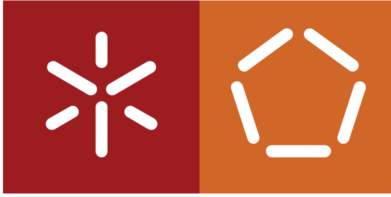
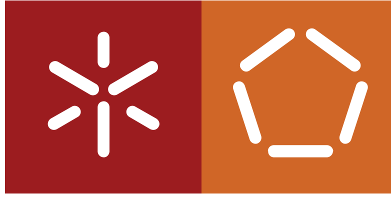


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tiago Fernandes Gonçalves

CoR-HPX
Uma nova abordagem à computação orientada ao recurso

Março 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tiago Fernandes Gonçalves

CoR-HPX
Uma nova abordagem à computação orientada ao recurso

Dissertação de Mestrado
Mestrado em Engenharia Informática

Dissertação orientada por
António Manuel Silva Pina

Março 2021

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

LICENÇA CONCEDIDA AOS UTILIZADORES DESTE TRABALHO:

CC BY

https://creativecommons.org/licenses/by/4.0/

AGRADECIMENTOS

Desejo exprimir os meus agradecimentos a todos aqueles que, de alguma forma, permitiram que esta dissertação de mestrado se concretizasse.

Em primeiro lugar, deixo o meu grande agradecimento ao professor António Pina pela sua orientação, interesse, pela sua visão crítica e oportuna, pelo rigor permanente e pela disponibilidade que sempre demonstrou. Agradeço ainda os valores e conselhos que me transmitiu, a seriedade e frontalidade em todas as conversas, a consideração, as palavras de incentivo e o contínuo estímulo intelectual.

Um agradecimento ao LIP Minho pela oportunidade e pelo financiamento.

Por fim, um agradecimento especial à minha família pelo suporte e encorajamento.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico.

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 conducentes à sua elaboração.

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

ABSTRACT

In the last few years, an emerging type of development and execution systems, based on asynchronous many-task programming (AMT), has shown enormous potential to address the scalability and efficiency needs of modern scientific parallel applications.

Within this scenario, HPX emerges as an alternative to the predominant programming models, representing a combination of ideas such as data flow, futures and CPS (Continuation-passing style), components and actions, through a uniform API with strict compliance to the modern C++ standard, with support for shared memory and distributed memory and accelerators.

Resource Oriented Computing (CoR) is a computing model that introduces the metaphor of the resource as a computational entity that incorporates the notion of state, concurrency, locality and distribution, through its common characteristics, evidenced in the triad: i) unique identification in the distributed system ii) coherence among replicas and iii) an instance that is the concrete realization of the resource in the host system.

This dissertation introduces the CoR-HPX platform as a new approach to CoR, starting from the establishment of the correspondence between the resource entity and its modeling as an HPX component. It is based on this assumption that we proceed to the design and development of CoR-HPX to address the constraints of the previous PlaCoR platform, namely: (i) execution - adopting the task-centric model instead of the thread-centric model, (ii) replication - adopting a centralized single copy policy instead of distributed replicas, compatible with the possibility of resource migration between domains, (iii) group communication - replacing the *spread* library with HPX's own synchronization mechanisms, (iv) point-to-point communication - using channels, and (v) API - extending the previous API in order to introduce asynchronous programming and futurization, based on the AMT programming model support.

The validation of CoR-HPX was made, in the first place, through the compiling and execution of application examples inherited from the previous platform, having verified that the results obtained faithfully reproduced the ones obtained from PlaCoR. The functionality of new resource classes such as Operon, UniChannel and MultiChannel was also verified. The first one enhances the multitasking execution of functions, according to the fork-join model, while the other two classes use the HPX channels as alternative way to send/receive messages between tasks, supporting simple bidirectional communication modes and predefined topologies.

We can say that CoR-HPX brings a higher level of abstraction to HPX programming, by using the different classes of resources as predefined modules that can be easily adapted to any application. For example, the Data resource includes the necessary mechanisms to transform instances of any C++ class, with minimal programmer involvement, into remotely addressable and migrable objects, almost completely hiding the HPX boilerplate code. Thus, the given resource is both a conventional component type (HPX) with state and an object that can be used to execute functions with any signatures, the equivalent of a simple action (HPX).

KEYWORDS Parallel Computing, Distributed Computing, HPX, Resource Oriented Computing, Asynchronous Many-Tasks

RESUMO

Nos últimos anos, uma classe emergente de sistemas de desenvolvimento e execução, baseados em programação assíncrona de muitas tarefas (AMT), tem vindo a mostrar um enorme potencial para responder às necessidades de escalabilidade e eficiência de aplicações paralelas científicas modernas.

É neste quadro que surge o HPX, como uma alternativa aos modelos de programação predominantes, que representa uma combinação de ideias tais como: fluxo de dados, futuros e CPS (*Continuation-passing style*), componentes e ações, através de uma API uniforme com aderência estrita ao padrão C++ moderno, com suporte para memória partilhada, memória distribuída e aceleradores.

A Computação orientada ao Recurso (CoR) é um modelo de computação que introduz a metáfora do Recurso como uma entidade computacional que incorpora a noção de estado, de concorrência, de localidade e de distribuição, através das suas características comuns, evidenciadas na tríade: i) identificação única, no sistema distribuído ii) coerência entre réplicas e iii) instância - realização concreta do Recurso no sistema hospedeiro.

Esta dissertação apresenta a plataforma CoR-HPX como uma nova abordagem ao CoR, a partir do estabelecimento da correspondência entre a entidade Recurso e a sua modelação como um *componente* HPX. É a partir desta base que se procede ao desenho e realização do CoR-HPX para responder às limitações da plataforma anterior PlaCoR, nomeadamente, no que concerne: a i) execução – adotando o modelo *task-centric* em detrimento do modelo *thread-centric*, ii) replicação – adotando uma política de cópia única centralizada em vez de réplicas distribuídas, compatível com a possibilidade de migração de Recursos entre domínios, iii) comunicação em grupo – substituindo a biblioteca *spread* por mecanismos de sincronização próprios do HPX, iv) comunicação ponto-aponto - usando canais e v) API – estendendo a API anterior de forma a introduzir programação assíncrona e futurização, baseadas no suporte na modelo de programação AMT.

A validação da CoR-HPX fez-se, em primeiro lugar, através da compilação e execução de exemplos de aplicações herdados da plataforma anterior, tendo-se verificado que os resultados obtidos reproduziam fidedignamente os obtidos do PlaCoR. Foi igualmente comprovada a funcionalidade de novas classes de Recursos como é o caso do Operon, do UniChannel e do MultiChannel. A primeira, potencia a execução multitarefa de funções, segundo o modelo *fork-join*, enquanto as outras duas classes usam os canais do HPX, como meios alternativos para o envio/receção de mensagem entre tarefas, dando suporte a modos de comunicação bidirecionais simples e em topologias pré-definidas.

Pode também afirmar-se que a CoR-HPX vem acrescentar à programação com HPX um nível superior de abstração, através da utilização das diferentes classes de Recursos como módulos pré-definidos facilmente adaptáveis ao código específico de cada nova aplicação. Tal é o caso do Recurso Dado, que inclui os mecanismos necessários para, com o envolvimento mínimo do programador, transformar as instâncias de qualquer classe C++, em objetos remotamente endereçáveis e migráveis, ocultando quase completamente o código *boilerplate* do HPX. Assim, o Recurso Dado é ao mesmo tempo um tipo de componente (HPX) convencional com

estado e um objeto que pode ser usado para executar funções com qualquer assinatura, o equivalente a uma ação simples (HPX).

PALAVRAS-CHAVE Computação Paralela, Computação Distribuída, HPX, Computação Orientada ao Recurso, Muitas-Tarefas Assíncronas

CONTEÚDO

Conteúdo iii

1 INTRODUÇÃO 3
1.1 Contextualização 3
1.2 Motivação 4
1.3 Objetivos 4
1.4 Estrutura do documento 5

2 ESTADO DA ARTE 6
2.1 Tarefas 6
2.2 Ambientes de programação baseados em tarefas 7
2.2.1 HPX vs OpenMP/MPI 7
2.2.2 Evolução dos modelos 9
2.2.3 Taxonomia 9
2.3 Paralelismo e concorrência em C++ 9
2.4 HPX - High Performance ParalleX 10
2.4.1 Gestão e escalonamento dos fios de execução 10
2.4.2 AGAS 11
2.4.3 Camada de transporte de parcelas 13
2.4.4 Objetos de controlo local (LCOs) 14
2.4.5 Sistema de monitorização de desempenho 15
2.5 O estudo de caso CoR 15
2.5.1 CoR - computação orientada ao recurso 15

3 APLICAÇÕES DISTRIBUÍDAS COM HPX 20
3.1 Declarar ações 20
3.1.1 Ações simples 20
3.1.2 Ações de componente 21
3.2 Executar ações 21
3.2.1 Assíncrono sem sincronização. 22
3.2.2 Assíncrono com sincronização 22
3.2.3 Síncrono 23
3.2.4 Assíncrono sem sincronização, com continuação 23

3.2.5 Assíncrono com sincronização, com continuação 23
3.3 Componentes 25
3.3.1 Criar um componente 25
3.3.2 Criar um componente template 29
3.3.3 Criar um componente template migrável 32
3.3.4 Criar um componente template migrável com herança 34
3.4 Canais 36
3.5 Serialização de objetos 37
3.6 Inicializar e executar o runtime HPX 39
3.7 Executar programas 41
4 PLATAFORMA PARA COMPUTAÇÃO ORIENTADA AO RECURSO 44
4.1 Antecedentes 44
4.2 Introdução ao CoR-HPX 45
4.3 Ferramentas de desenvolvimento 46
4.4 Arquitetura do Sistema 47
4.4.1 Classes dos Recursos 48
4.5 Recursos 50
4.5.1 Group 50
4.5.2 Domain 50
4.5.3 Closure 51
4.5.4 ProtoAgent 51
4.5.5 Agent 51
4.5.6 Barrier 51
4.5.7 Mutex 52
4.5.8 RWMutex 52
4.5.9 Data 52
4.5.10 Operon 53
4.5.11 UniChannel 54
4.5.12 MultiChannel 54
4.6 Elementos 54
4.6.1 Contentor 54
4.6.2 Organizador 56
4.6.3 Executor 58
4.6.4 Caixa-Postal 58
4.6.5 Classe auxiliar Mensagem 59
4.6.6 Valor 60

4.6.7 Sincronizador 60
4.6.8 ExecutorPool 61
4.6.9 SUniChannel 62
4.6.10 SMultiChannel 62

5 CODIFICAÇÃO E EXECUÇÃO DE PROGRAMAS 64
5.1 Inicialização de um programa 64
5.2 Estrutura de um programa 65
5.3 Troca de mensagens entre domínios distribuídos 65
5.4 Executar funções globais remotamente 67
5.5 Caso de uso Queue utilizando o Recurso Dado 67
5.5.1 Definição de Queue e Container 68
5.5.2 Definição das funções objeto 68
5.5.3 Programa 69
5.5.4 Observações 70
5.6 Caso de uso Stencil 72
5.6.1 Implementação em CoR-HPX 72
5.6.2 Observações 75

6 DISCUSSÃO E CONCLUSÕES 76
6.1 Discussão 76
6.2 Conclusões 78
6.3 Trabalho Futuro 79

A INSTALAÇÃO DO SISTEMA COR-HPX 85
a.1 Requisitos e download 85
a.2 Configuração do ambiente de execução 85
a.3 Execução de aplicações 86

B CÓDIGOS COMPLEMENTARES 87
b.1 Criar um componente template - extensão 87
b.1.1 Componente servidor 87
b.1.2 Componente cliente 88
b.2 Criar um componente template migrável - extensão 89
b.2.1 Componente servidor 89
b.3 Criar um componente template migrável com herança - extensão 90
b.3.1 Componente servidor - classe base Container 90

b.3.2 Componente servidor - classe derivada Queue 92
b.3.3 Componente cliente 94
b.4 Diagrama dos Recursos - Versão Ampliada 94
C API DOS ELEMENTOS QUE CONSTITUEM OS RECURSOS DE COR-HPX 96
c.1 Contentor 96
c.2 Organizador 98
c.2.1 Organizador Dinâmico 98
c.2.2 Métodos comuns 99
c.3 Executor 99
c.4 Caixa-Postal 100
c.5 Mensagem 101
c.6 Valor 101
c.7 Sincronizador 102
c.7.1 Barreira 102
c.7.2 Guarda 102
c.7.3 Guarda de leitura/escrita 102
c.8 ExecutorPool 103
c.9 SUniChannel 104
c.10 SMultiChannel 104
D API DO COR-HPX 105
d.1 Funções Globais 105
d.2 Domain 105
d.3 Group 108
d.4 Closure 109
d.5 ProtoAgent 109
d.6 Agent 110
d.7 Operon 111
d.8 Data 113
d.9 Barrier 114
d.10 Mutex 114
d.11 UniChannel 114
d.12 MultiChannel 115

LISTA DE FIGURAS

Figura 1 Arquitectura do HPX 11
Figura 2 Gestão de referências do AGAS 12
Figura 3 Processo de execução remota através de parcelas 13
Figura 4 Exemplo de execução de uma operação remota 14
Figura 5 Anatomia do Recurso CoR 16
Figura 6 Representação centralizada. Fonte:Moreira (2001) 18
Figura 7 Representação distribuída. Fonte:Moreira (2001) 18
Figura 8 Árvore hierárquica de dependências de uma aplicação CoR 19
Figura 9 Ilustração da migração de um componente 34
Figura 10 Ferramentas de desenvolvimento do protótipo PlaCoR 44
Figura 11 Ferramentas de desenvolvimento do protótipo CoR-HPX 46
Figura 12 Arquitetura da Plataforma CoR-HPX 47
Figura 13 Diagrama de classes dos Recursos 49
Figura 14 Diagrama de classes dos Recursos 95

LIST OF LISTINGS

1 plain_action.cpp . . . 21
2 plain_action_two_macros.cpp . . . 21
3 async_without_synchronization.cpp . . . 22
4 async_with_synchronization.cpp . . . 22
5 sync.cpp . . . 23
6 async_with_synchronization_with_continuation_I.cpp . . . 24
7 async_with_synchronization_with_continuation_II.cpp . . . 24
8 async_with_synchronization_with_continuation_III.cpp . . . 24
9 queue.hpp - Ficheiro cabeçalho de declaração do componente . . . 26
10 queue.cpp - Ficheiro de implementação do componente . . . 27
11 queue_client.hpp - Ficheiro cabeçalho de declaração do componente cliente . . . 28
12 program_hpx_queue.cpp . . . 29
13 queue.hpp - Ficheiro cabeçalho de declaração do componente template . . . 30
14 queue_client.hpp - Ficheiro cabeçalho de declaração do componente cliente templated . . . 31
15 program_hpx_queue_template.cpp . . . 31
16 queue.hpp - Ficheiro cabeçalho de declaração do componente template migrável . . . 32
17 queue_client.hpp - Ficheiro cabeçalho de declaração do componente cliente template migrável 33
18 program_hpx_queue_template_migrable.cpp . . . 33
19 program_hpx_queue_template_migrable_derived.cpp - Ficheiro de um exemplo de um programa para utilização do componente Queue com herança de Container . . . 35
20 local_channel.cpp . . . 36
21 local_channel_two_threads.cpp . . . 36
22 distributed_channel.cpp . . . 37
23 object_serialization.cpp . . . 38
24 start_runtime_implicit.cpp . . . 39
25 start_runtime_explicit.cpp . . . 40
26 start_runtime_explicit_async.cpp . . . 40
27 start_runtime_suspend_resume.cpp . . . 41
28 Execução de três programas em consolas separadas, no mesmo runtime . . . 42
29 Módulo de arranque de uma aplicação básica . . . 65
30 Exemplo send_an_object.cpp . . . 66
31 Exemplo create_remote.cpp . . . 68
32 queue.hpp . . . 69

33 queue_interface.hpp - Ficheiro cabeçalho de definição das funções objeto 70
34 queue_program.cpp - Ficheiro de um exemplo de um programa para utilização do recurso Data<Queue<Object» 71
35 Criação e execução do recurso Operon 73
36 Criação do recurso MultiChannel 74
37 Ciclo externo da simulação 74
38 Troca da linha de fronteira de cima com o vizinho superior 75
39 Processamento interior da matriz 75

1

INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Em computação de alto desempenho (HPC), os programadores de aplicações que usam métodos convencionais de programação paralela continuam a encontrar limitações e a enfrentar a dura realidade da falta de escalabilidade dos seus programas. Carecem de abordagens e facilidades de programação capazes de lidar com cargas de trabalho dinamicamente variáveis, que aumentem a escalabilidade e que tornem mais eficiente a utilização dos recursos do sistema.

Os padrões de programação paralela com MPI e OpenMP são amplamente utilizados e já bem estudados. Contudo, em algoritmos com balanceamento de carga dinâmico, com estruturas de dados não lineares e em sistemas Exascale, esses padrões enfrentam limitações Gropp (2009). Ao longo destes últimos anos têm-se feito esforços para tentar mitigar estes problemas, o que tem levado a um aparecimento crescente de modelos de programação baseados em tarefas, conhecidos também por Asynchronous Many-Task (AMT) Paul et al. (2019).

É neste contexto que surge o projeto HPX Kaiser et al. (2020), como uma biblioteca e um *runtime* que implementa o modelo de execução ParalleX, que visa ultrapassar as limitações dos modelos atuais de paralelismo, aliviando a contenção, latência, sobrecarga, inanição e consumo de energia. Criado por um grupo internacional de colaboração (*The Stellar Group* [1]), o seu código de desenvolvimento é de livre acesso e totalmente compatível com o padrão C++ moderno.

Tendo surgido com foco no desenvolvimento de aplicações paralelas e distribuídas de alto desempenho, o HPX fornece um novo modelo de programação para sistemas convencionais com uma arquitetura de sistema em tempo de execução bastante modular e bem concebida. As principais características desta biblioteca são:

- Possuir uma API uniforme, orientada ao padrão C++ moderno e compatível com a programação de aplicações paralelas em memória partilhada e memória distribuída;
- Possibilitar a escrita de código totalmente assíncrono e a execução de centenas de milhões de tarefas;
- Possuir uma semântica unificada com uma sintaxe homogénea para operações locais e remotas;
- Tornar manejável a concorrência através da sincronização baseada em *futures* e *dataflows*;
- Ter sido concebida para aplicações de qualquer escala;
- Ser a primeira implementação totalmente funcional do modelo de execução ParalleX;

[1] https://stellar-group.org/

- Ter sido publicada sob uma licença livre de código aberto e ter uma comunidade de desenvolvedores ativa e em crescimento.

1.2 MOTIVAÇÃO

Atualmente, com o volume de dados e a complexidade de programas a aumentar, para além das preocupações relativas à poupança de energia, tem havido um esforço no desenvolvimento e melhoramento de ambientes de programação paralela, de forma a extrair o máximo de desempenho dos sistemas computacionais. É neste âmbito que o HPX está a ganhar o seu espaço entre os programadores HPC, apesar de ser relativamente recente e carecer de uma mais ampla divulgação.

Numa outra dimensão, o CoR - Computação orientada ao Recurso Ribeiro and Pina (2019), surge como um paradigma de computação que recorre ao conceito de Recurso para explorar e extrair paralelismo em programas com memória partilhada e distribuída. Um Recurso pode ser encarado como uma abstração lógica ou física, como por exemplo processos, fios de execução, interfaces de comunicação, nós de computação ou ainda, grupos de outros Recursos. A abordagem procura atenuar as fronteiras que separam os componentes físicos de hardware dos componentes lógicos de software, combinando um conjunto de mecanismos de estruturação com os princípios da programação com múltiplos fios de execução, da memória partilhada/distribuída e da comunicação por passagem de mensagens.

Ao longo dos anos surgiram várias realizações práticas do modelo CoR, sendo a mais recente a PlaCoR Ribeiro (2019), desenvolvida em C++ Moderno, que recorre e integra um conjunto de bibliotecas externas para assegurar a funcionalidade esperada, como a computação em memória distribuída.

É da combinação das características do modelo CoR e das facilidades e conceitos do ambiente HPX que surgiu a motivação para o desenvolvimento do CoR-HPX, uma nova plataforma para programação concorrente/-paralela em ambiente de memória partilhada e distribuída.

1.3 OBJETIVOS

O objetivo principal desta dissertação é providenciar um ambiente de programação e de execução paralela para maximizar a utilização dos Recursos de um sistema computacional híbrido de grande escala. O processo de desenho e construção da nova plataforma inclui, ainda:

- o estudo exaustivo dos modelos e ferramentas associados ao HPX e do modelo ParalleX subjacente, em particular, dos que estão associados à extensão proposta de compatibilização do C++ moderno com a programação paralela distribuída;
- a revisão de alguns dos pressupostos do modelo de execução do CoR, com eventual reformulação dos conceitos na origem, nomeadamente no que diz respeito ao modelo de representação dos Recursos e à própria sintaxe da API;
- a inclusão dos mecanismos de programação assíncrona com futurização propostos pelo HPX;

- o estudo das vantagens da utilização do conceito Recurso no desenho de componentes HPX e da forma de executar funções remotas através de ações;
- a avaliação o poder expressivo do paradigma da orientação ao Recurso na escrita de programas concorrentes/paralelos/distribuídos.

Em termos gerais, pretende-se usar o HPX na construção de uma plataforma de programação, denominada CoR-HPX, para criar uma ambiente autónomo desenvolvido em C++ moderno, multi-plataforma - Linux e MacOS - e com facilidades para: comunicação inter-domínios, passagem de mensagem entre Recursos comunicantes, memória partilhada e distribuída, ativação remota de agentes, criação e gestão de Recursos, capacidade de consistência entre todas as réplicas de um Recurso e assincronismo.

Finalmente, como forma de validação e avaliação do trabalho realizado, realizou-se o desenho, programação e execução de exemplos paradigmáticos, em particular, de um algoritmo do tipo *stencil*, num ambiente de domínios distribuídos. Sendo que este trabalho é de código aberto, o código da plataforma e dos exemplos pode ser encontrado em https://github.com/tiagofgon/CoR-HPX.

1.4 ESTRUTURA DO DOCUMENTO

Este documento é constituído por 6 capítulos:

- Capítulo 1 - Introdução, motivação e objetivos desta dissertação.
- Capítulo 2 - Estado da arte das áreas que irão ser abordadas neste trabalho: primeiramente os ambientes de programação baseados em tarefas, seguindo-se o HPX e o seu modelo de execução e, por fim, o modelo CoR.
- Capítulo 3 - Principais funcionalidades de programação que o HPX oferece numa perspetiva prática - criar e executar ações e arranque do *runtime*;
- Capítulo 4 - Protótipo CoR-HPX, pondo a ênfase na arquitetura do sistema, na descrição dos Recursos e na interface de programação.
- Capítulo 5 - Metodologias de programação em CoR-HPX e exemplos de programas.
- Capítulo 6 - Discussão sobre o trabalho realizado, conclusão e perspetivas de trabalho futuro.

O documento contém ainda 4 anexos, onde é incluída informação complementar ao trabalho desenvolvido:

- Anexo A - Informações sobre como instalar e configurar o CoR-HPX.
- Anexo B - Códigos complementares, que estendem alguns códigos mostrados ao longo do documento.
- Anexo C - API dos elementos que constituem os Recursos.
- Anexo D - API dos Recursos, que constituem a interface de programação do CoR-HPX.

2

ESTADO DA ARTE

Com os modelos de programação paralela baseados nas normas MPI e OpenMP, já muito bem estudados e amplamente utilizados, obtêm-se bons algoritmos quando os dados são estruturados e o balanceamento de carga está bem definido. Mas quando as estruturas de dados são irregulares e a carga não é previsível, estes modelos revelam as suas limitações.

Por sua vez, o modelo de execução CSP (Communicating Sequential Processes) tem-se defrontado com a complexidade crescente dos CPUs, sockets multi-core e estruturas heterogéneas das GPUs. Tanto a eficiência como a escalabilidade para algumas aplicações e futuras aplicações *Exascale* exigem novas técnicas para expor novas fontes de paralelismo dos algoritmos e explorar recursos não utilizados através do uso adequado dos dados em tempo de execução.

Nos últimos anos tem-se assistido a vários esforços para mitigar estes problemas com o consequente aparecimento de novos modelos de programação, em particular o AMT - (Asynchronous Many-Task), que pretende responder com elevado desempenho aos desafios que surgirão na próxima geração de sistemas *Exascale*.

Sendo o HPX o foco desta dissertação, um sistema de *runtime* que suporta AMT, é de toda a conveniência fazer o estado da arte de sistemas que prosseguem o mesmo paradigma e, em simultâneo, fazer o enquadramento com os mais recentes e futuros desenvolvimentos do padrão C++ moderno. Complementarmente, apresenta-se o modelo CoR (e o seu percurso de desenvolvimento até agora), que irá servir de suporte para a construção da arquitetura dos Recursos em CoR-HPX.

2.1 TAREFAS

Uma tarefa (Thoman et al., 2018) é uma sequência de instruções dentro de um programa, que podem ser processadas em simultâneo. Pode ser encarada como uma chamada de função, incluindo o nome da função a ser invocada, o conjunto de argumentos para a invocar, e possivelmente meta-dados. A execução simultânea de tarefas pode ser condicionada por dependências de controlo e fluxo de dados, originando um grafo. Linguagens com este paradigma denominam-se *task-centric* (Alessandrini, 2015).

Uma outra abordagem, denominada *thread-centric*, usa o fio de execução como unidade de execução, aumentando a granularidade de concorrência/paralelismo. Apesar de continuar a desempenhar um papel importante e, em certos casos, ter bons resultados de escalabilidade, a sua utilização é mais problemática, em termos de balanceamento de carga, grão de paralelismo e sincronização.

2.2 AMBIENTES DE PROGRAMAÇÃO BASEADOS EM TAREFAS

Uma das plataformas pioneiras de programação eficiente de tarefas leves é a linguagem Cilk (Blumofe et al., 1996a) que desde o seu início (anos 90 do século XX) suporta o *work stealing* entre tarefas.

Para além das linguagens e extensões, surgiram também bibliotecas paralelas com base em tarefas, que se tornaram padrão na indústria, tais como Intel Cilk Plus (Blumofe et al., 1996b) ou Intel TBB (Voss et al., 2019). Existem também *runtimes* especificamente concebidos para melhorar o desempenho em memória partilhada dos *frameworks* existentes, tais como Qthreads (Wheeler et al., 2008) ou ainda Argobots (Seo et al., 2017). Ambientes baseados em tarefas para hardware heterogéneo também se desenvolveram com o surgimento de aceleradores de computação GPU como, por exemplo, o StarPU (Augonnet et al., 2011).

Ps modelos de programação paralela baseados em tarefas podem-se dividir em três categorias (Kaiser et al., 2014):

- Soluções de bibliotecas: bibliotecas como Intel TBB, Microsoft PPL (Campbell et al., 2010), Qthreads, StarPU, Argobots, o HPX, entre outras;
- Extensões de linguagem: exemplos como Intel Cilk Plus e OpenMP 5.0;
- Linguagens de programação experimentais: por exemplo, Chapel (Chamberlain et al., 2007) , Intel ISPC (int, 2020) , ou X10 (Charles et al.).

Todas estas soluções convergem na utilização de um modelo de programação AMT, sendo que a maior parte utiliza o estilo *continuation-passing style* (CPS). O HPX distingue-se porque ao invés de criar uma nova sintaxe ou semântica, implementou as definidas no C++11, o que se traduziu numa API homogénea, que se baseia numa interface de programação já amplamente aceite e facilmente usada por programadores com experiência prévia em C++.

Os *frameworks* AMT são cada vez mais utilizados em sistemas de memória distribuída, que constituem o alvo mais importante para o HPC. Neste contexto, o processamento de tarefas é frequentemente combinado com um modelo de programação de espaço de endereçamento global (GAS) e com programação de múltiplos processos que em conjunto formam a execução distribuída de um único programa baseado em tarefas. Embora alguns exemplos de ambientes de espaço de endereçamento global com paralelismo baseado em tarefas sejam linguagens especificamente concebidas, tais como Chapel e X10, é também possível implementar estes conceitos sob a forma de bibliotecas, tais como o HPX e o Charm++ (Kale and Krishnan, 1993) - *runtimes* assíncronos baseados em GAS, mais concretamente, PGAS (Yelick et al., 2007).

Muitos dos ambientes de programação baseados em tarefas são mantidos por uma comunidade dedicada de programadores e são frequentemente orientados para a investigação. Como tal, pode haver relativamente pouca documentação disponível sobre as suas características e funcionamento interno. Thoman et al. (2018)

2.2.1 *HPX vs OpenMP/MPI*

Apesar do HPX ser uma biblioteca relativamente recente e ainda pouco utilizada, na literatura surge como tendo muitas vantagens e resultados positivos.

Em (Khatami et al., 2016b) foi usada uma biblioteca, chamada OP2, que fornece uma *framework* para a execução paralela de aplicações em grelhas não estruturadas em diferentes arquiteturas de hardware *multi-core/many-core*. Essa biblioteca usa o OpenMP para gerar código para paralelizar ciclos. Com o objetivo de substituir o OpenMP pelo HPX foram realizados testes comparativos entre as duas versões numa aplicação chamada "Airfoil", num único nó com 32 threads. A versão HPX obteve um ganho de escalabilidade de 21% em relação à versão OpenMP. Tais resultados com HPX justificam-se com a redução da inanição, devido ao controlo de granularidade e da capacidade de interligar automaticamente ciclos consecutivos, ambos em tempo de execução.

Em (Bremer et al., 2019) comparam-se as diferenças de performance entre implementações em HPX e MPI de um código do método *discontinuous Galerkin* de elementos finitos para equações bidimensionais de águas rasas *(two-dimensional shallow water equations)*. Os resultados demonstraram que, com tarefas com uma granularidade adequada, a versão HPX supera o MPI um fator de aproximadamente 1,2 em 128 nós de computação.

Uma das vantagens do HPX é ter sincronização assíncrona baseada em *futures* e *dataflows*, que torna possível a remoção da sincronização com barreiras, levando a um ganho de desempenho. A granularidade também é um aspeto diferenciador. Com efeito, tarefas de grão fino têm menos quantidade de trabalho, o que leva a um melhor balanceamento de carga. Estas foram as razões que levaram à reimplementação do algoritmo N-Body Khatami et al. (2016a) com o HPX. Comparando com o uso do OpenMP, num só nó de computação, obtiveram-se ganhos por volta de 45%, e comparando com a versão hibrida OpenMP + MPI, em ambiente distribuído, os ganhos foram de 28%.

O HPX também demonstra bons resultados de eficiência com o algoritmo Octo-Tiger (Pfander et al., 2018), um código astrofísico que simula a evolução dos sistemas estrelares com base no método *fast multipole method* usando *octrees* adaptados como estrutura central de dados.

Além das características de programação, o HPX incorpora ainda outras capacidades de ajuda ao desenvolvimento, tais como contadores de desempenho e serialização.

Contadores de Desempenho

Em (Grubel, 2016) e (Grubel et al., 2016) foram usados contadores de desempenho, que dão uma visão detalhada sobre o comportamento das aplicações e a utilização dos Recursos em tempo de execução, assim como o impacto da granularidade das tarefas do HPX no desempenho.

Serialização

Em (Biddiscombe et al., 2017) foi demonstrado que a camada de serialização do HPX é capaz de competir com outras bibliotecas similares, pelo que, foi expandida para suportar funcionalidades de RMA e, desta forma, poder realizar chamadas RPC de cópia zero em aplicações distribuídas baseadas em tarefas HPX.

2.2.2 *Evolução dos modelos*

A necessidade de mudança de paradigma é evidenciada pela evolução recente das normas do MPI e do OpenMP. Na versão 3.0, o MPI passou a adotar novas características como RDMA e operações de memória partilhada. Pelo seu lado, o OpenMP, conhecido principalmente pelo modelo *fork/join*, integra *tasks* na API desde a versão 3.0 (ano de 2008). Esta linha de orientação às tarefas tem continuado com o desenvolvimento no OpenMP.

2.2.3 *Taxonomia*

Em (Thoman et al., 2018) é feito um estudo das taxonomias dos seguintes ambientes de programação baseados em tarefas: C++ STL, TBB, HPX, Legion (Bauer, 2014), PaRSEC (Bosilca et al., 2013), OpenMP, Charm++, OmpSs (Duran et al., 2011), AllScale (Jordan et al., 2018), StarPU, Cilk Plus, Chapel e X10. Comparando as respetivas APIs e *runtimes*, constata-se que os que suportam memória distribuída implícita também contemplam, de alguma forma, a heterogeneidade, geralmente possuem um espaço de endereçamento global e implementam particionamento de tarefas. *work stealing* também é uma funcionalidade comum intra-nó, com Legion e X10 a utilizá-la em ambiente inter-nó.

Outras bibliotecas como o TBB, StarPU e OpenMP, são também soluções em C++ orientadas à programação baseada em tarefas e que também tentam extrair o máximo de paralelismo, mas exclusivamente em memória partilhada.

2.3 PARALELISMO E CONCORRÊNCIA EM C++

A maior parte das bibliotecas que se baseiam em tarefas usam o C++ que está entre as dez linguagens de programação mais populares TIOBE (2020). O C++ moderno combina a eficiência herdada da linguagem C com novas características multi-paradigma tais como: a orientação ao objeto, a programação genérica e a funcional.

Em termos de concorrência/paralelismo, desde a norma C++11 que se inclui uma API multithreading e um novo modelo de memória, com suporte a concorrência (threads, locks, variáveis de condição, futures, ...) e operações atómicas, sobre tipos de dados atómicos. Com o C++14 e o C++17 estabeleceram-se padrões mais elevados de abstração e a introduziram-se algoritmos paralelos na Standard Template Library (STL).

Mais recentemente, o padrão C++20 introduziu novas funcionalidades relacionadas com a concorrência/paralelismo. Em particular, std::jthread, Smart Pointers atómicos, Latches e Barreiras, Semáforos e Co-rotinas. Está previsto para o C++23: Executores, extensão de Futures, Memória transacional, Task Blocks e Data-Parallel Vector Library. Podemos afirmar que a equipa de desenvolvimento do C++ tem feito um esforço contínuo para reforçar o suporte à programação paralela/concorrente, sendo uma das suas prioridades.

2.4 HPX - HIGH PERFORMANCE PARALLEX

O HPX, que adotou o modelo de execução e o paradigma de programação ParalleX (Gao et al., 2007; Kaiser et al., 2009), é uma biblioteca e um runtime totalmente alinhado com o padrão do C++ moderno, que integra computação em memória partilhada (num só nó) com computação distribuída.

O ParalleX é um modelo de computação paralela assíncrono com espaço de endereçamento global particionado, com as facilidades de programação necessárias para dar uma resposta inovadora a muitos dos desafios críticos atuais de programação paralela, conhecidos como "SLOW" *(Starvation, Latency, Overheads, Waiting for contention)*, através da manipulação direta e eficiente dos dados, objetos e um controlo dinâmico de recursos.

Explora uma metodologia de computação distribuída com operações *multithreaded* assíncronas que tira partido da sobreposição do cálculo e da comunicação de forma a diminuir a latência. Assim, deixa de haver distinção entre regiões paralelas e regiões sequenciais potenciando a execução de todo o código de forma paralela.

As tarefas em execução num programa têm individualmente a sua própria árvore de dependências cuja dimensão aumenta com a diminuição do grão de paralelismo.

O encadeamento de *futures* através de primitivas leves de sincronização é usado para resolver restrições e dependências, evitando, desta forma, a utilização de barreiras globais que estão habitualmente associadas ao desperdício de tempo de CPU. Este modelo de programação resulta numa cadeia de execução representada por um diagrama DAG, em explícita oposição ao modelo *fork-join*.

A arquitetura do HPX (ver figura 1) conjuga cinco módulos (Kaiser et al., 2014):

- Sistema de escalonamento de fios de execução;
- Espaço global ativo de endereçamento (AGAS);
- Camada de transporte de parcelas;
- Objetos de controlo local (LCOs);
- Sistema de monitorização de desempenho.

2.4.1 *Gestão e escalonamento dos fios de execução*

As *threads* HPX, chamadas usualmente de tarefas, são fios de execução do nível de utilizador que mantêm um estado num ambiente de execução com um conjunto específico de registos. No *kernel* existe uma *pool* de N fios de execução, geralmente um fio por cada núcleo do CPU, para executar M fios de execução do runtime HPX, normalmente M>N.

Os fios de execução HPX são escalonados de forma cooperativa pelo escalonador por cima de cada fio de execução do kernel. Desta forma, os fios de execução HPX podem ser escalonados sem uma transição para o kernel, o que proporciona um aumento de desempenho. Outra vantagem é que os fios de execução do kernel podem continuar a correr mesmo que um conjunto de fios de execução do HPX bloqueie por qualquer razão.

O sistema de escalonamento de fios de execução HPX implementa, atualmente, diversas políticas de escalonamento. De forma geral, em todas as políticas, as tarefas são escalonadas usando um modelo de execução

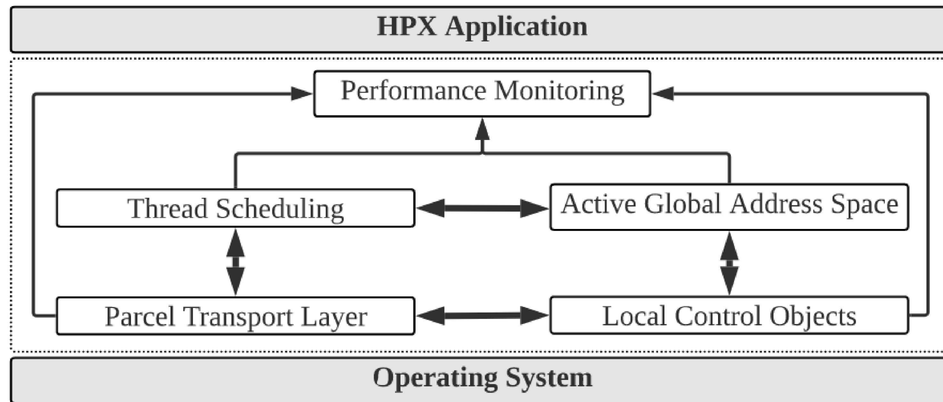


Figura 1: Arquitetura do HPX: estrutura modular do sistema em tempo de execução HPX que implementa as ferramentas que suportam todos os elementos necessários para o modelo ParalleX - AGAS, camada de transporte de parcelas, fios de execução do HPX e o seu sistema de escalonamento, contadores de desempenho, LCOs e a interligação destes componentes. Fonte:(Gupta et al., 2020)

baseado em filas de trabalho FIFO *(first-in-first-out)*, onde cada fio do sistema operativo opera a sua própria fila. As tarefas não são alocadas fixamente às filas FIFO, podendo ser transferidas entre fios através do mecanismo de *task stealing*.

2.4.2 *AGAS*

O objetivo dos sistemas de espaço global de endereçamento é tentar aumentar a produtividade e simplificar o ciclo de desenvolvimento de aplicações distribuídas, fornecendo uma camada de abstração omnipresente sobre os endereços de memória. Estes mecanismos são controlados pelo sistema operativo, garantindo que os endereços globais sejam válidos em cada nó de computação, num sistema de qualquer escala.

Sistemas como Charm++ e UPC++ (Bachan et al., 2019) também fornecem, em tempo de execução, um módulo que mapeia endereços globais para endereços virtuais para fornecer endereços globais verdadeiros. Contudo, apenas o HPX tem um sistema de endereços globais que permite que objetos de qualquer tipo sejam re-colocados (migrados) em tempo de execução.

Com o AGAS *(Active Global Address Space)* Gao et al. (2007) é possível fazer balanceamento de carga em tempo de execução, migrando os dados, e também a execução de mensagens ativas. As mensagens ativas reduzem significativamente a necessidade de movimentação de dados ao mover tarefas para onde os dados estão localizados em vez de mover os dados para onde o trabalho é executado. No entanto, o AGAS introduz alguma sobrecarga, pois precisa executar código para resolver e manter as referências, mas o impacto deste é desprezável (Amini and Kaiser, 2019).

A figura 2 mostra como o AGAS permite que objetos locais sejam acedidos a partir da existência de múltiplas referências em qualquer localidade, seja a local ou remotas. Para uma melhor compreensão do funcionamento do AGAS apresenta-se a definição de termos fulcrais:

GID: Todos os objectos globalmente endereçáveis, em HPX, são identificados por um GID - Global Identifier. Um GID é constituído por uma instância do tipo `hpx::id_type`, que serve para identificar objetos, a nível global, e localidades. É também possível associar o GID a uma cadeia arbitrária de caracteres, usada para registar e identificar objetos com o mesmo nome de base, mas com índices diferentes.

Localidade: Em HPX, os processos que fazem parte de um programa são chamados de localidades. Simbolizam um domínio síncrono de execução que, normalmente, representa um único nó de computação num cluster ou um domínio NUMA numa máquina SMP. Cada localidade é, por sua vez, identificada por um GID.

Ação: É um invólucro para uma função que viabiliza a sua execução remota. Ao encapsular funções, o HPX pode enviar pacotes de trabalho (parcelas) para diferentes unidades de processamento (localidades). Este mecanismo também é conhecido por RPC Soares (1992) e é uma das valiosas capacidades do HPX.

Componente: É um qualquer objeto C++, no espaço global de endereçamento, remotamente endereçável. Além do objeto, as suas funções membro também podem ser invocadas remotamente. É importante referir que um único objeto pode ser referenciado a partir de múltiplas localidades, ou seja, podem existir múltiplas instâncias do tipo `hpx::id_type` que contêm o mesmo GID. Todos os objetos remotamente endereçáveis possuem um GID único e mesmo que o objeto migre de localidade, as referências para ele permanecem válidas. Na figura 2, as estrelas a negro simbolizam componentes e as estrelas a cinzento simbolizam referências locais para esses componentes.

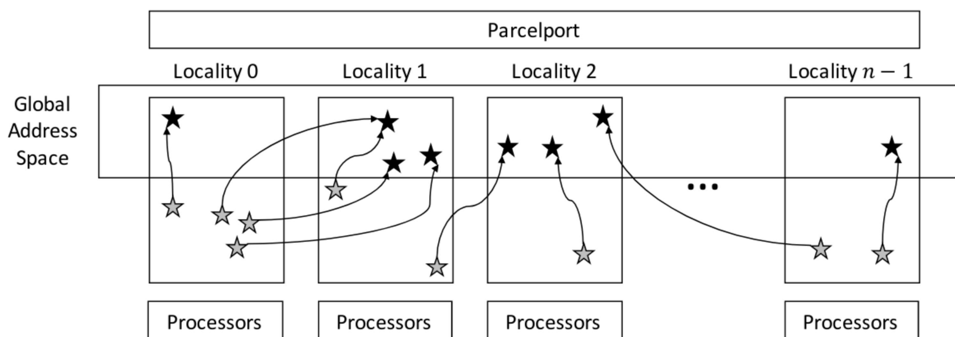


Figura 2: O AGAS fornece uma camada de abstração em cima de endereços virtuais locais para cada localidade. As estrelas negras representam objetos globais, e as estrelas cinzentas indicam as referências aos objetos globais. Cada referência está conectada ao objecto global com uma seta, para ilustração. Fonte: (Amini, 2020)

O mecanismo de propriedade partilhada em HPX (múltiplas referências para um mesmo objeto, potencialmente existentes em diferentes localidades), segue o padrão do C++ moderno no que diz respeito à gestão da sua vida útil. Assim, tal como acontece com os *smart pointers*, os objetos globais possuem uma contador de referências global, pelo que, quando um objeto saí do escopo, ou o contador chega a zero, o objeto é destruído.

A API do HPX contempla várias funções relacionadas com os endereços globais (GIDs) das localidades, usadas para obter:

- `hpx::find_here()`: o endereço global da localidade em que esta função é chamada;
- `hpx::find_all_localities()`: os GIDs de todas as localidades disponíveis para esta aplicação (incluindo a localidade na origem da chamada desta função);
- `hpx::find_remote_localities()`: os GIDs de todas as localidades remotas (não incluindo a localidade na origem da chamada desta função);
- `hpx::get_num_localities()`: o número de localidades disponíveis para esta aplicação;
- `hpx::get_locality_id()`: o id da localidade atual, valor entre [0, N-1];
- `hpx::find_locality()`: o endereço global de qualquer localidade que suporte um determinado tipo de componente;
- `hpx::get_colocation_id()`: o endereço global da localidade que atualmente hospeda o objeto com um determinado GID.

2.4.3 *Camada de transporte de parcelas*

Uma parcela é uma mensagem ativa que desencadeia uma operação após a sua receção/ativação numa determinada localidade (Von Eicken et al., 1992). Se a localidade for remota cabe ao AGAS serializar a própria função, denominada de "ação", incluindo os argumentos e a informações de estado, e encaminhá-la para a localidade alvo. Este processo, ilustrado na figura 3, abarca serialização e comunicação de dados transparente para o programador.

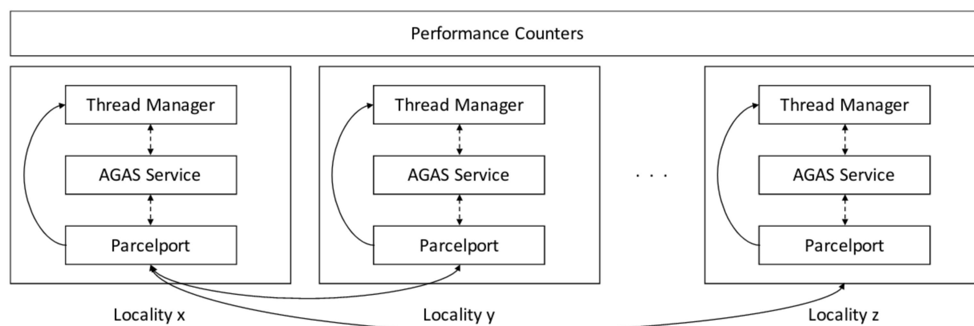


Figura 3: Quando um fio HPX acede a um objecto global (componente), o AGAS determina se o objecto pode ser acedido localmente. Se o objecto estiver numa localidade diferente, a tarefa HPX é serializada e entregue ao *parcelport* dessa localidade. O *parcelport* desserializa a tarefa, cria um fio HPX, e entrega-o ao gestor de fios de execução para execução na localidade de destino. Fonte: (Amini, 2020)

2.4.4 *Objetos de controlo local (LCOs)*

Um LCO (Objeto de Controlo Local) é um objeto que pode ser usado para criar, ativar, suspender ou reativar um fio de execução do HPX, assim como organizar o fluxo de execução e proteger recursos partilhados. O seu propósito é fornecer uma sincronização dinâmica e global, facilitar a migração de objetos e sua continuidade, bem como a gestão de recursos, permitindo assim o paralelismo com diversas granularidades.

Os LCOs, objetos de controlo local (ou controlo leve), são orientados ao evento e substituem as barreiras globais com sincronização baseada em restrições, através dos principais LCOs em HPX: *futures* e *dataflows*. Apesar disto, o termo é suficientemente abrangente para incluírem primitivas tradicionais de sincronização, como mutexes, semáforos, spin-locks e barreiras. É através do uso de LCOs que, em ambiente distribuído, o HPX é capaz de controlar a sincronização de tarefas entre localidades.

A imagem na figura 4 demonstra o conceito de continuação. Um futuro é criado na localidade 1 e é enviada uma parcela para a localidade 2 para iniciar a operação remota. Como o resultado da operação é enviado automaticamente para o future na localidade 1, este espera que tal aconteça através da função get(). O futuro encadeado com outros futuros, através de outros LCOs, cria uma árvore de fluxo de execução.

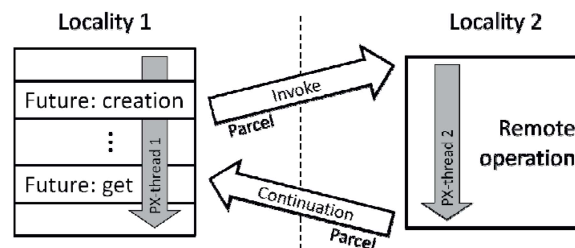


Figura 4: Exemplo de execução de uma operação remota. Fonte: (Kaiser et al., 2009)

Dentre a grande variedade de LCOs existentes, apresenta-se a seguir algumas das primitivas da API do HPX mais usadas:

- `hpx::async`: criar uma tarefa e retornar um *future*, recebendo como argumento uma função e os respetivos parâmetros de invocação.
- `get()`: para esperar e extrair o valor encapsulado por um *future*.
- `then()`: para encadear duas tarefas, de forma a que quando a primeira termina, o valor de retorno (*future*) é passado como argumento à tarefa seguinte que só é iniciada quando o *future* estiver pronto.
- `dataflow()`: associar uma função a uma lista de argumentos que são *futures*. A função só é efetivamente chamada quando todos os argumentos estiverem prontos. Consegue-se, desta forma, encadear fluxos de dados e construir uma árvore de execução do tipo DAG. À maneira de execução de `hpx::async`, este LCO retorna imediatamente um futuro, não pronto.
- `wait_all()`: esperar enquanto todos os futuros passados como argumentos não ficarem prontos.
- `make_ready_future()`: criar um futuro no estado pronto com o valor recebido como parâmetro

- `when_all()`: esperar que todos os futuros passados como argumentos estejam prontos; por sua vez, retorna encapsulados como futuros os valores obtidos.

2.4.5 *Sistema de monitorização de desempenho*

O Sistema de monitorização de desempenho expõe métricas: do hardware, da aplicação, do sistema operativo e do tempo de execução das aplicações através de um método intrusivo de instrumentação (Grubel et al., 2015). Para o efeito, são usados contadores de desempenho que coletam dinamicamente dados e informações sobre o desempenho do sistema ou da aplicação, que podem assistir na procura dos gargalos do sistema.

Os contadores são objetos de primeiro nível identificados por um nome simbólico que expõem uma interface uniforme para recolha arbitrária de dados de desempenho relativos, entre outros, ao escalonamento de fios de execução, à camada de transporte de parcelas e o endereçamento global.

2.5 O ESTUDO DE CASO COR

Nesta secção, é apresentado de forma resumida o modelo de programação CoR, nomeadamente os conceitos principais e uma definição formal do Recurso.

2.5.1 *CoR - computação orientada ao recurso*

A investigação iniciada com o Modelo de Computação Celular introduz os conceitos de célula, operão, gene e variável fisiológica que encontram expressão na plataforma para a computação celular PlaCC Pina (1997). Desenvolvimentos posteriores conduziram ao modelo CoR que incorpora na metáfora Recurso a noção de estado, de concorrência, de localidade e de distribuição (Moreira, 2001). Mais recentemente, foi desenhado e implementado um ambiente de desenvolvimento e execução de aplicações concorrentes e paralelas, construídas sobre o paradigma da computação orientada ao Recurso, denominado PlaCoR Ribeiro (2019).

Constituição da entidade recurso

Em CoR podem existir Recursos de múltiplas especializações mas que partilham uma base comum de propriedades e elementos, visível na figura 5:

- Identificação - identificação única no contexto do seu Recurso ascendente;
- Propriedades - propriedades individuais a cada Recurso, assim como propriedades referentes à coerência de estado entre todas as suas, eventuais, réplicas;
- Instância - uma referência à realização concreta do Recurso no sistema hospedeiro.

Os Recursos podem ser físicos ou lógicos. Os primeiros, estão intrinsecamente relacionados com as noções de localidade, de concorrência e de distribuição, intrinsecamente, associados aos Recursos físicos do sistema

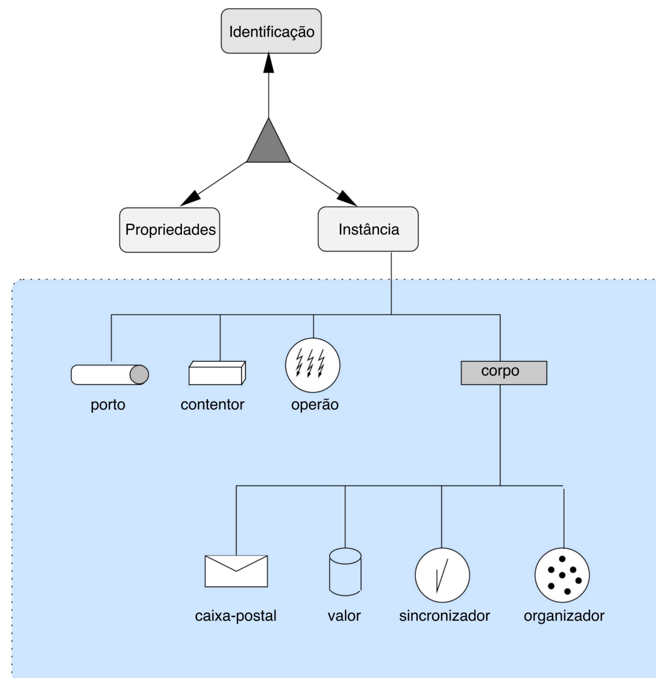


Figura 5: Anatomia do Recurso CoR. Fonte: Moreira (2001).

hospedeiro. Enquanto os segundos dão suporte ao estado da computação e à modularidade no desenho das aplicações. De entre os Recursos destacam-se:

- Tarefa - entidade executável, escalonada como um fio de execução independente;
- domínios - delimitam espaços de endereçamento onde têm lugar as interações entre os Recursos (o equivalente a um processo Unix);
- Dados - para a criação/manipulação de informação, globalmente partilhados num ambiente de computação distribuído;
- Grupos - para o desenho modular de aplicações complexas e de grande dimensão, estruturando os Recursos membros numa árvore hierárquica de dependências;
- Guardas e barreiras - instrumentos de sincronização distribuídos, usados para coordenação da execução de atividades concorrentes/paralelas e zonas de exclusão mútua em código partilhado;
- Portos - mecanismos de comunicação, usados para encaminhar informação entre domínios.

Anatomia do recurso CoR

Os Recursos, para serem modulares e extensíveis, são materializáveis através da composição de elementos mais simples, em que cada um deles concretiza funcionalidades específicas. Esta abordagem torna o modelo mais flexível, oferecendo ao programador um procedimento intuitivo para a construção dos seus próprios Recursos ou alteração dos existentes, na medida em que forem necessárias novas e mais funcionalidades. Por

exemplo, acrescentar um elemento caixa postal a um grupo permite dotar este recurso com a capacidade para enviar/receber mensagens para/de outros Recursos.

Elementos

Apresenta-se a seguir uma definição breve de diferentes elementos de uma instância de recurso no modelo CoR.

- Contentor - dá suporte aos requisitos de memória, aos meios de comunicação e computação;
- Executor - responsável pela atividade computacional no contexto de um domínio;
- Organizador - fornece um mecanismo de estruturação para a gestão e agregação de outros Recursos;
- Valor - região contígua de memória com um tipo associado (por exemplo, variável de um tipo simples, vetores, estruturas) num ambiente de memória partilhada e distribuída;
- Caixa-postal - para a interação local/remota entre Recursos através do envio/receção de mensagens;
- Porto - para selecionar o protocolo e o meio de comunicação que melhor se adequa à transferência de informação entre domínios da aplicação; opcional;
- Sincronizador - para a coordenação e sincronização entre fios de execução.

Definição de Recursos

Os Recursos podem ser formalmente descritos usando a notação:

- `domínio :: contentor . organizador . (porto)?`
- `grupo :: organizador`
- `tarefa :: (executor)+ . (caixa-postal)? . (valor)?`
- `dado :: valor`
- `guarda v barreira :: sincronizador`

Analisando a notação, pode ver-se que, por exemplo, o domínio é constituído por um elemento contentor, seguido por um elemento organizador, seguido por elemento porto (opcional). Notar que o mesmo elemento pode existir em diferentes Recursos, como é o caso do organizador, comum ao domínio e o grupo.

Identificação de Recursos

Para os Recursos poderem interagir foram definidos dois níveis diferentes de identificação: global e local. No primeiro é usado o idp, um valor único em todo o sistema de domínios distribuídos. O outro assenta na assunção de que todo o recurso existe no contexto de um outro recurso estruturado (grupo ou domínio), pelo que, em cada contexto é-lhe atribuído um nome (uma sequência de caracteres) e um identificador idm (um índice de membro) que deverão ser únicos, naquele contexto.

A identificação é gerada automaticamente quando um recurso é criado e/ou se junta a um recurso estruturado. A qualquer momento, um recurso pode tornar-se membro de um outro ascendente, criando-se assim um pseudónimo do recurso original. Podem usar-se, arbitrariamente, ambos os tipos de identificação para designar um recurso, sendo a semântica das operações, independente do modo de identificação utilizado.

Modelos de representação

O CoR permite a seleção do esquema de representação de Recursos que melhor se adequa aos requisitos de localidade e de concorrência da aplicação, dentre: cópia única, replicação e partição.

A existência de um recurso com uma única representação no sistema (cópia única) leva a que o modo de acesso ao seu corpo seja centralizado (figura 6). Sempre que for necessário aceder ao corpo do recurso para executar uma determinada ação, é feita uma RPC ao domínio onde o objeto se encontra instanciado.

Se o modelo for de replicação, leva à existência de múltiplas réplicas (ou de pseudónimos), do corpo do recurso, disseminadas pelo sistema de domínio distribuídos, constituindo assim uma política distribuída (figura 7).

Como uma alternativa ou complemento à replicação, a partição consiste em dividir um recurso por vários aliás, sendo atribuído a cada um uma partição, acessível a título individual.

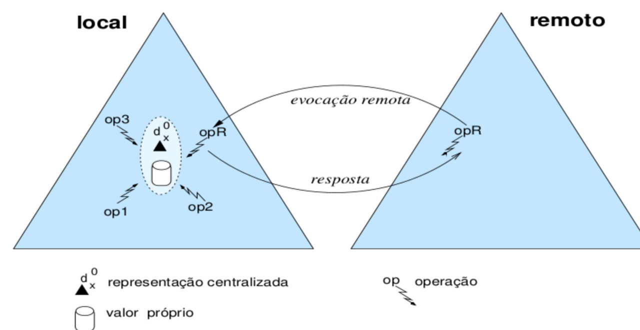


Figura 6: Representação centralizada. Fonte:Moreira (2001)

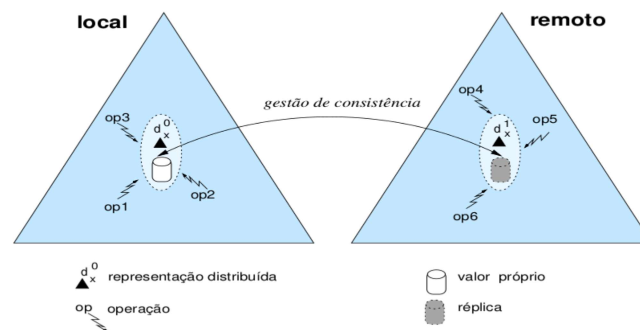


Figura 7: Representação distribuída. Fonte:Moreira (2001)

Estrutura de uma aplicação CoR

Em CoR, a criação/remoção de Recursos e/ou a adesão/saída de grupos em tempo de execução produz uma construção lógica que em cada momento corresponde a uma árvore de dependências que representa a organização lógica de todos os Recursos num sistema de domínios distribuídos, durante o tempo de vida de uma

aplicação. A raiz da árvore é o primeiro domínio da aplicação - meta-domínio - que é o ascendente primordial de todos os Recursos e, simultaneamente, o seu próprio primeiro membro fundador. Os nós da árvore correspondem a Recursos estruturados (domínios e grupos), e as folhas a Recursos simples (tarefas, dados, guardas, barreiras, ...). Na árvore de dependências da figura 8 é possível verificar as relações de afinidade entre as entidades envolvidas na computação, facilitando a estruturação da aplicação.

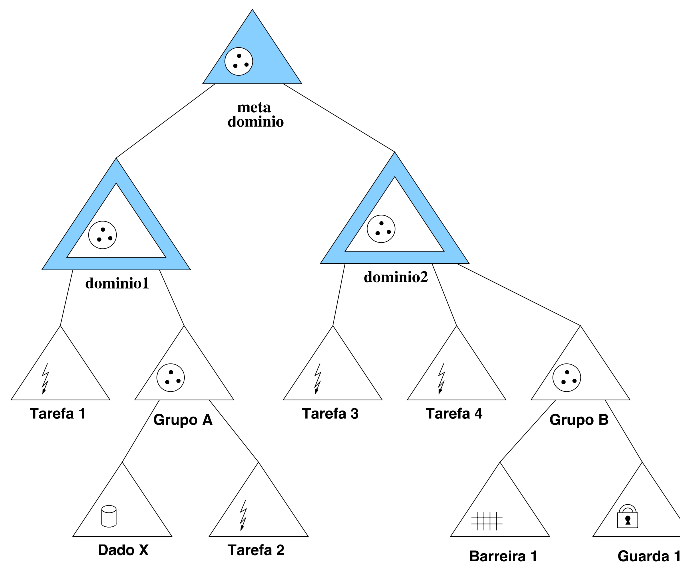


Figura 8: Árvore hierárquica de dependências de uma aplicação CoR. Fonte:Moreira (2001).

3

APLICAÇÕES DISTRIBUÍDAS COM HPX

Nas próximas secções descrevem-se os passos necessários para construir uma aplicação em HPX. É exposto o essencial para compreender a forma de utilizar os LCOs e de como distribuir a computação pelos nós de um sistema distribuído.

3.1 DECLARAR AÇÕES

Em HPX, uma ação é um tipo e um invólucro para uma função C++ "normal" que viabiliza a sua execução, tal como um RPC (*Remote Procedure Call*), mantendo a mesma sintaxe para execução local e remota. No caso da execução remota, os mecanismos de comunicação são implicitamente prestados pelo AGAS, através de parcelas.

As ações podem ter um qualquer número e tipo de argumentos, mas pelo menos um, o primeiro, é obrigatório, que é o GID da localidade de destino (se for uma ação simples) ou o GID do componente (se for uma ação de um componente).

3.1.1 *Ações simples*

Para criar funções globais, também chamadas ações simples (*plain actions*), é necessário usar a macro `HPX_PLAIN_ACTION()` para encapsular a função pretendida, condição necessária para executar em nós remotos. O exemplo que se encontra a seguir (listagem 1) ilustra um caso simples deste procedimento, desenvolvido mais à frente na secção 3.2. No exemplo, tanto a macro `HPX_PLAIN_ACTION` como a função a encapsular devem ser colocadas no espaço global do programa. Se se pretender que a ação seja circunscrita a um espaço, que não o global, a definição da ação tem de ser dividida em duas macros complementares: `HPX_DEFINE_PLAIN_ACTION()` e `HPX_REGISTER_ACTION()`. A primeira macro serve para definir a ação que encapsula a função em questão e a segunda macro expande-se para uma série de objetos globais, que suportam a serialização/iniciação necessárias para a invocação remota da função (ver listagem 2).

```
// Definição da função print_double no espaço global
void print_double(double d) {
    std::cout << d << std::endl;
}
// Macro no espaço global com o nome da função e o nome do tipo da ação a criar
HPX_PLAIN_ACTION(print_double, print_double_action);

int main() {
    print_double_action action;
    action(hpx::find_here(), 2.8); // Execução da ação, na localidade presente,
    return 0;                                               // de forma síncrona
}
```

Listing 1: plain_action.cpp

```
namespace app
{
    // Definição da função print_double dentro do espaço app
    void print_double(double d) {
        std::cout << d << std::endl;
    }
    HPX_DEFINE_PLAIN_ACTION(print_double, print_double_action);
}
// Macro no espaço global com o nome do tipo da ação
// e um nome único para fins internos ao HPX
HPX_REGISTER_ACTION(app::print_double_action, app_print_double_action);

int main() {
    app::print_double_action action;
    action(hpx::find_here(), 2.8); // Execução da ação, na localidade presente,
    return 0;                                              // de forma síncrona
}
```

Listing 2: plain_action_two_macros.cpp

3.1.2 *Ações de componente*

Em HPX, para além de ser possível criar ações que encapsulam funções globais, também é possível criar ações que encapsulam funções membro de classes de objetos. Esta facilidade recorre ao conceito de componente, uma classe que pode ser instanciada remotamente e que cujas funções membro também podem ser invocadas remotamente. Este tipo de objetos e ações serão discutidos em pormenor na secção 3.3.

3.2 EXECUTAR AÇÕES

Após uma ação ser declarada, para a sua invocação pode usar-se uma vasta gama de LCOs que, recorrendo a critérios de eficiência computacional, pode-se classificar em:

- Assíncrono sem sincronização - `hpx::apply();`
- Assíncrono com sincronização - `hpx::async();`

- Síncrona;
- Assíncrono sem sincronização, com continuação - `hpx::apply_continue`;
- Assíncrono com sincronização, com continuação - `hpx::async_continue`.

3.2.1 *Assíncrono sem sincronização.*

Este método, conhecido por "*fire and forget*", irá garantir que a função encapsulada pela ação seja executada na localidade pretendida, com a particularidade de que a função chamadora não fica à espera da execução da função chamada, nem pelo seu eventual valor de retorno. É, por isso, uma operação totalmente assíncrona.

O exemplo a seguir, (listagem 3), mostra como executar uma função desta forma através do LCO `hpx::apply()`. Primeiro é definida uma instância da ação (linha 2) e depois executada (linha 3): recebe um argumento do tipo double (2.8), tal como na assinatura da função original, e é executada na localidade local `hpx::find_here()`.

```
1 int main() {
2     print_double_action action;
3     hpx::apply(action, hpx::find_here(), 2.8);
4     return 0;
5 }
```

Listing 3: async_without_synchronization.cpp

3.2.2 *Assíncrono com sincronização*

Este método garante que a função encapsulada pela ação seja executada de forma assíncrona, na localidade pretendida com obtenção do valor de retorno. Para o efeito, é usado o LCO `hpx::async()`, que retorna sempre uma instância de um `hpx::future<>` que encapsula o resultado da execução, como na listagem 4. Ao executar a ação, o fluxo do programa continua e não fica à espera da execução/terminação da função, que só sincroniza com a operação remota através da chamada de `get()` no `hpx::future<>`, obtendo assim também o valor de retorno.

```
int main() {
    print_double_action action;
    hpx::future<void> f = hpx::async(action, hpx::find_here(), 2.8);
    // ... outro código pode ser executado aqui
    f.get();
    return 0;
}
```

Listing 4: async_with_synchronization.cpp

3.2.3 *Síncrono*

Neste método, (listagem 5), o fio de execução que chamou a ação será suspenso enquanto a função não retornar.

```
int main() {
    print_double_action action;
    action(hpx::find_here(), 2.8);
    return 0;
}
```

Listing 5: sync.cpp

Apesar deste método dar a ilusão de uma invocação convencional de uma função síncrona, internamente a ação `action` irá ser escalonada num novo fio de execução, escondendo uma chamada de função assíncrona.

3.2.4 *Assíncrono sem sincronização, com continuação*

Este método é muito semelhante ao método "Assíncrono sem sincronização (3.2.1)" descrito acima. A diferença reside na possibilidade de encadear operações assíncronas, em que o resultado de cada ação na cadeia é passado como argumento na ação seguinte. Enquanto que `hpx::apply` invoca uma única função usando a semântica "fire and forget",
`hpx::apply_continue` cria assincronamente uma cadeia de funções sem necessidade do fluxo de execução ter de "voltar" ao ponto original de invocação. O LCO `hpx::apply_continue` tem o mesmo funcionamento de `hpx::async_continue`, apresentado a seguir.

Notar que cada uma das funções assíncronas pode ser executada em localidades diferentes.

3.2.5 *Assíncrono com sincronização, com continuação*

Neste método, o LCO utilizado, `hpx::async_continue`, é muito semelhante a `hpx::async` descrito acima em "Assíncrona com sincronização (3.2.1)", mas recebe um argumento adicional: uma função que será usada como continuação da ação executada, tal como `hpx::apply_continue`. Este método encadeia operações de forma assíncrona, fornecendo uma operação de continuação que é automaticamente executada assim que a primeira ação tiver terminado.

No exemplo seguinte, (listagem 6), é mostrado o encadeamento de duas ações, onde o resultado da primeira ação é direcionado para a segunda e o resultado desta é enviado de volta para o local de invocação original.

```cpp
// primeira ação
double add_one(double i) {
    return i+1;
}
HPX_PLAIN_ACTION(add_one, add_one_action);

// segunda ação
double add_two(double i) {
    return i+2;
}
HPX_PLAIN_ACTION(add_two, add_two_action);

int main(){
    add_one_action act1;
    add_two_action act2;

    hpx::future<double> f = hpx::async_continue(
        act1, hpx::make_continuation(act2), hpx::find_here(), 2.8);
    std::cout << f.get() << "\n"; // Irá imprimir: 5.8: ((2.8 + 1) + 2)
    return 0;
}
```

Listing 6: async_with_synchronization_with_continuation_I.cpp

Por omissão, a continuação é executada na mesma localidade de onde `hpx::async_continue` tiver sido invocada. Se for necessário especificar a localidade de onde a continuação deve ser executada, o código da listagem acima (listagem 6) deve ser alterado para o código da listagem 7:

```cpp
int main(){
    add_one_action act1;
    add_two_action act2;
    hpx::future<double> f = hpx::async_continue(act1,
        hpx::make_continuation(act2, hpx::find_here()), hpx::find_here(), 2.8);
    std::cout << f.get() << "\n"; // Irá imprimir: 5.8: ((2.8 + 1) + 2)
    return 0;
}
```

Listing 7: async_with_synchronization_with_continuation_II.cpp

Se quisermos encadear mais de duas operações, de forma similar, pode usar-se o código da listagem 8:

```cpp
int main(){
    add_one_action act1;
    add_two_action act2;
    hpx::future<double> f = hpx::async_continue(act1,
        hpx::make_continuation(act2, hpx::make_continuation(act1)), hpx::find_here(), 2.8);
    std::cout << f.get() << "\n"; // Irá imprimir: 6.8: ((2.8 + 1) + 2 + 1)
    return 0;
}
```

Listing 8: async_with_synchronization_with_continuation_III.cpp

3.3 COMPONENTES

Em HPX, o componente é o nome da estrutura que permite que um objeto C++ possa ser acedido remotamente. Graças às caracteristicas do AGAS, os objetos componentes, identificados por um GID, são conhecidos no espaço de endereçamento global e não vivem limitados aos seus espaços de memória, sendo endereçáveis a partir de qualquer localidade.

Para tornar a programação com componentes mais robusta e ágil, foi também implementada em HPX uma classe que serve de interface para o componente, chamada "componente cliente". Este tipo de classe quando instanciada origina um objeto que é exclusivamente local e comporta-se como os normais objetos C++. A sua função é encapsular a implementação dos componentes - esconder a forma de como são criados e de como as ações são chamadas. São ainda objetos com estado próprio, o que lhes permite ter informação local sobre o componente (potencialmente remoto).

Resumidamente, um componente HPX pode ser representado por uma ou por duas classes C++:

- **Classe servidor**: o componente propriamente dito; possui a implementação das funcionalidades do componente - atributos, funções membro e definição das ações.
- **Classe cliente**: uma interface de alto nível que serve de intermediário de acesso à instância do componente.

É possível construir programas usando unicamente componentes servidor, mas não é aconselhável porque leva à repetição de código e aumenta a propensão a erros. Portanto, é recomendável usar componentes servidor e os correspondentes componentes cliente. Um outro aspeto importante é que o mesmo componente pode ser referenciado a partir de várias localidades, quer seja diretamente através de ações ou através de componentes cliente. Visto de uma outra perspetiva, um componente cliente é uma representação local do componente que está potencialmente remoto.

3.3.1 *Criar um componente*

Nesta subseção, pretende-se mostrar a criação e utilização de um componente em HPX através de um exemplo simples: a implementação de uma fila com acesso remoto.

O caso de uso que se pretende aqui explorar consiste na criação e utilização de um componente cuja classe servidor denomina-se `Queue` e que tem apenas três métodos: i) `Push` - adiciona um ou vários elementos na fila; ii) `Pop` - elimina o último elemento e iii) `Size` - retorna o número de elementos. Estes métodos atuam sobre a variável membro `_fifo` do tipo `std::vector<int>`.

Este exemplo como um todo é constituído por três partes: i) componente servidor - que dá origem aos ficheiros queue.hpp de definição da classe do componente (listagem 9) e queue.cpp de implementação (listagem 10); ii) componente cliente - que dá origem ao ficheiro queue_client.hpp de definição e implementação do componente cliente (listagem 11); e iii) programa - originando no ficheiro do programa principal, que, por sua vez, dá origem ao executável (listagem 12).

Componente servidor

A listagem 9, referente à definição da classe componente `Queue`, mostra que para que a classe seja um componente é necessário adicionar `hpx::components::component_base<>` à definição da classe na linha 1. Ao mesmo tempo, para garantir a exclusão mútua sobre a variável membro `_fifo`, usa-se o mecanismo de sincronização obtido por herança `hpx::components::locking_hook<>` (ainda na linha 1) que inclui internamente a primitiva `hpx::lcos::local::spinlock` a qual esconde e simplifica os mecanismos de sincronização, assegurando que o componente é *thread safe* por omissão.

```
1  class Queue: public hpx::components::locking_hook< hpx::components::component_base<Queue> >
2  {
3  public:
4      template <typename ... Args>
5      void Push(Args&& ... args) {
6          (_fifo.push_back(std::forward<Args>(args)), ...);
7      }
8      int Pop();
9      size_t Size();
10
11     HPX_DEFINE_COMPONENT_ACTION(Queue, Pop, Pop_action_Queue);
12     HPX_DEFINE_COMPONENT_ACTION(Queue, Size, Size_action_Queue);
13
14     template <typename ... Args>
15     struct Push_action_Queue
16     : hpx::actions::make_action<
17         decltype(&Queue::Push<Args...>),
18         &Queue::Push<Args...>
19     >::type {};
20
21 private:
22     std::vector<int> _fifo;
23 };
24
25 typedef Queue::Pop_action_Queue Pop_action_Queue;
26 typedef Queue::Size_action_Queue Size_action_Queue;
27 HPX_REGISTER_ACTION_DECLARATION(Pop_action_Queue);
28 HPX_REGISTER_ACTION_DECLARATION(Size_action_Queue);
```

Listing 9: queue.hpp - Ficheiro cabeçalho de declaração do componente

A invocação remota dos métodos do componente tem de ser mediada por ações. Notar que, `Pop` e `Size` são funções com tipo fixo e `Push` uma função *template*, o que irá dar lugar a diferentes formas de criar ações. As funções `Pop` e `Size` são definidas recorrendo à macro `HPX_DEFINE_COMPONENT_ACTION()` (linhas 11 e 12) que pode ter dois ou três parâmetros que são: o nome da classe, o nome do método e o último (opcional) o tipo da ação. No caso do terceiro parâmetro ser omitido, o nome do tipo da ação é formado pelo nome do método concatenado com o sufixo "_action". Uma vez que o tipo da ação tem de ser obrigatoriamente único, é boa prática escolher o nome explicitamente para evitar erros de compilação, que podem acontecer quando existem métodos com o mesmo nome, mas de classes diferentes.

Após a definição das ações é usado a macro `HPX_REGISTER_ACTION_DECLARATION()` (linhas 27 e 28). tendo como parâmetro o tipo da ação, para as registar no espaço global.

Agora foquemo-nos no método `Push`. Também é possível em HPX criar ações que encapsulam funções com parâmetros *variadic template*, isto é, funções que podem ser chamadas com um número arbitrário de

argumentos de qualquer tipo. Como não é possível iterar os elementos que estão no pacote de parâmetros "args" para fazer push de cada um deles na fila, a iteração tem de ser convertida em recursão. Mas, ao invés de fazer recursão de forma tradicional, foi adotado neste exemplo expressões fold (C++17) que permitem expressar recursão através de um código mais limpo, curto e possivelmente mais fácil de ler (linha 6). Note-se também que a função `Push` recebe os argumentos por referência universal (&&). Para o caso deste tipo de funções *template*, para criar a ação correspondente é necessário definir explicitamente a sua estrutura (linhas 14 a 19). Note-se que quando uma ação é declarada desta forma, fica pronta a usar, ou seja, não é necessário usar, nem separar a declaração e o registo em várias macros.

O ficheiro que incorpora o código da listagem 10 contém os métodos da classe juntamente com o código extra necessário para a sua implementação como um componente. Para registar o componente no AGAS, é necessário usar a macro `HPX_REGISTER_COMPONENT()` (linha 11), com os parâmetros tipo do componente e tipo da classe. As ações são registadas com a macro `HPX_REGISTER_ACTION()` com o tipo da ação (linhas 14 e 15).

```
1   int Queue::Pop() {
2       int element = _fifo.front();
3       _fifo.erase(_fifo.begin());
4       return element;
5   }
6   size_t Queue::Size() {
7       return _fifo.size();
8   }
9
10  typedef hpx::components::component<Queue> Queue_type;
11  HPX_REGISTER_COMPONENT(Queue_type, Queue);
12
13  typedef Queue::Pop_action_Queue Pop_action_Queue;
14  typedef Queue::Size_action_Queue Size_action_Queue;
15  HPX_REGISTER_ACTION(Pop_action_Queue);
16  HPX_REGISTER_ACTION(Size_action_Queue);
```

Listing 10: queue.cpp - Ficheiro de implementação do componente

Neste ponto, o componente `Queue` está definido e pronto a ser instanciado, bastando o seu GID para lhe aceder, através das suas ações, a partir de qualquer localidade.

Em complemento, segue-se-lhe a definição do componente cliente correspondente. Este permite encapsular o código de criação do componente servidor, usar os seus métodos através de ações sem ter de usar explicitamente o seu GID e, ao mesmo tempo, simplificar a gestão do assincronismo.

Componente cliente

A definição e implementação da classe componente cliente `Queue_Client` (listagem 11) herda o código *boilerplate* da classe *template* `hpx::components::client_base<>`, com o par de nomes `<classe_cliente, classe_componente>` (linha 1).

Para instanciar um componente servidor é necessário usar o operador `hpx::new_` (linha 5), similar ao operador `new` do C++, em que o primeiro argumento é o lugar de destino e os restantes são os argumentos do construtor da classe, retornando o GID da instância criada. Se o destino for a localidade atual pode usar-se como alternativa o método `hpx::local_new` (linha 4), com a vantagem de os argumentos poderem ser *non-copyable* e *non-movable*, com a garantia de que a instância é local.

Cada um dos métodos constituintes encapsula a chamada da ação correspondente, recorrendo a `hpx::async` e, por isso, devolvendo um futuro como retorno. Se, por ventura, se pretendesse que a ação ocorresse sem sincronização poder-se-ia ter usado `hpx::apply` (ver outras informações sobre como executar ações em 3.2).

```
1  class Queue_Client: hpx::components::client_base<Queue_Client, Queue> {
2  public:
3      typedef hpx::components::client_base<Queue_Client, Queue> base_type;
4          Queue_Client() : base_type(hpx::local_new<Queue>()) {}
5          Queue_Client(hpx::id_type locality) : base_type(hpx::new_<Queue>(locality)) {}
6
7          template <typename ... Args>
8          hpx::future<void> Push(Args&& ... args) {
9                  typedef Queue::Push_action_Queue<Args...> action_type;
10                 return hpx::async<action_type>(this->get_id(), std::forward<Args>(args)...);
11         }
12
13         hpx::future<int> Pop() {
14                 typedef Queue::Pop_action_Queue action_type;
15                 return hpx::async<action_type>(this->get_id());
16         }
17
18         hpx::future<size_t> Size() {
19                 typedef Queue::Size_action_Queue action_type;
20                 return hpx::async<action_type>(this->get_id());
21         }
22 };
```

Listing 11: queue_client.hpp - Ficheiro cabeçalho de declaração do componente cliente

Programa

Para completar e mostrar a utilização do componente num programa apresenta-se o código 12 de uma função main, a título de exemplo. Pode-se observar a construção do componente cliente, com a localidade atual como argumento, na linha 3, que tem como consequência a criação do componente servidor implicitamente. Na linha 4 é adicionado o valor 1 na fila e na linha 5 são adicionados os valores [2,5]. Notar que é omitido o GID do componente e a sua localidade.

Na linha 7 é utilizado o LCO `hpx::dataflow` para dar continuidade aos futuros f1 e f2, de forma a que, depois de f1 e f2 estarem prontos, seja removido o primeiro elemento da fila através da chamada da função `Pop` (linha 8).

Note-se também que não existe *data race* apesar do assincronismo das ações f1 e f2, dado que a serialização do acesso é garantido pelo componente servidor. Por isso, se a ação correspondente a f1 for executada primeiro, a ação `Pop` que se lhe segue, na linha 8, irá retirar o valor 1. Caso contrário, se a ação associada a f2 for executada primeiro, o elemento removido será 2.

Na linha 13, o futuro f3 obtido na linha 7 é usado pelo LCO `then` para, na continuidade da execução do `Pop` na linha 8, ser executada a função `Size` (linha 13) para imprimir o número de elementos presentes na fila.

```
1  int main(int argc, char* argv[]) {
2      hpx::id_type locality = hpx::find_here();
3      Queue_Client myqueue(locality);
4      auto f1 = myqueue.Push(1);
5      auto f2 = myqueue.Push(2, 3, 4, 5);
6
7      auto f3 = hpx::dataflow([&myqueue](auto f1, auto f2){
8          auto element = myqueue.Pop();
9          hpx::cout << element.get() << hpx::endl; // irá imprimir 1 ou 2,
10                             // depende se f1 foi computado primeiro que f2 ou não
11     }, f1, f2);
12
13     f3.then([&myqueue](auto f3){
14         auto size = myqueue.Size();
15         hpx::cout << size.get() << hpx::endl; // irá imprimir 4
16     }).get();
17 }
```

Listing 12: program_hpx_queue.cpp

Convém referir que no exemplo se optou por esconder totalmente a manipulação do componente servidor, disponibilizando apenas o acesso a ele através do componente cliente. Em alternativa: i) o componente servidor poderia ter sido instanciado explicitamente na função *main* e o seu GID passado como argumento no construtor do componente cliente; ii) os dois componentes poderiam ter sido criados de forma independente e o GID do servidor passado ao cliente posteriormente; iii) ou ainda criar unicamente o componente servidor e aceder a ele executando as ações de forma explícita, de certa forma, usando o código do componente cliente na função *main* diretamente. Notar ainda que no exemplo a utilização de `hpx::dataflow` e `then` é meramente ilustrativa, não tendo em consideração questões de desempenho.

3.3.2 *Criar um componente template*

O objetivo desta subsecção é, à luz da subsecção anterior, apresentar o modo de como se cria e usa um componente, mas agora para o caso de um componente *template*, ou seja, um componente com um tipo arbitrários de dados.

A introdução de *templates* altera significamente a estrutura do código, uma vez que, em C++, as classes *template* são obrigatoriamente definidas e implementadas em ficheiros *header*, sendo que o compilador precisa de

instanciar diferentes versões de classe, dependendo dos parâmetros fornecidos/deduzidos. Notar que, quando se usa *templates*, uma definição da classe não representa univocamente o código, mas um modelo para várias versões do mesmo. Uma solução comum é escrever a declaração modelo num ficheiro de *header* e depois escrever a implementação da classe num ficheiro de implementação, por exemplo .tpp, e incluir esse ficheiro de implementação no ficheiro *header*. No que segue, para simplificar, a implementação está diretamente colocada no ficheiro *header* (listagem 13).

Componente servidor

Em comparação com a listagem anterior 9, a primeira alteração que se pode observar é a utilização de `template <typename T>` (linha 1) que substituiu o tipo concreto `int` pelo tipo arbitrário `T`. A seguir a esta, as modificações mais significativas são a forma como as ações são declaradas e registadas. Uma vez que a classe é *template*, tanto a declaração como o registo das ações são feitas através de macros, definidas pelo programador na definição da classe. São então definidas as macros `REGISTER_QUEUE_DECLARATION` e `REGISTER_QUEUE` que deverão ser usadas quando se pretender usar este tipo de componente, para o registar a si e às suas ações. O código completo encontra-se no anexo B.1.1.

```
1  template <typename T>
2  class Queue: public hpx::components::locking_hook< hpx::components::component_base<Queue<T>> >
3  {
4  public:
5      template <typename ... Args>
6      void Push(Args ... args);
7      T Pop();
8      size_t Size();
9
10     HPX_DEFINE_COMPONENT_ACTION(Queue, Pop, Pop_action_Queue);
11     HPX_DEFINE_COMPONENT_ACTION(Queue, Size, Size_action_Queue);
12
13     template <typename ... Args>
14     struct Push_action_Queue
15     : hpx::actions::make_action<
16         decltype(&Queue::Push<Args...>),
17         &Queue::Push<Args...>
18     >::type {};
19 };
20 #define REGISTER_QUEUE_DECLARATION(type)
21 // ... declaração das ações
22 #define REGISTER_QUEUE(type)
23 // ... registo das ações e registo do componente
```

Listing 13: queue.hpp - Ficheiro cabeçalho de declaração do componente template

Componente Cliente

No componente cliente (listagem 14) a estrutura da classe foi adaptada para contemplar o tipo arbitrário `template <typename T>` e a forma como as ações são chamadas. De entre as alterações destacam-se, na linha 10,

a definição `typedef`, que para definir um novo tipo necessita de ser sucedida de `typename`, para informar o compilador que `action_type` é um tipo e não um membro estático de `Queue_Client`. O código completo encontra-se no anexo B.1.2.

```
1  template <typename T>
2  class Queue_Client: hpx::components::client_base<Queue_Client<T>, Queue<T>>
3  {
4  public:
5      typedef hpx::components::client_base<Queue_Client<T>, Queue<T>> base_type;
6  
7          template <typename ... Args>
8          hpx::future<void> Push(Args ... args)
9          {
10                 typedef typename Queue<T>::template Push_action_Queue<Args...> action_type;
11                 return hpx::async<action_type>(this->get_id(), args...);
12         }
13         // ...
14 };
```

Listing 14: queue_client.hpp - Ficheiro cabeçalho de declaração do componente cliente templated

Programa

O programa (listagem 15) é análogo ao programa anterior (listagem 12) substituindo na fila o tipo `int` pelo tipo `Object` serializável, uma classe arbitrária criada pelo utilizador. A macro `REGISTER_QUEUE` na linha 1, definida previamente na declaração do componente (listagem 13), é aqui usada para criar o código do componente com tipo `Object`.

```
1  REGISTER_QUEUE(Object);
2  
3  int main(int argc, char* argv[]) {
4  
5      hpx::id_type locality = hpx::find_here();
6      Queue_Client<Object> myqueue(locality);
7      Object objA, objB, objC;
8  
9      auto f1 = myqueue.Push(objA);
10     auto f2 = myqueue.Push(objB, objC);
11 
12     auto f3 = hpx::dataflow([&myqueue](auto f1, auto f2){
13         auto element = myqueue.Pop(); // irá retornar o objeto objA ou objB, depende se f1
14                                       // foi computado primeiro que f2 ou não
15     }, f1, f2);
16 
17     f3.then([&myqueue](auto f3){
18         auto size = myqueue.Size();
19         hpx::cout << size.get() << hpx::endl; // irá imprimir 2
20     }).get();
21 }
```

Listing 15: program_hpx_queue_template.cpp

3.3.3 *Criar um componente template migrável*

Por omissão, os componentes em HPX mantêm-se sempre na localidade onde foram criados. Para poder transferir um componente de uma localidade para outra, o programador tem de explicitamente o criar como um componente migrável. Nesta subsecção apresentamos a mesma classe "Queue" do exemplo anterior, mas agora com a funcionalidade de migrar entre localidades.

Componente servidor

Para definir um componente migrável, na definição da classe (listagem 16) é necessário acrescentar `hpx::components::migration_support<>` (linha 2 e 3), de forma a herdar o código complementar HPX. Atualmente, esta opção é incompatível com a herança de `hpx::components::locking_hook< >`, pelo que, neste caso, a serialização dos acessos ao componente é da responsabilidade do programador. Complementarmente, o componente para ser migrável precisa de ser serializável e construível por cópia, pelo que são adicionados construtores de cópia (linhas 9 à 23) e o método de serialização (linha 26). O código completo encontra-se no anexo B.2.1. (Para mais informações sobre serialização, ver a secção 3.5).

```
1  template <typename T>
2  class Queue: public hpx::components::migration_support<
3                      hpx::components::component_base<Queue<T>> >
4  {
5  public:
6      typedef hpx::components::migration_support<
7              hpx::components::component_base<Queue<T>> > base_type;
8
9      Queue(Queue const& rhs)
10       : base_type(rhs), _fifo(rhs._fifo) {}
11
12     Queue(Queue && rhs)
13       : base_type(std::move(rhs)), _fifo(rhs._fifo) {}
14
15     Queue& operator=(Queue const & rhs) {
16         _fifo = rhs._fifo;
17         return *this;
18     }
19
20     Queue& operator=(Queue && rhs) {
21         _fifo = rhs._fifo;
22         return *this;
23     }
24
25     template <typename Archive>
26     void serialize(Archive& ar, unsigned version) {
27         ar & _fifo;
28     }
29 };
```

Listing 16: queue.hpp - Ficheiro cabeçalho de declaração do componente template migrável

Componente cliente

Ao acrescentar a um componente a capacidade de migração, isso não implica que se tenha de alterar o componente cliente correspondente. Neste caso, é adicionada uma nova função membro ("Migrate") por conveniência (listagem 17), para encapsular a chamada de `hpx::components::migrate`, em que o primeiro argumento é o GID do componente, o segundo é o GID da localidade de destino e o retorno é o GID do componente migrado.

```
hpx::future<hpx::id_type> Migrate(hpx::id_type dest) {
        return hpx::components::migrate<Queue<T>>(this->get_id(), dest);
}
```

Listing 17: queue_client.hpp - Ficheiro cabeçalho de declaração do componente cliente template migrável

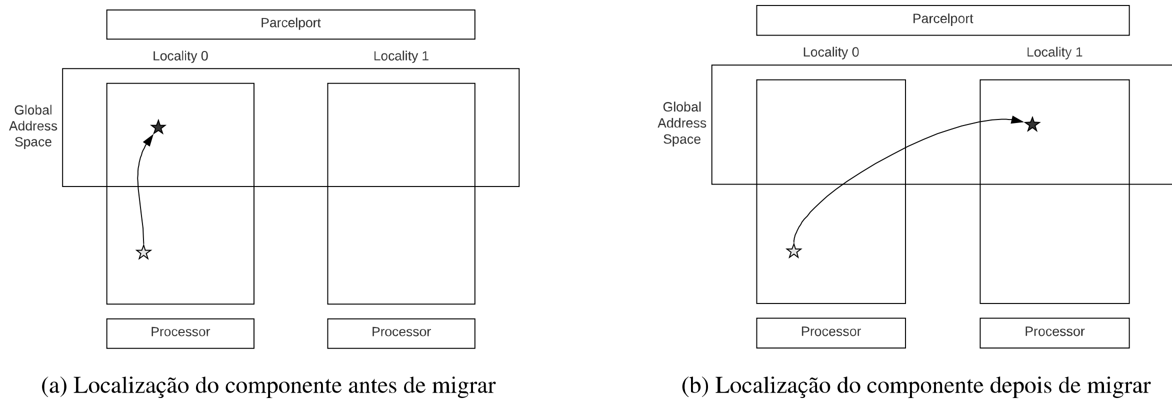
Programa

No programa (listagem 18) é mostrado a forma de criar um componente do tipo `Queue<Object>` e a migração entre localidades em tempo de execução, num ambiente distribuído com mais do que uma localidade.

Assim, as localidades remotas são descobertas através da função `hpx::find_remote_localities()` (linha 9), e escolhe-se a localidade de destino (linha 10) que é passada como parâmetro à função `Migrate` (linha 11) para migrar o componente (ver figura 9). Importa salientar que o GID do componente permanece inalterado, mesmo depois de ter migrado, e o fluxo do programa continua com a chamada da função Pop (linha 13).

```
int main(int argc, char* argv[]) {

    hpx::id_type locality = hpx::find_here();
    Queue_Client<Object> myqueue(locality);
    Object objA, objB, objC;

    myqueue.Push(objA, objB, objC).get();

    std::vector<hpx::id_type> remote_localities = hpx::find_remote_localities();
    hpx::id_type dest = remote_localities[0];
    myqueue.Migrate(dest).get(); // migrar o componente

    myqueue.Pop().get();

    auto size = myqueue.Size();
    std::cout << size.get() << std::endl; // irá imprimir 2

    return 0;
}
```

Listing 18: program_hpx_queue_template_migrable.cpp



(a) Localização do componente antes de migrar (b) Localização do componente depois de migrar

Figura 9: Ilustração da migração de um componente, da localidade `0` para a localidade `1` - a estrela cinzenta representa o objeto componente cliente e a estrela a negro representa o objeto componente servidor.

3.3.4 *Criar um componente template migrável com herança*

A herança múltipla de classes é uma funcionalidade importante no desenvolvimento de *software* que no caso do HPX acrescenta complexidade e dificuldade, razão pela qual é geralmente substituída por composição de classes. Contudo, continuando com o mesmo caso de uso, apresenta-se a constituição de um componente *template*, migrável e com herança simples, introduzindo para o efeito uma classe abstrata denominada `Container`, da qual a classe componente `Queue` é agora uma classe derivada. O código completo e a sua explicação detalhada encontra-se no anexo B.3.

Ao código de implementação fiz-se corresponder ficheiros distintos: i) declaração de `Container`, ii) implementação de `Container`, iii) declaração e implementação de `Queue`, iv) declaração `Queue_Client` e v) programa principal.

Componente servidor Container (classe base)

O código da classe `Container` foi subdividido num ficheiro cabeçalho e outro de implementação. O código encontra-se em anexo em B.3.1 Esta classe apenas serve como base de polimorfismo, uma vez que é uma classe abstrata. A classe define uma única variável membro `_id` do tipo `int` e um método `GetId` que retorna essa variável membro e, para fins de serialização, construtores de cópia e a função de serialização `serialize`. Ao transformar uma classe base abstrata num componente chama-se a atenção para o seguinte:

- Uma ação não pode encapsular diretamente uma função membro abstrata. Para tal, é necessário recorrer a uma função membro auxiliar não abstrata para servir de intermediário. Esta função auxiliar não abstrata, que irá ser a ação, é que irá chamar a função membro abstrata que se pretende.
- Para uma classe abstrata ser um componente, tem de ser executado no espaço global a macro `HPX_DEFINE_GET_COMPONENT_TYPE()` com o nome da classe, ao contrário das classes normais em que se executa `HPX_REGISTER_COMPONENT()`.

Componente servidor Queue (classe derivada)

A estrutura de base é análoga ao código da "Queue" *template* migrável (listagem 16), mas agora com algumas modificações no construtor e com o acrescento de algumas definições de tipos impostas para este tipo de componentes com herança. Código em anexo B.3.2

Componente cliente

O componente cliente é responsável por servir de interface ao componente servidor `Queue` e, por consequencia, à classe de base `Container`. Mantém a mesma linha do exemplo da secção anterior 3.3.3, sendo adicionado agora um novo método para encapsular a função `GetId` da classe base `Container`. Código em anexo B.3.3

Programa

Relativamente ao programa que serve de exemplo para a utilização do componente, considera-se a listagem 19. É análogo ao programa da secção anterior (código 18), com a diferença de que o componente é criado com mais um argumento (id) (linha 3) e uma nova chamada de função de `GetId` (linha 17). Realça-se com este exemplo a amplitude de possibilidades de construção e o polimorfismo que os componentes incorporam.

```
1  int main(int argc, char* argv[]) {
2      hpx::id_type locality = hpx::find_here();
3      Queue_Client<Object> myqueue(locality, 42);
4      Object objA, objB, objC;
5
6      myqueue.Push(objA, objB, objC).get();
7
8      std::vector<hpx::id_type> remote_localities = hpx::find_remote_localities();
9      hpx::id_type dest = remote_localities[0];
10     myqueue.Migrate(dest).get(); // migrar o componente
11
12     myqueue.Pop().get();
13
14     auto size = myqueue.Size();
15     std::cout << size.get() << std::endl; // irá imprimir 2
16
17     auto id = myqueue.GetId(); // função da classe base container
18     std::cout << id.get() << std::endl; // irá imprimir 42
19 }
```

Listing 19: program_hpx_queue_template_migrable_derived.cpp - Ficheiro de um exemplo de um programa para utilização do componente Queue com herança de Container

3.4 CANAIS

Num ambiente de memória distribuída, para trocar dados entre tarefas em diferentes localidades podem usar-se ações e canais. Estes últimos, como o nome indica, são funcionalmente equivalentes aos canais usados para passagem de mensagem (*sends* e *receives*) no MPI.

Por exemplo, num algoritmo de *stencil* de cálculo de transferência de calor numa placa, o padrão de programação é dividir o domínio em várias secções em que cada uma é atribuída a um fio de execução diferente, eventualmente, num processo remoto. A fronteira de cada secção tem de ser partilhada com as secções vizinhas. Para enviar esses dados podem ser usados canais.

Para além de comunicação (a troca de um valor), os canais podem também ser usados para sincronização.

Para ilustrar este conceito, na listagem 20, no mesmo fio de execução é usado um canal para enviar um valor entre dois pontos.

```
int main() {
    hpx::lcos::local::channel<int> canal; // cria um canal do tipo int
    canal.set(42); // coloca no canal o valor 42
    // recebe do canal o valor 42 dentro de um future
    hpx::future<int> msg_received = canal.get();
    std::cout << msg_received.get() << std::endl; // irá imprimir '42'
}
```

Listing 20: local_channel.cpp

O mesmo canal também pode ser utilizado por fios de execução diferentes, estabelecendo a comunicação entre dois locais independentes no programa, dentro da mesma localidade (listagem 21).

```
void troca_de_dados(hpx::lcos::local::receive_channel<int> canal,
                    hpx::lcos::local::send_channel<> done)
{
    hpx::future<int> received_data = canal.get(); // recebe o valor
    std::cout << received_data.get() << std::endl; // imprime 42
    done.set(); // envia o sinal de retorno
}

int main() {
    hpx::lcos::local::channel<int> canal; // canal para transmissão de um valor
    hpx::lcos::local::channel<> done; // canal para transmissão de um acknowledgement
    hpx::apply(&troca_de_dados, canal, done); // execução assincrona

    canal.set(42); // envia um valor do tipo int
    done.get(); // espera pelo sinal de retorno
}
```

Listing 21: local_channel_two_threads.cpp

Um canal componente é um outro tipo de canal que tem acesso remoto e portanto, pode ser usado para trocar dados entre localidades diferentes. A listagem 22 ilustra um exemplo com a utilização deste tipo de canal. Em canais componente é necessário definir a macro `HPX_REGISTER_CHANNEL()` com o tipo de dados a enviar, para fins de serialização. No exemplo também é mostrado como um canal pode ser usado para enviar uma gama de valores.

```
HPX_REGISTER_CHANNEL(double);

void troca_de_dados(hpx::lcos::channel<double> canal) {
    for (double d : canal)
        std::cout << d << std::endl;
}
HPX_PLAIN_ACTION(troca_de_dados, troca_de_dados_action);

int main() {
    // cria o canal na localidade atual
    hpx::lcos::channel<double> canal(hpx::find_here());
    // envia o canal para a ação, que potencialmente irá executar remotamente
    hpx::apply(troca_de_dados_action(), hpx::find_here(), canal);
    // envia os dados para o recetor
    std::vector<double> v = { 1.2, 3.4, 5.0 };
    for (double d : v)
        canal.set(d);

    // explicitamente encerra o canal (implícito quando sai fora do escopo)
    canal.close();
}
```

Listing 22: distributed_channel.cpp

3.5 SERIALIZAÇÃO DE OBJETOS

Para enviar e receber estruturas de dados arbitrárias através do meio físico e de forma transparente é necessário um protocolo de serialização. O HPX suporta serialização de dados de tipos nativos da biblioteca padrão do C++, tipos de dados construídos pelo programador e tipos de dados próprios do HPX (entre estes, o `hpx::id_type` e `hpx::future`).

A serialização do HPX é baseada na biblioteca "Boost serialization"[1], que também serve de inspiração para as bibliotecas de serialização "cereal"[2] e "YaS"[3].

De seguida, mostra-se um exemplo de serialização de um objeto definido pelo utilizador (listagem 23), que pertence à classe `MyClass`, tem os atributos, `length` e `width` e quatro funções membro (`getLength`,

1 www.boost.org/doc/libs/release/libs/serialization/
2 https://uscilab.github.io/cereal/
3 https://github.com/niXman/yas

getWidth, setLength e setWidth). Para tornar esta classe serializável pode-se usar o modo intrusivo ou o não intrusivo. No exemplo, para permitir que o utilizador tenha um maior controlo do que pretende serializar, usa-se o modo intrusivo que permite indicar explicitamente as variáveis membro que se pretende serializar.

Para o efeito, acrescenta-se à classe a função membro "serialize" que é *template* para poder ser usada nos dois processos de serialização - serializar (output archive) e desserializar (input archive).

```
class MyClass {
    private:
        int length;
        int width;

    public:
        template <typename Archive>
        void serialize(Archive& ar, unsigned) {
            ar & length;
            ar & width;
        }

        int getLength(){ return length; }
        int getWidth(){ return width; }
        void setLength(int length){ this->length = length; }
        void setWidth(int width){ this->width = width; }
};

int main() {
    // criação de um objeto do tipo MyClass
    std::unique_ptr<MyClass> p1 = std::make_unique<MyClass>();
    p1->setLength(90);
    p1->setWidth(40);

    // serializar
    std::vector<char> buffer;
    hpx::serialization::output_archive oarchive(buffer);
    oarchive << p1;

    // desserializar
    std::unique_ptr<MyClass> p2;
    hpx::serialization::input_archive iarchive(buffer);
    iarchive >> p2;

    return 0;
}
```

Listing 23: object_serialization.cpp

Na listagem 23, dentro da função main(), é criado um objeto p1 do tipo MyClass e um apontador para ele. O objeto será serializado para o vetor buffer. De seguida, o objeto será desserializado para p2. Este é um exemplo básico que ilustra a funcionalidade de serialização explícita do HPX.

3.6 INICIALIZAR E EXECUTAR O RUNTIME HPX

HPX é uma biblioteca C++ que oferece suporte a um conjunto de mecanismos críticos para gestão dinâmica de recursos e escalonamentos de tarefas num espaço de endereçamento global. Para que o ambiente de execução seja iniciado num programa é necessário adicionar algumas instruções no código fonte, dependendo dos modos seguintes:

- Implícito - A função main() é reutilizada e é o ponto de arranque do HPX.
- Explícito - Providenciar um ponto de arranque para o HPX enquanto que o fio de execução da função `main()` é bloqueado.
- Explícito assíncrono - Providenciar um ponto de arranque para o HPX mas evitando que o fio de execução da função `main()` seja bloqueado.
- Suspender e retomar - Como o modo anterior, com a diferença de ser possível suspender e retomar o *runtime* HPX para que outros *runtimes* sejam executados.

Implícito:

De forma implícita, o *runtime* é executado ao longo de todo o programa e de forma menos intrusiva no código. Arranca quando a função `main()` é chamada e para automaticamente após esta terminar. Aqui, a função `main()` definida pelo utilizador - que obrigatoriamente deve declarar uma instrução *return* - é substituída por uma função equivalente definida pelo HPX, resultante do pré-processamento assegurado pela utilização do cabeçalho `hpx/hpx_main.hpp` (listagem 24):

```
#include <hpx/hpx_main.hpp>
int main(int argc, char * argv[]) {
    return 0;
}
```

Listing 24: start_runtime_implicit.cpp

Explícito:

Neste modo, o código e *runtime* HPX é executado no escopo da função `hpx_main()`, que deverá ter como retorno a função `hpx::finalize()` e ser chamada dentro da função `main()`. Para tal, tem de se incluir o *header* `hpx/hpx_init.hpp` (listagem 25).

```
#include <hpx/hpx_init.hpp>
int hpx_main(int argc, char * argv[]) {
    // Código executado no runtime
    return hpx::finalize();
}
int main(int argc, char * argv[]) {
    // Inicializa o HPX e corre hpx_main na primeira hpx thread,
    // e espera que hpx::finalize seja chamado.
    return hpx::init(argc, argv);
}
```

Listing 25: start_runtime_explicit.cpp

A função `hpx_main` pode ter uma das seguintes assinaturas:

```
int hpx_main();
int hpx_main(int argc, char * argv[]);
int hpx_main(boost::program_options::variables_map& vm);
```

A função `hpx::init` arranca com uma *thread* do HPX e nela chama a função `hpx_main()`. A função `hpx::finalize` tem de ser declarada para terminar o *runtime* HPX. Esta forma de invocar o HPX traz a vantagem de se poder decidir a versão `hpx::init` a chamar o que também permite passar parâmetros de configuração adicionais na inicialização do *runtime* HPX.

Explícito assíncrono:

Esta forma é análoga à anterior, com algumas diferenças (listagem 26). São introduzidas duas novas funções: `hpx::start` e `hpx::stop`. O fio de execução que irá executar `hpx::start` não irá bloquear à espera que o *runtime* termine, mas irá retornar imediatamente. Este método de invocar o HPX é útil em aplicações em que o fio de execução principal é utilizado para operações específicas, tais como interfaces gráficas. O único cabeçalho que se tem de incluir é `hpx/hpx_start.hpp`.

```
#include <hpx/hpx_start.hpp>
int hpx_main(int argc, char * argv[]) {
    // Código executado no runtime
    return hpx::finalize();
}
int main(int argc, char * argv[]) {
    // Inicializa o HPX e corre hpx_main.
    hpx::start(argc, argv);
    // std::cout << "Hello from the main thread!" << std::endl;
    // Espera que hpx::finalize seja chamado.
    return hpx::stop();
}
```

Listing 26: start_runtime_explicit_async.cpp

Suspender e retomar:

Para possibilitar a coabitação de vários *runtimes* no mesmo programa, o HPX fornece duas funções: `hpx::suspend` e `hpx::resume` (listagem 27). `hpx::suspend` é uma função bloqueante que irá suspender os fios de execução da *pool* de fios de execução do sistema após as tarefas do HPX terminarem. `hpx::resume` acorda os fios de execução suspensos e retoma o *runtime*. Os únicos cabeçalhos que se tem de incluir são `hpx/hpx_start.hpp` e `hpx/hpx_suspend.hpp`.

```
#include <hpx/hpx_start.hpp>
#include <hpx/hpx_suspend.hpp>
int main(int argc, char * argv[]) {
    // Inicializa o HPX, mas não corre hpx_main
    hpx::start(nullptr, argc, argv);
    // Escalona uma função no runtime HPX
    hpx::apply(&my_function, ...);
    // Espera que todas as tarefas terminem e suspende o runtime HPX
    hpx::suspend();
    // Pode ser executado código não HPX
    // Retoma o runtime HPX
    hpx::resume();
    // Escalona mais trabalho no runtime HPX
    // hpx::finalize tem de ser chamado do runtime HPX antes de hpx::stop
    hpx::apply([]() { hpx::finalize(); });
    return hpx::stop();
}
```

Listing 27: start_runtime_suspend_resume.cpp

3.7 EXECUTAR PROGRAMAS

A maioria das aplicações HPX são executadas em *clusters* com vários nós, cujas plataformas fornecem tipicamente serviços integrados de gestão de *trabalhos* que facilitam a atribuição de recursos físicos para cada programa. O HPX inclui suporte para dois dos sistemas de gestão de trabalho mais comuns, o PBS[4] e o SLURM[5]. Por omissão, a comunicação entre diferentes localidades é efetuada usado o *parcelport* TCP, mas além deste, é possível usar outros, tais como MPI, VERBS e LIBFABRIC.

Para execução de programas com o MPI e o SLURM não é necessário definir as principais configurações de arranque, se assim for pretendido. Os processos irão conectar-se sobre o mesmo *runtime* e irão sincronizar no ínicio e no fim da execução, de forma automática. Além de executar programas de forma estática, isto é, com a mesma configuração das localidades durante toda a execução, também é possível fazê-lo de forma dinâmica.

Em HPX é possível executar programas separados partilhando o mesmo espaço de endereçamento de memória, através do AGAS. Por exemplo, se um programa HPX estiver a correr num certo nó `0`, e outro programa a correr no nó `1`, e estiverem ambos ligados ao mesmo endereço e porta de comunicação AGAS, ambos irão

4 https://www.openpbs.org/
5 https://slurm.schedmd.com/

partilhar o seu espaço de memória, podendo um aceder à memória local do outro. Isto é particularmente interessante para programas que precisem de mais unidades de processamento ao longo do tempo, sem que seja necessário terem de ser reiniciados para mudar a configuração inicial. Destacam-se algumas opções que podem ser introduzidas na linha de comandos:

- –hpx:hpx=ip:port
  Endereço IP em que o *parcelport* do HPX está a escutar, formato padrão: endereço:porta (predefinido: 127.0.0.1:7910).
- –hpx:expect-connecting-localities
  Esta localidade pressupõe que outras localidades se liguem de forma dinâmica.
- –hpx:agas=ip:port
  O endereço IP no qual o servidor principal AGAS está a funcionar, formato padrão: endereço:porta (predefinido: 127.0.0.1:7910).
- –hpx:connect
  Executar este exemplo em modo *worker*, mas com conexão posterior.
- –hpx:run-hpx-main
  Executar a função hpx_main, independentemente do modo de localidade.
- –hpx:worker
  Executar a instância do programa em modo *worker*.

Todas as opções podem ser encontradas na documentação oficial do HPX[6].

Para ilustrar a utilização destas opções de linha de comandos, suponha-se que existem três programas, "servidor", "cliente1" e "cliente2", e que cada um corre numa localidade diferente, nomeadamente, em nós diferentes. Neste caso de uso (listagem 28) os três programas são corridos manualmente em consolas separadas, sem recurso a nenhum escalonador de processos (PBS, SLURM, ...).

```
./servidor --hpx:hpx=10.1.2.1:1337 --hpx:expect-connecting-localities

./cliente1 --hpx:hpx=10.1.2.253:1340 --hpx:agas=10.1.2.1:1337
--hpx:connect --hpx:run-hpx-main --hpx:expect-connecting-localities --hpx:worker

./cliente2 --hpx:hpx=10.1.2.254:1334 --hpx:agas=10.1.2.1:1337
--hpx:connect --hpx:run-hpx-main --hpx:expect-connecting-localities --hpx:worker
```

Listing 28: Execução de três programas em consolas separadas, no mesmo runtime

O que acontece na listagem 28 é o seguinte: o programa "servidor" está a correr no nó com endereço `10.1.2.1:1337` e como é o primeiro programa a ser executado, o servidor de AGAS é naturalmente iniciado lá. Como foi executado com o comando `-hpx:expect-connecting-localities`, permite que outros programas se conectem a ele. O programa "cliente1" está a correr no nó com endereço

6 https://stellar-group.github.io/hpx-docs/branches/master/html/manual/launching_and_configuring_hpx_applications.html#hpx-command-line-options

`10.1.2.253:1340` e conecta-se ao servidor AGAS localizado no endereço do programa "servidor" partilhando assim o mesmo espaço de endereçamento, portanto, o mesmo *runtime*. O programa "cliente2" está a correr no nó com endereço `10.1.2.254:1334` e o seu comportamento é análogo ao do "cliente1".

Neste caso, estão os três programas a serem executados em nós diferentes, mas também é possível que os 3 programas sejam executados no mesmo nó, originando 3 localidades no mesmo nó (o IP *parcelport* HPX dos três programas será o mesmo, só mudará a porta). Apesar de ser possível ter mais do que uma localidade por nó, tal não é vantajoso porque irão partilhar os mesmo recursos físicos (cpu, memória, etc).

4

PLATAFORMA PARA COMPUTAÇÃO ORIENTADA AO RECURSO

4.1 ANTECEDENTES

O CoR-HPX não pode, de forma alguma, ser separado dos trabalhos anteriores que visavam animar o modelo de computação CoR Moreira (2001). Em particular, torna-se imperioso apresentar os princípios e resultados que culminaram na construção da plataforma PlaCoR Ribeiro (2019).

O PlaCoR foi criado como uma biblioteca de classes em C++ Moderno, como alternativa a versões prévias em C e Python, com enormes vantagens para o desenvolvimento do protótipo, com destaque para o suporte a: i) orientação aos objetos - na definição e construção dos Recursos assente em mecanismos de herança; ii) programação genérica - por forma a abstrair na API as diferentes classes de Recursos; iii) programação concorrente - para tirar partido dos fios de execução e estruturas de sincronização nativas da própria linguagem.

Para acomodar os conceitos do modelo de computação foi necessário recorrer a várias bibliotecas auxiliares (ver figura 10), que de alguma forma condicionaram a sua conceção e realização.

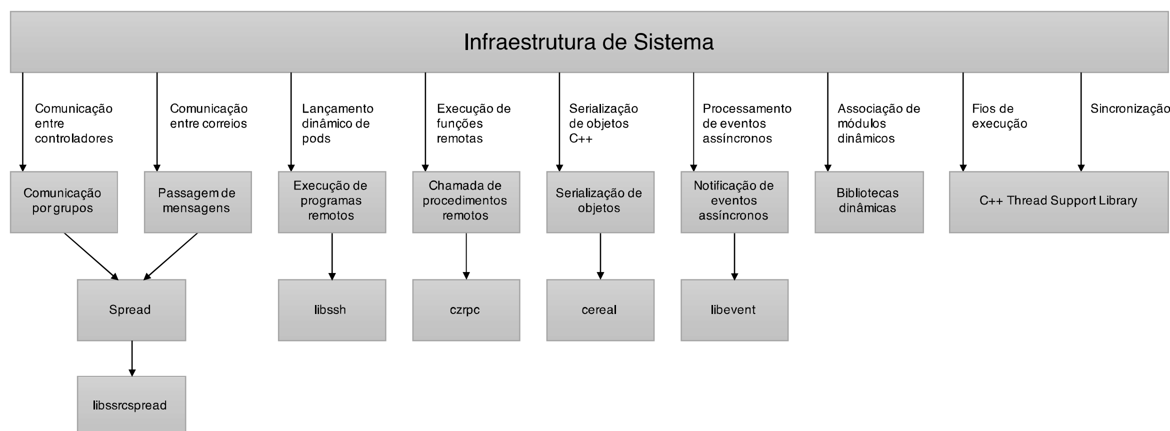


Figura 10: Ferramentas de desenvolvimento do protótipo PlaCoR, Fonte: Ribeiro (2019)

Tratando-se de um ambiente de execução paralelo e distribuído era necessário garantir a consistência entre as diferentes réplicas dos Recursos e a coordenação entre serviços, tais como: controladores e correio. Para o efeito, recorreu-se à biblioteca spread (Amir et al., 2004) que fornece mecanismos de comunicação i) por grupo ii) unicast e iii) multicast e garantia de ordem e de entrega das mensagens, mesmo no caso de falhas. Para

facilitar a integração da biblioteca spread no ambiente de programação recorreu-se ao binding C++ da biblioteca libssrcspread[1] e à biblioteca libevent[2] para o processamento de eventos.

As capacidades previstas na API para a instanciação de Recursos remotos e a execução de funções remotas foram realizadas através das facilidades da biblioteca czrpc3[3] que possui as seguintes caraterísticas: i) não necessita de código adicional para realizar chamadas remotas, ii) em tempo de compilação, deteta argumentos inválidos, iii) não tem dependências externas e iv) a camada de comunicação pode ser adaptada a bibliotecas de comunicação externas.

Uma outra questão fundamental, associada à criação de réplicas distribuídas de Recursos, é a necessidade de serialização de tipos de dados arbitrários e suporte à herança e polimorfismo. Para o efeito, recorreu-se à biblioteca *header-only* cereal[4] que dispõe de facilidades para serialização, não apenas de classes C++ Standard Library, mas de classes arbitrárias definidas pelo programador.

O lançamento e execução de aplicações através de conexões seguras em máquinas remotas foi assegurado pela biblioteca libssh[5], que disponibiliza uma API de fácil utilização e que pode ser usada num ambiente *multithreaded*.

Relativamente ao arranque e execução de aplicações, uma aplicação PlaCoR consiste num ou mais módulos, escritos na linguagem C++, compilados sob a forma de bibliotecas dinâmicas que se ligam à biblioteca PlaCoR. Um destes módulos é um módulo de arranque, onde tem de estar definido obrigatoriamente uma função com o nome *Main*, com a mesma assinatura da função padrão *main* da linguagem C/C++.

4.2 INTRODUÇÃO AO COR-HPX

O surgimento de uma nova abordagem ao CoR é o resultado da constatação das limitações do protótipo anterior, principalmente no que concerne às comunicações intra/inter domínios e ao modelo de execução adotado. Com efeito, a biblioteca spread é uma boa escolha para a comunicação em grupo, mas falta-lhe a capacidade para lidar diretamente com grandes volumes arbitrários de dados e o seu desempenho é significativamente reduzido quando comparado com outras bibliotecas de comunicação.

Por outro lado, a tendência atual é para substituir o modelo *thread-centric*, orientado ao fio de execução, pelo modelo *task-centric*, na expectativa de aumentar a escalabilidade, a eficiência e o desempenho.

Havia também muitas dependências de bibliotecas externas, o que obrigava a uma contínua e complexa atualização e compatibilização de pacotes, limitando a instalação e correspondente utilização da plataforma.

Durante o estudo prévio do HPX foi pensada a possibilidade de se usar como "estudo de caso" o PlaCoR, com vista a produzir uma versão melhorada daquela plataforma que ultrapassasse as limitações acima identificadas. É assim que surge o CoR-HPX, como uma nova etapa no processo de evolução do modelo CoR que tira partido

1 https://www.savarese.com/software/libssrcspread/
2 https://libevent.org/
3 https://www.gamasutra.com/blogs/RuiFigueira/20160608/274404/Modern_C_lightweight_binary_RPC_framework_without_code_generation.php
4 https://uscilab.github.io/cereal/
5 https://www.libssh.org

dos conceitos e *runtime* do HPX para criar uma nova plataforma, orientada à programação AMT, mas compatível com API anterior.

Atualmente o CoR-HPX é um protótipo, em C++ Moderno, de uma plataforma de computação paralela que oferece ao programador um modelo de programação híbrido com memória partilhada/distribuída. Pretende-se com este protótipo validar a utilização das entidades principais do modelo: os Recursos - unidades base de estruturação, de computação e de comunicação; e os mecanismos de comunicação inter-recurso - partilha de memória e de sincronização, em ambientes distribuídos.

As classes de Recursos disponíveis incluem os Recursos previstos pelo CoR (domínio, grupo, clausura, agente, proto-agente, dado, guarda, guarda de leitura/escrita e barreira), e também Recursos novos (operão, unichannel e multichannel).

4.3 FERRAMENTAS DE DESENVOLVIMENTO

A vantagem mais imediata na utilização do HPX resulta da possibilidade de, à exceção da biblioteca libssh, eliminar todas as bibliotecas externas, visto que integra direta ou indiretamente todas as respetivas facilidades, tais como: ambiente paralelo/distribuído, serialização, mensagens ativas e um *runtime* e modelo de programação assíncrono baseado em tarefas (*task-oriented*).

Na figura 11, pode ver-se que o lugar das bibliotecas externas é ocupado pelo HPX: i) a comunicação entre controladores e entre correios ficaram a cargo do componente AGAS. i) a execução de chamadas remotas é realizada através de ações; iii) a serialização é uma alternativa à biblioteca cereal; iv) o processamento de eventos assíncronos é intrínseco ao modelo; v) os fios de execução passam a tarefas (fios de execução leves HPX) e vi) a sincronização é resolvida pelos LCOs. Assim, é unicamente necessária a libssh para criar processos em tempo de execução e adicioná-los ao *runtime* dinamicamente.

A figura realça ainda o uso das bibliotecas dinâmicas como meio para a criação de módulos ELF gerados a partir da compilação de unidades de código C++ que podem ser posteriormente carregados em tempo de execução.

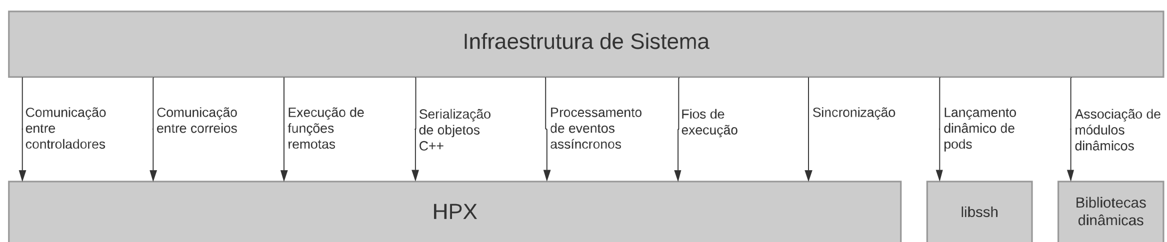


Figura 11: Ferramentas de desenvolvimento do protótipo CoR-HPX

4.4 ARQUITETURA DO SISTEMA

A figura 12 é uma representação geral da arquitetura do CoR-HPX. No topo, as aplicações CoR são desenvolvidas recorrendo à API dos Recursos que, por sua vez, usam a API do Pod. Na base, a infraestrutura é maioritariamente suportada pelo *runtime* HPX. Ainda, em cima, a ferramenta "corhpx" tira partido da API Pod para arrancar e executar as aplicações. Note-se que o Pod representa um processo, podendo haver mais do que um, como é natural num sistema de domínios distribuídos.

No centro destacam-se os objetos locais a cada Pod: o Controlador, o PageManager, o ResourceManager, e os clientes (idpManager_Cliente e ResourceManagerGlobal_Cliente), que interagem entre si. Os clientes, por sua vez, interatuam com os componentes globais (idpManager_Cliente e ResourceManagerGlobal_Cliente).

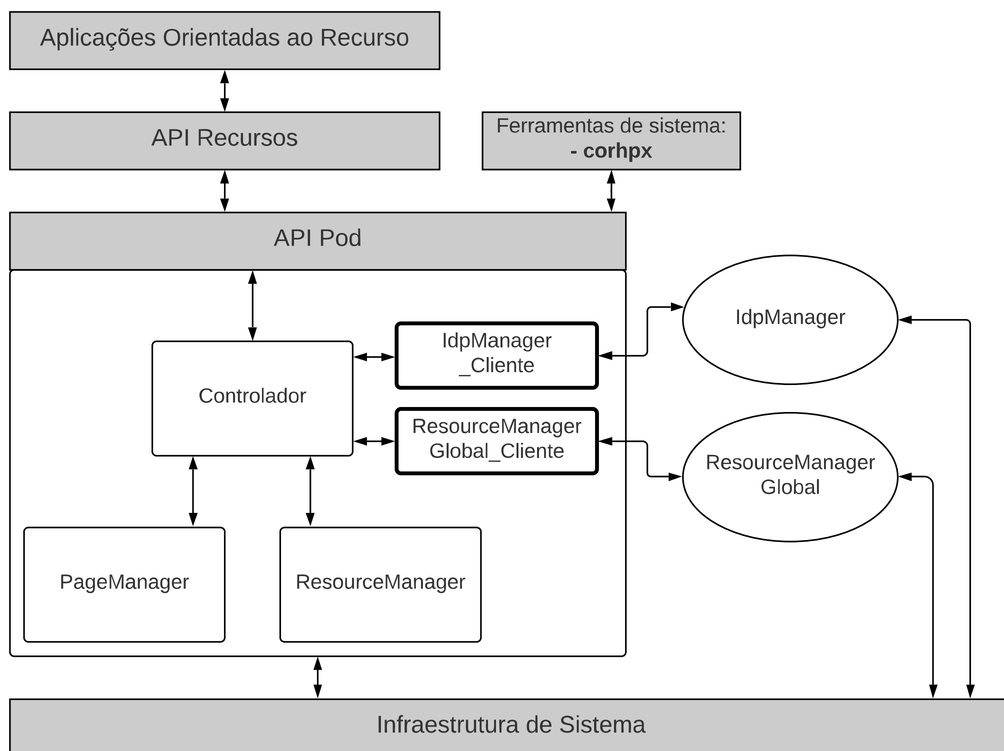


Figura 12: Arquitetura da Plataforma CoR-HPX

Componentes da arquitetura CoR-HPX:

- Pod
  O Pod, ao nível de implementação, é um componente HPX, pelo que toda a interação, local ou remota, pode ser realizada através de ações. As suas caraterísticas não são, atualmente, compatíveis com a possibilidade de ser um componente migrável, o que permitiria ser movido entre localidades (processos Unix). Nesta

entidade Pod coabitam os fios de execução do sistema e todas as tarefas associadas às instanciações dos Recursos CoR que herdam as propriedades do elemento executor.

- Controlador
  O controlador, ao contrário da plataforma anterior em que era a entidade responsável pela gestão de múltiplos serviços, serve maioritariamente de interface entre as entidades PageManager e ResourceManager e as entidades componentes HPX IdpManager e ResourceManagerGlobal

- Serviço de idps global - IdpManager
  O IdpManager é o serviço global responsável por gerar idps de forma consistente e centralizada, fornecendo conjuntos de novos idps (páginas) aos PageManager locais. Modelado como um componente HPX, é criado durante o arranque do primeiro Pod da aplicação. No arranque de uma aplicação paralela multi-domínio, o IdpManager (GID) é registado no IdpManager_Cliente de cada domínio, para que sempre que for necessário gerar um novo idp, seja obtido não um, mas um lote de novos identificadores, através de uma ação sobre o IdpManager global. Esta opção foi tomada com o intuito de reduzir a comunicação. A atomicidade desta operação é garantida, visto que este serviço é realizado através de uma ação HPX sobre um componente com a facilidade de `hpx::components::locking_hook`.

- Identificação local - PageManager
  No modelo CoR, a cada novo Recurso (ou uma das suas réplicas) corresponde um idp único. O processo de obtenção de um idp é iniciado no domínio pelo ResourceManagerGlobal_Cliente que faz o pedido ao PageManager local. No caso de não existirem identificadores disponíveis o pedido é direcionado para o Controlador que, por sua vez, através do idpManager_Cliente, reencaminha o pedido para o IdpManager global que retorna um novo lote de identificadores, do qual é retirado o idp pretendido. O PageManager serve como uma espécie de *cache* local a cada Pod para armazenar um conjunto de idps já calculados, com disponibilidade imediata.

- Gestão de Recursos global - ResourceManagerGlobal
  Tal como IdpManager, o ResourceManagerGlobal é um componente HPX criado pelo controlador do primeiro Pod do sistema. É usado para gestão e localização de Recursos ao nível global do sistema de domínios distribuídos.

- Gestão de Recursos local - ResourceManager
  Esta entidade diz respeito à gestão de todo o ciclo de vida dos Recursos. Em cada Pod existe uma instância desta classe que, visto de uma forma simplista, tem o papel de *cache* local do componente ResourceManagerGlobal. É através desta entidade que os Recursos são construídos, destruídos e geridos.

4.4.1 *Classes dos Recursos*

Em termos de implementação, a classe "Resource" é uma classe abstrata, superclasse de todas as categorias de Recursos definidos na plataforma, usada para introduzir polimorfismo uma vez que derivam dela todos os

tipos de Recursos. Na figura 13 (figura ampliada no anexo B) pode ver-se que a cada classe de Recursos está associada uma classe cliente e um ou mais elementos (caixas com fundo branco). Por exemplo, a classe Domain está associada à classe cliente Domain_Client e é composta pelos elementos Container e DynamicOrganizer, sendo este último elemento também constituinte da classe Group. Esta associação e composição dos Recursos corresponde a uma definição formal, ou assinatura, que inclui os elementos integrantes do Recurso e a respetiva classe cliente:

```
Domain :: Container . DynamicOrganizer | Domain_Client
```

Doravante, este tipo de definição será usado para mostrar que as propriedades de uma determinada classe de Recursos resulta na composição da sequência de classes integrantes, designadas por classes elemento, e da respetiva classe cliente.

Ainda na figura 13, em segundo plano, mostra-se a dependência e herança da infraestrutura HPX, que transforma as classes dos Recursos em componentes (caixas arredondadas). A herança comum representada pela classe "Resource"(e ResourceMigrable) assiste na criação de uma API unificada de Recursos que é posteriormente refinada em cada classe particular, de acordo com as características específicas dos respetivos elementos.

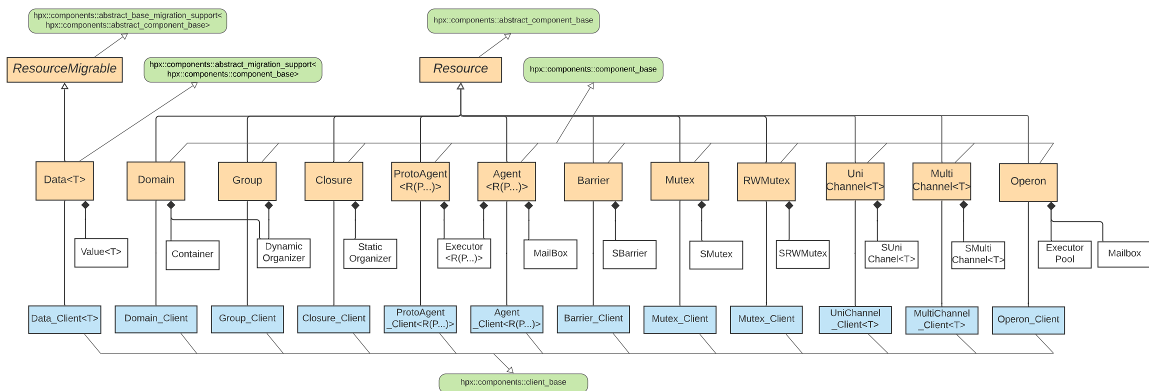


Figura 13: Diagrama de classes dos Recursos

Classe cliente

A criação e interação com os Recursos assenta na definição de uma classe cliente, em que cada instância cliente é uma camada de abstração que esconde o código necessário para criar e agir sobre as instâncias dos Recursos, local ou remotamente. Em tempo de execução, quando um cliente cria um Recurso, armazena o respetivo identificador global (GID) que é usado posteriormente para mediar todas as ações sobre o Recurso.

Réplica de um Recurso

Em CoR-HPX todos os Recursos foram desenhados como componentes HPX, pelo que, em geral, as interações com as respetivas instâncias são realizadas através de ações. Esta opção de desenho introduz um modelo de consistência baseado numa política de cópia única centralizada (2.5.1), oposta à da plataforma anterior baseada na existência de réplicas distribuídas. A política centralizada caracteriza-se pela existência de uma representação única de cada Recurso no sistema de domínios distribuídos, a mesma usada na definição e gestão dos componentes HPX.

Em CoR-HPX, uma réplica de um Recurso, em vez de ser uma cópia local do Recurso original, protegido por uma política de consistência distribuída, é uma instância de uma classe cliente, que representa o Recurso original da respetiva classe servidor (componente). Cada réplica é uma instância cliente que atua como procurador do Recurso original (componente) sem ter de conhecer a sua localização efetiva, recorrendo a ações sobre o GID (do componente) associado ao idp original. Os objetos cliente (réplicas) possuem estado próprio, o que é conciliável com a atribuição de um novo idp distinto do original, quando criados.

Convém referir que a política de cópia única centralizada é compatível com o desenho de classes de Recursos cujas instâncias podem ser movidas entre localidades (migrar), de forma transparente, sem necessidade de atualização das suas réplicas, uma vez que o GID original se preserva (tal como o respetivo idp).

4.5 RECURSOS

Nesta secção, são apresentados os Recursos disponibilizados em CoR-HPX, contendo a descrição e a sua constituição por elementos.

4.5.1 *Group*

O Recurso Group é constituído pelo elemento DynamicOrganizer, com interface Group_Client. Pertence à classe de Recursos estruturados, na medida em que organiza outros Recursos com base no elemento DynamicOrganizer. É um grupo dinâmico acessível através de operações de adesão (join) e abandono (leave). A identificação idp dos seus membros aderentes pode ser obtido quer por um número de ordem idm, quer por um nome.

- `Group :: DynamicOrganizer | Group_Client`

4.5.2 *Domain*

O Recurso Domain é constituído pelos elementos Container e DynamicOrganizer, com interface Domain_Client. Num sistema de domínios distribuídos, o Recurso Domain delimita uma região do espaço equivalente a um processo/localidade que suporta a representação, computação e comunicação entre Recursos locais ou remo-

tos. Na sua constituição, o elemento Container possui as características apropriadas para interação com o Pod enquanto que o elemento DynamicOrganizer lhe dá as caraterísticas já referidas para o Recurso Group.

- `Domain :: Container . DynamicOrganizer | Domain_Client`

4.5.3 *Closure*

O Recurso Closure é constituído pelo elemento StaticOrganizer, com interface Closure_Client. É um Recurso estruturado adequado para a realização de operações coletivas entre os seus membros, obrigatoriamente do tipo Agent.

- `Closure :: StaticOrganizer | Closure_Client`

4.5.4 *ProtoAgent*

O Recurso ProtoAgent é constituído pelo elemento Executor, com interface ProtoAgent_Client. É uma entidade cujo elemento constituinte Executor lhe dá a capacidade de executar uma função template registada no momento da sua criação.

- `ProtoAgent :: Executor | ProtoAgent_Client`

4.5.5 *Agent*

O Recurso Agent é constituído pelos elementos Executor e Mailbox, com interface Agent_Client. Partilha com o ProtoAgent a capacidade de executar uma função *template*, registada no momento da criação. O elemento adicional Mailbox transforma-o numa entidade comunicante capaz de receber/enviar mensagens através de um protocolo de comunicação ponto-a-ponto entre Recursos.

- `Agent :: Executor . Mailbox | Agent_Client`

4.5.6 *Barrier*

O Recurso Barrier é constituído pelo elemento Sbarrier, com interface Barrier_Client. Usa as propriedades do elemento SBarrier para atuar como coordenador das atividades paralelas de conjuntos estáticos (clausuras) de agentes/proto-agentes.

- `Barrier :: Sbarrier | Barrier_Client`

4.5.7 *Mutex*

O Recurso Mutex é constituído pelo elemento Smutex, com interface Mutex_Client. É um Recurso que serve de instrumento de sincronização distribuído, usados para definir zonas de exclusão mútua, entre agentes/proto-agentes.

- `Mutex:: Smutex | Mutex_Client`

4.5.8 *RWMutex*

O Recurso RWMutex é constituído pelo elemento RWSMutex, com interface RWMutex_Client. É um Recurso análogo ao Mutex, com a particularidade de definir zonas de exclusão mútua de leitura/escrita.

- `RWMutex :: RWSMutex | RWMutex_Client`

4.5.9 *Data*

O Recurso Data é constituído pelo elemento Value, com interface Data_Client. Representa um tipo de dados arbitrário com acesso remoto. Pode ser simplesmente um tipo nativo ou uma classe mais complexa (com métodos/herança/*template*). Ao contrário dos restantes Recursos, o Recurso Data é um caso particular pelo facto de poder ser movido entre domínios/localidades.

Em PlaCoR (protótipo anterior), o Dado foi implementado segundo o modelo de representação distribuída, o que quer dizer que para aceder à mesma instância de um Recurso Dado através de domínios distribuídos, essa instância tinha de estar replicada em cada um desses domínios. Para aceder ao Dado, este disponibilizava a função Get() que retornava um apontador para a posição de memória do objeto. Para garantir que o acesso era serializado, foi munido com as funcionalidades de *acquire/release*. Quando se executava *acquire()*, o Recurso ficava protegido contra acessos concorrentes. Assim que fosse libertado, com *release()*, as réplicas sincronizavam para atualizar o estado.

Em CoR-HPX (protótipo atual), o Dado, tal como todos os Recursos, foi implementado segundo o modelo de representação centralizado. Desta forma, há apenas uma única instância do Recurso no sistema de domínios distribuídos, modelado por um componente HPX. Esta mudança de paradigma, que levantou novas questões de desenvolvimento, levou a que o Dado fosse implementado tendo em vista dois objetivos: i) possuir uma API compatível com a anterior e ii) aproveitar o facto de ser um componente HPX para suportar a execução remota de funções.

Quando se afirma que a API do Dado tem de ser compatível com o modelo anterior, está em causa a função Get(). Para executar Get() e aceder à posição de memória do objeto, este tem de estar na localidade HPX de onde Get() é chamado. Se essa função for executada em duas localidades diferentes, e uma vez que o modelo é centralizado, não resta alternativa, o objeto tem de ser movido de uma localidade para a outra, de forma a

responder aos dois pedidos. Esta imposição levou a que o Recurso Dado, ao contrário dos restantes Recursos, fosse provido com a capacidade de migrar.

Uma vez que, no protótipo anterior PlaCoR, o Dado só poderia ser acedido através da sua posição de memória, agora, em CoR-HPX, ao ser um componente HPX, faz todo o sentido que também possa ser acedido remotamente, através de ações. Assim, foram adicionadas duas novas funções à API do Dado, na forma de ações: Fetch() e Run().

A função Fetch() retorna uma cópia do objeto que está albergado pelo Dado. A cópia retornada é um novo objeto independente. Pode ser executada numa localidade diferente de onde o Dado se encontra, não obrigando o Recurso a migrar. Contudo, é uma função exclusivamente de leitura, que não altera o estado do Dado.

A função Run() surgiu da necessidade de aceder ao Dado remotamente para escrever, sem ter de usar a função Get() que obriga a que o Dado esteja local, levando-o a migrar se estiver numa localidade diferente. A função Run() serve para executar quaisquer funções membro do objeto albergado pelo Dado, tanto local como remotamente.

Desta forma, o CoR-HPX, através do Recurso Dado, disponibiliza um meio para que qualquer objeto possa ser acedido local e/ou remotamente. Comparando o Dado com um componente HPX, o Dado exige menos esforço a construir e oferece mais funcionalidades por omissão, nomeadamente a de migração. Simplifica ainda a escrita do código de objetos complicados como, por exemplo, classes de componentes com hierarquia, que exige conhecimentos avançados que estão indisponíveis na documentação do HPX. No caso do Dado, o utilizador só necessita de o criar com o tipo que pretende e toda a implementação obscura dos componentes HPX fica omissa. Naturalmente, a migração inter-domínios de objetos, não serializáveis automaticamente pelo HPX, obriga a incluir o código de serialização através de uma função membro "serialize" para referir o que se pretende serializar. Para concretizar o tipo do Recurso é também necessário usar a macro `REGISTER_DATA()` no espaço global do código do utilizador com o tipo de dados.

- `Data :: Value | Data_Client`

4.5.10 *Operon*

Recurso Operon é constituído pelos elementos MailBox e ExecutorPool (conjunto de tarefas), com interface Operon_Client. Possui capacidades de computação e comunicação equivalentes às do Recurso Agent. A diferença reside na possibilidade de registar a todo momento uma nova função cuja execução é processada em paralelo por um coletivo de tarefas, tal como se de um "omp parallel" se tratasse. O operão também disponibiliza funcionalidades para paralelizar ciclos, como "omp parallel for", com escalonamento *estático*, *dinâmico* e *guided*.

- `Operon :: ExecutorPool . MailBox | Operon_Client`

4.5.11 *UniChannel*

O Recurso UniChannel é constituído pelo elemento SUniChannel, com interface UniChannel_Client. Surge da necessidade do modelo ter um mecanismo de troca de mensagens independente entre dois pontos, implementado, naturalmente, com um LCO hpx::channel. Este Recurso chama-se UniChannel porque está implementado com um só canal para comunicação, mas existe liberdade para troca mensagens nas duas direções. Contudo, recomenda-se a sua utilização para o envio de dados numa só direção, uma vez que pode não ficar clara a ordem das mensagens quando usado nos dois sentidos. Para comunicação bilateral, ver o Recurso MultiChannel.

- `UniChannel :: SUniChannel | UniChannel_Client`

4.5.12 *MultiChannel*

O Recurso MultiChannel é constituído pelo elemento SMultiChannel, com interface MultiChannel_Client. Expande o Recurso UniChannel para troca de mensagens entre vários interlocutores. Simplifica a programação no desenvolvimento de algoritmos em que seja necessário trocar dados entre vários agentes, contendo tantos canais quanto se deseje. Cria, internamente, dois canais de comunicação entre cada parceiro tornado possível a troca de dados entre eles em ambas as direções sem conflitos.

- `MultiChannel :: SMultiChannel | MultiChannel_Client`

4.6 ELEMENTOS

Em CoR-HPX a criação de Recursos obedece a um conjunto simples de regras modulares acessível ao programador, que permite combinar classes de elementos novas e/ou pré-existentes para, através da definição de novas classes clientes, acrescentar novas funcionalidades à plataforma. Por exemplo, no âmbito desta dissertação foram criados novas classes de elementos, ExecutorPool, SUniChannel e SMultiChannel, que estão na base das novas classes de Recursos Operon, UniChannel e MultiChannel, respetivamente.

No que se segue, apresentam-se as funções das classes de elementos do CoR-HPX. Estas funções são encapsuladas em ações nas classes dos Recursos a que fazem parte, e são mediadas posteriormente pelos componentes cliente que servem de interface e constituem a API final dos Recursos (anexo D).

4.6.1 *Contentor*

O contentor estabelece uma relação direta com as facilidades disponibilizadas pela infraestrutura, através do Pod. Em particular, permite: i) obter o contexto/arranque da aplicação e os identificadores/apontadores dos Recursos no domínio local, ii) criar dinamicamente Recursos/réplicas local/remotamente e iv) arrancar dinamicamente novos pods/domínios.

class Container

- **Container**(idp_t idp): construtor de um elemento contentor no contexto identificado por idp.
- std::string **GetGlobalContext**(): retorna o nome do contexto global da aplicação.
- std::string **GetLocalContext**(): retorna o nome do contexto de arranque do domínio local.
- unsigned int **GetTotalPods**(): retorna o número total de pods presentes na aplicação.
- unsigned int **GetTotalDomains**(): retorna o número total de domínios na aplicação.
- idp_t **GetActiveResourceIdp**(size_t id): retorna o identificador do Recurso ativo, identificado por id.
- idp_t **GetPredecessorIdp**(idp_t idp): retorna o ascendente do Recurso identificado por idp.
- template <typename T>
  std::unique_ptr<T> **GetLocalResource**(idp_t idp): retorna um apontador para o objeto cliente do tipo T, do Recurso identificador por idp.
- template <typename T, typename ... Args>
  std::unique_ptr<T> **CreateLocal**(idp_t ctx, std::string const& name, Args... args): cria no domínio local e no contexto ctx, um Recurso do tipo T com nome name e retorna um apontador para o respetivo cliente referente ao Recurso criado. O construtor do tipo T recebe uma lista de argumentos em args.
- template <typename T, typename ... Args>
  idp_t **CreateRemote**(idp_t ctx, std::string const& name, Args... args): cria um Recurso do tipo T no contexto identificado por ctx, num domínio remoto, com nome name e retorna o identificador do Recurso criado.
- template <typename T, typename ... Args>
  idp_t **Create**(idp_t ctx, std::string const& name, Args... args): cria um Recurso do tipo T no contexto identificado por ctx, num domínio local/remoto, com nome name e retorna o identificador do Recurso criado. O construtor do tipo T recebe uma lista de argumentos em args.
- template <typename T>
  std::unique_ptr<T> **CreateReference**(idp_t idp, idp_t ctx, std::string const& name): cria uma réplica no domínio local, no contexto identificado por ctx, do Recurso com tipo T identificado por idp e com o nome name.
- template <typename T, typename ... Args>
  std::unique_ptr<T> **CreateCollective**(idp_t ctx, std::string const& name, unsigned int total_members, Args...args): equivalente ao Create, chamado de forma coletiva por um total de total_members Recursos ativos (com elemento executor). O Recurso é criado na domínio do primeiro executor a chamar esta função, obtendo todos os participantes uma referência local para o cliente.
- template <typename T, typename ... Args>
  std::unique_ptr<T> **CreateCollective**(idp_t active_rsc_idp, idp_t clos,

`idp_t ctx, std::string const& name, Args... args)`: equivalente ao Create chamado de forma coletiva por todos os membros do Recurso identificado por clos, na qual todos os participantes terão uma referência local do Recurso. O Recurso é criado na domínio do primeiro executor a chamar esta função, obtendo todos os participantes uma referência local para o cliente.

- `template <typename T, typename ... Args>`
  `auto` **`Run`**`(idp_t idp, Args... args)`: usado para correr, local ou remotamente, a função com tipo T do elemento executor do Recurso identificado por idp. Os argumentos args que têm de coincidir com a assinatura da função. Este método é assíncrono e retorna um futuro com o valor de retorno da função.
- `idp_t` **`Spawn`**`(std::string const& ctx, unsigned int npods, idp_t parent, std::string const& mod, std::vector<std::string> const& args, std::vector<std::string> const& hosts)`: usado para lançar dinamicamente um total de npods novos pods/domínios no contexto de arranque ctx, nas máquinas em hosts utilizando um algoritmo de escalonamento round-robin. O módulo mod, carregado dinamicamente, contém a função de entrada do módulo que irá ser executada com os argumentos em args. Para a identificação das máquinas em hosts pode ser usado um endereço IP ou nome DNS.

4.6.2 *Organizador*

O elemento organizador cria as condições para o estabelecimento de árvores de dependências, que agregam Recursos, de alguma forma relacionados, atribuindo-lhes identificadores de membro locais (idm) e um nome no contexto de um outro Recurso, designado por Recurso ascendente. Na árvore de dependências, cada um dos Recursos que possuem um elemento do tipo organizador dá origem a uma nova subárvore.

Este elemento surge em duas versões, muito semelhantes ao nível do interface: dinâmico (DynamicOrganizer) e estático (StaticOrganizer). O primeiro permite a adesão/saída dinâmica de Recursos a qualquer momento, enquanto que o segundo é criado de uma só assentada, com um conjunto inicial fechado de Recursos membros.

A existência de organizadores estáticos está intrinsecamente ligada à necessidade de garantir a constância de um grupo, com uma aridade que se mantém fixa ao longo do tempo de vida do Recurso, condição fundamental para a realização de operações coletivas; por exemplo, durante o arranque de uma aplicação paralela que obriga à instanciação simultânea de múltiplos domínios, ou de comunicação em grupo fechado.

Organizador Dinâmico

O construtor deste tipo de elementos inclui um parâmetro (opcional), como um caminho no sistema de ficheiros para um módulo de biblioteca dinâmica. O módulo (se existir) é carregado em tempo de execução e disponibiliza um conjunto de funções que podem ser evocadas por Recursos ativos, i.e. com elemento executor.

```
class DynamicOrganizer
```

- **DynamicOrganizer**(idp_t idp, std::string const& module): construtor que cria no Recurso, com identificador idp, um elemento organizador dinâmico e, eventualmente, carrega a biblioteca dinâmica module que integra na aplicação.
- void **Join**(idp_t idp, std::string const& name): junta o Recurso identificado por idp ao organizador dinâmico, ao qual é atribuído o nome name e um identificador local (idm) no organizador.
- void **Leave**(idp_t idp): dissocia do organizador o Recurso identificado por idp.
- std::string const& **GetModuleName**() const: retorna nome do módulo no organizador.

Organizador Estático

O organizador estático inclui informação específica que permite distinguir se foi criado no contexto inicial de arranque da aplicação, parent==0, ou em tempo de execução e, neste caso, parent toma o valor do idp do Recurso ativo responsável pela criação do próprio Recurso, como é o caso do lançamento dinâmico de novos domínios.

```
class StaticOrganizer
```

- **StaticOrganizer**(idp_t idp, unsigned int members, idp_t parent): construtor que cria no Recurso, com identificador idp, um elemento organizador com o total de members membros e o identificador do Recurso parent que chamou a função.
- void **Join**(idp_t idp, std::string const& name): associa o Recurso ativo identificado por idp ao organizador estático atribuindo o nome name e um identificador idm local. É uma operação coletiva que deverá ser chamada um total de members vezes por diferentes Recursos ativos.
- void **Leave**(idp_t idp): dissocia o Recurso identificado por idp do organizador correspondente. Esta operação é coletiva, pelo que deverá ser chamada um total de members vezes por diferentes Recursos ativos.
- idp_t **GetParent**() const: usado para obter o identificador principal do Recurso que criou o Recurso deste organizador.

Métodos comuns

De seguida, são apresentados os métodos comuns a ambos os elementos organizador (dinâmico e estático).

- std::size_t **GetTotalMembers**() const: retorna o número total de membros do organizador correspondente.
- std::vector<idp_t> **GetMemberList**() const: retorna a lista dos identificadores principais dos Recursos que fazem parte do organizador correspondente.
- idp_t **GetIdp**(idm_t idm) const: retorna o identificador principal do Recurso com base no identificador de membro idm.
- idp_t **GetIdp**(std::string const& name) const: retorna o identificador principal do Recurso com base no nome name.

- `idm_t` **`GetIdm`**`(idp_t idp) const`: retorna o identificador de membro do Recurso com base no identificador principal idp.
- `idm_t` **`GetIdm`**`(std::string const& name) const`: retorna o identificador de membro do Recurso com base no nome name.

4.6.3 *Executor*

O elemento executor corresponde à definição de uma função template que será executada por um fio de execução HPX de acordo com a respetiva assinatura (que especifica os tipos de dados do retorno e dos parâmetros de invocação). Existem duas possibilidades para criar um executor: ou diretamente com o nome da função, ou acrescentando o nome do módulo ao nome da função.

```
template <typename R, typename ... P>
class Executor<R(P...)>
```

- **`Executor`**`(idp_t idp, std::string const& module, std::string const& function)`: cria o elemento executor do Recurso com identificador idp que irá carregar e executar uma função com nome function da biblioteca dinâmica module e assinatura `R(P...)`.
- **`Executor`**`(idp_t idp, std::function<R(P...)> const& f)`: cria o elemento executor do Recurso com identificador idp, que irá executar a função f recebida como parâmetro.
- `template <typename ... Args>`
  `hpx::future<R>` **`Run`**`(Args&&... args)`: executa a função template registada no executor os args correspondentes à respetiva assinatura. Retorna, imediatamente, um futuro de tipo R com o valor de retorno.
- `void` **`ChangeIdp`**`(idp_t idp)`: troca a identidade do executor para idp.
- `void` **`ResumeIdp`**`()`: restaura a identidade original do executor.
- `idp_t` **`CurrentIdp`**`() const`: retorna o identificador atual do Recurso.
- `idp_t` **`OriginalIdp`**`() const`: retorna o identificador do Recurso na origem da criação do executor.

4.6.4 *Caixa-Postal*

O elemento Caixa-Postal é usado para o envio/receção de mensagens entre Recursos que possuam este elemento. Tem a possibilidade do envio de mensagens para um ou mais destinos, desde que os seus identificadores sejam conhecidos. Também possui primitivas de comunicação por contexto, que permitem difundir mensagens para um Recurso com um elemento organizador estático, através do idp do mesmo, ou então o envio/receção de mensagens utilizando o idp do Recurso clausura e o idm no contexto do mesmo. Atualmente, as primitivas de comunicação disponibilizadas são bloqueantes.

```
class Mailbox
```

- **Mailbox**(idp_t idp): construtor de um elemento Caixa-Postal a integrar no Recurso idp.
- void **Send**(idp_t dest, Message const& msg) const: utilizado para enviar a mensagem msg para o Recurso dest.
- void **Send**(std::vector<idp_t> const& dests, Message const& msg) const: envia a mensagem msg para uma lista de destinatários dests.
- Message **Receive**() const: utilizado para receber uma mensagem da Caixa-Postal do Recurso.
- Message **Receive**(idp_t source) const: recebe uma mensagem com origem em source.
- void **Broadcast**(idp_t clos, Message const& msg) const: difunde uma mensagem msg para todos os membros do Recurso clos com elemento organizador estático.
- void **Send**(idm_t rank, idp_t clos, Message const& msg) const: envia uma mensagem msg para o Recurso identificado por rank, no contexto do Recurso clos com elemento organizador estático.
- Message **Receive**(idm_t rank, idp_t clos) const: recebe uma mensagem enviada pelo Recurso identificado por rank, no contexto do Recurso com elemento organizador estático clos.

4.6.5 *Classe auxiliar Mensagem*

A classe mensagem é uma classe auxiliar do elemento Caixa-Postal, cujas instâncias incluem um cabeçalho, formado por uma etiqueta e o idp do Recurso na origem, e um tampão de memória para os dados. É possível adicionar e obter dados de uma mensagem, desde que os respetivos tipos sejam serializáveis pelo HPX.

```
class Message
```

- **Message**(): construtor padrão de uma mensagem.
- std::size_t **Size**() const: retorna o tamanho da mensagem.
- void **Clear**(): apaga o conteúdo da mensagem.
- std::uint16_t **Type**() const: retorna o tipo da mensagem.
- void **SetType**(std::uint16_t type): modifica o tipo (etiqueta) da mensagem.
- idp_t **Sender**() const: retorna a identificação do Recurso que enviou a mensagem.
- void **SetSender**(idp_t): insere o identificador do emissor na mensagem.
- template <typename T>
  T **Get**(std::size_t index = 0) const: obtém do tampão dos dados da mensagem, o conteúdo do tipo T localizado na posição index.
- template <typename T>
  void **Add**(T const& data): adiciona ao tampão dos dados da mensagem o conteúdo da variável data do tipo T.

4.6.6 *Valor*

O valor é uma classe template usada para albergar uma instância parametrizada de um determinado tipo de dados. O programador pode criar as suas próprias classes de dados, desde que, tal como no caso da mensagem, sejam serializáveis.

```
template <typename T>
class Value
```

- `Value(idp_t idp, Args... args)`: construtor que cria o elemento valor do tipo `T` no Recurso com identificador idp, com o estado inicial especificado através de args.
- `T* Get()`: retorna um apontador para os dados do utilizador do tipo T no elemento valor. Depois de retornado o apontador, não é assegurada a consistência dos dados, sendo recomendável aquisição do direito de escrita, disponibilizado pela API do Dado.
- `T Fetch() const`: retorna uma cópia dos dados do utilizador do tipo `T` albergado pelo elemento valor. Para assegurar a consistência dos dados, enquanto a função é chamada através da API do Dado, o Recurso não permite escritas, unicamente leituras.

4.6.7 *Sincronizador*

O elemento sincronizador proposto em CoRes, desdobra-se nas variantes barreira (SBarrier), guarda (SMutex) e guarda para leituras/escritas (SRWMutex), que atuam como estruturas de sincronização distribuídas.

Barreira

A barreira, tal como o nome indica, fornece a capacidade de sincronização distribuída do conjunto dos membros de um organizador estático.

```
class SBarrier
```

- `SBarrier(idp_t idp, idp_t clos)`: construtor do elemento barreira do Recurso identificado por idp, usado como estrutura de sincronização distribuída dos Recursos ativos associados ao organizador estático clos.
- `void Synchronize()`: operação coletiva que só termina quando invocada por todo os Recursos ativos presentes no organizador estático implícito.

Guarda

A guarda serve para criar uma região de código de exclusão mútua.

```
class SMutex
```

- **SMutex**(idp_t idp): construtor da guarda do Recurso identificado por idp.
- void **Acquire**(): sinaliza a entrada numa região de exclusão mútua.
- void **Release**(): sinaliza a saída de uma região de exclusão mútua

Guarda de leitura/escrita

A guarda de leitura/escrita serve para criar uma região de exclusão mútua distinguindo as leituras das escritas.

```
class SRWMutex
```

- **SRWMutex**(idp_t idp): construtor da guarda do Recurso identificado por idp.
- void **AcquireRead**(): sinaliza a entrada numa região de exclusão mútua para leituras.
- void **ReleaseRead**(): sinaliza a saída de uma região de exclusão mútua para leituras.
- void **AcquireWrite**(): sinaliza a entrada de uma região de exclusão mútua para escritas.
- void **ReleaseWrite**(): sinaliza a saída de uma região de exclusão mútua para escritas.

4.6.8 *ExecutorPool*

O elemento ExecutorPool possui capacidades de computação, similarmente ao elemento Executor do Recurso ProtoAgent, mas com a capacidade de registar a todo momento uma nova função cuja execução é processada em paralelo por um coletivo de tarefas, como se de um "omp parallel"se tratasse. Disponibiliza funcionalidades para paralelizar ciclos, de forma análoga ao "omp parallel for", com escalonamento *estático*, *dinâmico* e *guided*.

```
class ExecutorPool
```

- **ExecutorPool**(idp_t idp, std::size_t num_hpx_threads) - Construtor do elemento ExecutorPool do Recurso identificado por idp. Recebe como argumentos o número de tarefas que irão ficar alocadas.
- int **GetRank**() - rank da tarefa.
- int **GetNumThreads**() - numero total de tarefas do alocadas.
- std::pair<int,int> **ScheduleStatic**(int Beg, int End) - método de escalonamento estático para processamento paralelo de ciclos. Fornecendo os índices inicial e final da gama de iterações de um ciclo, retorna um par com os índices inicial e final atribuídos à tarefa correspondente.
- std::vector<std::pair<int,int> > **ScheduleStatic**(int Beg, int End, int chunk) - método de escalonamento estático com *chunk* para processamento paralelo de ciclos. Fornecendo os índices inicial e final da gama de iterações de um ciclo, retorna um vetor de pares com os índices inicial e final atribuídos à tarefa correspondente.
- std::pair<int,int> **ScheduleDynamic**(int Beg, int End, int chunk) - método de escalonamento dinâmico com *chunk* para processamento paralelo de ciclos. Iterativamente, chamando esta função fornecendo os índices inicial e final da gama de iterações de um ciclo, retorna um par com os índices inicial e final atribuídos à tarefa correspondente.

- std::pair<int,int> **ScheduleGuided**(int Beg, int End, int chunk) - método de escalonamento *guided* com *chunk* para processamento paralelo de ciclos. Iterativamente, chamando esta função fornecendo os índices inicial e final da gama de iterações de um ciclo, retorna um par com os índices inicial e final atribuídos à tarefa correspondente.
- template < typename ... Args >
  void **Dispatch**(hpx::function<void(Args...)> func, Args ... args) - Executa a função func com os argumentos args em paralelo pelas tarefas constituintes do ExecutorPool.
- void **Wait**() - função de sincronização na barreira interna de ExecutorPool.

4.6.9 *SUniChannel*

Elemento de comunicação para troca de dados entre dois pontos. Chama-se "uni" porque é construído com um só LCO hpx::channel; contudo, os dados podem ser enviados nas duas direções. Para a sua utilização, na troca de dados entre dois agentes, ambos os parceiros têm de usar o mesmo Recurso que contém SUniChannel - um cria o Recurso e o outro uma réplica.

class SUniChannel

- **SUniChannel**(idp_t idp): construtor do SUniChannel do Recurso identificado por idp.
- void **Set**(T&& t, std::size_t step): envia uma mensagem com tipo T, com índice "step", de forma assincrona.
- hpx::future<T> **Get**(std::size_t step): recebe uma mensagem com tipo T, com índice "step".

4.6.10 *SMultiChannel*

Expande o elemento SUniChannel para troca de dados entre vários interlocutores. Simplifica a construção de algoritmos em que seja necessário trocar dados entre vários agentes. Cada Recurso pode conter tantos canais quanto deseje. Cria, internamente, dois canais de comunicação hpx::channel entre cada parceiro, tornado possível a troca de dados em ambas as direções, sem conflitos. Cada interlocutor, para comunicar com os restantes, tem de criar um novo Recurso que contenha SMultiChannel. Os canais conectam-se através dos nomes fornecidos no construtor.

class SMultiChannel

- template <typename ...Args>
  SMultiChannel(idp_t idp, std::string const& myself, Args ... args): construtor do SMultiChannel do Recurso identificado por idp, que é identificado também por "myself"para registo dos canais para comunicação com cada um dos parceiros recebidos em args.

- void **Set**(T&& t, std::string const& partner, std::size_t step): envia uma mensagem com tipo T, com índice "step" (etiqueta), para o parceiro identificado por "partner".
- hpx::future<T> **Get**(std::string const& partner, std::size_t step): recebe uma mensagem com tipo T, com índice "step" (etiqueta), do parceiro "partner".

5

CODIFICAÇÃO E EXECUÇÃO DE PROGRAMAS

Este capítulo pretende, numa primeira fase, explicar de forma sucinta como criar e executar programas em CoR-HPX. A explicação detalhada encontra-se na tese de mestrado PlaCoR Ribeiro (2019), que também é compatível. Posteriormente, é apresentado um exemplo com o Recurso Data, análogo ao exemplo Queue em 3.3.4, mostrando as suas funcionalidades e vantagens. Por fim é apresentado um algoritmo *stencil* construído em CoR-HPX, ilustrando a utilização de Recursos.

5.1 INICIALIZAÇÃO DE UM PROGRAMA

Para executar programa escritos em CoR-HPX é usado o comando de linha `corhpx` com uma sequência de argumentos que obedece ao formato:

```
corhpx <app group> <context> <number pods> <parent> <module> <args...>
```

Os três argumentos seguintes são definições gerais relativas ao contexto de execução:

- `app group` - contexto da aplicação que engloba todos os pods da aplicação;
- `context` - contexto de arranque que inclui exclusivamente os pods (um número dado por `number pods`) que arrancaram simultaneamente da linha de comando;
- `parent` - idp do Recurso na origem da criação dos pods (por omissão, `0`).

Os restantes argumentos estão relacionados com o número de réplicas (processos) do código, o nome da biblioteca dinâmica e os respetivos argumentos:

- `number pods` - quantidade de pods (processos) a arrancar, em paralelo;
- `module` - nome do módulo (biblioteca `.so`) com o código do programa;
- `args...` - argumentos passados à função Main do módulo.

5.2 ESTRUTURA DE UM PROGRAMA

O código da listagem 29 ilustra um exemplo básico de um programa CoR-HPX:

```
#include "cor/cor.hpp"

extern "C"
{
    void Main(int argc, char *argv[]);
}

void Main(int argc, char *argv[]) {
    auto domain = cor::GetDomain();
    std::cout << domain->GetActiveResourceIdp() << std::endl;
    std::cout << domain->GetActiveResourceIdp(hpx::launch::async).get() << std::endl;
}
```

Listing 29: Módulo de arranque de uma aplicação básica

O comando `corhpx`, usado para arrancar programas CoR-HPX, carrega a biblioteca dinâmica, referida na linha de comandos, cujo código contém as seguintes partes:

- `#include "cor/cor.hpp"`
- `extern "C"`
- `void Main(int argc, char *argv[]);`

Relativamente ao código da listagem 29, primeiramente é chamado `cor::GetDomain()` que retorna o domínio local e fornece uma primeira porta para interagir com a plataforma. De seguida é executada uma função essencial da API, `GetActiveResourceIdp()`, chamada de duas formas, síncrona e assíncrona (linhas 10 e 11), respetivamente, retornando o idp do agente ativo que a executa.

Para o arranque da aplicação foi utilizado o comando `corhpx app ctx 1 0 libmodule.so`, obtendo-se como resultado[1]:

```
4294967038
4294967038
```

5.3 TROCA DE MENSAGENS ENTRE DOMÍNIOS DISTRIBUÍDOS

O código da listagem 30 serve para mostrar uma forma de trocar de mensagens entre dois agentes da mesma clausura. Os agentes, apesar de estarem na mesma clausura, pertencem a domínios diferentes. O agente com *rank* `0` envia uma mensagem para o agente com rank `1`, o qual imprime o seu conteúdo.

1 corhpx e libmodule.so têm os seus caminhos inseridos em, respetivamente, $PATH e $LD_LIBRARY_PATH; caso contrário é necessário fornecer esses caminhos explicitamente.

```
1  typedef std::vector<std::string> MsgType;
2  void Main(int argc, char *argv[])
3  {
4      auto domain = cor::GetDomain();
5      auto agent_idp = domain->GetActiveResourceIdp();
6      auto agent = domain->GetLocalResource<cor::Agent_Client<void(char**)>>(agent_idp);
7      auto clos_idp = domain->GetPredecessorIdp(agent_idp);
8      auto clos = domain->GetLocalResource<cor::Closure_Client>(clos_idp);
9      auto rank = clos->GetIdm(agent_idp);
10     if(clos->GetTotalMembers() != 2) return;
11
12     // rank 0 envia a menagem para o rank 1
13     if(rank==0) {
14         cor::Message msg;
15         MsgType vec = {"Olá", "mundo!"};
16         msg.Add<MsgType>(vec);
17         auto other_rank = clos->GetIdp(1);
18         agent->Send(other_rank, msg);
19     }
20     // o rank 1 recebe a mensagem
21     else {
22         auto msg = agent->Receive();
23         MsgType vec = msg.Get<MsgType>();
24         for(auto &str : vec) {
25             std::cout << str << " ";
26         }
27         std::cout << ", recebido de " << msg.Sender() << std::endl;
28     }
29 }
```

Listing 30: Exemplo send_an_object.cpp

Primeiro é obtido um apontador para o Domínio local (linha 4); depois é obtido o idp do Agente local que está a correr o código (linha 5) e um apontador para ele (linha 6); a seguir é obtido o idp da clausura (linha 7), que é o predecessor do Agente, e um apontador para ela (linha 8); logo após é obtido o rank - idm que identifica o Agente no seu contexto ascendente (linha 9); é entretanto testada (linha 10) uma condição para o programa ser executado unicamente se existirem dois Agentes na clausura (1 Agente de cada Domínio). Até aqui todo o código é executado nos dois domínios pelos seus agentes.

O Agente com rank `0` cria uma mensagem (um vetor de *strings*), e envia-a para o Agente com rank `1`. Este último espera sincronamente por uma mensagem e quando a receber imprime o seu conteúdo.

Se se quiser que este programa seja executado por mais do que dois processos e que todos eles recebam a mensagem enviada pelo rank `0`, pode-se usar a primitiva `Broadcast` ao invés de `Send` (`Broadcast` envia uma mensagem para todos os agentes da mesma clausura - ver API do Agente).

Para executar o programa da listagem 30 com dois processos, através do mpi, executa-se:

```
$ mpirun -np 2 corhpx apps ctx 2 0 libsend_an_object_simple.so
```

```
Olá mundo! , received from 4294967038
```

5.4 EXECUTAR FUNÇÕES GLOBAIS REMOTAMENTE

As classes de Recursos que possuem o elemento executor, tais como o ProtoAgent e o Agent, para além das propriedades do modelo CoR , são o equivalente natural das ações globais em HPX. Pretende-se mostrar com o exemplo da listagem 31, a execução remota de funções através do Recurso ProtoAgent. O exemplo foi criado para ser executado com dois domínios (por exemplo, Domínio A e Domínio B), em que os agentes de ambos irão executar uma função no Domínio remoto (o Agente do Domínio A irá executar código no Domínio B e o Agente do Domínio B irá executar código no Domínio A).

Nas linhas 1 a 8 é definida a função `operator()`, membro da estrutura `function_object` que irá encapsular a ação (código a ser processado remotamente). A ação é implicitamente produzida aquando da instância da classe ProtoAgent (linhas 24 e 25) através da função `domain->Create()`, esta recebe no primeiro argumento o Domínio remoto de destino, no segundo o nome da nova instância de ProtoAgente e no terceiro a função a executar. Notar que se a criação do ProtoAgente fosse local, usando `domain->CreateLocal()`, poder-se-ia ter usado uma definição convencional da função que representa a ação, em vez de uma função objeto. Posteriormente, a chamada `domain->Run()` (linhas 28, 29, 32 e 33) efetua a chamada da ação, no Domínio remoto `remote_domain_id`, passando como primeiro argumento o idp do Agente em questão e como segundo argumento o argumento da função (`domain_id`).

A execução em paralelo de duas instâncias de corhpx com o módulo libcreate_remote.so, produz o resultado visível a seguir[2]:

```
$ mpirun -n 2 corhpx ctx app 2 0 libcreate_remote.so --hpx:ini=hpx.component_paths=<path>
Function spawned from 4294967040 executed on domain 4294966784
Function spawned from 4294966784 executed on domain 4294967040
```

5.5 CASO DE USO QUEUE UTILIZANDO O RECURSO DADO

O intuito desta secção é mostrar a criação e utilização do Recurso Dado, com o tipo de dados Queue, funcionalmente equivalente ao desenho de um componente *template*, migrável e com herança, que foi apresentado na secção 3.3.4 (que contém a explicação funcional do exemplo), com os métodos Push, Pop e Size que atuam sobre uma variável membro "fifo" do tipo `std::vector<Object>`.

O código compreende ficheiros de: i) definição e implementação das classes Queue e Container (listagem 32); ii) definição e implementação das funções objeto (listagem 33); iii) programa principal (listagem 34).

2 <path> = localização do exemplo libcreate_remote.so, caminho absoluto ou relativo

```
struct function_object {
    void operator()(idp_t remote_idp) {
        auto domain = cor::GetDomain();
        auto domain_idp = domain->Idp();
        std::cout << "Function spawned from " << remote_idp
            << " executed on domain " << domain_idp << "" << std::endl;
    }
};
hpx::function<void(idp_t)> Function = function_object();

void Main(int argc, char *argv[]) {
    auto domain = cor::GetDomain();
    auto domain_idp = domain->Idp();
    auto agent_idp = domain->GetActiveResourceIdp();
    auto agent = domain->GetLocalResource<cor::ProtoAgent_Client<void(char**)>>(agent_idp);
    auto clos_idp = domain->GetPredecessorIdp(agent_idp);
    auto clos = domain->GetLocalResource<cor::Closure_Client>(clos_idp);
    auto rank = clos->GetIdm(agent_idp);

    auto remote_domains = domain->GetRemoteDomains();
    idp_t remote_domain_idp = remote_domains[0];

    // criacao de um novo agente no dominio remoto
    auto remote_agent_idp = domain->Create<cor::ProtoAgent_Client<void(idp_t)>>
        (remote_domain_idp, "", Function);

    if(rank == 0) {
        domain->Run<cor::ProtoAgent_Client<void(idp_t)>>
            (remote_agent_idp, domain_idp).get();
    }
    else {
        domain->Run<cor::ProtoAgent_Client<void(idp_t)>>
            (remote_agent_idp, domain_idp).get();
    }
}
```

Listing 31: Exemplo create_remote.cpp

5.5.1 *Definição de Queue e Container*

Pode-se ver que, na listagem 32, as classes `Container` e `Queue` são definidas de forma comum como qualquer outra classe em C++. O único código HPX extra que é necessário incluir é relativo a questões de serialização (função `serialize` e inclusão de `friend class hpx::serialization::access` - linhas 5, 8, 20 e 23).

5.5.2 *Definição das funções objeto*

Para chamar as funções membro da classe `Queue` remotamente, têm de ser criado para cada função uma estrutura auxiliar na forma de uma função objeto. Estas tem de ter como estrutura base, na sua assinatura,

```
class Container {
public:
    Container(int id = 42);
    int GetId() const;
    template <typename Archive> void serialize(Archive& ar, unsigned version);

private:
    friend class hpx::serialization::access;
    int _id;
};

template <typename T>
class Queue : public Container {
public:
    Queue() = default;
    Queue(int id);
    template <typename ... Args> void Push(Args ... args);
    T Pop();
    size_t Size();
    template <typename Archive> void serialize(Archive& ar, unsigned);

private:
    friend class hpx::serialization::access;
    std::vector<T> _fifo;
};
```

Listing 32: queue.hpp

o primeiro parâmetro com o formato de uma referência para o tipo de dados do Dado, que é neste caso `Queue<Object>`.

Este procedimento é pode ser observado na listagem 33, onde é mostrado o encapsulamento de `Push` e `GetId`. Analisando o código, pode-se observar, na linha 3, a criação de uma estrutura chamada `Function_object1`, que contém o operador de função que tem como primeiro parâmetro uma referência para um objeto do tipo de dados "_queue_type" (`Queue<Object>`). Como esta função objeto tem o intuito de chamar a função Push, recebe como restantes argumentos os argumentos que serão passados a Push. Na linha 6, observa-se que a função `Push` é chamada no objeto que é proveniente do primeiro argumento, objeto este que será o albergado pelo Recurso Dado. O último passo é converter a função objeto para uma função do tipo `hpx::function` (linha 9,10). O nome escolhido foi "Push", alusivo à função que encapsula.

O procedimento para a função membro `GetId` é análogo ao usado para `Push`, com a diferença de ter um retorno, do tipo `int`.

Estas funções membro irão ser chamadas através da função `Run` do Dado. Desta forma, o Dado consegue computar quaisquer funções membro do objeto que alberga, independentemente das suas assinaturas.

5.5.3 *Programa*

A listagem 34 inclui, apenas, os extratos do programa considerados necessários para a sua compreensão. Nas linha 1 e 2, faz-se a inclusão, obrigatória, dos cabeçalhos da biblioteca CoR e das funções objeto que (por sua vez inclui a definição da classe Container e Queue). Na linha 5 é usado a macro `REGISTER_DATA()` para registar o tipo de dados do Recurso Data.

```
1  typedef Queue<Object> _queue_type;
2
3  struct Function_object1 {
4      template <typename ... Args>
5      void operator()(_queue_type& obj, Args ... args) {
6          obj.Push(args...);
7      }
8  };
9  template <typename ... Args>
10 hpx::function<void(_queue_type&, Args ... args)> Push = Function_object1();
11
12 struct Function_object4 {
13     int operator()(_queue_type& obj) {
14         return obj.GetId();
15     }
16 };
17 hpx::function<int(_queue_type&)> GetId = Function_object4();
```

Listing 33: queue_interface.hpp - Ficheiro cabeçalho de definição das funções objeto

Na função Main, é obtido o domínio local (linha 10) e nele é criada uma instância local do Recurso Data, com o tipo `Queue<Object>` e com nome "data" (linha 13).

Na linha 14, são criados três objetos do tipo Object que são inseridos na fila (linhas 16 e 17) recorrendo ao método Run, encapsulando a chamada de `Push`. Note-se que a função `Push` é a função objeto definida na listagem 33 que, por sua vez, irá chamar a função membro `Push` que pertence à `Queue`.

Escolhido o domínio remoto (linhas 19 e 20) o Dado é migrado (linha 21).

Notar que a ações `Pop`, `Size` e `Get` (linhas 23,25 e 27) atuam remotamente sobre o Recurso Dado enquanto que as ações `Push`, executadas anteriormente, atuavam sobre o dado no domínio local.

5.5.4 *Observações*

Pretende-se, nesta subsecção, comparar as duas formas de criar a classe `Queue` como instâncias migráveis com acesso remoto, enunciadas neste trabalho na secção atual e na subsecção 3.3.4.

A classe `Queue` para ser um objeto com acesso remoto exige que o utilizador a transforme num componente. Além disso, para cumprir com os nossos requisitos, precisa de ser migrável, *template* e com hierarquia (derivação de `Container`) (ver secção 3.3.4). Para uma melhor manipulação, é também recomendado ao utilizador criar o componente cliente correspondente.

Este caso particular de componente exige conhecimentos avançados de HPX e ultrapassa os casos de uso triviais disponibilizados na documentação. Constata-se, também, que é fácil haver erros na construção e que estes são de difícil resolução sem o suporte direto dos programadores do HPX.

Utilizando o CoR-HPX, o utilizador pode construir a classe *template* `Queue`, derivada de `Container`, e dar acesso remoto sem a necessidade de aplicar código intrusivo HPX (exceto serialização). Posteriormente, para efetivamente a classe possuir acesso remoto, o utilizador tem de criar uma instância do Recurso Dado com o tipo de dados em questão (`cor::Data<Queue<Object»`) (listagem 34). Ao mesmo tempo, para

```
#include "cor/cor.hpp"
#include "queue_interface.hpp"

typedef Queue<Object> myqueue_type;
REGISTER_DATA(myqueue_type); // necessário para registar o novo tipo do Data

void Main(int argc, char *argv[])
{
    // obter o domínio local
    auto domain = cor::GetDomain();

    // criar um Dado que irá conter a um objeto da classe Queue<Object>
    auto myqueue = domain->CreateLocal<cor::Data_Client<queue_type>>(domain->Idp(), "data");
    Object objA, objB, objC;

    myqueue->Run(Push<Object>, objA).get();
    myqueue->Run(Push<Object, Object>, objB, objC).get();

    std::vector<idp_t> remote_domains = domain->GetRemoteDomains();
    idp_t dest = remote_domains[0];
    myqueue->Migrate(dest);

    auto element = myqueue->Run(Pop).get();

    std::cout << myqueue->Run(Size).get() << std::endl; // irá imprimir 2

    std::cout << myqueue->Run(GetId).get() << std::endl; // irá imprimir 42

    return;
}
```

Listing 34: queue_program.cpp - Ficheiro de um exemplo de um programa para utilização do recurso Data<Queue<Object»

executar as funções membro de `Queue` remotamente, como ações, o utilizador tem de criar funções objeto, seguindo a estrutura mostrada na listagem 33.

Resumindo, encontram-se a seguir as diferenças entre o HPX e CoR-HPX, relativamente à construção da classe `Queue`, classe esta com hierarquia (derivada de `Container`), *template* e migrável:

Construção em HPX:

1. Construir as classes Queue (*template*) e Container;
2. Adicionar código HPX para transformar as classes em componentes;
3. Adicionar código HPX para serem componentes migráveis;
4. Adicionar código HPX para registar as ações e o tipo de componente (passos acrescidos para as ações de Queue, uma vez que Queue é *template*);
5. Adicionar código HPX por causa da relação de hierarquia entre ambas as classes;
6. Criar o componente cliente.

Construção em CoR-HPX:

1. Construir as classes Queue (*template*) e Container;
2. Criar uma função objeto para cada função membro, para estas serem executadas remotamente;
3. Criar o Recurso Data com o tipo de dados Queue.

5.6 CASO DE USO STENCIL

Para demonstrar a aplicabilidade de CoR-HPX na resolução de problemas mais complexos que exigem grande atividade computacional e de comunicação, foi desenvolvido um programa *stencil* de transferência de calor numa grelha bidimensional, estrutura recorrente em muitos programas científicos. Optou-se pela escolha deste algoritmo porque já existe uma versão desenvolvida em HPX, encontrada em https://github.com/STEllAR-GROUP/tutorials/tree/master/examples/03_stencil, com explicação da implementação em https://github.com/STEllAR-GROUP/tutorials/tree/master/cscs2016/session5.

Não entrando em pormenores matemáticos e ficando pelo nível da implementação, o algoritmo consiste no processamento iterativo de uma matriz bidimensional, que se subdivide em duas: matriz `A` e a `B`. Os dados da matriz `A` são o resultado do processamento da matriz `B`, onde cada elemento da matriz `A` é calculado pela média dos elementos circundantes da mesma posição da matriz `B`. No fim de cada iteração, os valores da matriz `A` são copiados para a matriz `B` e o processo repete-se.

Em pseudo-código, para cada elemento da matriz `A`, efetua-se o seguinte cálculo:

```
A(x,y) = 0.25 * (B(x-1,y+1) + B(x+1,y+1) + B(x-1,y-1) + B(x+1,y-1))
- B(x,y)
```

Para o algoritmo ser computado com memória distribuída, seguiu-se o padrão de envio/recepção de mensagens, normalmente encontrado em aplicações baseadas em MPI, seguindo o modelo SPMD *(single program, multiple data)*. A sincronização e troca de dados entre os agentes (fios de execução HPX) é conseguida através da utilização de canais, recorrendo ao Recurso MultiChannel. A decomposição da matriz foi feita por blocos de linhas, por uma questão de simplicidade e de demonstração. Considera-se, daqui em diante, um bloco de linhas como uma partição. Cada partição terá um comunicador para trocar a linha superior com a partição vizinha superior e a linha inferior com a partição vizinha inferior.

A nível de distribuição dos dados, optou-se por atribuir duas partições por Domínio CoR (processo unix/localidade HPX). Esta opção tem como objetivo reduzir latências de comunicação. A discussão e explicação deste algoritmo em HPX, análogo ao de CoR-HPX, pode ser encontrado em Heller (2019).

5.6.1 *Implementação em CoR-HPX*

No início do programa, são definidas as variáveis que contêm as informações gerais do algoritmo, como por exemplo, o número de linhas, de colunas, de partições e o número de iterações. Por conseguinte, é executada uma instância da função "worker" por cada partição, inicializando a simulação. A função "worker" é a função onde se encontra o núcleo da simulação.

Para criar essas partições, duas por cada Domínio, recorreu-se ao Recurso Operon, executando duas instâncias da função "worker" paralelamente. A vantagem de usar o Recurso Operon é lançar tantas partições por Domínio quantas se queira. Neste caso, como se optou por usar duas partições por Domínio, o Recurso Operon irá ser executado com duas tarefas.

Para um melhor esclarecimento atente-se na listagem 35, onde se pode ver o Recurso Operon ser criado e acessível através de um apontador global. Recebe como argumento o número de partições (número de tarefas). A função "worker" é de seguida executada recorrendo à função "Dispatch", que recebe a função ("worker") e os seus argumentos, que são, respetivamente, o apontador do Domínio, o rank do agente * o número de partições (totalizando o rank inicial de cada partição), o número de partições, o número de linhas e o número de colunas atribuídos à partição e por fim o número de ciclos que serve de condição para terminar a simulação. Relativamente à atribuição de ranks às partições, chama-se rank inicial a multiplicação do rank do agente inicial do Domínio pelo número de partições por Domínio; o rank final é a soma do resultado dessa operação mais o rank interno que o operão atribui a cada tarefa, que é efetuado dentro da função "worker".

```
1  std::shared_ptr<cor::Operon_Client> operon;
2  void Main(int argc, char *argv[]) {
3      // ...
4      operon = domain->CreateLocal<cor::Operon_Client>(domain->Idp(),  "", num_local_partitions);
5      auto res = operon->Dispatch(&worker, domain, (agent_rank * num_local_partitions),
6                                          num_partitions, Nx, Ny, steps);
7      res.get();
8  }
```

Listing 35: Criação e execução do recurso Operon

Inicialização

Em cada partição (função "worker"), são criadas duas matrizes que irão servir para armazenar os dados computados. Em cada iteração da simulação, a matriz U[1] irá conter o estado atual e U[0] o estado anterior. As suas fronteiras são preenchidas com uns e o interior com zeros.

Depois da criação das matrizes é construído um comunicador (listagem 36), recorrendo ao Recurso MultiChannel. Cada instância de MultiChannel é criada recebendo como argumentos o seu nome (rank) e o nome dos vizinhos (rank-1 e rank+1), se existirem. Posteriormente são enviadas as linhas de fronteira aos vizinhos através da função "Set", que recebe como argumentos os dados a enviar, o nome do parceiro e o índice da mensagem (0 porque é a iteração 0).

Simulação

A simulação propriamente dita consiste num ciclo (listagem 37), em que dentro de cada iteração acontece a computação assíncrona de três futuros: um futuro representa a troca da linha de cima com o vizinho superior (linha 2), outro futuro representa a troca da linha de baixo com o vizinho inferior (linha 4), e um outro futuro representa a computação interior da matriz (linha 3). Estes três futuros são processados em cada iteração do ciclo da simulação. Se por ventura não existirem vizinhos, ou se só existir um deles, os futuros correspondentes aos vizinhos que faltam são colocados imediatamente como prontos. Assim que os três futuros fiquem prontos, o LCO `hpx::wait_all` (linha 5) retorna e a matriz U[1] é copiada para a matriz U[0]. O número de iterações do ciclo é limitada pela variável "steps", definida pelo utilizador.

```
// ...
std::string myself = "rank" + std::to_string(rank);
std::string upper_neighbor = "";
std::string bottom_neighbor = "";

std::unique_ptr<cor::MultiChannel_Client<std::vector<double>>> multichannel;

// We have an upper and bottom neighbors
if ((rank > 0) && (rank < num - 1))
{
    upper_neighbor = "rank" + std::to_string(rank-1);
    bottom_neighbor = "rank" + std::to_string(rank+1);
    multichannel = std::move(domain->CreateLocal<cor::MultiChannel_Client<std::vector<double>>>
                            (domain->Idp(), "",  myself, upper_neighbor, bottom_neighbor));

    // send initial value to our upper and bottom neighbors
    multichannel->Set(std::vector<double>(U[0].begin(), U[0].begin() + Nx), upper_neighbor, 0);
    multichannel->Set(std::vector<double>(U[0].end() - Nx, U[0].end()), bottom_neighbor, 0);
}
// ...
```

Listing 36: Criação do recurso MultiChannel

```
for(std::size_t t = 0; t < steps; ++t) {
    // top_boundary_future = ...
    // interior_future = ...
    // bottom_boundary_future = ...
    hpx::wait_all(top_boundary_future, interior_future, bottom_boundary_future);
    U[0].swap(U[1]);
}
```

Listing 37: Ciclo externo da simulação

Examine-se com mais detalhe o futuro que contém a troca de linha com o vizinho superior (listagem 38). A troca com o vizinho inferior é análoga. Primeiramente, verifica-se se o vizinho existe (linha 2). Se sim, é criada uma operação assíncrona, armazenando o seu estado no futuro `top_boundary_future` (linha 3). De seguida, é chamada a função `Get` com o nome do parceiro e o índice da mensagem (linha 3). Assim que a mensagem for recebida, é executado imediatamente a continuação que o `then` inicia (linha 4), que consiste no processamento da linha superior da matriz daquela partição (linhas 9 e 10). Essa mesma linha é enviada, depois de computada, para o mesmo vizinho de cima, que será a sua linha de baixo. Note-se que a mensagem de envio é assíncrona sem aviso de receção e o índice é o número da iteração seguinte (t é a iteração atual). Caso não exista vizinho de cima, o futuro é colocado como pronto (linha 18).

O processamento do interior da matriz consiste no código da listagem 39. É usado o LCO `hpx::for_loop` para iterar as linhas e as computar em paralelo. Este LCO foi utilizado porque é uma primitiva HPX já otimizada e orientada para este tipo de casos. Dá a opção de escolher uma política de paralelização "policy" (linha 1) para adequar o tipo de escalonamento ao tipo de problema. Para saber mais sobre o `hpx::for_loop`, consulte a documentação do HPX.

```
hpx::future<void> top_boundary_future;
if (upper_neighbor != "") {
    top_boundary_future = multichannel->Get(upper_neighbor, t)
    .then([&U, &multichannel, upper_neighbor, Nx, Ny, t] (auto up_future){
        std::vector<double> up = up_future.get();

        // Iterate over the interior: skip the last and first element
        for(int j = 1; j < Nx-1; j++) {
            U[1][j] = 0.25 * (up[j-1] + up[j+1] + U[0][(j + Nx) - 1]
                + U[0][(j + Nx) + 1]) - U[0][j];
        }

        std::vector<double> newVec(U[1].begin(), U[1].begin()+Nx);
        multichannel->Set(std::move(newVec), upper_neighbor, t + 1);
    });
}
else {
    top_boundary_future = hpx::make_ready_future();
}
```

Listing 38: Troca da linha de fronteira de cima com o vizinho superior

```
hpx::future<void> interior_future = hpx::for_loop(policy, 1, Ny-1,
    [&U, Nx, Ny](std::size_t i) {
        for(std::size_t j = 1; j < Nx-1; ++j)
        {
            U[1][i*Nx + j] = 0.25 * (U[0][(i-1)*Nx + j-1] + U[0][(i-1)*Nx + j+1]
            + U[0][(i+1)*Nx + j-1] + U[0][(i+1)*Nx + j+1]) - U[0][i*Nx + j];
        }

    });
```

Listing 39: Processamento interior da matriz

5.6.2 *Observações*

Concluindo, a utilização do CoR-HPX facilitou a programação relativamente: i) ao lançamento de partições e ii) à troca de mensagens entre as partições.

Sem a utilização do Recurso Operon para executar a função "worker", as tarefas em que esta iria correr teriam de ser criadas explicitamente em HPX. O Operon, além de lançar tarefas, também permite a sincronização entre elas e disponibiliza facilidades para execução paralela de ciclos, não usados neste exemplo.

No que diz respeito às trocas de mensagens, sem a utilização do Recurso MultiChannel os canais teriam de ser criados e conectados explicitamente, 4 por cada partição, levando à necessidade do programador ter de criar uma estratégia de atribuição de nomes aos canais para estes se conectarem corretamente. O Recurso MultiChannel oferece a possibilidade de criar pontos de comunicação entre vários agentes de uma forma intuitiva.

6

DISCUSSÃO E CONCLUSÕES

6.1 DISCUSSÃO

Antecedentes

No início dos trabalhos desta dissertação nada fazia prever que o tema original "estudo, avaliação e aplicação da plataforma HPX" confluiria no desenho da plataforma CoR-HPX. Havia, evidentemente, a ideia de encontrar um tema suficiente interessante e complexo para justificar o investimento no estudo e utilização de uma plataforma atrativa e certamente poderosa. Foi neste contexto que surgiu a hipótese, também ela estimulante, de reconstruir o ambiente PlaCoR, sobre a infraestrutura de *runtime* do HPX que veio a culminar na criação da plataforma CoR-HPX.

O projeto era ambicioso na medida em que iria acrescentar aos trabalhos inicialmente previstos a necessidade de: i) estudar em profundidade os conceitos envolvidos no modelo CoR na origem do PlaCoR, por forma a encontrar as relações de proximidade com o modelo ParalleX, na base do HPX, ii) conhecer com grande detalhe a plataforma PlaCoR, com o objetivo de minimizar o esforço de reescrita do código, mantendo a compatibilidade com a API herdada, iii) integrar no *runtime* do HPX as ferramentas de arranque das aplicações CoR, de forma a maximizar os recursos de comunicação e computação disponíveis em ambientes de *clustering*.

1ª fase

O ponto de partida dos trabalhos foi, naturalmente, o estudo alargado da plataforma HPX realizada a partir da instalação do software, a consulta da documentação e a escrita de programas de teste e de avaliação das facilidades disponíveis.

Do estudo realizado, pode-se constatar que o HPX é uma realização eficiente de um sistema de *runtime* concorrente/paralelo/distribuído especialmente projetado para o desenvolvimento de aplicações de sistema e científicas para arquiteturas de hardware híbridas e de vários núcleos.

Na análise realizada, a ênfase foi posta na aplicabilidade dos seguintes conceitos presentes em HPX: i) no paralelismo de grão fino utilizando tarefas leves, em vez de fios de execução do sistema de operação, ii) na programação assíncrona com e sem continuação, iii) nos mecanismos de sincronização baseados em restrições,

iv) nas extensões da API do C++ moderno para a adaptar a ambientes de sistemas distribuídos, v) no controlo de localidade dinâmico, adaptativo e espaço de endereçamento uniforme e global para as aplicações distribuídas e vi) nas ações, componentes e canais.

Durante esta fase é de referir a insuficiência (e muitas vezes ausência) de documentação em muitos dos tópicos envolvidos, o que teve a vantagem de propiciar o contacto direto, via "chat", com responsáveis daquele projeto, mas que se traduziu em muito tempo perdido, até obter a informação desejada.

Na perspetiva de fixar o conhecimento adquirido nesta fase, um dos primeiros contributos deste trabalho foi a escrita de um capítulo integralmente dedicado aos tópicos referidos no ponto vi) acima, que em conjunto com o código disponível em `https://github.com/tiagofgon/thesis_examples` , de alguma forma complementa a documentação.

2ª fase

Esta fase foi consagrada, em primeiro lugar, ao estudo do modelo de computação orientado ao Recurso e posteriormente à exploração da PlaCoR num ambiente de computação distribuído em "cluster". A partir da introdução da metáfora do Recurso que incorpora a noção de estado, de concorrência, de localidade e de distribuição, progrediu-se no reconhecimento das suas características comuns, evidenciadas na tríade: i) identificação única, no sistema distribuído ii) propriedades específicas, tais como a coerência entre réplicas e iii) corpo, que é uma realização concreta do Recurso no sistema hospedeiro.

De entre as classes de Recursos de raiz destacam-se: os domínios que delimitam espaços de endereçamento onde têm lugar as interações entre os Recursos; as tarefas, que são entidades executáveis, escalonadas como fios de controlo independentes; e os dados, usados para a criação/manipulação de informação, globalmente, partilhada num ambiente de computação distribuído.

Na posse deste conhecimento, iniciou-se o estudo da PlaCoR com base no desenho da sua arquitetura e na identificação das ferramentas de desenvolvimento usadas, e na realização do protótipo, para acomodar os conceitos do modelo de computação.

A plataforma constitui-se num ambiente de execução paralelo de domínios distribuídos, realizados sob a forma de Pods, o equivalente de um processo Unix, onde se processa a interação de Recursos, se garante a consistência entre réplicas destes e se faz a coordenação de serviços essenciais, tais como controladores e correio.

O passo seguinte, foi a familiarização com a API de Recursos, através da análise, compilação e execução de códigos de programas orientados ao Recurso.

3ª fase

Esta fase corresponde ao desenho e realização do CoR-HPX como uma nova abordagem ao CoR que dá resposta às limitações do protótipo anterior, nomeadamente, no que concerne: i) a execução – adotando o modelo task-centric em detrimento do modelo thread-centric, ii) replicação– adotando uma política de cópia

única centralizada em vez de réplicas distribuídas, iii) comunicação em grupo – substituindo a biblioteca spread por mecanismos de sincronização próprios do HPX, iv) comunicação ponto-aponto - usando canais como meio de transferência de grandes volumes de dados arbitrários de dados em vez do envio/receção de mensagens tradicional e v) API – estender a API anterior de forma a introduzir a programação assíncrona.

Convém referir que o desenho e escrita do código, tanto quanto foi possível, seguiu de perto as grandes opções e inovações introduzidas no trabalho anterior, incluindo a API, pelo que a leitura da documentação associada é fundamental para a compreensão da nova plataforma.

6.2 CONCLUSÕES

Pode-se afirmar que todos os objetivos previstos no plano inicial desta dissertação foram atingidos.

Começando pela primeira fase, foi possível confirmar a validade da abordagem seguida pelo HPX no suporte à execução e desenvolvimento de aplicações em ambiente de memória partilhada e distribuída que decorre da implementação do modelo ParalleX. Conflui para esta conclusão, a inegável vantagem decorrente da definição de uma API em permanente evolução, mas totalmente alinhada com as mais recentes normas C++, que estende de forma inovadora a sintaxe e semântica da programação assíncrona concorrente e paralela, em memória partilhada, à programação em memória distribuída. É também importante salientar que se deve à seleção do estudo de caso, com a inerente complexidade do modelo de computação orientado ao Recurso, a necessidade de estudar profunda e detalhadamente o HPX, de forma a encontrar as soluções mais apropriadas para a realização eficiente dos conceitos e facilidades da plataforma CoR-HPX.

É na segunda fase que assume uma importância capital o estabelecimento da correspondência entre a metáfora do Recurso em CoR e a sua modelação / realização como um componente HPX. Esta opção, consentânea com um modelo de representação de Recursos de cópia única centralizada, é oposta à política de existência de réplicas distribuídas usadas na plataforma anterior. Convém referir que a cópia única centralizada é compatível com a definição de classes de Recursos, como, por exemplo, o Dado, cujas instâncias podem ser movidas entre Domínios (migrar), de forma transparente, dando a ilusão da existência de réplicas do Recurso original.

Com a adoção do paradigma da representação centralizada de Recursos deixa de ser necessário assegurar a consistência entre réplicas de Recursos, antes assegurada pelos mecanismos de comunicação em grupo do Spread e, consequentemente, a possibilidade de substituir todas as funções desempenhadas por aquele componente por facilidades presentes na API do HPX. Em concordância com esta premissa, à exceção da biblioteca libssh, todas as ferramentas usadas em PlaCoR, foram eliminadas.

A validação da nova plataforma fez-se, em primeiro lugar, através da compilação e execução de exemplos de aplicações herdados da plataforma anterior, tendo-se confirmado que os resultados obtidos reproduziam fidedignamente os obtidos do PlaCoR.

Foi igualmente comprovada a funcionalidade de novas classes de Recursos como é o caso do Operon, do UniChannel e do MultiChannel. A primeira potencia a execução multi tarefa de funções segundo o modelo *fork-join*, enquanto as outras duas classes usam os channel do HPX como meios alternativos para o envio/receção

de mensagem entre tarefas, dando suporte a modos de comunicação bidirecionais simples e em topologias pré-definidas.

Na senda da tendência crescente para o surgimento de linguagens que suportam a programação AMT e, desta forma, maximizar a execução assíncrona local/remota, à API de base foram adicionadas novas funcionalidades em consonância com aquele paradigma.

Pode também afirmar-se que a nova aproximação à programação orientada ao Recurso assente na CoR-HPX vem acrescentar ao próprio HPX um nível superior de abstração, através da utilização das diferentes classes de Recursos como módulos pré-definidos facilmente adaptáveis ao código específico de cada nova aplicação. Exemplo paradigmático desta possibilidade é a classe Dado que inclui os mecanismos necessários para, com o envolvimento mínimo do programador, transformar as instâncias de qualquer classe Value, definida da forma convencional em C++, em objetos *template*, remotamente endereçáveis e migráveis, ocultando quase completamente o *boilerplate* do HPX. Assim, o Recurso Dado é em simultâneo: i) um tipo de componente (HPX) convencional com estado (instância de Value), mas que não está preso à definição prévia das suas funções membro para serem usadas como ações e ii) um objeto que pode ser usado para executar funções com qualquer assinatura, o equivalente a uma ação simples (HPX).

6.3 TRABALHO FUTURO

Durante os trabalhos desta dissertação independentemente das opções tomadas e do trabalho surgiram, naturalmente, ideias tanto ao nível da conceção como da realização que parecerem interessantes, mas que não foram incluídas nos objetivos a alcançar. No que segue enunciamos algumas daquelas ideias, como perspetivas de trabalho futuro:

- Domínios virtuais - para quebrar a associação direta existente entre a realização do Recurso Domínio, representado pelo Pod (na arquitetura do CoR-HPX) com o processo (ou localidade em HPX) do sistema hospedeiro. Assim, o lançamento em paralelo de vários Domínios diretamente da consola ou através da primitiva spawn da API daria lugar à possibilidade de que em cada localidade (local ou remota) coexistissem, para além de um Pod convencional, múltiplos Pods virtuais que de forma transparente seriam integrados no sistema de Domínios distribuído que constitui cada uma das aplicações CoR. Existe a expectativa que a virtualização de Pods possa contribuir para melhorar o desempenho de aplicações em sintonia com a abordagem do HPX que, por omissão, assume que todos os recursos físicos num nó de computação estão associados a uma única localidade, por forma a otimizar o *runtime* e minimizar a comunicação inter-processo.
- API menos verbosa - a instanciação de novos Recursos é efetuada através do Recurso Domínio de forma explícita, o que leva a que a escrita do código fique mais extensa. Um melhoramento possível seria encapsular a conexão ao Domínio dentro da classe cliente dos próprios Recursos. Assim, seria possível criar Recursos espontaneamente, como qualquer outro objeto.
- Desenho automático da árvore de dependências dos programas - a execução de um programa origina uma árvore de dependências que resulta dos vários Recursos pendurados nos Recursos ascendentes. Essa

árvore poderia ser gerada por uma ferramenta auxiliar de forma a ilustrar ao programador as dependências dos Recursos criadas.

- Análise de desempenho com os contadores disponibilizados pelo HPX - o HPX oferece uma vasta gama de contadores de desempenho que poderiam ser usados para uma análise mais minuciosa do comportamento da plataforma. Contadores HPX não foram abordados nesta dissertação.

BIBLIOGRAFIA

Intel SPMD Program Compiler. http://ispc.github.io/, 2020. [Online; accessed 20-March-2020].

Victor Alessandrini. *Shared Memory Application Programming: Concepts and strategies in multicore application programming*. Morgan Kaufmann, 2015.

Parsa Amini. Adaptive data migration in load-imbalanced hpc applications. 2020.

Parsa Amini and Hartmut Kaiser. Assessing the performance impact of using an active global address space in hpx: A case for agas. In *2019 IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*, pages 26–33. IEEE, 2019.

Yair Amir, Claudiu Danilov, Michal Miskin-Amir, John Schultz, and Jonathan Stanton. The spread toolkit: Architecture and performance. *Johns Hopkins University, Tech. Rep. CNDS-2004-1*, 2004.

Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

John Bachan, Scott B Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H Hargrove, and Hadia Ahmed. Upc++: A high-performance communication framework for asynchronous computation. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 963–973. IEEE, 2019.

Michael Edward Bauer. *Legion: Programming distributed heterogeneous architectures with logical regions*. PhD thesis, Stanford University, 2014.

John Biddiscombe, Anton Bikineev, Thomas Heller, and Hartmut Kaiser. Zero copy serialization using rma in the hpx distributed task-based runtime. 2017.

Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1): 55–69, 1996a.

Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1): 55–69, 1996b.

George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

Maximilian Bremer, Kazbek Kazhyken, Hartmut Kaiser, Craig Michoski, and Clint Dawson. Performance comparison of hpx versus traditional parallelization strategies for the discontinuous galerkin method. *Journal of Scientific Computing*, 80(2):878–902, 2019.

Colin Campbell, Ralph Johnson, Ade Miller, and Stephen Toub. *Parallel programming with Microsoft. NET: design patterns for decomposition and coordination on multicore architectures.* Microsoft Press, 2010.

Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538.

Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters*, 21(02):173–193, 2011.

Guang R Gao, Thomas Sterling, Rick Stevens, Mark Hereld, and Weirong Zhu. Parallex: A study of a new parallel computation model. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–6. IEEE, 2007.

William Gropp. Mpi at exascale: Challenges for data structures and algorithms. *PVM/MPI*, 3, 2009.

Patricia Grubel, Hartmut Kaiser, Jeanine Cook, and Adrian Serio. The performance implication of task size for applications on the hpx runtime system. In *2015 IEEE International Conference on Cluster Computing*, pages 682–689. IEEE, 2015.

Patricia Grubel, Hartmut Kaiser, Kevin Huck, and Jeanine Cook. Using intrinsic performance counters to assess efficiency in task-based parallel applications. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1692–1701. IEEE, 2016.

Patricia A Grubel. *Dynamic Adaptation in HPX: A Task-based Parallel Runtime System.* PhD thesis, New Mexico State University, 2016.

Nikunj Gupta, Steve R Brandt, Bibek Wagle, Nanmiao Wu, Alireza Kheirkhahan, Patrick Diehl, Felix W Baumann, and Hartmut Kaiser. Deploying a task-based runtime system on raspberry pi clusters. In *2020 IEEE/ACM 5th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pages 11–20. IEEE, 2020.

Thomas Heller. Extending the c++ asynchronous programming model with the hpx runtime system for distributed memory computing. 2019.

Herbert Jordan, Thomas Heller, Philipp Gschwandtner, Peter Zangerl, Peter Thoman, Dietmar Fey, and Thomas Fahringer. The allscale runtime application model. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 445–455. IEEE, 2018.

Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *2009 International Conference on Parallel Processing Workshops*, pages 394–401. IEEE, 2009.

Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, pages 1–11, 2014.

Hartmut Kaiser, Patrick Diehl, Adrian S. Lemoine, Bryce Adelstein Lelbach, Parsa Amini, Agustín Berge, John Biddiscombe, Steven R. Brandt, Nikunj Gupta, Thomas Heller, Kevin Huck, Zahra Khatami, Alireza Kheirkhahan, Auriane Reverdell, Shahrzad Shirzad, Mikael Simberg, Bibek Wagle, Weile Wei, and Tianyi Zhang. Hpx - the c++ standard library for parallelism and concurrency. *Journal of Open Source Software*, 5 (53):2352, 2020. doi: 10.21105/joss.02352. URL https://doi.org/10.21105/joss.02352.

Laxmikant V Kale and Sanjeev Krishnan. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, 1993.

Zahra Khatami, Hartmut Kaiser, Patricia Grubel, Adrian Serio, and J Ramanujam. A massively parallel distributed n-body application implemented with hpx. In *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pages 57–64. IEEE, 2016a.

Zahra Khatami, Hartmut Kaiser, and J Ramanujam. Using hpx and op2 for improving parallel scaling performance of unstructured grid applications. In *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, pages 190–199. IEEE, 2016b.

Cecília Moreira. CoRes - Computação orientada ao Recurso - uma especificação. Master's thesis, Universidade do Minho, Braga, Portugal, 2001.

Sri Raj Paul, Akihiro Hayashi, Nicole Slattengren, Hemanth Kolla, Matthew Whitlock, Seonmyeong Bak, Keita Teranishi, Jackson Mayo, and Vivek Sarkar. Enabling resilience in asynchronous many-task programming models. In *European Conference on Parallel Processing*, pages 346–360. Springer, 2019.

David Pfander, Gregor Daiß, Dominic Marcello, Hartmut Kaiser, and Dirk Pflüger. Accelerating octo-tiger: Stellar mergers on intel knights landing with hpx. In *Proceedings of the International Workshop on OpenCL*, pages 1–8, 2018.

António Pina. *MC^2 - Modelo de Computação Celular - Origem e Evolução*. PhD thesis, Universidade do Minho, Braga, Portugal, 1997.

Bruno Ribeiro. PlaCoR - Plataforma para a Computação orientada ao Recurso. Master's thesis, Universidade do Minho, Braga, Portugal, 2019. URL http://hdl.handle.net/1822/66578.

Bruno Ribeiro and António Pina. Placor: plataforma para a computação orientada ao recurso. 2019. URL http://hdl.handle.net/1822/69658.

Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, et al. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, 2017.

Patrícia Gomes Soares. On remote procedure call. In *Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research-Volume 2*, pages 215–267. IBM Press, 1992.

Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, et al. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4):1422–1434, 2018.

TIOBE. Tiobe group et al. tiobe index for ranking the popularity of programming languages., 2020. (Accessed: 3.12.2020. 2020. url: http://www.tiobe.com/tiobe-index/).

Thorsten Von Eicken, David E Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. *ACM SIGARCH Computer Architecture News*, 20 (2):256–266, 1992.

Michael Voss, Rafael Asenjo, and James Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. Apress, 2019.

Kyle B Wheeler, Richard C Murphy, and Douglas Thain. Qthreads: An api for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, 2007.

INSTALAÇÃO DO SISTEMA COR-HPX

A.1 REQUISITOS E DOWNLOAD

O CoR-HPX pode ser instalado e explorado nos sistemas Linux, Solaris e MacOS. Foi compilado e testado com os compiladores GNU GCC e Clang, com versões que suportam a versão padrão C++17. Para compilação recomendamos a ferramenta CMake. O libssh é também necessário para lançar processos dinamicamente em tempo de execução.

Resumindo, os requisitos para instalar CoR-HPX são:

- GCC/Clang - com suporte para C++17
- HPX - (>= 1.5.0)
- libssh - (>= 0.8)
- CMake - (>= 3.12)

Para instalar o HPX recomendamos consultar:

- Documentação do HPX - Informação sobre a instalação do HPX, exemplos de programas, descrição da API, etc.
- GitHub do HPX - Contem a plataforma para download em conjunto com informação útil.

CoR-HPX encontra-se *online* disponível para download em `https://github.com/tiagofgon/CoR-HPX`, contendo também as instruções de instalação.

A.2 CONFIGURAÇÃO DO AMBIENTE DE EXECUÇÃO

As aplicações CoRHPX são executadas através do programa `corhpx`. Este cria o *pod* local e executa a função de entrada do módulo do utilizador. Assim, é necessário colocar o caminho de corhpx na variável de ambiente `PATH`.

Para utilizar a aplicação sem ter que definir caminhos é recomendável que se insira na variável de ambiente `LD_LIBRARY_PATH` o caminho da biblioteca CoR-HPX chamada libcor.so, assim como os caminhos dos exemplos.

A.3 EXECUÇÃO DE APLICAÇÕES

Para excutar uma aplicação em ambiente distribuído, podem ser arrancados manualmente cada um dos processos com corhpx com as respetivas configurações do HPX para conectar os processos e agregá-los no mesmo runtime, ou então, por exemplo, utilizar o MPI para servir de escalonador de processos. A seguir encontra-se um exemplo de como executar o o modulo "parallel"das duas formas, com nós: node1 - 10.1.2.1 e node2 - 10.1.2.253.

Execução em dois nós com arranque manual de corhpx numa consola em ambos:

- node1: `$ corhpx app ctx 2 0 libparallel.so -hpx:hpx=10.1.2.1:1337 -hpx:expect-connecting-localities`
- node2: `$ corhpx app ctx 2 0 libparallel.so -hpx:hpx=10.1.2.253:1338 -hpx:agas=10.1.2.1:1337 -hpx:run-hpx-main -hpx:expect-connecting-localities -hpx:worker`

Execução com MPI:

- `$ mpirun -np 2 -map-by node -host node1,node2 corhpx app ctx 2 0 libparallel.so`

B

CÓDIGOS COMPLEMENTARES

B.1 CRIAR UM COMPONENTE TEMPLATE - EXTENSÃO

B.1.1 *Componente servidor*

```
template <typename T>
class Queue: public hpx::components::locking_hook< hpx::components::component_base<Queue<T>> >
{
public:
    template <typename ... Args>
    void Push(Args ... args) {
        (_fifo.push_back(args), ...);
    }

    T Pop() {
        T element = _fifo.front();
        _fifo.erase(_fifo.begin());
        return element;
    }

    size_t Size() {
        return _fifo.size();
    }

    HPX_DEFINE_COMPONENT_ACTION(Queue, Pop, Pop_action_Queue);
    HPX_DEFINE_COMPONENT_ACTION(Queue, Size, Size_action_Queue);

    template <typename ... Args>
    struct Push_action_Queue
    : hpx::actions::make_action<
        decltype(&Queue::Push<Args...>),
        &Queue::Push<Args...>
    >::type {};

private:
    std::vector<T> _fifo;
};

#define REGISTER_QUEUE_DECLARATION(type)                              \
    HPX_REGISTER_ACTION_DECLARATION(                                   \
```

```
36          Queue<type>::Pop_action_Queue,                          \
37          HPX_PP_CAT(__Queue_Pop_action_Queue_, type));           \
38      HPX_REGISTER_ACTION_DECLARATION(                            \
39          Queue<type>::Size_action_Queue,                         \
40          HPX_PP_CAT(__Queue_Size_action_Queue_, type));
41  /**/
42
43  #define REGISTER_QUEUE(type)                                    \
44      HPX_REGISTER_ACTION(                                        \
45          Queue<type>::Pop_action_Queue,                          \
46          HPX_PP_CAT(__Queue_Pop_action_Queue_, type));           \
47      HPX_REGISTER_ACTION(                                        \
48          Queue<type>::Size_action_Queue,                         \
49          HPX_PP_CAT(__Queue_Size_action_Queue_, type));          \
50      typedef ::hpx::components::component< Queue<type> > HPX_PP_CAT(__Queue_, type); \
51      HPX_REGISTER_COMPONENT(HPX_PP_CAT(__Queue_, type))
52  /**/
```

B.1.2 *Componente cliente*

```
1   template <typename T>
2   class Queue_Client: hpx::components::client_base<Queue_Client<T>, Queue<T>>
3   {
4   public:
5       typedef hpx::components::client_base<Queue_Client<T>, Queue<T>> base_type;
6
7           Queue_Client() : base_type(hpx::local_new<Queue<T>>())
8           {}
9           Queue_Client(hpx::id_type locality) : base_type(hpx::new_<Queue<T>>(locality))
10          {}
11          template <typename ... Args>
12          hpx::future<void> Push(Args ... args)
13          {
14                  typedef typename Queue<T>::template Push_action_Queue<Args...> action_type;
15                  return hpx::async<action_type>(this->get_id(), args...);
16          }
17          hpx::future<T> Pop() {
18                  typedef typename Queue<T>::Pop_action_Queue action_type;
19                  return hpx::async<action_type>(this->get_id());
20          }
21          hpx::future<size_t> Size() {
22                  typedef typename Queue<T>::Size_action_Queue action_type;
23                  return hpx::async<action_type>(this->get_id());
24          }
25  };
```

B.2 CRIAR UM COMPONENTE TEMPLATE MIGRÁVEL - EXTENSÃO

B.2.1 *Componente servidor*

```
template <typename T>
class Queue: public hpx::components::migration_support<
                    hpx::components::component_base<Queue<T>> >
{

public:
    friend class hpx::serialization::access;

    typedef hpx::components::migration_support< hpx::components::component_base<Queue<T>> > base_type;
    // Components which should be migrated using hpx::migrate<> need to
    // be Serializable and CopyConstructable. Components can be
    // MoveConstructable in which case the serialized data is moved into the
    // component's constructor.
    Queue(Queue const& rhs)
      : base_type(rhs), _fifo(rhs._fifo)
    {}

    Queue(Queue && rhs)
      : base_type(std::move(rhs)), _fifo(rhs._fifo)
    {}

    Queue& operator=(Queue const & rhs)
    {
        _fifo = rhs._fifo;
        return *this;
    }
    Queue& operator=(Queue && rhs)
    {
        _fifo = rhs._fifo;
        return *this;
    }

    Queue() = default;

    template <typename ... Args>
    void Push(Args ... args) {
        (_fifo.push_back(args), ...);
    }

    T Pop() {
        T element = _fifo.front();
        _fifo.erase(_fifo.begin());
        return element;
    }

    size_t Size() {
        return _fifo.size();
    }

```

```
50      HPX_DEFINE_COMPONENT_ACTION(Queue, Pop, Pop_action_Queue);
51      HPX_DEFINE_COMPONENT_ACTION(Queue, Size, Size_action_Queue);
52
53      template <typename ... Args>
54      struct Push_action_Queue
55      : hpx::actions::make_action<
56          decltype(&Queue::Push<Args...>),
57          &Queue::Push<Args...>
58      >::type {};
59
60      template <typename Archive>
61      void serialize(Archive& ar, unsigned version)
62      {
63          ar & _fifo;
64      }
65
66  private:
67      std::vector<T> _fifo;
68  };
69
70
71  #define REGISTER_QUEUE_DECLARATION(type)                    \
72      HPX_REGISTER_ACTION_DECLARATION(                        \
73          Queue<type>::Pop_action_Queue,                      \
74          HPX_PP_CAT(__Queue_Pop_action_Queue_, type));       \
75      HPX_REGISTER_ACTION_DECLARATION(                        \
76          Queue<type>::Size_action_Queue,                     \
77          HPX_PP_CAT(__Queue_Size_action_Queue_, type));      \
78
79  #define REGISTER_QUEUE(type)                                \
80      HPX_REGISTER_ACTION(                                    \
81          Queue<type>::Pop_action_Queue,                      \
82          HPX_PP_CAT(__Queue_Pop_action_Queue_, type));       \
83      HPX_REGISTER_ACTION(                                    \
84          Queue<type>::Size_action_Queue,                     \
85          HPX_PP_CAT(__Queue_Size_action_Queue_, type));      \
86      typedef ::hpx::components::component< Queue<type> > HPX_PP_CAT(__Queue_, type); \
87      HPX_REGISTER_COMPONENT(HPX_PP_CAT(__Queue_, type))
```

B.3 CRIAR UM COMPONENTE TEMPLATE MIGRÁVEL COM HERANÇA - EXTENSÃO

B.3.1 *Componente servidor - classe base Container*

O código da classe `Container` foi subdividido num ficheiro cabeçalho e outro de implementação. Esta classe apenas serve como base de polimorfismo, uma vez que é uma classe abstrata. No ficheiro cabeçalho, para transformar num componente e garantir que também é migrável usa-se a definição
`hpx::components::abstract_base_migration_support< hpx::components::abstract_component_base<Container> >`.

A classe define uma única variável membro (`_id`) do tipo `int` e um método `GetId` que retorna essa variável membro e, para fins de serialização, construtores de cópia e a função de serialização `serialize`.

Sendo a classe abstrata, para criar uma ação que envolva o método `GetId` este tem que ser virtual e é necessário criar um método não virtual `GetId_nonvirt` para invocar `GetId`. A ação não pode encapsular diretamente a função virtual `GetId`, mas pode encapsular a função não virtual que a irá invocar GetID. Ações que englobam diretamente métodos de componentes abstratos, virtuais ou não virtuais, não funcionam corretamente.

Relativamente ao ficheiro de implementação, destaca-se a instrução `HPX_DEFINE_GET_COMPONENT_TYPE(Container)` (linha 10), necessária para definir o novo tipo para as classes derivadas herdarem.

```
1  class Container: public hpx::components::abstract_base_migration_support<
2                      hpx::components::abstract_component_base<Container> >
3  {
4
5  public:
6      typedef hpx::components::abstract_base_migration_support<
7      hpx::components::abstract_component_base<Container> > base_type;
8
9      Container(int id = 42) : _id(id) {}
10     virtual ~Container() = default;
11     Container(Container const& rhs) : base_type(rhs), _id(rhs._id) {}
12     Container(Container && rhs) : base_type(std::move(rhs)), _id(rhs._id) {}
13
14     Container& operator=(Container const & rhs) {
15         _id = rhs._id;
16         return *this;
17     }
18     Container& operator=(Container && rhs) {
19         _id = rhs._id;
20         return *this;
21     }
22
23     virtual int GetId() const;
24     int GetId_nonvirt() const;
25
26     HPX_DEFINE_COMPONENT_ACTION(Container, GetId_nonvirt, GetId_action_Container);
27
28     template <typename Archive>
29     void serialize(Archive& ar, unsigned version){
30         ar & _id;
31     }
32
33 private:
34     friend class hpx::serialization::access;
35     int _id;
36 };
37
38 typedef Container::GetId_action_Container GetId_action_Container;
39 HPX_REGISTER_ACTION_DECLARATION(GetId_action_Container);
```

```
int Container::GetId() const {
    return _id;
}

int Container::GetId_nonvirt() const {
    return GetId();
}

typedef Container Container;
HPX_DEFINE_GET_COMPONENT_TYPE(Container);

typedef Container::GetId_action_Container GetId_action_Container;
HPX_REGISTER_ACTION(GetId_action_Container);
```

B.3.2 *Componente servidor - classe derivada Queue*

```
#include <hpx/hpx.hpp>
#include "container.hpp"


template <typename T>
class Queue: public hpx::components::abstract_migration_support< hpx::components::component_base<Queue<T>
{

public:
    using base_type = hpx::components::abstract_migration_support<
        hpx::components::component_base<Queue<T>>, Container >;

    typedef typename hpx::components::component_base<Queue<T>>::wrapping_type wrapping_type;
    typedef Queue type_holder;
    typedef Container base_type_holder;

    Queue() = default;
    ~Queue() = default;

    Queue(int id) : base_type(id) {}

    // Components which should be migrated using hpx::migrate<> need to
    // be Serializable and CopyConstructable. Components can be
    // MoveConstructable in which case the serialized data is moved into the
    // component's constructor.

    Queue(Queue && rhs) :
        base_type(std::move(rhs)),
        _fifo(std::move(rhs._fifo))
    {}

    Queue& operator=(Queue && rhs)
    {
        this->Container::operator=(std::move(static_cast<Container&>(rhs)));
        _fifo = rhs._fifo;
        return *this;
    }
```

```
38
39      template <typename ... Args>
40      void Push(Args ... args) {
41          (_fifo.push_back(args), ...);
42      }
43
44      T Pop() {
45          T element = _fifo.front();
46          _fifo.erase(_fifo.begin());
47          return element;
48      }
49
50      size_t Size() {
51          return _fifo.size();
52      }
53
54      HPX_DEFINE_COMPONENT_ACTION(Queue, Pop, Pop_action_Queue);
55      HPX_DEFINE_COMPONENT_ACTION(Queue, Size, Size_action_Queue);
56
57      template <typename ... Args>
58      struct Push_action_Queue
59      : hpx::actions::make_action<
60          decltype(&Queue::Push<Args...>),
61          &Queue::Push<Args...>
62      >::type {};
63
64      template <typename Archive>
65      void serialize(Archive& ar, unsigned version)
66      {
67          ar & hpx::serialization::base_object<Container>(*this);
68          ar & _fifo;
69      }
70
71  private:
72      std::vector<T> _fifo;
73  };
74
75
76  #define REGISTER_QUEUE_DECLARATION(type)                  \
77      HPX_REGISTER_ACTION_DECLARATION(                      \
78          Queue<type>::Pop_action_Queue,                    \
79          HPX_PP_CAT(__Queue_Pop_action_Queue_, type));     \
80      HPX_REGISTER_ACTION_DECLARATION(                      \
81          Queue<type>::Size_action_Queue,                   \
82          HPX_PP_CAT(__Queue_Size_action_Queue_, type));    \
83
84  #define REGISTER_QUEUE(type)                              \
85      HPX_REGISTER_ACTION(                                  \
86          Queue<type>::Pop_action_Queue,                    \
87          HPX_PP_CAT(__Queue_Pop_action_Queue_, type));     \
88      HPX_REGISTER_ACTION(                                  \
89          Queue<type>::Size_action_Queue,                   \
90          HPX_PP_CAT(__Queue_Size_action_Queue_, type));    \
91      typedef ::hpx::components::component<Queue<type>>    \
92          HPX_PP_CAT(__Queue_type, type);                   \
```

```
93      typedef Queue<type>                                       \
94          HPX_PP_CAT(__Queue_, type);                           \
95      HPX_REGISTER_DERIVED_COMPONENT_FACTORY(HPX_PP_CAT(__Queue_type, type), \
96                                      HPX_PP_CAT(__Queue_, type), "Container")
```

B.3.3 *Componente cliente*

```
1  template <typename T>
2  class Queue_Client: hpx::components::client_base<Queue_Client<T>, Queue<T>>
3  {
4  public:
5      typedef hpx::components::client_base<Queue_Client<T>, Queue<T>> base_type;
6
7          Queue_Client() : base_type(hpx::local_new<Queue<T>>())
8          {}
9          Queue_Client(hpx::id_type locality, int id) : base_type(hpx::new_<Queue<T>>(locality, id))
10         {}
11         Queue_Client(hpx::id_type locality) : base_type(hpx::new_<Queue<T>>(locality))
12         {}
13
14         template <typename ... Args>
15         hpx::future<void> Push(Args ... args)
16         {
17                 typedef typename Queue<T>::template Push_action_Queue<Args...> action_type;
18                 return hpx::async<action_type>(this->get_id(), args...);
19         }
20
21         hpx::future<T> Pop() {
22                 typedef typename Queue<T>::Pop_action_Queue action_type;
23                 return hpx::async<action_type>(this->get_id());
24         }
25
26         hpx::future<size_t> Size() {
27                 typedef typename Queue<T>::Size_action_Queue action_type;
28                 return hpx::async<action_type>(this->get_id());
29         }
30
31         hpx::future<int> GetId() {
32                 typedef Container::GetId_action_Container action_type;
33                 return hpx::async<action_type>(this->get_id());
34         }
35
36         hpx::id_type get_gid() {
37                 return this->get_id();
38         }
39 };
```

B.4 DIAGRAMA DOS RECURSOS - VERSÃO AMPLIADA

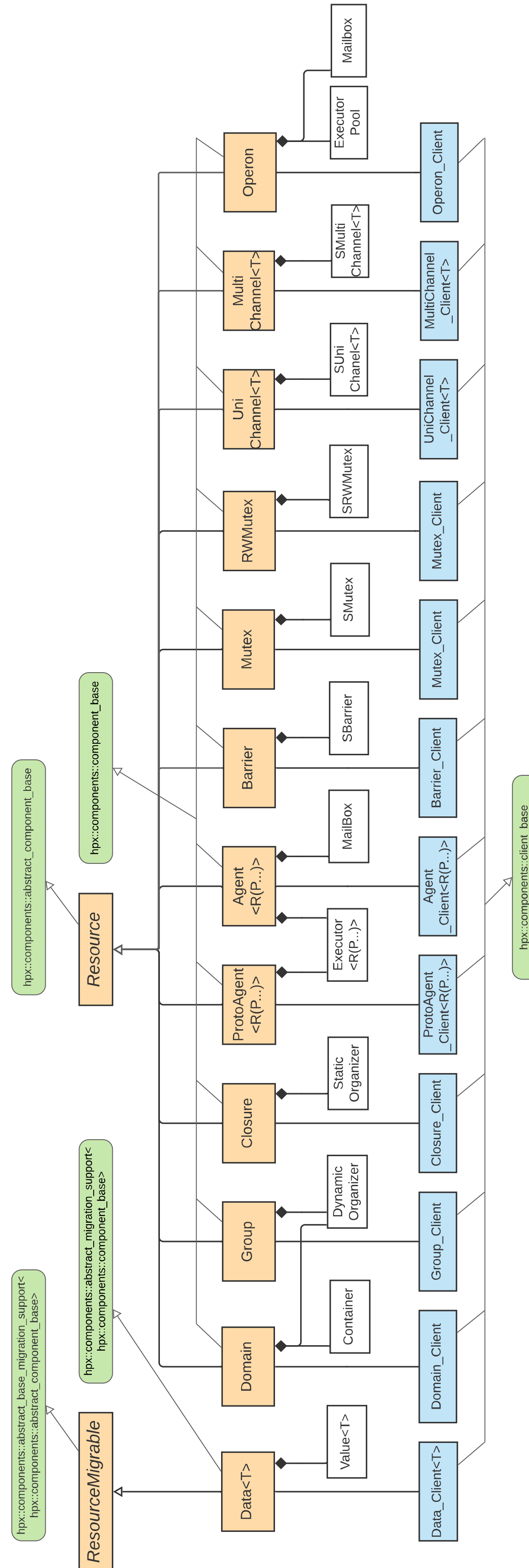
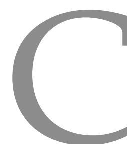


Figura 14: Diagrama de classes dos Recursos



API DOS ELEMENTOS QUE CONSTITUEM OS RECURSOS DE COR-HPX

Em CoR-HPX a criação de Recursos obedece a um conjunto simples de regras modulares acessível ao programador que permite combinar classes de elementos novas e/ou pré-existentes para, através da definição de novas classes clientes, acrescentar novas funcionalidades à plataforma .

No que se segue, apresentam-se as funções das classes de elementos do CoR-HPX. Estas funções são encapsuladas em ações nas classes dos Recursos de que fazem parte, e são mediadas posteriormente pelos componentes cliente que servem de interface e constituem a API final dos Recursos - a API encontra-se no anexo D, contendo um maior detalhe sobre os Recursos e respetivas funções.

C.1 CONTENTOR

O contentor estabelece uma relação direta com as facilidades disponibilizadas pela infraestrutura, através do Pod. Em particular, permite: i) obter o contexto/arranque da aplicação e os identificadores/apontadores dos Recursos no Domínio local, ii) criar dinamicamente Recursos/réplicas local/remotamente e iv) arrancar dinamicamente novos pods/Domínios.

class Container

- **Container**(idp_t idp): construtor de um elemento contentor no contexto identificado por idp.
- std::string **GetGlobalContext**(): retorna o nome do contexto global da aplicação.
- std::string **GetLocalContext**(): retorna o nome do contexto de arranque do Domínio local.
- unsigned int **GetTotalPods**(): retorna o número total de pods presentes na aplicação.
- unsigned int **GetTotalDomains**(): retorna o número total de Domínios na aplicação.
- idp_t **GetActiveResourceIdp**(size_t id): retorna o identificador do Recurso ativo, identificado por id.
- idp_t **GetPredecessorIdp**(idp_t idp): retorna o ascendente do Recurso identificado por idp.
- template <typename T>
  std::unique_ptr<T> **GetLocalResource**(idp_t idp): retorna um apontador para o objeto cliente do tipo T, do Recurso identificador por idp.

- template <typename T, typename ... Args>
  std::unique_ptr<T> **CreateLocal**(idp_t ctx, std::string const& name, Args... args): cria no Domínio local e no contexto ctx, um Recurso do tipo T com nome name e retorna um apontador para o respetivo cliente referente ao Recurso criado. O construtor do tipo T recebe uma lista de argumentos em args.
- template <typename T, typename ... Args>
  idp_t **CreateRemote**(idp_t ctx, std::string const& name, Args... args): cria um Recurso do tipo T no contexto identificado por ctx, num Domínio remoto, com nome name e retorna o identificador do Recurso criado.
- template <typename T, typename ... Args>
  idp_t **Create**(idp_t ctx, std::string const& name, Args... args): cria um Recurso do tipo T no contexto identificado por ctx, num Domínio local/remoto, com nome name e retorna o identificador do Recurso criado. O construtor do tipo T recebe uma lista de argumentos em args.
- template <typename T>
  std::unique_ptr<T> **CreateReference**(idp_t idp, idp_t ctx, std::string const& name): cria uma réplica no Domínio local, no contexto identificado por ctx, do Recurso com tipo T identificado por idp e com o nome name.
- template <typename T, typename ... Args>
  std::unique_ptr<T> **CreateCollective**(idp_t ctx, std::string const& name, unsigned int total_members, Args...args): equivalente ao Create, chamado de forma coletiva por um total de total_members Recursos ativos (com elemento executor). O Recurso é criado na Domínio do primeiro executor a chamar esta função, obtendo todos os participantes uma referencia local para o cliente.
- template <typename T, typename ... Args>
  std::unique_ptr<T> **CreateCollective**(idp_t active_rsc_idp, idp_t clos, idp_t ctx, std::string const& name, Args... args): equivalente ao Create chamado de forma coletiva por todos os membros do Recurso identificado por clos, na qual todos os participantes terão uma referencia local do Recurso. O Recurso é criado na Domínio do primeiro executor a chamar esta função, obtendo todos os participantes uma referencia local para o cliente.
- template <typename T, typename ... Args>
  auto **Run**(idp_t idp, Args... args): usado para correr, local ou remotamente, a função com tipo T do elemento executor do Recurso identificado por idp. Os argumentos args que têm de coincidir com a assinatura da função. Este método é assíncrono e retorna um futuro com o valor de retorno da função.
- idp_t **Spawn**(std::string const& ctx, unsigned int npods, idp_t parent, std::string const& mod, std::vector<std::string> const& args, std::vector<std::string> const& hosts): usado para lançar dinamicamente um total de npods novos pods/Domínios no contexto de arranque ctx, nas máquinas em hosts utilizando um algoritmo de escalonamento round-robin. O módulo mod, carregado dinamicamente, contém a função de entrada do

módulo que irá ser executada com os argumentos em args. Para a identificação das máquinas em hosts pode ser usado um endereço IP ou nome DNS.

C.2 ORGANIZADOR

O elemento organizador cria as condições para o estabelecimento de árvores de dependências, que agregam Recursos, de alguma forma relacionados, atribuindo-lhes identificadores de membro locais (idm) e um nome no contexto de um outro Recurso, designado por Recurso ascendente. Na árvore de dependências, cada um dos Recursos que possuem um elemento do tipo organizador dá origem a uma nova sub-árvore.

Este elemento surge em duas versões, muito semelhantes ao nível do interface: dinâmico (DynamicOrganizer) e estático (StaticOrganizer). O primeiro permite a adesão/saída dinâmica de Recursos a qualquer momento, enquanto que o segundo é criado de uma só assentada, com um conjunto inicial fechado de Recursos membros.

A existência de organizadores estáticos está intrinsecamente ligada à necessidade de garantir a constância de um grupo, com uma aridade que se mantém fixa ao longo do tempo de vida do Recurso, condição fundamental para a realização de operações coletivas; por exemplo, durante o arranque de uma aplicação paralela que obriga à instanciação simultânea de múltiplos Domínios, ou de comunicação em grupo fechado.

C.2.1 *Organizador Dinâmico*

O construtor deste tipo de elementos inclui um parâmetro (opcional), como um caminho no sistema de ficheiros para um módulo de biblioteca dinâmica. O módulo (se existir) é carregado em tempo de execução, disponibilizam um conjunto de funções que podem ser evocadas por Recursos activos, i.e. com elemento executor.

`class DynamicOrganizer`

- `**DynamicOrganizer**(idp_t idp, std::string const& module)`: construtor que cria no Recurso, com identificador idp, um elemento organizador dinâmico e, eventualmente, carrega a biblioteca dinâmica module que integra na aplicação.
- `void **Join**(idp_t idp, std::string const& name)`: junta o Recurso identificado por idp ao organizador dinâmico, ao qual é atribuído o nome name e o identificador local (idm) no organizador.
- `void **Leave**(idp_t idp)`: desassocia do organizador o Recurso identificado por idp.
- `std::string const& **GetModuleName**() const`: retorna nome do módulo no organizador.

Organizador Estático

O organizador estático inclui informação especifica que permite distinguir se foi criado no contexto inicial de arranque da aplicação, parent==0, ou em tempo de execução e, neste caso, parent tomo valor do idp do Recurso ativo responsável pela criação do próprio Recurso, como é o caso do lançamento dinâmico de novos Domínios.

class StaticOrganize

```
class StaticOrganizer
```

- StaticOrganizer(idp_t idp, unsigned int members, idp_t parent): construtor que cria no Recurso, com identificador idp, um elemento organizador com o total de members membros e o identificador do Recurso parent que chamou a função.
- void **Join**(idp_t idp, std::string const& name): associa o Recurso ativo identificado por idp do organizador estático atribuindo o nome name e um identificador idm local. É uma operação é coletiva que deverá ser chamada um total de members vezes por diferentes Recursos ativos.
- void **Leave**(idp_t idp): desassocia o Recurso identificado por idp do organizador correspondente. Esta operação é coletiva, pelo que deverá ser chamada um total de members vezes por diferentes Recursos ativos.
- idp_t **GetParent**() const: usado para obter o identificador principal do Recurso que criou o Recurso deste organizador.

C.2.2 *Métodos comuns*

De seguida, são apresentados os métodos comuns a ambos os elementos organizador (dinâmico e estático).

- std::size_t **GetTotalMembers**() const: retorna o número total de membros do organizador correspondente.
- std::vector<idp_t> **GetMemberList**() const: retorna a lista dos identificadores principais dos Recursos que fazem parte do organizador correspondente.
- idp_t **GetIdp**(idm_t idm) const: retorna o identificador principal do Recurso com base no identificador de membro idm.
- idp_t **GetIdp**(std::string const& name) const: retorna o identificador principal do Recurso com base no nome name.
- idm_t **GetIdm**(idp_t idp) const: retorna o identificador de membro do Recurso com base no identificador principal idp.
- idm_t **GetIdm**(std::string const& name) const: retorna o identificador de membro do Recurso com base no nome name.

C.3 EXECUTOR

O elemento executor corresponde à definição de uma função template que será executada por um fio de execução HPX de acordo com a respetiva assinatura que especifica os tipos de dados do retorno e dos parâmetros de invocação. Existem duas possibilidades para criar um executor, ou diretamente com o nome da função, ou acrescentando o nome do módulo ao nome da função.

```
template <typename R, typename ... P>
class Executor<R(P...)>
```

- `Executor(idp_t idp, std::string const& module, std::string const& function)`: cria o elemento executor do Recurso com identificador idp que irá carregar e executar uma função com nome function da biblioteca dinâmica module e assinatura `R(P...)`.
- `Executor(idp_t idp, std::function<R(P...)> const& f)`: cria o elemento executor do Recurso com identificador idp, que irá executar a função f recebida como parâmetro.
- `template <typename ... Args>`
  `hpx::future<R> Run(Args&&... args)`: executa a função template registada no executor os args correspondentes à respetiva assinatura. Retorna, imediatamente, um futuro de tipo R com o valor de retorno.
- `void ChangeIdp(idp_t idp)`: troca a identidade do executor para idp.
- `void ResumeIdp()`: restaura a identidade original do executor.
- `idp_t CurrentIdp() const`: retorna o identificador atual do Recurso.
- `idp_t OriginalIdp() const`: retorna o identificador do Recurso na origem da criação do executor.

C.4 CAIXA-POSTAL

O elemento caixa-postal é usado para o envio/receção de mensagens entre Recursos que possuam este elemento. Possui a possibilidade do envio de mensagens para um ou mais destinos, desde que os seus identificadores sejam conhecidos. Também possui primitivas de comunicação por contexto, que permitem difundir mensagens para um Recurso com um elemento organizador estático, através do idp do mesmo, ou então o envio/receção de mensagens utilizando o idp do Recurso clausura e o idm no contexto do mesmo. Atualmente, as primitivas de comunicação disponibilizadas são bloqueantes.

O elemento caixa-postal é usado para o envio/receção de mensagens ponto-a-ponto entre Recursos que incluem este tipo de elemento, sendo o(s) destinatário(s) identificados pelo respetivo idp. É, também, possível a comunicação por contexto (grupo estático) em que o destinatário/receptor é identificado pelo par <contexto,idm>.

`class Mailbox`

- `Mailbox(idp_t idp)`: construtor de um elemento caixa-postal a integrar no Recurso idp.
- `void Send(idp_t dest, Message const& msg) const`: utilizado para enviar a mensagem msg para o Recurso dest.
- `void Send(std::vector<idp_t> const& dests, Message const& msg) const`: envia a mensagem msg para uma lista de destinatários dests.
- `Message Receive() const`: utilizado para receber uma mensagem da caixa-postal do Recurso.
- `Message Receive(idp_t source) const`: recebe uma mensagem do Recurso com origem em source.
- `void Broadcast(idp_t clos, Message const& msg) const`: difunde uma mensagem msg para todos os membros do Recurso clos com elemento organizador estático.

- `void` **`Send`**`(idm_t rank, idp_t clos, Message const& msg) const`: envia uma mensagem msg para o Recurso identificado por rank, no contexto do Recurso clos com elemento organizador estático.
- `Message` **`Receive`**`(idm_t rank, idp_t clos) const`: recebe uma mensagem enviada pelo Recurso rank, no contexto do Recurso com elemento organizador estático identificado por clos.

C.5 MENSAGEM

A classe mensagem é uma classe auxiliar do elemento caixa-postal, cujas instâncias incluem um cabeçalho, formado por uma etiqueta e o idp do Recurso na origem, e um tampão de memória para os dados. É possível adicionar e obter dados de uma mensagem, desde que os respetivos tipos sejam serializáveis pelo HPX.

```
class Message
```

- **`Message`**`()`: construtor padrão de uma mensagem.
- `std::size_t` **`Size`**`() const`: retorna o tamanho da mensagem.
- `void` **`Clear`**`()`: utilizado para apagar o conteúdo da mensagem.
- `std::uint16_t` **`Type`**`() const`: retorna o tipo da mensagem.
- `void` **`SetType`**`(std::uint16_t type)`: utilizado para modificar o tipo (etiqueta) da mensagem.
- `idp_t` **`Sender`**`() const`: retorna a identificação do Recurso que enviou a mensagem.
- `void` **`SetSender`**`(idp_t)`: utilizado para inserir o identificador do emissor na mensagem.
- `template <typename T>`
  `T` **`Get`**`(std::size_t index = 0) const`: obtém do tampão dos dados da mensagem, o conteúdo do tipo `T` localizado na posição index.
- `template <typename T>`
  `void` **`Add`**`(T const& data)`: adiciona ao tampão dos dados da mensagem o conteúdo da variável data do tipo `T`.

C.6 VALOR

O valor é uma classe template usada para albergar uma instância parametrizada de um determinado tipo de dados. O programador pode criar as suas próprias classes de dados, desde que, tal como no caso da mensagem, sejam serializáveis.

```
template <typename T>
class Value
```

- **`Value`**`(idp_t idp, Args... args)`: construtor que cria o elemento valor do tipo `T` no Recurso com identificador idp, com o estado inicial especificado através de args.

- `T*` **`Get`**`()`: retorna um apontador para os dados do utilizador do tipo T no elemento valor. Depois de retornado o apontador, não é assegurado a consistência dos dados, sendo recomendável aquisição do direito de escrita, disponibilizado pela API do Dado.
- `T` **`Fetch`**`() const`: retorna uma cópia dos dados do utilizador do tipo `T` albergado pelo elemento valor. Para assegurar a consistência dos dados, enquanto a função é chamada através da API do Dado, o Recurso não permite escritas, unicamente leituras.

C.7 SINCRONIZADOR

O elemento sincronizador proposto em CoRes, desdobra-se nas seguintes variantes: barreira (SBarrier), guarda (SMutex) e guarda para leituras/escritas (SRWMutex) que atuam como estruturas de sincronização distribuídas.

C.7.1 *Barreira*

A barreira, tal como o nome indica, fornece a capacidade de sincronização distribuída do conjunto dos membros de um organizador estático.

```
class SBarrier
```

- **`SBarrier`**`(idp_t idp, idp_t clos)`: construtor do elemento barreira do Recurso identificado por idp, usado como estrutura de sincronização distribuída dos Recursos ativos associados ao organizador estático clos.
- `void` **`Synchronize`**`()`: operação coletiva que só termina quando invocada por todo os Recursos ativos presentes no organizador estático implícito.

C.7.2 *Guarda*

A guarda serve para criar uma região de código de exclusão mútua.

```
class SMutex
```

- **`SMutex`**`(idp_t idp)`: construtor da guarda do Recurso identificado por idp.
- `void` **`Acquire`**`()`: sinaliza a entrada numa região de exclusão mútua.
- `void` **`Release`**`()`: sinaliza a saída de uma região de exclusão mútua

C.7.3 *Guarda de leitura/escrita*

A guarda leitura/escrita serve para criar uma região de exclusão mútua distinguindo as leituras das escritas.

```
class SRWMutex
```

- **`SRWMutex`**`(idp_t idp)`: construtor da guarda do Recurso identificado por idp.
- `void` **`AcquireRead`**`()`: sinaliza a entrada numa região de exclusão mútua para leituras.
- `void` **`ReleaseRead`**`()`: sinaliza a saída de uma região de exclusão mútua para leituras.
- `void` **`AcquireWrite`**`()`: sinaliza a saída de uma região de exclusão mútua para escritas.
- `void` **`ReleaseWrite`**`()`: sinaliza a saída de uma região de exclusão mútua para escritas.

C.8 EXECUTORPOOL

O elemento ExecutorPool possui capacidades de computação, similarmente ao Recurso ProtoAgent, mas com a capacidade de registar a todo momento uma nova função cuja execução é processada em paralelo por um coletivo de tarefas, como se de um "omp parallel"se tratasse. Disponibiliza funcionalidades para paralelizar ciclos, análogo ao "omp parallel for", com escalonamento estático, dinâmico e *guided*.

```
class ExecutorPool
```

- **ExecutorPool**(idp_t idp, std::size_t num_hpx_threads) - Construtor do elemento ExecutorPool do Recurso identificado por idp. Recebe como argumentos o número de tarefas que irão ficar alocadas.
- int **GetRank**() - rank da tarefa.
- int **GetNumThreads**() - numero total de tarefas do alocadas.
- std::pair<int,int> **ScheduleStatic**(int Beg, int End) - método de escalonamento estático para processamento paralelo de ciclos. Fornecendo os índices inicial e final da gama de iterações de um ciclo, retorna um par com os índices inicial e final atribuídos à tarefa correspondente.
- std::vector<std::pair<int,int> > **ScheduleStatic**(int Beg, int End, int chunk) - método de escalonamento estático com *chunk* para processamento paralelo de ciclos. Fornecendo os índices inicial e final da gama de iterações de um ciclo, retorna um vetor de pares com os índices inicial e final atribuídos à tarefa correspondente.
- std::pair<int,int> **ScheduleDynamic**(int Beg, int End, int chunk) - método de escalonamento dinâmico com *chunk* para processamento paralelo de ciclos. Iterativamente, chamando esta função fornecendo os índices inicial e final da gama de iterações de um ciclo, retorna um par com os índices inicial e final atribuídos à tarefa correspondente.
- std::pair<int,int> **ScheduleGuided**(int Beg, int End, int chunk) - método de escalonamento *guided* com *chunk* para processamento paralelo de ciclos. Iterativamente, chamando esta função fornecendo os índices inicial e final da gama de iterações de um ciclo, retorna um par com os índices inicial e final atribuídos à tarefa correspondente.
- template < typename ... Args >
  void **Dispatch**(hpx::function<void(Args...)> func, Args ... args) - Executa a função func com os argumentos args em paralelo pelas tarefas constituintes do ExecutorPool.
- void Wait() - função de sincronização na barreira interna de ExecutorPool.

C.9 SUNICHANNEL

Elemento de comunicação para troca de dados entre dois pontos. Chamas-se "uni"porque é construído com um só LCO `hpx::channel`; contudo, os dados podem ser enviados nas duas direções. Para a sua utilização, na troca de dados entre dois agentes, ambos os parceiros tem de usar o mesmo Recurso que contém SUniChannel - um cria o Recurso e o outro uma réplica.

`class SUniChannel`

- **`SUniChannel`**`(idp_t idp)`: construtor do SUniChannel do Recurso identificado por idp.
- `void` **`Set`**`(T&& t, std::size_t step)`: envia uma mensagem com tipo `T`, com índice "step", de forma assincrona.
- `hpx::future<T>` **`Get`**`(std::size_t step)`: recebe uma mensagem com tipo `T`, com índice "step".

C.10 SMULTICHANNEL

Expande o elemento SUniChannel para troca de dados entre vários interlocutores. Simplifica a programabilidade de algoritmos em que seja necessário trocar dados entre vários agentes. Cada Recurso pode conter tantos canais quanto deseje. Cria, internamente, dois canais de comunicação `hpx::channel` entre cada parceiro, tornado possível a troca de dados em ambas as direções, sem conflitos. Cada interlocutor, para comunicar com os restantes, tem de criar um novo Recurso que contenha SMultiChannel. Os canais conectam-se através dos nomes fornecidos no construtor.

`class SMultiChannel`

- `template <typename ...Args>`
  `SMultiChannel``(idp_t idp, std::string const& myself, Args ... args)`: construtor do SMultiChannel do Recurso identificado por idp, que é identificado também por "myself"para registo dos canais para comunicação com cada um dos parceiros recebidos em args.
- `void` **`Set`**`(T&& t, std::string const& partner, std::size_t step)`: envia uma mensagem com tipo `T`, com índice "step", para o parceiro identificado por "partner".
- `hpx::future<T>` **`Get`**`(std::string const& partner, std::size_t step)`: recebe uma mensagem com tipo `T`, com índice "step", do parceiro "partner".

D

API DO COR-HPX

Disponibiliza-se a seguir a API do CoR-HPX.

Notas adicionais:

- O Recurso Domain difere dos outros Recursos porque está diretamente associado ao Pod e, por consequência, a uma localidade. Para criar um novo Domain é necessário recorrer à função `Spawn` (Domain), ao contrário dos outros Recursos que são criados através de funções como `CreateLocal` ou `Create`.
- Os construtores dos Recursos apresentados na API servem unicamente para disponibilizar a assinatura com os parâmetros. Como referido no ponto anterior, os Recursos devem ser construídos recorrendo às funções indicadas para o efeito, no Domain.
- Todas as funções disponíveis na API podem ser executadas assincronamente. Para tal, basta unicamente inserir `hpx::launch::async` como primeiro argumento. O valor de retorno é encapsulado num futuro.

D.1 FUNÇÕES GLOBAIS

- `std::shared_ptr<cor::Domain_Client>` **`cor::GetDomain`**`()` - função utilizada para obter um apontador para o Domínio local. Apenas deve ser chamada pelos fios de execução dos Recursos ativos.

D.2 DOMAIN

O Recurso Domain delimita uma região do espaço equivalente a um processo/localidade que suporta a representação, computação e comunicação entre Recursos locais ou remotos. Disponibiliza ao utilizador a capacidade de interagir com o Pod, nomeadamente criar novos Recursos, executar funções remotas e lançar novos Pods/-Domínios, enquanto que oferece também as caraterísticas do Recurso Group, para agrupar outros Recursos.

É encapsulado pela classe componente cliente Domain_Client e disponibiliza a seguinte interface de utilização:

- **Domain_Client**`(std::string const& module)` - Construtor do componente cliente Domain_Client, que por sua vez constrói o Recurso, e componente servidor, Domain. Este construtor não é público. Recursos do tipo Domain são construídos automaticamente assim que um novo Pod for criado ou a função "spawn"chamada.
- `idp_t` **IdpGlobal**`()` - idp global do Recurso.
- `idp_t` **Idp**`()` - idp local do Recurso.
- `hpx::id_type` **GetLocalityGID**`()` - GID da localidade HPX do Recurso.
- `unsigned int` **GetLocalityID**`()` - ID da localidade HPX do Recurso.
- `void` **Join**`(idp_t idp, std::string const& name)` - junta o Recurso identificado por idp ao Domínio, ao qual é atribuído o nome name e um identificador idm locais.
- `void` **Leave**`(idp_t idp)` - desassocia do Domínio o Recurso identificado por idp.
- `std::string` **GetModuleName**`()` - retorna nome do módulo associado ao Domínio.
- `std::size_t` **GetTotalMembers**`()` - retorna o número total de membros do Domínio.
- `std::vector<idp_t>` **GetMemberList**`()` - retorna a lista dos identificadores principais dos Recursos que fazem parte do Domínio.
- `idp_t` **GetIdp**`(idm_t idm)` - retorna o identificador principal do Recurso com identificador de membro idm, que faz parte do Domínio.
- `idp_t` **GetIdp**`(std::string const& name)` - retorna o identificador principal do Recurso com nome name, que faz parte do Domínio.
- `idm_t` **GetIdm**`(idp_t idp)` - retorna o identificador de membro do Recurso com identificador principal idp, que faz parte do Domínio.
- `idm_t` **GetIdm**`(std::string const& name)` - retorna o identificador de membro do Recurso com nome name, que faz parte do Domínio.
- `std::string` **GetGlobalContext**`()` - retorna o nome do contexto global da aplicação.
- `std::string` **GetLocalContext**`()` - retorna o nome do contexto de arranque do Domínio.
- `unsigned int` **GetNumPods**`()` - retorna o número total de pods presentes na aplicação.
- `unsigned int` **GetNumDomains**`()` - retorna o número total de Domínios na aplicação.
- `std::vector<idp_t>` **GetPods**`()` - retorna um vetor com os idps dos pods da aplicação.
- `std::vector<idp_t>` **GetDomains**`()` - retorna um vetor com os idps dos Domínios da aplicação.
- `std::vector<idp_t>` **GetRemotePods**`()` - retorna um vetor com os idps dos pods remotos da aplicação.
- `std::vector<idp_t>` **GetRemoteDomains**`()` - retorna um vetor com os idps dos Domínios remotos da aplicação.
- `idp_t` **GetActiveResourceIdp**`()` - retorna o identificador do Recurso ativo.
- `idp_t` **GetPredecessorIdp**`(idp_t idp)` - retorna o ascendente do Recurso identificado por idp.

- template <typename T>
  std::unique_ptr<T> **GetLocalResource**(idp_t idp) - retorna um apontador para o Recurso identificador por idp, com tipo T.
- template <typename T, typename ... Args>
  std::unique_ptr<T> **CreateLocal**(idp_t ctx, std::string const& name, Args ... args) - cria no Domínio local e no contexto ctx um Recurso do tipo T, com argumentos args e com nome name, retornando um apontador para o Recurso criado.
- template <typename T, typename ... Args>
  idp_t **Create**(idp_t ctx, std::string const& name, Args ... args) - cria um Recurso do tipo T no contexto identificado por ctx, num Domínio local/remoto, com argumentos args e com nome name, retornando o identificador do Recurso criado.
- template <typename T, typename ... Args>
  idp_t **CreateRemote**(idp_t ctx, std::string const& name, Args ... args) - cria um Recurso do tipo T no contexto identificado por ctx, num Domínio remoto, com argumentos args e com nome name, retornando um identificador do Recurso criado.
- template <typename T, typename ... Args>
  std::unique_ptr<T> **CreateReference**(idp_t idp, idp_t ctx, std::string const& name) - cria uma réplica no Domínio local, no contexto identificado por ctx, do Recurso com tipo T identificado por idp e com o nome name.
- template <typename T, typename ... Args>
  std::unique_ptr<T> **CreateCollective**(idp_t ctx, std::string const& name, unsigned int total_members, Args ... args) - equivalente ao Create, chamado de forma coletiva por um total de total_members Recursos ativos (com elemento executor). O Recurso é criado na Domínio do primeiro executor que chamar esta função, obtendo todos os participantes uma referencia local para ele.
- template <typename T, typename ... Args>
  std::unique_ptr<T> **CreateCollective**(idp_t clos, idp_t ctx, std::string const& name, Args ... args) - equivalente ao Create chamado de forma coletiva por todos os membros do Recurso identificado por clos, na qual todos os participantes terão uma referencia local do Recurso. O Recurso é criado na Domínio do primeiro executor a chamar esta função, obtendo todos os participantes uma referencia local para ele.
- template <typename T, typename ... Args>
  auto **Run**(idp_t idp, Args... args) - usado para correr, local ou remotamente, a função com tipo T do elemento executor do Recurso identificado por idp. Os argumentos args têm de coincidir com a assinatura da função. Este método é assíncrono e retorna um futuro com o valor de retorno da função.
- idp_t **Spawn**(std::string const& ctx, unsigned int npods, std::string const& module, std::vector<std::string> const& args, std::vector<std::string> const& hosts) - usado para lançar dinamicamente um total

de npods novos pods/Domínios no contexto de arranque ctx, nas máquinas em hosts utilizando um algoritmo de escalonamento round-robin. O módulo mod, carregado dinamicamente, contém a função de entrada do módulo que irá ser executada com os argumentos em args. Para a identificação das máquinas em hosts pode ser um endereço IP ou DNS.

D.3 GROUP

O Recurso Group pertence a classe de Recursos estruturados, na medida em que organiza outros Recursos. É um grupo dinâmico acessível através de operações de adesão (join) e abandono (leave). Os seus membros são identificados no contexto por um identificador (idm) e por um nome.

É encapsulado pela classe componente cliente Group_Client e disponibiliza a seguinte interface de utilização:

- `Group_Client(std::string const& module)` - Construtor do componente cliente Group_Client, que por sua vez constrói o Recurso, e componente servidor, Group. Tem como argumentos o nome do modulo que irá carregar dinamicamente.
- `idp_t IdpGlobal()` - idp global do Recurso.
- `idp_t Idp()` - idp local do Recurso.
- `hpx::id_type GetLocalityGID()` - GID da localidade HPX do Recurso.
- `unsigned int GetLocalityID()` - ID da localidade HPX do Recurso.
- `void Join(idp_t idp, std::string const& name)` - junta o Recurso identificado por idp ao grupo, ao qual é atribuído o nome name e um identificador idm locais.
- `void Leave(idp_t idp)` - desassocia do grupo o Recurso identificado por idp.
- `std::string GetModuleName()` - retorna nome do módulo associado ao Domínio.
- `std::size_t GetTotalMembers()` - retorna o número total de membros do Domínio.
- `std::vector<idp_t> GetMemberList()` - retorna a lista dos identificadores principais dos Recursos que fazem parte do grupo.
- `idp_t GetIdp(idm_t idm)` - retorna o identificador principal do Recurso com identificador de membro idm, que faz parte do grupo.
- `idp_t GetIdp(std::string const& name)` - retorna o identificador principal do Recurso com nome name, que faz parte do grupo.
- `idm_t GetIdm(idp_t idp)` - retorna o identificador de membro do Recurso com identificador principal idp, que faz parte do grupo.
- `idm_t GetIdm(std::string const& name)` - retorna o identificador de membro do Recurso com nome name, que faz parte do grupo.

D.4 CLOSURE

O Recurso Closure é também um Recurso estruturado semelhante ao Group, mas os seus membros são estáticos. É adequado para a realização de operações coletivas entre os seus membros, obrigatoriamente do tipo Agent.

É encapsulado pela classe componente cliente Clousure_Client e disponibiliza a seguinte interface de utilização:

- **`Closure_Client`**`(unsigned int total_members, idp_t parent);` - Construtor do componente cliente Closure_Client, que por sua vez constrói o Recurso, e componente servidor, Closure. Tem como argumentos o número de Recursos ativos e o idp do Recurso que chamou o construtor.
- `idp_t` **`IdpGlobal`**`()` - idp global do Recurso.
- `idp_t` **`Idp`**`()` - idp local do Recurso.
- `hpx::id_type` **`GetLocalityGID`**`()` - GID da localidade HPX do Recurso.
- `unsigned int` **`GetLocalityID`**`()` - ID da localidade HPX do Recurso.
- `void` **`Join`**`(idp_t idp, std::string const& name)` - associa o Recurso ativo identificado por idp à clausura, atribuindo o nome name e um identificador idm local. É uma operação é coletiva que deverá ser chamada um total de members vezes por diferentes Recursos ativo.
- `void` **`Leave`**`(idp_t idp)` - desassocia o Recurso identificado por idp da clausura. Esta operação é coletiva, pelo que deverá ser chamada um total de members vezes por diferentes Recursos ativos.
- `idp_t` **`GetParent`**`()` - usado para obter o identificador principal do Recurso ativo que criou a clausura.
- `std::size_t` **`GetTotalMembers`**`()` - retorna o número total de membros da clausura efetivos.
- `std::size_t` **`GetFixedTotalMembers`**`()` - retorna o número total de membros da clausura predefinidos pelo construtor.
- `std::vector<idp_t>` **`GetMemberList`**`()` - retorna a lista dos identificadores principais dos Recursos que fazem parte da clausura.
- `idp_t` **`GetIdp`**`(idm_t idm)` - retorna o identificador principal do Recurso com identificador de membro idm, que faz parte da clausura.
- `idp_t` **`GetIdp`**`(std::string const& name)` - retorna o identificador principal do Recurso com nome name, que faz parte da clausura.
- `idm_t` **`GetIdm`**`(idp_t idp)` - retorna o identificador de membro do Recurso com identificador principal idp, que faz parte da clausura.
- `idm_t` **`GetIdm`**`(std::string const& name)` - retorna o identificador de membro do Recurso com nome name, que faz parte da clausura.

D.5 PROTOAGENT

O Recurso ProtoAgent tem a capacidade de executar uma função de qualquer tipo, registada no momento da sua criação.

É encapsulado pela classe componente cliente ProtoAgent_Client e disponibiliza a seguinte interface de utilização:

- **ProtoAgent_Client**(std::function<R(P...)> const& f) - Construtor do componente cliente ProtoAgent_Client, que por sua vez constrói o Recurso, e componente servidor, ProtoAgent. Recebe como argumentos a função a executar.
- **ProtoAgent_Client**(std::string const& module, std::string const& function) - Construtor do componente cliente ProtoAgent_Client, que por sua vez constrói o Recurso, e componente servidor, ProtoAgent. Recebe como argumentos o modulo que irá carregar dinamicamente e o nome da função a executar, contida no modulo.
- idp_t **IdpGlobal**() - idp global do Recurso.
- idp_t **Idp**() - idp local do Recurso.
- hpx::id_type **GetLocalityGID**() - GID da localidade HPX do Recurso.
- unsigned int **GetLocalityID**() - ID da localidade HPX do Recurso.
- template <typename ... Args> hpx::future<R> **Run**(Args&&... args) - executa a função template registada no executor os args correspondentes à respetiva assinatura. Retorna, imediatamente, um futuro de tipo R que corresponde ao valor de retorno.
- void **ChangeIdp**(idp_t idp) - troca a identidade do executor para idp.
- void **ResumeIdp**() - restaura a identidade original do executor.
- idp_t **CurrentIdp**() - restaura a identidade original do executor.
- idp_t **OriginalIdp**() - retorna o identificador atual do Recurso.
- idp_t **GetExecutorIdp**() - retorna o identificador do elemento executor na origem da criação do Recurso.

D.6 AGENT

O Recurso Agent partilha com o ProtoAgent a capacidade de executar uma função de qualquer tipo, registada no momento da criação. O elemento adicional Mailbox transforma-o numa entidade comunicante capaz de receber/enviar mensagens através de um protocolo de comunicação ponto-a-ponto entre Recursos.

É encapsulado pela classe componente cliente Agent_Client e disponibiliza a seguinte interface de utilização:

- **Agent_Client**(std::function<R(P...)> const& f) - Construtor do componente cliente Agent_Client, que por sua vez constrói o Recurso, e componente servidor, Agent. Recebe como argumentos a função a executar.
- **Agent_Client**(std::string const& module, std::string const& function) - Construtor do componente cliente Agent_Client, que por sua vez constrói o Recurso, e componente servidor, Agent. Recebe como argumentos o modulo que irá carregar dinamicamente e o nome da função a executar, contida no modulo.

- `idp_t` **`IdpGlobal`**`()` - idp global do Recurso.
- `idp_t` **`Idp`**`()` - idp local do Recurso.
- `hpx::id_type` **`GetLocalityGID`**`()` - GID da localidade HPX do Recurso.
- `unsigned int` **`GetLocalityID`**`()` - ID da localidade HPX do Recurso.
- `template <typename ... Args>`
  `hpx::future<R>` **`Run`**`(Args&&... args)` - executa a função template registada no executor os args correspondentes à respetiva assinatura. Retorna, imediatamente, um futuro de tipo R que corresponde ao valor de retorno.
- `void` **`ChangeIdp`**`(idp_t idp)` - troca a identidade do executor para idp.
- `void` **`ResumeIdp`**`()` - restaura a identidade original do executor.
- `idp_t` **`CurrentIdp`**`()` - restaura a identidade original do executor.
- `idp_t` **`OriginalIdp`**`()` - retorna o identificador atual do Recurso.
- `idp_t` **`GetExecutorIdp`**`()` - retorna o identificador do Recurso na origem da criação do executor.
- `void` **`Send`**`(idp_t dest, Message const& msg) const`: utilizado para enviar a mensagem msg para o Recurso dest.
- `void` **`Send`**`(std::vector<idp_t> const& dests, Message const& msg) const`: envia a mensagem msg para uma lista de destinatários dests.
- `Message` **`Receive`**`() const`: utilizado para receber uma mensagem.
- `Message` **`Receive`**`(idp_t source) const`: recebe uma mensagem do Recurso com origem em source.
- `void` **`Broadcast`**`(idp_t clos, Message const& msg) const`: difunde uma mensagem msg para todos os membros do Recurso clausura clos.
- `void` **`Send`**`(idm_t rank, idp_t clos, Message const& msg) const`: envia uma mensagem msg para o Recurso identificado por rank, no contexto do Recurso clausura clos.
- `Message` **`Receive`**`(idm_t rank, idp_t clos) const`: recebe uma mensagem enviada pelo Recurso rank, no contexto do Recurso clausura clos.

D.7 OPERON

O Recurso Operon possui capacidades de computação e comunicação, similarmente ao Recurso Agent, mas com a capacidade de registar a todo momento uma nova função cuja execução é processada em paralelo por um coletivo de tarefas, como se de um "omp parallel"se tratasse. O operão também disponibiliza funcionalidades para paralelizar ciclos, análogo ao "omp parallel for", com escalonamento estático, dinâmico e *guided*.

É encapsulado pela classe componente cliente Operon_Client e disponibiliza a seguinte interface de utilização:

- **`Operon_Client`**`(std::size_t num_hpx_threads)` - Construtor do componente cliente Operon_Client, que por sua vez constrói o Recurso, e componente servidor, Operon. Recebe como argumentos o número de tarefas que irão ficar alocadas.

- idp_t **IdpGlobal**() - idp global do Recurso.
- idp_t **Idp**() - idp local do Recurso.
- hpx::id_type **GetLocalityGID**() - GID da localidade HPX do Recurso.
- unsigned int **GetLocalityID**() - ID da localidade HPX do Recurso.
- int **GetRank**() - rank da tarefa.
- int **GetNumThreads**() - numero de tarefas do operão.
- std::pair<int,int> **ScheduleStatic**(int Beg, int End) - método de escalonamento estático para processamento paralelo de ciclos. Fornecendo os índices inicial e final da gama de iterações de um ciclo, retorna um par com os índices inicial e final atribuídos à tarefa correspondente.
- std::vector<std::pair<int,int» **ScheduleStatic**(int Beg, int End, int chunk) - método de escalonamento estático com *chunk* para processamento paralelo de ciclos. Fornecendo os índices inicial e final da gama de iterações de um ciclo, retorna um vetor de pares com os índices inicial e final atribuídos à tarefa correspondente.
- std::pair<int,int> **ScheduleDynamic**(int Beg, int End, int chunk) - método de escalonamento dinâmico com *chunk* para processamento paralelo de ciclos. Iterativamente, chamando este função fornecendo os índices inicial e final da gama de iterações de um ciclo, retorna um par com os índices inicial e final atribuídos à tarefa correspondente.
- std::pair<int,int> **ScheduleGuided**(int Beg, int End, int chunk) - método de escalonamento *guided* com *chunk* para processamento paralelo de ciclos. Iterativamente, chamando este função fornecendo os índices inicial e final da gama de iterações de um ciclo, retorna um par com os índices inicial e final atribuídos à tarefa correspondente.
- template < typename Func, typename ... Args >
  hpx::future<void> **Dispatch**(Func&& func, Args&&... args) - Executa a função func com os argumentos args em paralelo. O numero de tarefas é definido no construtor.
- void **Send**(idp_t dest, Message const& msg) const: utilizado para enviar a mensagem msg para o Recurso dest.
- void **Send**(std::vector<idp_t> const& dests, Message const& msg) const: envia a mensagem msg para uma lista de destinatários dests.
- Message **Receive**() const: utilizado para receber uma mensagem.
- Message **Receive**(idp_t source) const: recebe uma mensagem do Recurso com origem em source.
- void **Broadcast**(idp_t clos, Message const& msg) const: difunde uma mensagem msg para todos os membros do Recurso clausura clos.
- void **Send**(idm_t rank, idp_t clos, Message const& msg) const: envia uma mensagem msg para o Recurso identificado por rank, no contexto do Recurso clausura clos.
- Message **Receive**(idm_t rank, idp_t clos) const: recebe uma mensagem enviada pelo Recurso rank, no contexto do Recurso clausura clos.

D.8 DATA

O Recurso Data representa um tipo de Dados arbitrário, como uma classe nativa ou uma classe simples/com herança/template definida pelo utilizador, com a capacidade de migrar e de ser acedido remotamente.

É encapsulado pela classe componente cliente Data_Client e disponibiliza a seguinte interface de utilização:

- `Data_Client(Args&&... args)` - Construtor do componente cliente Data_Client, que por sua vez constrói o Recurso, e componente servidor, Data. Recebe como argumentos os argumentos do construtor do objeto que irá construir.
- `idp_t IdpGlobal()` - idp global do Recurso.
- `idp_t Idp()` - idp local do Recurso.
- `hpx::id_type GetLocalityGID()` - GID da localidade HPX do Recurso.
- `unsigned int GetLocalityID()` - ID da localidade HPX do Recurso.
- `void AcquireRead()` - sinaliza a entrada numa região de exclusão mútua para leitura.
- `void ReleaseRead()` - sinaliza a saída duma região de exclusão mútua para leitura.
- `void AcquireWrite()` - sinaliza a entrada numa região de exclusão mútua para escrita.
- `void ReleaseWrite()` - sinaliza a saída duma região de exclusão mútua para escrita.
- `T Fetch()` - retorna uma cópia dos Dados do tipo `T` albergado pelo Dado. É uma operação atómica com aquisição do direito de leitura.
- `T* Get()` - retorna um apontador para os Dados do tipo T albergado pelo Dado. Não é uma operação atómica. Para assegurar a consistência dos Dados, a chamada da função deve ser precedida da aquisição do direito de escrita. Se o Dado não estiver na localidade de onde esta função for chamada, o objeto é migrado automaticamente.
- `template <typename F, typename ... Args>`
  `auto Run(hpx::function<F> func, Args... args)` - Esta função foi desenhada para executar funções objeto sobre o objeto albergado pelo Dado, funções estas que irão ser definidas pelo utilizador para executar funções membro. Executa a função func, no Domínio do Dado, com os argumentos args. Internamente é acrescentado na primeira posição dos argumentos args a referência do objeto que o Dado alberga. Assim, o utilizador tem de criar as funções objeto com o primeiro parâmetro sendo uma referência para um objeto do tipo do Dado. Depois, na definição da função objeto, o utilizador chama a função membro que pretender no objeto recebido como argumento, objeto este passado em tempo de execução e que é o objeto albergado pelo Dado. Uma estrutura que serve de exemplo para as funções objeto pode ser observado na listagem 33.

  Para executar Run ignorando o estado do Dado, a função objeto passada como argumento tem de ter na mesma como primeiro parâmetro a referencia do objeto albergado pelo Dado, mas depois na definição o utilizador tem a liberdade de colocar o código que quiser, executando assim código no Domínio do Dado, tal como uma ação simples HPX.
- `void Migrate(idp_t domain_target)` - migra o Dado para a localidade do Domínio identificado por domain_target.

D.9 BARRIER

O Recurso Barrier serve como um coordenador de atividades paralelas de conjuntos estáticos (clausuras) de agentes/protoAgentes.

É encapsulado pela classe componente cliente Barrier_Client e disponibiliza a seguinte interface de utilização:

- **Barrier_Client**(idp_t clos) - Construtor do componente cliente Barrier_Client, que por sua vez constrói o Recurso, e componente servidor, Barrier. Recebe como argumentos o idp da clausura associada.
- idp_t **IdpGlobal**() - idp global do Recurso.
- idp_t **Idp**() - idp local do Recurso.
- hpx::id_type **GetLocalityGID**() - GID da localidade HPX do Recurso.
- unsigned int **GetLocalityID**() - ID da localidade HPX do Recurso.
- void **Synchronize**(): operação coletiva que só termina quando invocada por todos os Recursos ativos presentes na clausura implícita.

D.10 MUTEX

O Recurso Mutex é um instrumento de sincronização distribuído, usados para definir zonas de exclusão mútua, entre agentes/protoAgentes.

É encapsulado pela classe componente cliente Mutex_Client e disponibiliza a seguinte interface de utilização:

- **Mutex_Client()** - Construtor do componente cliente Mutex_Client, que por sua vez constrói o Recurso, e componente servidor, Mutex. Construtor sem argumentos.
- idp_t **IdpGlobal**() - idp global do Recurso.
- idp_t **Idp**() - idp local do Recurso.
- hpx::id_type **GetLocalityGID**() - GID da localidade HPX do Recurso.
- unsigned int **GetLocalityID**() - ID da localidade HPX do Recurso.
- void **Acquire**(): sinaliza a entrada numa região de exclusão mútua.
- void **Release**(): sinaliza a saída de uma região de exclusão mútua.

D.11 UNICHANNEL

O Recurso UniChannel é um Recurso de comunicação para troca de Dados entre dois pontos. Chama-se "uni"porque é construído com um só LCO hpx::channel; contudo, os Dados podem ser enviados nas duas direções. Para a sua utilização, troca de Dados entre dois agentes, ambos os parceiros tem de usar o mesmo Recurso UniChannel. O primeiro cria o Recurso e o segundo uma réplica.

É encapsulado pela classe componente cliente Mutex_Client e disponibiliza a seguinte interface de utilização:

- **`UniChannel_Client`**`(std::string const& type)` - Construtor do componente cliente UniChannel_Client, que por sua vez constrói o Recurso, e componente servidor, UniChannel. Recebe como argumentos o efeito a que se destina ("r", "w", "rw"): "r- ler, "w- escrever, "rw- ler e escrever.
- `idp_t` **`IdpGlobal`**`()` - idp global do Recurso.
- `idp_t` **`Idp`**`()` - idp local do Recurso.
- `hpx::id_type` **`GetLocalityGID`**`()` - GID da localidade HPX do Recurso.
- `unsigned int` **`GetLocalityID`**`()` - ID da localidade HPX do Recurso.
- `hpx::future<T>` **`Get`**`(std::size_t step)` : recebe uma mensagem assincronamente com índice step.
- `void` **`Set`**`(T&& t, std::size_t step)`: envia uma mensagem com índice step, com o método *"fire and forget"*, ou seja, assincronamente e sem aviso de receção.

D.12 MULTICHANNEL

O Recurso MultiChannel expande o Recurso UniChannel para troca de Dados entre vários interlocutores. Simplifica a programabilidade de algoritmos em que seja necessário trocar Dados entre vários agentes. Cada Recurso pode conter tantos canais quanto se deseje. Cria, internamente, dois canais de comunicação `hpx::channel` entre cada parceiro, tornado possível a troca de Dados em ambas as direções, sem conflitos. Cada interlocutor, para comunicar com os restantes, tem de criar um novo Recurso MultiChannel. Os canais conectam-se através dos nomes fornecidos no construtor.

É encapsulado pela classe componente cliente MultiChannel_Client e disponibiliza a seguinte interface de utilização:

- **`MultiChannel_Client`**`(std::string const& myself, Args ...args)` - Construtor do componente cliente MultiChannel_Client, que por sua vez constrói o Recurso, e componente servidor, MultiChannel. Recebe como argumentos o nome próprio e o os nomes dos restantes parceiros.
- `idp_t` **`IdpGlobal`**`()` - idp global do Recurso.
- `idp_t` **`Idp`**`()` - idp local do Recurso.
- `hpx::id_type` **`GetLocalityGID`**`()` - GID da localidade HPX do Recurso.
- `unsigned int` **`GetLocalityID`**`()` - ID da localidade HPX do Recurso.
- `hpx::future<T>` **`Get`**`(std::string const& partner, std::size_t step)`: recebe uma mensagem assincronamente de "partner", com índice step.
- `void` **`Set`**`(T&& t, std::string const& partner, std::size_t step)`: envia uma mensagem para partner, com índice step. A mensagem é enviada com o método *"fire and forget"*, ou seja, assincronamente e sem aviso de receção.

A presente dissertação foi desenvolvida com apoio de uma bolsa de investigação do LIP (Laboratório de Instrumentação e Física Experimental de Partículas) com a referência LIP / BI - 05/2020, no âmbito do Projeto "BigDataHEP: Understanding Big Data in High Energy Physics: finding a needle in many haystacks", com referência: Projeto ICDT, POCI-01-0145-FEDER-029147.