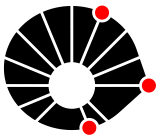

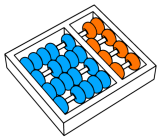

OpenMP Cluster Nextgen Plugin

J. Cléto *H. Hyviquel*

Relatório Técnico - IC-PFG-23-38
Projeto Final de Graduação
2023 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

OpenMP Cluster Nextgen Plugin

Jhonatan Cléto* Hervé Yviquel*

Resumo

Para atender às demandas de aplicações modernas, o OpenMP Cluster visa simplificar a programação em clusters de High Performance Computing, aproveitando sistemas multi-núcleo, multi-nó e aceleradores. Como uma extensão do OpenMP, ele melhora a gestão de tarefas e transferência de dados entre os nós de um cluster utilizando MPI e foi projetado para ser integrado ao LLVM/OpenMP com um plugin para a libomptarget. Este projeto apresenta uma atualização significativa do OpenMP Cluster, adequando-o à nova estrutura de plugins da libomptarget, modernizando sua implementação e proporcionando uma solução escalável para a execução de aplicações HPC em diversos ambientes computacionais.

1 Introdução

As aplicações científicas modernas necessitam de uma capacidade computacional cada vez maior. No entanto, o desempenho de chips de núcleo único é limitado devido às restrições de consumo de energia e dissipação de calor [7]. Assim, essas aplicações recorrem a sistemas multi-núcleo e multi-nó com aceleradores, como GPUs e FPGAs [10, 2]. Contudo, as aplicações para esses sistemas heterogêneos são complexas e não portáteis, pois utilizam diferentes modelos de programação para otimizar a utilização dos recursos do sistema [8, 9]. De fato, mais de 75% das aplicações de *High Performance Computing* (HPC) de código aberto combinam OpenMP, para programação multi-thread, com MPI, para tarefas de troca de mensagens [5].

A combinação de modelos de programação é comum em HPC, porém exige esforço e otimização [5]. O OpenMP facilita isso ao usar anotações para especificar paralelismo, tornando o código mais legível e fácil de escrever. Ele indica as partes do código que devem ser executadas em paralelo e suporta programação baseada em tarefas, permitindo a expressão de paralelismo através da definição de unidades de código e suas dependências. As tarefas são agendadas automaticamente e as versões mais recentes da biblioteca incluem diretrizes de offloading, permitindo a execução do código anotado em um acelerador, como uma GPU.

O OpenMP Cluster (OMPC) [11] é uma extensão do OpenMP implementada no LLVM, que introduz um novo dispositivo de offloading, um cluster de computação. Este plugin para a biblioteca libomptarget [1] automatiza a programação de tarefas OpenMP e a transferência de dados entre nós usando MPI.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

Neste projeto, a nossa proposta é atualizar a implementação do OpenMP Cluster, incorporando as mudanças da nova infraestrutura de plugins da libomptarget ao projeto. O intuito é tornar o plugin consonante com a nova arquitetura de modo a viabilizar a incorporação dele no repositório oficial do LLVM, visando oferecer a comunidade do LLVM/OpenMP uma estratégia que facilite a escalabilidade de uma aplicação, desde um protótipo em uma única máquina até um robusto cluster HPC.

2 Contexto

2.1 OpenMP

OpenMP [4] é uma API popular que facilita a programação paralela em linguagens como C, C++ e Fortran, especialmente em sistemas com múltiplos núcleos de processamento ou CPUs. Seu diferencial está na capacidade de permitir que os desenvolvedores paralelizem o código facilmente através de diretivas específicas, sem a necessidade de grandes alterações na estrutura do programa. Isso é feito inserindo comandos (diretivas e cláusulas) que orientam o compilador a executar certas partes do código em paralelo, simplificando o processo de aproveitar o poder de processamento disponível e melhorando o desempenho das aplicações. O OpenMP também fornece ferramentas para controlar a execução de threads e a sincronização de dados, tornando-o uma escolha eficiente para otimizar aplicações para ambientes de computação de alto desempenho.

Mesmo com todas essas facilidades, às vezes o OpenMP não é suficiente para atender as demandas computacionais das aplicações modernas, porque ele foi feito para trabalhar com memória que todos os núcleos ou threads podem acessar ao mesmo tempo. Quando o programa é muito grande ou complexo, pode ser necessário usar sistemas que conectam vários computadores juntos, cada um com sua própria memória, para dar conta do trabalho.

2.2 MPI

O padrão MPI (Message Passing Interface) [3] pode ser considerado uma alternativa viável para resolver problemas encontrados com o uso do OpenMP, definindo padrões que sistemas compostos por múltiplos nós, como clusters, devem seguir para permitir a troca de mensagens em um ambiente de memória distribuída. O MPI é responsável por determinar a interface, o protocolo e as regras semânticas necessárias para essa comunicação. Apesar de ser suscetível a falhas, onde um único nó defeituoso pode comprometer todo o processo, as implementações mais comuns do MPI são capazes de fornecer execuções confiáveis e de alto desempenho que se adaptam bem ao aumento de escala, o que explica sua ampla aceitação no setor industrial.

Contudo, o uso direto do MPI traz uma série de desafios e preocupações adicionais para os desenvolvedores. Eles precisam se atentar não apenas à resolução do problema em si, mas também à paralelização do código e à eficiência na troca de dados entre os nós. Isso significa que o programador precisa ter um conhecimento abrangente sobre todos esses aspectos. Como resultado, a complexidade aumentada pode tornar o programa mais difícil de manter e atualizar. Isso frequentemente resulta em uma utilização subótima dos recursos

de software e hardware disponíveis, limitando a flexibilidade do código. Uma solução mais adequada seria aquela que confia a gestão da troca de dados a uma biblioteca externa especializada, permitindo que o desenvolvedor se concentre exclusivamente nos aspectos cruciais da aplicação.

2.3 OpenMP Cluster

O OpenMP Cluster surge como uma solução promissora para atender às necessidades que nem o OpenMP nem o MPI conseguem suprir completamente. Esse modelo de programação paralela baseado em tarefas possibilita que os desenvolvedores utilizem a familiar sintaxe do OpenMP para distribuir automaticamente tarefas computacionais nos nós de um cluster, ampliando a aplicabilidade do OpenMP para sistemas com arquiteturas distribuídas.

Para facilitar a comunicação entre diferentes nós do cluster, o OMPC utiliza o MPI por trás das cenas, eliminando a necessidade do programador escrever código MPI diretamente. O sistema vai além, realizando o mapeamento automático de dados para as áreas de processamento e organizando as tarefas criadas para que sejam executadas no momento certo, sempre levando em conta as dependências entre tarefas estabelecidas pelo usuário. Com isso, o OMPC cuida de todo o escalonamento de tarefas, permitindo que o programador se concentre exclusivamente na definição das tarefas e no mapeamento eficiente dos dados.

Por exemplo, o código OpenMP no Listagem 1 não precisa ser alterado quando é executado em um cluster utilizando o runtime do OMPC. Neste cenário, as tarefas foo e bar são atribuídas aos nós do cluster e o vetor A é transferido entre os nós usando a implementação OMPC para a cláusula depend, que utiliza chamadas MPI eficientes para mover A de foo para bar.

```
void target_task(char *A, int N){
    #pragma omp target enter data map(to: A[:N]) nowait depend (out: *A)
    #pragma omp target nowait depend(inout: *A)
    foo(A)
    #pragma omp target nowait depend(inout: *A)
    bar(A)
    #pragma omp target exit data map(release: A[:N]) nowait depend(out: *A)
}
```

Listing 1: Exemplo de tarefa target do OpenMP.

A implementação do OMPC foi estruturada em abstrações que categorizam as funcionalidades de acordo com a natureza de cada operação. Por exemplo, as operações de comunicação são gerenciadas pelo *MPIManager*, que facilita a troca de mensagens entre o host e os nós do cluster.

Outro componente crucial do OMPC é o Sistema de Eventos, que simplifica a distribuição de tarefas e o gerenciamento de dados, aproveitando o paralelismo de tarefas e o offloading do OpenMP. Este sistema é composto por Eventos, uma *Gate Thread* e *Event Handlers*.

Os eventos são entidades lógicas que simbolizam uma sequência de mensagens MPI para ações como operações de memória e transferências de dados, com origens e destinos definidos para assegurar a comunicação MPI correta. A *Gate Thread* administra a fila de notificações de eventos para processamento, enquanto os *Event Handlers* são responsáveis por executar os eventos enfileirados, reenfileirando-os se necessário ou enviando notificações de conclusão ao processo de origem.

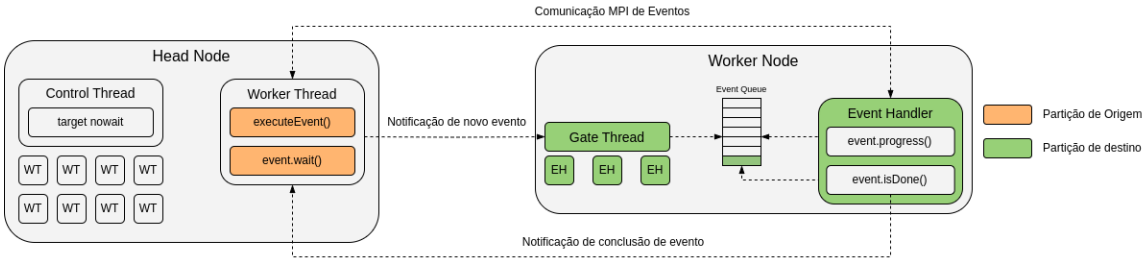

Figura 1: Arquitetura do Sistema de Eventos do OMPC.

Na Figura 1, ilustramos um *workflow* que envolve a criação de um evento por uma *worker thread*, seguida pelo enfileiramento pela *Gate Thread* e a execução pelo *Event Handler*, que pode reenfileirar o evento para operações de E/S pendentes. Após a conclusão de um evento, uma notificação é enviada ao processo de origem. Esse sistema assegura comunicações MPI eficientes e simultâneas, essenciais para o desempenho e escalabilidade do OMPC.

Além disso, o OMPC implementa a interface de plugins do libomptarget, convertendo operações em eventos gerenciados pelo *MPIManager* e pelo Sistema de Eventos. O OMPC também oferece funcionalidades adicionais para aprimorar a eficiência da comunicação entre nós, o escalonamento de tarefas e a robustez do ambiente de execução.

O *DataManager* (DM) é um componente vital do OMPC, otimizando o movimento de dados e mantendo a coerência em ambientes de computação distribuída. Ele gerencia o registro e a reutilização de buffers de dados, reduzindo transferências repetitivas entre o host e os dispositivos. Ao analisar dependências de tarefas, o DM coordena automaticamente o tráfego de dados entre os nós, assegurando que a versão mais atualizada esteja disponível onde e quando necessário. Isso é feito por meio de mapas de dados que rastreiam a localização e o estado de cada buffer, determinando o fluxo de dados com base nas necessidades de cada região-alvo.

Durante a execução do programa, o DM intercepta cláusulas target para encaminhar buffers aos nós apropriados antes da execução da tarefa e os recupera depois, atualizando os mapas de dados. Por exemplo, se um buffer é modificado, o DM assegura que a versão mais recente fique no nó que realizou a atualização, enquanto dados de leitura podem ser distribuídos para múltiplos nós. Esse sistema promove o compartilhamento eficiente de dados e reduz a sobrecarga de sincronização, acelerando a execução de aplicações OpenMP em ambientes de memória distribuída.

O funcionamento do DM está intimamente ligado ao fluxo de execução de tarefas no cluster. Ele analisa o gráfico de tarefas para determinar a localização inicial dos buffers de dados, enviando-os ao primeiro nó que os necessita. Conforme as tarefas são realizadas, o

DM ajusta a localização dos buffers com base em seu uso. Se um buffer é atualizado por uma tarefa em um nó, o DM garante que as tarefas subsequentes recebam os dados atualizados diretamente desse nó, evitando o nó principal para diminuir a latência e o tráfego de rede. Essa abordagem simplifica o gerenciamento de dados e aumenta a clareza do código, pois abstrai a complexidade do gerenciamento de dados do programador.

2.4 OpenMP Target Library

A libomptarget é uma biblioteca auxiliar, desenvolvida para implementar as funcionalidades de offloading de tarefas para aceleradores, introduzidas na versão 4.0 do OpenMP, no LLVM. Ela foi desenvolvida de forma modular, pensando na possibilidade de expansão futura.

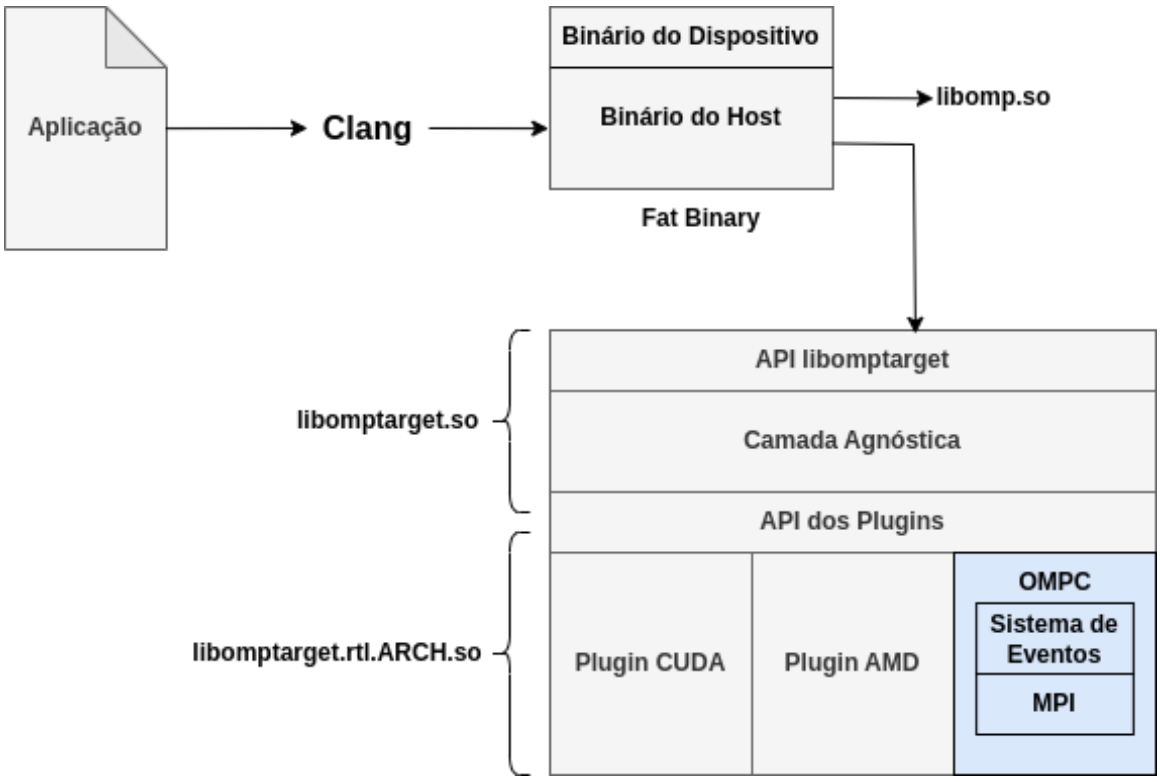

Figura 2: Estrutura de componentes da libomptarget.

Na Figura 2, apresentamos uma visão geral da arquitetura da libomptarget. No nível mais alto, a libomptarget fornece APIs síncronas e assíncronas que a aplicação pode utilizar para gerenciar operações de offloading. Durante a compilação, regiões target OpenMP são compiladas em imagens específicas do dispositivo para todas as arquiteturas suportadas, que são então incorporadas no binário da aplicação. Isso permite que um único binário (Fat binary) suporte múltiplos aceleradores. As diretivas de offloading OpenMP no código host são traduzidas pelo compilador em chamadas para rotinas de runtime do libomptarget, integrando-se de maneira transparente ao fluxo de compilação de offloading do LLVM/O-

penMP.

O componente agnóstico da libomptarget desempenha um papel crucial no offloading de regiões target OpenMP para uma variedade de dispositivos, como CPUs e aceleradores. Ele é projetado para ser independente de arquitetura, oferecendo uma interface universal para tarefas de offloading sem estar restrito a qualquer hardware específico.

A infraestrutura de offloading do LLVM/OpenMP inclui plugins de runtime dependentes da arquitetura que são essenciais para gerenciar a execução de tarefas em diferentes aceleradores de hardware, como GPUs. Esses plugins, chamados *libomptarget.rtl.ARCH*, são personalizados para várias arquiteturas e realizam operações de baixo nível essenciais, como inicialização do dispositivo, gerenciamento de memória, execução de kernels e sincronização. Eles aderem a uma API padronizada, permitindo que o runtime independente do target interaja de maneira uniforme com qualquer dispositivo suportado.

Esses plugins também atuam como intermediários entre as chamadas genéricas de offloading e os runtimes nativos do dispositivo, garantindo que o código compilado para regiões target, embutido no binário do host, seja executado corretamente no dispositivo.

2.5 Primeira Interface de Plugins

Inicialmente a interface dos plugins da libpomptarget era definida unicamente por meio de um conjunto de funções que todos os plugins deveriam implementar para garantir o correto funcionamento de toda a especificação target do OpenMP. Nessa primeira implementação, era responsabilidade do desenvolvedor do plugin implementar as abstrações e funcionalidades necessárias para que os dispositivos associados ao plugin atendessem todos os serviços expostos pela API. As principais operações implementadas pelos plugins da libomptarget são descritas a seguir:

1. **Validação de Binários**: Verifica se o código binário fornecido é compatível com o dispositivo de destino.

2. **Gerenciamento de Dispositivos**: Inicializa dispositivos de destino, recupera o número de dispositivos disponíveis e inicia os dispositivos com base em uma descrição binária.

3. **Alocação e Gerenciamento de Memória**: Aloca e libera memória no dispositivo de destino, envia e recupera dados entre o host e o dispositivo, e realiza troca de dados entre dispositivos.

4. **Execução de Código**: Executa regiões de código no dispositivo de destino, tanto de forma síncrona quanto assíncrona, com suporte a especificação de número de equipes e limite de threads.

5. **Sincronização**: Sincroniza o dispositivo de destino para garantir que todas as operações assíncronas sejam concluídas e gerencia eventos para controle de execução assíncrona.

2.6 Nova Interface dos Plugins

Na versão original da libomptarget, cada vez que um novo plugin era criado, as operações e abstrações comuns eram re-implementadas, o que resultava em redundância e esforço duplicado. Para resolver essa questão, os desenvolvedores da libomptarget no projeto LLVM reformularam a infraestrutura dos plugins.

A API dos plugins passou por um processo de modernização, com a implementação baseada em um sistema de classes que estabelece padrões para abstrações e operações dentro da biblioteca. Algumas classes novas, que são aplicáveis a todos os plugins, já estão disponíveis para uso pelos plugins específicos. Além dessas funcionalidades que são compartilhadas, a semântica dos plugins e dos dispositivos foi organizada em classes genéricas, o que simplifica a integração de novos plugins. As classes genéricas mais importantes incluem:

- **GenericPluginTy**: Define as operações e atributos essenciais para todos os plugins, garantindo a conformidade com a especificação do OpenMP target.
- **GenericDeviceTy**: Serve como uma abstração para os aceleradores, implementando funcionalidades comuns a todos os aceleradores.
- **GenericKernelTy**: Fornece implementações comuns para transferência e execução de kernels nos aceleradores.
- **DeviceImageTy**: Encarregado de envolver e administrar as imagens dos dispositivos, as quais consistem em representações de código compilado e informações que têm a capacidade de ser transferidas para os aceleradores.

Com base nessas classes, cada plugin específico deve estender e implementar os métodos virtuais de acordo com suas particularidades. Essa nova estrutura simplifica o desenvolvimento de novos plugins, pois muitas funcionalidades já estão incorporadas ao núcleo comum da biblioteca.

3 Novo Plugin MPI

O OMPC foi desenvolvido com base na implementação original da libomptarget e é mantido em um repositório bifurcado [6], distinto da linha principal do LLVM. Contudo, almejamos torná-lo acessível à comunidade no repositório oficial do LLVM. Portanto, o propósito deste trabalho é aprimorar a implementação do OMPC na libomptarget, de forma a viabilizar sua integração ao conjunto de plugins oficiais da biblioteca.

Essa atualização requer uma série de alterações significativas, pois, para facilitar o offloading de tarefas em um cluster HPC, implementamos modificações na interface prévia da libomptarget, permitindo assim a comunicação entre a aplicação host e os nós remotos. Com o intuito de incorporar nosso plugin à biblioteca principal, foi necessário adaptar nossa infraestrutura à nova arquitetura baseada em classes dos plugins da libomptarget, o que demandou inúmeros refatoramentos e a reescrita de partes fundamentais do OMPC.

Agora, o OMPC é implementado como um plugin MPI, aproveitando as abstrações da nova arquitetura da libomptarget. Utilizamos o *GenELF64Plugin*, que suporta arquiteturas

de CPU 64 bits, como base para o *MPIPlugin*, que facilita o uso de CPUs remotas via comunicação MPI. Introduzimos classes como *MPIPluginTy*, *MPIDeviceTy*,*MPIDeviceImageTy* e *MPIKernelTy*, além de outras auxiliares, para adaptar o OMPC à nova interface.

Uma mudança significativa da nova implementação foi a separação do runtime em dois binários distintos: um dedicado à lógica completa do plugin para o nó host e outro projetado para as funcionalidades de execução de tarefas nos nós remotos, conhecidos como workers. Essa divisão é estratégica, pois os workers são responsáveis apenas por executar as tarefas que lhes são atribuídas. Isso envolve a comunicação com o nó host para transferência de dados, notificações de conclusão de tarefas e processos de sincronização. Como resultado, o binário destinado aos workers é mais simplificado do que o do nó host, já que não precisa incorporar todas as funcionalidades necessárias para a operação integral do plugin.

O binário dos workers foca exclusivamente nas funcionalidades do sistema de eventos, como a *Gate Thread* e os *Event Handlers*. Isso permite que os workers recebam notificações de novos eventos e procedam com a execução correspondente. Na Listagem 2, apresentamos um exemplo de como poderia ser a implementação do código nos nós remotos. É importante notar que a implementação se concentra no uso dos recursos do sistema de eventos do OMPC.

```
#include "EventSystem.h"

int main() {
  EventSystemTy EventSystem;

  EventSystem.initialize();

  EventSystem.runGateThread();

  EventSystem.deinitialize();

  return 0;
}
```

Listing 2: Implementação do código dos nós remotos no OMPC.

3.1 MPIPluginTy

A classe*MPIPluginTy* desempenha um papel central na nova implementação do OMPC, atuando como o gerenciador dos dispositivos MPI. Ela é projetada para interagir com os nós de um cluster HPC, que são tratados como dispositivos de offload no contexto do OpenMP. Mantivemos o sistema de eventos do OMPC, incluindo um atributo para uma instância da classe EventSystem no plugin. Assim, ao construir o plugin, também inicializamos o sistema de eventos e, consequentemente, os mecanismos de comunicação MPI.

Além de gerenciar os dispositivos, a *MPIPluginTy* é responsável por iniciar e gerenciar

a comunicação entre o host e os dispositivos MPI. Isso envolve o estabelecimento de sessões MPI, o envio e recebimento de dados e a coordenação da execução das tarefas de offload.

No construtor da *MPIPluginTy*, além de inicializar o sistema de eventos, a classe configura o ambiente MPI necessário para a execução distribuída. Isso inclui a inicialização da biblioteca MPI, a identificação do número de processos disponíveis e a configuração de qualquer parâmetro específico necessário para a comunicação MPI.

Outra responsabilidade da *MPIPluginTy* é a criação de instâncias das classes *MPIDeviceTy*, *MPIDeviceImageTy* e *MPIKernelTy*, que são especializações das classes genéricas para o contexto MPI. Essas instâncias são usadas para representar e gerenciar os kernels de offload, as imagens de dispositivos e os próprios dispositivos dentro do ambiente MPI.

3.2 MPIDeviceTy

A classe *MPIDeviceTy* é uma extensão da classe genérica *GenericDeviceTy* e é especificamente adaptada para o contexto de um ambiente MPI no plugin OMPC. Ela representa um dispositivo de offload individual dentro de um cluster HPC, que, neste caso, é um nó de processamento ou um processo MPI. A *MPIDeviceTy* encapsula as funcionalidades e informações necessárias para interagir com esse nó de processamento, permitindo a execução de tarefas de offload de maneira eficiente. As principais responsabilidades e características da *MPIDeviceTy* incluem:

1. **Identificação do Dispositivo**: A classe armazena informações de identificação para cada dispositivo MPI, como o rank do processo MPI no comunicador global. Isso é essencial para direcionar as operações de offload para o nó correto.

2. **Gerenciamento de Memória**: A *MPIDeviceTy* é responsável por gerenciar a alocação e liberação de memória no dispositivo de offload. Isso inclui a transferência de dados entre a memória do host e a memória do dispositivo, o que é crítico para a execução de tarefas de offload.

3. **Execução de Kernels**: A classe fornece mecanismos para lançar kernels de offload no dispositivo MPI. Isso envolve a preparação dos argumentos do kernel, a configuração do ambiente de execução e a comunicação com o nó de processamento para iniciar a execução do kernel.

4. **Sincronização**: A *MPIDeviceTy* implementa métodos para sincronizar a execução de tarefas entre o host e o dispositivo. Isso garante que as tarefas de offload sejam concluídas antes que o controle seja retornado ao programa principal no host ou que outras tarefas dependentes sejam iniciadas.

5. **Comunicação MPI**: A classe encapsula as chamadas de comunicação MPI necessárias para interagir com os dispositivos de offload. Isso inclui o envio de comandos, a transferência de dados e a recepção de resultados das tarefas de offload. Para isso, A *MPIDeviceTy*, transforma essas operações em eventos do sistema de eventos do OMPC, reduzindo a complexidade da comunicação entre os nós do cluster, abstraídos como dispositivos no plugin.

Ao encapsular a complexidade da comunicação e gerenciamento de dispositivos MPI, a *MPIDeviceTy* desempenha um papel fundamental na abstração do cluster HPC como um conjunto de dispositivos de offload, permitindo que os desenvolvedores de aplicativos OpenMP se concentrem na lógica de programação paralela sem se preocupar com os detalhes de baixo nível da computação distribuída.

3.3 MPIKernelTy

A classe *MPIKernelTy* é uma especialização da classe genérica *GenericKernelTy*, projetada para atender às necessidades específicas de execução de kernels de offload em um ambiente MPI dentro do plugin OMPC. Esta classe é responsável por encapsular e gerenciar a execução de kernels de computação paralela nos aceleradores, que, no contexto do OMPC, são os nós um cluster HPC. As principais funções e características da *MPIKernelTy* incluem:

1. **Representação de Kernels**: A MPIKernelTy representa um kernel de offload, que é um bloco de código compilado para ser executado em um acelerador. A classe armazena informações sobre o kernel, como seu identificador e os argumentos necessários para sua execução.

2. **Preparação de Argumentos**: Antes de um kernel ser executado, é necessário preparar seus argumentos. A MPIKernelTy é responsável por organizar os argumentos do kernel, garantindo que eles estejam no formato correto e prontos para serem transferidos para o dispositivo.

3. **Lançamento de Kernels**: A classe fornece métodos para iniciar a execução de kernels nos dispositivos MPI. Isso envolve a comunicação com o nó de processamento apropriado e o envio de instruções para que o kernel seja executado com os argumentos fornecidos.

4. **Integração com o Sistema de Eventos**: A classe trabalha em conjunto com o sistema de eventos do OMPC para notificar o host sobre o início e a conclusão da execução do kernel.

5. **Comunicação com o MPIDeviceTy**: A *MPIKernelTy* interage estreitamente com a classe *MPIDeviceTy* para garantir que os kernels sejam executados no dispositivo de offload correto e que a comunicação entre o kernel e o dispositivo seja gerenciada de forma eficaz.

Através da abstração fornecida pela *MPIKernelTy*, os desenvolvedores podem se concentrar na lógica dos kernels de offload sem se preocupar com os detalhes intrincados da execução de código em um ambiente de computação distribuída. Isso simplifica o processo de desenvolvimento e manutenção de aplicações paralelas e distribuídas, permitindo que os desenvolvedores aproveitem a potência do MPI para offloading em clusters HPC com a facilidade de uso do OpenMP.

3.4 MPIDeviceImageTy

A classe *MPIDeviceImageTy* é uma especialização da classe *DeviceImageTy*, adaptada para o contexto de execução de offload em um ambiente MPI no plugin OMPC. Esta classe tem a função de encapsular e gerenciar as imagens de dispositivos, que são representações de código compilado e dados que podem ser *offloaded* para os dispositivos de computação remotos, ou seja, os nós de um cluster HPC. A *MPIDeviceImageTy* é responsável por armazenar informações sobre as imagens, como metadados, e por gerenciar o ciclo de vida das imagens no dispositivo, incluindo o carregamento e descarregamento das mesmas. Ela também lida com a localização e o acesso às entradas de offload específicas dentro da imagem, o que é crucial para a execução correta dos kernels. Ao fornecer uma interface clara para o gerenciamento de imagens de dispositivos, a *MPIDeviceImageTy* facilita a complexa tarefa de transferir e manter o código executável e os dados associados entre o host e os dispositivos de offload em um ambiente de computação distribuída.

4 Desafios na Implementação do novo Plugin MPI

A transição do antigo plugin OMPC para a nova arquitetura de classes do plugin MPI apresentou vários desafios técnicos e exigiu adaptações significativas no código. A seguir, são detalhados alguns dos principais desafios encontrados e as estratégias adotadas para superá-los:

1. **Integração com a Nova Arquitetura**: O antigo OMPC foi construído com uma arquitetura que não estava alinhada com as novas classes genéricas da libomptarget. A integração exigiu um mapeamento cuidadoso das funcionalidades existentes para as novas classes, como *MPIPluginTy*, *MPIDeviceTy*, *MPIKernelTy* e *MPIDeviceImageTy*. Isso envolveu a reestruturação do código para se adequar ao novo modelo orientado a objetos e garantir a compatibilidade com as interfaces genéricas. Nessa atualização do plugin do OMPC, focamos em portar as funcionalidades essenciais para o funcionamento do runtime. Nesse sentido, features como mecanismos de tolerância a falhas e métodos para broadcasting de dados ficaram de fora dessa nova implementação.

2. **Gerenciamento de Comunicação MPI**: O *MPIManager* do OMPC antigo era responsável por toda a comunicação MPI. Na nova implementação, essa funcionalidade precisou ser distribuída entre as classes *MPIPluginTy* e *MPIDeviceTy*, requerendo uma redefinição de responsabilidades e a implementação de mecanismos de comunicação eficientes dentro de cada classe.

3. **Sistema de Eventos**: O Sistema de Eventos era central para o OMPC, gerenciando a execução assíncrona e a comunicação. A nova implementação precisou incorporar esse sistema dentro da estrutura das novas classes, mantendo a eficiência e a capacidade de resposta do plugin. Isso significou integrar o sistema de eventos de forma a trabalhar harmoniosamente com o ciclo de vida e os eventos das classes genéricas.

4. **DataManager (DM)**: O DM desempenhava um papel essencial na otimização do fluxo de dados e na preservação da consistência. Com a nova implementação, foi necessário distribuir as funções do DM entre as diferentes classes da recém-desenvolvida arquitetura de plugins. Por exemplo, parte da responsabilidade pela transferência de dados entre o host e os dispositivos foi atribuída à classe *MPIDeviceTy*. Esse processo exigiu a reescrita do código para aproveitar as novas abstrações de imagem de dispositivo e para se integrar ao sistema de eventos. Contudo, nem todas as funcionalidades do DM foram transferidas para a nova implementação, o que pode resultar em uma performance inferior em comparação com a implementação anterior. Isso ocorre porque o gerenciamento de dados na versão atual não possui todas as capacidades que estavam presentes na versão antiga.

5. **Divisão do Runtime**: A decisão de dividir o runtime em dois binários distintos trouxe desafios adicionais, como a necessidade de garantir a comunicação e a coordenação eficazes entre os binários. Isso exigiu uma abordagem cuidadosa para o design do sistema de inicialização e a gestão do ciclo de vida dos processos.

5 Conclusão e Trabalhos Futuros

A implementação do novo plugin MPI para o OMPC representa um avanço significativo na integração do modelo de programação OpenMP com ambientes de computação distribuída baseados em MPI. A reestruturação do código em novas abstrações de classes, como *MPIPluginTy*, *MPIDeviceTy*, *MPIKernelTy* e *MPIDeviceImageTy*, oferece uma base sólida para a execução eficiente de tarefas de offload em clusters HPC. A adaptação do DataManager (DM) e a integração do Sistema de Eventos dentro dessa nova arquitetura são passos importantes para otimizar o movimento de dados e manter a coerência em ambientes de memória distribuída.

Apesar desses avanços, o projeto enfrenta desafios no mapeamento correto dos endereços das funções entre o host e os dispositivos. Este problema é crítico, pois afeta a capacidade do sistema de executar corretamente os kernels de offload, que são essenciais para o aproveitamento do paralelismo em clusters HPC.

Considerando o estado atual da implementação do novo plugin do OMPC, identificamos os seguintes trabalhos futuros necessários para que o runtime alcance plenamente os objetivos propostos:

1. **Resolução de Mapeamento de Endereços**: O foco imediato é resolver os problemas de mapeamento de endereços das funções. Isso pode envolver a investigação e a implementação de mecanismos de mapeamento de memória mais robustos, garantindo que os endereços das funções no host correspondam aos endereços nos dispositivos.

2. **Testes Extensivos**: Será necessário realizar uma bateria de testes mais abrangente para garantir a compatibilidade do plugin com uma variedade de configurações de cluster e para identificar quaisquer problemas residuais de estabilidade ou desempenho.

3. **Permitir que o plugin MPI utilize outros aceleradores**: Explorar a integração do plugin MPI com outros aceleradores, como GPUs, pode expandir ainda mais o alcance e a utilidade do OMPC. Um dos objetivos dessa atualização do plugin é num futuro próximo transformá-lo em um plugin proxy. Isso significa que o OMPC se tornaria uma camada de comunicação para permitir o acesso a todos os recursos em nós remotos no cluster HPC, de modo a permitir a construção de aplicações OpenMP com offload de tarefas para outros aceleradores remotos além das CPUs dos outros nós e num futuro mais distante, permitir a construção de aplicações heterogêneas que conseguem fazer uso de todos os recursos dos nós do cluster, por exemplo fazer o offloading de tarefas para os diferentes tipos de aceleradores presentes no cluster.

Por fim, embora a nova implementação do plugin MPI ainda enfrente desafios técnicos, o progresso alcançado até o momento é promissor. Com a resolução dos problemas de mapeamento de endereços e a continuação dos esforços de otimização e testes, o projeto tem o potencial de se tornar uma ferramenta valiosa para a comunidade de computação de alto desempenho, permitindo que aplicações OpenMP sejam executadas de forma eficiente em clusters HPC.

Referências

[1] Samuel F. Antao, Alexey Bataev, Arpith C. Jacob, Gheorghe Teodor Bercea, Alexandre E. Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, Tong Chen, Hyojin Sung, Carlo Bertolli, and Kevin O'Brien. Offloading support for Openmp in clang and LLVM. *Proceedings of LLVM-HPC 2016: The 3rd Workshop on the LLVM Compiler Infrastructure in HPC - Held in conjunction with SC 2016: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2017.

[2] Sanjay Chatterjee, Sağnak Taşirlar, Zoran Budimlić, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. Integrating asynchronous task parallelism with MPI. *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*, pages 712–725, 2013.

[3] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI message passing interface standard. In *Programming Environments for Massively Parallel Distributed Systems*, pages 213–218. Birkhäuser Basel, 1994.

[4] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[5] Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. A large-scale study of MPI usage in open-source HPC applications. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2019.

[6] OmpCluster. Repositório do openmp cluster. https://ompcluster.gitlab.io/. Acesso: 01-12-2023.

[7] Cherri M. Pancake. Is parallelism for you? *IEEE computational science & engineering*, 3(2):18–37, 1996.

[8] Cherri M. Pancake. What computational scientists and engineers should know about parallelism and performance. *Computer Applications in Engineering Education*, 4(2):145–160, 1996.

[9] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading. *ACM SIGARCH Computer Architecture News*, 23(2):392–403, 1995.

[10] Oreste Villa, Daniel R. Johnson, Mike O'Connor, Evgeny Bolotin, David Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero, Stephen W. Keckler, and William J. Dally. Scaling the Power Wall: A Path to Exascale. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2015-Janua(January):830–841, 2014.

[11] Hervé Yviquel, Marcio Machado Pereira, Emílio Francesquini, Guilherme Valarini, Gustavo Leite, Pedro Rosso, Rodrigo Ceccato, Carla Cusihualpa, Vitoria Dias, Sandro Rigo, Alan Souza, and Guido Araújo. The openmp cluster programming model. *Workshop Proceedings of the 51st International Conference on Parallel Processing*, 2022.