2

Programação Paralela em CHARM++

Laércio Lima Pilla – laercio.pilla@ufsc.br [1]
Esteban Meneses – esteban.meneses@acm.org [2]

Resumo:

Este curso tem como objetivo apresentar conceitos básicos para o desenvolvimento de programas em CHARM++. CHARM++ segue um paradigma de programação paralela orientado a objetos baseado na troca de mensagens assíncronas que suporta o desenvolvimento de aplicações regulares e irregulares, assim como paralelismo de dados e de tarefas. Programas em CHARM++ são descritos através de objetos ativos chamados chares e suas interações com chamadas de métodos remotos não bloqueantes. Códigos desenvolvidos nesse paradigma usam a linguagem C++ com algumas extensões e se beneficiam de um sistema de execução maduro e otimizado. O sistema de execução suporta tanto ambientes multiprocessados quanto multicomputadores. Entre os diferenciais de CHARM++ podem ser citados: (i) o uso de sobredecomposição, o que envolve a divisão da aplicação em um grande número de unidades de dados e computação (usualmente maior do que o número de unidades de processamento disponíveis) e facilita a implementação de algoritmos; (ii) a execução direcionada por mensagens assíncronas, o que possibilita esconder latências de comunicação e sobrepor cálculo e comunicação junto à sobredecomposição; e (iii) a migrabilidade de objetos, o que permite o balanceamento dinâmico de carga, a tolerância a falhas de componentes e a expansão e redução de recursos computacionais durante a execução.

[1]Doutor em Ciência da Computação pela Universidade Federal do Rio Grande do Sul e pela *Université de Grenoble*, França (2014). Atualmente é professor na Universidade Federal de Santa Catarina, atuando nas áreas de balanceamento de carga em arquiteturas hierárquicas e tolerância a falhas em processadores gráficos.

[2]Doutor em Ciência da Computação pela *University of Illinois at Urbana-Champaign*, Estados Unidos da América (2013). Atualmente é professor na *University of Pittsburgh*, atuando nas áreas de tolerância a falhas energeticamente eficiente e balanceamento de carga para sistemas de larga escala.

2.1. Introdução

O desenvolvimento de programas paralelos é necessário para o uso eficiente das plataformas computacionais atuais. Por exemplo, processadores convencionais com múltiplos núcleos e aceleradores com centenas ou milhares de núcleos são usados para a construção de sistemas embarcados, computadores domésticos e supercomputadores. O paralelismo desses processadores e plataformas tende a crescer com o passar dos anos.

Infelizmente, a programação paralela é uma atividade sabidamente não trivial. O uso de interfaces de programação de aplicativos (APIs) padrão para a escrita de programas paralelos, tais como OPENMP [Dagum and Menon 2002] e MPI [Gropp et al. 1999], facilita esse trabalho para aplicações regulares e ambientes homogêneos. Entretanto, variações dinâmicas na plataforma ou no problema sendo modelado podem levar à perda de desempenho, eficiência e produtividade. Exemplos disso incluem a mudança de frequência de relógio nos processadores em uso, a perda de nós de computação na plataforma paralela e o crescimento da carga computacional em partes de uma área sendo simulada. Nesse contexto, o paradigma de programação paralela CHARM++ [Kale and Krishnan 1993] surge como uma possibilidade para mitigar esses problemas.

CHARM++ é um paradigma de programação paralela voltado à produtividade desenvolvido desde 1993 pelo Laboratório de Programação Paralela (PPL) da *University of Illinois at Urbana-Champaign* [Kale 2015]. CHARM++ é usado para o desenvolvimento de grandes aplicações científicas como NAMD [Phillips et al. 2005], um simulador de dinâmica molecular. O paradigma de programação é baseado na ideia de objetos migráveis cuja execução é guiada pela troca de mensagens. O seu sistema de execução (*runtime*) possibilita a execução de programas paralelos com mecanismos avançados, como balanceamento de carga automático e *checkpointing*. CHARM++ suporta tanto ambientes de multiprocessadores quanto multicomputadores, conseguindo realizar comunicação através de memória compartilhada e de protocolos ou infraestruturas de comunicação em rede, tais como TCP e o próprio MPI.

Este curso busca apresentar uma visão introdutória sobre o modelo de programação de CHARM++ através da discussão de suas principais características e do desenvolvimento de programas simples. Para seu melhor entendimento, é esperado que o leitor esteja familiarizado com conceitos de orientação a objetos, programação em C++ e programação paralela ou concorrente.

O curso é estruturado da forma seguinte: as principais características da linguagem de programação CHARM++ são apresentadas na Seção 2.2.; os atributos fundamentais de seu modelo de execução são discutidos na Seção 2.3.; a forma pela qual o sistema de execução de CHARM++ é organizado e como isso afeta o modelo de programação é detalhado na Seção 2.4.; dois exemplos de construção de programas paralelos em CHARM++ são abordados na Seção 2.5.; por fim, as conclusões são discutidas na Seção 2.6..

2.2. Modelo de Programação

O modelo de programação de CHARM++ envolve o particionamento da aplicação paralela em objetos que se comunicam através da troca de mensagens. Esses objetos executam sobre unidades de processamento (*processing elements* em inglês, ou PEs), as quais refletem núcleos ou processadores espalhados em uma plataforma paralela. Esse

modelo compartilha algumas similaridades com o modelo de atores usado em linguagens como ERLANG [Armstrong 2007], onde entidades autônomas se comunicam de forma assíncrona e explícita e podem criar novas entidades.

2.2.1. Objetos Ativos - Chares

Em programas escritos em CHARM++, toda a computação é realizada por objetos especiais chamados de ***objetos chare***. Nesse contexto, objetos chare são similares a threads em OPENMP e processos em MPI, executando de forma concorrente ou paralela em uma plataforma computacional. O conjunto de todos os objetos chare que compõem uma aplicação é chamado de *espaço global de objetos*. O espaço global de objetos pode mudar durante a execução da aplicação através da criação de novos objetos chare, assim como a deleção dos mesmos.

Como o modelo de CHARM++ é baseado na orientação a objetos, *objetos chare* são instanciados a partir de *classes chare*. Por um motivo de simplicidade, este curso se referenciará aos objetos chare como *chares* apenas, especificando claramente quando classes chare forem tratadas.

Cada chare conta com seus próprios dados e estado privados, o que implica que outros objetos na aplicação não podem acessar essas informações diretamente. Todo o compartilhamento de dados acontece através da comunicação direta entre chares por troca de mensagens. A Figura 2.1 ilustra a interação entre chares. O modelo de programação permite que um chare se comunique com qualquer outro chare que ele conheça no espaço global de objetos.

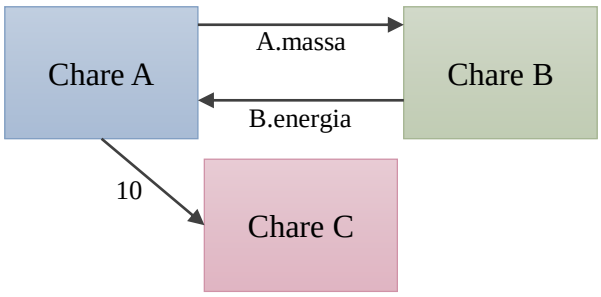
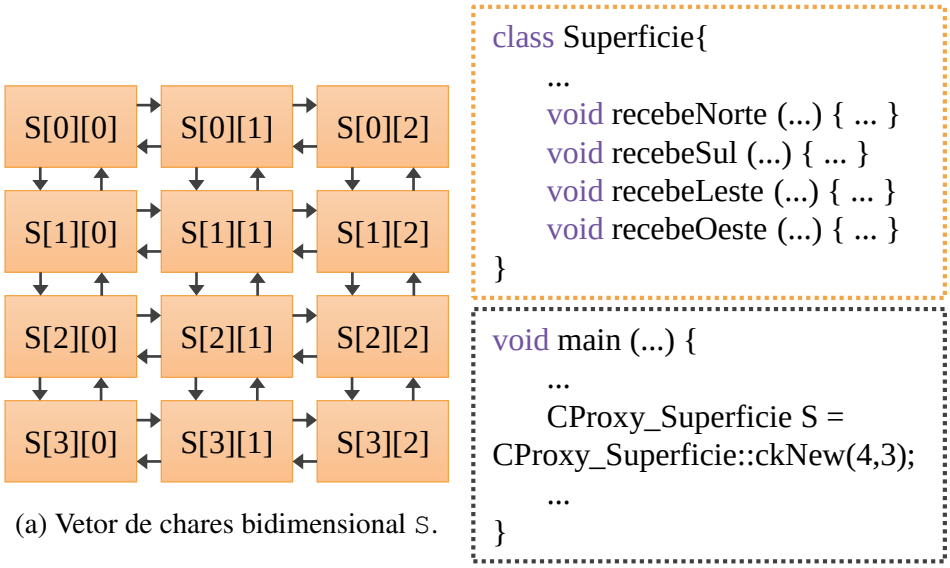


Figura 2.1: Comunicação entre os chares `A`, `B` e `C` através de troca de mensagens.

Chares podem ser criados de forma agrupada com o uso de **vetores de chares** (do inglês *chare arrays*). Um vetor de chares organiza chares de uma mesma classe chare através de índices lógicos da mesma forma que vetores são criados para outros tipos de dados em C++. CHARM++ suporta vetores de chares de uma até seis dimensões. O uso de vetores de chares facilita a comunicação entre chares e possibilita a comunicação via *broadcast*.

A organização de chares em vetores auxilia no desenvolvimento de programas paralelos ao facilitar o mapeamento entre o problema a ser resolvido e chares. Por exemplo, uma aplicação que simula a dissipação de calor em uma superfície poderia usar um vetor de chares bidimensional da classe *Superficie*. Em tal programa, um dado chare `S[i][j]` seria responsável pelo processamento de uma parte da superfície simulada e se comunicaria com seus vizinhos `S[i-1][j]`, `S[i][j-1]`, etc. Essa situação é ilustrada na Figura 2.2a. O chare `S[i][j]` saberia sua posição no vetor através

dos campos `thisIndex.x` e `thisIndex.y`, os quais são herdados por sua classe chare. Além disso, a comunicação com seus vizinhos usaria o campo `thisProxy`, o qual possibilita ao chare enxergar todos os outros membros do vetor de chares. No caso, o vizinho `S[i-1][j]` seria contactado através de uma chamada usando o *proxy* `thisProxy(thisIndex.x-1,thisIndex.y)`. Mais detalhes sobre *proxies* serão discutidos posteriormente no curso na Seção 2.4.2..



(a) Vetor de chares bidimensional `S`.

(b) Pseudocódigo do vetor de chares.

Figura 2.2: Exemplo de vetor de chares.

Além dos chares usados para modelar um problema, toda aplicação em CHARM++ tem um objeto especial chamado **main chare**. O *main chare* atua como o ponto de entrada da aplicação. Ele recebe parâmetros passados pelo usuário, inicializa outros chares e variáveis globais somente de leitura e usualmente coordena etapas de computação.

2.2.2. Troca de Mensagens

Em geral, programas paralelos envolvem o compartilhamento de dados entre suas partes, seja através de estruturas de dados compartilhadas em memória como em OPENMP, ou através de mensagens enviadas entre processos como no caso de MPI. CHARM++ usa a segunda abordagem para a comunicação de seus chares. Como chares possuem dados privados que não podem ser vistos por outros chares, eles trocam mensagens explicitamente. Isso facilita o controle e compreensão de quais informações cada chare tem acesso, assim como permite o uso de plataformas de multiprocessadores ou multicomputadores.

Mais especificamente, a comunicação entre chares é feita através de **chamadas remotas de métodos**. Por exemplo, na interação entre chares ilustrada na Figura 2.1, os chares `A` e `B` trocam informações privadas. O envio de uma mensagem de `A` para `B` com sua variável de massa acontece quando `A` invoca `B.recebeDado(massa)`, onde `recebeDado` é um método implementado por `B`. A Figura 2.3 apresenta o pseudocódigo que implementa essa interação. Esse método de comunicação é menos complexo do que o uso de chamadas de envio e recebimento de mensagens (como `MPI_Send` e `MPI_Receive` em MPI), o que facilita o desenvolvimento de programas.

```
Chare A{
    int massa;
    void meuMetodo () {
        ...
        B.recebeDado (massa);
        C.recebeValor (10);
        ...
    }
    void recebeResposta (int dado){
        ...
    }
}
```

```
Chare B{
    int energia;
    void recebeDado (int dado) {
        ...
        A.recebeResposta (energia);
        ...
    }
}
```

```
Chare C{
    void recebeValor (int dado) {
        ...
    }
}
```

Figura 2.3: Pseudocódigo da comunicação entre os chares `A`, `B` e `C` apresentados na Figura 2.1.

Os métodos de um chare que podem ser invocados por outros são chamados de **métodos de entrada**. É importante ressaltar que nem todos os métodos implementados por um chare serão métodos de entrada, ou seja, chares podem ter seus próprios métodos privados para auxiliar na organização do código. Um caso específico de método de entrada são os construtores, os quais são invocados na criação e inicialização de novos chares espalhados pela plataforma paralela.

Métodos de entrada podem receber mensagens com dados simples (por exemplo, um inteiro), estruturas de dados complexas ou, inclusive, nenhum dado. No caso de estruturas de dados e objetos complexos, uma função de serialização precisa ser implementada para o empacotamento e desempacotamento dos dados transferidos. Mais detalhes sobre isso serão discutidos na Seção 2.3.3..

A troca de mensagens em CHARM++ é feita de forma assíncrona. A **troca de mensagens assíncrona** faz com que a invocação de um método de entrada em outro chare tenha retorno imediato. Em outras palavras, o chare que envia uma mensagem continua sua execução sem esperar por uma resposta. Adicionalmente, o chare cujo método é chamado pode não começar a executá-lo imediatamente. Esse mecanismo é usado para guiar a execução da aplicação, como será explicado na Seção 2.3.1..

2.3. Modelo de Execução

O modelo de execução de CHARM++ é baseado nos preceitos de execução direcionada por mensagens assíncronas, sobredecomposição da aplicação e migrabilidade dos chares. Esses atributos influenciam diretamente as capacidades do sistema de execução, possibilitando um controle refinado no escalonamento de tarefas e no uso de recursos sujeitos a variações de desempenho e disponibilidade.

2.3.1. Execução Direcionada por Mensagens Assíncronas

Em um sistema onde a execução da aplicação é direcionada por mensagens, tarefas somente serão escalonadas para execução após o recebimento de mensagens. Dessa forma, o fluxo de controle da aplicação não é predeterminado e uma maior concorrência é evidenciada. Esse tipo de comunicação também é conhecido como mensagens ativas. Em CHARM++, um chare somente executará após outro chare invocar algum de seus métodos de entrada. Dado que a comunicação é feita de forma assíncrona, um chare que invoca um método de outro chare não tem uma resposta imediata, continuando a execução de seu próprio método até o fim. Nesse caso, o envio de uma resposta será feito através da invocação de um método de entrada do primeiro chare pelo segundo, o que indicará que esse poderá ser escalonado para execução novamente.

O uso de um modelo de comunicação assíncrona é benéfico pois remove a obrigação de uma sincronização entre o emissor e o receptor de uma mensagem. Isso remove tanto a espera do emissor por uma resposta – escondendo parte da latência de comunicação da plataforma – quanto o bloqueio da unidade de processamento do receptor enquanto espera pela recepção de uma mensagem.

Para que o fluxo de execução seja direcionado por mensagens, o ambiente de execução de CHARM++ sonda por mensagens enviadas (chamadas de métodos de entrada). Quando uma mensagem é descoberta, o próprio ambiente de execução identifica para qual chare ela foi enviada e a coloca em uma fila de mensagens. Essa fila de mensagens é usada para guiar o escalonamento de chares na unidade de processamento, o que será discutido em mais detalhes na Seção 2.4.1..

2.3.2. Sobredecomposição

A ideia de sobredecomposição, no contexto de CHARM++, trata do particionamento da aplicação em um número de chares muito maior do que o número de unidades de processamento da máquina onde a aplicação executará. Tal característica possibilita a sobreposição de computação e comunicação de forma natural, pois alguns chares podem estar prontos para execução enquanto outros esperam por chamadas a seus métodos de entrada. Essa sobreposição leva a uma menor ociosidade na plataforma paralela, o que é especialmente importante para plataformas de larga escala. O particionamento dos dados em pequenas unidades acarreta em uma maior localidade de dados, o que é benéfico para o uso eficiente de memórias cache [Acun et al. 2014].

O uso de sobredecomposição também leva à definição de aplicações em termos de suas entidades lógicas de forma independente do número de unidades de processamento das plataformas onde serão executadas. Essa abstração beneficia a produtividade dos programadores e a portabilidade de desempenho da aplicação ao possibilitar que o sistema de execução gerencie o mapeamento e escalonamento dos chares de forma mais otimizada.

Por fim, um código escrito dessa forma é mais modular, o que permite a composição de múltiplos módulos concorrentes e independentes para o desenvolvimento de novas aplicações paralelas.

2.3.3. Migrabilidade

A última (mas não menos importante) característica do modelo de execução de CHARM++ é a migrabilidade. A migrabilidade em CHARM++ envolve a habilidade de

mover chares entre unidades de processamento da plataforma paralela. Para possibilitar que chares sejam migrados, métodos de empacotamento e desempacotamento precisam ser implementados. Mais detalhes sobre isso são apresentados na Seção 2.4.3..

A migrabilidade possibilita tolerância a falhas em aplicações em CHARM++. Uma aplicação que realiza *checkpointing* de seus chares pode se prevenir ou recuperar de uma falha através da migração ou reinicialização dos chares afetados em outros recursos computacionais da plataforma. Através de uma combinação de *checkpointing* com a manutenção de *logs* de mensagens, aplicações em CHARM++ podem inclusive se recuperar de uma falha sem precisar da interrupção e reinicialização da aplicação por completo.

A combinação de migrabilidade com sobredecomposição permite a migração de chares para o balanceamento da carga da aplicação na plataforma. Dado que a carga computacional de chares pode ser desconhecida durante a inicialização da aplicação, ou pode ser dinâmica e mudar durante a execução da aplicação, o mapeamento inicial de chares para unidades de processamento pode não ser ideal. Nesse contexto, algoritmos de balanceamento de carga externos à aplicação podem ser usados para migrar chares de unidades de processamento sobrecarregadas para outras ociosas, melhorando o desempenho da aplicação. O mesmo pode ser feito para aprimorar a comunicação entre chares, assim como reduzir o consumo de potência ou temperatura das unidades de processamento [Acun et al. 2014]. Mais detalhes sobre como o balanceamento de carga é feito em CHARM++ serão apresentados na Seção 2.4.4..

A combinação dessas características – migrabilidade e sobredecomposição – possibilita também uma elasticidade à aplicação, a qual pode se adaptar a mudanças na quantidade de recursos computacionais disponíveis para sua execução. No caso de um aumento no número de unidades de processamento disponíveis, chares podem ser migrados para os novos recursos. Já no caso de recursos precisarem ser liberados (no caso da aplicação executar de forma oportunista, por exemplo), chares presentes em tais recursos são migrados para as unidades de processamento que serão mantidas.

2.4. Sistema de Execução

CHARM++ possui um sistema de execução (*runtime*) maduro e otimizado para o uso de diferentes plataformas computacionais, sejam elas multiprocessadas ou multicomputadores. O sistema de execução é responsável pelo mapeamento e migração de chares entre unidades de processamento, pelo gerenciamento da troca de mensagens entre chares e por atividades avançadas como *checkpointing*. Essas funcionalidades são discutidas nas seções seguintes.

2.4.1. Filas de Mensagens e Escalonamento

O sistema de execução é responsável pelo mapeamento inicial dos chares, pela seleção de qual chare deve executar em uma dada unidade de processamento e pelo gerenciamento das mensagens trocadas entre chares. No momento de criação e inicialização de um chare, o próprio ambiente escolherá uma unidade de processamento adequada para este, não dependendo de controle pelo programador. Após sua criação, um chare ficará ocioso esperando por uma chamada a um de seus métodos de entrada. Um chare existe apenas em uma unidade de processamento em um dado momento.

Todas as mensagens enviadas para chares em uma unidade de processamento são colocadas em sua fila de mensagens, como ilustrado na Figura 2.4. Cada entrada na fila de mensagens inclui informações sobre qual chare é o receptor, o método invocado e os dados enviados na mensagem. Por exemplo, a primeira entrada na fila de mensagens da unidade de processamento 0 na Figura 2.4 informa que o método `recebe` do chare `C` foi invocado com o valor 10.

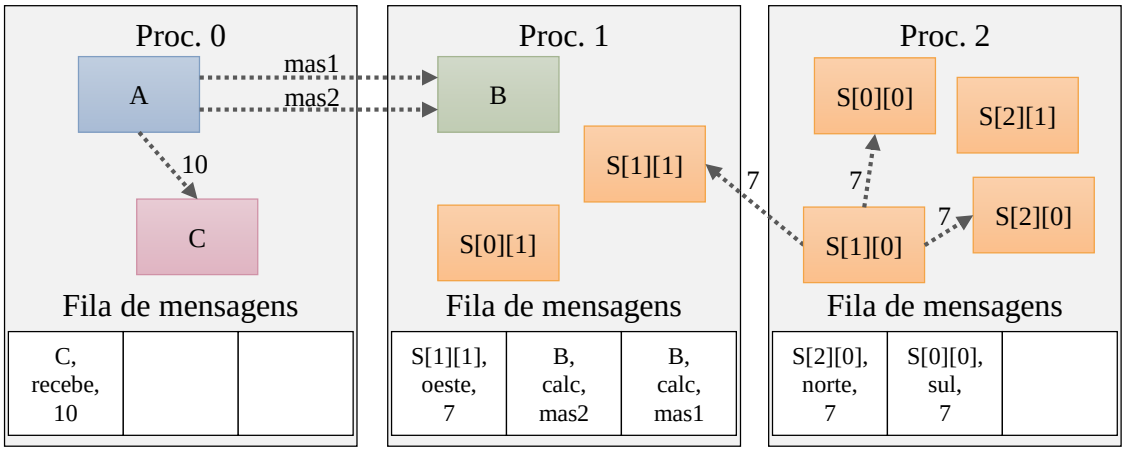


Figura 2.4: Distribuição de chares em unidades de processamento e suas filas de mensagens.

Quando a unidade de processamento se encontra sem chares em execução, o escalonador local seleciona a primeira mensagem na fila e coloca o respectivo chare a executar o método de entrada invocado. Tal chare executa sem preempção pelo escalonador. Caso o chare invocasse um de seus próprios métodos de entrada, a chamada seria registrada na fila de mensagens e o chare continuaria sua execução sem alterações. Ao terminar de executar seu método, o chare volta à ociosidade e o escalonador busca outra mensagem em sua fila para tratar.

O escalonamento de chares direcionado por mensagens e a propriedade de não preempção de um chare garantem que um chare não irá executar dois métodos concomitantemente, o que passa uma visão de atomicidade na execução de métodos de entrada similar a métodos *synchronized* em Java [Deitel and Deitel 2015].

Vale salientar que o ambiente de execução de CHARM++ não provê garantias de ordenamento na recepção de mensagens. Por exemplo, caso um chare `A` invocasse os métodos de entrada `B.primeiroMetodo()` e `B.segundoMetodo()` em sequência, o sistema poderia executar o segundo antes do primeiro.

2.4.2. Roteamento de Mensagens

A comunicação entre chares em uma plataforma paralela é abstraída do programador pelo sistema. No caso, um chare não precisa saber exatamente em qual unidade de processamento o receptor de sua mensagem se encontra no momento, podendo este ter inclusive migrado durante sua execução. Para garantir que uma mensagem chegue a seu destinatário, o sistema atribui a responsabilidade pelo conhecimento da localização de um chare a uma unidade de processamento. Caso o chare venha a ser migrado, a comunicação ainda chegará para tal unidade de processamento, a qual irá encaminhar a mensagem

para o local correto.

Adicionalmente, chares não possuem referências diretas a outros chares. No caso, toda a comunicação é feita através de *proxies*. Um *proxy* serve como um "manequim" do chare de interesse. Ele é responsável pela preparação da mensagem enviada (serializando os dados, por exemplo) e por seu encaminhamento para o destino correto. Um exemplo de *proxy* é o campo `thisProxy`, usado por um chare para enxergar a si ou aos outros membros de seu vetor de chares.

2.4.3. Serialização de Objetos

A serialização de objetos em CHARM++ tem duas funcionalidades importantes: a serialização de mensagens para envio e a serialização de chares para migração ou *checkpoint*. Felizmente, CHARM++ disponibiliza um arcabouço com métodos chamados PUP (do inglês *pack and unpack*) que auxiliam na serialização de objetos complexos.

Para possibilitar a serialização de um objeto, um método `pup` precisa ser implementado para sua classe. A Figura 2.5b ilustra uma possível implementação de método `pup` para uma classe composta por dois valores inteiros (`i` e `t`), um vetor de 10 caracteres (`nome[10]`) e um vetor de `t` inteiros (`valores`), como apresentado na Figura 2.5a. No método `pup`, o operador "`|`" é usado para o empacotamento e desempacotamento de dados simples, enquanto o método `PUParray` é usado para os vetores. É importante ressaltar que o método `pup` para a serialização de uma classe chare requer a chamada do método de serialização de sua superclasse.

```
class minhaClasse {
    int i, t;
    char nome[10];
    int *valores;
}
```

(a) Classe a ser serializada.

```
void minhaClasse::pup ( PUP::er &p ) {
    p|i;
    p|t;
    PUParray ( p, nome, 10 );
    PUParray ( p, valores, t );
}
```

(b) Implementação de método `pup`.

Figura 2.5: Exemplo de método para a serialização de objetos.

2.4.4. Balanceamento de Carga

O arcabouço de balanceamento de carga de CHARM++ [Sinha and Kale 1993] é uma das partes mais importantes de seu sistema. Através de classes de balanceadores de carga, desenvolvidas independentemente de aplicações, é possível reorganizar chares em uma plataforma de forma a equalizar a carga de trabalho entre as unidades de processamento [Pilla et al. 2014], reduzir o tempo de comunicação entre chares, reduzir o consumo de potência em unidades de processamento mais ociosas, entre outros. CHARM++ disponibiliza dezenas de algoritmos de balanceamento de carga para uso por desenvolvedores.

O balanceamento de carga em CHARM++ é baseado na medição do tempo de atividade dos chares e das unidades de processamento. Ele é baseado no *princípio da persistência*, o qual indica, através de observações, que a carga computacional e os padrões de comunicação de aplicações científicas tendem a persistir ao longo do tempo; e que o passado recente da aplicação é um bom preditor de seu futuro próximo.

Durante um certo período de execução de uma aplicação (por exemplo, um determinado número de iterações ou segundos), o sistema coleta informações como: quanto tempo cada chare passou executando; quanto tempo uma unidade de processamento passou ociosa; quantas mensagens um dado chare enviou para outro e qual a quantidade de dados comunicados; e o mapeamento atual de chares para unidades de processamento. Esses dados são organizados pelo ambiente em um vetor de informações sobre chares e um vetor de informações sobre unidades de processamento, os quais são encaminhados a um algoritmo de balanceamento de carga. O algoritmo é responsável por avaliar as informações atuais recebidas e prover um novo mapeamento de chares para unidades de processamento.

Para habilitar o balanceamento de carga em uma aplicação em CHARM++, o programador é responsável apenas por implementar métodos de serialização para os objetos, por inserir uma chamada ao método bloqueante `AtSync()` em algum momento da execução de todos os chares (outros formas para o balanceamento de carga são disponibilizadas pelo ambiente mas não serão discutidas neste curso) e por definir um método de entrada `ResumeFromSync()`, o qual será chamado após o término do balanceamento de carga. Quando todos os chares alçarem a chamada `AtSync`, um algoritmo de balanceamento de carga externo, o qual foi compilado em anexo à aplicação e escolhido no momento de sua execução, computará um novo mapeamento.

2.4.5. Checkpointing

A execução de uma aplicação paralela pode ser entendida como um histórico distribuído, onde o estado da aplicação a qualquer momento no tempo é uma fotografia composta pelo estados de nós individuais e pelas mensagens em trânsito. Tal fotografia do sistema é também chamada de *checkpoint*. Funções de *checkpointing* são usualmente acionadas em momentos de sincronização global. Desse modo, o *checkpoint* não precisa incluir o estado dos canais de comunicação, pois não há mensagens em trânsito, e a interferência na execução da aplicação é mantida pequena.

Existem diversos usos para *checkpoints*. Primeiro, eles são utilizados para a análise da evolução de um sistema, pois o processo que leva ao resultado final é tão importante quanto o próprio resultado em algumas aplicações. Por exemplo, em uma simulação de interação de partículas, as trajetórias das partículas em movimento no espaço podem ser de interesse. Este tipo de análise *post mortem* é comum em diversas áreas de ciência computacional. Segundo, *checkpoints* são úteis para a execução de uma aplicação em intervalos de tempo separados, o que acontece quando o tempo total de execução de uma aplicação ultrapassa o tempo máximo permitido para alocação pelo sistema. Além disso, alguns trechos da execução podem compartilhar fases que não precisam ser recomputadas. Por fim, *checkpoints* são úteis para a tolerância a falhas. Se uma parte da plataforma ou o sistema todo falha por parada, *checkpoints* são usados para a recuperação da aplicação e manutenção do progresso dela mesmo com falhas recorrentes.

Em CHARM++, *checkpointing* é implementado naturalmente através do arcabouço de serialização de objetos (PUP) discutido na Seção 2.4.3.. Como chares são migráveis

por definição, os métodos PUP são usados para copiar uma imagem dos chares da memória e armazená-la. Ao escrever os métodos de serialização, o programador precisa estar ciente que tais métodos podem ser usados para reiniciar a execução do programa e, logo, todas as informações necessárias para recuperar o estado da aplicação para sua re-execução precisam ser incluídos.

CHARM++ inclui diferentes implementações de *checkpoint* [Meneses et al. 2014]. A *implementação baseada em discos em rede* é empregada para a execução particionada da aplicação e para a análise *post mortem*. O sistema de execução descarrega o estado de todos os chares em arquivos na rede em um sistema de arquivos compartilhado. Esse mecanismo costuma saturar a largura de banda da rede de interconexão e aumentar o sobrecusto do *checkpoint*. Entretanto, os dados são considerados seguros e os *checkpoints* podem ter tamanhos arbitrários. CHARM++ possibilita o resumo da execução da aplicação com um número diferentes de recursos computacionais, o que também permite execuções com expansão e encolhimento destes.

A segunda implementação de *checkpoints* em CHARM++ é *baseada em armazenamento local*. Esse mecanismo é empregado para a recuperação de falhas durante o tempo de execução da aplicação. O sistema de execução descarrega o estado dos chares para um meio de armazenamento local nos nós sendo utilizados, podendo isso ser feito em memória principal ou secundária. Por exemplo, no mecanismo de *checkpoint* de par em memória [Zheng et al. 2004], duas cópias do estado de execução são mantidas. A primeira cópia é mantida na memória primária local e a segunda na memória primária de um nó *parceiro*. No caso de uma parada no nó X, o seu nó parceiro Y pode prover o *checkpoint* de X para um nó que o irá substituir. Enquanto isso, todos os outros nós irão reiniciar suas execuções com seus estados armazenados localmente. Dessa forma, a recuperação da aplicação por *checkpoint/restart* é implementada eficientemente.

2.5. Exemplos de Código Paralelos

O desenvolvimento de aplicações em CHARM++ é organizado através da escrita de classes para os chares e para o *main chare*. Os componentes da aplicação são comumente divididos em módulos de forma a aprimorar o particionamento e a reutilização de códigos, levando a uma maior produtividade.

Neste curso, são discutidos dois exemplos de aplicações adaptados do tutorial de CHARM++ em inglês [Laboratory 2015]. Os exemplos tratam de um programa "Olá Mundo" usado para exemplificar as ideias básicas de definição de métodos de entrada, módulos, arquivos e processo de compilação, e de um programa stencil de duas dimensões usado para ilustrar conceitos de comunicação avançada entre chares, sincronização e cálculo em etapas. Os códigos de ambos exemplos são apresentados por completo nas seções seguintes.

2.5.1. "Olá Mundo" com Vetor de Chares

O programa "Olá Mundo" contém um conjunto de chares onde cada um é responsável por imprimir uma mensagem na saída padrão e pedir para que o próximo chare faça o mesmo. Para isso, são usados um vetor de chares `OlaMundo` e um *main chare* responsável pela inicialização e finalização do programa. Cada uma dessas partes do programa

é organizada em módulos.

Um módulo em CHARM++ costuma envolver arquivos identificados pelas extensões "*.C*", "*.h*" e "*.ci*". Arquivos "*.C*" contêm a implementação de classes, enquanto arquivos "*.h*" contêm os cabeçalhos e definições de classes. Arquivos "*.ci*" são usados para definir quais classes no módulo são classes chare, quais são seus métodos de entrada, além das variáveis globais somente de leitura.

OlaMundo.C

```
1.  #include "OlaMundo.decl.h"
2.  #include "OlaMundo.h"
3.  #include "main.decl.h"

4.  extern /* readonly */ CProxy_Main mainProxy;
5.  extern /* readonly */ int numChares;

6.  OlaMundo::OlaMundo () { }
7.  OlaMundo::OlaMundo (CkMigrateMessage *msg)
    { }

8.  void OlaMundo::dizerOi (int emissor) {
9.    CkPrintf("\"Ola mundo\" do chare %d no PE %d
    (pedido pelo chare %d).\n", thisIndex, CkMyPe(),
    emissor);

10.   if (thisIndex < (numChares - 1))
11.     thisProxy[thisIndex + 1].dizerOi(thisIndex);
12.   else
13.     mainProxy.terminado();
14. }

15. #include "OlaMundo.def.h"
```

OlaMundo.h

```
16. #ifndef __OlaMundo_H__
17. #define __OlaMundo_H__

18. class OlaMundo : public CBase_OlaMundo
    {
19.   public:
20.     OlaMundo ();
21.     OlaMundo (CkMigrateMessage *msg);
22.     void dizerOi(int emissor);};

23. #endif //__OlaMundo_H__
```

OlaMundo.ci

```
24. module OlaMundo {
25.   array [1D] OlaMundo {
26.     entry OlaMundo();
27.     entry void dizerOi(int);
28.   };
29. };
```

Figura 2.6: Programa "Olá Mundo": módulo com a classe de chares que imprimem mensagens.

O código do módulo *OlaMundo* é ilustrado na Figura 2.6. O arquivo *OlaMundo.h* declara que a classe é filha da classe `CBase_OlaMundo`, o que é comum para a escrita de classes chare, e que ela tem três métodos, sendo eles dois construtores e um método `dizerOi()`. Esses três métodos são métodos de entrada, os quais são identificados como tais através da palavra-chave `entry` presente no arquivo *OlaMundo.ci*. O construtor com o parâmetro `CkMigrateMessage` não precisa ser declarado no arquivo *.ci*. Ele é usado apenas para a migração de chares, o que não é tratado no exemplo.

A implementação do principal método da classe chare `OlaMundo`, `dizerOi()`, é composta das linhas 8 a 14 da Figura 2.6. Nesse método, um chare recebe uma mensagem com um parâmetro contendo seu emissor. Na linha 9, a função de CHARM++ `CkPrintf` é usada para a impressão de uma mensagem na saída padrão. A mensagem usa o campo `thisIndex` para indicar a posição do chare no vetor de chares e a chamada `CkMyPe()` que provê o identificador da unidade de processamento onde o chare se encontra mapeado. Após a escrita da mensagem, o chare invoca o método `dizerOi()` no próximo chare do vetor através do *proxy* `thisProxy` ou, caso ele seja o último chare do vetor, invoca o método `terminado()` no *main chare*, cujo *proxy* é disponibilizado como uma variável global somente de leitura.

É notável que o código em *OlaMundo.C* inclui arquivos "*.decl.h*" em seu início e "*.def.h*" em seu final, os quais não fazem parte dos arquivos do módulo. A razão disso é que esses arquivos são gerados automaticamente no processo de compilação de CHARM++ baseados no arquivo "*.ci*". Esses arquivos descrevem partes importantes do código em CHARM++ que são abstraídas dos programadores. Eles devem sempre ser incluídos em seus respectivos lugares, podendo gerar erros de compilação difíceis de identificar caso esquecidos.

A Figura 2.7 apresenta o código do módulo *main*. O principal método do *main chare* é seu construtor, o qual organiza o início da execução do programa. A criação do vetor de chares da classe `OlaMundo` acontece na linha 13 e resulta no *proxy* `OlaMundoArray`. Na linha 14, a primeira mensagem é enviada ao chare de índice 0. Nela, o *main chare* se identifica como membro de índice -1 no vetor de chares. Ao fim do programa, o método `terminado()` é invocado, o que resulta na chamada da função `CkExit()` pelo *main chare* e na respectiva terminação da execução.

Todas as variáveis globais (somente de leitura) são inicializadas pelo *main chare*. No caso do programa "Olá Mundo", as únicas informações externas necessárias para a execução dos chares são o número de chares no vetor e o *proxy* para o *main chare*. Essas variáveis são declaradas como somente de leitura através da palavra-chave `readonly`.

main.C

```
1.  #include "main.decl.h"
2.  #include "main.h"
3.  #include "OlaMundo.decl.h"

4.  /* readonly */ CProxy_Main mainProxy;
5.  /* readonly */ int numChares;

6.  Main::Main (CkArgMsg* msg) {
7.    numChares = 7;
8.    if (msg->argc > 1)
9.      numChares = atoi (msg->argv[1]);
10.   delete msg;

11.   CkPrintf ("Rodando o programa \"Ola Mundo\"
      com %d chares em %d unidades de
      processamento.\n", numChares, CkNumPes());

12.   mainProxy = thisProxy;
13.   CProxy_OlaMundo OlaMundoArray =
      CProxy_OlaMundo::ckNew (numChares);
14.   OlaMundoArray[0].dizerOi (-1);
15. }

16. Main::Main (CkMigrateMessage* msg) { }

17. void Main::terminado () {
18.   CkExit ();
19. }

20. #include "main.def.h"
```

main.h

```
21. #ifndef __MAIN_H__
22. #define __MAIN_H__

23. class Main : public CBase_Main {
24.   public:
25.     Main (CkArgMsg* msg);
26.     Main (CkMigrateMessage* msg);
27.     void terminado();
28. };

29. #endif //__MAIN_H__
```

main.ci

```
30. mainmodule main {
31.   readonly CProxy_Main mainProxy;
32.   readonly int numChares;
33.   extern module OlaMundo;

34.   mainchare Main {
35.     entry Main (CkArgMsg* msg);
36.     entry void terminado ();
37.   };
38. };
```

Figura 2.7: Programa "Olá Mundo": módulo com o *main chare*, responsável pela inicialização e finalização do programa.

Makefile

```
1.  CHARMDIR = [caminho para o diretório de CHARM++ na máquina]
2.  CHARMC = $(CHARMDIR)/bin/charmc $(OPTS)

3.  default: OlaMundo

4.  OlaMundo : main.o OlaMundo.o
5.    $(CHARMC) -language charm++ -o OlaMundo main.o OlaMundo.o

6.  main.o : main.C main.h main.decl.h main.def.h OlaMundo.decl.h
7.    $(CHARMC) -o main.o main.C

8.  main.decl.h main.def.h : main.ci
9.    $(CHARMC) main.ci

10. OlaMundo.o : OlaMundo.C OlaMundo.h OlaMundo.decl.h OlaMundo.def.h main.decl.h
11.   $(CHARMC) -o OlaMundo.o OlaMundo.C

12. OlaMundo.decl.h OlaMundo.def.h : OlaMundo.ci
13.   $(CHARMC) OlaMundo.ci

14. clean:
15.   rm -f main.decl.h main.def.h main.o OlaMundo.decl.h OlaMundo.def.h
16.   rm OlaMundo.o OlaMundo charmrun
```

Figura 2.8: Programa "Olá Mundo": arquivo Makefile para compilação e ligação dos módulos.

Dado que o processo de compilação de um programa em CHARM++ envolve a ligação de diferentes módulos, a Figura 2.8 apresenta o arquivo de compilação usado para o exemplo. A compilação em CHARM++ ocorre através do uso da ferramenta `charmc`, a qual abstrai o uso do compilador nativo instalado (por exemplo, `gcc`). As linhas 6 a 9 envolvem a compilação do módulo *main*, enquanto as linhas 10 a 13 tratam do módulo *OlaMundo*. Para cada módulo, primeiramente o arquivo "*.ci*" é compilado, gerando os arquivos "*.decl.h*" e "*.def.h*". Por fim, o arquivo objeto "*.o*" é gerado e os módulos são combinados na linha 5 para a geração do arquivo executável.

Compilado o programa, a forma de começar sua execução dependerá de como foi instalado o sistema de execução de CHARM++. No caso de um ambiente multiprocessado, o comando *./OlaMundo +p3* fará com que o programa execute sobre três unidades de processamento. Caso contrário, a execução em uma plataforma multicomputador utiliza o executável *charmrun* (gerado no processo de compilação) através do comando *./charmrun +p3 ./OlaMundo*.

2.5.2. Jacobi2D com sincronização

O segundo exemplo deste curso ilustra a implementação de um cálculo de Jacobi iterativo sobre uma matriz bidimensional. Neste exemplo, o valor de uma célula é atualizado baseado na média de seu valor atual e os de seus quatro vizinhos. Esse processamento é feito para cada célula da matriz em uma iteração. Ao fim da iteração, a maior variação de valor entre todas as células (diferença entre o valor anterior e o atual) é descoberta e uma nova iteração somente será necessária se este valor estiver acima de um

limiar pré-determinado. No caso contrário, a aplicação termina.

No exemplo implementado nesta seção, uma matriz de 50×50 células tem sua primeira célula com o valor mantido em 1, enquanto as outras células começam com o valor 0. Na primeira iteração, os vizinhos da primeira célula acabam com seu valor atualizado para $0,2$ ($\frac{1}{5}$), o que representa uma variação de $0,2$, como ilustra a Figura 2.9. Na iteração seguinte, estas mesmas células e suas vizinhas acabam com novos valores. Dessa forma, o valor da primeira célula é propagado através das iterações.

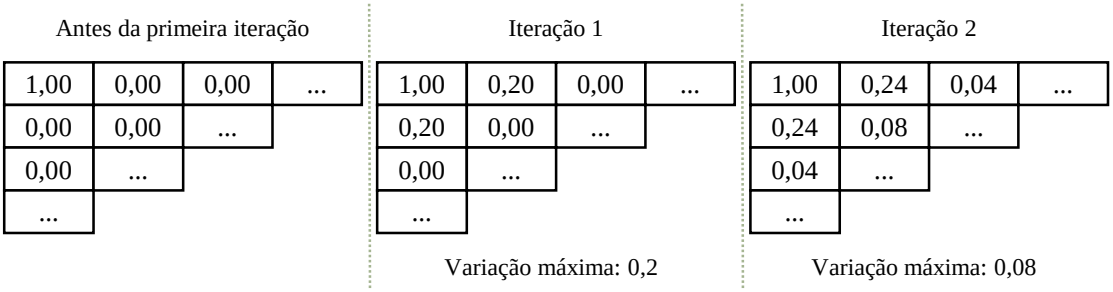


Figura 2.9: Programa Jacobi2D: iterações iniciais da aplicação.

jacobi.h

```
1.  #ifndef __JACOBI_H__
2.  #define __JACOBI_H__

3.  #include "jacobi.decl.h"

4.  class Jacobi : public CBase_Jacobi {
5.   private:
6.    double* valores;
7.    double* temp;
8.    int alvoChamadas;
9.    int chamadas;
10.   void tentaCalcular();
11.   double calcula();

12.  public:
13.   Jacobi();
14.   Jacobi(CkMigrateMessage *msg);
15.   ~Jacobi();
16.   void iniciaEtapa();
17.   void dadosLeste(int tam, double* vals);
18.   void dadosOeste(int tam, double* vals);
19.   void dadosSul(int tam, double* vals);
20.   void dadosNorte(int tam, double* vals);
21. };

22. #endif //__JACOBI_H__
```

jacobi.ci

```
23. module jacobi {
24.  array [2D] Jacobi {
25.   entry Jacobi();
26.   entry void iniciaEtapa();
27.   entry void dadosLeste(int tam, double vals[tam]);
28.   entry void dadosOeste(int tam, double vals[tam]);
29.   entry void dadosSul(int tam, double vals[tam]);
30.   entry void dadosNorte(int tam, double vals[tam]);
31.  };
32. };
```

Figura 2.10: Programa Jacobi2D: definição do módulo da matriz de chares.

Similar ao exemplo anterior, o código do programa Jacobi2D é dividido em dois módulos: um módulo que define o vetor bidimensional de chares responsáveis pela computação e um módulo para o *main chare*. A Figura 2.10 apresenta os arquivos "*.h*" e "*.ci*"

jacobi.C (parte 1)

```
1.  #include "jacobi.h"
2.  #include "main.h"

3.  extern /* readonly */ CProxy_Main mainProxy;
4.  extern /* readonly */ int linhas;
5.  extern /* readonly */ int colunas;
6.  extern /* readonly */ int linhasChares;
7.  extern /* readonly */ int colunasChares;

8.  Jacobi::Jacobi() {
9.    int numElementos = (linhas + 2) * (colunas +
    2);
10.   valores = new double[numElementos];
11.   memset(valores, 0, sizeof(double) *
    numElementos);
12.   temp = new double[numElementos];

13.   if (thisIndex.x == 0 && thisIndex.y == 0 &&
    valores != NULL)
14.     valores[(colunas + 2) + 1] = 1.0;

15.   alvoChamadas = 1;
16.   if (thisIndex.x > 0) alvoChamadas++;
17.   if (thisIndex.x < colunasChares - 1)
18.     alvoChamadas++;
19.   if (thisIndex.y > 0) alvoChamadas++;
20.   if (thisIndex.y < linhasChares - 1)
21.     alvoChamadas++;
22.   chamadas = 0;
23. }
```

```
24. Jacobi::Jacobi(CkMigrateMessage *msg) { }
25. Jacobi::~Jacobi() {
26.   if (valores != NULL) delete [] valores;
27.   if (temp != NULL) delete [] temp;
28. }

29. void Jacobi::iniciaEtapa() {
30.   if (thisIndex.x > 0) {
31.     for (int i = 0; i < linhas; i++)
32.       temp[i] = valores[1 + ((colunas + 2) * (i +
    1))];
33.     thisProxy(thisIndex.x - 1,
    thisIndex.y).dadosLeste(linhas, temp);
34.   }
35.   if (thisIndex.x < colunasChares - 1) {
36.     for (int i = 0; i < linhas; i++) temp[i] =
    valores[colunas + ((colunas + 2) * (i + 1))];
37.     thisProxy(thisIndex.x + 1,
    thisIndex.y).dadosOeste(linhas, temp);
38.   }
39.   if (thisIndex.y > 0) {
40.     double* bufferPtr = &(valores[(colunas + 2) +
    1]);
41.     thisProxy(thisIndex.x, thisIndex.y -
    1).dadosSul(colunas, bufferPtr);
42.   }
43.   if (thisIndex.y < linhasChares - 1) {
44.     double* bufferPtr = &(valores[((colunas + 2)
    * colunas) + 1]);
45.     thisProxy(thisIndex.x, thisIndex.y +
    1).dadosNorte(colunas, bufferPtr);
46.   }
47.   tentaCalcular();
48. }
```

Figura 2.11: Programa Jacobi2D: primeira parte da implementação da matriz de chares.

do módulo *Jacobi*. Neles são definidos as variáveis privadas dos chares, seus métodos privados e seus métodos de entrada. O método de entrada `iniciaEtapa()` é usado pelo *main chare* para começar uma iteração, enquanto os métodos de entrada `dados*` servem para a comunicação entre chares da mesma classe, cada um para uma direção.

As Figuras 2.11 e 2.12 apresentam a implementação da classe *Jacobi*. Em sua inicialização, cada elemento do vetor de chares aloca duas matrizes para o armazenamento de suas células locais. As matrizes possuem duas colunas e duas linhas extras para armazenar os dados enviados pelos vizinhos para realizar suas computações internas. Uma das matrizes guarda os dados da última iteração, enquanto a outra é usada temporariamente para manter os novos valores sendo calculados. Durante essa etapa, cada chare descobre a quantidade de chares (vizinhos e si próprio) que precisam lhe enviar dados para que ele possa realizar seu cálculo. Um chare precisa receber uma confirmação de si próprio para garantir que ele não realizará o envio de mensagens para seus vizinhos no método `iniciaEtapa()` com seus dados já calculados para a iteração, o que resultaria numa condição de corrida e na corrupção dos dados.

O método `iniciaEtapa()`, apresentado nas linhas 29 a 48 da Figura 2.11, é invocado para todos os chares no início de uma iteração. Durante sua execução, o chare

jacobi.C (parte 2)

```
49. void Jacobi::dadosLeste(int tam, double* vals) {
50.   for (int i = 0; i < linhas; i++) valores[((colunas
    + 2) * (i + 1)) + colunas + 1] = vals[i];
51.   tentaCalcular();
52. }
53. void Jacobi::dadosOeste(int tam, double* vals) {
54.   for (int i = 0; i < linhas; i++) valores[((colunas
    + 2) * (i + 1))] = vals[i];
55.   tentaCalcular();
56. }
57. void Jacobi::dadosNorte(int tam, double* vals) {
58.   memcpy(&(valores[1]), vals, sizeof(double) *
    tam);
59.   tentaCalcular();
60. }
61. void Jacobi::dadosSul(int tam, double* vals) {
62.   memcpy(&(valores[((colunas + 2) * (linhas +
    1)) + 1]), vals, sizeof(double) * tam);
63.   tentaCalcular();
64. }

65. void Jacobi::tentaCalcular() {
66.   chamadas++;
67.   if (chamadas >= alvoChamadas) {
68.     chamadas = 0;
69.     double difMax = calcula();
70.     mainProxy.recebeDiferenca(difMax);
71.   }
72. }
```

```
73. double Jacobi::calcula() {
74.   memset(temp, 0, sizeof(double) * ((colunas +
    2) * (linhas + 2)));
75.   int inicioX = 1;
76.   if (thisIndex.x == 0 && thisIndex.y == 0) {
77.     inicioX = 2;
78.     temp[(colunas + 2) + 1] = 1.0;
79.   }

80.   double difMax = 0.0;
81.   for (int y = 1; y <= linhas; y++) {
82.     for (int x = inicioX; x <= colunas; x++) {
83.       int indice = (y * (colunas + 2)) + x;
84.       double newVal = valores[indice];
85.       newVal += valores[indice - 1];
86.       newVal += valores[indice + 1];
87.       newVal += valores[indice - (colunas + 2)];
88.       newVal += valores[indice + (colunas + 2)];
89.       temp[indice] = newVal / 5.0;
90.       double diferenca = temp[indice] -
    valores[indice];
91.       if (diferenca < 0) diferenca *= -1.0;
92.       difMax = ((difMax > diferenca) ? (difMax) :
    (diferenca));
93.     }
94.     inicioX = 1;
95.   }
96.   double* pontTemp = valores;
97.   valores = temp;
98.   temp = pontTemp;
99.   return difMax;
100. }

101. #include "jacobi.def.h"
```

Figura 2.12: Programa Jacobi2D: segunda parte da implementação da matriz de chares.

verifica em quais direções ele tem vizinhos e envia mensagens para estes com as células de suas linhas ou colunas de fronteira. Ao fim do método, o chare invoca seu método privado `tentaCalcular()`, o qual verifica se a etapa inicial da iteração já foi executada e se todas as mensagens foram recebidas. No momento em que todas essas dependências estão resolvidas, o método `calcula()` é chamado para atualizar as células locais e obter a maior variação entre elas, sendo essa variação enviada para o *main chare* através do método `recebeDiferenca()`.

A implementação do módulo do *main chare* é ilustrada na Figura 2.13. O método construtor da classe chare é responsável pela inicialização de variáveis globais que definem o tamanho das matrizes internas e as dimensões do vetor de chares, assim como a criação deste vetor. O método `iniciaEtapa()` inicializa o contador de mensagens recebidas de chares `Jacobi` e invoca o método que inicia a iteração nesses chares através de uma mensagem em *broadcast* (o método é invocado em todos os chares do vetor). Por fim, o método `recebeDiferenca()` armazena a maior diferença recebida entre todos os chares e termina a execução da aplicação caso essa diferença seja menor do que o limiar pré-estabelecido.

2.6. Conclusão

Este curso introduziu o paradigma de programação paralela CHARM++ e caracterizou seu uso através de dois exemplos simples. Através do curso, o leitor aprendeu que o ambiente de CHARM++ é baseado nos conceitos de execução direcionada por mensagens assíncronas, sobredecomposição e migrabilidade, e que essas características o possibilitam desenvolver aplicações paralelas que executarão de forma eficiente mesmo em situações sujeitas a variabilidade de recursos e de cargas computacionais graças aos mecanismos de balanceamento de carga e *checkpointing*.

Devido à natureza introdutória do curso, diversos aspectos interessantes e avançados de CHARM++ não foram abordados, tais como a programação com chamadas de retorno e reduções, ou o desenvolvimento de algoritmos de balanceamento de carga através do arcabouço disponibilizado. Além disso, o ambiente de CHARM++ também disponibiliza uma implementação da API MPI chamada ADAPTIVE MPI que se beneficia do mesmo sistema de execução (com os mesmos mecanismos avançados). CHARM++ também conta com um simulador e emulador de ambientes paralelos chamado BIGSIM. O estudo de tais aspectos e funcionalidades adicionais são deixados a cargo do leitor capacitado através deste curso.

2.7. Bibliografia

[Acun et al. 2014] Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., Wesolowski, L., and Kale, L. (2014). Parallel Programming with Migratable Objects: Charm++ in Practice. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '14, New York, NY, USA. ACM.

[Armstrong 2007] Armstrong, J. (2007). *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf.

[Dagum and Menon 2002] Dagum, L. and Menon, R. (2002). OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55.

[Deitel and Deitel 2015] Deitel, H. M. and Deitel, H. (2015). *Java: How to Program (Late Objects)*. Pearson Education Limited.

[Gropp et al. 1999] Gropp, W., Lusk, E., and Skjellum, A. (1999). Using MPI: portable parallel programming with the message-passing interface. *MIT Press Cambridge, MA, USA*, page 371.

[Kale 2015] Kale, L. (2015). Parallel Programming Laboratory website. http://charm.cs.uiuc.edu/. Acessado em fevereiro de 2015.

[Kale and Krishnan 1993] Kale, L. and Krishnan, S. (1993). CHARM++: A Portable Concurrent Object Oriented System Based on C++. In Paepcke, A., editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press.

[Laboratory 2015] Laboratory, P. P. (2015). Charm++ Programming Tutorial. http://charm.cs.illinois.edu/tutorial/. Acessado em fevereiro de 2015.

[Meneses et al. 2014] Meneses, E., Ni, X., Zheng, G., Mendes, C. L., and Kale, L. V. (2014). Using migratable objects to enhance fault tolerance schemes in supercomputers. In *IEEE Transactions on Parallel and Distributed Systems*.

[Phillips et al. 2005] Phillips, J. C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R. D., Kale, L., and Schulten, K. (2005). Scalable molecular dynamics with namd. *Journal of computational chemistry*, 26(16):1781–1802.

[Pilla et al. 2014] Pilla, L. L., Ribeiro, C. P., Coucheney, P., Broquedis, F., Gaujal, B., Navaux, P. O. A., and Méhaut, J.-F. (2014). A Topology-Aware Load Balancing Algorithm for Clustered Hierarchical Multi-Core Machines. *Future Generation Computer Systems*, 30(0):191–201.

[Sinha and Kale 1993] Sinha, A. and Kale, L. (1993). A load balancing strategy for prioritized execution of tasks. In *Seventh International Parallel Processing Symposium*, pages 230–237, Newport Beach, CA.

[Zheng et al. 2004] Zheng, G., Shi, L., and Kalé, L. V. (2004). FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *2004 IEEE Cluster*, pages 93–103, San Diego, CA.

main.C

```
1.  #include "main.h"
2.  #include "jacobi.decl.h"

3.  /* readonly */ CProxy_Main mainProxy;
4.  /* readonly */ int linhas;
5.  /* readonly */ int colunas;
6.  /* readonly */ int linhasChares;
7.  /* readonly */ int colunasChares;

8.  Main::Main(CkMigrateMessage* msg) { }
9.  void Main::iniciaEtapa() {
10.   difMax = 0.0;
11.   recebimentos = 0;
12.   vetorJacobi.iniciaEtapa();
13. }

14. void Main::recebeDiferenca(double valorEnviado) {
15.   difMax = ((valorEnviado > difMax) ? (valorEnviado) :
    (difMax));
16.   recebimentos++;
17.   if (recebimentos >= (linhasChares * colunasChares)) {
18.     CkPrintf("Main::recebeDiferenca() -> difMax =
    %lf\n", difMax);
19.     if (difMax > limiar)
20.       iniciaEtapa();
21.     else
22.       CkExit();
23.   }
24. }
25. Main::Main(CkArgMsg* msg) {
26.   delete msg;

27.   linhas = 10;
28.   colunas = 10;
29.   linhasChares = 5;
30.   colunasChares = 5;
31.   limiar = 0.005;

32.   CkPrintf("Executando \"Jacobi2D\" em %d PEs com
    uma matriz de valores de tamanho %dx%d
    particionada em %dx%d chares e com um limiar de
    %lf.\n", CkNumPes(), linhas * linhasChares, colunas *
    colunasChares, linhasChares, colunasChares, limiar);

33.   mainProxy = thisProxy;
34.   vetorJacobi = CProxy_Jacobi::ckNew(linhasChares,
    colunasChares);
35.   iniciaEtapa();
36. }
37. #include "main.def.h"
```

main.h

```
38. #ifndef __MAIN_H__
39. #define __MAIN_H__

40. #include "main.decl.h"
41. #include "jacobi.decl.h"

42. class Main : public CBase_Main {
43.   private:
44.     double limiar;
45.     int recebimentos;
46.     double difMax;
47.     CProxy_Jacobi vetorJacobi;
48.     void iniciaEtapa();

49.   public:
50.     Main(CkArgMsg* msg);
51.     Main(CkMigrateMessage* msg);
52.     void recebeDiferenca(double
    valorEnviado);
53. };

54. #endif //__MAIN_H__
```

main.ci

```
55. mainmodule main {
56.   readonly CProxy_Main mainProxy;
57.   readonly int linhas;
58.   readonly int colunas;
59.   readonly int linhasChares;
60.   readonly int colunasChares;

61.   extern module jacobi;

62.   mainchare Main {
63.     entry Main(CkArgMsg* msg);
64.     entry void recebeDiferenca(double);
65.   };
66. };
```

Figura 2.13: Programa Jacobi2D: implementação do módulo do *main chare*.