



Universidade do Minho

Escola de Engenharia

Departamento de Informática

José António Pereira Pinto

HpxTrace:
monitorização de aplicações em HPX

September 2022



Universidade do Minho

Escola de Engenharia

Departamento de Informática

José António Pereira Pinto

HpxTrace:
monitorização de aplicações em HPX

Dissertação de Mestrado

Mestrado Integrado em Engenharia Informática

Dissertação supervisionada por

António Manuel da Silva Pina

September 2022

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/>

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

As the complexity of the parallelism in high performance computing continues to increase, traditional tools such as MPI and OpenMP are starting to become inadequate. Within this context, the HPX C++ library emerges as the first implementation of ParalleX, an execution model that describes message-driven and task-centric computing with the use of lightweight threading, futures, global address spaces and other mechanisms, in order to mitigate problems such as starvation, latency, management costs, and resource contention.

In terms of monitoring HPX applications, this work includes the study of: i) HPX performance counters, a framework that provides real-time access to a number of performance metrics; and ii) APEX, a tool with an event API that enables runtime auto-tuning of application parameters and the generation of execution profiles, with the aim of finding areas of application. This resulted in an extensive report on the inner workings of these tools, which, due to its importance, was incorporated into the dissertation itself. During the development of this project, a new HPX counter was defined that monitors component variables and exemplifies how the framework of counters can be adapted to handle situations outside of its initial scope.

This dissertation's main contribution is HpxTrace, a tracing tool for HPX applications that was modeled after Dtrace and built with APEX's policy engine. HpxTrace provides a system of probes that trigger upon the occurrence of certain events. The user can associate actions with each probe to collect and manipulate event data. Events include readings from HPX counters, execution of tasks, message exchanges, and events instrumented in the application by the users themselves.

To describe the actions each probe should take, a scripting language was developed using the Spirit library from the Boost library collection. In addition to providing the basic operations and aggregations found in Dtrace, the language had to be adapted to the distributed nature of HPX. This was accomplished by taking measures such as separating variables into local and global contexts and adding synchronization mechanisms.

To validate HpxTrace, two implementations of stencil algorithms that solve heat distribution problems were used as case studies. HpxTrace detected the differences in the optimizations between the two versions and their impact on performance.

Following testing, it was concluded that HpxTrace can be used to analyse and measure the performance of HPX applications. Additionally, it proved to be particularly useful in comprehending the internal behavior of HPX, which is valuable for tasks like finding performance optimizations, as well as debugging and learning the platform itself.

KEYWORDS Parallel Computing, Distributed Computing, Task Oriented Computing, HPX, APEX, Application Tracing

RESUMO

Com o aumento da complexidade do paralelismo na computação de alto desempenho, ferramentas tradicionais como o MPI e OpenMP começam a ser inadequadas. É neste contexto que surge o HPX, uma biblioteca C++ que é a primeira implementação do modelo de execução ParalleX. Este modelo descreve computação *message-driven* e *task-centric* com o uso de mecanismos como *lightweight threading*, futuros e espaços de endereçamento global de forma a mitigar problemas como inanição, latência, custos de gestão e contenção de recursos.

Em termos de monitorização de aplicações em HPX, o trabalho incluiu o estudo: i) da *framework* de contadores de desempenho do HPX, que permite o acesso a diversas métricas de desempenho em tempo real; e ii) do APEX, uma ferramenta cuja API de eventos permite afinar em tempo de execução os parâmetros da aplicação e gerar perfis de execução, com a intenção de encontrar áreas de aplicação. Isto resultou num relatório extensivo sobre o funcionamento interno destas ferramentas, que pela sua importância foi incorporado na própria dissertação.

Durante o desenvolvimento deste projeto, foi definido um novo contador HPX que monitoriza as variáveis de componentes e que exemplifica como a *framework* dos contadores pode ser adaptada para lidar com situações fora do seu âmbito inicial.

A principal contribuição desta dissertação é o HpxTrace, uma ferramenta de rastreamento de aplicações em HPX, inspirada no Dtrace e construída com o sistema de políticas do APEX. O HpxTrace disponibiliza um sistema de sondas que disparam mediante a ocorrência de certos eventos. O utilizador pode associar ações a cada sonda para colecionar e manipular os dados dos eventos. Os eventos incluem leituras dos contadores do HPX, execução de tarefas, troca de mensagens e eventos instrumentados na aplicação pelos próprios utilizadores.

Para descrever as ações que cada sonda deve tomar, foi desenvolvida uma linguagem de *scripting* com a biblioteca Spirit da coleção de bibliotecas Boost. Para além de disponibilizar as operações básicas e as agregações existentes no Dtrace, a linguagem teve que ser adaptada à natureza distribuída do HPX. Para isso foram tomadas medidas, tais como a separação de variáveis em contextos locais e globais e a adição de mecanismos de sincronização.

Para validar o HpxTrace, foram usadas como caso de estudo duas implementações de algoritmos *stencil* que resolvem o problema de distribuição de calor. O HpxTrace detetou as diferenças nas otimizações entre as duas versões e o impacto destas no desempenho.

Realizados os testes chegou-se à conclusão que o HpxTrace pode ser usado para a análise e medição do desempenho de aplicações em HPX. Adicionalmente, provou ser particularmente útil na compreensão do comportamento interno do HPX, o que é valioso não só para procurar otimizações de desempenho como também para tarefas como *debugging* e aprendizagem da própria plataforma.

PALAVRAS-CHAVE Computação Paralela, Computação Distribuída, Computação Orientado às Tarefas, HPX, APEX, Rastreamento de Aplicações

CONTEÚDO

1	INTRODUÇÃO	3
1.1	Contextualização	3
1.2	Motivação	3
1.3	Objetivos	4
1.4	Estrutura do documento	5
2	ESTADO DA ARTE	6
2.1	HPX - High Performance ParallelX	6
2.1.1	Localidades	6
2.1.2	Multi-threading	7
2.1.3	Active Global Address Space (AGAS)	7
2.1.4	Parcelas	7
2.1.5	Local Control Objects (LCOs)	7
2.1.6	Percolação	8
3	COMPONENTES	9
3.1	Classe servidor	9
3.2	Classe cliente	11
3.3	Componentes com templates	13
3.3.1	Componentes com um parâmetro template	14
3.3.2	Componentes com dois parâmetros template	15
3.3.3	Cliente de um componente template	16
4	CONTADORES DE DESEMPENHO	18
4.1	Formato dos contadores	18
4.2	Acesso aos contadores de desempenho	19
4.2.1	Acesso aos contadores de desempenho através da linha de comandos	19
4.2.2	Acesso aos contadores de desempenho através da API do HPX	20
4.3	Contadores PAPI	22
4.4	Definição de novos contadores	23
4.4.1	Contadores simples	23
4.4.2	Contadores completos	25

5	CONTADORES DE COMPONENTES	31
5.1	Solução sem Templates	31
5.2	Solução com Templates	33
6	APEX	37
6.1	Introdução	37
6.2	API	39
6.2.1	Tipos e Valores Pré-definidos	39
6.2.2	Registo de políticas	40
6.2.3	Eventos definidos pelo utilizador	41
6.2.4	Profiling	42
6.3	Funcionamento Interno e Integração com o HPX	44
6.3.1	Leitura de contadores - Integração HPX/APEX	44
6.3.2	Leitura de contadores - Funcionamento interno do APEX	46
6.3.3	Eventos de execução de tarefas	48
6.3.4	Eventos de troca de parcelas	51
6.3.5	Conclusão	54
7	HPXTRACE	55
7.1	DTrace	55
7.2	HpxTrace	57
7.3	Descrição da API	58
7.3.1	Estrutura das cláusulas	58
7.3.2	Definição da linguagem do predicado e das ações	58
7.3.3	Políticas	62
7.3.4	Sondas disponíveis	63
7.3.5	Funções da API	67
7.4	Implementação da API	68
7.4.1	Arquitetura e Armazenamento dos dados	68
7.4.2	Interpretadores	73
7.4.3	Implementação da linguagem	76
7.4.4	Deteção de erros de sintaxe	83
7.4.5	Implementação das sondas	83
7.5	Alterações ao HPX e extração de novos dados	89
7.5.1	Sondas task	91
7.5.2	Sondas message	92
7.5.3	Sonda definidas pelo utilizador	93

8	CASO DE ESTUDO - <i>stencil</i>	94
8.1	Introdução do problema	94
8.2	Versão 6	96
8.3	Análise de uma execução	97
8.4	Versão 7	101
8.5	Métricas de desempenho	104
9	DISCUSSÃO E CONCLUSÕES	108
9.1	Discussão	108
9.1.1	Antecedentes	108
9.1.2	Fase preparatória	108
9.1.3	1ª fase	108
9.1.4	2ª fase	109
9.2	Conclusões	111
9.3	Trabalho futuro	111
A	ANEXOS	115

LISTA DE FIGURAS

Figura 1	Arquitetura do APEX [Wagle et al. (2019)]	37
Figura 2	Perfil de execução de um programa que calcula o fibonacci de 10 através de ações	38
Figura 3	Perfil de execução da aplicação exemplo	43
Figura 4	Perfil de execução da aplicação sem a função <code>resume</code>	44
Figura 5	Arquitetura do HPX [Heller et al. (2013)]	51
Figura 6	Agregações disponíveis no Dtrace [(alias?)]	56
Figura 7	Diagrama dos componentes na localidade 0	69
Figura 8	Diagrama dos componentes nas restantes localidades	70
Figura 9	Diagrama das classes de agregação	72
Figura 10	Resultado do script	98
Figura 11	Trace de uma execução da versão 6 do stencil	100
Figura 12	Execução monitorizada da versão 7 do stencil	104

LISTA DE EXCERTOS DE CÓDIGO

3.1	Exemplo de declaração do servidor de um componente	9
3.2	Implementação do servidor <code>CompServer</code>	10
3.3	Instanciação de um servidor e invocação das suas ações	11
3.4	Implementação do cliente <code>CompClient</code> do servidor <code>CompServer</code>	11
3.5	Instanciações de clientes <code>CompClient</code>	12
3.6	Construtores alternativos de <code>CompClient</code>	13
3.7	Re-implementação do código 3.3 com o uso de um cliente	13
3.8	Servidor de um componente com um parâmetro <i>template</i>	14
3.9	Macros de declaração das ações de um servidor com um parâmetro <i>template</i>	14
3.10	Macros de registo de um servidor com um parâmetro <i>template</i>	15
3.11	Servidor de um componente com dois parâmetros <i>template</i>	15
3.12	Macros de um servidor com dois parâmetros <i>template</i>	16
3.13	Cliente de um componente com dois parâmetros <i>template</i>	16
3.14	Utilização de um componente com dois parâmetros <i>template</i>	17
4.1	Exemplo de listagem dos contadores relacionados com filas	19
4.2	Exemplo de listagem com informação extra dos contadores relacionados com filas de parcelas	19
4.3	Exemplo da medição do total de <i>user threads</i> executadas de segundo em segundo.	19
4.4	Declaração da função de instanciação de um contador	20
4.5	Código fonte do programa de exemplo de contadores	21
4.6	<i>Output</i> do programa de exemplo de contadores	21
4.7	Listagem dos contadores relacionados com PAPI	22
4.8	Exemplo de medição de <i>misses</i> da <i>cache</i> de nível 1 por fio de execução do sistema operativo	22
4.9	Simulação do jogo cara ou coroa	23
4.10	Assinaturas possíveis para as funções de contadores simples	23
4.11	Função que retorna o valor do contador heads-tails	24
4.12	Função de registo de um contador simples	24
4.13	Registo do contador head-tails	24
4.14	Processo para registar um contador simples	25
4.15	Declaração do servidor do componente completo	26
4.16	Campos do contador completo	26
4.17	Métodos de começo e paragem de medição do contador completo	26
4.18	Função de leitura do contador completo	27
4.19	Função de registo de um contador completo	28

4.20	Assinatura da função de descoberta de contadores	29
4.21	Decomposição e validação do nome fornecido	29
4.22	Invocação da função <code>f</code> para a única variação do nome da instância	29
4.23	Invocação da função <code>f</code> para todas as variações dos nomes das instâncias	30
4.24	Assinatura da função de criação de contadores	30
4.25	Instanciação do servidor de um componente	30
5.1	Resolução do nome de um componente e invocação da sua ação	32
5.2	Função de criação do novo contador	32
5.3	Declaração do servidor de um contador <i>template</i>	33
5.4	Função de leitura de um contador <i>template</i>	33
5.5	Construção de um contador <i>template</i>	33
5.6	Registo de um contador <i>template</i>	34
5.7	Função de registo do novo contador	34
5.8	Registo da fábrica de componentes	34
5.9	Novo macro de registo da fábrica de componentes	35
5.10	Exemplo de utilização do novo contador	35
5.11	Resultado da utilização do novo contador	36
6.1	Funções de registo de políticas APEX	40
6.2	Contexto de um evento APEX	40
6.3	Função de registo de políticas periódicas	41
6.4	Função de registo de eventos	41
6.5	Função de notificação da ocorrência de um evento definido pelo utilizador	41
6.6	Funções dos temporizadores do APEX	42
6.7	Exemplo de utilização dos temporizadores do APEX	42
6.8	Exemplo de utilização dos temporizadores do APEX sem a função <code>resume</code>	43
6.9	Assinatura da função <code>sample_value</code>	44
6.10	Definições de <code>hpx::util::external_timer::sample_value</code>	45
6.11	Definição e exportação do apontador para a função do evento <i>sample_value</i>	45
6.12	Exportação da função que permite o registo de funções para os eventos	45
6.13	Estrutura de um registo	45
6.14	Registo da função <code>apex::sample_value</code> no APEX	46
6.15	Adaptador da função <code>apex::sample_value</code>	46
6.16	Classe <code>sample_value_event_data</code>	46
6.17	Notificação dos <i>listeners</i> registados para os eventos de leitura de contadores	47
6.18	Função <code>on_sample_value</code> do <code>policy_handler</code>	47
6.19	Preenchimento do contexto do evento de leitura de um contador	47
6.20	Assinaturas das funções de temporizadores usadas internamente	48
6.21	Instrumentação do escalonador de tarefas	48

6.22	Função functor de <code>thread_data</code>	49
6.23	Inicialização do <code>profiler</code> pelo <code>profiler_listener</code>	50
6.24	Armazenamento do <code>profiler</code> no <code>task_wrapper</code>	50
6.25	Paragem do <code>profiler</code> pelo <code>profiler_listener</code>	50
6.26	Assinatura das funções dos eventos das mensagens	51
6.27	Notificação do APEX do envio de uma parcela	52
6.28	Classe <code>message_event_data</code>	52
6.29	Adição da localidade fonte na função <code>apex::send</code>	52
6.30	Desserialização das parcelas recebidas	53
6.31	Notificação da receção de uma parcela	53
7.1	Formato das cláusulas <code>Dtrace</code>	55
7.2	Formato de agregações no <code>Dtrace</code>	56
7.3	Estrutura das cláusulas <code>HpxTrace</code>	58
7.4	Descrição das sondas <code>HpxTrace</code>	58
7.5	Atribuição de variáveis	59
7.6	Formato de agregações no <code>HpxTrace</code>	60
7.7	Impressão de agregações	60
7.8	Exemplo de um erro de sintaxe no predicado	61
7.9	Exemplo de um erro de sintaxe numa ação	61
7.10	Exemplo de um erro de redefinição de uma agregação	61
7.11	Exemplo de um erro de redefinição dos parâmetros de uma agregação <i>lquantize</i>	61
7.12	Exemplo de um erro de variável não inicializada	62
7.13	Exemplo de um erro de atribuição de um tipo diferente	62
7.14	Pseudocódigo de uma registo de uma cláusula	63
7.15	Exemplo de um disparo de uma sonda definida pelo utilizador	63
7.16	Exemplo de uma cláusula de uma sonda definida pelo utilizador	63
7.17	Monitorização das <i>user threads</i> executadas	64
7.18	Adição das opções da linha de comandos a uma aplicação HPX	67
7.19	Assinaturas da função <code>hpx_main</code>	68
7.20	Ações fornecidas por <code>ScalarVarsServer</code>	70
7.21	Ações fornecidas por <code>MapVarsServer</code>	71
7.22	Ações fornecidas por <code>AggregationsServer</code>	73
7.23	Funções de agregação e impressão de agregações	73
7.24	Interpretadores de números, carateres alfabéticos e de apenas o carácter <code>+</code>	73
7.25	Interpretador de uma soma	74
7.26	Interpretador de todos os carateres exceto <code>'.'</code> e interpretador dos símbolos <code>'+' e <code>'-'</code></code>	74
7.27	Interpretador de um número ou uma sequência de somas	74
7.28	Regra de somas	74

7.29	Cálculo de somas com uma função auxiliar	75
7.30	Cálculo de somas com uma expressão Phoenix	75
7.31	Comparação de números com ações semânticas	75
7.32	<i>Binding</i> da função \mathbb{f} ao padrão $x \% y$	76
7.33	Verificação da paridade de um número com ações semânticas	76
7.34	Definição dos tipos das regras usadas	76
7.35	Regras de deteção de <i>strings</i>	77
7.36	Regras do contexto das variáveis	77
7.37	Obtenção do valor de uma variável <i>string</i>	77
7.38	Regra para variáveis <i>string</i> locais	78
7.39	Regra de variáveis <i>string</i> e regra de variáveis numéricas	78
7.40	Regra de dicionário de <i>strings</i> local	78
7.41	Declaração de um valor de qualquer tipo e definição de uma lista de chaves	78
7.42	Consolidação de valores <i>string</i> e de valores numéricos	78
7.43	Regra para expressões <i>string</i>	79
7.44	Regra para expressões aritméticas	79
7.45	Atribuição de um valor a variável escalar local	79
7.46	Registo do sucesso da atribuição para variáveis escalares	80
7.47	Aglomerção das regras de atribuição	80
7.48	Regras das funções de conversão de tipo	80
7.49	Redefinição de <code>string_value</code> e <code>double_value</code>	80
7.50	Aglomerção de todos os valores numa só regra	81
7.51	Regra das agregações	81
7.52	Regra de impressão de variáveis se agregações	81
7.53	Regra dos índices das localidades	82
7.54	Regra do <code>global_lock()</code>	82
7.55	Definição de ação e sequência de ações	82
7.56	Regra de comparação entre expressões	82
7.57	Regras da lógica booleana	83
7.58	Definição de um novo tipo de evento APEX	84
7.59	Tipo dos dados incluídos no contexto	84
7.60	Função de disparo das sondas definidas pelo utilizador	84
7.61	Inserção dos argumentos da sonda como variáveis de sonda	85
7.62	Invocação dos interpretadores das sondas	85
7.63	Função <code>on_sample_value</code> do <code>policy_handler</code>	85
7.64	Classe <code>sample_value_event_data</code>	86
7.65	Inicialização do temporizador	86
7.66	Função de leitura do contador	86

7.67	Funções <code>on_start</code> e <code>on_resume</code> do <code>policy_listener</code>	87
7.68	Funções <code>on_stop</code> e <code>on_yield</code> do <code>policy_listener</code>	87
7.69	Campos da classe <code>task_identifier</code>	88
7.70	Classe <code>apex::message_event_data</code>	88
7.71	Adição da nova opção no <code>CMakeLists</code>	89
7.72	Registo da função <code>apex::start</code>	90
7.73	Adição dos novos tipos de funções	90
7.74	Invocação da função <code>send</code> registada	90
7.75	Instrumentação do <code>scheduling loop</code> com o <code>HpxTrace</code>	91
7.76	Instrumentação do envio de parcelas com o <code>HpxTrace</code>	92
7.77	Instrumentação da receção de parcelas com o <code>HpxTrace</code>	92
7.78	Adição do fio de execução no disparo das sondas definidas pelo utilizador	93
8.1	Função <code>get_data</code> do servidor da partição	95
8.2	Função <code>get_data</code> do cliente da partição	95
8.3	Ação <code>heat_part</code> da versão 6	96
8.4	Versão instrumentada da ação <code>heat_part</code> da versão 6	97
8.5	<i>Script</i> com as sondas usadas	98
8.6	Anotação da tarefa <code>heat_part_data</code>	99
8.7	Sonda para a receção do resultado da ação <code>get_data</code>	99
8.8	Primeira parte da ação <code>heat_part</code> da versão 7	101
8.9	Segunda parte da ação <code>heat_part</code> da versão 7	101
8.10	Primeira parte da versão instrumentada da ação <code>heat_part</code> da versão 7	102
8.11	Segunda parte da versão instrumentada da ação <code>heat_part</code> da versão 7	103
8.12	Sondas para calcular o tempo médio das iterações	105
8.13	Sondas para calcular o tempo médio a comunicar com cada localidade vizinha numa iteração	105
8.14	Resultados parciais em milissegundos da versão 6 com o <i>script</i> de medição de desempenho	106
8.15	Resultados parciais em milissegundos da versão 7 com o <i>script</i> de medição de desempenho	106
A.1	Definição completa do cliente de um componente com dois parâmetros <i>template</i>	115
A.2	Campos da classe <code>task_wrapper</code> implementada pelo APEX	117
A.3	Campos da classe <code>profiler</code> implementada pelo APEX	118
A.4	Classe <code>hpx::external_timer::scoped_timer</code>	119
A.5	<i>Script</i> usado para analisar a execução da versão 6 do <i>stencil</i>	120
A.6	<i>Script</i> usado para analisar a execução da versão 7 do <i>stencil</i>	121
A.7	Resultados em milissegundos da versão 6 com o <i>script</i> de medição de desempenho	122
A.8	Resultados em milissegundos da versão 7 com o <i>script</i> de medição de desempenho	123

INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

No ambiente da computação de alto desempenho, o paralelismo é cada vez mais prevalente e complexo. Este aumento de complexidade está presente não só nas arquiteturas do *hardware* utilizado como também nos algoritmos e problemas a resolver. Os *clusters* modernos são compostos por elementos com crescente heterogeneidade (e.g., *sockets multi-core* combinados com GPUs), cada um com capacidades e especialidades de computação diferentes. Com a adição de algoritmos com estruturas de dados não lineares, as plataformas tradicionais começam a ser incapazes de continuar a fornecer o desempenho, a escalabilidade, a flexibilidade, a eficiência e a facilidade de uso requeridos atualmente. O uso conjunto de ferramentas de gestão de fios de execução com a interface MPI tem sido popular neste meio. No entanto, esta abordagem começa a ser inadequada. Algumas das ferramentas de gestão de fios de execução disponíveis, em particular as *thread-centric*, são de demasiado baixo nível, tornando o desenvolvimento um processo lento e difícil. Outras garantem acessibilidade em troca de um decréscimo de desempenho. Por sua vez, a rigidez do MPI torna-o algo inadequado para muitos dos problemas modernos, principalmente nas questões de tolerância a falhas e rebalanceamento dinâmico da carga [STEIIAR Group (2022c)].

Em 2008 surgiu uma nova resposta a estes desafios, o HPX (High Performance ParallelX), uma biblioteca C++ que é a primeira implementação do modelo de execução ParallelX [Kaiser et al. (2009)]. O ParallelX propõe ser uma alternativa que tenta responder a estes problemas através de um paradigma de computação orientada à tarefa, com um foco em paralelismo de grão fino e com conceitos e mecanismos como *multi-threading*, parcelas, espaço de endereçamento global, percolação e objetos de controlo locais [Kaiser et al. (2009)].

O HPX é um projeto *open source* em constante desenvolvimento, procurando sempre implementar os conceitos do ParallelX aderindo às diretrizes dos padrões mais modernos de C++ [STEIIAR Group (2022b)].

1.2 MOTIVAÇÃO

Mesmo com todas as vantagens do HPX, as aplicações desenvolvidas podem ser sempre alvo de significativas otimizações de desempenho. Para tal, é necessária a obtenção e agregação de informação relativa ao desempenho, que é naturalmente complexa em aplicações paralelas e distribuídas, sendo que muitas ferramentas tradicionais não foram desenvolvidas para lidar adequadamente com essa complexidade.

Por estas razões o HPX apresenta uma API própria para esse fim. A API permite aceder em tempo real aos contadores de desempenho, tanto aos da máquina local como aos das máquinas remotas [Group (2022a)]. Para além disso, permite a integração com outras ferramentas de medição de desempenho. As ferramentas existentes já disponibilizam capacidades consideráveis. Mesmo assim, considera-se que existem amplas oportunidades para desenvolver novas utilidades.

A complexidade subjacente ao HPX pode tornar a análise do seu comportamento e desempenho desafiante. Assim sendo, ferramentas que auxiliem nesta tarefa podem desempenhar um papel vital na obtenção de ganhos de desempenho.

1.3 OBJETIVOS

O objetivo principal da dissertação é contribuir para o ecossistema do HPX, na área de monitorização e análise do desempenho do sistema e das aplicações HPX.

A prossecução deste objetivo passa pelas seguintes etapas:

- Estudo e familiarização com a plataforma HPX.
- Aprendizagem da definição de componentes, como suporte para o resto do trabalho.
- Estudo da *framework* de contadores de desempenho (que métricas estão disponíveis e como são usadas).
- Detalhar o processo de definição de novos contadores e determinar como este pode adaptado.
- Desenvolvimento de novos contadores de desempenho.
- Estudo do APEX e da sua integração com o HPX.
- Analisar o Dtrace e que características podem ser adaptadas ao ambiente do HPX.
- Desenvolvimento de uma API de rastreamento de aplicações HPX, usando o APEX como base.
- Validação experimental da API com casos de teste.

1.4 ESTRUTURA DO DOCUMENTO

O capítulo 2 introduz os conceitos básicos do HPX necessários para o resto do trabalho. Também aborda os contadores de desempenho do HPX e a forma como são usados.

O capítulo 3 foca-se na sua totalidade na definição de componentes. Primeiro descreve a definição dos servidores e dos clientes dos componentes. Depois instrui como definir contadores com um ou dois parâmetros *template*.

O capítulo 4 é sobre a definição de contadores. Aborda não só a implementação das funcionalidades dos contadores mas também a forma como o HPX interpreta os nomes dos contadores.

O capítulo 5 unifica o conhecimento adquirido nos capítulos anteriores para definir um contador que monitoriza valores de campos dos componentes.

No capítulo 6 é examinada a ferramenta APEX. Primeiro, são exploradas as capacidades da API disponibilizada. Depois é investigada a integração próxima entre o HPX e o APEX.

O capítulo principal é o capítulo 7, onde é apresentada a ferramenta HpxTrace. O capítulo começa com uma breve secção sobre o Dtrace, a principal fonte de inspiração para o HpxTrace. De seguida, a API e todas as suas funcionalidades são descritas. Após isso, é detalhado como foram implementadas as sondas, as estruturas de dados e a linguagem de *scripting*. Por fim são descritas as alterações que foram efetuadas ao código fonte do HPX para permitir a extração de mais informação, de forma a expandir a utilidade do HpxTrace.

A validação das capacidades do HpxTrace ocorre no capítulo 8, onde duas versões de algoritmos *stencil* são seleccionadas como alvos de monitorização.

No final é discutido o trabalho desenvolvido, são apresentadas as conclusões e são sugeridas tarefas para trabalho futuro.

ESTADO DA ARTE

2.1 HPX - HIGH PERFORMANCE PARALLELX

HPX é uma biblioteca de C++ de concorrência e paralelismo. Corresponde à primeira implementação do modelo de execução ParalleX, um modelo focado em computação paralela assíncrona de alto desempenho, que visa mitigar os fatores limitadores de desempenho em arquiteturas convencionais de computação paralela, como processadores SMP e *commodity clusters*. De acordo com [Kaiser et al. \(2014\)](#), esses fatores, apelidados de SLOW, são:

- **Starvation** - quando não existe paralelismo suficiente para ocupar todos os recursos disponíveis;
- **Latencies** - custo de acesso a recursos e a serviços remotos;
- **Overhead** - custos de gestão do paralelismo e da concorrência, como a gestão de fios de execução e de mecanismos de sincronização;
- **Waiting for contention resolution** - atrasos devido à indisponibilidade de recursos partilhados.

Para lidar com estes desafios, é abandonada a rigidez do modelo tradicional de *message passing* do MPI, composto de processos alocados estaticamente que efetuam comunicação e sincronização através da passagem explícita de mensagens. Em troca procura-se implementar computação *message-driven*, onde as mensagens recebidas são processadas implicitamente e de forma assíncrona, em conjunção com filas de espera, endereçamento global de objetos, paralelismo de grão fino, mecanismos de sincronização leves, equivalência semântica entre execuções locais ou remotas e suporte a aceleradores de hardware através de percolação [[Kaiser et al. \(2009\)](#)].

Nas secções seguintes, estes e outros conceitos associados são esclarecidos.

2.1.1 Localidades

Um conceito importante em HPX é o de localidades. Uma localidade é um domínio físico em que os recursos podem operar de forma síncrona [[Gao et al. \(2007\)](#)]. Normalmente equivale a um nó de um *cluster*.

2.1.2 Multi-threading

No HPX, as *user threads* (fios de execução de nível utilizador) são leves e efémeras [Gao et al. (2007)] e estão associadas a uma tarefa ou a uma sequência de tarefas. É atribuída grande importância à rapidez de troca de contexto entre *user threads* de forma a esconder a latência de certas operações, permitindo a execução de outras tarefas enquanto a *user thread* se encontra bloqueada [Kaiser et al. (2020)].

As *user threads* são executadas por *worker threads*, que correspondem aos fios de execução do sistema operativo. Por omissão, cada *worker thread* tem a sua própria fila de tarefas a executar.

O HPX é flexível quanto à execução das *user threads*, disponibilizando ao utilizador várias políticas de escalonamento, sendo possível escolher como as tarefas são distribuídas pelas filas (*round robin*, por exemplo), a existência de filas com prioridades diferentes e ativar ou não *task/thread-stealing* (onde em certas circunstâncias, uma *work thread* pode "roubar" tarefas das filas das outras *work threads*) [Raj and Chen (2014)].

2.1.3 Active Global Address Space (AGAS)

O AGAS permite endereçar objetos de uma forma global a todas as localidades. Isto é importante para permitir a existência de uma API uniforme para execução local ou remota. Também permite mover objetos entre localidades, mantendo um endereço conhecido, o que é importante para suportar balanceamento de carga.

2.1.4 Parcelas

A comunicação entre localidades é assíncrona e é baseada no conceito de parcelas. Em Gao et al. (2007), uma parcela é descrita como sendo uma mensagem que consiste nas seguintes partes: o endereço global do objeto destinatário, a ação (função invocada remotamente) que será efetuada neste, os argumentos da ação e uma continuação. A continuação permite encadear várias operações remotas sem necessidade de o fluxo de execução retornar à localidade invocadora. As parcelas também são usadas para retornar os resultados das ações [Heller et al. (2013)].

2.1.5 Local Control Objects (LCOs)

LCOs são primitivas de sincronização disponibilizadas pelo HPX para controlar o paralelismo sem a necessidade de recorrer a barreiras globais. A gama de primitivas disponibilizadas pelo HPX é vasta, desde simples *mutexes* a mais complexos *dataflows*. Entre estes, destacam-se os futuros, sendo que o HPX oferece várias funcionalidades que permitem descrever o fluxo de algoritmos complexos através da composição de futuros.

2.1.6 *Percolação*

Percolação é uma variação do sistema de parcelas para mover antecipadamente trabalho e dados para localizações onde irão ser executadas de forma a mitigar latências [Gao et al. (2007)].

COMPONENTES

Dada a importância dos contadores nas aplicações HPX e no trabalho desenvolvido, é essencial estabelecer uma base de compreensão. Para tal, será explorado o código de um componente simples.

Componentes são um fator importante na implementação da distribuição da computação no HPX. São objetos C++ com a capacidade de serem criados e acedidos remotamente. Para que isto seja possível, os seus métodos são expostos como sendo ações, permitindo a sua invocação remota. No contexto HPX, ações são funções que podem ser executadas tanto local como remotamente, de forma síncrona ou assíncrona. Ações simples são executadas numa localidade alvo. No caso de ações de componentes o alvo é uma instância do componente.

Dada a natureza distribuída do HPX, cada instância de um componente tem o seu próprio identificador global, o GID, que é fornecido pelo AGAS. O GID é um elemento vital na interação com os componentes e respetivas ações, sendo representado pelo tipo `hpx::id_type`.

A implementação de um componente consiste normalmente em duas classes, a classe servidor e a classe cliente.

3.1 CLASSE SERVIDOR

A classe servidor é responsável pela implementação das funcionalidades do componente. Contém todos os atributos e métodos que existiriam num objeto convencional análogo.

```
class CompServer : public hpx::components::locking_hook<hpx::components::
    component_base<CompServer> >{
private: int value;
public:
    CompServer(int n);
    void add(int n);
    int get();
    HPX_DEFINE_COMPONENT_ACTION(CompServer, add);
    HPX_DEFINE_COMPONENT_ACTION(CompServer, get);
};
HPX_REGISTER_ACTION_DECLARATION(CompServer::add_action, comp_add_action);
HPX_REGISTER_ACTION_DECLARATION(CompServer::get_action, comp_get_action);
```

Código 3.1: Exemplo de declaração do servidor de um componente

A classe servidor de um componente tem que obrigatoriamente derivar da classe *template* `hpx::components::component_base<>` parametrizada com a própria classe a ser implementada (padrão CRTP - Curiously Recurring Template Pattern [Coplien (1995)]).

Para que o acesso seja *thread-safe*, a classe servidor também deve ser derivada da classe `hpx::components::locking_hook<>`. Desta forma o sistema de execução do HPX impossibilita a execução simultânea de várias ações sobre a mesma instância de um componente.

Como um objeto normal, o servidor possui variáveis para armazenar valores e métodos para os manipular.

Tal como mencionado, os métodos devem ser encapsulados em ações de forma a que seja gerado todo o código auxiliar que vai suportar chamadas remotas. A transformação de funções em ações é feita recorrendo a vários macros. Os macros específicos a utilizar variam conforme sejam ações simples ou ações de componentes mas a lógica do seu uso é a mesma.

O macro `HPX_DEFINE_COMPONENT_ACTION` é utilizado para definir os métodos a expor como sendo ações. Por omissão, o nome do tipo da ação é o nome da função com o sufixo `_action` (os métodos `add` e `get` dão origem aos tipos `add_action` e `get_action`), sendo também possível definir manualmente o nome do tipo através de um terceiro argumento opcional.

Definido o tipo da ação, é necessário proceder ao seu registo. Esta etapa tem que obrigatoriamente ser efetuada no *namespace* global. O macro a usar é `HPX_REGISTER_ACTION_DECLARATION`. Este macro declara um conjunto de objetos globais responsáveis por tarefas como serialização e inicialização, as quais permitem a invocação remota da ação. De forma análoga ao macro anterior, existe um argumento secundário opcional (`comp_add_action` e `comp_get_action`). Neste caso corresponde ao nome da ação e é usado por questões de serialização. A sua existência é mais do que mera conveniência, o nome tem que necessariamente ser único em todo o sistema. Por essa razão, é aconselhável definir manualmente o nome em vez de depender do nome gerado implicitamente pelo HPX.

No ficheiro de implementação, para além das implementações dos métodos declarados, é necessário recorrer a mais alguns macros.

```
CompServer::CompServer() : value(0) { }
CompServer::CompServer(int n) : value(n) { }

void CompServer::add(int n) {
    value += n;
}

int CompServer::get() {
    return value;
}

HPX_REGISTER_ACTION(CompServer::add_action, comp_add_action);
HPX_REGISTER_ACTION(CompServer::get_action, comp_get_action);

typedef hpx::components::component<CompServer> comp_type;
HPX_REGISTER_COMPONENT(comp_type, CompServer);
```

```
HPX_REGISTER_COMPONENT_MODULE();
```

Código 3.2: Implementação do servidor `CompServer`

O macro `HPX_REGISTER_ACTION_DECLARATION` é complementado pelo macro `HPX_REGISTER_ACTION`, que define os objetos globais declarados previamente.

Para além de registar as ações, é necessário registar o componente em si através do macro `HPX_REGISTER_COMPONENT`. Isto gera uma fábrica de componentes e torna possível instanciar o componente através do mecanismo `new_`.

Ao invés de utilizar directamente os construtores, é preferível utilizar a função `hpx::new_<>`. Esta permite criar uma ou mais instâncias de componentes numa localidade específica. Recebe como argumentos a localidade e os argumentos que vão ser passados ao construtor do servidor e retorna um futuro com o identificador global. Com esse identificador é possível invocar as ações do componente.

```
std::vector<hpx::id_type> localities = hpx::find_all_localities();

hpx::id_type id = hpx::new_<CompServer>(localities[0], 5).get();

CompServer::add_action()(id, 1);
CompServer::get_action()(id);
```

Código 3.3: Instanciação de um servidor e invocação das suas ações

Como as ações são *functors* (objetos funções), é necessária a instanciação seguida da invocação, daí a existência dos dois pares de parênteses.

3.2 CLASSE CLIENTE

A classe cliente serve como uma interface de instanciação de componentes e de acesso às suas ações. A sua utilização não é necessária mas é recomendada pois simplifica a utilização das ações.

Podem existir vários clientes para cada servidor. Assim, podem existir instâncias locais de clientes em várias localidades que interagem de forma transparente com o servidor noutra localidade.

```
class CompClient : public hpx::components::client_base<CompClient, CompServer>{

    typedef hpx::components::client_base<CompClient, CompServer> base_type;

public:
    CompClient(hpx::future<hpx::id_type> && id) : base_type(std::move(id)) {}
    CompClient(hpx::id_type && id) : base_type(std::move(id)) {}

    //Synchronous
    void add(int n){
        HPX_ASSERT(this->get_id());
    }
```



```

        CompServer::add_action() (this->get_id(), n);
    }
    //Asynchronous
    hpx::future<void> add(hpx::launch::async_policy, int n){
        HPX_ASSERT(this->get_id());
        return hpx::async<CompServer::add_action>(hpx::launch::async, this->
            get_id(), n);
    }
    //Fire-and-forget
    void add(hpx::launch::apply_policy, int n){
        HPX_ASSERT(this->get_id());
        hpx::apply<CompServer::add_action>(this->get_id(), n);
    }
    //Synchronous
    int get(){
        HPX_ASSERT(this->get_id());
        return CompServer::get_action() (this->get_id());
    }
    //Asynchronous
    hpx::future<int> get(hpx::launch::async_policy){
        HPX_ASSERT(this->get_id());
        return hpx::async<CompServer::get_action>(hpx::launch::async, this->
            get_id());
    }
};

```

Código 3.4: Implementação do cliente CompClient do servidor CompServer

De forma análoga ao servidor do componente, o cliente necessita de ser derivado da classe *template* `hpx::components::client_base<>`, parametrizada com a classe a ser implementada e com a classe servidor do componente.

Em relação aos construtores do cliente, existem várias possibilidades. A abordagem presente na documentação é usar construtores que recebem o identificador global de um servidor e invocam o construtor da classe base, sendo o identificador guardado num atributo herdado do `client_base` e acessível através de `this->get_id()`. Estes construtores permitem duas alternativas para criar os clientes. Evidentemente, é possível criar independentemente o servidor e passar o identificador global recebido ao construtor.

A outra funcionalidade possibilitada por estes construtores é o uso direto de `hpx::new_<>`, de forma semelhante ao servidor.

```

CompClient c1 = CompClient(hpx::new_<CompServer>(locality, 5));
CompClient c2 = hpx::new_<CompClient>(locality, 5);

```

Código 3.5: Instanciações de clientes CompClient

O uso de `hpx::new_<>` gera implicitamente um servidor e depois invoca o construtor do cliente com o identificador do servidor. Todos os argumentos passados depois da localidade são fornecidos ao servidor.

Uma abordagem alternativa em relação aos construtores é implementar construtores que escondam toda esta complexidade.

```
CompClient(int n) : base_type(hpx::local_new<CompServer>(n)) {}
CompClient(hpx::id_type locality, int n) : base_type(hpx::new_<CompServer>(
    locality, n)) {}
```

Código 3.6: Construtores alternativos de `CompClient`

O primeiro construtor constrói um servidor na mesma localidade que o cliente. O segundo constrói um servidor na localidade fornecida.

Os métodos do cliente consistem em invocar as ações do respetivo servidor usando o GID armazenado. As ações podem ser invocadas de forma síncrona, assíncrona ou não-bloqueante. Para representar estes conceitos existe a estrutura *launch*. O seu uso como argumento faculta a identificação do procedimento a seguir. Em caso de omissão assume-se execução síncrona. A execução síncrona não apresenta qualquer complexidade, consiste simplesmente em invocar a ação da instância servidor e retornar o resultado. A execução assíncrona funciona à base de futuros, tal como é a norma no HPX. Para isso é necessário recorrer à função `hpx::async`. A execução *fire and forget* consiste em invocar a ação através de `hpx::apply`, não existe qualquer espera pelo fim da execução da ação ou por um eventual valor de retorno.

Neste componente exemplo, são implementadas as três variações para a ação `add`. Logicamente, para a ação `get` não é implementada a execução *fire and forget*.

O resultado final é uma sintaxe mais clara e simples do que lidar diretamente com a invocação das ações.

```
std::vector<hpx::id_type> localities = hpx::find_all_localities();

CompClient c(localities[1], 5);
c.add(1);
c.get();
```

Código 3.7: Re-implementação do código 3.3 com o uso de um cliente

3.3 COMPONENTES COM TEMPLATES

Os componentes suportam o uso de *templates*. Para que uma classe servidor suporte *templates*, tem que sofrer as alterações expectáveis. O desafio da utilização de *templates* encontra-se no uso dos macros para registar as ações. A complexidade desta tarefa aumenta conforme o número de parâmetros *template* a adicionar. De seguida são apresentadas as definições de servidores para componentes com um e dois parâmetros *template* e a definição de um cliente para um componente com dois parâmetros *template*.

3.3.1 Componentes com um parâmetro *template*

Para introduzir os componentes com um parâmetro *template*, é definida a classe `SMapServer<T>`. Este componente desempenha a função de uma correspondência em que as chaves são do tipo `std::string` e os valores do tipo genérico `T`.

```
template <typename T>
class SMapServer
: public hpx::components::locking_hook< hpx::components::component_base<
    SMapServer<T>>>{
private:
    std::map<T> values;
public:
    typedef T value_type;
    void store(std::string name, value_type value){
        values[name] = value;
    }
    bool exists(std::string name){
        return values.find(name) != values.end();
    }
    hpx::util::optional<value_type> get(std::string name){
        if (exists(name)) return values[name];
        return {};
    }
    HPX_DEFINE_COMPONENT_ACTION(SMapServer, store);
    HPX_DEFINE_COMPONENT_ACTION(SMapServer, get);
};
```

Código 3.8: Servidor de um componente com um parâmetro *template*

A serialização dos dados das ações está dependente do tipo com que o servidor é parametrizado. Por essa razão, o registo da declaração da ação com o macro `HPX_REGISTER_ACTION_DECLARATION` tem que ser efetuado para cada tipo a ser utilizado. A melhor forma de encapsular é definir um macro que recebe um tipo e invoca os restantes macros.

```
#define REGISTER_SMAPSERVER_DECLARATON(type) \
    HPX_REGISTER_ACTION_DECLARATION( \
        SMapServer<type>::store_action, \
        BOOST_PP_CAT(__SMapServer_store_action_, type)); \
    HPX_REGISTER_ACTION_DECLARATION( \
        SMapServer<type>::get_action, \
        BOOST_PP_CAT(__SMapServer_get_action_, type)); \
```

Código 3.9: Macros de declaração das ações de um servidor com um parâmetro *template*

Uma vez que a declaração do tipo da ação deve receber um nome único, o argumento recebido pelo macro é concatenado para garantir a unicidade do nome.

O macro de definição das ações e o macro de registo do componente requerem o mesmo tratamento.

```
#define REGISTER_SMAPSERVER(type) \
    HPX_REGISTER_ACTION( \
        SMapServer<type>::store_action, \
        BOOST_PP_CAT(__SMapServer_store_action_, type)); \
    HPX_REGISTER_ACTION( \
        SMapServer<type>::get_action, \
        BOOST_PP_CAT(__SMapServer_get_action_, type)); \
    typedef ::hpx::components::component< \
        SMapServer<type> \
    > BOOST_PP_CAT(__SMapServer_, type); \
    HPX_REGISTER_COMPONENT(BOOST_PP_CAT(__SMapServer_, type))
```

Código 3.10: Macros de registo de um servidor com um parâmetro *template*

3.3.2 Componentes com dois parâmetros *template*

A classe `MapServer<K, V>` exemplifica um componente com dois parâmetros *template*. O parâmetro `K` é o tipo das chaves e o parâmetro `V` é o tipo dos valores de um mapa.

```
template <typename K, typename V>
class MapServer : public hpx::components::locking_hook<
    hpx::components::component_base<MapServer<K, V>>>{
    private:
        std::map<K, V> vars;
    public:
        typedef K key_type;
        typedef V value_type;
        void store(key_type name, value_type value){
            vars[name] = value;
        }
        bool exists(key_type name){
            return vars.find(name) != vars.end();
        }
        hpx::util::optional<value_type> get(key_type name){
            if (exists(name)) return vars[name];
            return {};
        }
        HPX_DEFINE_COMPONENT_ACTION(MapServer, store);
        HPX_DEFINE_COMPONENT_ACTION(MapServer, get);
};
```

Código 3.11: Servidor de um componente com dois parâmetros *template*

A existência de um parâmetro adicional aumenta a complexidade dos macros de registo.

```
#define REGISTER_MAP_DECLARATION(a, b) \
    typedef MapServer<a, b> HPX_PP_CAT(MapServer, __LINE__); \
    HPX_REGISTER_ACTION_DECLARATION( \
        HPX_PP_CAT(MapServer, __LINE__)::get_action, \
        HPX_PP_CAT(__MapServer_get_action_, HPX_PP_CAT(a,b))) \
    HPX_REGISTER_ACTION_DECLARATION( \
        HPX_PP_CAT(MapServer, __LINE__)::store_action, \
        HPX_PP_CAT(__MapServer_store_action_, HPX_PP_CAT(a,b))) \

#define REGISTER_MAP(a, b) \
    typedef MapServer<a,b> HPX_PP_CAT(MapServer, __LINE__); \
    HPX_REGISTER_ACTION( \
        HPX_PP_CAT(MapServer, __LINE__)::get_action, \
        HPX_PP_CAT(__MapServer_get_value_action_, HPX_PP_CAT(a,b))) \
    HPX_REGISTER_ACTION( \
        HPX_PP_CAT(MapServer, __LINE__)::store_action, \
        HPX_PP_CAT(__MapServer_store_value_action_, HPX_PP_CAT(a,b))) \
    typedef ::hpx::components::component<HPX_PP_CAT( \
        MapServer, __LINE__)> \
        HPX_PP_CAT(__MapServer_, HPX_PP_CAT(a,b)); \
    HPX_REGISTER_COMPONENT(HPX_PP_CAT(__MapServer_, HPX_PP_CAT(a,b))) \
```

Código 3.12: Macros de um servidor com dois parâmetros *template*

3.3.3 Cliente de um componente *template*

O cliente de um componente também pode ser alterado para suportar *templates*, não sendo necessário implementar um cliente para cada conjunto de parâmetros da classe servidor. A definição apresentada apenas inclui uma ação síncrona e uma assíncrona. Uma definição completa pode ser encontrada no anexo A.1.

```
template <typename K, typename V>
class MapClient : public hpx::components::client_base<
    MapClient<K,V>, MapServer<K,V>>{

    typedef hpx::components::client_base<MapClient<K,V>, MapServer<K,V>>
        base_type;
    typedef typename MapServer<K,V>::key_type key_type;
    typedef typename MapServer<K,V>::value_type value_type;
public:
    void store(key_type key, value_type value){
        HPX_ASSERT(this->get_id());
        typedef typename MapServer<K,V>::store_action action_type;
        action_type()(this->get_id(), key, value);
    }
}
```

```

    hpx::future<hpx::util::optional<value_type>> get(hpx::launch::
        async_policy, key_type key){
        HPX_ASSERT(this->get_id());
        typedef typename MapServer<K,V>::get_action action_type;
        return hpx::async<action_type>(hpx::launch::async, this->get_id(),
            key);
    }
};

```

Código 3.13: Cliente de um componente com dois parâmetros *template*

A presença da palavra-chave `typename` é necessária para indicar ao compilador que `MapServer<K,V>::get_action` é um tipo.

A utilização de um componente *template* requer não só a atribuição dos parâmetros na declaração e instanciação, como também o seu registo através do uso dos macros explorados anteriormente no espaço global.

```

REGISTER_MAP_DECLARATION(string,int);
REGISTER_MAP(string,int);

int hpx_main()
{
    hpx::naming::id_type locality = hpx::find_here();
    MapClient<std::string,int> m = hpx::new_<MapClient<std::string,int>>(locality);
    m.store("a", 26);
    m.get("a");

    hpx::finalize();
    return 0;
}

```

Código 3.14: Utilização de um componente com dois parâmetros *template*

CONTADORES DE DESEMPENHO

Uma ferramenta essencial para a análise do desempenho de aplicações são os contadores de desempenho. Estes disponibilizam métricas dos vários subsistemas do HPX. Por exemplo, uma métrica pode ser o tamanho da fila de espera de parcelas numa localidade ou o *idle-rate* de um fio de execução do sistema operativo. A análise destes valores permite identificar possíveis fatores responsáveis por perdas de desempenho. Também existem contadores de agregação que calculam estatísticas como a média ou o máximo das leituras de outros contadores.

O HPX disponibiliza uma API própria que permite criar contadores e ler os seus valores em tempo real. Quando uma aplicação HPX começa, os contadores disponíveis em cada localidade são registados, sendo que o acesso a estes pode ser local (aceder aos contadores da mesma localidade) ou remoto (aceder aos contadores de uma localidade diferente).

4.1 FORMATO DOS CONTADORES

Cada instância de um contador é identificada por um nome único. Todos os nomes seguem a estrutura `/objectname{full_instancename}/countername@parameters` [Group (2022a)]. O tipo de um contador corresponde à junção de `objectname` (grupo a que o contador pertence) e `countername`. Os parâmetros permitem passar informação adicional, como um valor inicial. Uma instância de um tipo é identificada por `full_instancename`. No caso de contadores de agregação, `full_instancename` é o nome de outro contador. Para os restantes contadores, `full_instancename` é decomposto em `parentinstancename#parentindex/instancename#instanceindex` [Group (2022a)]. Por norma, a instância pai designa a localidade do contador e a instância o alvo de monitorização. Um exemplo é o nome `/threadslocality#0/worker-thread#0/count/cumulative`, que corresponde ao número cumulativo de *user threads* que foram executadas pela *worker threads* 0 da localidade 0. Nem todos os contadores seguem estritamente a estrutura do `full_instancename`. O `instanceindex` em particular é frequentemente omitido.

Uma funcionalidade que é importante abordar é a expansão de contadores. Em certas situações, um nome é expandido para várias e são instanciados vários contadores. Isto normalmente acontece quando é usada uma *wildcard* num dos índices, o que resulta na criação de contadores para todos os índices possíveis. Para certos contadores ocorre uma expansão automática para várias instâncias. Um exemplo disto (e das variações

do `instanceindex`) é o nome `/threads{locality#0/total}/count/cumulative`, em que `full_instancename` é expandido para `locality#0/total/total` e para todas as possibilidades de `locality#0/pool#default/worker-thread#*`.

4.2 ACESSO AOS CONTADORES DE DESEMPENHO

Os dados dos contadores podem ser acedidos através da linha de comandos ou através da API no código.

4.2.1 Acesso aos contadores de desempenho através da linha de comandos

A utilidade da linha de comandos é limitada mas é uma forma fácil e rápida de obter algumas medições preliminares. É também o único método que suporta o uso de *wildcards* nos nomes dos contadores.

A opção `--hpx:list-counters` imprime todos os contadores disponíveis.

```
./program --hpx:list-counters | grep queue
/parcelqueue{locality#*/total}/length/receive
/parcelqueue{locality#*/total}/length/send
/threadqueue{locality#*/total}/length
```

Código 4.1: Exemplo de listagem dos contadores relacionados com filas

Para obter o mesmo resultado mas com mais informação sobre cada contador pode ser usada a opção `--hpx:list-counter-infos`.

```
./program --hpx:list-counter-infos | grep parcelqueue -A 3

fullname: /parcelqueue{locality#*/total}/length/receive
helptext: returns the number current length of the queue of incoming parcels
type:     counter_raw
version:  1.0.0
--
fullname: /parcelqueue{locality#*/total}/length/send
helptext: returns the number current length of the queue of outgoing parcels
type:     counter_raw
version:  1.0.0
```

Código 4.2: Exemplo de listagem com informação extra dos contadores relacionados com filas de parcelas

A opção `--hpx:print-counter` imprime o valor do contador especificado. Esta opção pode ser combinada com outras opções, como `--hpx:print-counter-interval`, que define o intervalo entre medições; `--hpx:print-counter-reset`, que reinicia o contador de cada vez que é medido; `--hpx:print-counter-destination`, que redireciona o resultado para um ficheiro; etc.

```
./program --hpx:print-counter=/threads{locality#0/total/total}/count/cumulative
--hpx:print-counter-interval=1000
```



```

/threads{locality#0/total/total}/count/cumulative,1,0.037437,[s],7
/threads{locality#0/total/total}/count/cumulative,2,1.037675,[s],14
/threads{locality#0/total/total}/count/cumulative,3,2.037955,[s],18
/threads{locality#0/total/total}/count/cumulative,4,3.038137,[s],22
/threads{locality#0/total/total}/count/cumulative,5,4.038343,[s],26
/threads{locality#0/total/total}/count/cumulative,6,5.038552,[s],30
/threads{locality#0/total/total}/count/cumulative,7,6.181618,[s],64
/threads{locality#0/total/total}/count/cumulative,8,6.214224,[s],87

```

Código 4.3: Exemplo da medição do total de *user threads* executadas de segundo em segundo.

Cada linha imprimida tem 5 ou 6 campos. Estes são:

- nome do contador;
- o contador foi lido pela *n*ésima vez;
- a *timestamp* em que esta medição ocorreu;
- a unidade de tempo da *timestamp*, neste caso segundos;
- o valor do contador;
- opcionalmente, a unidade de medida do valor do contador, que neste caso não é necessária.

4.2.2 Acesso aos contadores de desempenho através da API do HPX

O uso da API é mais flexível do que o acesso pela linha de comandos e possibilita adaptar o comportamento da aplicação de acordo com os valores medidos. Antes de ser usado, um contador tem que ser instanciado manualmente.

```
hpx::performance_counters::performance_counter counter(std::string const& name);
```

Código 4.4: Declaração da função de instanciação de um contador

Como só pode existir uma única instância ativa por contador, apenas a primeira invocação desta função instancia o contador. Futuras tentativas de instanciação retornam uma referência para o mesmo contador.

O objeto representativo do contador possui todas as funções necessárias para interagir com o mesmo, como as funções `get_value`, `stop` e `reset`.

Como exemplo foram medidos os fios de execução de nível utilizador do HPX executados por cada fio de execução do sistema operativo. O programa medido calcula a raiz quadrada dos índices de um vetor utilizando a função `for_loop` que automaticamente divide o trabalho em tarefas.

```

int main() {
    const int n = 10000000;
    std::vector<double> v(n);

    //Inicializar contadores
    std::size_t const os_threads = hpx::get_os_thread_count();
    std::vector<hpx::performance_counters::performance_counter> counters(os_threads
    );
    for (int i = 0; i < os_threads; i++) {
        counters[i] = hpx::performance_counters::performance_counter("/threads{
            locality#0/worker-thread#" + std::to_string(i) + "/total}/count/cumulative
            ");
    }

    hpx::for_loop(hpx::execution::par, 0, n, [&v](auto i) { v[i] = std::sqrt(i);});

    //Ler contadores
    for (int i = 0; i < os_threads; i++) {
        hpx::cout << "worker-thread#" + std::to_string(i) + ": " << counters[i].
            get_value<int>().get() << hpx::endl;
    }

    return 0;
}

```

Código 4.5: Código fonte do programa de exemplo de contadores

```

./program
worker-thread#0: 5
worker-thread#1: 6
worker-thread#2: 6
worker-thread#3: 7
worker-thread#4: 7
worker-thread#5: 8

```

Código 4.6: *Output* do programa de exemplo de contadores

Os resultados indicam que o HPX utilizou 6 fios de execução ao nível do sistema operativo, o que corresponde aos 6 *cores* disponíveis na máquina usada. Para além disso dividiu de forma algo equilibrada as tarefas pelas várias *worker threads*.

4.3 CONTADORES PAPI

Os processadores modernos possuem registos especiais, chamados de contadores de *hardware*, que contabilizam as várias ações efetuadas pelo processador. Estes permitem analisar o impacto do código no *hardware*, possibilitando a identificação de gargalos de desempenho sem impactos de sobrecarga significativos no desempenho do código.

O PAPI é uma interface de acesso a estes contadores [Mucci et al. (1999)]. O HPX suporta a integração do PAPI, permitindo a utilização dos seus contadores através da mesma *framework* que os restantes. Para o HPX permitir a utilização dos contadores PAPI é necessário especificar essa opção durante a sua instalação.

Filtrando o resultado da opção `--hpx:list-counters` (que imprime todos os contadores disponíveis) por ocorrências de `papi` obtemos o seguinte:

```
./program --hpx:list-counters | grep papi

/papi{locality#*/total}
/papi{locality#*/worker-thread#*}
```

Código 4.7: Listagem dos contadores relacionados com PAPI

Como pode ser observado, é suportada a instrumentação por localidade ou por fio de execução do sistema operativo. No entanto não é suportada instrumentação por fios de execução de nível utilizador.

Para utilizar um contador é preciso completar o seu nome com o evento a medir. Para obter todos os eventos disponíveis pode ser usada a opção `--hpx:papi-event-info`.

A utilização destes contadores é igual à dos contadores nativos do HPX.

```
./program --hpx:papi-domain=all --hpx:print-counter=/papi{locality#0/worker-
thread#*}/PAPI_L1_TCM

/papi{locality#0/worker-thread#0}/PAPI_L1_TCM,1,0.322130,[s],4.2963e+06
/papi{locality#0/worker-thread#1}/PAPI_L1_TCM,1,0.321906,[s],6.53578e+06
/papi{locality#0/worker-thread#2}/PAPI_L1_TCM,1,0.322166,[s],4.8375e+06
/papi{locality#0/worker-thread#3}/PAPI_L1_TCM,1,0.322061,[s],4.98386e+06
```

Código 4.8: Exemplo de medição de *misses* da *cache* de nível 1 por fio de execução do sistema operativo

O HPX permite a definição de novos contadores. Isto é útil pois permite coleccionar informação específica à aplicação tirando proveito de todas as vantagens da *framework* já existente.

A definição de novos contadores é flexível. A opção mais simples é a definição de um contador básico que invoca uma função de cada vez que é lido. A opção mais complexa é a definição de um contador completo que suporta mais funcionalidades como parar, continuar ou reiniciar o contador.

4.4 DEFINIÇÃO DE NOVOS CONTADORES

O HPX também permite a definição de novos contadores. Isto é útil pois permite coleccionar informação específica à aplicação tirando proveito de todas as vantagens da *framework* já existente.

A definição de novos contadores é flexível. A opção mais simples é a definição de um contador básico que invoca uma função de cada vez que é lido. A opção mais complexa é a definição de um contador completo que suporta mais funcionalidades tais como parar, continuar ou reiniciar o contador. Nas secções seguintes estas duas possibilidades são exploradas.

4.4.1 Contadores simples

Considere-se o seguinte programa que simula o simples jogo de cara ou coroa.

```
std::int64_t heads(0);
std::int64_t tails(0);

for (int i = 0; i < 300000000; ++i) {
    if ((std::rand() % 2) == 0)
        heads++;
    else
        tails++;
}

return hpx::finalize();
```

Código 4.9: Simulação do jogo cara ou coroa

Pretende-se que por razões de conveniência seja possível observar a quantidade de caras ou coroas através da *framework* dos contadores HPX.

Tal como mencionado, a implementação mais simples consiste em fornecer uma função que é invocada de cada vez que o contador é lido. A função tem que corresponder a uma das seguintes assinaturas:

```
std::int64_t some_performance_data(bool reset);

std::vector<std::int64_t> some_performance_data(bool reset);
```

Código 4.10: Assinaturas possíveis para as funções de contadores simples

Ou seja, o resultado da função tem que ser obrigatoriamente um inteiro ou um vetor de inteiros. Neste último caso, se os valores forem impressos na linha de comandos então serão separados pelo símbolo ":". O argumento `reset` existe para indicar se o valor do contador deve ou não ser reiniciado após a leitura.

Surgem assim duas opções: implementar dois contadores, sendo que um retorna o valor de caras e o outro de coroas, ou um único contador que retorna ambos num vetor. Optando pela última opção, fica-se com a seguinte função:

```
std::vector<std::int64_t> heads_tails_counter(bool reset) {
    std::vector<std::int64_t> result;
    result.push_back(heads);
    result.push_back(tails);
    return result;
}
```

Código 4.11: Função que retorna o valor do contador heads-tails

Antes de poder ser utilizado, o contador tem que ser registrado no sistema de tempo de execução do HPX. Isto é efetuado através da função `hpx::performance_counters::install_counter_type`.

Existem várias definições desta função com diferentes conjuntos de argumentos. A assinatura da função que permite registar um contador simples é a seguinte:

```
counter_status install_counter_type(
    std::string const &name,
    hpx::util::function_nonser<std::int64_t(bool)> const &counter_value,
    std::string const &helptext = "",
    std::string const &uom = "",
    counter_type type = counter_raw,
    error_code &ec = throws
)
```

Código 4.12: Função de registo de um contador simples

Os argumentos desta assinatura são:

- `name` - o nome do tipo contador;
- `counter_value` - a função que vai ser chamada de cada vez que o contador é lido;
- `helptext` (opcional) - um breve texto que descreve o propósito do contador;
- `uom` (opcional) - a unidade de medida do valor do contador;
- `type` (opcional) - o tipo do contador;
- `ec` (opcional) - um objeto representativo de uma condição de erro e onde será registada a ocorrência de possíveis erros de fontes como o sistema operativo.

O contador ficou registado com o nome de `/example/heads-tails`.

```
hpx::performance_counters::install_counter_type(
    "/example/heads-tails",           //name
    &heads_tails_counter,             //counter_value
    "returns the number of heads and tails", //helptext
    "heads:tails"                     //uom
)
```

```
);
```

Código 4.13: Registo do contador head-tails

Ao ser registado desta forma, os nomes das instâncias do contador seguem o seguinte formato `/example{locality#<*>/total}/heads-tails`, onde `<*>` representa a localidade na qual o contador vai ser criado. Importante notar que desta forma apenas é possível criar um contador em cada localidade.

Para possibilitar a utilização através da linha de comandos do contador definido, este tem que ser registado durante o arranque da aplicação, antes da execução do `hpx_main`. Isto é possível através do uso da função `hpx::register_startup_function`, que garante que a função registada é executada entre o início do processo de *startup* e a invocação do `hpx_main` (que é invocado por `hpx::init`).

```
void register_counter_type() {

    hpx::performance_counters::install_counter_type(
        (...)
    );
}

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

    hpx::register_startup_function(&register_counter_type);

    return hpx::init(argc, argv);
}
```

Código 4.14: Processo para registar um contador simples

Devido à sua simplicidade, estes contadores são fácil e rapidamente implementados. Porém, esta mesma simplicidade impõe bastantes restrições no que é possível conseguir fazer com os contadores. Se for necessário manter um estado entre leituras de medições é necessário recorrer a variáveis externas à função, o que não é ideal em termos de organização.

4.4.2 Contadores completos

Para tirar proveito de todas as capacidades da *framework* dos contadores em HPX, pode ser implementado um contador de desempenho completo. Em troca de um aumento significativo na complexidade de implementação, são oferecidas múltiplas vantagens.

Os contadores completos são baseados em componentes e correspondem a um objeto C++, logo estes possuem um estado interno que pode ser utilizado. Em termos de funcionalidades, os contadores completos suportam funcionalidades como definir valores iniciais e parar, retornar ou reiniciar a medição.

Para auxiliar a descrição do processo de implementação de um contador completo, será usado um contador responsável por medir o tempo, o `example_counter`. A utilidade deste contador não é relevante, servindo apenas como um simples exemplo.

A implementação de um contador de desempenho completo traduz-se na criação de um componente especial responsável pelas funcionalidades do contador. Sendo assim, a implementação não vai estar integrada diretamente no código da aplicação mas sim num módulo separado.

Tal como num componente normal, é implementada uma classe servidor. A classe cliente não é implementada pois a interação com os componentes é responsabilidade da *framework* abordada em 4.2.

Classe servidor

Mais uma vez, a classe servidor deve derivar da classe base `base_performance_counter` parametrizada com o tipo da classe a implementar.

```
class example_counter
: public hpx::performance_counters::base_performance_counter<example_counter>
```

Código 4.15: Declaração do servidor do componente completo

Sendo uma classe C++, o contador pode ter o seu estado interno.

```
private:
    std::int64_t time_started;
    std::int64_t total_time;
    bool counting;
```

Código 4.16: Campos do contador completo

Devem ser implementados os métodos correspondentes às funcionalidades desejadas. Os métodos `start` e `stop`, como o nome sugere, devem começar e parar a medição do valor.

```
bool example_counter::start()
{
    time_started = static_cast<std::int64_t>(hpx::get_system_uptime());
    counting = true;
    return counting;
}

bool example_counter::stop()
{
    counting = false;
    return counting;
}
```

Código 4.17: Métodos de começo e paragem de medição do contador completo

O significado do booleano retornado fica à descrição do implementador (a documentação do HPX não atribui nenhum significado específico).

O método mais importante é o `get_counter_value`. Este recebe um booleano e retorna um objeto do tipo `hpx::performance_counters::counter_value`. De forma análoga à função do contador simples, este é invocado para ler o valor do contador. Novamente, recebe o argumento `reset` que indica se o valor do contador deve ou não ser reiniciado após a leitura. A maior diferença é no valor de retorno, que em vez de um inteiro deve ser um `counter_value`.

O tipo `counter_value` consiste numa estrutura com os seguintes campos:

- `value_` - o valor atual do contador;
- `time_` - a data da medição;
- `scaling_` - a escala do valor medido;
- `scale_inverse_` - se o valor deve ser multiplicado ou dividido pela escala;
- `status_` - o estado do valor do contador ou possíveis códigos de erro (ex: `status_new_data`, `status_invalid_data`);
- `count_` - quantas vezes o contador foi lido.

No exemplo a ser explorado, `get_counter_value` trata de implementar a lógica de medir o tempo. É importante ter em consideração se deve ocorrer ou não um reinício do valor, se a contagem deve ser parada e que devem ser preenchidos os vários atributos do `counter_value`.

```
hpx::performance_counters::counter_value
example_counter::get_counter_value(bool reset)
{
    std::int64_t current_time =
        static_cast<std::int64_t>(hpx::get_system_uptime());

    std::int64_t const scaling = 1000000;
    hpx::performance_counters::counter_value value;

    if(counting){
        total_time += current_time - time_started;
        time_started = current_time;
    }

    value.value_ = total_time;
    value.time_ = current_time;
    value.scaling_ = scaling;
    value.scale_inverse_ = true;
    value.status_ = hpx::performance_counters::status_new_data;
    value.count_ = ++invocation_count_;
```



```

    if (reset){
        total_time = 0;
        time_started = current_time;
    }

    return value;
}

```

Código 4.18: Função de leitura do contador completo

Registo do contador

O registo do contador completo é a parte onde a implementação é mais complexa e, ao mesmo tempo, menos documentada. Este processo envolve vários passos e começa pelo registo do módulo em si com o macro `HPX_REGISTER_COMPONENT_MODULE`.

O contador completo precisa de ser registado antes da execução do `hpx_main`. Devido a ser um módulo separado, este processo é diferente do registo do contador simples. Primeiro é necessário utilizar o macro `HPX_REGISTER_STARTUP_MODULE` para registar uma função que irá ser executada quando o módulo é carregado durante o arranque do sistema de execução HPX. Nessa função podemos novamente usar a função `install_counter_type` para registar os contadores mas desta vez com um conjunto diferente de argumentos.

```

counter_status install_counter_type(
    std::string const &name,
    counter_type type,
    std::string const &helptext,
    create_counter_func const &create_counter,
    discover_counters_func const &discover_counters,
    std::uint32_t version = HPX_PERFORMANCE_COUNTER_V1,
    std::string const &uom = "",
    error_code &ec = throws
)

```

Código 4.19: Função de registo de um contador completo

A função `discover_counters_func` é responsável por determinar quais as instâncias do contador que devem ser criadas. Por exemplo, se o nome de um contador conter *wildcards* (*), devem ser criadas instâncias para todas as versões possíveis do nome. Cabe à função `discover_counters_func` validar o uso das *wildcards* e descobrir quais as instâncias que devem ser criadas. Esta função apenas é invocada quando os contadores são criados a partir da linha de comandos, visto que apenas este modo suporta o uso de *wildcards* para pedir vários contadores em simultâneo. No caso do uso da API no código, é invocada diretamente a função `create_counter_func`, responsável por instanciar um contador específico.

A documentação desta função é muito limitada, sendo que os passos seguintes foram deduzidos através da análise e experimentação de exemplos incluídos no código fonte do HPX.

```
bool explicit_example_counter_discoverer(
    hpx::performance_counters::counter_info const& info,
    hpx::performance_counters::discover_counter_func const& f,
    hpx::performance_counters::discover_counters_mode mode,
    hpx::error_code& ec
)
```

Código 4.20: Assinatura da função de descoberta de contadores

No `counter_info` está presente informação como o tipo, o estado e a unidade de medida. A mais importante é uma *string* com o nome fornecido do contador (ou contadores, se conter *wildcards*) a criar. Para analisar esta função é importante lembrar que o nome completo de uma instância (`full_instancename`) é formado pela composição do nome da instância pai (`parentinstancename#parentindex`) com o nome da instância em si (`instancename#instanceindex`).

É possível suportar o uso de *wildcards* na instância pai e na instância em si, normalmente no índice. A função `f` deve ser chamada para todas as variações possíveis do nome da instância em si. Do ponto de vista da implementação de um contador, é apenas relevante saber que a função `f` irá automaticamente descobrir todas as variações possíveis do nome da instância pai (normalmente, todas as localidades) e invocar a função `create_counter_func` para cada. Esta última é encarregue de criar um contador para o nome que lhe é fornecido. Ou seja, a função de criação é invocada para todas as variações possíveis do `full_instancename`.

No início da função de descoberta, a *string* com nome completo do contador deve ser decomposta. Esta tarefa é facilmente efetuada com o uso da estrutura `counter_path_elements`, cujos atributos correspondem aos elementos do nome, e com o uso da função `get_counter_path_elements`, que preenche esses atributos. Ao verificar o `status` retornado pela função também é possível saber se o nome da instância segue o formato requerido.

```
hpx::performance_counters::counter_path_elements p;

hpx::performance_counters::counter_status status =
    get_counter_path_elements(info.fullname_, p, ec);

if (!status_is_valid(status)) return false;
```

Código 4.21: Decomposição e validação do nome fornecido

No caso de o nome fornecido estar correto e não conter nenhuma *wildcard*, a função `f` deve ser invocada com a informação necessária. Depois de invocada deve ser verificado se foi bem sucedida.

```
if (!f(i, ec) || ec) {
    return false;
```

```

}
ec = hpx::make_success_code();
return true;

```

Código 4.22: Invocação da função `f` para a única variação do nome da instância

Caso o nome da instância contenha uma *wildcard*, os vários nomes possíveis devem ser compostos e `f` invocada para cada um. O excerto 4.23 lida com o caso habitual de existir uma *wildcard* no `instanceindex`.

```

else if (p.instance_name_ == "instance#*") {
    for (int n = 0; n < MAX_INSTANCES; n++) {
        p.instance_name_ = "instance";
        p.instanceindex_ = n;
        status = get_counter_name(p, i.fullname_, ec);
        if (!status_is_valid(status) || !f(i, ec) || ec)
            return false;
    }
}

```

Código 4.23: Invocação da função `f` para todas as variações dos nomes das instâncias

Quando invocada, seja após o processo de descoberta ou imediatamente no início, a função `create_counter` vai proceder à criação do contador, invocando o construtor do servidor do contador.

```

hpx::naming::gid_type explicit_example_counter_creator(
    hpx::performance_counters::counter_info const& info,
    hpx::error_code& ec
)

```

Código 4.24: Assinatura da função de criação de contadores

Novamente, é recebido o `counter_info` e o nome deve ser verificado e decomposto com a função `get_counter_path_elements`. A verificação é importante pois nos casos em que a função é invocada diretamente, não ocorre a verificação presente em `discover_counters_func`.

De forma a instanciar o servidor responsável pela lógica do contador, deve ser usada a função `hpx::components::server::construct<>` que funciona de forma algo semelhante ao `hpx::new<>`. É parametrizada com o tipo do contador a instanciar e recebe argumentos que vai passar ao construtor do servidor.

```

hpx::naming::gid_type id;

id = hpx::components::server::construct<example_counter_type>(info);

```

Código 4.25: Instanciação do servidor de um componente

CONTADORES DE COMPONENTES

Durante o desenvolvimento da dissertação surgiu a possibilidade de utilizar contadores para monitorizar algum aspeto dos componentes. Um dos possíveis alvos de monitorização são os valores dos campos do componente. Apesar de isso não corresponder a uma medição de desempenho, pode potencialmente auxiliar no processo de análise de desempenho pois permite ter uma ideia do progresso da execução do programa ou contextualizar medições de outros contadores.

Mais especificamente, pretende-se desenvolver um contador em que o utilizador possa, através da API no código ou da linha de comandos, especificar no nome do contador uma instância de um componente para monitorizar. Devido à natureza remota dos componentes, a obtenção do valor de uma variável terá que ocorrer através de uma ação do componente já existente. Caso o componente ainda não exista, o contador deverá devolver um valor predefinido à descrição do utilizador. O elemento `parameter` do nome poderá ser usado para definir esse valor.

Os nomes das instâncias do contador que se pretende desenvolver deverão seguir o formato `/example{locality#index/component_name}/name@default_value`

Considerando o objetivo pretendido, a implementação de um contador completo é mais adequada do que uma implementação simples, dado que esta última é demasiado restritiva. Por exemplo, não permite a existência de um estado interno para cada instância individual do contador nem permite a liberdade necessária em relação ao nome do contador.

5.1 SOLUÇÃO SEM TEMPLATES

Um dos desafios principais desta tarefa é como ter acesso ao componente dentro do contexto de execução do contador. Toda a informação disponível para a criação do contador provém somente do nome que é fornecido. Então é importante conseguir identificar uma instância de um componente através de uma *string*. Uma possibilidade seria converter o seu identificador global numa *string*. Porém, esta hipótese não seria uma solução muito elegante e tornaria impossível a utilização destes contadores através da linha de comandos.

Felizmente, o AGAS do HPX possui um mecanismo que permite associar nomes definidos pelo utilizador a GIDs. A função `register_name` permite associar uma string a um determinado GID. A partir de qualquer contexto de execução ou localidade é possível obter o identificador registado através da função `resolve_name`.

Desta forma, na função de leitura do contador é possível obter o identificador de um componente a partir do nome de um componente.

```
if(component_id == hpx::naming::invalid_id){
    component_id = hpx::agas::resolve_name(component_name).get();
    value.value_ = default_value;
}
if(component_id != hpx::naming::invalid_id){
    value.value_ = ::server::comp::get_action()(component_id);
}
```

Código 5.1: Resolução do nome de um componente e invocação da sua ação

Como o contador pode ser lido antes de o componente ter sido registrado, principalmente no caso de ser usado através da linha de comandos, o identificador retornado pelo AGAS pode ser inválido. Neste caso é retornado um valor pré-definido. Numa tentativa de minimizar impactos extra no desempenho, o identificador global também é armazenado para evitar consultas repetidas do AGAS. A desvantagem disto é que o componente nunca é destruído automaticamente pelo *garbage collector* do HPX, já que existe sempre uma referência (o identificador) para o componente. Com o identificador validado invoca-se a ação para adquirir o valor pretendido.

A função de descoberta de contadores não contém nenhum aspeto relevante de mencionar, já que o uso de *wildcards* não é suportado. Por sua vez, a função de criação tem a responsabilidade acrescida de extrair o nome do componente a monitorizar e o valor inicial.

```
template <class T>
hpx::naming::gid_type explicit_name_counter_creator(
    hpx::performance_counters::counter_info const& info, hpx::error_code& ec)
{
    hpx::performance_counters::counter_path_elements paths;
    get_counter_path_elements(info.fullname_, paths, ec);

    if (paths.parameters_.empty()){
        paths.parameters_ = "0";
    }

    hpx::naming::gid_type id;
    id = hpx::components::server::construct<example_counter_type>(
        complemented_info, paths.instancename_, paths.parameters_
    );
    return id;
}
```

Código 5.2: Função de criação do novo contador

Apesar desta solução resolver o problema de como identificar a instância do contador, continua com alguns problemas. Em primeiro lugar, não é transparente dado que a utilização do contador necessita da intervenção do utilizador na implementação para definir a ação a ser usada. Para além disso, requer uma nova implementação para cada tipo de componente.

5.2 SOLUÇÃO COM TEMPLATES

O objetivo seria uma única implementação que funcionaria para qualquer tipo de componente e cuja complexidade estivesse escondida do utilizador. Ora, uma resposta comum em C++ para situações deste tipo é o uso de *templates*.

O recurso a *templates* em contextos semelhantes ocorre no código fonte do HPX. Um exemplo são os contadores de agregação do grupo *arithmetics*. Cada tipo corresponde a uma operação (adição, média, máximo, etc.) que é aplicada ao resultado de outros contadores. Todos partilham a mesma classe servidor, uma classe *template* em que o parâmetro é a operação a aplicar.

Com base neste precedente, a classe do contador é facilmente alterada para uma classe *template* em que o parâmetro é a ação que vai ser invocada.

```
template <class T> class name_counter
: public hpx::performance_counters::base_performance_counter<name_counter<T>>
```

Código 5.3: Declaração do servidor de um contador *template*

```
template <class T>
hpx::performance_counters::counter_value
name_counter<T>::get_counter_value(bool reset) {
    (...)
    if(component_id != hpx::naming::invalid_id)
        value.value_ = T()(component_id);
    return value;
}
```

Código 5.4: Função de leitura de um contador *template*

A abordagem utilizada nos contadores aritméticos deixa de ser viável no que toca ao registo dos contadores. Mais especificamente, na função de criação de instâncias onde é necessário o tipo especificado do contador para proceder à sua construção. Como apenas existe um conjunto fixo de operações suportadas, os diferentes tipos dos contadores aritméticos encontram-se *hardcoded*.

Mesmo assim, a utilização de *templates* continua a ser uma possibilidade. Para isso é preciso alterar a função de criação para funcionar com *templates*.

```
template <class T>
hpx::naming::gid_type name_counter_creator_func(
    hpx::performance_counters::counter_info const& info, hpx::error_code& ec)
{
    (...)
    hpx::naming::gid_type id;
    id = hpx::components::server::construct<T>(
        complemented_info, paths.instancename_, paths.parameters_);
    return id;
}
```

```
}
```

Código 5.5: Construção de um contador *template*

E também deve ser alterada a função que regista o contador.

```
template <class T>
void register_counter(std::string name) {
    using namespace hpx::performance_counters;

    install_counter_type(
        name,
        counter_raw,
        " ",
        &explicit_name_counter_creator_name<T>,
        &explicit_name_counter_discoverer,
        HPX_PERFORMANCE_COUNTER_V1,
        " " ,
    );
}
```

Código 5.6: Registo de um contador *template*

No exemplo do capítulo anterior, a função que instalava o contador era registada como sendo uma função *startup* do módulo, ou seja, uma função que iria ser executada durante a inicialização do HPX caso o módulo fosse incluído. Este procedimento tem que necessariamente ser alterado para cumprir os objetivos pretendidos. A resposta encontrada consiste em oferecer uma função para o utilizador invocar no *main* que recorre às expressões *lamda* para gerar, durante a inicialização, uma função que irá registar o contador.

```
template <class T>
void register_component_counter(std::string name) {

    hpx::register_startup_function([name]() -> void {
        performance_counters::examples::register_counter<T>(name);
    });
}
```

Código 5.7: Função de registo do novo contador

Como agora o módulo pode ser usado para registar vários tipos de contadores, é incluído um argumento para o utilizador poder definir o nome do tipo de cada um.

O único aspeto que falta é o registo da fábrica de componentes.

```
typedef hpx::components::component<
    ::performance_counters::examples::server::name_counter<::server::comp::
        get_action>
> name_counter_type;
```

```
HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC(
    name_counter_type, name_counter, "base_performance_counter");
```

Código 5.8: Registo da fábrica de componentes

O ideal seria ocultar este passo, incluindo-o na função anterior. Porém, o macro necessário define um *namespace*. Como o C++ não permite a declaração de *namespaces* dentro da definição de uma função isto não é possível. Mesmo assim, é conveniente reduzir a complexidade deste passo. Para isso é definido um novo macro.

```
#define REGISTER_COMPONENT_COUNTER_FACTORY(action) \
    HPX_REGISTER_DERIVED_COMPONENT_FACTORY_DYNAMIC( \
        hpx::components::component<::performance_counters::examples::server:: \
            name_counter<action>>, \
        name_counter, "base_performance_counter");
```

Código 5.9: Novo macro de registo da fábrica de componentes

Em resumo, a preparação necessária para utilizar o contador consiste em:

- Registrar a fábrica do contador com o macro `REGISTER_COMPONENT_COUNTER_FACTORY`, cujo argumento é a ação que vai ser invocada pelo contador.
- Registrar no *main* o tipo do contador com a função `register_component_counter`, parametrizada com a ação para obter o valor e cujo argumento é o nome do contador.
- Registrar o nome do componente a monitorizar com a função `hpx::agas::register_name`, cujos argumentos são o nome e o GID do componente.

Juntando todos estes elementos num pequeno exemplo em que o alvo é um componente que funciona como um contador de inteiros e *get* é a ação usada para obter o valor atual, obtém-se:

```
int hpx_main() {
    usleep(1000000); //ms
    comp c = hpx::new_<server::comp>(hpx::find_here());
    hpx::agas::register_name("component", c.get_id());
    for (int i = 0; i < 1000000; i++)
        c.add(1);
    hpx::finalize();
    return 0;
}

int main(int argc, char* argv[]) {
    register_component_counter<::server::comp::get_action>("/example/name");
    hpx::init(argc, argv);
    return 0;
}
```



```

}
REGISTER_COMPONENT_COUNTER_FACTORY(::server::comp::get_action);

```

Código 5.10: Exemplo de utilização do novo contador

O `usleep` tem o propósito de testar o comportamento do contador antes do componente ser registado. Tal como pretendido, através da linha de comandos é possível observar o valor pré-definido, seguido pela evolução do valor do componente.

```

$ /api --hpx:print-counter=/example{locality#0/component}/name@-1 --hpx:print-
  counter-interval=2000
/example{locality#0/component}/name@-1,1,0.003699,[s],-1
/example{locality#0/component}/name@-1,1,1.004074,[s],117184
/example{locality#0/component}/name@-1,1,3.004144,[s],352955
/example{locality#0/component}/name@-1,1,5.004077,[s],584359
/example{locality#0/component}/name@-1,1,7.004325,[s],814563
/example{locality#0/component}/name@-1,1,9.004313,[s],1e+06

```

Código 5.11: Resultado da utilização do novo contador

Embora o resultado final deste objetivo seja algo simples em termos de funcionalidades, a sua flexibilidade em termos de tipos é algo que o torna aplicável em vários contextos. Também não obriga a alterações no código do componente a monitorizar, o que é sempre vantajoso.

Para além disso, serviu para determinar se o uso extensivo de *templates* na definição de novos contadores é viável. Embora seja perceptível que a interface de registo de contadores não foi projetada com este uso em mente, é mesmo assim uma possibilidade.

APEX

Para além dos contadores, existem outras ferramentas que lidam com o desempenho de aplicações HPX. Entre estas destaca-se o APEX (*Autonomic Performance Environment for eXascale*). Este capítulo resume o estudo que se fez desta ferramenta.

6.1 INTRODUÇÃO

O APEX é uma biblioteca de medição e adaptação de desempenho, principalmente para sistemas de tempo de execução distribuídos e assíncronos. O seu principal propósito consiste em adquirir medições das várias camadas (*hardware*, sistema operativo, sistema de tempo de execução, etc.) e utilizar em tempo real essa informação para afinar parâmetros da aplicação em execução [Huck et al. (2015)]. De forma secundária, o APEX também funciona como *profiler*. Através da integração com outras ferramentas como TAU e com formatos como o OTF2, as medições podem ser usadas para gerar perfis de execução para análises *post-mortem* [PI et al. (2019)].

O HPX e o APEX estão fortemente integrados, sendo que as respetivas implementações são circularmente dependentes [PI et al. (2019)]. Apesar de o APEX ter sido desenvolvido principalmente para suportar sistemas de execução ParallelX, também suporta outros modelos, como o OpenMP ou fios de execução C++ [APEX (2021)]. Naturalmente, o foco deste capítulo será apenas na utilização relativa ao HPX.

De acordo com os seus objetivos, a arquitetura do APEX está dividida em dois sistemas principais: o de introspeção, responsável pela recolha de dados, e o de políticas, responsável pelo controlo do desempenho.

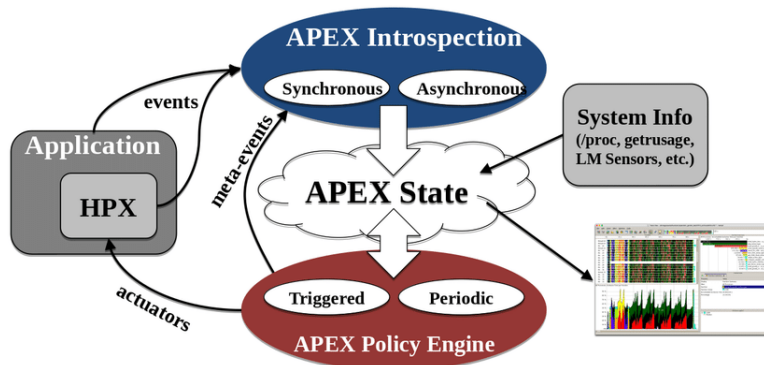


Figura 1: Arquitetura do APEX [Wagle et al. (2019)]

A introspeção no APEX é feita por inspetores síncronos e assíncronos [Wagle et al. (2019)].

A coleção de dados assíncrona, também denominada de *bottom-up*, consiste simplesmente em observar periodicamente medições de outras fontes, tais como contadores PAPI ou a diretoria /proc.

A coleção síncrona, ou *top-down*, é mais complexa. Esta consiste numa API baseada em eventos e em *event listeners*. Eventos incluem situações como:

- criação, destruição e outros eventos relacionados com fios de execução de nível utilizador (que em HPX correspondem a tarefas);
- começar e parar temporizadores;
- leituras de contadores do HPX;
- meta-eventos (eventos definidos pelo próprio utilizador).

Estes entram no ambiente APEX através da instrumentação do código com a API do APEX. Os eventos são gerados pelas chamadas para esta API. Por exemplo, o escalonador de fios de execução do HPX contém código APEX que gera eventos com a informação adequada de cada vez que ocorre alguma transição do estado de um fio de execução [PI et al. (2019)]. A leitura de contadores é outro exemplo de instrumentação. De cada vez que ocorre uma leitura de um contador, o nome e respetivo valor são passados ao APEX [PI et al. (2019)].

O APEX possui vários *listeners* que agem em conformidade com a ocorrência dos eventos [Huck et al. (2015)]. Vários *listeners* podem reagir ao mesmo evento, como é o caso para eventos relacionados com temporizadores.

Dos vários *listeners*, destaca-se o *profiling listener*. Este reage aos eventos dos temporizadores e das leituras de contadores, armazena os dados dos eventos numa fila de espera e, assincronamente, processa-os para gerar um perfil de execução [Huck et al. (2015)].

```
z@pc:~/Desktop/HPX-Performance-Counters/apex_unit_test/build$ export APEX_SCREEN_OUTPUT=1
z@pc:~/Desktop/HPX-Performance-Counters/apex_unit_test/build$ ./apex_action_count_test
fibonacci(10) == 55
elapsed time: 0.003231 [s]

Elapsed time: 0.0129446 seconds
Cores detected: 12
Worker Threads observed: 6
Available CPU time: 0.0776675 seconds

Counter                                : #samples | minimum | mean | maximum | stddev
-----
1 Minute Load average :          1    1.170    1.170    1.170    0.000

Timer                                : #calls | mean | total | % total
-----
APEX MAIN :              1    0.013    0.013   100.000
  async :                2    0.000    0.000    0.011
    async_launch_policy_dispatch :    5    0.000    0.000    0.463
      broadcast_call_shutdown_functions_action :    2    0.000    0.000    0.060
        call_shutdown_functions_action :    2    0.000    0.000    0.199
          fibonacci_action :       174    0.000    0.017   22.149
            load_components_action :    1    0.001    0.001    1.748
              primary_namespace_colocate_action :    2    0.000    0.000    0.429
                run_helper :    1    0.001    0.001    1.809
                  shutdown_all_action :    1    0.000    0.000    0.111
Total timers : 190
```

Figura 2: Perfil de execução de um programa que calcula o *fibonacci* de 10 através de ações

Outros *listeners* incluem o *TAU Listener* e o *OTF2 Listener*, responsáveis por passar as medições dos eventos às respetivas bibliotecas, e o *concurrency listener*, que mantém um registo da quantidade de fios de execução existentes em qualquer momento [Huck et al. (2015)].

As políticas do APEX são o mecanismo pelo qual este possibilita controlar e afinar a execução da aplicação. As políticas são regras que permitem executar tarefas como alterar parâmetros de acordo com estado do programa detetado pela introspeção [Wagle et al. (2019)]. Podem ser assíncronas, ocorrendo periodicamente, ou podem ser síncronas, sendo despoletadas pela ocorrência de eventos específicos [Wagle et al. (2019)]. Estes eventos podem ser os pré-definidos ou novos eventos definidos pelo utilizador [Wagle et al. (2019)].

6.2 API

Esta secção foca-se nas funcionalidades da API do APEX que são relevantes para o contexto desta dissertação. É abordada especificamente a versão C++ da API. Porém, a versão C é equivalente em termos de funcionalidades.

6.2.1 Tipos e Valores Pré-definidos

Antes de proceder à exploração das funções da API é útil abordar os tipos e valores pré-definidos mais importantes.

O tipo `apex_policy_handle` é um apontador cujo papel é identificar uma política em particular. Por sua vez, o tipo `apex_event_type` é uma enumeração dos tipos de eventos já definidos. A listagem dos tipos oferece uma noção das capacidades do APEX. A documentação oficial indica a existência dos seguintes valores que representam eventos:

- `APEX_INVALID_EVENT = -1` - evento inválido;
- `APEX_STARTUP = 0`, `APEX_SHUTDOWN` - inicialização e terminação do APEX;
- `APEX_NEW_NODE` - adição de um novo nodo HPX/processo;
- `APEX_NEW_THREAD`, `APEX_EXIT_THREAD` - criação e terminação de fios de execução do sistema operativo;
- `APEX_START_EVENT`, `APEX_RESUME_EVENT`, `APEX_STOP_EVENT`, `APEX_YIELD_EVENT` - alterações no estado de um temporizador (usados no HPX para medir tarefas);
- `APEX_SAMPLE_VALUE` - leitura de um contador;
- `APEX_PERIODIC` - política periódica;
- `APEX_CUSTOM_EVENT_1` ... `APEX_CUSTOM_EVENT_8`, eventos definidos pelo utilizador;
- `APEX_UNUSED_EVENT = APEX_MAX_EVENTS` - limite do número de eventos.

Contudo, existem mais 4 tipos de eventos documentados que foram descobertos no código fonte do APEX:

- APEX_DUMP - *output* de dados, por exemplo para a linha de comandos;
- APEX_RESET - reinício de um perfil de execução;
- APEX_SEND - APEX_RECV - envio e receção de parcelas.

A enumeração inclui 8 valores para eventos definidos pelo utilizador. Estes são irrelevantes e apenas existem devido a questões de testagem interna, sendo possível definir mais do que 8 novos eventos. O valor máximo de eventos, APEX_MAX_EVENTS, é de 128. Como os valores até 15 já estão ocupados, a quantidade máxima de novos eventos é de 113, quantidade certamente suficiente para a grande maioria dos casos.

6.2.2 Registo de políticas

O registo de políticas baseadas em eventos é efetuado com a função `apex::register_policy`. É possível registar uma função que é invocada conforme a ocorrência de um determinado tipo de evento. Também é possível registar simultaneamente a mesma função para um conjunto de tipo de eventos, o que resulta numa nova política para cada um.

```
apex_policy_handle apex::register_policy (
    const apex_event_type when,
    std::function<int (apex_context const*)> f
);

std::set<apex_policy_handle> apex::register_policy (
    std::set<apex_event_type> when,
    std::function<int (apex_context const*)> f
);
```

Código 6.1: Funções de registo de políticas APEX

O resultado retornado consiste em um ou múltiplos apontadores para as políticas registadas. Estes podem ser utilizados para anular os registos das respetivas funções com a função `apex::deregister_policy`.

As funções registadas recebem o `apex_context`, uma estrutura que contém a informação sobre a ocorrência de um evento.

```
typedef struct _context {
    apex_event_type event_type;
    apex_policy_handle* policy_handle;
    void * data;
} apex_context;
```

Código 6.2: Contexto de um evento APEX

O `event_type` é útil identificar que tipo de evento ocorreu, principalmente no caso de funções que são associadas a vários tipos de eventos. Por sua vez, o membro `policy_handle` aponta para a política à qual a função invocada está associada. Quando a política deixar de ser necessária, pode ser usado para cancelar o seu registo. O argumento `data` serve para transmitir dados específicos do evento. Cada tipo de evento tem a sua própria classe que pode ser obtida através de um `cast` do apontador `void`. No caso de eventos definidos pelo utilizador, é o utilizador que determina o conteúdo do apontador.

Em termos da API, as políticas periódicas consistem num caso específico do conceito mais generalizado de políticas de eventos. Na prática, existe um temporizador que periodicamente gera um evento do tipo `APEX_PERIODIC`.

O registo de políticas periódicas é efetuado através da função `apex::register_periodic_policy`. Em vez de receber o tipo do evento, esta recebe um intervalo de tempo em micro-segundos.

```
apex_policy_handle apex::register_periodic_policy(
    const unsigned long period,
    std::function<int(apex_context const*)> f
);
```

Código 6.3: Função de registo de políticas periódicas

6.2.3 Eventos definidos pelo utilizador

A utilização de eventos definidos pelo utilizador começa com o registo do novo tipo.

```
apex_event_type apex::register_custom_event(const std::string& name);
```

Código 6.4: Função de registo de eventos

O nome fornecido não é muito relevante. É usado na API do C mas nunca chega a ser usado na API do C++. Mais importante é o valor retornado. Este representa o novo tipo registado e, tal como explicado anteriormente, deve ser usado na função `apex::register_policy`. Também deve ser usado para indicar a ocorrência do evento com a função `apex::custom_event`.

```
void apex::custom_event(apex_event_type event_type, const void* event_data);
```

Código 6.5: Função de notificação da ocorrência de um evento definido pelo utilizador

O apontador `event_data` corresponde ao campo `data` do `apex_context`, podendo ser usado para passar informação à política do evento.

6.2.4 Profiling

A API do APEX permite o acesso aos mecanismos de temporização utilizados para construir o perfil de execução das tarefas. Esta é uma oportunidade para introduzir e testar as transições de estado da execução das tarefas.

Iniciar um temporizador requer um identificador, que pode ser uma *string* ou um apontador para uma função (apenas serve como identificação e não afeta a função). Um objeto que representa a execução é devolvido. Esse objeto é depois usado para parar o temporizador. Para além de ser iniciado e parado, um temporizador pode ser cedido e resumido.

```
apex_profiler_handle apex::start(const std::string &timer_name);
apex_profiler_handle apex::start(const apex_function_address function_address);
apex_profiler_handle apex::resume (const std::string &timer_name);
apex_profiler_handle apex::resume (const apex_function_address function_address);

void apex::stop(apex_profiler_handle the_profiler);
void apex::yield (apex_profiler_handle the_profiler);
```

Código 6.6: Funções dos temporizadores do APEX

Ceder e resumir temporizadores deve-se ao facto de que a execução de uma tarefa pode ser suspensa antes de ser completada e, conseqüentemente, pode ser resumida posteriormente. A diferença prática é que o uso destas funções permite interromper a medição do tempo sem incrementar o número de invocações no perfil de execução.

Como demonstração, o temporizador `t1` é iniciado, parado, iniciado e parado enquanto que o temporizador `t2` é iniciado, cedido, resumido e parado.

```
p = apex::start("t1");
usleep(100000); //100 ms
apex::stop(p);
usleep(100000);
p = apex::start("t1");
usleep(100000);
apex::stop(p);

p = apex::start("t2");
usleep(100000);
apex::yield(p);
usleep(100000);
p = apex::resume("t2");
usleep(100000);
apex::stop(p);
```

Código 6.7: Exemplo de utilização dos temporizadores do APEX

A diferença é expressa no perfil de execução final.

Timer	: #calls		mean		total		% total
<unknown>	2		0.000		0.000		0.006
APEX MAIN	1		0.611		0.611		100.000
broadcast_call_shutdown_functions_action	1		0.000		0.000		0.001
load_components_action	1		0.001		0.001		0.028
primary_namespace_colocate_action	1		0.000		0.000		0.001
run_helper	1		0.601		0.601		16.393
shutdown_all_action	1		0.000		0.000		0.004
t1	2		0.100		0.200		5.458
t2	1		0.200		0.200		5.459
APEX Idle					2.665		72.651

Total timers : 10							

Figura 3: Perfil de execução da aplicação exemplo

O APEX interpreta `t1` como duas chamadas distintas com duração média de 100 milissegundos e `t2` como apenas uma chamada com duração de 200 milissegundos.

Ao testar verificou-se que o uso simultâneo de `resume` e de `yield` apresenta alguma redundância, dado que apenas o uso de um deles tem o mesmo resultado.

```
p = apex::start("t3");
usleep(100000);
apex::yield(p);
usleep(100000);
p = apex::start("t3");
usleep(100000);
apex::stop(p);

p = apex::start("t4");
usleep(100000);
apex::stop(p);
usleep(100000);
p = apex::resume("t4");
usleep(100000);
apex::stop(p);
```

Código 6.8: Exemplo de utilização dos temporizadores do APEX sem a função `resume`

Timer	: #calls		mean		total		% total
<unknown>	2		0.000		0.000		0.003
APEX MAIN	1		1.212		1.212		100.000
broadcast_call_shutdown_functions_action	1		0.000		0.000		0.001
load_components_action	1		0.001		0.001		0.018
primary_namespace_colocate_action	1		0.000		0.000		0.001
run_helper	1		1.202		1.202		16.526
shutdown_all_action	1		0.000		0.000		0.003
t1	2		0.100		0.200		2.752
t2	1		0.200		0.200		2.754
t3	1		0.200		0.200		2.753
t4	1		0.200		0.200		2.754
APEX Idle					5.269		72.435
Total timers : 12							

Figura 4: Perfil de execução da aplicação sem a função `resume`

Isto acontece pois no código fonte é usada uma *flag*, que é tanto alterada por um como pelo outro.

Por fim, a função `sample_value` pode ser usada para adicionar medições ao perfil de execução.

```
void apex::sample_value (const std::string & name, const double value)
```

Código 6.9: Assinatura da função `sample_value`

A utilização destas funções leva à ativação das políticas registadas, tal como se fosse um evento interno.

6.3 FUNCIONAMENTO INTERNO E INTEGRAÇÃO COM O HPX

O objetivo desta secção é descrever como ocorre a transmissão de informação do HPX para o APEX, quais os aspetos que devem ser considerados na sua utilização e estabelecer um base de conhecimento para alterações futuras.

O funcionamento interno do APEX pode ser sintetizado em três etapas: i) o HPX fornece os dados de um evento ao APEX, ii) o APEX notifica os *listeners* registados e iii) estes reagem em conformidade.

Nesta secção serão exploradas duas interações diferentes. Para introduzir os mecanismos de integração com o HPX e os de funcionamento interno, é abordado o fluxo de informação desde a leitura de um contador até à invocação das políticas registadas pelo utilizador. Depois, são observados os eventos *start*, *stop*, *yield* e *resume*, cuja integração é a mais complexa e apresenta características únicas.

6.3.1 Leitura de contadores - Integração HPX/APEX

Inicialmente, e como descrito em [PI et al. \(2019\)](#), o HPX invocava diretamente as funções da API do APEX para indicar a ocorrência dos eventos. Na função responsável pela leitura e impressão periódica dos contadores, a função `apex::sample_value` era invocada. Ao consultar o código fonte do HPX, verificou-se que em versões mais recentes esta chamada foi substituída pela chamada da função `hpx::util::external_timer::sample_value`. O propósito desta alteração é abstrair o APEX de forma a facilitar a gestão das dependên-

cias circulares. No *namespace* `external_timer` é mantido um conjunto de apontadores para funções onde são registadas as funções do APEX. As funções expostas pelo `external_timer` invocam as funções registadas nesses apontadores.

```
static inline void sample_value(const std::string& name, double value)
{
    if (sample_value_function != nullptr)
        sample_value_function(name, value);
}
static inline void sample_value(
    hpx::performance_counters::counter_info const& info, double value)
{
    if (sample_value_function != nullptr)
        sample_value_function(info.fullname_, value);
}
```

Código 6.10: Definições de `hpx::util::external_timer::sample_value`

O apontador é exportado para poder ser referenciado fora da biblioteca HPX.

```
typedef void sample_value_t(const std::string&, double);
(...)
HPX_CORE_EXPORT extern sample_value_t* sample_value_function;
```

Código 6.11: Definição e exportação do apontador para a função do evento *sample_value*

A função de registo dos apontadores também é exportada para poder ser utilizada pelo APEX.

```
HPX_CORE_EXPORT void register_external_timer(
    registration_t& registration_record);
```

Código 6.12: Exportação da função que permite o registo de funções para os eventos

O tipo `registration_t` é uma estrutura com um apontador para a função a registar e uma *flag* a indicar o seu tipo.

```
typedef struct registration {
    functions_t type;
    union {
        init_t* init;
        (...)
        sample_value_t* sample_value;
        (...)
        stop_t* stop;
    } record;
} registration_t;
```

Código 6.13: Estrutura de um registo

Na inicialização do APEX, é preenchido e aplicado o registo de cada função.

```
reg.type = hpx::util::external_timer::sample_value_flag;
reg.record.sample_value = &sample_value_adapter;
hpx::util::external_timer::register_external_timer(reg);
```

Código 6.14: Registo da função `apex::sample_value` no APEX

Antes de ser registada, a função `apex::sample_value` tem que ser embrulhada num adaptador.

```
void sample_value_adapter(const std::string &name, double value) {
    APEX_TOP_LEVEL_PACKAGE::sample_value(name, value, false);
}
```

Código 6.15: Adaptador da função `apex::sample_value`

As funções registadas nos apontadores não necessitam de ser as funções do APEX. Ou seja, outra vantagem desta solução é que simplifica o desenvolvimento e integração de novas ferramentas que pretendam tirar vantagem do sistema de eventos já existente.

6.3.2 Leitura de contadores - Funcionamento interno do APEX

O APEX tem uma instância por localidade. Cada instância tem uma lista de *listeners* que são alertados para a ocorrência dos eventos. Em cada função da API de eventos, os dados do evento são empacotados numa classe específica antes de serem fornecidos aos *listeners* registados. Para descobrir a classe é preciso consultar o código fonte, já que na documentação essa informação está ausente.

```
class sample_value_event_data : public event_data {
public:
    std::string * counter_name;
    double counter_value;
    bool is_threaded;
    bool is_counter;
    sample_value_event_data(int thread_id, std::string counter_name, double
        counter_value, bool threaded);
    ~sample_value_event_data();
};
```

Código 6.16: Classe `sample_value_event_data`

A classe `event_listener` têm um método virtual para cada tipo de evento. Os nomes dos métodos seguem o formato "on_x", sendo x o evento. Cada *listener* deve herdar a classe `event_listener`, o que força a implementação de todos os métodos virtuais.

```

void sample_value(const std::string &name, double value, bool threaded)
{
    (...)
    sample_value_event_data data(tid, name, value, threaded);
    if (_notify_listeners) {
        for (unsigned int i = 0 ; i < instance->listeners.size() ; i++) {
            instance->listeners[i]->on_sample_value(data);
        }
    }
    (...)
}

```

Código 6.17: Notificação dos *listeners* registrados para os eventos de leitura de contadores

Um detalhe que depois será importante é que os *listeners* são notificados pela ordem em que foram registrados. Em particular, o `policy_handler` é invocado após o `profile_listener`.

A função `on_sample_value` do `profile_listener`, classe responsável por construir o perfil de execução, coloca os dados numa fila de espera para serem processados assincronamente.

O *listener* que trata das políticas definidas pelo utilizador, mantém um conjunto das funções registradas para cada tipo de evento. As suas funções "on_x" invocam a função `call_policies`.

```

void policy_handler::on_sample_value(sample_value_event_data &data) {
    call_policies(sample_value_policies, &data, APEX_SAMPLE_VALUE);
}

```

Código 6.18: Função `on_sample_value` do `policy_handler`

Algumas funções "on_x" alteram os dados enviados a `call_policies`, sendo por isso necessário verificar esta etapa no código fonte para descobrir o tipo de dados incluído no contexto. A função `call_policies` trata de criar o contexto para cada política e de colocar os dados recebidos em `context.data`.

```

void policy_handler::call_policies(
    const std::list<std::shared_ptr<policy_instance> > & policies,
    void *data,
    const apex_event_type& event_type) {
    (...)
    for(const std::shared_ptr<policy_instance>& policy : policies) {
        apex_context my_context;
        my_context.event_type = event_type;
        my_context.policy_handle = nullptr;
        my_context.data = data;
        (...)
        const bool result = policy->func(my_context);
        (...)
    }
}

```

Código 6.19: Preenchimento do contexto do evento de leitura de um contador

6.3.3 Eventos de execução de tarefas

Na integração com o HPX, o APEX também utiliza as funções `start`, `stop`, `yield` e `resume`. Contudo, em vez de receberem uma string ou função como argumento, recebem um `task_wrapper`.

```
void start(std::shared_ptr<task_wrapper> task_wrapper_ptr);
void stop(std::shared_ptr<task_wrapper> task_wrapper_ptr);
void yield(std::shared_ptr<task_wrapper> task_wrapper_ptr);
void resume(std::shared_ptr<task_wrapper> task_wrapper_ptr)
```

Código 6.20: Assinaturas das funções de temporizadores usadas internamente

A classe `task_wrapper` é declarada no HPX. O APEX (ou qualquer outra API que queira interagir com esta parte do HPX) deve definir a classe. O `task_wrapper` definido pelo APEX contém informação como o identificador interno, que função está a ser executada e as dependências da tarefa.

Um dos campos do `task_wrapper` é um objeto `profiler`. O `profiler` regista detalhes sobre a execução como o tempo inicial, o tempo final e o total de *bytes* alocados, entre outros. Ambos têm um apontador para o outro. Os campos de ambas as classes podem ser consultados nos anexos [A.2](#) e [A.3](#).

A função `apex::start` é fornecida pelo HPX com um apontador para um `task_wrapper` já inicializado, apesar de o HPX desconhecer a definição da classe. A resposta para esta contradição é o evento *new task*. A função `new_task` é invocada pelo HPX nas zonas onde são geradas novas tarefas, tal como na implementação do mecanismo `apply` onde, nas versões que correspondem a ações cujo alvo é local, vão ser geradas novas tarefas na localidade atual. O apontador para o `task_wrapper` pode então ser armazenado pelo HPX, juntamente com outra informação sobre a tarefa.

O código do HPX não está instrumentado diretamente com as funções dos temporizadores. Em vez disso é utilizado um `scoped_timer`. Um `scoped_timer` é um objeto que encapsula as funções de temporização e oferece algumas proteções adicionais. As tarefas internas ao APEX (`task_wrapper` não inicializado) são ignoradas de forma a evitar sobrecargas e recursividade infinita. O construtor recebe um `task_wrapper`, armazena-o e invoca automaticamente a função `start`. O objeto impede invocações de `stop` e `yield` se a tarefa já estiver parada. As funções `stop` e `yield` utilizam o `task_wrapper` armazenado. O destrutor do objeto invoca a função `stop`. O código fonte da classe está disponível no anexo [A.4](#).

O `scoped_timer` é utilizado no ciclo do HPX responsável pelo escalonamento das tarefas/fios de execução do HPX.

```
void scheduling_loop(...)

    while (true)
    {
        thread_data* thrd = next_thrd;

        (...)
    }
```

```

#if defined(HPX_HAVE_APEX)

    util::external_timer::scoped_timer profiler(
        thrd->get_timer_data());

    thrd_stat = (*thrd)(context_storage);

    if (thrd_stat.get_previous() == thread_schedule_state::terminated)
    {
        profiler.stop();
        // just in case, clean up the now dead pointer.
        thrd->set_timer_data(nullptr);
    }
    else
    {
        profiler.yield();
    }
#else
    thrd_stat = (*thrd)(context_storage);
#endif

```

Código 6.21: Instrumentação do escalonador de tarefas

A variável `thrd` é um apontador para o objeto representativo do fio de execução HPX prestes a ser executado. O seu `task_wrapper`, ainda não usado desde que foi gerado por `new_task` ou reutilizado de outra execução anterior, é utilizado para instanciar um novo `scoped_timer`.

Depois de instanciado o temporizador, o que resulta no começo da medição de tempo, é começada a execução do fio de execução. O tipo `thread_data` é um functor e a sua invocação leva à execução do fio de execução e retorna informação sobre o estado da execução.

```

inline coroutine_type::result_type thread_data::operator() (
    hpx::execution_base::this_thread::detail::agent_storage* agent_storage)
{
    if (is_stackless_)
    {
        return static_cast<thread_data_stackless*>(this)->call();
    }
    return static_cast<thread_data_stackful*>(this)->call(agent_storage);
}

```

Código 6.22: Função functor de `thread_data`

De seguida o temporizador é parado com `stop` ou `yield`, dependendo da informação recebida.

Caso o HPX não tenha sido instalado com o APEX, o functor é invocado sem qualquer uso de temporizadores.

O HPX nunca utiliza a função de resumo de execução. Como foi verificado anteriormente em 6.2.4, o uso do `resume` não é obrigatório para o APEX efetuar medições corretas. Apesar de não afetar a construção do perfil de execução pelo APEX, a ausência desta informação adicional é um fator negativo para outras utilizações da API dos eventos.

Voltando ao lado do APEX, a função `apex::start` trata de propagar os dados por todos os *listeners* registados invocando as suas funções `on_start`, tal como acontecia para os eventos de leitura de contadores. O *listener* responsável pela construção do perfil de execução é o `profiler_listener`. A sua função `on_start` invoca diretamente a função `_common_start`. Essa função, que também é usada por `on_resume`, utiliza o `task_wrapper` fornecido pelo HPX para gerar um `profiler`.

```
inline bool profiler_listener::_common_start(
    std::shared_ptr<task_wrapper> &tt_ptr, bool is_resume)
{
    (...)
    profiler * p = new profiler(tt_ptr, is_resume);
    p->guid = tt_ptr->guid;
    thread_instance::instance().set_current_profiler(p);
    (...)
}
```

Código 6.23: Inicialização do `profiler` pelo `profiler_listener`

Novamente na função `apex::start`, o `profiler` construído é adicionado ao `task_wrapper`.

```
tt_ptr->prof = thread_instance::instance().get_current_profiler()
```

Código 6.24: Armazenamento do `profiler` no `task_wrapper`

Quando a execução de uma tarefa é terminada ou suspendida, o mesmo `task_wrapper` volta a ser fornecido ao APEX. O `profiler_listener` trata de parar o `profiler`, o que resulta na medição do tempo de execução. Além disso, coloca-o numa fila de espera para eventualmente ser processado e usado para completar o perfil de execução da aplicação.

```
inline void profiler_listener::_common_stop(
    std::shared_ptr<profiler> &p, bool is_yield)
{
    (...)
    p->stop(is_yield);
    (...)
    push_profiler(_pls.my_tid, p);
}
```

Código 6.25: Paragem do `profiler` pelo `profiler_listener`

6.3.4 Eventos de troca de parcelas

A camada de transporte do HPX em cada localidade é composta por dois elementos, o *parcel port* e o *parcel handler*.

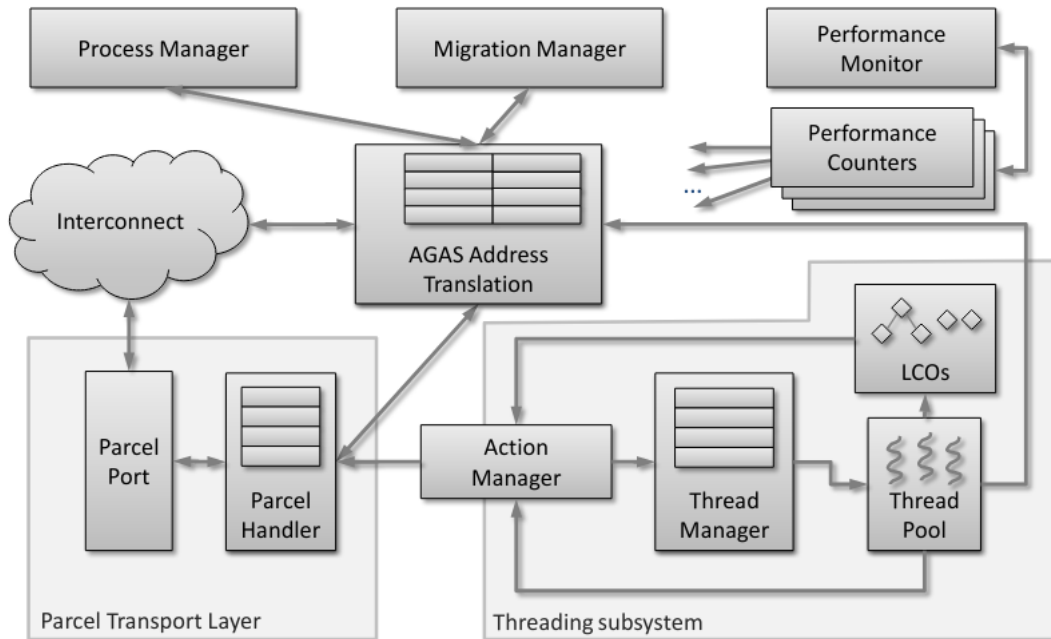


Figura 5: Arquitetura do HPX [Heller et al. (2013)]

Quando o AGAS determina que uma ação tem um destinatário remoto, o *parcel handler* trata de serializar os dados [Heller (2019)]. A mensagem depois é enviada assincronamente pelo *parcel port* para a localidade destino [Heller et al. (2013)].

Na localidade destino, o *parcel port* recebe e desserializa a mensagem [Heller et al. (2013)]. O *parcel handler* verifica se o objeto destino se encontra na localidade e, caso isto se verifique, o *action manager* gera um novo fio de execução HPX [Heller et al. (2013)].

A transmissão de informação sobre o envio e a recepção de parcelas entre o HPX e o APEX dá-se através das funções `send` e `recv`.

```

APEX_EXPORT void send (uint64_t tag, uint64_t size, uint64_t target);
APEX_EXPORT void recv (
    uint64_t tag, uint64_t size, uint64_t source_rank, uint64_t source_thread);
  
```

Código 6.26: Assinatura das funções dos eventos das mensagens

Envio de parcelas

No código do *parcel handler* encontra-se o uso de `hpx::util::external_timer::send()`. Mais especificamente na função `parcel_sent_handler`, que é invocada após o envio de uma parcela.

```
void parcel_sent_handler(
    parcelhandler::write_handler_type& f,
    std::error_code const& ec, parcel const& p)
{
    (...)
    #if defined(HPX_HAVE_APEX) && defined(HPX_HAVE_PARCEL_PROFILING)
        util::external_timer::send(
            p.parcel_id().get_lsb(),
            p.size(),
            p.destination_locality_id()
        );
    #endif
}
```

Código 6.27: Notificação do APEX do envio de uma parcela

O uso de `util::external_timer::send` invoca a função `apex::send`, registrada pelos mecanismos de registo já explorados. Nesta, os dados são empacotados na classe `message_event_data` antes de serem distribuídos.

```
class message_event_data : public event_data {
public:
    uint64_t tag;
    uint64_t size;
    uint64_t source_rank;
    uint64_t source_thread;
    uint64_t target;
    (...)
}
```

Código 6.28: Classe `message_event_data`

Dos 5 campos, apenas 3 são fornecidos ao APEX. O campo `source_rank` é obtido pelo próprio APEX consultando a localização da sua instância. O campo `source_thread` ainda não estava implementado na versão do APEX utilizada.

```
if (_notify_listeners) {
    message_event_data data(tag, size, instance->get_node_id(), 0, target);
    if (_notify_listeners) {
        for (unsigned int i = 0 ; i < instance->listeners.size() ; i++) {
            instance->listeners[i]->on_send(data);
        }
    }
}
```

Código 6.29: Adição da localidade fonte na função `apex::send`

Receção de parcelas

O HPX suporta os protocolos de comunicação TCP e MPI [Heller (2019)]. No código dos *parcel ports* de ambos, após a receção de uma transmissão é invocada a função `decode_parcels`. Como o nome da função sugere, uma mensagem TCP ou MPI pode conter informação de várias parcelas.

Seguindo a cadeia de invocações, chega-se à função `decode_message_with_chunks`. Nessa função os *bytes* recebidos são colocados num objeto `archive` que permite a desserialização de objetos C++. Na função está presente um ciclo com uma iteração para cada parcela contida no arquivo.

```
for(std::size_t i = 0; i != parcel_count; ++i)
{
    (...)
    parcel p;
    bool migrated = p.load_schedule(archive, num_thread , deferred_schedule);
}
```

Código 6.30: Desserialização das parcelas recebidas

Em cada iteração é instanciado um objeto `parcel` e invocado o método `load_schedule()`, onde o arquivo é lido e os campos do objeto são preenchidos. Após a leitura do arquivo, o APEX pode ser notificado.

```
bool parcel::load_schedule(serialization::input_archive & ar,
    std::size_t num_thread, bool& deferred_schedule)
{
    load_data(ar);
    (...)
    #if defined(HPX_HAVE_APEX) && defined(HPX_HAVE_PARCEL_PROFILING)
        util::external_timer::recv(data_.parcel_id_.get_lsb(), size_,
            naming::get_locality_id_from_gid(data_.source_id_),
            reinterpret_cast<std::uint64_t>(action_->get_parent_thread_id().get()));
    #endif
}
```

Código 6.31: Notificação da receção de uma parcela

O método `load_data` preenche o campo `data_` do objeto `parcel`. O identificador da parcela e a localidade de origem fornecidos ao APEX são retirados de `data_`. Contudo, o tamanho da mensagem é dado pelo campo `size_` que nunca chega a ser inicializado, sendo sempre 0. Ao contrário de no evento *send*, o `source_thread` é fornecido. Este é convertido do formato hexadecimal original para um inteiro. De resto, o funcionamento dentro do APEX é idêntico ao do evento *send*, sendo novamente usada a classe `message_event_data`.

6.3.5 Conclusão

Com esta análise do funcionamento interno é possível compreender melhor as capacidades e as limitações da ferramenta. Para além disso, permite colmatar as falhas da documentação. De particular relevância é a capacidade de verificar que tipos de dados são incluídos no contexto dos eventos, um aspeto importante que está ausente na documentação. A exploração do código onde o HPX é instrumentado foi vital para compreender que informação adicional pode ser extraída, tanto do HPX como do APEX, e usada nas políticas.

HPXTRACE

Ao explorar as capacidades da API do APEX surgiu a ideia de o utilizar para implementar uma API inspirada no Dtrace. O objetivo não é desenvolver um equivalente direto do Dtrace. Aliás, várias características do Dtrace (como instrumentação do *kernel* do sistema operativo) seriam inviáveis de replicar no contexto desta dissertação. A intenção é aproveitar alguns conceitos e aplicá-los ao HPX usando o APEX como base.

7.1 DTRACE

O Dtrace (Dynamic Tracing) é uma ferramenta de análise de desempenho e de *troubleshooting* disponível nos sistemas operativos Solaris, FreeBSD e Mac OS X [Gregg and Mauro (2011)]. É capaz tanto de rastreamento dinâmico como de rastreamento estático. O rastreamento dinâmico está dependente da presença de *probes*, sondas embebidas em vários níveis, como no *kernel* do sistema operativo ou em aplicações do utilizador. Estes sensores são acionados, ou disparam segundo a terminologia do Dtrace, mediante a ocorrência de certos eventos e colecionam informação nesse instante.

O que torna esta ferramenta interessante é a disponibilização de uma linguagem simples de *scripting* que permite associar às sondas conjuntos de operações básicas sobre dados adquiridos. O dinamismo é devido ao facto de que as sondas só são ativadas quando têm ações associadas, reduzindo assim o impacto no desempenho. O uso da ferramenta consiste num executável que recebe o *script*. Para além de poder ser definido num ficheiro, o *script* também pode ser escrito diretamente na linha de comandos e fornecido ao Dtrace, o que simplifica e acelera a sua utilização.

Uma breve abordagem da API do Dtrace é importante. O objetivo não é explorar em detalhe nem explicar a sua utilização mas sim enquadrar o trabalho desenvolvido. Por essa razão serão omitidos detalhes referentes a aspetos como a compilação e os processos.

Um programa Dtrace consiste em cláusulas do seguinte formato:

```
description
/ predicate /
{
    actions
}
```

Código 7.1: Formato das cláusulas Dtrace

A descrição indica a quais sondas se aplica a cláusula. Existe um conjunto substancial de sondas, que são agrupadas em diferentes *providers*. Existem *providers* como o *syscall* e o *profile*. O primeiro fornece sondas que lidam com chamadas de sistemas, como a abertura ou a leitura de ficheiros. O segundo permite efetuar ações periodicamente. Dentro de cada *provider* as sondas podem estar subdivididas em módulos e funções. O formato da descrição reflete esta organização: *provider:module:function:name*. Cada elemento pode ser omitido e caracteres de *pattern matching* são suportados.

Cada sonda pode disponibilizar argumentos, que são variáveis com informação obtida pela sonda. No caso das sondas do *provider* *syscall*, correspondem aos valores dos argumentos das próprias chamadas de sistema. Também existem variáveis *built-in* sempre disponíveis. Estas contêm informação como o pid do processo ou o cpu onde está a ser executado.

O predicado determina se as ações devem ser executadas quando uma sonda dispara. A sua ausência significa que as ações devem sempre ser executadas.

As ações suportam os tipos de dados básicos e respetivas operações que se esperam numa linguagem, para além de um pequeno conjunto de funções.

A obtenção dos dados por si só é útil, mas a capacidade de agrupá-los num formato adequado é essencial para facilitar a chegada às conclusões corretas. Algo particularmente relevante neste aspeto são as agregações disponibilizadas pelo Dtrace. As agregações funcionam através de dois elementos. As variáveis de agregação, que equivalem a dicionários, e as funções de agregação.

```
@name[ keys ] = aggfunc ( args );
```

Código 7.2: Formato de agregações no Dtrace

A chave pode ser um único valor ou uma lista. As funções de agregação operam sobre os valores já agregados e o novo valor a adicionar.

Function Name	Arguments	Result
count	none	The number of times called.
sum	scalar expression	The total value of the specified expressions.
avg	scalar expression	The arithmetic average of the specified expressions.
min	scalar expression	The smallest value among the specified expressions.
max	scalar expression	The largest value among the specified expressions.
lquantize	scalar expression, lower bound, upper bound, step value	A linear frequency distribution, sized by the specified range, of the values of the specified expressions. Increments the value in the <i>highest</i> bucket that is <i>less</i> than the specified expression.
quantize	scalar expression	A power-of-two frequency distribution of the values of the specified expressions. Increments the value in the <i>highest</i> power-of-two bucket that is <i>less</i> than the specified expression.

Figura 6: Agregações disponíveis no Dtrace [Dynamic Tracing Guide (2008)]

Tal como foi mencionado, o Dtrace suporta a inserção de sondas no código fonte das aplicações do utilizador. Ignorando os detalhes do processo de compilação, o conceito pode ser resumido da seguinte forma.

A primeira etapa é a definição da nova sonda num ficheiro separado. Nesse ficheiro é definido um novo *provider* e respetivas sondas. Para cada sonda é preciso definir o nome e o tipo de cada argumento, caso existam.

No código fonte utiliza-se o macro `DTRACE_PROBE` para sinalizar a localização onde a sonda vai ser acionada. O macro recebe como argumentos o nome do *provider* e da sonda a disparar. Caso a sonda receba argumentos, o número destes deve sufixado ao macro (ex: `DTRACE_PROBE2`).

Após a sua definição e a instrumentação no código, a sonda pode ser usada da mesma forma que uma sonda já existente.

7.2 HPXTRACE

Apesar de nem todas as características do Dtrace exploradas serem aplicáveis no contexto do HPX, o conceito de sondas, pré-definidas ou personalizada, que fornecem dados quando disparam e às quais podem ser associadas ações através da linha de comandos foi o que pareceu mais adequado a adaptar no HPX. Aspectos como a existência de um executável separado e necessidade de compilar novas sondas foram descartados.

Clarificando o resultado pretendido, este consiste num mecanismo em que seja possível, através da linha de comandos ou através de ficheiros, definir comportamentos básicos associados à ocorrência de determinados eventos e medições. A principal diferença em relação ao Dtrace exibe-se na fonte dos dados. Em vez da instrumentação, as fontes serão os contadores do HPX e as medições e eventos disponibilizados pelo APEX. Mantém-se a capacidade de o utilizador definir as suas próprias sondas.

É preciso adaptar o conceito às características do HPX e do APEX, em particular à existência de localidades. Cada localidade do HPX tem a sua respetiva instância do APEX. Isto significa que para reagir a ocorrências de eventos numa localidade, as sondas devem ser registadas na mesma. A única exceção são as sondas com acesso aos contadores HPX, pois todos os eventos associados às leituras dos mesmos ocorrem na localidade 0.

Também é preciso considerar a natureza distribuída do HPX no que toca aos dados, o que leva à existência de três contextos distintos. Variáveis de sonda apenas existem durante a execução da sonda, variáveis locais são partilhadas por todas as sondas de uma localidade e variáveis globais são partilhadas por todas as sondas.

Um problema causado por esta abordagem é a possibilidade de ocorrerem *race conditions*, principalmente com variáveis partilhadas que são acedidas frequentemente. Para lidar com esse problema existe um mecanismo básico de sincronização intra e inter localidades. As variáveis de sonda são úteis neste aspeto pois facilitam cálculos intermédios sem qualquer preocupação em relação a sincronização e desempenho.

O conceito de agregações é reutilizado e é disponibilizado o mesmo conjunto de funções de agregação. As variáveis de agregação existem somente no contexto local. Contudo, como todas as funções de agregação suportadas podem ser decompostas (o resultado de aplicá-las a subconjuntos de dados e aplicá-las novamente aos resultados é idêntico à aplicação sobre o conjunto inteiro), o valor global pode ser calculado quando requerido.

Para conseguir esta disposição de dados e acessos inter-localidade, são utilizados componentes.

O projeto não pretende concluir com uma ferramenta totalmente acabada e com um grande conjunto de funcionalidades, mas sim servir como uma prova de conceito e como uma base para futuras melhorias.

O projeto desenvolvido usa como base o APEX e a biblioteca de processamento de texto Boost.Spirit.

7.3 DESCRIÇÃO DA API

7.3.1 Estrutura das cláusulas

A estrutura descrição-predicado-ações permanece inalterada.

```
description
/ predicate /
{
    actions
}
```

Código 7.3: Estrutura das cláusulas HpxTrace

A organização hierárquica das sondas Dtrace em *provider*, *module*, *function* e *name* não é reutilizada, dado que não existirá uma quantidade de sondas que a justifique. Além disso, é preciso considerar a natureza distribuída do HPX. Tendo em consideração estes fatores, a descrição das sondas HpxTrace segue o seguinte formato:

```
tipo[localidade0, ..., localidadeN]::arg1::arg2::
```

Código 7.4: Descrição das sondas HpxTrace

O tipo da sonda, que é obrigatório, pode referir-se a um dos tipos existentes ou a um tipo de sonda definido pelo utilizador. A presença dos restantes elementos varia conforme o tipo da sonda.

A lista das localidades indica onde a sonda deve ser registada. Os elementos da lista devem ser os índices das respetivas localidades. Uma lista vazia ([]) significa que a sonda deve ser registada em todas as localidades. Algumas sondas estão associadas a eventos que apenas ocorrem na localidade 0. Nesses casos a lista é omitida.

Existem sondas que recebem um ou dois argumentos. Dependendo da sonda estes podem ser obrigatórios ou opcionais. Os argumentos são delimitados por ::. A seleção de :: como delimitador em lugar de uma opção mais ortodoxa, como parênteses, deve-se a duas razões. Em primeiro lugar, como as sondas vão lidar com contadores HPX é mais fácil em termos de processamento de texto se o delimitador não for utilizado nos nomes dos contadores. Além disso, facilita a leitura do *script* na linha de comandos. Em determinados casos, os argumentos são listas e seguem a mesma sintaxe que a lista de localidades.

7.3.2 Definição da linguagem do predicado e das ações

As ações de cada cláusula são separadas por ; .

Variáveis

A linguagem suporta dois tipos de valores: números e *strings*. As *strings* são delimitadas por " e podem conter qualquer outro símbolo. Estes valores podem ser armazenados em variáveis escalares ou em dicionários. Cada dicionário apenas contém elementos de um tipo. As chaves são listas de valores que podem ser de qualquer tipo.

Os nomes das variáveis podem conter letras, dígitos e *underscores*. Para identificar o contexto das variáveis, são adicionados prefixos. As variáveis da sonda são prefixadas com o símbolo &, as variáveis locais com o símbolo : e as variáveis globais com o símbolo #.

A operação mais básica é a de atribuição de um valor a uma variável escalar. Não é necessário especificar o tipo da variável, este é induzido através do valor atribuído.

```
n = 1.5;
s[n] = "Hello world!";
```

Código 7.5: Atribuição de variáveis

Uma variável escalar não pode ser acedida sem antes lhe ter sido atribuída um valor. O mesmo acontece com os dicionários, que precisam de ser inicializados com pelo menos um valor. Após isso, qualquer valor associado a uma chave pode ser acedido sem atribuição prévia. No caso de dicionários numéricos isto resulta no valor 0 e no caso de dicionários de *strings*, resulta uma *string* vazia.

Para além da possibilidade de definir novas variáveis, podem ser usadas variáveis pré-definidas. Cada sonda tem o seu conjunto de variáveis pré-definidas. Como estas variáveis contêm informação sobre o evento que causou o disparo, são variáveis de sonda prefixadas com &.

Operações

As operações aritméticas básicas de adição, subtração, multiplicação, divisão e módulo podem ser aplicadas a variáveis e valores numéricos através dos operadores $+$, $-$, \times , $/$ e $\%$. Também podem ser usados parênteses para agrupar as expressões aritméticas. Para além disso, é possível inverter um valor através de $-$. Aplica-se a prioridade de operações normal.

No que diz respeito a *strings*, é suportada a concatenação através do operador $+$.

No predicado é possível efetuar várias operações booleanas sobre valores e variáveis. Existem os operadores de comparação $==$, $!=$, $<$, $<=$, $>$, $>=$ que permitem comparar números ou *strings*. Várias expressões podem ser avaliadas através dos operadores $&&$, $||$ e $!$. Os predicados também permitem parênteses. A ordem de prioridades é $()$, $&&$ e $||$.

Funções

São fornecidas ainda funções adicionais com fins específicos. Existe necessariamente a função `print` que imprime qualquer valor ou variável na linha de comandos. Para converter entre os dois tipos suportados, existem as funções `str` (número para *string*) e `dbl` (*string* para número). Para efetuar arredondamentos existem as

função `round`, `ceil` e `floor` que arredondam, respetivamente, para o inteiro mais próximo, por excesso e por defeito.

Para auxiliar na organização dos dados, existe um conjunto de funções relativo às localidades. A função `locality_name` retorna o nome da localidade onde a sonda disparou e a função `locality_rank` o seu *rank*. A função `nl` retorna o número total de localidades.

A última função é a função `timestamp`, que indica em milissegundos o tempo decorrido desde a inicialização da API. É importante ter em consideração que este valor é local. Ou seja, não deve ser usada para comparações entre localidades.

Agregações

A linguagem reaproveita o conceito de agregações do Dtrace. Começando pela sintaxe, esta é equivalente.

```
@name[ keys ] = aggfunc ( args );
```

Código 7.6: Formato de agregações no HpxTrace

As variáveis de agregação são identificadas pelo prefixo `@` e são indexadas com uma lista de chaves separadas por vírgulas. As chaves podem ser expressões do tipo numérico ou *string*. As funções de agregação são as mesmas descritas na figura 6.

Cada variável de agregação apenas pode estar associada a uma função de agregação. Tentativas de agregar com uma função diferente resultam em erro. Para além do valor a agregar, a função de agregação `lquantize` tem a particularidade de receber argumentos que definem os parâmetros da agregação. Tentativas de definir parâmetros diferentes para a mesma variável também resultam em erro.

A função `print` suporta agregações. Para consolidar os valores de agregações de várias localidades, existe a função `global_print`. Esta função pode receber as localidades pretendidas, caso contrário são consideradas todas as localidades.

```
print(@agg);
global_print(@agg);
global_print(@agg, 1, 3);
```

Código 7.7: Impressão de agregações

Diretivas de sincronização

Disparos simultâneos da mesma sonda ou de sondas que partilhem variáveis podem causar acessos concorrentes às mesmas variáveis. Isto pode causar problemas, particularmente no caso de cláusulas que sejam executadas muito frequentemente. Sempre que possível a melhor opção é utilizar as agregações, pois estas possuem mecanismos de sincronização internos para lidar corretamente com inserções concorrentes. Para lidar com as restantes situações foram introduzidas diretivas de exclusão mútua `lock(x)` e `unlock(x)`, onde `x` é uma lista de valores. Após uma sonda executar a ação `lock(x)`, qualquer outra sonda que tentar executar `lock(x)` ficará

bloqueada até a sonda original executar `unlock(x)`. Para sincronização entre sondas da mesma localidade, o acesso a variáveis locais pode ser controlado através das funções `local_lock` e `local_unlock`. O mesmo é possível no contexto global através das funções `global_lock` e `global_unlock`.

Erros

Existe um sistema rudimentar de detecção de erros de sintaxe que é aplicado no início da execução. Se um predicado tiver um erro então é gerada uma mensagem de erro com o predicado incluído.

```
./probes --script 'probeX/"x" > 100/{print(x);print(y);}'

<unknown>
terminate called after throwing an instance of 'std::runtime_error'
  what(): Syntax error in predicate:
/"x" > 100/
```

Código 7.8: Exemplo de um erro de sintaxe no predicado

O mesmo acontece para as ações.

```
./probes --script 'probeX{print(x);prnt(y);}'

<unknown>
terminate called after throwing an instance of 'std::runtime_error'
  what(): Syntax error in action:
prnt(y)
```

Código 7.9: Exemplo de um erro de sintaxe numa ação

Para além das mensagens de erro básicas, existem mensagens de erro para as agregações. Existe uma mensagem de erro para tentativas de usar funções de agregação diferentes na mesma agregação.

```
./probes --script 'probeX{@ag[x] = count(); @ag["total"] = sum(x);}'

<unknown>
terminate called after throwing an instance of 'std::runtime_error'
  what(): aggregation redefined
  current: @ag = sum()
  previous: @ag = count()
```

Código 7.10: Exemplo de um erro de redefinição de uma agregação

Existe outra mensagem de erro por se tentar definir parâmetros diferentes para a mesma agregação `lquantize`.

```
./probes --script 'probe{@ag["x"] = lquantize(x,0,10,1); @ag["y"] = lquantize(y
,0,100,10);}'
```

```

terminate called after throwing an instance of 'std::runtime_error'
  what():  lquantization parameters redefined
  current: @ag = lquantize(_, 0, 100, 10)
  previous: @ag = lquantize(_, 0, 10, 1)

```

Código 7.11: Exemplo de um erro de redefinição dos parâmetros de uma agregação *lquantize*

Para além da verificação inicial, existem certos tipos de erros que podem ocorrer quando uma sonda dispara. Como o utilizador pode definir sondas e respetivas variáveis, existem variáveis cujo tipo não pode ser deduzido a partir do *script* e que só é determinado na execução da sonda. Ou seja, só nesse momento é que vão ser efetuadas as validações sobre o tipo da variável e o seu estado de inicialização.

Falhas nestas validações podem resultar em dois erros. O primeiro corresponde à tentativa de utilizar uma variável do tipo errado. Como uma variável não inicializada não tem um tipo associado, o seu uso resulta no mesmo erro.

```

./probes --script 'user_probe[] {print(x);}'

terminate called after throwing an instance of 'std::runtime_error'
  what():  Variable with wrong type/uninitialized variable in action:
  print(x)

```

Código 7.12: Exemplo de um erro de variável não inicializada

O segundo erro consiste na tentativa de atribuir um valor de um tipo a uma variável inicializada previamente com outro tipo.

```

./probes --script 'user_probe[] {{x = 3; x = "a";}}'

terminate called after throwing an instance of 'std::runtime_error'
  what():  Variable type and assigned value type don't match:
  x = "a"

```

Código 7.13: Exemplo de um erro de atribuição de um tipo diferente

Apesar de simples as mensagens de erro podem ser de grande utilidade para a maioria dos casos. Em princípio, os predicados e as ações individuais são breves, logo a identificação do predicado ou ação onde ocorre o erro deve ser suficiente. Como as agregações são um elemento mais distinto e que pode causar confusão inicial, foram adicionadas mensagens de erro mais esclarecedoras.

7.3.3 Políticas

O comportamento de disparo pode ser alcançado usando o sistema de políticas do APEX. O disparo de sondas traduz-se em eventos APEX e as cláusulas podem ser registadas como sendo as políticas dos respetivos eventos.

```

apex::register_policy(event_type,
    [predicate, actions](apex_context const& context) -> int{
        get_variables(context);
        if(evaluate(predicate))
            execute(actions);
    });

```

Código 7.14: Pseudocódigo de uma registo de uma cláusula

Ou seja, a política registada deve ter acesso às variáveis da sonda, avaliar o predicado e, caso este seja verdadeiro, executar as ações.

7.3.4 Sondas disponíveis

Sondas **BEGIN** e **END**

A API disponibiliza as sondas **BEGIN** e **END**. A sonda **BEGIN** dispara com a inicialização da API e é útil para tarefas como definir valores iniciais de variáveis. A sonda **END** dispara quando a API é finalizada e pode ser usada para tarefas como imprimir o valor final de variáveis.

As restantes sondas disponíveis são um produto dos eventos suportados pelo APEX.

Sondas definidas pelo utilizador

Os eventos `APEX_CUSTOM_EVENT` permitem replicar a capacidade do DTrace de definir sondas em qualquer ponto da aplicação. Em vez de macros, o disparo da sonda é efetuado através da função `trigger_probe`. Esta recebe como argumentos o nome da sonda e dois mapas. Um mapa representa as variáveis numéricas e o outro variáveis *strings*.

```

HpXTrace::trigger_probe("abc", {{ "a", 5 }}, {{ "s", "Hello world!" }} );

```

Código 7.15: Exemplo de um disparo de uma sonda definida pelo utilizador

As variáveis podem ser utilizadas na cláusula com o mesmo nome.

```

abc[]/a > 0/
{print(s);}

```

Código 7.16: Exemplo de uma cláusula de uma sonda definida pelo utilizador

Sondas de contadores

O evento `APEX_SAMPLE_VALUE` permite o acesso às medições dos contadores HPX. Os contadores são uma fonte rica de informação sobre a execução das aplicações HPX. A sua inclusão na API permite tratar os dados dos contadores através da linha de comandos sem ser preciso desenvolver ferramentas adicionais.

Existem 3 sondas relacionadas com a leitura de contadores HPX:

- `counter::name`
- `counter-type::type`
- `counter-create::name::time`

Como o evento de leitura de contadores apenas ocorre na localidade 0, estas sondas não recebem uma lista de localidades e são registadas exclusivamente na localidade 0.

A sonda **counter** reage às leituras do contador especificado. É preferível utilizar a API do HPX da linha de comandos para definir os contadores pois a leitura de contadores no código não está instrumentada com o APEX. Uma forma de contornar este obstáculo é invocar manualmente no código a função `sample_value` após cada leitura.

Ao utilizar esta sonda é importante ter em conta a expansão dos nomes que acontece na API da linha de comandos. Por exemplo, apesar de parecer correto o seguinte *script* não funciona como desejado.

```
./probes \
--script 'counter::/threads{locality#0/total}/count/cumulative::{print(
    counter_value);}' \
--hpx:print-counter=/threads{locality#0/total}/count/cumulative \
--hpx:print-counter-interval=200
```

Código 7.17: Monitorização das *user threads* executadas

O pedido para `/threads{locality#0/total}/count/cumulative` vai ser expandido para `locality#0/pool#default/worker-thread#*` e para `locality#0/total/total`.

Para permitir mais flexibilidade e para facilitar o uso em situações como a anterior foram definidas sondas derivadas desta.

A sonda **counter-type** é semelhante mas em vez de ser indicado o nome completo é apenas indicado o tipo do contador. Assim no exemplo anterior do contador dos fios de execução basta fornecer o tipo `/threads/count/cumulative` para lidar com a expansão de contadores. Para além disso, se o tipo fornecido for parcial então a sonda dispara para todos os subtipos. Ou seja, uma sonda com o tipo `threads/count/` dispara para todos os contadores dos tipos `/threads/count/cumulative` e `/threads/count/cumulative-phases`.

Por fim, existe a sonda **counter-create**. Tal como o nome indica, esta sonda trata de criar o contador explicitado. Desta forma pode-se evitar usar a API da linha de comandos, o que tem vantagens como não imprimir para o terminal todas as leituras.

Todas as variantes têm as mesmas variáveis de cláusula. O valor da leitura do contador é exposto através da variável `counter_value`. Como as variáveis podem ser usadas não só nas ações mas também no predicado, é possível controlar o disparo da sonda de acordo com o valor do contador.

O nome do contador lido é dado por `counter_name` e também estão disponíveis variáveis com os componentes do nome:

- `counter_type`
- `counter_parameters`
- `counter_parent_instance_name`
- `counter_parent_instance_index`
- `counter_instance_name`
- `counter_instance_index`

Estas variáveis são todas do tipo *string*, incluindo os índices. Apesar de estes últimos serem números, em princípio devem ser usados para organizar informação e não em operações aritméticas.

O uso destas variáveis nos predicados das cláusulas permite definir com mais detalhe qualquer padrão de disparo que não seja satisfatoriamente representado pelas três variações das sondas dos contadores.

Sonda *proc*

Para além de leituras de contadores, o `APEX_SAMPLE_VALUE` também ocorre com leituras da diretoria Linux `/proc`. Para aceder a estas métricas, existe a sonda **proc::type**. As leituras disponíveis por padrão são:

- CPU User %
- CPU Nice %
- CPU System %
- CPU Idle %
- CPU I/O Wait %
- CPU IRQ %
- CPU soft IRQ %
- CPU Steal %
- CPU Guest %
- Package-0 Energy
- DRAM Energy

Mais leituras podem ser acedidas alterando a configuração do APEX. A forma mais simples de o fazer é através das variáveis de ambiente. Por exemplo, a variável de ambiente `APEX_PROC_SELF_IO` permite ativar as leituras de `/proc/self/io`. A lista completa de variáveis de ambiente é:

- `APEX_PROC_STAT`
- `APEX_PROC_CPUINFO`

- APEX_PROC_MEMINFO
- APEX_PROC_NET_DEV
- APEX_PROC_SELF_STATUS
- APEX_PROC_SELF_IO
- APEX_PROC_STAT_DETAILS

O argumento da sonda indica as medições que devem ser processadas. Para facilidade de uso, a sonda é disparada se o argumento estiver incluído no nome da medição. Ou seja, o argumento "User" corresponde à medição CPU User%, o argumento "CPU" corresponde a todas as medições com esse prefixo e a omissão do argumento indica que todas as medições devem ser processadas.

As variáveis desta sonda são `proc_name` e `proc_value` e contêm, respetivamente, o nome e o valor da medição.

Uma forma de verificar que métricas estão disponíveis é utilizar a sonda sem definir o argumento e imprimir `proc_name`.

Sonda *task*

A ideia desta sonda é recorrer aos mesmos eventos que o APEX utiliza para construir o perfil final de execução com o número de invocações e com os tempos de execução das tarefas. A sonda dispara de cada vez que ocorre uma alteração no estado de execução de uma tarefa e fornece vários dados sobre a mesma. As transições possíveis são indicadas pelos eventos usados pelo APEX: APEX_START_EVENT, APEX_STOP_EVENT, APEX_RESUME_EVENT e APEX_YIELD_EVENT.

A sonda recebe dois argumentos. O primeiro, que é obrigatório, indica os eventos que devem resultar num disparo. É uma lista cujos valores podem ser `start`, `stop`, `resume` ou `yield`. O segundo argumento, que é opcional, filtra os disparos conforme o nome atribuído à tarefa. Normalmente este é o nome da ação ou função a ser executada. A ausência deste argumento indica que todos os nomes são aceites. A utilização recomendada desta sonda consiste numa execução inicial para descobrir os nomes de todas as tarefas, seguida de uma execução em que apenas um conjunto específico de tarefas são observadas. Desta forma o impacto no desempenho é minimizado, o que resulta numa observação mais correta.

Passando às variáveis da sonda, as essenciais são as que identificam a tarefa. Para além da variável `name` com o nome da tarefa, também existe a variável `guid`, que corresponde ao identificador interno usado pelo APEX (e que é particularmente útil em casos com recursividade). Também existe o `parent_guid`, que identifica a tarefa que deu origem à atual. Para sondas com vários eventos, a variável `event_type` também é importante e, tal como o primeiro argumento, pode assumir os valores `start`, `stop`, `resume` e `yield`. De resto, existem algumas variáveis com informação sobre a execução, apenas para os eventos `stop` e `yield`:

- `start` - início da execução (em nanosegundos);
- `end` - fim da execução (em nanosegundos);

- `allocations` - número de invocações de funções de alocação e realocação de memória;
- `frees` - número de invocações de funções de libertação e realocação de memória;
- `bytes_allocated` - total de *bytes* alocados na tarefa;
- `bytes_freed` - total de *bytes* libertados na tarefa.

Sonda *message*

Por fim, existe a sonda **message**, que monitoriza o envio de parcelas entre localidades. O disparo desta sonda corresponde à ocorrência dos eventos `APEX_SEND` e `APEX_RECV`.

A sonda recebe como argumento obrigatório uma lista com os eventos a que deve reagir (`send` para `APEX_SEND` e `receive` para `APEX_RECV`). Uma lista vazia indica que a sonda deve ser registada para ambos os eventos.

Para além da variável de sonda `event`, que indica o tipo do evento, existem as seguintes variáveis de sonda com informação sobre a mensagem:

- `tag` - identificador da mensagem;
- `size` - tamanho da mensagem (em *bytes*);
- `source_rank` - a localidade de origem;
- `source_thread` - o fio de execução que originou a mensagem;
- `target` - a localidade de destino.

7.3.5 Funções da API

Para integrar o uso do `HpxTrace` numa aplicação, existe um pequeno conjunto de funções.

A função `HpxTrace::register_command_line_options`, que deve ser invocada na função `main`, recebe um objeto da classe `options_description` e adiciona as opções da linha de comandos. Adiciona a opção `--script` para a leitura do *script* a partir da linha de comandos e a opção `--file` para a leitura do *script* a partir de um ficheiro de texto.

```
int main(int argc, char* argv[])
{
    using namespace hpx::program_options;
    options_description desc_commandline;
    HpxTrace::register_command_line_options(desc_commandline);

    // Initialize and run HPX
    hpx::init_params init_args;
    init_args.desc_cmdline = desc_commandline;
```



```

    return hpx::init(argc, argv, init_args);
}

```

Código 7.18: Adição das opções da linha de comandos a uma aplicação HPX

A invocação de `hpx::init` vai resultar na invocação de `hpx_main`. Deve ser usada a assinatura que recebe as opções do programa.

```

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

```

Código 7.19: Assinaturas da função `hpx_main`

Quando o utilizador quiser iniciar o `HpxTrace`, deve invocar a função `HpxTrace::init` com as opções da linha de comandos. Esta função valida o *script*, dispara as sondas **BEGIN** e regista as restantes.

O uso das opções do programa é opcional. Se o utilizador pretender, por exemplo, usar outra assinatura do `hpx_main`, pode invocar a função `HpxTrace::init` com o *script* numa *string*.

Como referido na secção 7.3.4, é usada a função `HpxTrace::trigger_probe` para disparar uma sonda definida pelo utilizador.

Para finalizar o uso da API existe a função `HpxTrace::finalize`. Esta trata de cancelar o registo das políticas do APEX em todas as localidades e de disparar as sondas **END**.

Ao permitir o controlo sobre quando a API é inicializada e terminada, o utilizador pode escolher apenas uma secção da execução de uma aplicação para ser monitorizada.

7.4 IMPLEMENTAÇÃO DA API

Esta secção detalha a implementação da ferramenta desenvolvida, incluindo o uso de componentes, de processadores de linguagem e do sistema de políticas do APEX.

7.4.1 Arquitetura e Armazenamento dos dados

Em primeiro lugar, os valores numéricos são representados apenas através do tipo C++ `double` de forma a simplificar a implementação inicial do interpretador da linguagem.

A API desenvolvida recorre ao uso de componentes HPX para armazenar e distribuir os dados pelas várias localidades. Existem 4 tipos de componentes:

- `ScalarVarsServer` - responsável pelo armazenamento das variáveis escalares;
- `MapVarsServer` - responsável pelo armazenamento dos dicionários;

- `AggregationsServer` - responsável pelo armazenamento das variáveis de agregação;
- `MutexesServer` - responsável pelo mecanismo de sincronização `lock()` e `unlock()`.

As ações expostas pelos componentes serão depois utilizadas pelo interpretador da linguagem.

A quantidade e a colocação das instâncias servidor dos componentes são regidas pela dinâmica local vs global presente na linguagem.

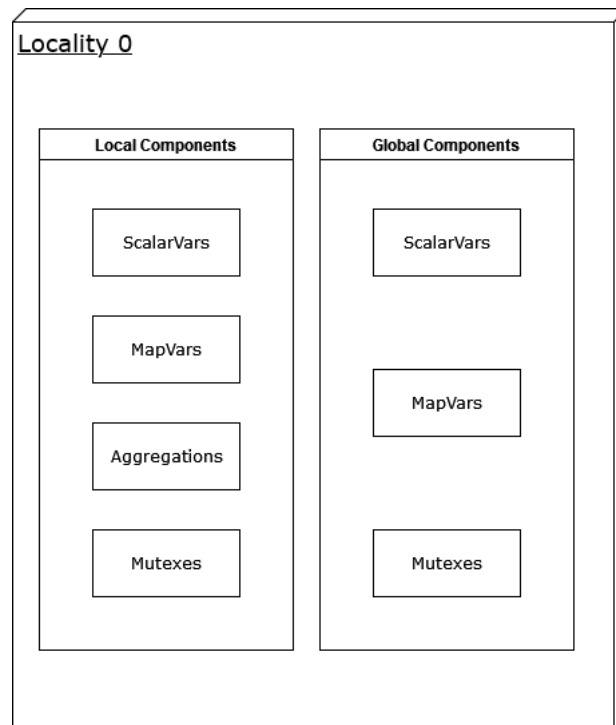


Figura 7: Diagrama dos componentes na localidade 0

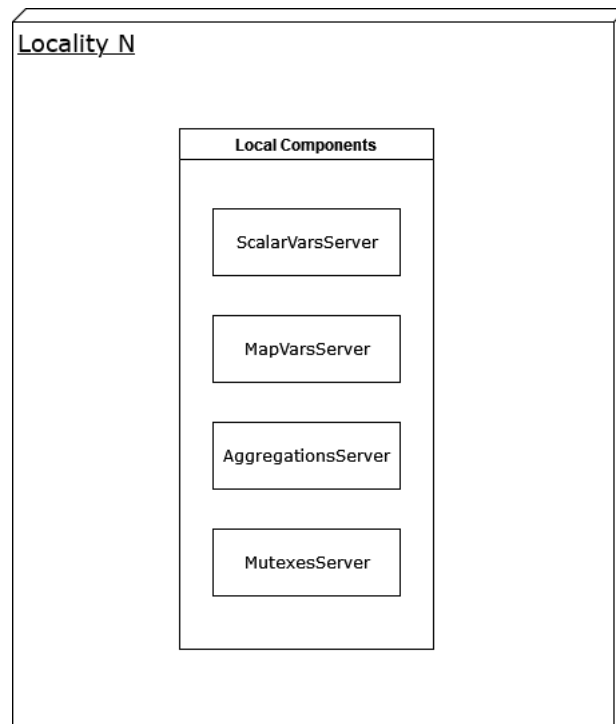


Figura 8: Diagrama dos componentes nas restantes localidades

Para lidar com variáveis e sincronismos locais, cada localidade aloja um servidor de cada componente. Desta forma são evitados custos de comunicação entre localidades. As instâncias responsáveis pelo lado global existem na localidade 0, dado que é a localidade que em média deverá ter mais disparos de sondas (por exemplo, todas as leituras de contadores). O tipo `Aggregations` não tem um servidor global pois as agregações globais são calculadas quando necessário.

Componente `ScalarVarsServer`

O componente `ScalarVarsServer` contém as variáveis escalares de ambos os tipos e as respetivas ações.

```

bool store_double(std::string name, double d)
bool store_string(std::string name, std::string s)
hpx::util::optional<double> get_double(std::string name)
hpx::util::optional<std::string> get_string(std::string name)

```

Código 7.20: Ações fornecidas por `ScalarVarsServer`

Nas ações que guardam variáveis, a operação só é aceite se não existir nenhuma variável de outro tipo com o nome fornecido. O booleano retornado indica se o valor foi adicionado com sucesso. As ações de obtenção de valores retornam um `hpx::util::optional` (equivalente ao `std::optional`, é um contentor que

pode ou não conter um valor) de forma a poderem representar a inexistência da variável requerida. A existência desta informação adicional permitirá ao processador do *script* detetar a ocorrência de erros.

Inicialmente, era utilizado o componente *template Map* (definido na secção 3.3), sendo que cada tipo de variável tinha a sua própria instância. Desta forma, o custo de implementação era minimizado e a adição de novos tipos era facilitada. No entanto, tarefas como evitar a existência de variáveis com o mesmo nome e tipos diferentes passariam a ser responsabilidade da implementação do interpretador da linguagem e implicariam comunicações com os componentes de cada tipo. Ao centralizar os diferentes tipos num só componente, a gestão destes aspetos é simplificada.

O componente *ScalarVarsServer* utiliza internamente a classe *ScalarVars* para tratar da gestão das variáveis escalares. Isto tem a vantagem dessa classe poder ser utilizada independentemente, o que é o caso na implementação das variáveis de sonda.

O componente é derivado de `hpx::components::locking_hook`, o que garante a serialização dos acessos.

Componente MapVarsServer

O componente *MapVarsServer* é muito semelhante ao *ScalarVarsServer*, com as adaptações necessárias para lidar com dicionários de dicionários. O componente também é derivado da classe `hpx::components::locking_hook` e utiliza a classe *MapVars* para implementar as funcionalidades, sendo que esta também pode ser usada separadamente.

```
bool store_double(std::string name, VariantList keys, double d)
bool store_string(std::string name, VariantList keys, std::string s)
hpx::util::optional<double> get_double(VariantList keys, std::string name)
hpx::util::optional<std::string> get_string(VariantList keys, std::string name)
```

Código 7.21: Ações fornecidas por *MapVarsServer*

O único detalhe importante é que a ausência de um valor encapsulado no `optional` apenas indica a ausência da variável dicionário. Se a chave fornecida não existir é retornado um valor padrão.

Componente MutexesServer

O componente *MutexesServer* consiste num dicionário com objetos `std::mutex` em que as chaves correspondem à lista fornecida como argumento às funções `lock` e `unlock`. Este componente também herda da classe `hpx::components::locking_hook`, de forma a que a criação de novos *mutexes* seja *thread-safe*.

Componente `AggregationsServer`

Tal como os restantes componentes, o componente `AggregationsServer` utiliza um dicionário para guardar as variáveis de agregação. O conceito de agregação é representado pela classe `Aggregations`. Os tipos de agregação são divididos em subclasses de acordo com os dados auxiliares.

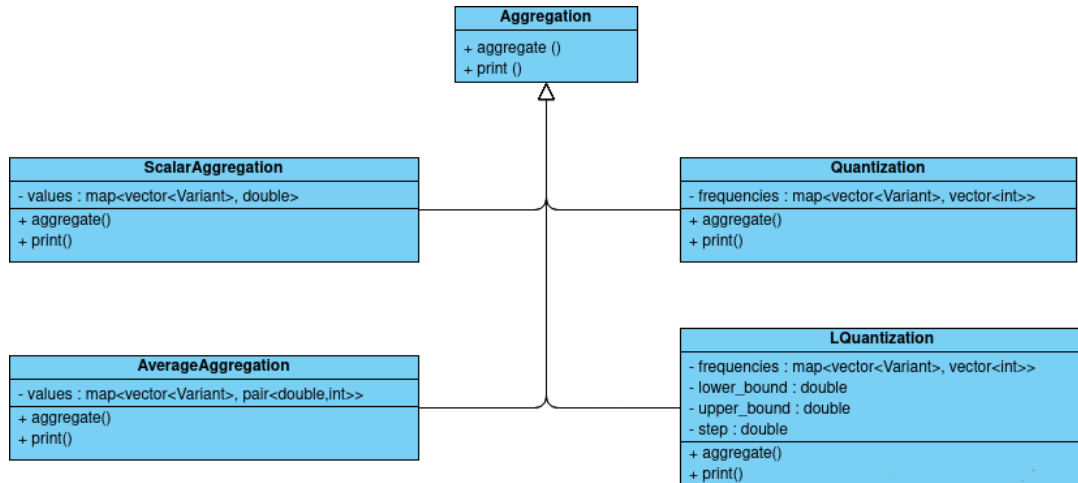


Figura 9: Diagrama das classes de agregação

Todas as classes contêm um dicionário em que as chaves são os índices das agregações, representados por vetores de `Variants`. As agregações *count*, *sum*, *max* e *min*, representadas por `ScalarAggregation`, apenas precisam de armazenar o valor de agregação. A agregação *avg* precisa de manter o número de elementos agregados para poder calcular a média.

Os valores das agregações *quantize* e *lquantize* são ambos distribuições de frequências. O vector em `Quantization` tem um tamanho fixo de 32 elementos. Como é uma distribuição de base 2, os limites são 0 e 4294967296, suficientes para a maioria das situações. Por sua vez, o tamanho do vector em `LQuantization` é determinado pelos parâmetros definidos pelo utilizador.

No arranque da API o *script* é processado para verificação de erros. Em particular, é necessário validar a definição dos parâmetros das agregações. Esta oportunidade é aproveitada para inicializar as instâncias das agregações. A validação do *script* apenas ocorre na localidade 0, porém as agregações são inicializadas simultaneamente em todas as localidades. Desta forma, em troca de um custo inicial mínimo, não é preciso verificar se a variável de agregação está inicializada antes de agregar e o controlo de concorrência pode ser otimizado.

Como as agregações são todas previamente inicializadas, a única preocupação em termos de concorrência são tentativas simultâneas de agregar um valor na mesma variável de agregação. A adição de um `scoped_lock` (*mutex* com comportamento RAII) no método de agregação resolve este obstáculo.

As ações expostas pelo componente permitem registar novas variáveis de agregação, agregar novos valores e imprimir agregações.

```
std::string new_aggregation(std::string func, std::string name)
LquantizeResult new_lquantize(
    std::string name, int lower_bound, int upper_bound, int step
)
bool aggregate(std::string name, VariantList keys, double value)
void print_aggregation(hpx::naming::id_type id, std::string name)
```

Código 7.22: Ações fornecidas por AggregationsServer

Uma tentativa de registar uma variável de agregação que já exista resulta na devolução dos parâmetros da agregação existente. Desta forma o processador do *script* poderá comparar o resultado da ação com os novos parâmetros fornecidos para verificar se deve gerar um erro.

Outras ações expostas retornam os dados de variáveis de agregação. O seu uso permite agregar agregações. Este processo é encapsulado em métodos estáticos da classe AggregationsServer.

```
partial_print_aggregation(std::vector<hpx::naming::id_type> ids,
    std::vector<int> loc_indexes, std::string name
)
void global_print_aggregation(
    std::vector<hpx::naming::id_type> ids, std::string name
)
```

Código 7.23: Funções de agregação e impressão de agregações

7.4.2 Interpretadores

O ideal para minimizar impactos no desempenho das aplicações monitorizadas seria desenvolver um compilador para a linguagem estabelecida. No entanto, isso seria demasiado complexo. Como alternativa foram implementados interpretadores através da biblioteca Boost.Spirit.Qi. Esta biblioteca permite implementar interpretadores *recursive-descent* de uma forma acessível, motivo pela qual foi selecionada. A desvantagem desta abordagem é que implica o processamento repetido das ações de cada vez que devem ser executadas.

A biblioteca funciona com base na composição de interpretadores. Existem interpretadores já definidos que tratam de identificar números, caracteres específicos e outros padrões básicos.

```
double_
char_("a-zA-Z")
'+'
```

Código 7.24: Interpretadores de números, carateres alfabéticos e de apenas o carácter +

Os interpretadores podem ser compostos através do operador `>>`, que significa "seguido de". Se um interpretador falhar, os seguintes não são avaliados.

```
double_ >> '+' >> double_
```

Código 7.25: Interpretador de uma soma

Outros operadores de interpretadores vitais são o operador de diferença `-` e o operador de alternativa `|`.

```
char_ - ':'  
'+' | '-'
```

Código 7.26: Interpretador de todos os caracteres exceto ':' e interpretador dos símbolos '+' e '-'

Tal como nas expressões regulares, estão disponíveis operadores que lidam com a repetição de padrões. O operador `*` significa que o padrão pode ocorrer 0 ou mais vezes, o operador `+` significa que o padrão pode ocorrer 1 ou mais vezes e o operador `?` significa que o padrão pode ocorrer 0 ou 1 vez.

```
double_ >> *('+' >> double_)
```

Código 7.27: Interpretador de um número ou uma sequência de somas

As composições dos interpretadores podem ser organizadas em regras:

```
boost::spirit::qi::rule<Iterator> addition;  
addition = double_ >> *('+' >> double_);
```

Código 7.28: Regra de somas

Obrigatoriamente, as regras necessitam de ser parametrizadas com o iterador do tipo a ser processado. Além disso, podem ser parametrizadas com um interpretador *skipper*. O interpretador *skipper* é responsável por identificar os caracteres entre os elementos da regra. Normalmente isto significa usar o `ascii::space_type`, um interpretador que identifica espaços.

Apenas reconhecer padrões não é suficiente para realizar os objetivos pretendidos, sendo também necessário manipular a informação obtida. Para isto a biblioteca oferece dois conceitos: atributos e ações semânticas.

Cada interpretador pode ter o seu atributo. Este corresponde a um tipo C++ representativo da informação processada. Os atributos de interpretadores elementares são evidentes. O atributo do interpretador `double_` é um `double` e os interpretadores literais (ex: `'x'`) não necessitam de ter atributos. Dado que as regras correspondem a interpretadores, então também podem ter os seus atributos. O valor do atributo de uma regra pode ser inferido automaticamente pela biblioteca ou pode ter que ser determinado explicitamente, dependendo da complexidade. No último caso, os atributos são definidos adicionando mais um parâmetro *template* à declaração da regra. Este aspeto será mencionado em mais detalhe conforme necessário.

Ações semânticas são funções ou objetos função que podem ser associados aos elementos da gramática através dos operadores `[]`. Quando um elemento é detetado, a ação é invocada e recebe como argumento o atributo do interpretador associado.

O exemplo seguinte utiliza ações semânticas para somar o valor de todos os elementos de uma expressão do tipo " $x_1 + x_2 \dots + x_n$ " na variável n .

```
double sum = 0;
void add (double const& n)
{
    sum += n;
}
double_[&add] >> *('+' >> double_[&add])
```

Código 7.29: Cálculo de somas com uma função auxiliar

A inclusão de outra biblioteca da coleção Boost, a biblioteca Phoenix, adiciona muita flexibilidade ao processo de definição de ações semânticas. De forma semelhante ao uso de expressões *lambda*, permite definir ações básicas sem ser necessário definir funções separadas.

A funcionalidade anterior pode ser obtida da seguinte forma:

```
double sum = 0;

double_[boost::phoenix::ref(sum) += _1] >>
    *('+' >>
        double_[boost::phoenix::ref(sum) += _1]
    )
```

Código 7.30: Cálculo de somas com uma expressão Phoenix

O símbolo `_1` representa o atributo do interpretador e `boost::phoenix::ref` indica que `sum` é uma referência de uma variável mutável.

A biblioteca Phoenix também é útil para definir ações para grupos de interpretadores. Os atributos de cada interpretador do grupo são representados pelos símbolos `_1` - `_N` e o atributo da regra é representado por `_val`. O exemplo seguinte demonstra o uso deste mecanismo para conseguir a comparação de dois números de uma forma concisa.

```
qi::rule<Iterator, ascii::space_type, bool> equals;

equals = (double_ >> "==" >> double_) [_val = _1 == _2]
```

Código 7.31: Comparação de números com ações semânticas

O atributo do terceiro interpretador é representado por `_2` pois como o interpretador literal `"=="` não tem atributo, não é considerado.

Outra funcionalidade do Phoenix é o *binding* de funções. Este introduz possibilidades como utilizar funções membro de classes ou transmitir múltiplos atributos a uma única função.


```
rule = (double_ >> '%' >> double_) [_val = boost::phoenix::bind(&f, _1, _2)]
```

Código 7.32: *Binding* da função f ao padrão $x \% y$

Neste exemplo, a ocorrência do padrão $x \% y$ resulta na invocação da função f com os argumentos x e y .

Outra funcionalidade importante das ações semânticas é a possibilidade de invalidar a regra, atribuindo um booleano ao parâmetro `_pass`. Isto permite situações como distinguir entre números pares e ímpares.

```
even = int_[_pass = _1 % 2 == 0];
odd  = int_[_pass = _1 % 2 == 1];
```

Código 7.33: Verificação da paridade de um número com ações semânticas

Com os interpretadores definidos, resta invocá-los. Isto pode ser alcançado com a função `phrase_parse`, entre outras. A esta função devem ser fornecidos os iteradores do início e do fim do texto a processar, o interpretador principal (definido diretamente ou através de uma regra) e um interpretador *skipper*. O iterador inicial vai avançando conforme os elementos que compõem o interpretador vão sendo detetados. Desta forma é possível verificar se o processamento teve sucesso ou em que ponto falhou.

7.4.3 Implementação da linguagem

Com o estabelecimento das bases do funcionamento da biblioteca Spirit, é finalmente possível proceder à definição da linguagem escolhida.

O processador da linguagem tem acesso aos identificadores dos componentes da sua localidade, aos dos globais e a uma lista de todos os componentes *Aggregations*, de forma a permitir juntar agregações. Os identificadores estão contidos na classe `ScriptData` que é usada extensivamente nas ações semânticas das regras.

O uso de atributos é vital. Como tal, são definidos tipos de regras para os tipos de atributos necessários.

```
typedef qi::rule<Iterator, ascii::space_type> Rule;
typedef qi::rule<Iterator, ascii::space_type, std::string()> RuleSt;
typedef qi::rule<Iterator, ascii::space_type, double> RuleN;
typedef qi::rule<Iterator, ascii::space_type, bool> RuleBool;

typedef boost::variant<double, std::string> Variant;
typedef qi::rule<Iterator, ascii::space_type, Variant()> RuleVariant;

typedef std::vector<Variant> VariantList;
qi::rule<Iterator, ascii::space_type, VariantList()> RuleVariantList;
```

Código 7.34: Definição dos tipos das regras usadas

Para suportar regras cujos atributos podem ter qualquer tipo de valor é usado o tipo `boost::variant`, que permite manipular dados heterogêneos de uma forma uniforme. Uma lista de valores de qualquer tipo corresponde ao tipo `VariantList`.

A tarefa inicial consiste na detecção dos elementos mais básicos da linguagem: nomes de variáveis, números e *strings*. Para números basta utilizar o interpretador já existente `double_`. O interpretador das *strings* deve identificar sequências de caracteres delimitadas por aspas.

```
qi::rule<Iterator, std::string()> string_content = +(char_ - '"');
RuleSt string = qi::lexeme['"' >> string_content >> '"'][_val = _1];
```

Código 7.35: Regras de detecção de *strings*

A primeira regra trata de capturar uma sequência sem a presença de ". Dado que os espaços devem ser incluídos, é omitido o interpretador *skip* na parametrização da regra. Todos os caracteres encontrados são automaticamente sintetizados no atributo da regra.

Na segunda regra é utilizada a diretiva `lexeme` que indica que os espaços naquele segmento não devem ser ignorados. A sua presença é necessária para que a primeira regra possa funcionar corretamente.

Passando às variáveis, os nomes destas podem conter letras, dígitos e *underscores*. Para distinguir entre variáveis de sonda, locais e globais, é verificada a presença dos prefixos `&`, `:` e `#`.

```
RuleSt probe_var = (char_("&")) >> *char_("a-zA-Z0-9_");
RuleSt local_var = (char_(":")) >> *char_("a-zA-Z0-9_");
RuleSt global_var = (char_("#")) >> *char_("a-zA-Z0-9_");
```

Código 7.36: Regras do contexto das variáveis

Usar diretamente estas regras para definir operações e funções implicaria verificações constantes dos tipos das variáveis. Para simplificar definições futuras, é vantajoso adicionar regras que traduzam o nome de uma variável no respetivo valor. Isto implica determinar o tipo da variável e colocar o valor no atributo da regra. Ambas tarefas cabem às ações semânticas pois não existe nenhum elemento sintático que indique o tipo das variáveis.

Utilizando as variáveis locais do tipo *string* como exemplo, para comunicar com o componente das variáveis escalares (neste caso o local), a função `get_comp_stvar` invoca a ação `get_string_action` e verifica o conteúdo do `optional` recebido. O valor de retorno indica a existência da variável e caso exista, o valor da variável é atribuído à referência recebida.

```
bool get_comp_stvar(hpx::naming::id_type id, std::string name, std::string& value)
{
    hpx::util::optional op = ScalarVarsServer::get_string_action()(id, name);
    if(!op.has_value()) return false;
    value = op.value();
    return true;
}
```

Código 7.37: Obtenção do valor de uma variável *string*

A função é depois usada na ação semântica da regra para variáveis locais *string*. O booleano que indica a existência da variável é atribuído ao parâmetro `_pass`. Desta forma, se não existir uma variável *string*, a regra é invalidada. Para armazenar o valor da variável é passado o parâmetro que representa o atributo da regra.

```
RuleSt local_stvar = local_var
[_pass = phx::bind(&get_comp_stvar, phx::ref(data.local_scalar_vars), _1, _val)];
```

Código 7.38: Regra para variáveis *string* locais

O mesmo acontece para as restantes variáveis escalares. A única diferença é que no caso das variáveis da sonda, é invocado o método de `ScalarVars` em vez da ação do componente.

As várias regras das variáveis escalares são agrupadas em duas regras, uma para cada tipo.

```
RuleSt string_var = probe_stvar | local_stvar | global_stvar;
RuleN double_var = probe_dvar | local_dvar | global_dvar;
```

Código 7.39: Regra de variáveis *string* e regra de variáveis numéricas

As regras que definem os dicionários seguem a mesma lógica das variáveis escalares. A única diferença é que devem detetar e capturar as chaves do dicionário.

```
RuleSt local_stmap = (local_var >> '[' >> keys >> ']')
[_pass = phx::bind(&get_comp_stmap, phx::ref(data.local_map_vars), _1,_2, _val)];
```

Código 7.40: Regra de dicionário de *strings* local

As chaves podem ser valores de qualquer tipo.

```
RuleVariant value;
RuleVariantList keys = value % ',';
```

Código 7.41: Declaração de um valor de qualquer tipo e definição de uma lista de chaves

A regra `value` só será definida após as regras das quais depende estarem definidas.

Tal como acontece com as variáveis escalares, os dicionários de sonda, locais e globais são agrupados em `double_map` e `string_map`.

O passo seguinte consiste na definição de expressões, aritméticas ou com *strings*. Como os elementos das expressões podem ser variáveis escalares, dicionários ou valores literais, é útil agrupá-los numa só regra.

```
RuleSt string_value = string | string_map | string_var;
RuleN double_value = double_ | double_map | double_var;
```

Código 7.42: Consolidação de valores *string* e de valores numéricos

As expressões com *strings* apenas suportam a operação de concatenação.

```
RuleSt string_expression =
  string_value          [_val = _1]
  >> *('+' >> string_value [_val = _val + _1]);
```

Código 7.43: Regra para expressões *string*

As expressões aritméticas suportam uma gama maior de operações, o que é refletido na complexidade acrescida das regras. É preciso suportar a aplicação de cinco operações aritméticas básicas, dos parênteses e da negação, tendo em consideração as respectivas prioridades.

```
RuleN arithmetic_expression, term, factor;

arithmetic_expression =
  term          [_val = _1]
  >> *( ('+' >> term [_val = _val + _1])
        | ('-' >> term [_val = _val - _1])
        );

term =
  factor          [_val = _1]
  >> *( ('*' >> factor [_val = _val * _1])
        | ('/' >> factor [_val = _val / _1])
        | ('%' >> factor [_val = boost::phoenix::bind(&fmod, _val, _1)])
        );

factor =
  double_value          [_val = _1]
  | '(' >> arithmetic_expression [_val = _1] >> ')'
  | ('-' >> factor          [_val = -_1])
  | ('+' >> factor          [_val = _1]);
```

Código 7.44: Regra para expressões aritméticas

A prioridade das operações é expressa através das dependências entre as regras. O uso de parênteses é conseguido através de recursividade.

Com estas definições elementares, a capacidade de atribuição de valores a variáveis pode ser descrita. A atribuição de valores pode falhar caso já exista uma variável com o mesmo nome e tipo diferente. De forma semelhante à obtenção do valor de uma variável, é usada uma função para comunicar com os componentes que retorna um booleano que indica o sucesso da operação. Ao invés do booleano ser usado para determinar a validade da regra, é armazenado no atributo da regra.

```
RuleBool local_var_assignment =
  (local_var >> '=' >> arithmetic_expression)
  [_val = phx::bind(&store_comp_dvar, phx::ref(data.local_scalar_vars), _1, _2)]
  | (local_var >> '=' >> string_expression)
```

```
[_val = phx::bind(&store_comp_stvar, phx::ref(data.local_scalar_vars), _1,_2)];
```

Código 7.45: Atribuição de um valor a variável escalar local

As regras de atribuições de variáveis escalares são agrupadas numa só. É nessa regra que o booleano é usado para determinar a validade da regra. Caso a regra falhe, como não existe mais nenhuma regra que siga a estrutura da atribuição escalar (`scalar_var = value`), o processamento do *script* vai falhar. Uma *flag* é sinalizada para posteriormente ser possível verificar a origem da falha.

```
bool scalar_assignment_error = false;

Rule scalar_var_assignment =
    (probe_var_assignment | local_var_assignment | global_var_assignment)
    [_pass = _1, phx::ref(scalar_assignment_error) = !_1];
```

Código 7.46: Registo do sucesso da atribuição para variáveis escalares

Mais uma vez o processo para variáveis dicionário é muito semelhante e não justifica a sua exposição. Como tem sido prática, ambos os tipos de atribuição são agrupados.

```
Rule assignment = scalar_var_assignment | map_var_assignment;
```

Código 7.47: Aglomeração das regras de atribuição

Passando às funções, uma função é formada pelo seu nome e por um conjunto de argumentos (que pode ser vazio) entre parênteses. Graças à função `bind`, a funcionalidade pretendida é implementada numa função C++ separada e o valor retornado é armazenado no atributo da regra.

```
RuleSt str = ("str(" >> arithmetic_expression >> ')')
    [_val = boost::phoenix::bind(&to_string, _1)];

RuleN dbl = ("dbl(" >> string_expression >> ')')
    [_val = boost::phoenix::bind(&to_double, _1)];
```

Código 7.48: Regras das funções de conversão de tipo

Para que as funções possam ser usadas nas expressões, basta adicioná-las às definições de valores numéricos e *string*.

```
RuleSt string_func = str | locality_name;
RuleN double_func = dbl | nl | locality_rank | round | ceil | floor | timestamp;

RuleSt string_value = string | string_func | string_map | string_var;
RuleN double_value = double_ | double_func | double_map | double_var;
```

Código 7.49: Redefinição de `string_value` e `double_value`

Caso mais funções sejam necessárias no futuro, basta adicionar as regras à regra `string_func` ou à regra `double_func`.

Finalmente é possível definir a regra `value` declarada previamente.

```
//RuleVariant value
value = arithmetic_expression[_val = _1]
      | string_expression[_val = _1];
```

Código 7.50: Aglomeração de todos os valores numa só regra

Passando às agregações, a maioria segue o mesmo padrão. É preciso identificar o nome da variável, as chaves, a função de agregação e os argumentos. A função de agregação `count` não tem argumentos, a função `lquantize` recebe uma expressão numérica e 3 literais `double` e as restantes funções recebem uma expressão numérica. Em todos os casos a função `aggregate` é invocada com os argumentos colecionados.

```
Rule agg_funcs = lit("sum") | "avg" | "min" | "max" | "quantize";

Rule aggregation =
  ('@' >> var >> '[' >> keys >> ']' >> "=" >> "count()")
  [phx::bind(&aggregate, phx::ref(data.local_aggregation), _1, _2, 0)]
  |
  ('@' >> var >> '[' >> keys >> ']' >> "=" >>
    agg_funcs >> '(' >> arithmetic_expression >> ')')
  [phx::bind(&aggregate, phx::ref(data.local_aggregation), _1, _2, _3)]
  |
  ('@' >> var >> '[' >> keys >> ']' >> "=" >> "lquantize" >> '(' >>
    arithmetic_expression >> ',' >>
    double_ >> ',' >> double_ >> ',' >> double_ >>
    ')')
  [phx::bind(&aggregate, phx::ref(data.local_aggregation), _1, _2, _3)];
```

Código 7.51: Regra das agregações

A presença de `lit("sum")` na regra `agg_funcs` é para explicitar que é um interpretador e não uma string. Caso contrário o compilador assume que o símbolo `|` representa a operação *bitwise OR*.

Com todos os tipos de dados presentes, é possível definir as funções que permitem imprimir os dados no terminal. A função `print` pode receber um valor numérico ou um valor *string*, o que resulta na impressão do objeto `Variant`, ou uma agregação, o que leva à invocação da ação do componente `Aggregations` local.

```
Rule print = ("print(" >> value >> ')')[phx::bind(&print_value, _1)]
  | (lit("print(" >> '@' >> var >> ')')
  [phx::bind(&print_aggregation, phx::ref(data.local_aggregation), _1)]
  | (lit("global_print(" >> '@' >> var >> ')')
  [phx::bind(&global_print_aggregation, phx::ref(data.aggregations), _1)]
  | (lit("global_print(" >> '@' >> var >> ',' >> localities >> ')')
  [phx::bind(&global_print_aggregation, phx::ref(data.aggregations), _1)]
```

```
[phx::bind(&partial_print_aggregation, phx::ref(data.aggregations), _2, _1)];
```

Código 7.52: Regra de impressão de variáveis e agregações

Para a função `global_print` é preciso verificar a presença dos índices das localidades.

```
qi::rule<Iterator, ascii::space_type, vector<int>> localities = int_ % ',';
```

Código 7.53: Regra dos índices das localidades

Por fim, são definidas as funções de controlo de concorrência que processam a lista de chaves e invocam as ações dos componentes `Mutexes`.

```
Rule global_lock = (lit("global_lock") >> '(' >> keys >> ')')
[phx::bind(&comp_lock, phx::ref(data.global_mutexes), _1)];
```

Código 7.54: Regra do `global_lock()`

Como nos casos anteriores, estas regras também são agrupadas, desta vez na regra `locks`.

Para completar a linguagem das ações falta descrever uma sequência das ações. As ações podem ser atribuições, impressões para o terminal, agregações ou controlo de concorrência, separadas por `;`.

```
Rule action = assignment | print | aggregation | locks;
Rule actions = action >> ';' >> *(action >> ';');
```

Código 7.55: Definição de ação e sequência de ações

Com as ações definidas, falta implementar os predicados. Muitas das regras são utilizadas novamente, sendo apenas necessário definir os aspetos de comparação e de lógica booleana.

A primeira adição é a da comparação. É permitida a comparação entre *strings* e entre números

```
RuleBool comparison =
  (string_expression >> "==" >> string_expression) [_val = _1 == _2]
| (string_expression >> "!=" >> string_expression) [_val = _1 != _2]
| (string_expression >> "in" >> string_expression) [_val = phx::bind(&contains,
  _1, _2)]
| (arithmetic_expression >> "==" >> arithmetic_expression) [_val = _1 == _2]
| (arithmetic_expression >> "!=" >> arithmetic_expression) [_val = _1 != _2]
| (arithmetic_expression >> "<" >> arithmetic_expression) [_val = _1 < _2]
| (arithmetic_expression >> "<=" >> arithmetic_expression) [_val = _1 <= _2]
| (arithmetic_expression >> ">" >> arithmetic_expression) [_val = _1 > _2]
| (arithmetic_expression >> ">=" >> arithmetic_expression) [_val = _1 >= _2];
```

Código 7.56: Regra de comparação entre expressões

A definição da lógica booleana é semelhante à das expressões aritméticas. Existe recursividade e é necessário ter em consideração a prioridade das operações.

```

RuleBool parentheses, ands, ors;
parentheses = comparison | ('(' >> ors >> ')') ;
ands = parentheses[_val = _1] >> *("&&" >> parentheses) [_val &= _1];
ors  = ands[_val = _1] >> *("||" >> ands)                [_val |= _1];

RuleBool predicate;
predicate = ('/' >> ors >> '/') [phx::ref(result) = _1];

```

Código 7.57: Regras da lógica booleana

O valor final do predicado fica armazenado na variável `result`, que depois será usada para determinar se a sonda deve ou não disparar.

7.4.4 Detecção de erros de sintaxe

O sistema de detecção de erros consiste na utilização da maioria das regras mencionadas mas sem as ações. Algumas regras foram alteradas devido à inabilidade de distinguir entre variáveis numéricas e *string* antes da execução. Na inicialização, os predicados e as ações de cada sonda são avaliados. Como cada predicado é processado individualmente, caso o interpretador falhe, basta imprimir o predicado. Para as ações é algo diferente pois estas são processadas em grupo.

Para determinar que ação causou o erro é preciso verificar o iterador inicial fornecido. O iterador só avança após um elemento da regra inicial ter sido detetado, então vai avançar de ação em ação. No caso do processamento falhar basta iterar até ao próximo `' ; '` ou até ao final para obter a totalidade da ação com o erro.

Às regras das agregações são adicionadas ações que mantêm um registo de variáveis e funções de agregação. Quando existe uma tentativa de registar uma variável com uma função diferente, é gerado um erro. O mesmo acontece para as variáveis de agregação *lquantize* e respetivos parâmetros.

Durante a execução das ações, o interpretador pode falhar. Se a causa foi uma atribuição incorreta, uma das *flags* (atribuição de variável escalar ou de dicionário) vai estar sinalizada. Caso contrário é porque ocorreu um problema com o tipo de uma variável ou porque uma variável não foi inicializada. Novamente, utiliza-se o iterador inicial para verificar a ocorrência de um erro e qual a ação errónea.

7.4.5 Implementação das sondas

Esta secção aborda os detalhes mais importantes sobre a implementação das sondas.

As funções de registo de sondas geram funções *lambda* com o comportamento da sonda e estas são registadas como políticas do APEX através da função `apex::register_policy`. Quanto às sondas que operam em várias localidades, as suas funções de registo são expostas como ações para permitirem o registo de políticas nas instâncias do APEX de outras localidades. Os *handlers* das políticas são armazenados para as políticas depois poderem ser canceladas antes do processo de finalização do HPX. Isto é particularmente importante para

o caso das sondas **task**, pois o HPX gera num curto intervalo de tempo uma quantidade muito significativa de eventos de execuções de tarefas irrelevantes após a chamada de `hpx::finalize`.

As ações das sondas **BEGIN** e **END** são executadas no arranque e na finalização do HpxTrace, respetivamente. Todas as restantes sondas seguem o padrão descrito na secção de políticas (7.3.3): ocorrência do evento, obtenção das variáveis, avaliação do predicado e execução das ações. Estas últimas duas etapas correspondem à invocação dos interpretadores e são idênticas independentemente da sonda, mas as primeiras duas etapas devem ser explicadas.

Sondas definidas pelos utilizadores

O registo de uma sonda definida pelo utilizador começa com a definição de um novo tipo de evento APEX.

```
apex_event_type event_type = apex::register_custom_event(probe_name);
event_types[probe_name] = event_type;
```

Código 7.58: Definição de um novo tipo de evento APEX

Cada evento APEX tem o seu contexto que inclui um apontador para permitir a passagem de dados. Neste caso esses dados correspondem ao tipo `arguments`, que é um par dos dicionários com os argumentos da sonda.

```
typedef pair<map<string, double>, map<string, string>> arguments;
```

Código 7.59: Tipo dos dados incluídos no contexto

A função `trigger_probe` trata de converter os dados fornecidos no tipo `arguments`, de obter o identificador do fio de execução e de disparar a sonda com a função `apex::custom_event`.

```
void trigger_probe(std::string probe_name,
    std::map<std::string, double> double_arguments,
    std::map<std::string, std::string> string_arguments) {

    arguments args;
    args.first = double_arguments;
    args.second = string_arguments;

    std::uint64_t thread_id = reinterpret_cast<std::uint64_t>(hpx::threads::
        get_self_id().get());
    args.second["&thread"] = std::to_string(thread_id);

    apex::custom_event(event_types[probe_name], &args);
}
```

Código 7.60: Função de disparo das sondas definidas pelo utilizador

A política registada para o novo evento introduz o conteúdo de `arguments` nas variáveis locais da sonda.

```
arguments& args = *reinterpret_cast<arguments*>(context.data);
ScalarVars probe_svars;
MapVars probe_mvars;

std::map<std::string, double>& double_arguments = args.first;
for (auto const& arg : double_arguments){
    //arg.first -> variable name
    //arg.second -> variable value
    probe_svars.store_double(arg.first, arg.second);
}
std::map<std::string, std::string>& string_arguments = args.second;
for (auto const& arg : string_arguments){
    //arg.first -> variable name
    //arg.second -> variable value
    probe_svars.store_string(arg.first, arg.second);
}
```

Código 7.61: Inserção dos argumentos da sonda como variáveis de sonda

Com as variáveis obtidas, é invocado o interpretador do predicado, caso este exista. Se o predicado for inexistente ou verdadeiro, é invocado o interpretador das ações.

```
if(probe_predicate == "")
    || parse_predicate(probe_predicate.begin(), probe_predicate.end())
{
    parse_probe(script.begin(), script.end());
}
```

Código 7.62: Invocação dos interpretadores das sondas

Sondas dos contadores e sonda proc

As sondas dos contadores e a sonda das medições da diretoria */proc* correspondem a políticas do evento APEX_SAMPLE_VALUE. Este evento corresponde à invocação de `apex::sample_value`. A documentação do APEX menciona a existência deste tipo de evento mas não descreve que dados estão presentes no seu contexto. A solução consistiu em verificar o código fonte. Devido à exploração anterior do código fonte, sabe-se que para encontrar a classe presente no contexto deve-se verificar a função `on_sample_value` do `policy_handler`.

```
void policy_handler::on_sample_value(sample_value_event_data &data) {
    call_policies(sample_value_policies, &data, APEX_SAMPLE_VALUE);
}
```

Código 7.63: Função `on_sample_value` do `policy_handler`

Os dados inseridos no contexto são do tipo `sample_value_event_data`.

```
class sample_value_event_data : public event_data {
public:
    std::string * counter_name;
    double counter_value;
    bool is_threaded;
    bool is_counter;
    sample_value_event_data(int thread_id, std::string counter_name,
        double counter_value, bool threaded);
    ~sample_value_event_data();
};
```

Código 7.64: Classe `sample_value_event_data`

A classe `sample_value_event_data` contém dois campos importantes: o nome e o valor da medição. Apesar dos nomes dos campos referirem-se aos contadores, esta também é a estrutura de dados usada para leituras da diretoria `/proc`. Cada política registada avalia o nome da medição para determinar se deve ou não reagir.

O nome e o valor da medição são diretamente armazenados nas estruturas de dados das variáveis. No caso dos contadores HPX, o nome é decomposto através da função `get_counter_path_elements`, os índices são convertidos para *strings* e todos os componentes do nome são introduzidos como variáveis locais.

A sonda **counter-create** tem a responsabilidade extra de criar e periodicamente ler o contador requerido. Uma possibilidade para a leitura periódica seria utilizar os eventos `APEX_PERIODIC`. Optou-se no entanto por tentar simular a forma como o HPX lê os contadores periodicamente, tanto por uma questão de desempenho como para manter inalterado o comportamento esperado pelo utilizador. O mecanismo utilizado pelo HPX é o `hpx::util::interval_timer`. Este recebe uma função e um intervalo de tempo.

```
hpx::util::interval_timer* it = new hpx::util::interval_timer(
    hpx::util::bind_front(&read_counter, counter, counter_name),
    period,
    "",
    true);
```

Código 7.65: Inicialização do temporizador

A função fornecida trata de ler o contador e de causar a ocorrência do evento `APEX_SAMPLE_VALUE`.

```
bool read_counter(
    hpx::performance_counters::performance_counter counter,
    std::string counter_name)
{
    int value = counter.get_value<int>().get();
    apex::sample_value(counter_name, value);
    return true;
}
```

Código 7.66: Função de leitura do contador

Como a leitura de um contador através da API no código não causa um evento, o acontecimento tem que ser notificado manualmente.

Sondas das tarefas

A implementação da sonda task necessitou da alteração de alguns detalhes do APEX para tornar a sonda possível. Como foi estudado na seção sobre o seu funcionamento interno, os *listeners* recebem objetos `task_wrapper` para os eventos *start* e *resume*. Porém, o `policy_handler` não inclui estes objetos no contexto dos eventos. Apenas incluiu os respetivos `task_identifier`.

```
bool policy_handler::on_start(std::shared_ptr<task_wrapper> &tt_ptr) {
    call_policies(start_event_policies, (void *)tt_ptr->get_task_id(),
        APEX_START_EVENT);
    return true;
}

bool policy_handler::on_resume(std::shared_ptr<task_wrapper> &tt_ptr) {
    call_policies(resume_event_policies, (void *)tt_ptr->get_task_id(),
        APEX_RESUME_EVENT);
    return true;
}
```

Código 7.67: Funções `on_start` e `on_resume` do *policy_listener*

Algo equivalente ocorre com os eventos *stop* e *yield*. São fornecidos ao `policy_listener` objetos `profiler` mas este apenas inclui o `task_identifier` nos contextos dos eventos.

```
void policy_handler::on_stop(std::shared_ptr<profiler> &p) {
    call_policies(stop_event_policies, (void *)p->tt_ptr->get_task_id(),
        APEX_STOP_EVENT);
}

void policy_handler::on_yield(std::shared_ptr<profiler> &p) {
    call_policies(yield_event_policies, (void *)p->tt_ptr->get_task_id(),
        APEX_YIELD_EVENT);
}
```

Código 7.68: Funções `on_stop` e `on_yield` do *policy_listener*

O que isto significa é que através das políticas do APEX só se tem acesso à informação presente no `task_identifier`.

```
apex_function_address address;
std::string name;
std::string _resolved_name;
bool has_name;
```

Código 7.69: Campos da classe `task_identifier`

Só com o nome e com o tipo do evento, seria possível implementar uma sonda que indicasse qual a tarefa cujo estado de execução tivesse acabado de ser alterado. Contudo, esta solução não lida satisfatoriamente com a recursividade. Como podem existir várias tarefas em simultâneo com o mesmo nome, não seria claro qual acabou de executar ou foi colocada em pausa. Foi esta razão que justificou a alteração do código fonte do APEX. Para isso as funções do `policy_listener` foram alteradas para incluírem o `task_wrapper` no contexto dos eventos.

Os novos valores disponíveis são inseridos como variáveis de sonda (consultar 7.3.4). Os valores sobre a gestão da memória não foram testados na totalidade devido à inabilidade de ativar com sucesso a monitorização deste aspeto na instalação do HPX/APEX utilizada. Como a obtenção dos outros parâmetros ocorre corretamente, em princípio também deve funcionar para instâncias com a monitorização de memória ativada.

Sondas das mensagens

A sonda `message` dispara conforme o envio e receção de parcelas. Corresponde aos eventos `APEX_SEND` e `APEX_RECV`. No contexto destes eventos está presente a classe `apex::message_event_data`.

```
class message_event_data : public event_data {
public:
    uint64_t tag;
    uint64_t size;
    uint64_t source_rank;
    uint64_t source_thread;
    uint64_t target;
    (...)
}
```

Código 7.70: Classe `apex::message_event_data`

Todos os campos são inseridos como variáveis de sonda. Na versão do APEX utilizada no desenvolvimento da ferramenta (v2.4.1) o campo `source_thread` é sempre 0 pois ainda não está implementado.

7.5 ALTERAÇÕES AO HPX E EXTRAÇÃO DE NOVOS DADOS

A informação disponibilizada pelo HPX ao APEX é algo limitada para conseguir um rastreamento interpretável. Para colmatar isso o código fonte do HPX e do APEX foram alterados para permitir a extração de mais informação e a adição de novas variáveis de sonda.

- Sonda **task**
 - `&thread` - identificador do fio de execução HPX cujo estado de execução foi alterado
- Sondas definidas pelo utilizador
 - `&thread` - identificador do fio de execução HPX que causou o disparo da sonda
- Sonda **message**
 - `&action` - ação associada à parcela enviada/recebida

Para além destas adições, as variáveis `&source_thread` e `&size` foram alteradas. Para eventos de envio a variável foi implementada. Para eventos de receção `recv`, a variável `&size` passa a ser preenchida com o tamanho correto.

Com a adição dos identificadores dos fios de execução às variáveis das sondas, é possível uma análise mais granular. Por exemplo, ao instrumentar uma tarefa que é executada para cada segmento de dados com uma sonda personalizável que recolhe informação da aplicação, com a comparação dos identificadores dos fios de execução é possível associar as métricas de uma execução específica com um dos segmentos de dados.

A informação originalmente disponibilizada pelos eventos *send* e *recv* é bastante limitada. A *tag* permite associar um envio particular com uma receção e o tamanho do envio sugere o propósito da mensagem. No mínimo era necessária informação sobre a ação associada à parcela. Acabar de implementar `&source_thread` foi uma tentativa de associar uma parcela à tarefa que a gerou. No entanto, o valor de `&source_thread` nunca correspondia aos valores `&thread` das tarefas, o que sugere que o valor obtido corresponde a um fio de execução responsável pelo envio de todas as parcelas de uma localidade.

Ao alterar o código fonte do HPX para suportar as novas funcionalidades do HpxTrace, é importante manter a modularidade do processo *build*. Ou seja, deve ser possível alternar entre a versão que suporta o HpxTrace e a versão que suporta o APEX original (ou outra ferramenta que utilize a mesma API).

Foi adicionada a opção `-DHPX_WITH_HPXTRACE:BOOL=X` ao build do HPX. O uso desta opção também requer a ativação do APEX com `-DHPX_WITH_APEX:BOOL=X`.

```
if (HPX_WITH_APEX)
  hpx_option(
    HPX_WITH_HPXTRACE BOOL "Enable HpxTrace instrumentation support." OFF
    CATEGORY "Profiling"
  )
if (HPX_WITH_HPXTRACE)
  hpx_add_config_define(HPX_HAVE_HPXTRACE) # tell HPX that we use HpxTrace
```

```
endif()
(...)
```

Código 7.71: Adição da nova opção no CMakeLists

Ao ativar o HpxTrace, o identificador `HPX_HAVE_HPXTRACE` fica definido. Deste modo pode ser usado em conjunção com diretivas de pré-processamento para manter duas versões do HPX.

No APEX são definidas novas versões das funções `start`, `send` e `recv` com os argumentos adicionais necessários, e os respetivos *adapters*. Depois, conforme o estado de `HPX_HAVE_HPXTRACE`, são registadas as funções originais ou as adaptadas.

```
static void apex_register_with_hpx(void) {
    (...)
    #ifdef HPX_HAVE_HPXTRACE
        reg.type = hpx::util::external_timer::start_flag;
        reg.record.start = &HpxTrace_start_adapter;
        hpx::util::external_timer::register_external_timer(reg);
    #else
        reg.type = hpx::util::external_timer::start_flag;
        reg.record.start = &start_adapter;
        hpx::util::external_timer::register_external_timer(reg);
    #endif
}
```

Código 7.72: Registo da função `apex::start`

No lado do HPX, os tipos das funções devem concordar com as funções registadas.

```
#ifdef HPX_HAVE_HPXTRACE
    typedef void start_t(std::shared_ptr<task_wrapper>, uint64_t thread_id);
    typedef void send_t(uint64_t, uint64_t, uint64_t, uint64_t, char const*);
    typedef void recv_t(uint64_t, uint64_t, uint64_t, uint64_t, char const*);
#else
    typedef void start_t(std::shared_ptr<task_wrapper>);
    typedef void send_t(uint64_t, uint64_t, uint64_t);
    typedef void recv_t(uint64_t, uint64_t, uint64_t, uint64_t);
#endif
```

Código 7.73: Adição dos novos tipos de funções

Por fim, falta alterar os *wrappers* que invocam as funções registadas.

```
#ifdef HPX_HAVE_HPXTRACE
static inline void send(uint64_t tag, uint64_t size, uint64_t target,
    uint64_t source_thread, char const* action_name)
{
    if (send_function != nullptr)
    {
```

```

        send_function(tag, size, target, source_thread, action_name);
    }
}
#else
    static inline void send(uint64_t tag, uint64_t size, uint64_t target)
    {
        if (send_function != nullptr)
        {
            send_function(tag, size, target);
        }
    }
#endif

```

Código 7.74: Invocação da função send registada

7.5.1 Sondas *task*

Como o HpxTrace extrai os dados da estrutura `task_wrapper`, é-lhe adicionado um novo campo relativo ao identificador do fio de execução. Como o HPX apenas mantém apontadores dos `task_wrappers` e não tem qualquer conhecimento sobre a estrutura em si, a inserção do identificador tem que ser mediada através dos métodos expostos pela API do APEX. Aqui surgiram duas possibilidades, alterar o `new_task`, que instancia o `task_wrapper`, ou a função `start`. Ora, enquanto que `new_task` é usada em 7 localizações diferentes, `start` apenas é usada no construtor do `scoped_timer` (consultar 6.3.3). Para além disso, o `scoped_timer` é usado no ciclo de escalonamento de tarefas, onde se tem acesso direto à informação do fio de execução, de onde pode ser obtido o identificador desejado. As funções `apex::yield` e `apex::stop` permanecem inalteradas já que o identificador permanece armazenado no `task_wrapper`.

```

#ifdef HPX_HAVE_APEX

    #ifdef HPX_HAVE_HPXTRACE

        std::uint64_t thread_id =
            reinterpret_cast<std::uint64_t>(thrd->get_thread_id().get());

        util::external_timer::scoped_timer profiler(
            thrd->get_timer_data(), thread_id);
    #else
        util::external_timer::scoped_timer profiler(
            thrd->get_timer_data());
    #endif
#endif

```

Código 7.75: Instrumentação do *scheduling loop* com o HpxTrace

7.5.2 Sondas *message*

Tal como a função `start`, são adicionadas novas versões das funções `send` e `recv` que recebem como argumento extra o nome da ação.

Os objetos `parcel` contêm um apontador para um objeto `base_action`, que representa a ação associada à parcela. Tal como já acontece no evento de receção, a partir deste objeto é possível obter o identificador do fio de execução que deu origem à ação. Além disso também é possível obter o nome da ação.

```
#if defined(HPX_HAVE_APEX) && defined(HPX_HAVE_PARCEL_PROFILING)

#ifdef HPX_HAVE_HPXTRACE
    util::external_timer::send(p.parcel_id().get_lsb(), p.size(),
        p.destination_locality_id(),
        reinterpret_cast<uint64_t>(p.get_action()->get_parent_thread_id().get()),
        p.get_action()->get_action_name());
#else
    util::external_timer::send(p.parcel_id().get_lsb(), p.size(),
        p.destination_locality_id(), 0, "");
#endif
#endif
```

Código 7.76: Instrumentação do envio de parcelas com o HpxTrace

Tal como foi descoberto na secção sobre a integração HPX/APEX, o valor do tamanho da parcela dado a `recv` nunca é inicializado. Uma forma de estimar o tamanho é comparar a posição do arquivo antes e depois da leitura de uma parcela para contar quantos *bytes* são lidos. O nome da ação também é adicionado.

```
bool parcel::load_schedule(serialization::input_archive & ar,
    std::size_t num_thread, bool& deferred_schedule)
{
    std::size_t archive_pos = ar.current_pos();
    load_data(ar);
    (...)

#if defined(HPX_HAVE_APEX) && defined(HPX_HAVE_PARCEL_PROFILING)
#ifdef HPX_HAVE_HPXTRACE

    util::external_timer::recv(
        data_.parcel_id().get_lsb(),
        ar.current_pos() - archive_pos,
        naming::get_locality_id_from_gid(data_.source_id_),
        reinterpret_cast<std::uint64_t>(action_->get_parent_thread_id().get()),
        action_->get_action_name());
#else
    util::external_timer::recv(
        data_.parcel_id().get_lsb(),
```

```

        size_,
        naming::get_locality_id_from_gid(data_.source_id_),
        reinterpret_cast<std::uint64_t>(action_>get_parent_thread_id().get()));
    #endif
#endif
    (...)
}

```

Código 7.77: Instrumentação da receção de parcelas com o HpxTrace

Ao testar experimentalmente, o valor obtido era sempre menor em 17 *bytes* quando comparado com o valor dado por `send`. Esta diferença poderá corresponder a metadados que não são incluídos no arquivo. Para os valores serem consistentes de ambos os lados, são adicionados 17 *bytes* ao valor dado a `recv`.

7.5.3 Sonda definidas pelo utilizador

A adição da nova variável a estas sondas foi a mais simples e não implicou alterar o código fonte do HPX nem o do APEX. A única consideração relevante a ter neste objetivo foi obter o identificador de fios de execução com o mesmo formato dos identificadores das restantes sondas. O identificador do fio de execução atual pode ser obtido através da função `hpx::threads::get_self_id()` e depois convertido para o formato `std::uint64_t`.

Este pedido é então adicionado à função `trigger_probe`, onde o identificador é adicionado ao conjunto de variáveis locais colocadas no contexto do evento APEX.

```

void trigger_probe(std::string probe_name,
    std::map<std::string, double> double_arguments,
    std::map<std::string, std::string> string_arguments){

    arguments args;
    args.first = double_arguments;
    args.second = string_arguments;

    std::uint64_t thread_id = reinterpret_cast<std::uint64_t>(hpx::threads::
        get_self_id().get());
    args.second["&thread"] = std::to_string(thread_id);

    apex::custom_event(event_types[probe_name], &args);
}

```

Código 7.78: Adição do fio de execução no disparo das sondas definidas pelo utilizador

CASO DE ESTUDO - *STENCIL*

8.1 INTRODUÇÃO DO PROBLEMA

O caso de estudo escolhido para exemplificar o uso do HpxTrace é um problema de distribuição de calor unidimensional e circular (o primeiro e último elementos são vizinhos). Este problema é um *stencil*, um algoritmo que calcula um valor de uma posição com base nos valores das posições vizinhas.

O HPX disponibiliza um conjunto de implementações do *stencil*. Começa com uma versão sequencial e iterativamente o código é paralelizado e otimizado até alcançar uma versão que incorpora o máximo possível de capacidades do HPX como futurização, componentes e ações. Destas foram selecionadas as versões 6 e 7. Ambas são suficientemente complexas para que uma análise com o HpxTrace seja mais útil do que uma instrumentação mais informal e as otimizações introduzidas na versão 7 são um bom alvo de observação com as sondas disponíveis.

Neste capítulo apenas são abordados os aspetos suficientes para perceber o que está a acontecer e para interpretar os resultados. O código fonte completo das versões 6 e 7 e o resto dos detalhes podem ser consultados no guia completo, em https://hpx-docs.stellar-group.org/latest/html/examples/1d_stencil.html

Os dados do problema são divididos em partições de igual dimensão e distribuídos pelas várias localidades. Em cada partição, o cálculo da distribuição do calor é aplicado aos dados nela presente. Para que o cálculo seja aplicado ao primeiro e último elemento, cada partição necessita dos dados dos elementos vizinhos. Estes elementos podem estar numa partição atribuída a outra localidade o que implica comunicação e dependências entre localidades.

Numa versão mais tradicional, cada iteração só é iniciada após todos os valores da iteração anterior terem sido calculados. Com a distribuição dos dados e com o uso dos mecanismos de futurização, as iterações de cada partição podem ser mais desfasadas. Em vez de esperar que todos os valores em todas as partições sejam calculados, o avanço de uma partição para a próxima iteração apenas depende diretamente das iterações das partições vizinhas o que pode resultar em ganhos de desempenho (indiretamente existem dependências entre todas as partições pois cada partição depende da próxima vizinha).

Como seria de esperar, em termos de HPX uma partição é um componente. A sua ação principal é `get_data`. Esta é usada para obter dados e distingue entre a partição ser o alvo da iteração ou uma das partições vizinhas.

```
partition_data get_data(partition_type t) const
{
    switch (t)
    {
        case left_partition:
            return partition_data(data_, data_.size()-1);

        case middle_partition:
            break;

        case right_partition:
            return partition_data(data_, 0);

        default:
            HPX_ASSERT(false);
            break;
    }
    return data_;
}
```

Código 8.1: Função `get_data` do servidor da partição

O parâmetro recebido indica que elementos são necessários para uma operação poder ser efetuada. No caso de ser a partição vizinha à esquerda, é retornado apenas o último elemento. No caso de ser a partição vizinha à direita, é retornado apenas o primeiro elemento. Caso seja a partição a ser iterada, todos os dados são retornados.

No cliente da partição, o método `get_data` resulta na invocação assíncrona da ação.

```
hpx::future<partition_data> get_data(partition_server::partition_type t) const
{
    partition_server::get_data_action act;
    return hpx::async(act, get_id(), t);
}
```

Código 8.2: Função `get_data` do cliente da partição

O núcleo do programa é a ação `heat_part`, que é direcionada para a localidade onde se encontra a partição a ser iterada. Consiste em requisitar os dados das três partições, calcular a operação de distribuição de calor e colocar o resultado numa nova partição na mesma localidade.

8.2 VERSÃO 6

Na versão 6 os dados são pedidos assincronamente e, assim que estes estejam disponíveis, é aplicada a operação do *stencil*. O HPX tem um *local control object* (LCO) para expressar exatamente este comportamento. LCOs são mecanismos de sincronização que quando ativados causam a execução de uma nova tarefa. No caso do LCO *dataflow*, uma função é associada a um conjunto de futuros. Quando todos os futuros estiverem prontos, a função é invocada com os dados dos futuros.

```
static partition heat_part(partition const& left, partition const& middle,
    partition const& right)
{
    using hpx::dataflow;
    using hpx::unwrapping;

    return dataflow(
        hpx::launch::async,
        unwrapping(
            [left, middle, right](partition_data const& l, partition_data
                const& m,
                partition_data const& r)
            {
                HPX_UNUSED(left);
                HPX_UNUSED(right);

                return partition(middle.get_id(), heat_part_data(l, m, r));
            }
        ),
        left.get_data(partition_server::left_partition),
        middle.get_data(partition_server::middle_partition),
        right.get_data(partition_server::right_partition)
    );
}
```

Código 8.3: Ação `heat_part` da versão 6

O primeiro argumento de um *dataflow* especifica se a função deve ser invocada sincronamente ou assincronamente. O segundo argumento é a função a invocar. Por fim, recebe os futuros pelos quais deve esperar.

O uso de *unwrapping* lida com a extração dos resultados dos futuros para que possam ser usados diretamente pela função. Com todos os dados disponíveis, a função `heat_part_data` calcula a nova iteração que é armazenada num novo componente de partição.

8.3 ANÁLISE DE UMA EXECUÇÃO

Nesta secção a análise da execução é feita adicionando progressivamente as várias sondas até ser obtida uma imagem final completa.

Para demonstrar como o HpxTrace pode ser usado para analisar em detalhe a execução de um segmento do algoritmo, a execução do problema de distribuição do calor é parametrizada com duas partições (uma por localidade) e com apenas uma iteração.

Para analisar a execução são usadas sondas definidas pelo utilizador, sondas de tarefas e sondas de mensagens. As sondas definidas pelo utilizador são colocadas em pontos de interesse para contextualizar as restantes sondas. A sonda **start_heat** é colocada no início de `heat_part` para sinalizar o começo da iteração. A sonda **received_data** é colocada no início da função dada ao `dataflow` para sinalizar a receção dos dados e a sonda **end_heat** é colocada no fim da mesma função para sinalizar a conclusão da iteração.

```
static partition heat_part(partition const& left, partition const& middle,
    partition const& right)
{
    using hpx::dataflow;
    using hpx::unwrapping;

    HpxTrace::trigger_probe("start_heat");

    return dataflow(
        hpx::launch::async,
        unwrapping(
            [left, middle, right](partition_data const& l, partition_data const&
                m,
                partition_data const& r)
            {
                HPX_UNUSED(left);
                HPX_UNUSED(right);

                HpxTrace::trigger_probe("received_data");
                partition p = partition(middle.get_id(), heat_part_data(l, m, r))
                ;
                HpxTrace::trigger_probe("end_heat");
                return p;
            }
        ),
        left.get_data(partition_server::left_partition),
        middle.get_data(partition_server::middle_partition),
        right.get_data(partition_server::right_partition)
    );
}
```

Código 8.4: Versão instrumentada da ação `heat_part` da versão 6

A próxima adição é a sonda das mensagens para monitorizar a comunicação inter-localidades do algoritmo. O primeiro passo é listar todas as mensagens trocadas para depois serem seleccionadas as mais relevantes.

```
start_heat[0]{
    print("START_HEAT");
}
end_heat[0]{
    print("END_HEAT");
}
received_data[0]{
    print("RECEIVED_DATA");
}

message[0]
{
    print(&source_rank + " -> " + &target + " tag: " + &tag + " action: " + &action);
}
```

Código 8.5: *Script* com as sondas usadas

A execução analisada envolve duas localidades, cada uma com uma partição. As sondas apenas são registadas na localidade 0. Como a comunicação é simétrica, sondas na localidade 1 seriam redundantes.

```
z@pc: ~/Desktop/HPX-Performance-Counters/1d_stencil/build
File Edit View Search Terminal Help
z@pc:~/Desktop/HPX-Performance-Counters/1d_stencil/build$ mpirun -np 2 ./1d_stencil_6 --nx=1000000 --np=2 --nt=1 --file messages_script.txt
Initializing HpxTrace
HpxTrace Initialized
0 -> 1 tag: 83 action: primary_namespace_decrement_credit_action
0 -> 1 tag: 84 action: N3hpx10components6server23create_component_actionI16partition_serverJmdiEEE
1 -> 0 tag: 80 action: set_value_action_id_typedmanaged_component_tag
0 -> 1 tag: 85 action: heat_part_action
START HEAT
0 -> 1 tag: 86 action: get_data_action
1 -> 0 tag: 81 action: get_data_action
0 -> 1 tag: 88 action: N3hpx4lcos19base_lco_with_valueI14partition_dataS2_NS_6traits6detail21managed_component_tagEE16set_value_actionE
0 -> 1 tag: 87 action: get_data_action
1 -> 0 tag: 84 action: N3hpx4lcos19base_lco_with_valueI14partition_dataS2_NS_6traits6detail21managed_component_tagEE16set_value_actionE
1 -> 0 tag: 82 action: get_data_action
0 -> 1 tag: 89 action: N3hpx4lcos19base_lco_with_valueI14partition_dataS2_NS_6traits6detail21managed_component_tagEE16set_value_actionE
1 -> 0 tag: 85 action: N3hpx4lcos19base_lco_with_valueI14partition_dataS2_NS_6traits6detail21managed_component_tagEE16set_value_actionE
RECEIVED_DATA
1 -> 0 tag: 83 action: N3hpx4lcos19base_lco_with_valueI9partitionS2_NS_6traits6detail21managed_component_tagEE16set_value_actionE
END HEAT
1 -> 0 tag: 86 action: primary_namespace_decrement_credit_action
0 -> 1 tag: 90 action: primary_namespace_decrement_credit_action
0 -> 1 tag: 91 action: get_data_action
HpxTrace finalizing
1 -> 0 tag: 87 action: N3hpx4lcos19base_lco_with_valueI14partition_dataS2_NS_6traits6detail21managed_component_tagEE16set_value_actionE
HpxTrace finalized
Localities,OS_Threads,Execution_Time_sec,Points_per_Partition,Partitions,Time_Steps
2, 12, 0.656790275, 1000000, 2, 1
z@pc:~/Desktop/HPX-Performance-Counters/1d_stencil/build$
```

Figura 10: Resultado do *script*

A primeira parcela detetada pertence ao sistema de endereçamento global AGAS. Para manter a contagem de referências globais, o AGAS utiliza um sistema de créditos, onde os créditos são "emprestados" ao GID e depois devolvidos ao AGAS quando o GID ficar fora de contexto. Quando o AGAS recebe todos os créditos de volta, o objeto pode ser destruído. Não é uma parcela relevante para a análise atual e pode ser filtrada.

Antes de começar o algoritmo, é iniciado um componente em cada localidade. A parcela enviada para a localidade 1 contém um pedido para a criação de um componente (`components6server23create_component_actionI16`) do tipo `partition_server` (`partition_serverJmdiEEE`).

A parcela com a ação `set_value_action_id_typedmanaged_component_tag` é a resposta desse pedido e transporta o identificador do componente criado para a localidade que o pediu.

O início da iteração na localidade 1 é evidente com o envio de `heat_part_action`.

Dentro da iteração, cada localidade efetua dois pedidos com `get_data`. De forma semelhante à instanciação de um componente, a resposta consiste numa ação do tipo `set_value_action`. Mais especificamente, `base_lco_with_valueI14partition_data` sugere que é colocado um valor do tipo `partition_data` num LCO, que neste caso é um dos futuros dados ao *dataflow*.

Com a sonda das tarefas apenas pretende-se monitorizar as tarefas associadas diretamente à execução da iteração. Outras tarefas devem ser ignoradas, tanto para minimizar impactos na execução, como para manter a legibilidade do perfil de execução. As tarefas geradas por ações ficam com o mesmo nome da tarefa, então `heat_part_action` é um alvo. Quando o *dataflow* é ativado, é gerada uma nova tarefa que também devia ser observada. Neste caso não é atribuído automaticamente um nome à tarefa, ficando a variável com o nome `<unknown>` por defeito. Para remediar essa situação, recorre-se ao mecanismo de anotação.

```
return dataflow(
    hpx::launch::async,
    hpx::util::annotated_function(
        unwrapping(
            (...)
        ), "heat_part_data"),
    left.get_data(partition_server::left_partition),
    middle.get_data(partition_server::middle_partition),
    right.get_data(partition_server::right_partition)
);
```

Código 8.6: Anotação da tarefa `heat_part_data`

Com esta alteração, a tarefa resultante irá ficar com o nome `heat_part_data`.

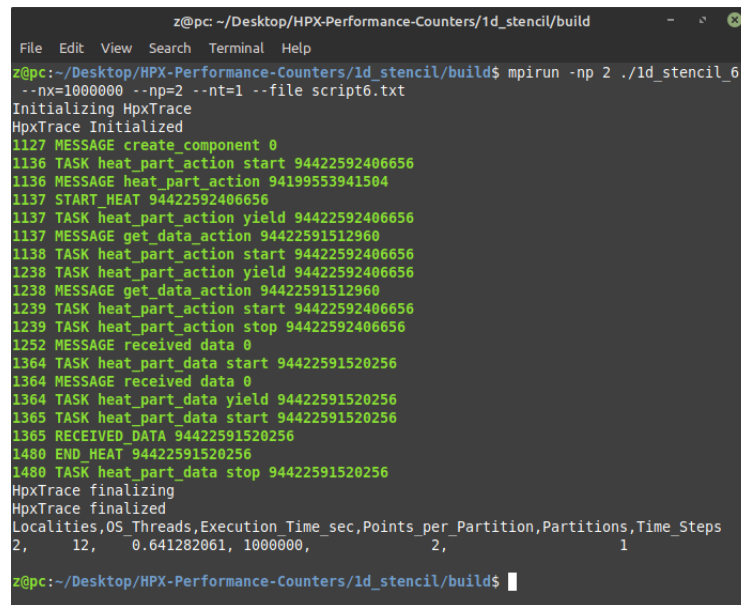
Finalmente, foi desenvolvido um *script* que filtra e formata todos os dados considerados relevantes. Todas as sondas estão apenas registadas na localidade 1 e imprimem a *timestamp* em milissegundos e o identificador do fio de execução. As sondas de tarefas apenas aceitam as tarefas mencionadas e imprimem a alteração de estado. As sondas das mensagens monitorizam as parcelas relevantes na perspetiva de uma iteração na localidade 1. Ou seja, a receção de um pedido de criação de um componente, a receção da ação `heat_part_action`, o envio de um pedido `get_data` e a receção do seu resultado. Os nomes das ações mais complexos são transformados num formato mais legível.

```
message[1]::[receive]::/"base_lco_with_value" in &action/
{
    print(str(timestamp()) + " MESSAGE received data " + &source_thread);
}
```


Código 8.7: Sonda para a receção do resultado da ação `get_data`

O *script* completo pode ser consultado no anexo [A.5](#)

Com o uso de todas as sondas é obtida uma imagem mais completa da execução de uma iteração.



```

z@pc: ~/Desktop/HPX-Performance-Counters/1d_stencil/build
File Edit View Search Terminal Help
z@pc:~/Desktop/HPX-Performance-Counters/1d_stencil/build$ mpirun -np 2 ./1d_stencil_6
--nx=1000000 --np=2 --nt=1 --file script6.txt
Initializing HpxTrace
HpxTrace Initialized
1127 MESSAGE create_component 0
1136 TASK heat_part_action start 94422592406656
1136 MESSAGE heat_part_action 94199553941504
1137 START_HEAT 94422592406656
1137 TASK heat_part_action yield 94422592406656
1137 MESSAGE get_data_action 94422591512960
1138 TASK heat_part_action start 94422592406656
1238 TASK heat_part_action yield 94422592406656
1238 MESSAGE get_data_action 94422591512960
1239 TASK heat_part_action start 94422592406656
1239 TASK heat_part_action stop 94422592406656
1252 MESSAGE received data 0
1364 TASK heat_part_data start 94422591520256
1364 MESSAGE received data 0
1364 TASK heat_part_data yield 94422591520256
1365 TASK heat_part_data start 94422591520256
1365 RECEIVED_DATA 94422591520256
1480 END_HEAT 94422591520256
1480 TASK heat_part_data stop 94422591520256
HpxTrace finalizing
HpxTrace finalized
Localities,OS_Threads,Execution_Time_sec,Points_per_Partition,Partitions,Time_Steps
2, 12, 0.641282061, 1000000, 2, 1
z@pc:~/Desktop/HPX-Performance-Counters/1d_stencil/build$

```

Figura 11: Trace de uma execução da versão 6 do *stencil*

Nos primeiros eventos são visíveis os passos para o arranque da iteração. O componente é criado, a ação `heat_part_action` é recebida e a tarefa `heat_part_action` é iniciada.

O arranque da tarefa `heat_part_action` aparece primeiro do que a receção da ação, embora a *timestamp* seja a mesma. Isto é resultado do paralelismo, já que a execução da ação recebida, a notificação do APEX sobre a parcela recebida e o processamento do disparo da sonda ocorrem em fios de execução diferentes. A relevância deste aspeto é que ao definir sondas de eventos muito próximos, é importante não depender da ordem exata do disparo (por exemplo, iniciar uma variável que é usada na outra sonda). Além disso, ao analisar os resultados é importante considerar esta possibilidade e não chegar a conclusões usando apenas a ordem pela qual os eventos aparecem.

O *trace* demonstra que a tarefa `heat_part_action` é executada e suspensa várias vezes. A frequência deste comportamento varia entre execuções, podendo em alguns casos ocorrer múltiplas transições seguidas entre *start* e *stop*.

Por volta dos 1239 milissegundos, a tarefa `heat_part_action` termina após o envio de ambos os pedidos `get_data`. A tarefa `heat_part_data` começa a ser executada num novo fio de execução (como pode ser observado pelo identificador) assim que o resultado de ambos os pedidos é recebido (1364ms). A secção entre os 1364 e os 1365 milissegundos é outro exemplo de que a ordem dos eventos próximos não

deve ser interpretada literalmente. Mesmo com essa adversidade, o *trace* permite deduzir razoavelmente o comportamento interno do HPX.

Após essa sequência de eventos agrupados, os próximos 115 milissegundos são gastos na tarefa `heat_part_data`, concluindo uma iteração.

Por fim, um aspecto que vale a pena mencionar é que o fio de execução que envia a ação `get_data` não é o mesmo que executa a tarefa `heat_part_action`. Isto suporta a teoria que o envio de mensagens é responsabilidade de um fio de execução separado.

8.4 VERSÃO 7

A versão 7 do *stencil* procura eliminar o tempo de inatividade à espera dos dados dos vizinhos. Durante esse tempo de espera, os valores interiores da partição podem ser calculados. Quando os dados dos vizinhos estiverem disponíveis, a iteração pode ser terminada com o cálculo dos elementos fronteiros.

Os mecanismos de sincronização dos LCOs são novamente usados para expressar o comportamento pretendido. Com o uso de `then`, os valores interiores são calculados assim que os dados estejam prontos (o que deve ser quase instantâneo considerando que os dados estão na mesma localidade). Com esta abordagem o cálculo é encadeado ao `get_data` da partição central, mantendo a mesma interface de assincronia e futuros já existente.

```
hpx::shared_future<partition_data> middle_data =
    middle.get_data(partition_server::middle_partition);

hpx::future<partition_data> next_middle = middle_data.then(
    unwrapping(
        [middle](partition_data const& m) -> partition_data
        {
            HPX_UNUSED(middle);

            std::size_t size = m.size();
            partition_data next(size);
            for (std::size_t i = 1; i != size-1; ++i)
                next[i] = heat(m[i-1], m[i], m[i+1]);

            return next;
        }
    )
);
```

Código 8.8: Primeira parte da ação `heat_part` da versão 7

O *dataflow* recebe novamente um futuro por partição. A diferença é que agora a tarefa gerada pelo *dataflow* apenas precisará de calcular os extremos da partição.

```
return dataflow(
```

```

hpx::launch::async,
unwrapping(
    [left, middle, right](partition_data next, partition_data const& l,
        partition_data const& m, partition_data const& r) -> partition
    {
        HPX_UNUSED(left);
        HPX_UNUSED(right);

        std::size_t size = m.size();
        next[0] = heat(l[size-1], m[0], m[1]);
        next[size-1] = heat(m[size-2], m[size-1], r[0]);

        return partition(middle.get_id(), std::move(next));
    }
),
std::move(next_middle),
left.get_data(partition_server::left_partition),
middle_data,
right.get_data(partition_server::right_partition)
);

```

Código 8.9: Segunda parte da ação `heat_part` da versão 7

Esta versão é também instrumentada com as sondas `start_heat`, `end_heat` e `received_data` nos mesmos sítios. Para além dessas é adicionada a sonda `finished_middle` no final da função fornecida ao `then`.

Como a nova versão não altera a forma como as mensagens são trocadas, são usadas as mesmas sondas de mensagem.

O uso do mecanismo de anotação volta a ser essencial para identificar as tarefas das duas fases. A tarefa do `then` que calcula os valores interiores é etiquetada com `middle_heat_part`. A tarefa gerada pelo `dataflow` é etiquetada com `order_heat_part`.

```

HpxTrace::trigger_probe("start_heat");

hpx::future<partition_data> next_middle = middle_data.then(
    hpx::util::annotated_function(
        unwrapping(
            [middle](partition_data const& m) -> partition_data
            {
                HPX_UNUSED(middle);

                std::size_t size = m.size();
                partition_data next(size);
                for (std::size_t i = 1; i != size-1; ++i)
                    next[i] = heat(m[i-1], m[i], m[i+1]);
                HpxTrace::trigger_probe("finished_middle");
                return next;
            }
        )
    )
);

```

```

    }
    ), "middle_heat_part")
);

```

Código 8.10: Primeira parte da versão instrumentada da ação `heat_part` da versão 7

```

return dataflow(
    hpx::launch::async,
    hpx::util::annotated_function(
        unwrapping(
            [left, middle, right](partition_data next, partition_data const& l,
                partition_data const& m, partition_data const& r) -> partition
            {

                HpxTrace::trigger_probe("received_data");

                HPX_UNUSED(left);
                HPX_UNUSED(right);

                std::size_t size = m.size();
                next[0] = heat(l[size-1], m[0], m[1]);
                next[size-1] = heat(m[size-2], m[size-1], r[0]);

                HpxTrace::trigger_probe("end_heat");

                return partition(middle.get_id(), std::move(next));
            }
        ), "border_heat_part"),
    std::move(next_middle),
    left.get_data(partition_server::left_partition),
    middle_data,
    right.get_data(partition_server::right_partition)
);

```

Código 8.11: Segunda parte da versão instrumentada da ação `heat_part` da versão 7

O *script* utilizado pode ser consultado no anexo [A.6](#).

```

z@pc: ~/Desktop/HPX-Performance-Counters/1d_stencil/build
File Edit View Search Terminal Help
z@pc:~/Desktop/HPX-Performance-Counters/1d_stencil/build$ mpirun -np 2 ./1d_stencil_7
--nx=1000000 --np=2 --nt=1 --file script7.txt
Initializing HpxTrace
HpxTrace Initialized
1090 MESSAGE create_component 0
1151 TASK heat_part_action start 94597092536064
1151 MESSAGE heat_part_action 94810266238976
1151 START_HEAT 94597092536064
1252 TASK heat_part_action yield 94597092536064
1253 TASK heat_part_action start 94597092536064
1253 TASK heat_part_action yield 94597092536064
1253 MESSAGE get_data_action 94597091681024
1254 TASK heat_part_action start 94597092536064
1254 MESSAGE get_data_action 94597091681024
1252 TASK middle_heat_part start 94597091685248
1267 TASK middle_heat_part yield 94597091685248
1268 TASK middle_heat_part start 94597091685248
1254 TASK heat_part_action yield 94597092536064
1271 TASK heat_part_action start 94597092536064
1271 TASK heat_part_action stop 94597092536064
1355 MESSAGE received data 0
1359 MESSAGE received data 0
1366 FINISHED_MIDDLE 94597091685248
1366 TASK middle_heat_part stop 94597091685248
1366 TASK border_heat_part start 94597092474880
1367 TASK border_heat_part yield 94597092474880
1367 TASK border_heat_part start 94597092474880
1368 RECEIVED_DATA 94597092474880
1368 END_HEAT 94597092474880
1369 TASK border_heat_part stop 94597092474880
HpxTrace finalizing
HpxTrace finalized
Localities, OS_Threads, Execution_Time_sec, Points_per_Partition, Partitions, Time_Steps
2, 12, 0.504371896, 1000000, 2, 1
z@pc:~/Desktop/HPX-Performance-Counters/1d_stencil/build$

```

Figura 12: Execução monitorizada da versão 7 do *stencil*

Inicialmente aparecem os mesmos eventos que na versão 6, e a tarefa `heat_part_action` também termina após o envio de ambos os `get_data`. No tempo que normalmente seria desperdiçado a esperar pelos vizinhos, a tarefa `middle_heat_part` começa a ser executada num novo fio de execução e demora aproximadamente 114 segundos, equivalente ao tempo de `heat_part_data` da versão 6. Quando as três dependências são cumpridas, a tarefa `border_heat_part` é rapidamente executada.

8.5 MÉTRICAS DE DESEMPENHO

Para além da capacidade de analisar em detalhe um segmento de código, o HpxTrace pode ser usado para coletar medições de desempenho que não são viáveis com os contadores do HPX.

Num cenário de execução em que cada localidade tem uma partição, o objetivo é medir e comparar os tempos de execução das iterações com o tempo de comunicação com as localidades vizinhas.

Cada partição tem um tamanho de 1 000 000. Não pode ser muito grande, caso contrário o impacto dos tempos de comunicação (que não dependem do tamanho) é insignificante.

O ambiente de execução consiste numa máquina com um processador com 6 cores. Devido à ausência de um ambiente com várias máquinas, foi adicionado uma pausa artificial para simular tempos de comunicação entre localidades. O método `get_data` do componente `partition_server` espera por 100 milissegundos antes de retornar os dados.

Com o código já instrumentado, o início de uma iteração é sinalizado pelo disparo de **start_heat** e o fim pelo disparo de **end_heat**. Uma partição por localidade significa que num dado momento apenas uma iteração está a

ser executada numa localidade. Com base nisso, o uso de variáveis escalares locais é adequado. Para obter o tempo médio por localidade, usa-se uma agregação *avg* com o *rank* da localidade como chave.

```
start_heat[] {
    :it_t = timestamp();
}

end_heat[] {
    @it_avg[locality_rank()] = avg(timestamp() - :it_t);
}
```

Código 8.12: Sondas para calcular o tempo médio das iterações

Uma alternativa sem o uso de sondas definidas pelo utilizador seria monitorizar o estado de execução das tarefas. Ou seja, o início seria o primeiro evento *start* da tarefa *heat_part_action*. O fim seria o evento *stop* de *heat_part_data* para a versão 6 do *stencil* ou de *border_heat_part* para a versão 7. Esta opção envolveria mais custos no desempenho e um *script* mais complexo (sempre que *heat_part_data* começasse a ser executado era preciso verificar se era a primeira vez) e seria na mesma necessário alterar o código fonte para anotar os nomes das tarefas.

A comunicação com uma localidade vizinha é iniciada com o envio de uma parcela da ação *get_data* e termina com a receção de uma parcela de uma ação do tipo *base_lco_with_value*. Em cada iteração existe comunicação simultânea com as duas localidades vizinhas. Para separar as medições é usada a variável *&target* no caso de um envio e *&source_rank* no caso de uma receção.

```
message[]::[send]::/&action == "get_data_action"/
{
    :msg_t[&target] = timestamp();
}

message[]::[receive]::/"base_lco_with_value" in &action && "partition_data" in &
    action/
{
    @msg_avg[&target, "->", &source_rank] = avg(timestamp() - :msg_t[&source_rank])
    ;
}
```

Código 8.13: Sondas para calcular o tempo médio a comunicar com cada localidade vizinha numa iteração

Um *script* com as quatro sondas mencionadas foi aplicado a uma execução da versão 6 do *stencil* e a uma execução da versão 7, ambas parametrizadas com 6 partições de tamanho 1000000 e 50 iterações. O tamanho das partições foi escolhido para que a melhoria da versão 7 seja perceptível, já que com um tamanho demasiado grande o aproveitamento do tempo de execução não seria significativo. Com o tamanho limitado, o tempo de execução é controlado com o número de iterações. O tempo de execução obtido é suficiente para demonstrar o ganho da otimização e para fornecer medições que não sejam demasiado afetadas por possíveis iterações anómalas.

Dos resultados obtidos, são apresentados os segmentos relacionados com a localidade 2. Os resultados completos podem ser consultados nos anexos [A.7](#) e [A.8](#)

```
Localities,OS_Threads,Execution_Time_sec,Points_per_Partition,Partitions,
    Time_Steps
6,      36,      47.210080743, 1000000,      6,      50
global_print_aggregation @it_avg
avg
(...)
2 : 665.68
(...)
global_print_aggregation @msg_avg
sum
(...)
0 -> 2 : 2.06
(...)
1 -> 2 : 134.58
2 -> 1 : 125.24
2 -> 3 : 155.64
3 -> 2 : 129.52
(...)
```

Código 8.14: Resultados parciais em milissegundos da versão 6 com o script de medição de desempenho

```
Localities,OS_Threads,Execution_Time_sec,Points_per_Partition,Partitions,
    Time_Steps
6,      36,      31.395991553, 1000000,      6,      50
global_print_aggregation @it_avg
avg
(...)
2 : 431.1
(...)
global_print_aggregation @msg_avg
sum
(...)
0 -> 2 : 4.04
(...)
1 -> 2 : 128.54
2 -> 1 : 131.02
2 -> 3 : 138.8
3 -> 2 : 141.5
(...)
```

Código 8.15: Resultados parciais em milissegundos da versão 7 com o script de medição de desempenho

Como esperado o tempo de execução da versão 7 é significativamente melhor. Graças à otimização adicionada, o tempo de execução da versão 7 é apenas 66% do tempo de execução da versão 6. Observando os tempos de

comunicações entre localidades, não existem variações significativas entre ambas as versões. Os pequenos valores no caso de pedidos da localidade 0 para as localidades 2, 3 e 4 correspondem a comunicações fora das iterações normais. Os tempos médios das iterações da versão 7 são aproximadamente 65% (430/665) dos da versão 6, o que é consistente com os tempos totais.

DISCUSSÃO E CONCLUSÕES

9.1 DISCUSSÃO

9.1.1 *Antecedentes*

O ponto de partida do projeto foi o estudo das ferramentas existentes para análises de execução e para a otimização do desempenho do sistema e das aplicações HPX. Como o HPX é uma plataforma inovadora e relativamente recente, procurou-se encontrar lacunas nas ferramentas existentes e oportunidades para a sua utilização em áreas onde uma contribuição fosse valiosa. Ao longo do projeto, este objetivo geral foi sendo detalhado e refinado conforme o domínio sobre o tópico aumentava.

9.1.2 *Fase preparatória*

Como preparação para o projeto, foi efetuada uma investigação sobre o debate entre programação orientada aos fios de execução e a programação orientada às tarefas, de forma a perceber qual a importância de plataformas como o HPX e porque vale a pena investir na aprendizagem e no desenvolvimento de ferramentas para este meio.

Feito isso, iniciou-se o estudo do HPX em si mesmo, com base na consulta da documentação oficial e dos exemplos disponíveis online. A elaboração de pequenos programas permitiu a testagem e a familiarização dos conceitos do HPX. Foram abordados conceitos base como a futurização, objetos de controlo local, sistema de endereçamento global, ações, componentes e parcelas. Foram também abordados os meios de medição de desempenho disponíveis nativamente, os contadores de desempenho. A abordagem inicial focou-se em como estes são utilizados e que métricas estão disponíveis.

9.1.3 *1ª fase*

Naturalmente, a primeira ferramenta a ser analisada em busca de potenciais contribuições foi a *framework* de contadores de desempenho do HPX. A grande variedade de contadores já existentes era maioritariamente orientada às localidades e os novos componentes que podiam ser definidos também estavam organizados por

omissão em termos de localidades. Ao considerar em que áreas os contadores poderiam ser expandidos, surgiu a ideia de tentar quebrar este molde e definir contadores com uma granularidade diferente.

A escolha de contadores associados a componentes deveu-se por várias razões. Os componentes são um elemento essencial para qualquer aplicação HPX com distribuição de dados, o que significa que qualquer ferramenta que os analise teria potencial para ser aplicada numa vasta gama de situações. Além disso, os contadores, tal como a maioria dos mecanismos no HPX, usam componentes como base. Finalmente, a probabilidade do emprego de contadores no desenvolvimento de novas ferramentas é considerável, pelo que o estudo dos componentes poderia ser um investimento relevante em várias vertentes,

O estudo dos componentes começou com a definição e utilização de componentes regulares e acabou com a definição de componentes com um ou dois parâmetros *template*. Componentes *templated* com mais de um parâmetro acabaram por não serem utilizados no resto do trabalho mas a secção que documenta a sua utilização acaba por ser uma pequena contribuição adicional, pois a escassa informação existente sobre esta possibilidade encontrava-se somente em *issues* no repositório GitHub do HPX.

Esta fase resultou num novo contador que, embora um pouco inortodoxo, é funcional e comprova que é viável o uso extensivo de templates na definição de contadores. Uma contribuição importante desta etapa foi o detalhamento do processo de definição de novos contadores, em particular em termos de como a nomenclatura pode ser flexibilizada.

9.1.4 2ª fase

A outra ferramenta estudada foi o APEX, devido à sua integração próxima do HPX. Ao examinar o APEX, surgiu a ideia de utilizar o mecanismo de políticas para implementar uma API de *tracing* semelhante à do Dtrace.

A documentação do APEX apresenta algumas falhas. Conteúdo vital, como alguma informação sobre dados fornecidos às políticas, estava ausente. Novamente isto teve que ser colmatado com exploração do código e com experimentação através de pequenos exemplos.

Uma decisão importante no trabalho foi que biblioteca devia ser usada para implementar o processador da linguagem da API. Tendo em vista que o foco do trabalho era a monitorização da plataforma HPX e não técnicas de processamento de texto, a biblioteca tinha que equilibrar vários critérios. Em primeiro lugar tinha que ser compatível com o código HPX e permitir a invocação de ações HPX. Depois, tinha que ser suficientemente poderosa para permitir programar em alto nível uma linguagem semelhante à linguagem "D" do Dtrace. Ao mesmo tempo, tinha que ser acessível, mesmo sem conhecimentos avançados de processamento de texto.

A biblioteca Spirit da coleção de bibliotecas Boost acabou por ser escolhida. Pode ser usada diretamente em código C++ (e consequentemente HPX) sem a necessidade de ferramentas adicionais de compilação ou de integração. O conceito de atributos simplificou a implementação de variáveis. O suporte das ações da biblioteca Boost Phoenix auxiliou na integração com o HPX ao permitir o *binding* de funções. Todos estes fatores apoiaram significativamente o desenvolvimento do HpxTrace.

A gramática do processador da linguagem e a arquitetura de dados foram desenvolvidos em paralelo, conforme surgiam novos requisitos para a linguagem. Embora algumas decisões da arquitetura de dados tenham sido

afetadas pela biblioteca usada, em princípio a biblioteca pode ser substituída por outra ferramenta que seja capaz de invocar as funções de interface com os dados.

Implementar uma ferramenta para monitorizar aplicações num ambiente paralelo e distribuído foi um desafio. Em termos de implementação, a solução foi incorporar o uso de componentes e ações. A existência de domínios de computação separados implicou a adição de variáveis globais o que por sua vez implicou a adição de mecanismos de sincronização global. Embora o uso frequente destes não seja encorajado, são mais uma ferramenta à disposição do utilizador que pode ser usada para operações consideradas críticas.

O aproveitamento do conceito de agregações do Dtrace e a sua adaptação para um meio multi-localidades foi uma mais-valia. As agregações transformam o que seria um conjunto de várias ações necessárias para conseguir certas formas de tratamento de dados em apenas uma instrução.

Como foi a norma ao longo do trabalho, para chegar a um entendimento de como ocorre a integração do HPX com o APEX foi preciso a pesquisa do código fonte de ambos. Neste caso o documento [PI et al. \(2019\)](#), embora já algo desatualizado devido ao desenvolvimento contínuo de ambas as ferramentas, serviu como uma base. Esta fase serviu não só para conseguir interpretar corretamente a informação fornecida pelo APEX como também para descobrir que informação adicional podia ser extraída e usada no HpxTrace.

A experimentação de versões iniciais do HpxTrace com pequenos casos de teste revelou que a informação fornecida pelo APEX às sondas das tarefas e das mensagens era bastante reduzida. Com as alterações ao código fonte do HPX e do APEX, estas sondas tornam-se mais potentes. Para além das métricas de desempenho, as novas variáveis facilitaram a associação entre disparos diferentes. As adições na sonda das tarefas permitiram a distinção entre tarefas com o mesmo nome e a associação com disparos de sondas definidas pelo utilizador, o que permitiu associar informação interna da plataforma HPX com informação do algoritmo. Na sonda das mensagens, a ausência do nome da ação era uma falha grave que foi colmatada. Uma intenção que não foi cumprida era encontrar um parâmetro que associasse garantidamente uma parcela a uma tarefa. A finalização da implementação do campo com o fio de execução fonte de uma parcela foi uma tentativa de alcançar esse fim mas como verificado não foi bem sucedida. Com o desenvolvimento contínuo do HPX, existe a possibilidade que no futuro esta opção seja mais viável.

Para terminar o desenvolvimento do projeto, foi necessário selecionar um caso de estudo, não só para validar as funcionalidades do HpxTrace mas também para detetar e corrigir *bugs* e para afinar pequenos detalhes de implementação. A existência de múltiplas versões do algoritmo *stencil* deu a liberdade de escolher uma otimização em que várias das capacidades funcionalidades do HpxTrace pudessem ser aplicadas. Uma vantagem de utilizar um exemplo já existente foi que mais tempo pode ser investido no desenvolvimento da ferramenta em si.

Para além do seu propósito original, o HpxTrace pode auxiliar no processo de aprendizagem do HPX, principalmente nos casos em que pequenos exemplos são elaborados e a complexidade é gradualmente adicionada. A cada passo, as sondas podem ser usadas para constatar o impacto de cada alteração, em áreas como a geração de tarefas ou a comunicação entre localidades.

O trabalho no projeto não chegou ao ponto de testar o HpxTrace num *cluster* com as versões alteradas do HPX e do APEX. Em princípio o HpxTrace deve funcionar sem problemas num ambiente com várias máquinas.

O código fonte do HpxTrace e alguns exemplos de exploração do HPX podem ser encontrados no repositório <https://github.com/JAPPinto/HpxTrace>.

9.2 CONCLUSÕES

O estudo do HPX demonstrou que este tem potencial num futuro em que a computação em *clusters* e orientada a tarefas é cada vez mais importante. Por outro lado, também demonstrou que é uma plataforma com falta de recursos introdutórios e de documentação, que são abundantes em plataformas mais estabelecidas. A mesma conclusão pode ser aplicada ao APEX. O facto de serem relativamente recentes e ainda estarem numa fase de desenvolvimento significativo apresentou os seus desafios. Nomeadamente, algumas ausências na documentação, código fonte com poucos comentários, funcionalidades por implementar e a existência ocasional de bugs. Eventualmente estas dificuldades foram ultrapassadas e o esforço despendido resultou em conteúdo que pode ser de uso para outros projetos que envolvam os tópicos abordados.

A primeira fase resultou num novo contador e em contribuições no processo de definição de novos contadores.

O foco do trabalho numa ferramenta do estilo Dtrace foi bem sucedido. Com o caso de estudo do *stencil*, foi comprovado que o HpxTrace apresenta utilidade tanto para analisar em detalhe o comportamento do HPX como também para efetuar medições de desempenho. Também apresenta valor educacional, sendo uma ferramenta que pode auxiliar o estudo do HPX, desvendando aspetos do funcionamento interno que são transparentes para o utilizador normal.

9.3 TRABALHO FUTURO

A principal tarefa de um projeto que procure expandir o trabalho desenvolvido seria a melhoria do processador da linguagem. Certamente, otimizar o processador de forma a tirar proveito de todas as funcionalidades da biblioteca Boost Spirit seria benéfico para a API. Um passo mais longe seria substituir totalmente a implementação em Boost Spirit por bibliotecas ou ferramentas que potencialmente sejam mais eficientes.

Com a aquisição de conhecimento sobre o APEX e a sua integração com o HPX, chegou-se ao ponto em que a remoção deste intermediário começou a ser uma possibilidade. Ao utilizar diretamente a API de eventos do HPX, a sobrecarga computacional e as latências seriam reduzidas.

Outra proposta seria estudar a viabilidade de usar, e provavelmente adaptar, o HpxTrace ao CoR-HPX, uma plataforma de computação orientada ao recursos que modela recursos usando componentes do HPX.

A linguagem das sondas do HpxTrace pode ser expandida. Podem ser adicionadas novas funcionalidades e melhorias que facilitem a sua utilização

- Adicionar novos tipos de dados à linguagem, em particular vários tipos numéricos (int, float, long, etc).
- Adicionar mais operadores à linguagem, incluído operadores *bitwise*, operadores de incrementação (++) ou de atribuição (+=).
- Suportar expressões condicionais (`x = i == 0 ? "zero": "non-zero";`).

- Definição de estruturas de dados simples.

A utilidade de manipular os valores das medições depende das fontes de dados disponíveis. Ao expandir as fontes de dados, também é expandida a quantidade de situações e problemas em que o HpxTrace é útil.

- Continuar a procura de mais informação útil que pode ser extraída do HPX.
- Mais variáveis *built-in*. Por exemplo, em que processador a localidade está a ser executada.
- Sondas que monitorizem mais métricas do sistema.

Como em qualquer outra ferramenta, existirão sempre melhorias miscelâneas a efetuar.

- Adição de *wildcards* nos argumentos das sondas. Por exemplo, na sonda das tarefas, uma *wildcard* poderia ser usada na filtração dos nomes das tarefas.
- Expandir o sistema de erros para incluir a sintaxe das sondas em si e não só os predicados e as ações. Por exemplo, notificar quando o utilizador não define todos os argumentos de uma sonda.
- Diversificar o controlo sobre o *output* da ferramenta. Por exemplo, escrever resultados em ficheiros diferentes, conforme a localidade.
- De forma semelhante às sondas dos contadores HPX, permitir controlar com a sonda *proc* a frequência das leituras da diretoria */proc*.

BIBLIOGRAFIA

- Aggregations, 2008. URL <https://illumos.org/books/dtrace/chp-aggs.html#chp-aggs>.
- Apex: Autonomic performance environment for exascale, 01 2021. URL http://www.nic.uoregon.edu/~khuck/apex_docs/doc/html/index.html.
- James O. Coplien. Curiously recurring template patterns. *C++ Rep.*, 7(2):24–27, feb 1995. ISSN 1040-6042.
- G. R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. ParalleX: A study of a new parallel computation model. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–6, 2007. doi: 10.1109/IPDPS.2007.370484.
- Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.
- STELLAR Group. Optimizing hpx applications, May 2022a. URL https://hpx-docs.stellar-group.org/latest/html/manual/optimizing_hpx_applications.html.
- STELLAR Group. Welcome to the hpx documentation!, May 2022b. URL <https://hpx-docs.stellar-group.org/latest/html/index.html>.
- STELLAR Group. Welcome to the hpx documentation!, May 2022c. URL https://hpx-docs.stellar-group.org/branches/master/html/why_hpx.html.
- Thomas Heller. *Extending the C++ Asynchronous Programming Model with the HPX Runtime System for Distributed Memory Computing*. PhD thesis, 05 2019.
- Thomas Heller, Hartmut Kaiser, Andreas Schäfer, and Dietmar Fey. Using hpx and libgeodecomp for scaling hpc applications on heterogeneous supercomputers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, Scala '13*, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450325080. doi: 10.1145/2530268.2530269. URL <https://doi.org/10.1145/2530268.2530269>.
- K.A. Huck, Allan Porterfield, Nicholas Chaimov, H. Kaiser, A.D. Malony, T. Sterling, and R. Fowler. An autonomic performance environment for exascale. *Supercomputing Frontiers and Innovations*, 2:49–66, 09 2015. doi: 10.14529/jsfi150305.

- H. Kaiser, M. Brodowicz, and T. Sterling. Parallelex an advanced parallel execution model for scaling-impaired applications. In *2009 International Conference on Parallel Processing Workshops*, pages 394–401, 2009. doi: 10.1109/ICPPW.2009.14.
- Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx – a task based programming model in a global address space. 10 2014. doi: 10.13140/2.1.2635.5204.
- 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>.
- Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710. Citeseer, 1999.
- Kevin A Huck PI, Allen D Malony, and Monil Mohammad Alaul Haque. Apex/hpx integration specification for phylanx. 2019.
- R. Raj and J. Chen. Performance analysis with hpx. 2014.
- Bibek Wagle, Mohammad Alaul Haque Monil, Kevin Huck, Allen Malony, Adrian Serio, and Hartmut Kaiser. Runtime adaptive task inlining on asynchronous multitasking runtime systems. pages 1–10, 08 2019. ISBN 978-1-4503-6295-5. doi: 10.1145/3337821.3337915.

ANEXOS

```

template <typename K, typename V>
class MapClient : public hpx::components::client_base<MapClient<K,V>, MapServer<K
,V>>{

    typedef hpx::components::client_base<MapClient<K,V>, MapServer<K,V>>
        base_type;
    typedef typename MapServer<K,V>::key_type key_type;
    typedef typename MapServer<K,V>::value_type value_type;

public:

    MapClient() {}
    MapClient(hpx::future<hpx::id_type> && id) : base_type(std::move(id)) {}
    MapClient(hpx::id_type && id) : base_type(std::move(id)) {}

    //Synchronous
    void store(key_type key, value_type value){
        HPX_ASSERT(this->get_id());
        typedef typename MapServer<K,V>::store_action action_type;
        action_type() (this->get_id(), key, value);
    }

    //Asynchronous
    hpx::future<void> store(hpx::launch::async_policy, key_type key,
        value_type value){
        HPX_ASSERT(this->get_id());
        typedef typename MapServer<K,V>::store_action action_type;
        return hpx::async<action_type>(hpx::launch::async, this->get_id(),
            key, value);
    }

    //Non-blocking
    hpx::future<void> store(hpx::launch::apply_policy, key_type key,
        value_type value){

```



```

        HPX_ASSERT(this->get_id());
        typedef typename MapServer<K,V>::store_action action_type;
        return hpx::apply<action_type>(this->get_id(), key, value);
    }

    //Synchronous
    hpx::util::optional<value_type> get(key_type key){
        HPX_ASSERT(this->get_id());
        typedef typename MapServer<K,V>::get_action action_type;
        return action_type()(this->get_id(), key);
    }

    //Asynchronous
    hpx::future<hpx::util::optional<value_type>> get(hpx::launch::
        async_policy, key_type key){
        HPX_ASSERT(this->get_id());
        typedef typename MapServer<K,V>::get_action action_type;
        return hpx::async<action_type>(hpx::launch::async, this->get_id(),
            key);
    }
};

```

Código A.1: Definição completa do cliente de um componente com dois parâmetros *template*

```

//A pointer to the task_identifier for this task_wrapper.
task_identifier * task_id;

//A pointer to the active profiler object timing this task.
profiler * prof;

//An internally generated GUID for this task.
uint64_t guid;

//An internally generated GUID for the parent task of this task.
uint64_t parent_guid;

//A managed pointer to the parent task_wrapper for this task.
std::shared_ptr<task_wrapper> parent;

//A node in the task tree representing this task type
dependency::Node* tree_node;

/*
    Internal usage, used to manage HPX direct actions when their
    parent task is yielded by the runtime.
*/
std::vector<profiler*> data_ptr;

/*
    If the task changes names after creation
    (due to the application of an annotation)
    then the alias becomes the new task_identifier for the task.
*/
task_identifier* alias;

```

Código A.2: Campos da classe `task_wrapper` implementada pelo APEX

```

private:
    task_identifier * task_id; // for counters, timers
public:
    std::shared_ptr<task_wrapper> tt_ptr;    // for timers
    uint64_t start_ns;
    uint64_t end_ns;
#if APEX_HAVE_PAPI
    long long papi_start_values[8];
    long long papi_stop_values[8];
#endif
    double allocations;
    double frees;
    double bytes_allocated;
    double bytes_freed;
    double value;
    double children_value;
    uint64_t guid;
    bool is_counter;
    bool is_resume; // for yield or resume
    reset_type is_reset;
    bool stopped;

```

Código A.3: Campos da classe `profiler` implementada pelo APEX

```

struct scoped_timer {
    explicit scoped_timer(std::shared_ptr<task_wrapper> data_ptr)
        : stopped(false), data_(nullptr)
    {
        // APEX internal actions are not timed. Otherwise, we would end
        // up with recursive timers. So it's possible to have a null
        // task wrapper pointer here.
        if (data_ptr != nullptr)
        {
            data_ = data_ptr;
            hpx::util::external_timer::start(data_);
        }
    }
    ~scoped_timer() {stop();}
    void stop()
    {
        if (!stopped)
        {
            stopped = true;
            // APEX internal actions are not timed. Otherwise, we would
            // end up with recursive timers. So it's possible to have a
            // null task wrapper pointer here.
            if (data_ != nullptr)
            {
                hpx::util::external_timer::stop(data_);
            }
        }
    }
    void yield()
    {
        if (!stopped)
        {
            stopped = true;
            // APEX internal actions are not timed. Otherwise, we would
            // end up with recursive timers. So it's possible to have a
            // null task wrapper pointer here.
            if (data_ != nullptr)
            {
                hpx::util::external_timer::yield(data_);
            }
        }
    }
    bool stopped;
    std::shared_ptr<task_wrapper> data_;
};

```

Código A.4: Classe `hpx::external_timer::scoped_timer`

```

start_heat[1]{
    print(str(timestamp()) + " START_HEAT " + &thread);
}
end_heat[1]{
    print(str(timestamp()) + " END_HEAT " + &thread);
}
received_data[1]{
    print(str(timestamp()) + " RECEIVED_DATA " + &thread);
}

task[1]::[start,stop,yield]::heat_part_action::
{
    print(str(timestamp()) + " TASK " + &name + " " + &event + " " + &thread);
}
task[1]::[start,stop,yield]::heat_part_data::
{
    print(str(timestamp()) + " TASK " + &name + " " + &event + " " + &thread);
}

message[1]::[receive]::/&action == "heat_part_action"/
{
    print(str(timestamp()) + " MESSAGE " + &action+ " " + &source_thread);
}

message[1]::[send]::/&action == "get_data_action"/
{
    print(str(timestamp()) + " MESSAGE " + &action + " " + &source_thread);
}

message[1]::[receive]::/"base_lco_with_value" in &action/
{
    print(str(timestamp()) + " MESSAGE received data " + &source_thread);
}

message[1]::[receive]::/"create_component" in &action/
{
    print(str(timestamp()) + " MESSAGE create_component " + &source_thread);
}

```

Código A.5: *Script* usado para analisar a execução da versão 6 do *stencil*

```

start_heat[1]{
    print(str(timestamp()) + " START_HEAT " + &thread);
}
end_heat[1]{
    print(str(timestamp()) + " END_HEAT " + &thread);
}
received_data[1]{
    print(str(timestamp()) + " RECEIVED_DATA " + &thread);
}
finished_middle[1]{
    print(str(timestamp()) + " FINISHED_MIDDLE " + &thread);
}

task[1]::[start,stop,yield]::heat_part_action::
{
    print(str(timestamp()) + " TASK " + &name + " " + &event + " " + &thread);
}
task[1]::[start,stop,yield]::middle_heat_part::
{
    print(str(timestamp()) + " TASK " + &name + " " + &event + " " + &thread);
}
task[1]::[start,stop,yield]::border_heat_part::
{
    print(str(timestamp()) + " TASK " + &name + " " + &event + " " + &thread);
}

message[1]::[receive]::/&action == "heat_part_action"/
{
    print(str(timestamp()) + " MESSAGE " + &action+ " " + &source_thread);
}
message[1]::[send]::/&action == "get_data_action"/
{
    print(str(timestamp()) + " MESSAGE " + &action + " " + &source_thread);
}

message[1]::[receive]::/"base_lco_with_value" in &action/
{
    print(str(timestamp()) + " MESSAGE received data " + &source_thread);
}
message[1]::[receive]::/"create_component" in &action/
{
    print(str(timestamp()) + " MESSAGE create_component " + &source_thread);
}

```

Código A.6: *Script* usado para analisar a execução da versão 7 do *stencil*

```

mpirun -np 6 ./1d_stencil_6 --nx=1000000 --np=6 --nt=50 --file times.txt
Initializing HpxTrace
HpxTrace Initialized
HpxTrace finalizing
HpxTrace finalized
Localities,OS_Threads,Execution_Time_sec,Points_per_Partition,Partitions,
    Time_Steps
6,      36,    47.210080743, 1000000,      6,      50
global_print_aggregation @it_avg
avg
0 : 692.8
1 : 701.06
2 : 665.68
3 : 672.72
4 : 663.96
5 : 654.26
global_print_aggregation @msg_avg
sum
0 -> 1 : 144.52
0 -> 2 : 2.06
0 -> 3 : 3.26
0 -> 4 : 3.8
0 -> 5 : 117.96
1 -> 0 : 139.52
1 -> 2 : 134.58
2 -> 1 : 125.24
2 -> 3 : 155.64
3 -> 2 : 129.52
3 -> 4 : 116.98
4 -> 3 : 143.76
4 -> 5 : 125.78
5 -> 0 : 159.38
5 -> 4 : 131.18

```

Código A.7: Resultados em milissegundos da versão 6 com o *script* de medição de desempenho

```

mpirun -np 6 ./1d_stencil_7 --nx=1000000 --np=6 --nt=50 --file times.txt
Initializing HpxTrace
HpxTrace Initialized
HpxTrace finalizing
HpxTrace finalized
Localities,OS_Threads,Execution_Time_sec,Points_per_Partition,Partitions,
    Time_Steps
6,      36,    31.395991553, 1000000,      6,      50
global_print_aggregation @it_avg
avg
0 : 434.64
1 : 409.26
2 : 431.1
3 : 430
4 : 423.3
5 : 422.68
global_print_aggregation @msg_avg
sum
0 -> 1 : 150.5
0 -> 2 : 4.04
0 -> 3 : 3.82
0 -> 4 : 2.6
0 -> 5 : 148.1
1 -> 0 : 136.68
1 -> 2 : 128.54
2 -> 1 : 131.02
2 -> 3 : 138.8
3 -> 2 : 141.5
3 -> 4 : 145.08
4 -> 3 : 139.18
4 -> 5 : 136.14
5 -> 0 : 138.2
5 -> 4 : 138.7

```

Código A.8: Resultados em milissegundos da versão 7 com o *script* de medição de desempenho

