PlaCoR *

Plataforma para a Computação orientada ao Recurso

Bruno Ribeiro[1] and António Pina[2]

[1] Departamento de Informática, UMinho
a73269@alunos.uminho.pt
[2] Laboratório de Instrumentação e Física Experimental de Partículas, UMinho
pina@di.uminho.pt

Resumo A Plataforma para a Computação orientada ao Recurso (PlaCoR) foi desenhada como um ambiente de programação e execução de aplicações baseadas no modelo da computação orientada ao recurso (CoR), integralmente escrito em C++ Moderno.
A plataforma possui facilidades para: i) comunicação inter-domínios, ii) passagem de mensagens entre recursos comunicantes, iii) memória partilhada distribuída (DSM), iv) ativação remota de fios de execução (RPC), v) criação e gestão de recursos e vi) gestão da consistência entre todas as réplicas de um recurso.
Atualmente, o desenho de aplicações CoR assenta nos recursos: domínio, grupo, clausura, agente, proto-agente, dado, barreira, guarda e guarda para leituras/escritas.
A avaliação da plataforma tomou como exemplo a leitura e processamento de eventos registados em TTree, recorrentemente usados na experiência ATLAS. O experimento confirmou a viabilidade da orientação ao recurso como paradigma de programação híbrido que integra múltiplos fios de execução e sincronização distribuída, com facilidades de comunicação de grão fino para a passagem de mensagens e de comunicação em contextos seguros, o acesso remoto a memória e a ativação remota de agentes.

Keywords: Orientação ao Recurso · Memória Partilhada Distribuída · Passagem de Mensagens · Grão Fino Paralelismo · Execução Remota.

1 Introdução

Atualmente acentua-se a tendência para a construção e utilização de sistemas de computação que recorrem, simultaneamente, a modelos de memória partilhada e de memória distribuída, tais como *clusters* de multiprocessadores de memória partilhada, interligados por redes de alta velocidade.

O CoR (Computação orientada ao Recurso) [7] é um paradigma de computação e de programação que recorre à definição do recurso como meio para explorar

* Supported by LIP - Laboratório de Instrumentação e Física Experimental de Partículas.

a computação paralela e distribuída de elevado desempenho. A particularidade deste modelo prende-se com a forma como se combinam mecanismos de estruturação/coordenação, com a programação com múltiplos fios de execução, com a memória partilhada distribuída e a comunicação por passagem de mensagens.

As aplicações construídas sobre o paradigma CoR assentam na gestão de um sistema de domínios distribuídos que identificam e representam os diferentes tipos de recursos, que representam entidades e conceitos externos ao modelo, tais como processos, fios de execução, interfaces de comunicação ou nós de computação.

O protótipo PlaCoR (Plataforma para a Computação orientada ao Recurso) [11] não pretende constituir-se como um competidor direto de outros ambientes que exploram o paralelismo e a distribuição dos dados, como é o caso das atuais implementações do MPI [8]. Assume, principalmente, como uma alternativa viável que utiliza a abstração recurso para intervir nos aspectos mais relevantes da computação paralela, tais como, a possibilidade de ajustar o grão de paralelismo e a distribuição dos dados, para tirar partido da capacidade computacional da infraestutura de computação subjacente.

1.1 Objetivos

Em termos gerais, o objetivo é criar um ambiente de suporte à criação e execução de aplicações distribuídas, com um elevado número de atividades paralelas e um grão fino de paralelismo e de comunicação, com distribuição de dados.

Um outro objetivo a alcançar é a construção de uma infraestrutura autónoma desenvolvida em C++ Moderno, multiplataforma - Linux, Solaris e MacOS -, com facilidades para: comunicação inter-domínios, passagem de mensagem entre recursos comunicantes, memória partilhada distribuída, ativação remota de agentes, criação e gestão de recursos, e capacidade de manter a consistência entre todas as réplicas de um recurso.

O objetivo final é a validação da plataforma, através da criação e execução de um número significativo de aplicações, num ambiente de domínios distribuídos, de forma a avaliar o poder expressivo da orientação ao recurso, na escrita de programas concorrentes.

2 Protótipo PlaCoR

Este projeto constitui-se como um ambiente de programação e execução de aplicações baseadas no modelo CoR [7] que dá continuidade a trabalhos anteriores, nomeadamente o pCoR [9]. O pCoR foi construído como uma biblioteca de funções C, suportado pelo PVM [5] e o DoTS [10] ao nível da infraestrutura. O PVM disponibiliza facilidades para a gestão de múltiplos nós de computação heterogéneos, enquanto o DoTS é uma camada lógica que combina a programação com múltiplos fios de execução, memória partilhada e passagem de mensagens, na linha do TPVM [3].

2.1 Apresentação

O PlaCoR, foi criado como uma biblioteca de classes em C++ Moderno. A escolha desta linguagem trouxe enormes vantagens para o desenvolvimento do protótipo, com destaque para o suporte à: programação orientada aos objetos - usada para a construção dos recursos assente em mecanismos de herança múltipla; programação genérica - por forma a abstrair na API as diferentes classes de recursos; programação concorrente - para tirar partido dos fios de execução e estruturas de sincronização nativas da própria linguagem.

2.2 Recursos

O comportamento dos recursos é integralmente dedutível das propriedades dos **elementos** (classes) que compõem a sua hierarquia, a saber: contentor, organizador dinâmico e estático, executor, caixa-postal, valor, barreira, guarda e guarda para leituras/escritas. Sendo cada um dos diferentes tipos de recursos definidos a partir da seleção das classes de elementos apropriadas, por forma a obter a funcionalidade requerida. A generalização do conceito de recursos implica o uso da classe **Recurso** como uma das suas classes de base.

Presentemente, a plataforma disponibiliza as classes de recursos: domínio, grupo, clausura, agente, proto-agente, dado, barreira, guarda e guarda para leituras/escritas.

É possível a existência de múltiplas réplicas por recurso, disseminadas pelo sistema de domínios distribuídos, no qual cada uma das quais possui um único objeto de consistência, usado para garantir a consistência entre todas as réplicas.

Identificação Em PlaCoR, os recursos possuem uma identificação global (**idp**) e uma identificação contextual, através de um identificador de membro (**idm**) e, opcionalmente, um nome no contexto do seu recurso ascendente. A criação de um recurso obriga à especificação do seu ascendente, obrigatoriamente um recurso estruturado, i.e., que possui um elemento do tipo organizador na sua hierarquia. No caso do ascendente ser remoto, a API inclui funcionalidades para a criação de réplicas locais, enquanto o sistema assegura automaticamente a consistência entre todas as réplicas, bem como a libertação automática das suas representações locais.

Árvore de dependências Em tempo de execução, cada aplicação organiza logicamente, numa árvore de dependências, todos os recursos através da associação entre o idp do recurso e o idp do respetivo ascendente. A criação de recursos estruturados introduz novos caminhos na árvore de dependências, onde novos recursos se podem pendurar.

Escalonamento de domínios O domínio estabelece uma correspondência direta com a plataforma de computação subjacente, já que traduz os módulos do utilizador em processos, estabelecendo o primeiro nível de concorrência/paralelismo. Os domínios são escalonados estaticamente no arranque das

aplicações através do comando apropriado, ou então dinamicamente através de um método que permite lançar novos domínios na aplicação, com o mesmo ou um novo módulo do utilizador, estando previsto o arranque de aplicações do tipo SPMD.

Infraestrutura de comunicação Presentemente, a plataforma apenas contempla primitivas de comunicação bloqueantes, entre recursos comunicantes de duas formas distintas: global através do identificador principal (idp); por contexto através de um identificador de membro (idm). Uma característica distintiva da passagem de mensagens em PlaCoR, é a possibilidade de as mensagens poderem incluir tipos de dados definidos pelo programador, suportadas pela biblioteca de serialização **cereal** [6].

2.3 Arquitetura da Plataforma

A arquitetura do protótipo PlaCoR, na figura 1, está centrada no *pod*[3]. O *pod*, enquanto núcleo do sistema, vive como um processo no sistema operativo, que se constitui em sub-sistemas independentes, organizados em camadas hierárquicas que suportam os recursos e a execução de aplicações.

De um modo geral, os programas são desenvolvidos com base na interface de programação (API) dos recursos, e o arranque das aplicações é realizado com ferramentas específicas. Há também a possibilidade de estender a plataforma com o desenho de novos recursos.

No ***pod*** coabitam os fios de execução do utilizador e de sistema, sendo os primeiros responsáveis pela execução dos programas do utilizador, enquanto os fios de sistema estão associados aos serviços de gestão do sistema de domínios distribuídos.

O **controlador** faz a interligação entre os vários domínios presentes na aplicação, através dos seguinte serviços:

- identificação global - geração dinâmica de identificadores;
- recursos distribuídos - gestão de réplicas e gestão de consistência das múltiplas representações dos recursos;
- sessões remotas - lançamento dinâmico de domínios em máquinas locais ou remotas.

O **correio** é responsável pelos mecanismos de comunicação inter/intra domínios que suportam os mecanismos de passagem de mensagens entre recursos comunicantes.

O serviço **rpc** está relacionado com a criação de recursos em domínios remotos, e a execução, espera e obtenção dos resultados produzidos pela ativação remota de recursos do tipo agente, i.e., que herdam o elemento executor na sua hierarquia de classes.

[3] o termo pod, recetáculo em português, tem origem num conceito usado em TPVM com um propósito semelhante.

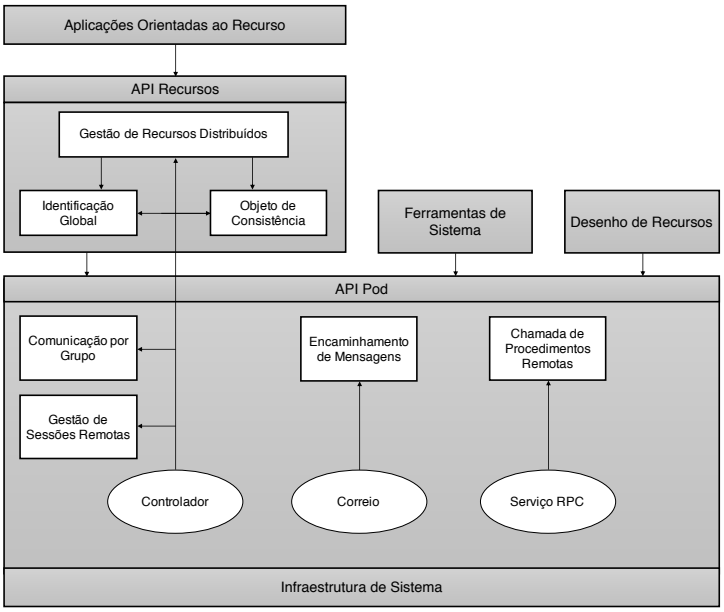

Figura 1. Arquitetura da Plataforma.

2.4 Ferramentas de desenvolvimento

O desenho da plataforma foi substancialmente influenciado pela necessidade de selecionar e adaptar diversos pacotes de *software* às necessidades de desenvolvimento, apresentados na figura 2.

Uma das questões centrais está relacionada com a obrigatoriedade de garantir a comunicação confiável entre os controladores. Dada a complexidade do tema e a dificuldade de desenho e projeto das facilidades necessárias, recorreu-se ao Spread [1,2] que suporta mecanismos de comunicação por grupo, comunicação *unicast* e *multicast*, e a garantia de ordem e da entrega das mensagens, mesmo no caso de falhas. Em conjunto com o Spread, usamos a biblioteca libssrcspread que fornece um conjunto de *bindings* C++ para o Spread, e a biblioteca libevent [4] para o processamento de eventos assíncronos.

Numa aplicação CoR, para tornar possível a instanciação de recursos remotamente e ativação remota de agentes, recorremos às facilidades da biblioteca czrpc [4] pelas seguintes razões: não necessita de código adicional para realizar chamadas remotas; em tempo de compilação, deteta argumentos inválidos; não tem dependências externas; a camada de transporte pode ser adaptada a bibliotecas de comunicação externas.

Num sistema de domínios distribuídos, outra das questões cruciais a resolver é a criação de réplicas locais de recursos. Para o efeito, recorremos à biblioteca

[4] https://libevent.org

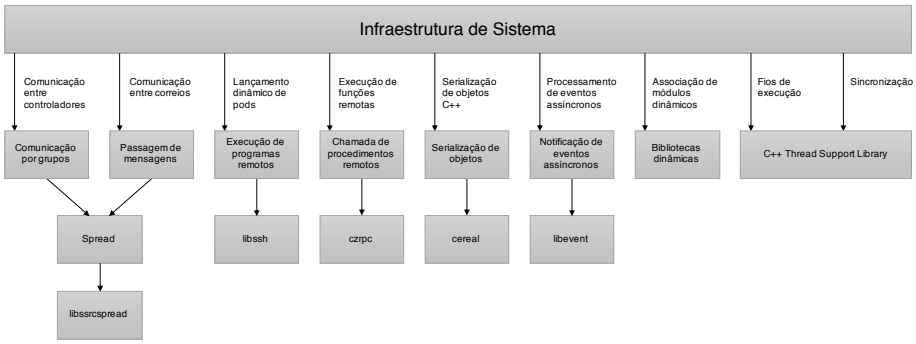

Figura 2. Ferramentas de desenvolvimento.

hearder-only cereal que dispõe de facilidades para a serialização de tipos de dados arbitrários e suporte à herança e polimorfismo. Estas características permitem ao programador serializar as suas próprias classes de dados, incluindo os tipos de dados da C++ Standard Library.

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

3 Arranque e execução de aplicações

Uma aplicação PlaCoR consiste num ou mais módulos escritos na linguagem C++, compilados sob a forma de bibliotecas dinâmicas e ligados à biblioteca PlaCoR. Um destes módulos é obrigatoriamente um módulo de arranque, que se carateriza por ter de definir uma função de entrada com o nome `Main`, com assinatura semelhante à função padrão `main` da linguagem C/C++. Os demais módulos a serem, eventualmente, incluídos na aplicação poderão ou não conter aquela função.

3.1 Arranque básico

A figura 3 mostra o código `module.cpp` de um módulo básico de arranque de uma aplicação que apresentamos, referindo os números de linhas do código. Em (1) é incluída a referência ao interface da biblioteca PlaCoR `"cor/cor.hpp"`, enquanto que em (2) é usada a diretiva `extern "C"`, necessária para permitir carregar dinamicamente a função de entrada do módulo (biblioteca dinâmica).

Em (4) é declarada a função de entrada do módulo, `Main`. Em (8) é usado o método `GetDomain` da API para obter um apontador para o domínio de arranque,

[5] https://www.libssh.org

presente na variável `domain`, usado para interagir com o sistema. Em (9) é obtido o identificador global do recurso que executou o método `GetActiveResourceIdp`, i.e. o idp do agente criado para executar a função de entrada do módulo de arranque.

```
1  #include "cor/cor.hpp"
2  extern "C"
3  {
4      void Main(int argc, char *argv[]);
5  }
6  void Main(int argc, char *argv[])
7  {
8      auto domain = cor::GetDomain();
9      std::cout << domain->GetActiveResourceIdp() << "\n";
10 }
```

Figura 3. Módulo de arranque.

3.2 Lançamento Estático de Aplicações

O código apresentado na figura 4 (`basic_operations.cpp`), mostra as operações básicas para a criação/interação de recursos e as diferentes alternativas existentes para obter os mesmos resultados. Notar que, por simplificação, o código não inclui as linhas (1-5) do exemplo anterior e que está implícita a existência de um outro módulo (`callable_module.cpp`), apresentado na figura 6, que irá ser carregado dinamicamente, para posteriormente ser integrado na aplicação original.

Em (3) é iniciada a variável `domain` com um apontador para o domínio local, obtido em (4,5) o idp do agente local e um apontador para o objeto que representa o recurso agente em execução, através do método `GetLocalResource` com a assinatura da função `Main` que está a executar. Segue-se o método `CreateLocal` para criar os recursos: `group` (6), `data` (7) e `new_agent` (8-10); o primeiro criado no contexto do domínio local e os restantes no contexto do grupo.

A criação dos recursos obriga a declarar o respetivo tipo (`Group`, `Data<idp_t>` e `Agent<idp_t(idp_t)>`, respetivamente). Enquanto, os argumentos incluem: i) idp do recurso ascendente (`domain->Idp()` e `group->Idp()`); ii) nome do recurso no contexto (`"group"`, `"data"` e `"agent"`); iii) argumentos usados na sua iniciação. O grupo necessita do nome do módulo (biblioteca dinâmica) a carregar, neste caso `"libcallable_module.so"`. O agente recebe o nome do módulo carregado pelo recurso grupo, obtido através do método `GetModuleName`, e o nome da função `"Test"` a carregar dinamicamente para execução. Por fim, o dado, é iniciado com um valor por omissão.

Na linha (11) o agente ativo chama o método `Run` do agente para executar a função `Test` com o argumento `agent_idp`, ficando em (12) à espera pelo término da função, obtendo o valor de retorno da mesma na linha (13) para a variável

```
1  void Main(int argc, char *argv[])
2  {
3      auto domain = cor::GetDomain();
4      auto agent_idp = domain->GetActiveResourceIdp();
5      auto agent = domain->GetLocalResource<cor::Agent<void(int,char**)>>(agent_idp);
6      auto group = domain->CreateLocal<cor::Group>(domain->Idp(), "group", "libcallable_module.so");
7      auto data = domain->CreateLocal<cor::Data<idp_t>>(group->Idp(), "data");
8      auto new_agent = domain->CreateLocal<cor::Agent<idp_t(idp_t)>>(
9          group->Idp(), "agent", group->GetModuleName(), "Test"
10     );
11     new_agent->Run(agent_idp);
12     new_agent->Wait();
13     auto res1 = new_agent->Get();
14
15     auto msg = agent->Receive();
16     auto res2 = msg.Get<idp_t>();
17
18     data->AcquireRead();
19     auto res3 = data->Get();
20     data->ReleaseRead();
21
22     std::cout << res1 << "\t" << res2 << "\t" << *res3 << "\n";
23 }
```

Figura 4. Código do exemplo `basic_operations.cpp`.

`res1`. Em (15) o agente ativo aguarda a receção de uma mensagem, via o método `Receive`, presumivelmente enviada pelo agente `new_agent`, e em (16) usa o método `Get` para obter o conteúdo da mensagem recebida, neste caso um identificador principal de um recurso guardado na variável `res2`. Em (18), o agente ativo requer o direito de leitura do recurso dado, com o método `AcquireRead`, obtendo um apontador para o valor através do método `Get` (16), guardado na variável `res3`, libertando posteriormente o direito de leitura em (20), através do método `ReleaseRead`. Por fim, em (22), são imprimidos os valores obtidos usando três formas diferentes disponíveis em CoR.

A árvore de dependências da aplicação, visível na figura 5, corresponde à criação: (a) do meta-domínio que representa a própria aplicação; (b) do domínio inicial e carregamento do módulo de arranque, para aceder às funções declaradas na diretiva `extern "C"`; (c) da clausura associada ao contexto de arranque da aplicação e (d) do agente que executa a função de entrada do módulo, no contexto da clausura. A clausura está intimamente ligada à criação de um contexto estático (fechado) que inclui apenas os domínios que arrancaram simultaneamente (neste caso apenas um), em associação com o agente inicial (d). Em (e) é criado o recurso grupo (`group`) no contexto do domínio (b) (`domain`), e a junção, em (f) e (g), dos recursos dado (`data`) e agente (`new_agent`) criados no contexto do grupo, respetivamente.

3.3 Módulo adicional

Na figura 6 é apresentado o código fonte do módulo `libcallable_module.so`, carregado pelo grupo no exemplo anterior. Este módulo disponibiliza a função `Test`, declarada (1-4) na diretiva `extern "C"`.

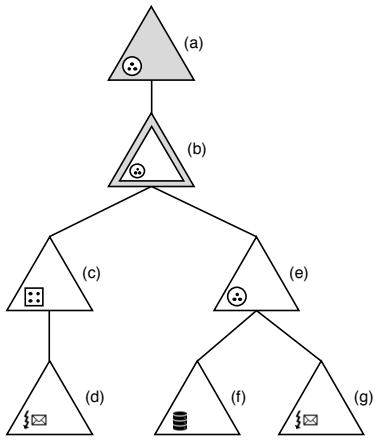

Figura 5. Árvore de dependências do exemplo `basic_perations.cpp`.

```
1  extern "C"
2  {
3      idp_t Test(idp_t rsc_idp);
4  }
5
6  idp_t Test(idp_t rsc_idp)
7  {
8      auto domain = cor::GetDomain();
9      auto agent_idp = domain->GetActiveResourceIdp();
10
11     auto group_idp = domain->GetPredecessorIdp(agent_idp);
12     auto group = domain->GetLocalResource<cor::Group>(group_idp);
13     auto data_idp = group->GetIdp("data");
14     auto data = domain->GetLocalResource<cor::Data<idp_t>>(data_idp);
15
16     data->AcquireWrite();
17     auto value = data->Get();
18     *value = agent_idp;
19     data->ReleaseWrite();
20
21     cor::Message msg;
22     msg.Add<idp_t>(agent_idp);
23     auto agent = domain->GetLocalResource<cor::Agent<idp_t(idp_t)>>(agent_idp);
24     agent->Send(rsc_idp, msg);
25
26     return agent_idp;
27 }
```

Figura 6. Código do exemplo `callable_module.cpp`.

Em (11), é obtido o idp do ascendente do agente ativo, neste caso, o grupo criado em `libbasic_operations.so`, com o método `GetPredecessorIdp` e em (12) o próprio objeto. Utilizando o método `GetIdp` com o nome atribuído a este (`"data"`), em (13) é obtido o idp do dado e um apontador para o objeto em (14).

Para garantir a exclusividade na modificação do dado, é usado o método `AcquireWrite` em (16), enquanto em (17-18) é obtido um apontador para o

respetivo valor e se procede à sua modificação localmente. Por fim, é feita a libertação do direito de escrita em (19), tendo como consequência a invalidação das possíveis réplicas existentes.

Seguidamente, em (21) é criado um objeto `msg`, do tipo `Message` ao qual é adicionado o conteúdo a enviar em (22) com o método `Add` que especifica o tipo dos dados `idp_t`. Em (23) obtém-se um apontador para o agente local que em (24) envia uma mensagem para o recurso com o identificador `rsc_idp`, que corresponde ao argumento de chamada da função `Test`. Em (25) o idp do agente local na variável `agent_idp` é usado como valor de retorno da função.

O comando `corx` usado para correr aplicações CoR pode ser usado para confirmar que o código produz o mesmo valor usando os três métodos apresentados.

```
$ corx app ctx 1 0 libbasic_operations.so
4294967035 4294967035 4294967035
```

3.4 Arranque paralelo

Introduzimos a seguir o tema do arranque paralelo de uma aplicação com mais do que um domínio que corresponde a uma infraestrutura formada por múltiplos *pods*. O exemplo da figura 7 (`parallel.cpp`) expõe os elementos necessários para a sua compreensão que assentam no recurso clausura.

```
1  void Main(int argc, char *argv[])
2  {
3    auto domain = cor::GetDomain();
4    auto agent_idp = domain->GetActiveResourceIdp();
5    auto clos_idp = domain->GetPredecessorIdp(agent_idp);
6    auto clos = domain->GetLocalResource<cor::Closure>(clos_idp);
7    auto clos_size = clos->GetTotalMembers();
8    auto rank = clos->GetIdm(agent_idp);
9    auto parent_idp = clos->GetParent();
10
11   std::cout << agent_idp << "\t" << rank << "\t" << clos_idp << "\t"
12     << clos_size << "\t" << parent_idp << "\n";
13 }
```

Figura 7. Código do exemplo `parallel.cpp`.

A necessidade de arrancar, em simultâneo, com múltiplas instância do mesmo módulo deu lugar à introdução do comando `corun`. Pode-se, assim, usar o comando `corun --hostfile nameOfFile --np 2 libparallel.so`, que implicitamente usa a ferramenta `corx`, para lançar `--np > 1` instâncias de `libparallel.so`, local/remotamente, de acordo com o conteúdo de `nameOfFile`.

3.5 Lançamento dinâmico de domínios

O lançamento dinâmico de domínios acrescenta à criação estática de aplicações, a possibilidade de, em tempo de execução, poder arrancar com múltiplas instâncias simples ou paralelas de módulos de bibliotecas dinâmicas que fazem parte integrante da aplicação inicial, através do método `Spawn` da API.

Na figura 8, o código `spawn.cpp` até à linha (9) é idêntico ao código anterior na figura 7. No caso paralelo, através do comando `corun` e `--np > 1` a aplicação é arrancada simultâneamente com várias instâncias de `libparallel.so`, enquanto neste caso de estudo, a aplicação é arrancada com `nReplicas` do mesmo módulo.

```
1  void Main(int argc, char *argv[])
2  {
3    auto domain = cor::GetDomain();
4    auto agent_idp = domain->GetActiveResourceIdp();
5    auto clos_idp = domain->GetPredecessorIdp(agent_idp);
6    auto clos = domain->GetLocalResource<cor::Closure>(clos_idp);
7    auto clos_size = clos->GetTotalMembers();
8    auto rank = clos->GetIdm(agent_idp);
9    auto parent_idp = clos->GetParent();
10
11   if (parent_idp == 0)
12     auto new_clos_idp = domain->Spawn("ctx2", nReplicas, "libspawn.so",
13         {}, { "localhost" });
14
15   std::cout << agent_idp << "\t" << rank << "\t" << clos_idp << "\t"
16         << clos_size << "\t" << parent_idp << "\n";
17 }
```

Figura 8. Código do exemplo `spawn.cpp`.

4 Discussão e Perspetivas Futuras

O PlaCoR constitui-se como uma nova abordagem indissociável do legado recebido dos desenvolvimentos anteriores, marcada pela escolha da linguagem C++ Moderno e de um conjunto significativo de outras bibliotecas e ferramentas. Esta perspetiva veio concretizar-se na definição do recurso, como um objeto, cujo comportamento é descrito por uma classe que herda, por múltipla herança, as funcionalidades das suas classes de base (elementos). Numa outra dimensão, procuramos também simplificar a API, de forma a criar e lidar com os recursos de forma genérica, o que se traduziu na exploração do paradigma da programação genérica, através dos *templates*.

A introdução do conceito de arranque estático paralelo e distribuído de aplicações traduziu-se na necessidade de adaptação da plataforma a esta exigência, que foi posteriormente estendida para contemplar o arranque dinâmico de domínios (*Spawn*). O modelo CoR pressupõe ainda a possibilidade de criar recursos remotamente, o que veio a ser concretizado através da introdução de um serviço rpc, enquanto que existência de múltiplas réplicas, disseminadas pelos vários domínios locais ou remotos, levanta a questão da serialização de objetos, compatível com um ambiente multiplataforma, em particular Linux, Solaris e MacOS.

Para a avaliação da plataforma, foi usado como exemplo a leitura e processamento de eventos, registados em coleções de TTree organizadas numa TChain, recorrentemente utilizados em física das altas energias (HEP), nomeadamente

na experiência ATLAS do CERN. O código e os resultados obtidos em termos de desempenho podem ser consultados em [11].

O experimento veio confirmar a viabilidade da orientação ao recurso como um paradigma de programação híbrido, que integra múltiplos fios de execução e sincronização distribuída, com facilidades de comunicação de grão fino por passagem de mensagens e de comunicação em contextos seguros, o acesso remoto a memória e a ativação remota de agentes .

4.1 Trabalho Futuro

A partir do trabalho já realizado temos previsto os seguintes desenvolvimentos:

- melhorar o suporte à criação de *pods* e à gestão dinâmica dos nós de computação alocados à aplicação;
- estudo e definição de novos elementos/recursos que contemplem modelos *thread-centric* e *task-centric*, num ambiente de memória partilhada e distribuída;
- novas facilidades para a interação com os recursos - primitivas de comunicação não bloqueantes e coletivas;
- criação do elemento porto - para tirar partido de transportes de comunicação de forma a aumentar o desempenho da passagem de mensagens;

Referências

1. Amir, Y., Stanton, J.: The spread wide area group communication system. Tech. rep. (1998)
2. Amir, Y., Danilov, C., Miskin-amir, M., Schultz, J., Stanton, J.: The spread toolkit: Architecture and performance (01 2004)
3. Ferrari, A., S. Sunderam, V.: Tpvm: Distributed concurrent computing with lighweight processes. pp. 211– (01 1995). https://doi.org/10.1109/HPDC.1995.518712
4. Figueira, R.: Modern c++ lightweight binary rpc framework without code generation. Tech. rep. (2016)
5. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.: PVM: Parallel Virtual Machine. A Users Guide and Tutorial for Networked Parallel Computing. Scientific and Engineering Computation, MIT Pres (1994)
6. Grant, W.S., Voorhies, R.: cereal - a c++11 library for serialization. Tech. rep. (2017), http://uscilab.github.io/cereal
7. Moreira, C.: CoRes - Computação orientada ao Recurso - uma especificação. Master's thesis, Universidade do Minho, Braga, Portugal (2001)
8. MPI-Forum: MPI: A Message-Passing Interface Standard. Version 3.1 (2015)
9. Pina, A., Oliveira, V., Moreira, C., Alves, A.: pCoR: a prototype for Resource oriented Computing. In: 7th International Conference on Applications of High-Performance Computers in Engineering (HPC '02). pp. 251–262. WIT Press (2002)
10. Pina, A., Moreira, C., Oliveira, V.: Domains, threads and shared memory in a message passing environment. Tech. rep., Universidade Minho, Braga, Portugal (1997)
11. Ribeiro, B.: PlaCoR - Plataforma para a Computação orientada ao Recurs . Master's thesis, Universidade do Minho, Braga, Portugal (2019)