observa na Figura 10, relativamente às versões anteriores, é a introdução da classe genérica Executor.Esta classe , uma classe faz uma composição estática de um adaptador com um esqueleto, ambos seleccionados pelo utilizador. De notar que este esqueleto pode ser a fusão de um número arbitrário de esqueletos.

A camada **Adapter** substitui a anteriormente denominada **Data**. Nesta versão, as classes desta camada passaram a ter um comportamento mais coeso de adaptação explicita do esqueleto. Por exemplo, o esqueleto map_i deixou de existir para passar a ser disponibilizado um adaptador dos esqueletos, representado pela classe Index da camada de adaptação. Ou seja, o comportamento de interacção com índices foi desacoplado do esqueleto map_i para ser disponibilizado como um *pluggable adapter*, que pode ser conectado a qualquer um dos esqueletos. Em geral o utilizador pode facultar qualquer um dos esqueletos com um adaptador específico.

Relativamente à fusão dos esqueletos, o que se fez foi desconstruir os esqueletos num conjunto de comportamentos/fases. Na Figura 10, as classes que representam os esqueletos são *mixins* compostos por métodos que correspondem a cada uma dessas fases do esqueleto. A invocação de métodos de um esqueleto invoca os métodos dos outros esqueletos de forma

recursiva, uma vez que estes são *mixins*. Os *mixins* são construções estáticas que permitem ao compilador fazer a invocação em linha dos métodos, e consequentemente dos functores.

O Executor delega estaticamente os comportamentos relativos à iteração ao adaptador, por exemplo, a incrementação do iterador; e os relativos às fases do esqueleto ao esqueleto seleccionado pelo utilizador. A refinação do esqueleto para suportar de paralelização é também ela um *mixin*, pelo que o *Executor* delega , por exemplo, no caso da paralelização com recurso a memória partilhada, a fase de redução das variáveis locais de cada *thread* a esse *mixin*, classe omp na Figura 10.

O Executor delega estaticamente os comportamentos relativos à iteração e às fases dos esqueletos, respectivamente, ao adaptador e ao esqueleto seleccionados pelo utilizador. Por exemplo, a incrementação do iterador é delegada para o adaptador e aplicação do functor para o esqueleto. Para além destes comportamentos o Executor delega também as fases da paralelização. Neste caso, a delegação é feita automaticamente porque estes refinamentos são *mixins* sobre os esqueletos, ou seja, as fases são invocadas através do mesmo processo descrito na fusão dos esqueletos. Por exemplo, no caso da paralelização com recurso a memória partilhada, o Executor delega a fase de redução das variáveis locais de cada *thread* à especialização da classe omp que refina o esqueleto.

De notar que o uso do adaptador traz vantagens tanto para o desenvolvedor da biblioteca como para o utilizador. Para o primeiro reduz a replicação de código, ou seja, para implementar novos adaptadores é apenas necessário implementar as diferenças em relação aos adaptadores já existentes, uma vez que a classe Executor contém o código comum entre os esqueletos e as suas variantes. Para o utilizador, é vantajoso porque este pode implementar os seus próprios adaptadores, isto é as suas próprias variantes, sem o custo de ter de implementar um esqueleto de raiz e sem ter de lidar com a programação paralela. O adaptador é assim uma via para se poder adaptar os esqueletos, sem a necessidade de se alterar directamente/internamente a biblioteca.

A Figura 10 apresenta várias similaridades com a figura apresentada das *mixin layers*. Apesar de a arquitectura ser semelhante estas divergem no facto de no artigo original as camadas serem específicas para uma aplicação em concreto. Neste caso o utilizador faz uma parametrização das componentes da camada, por exemplo a utilização de um esqueleto específico requer a parametrização de um adaptador, ou seja a camada só é composta por um dos **Adapters** da figura ou o **Zip** ou o Simple ou Index, o mesmo acontece com os Skeletons. Note-se que apesar de poderem ser usados mais do que um esqueleto estes são fundidos numa só classe e assim tratados com se apenas de um se tratasse.

Apesar desta diferença aquando da parametrização da camada o diagrama de classes originado é semelhante ao das versões anteriores, com a mesma arquitectura por camadas. De salientar que o refinamento da camada é feito automaticamente sobre as componente seleccionadas pelo utilizador, isto é, se for utilizado o *map*, é usado o refinamento do *map*,

no casos de ser seleccionado o *reduce* é utilizado o refinamento do *reduce*. Isto é feito com recurso à mesma técnica apresentada anteriormente de especialização de classes genéricas.

3.2 UTILIZAÇÃO DA BIBLIOTECA DE ESQUELETOS

Nesta secção são apresentados sobre forma de tabelas os construtores desenvolvidos, as suas especificações e a sua assinatura. Depois de apresentados os construtores seguem-se alguns exemplos de utilização da biblioteca.

No final da secção é abordada a compilação/linkagem da biblioteca, com foco na activação das camadas paralelas. Note-se que a activação das camadas de paralelização é feita exclusivamente em tempo de compilação, isto é, sem a necessidade de alterar código ou o uso de esqueletos com uma sintaxe diferente.

3.2.1 *Sintaxe e descrição do esqueletos*

Nome	*Descrição*	*Sintaxe*
map	Aplica o functor f a um conjunto valores apontados por um tuplo de iteradores das respectivas colecções c.	void map(Fn&& f, Collection&... c)
		void map<Fn>(Collection&... c)
reduce	Aplica o functor com estado sf a um conjunto de elementos das colecções c. O retorno é o functor sf que contém como estado a variável de acumulação.	StFn reduce(StFn&& sf, Collection&... c)
		StFn reduce<StFn>(Collection&... c)
reduce2	*BiFn* é um functor cujos argumentos são a variável redução e um conjunto de elementos das colecções *c*. O valor de retorno é a variável de redução, que tem o mesmo tipo que o valor de retorno do functor.	
		auto reduce2<BiFn>(Collection&... c)
map_reduce	Aplica consecutivamente os functores f e sf, a um conjunto de elementos das colecções *c*.	StFn map_reduce(Fn&& f, StFn&& sf, Collection&... c)
		StFn map_reduce<Fn,StFn>(Collection&... c)
map_reduce2	Aplica consecutivamente os functores *Fn* e *BiFn*, a um conjunto de elementos das colecções c. O Functor BiFn é o mesmo do explicado para o reduce2.	
		auto map_reduce2<Fn, BiFn>(Collection&... c)
map_i	Aplica o functor f a todos os índices respectivamente de elementos da colecção c. Os argumentos do functor são o índice da iteração, a colecção c e os argumento args.	void map_i(Fn&& f, Collection& c, Args&... args)
		void map_i<Fn>(Collection& c, Args&... args)
compose	Faz a fusão dos esqueletos skeletons, aplicando consecutivamente os functores de cada um dos esqueletos a um elemento criado pelo adapter, para todos os elementos.	auto compose(Adapter&& a, Skeletons&&... skeletons)

Tabela 1: Descrição e sintaxe dos esqueletos/funcionalidades disponibilizados.

A Tabela 1 descreve os construtores das três versões da biblioteca desenvolvida, e a sua sintaxe. Como pode ser observado para cada esqueleto são apresentadas duas sintaxes semelhantes, apenas com a diferença de na primeira o functor ser passado como argumento da rotina e na segunda o functor ser passado exclusivamente como parâmetro. Os construtores reduce2 e map_reduce2 só foram implementados para a segunda versão, porque as versões por passagem por parâmetro das versões reduce e map_reduce apresentam um desempenho fraco relativamente à passagem por argumentos, tópico analisado no caso de estudo do *kmeans* (4.1.2)

No caso do compose este foi apenas implementado com a passagem do functor por argumento.

De seguida apresentam-se duas tabelas que especificam os construtores que podem ser passados nos argumentos Adapter e Skeletons do construtor compose.

Esqueleto	*Descrição*	*Sintaxe*
map	Aplica o functor f a todos os elementos de uma dada colecção.	auto map(Fn&& f, MiniAdapter&& a=identity)
reduce	Aplica o functor binário f , para todos os elementos de colecção. Os argumentos do functor são a variável de redução e um elemento de uma colecção.	auto reduce(BiFn&& f, MiniAdapter&& a=identity)

Tabela 2: Descrição e sintaxe dos esqueletos da versão SKL3.

De notar que os esqueletos apresentados na Tabela 2, têm de ser aplicados como argumentos do construtor compose. O MiniAdapter apenas tem relevância quando utilizado o adaptador zip como argumento do compose, nesse caso o elemento iterativo tem vários iteradores e alguns dos functores podem não necessitar de todos os iteradores, o MiniAdapter serve como filtro de iteradores desnecessários de alguns esqueletos. No caso do MiniAdapeter não ser especificado/necessário este é escolhido por defeito como sendo a identidade, ou seja, não filtra nenhum dos iteradores.

Adaptador	*Descrição*	*Sintaxe*	*Argumentos do functor*
simple	Adapta o esqueleto de modo a que este possa iterar qualquer objecto que implemente a interface de uma *Collection*, isto é, a classe do objecto implemente os métodos *begin()* e *end()* a retornarem um iterador.	simple(Collection&& c)	Iterator&& ite
zip	O zip recebe um conjunto arbitrário de colecções e faz com que o esqueleto seja capaz de iterá-las simultaneamente. As colecções necessitam de implementar a interface *Collection*.	zip(Collection&&... cs)	Iterator&&... ite
index	Recebe como argumento uma colecção, tal que seja possível obter o tamanho desta através do método *size()*. Adapta o esqueleto de modo que este faça a iteração da colecção através de índices ao invés de apontadores.	index(Collection&& c)	size_t i, Collection& c
dummy	Transforma um objecto de um tipo T não iterável numa iteração infinita desse mesmo valor. Não sendo esta variável uma colecção pede ser utilizada para comunicar dados entre functores dos diversos esqueletos.	dummy<T>(T&& value)	T&& value

Tabela 3: Descrição e sintaxe dos adaptadores da versão SKL3.

A Tabela 3 apresenta os adaptadores implementados, mas note-se que o utilizador pode criar os seu próprios adaptadores.

De salientar que os adaptadores podem ser aninhados, o que é útil, como se pode observar no caso de estudo *kmeans*(4.1.2), com o aninhamento dos adaptadores *zip* e *dummy*.

3.2.2 *Exemplos do uso de esqueletos esqueletos*

De seguida são apresentados alguns exemplos de utilização da biblioteca. Exemplos que, na sua generalidade, são simples e carecem de utilidade, mas que ilustram bem as funcionalidades essências da biblioteca.

```
1 vector<int> vx;
2 struct inc { void operator()(int& x){ x++; } }
3 skl::map<inc>(vx);
4 skl::map(inc, vx);
```

Listagem 3.1: Exemplo de dois *maps*, passagem por parâmetros e por argumento

Na Listagem 3.1 pode observar-se a incrementação de todos os elementos de um vector através do uso do esqueleto *map*, na linha 3 pode ver-se o uso da versão em que o functor é passado como parâmetro. A linha seguinte obtém exactamente o mesmo resultado mas utilizando a versão do esqueleto *map* em que o functor é passado por argumento da função.

```
1   const size_t size = 100;
2   std::vector<int> x(size, 1);
3   std::vector<int> y(size, 2);
4   auto axpy = [](const size_t i, const std::vector<int>& x, std::vector<int>& y) { y[i
        ] += 2 * x[i]; };
5   skl::map_i(axpy, x, y);
```

Listagem 3.2: Exemplo do *map_i*

Na Listagem 3.2 pode observar-se a operação *axpy* entre dois vectores em que o vector y é modificado para conter o resultado da computação entre vectores $y + ax$. Neste exemplo, ao contrario do anterior, o functor utilizado é um uma função lambda, que é apenas açúcar sintático para declarar, implementar e instanciar um functor. O esqueleto utilizado é o *mapi_i*, que é similar ao *map* mas neste é possível saber qual o índice de iteração. Este esqueleto tem a particularidade de poder receber um número arbitrário de argumentos. Estes são passados ao functor como argumentos a cada iteração, desta forma, no exemplo, o esqueleto é invocado com dois argumentos, sendo estes duas referências para os vectores x e y, que são também passadas ao functor ao invés de apenas o elemento, neste caso o elemento a iterar é acessível utilizando o primeiro argumento do functor ou seja o índice da iteração.

```
1  struct sum
2  {
3    int res;
4    sum() : res(0) {}
5    void operator()(const int& x) { res += x; }
6    void operator()(const sum other) { res += other.res; }
7  };
8  std::vector<int> x(size, value);
9  sum s = skl::reduce(sum(), x);
10 int r = s.res;
11 int res = skl::reduce2<std::plus<int>>(x);
```

Listagem 3.3: Exemplo das diferenças do *reduce* e *reduce2*

Na Listagem 3.3 podem ver-se as versões alternativas do esqueleto *reduce*, nas linhas 9 e 10 pode ver-se a utilização da versão que utiliza o estado do functor para armazenar o resultado final. Isto é na declaração do functor é declarada a variável res, que será utilizada pela operação do functor para armazenar a redução (linha 5). Este esqueleto retorna o próprio functor, assim sendo é da responsabilidade do utilizador ir buscar o valor ou valores da redução (linha 10).

Por fim na linha 11 é apresentada a outra versão do *reduce*, o *reduce2*. Esta versão recebe um functor binário, cujos argumentos são a variável de redução e o elemento da colecção, este calcula uma operação, neste caso a soma de inteiros por via do functor std::plus<int> disponibilizado pela biblioteca padrão do C++, e o valor de retorno desta operação é escrito na variável de redução. Neste esqueleto a variável de redução é inicializada com o valor do primeiro elemento da colecção. Observe-se que neste caso, ao contrario do anterior, o retorno do esqueleto é do tipo *int*, uma vez que o esqueleto tem o mesmo retorno que o functor, isto é, é retornada a variável de redução directamente.

```
1  struct column_mean
2  {
3    int x_acc;
4    int y_acc;
5    int m_acc;
6    int count;
7    column_mean() : x_acc(0), y_acc(0), m_acc(0), count(0) {}
8    void operator()(int x, int y int m) {
9      x_acc += x;
10     y_acc += y;
11     m_acc += m;
12     ++count;
13   }
14   void operator()(const column_mean& other) {
15     x_acc += other.x_acc;
16     y_acc += other.y_acc;
17     m_acc += other.m_acc;
18     count += othe.count;
19   }
20   auto get_result(){ return std::make_tuple(xm/count, ym/count, mm/count); }
21 };
22 auto line_mean =  [](int x, int y, int& m){ m = (x + y)/2; } }
23 std::vector<int> vx(size, 1);
24 std::vector<int> vy(size, 2);
25 std::vector<int> vm(size, 0);
26 auto[xm, ym, mm] = skl::map_reduce(line_mean, column_mean(), vx, vy, vm).get_result();
```

Listagem 3.4: Exemplo *map_reduce*

Na Listagem 3.4 é exemplificado o uso do *map_reduce*. Para o *map* é utilizado uma função lambda line_mean (linha 17). Este functor recebe três argumentos, dois de leitura e o ultimo de escrita. O functor calcula a media dos primeiro dois argumentos e guarda no último. Este functor ao ser utilizado pelo *map* transforma o vector vm no resultado da media dos vectores vx e vy ponto a ponto.

O functor do *reduce* é mais complexo, ao invés de calcular a media de uma linha, computa a media das coluna, ou seja calcula a media de cada um dos vectores. Como se pode observar nas linhas 3 a 5 o functor é composto por três variáveis de redução (x_acc, y_acc, m_acc), uma para cada vector. Estas variáveis servem para acumular os valores de cada iteração, como se pode ver na linha 8 a variável x_acc a acumular x e analogamente para as outras variáveis. Neste caso o utilizador não necessita de ir buscar directamente as variáveis ao functor, uma vez que este tem implementado o método get_result. Este método computa a média dos vectores, dividindo cada uma das variáveis de acumulação pela variável count, que durante a iteração dos vectores contabiliza o número de elementos do vector, por fim o método retorna o calculo das média num tuplo.

De notar que este functor necessita de fazer a sobrecarga do operator() recebendo como argumento o próprio tipo, para que o esqueleto seja capaz de fazer a redução dos functores locais a cada *thread*.

```
1 struct min {
2   void operator()(const int& x, const int& y) { return (x>y)? y : x}
3 };
4 struct max {
5   void operator()(const int& x, const int& y) { return (x>y)? x : y}
6 };
7 std::vector<int> x(size, value);
8 auto [v_min, v_max] = skl::compose(x, reduce(min()), reduce(max()));
```

Listagem 3.5: Exemplo da fusão de dois *reduces*

Na Listagem 3.5 ilustra a utilização do construtor compose através da fusão de dois *reduces*. O primeiro reduz a colecção no seu valor mínimo e o segundo reduz a colecção no valor máximo. A fusão é feita de modo que o calculo destes dois valores seja feito em simultâneo, isto é com único ciclo de iteração da colecção.

Observe-se que a utilização do esqueleto *reduce* gera um variável de saída/escrita. Neste caso são utilizados dois *reduces* pelo que o *compose* é capaz de unir as duas variáveis de saída, automaticamente, e retorná-las num tuplo, pelo contrario ao apresentado na Listagem 3.4, em o utilizador necessita programar explicitamente este comportamento.

```
1 struct initialize_index {
2   void operator()(int i, std::vector<int>& c){ c[i] = i; }
3 };
4 struct mini_adapter {
5   auto operator()(const auto& ite)  {
6     auto [i, c] = ite;
7     return make_tuple(c[i]);
8  }
9 };
10 std::vector<int> x(size, value);
11 auto [sum] = skl::compose(index(x)
12               , map(initialize_index())
13               , reduce(std::plus<int>(), mini_adapter()));
```

Listagem 3.6: Exemplo da fusão do *map* e do *reduce*, utilizando o adaptador *index*

O exemplo da Listagem 3.6 é um pouco complexo e de pouca utilidade, ainda assim é apresentado para demonstrar algumas das funcionalidades da biblioteca, como a adaptação dos esqueletos aos elemento iterados e a utilização de um adaptador index. Como se pode observar, é uma fusão de um *map* com um *reduce*, utilizando o adaptador *index*. A utilização deste adaptador, faz com que seja passado functor deste (initialize_index) o índice da iteração e a colecção, o que permite inicializar os elementos do *vector* com o índice da sua posição, o que não seria possível utilizando o *map* , sem adaptador. Como se pode observar, neste caso, o functor initialize_index, acede a um elemento da colecção fazendo a habitual aritmética de apontadores *c[i]* (linha 2). Note-se que os argumentos passados ao *reduce* são os mesmos que ao *map*, o que seria impossível de compilar uma vez que o functor do *reduce* (*std::plus<int>*)

recebe dois argumentos, a variável de redução e um elemento da colecção. Ou seja para se poder compilar é necessário adaptar os argumentos que são passados ao functor do *reduce*, funcionalidade exercida pelo *mini_adapter*, isto é, transforma os dois argumentos índice e colecção num só, o elemento a reduzir.

3.2.3 *Interfaces*

1. **Collection**
   a) Iterator begin()
   b) Iterator end()
2. **Adapter**
   a) Iterator begin()
   b) Iterator end()
3. **Iterator**
   a) iterator& operator++();
   b) reference operator*() const;
   c) friend bool operator!=(const iterator&, const iterator&);
4. **Skeleton**
   a) void pre_for()
   b) void inside_for(Iterator& i)
   c) void post_for()
5. **Functor**
   a) auto operator()(Args... a)
6. **MiniAdapter**
   a) std::tuple<T...> operator()(std::tuple<T...> ite)

3.2.4 *Compilação e execução*

A utilização da biblioteca pode ser feita de dois modos, o sequencial e o paralelo, em ambos os casos o código fonte da aplicação é o mesmo, mudando apenas a implementação interna da biblioteca. A activação da paralelização é feita em tempo de compilação, para isso é acrescentada a opção "-DSKL_SM"no processo de compilação, que irá estender os esqueletos, refinando-os assim com as funcionalidades paralelas.

A quantidade de threads utilizadas na paralelização com recurso a memória partilhada pode ser feita inicializando a variável de ambiente "SKL_NUM_THREADS", antes da execução, com o numero de *therads* desejado.

3.3 IMPLEMENTAÇÃO

3.3.1 *SKL1*

Os esqueletos foram implementados sequencialmente através de classes e o tipo de paralelização é implementado nas suas sucessivas extensões. Estas extensões implementam apenas as modificações necessárias para a paralelização do esqueleto base (sequencial).

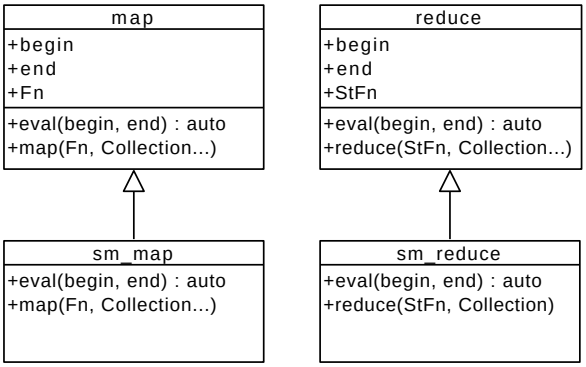


Figura 11: Diagrama de classes da SKL1 versão argumentos.

Como se pode ver na Figura 11 as classes que representam os esqueletos são compostas por três variáveis:

begin é um iterador, que aponta para o primeiro elemento da colecção

end é também um iterador, mas neste caso aponta para o último[1] elemento da colecção

Fn é um functor. É utilizado pelo esqueleto invocando-o para todos os elementos da colecção, com os mesmos como argumentos. Os functores do *map* e do *reduce* têm comportamentos diferentes, por exemplo, o functor StFn do *reduce* tem estado, sendo esse estado constituído por uma ou mais variáveis de redução/acumulação, ou seja, de cada vez que este functor é invocado esta variáveis sofrem alterações.

Para além destes variáveis cada esqueleto é também composto por por dois métodos:

1 Para se ser mais correto aponta para o elemento imediatamente a seguir ao último, para que o caso de paragem da iteração ser o iterador ter o mesmo valor que o *end*, ou seja, apontar exactamente para o mesmo elemento que este iterador *end*

Construtor utilizado para receber os argumentos do esqueleto, isto é, o functor e a colecção que se pretende iterar, as variáveis begin e end são respectivamente inicializadas com os valores de retorno da invocação dos métodos begin() e end() sobre essa colecção.

eval é o método onde se encontra implementada a lógica do esqueleto, este método recebe dois argumentos, dois iteradores que representam um intervalo de uma colecção a que se pretende iterar o esqueleto.

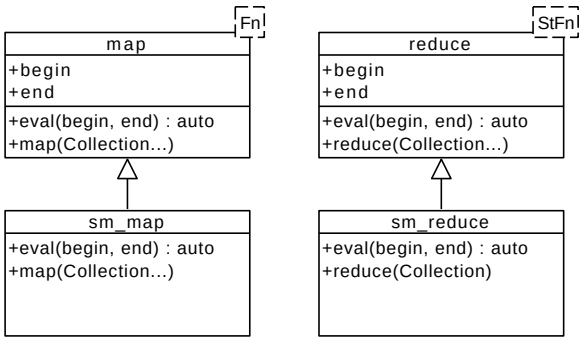


Figura 12: Diagrama de classes da SKL1 versão parâmetros.

A Figura 12 a presenta o diagrama de classes da versão parâmetros. Em comparação com a figura anterior pode observar-se que neste caso o functor deixa de ser uma variável de instância para ser apenas um parâmetro da classe genérica. Neste caso não há necessidade de se inicializar essa variável por isso foi possível deixar cair o functor como argumento do construtor. Isto significa o esqueleto e o functor são acoplados em tempo de compilação, por sua vez, isto significa que não existe qualquer passagem de dados relativos ao functor para o esqueleto em tempo de execução. Por este motivo era esperado que esta versão apresentasse uma melhor eficiência/performance, o que acabou por não se verificar, como é analisado no capitulo 4.

Observe-se ainda que os refinamentos sm_map e sm_reduce têm exactamente os mesmos métodos. Até as suas implementações são semelhantes, apenas pequenas diferenças. O que significa que esta não é a melhor forma de estes serem implementados.

Paralelização

Nesta versão da biblioteca a camada paralelização implementada foi com recurso a memória partilhada. Mais especificamente foi implementada uma *ParallelThreadPool* que tem um comportamento semelhante a uma *thread pool* normal mas neste caso a tarefa é executadas por todas as threads em paralelo (sendo que a execução da tarefa difere de thread para thread devido a variáveis internas da thread como por exemplo o identificador de thread) outra das diferenças é a thread principal (que faz a submissão da tarefa) ficar

bloqueada até que todas as threads terminem. Esta *ParallelThreadPool* foi desenvolvida utilizando o suporte a *threads* disponibilizado pela biblioteca padrão do c++.

As versões dos esqueletos paralelizados são extensões destas classes representativas dos diversos esqueletos. Estas classes não contêm qualquer variável adicional, apenas modificando os métodos da classe base de modo a acrescentarem a paralelização. Assim os métodos apresentados anteriormente passam nesta camada a ter as seguintes funções:

Construtor redirecciona os argumentos para a classe base, ou seja, invoca o construtor da classe estendida com os argumentos recebidos;

eval passa a ser responsável pela introdução do paralelismo assim como da reutilização do código das camadas inferiores. A paralelização consiste na criação e submissão de uma tarefa para uma ParallelThreadPool. Esta tarefa consiste no escalonamento de intervalos das colecções pelas threads. As threads tem assim apenas de aplicar o *eval* da camada inferior com os iteradores de inicio e fim do intervalo atribuído.

O facto de o construtor apenas servir para redireccionar os argumentos para a classe base deve-se ao facto de esta camada não ter variáveis de instância. Por exemplo, o construtor desta camada poderia servir para implementar o padrão estratégia em relação escalonador das *threads*, isto é, o construtor poderia receber um escalonador concreto que seria atribuído a uma variável de instância, assim o comportamento de escalonamento das *threads* seria delegado para essa variável. Nesse caso o construtor servira para filtrar os argumentos destinados a essa camada.

Note-se que apesar de os esqueletos *map* e *reduce* (com a utilização de um functor com estado) na sua implementação sequencial serem idênticos, necessitam de ser implementados como classes diferentes porque os seus refinamentos são implementados de forma diferente, por exemplo, a utilização do esqueleto *map* com um functor do tipo do *reduce* origina um *data race* na variável de redução.

Note-se ainda que as classes *sm_map* e *sm_reduce* são *mixins* o que significa que não estendem objectivamente a implementação sequencial dos esqueleto mas sim uma classe genérica. Isto permite que este refinamento seja aplicado não só à implementação sequencial mas também a outros refinamentos, ou seja, permite utilizar diferentes tipos de combinações de refinamentos.

```
1  vector<int> vx;
2  struct inc { void operator()(int& x){ x++; } }
3
4  map_impl<inc> m(vx);
5  m.eval(m.begin, m.end);
6
7  sm_map<map_impl<inc>> m(vx);
8  m.eval(m.begin, m.end);
9
10 //dm_map<map_impl<inc>> m(vx);
11 //dm_map<sm_map<map_impl<inc>>> m(vx);
```

Listagem 3.7: Exemplo da refinação da implementação do *map_impl* por diversos tipos de paralelização.

Na Listagem 3.7 pode observar-se a utilização directa destas classes que representam os esqueletos e os seus refinamentos, este é um exemplo da versão parâmetros. Na linha 4 é instanciado objecto m que representa a aplicação do esqueleto *map* com functor inc à colecção vx. Na linha seguinte é invocado, a esse objecto, o método eval com os iteradores da colecção, só aqui com a evocação deste método se dá verdadeiramente a execução do *map*.

Nas linhas 7 e 8 é programado exactamente o mesmo processo, mas desta vez o objecto m é dotado do refinamento que torna a sua execução paralela, esta extensão é obtida passando como parâmetro do *mixin* sm_map o tipo do objecto do exemplo anterior.

Nas linhas em comentário 10 e 11 pode ver-se e exemplo de como poderia ser feito o refinamento de uma paralelização com recurso a memória distribuída ou híbrida do mesmo esqueleto apresentado anteriormente. Estas linha aparecem em comentário porque isto não é possível, uma vez que a camada de memória distribuída não está implementada.

```
1  namespace skl {
2
3  template<typename Fn, typename Collection>
4  void map(Collection& c) {
5  using map_impl =
6  #ifdef SKL_SM
7    skeleton::sm_map<
8  #endif
9    skeleton::map
10 #ifdef SKL_SM
11   >
12 #endif
13 ;
14   map_impl<Fn, Collection> m(c);
15   return m.eval(m.begin, m.end);
16 }
17
18 } // namespace skl
```

Listagem 3.8: Exemplo da implemntação rotina *map* disponibilizada pela API da biblioteca.

A Listagem 3.8 é o exemplo da implementação da rotina disponibilizada pela API da biblioteca. Nas linhas 14 e 15 pode observar-se a utilização do esqueleto, que é, exactamente o processo apresentado na listagem anterior.

Apesar disso, os refinamentos são feitos de forma ligeiramente diferente. Neste caso é utilizada a directiva *"#ifdef"* para se declara o tipo do *map* como sendo simplesmente a classe sequencial, ou utilizando refinamentos sobre esta por via de um identificador definido em tempo de compilação, ou seja, os refinamentos utilizado são seleccionados sem a necessidade de alterar o código fonte. Isto significa que o esqueleto é refinado em tempo de compilação.

O uso destas funções, ao invés de directamente as classes que representam os esqueletos, permite omitir do utilizador este processo repetitivo de declaração, extensão, instanciação e execução dos esqueletos, agilizando uso destes.

3.3.2 *SKL2*

Recapitulando, nesta versão foram adicionadas as seguintes funcionalidades: novo esqueleto, o *map_reduce*; a iteração de múltiplas colecções em simultaneamente e a iteração através de índices, que originou também um novo esqueleto, o *map_i*.

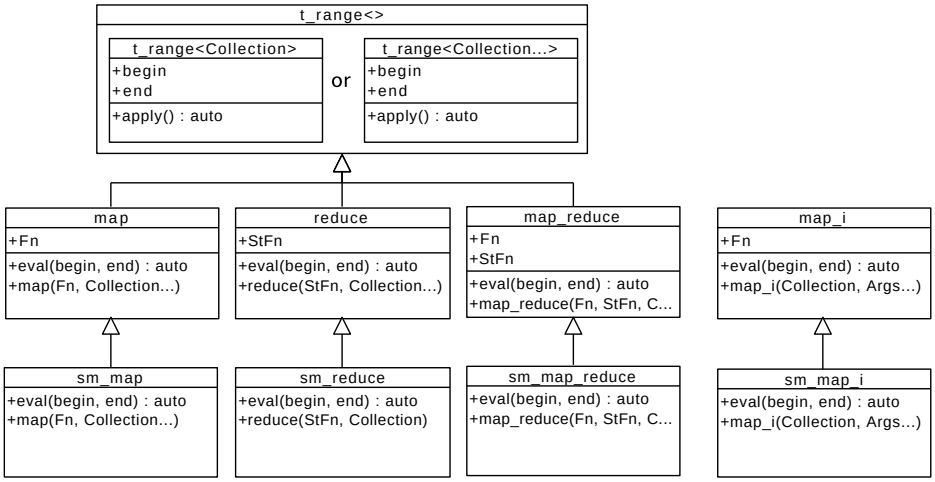


Figura 13: Diagrama de classes da SKL2, versão argumentos.

A versão anterior apresenta uma falha arquitectural. Todos os esqueletos são compostos da mesma forma, isto é, por dois iteradores. Isto é um problema, desde logo porque isto requer uma replicação do código por todos os esqueletos. Mas também porque, uma vez que os comportamentos de, por exemplo, selecção, incrementação dos iteradores e condição de paragem estão contidos no esqueleto. Assim a alteração destes comportamentos requer uma modificação ou a criação de novos esqueletos, ao invés de substituir esse componente por um com os comportamentos desejados.

A arquitectura proposta nesta versão separa este comportamento do esqueleto, dando origem à classe *t_range*, que é estendida por todos os esqueletos, evitando a replicação de código. Como se pode ver na Figura 13 esta classe tem duas especializações, uma em que o construtor recebe apenas uma colecção e a outra especialização recebe múltiplas colecções, de forma que o compilador possa inferir automaticamente através dos argumentos do constructor qual a classe que deve ser estendida pelo esqueleto. Em ambos os caso os iteradores passam a compor o esqueleto, mas de uma classe para a outra os iteradores mudam o seu tipo. De notar que variáveis e métodos de cada uma são as mesmas, simulando uma espécie de interface comum, que é utilizada pelos esqueletos que estendem esta classes.

```
1  template<typename... Collections>
2  struct t_range
3  {
4    using iterator = zip_iterator<std::tuple<typename Collections::iterator...>>;
5
6    iterator begin_;
7    iterator end_;
8
9    explicit t_range(Collections&... collections)
10     : begin_( std::make_tuple(begin(collections)...) )
11     , end_( std::make_tuple(end(collections)...) )
12   {}
13
14   template<typename Fn, typename Args>
15   static auto apply(Fn&& f, Args&& args)
16   {
17     return std::apply(f, args);
18   }
19 }
```

Listagem 3.9: Implementação da especialização classe *t_range*

Na Listagem 3.9 pode ver-se a implementação da especialização classe *t_range*. Esta é a especialização que permite iterar múltiplas colecções. Na linha 4 é definido o tipo dos iteradores, como sendo um tuplo dos tipos dos iteradores, envolvido pelo *mixin* zip_iterator. Este *mixin* dota um qualquer tuplo com a mesma interface de iteração dos iteradores que o constituem. Esta classe é uma concretização do padrão compósito para a interface de iteradores, ou seja, a classe invoca os métodos, a si invocados, a cada um dos iteradores que compõem o tuplo.

Na linha 9 pode ver-se que o construtor desta classe recebe múltiplas colecções é através destes argumentos que o compilador decide seleccionar esta especialização.

Nas linhas 10 e 11 são inicializados os iteradores. O *zip_iterator* é inicializado com um tuplo de iteradores. Para isso, é utilizada uma técnica de meta-programação para desempacotar as colecções passadas como argumento e é lhes aplicada uma função, neste caso os métodos *begin* e *end*, respectivamente para obter os iteradores iniciais e finais de cada uma das colecções. Estes iteradores são por fim transformados num tuplo para serem passados ao construtor do *zip_iterator*

Neste caso, o método apply (linha 15) simplesmente invoca a rotina std::apply que recebe como argumentos um functor e um tuplo. Este tuplo de elementos é desempacotado, ou seja, cada elemento é passado ao functor como um argumento diferente. No caso da especialização para iterar uma só colecção não é necessário fazer o desempacotamento uma vez que os iteradores não são um tuplo, assim sendo é invocado simplesmente a rotina *std::invoke*. Já no caso da *map_i* esta função invoca o método *std::invoke*, mas desta vez com um functor, um índice, uma colecção e um número arbitrário de argumentos.

```
1  template<typename CollectionTuple>
2  struct zip_iterator
3  {
4    CollectionTuple tuple_;
5
6    zip_iterator(CollectionTuple tuple)
7      : tuple_(tuple)
8    {}
9
10   template<typename CollectionTupleOther>
11   bool tuple_any_equal(CollectionTupleOther& other) const
12   {
13     auto tuple_of_pairs = util::tuple::zip(tuple_, other.tuple_);
14     return std::apply(
15         [&](auto&... x) { return (... || (std::get<0>(x) == std::get<1>(x))); }
16         , tt);
17   }
18
19   auto operator*()
20   {
21     auto dereference_iterator = [](auto&& iterator) { return std::ref(*iterator); };
22     return util::tuple::make_for_each(dereference_iterator, tuple_);
23   }
24
25   void operator++()
26   {
27     auto increment_iterator = [](auto&& iterator) { ++iterator; };
28     util::tuple::for_each(increment_iterator, tuple_);
29   }
30
31   template<typename CollectionTupleOther>
32   bool operator!=(CollectionTupleOther& other) const
33   {
34     auto tuple_of_pairs = util::tuple::zip(tuple_, other.tuple_);
35     return std::apply(
36         [&](auto&... x)  { return (... && (std::get<0>(x) != std::get<1>(x))); }
37         , tuple_of_pairs);
38   }
39 };
```

Listagem 3.10: Implementação da classe *zip_iterator*

A Listagem 3.10 ilustra a implementação do *zip_iterator*. Observe-se que esta é uma classe genérica, em que o seu parâmetro é um tuplo o que significa que esta classe pode envolver um tuplo com quaisquer tipos (linhas 1 e 4).

Nas linhas 19, 25 e 31 são declarados e implementados os métodos principais de um iterador. Como referido anteriormente cada um desses métodos invoca os respectivos métodos a cada um dos elementos do tuplo. Por exemplo, na implementação do método operator++(), é instanciada uma função lambada que incrementa um qualquer tipo de iterador. De seguida, com recurso à rotina auxiliar util::tuple::for_each, é aplicada esta lambda a todos os elementos do tuplo. De notar que tudo isto é feito através de técnicas de

meta-programação, ou seja, esta aplicação de um functor a todos os elementos de um tuplo não é feita em tempo de execução mas sim gerado código em tempo de compilação para fazer essa aplicação.

```
1  template<typename Fn, typename... Collection>
2  struct map_impl : t_range<Collection...>
3  {
4    using range = t_range<Collection...>;
5    using iterator = typename range::iterator;
6
7    Fn&& mapf;
8
9    map_impl(Fn&& f, Collection&... collections)
10     : range(collections...)
11     , mapf(std::forward<Fn>(f))
12   {}
13
14   void eval(iterator begin, iterator end)
15   {
16     for (iterator& ite = begin; begin != end; ++ite)
17     {
18       range::apply(mapf, *ite);
19     }
20   }
21 }
```

Listagem 3.11: Implementação sequencial *map* passando o morfismo como argumento

A Listagem 3.11 apresenta a implementação do *map* em SKL2, como se pode ver, as colecções são passadas na linha 10 à classe base *range* que é apenas um nome adicional para a classe estendida t_range<Collection...>, através deste argumento é inferida a especialização correcta do *range* e portanto o esqueleto faz a extensão da especialização inferida (linha 2).

Observe-se que na linha 18 a forma como o functor é aplicado ao iterador é delegada estaticamente ao *range*. Esta delegação é estática uma vez que o *range* é estendido pelo esqueleto, ainda assim esta extensão não é de uma classe em concreto, podendo esta mudar através da especialização. O facto de se tratar de uma delegação estática significa que o compilador pode fazer a invocação desse método em linha.

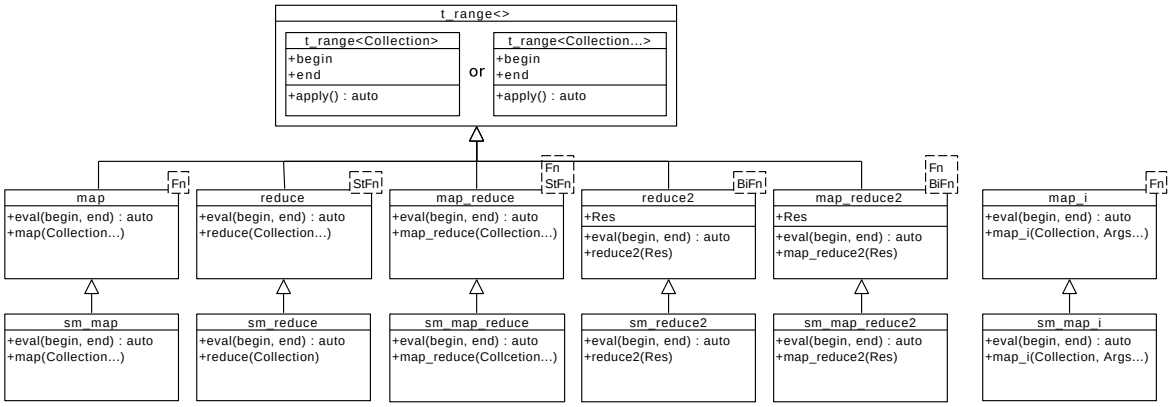


Figura 14: Diagrama de classes da SKL2, versão parâmetros.

A Figura 14 apresenta algumas diferenças em relação à versão anterior. Nesta versão foram implementadas duas versões de esqueletos já existentes, o *reduce2* e *map_reduce2*. Isto foi necessário porque os esqueletos *map* e *map_reduce* base apresentaram um mau desempenho em algumas situações quer sequencialmente quer paralelamente. Este problema é analisado em detalhe no caso de estudo *kmeans* 4.1.2, mas note-se que o problema consistiu no uso de functores com estado e a passagem destes como parâmetros. Para resolver o problema os functores de redução dos novos esqueletos deixaram de ser functores que contêm estado para functores binários que recebem como argumentos a variável de redução e um elemento da colecção. Ou seja, foi separado os dados do functor, o que permite que a variável de redução seja guardada como uma variável de instância do esqueleto. Na figura esta variável corresponde à variável Res dos esqueletos *reduce2* e *map_reduce2*.

3.3.3 *SKL3*

Como pode ser visto na secção Visão Geral, uma das diferenças desta versão foi a implementação de uma nova classe Executor, que é a peça central de todos os esqueletos.

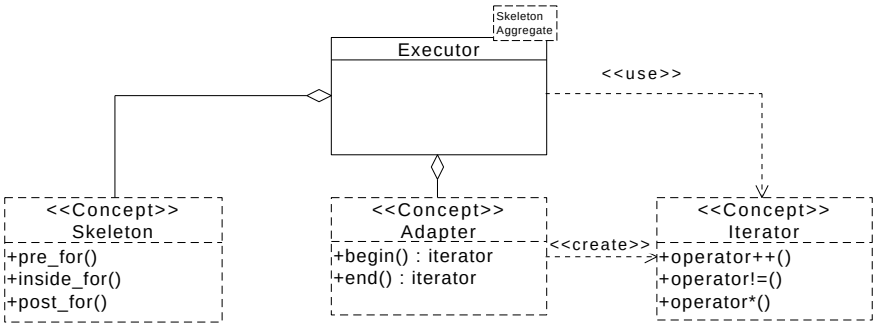


Figura 15: Diagrama de classes da biblioteca SKL3.

Como se pode ver na Figura 15, o Executor é uma classe genérica composta por duas outras classes o Skeleton e o Adapter, ambas passadas como parâmetros genéricos. De notar que o *Iterator* é essencial para que este consiga desempenhar a sua funcionalidade. O Iterator é o tipo do resultado de invocar os métodos begin e end a um objecto que implemente o conceito Adapter.

```
1  template<typename Adapter_t, typename Skeleton_t>
2  struct Executor
3  {
4    Skeleton_t skeleton;
5    Adapter_t adapter;
6
7    template<typename Iterator_t>
8    auto execute(Iterator_t begin, Iterator_t end, Skeleton_t skeleton)
9    {
10     if(begin != end)
11     {
12       skeleton.pre_for(*begin);
13       for (++begin; begin != end; ++begin)
14       {
15         skeleton.inside_for(*begin);
16       }
17     }
18     return skeleton.post_for();
19   }
20 };
```

Listagem 3.12: Implementação sequencial do Executor

Como se pode observar na Figura 3.12, o Executor é composto por duas classes o Skeleton_t e Adapter_t, às quais a classe delega responsabilidades relativas, respectivamente, aos esqueletos e à iteração. Por exemplo, na linha 12, o executor delega para o esqueleto através do método pre_for operações que este tem de realizar antes do ciclo. Analogamente é feito o mesmo para as etapas dentro do ciclo e depois do ciclo, respectivamente pelos métodos inside_for (linha 15) e post_for (linha 18). Neste caso a delegação é feita explicitamente, já no

caso do Adapter a delegação é feita na invocação do método execute que é feito da seguinte maneira: "e.execute(e.adapter.begin(), e.dapter.end(), s)", assumindo "e" como um objecto do tipo Executor, isto é, a invocação do método genérico execute recebe como argumentos objectos criados pelo adapter. Assim o adapter adapta os esqueletos fazendo com que as operações relativas à iteração sejam redireccionadas para um tipo de iterador diferente, com o comportamento desejado.

Note-se que os tipo Skeleton e Adapter não são classes abstractas mas sim conceitos, o que significa que é compilado um Executor para cada parametrização concreta destes conceitos. Ao contrário das interfaces/classes abstractas, a utilização destes não requer a virtualização dos seus métodos, o que permite uma melhor optimização do código por parte do compilador. Na prática o que acontece é a injecção de código em partes sinalizadas através do métodos que as classes concretas dos conceitos Skeleton e Adapter/Iterator implementam.

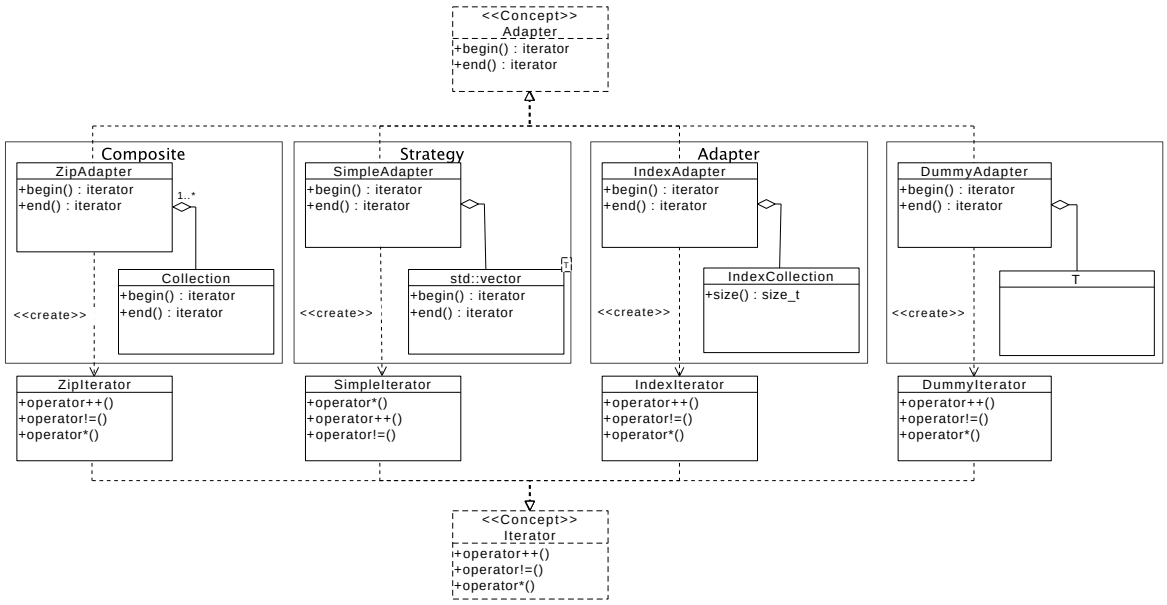


Figura 16: Diagrama de classes da interface *Adapters* e dos adaptadores que a implementam.

O diagrama da Figura 16 apresenta as classes da biblioteca que implementam o conceito *Adapter*. Estas classes servem para a adaptar o esqueleto ao código do utilizador. Ainda assim observe-se, por nota de curiosidade, que o acontece é que ao estes adaptadores serem conectáveis ao esqueleto modificam o padrão com que este esqueleto se relaciona com os objectos que se pretendem iterar.

O *ZipAdapter* simplesmente substitui os iteradores por um tuplo de iteradores envolvidos por um *mixin* que implementa esta interface de iterador e com um comportamento do padrão compósito, isto é, invoca a cada um dos elementos do tuplo os métodos disponibilizados

pela interface que implementa. O *mixin zip_iterator* utilizado é o mesmo que foi explicado para a versão anterior da biblioteca.

Por exemplo, o *SimpleAdapter* funciona como uma espécie padrão estratégia. Em que podem ser utilizados um conjunto, por exemplo, de *std::vector*<*T*> com tipos *T* diferentes, sendo que todos disponibilizam a mesma interface de iteração.

Já, por exemplo o IndexAdapter, permite alterar a interface utilizada, por exemplo quando o Executor invoca o método *end* este é redireccionado para uma interface diferente e é invocado o método que retorna o tamanho da colecção, fazendo com seja possível iterar através dos índices.

Por fim pode ver-se o DummyAdapter em que este é parametrizado com um tipo *T* e neste caso as funções de criação de iteradores retornam o mesmo objecto envolvido por um objecto com que implementa a interface iterador, mas com implementações vazias, ou seja, esse iterador simplesmente não itera, ficando sempre apontar para o mesmo objecto.

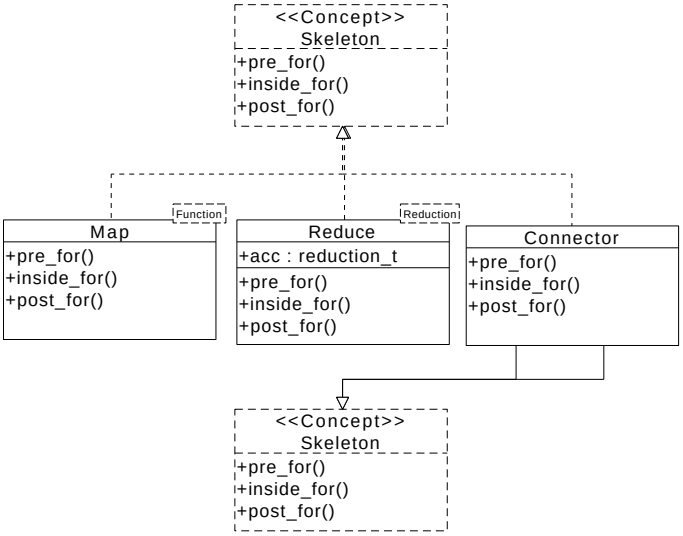


Figura 17: Diagrama de classes da interface *Skeleton* e dos esqueletos que a implementam.

A Figura 17 ilustra o conceito Skeleton e as classes que o implementam. Esta classe serve para fazer a fusão dos esqueletos. Como se pode constatar esta classe é composta por duas classes que implementam o conceito Skeleton, ambos parâmetros desta classes. O primeiro é um esqueleto simples, que pode ser o *map* ou o *reduce*, e o segundo é um outro Connector. A implementação dos métodos deste conector invocam o mesmo método cada um dos esqueletos, a invocação do próprio método ao segundo esqueleto, um *Connector*, criando assim uma recursividade de invocação do mesmo método a todos os esqueletos. O caso base desta recursividade é um *Connector* que tem como segundo esqueleto uma implementação em que os seus métodos são vazios.

A implementação dos esqueletos nesta versão é mais simples, estes apenas implementam os métodos que lhes são delegados pelo *Executor*.

```
1  struct reduce_impl
2  {
3    using reduction_t = typename std::decay_t<Fn>::result_type;
4    reduction_t reduction_;
5    Fn& fn_;
6
7    explicit reduce_impl(Fn& fn) : fn_(fn) { }
8
9    template<typename Iterator>
10   void pre_for(Iterator&& ite) { reduction_ = ite; }
11
12   template<typename Iterator>
13   void inside_for(Iterator&& ite) { reduction_ = fn_(reduction_, ite); }
14
15   auto post_for() { return std::make_tuple(reduction_); }
16 };
```

Listagem 3.13: Implementação sequencial do esqueleto *reduce*

A Listagem 3.13 ilustra a implementação do esqueleto reduce. Neste caso o functor é binário, assim a variável de redução é do tipo do retorno do functor. Nas linhas 3 e 4 pode ver-se a declaração de uma variável com o mesmo tipo que o functor binário genérico.

Na linha 10, pode observar-se a inicialização da variável de redução com o elemento apontado pelo iterador.

Na linha 13, pode ver-se a injecção do código da aplicação do functor ao corpo do ciclo através da invocação em linha do método *inside_for*, observe-se que neste caso o functor recebe como argumento a variável de redução e um elemento da colecção, o seu retorno é escrito de novo na variável de redução.

Por fim, na linha 15, pode ver-se que o retorno do método *post_for*, que é também o retorno do esqueleto, é feito através de um tuplo, para que no caso deste esqueleto fazer parte de uma fusão de esqueletos se possam concatenar os resultados com uma maior facilidade.

Parelelização

Nesta versão a camada de paralelização foi implementada com recurso a memória partilha, mas ao contrário da versão anterior, esta utilizou os mecanismos proporcionados pelo OpenMP. A paralelização requer também pequenos refinamentos sobre os esqueletos, por exemplo, para o esqueleto *reduce* é necessário uma barreira para sincronizar as *threads* de modo que a *thread* principal possa reduzir cada uma das variáveis de redução locais das *thread* sem que haja *data races*. O código da sincronização na barreira e redução das variáveis é injectado reescrevendo o método *post_for*.

```
1  template<typename Super>
2  struct Executor: Super
3  {
4    template<typename Iterator_t>
5    auto execute(Iterator_t begin, Iterator_t end, Super::Skeleton_t skeleton)
6    {
7      using result_t = decltype(Super::execute(begin, end, skeleton));
8      if constexpr(std::is_void_v<result_t>)
9      #pragma omp parallel firstprivate(skeleton)
10     {
11       int tid = omp_get_thread_num();
12       int n_threads = omp_get_num_threads();
13       auto s = scheduler::constant(tid, n_threads);
14       auto[b, e] = s.next(begin, end, skeleton);
15       Super::execute(b, e);
16     }
17     else
18     {
19       result_t res;
20       #pragma omp parallel firstprivate(skeleton)
21       {
22         int tid = omp_get_thread_num();
23         int n_threads = omp_get_num_threads();
24         auto s = scheduler::constant(tid, n_threads);
25         auto[b, e] = s.next(begin, end);
26         if(tid != 0) Super::execute(b, e, skeleton);
27         else res = Super::execute(b, e, skeleton);
28       }
29       return res;
30     }
31   }
32 };
```

Listagem 3.14: Implementação do *mixin* que refina o Executor

A Listagem 3.14 ilustra o *mixin* que faz o refinamento do *Executor* através de uma camada de OpenMP. Este código tem duas alternativa em tempo de compilação. No caso do resultado do esqueleto ser void (significa que só foram usados esqueletos sem retorno como o *map*) é compilado o código das linha 10 a 16, caso contrário é compilado o bloco de código entra as linhas 21 e 28.

Ambos os blocos são uma região paralela do OpenMP, nesta região cada uma das *threads* vai buscar o seu *tid* e o número total de *threads* que estão a executar (linhas 11 e 12). De seguida é alocado um escalonador com base nesses dados e são lhe passados o intervalo da colecção que se pretende aplicar o esqueleto (linha 13). O escalonador retorna um intervalo específico para cada *thread* e este intervalo é passado como argumento na invocação do método excute que faz a aplicação dos esqueletos especificados.

A diferença entre este código é a *thread* principal (com *tid* igual a zero) atribuir o resultado do retorna da invocação da execução dos esqueletos à variável *res* (linhas 26 e 27).

Observe se que a variável skeleton é declarada como sendo *firstprivate* (linha 9/20), o que significa que cada uma das *threads* aloca uma cópia local deste objecto e uma vez que as variáveis de redução são variáveis de instância dos *mixins* que compõe o esqueleto estas ficam automaticamente locais a cada uma das *threads*

4

RESULTADOS

AMBIENTE EXPERIMENTAL

Relativamente ao ambiente de teste, recorreu-se a uma máquina Linux, a correr o sistema operativo CentOS 6.3. Ao nível do hardware, a maquina utilizada tinha a seguinte especificação: 2 Xeon E5-2670v2, com 10 cores cada o que perfaz um total de 20 cores físicos. Esses cores possuem tecnologia Hyper-Threading Technology (HT) o que aumenta o número total de cores lógicos para 40.

METODOLOGIA

As medições do tempo de execução são referentes apenas à computação das diversas aplicações, isto significa que tempos como a criação da *threads* ou como inicialização de variáveis, entre outros, não foram contabilizados. A medição dos tempos de execução foram obtidos com recurso ao std::chrono disponibilizado no C++11. Cada medição advém da utilização da métrica *kbest* a um conjunto de execuções da aplicação, neste caso foram utilizados oito medições e feita a média das melhores três.

Para cada um dos casos de estudo são apresentados duas figura, uma onde se pode observar o impacto do uso dos esqueletos nas suas versões sequências e paralelas relativamente a uma implementação sequencial sem uso de esqueletos. Esta figura é apresentada como um gráfico de barras em que cada grupo corresponde a um tamanho de problema diferente. Os tamanho foram escolhidos de forma a que os dados do problema coubessem nos vários tipos de armazenamento (p.e., Cache L1, Cache L2, Cache L3, Random Access Memory (RAM)) da máquina. A outra figura é o gráfico de escalabilidade forte, isto é, foram medidos os tempos de execução de cada uma das versões paralelas do algoritmo, utilizando diferentes números de *threads*. Estes tempos de execução foram utilizados para calcular o *speedup* em relação a uma implementação sequencial sem esqueletos.

4.1 CASOS DE ESTUDO

4.1.1 *AXPY*

AXPY é um algoritmo que implementa uma das operações da álgebra linear. Esta operação é implementada na Basic Linear Algebra Subprograms (BLAS) e é uma rotina de nível um, o que faz desta uma rotina com complexidade linear. A rotina consiste na multiplicação de um escalar por um vector e a soma deste com um outro vector.

Algoritmo

Este é um algoritmo simples que consiste na iteração de ambos os vectores em simultâneo em que os elementos de um vector são multiplicados pelo valor *a* e o resultados somado com um elemento do vector Y.

A paralelização deste algoritmo advém do facto de o cálculo de cada posição do vector Y apenas depender da posição com o mesmo índice do vector X, deste modo cada posição pode ser calculada isoladamente e por consequente em paralelo.

Implementação

Na implementação deste algoritmo os vectores foram representados pelo tipo de dados `std::vector<double>`.

A implementação da paralelização é feita dividindo os vectores em segmentos, desta forma, em paralelo, são realizados por cada uma das threads a rotina *axpy* nos segmentos a si atribuídos.

Implementação com esqueletos

Como se pode observar na descrição do algoritmo este padrão de iteração dos dados é claramente o descrito pelo esqueleto `map`, em que o kernel é simplesmente a multiplicação da constante `a` por um valor correspondente a um elemento do vector X, e a soma deste resultado com um outro valor correspondente a um elemento do vector Y.

Análise de Resultados

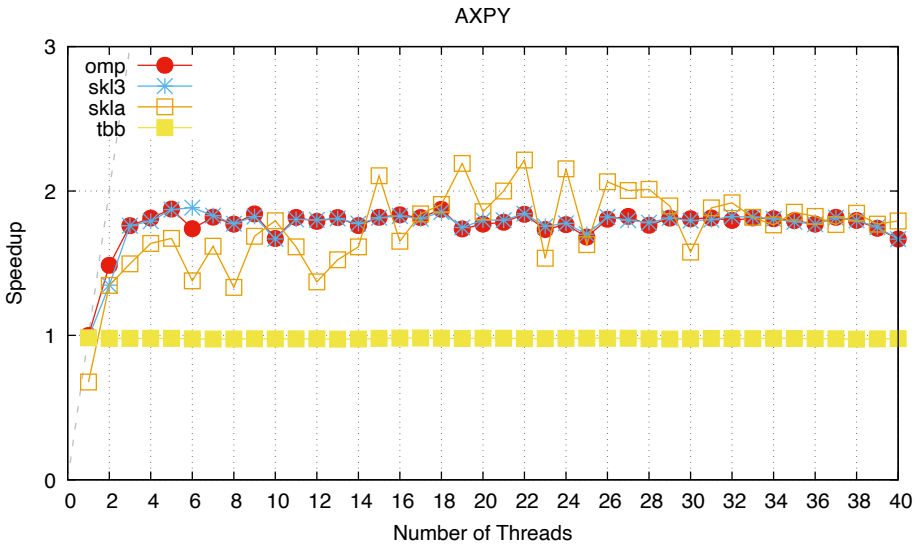


Figura 18: *Speedup* das várias versões do problema *axpy* paralelas em relação à versão sequencial sem esqueletos.

Como se pode observar na Figura 18, este algoritmo não aintgiu bons *speeedups* em relação à versão sequencial, o que já era esperado uma vez que este algoritmo é limitado pela memória.

Note-se que em ambas as versões que usam OpenMP para paralelizar o código, quer sejam utilizados esqueletos (caso do *skl3*), quer não sejam (versão *omp*), os *speedups* apresentados são idênticos.

De salientar que versão da biblioteca *skla*, que corresponde a versão da biblioteca SKL2 com a passagem do functor com argumentos, na paralelização com apenas uma *thread* teve um tempo de execução pior do que a versão sequencial, o que se deve a dois motivos. O primeiro foi a camada de paralelização utilizar *ParallelThreadPool*. O uso desta faz com que na realidade sejam utilizadas duas *threads* - uma principal, e uma que realiza o trabalho. Isto faz com que os dados alocados na *thread* principal sejam posteriormente enviados para a outra *thread*. Houve outro potencial motivo, que foi o código não ter sido vectorizado, possivelmente pelo uso de uma lambada em que a variável *a* é declarada como mutável. Ainda assim, com o aumento das *threads* esta diferença de desempenho acabou por ser atenuada.

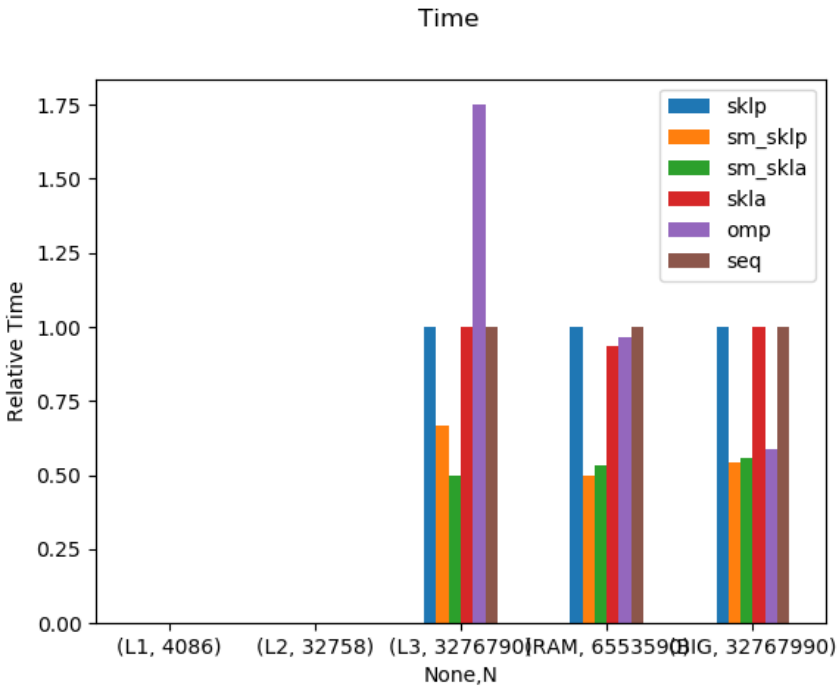


Figura 19: *Speedup* do problema *axpy* em relação à versão sequencial sem esqueletos, com tamanhos de problemas diferentes

Na figura 19 pode observar-se que para os diversos tamanhos do problema, em que foi possível realizar medições, as versões sequências apresentam os mesmo tempos de execução, isto é, tempos relativos igual a um, face à implementação sequencial sem esqueletos.

4.1.2 *Kmeans*

O algoritmo Kmeans consiste no agrupamento de um conjunto de dados, ou seja dados um conjuntos de n observações o algoritmo pretende agrupar estas em k partições relativamente à sua semelhança. Este algoritmo encontra-se na classe de problemas np-dificil. Apesar disso, existem várias heurísticas que computam soluções óptimas locais.

Neste caso foi implementada uma dessas heurísticas, conhecida como "*Lloyd's algorithm*", num espaço euclidiano bidimensional e a relação de semelhança utilizada foi a distância euclidiana.

Esta heurística apesar de não ser a que tem o melhor tempo de convergência nem o melhor resultado tem uma implementação relativamente simples. Ainda assim esta heurística tem as suas particularidades que a tornam um pouco complexa na sua implementação usando esqueletos. Por exemplo, a função de redução dos pontos no centroide associado é complexa pois o resultado da redução é uma colecção. Outro exemplo é o facto de este algoritmo

necessitar de iterar múltiplas colecções simultaneamente, o que levou à necessidade de acrescentar esta funcionalidade à biblioteca.

Algoritmo

O *Lloyd's algorithm* é caracterizado por duas fases:

- A primeira é a atribuição de um centroide a cada observação, esta atribuição é feita de modo que a cada observação seja atribuído o centroide mais próximo. Para esta fase é então necessário iterar todas as n observações e para cada uma destas iterações computar a distancia a todos os k centroides e por fim é necessário computar o mínimo das distancias calculadas.

- A segunda consiste no cálculo das novas médias dos centroides. Ou seja, é necessário somar todas as distancias de um centroide para as observações a si atribuídas para todos centroides e dividir pelo número de atribuições feitas ao respectivo centroide.

Estas duas fases são iteradas repetidamente até que não existam alterações na fase de atribuição dos centroides às observações. Desta forma a heurística encontra uma solução local óptima tal que o valor da soma das distâncias das observações aos respectivos centroides é mínima.

De notar que durante o processo iterativo podem não ser atribuídas quaisquer observações a um centroide. De modo a resolver este problema, o centroide é deslocado para a posição da observação com a maior distância ao seu respectivo centroide.

Implementação sequencial e paralela

As implementações sequências e paralelas deste algoritmo podem ser consultadas em anexo, Listagens B.2 e B.1.

Implementação com esqueletos

Para a implementação com esqueletos é necessário identificá-los primeiro. Assim foram identificados dois esqueletos. O primeiro é facilmente identificado como sendo o esqueleto *map* uma vez que se trata de computar a distância de um ponto a todos os centroides, para todos os pontos. O segundo esqueleto pode não ser tão evidente, uma vez que se tratam de duas reduções: a primeira acumula os pontos numa colecção, que representa os centroides; e a segunda redução contabiliza as diferenças das atribuições dos pontos ao centroides.

A redução encontra-se imediatamente a seguir, e em ambos os casos a colecção a iterar é o conjunto dos pontos. Por este motivo deve utilizar-se o esqueleto *map_reduce* pois este é mais eficiente, dado que, ao invés de iterar duas vezes a colecção - uma vez para o *map* e outra para *reduce* - é iterada apenas uma vez.

```
1  void compute_k_means(vector<point>& points,
2                       vector<cluster_id>& id,
3                       vector<point>& centroid)
4  {
5    centroid_ = &centroid;
6    K_ = centroid.size();
7    std::vector<size_t> changes(points.size(), 0);
8    auto view = reduce<view_sum_and_count>(points, id, changes);
9    do
10   {
11     repair_empty_clusters(points, id, centroid, view.centroid_acc);
12     for(size_t j=0; j<centroid.size(); ++j)
13       centroid[j] = view.centroid_acc[j].mean();
14     view = map_reduce<compute_clusters, view_sum_and_count>(points, id, changes);
15   } while(view.change != 0);
16 }
```

Listagem 4.1: Implemntação do algoritmo Kmeans utilizando a biblioteca de esqueletos SKL2

Como se pode verificar pela Listagem 4.1 a implementação do algoritmo *Kmeans* utilizando a biblioteca esqueletos SKL2 permite um abstracção maior relativamente à implementação do algoritmo e da sua paralelização, face às versões OpenMP e TBB.

```
1  void compute_k_means(std::vector<point>& points,
2                       std::vector<cluster_id>& ids,
3                       std::vector<point>& centroid)
4  {
5    k_centroid = centroid.size();
6    auto[centroid_acc] = skl::composition(skl::zip(points, ids, skl::dummy<size_t>(0)),
         skl::reduce(insert_point(), point_assigment));
7    size_t changes = 0;
8    do
9    {
10     repair_empty_clusters(points, ids, centroid, centroid_acc.acc);
11     for(size_t j=0; j<k_centroid; ++j) centroid[j] = centroid_acc[j].mean();
12     std::tie(centroid_acc, changes) = skl::composition(skl::zip(points, ids, skl::
           dummy<size_t>(0))
13     , skl::map(compute_clusters(centroid))
14     , skl::reduce(insert_point(), [](auto& ite) { return std::make_tuple(std::get<1>(
           ite), std::get<0>(ite)); })
15     , skl::reduce(std::plus<size_t>(), skl::get<2>()) );
16   } while(changes != 0);
17 }
```

Listagem 4.2: Implemntação do algoritmo Kmeans utilizando a biblioteca de esqueletos SKL3

Na Listagem 4.2 pode ver-se a implementação do algoritmo com a versão *SKL3* que permitiu simplificar e melhorar alguns aspectos. Por exemplo, na linha 6, a biblioteca evita trabalho redundante, uma vez permite utilizar uma redução com um functor mais simples, sendo depois reutilizado (linha 14) fazendo a fusão com outra redução. A fusão

disponibilizada permitiu também uma melhor expressividade através do uso de functores mais simples. Por exemplo, nesta versão facilmente se observa que o esqueleto da linha 12, executa duas reduções: uma que representa a acumulação de pontos por um centroide, e a outra que correspondes à contabilização das alterações ocorridas.

Para além disso, nesta versão, através do adaptador *dummy*, foi possível evitar a alocação do vector changes da versão anterior, uma vez que através deste adaptador é possível partilhar esta variável pelos diversos functores, passando a variável como argumento dos mesmos.

Análise de Resultados

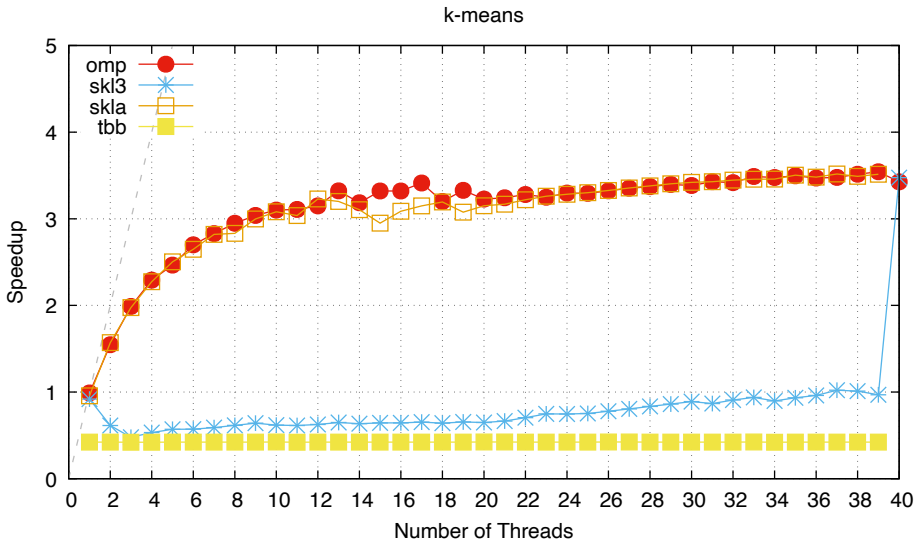


Figura 20: *Speedup* das várias versões do problema *Kmeans* paraleleas em relação à versão sequencial sem esqueletos.

Na Figura 20 pode observar-se o *speedup* das várias versões. Os *speedups* alcançados foram piores do que o que era esperado para esta algoritmo. O que se deveu ao facto de uma das partes do código não estar paralelizado. Essa parte corresponde à rotina *repair_empty_clusters* utilizada em toda as implementações.

Como se pode ver na figura, as versões *omp* e *skla* têm *speedups* idênticos. Já a versão *skl3*, não alcançou um bom desempenho, ficando pior do que a implementação sequencial. O que se deve a um problema na implementação da camada paralela da biblioteca. O problema ocorre na fusão de múltiplas reduções. Esta fusão faz com que as *threads* sejam sincronizadas através de uma barreira por cada redução utilizada na fusão, quando só é necessária uma sincronização no final do ciclo.

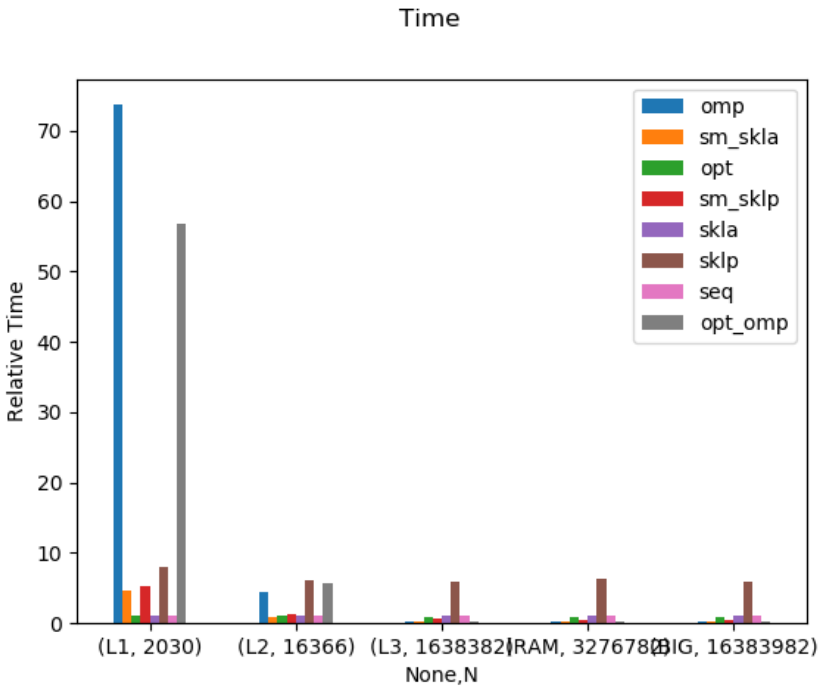


Figura 21: *Speedup* do problema *Kmeans* em relação à versão sequencial sem esqueletos, com tamanhos de problemas diferentes

Na Figura 21 pode verificar-se mais uma vez que as versões sequenciais com o uso de esqueletos não tem impacto significativo na performance. Observar-se que a utilização do OpenMP tem um impacto bastante negativo causado pela sobrecarga da gestão das *threads*. Esta sobrecarga é apenas notável em problemas com uma reduzida quantidade da dados. Ainda assim tentou-se fazer uma implementação (na figura representada por *opt_omp*) mais optimizada (de mais baixo nível) também com recurso ao OpenMP, mas como se pode verificar os ganhos não foram suficientes para se tornar competitiva com as implementações das versões paralelas da biblioteca de esqueletos.

De notar que a versão *sklp* apresenta um impacto negativo, demorando para qualquer quantidade de dados cerca de cinco vezes mais tempo a executar comparativamente à versão sequencial sem esqueletos. Isto não se deve ao uso de esqueletos mas sim a uma limitação desta variante do esqueleto causada pela passagem do functor como parâmetro. A passagem do functor como parâmetro permite alocar um objecto dentro da *stack* de uma rotina invocada (mais interior), evitando a passagem de argumentos. O que neste caso não se revelou vantajoso, pelo contrario diminui a performance, uma vez que o functor, por exemplo no caso do *reduce*, conter estado, neste caso uma colecção de centroides, que se pretende retornar. Desta forma retornar uma variável da rotina chamada para a chamadora requer uma cópia da variável. Já na versão da passagem por argumento o functor pode ser

passado por referência o que significa que a rotina invocada altera directamente variáveis da rotina chamadora.

4.1.3 *Multiplicação de Matrizes*

A multiplicação de matrizes é também um algoritmo da álgebra linear. Este algoritmo opera de sobre duas matrizes (A e B) e uma matriz de resultado (C). O algoritmo original de multiplicação de matrizes opera sobre matrizes com um qualquer número de linhas e colunas, apenas com a condição de que o número de linhas da matriz A é igual ao número de colunas de B.

Assumindo n como o número de elementos de uma linha da matriz, a complexidade deste algoritmo é dada pela expressão n^3. Uma vez que são iterados todos os elementos da matriz, ou seja, os n^2 elementos e por cada elemento é invocada a função *axpy* sobre uma linha e uma coluna das matrizes, em que este algoritmo tem uma com complexidade linear.

Algoritmo

O algoritmo mais comum é a aplicação de um produto interno, para cada elemento da matriz c_{ij}, assim este elemento resulta do produto interno da linha i de a pela coluna j de b, que pode ser descrito pela seguinte formula matemática:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj}$$

Implementação

As matrizes são representadas por arrays de doubles, e portanto são completamente contíguas na memória. Assim numa matriz a$n \times n$ x o acesso do elemento x_{ij} é feito através de $x[i * n + j]$. Na implementação deste algoritmo foram utilizados três arrays de doubles... O algoritmo de multiplicação de matrizes implementado é aplicado apenas a matrizes quadradas.

Este algoritmo é descrito com recurso a três ciclos, dois desses ciclos servem para iterar a matriz c, um ciclo para iterar os índice da linha e o outro para iterar os índice da coluna, o último ciclo serve para iterar tanto sobre a i-ésima linha da matriz a como sobre j-ésima coluna de b. Assim o algoritmo itera cada elemento da matriz c um única vez e por cada elemento de c aplica um produto interno de uma linha de a com uma coluna de b. Esta é uma versão ineficiente dado que a matriz b é iterada por coluna e elementos seguidos da coluna não se encontram contíguos na memória da máquina, o que prejudica a vectorização, e para além disso em caso de matrizes grandes aumenta a possibilidade de *cache misses* uma vez que existe um maior espaçamento entre elementos consecutivos de uma coluna.

Outras expressões equivalentes podem ser derivadas da apresentada anteriormente. Estas têm como efeito prático a alteração da ordem dos ciclos que se traduz numa forma diferente de iterar os elementos das matrizes. Por exemplo é possível obter um algoritmo que não itera a matriz b por colunas mas por linhas, tornando a execução do mesmo mais eficiente. Assim nesta versão é iterada a matriz a e para cada elemento desta é realizada uma operação *axpy*, em que cada elemento a_{ij} corresponde à constante a do *axpy*, e portanto este valor e multiplicado ponto a ponto com uma linha de b e esta é somada a uma linha de c.

Análise de Resultados

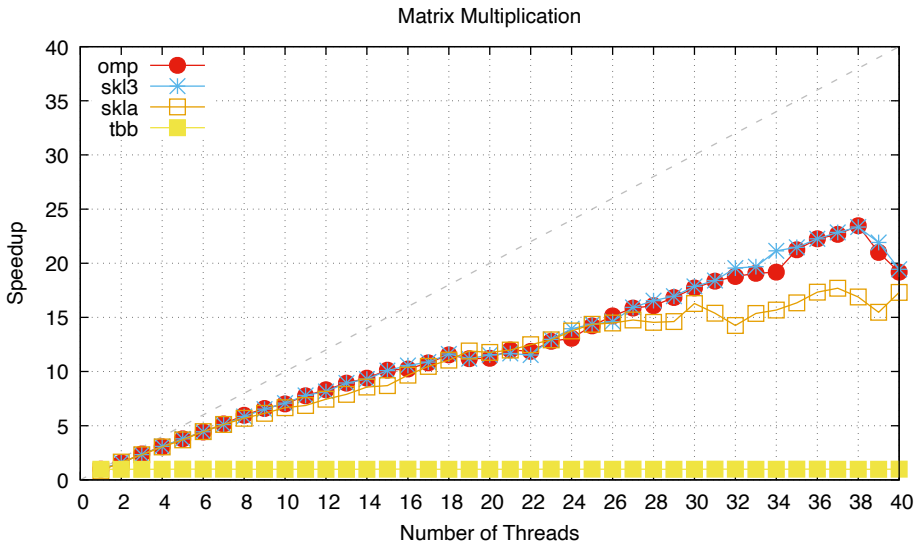


Figura 22: *Speedup* das várias versões do problema *Matrix Multiplication* paralelas em relação à versão sequencial sem esqueletos.

Na Figura 22 pode observar-se que as implementações obtiveram uma boa escalabilidade. O que era esperado deste algoritmo, uma vez que, a maioria, dos algoritmos com maior grau de complexidade tendem a conseguir tirara mais proveito do paralelismo. Por exemplo, neste algoritmo com aumento do tamanho do problema, aumenta-se a carga de trabalho por elemento, o que é positivo pois permite aumentar a localidade da execução. Isto seria ainda mais evidente num gráfico de escalabilidade fraca.

As curvas *omp* e *skl3*, como se pode ver no gráfico, aumentaram sempre a escalabilidade sempre que se aumentou o número de *threads* mais uma vez estas duas versões tiveram desempenhos idênticos. De notar a partir das vinte *threads*, as *threads* utilizadas passam a ser *hyper-threads*, o que resulta em acréscimos de escalabilidade menores. Já a curva *skla* sofreu de uma maior estagnação ao começar a utilizar *hyper-threads*. Este diferença pode ser explicada, não pelo uso de esqueletos, mas pelo facto de as curvas *omp* e *skl3* serem

implementadas com OpenMP enquanto que a versão *skla* utiliza uma implementação criada com recurso às *threads* disponibilizadas pela biblioteca estandarte do C++.

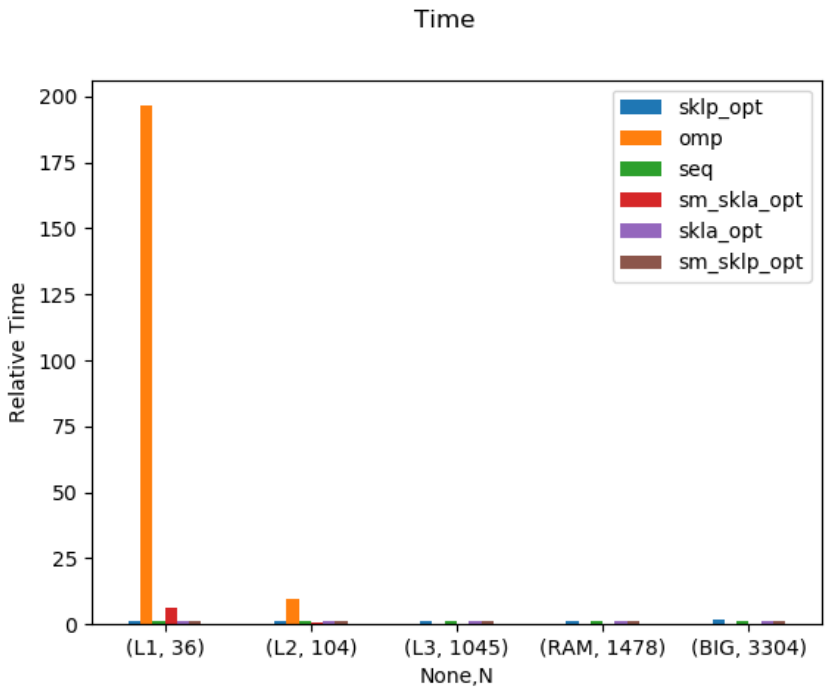


Figura 23: *Speedup* do problema *Matrix Multiplication* em relação à versão sequencial sem esqueletos, com tamanhos de problemas diferentes

A Figura 23 é idêntica há Figura 21 do algoritmo *kmeans*, pelo que se faz uma análise análoga. As implementações das versões sequenciais dos esqueletos têm um impacto negligenciável relativamente a uma implementação sem o uso de esqueletos, apesar de em alguns casos dificultar a vectorização que é feita automaticamente pelo compilador. O único impacto, substancial, que se observa é o do uso de OpenMP, que para algoritmos com poucos dados o trabalho realizado em paralelo não compensa a sobrecarga da gestão das *threads*.

Importa referir que ocorreram alguns problemas técnicos com ambiente de teste. Pelo que, a curva da versão das implementações da versão SKL2 com a passagem do functor como parâmetros não se encontra nos gráficos apresentados nesta secção. Ainda assim podem ser consultados em anexo C, resultados preliminares aonde se verificou as diferenças entra as versão da SKL2. Para além disso, como se pode observar, não se conseguiu obter a curva de escalabilidade para a implementação com recurso ao *TBB*.

5

CONCLUSÃO

Esta dissertação incidiu sobre duas áreas que são hoje mais do que nunca fundamentais para o mundo da programação, a arquitectura do software e programação paralela (com foco na heterogeneidade do hardware). Estes duas áreas são muitas vezes difíceis de conciliar, pois enquanto que por um lado se procurava uma forma de estruturar melhor o código por outro procurava-se uma biblioteca com um bom desempenho o que nem sempre é possível. Assim o caminho fez-se procurando abstracções que tivessem um custo nulo em tempo de execução, isto é, com recurso a variadíssimas técnicas de metaprogramação disponibilizadas pelo C++.

Pelos resultados obtidos, apesar de algumas limitações, pode-se dizer que é possível conciliar as duas áreas. Mais concretamente, ao nível da arquitectura implementou-se uma biblioteca dos já conhecidos esqueletos algorítmicos, que por si só melhoram consideravelmente a estruturação do código. Um dos requisitos era a estruturação através de camadas de forma a adaptar as aplicações ao ambiente usado na execução da mesmas. Este ponto foi parcialmente concluído. Parcialmente concluído porque apesar de terem sido implementadas duas camadas ambas utilizam o mesmo tipo de paralelização (com recurso a memória partilhada), assim servem apenas como prova do conceito. Outro dos pontos importantes, que de certa forma apareceu os com a evolução da biblioteca foi facultar os esqueletos com *plugabble adapters* de forma a aumentar a usabilidade dos mesmos. Isto enquadrou-se perfeitamente com o âmbito desta tese, uma vez que estes *plugabble adapters* foram desenvolvidos criando uma nova camada de software, relativamente aos dados que se pretendem iterar. Assim foi possível manter uma boa estruturação interna da biblioteca, o que permitiu uma forma fácil de integrar novas variantes dos esqueletos sem replicar código. Para além disso os esqueletos foram implementados como *mixins*, o que permitiu a fusão dos esqueletos em tempo de compilação. Como referido anteriormente, isto faz com que seja possível utilizar functores mais pequenos e menos complexos, o que os torna mais reutilizáveis uma vez que ficam menos específicos da aplicação. Além disso, a fusão aumenta a abstracção do uso dos esqueletos e melhora expressividade do código.

Ao nível da performance existiram dois focos principais. O primeiro foi a passagem dos functores por parâmetro que se antevia que pudessem atingir uma melhor desempenho mas

que acabou por não se verificar. Apresentando performances semelhantes às da passagem por argumentos, apesar de alguns casos com pior desempenho, que foram resolvidas com a introdução de novas variantes do esqueletos. Em relação ao segundo foi possível desenvolver a biblioteca de modo a que seja competitiva com tecnologias como OpenMP ou TBB.

Em suma, com esta dissertação pode comprovar-se que é possível ter uma boa estruturação do código paralelo através do uso de uma biblioteca esqueletos sem haver um impacto negativo na performance.

5.1 TRABALHO FUTURO

Como se pode ver na analise de resultados, a versão SKL3 apresentou alguns casos com uma fraca performance, os problemas foram identificados. Pelo que a realizar um trabalho futuro este começaria por eliminar as limitações das quais se destacam:

1. uso de múltiplas barreiras na fusão dos esqueletos quando só é necessária uma;
2. a não vectorização do código de alguns dos functores;
3. certificar que os métodos do *mixin* são efectivamente compilados em linha, com o atributo; *"__attribute__((always_inline))"*.

Estando estes problemas solucionados o trabalho futuro passaria por:

1. implementar mais esqueletos;
2. implementar mais adaptadores (experimentar implementar o *stencil* através de uma adaptação do esqueleto map);
3. implementar diversas camadas de paralelização;
4. utilizar o padrão estratégia para *tunnig* dos esqueletos, por exemplo implementar uma família de escalonadores que possam ser permutáveis entre si.

Uma biblioteca em que, especificamente, os itens 3 e 4, fossem implementados, faria com que fosse possível pensar em automatizar:

1. na escolha camadas de paralelização;
2. a escolha do *tunning* indicado para cada camada.

Isto poderia ser feito acrescentando lógica à camada representada pela API, de tal forma que esta fosse capaz de tomar decisões sobre estes dois ponto quer estática, quer dinamicamente. Para a selecção das camadas e *tunning* indicados poderiam ser utilizadas como métricas o tamanho dos functores, quantidade de escritas leituras dos functores, quais os esqueletos

utilizados, tamanho da colecções, tamanho dos elementos das colecções. A forma para tomar uma decisão poderia ser feita através de vários métodos como árvores de decisão, *machine learning*, entre outras.

BIBLIOGRAFIA

[1] M. I. Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[2] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[3] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

[4] A. E. L. L. C. Kessler. Flexible and type-safe skeleton programming for heterogeneous parallel systems. 2016.

[5] R. C. Martin. Clean architecture, 2017.

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

[7] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570. Springer, 1998.

[8] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

A

TABELA DE RESULTADOS

#n_threads	omp	sm_skl3	sm_skla	tbb
1	50329	51084.3	74243	51174.3
2	33854.3	37318	37375.3	51374
3	28581.3	28592	33646	51372.3
4	27760.7	28095.7	30735	51356.3
5	26827.7	26889	30090.7	51364.7
6	28928	26692.3	36504.7	51527.7
7	27653.7	27501.7	31101	51599
8	28408.7	28359.7	37740.3	51573.3
9	27324.3	27593.3	29820	51497
10	30095.3	30159.3	28040.3	51552.3
11	27688.3	27893	31180.3	51566.3
12	28112	27953	36715	51489.3
13	27696.7	27802.7	33014.3	51638.7
14	28563.3	28351.3	31184.7	51585
15	27672	27714	23894	51381
16	27419	27536	30449.3	51258.7
17	27704.7	27825	27320.7	51193
18	26853.7	27126	26386.7	51295
19	28958.3	28790.7	22959.3	51395.7
20	28411.3	27968	27088.3	51291.7
21	28208	27985.7	25152	51307.3
22	27354.7	27290	22713.7	51438
23	29023.3	28654.3	32781	51460.3
24	28470.3	28317.3	23356.7	51357.7
25	29929.7	29590.7	30885.3	51331.7
26	27882	27639.3	24376.3	51291.7
27	27583.7	27976.3	25105.3	51309.7
28	28537.3	28250	25005.3	51503.7
29	27757	27743.3	26519.3	51618.7
30	27828.3	28026	31891	51493
31	27744	27918.3	26712	51411
32	28010.7	27589.3	26228.7	51401.3
33	27738	27643.7	27672	51463
34	27832.3	27837.3	28501	51379.3
35	28027	28143.3	27150.3	51389.3
36	28410.3	28427	27581	51438
37	27671.3	27714.7	28450.3	51513.7
38	28002.3	28005	27254.3	51661.7
39	28888.3	28729.3	28403	51562.7
40	30172.3	30182.3	28064	51445.7

Tabela 4: Tempos de execução, em micro segundo, da aplicação *axpy*

#n_threads	omp	sm_skl3	sm_skla	tbb
1	1.19922e+07	1.29058e+07	1.23956e+07	28119546
2	7670999	1.93121e+07	7.55356e+06	2.81019e+07
3	5.96644e+06	2.48482e+07	6006529	28178543
4	5.17638e+06	2.22699e+07	5.21926e+06	2.81496e+07
5	4.81324e+06	2.0796e+07	4.73982e+06	2.81355e+07
6	4.3951e+06	20742508	4482373	28116951
7	4195515	2.0139e+07	4.20692e+06	28131393
8	4.02447e+06	19253457	4189735	2.81186e+07
9	3.90583e+06	1.84889e+07	3957729	2.80885e+07
10	3.82623e+06	1.91709e+07	3851367	2.80316e+07
11	3.81825e+06	19331870	3.90599e+06	2.81516e+07
12	3.76894e+06	1.89574e+07	3.67454e+06	28125615
13	3.57223e+06	18287015	3.7085e+06	2.8112e+07
14	3725207	1.86625e+07	3821228	2.8123e+07
15	3572634	1.84143e+07	4.02318e+06	2.80688e+07
16	3.57243e+06	18416734	3843419	28118508
17	3.47752e+06	1.81538e+07	3.76804e+06	28092487
18	3.70742e+06	1.84958e+07	3.71432e+06	2.81231e+07
19	3564643	18125123	3856737	2.80414e+07
20	3.67707e+06	1.82893e+07	3.76675e+06	28012352
21	3.66042e+06	1.78518e+07	3744255	28021269
22	3.61823e+06	1.68242e+07	3685070	28089375
23	3.65264e+06	15857774	3640183	2.81367e+07
24	3.59959e+06	1.58978e+07	3.61406e+06	28102992
25	3.6034e+06	1.57618e+07	3.59405e+06	2.81361e+07
26	3571449	1.52499e+07	3.56658e+06	28031368
27	3.53937e+06	1.47332e+07	3.53702e+06	28030987
28	3519965	1.42185e+07	3.51178e+06	2.81159e+07
29	3.4931e+06	1.37842e+07	3.48647e+06	28109893
30	3.50614e+06	1.33332e+07	3469679	28126615
31	3.4616e+06	1.36665e+07	3.46144e+06	28098459
32	3.47178e+06	1.30613e+07	3443983	28118201
33	3.40158e+06	1.26451e+07	3.43358e+06	2.80377e+07
34	3.41395e+06	1.32117e+07	3.43141e+06	2.81395e+07
35	3.3945e+06	1.27058e+07	3.38759e+06	28095800
36	3419737	12355847	3409620	2.81035e+07
37	3.41264e+06	1.16252e+07	3.37403e+06	2.80925e+07
38	3.37748e+06	11743641	3.40003e+06	2.81023e+07
39	3349777	12246987	3377034	2.81343e+07
40	3462460	3414429		

Tabela 5: Tempos de execução, em micro segundos, da aplicação *kmeans*

#n_threads	omp	sm_skl3	sm_skla	tbb
1	2.85475e+07	2.89946e+07	34597622	2.89283e+07
2	18307705	1.76353e+07	17279203	28932032
3	11890941	1.22151e+07	1.2228e+07	2.8926e+07
4	9302959	9.19831e+06	9350557	2.89222e+07
5	7.49903e+06	7498222	7.69469e+06	2.8856e+07
6	6.42342e+06	6.39809e+06	6.38162e+06	2.88419e+07
7	5492488	5.55881e+06	5.55306e+06	2.88657e+07
8	4759939	4.85042e+06	4.99661e+06	2.89412e+07
9	4309320	4.34702e+06	4.61915e+06	2.89152e+07
10	4.04759e+06	4.00839e+06	4265633	2.89373e+07
11	3661792	3.64339e+06	4.13346e+06	2.88786e+07
12	3.41662e+06	3482688	3816134	2.89363e+07
13	3.17912e+06	3.17614e+06	3.59636e+06	28937099
14	3.02421e+06	3.05036e+06	3313401	2.89279e+07
15	2.80477e+06	2.81384e+06	3.26556e+06	2.88622e+07
16	2775899	2.68827e+06	2.93638e+06	2.88409e+07
17	2628843	2.6123e+06	2.7108e+06	28895414
18	2.46057e+06	2.44306e+06	2.56935e+06	2.8879e+07
19	2538264	2.53823e+06	2.38672e+06	2.88846e+07
20	2.5317e+06	2.46259e+06	2.41358e+06	2.8931e+07
21	2.36682e+06	2433852	2369360	2.88984e+07
22	2394343	2.46513e+06	2.27493e+06	2.88807e+07
23	2.22036e+06	2.18895e+06	2196330	2.88467e+07
24	2.17194e+06	2035546	2065706	28886152
25	1.99799e+06	1.97573e+06	1.97739e+06	2.88726e+07
26	1.87656e+06	1.94258e+06	1.96119e+06	28897970
27	1.79143e+06	1.78431e+06	1.92383e+06	2.88667e+07
28	1.76665e+06	1.71632e+06	1.95232e+06	2.89352e+07
29	1.68692e+06	1674951	1943122	28914023
30	1.60274e+06	1590217	1.74497e+06	2.89271e+07
31	1.54841e+06	1543612	1846248	2.88682e+07
32	1512718	1452811	1992713	2.88498e+07
33	1.48901e+06	1.44276e+06	1.84876e+06	2.89052e+07
34	1.48116e+06	1342921	1.8122e+06	28891272
35	1.3367e+06	1322718	1737618	28867829
36	1275909	1.27627e+06	1.63896e+06	2.8867e+07
37	1253097	1.24321e+06	1.60506e+06	2.88895e+07
38	1210733	1.21703e+06	1.68268e+06	28885307
39	1.35218e+06	1.29554e+06	1.83547e+06	2.89091e+07
40	1.48096e+06	1.46057e+06	1.64148e+06	2.88626e+07

Tabela 6: Tempos de execução, micro segundos, da aplicação *Matrix Multiplication*

B

LISTAGENS

```
1  void compute_k_means(std::vector<point>& points,
2                       std::vector<cluster_id>& id,
3                       std::vector<point>& centroid)
4  {
5    const auto n = points.size();
6    const auto k = centroid.size();
7    tls_type tls([&]{return k;});
8    view global(k);
9    tbb::parallel_for(
10     tbb::blocked_range<size_t>(0,n),
11     [=,&tls,&global, &id]( tbb::blocked_range<size_t> r )
12     {
13       view& v = tls.local();
14       for( size_t i=r.begin(); i!=r.end(); ++i )
15       {
16         v.array[id[i]].tally(points[i]);
17       }
18     }
19   );
20   size_t change;
21   do
22   {
23     reduce_local_sums_to_global_sum( k, tls, global );
24     repair_empty_clusters(points, id, centroid, global.array);
25     for( size_t j=0; j<k; ++j )
26     {
27       centroid[j] = global.array[j].mean();
28       global.array[j].clear();
29     }
30     tbb::parallel_for(
31       tbb::blocked_range<size_t>(0,n),
32       [=,&tls,&global, &id]( tbb::blocked_range<size_t> r ) {
33         view& v = tls.local();
34         for( size_t i=r.begin(); i!=r.end(); ++i ) {
35           cluster_id j = reduce_min_ind(centroid, points[i]);
36           if( j!=id[i] )
37           {
38             id[i] = j;
39             ++v.change;
40           }
41           v.array[j].tally(points[i]);
```

```
42          }
43        }
44      );
45      reduce_local_counts_to_global_count( tls, global );
46    } while(global.change != 0);
47 }
```

Listagem B.1: Exemplo da implementação do algoritmo kmeans utilizando o TBB

```
1 #pragma omp declare reduction(\
2 vec_sum_and_count_reducer\
3 : std::vector <sum_and_count>\
4 : std::transform(\
5 omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus <
      sum_and_count>()))\
6 initializer(omp_priv = decltype(omp_orig)(omp_orig.size()))
7
8
9 void compute_k_means(std::vector<point>& points, std::vector<cluster_id>& id, std::
      vector<point>& centroid)
10 {
11   const auto n = points.size();
12   const auto k = centroid.size();
13
14   std::vector<sum_and_count> centroid_acc(k);
15   // Create initial clusters and compute their sums.
16 #pragma omp parallel for shared(n, id, points) default(none) reduction(
      vec_sum_and_count_reducer \
17                                     : centroid_acc)
18   for (size_t i = 0; i < n; ++i)
19   {
20     centroid_acc[id[i]].tally(points[i]);
21   }
22
23   // Loop until ids do not change
24   size_t change;
25   do
26   {
27     change = 0;
28     // Repair any empty clusters
29     repair_empty_clusters(points, id, centroid, centroid_acc);
30
31     // "Divide step": Compute centroids from global sums
32     for (size_t j = 0; j < k; ++j)
33     {
34       centroid[j] = centroid_acc[j].mean();
35       centroid_acc[j].clear();
36     }
37
38     // Compute new clusters and their local sums
39 #pragma omp parallel for shared(n, points, centroid, id) default(none) reduction(
      vec_sum_and_count_reducer \
40                                     : centroid_acc) reduction(+ : change)
41     for (size_t i = 0; i < n; ++i)
```

```
42      {
43        // "Reassign step": Find index of centroid closest to points[i]
44        cluster_id j = reduce_min_ind(centroid, points[i]);
45        if (j != id[i])
46        {
47          id[i] = j;
48          ++change;
49        }
50        // "Sum step"
51        centroid_acc[j].tally(points[i]);
52      }
53    } while (change != 0);
54  }
```

Listagem B.2: Exemplo da implementação do algoritmo kmeans utilizando o OpenMP

C

FIGURAS

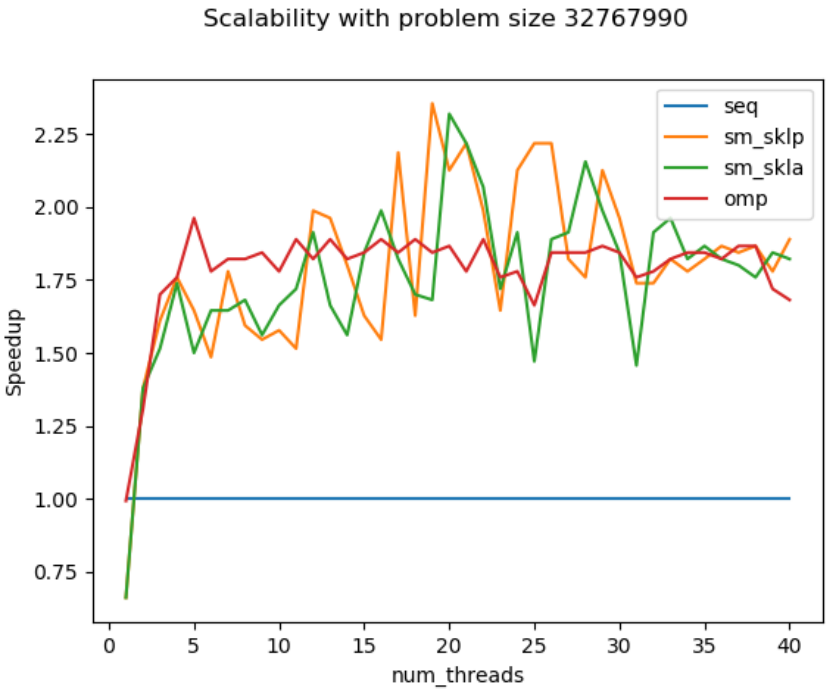


Figura 24: Escalabilidade das várias implementações da aplicação *axpy*.

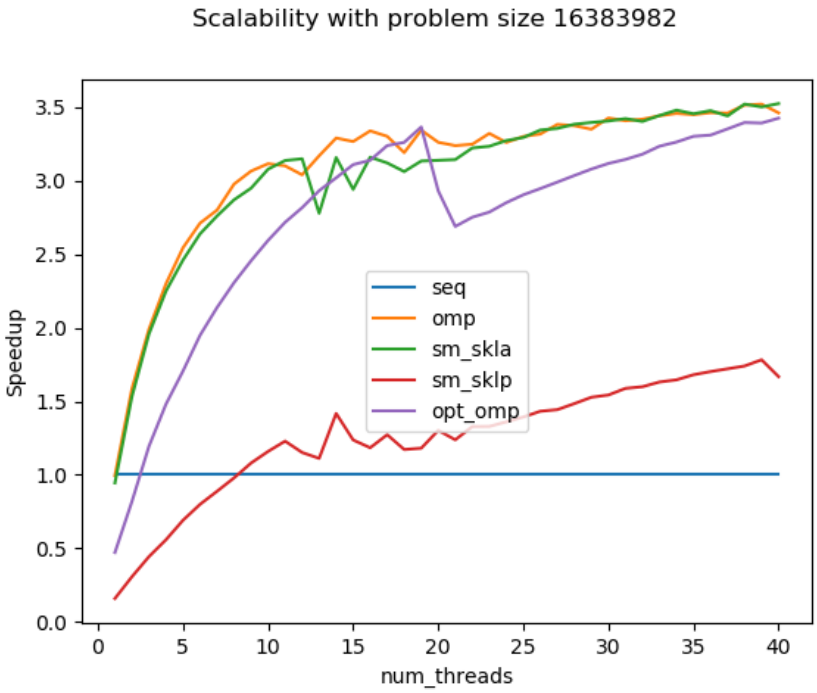


Figura 25: Escalabilidade das várias implemntações da aplicação *kmeans*.

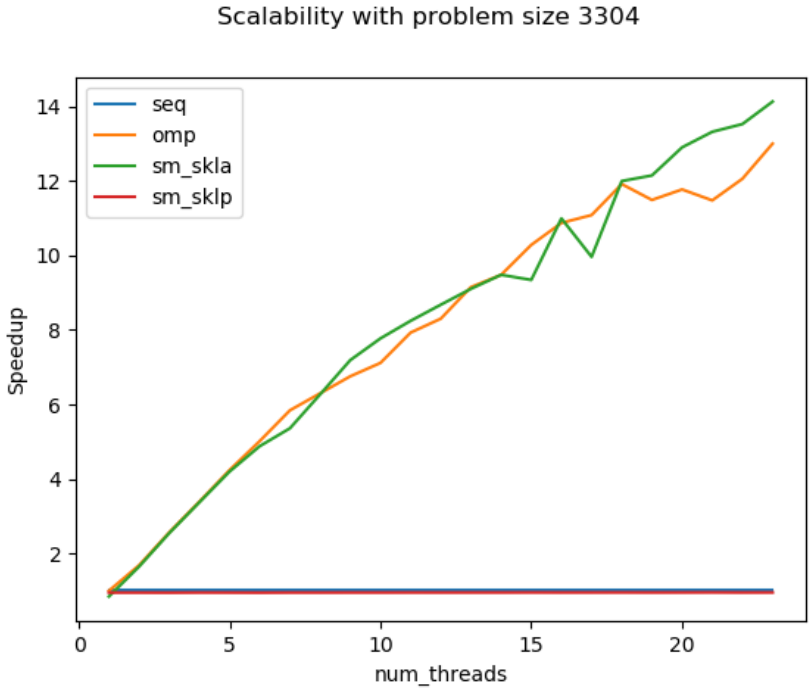


Figura 26: Escalabilidade das várias implementações da aplicação *matrix multiplication*.