2
Charm++

O Charm++ é uma linguagem orientada a objetos para programação paralela baseada em C++ (34). Ela possui uma biblioteca de execução para suporte a computação paralela que se chama *Kernel do Charm* (*Charm Kernel*). O kernel provê uma clara separação entre objetos paralelos e objetos sequenciais. O modelo de execução de Charm++ é orientado a eventos, onde os eventos são chamadas assíncronas de métodos de objetos distribuídos. Charm++ tem suporte a balanceamento de carga dinâmico, tendo várias políticas de balanceamento implementadas no seu kernel.

Charm++ é um projeto maduro, dando suporte a diversas plataformas paralelas como PSC Lemieux, IBM SP, Bluegene/L, Cray X1, Cray T3E, Origin2000 ou clusters de máquinas de diversas arquiteturas, como Sun, HP, SGO, x86 (Linux e Windows) e IA64.

Diferente de modelos tradicionais de programação paralela, como passagem de mensagem (ex. MPI (09)) e variáveis compartilhadas (ex. BSP (10)), o modelo de funcionamento de Charm++ é orientado a eventos. Os eventos são chegadas de mensagens, que não são visíveis ao programador. Cada mensagem recebida está associada à execução de um método de um objeto distribuído. Quando uma mensagen é recebida, ela é tratada pelo sistema de execução de Charm++, que é responsável por fazer a chamada do método. Essa computação pode gerar novas mensagens (eventos) para outros processos e disparar execuções locais ou remotas.

Um programa em Charm++ consiste de um número de objetos distribuídos espalhados através do número de processadores disponíveis. Assim, a unidade básica de computação paralela em Charm++ é o objeto distribuído, chamado de *chare*, que pode ser criado em qualquer dos processadores disponíveis e pode ser acessado de outro processador remotamente.

Charm++ é construído sobre C++, e também baseado em encapsulamento. A principal parte da implementação de um chare é uma classe C++ que implementa todos os métodos do chare. Assim, como em C++, entidades de Charm++ podem conter dados e métodos públicos e/ou privados. A maior diferença é que em Charm++ esses métodos podem ser chamados de

processos remotos assincronamente. A chamada assíncrona (11) significa que o objeto que chama o método não espera que ele seja finalizado para continuar a execução; logo, em Charm++, os métodos não têm valor de retorno. Como o objeto onde o método está sendo chamado pode estar em um outro processador, a referência a um objeto via ponteiro não é válida. Ao invés disso é usado um *proxy* para referenciar um objeto.

O conceito de proxy utilizado em Charm++ é igual ao de vários modelos de componentes, como CORBA (12), no qual o proxy é um objeto que tem por objetivo referenciar o objeto real, que pode ser remoto. Para cada classe em Charm++, existe uma classe de proxy. Os métodos do proxy são os mesmos que os do objeto referenciado, e agem como encaminhadores, ou seja, quando um método do proxy é chamado, ele encaminha a requisição para o método do objeto remoto. Toda instanciação e manipulação de objetos remotos em Charm++ é feita através de um proxy.

Métodos de um chare são chamados *entry methods*. Entry methods podem receber parâmetros ou ponteiros para objetos de mensagem.

Charm++ oferece balanceamento de carga dinâmico. Ao criar um chare, a localização não precisa ser indicada: o kernel irá colocar o chare no processador menos carregado disponível.

Todo programa Charm++ deve conter pelo menos um *mainchare*. Um mainchare é um chare que é criado pelo sistema no processador principal (número 0) quando o programa é iniciado. A execução de um programa Charm++ começa com o Charm Kernel construindo todos os mainchares.

A forma principal de comunicação entre os processos em Charm++ são chamadas assíncronas de entry methods em chares remotos. Para isso, o kernel precisa conhecer os tipos de chares no programa do usuário, os métodos que podem ser chamados desse chare remotamente e os parâmetros que esses métodos recebem. Todos esses dados são descritos em um arquivo de interface. Esse arquivo de interface é utilizado para geração de código dos proxies dos objetos.

A geração de código baseada no arquivo de interface é muito importante para Charm++. Ela é usada para gerar os proxies que são responsáveis pela criação do objeto remoto, pela serialização e desserialização (*marshaling* e *unmarshaling*) dos parâmetros e pela chamada dos métodos do objeto C++ em si.

O processo de geração e compilação de código é demonstrado na Figura 2.1. Primeiramente, o parser lê o arquivo de interface e gera dois arquivos C++, um arquivo com extensão .decl.h, que é o arquivo de definição das classes do proxy, e o .def.h, que implementa as classes declaradas no primeiro

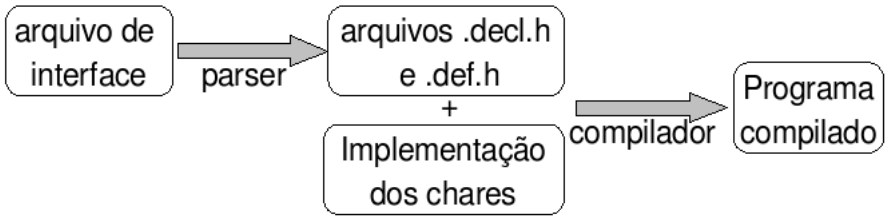

Figura 2.1: Compilação de um Programa Charm++

arquivo. Esses arquivos gerados são compilados juntamente com os arquivos que implementam os chares e são gerados dois arquivos: *charmrun*, que é o programa usado para iniciar a execução paralela, e um outro arquivo que possui a implementação da aplicação. Para executar um programa chamado *pgm* em 8 processadores, a linha de comando utilizada é: `charmrun pgm +p8`.

Um programa Charm++ acaba quando é feita uma chamada para `CkExit`. Como a função `exit`, `CkExit` nunca retorna. O Charm Kernel garante que nenhuma mensagem é processada e nenhum entry method é chamado após `CkExit` ser executada. `CkExit` não precisa ser chamada em todos os processadores. É suficiente que apenas um processo chame a função para indicar o fim do processamento.

2.1 Tipos de Objetos

A seguir descrevemos os tipos de objetos mais importantes para esta dissertação:

2.1.1 Chares

Chares são as entidades mais importantes em programas Charm++. Esses objetos concorrentes são diferentes de objetos C++ seqüenciais em vários aspectos.

Chares são instâncias de classes C++ que são derivadas de uma classe da API de Charm++ chamada `Chare`. Além de membros públicos e privados normais de C++, chares contêm alguns métodos públicos chamados *entry methods*. Esses entry methods não retornam nada (void) e podem receber como parâmetro tipos primitivos, outros chares ou ponteiros para objetos Message. Chares são acessados por proxy ou por um *handle*, que é um identificador único de um chare para toda a aplicação.

Chares são criados assincronamente de processadores remotos, e seus entry methods também podem ser chamados assincronamente de um processador remoto. Como o construtor vai ser chamado assincronamente, e pode ser executado em um processador remoto, todos os chares devem ter o seu construtor como um entry method.

2.1.2 Grupo (Chare Groups)

Grupos são um tipo especial de objeto concorrente. Cada grupo de chares é uma coleção de chares, com um representante (membro) em cada processador. Todos os membros de um grupo compartilham um nome global único definido pelo kernel de Charm++.

Um grupo pode ser endereçado como um todo ou um único chare pode ser acessado usando o seu nome global e o número de um processador. Grupos de chares são instâncias de classes C++ que estendem uma classe do sistema chamada `Group`.

2.1.3 Array (Chare Array)

Um *chare array* é uma coleção de chares. Porém, diferentemente de grupos, arrays não são restritos por características da arquitetura paralela, como número de processadores, assim um array pode ter qualquer número de elementos. Os elementos do array são chares e seus métodos podem ser chamados individualmente.

Arrays são tratados de modo especial pelo framework de balanceamento de carga, podendo ser migrados entre processadores. Assim, o sistema de execução coleta dados de carga no sistema e de tempo gasto na execução de cada método dos elementos de um array, e aplica uma das diversas estratégias para redistribuir os elementos entre os processadores disponíveis.

2.1.4 Readonly

Dados *Readonly* não são objetos paralelos. O seu valor é atribuído em um mainchare na inicialização do sistema e é enviado por broadcast para todos os processadores. Proxies do mainchare, de outros chares ou de arrays são exemplos típicos de dados declarados como readonly nos arquivos de interface.

2.1.5 Outros tipos

Além dos tipos principais descritos acima, Charm++ oferece outros tipos de objetos que são menos importantes para este trabalho por não terem sido usados no desenvolvimento desta dissertação. Um deles é o objeto *Mensagem* (*Message*), que é um objeto utilizado para o envio de dados numa chamada assíncrona. Tratado de modo diferente pelo sistema de execução, é mais eficiente para a transmissão de dados complexos. Além de mensagens, Charm++ também possui os tipos *Node* e *Node Group*.

2.2 A linguagem Charm++

Um programa Charm++ é estruturalmente similar a um programa C++. A maior parte de um programa Charm++ é código C++. A maior unidade sintática em programas Charm++ são as definições de classes.

Um programa em Charm++ é composto por um arquivo de interface, que é escrito em uma linguagem própria, e código C++ com a implementação das classes de chares. Um tradutor (parser) é usado para gerar o código necessário para lidar com as estruturas de Charm++. Esse parser gera código C++ que deve ser compilado com o código do usuário.

A seguir mostramos um exemplo de um programa Charm++ bastante simples:

Arquivo de interface:

```
mainmodule Hello {
   readonly CProxy_HelloMain mainProxy;
   mainchare HelloMain {
      entry HelloMain(); // argumento CkArgMsg * implicito
      entry void PrintDone(void);
   };
   group HelloGroup {
      entry HelloGroup(void);
   };
};
```

Arquivo de declaração das classes que vão implementar os chares:

```
#include "Hello.decl.h"
class HelloMain: public Chare {
```

```
   public:
      HelloMain(CkArgMsg *);
      void PrintDone(void);
   private:
      int count;
};
class HelloGroup: public Group {
   public:
      HelloGroup(void);
};
```

Arquivo de implementação dos chares:

```
#include "pgm.h"
CProxy_HelloMain mainProxy;
HelloMain::HelloMain(CkArgMsg *msg) {
   delete msg;
   count = 0;
   mainProxy=thishandle;
   CProxy_HelloGroup::ckNew(); // Cria um novo "HelloGroup"
}
void HelloMain::PrintDone(void) {
   count++;
   // Espera que todos os membros do grupo acabem
   if (count == CkNumPes()) {
      CkExit();
   }
}
HelloGroup::HelloGroup(void) {
   ckout << "Hello World from processor " << CkMyPe() << endl;
   mainProxy.PrintDone();
}
#include "Hello.def.h"
```

HelloMain é descrito como *mainchare*. Assim o Charm Kernel começa a execução deste programa criando uma instância de HelloMain no processador 0. O construtor de HelloMain cria um grupo de chares HelloGroup, e guarda uma referência de si próprio. A chamada de criação do grupo retorna imediatamente após direcionar o kernel para fazer a criação, que também é assíncrona. Pouco depois, o kernel cria um objeto do tipo HelloGroup em cada processador e chama os seus construtores. O construtor então imprime "Hello

World...” e depois chama o método `PrintDone` de `HelloMain`. Quando todos os chares chamarem `PrintDone`, o controlador estará no estado de finalização da execução, e chamará `CkExit` para o programa terminar.

Como a comunicação entre processos é feita através eventos, o chare controlador não pode fazer um loop no qual chame um método `receive` para receber todas mensagens de resposta, como seria possível em outros modelos, como MPI. Esse modelo torna o programa similar a uma máquina de estados, e seu estado deve ficar armazenado em variáveis globais, para que na próxima execução de um método, ele possa saber em que estado está e tomar a decisão do que fazer.

2.2.1 Entry methods

Em Charm++, todos os objetos se comunicam usando chamadas remotas. Esse método remoto pode receber parâmetros serializados ou objetos Message. Um *entry method* é sempre parte de um chare. Não existem métodos globais em Charm++. Os entry methods são declarados no arquivo de interface como:

```
entry void foo(int i, int j);
```

Chamar `foo(2, 3)` no proxy de um chare implica em chamar `foo(2, 3)` no chare remoto.

2.2.2 Atributos de entry methods e chares

Entry methods e chares podem ter vários atributos na sua declaração. Os dois principais atributos utilizados nesta dissertação são *lua* e *python*. Esses atributos são utilizados pelo gerador de código para que o chare tenha a funcionalidade que lhe é associada. O atributo python é usado no capítulo 3, na interface com python que foi desenvolvida pela equipe do Charm++, e o atributo lua é utilizado no capítulo 4, no binding desenvolvido durante esta dissertação.

Um exemplo de entry method com atributo é:

```
entry [python] void entryMethod(parameters);
```

e um exemplo de um chare com um atributo é:

```
chare [lua] chareName { ... }
```

Alguns outros atributos interessantes presentes em Charm++ e utilizado em entry methods são:

- *threaded* faz com que a execução do entry method seja feita em uma thread não preemptiva separada;
- *sync* permite que os métodos sejam chamados sincronamente por outro chare e que retornem um valor, que deve ser um ponteiro para uma mensagem.

Um método sync deve ser chamado por um entry method com o atributo threaded, para que sua execução possa ser suspensa sem pausar a thread principal do chare.

2.2.3 Chares

Chares são objetos concorrentes com métodos que podem ser chamados remotamente. Esses métodos são os entry methods e devem ser descritos nos arquivos de interface da seguinte maneira:

```
chare ChareName {
  entry ChareName (parameters);
  entry void methodName(parameters);
};
```

Chares podem ser dinamicamente criados em qualquer processador, e podem haver milhares de chares em um único processador. A localização de um chare é definida pelo balanceador de carga, mas uma vez que o chare comece a execução, ele não migra mais de processador, a menos que ele faça parte de um array.

2.3 Implementação em C++

Como descrito anteriormente, chares são descritos no arquivo de interface e implementados como uma classe em C++. Esta classe em C++ deve estender a classe `Chare` ou a classe `CBase_chareType`, onde `chareType` é o nome do chare declarado no arquivo de interface.

O chare descrito na seção 2.3.3 deveria ter sua classe declarada da seguinte maneira:

```
class ChareName : public CBase_ChareName {
  public:
```

```
    ChareName(parameters);
    void methodName(parameters);
};
```

e a sua implementação em C++ seria:

```
ChareName::ChareName(parameters){
  // codigo C++
}
void ChareName::methodName(parameters){
  // codigo C++
}
```

2.4 Converse

Converse (14) é a biblioteca de execução na qual o kernel de Charm++ se baseia. Inicialmente Converse era parte do kernel de Charm++, mas ele foi separado do kernel e passou a ser desenvolvido como uma biblioteca autônoma.

Converse é responsável pela troca de mensagens e pela interpretação das mensagens como eventos, que são registrados e tratados pelo kernel de Charm++. Converse também é responsável pelo controle do balanceamento de carga, pela migração de objetos serializados e ele implementa uma API de troca de mensagens no estilo servidor/cliente, que pode ser utilizada tanto por chares quanto por aplicações externas ao programa Charm++ para comunicação com chares.

2.5 Balanceamento de Carga

Uma grande preocupação no desenvolvimento de aplicações paralelas é fazer uso uniforme de todos os processadores, maximizando o aproveitamento dos recursos computacionais. Ainda assim, muitas aplicações paralelizáveis não têm uma estrutura regular para paralelização eficiente, e precisam de um balanceamento de carga para otimizar a distribuição de trabalho (19).

Outro fator que justifica a presença de um sistema de balanceamento de carga é a possibilidade da execução estar acontecendo em um ambiente não exclusivo, no qual carga externa à aplicação pode causar um desbalanceamento mesmo durante a execução de problemas paralelos com carga regular.

Em Charm++, com exceção de grupos, quando um objeto é criado, sua localização é definida pelo balanceador de carga. O balanceamento de carga de Charm++ é especialmente integrado com objetos do tipo array, que podem ser

migrados mesmo após a sua criação. Quando um array é criado, seus elementos são automaticamente registrados no framework de balanceamento de carga e podem ser migrados de processador a qualquer momento.

Durante a migração de chares em Charm++, o objeto será destruído em um processador e recriado em outro. Assim, os dados armazenados em variáveis do chare serão perdidos. Charm++ oferece um framework chamado *PUP*, que é usado para serialização de dados que serão enviados para o novo objeto. Para isso o chare deve implementar um método chamado *pup*, que será responsável tanto pelo empacotamento quanto pelo desempacotamento dos dados do chare.

O método `pup` recebe como parâmetro um objeto do tipo `PUP::er`. Esse objeto contém operadores e funções que com comportamento modificados, que funcionam tanto para a serialização, quanto para a desserialização. Quando chamado antes da migração, o objeto coleta os valores vindos das variáveis do chare, e, quando chamado após a migração, ele traz os valores armazenados de volta que serão inseridos nas variáveis do chare. O principal operador é o '|', que é um operador definido para receber tipos primitivos e armazená-los. Para migração de arrays, o objeto p deve receber o array e o seu tamanho com a seguinte sintaxe: `p(data, len)`.

Segue abaixo um exemplo simples de um chare migrável que utiliza o framework pup para a migração dos seus dados.

```
class ChareName : public CBase_ChareName {
  private:
    double d;
    char*  str;
  public:
    ....
    void pup(PUP::er &p){
      // serializa/desserializa valor de d
      p|d;
      int len = strlen(str);
      p|len;
      if (p.isUnpacking())
        str = new char[len];
      // serializa/desserializa len valores da string str
      p(str, len);
    }
};
```