ERAD 2015

Programação Paralela em Charm++

Prof. Dr. Laércio Lima Pilla

Universidade Federal de Santa Catarina

Prof. Dr. Esteban Meneses

University of Pittsburgh

Autores

- **Laércio Lima Pilla**
  - Prof. UFSC
    - Doutor UFRGS + Université de Grenoble
    - Bacharel UFRGS

- **Esteban Meneses**
  - Prof. University of Pittsburgh
    - Doutor University of Illinois at Urbana-Champaign
    - Bacharel e Mestre Instituto Tecnológico de Costa Rica

Agenda

1. Introdução a Charm++
2. Modelo de Programação
3. Modelo de Execução
4. Exemplo 1: Olá Mundo
5. Sistema de Execução
6. Exemplo 2: Jacobi2D
7. Conclusões


1. INTRODUÇÃO A CHARM++

1. Introdução a Charm++

- **Arquitetura de sistemas atuais e futuros**
  - **Exascale**
    - Supercomputadores com EFLOPS (10^{18})
    - Computadores institucionais com PFLOPS (10^{15})
    - Computadores domésticos com TFLOPS (10^{12})

1. Introdução a Charm++

- **Arquitetura de sistemas atuais e futuros**
  - **Processadores paralelos**
    - Múltiplos núcleos
    - Frequências que não aumentam
  - **Altos custos de memória**
    - Pouca memória primária ou cache por núcleo
      - Xeon Phi 5100: 133 MB/núcleo
      - Power BGQ: 1 GB/núcleo

1. Introdução a Charm++

- **Arquitetura de sistemas atuais e futuros**
  - **Crescente heterogeneidade**
    - Aceleradores paralelos, núcleos dedicados, NUMA
  - **Limitações energéticas**
    - Preocupações com energia, potência e calor
  - **Desbalanceamento de carga**
    - Núcleos mais carregados ditam o ritmo das aplicações
  - **Falhas de componentes mais frequentes**
    - Sistemas maiores, menores MTBF

1. Introdução a Charm++

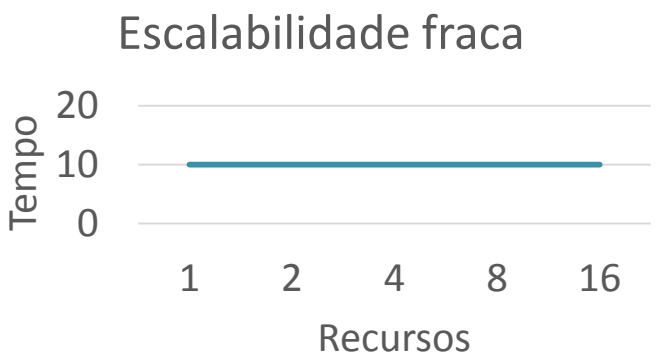
- **Exemplos de plataformas paralelas**
  - Três supercomputadores de maior desempenho segundo Top500
    - **Tianhe-2**: **3.120.000 núcleos** (Xeon + Xeon Phi)
    - **Titan**: **560.640 núcleos** (Opteron + Nvidia K20x)
    - **Sequoia**: **1.572.864 núcleos** (Power BQC)

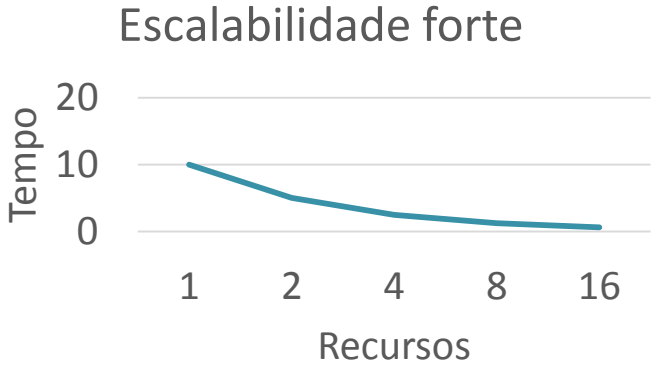
1. Introdução a Charm++

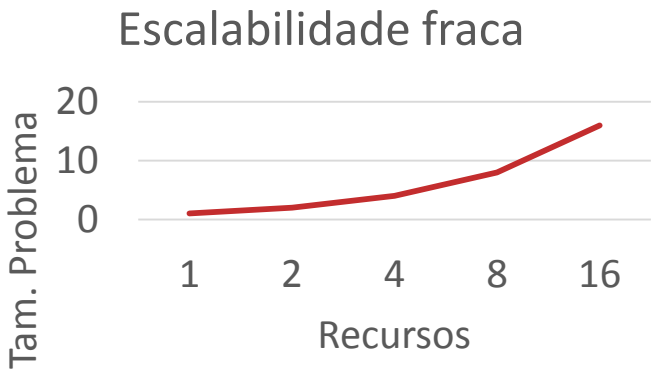
- **Implicações**
  - Cada fluxo de execução precisará
    - **Trabalhar com menos dados**
    - **Esperar mais por dados remotos**
  - **Necessidade de escalabilidade forte**

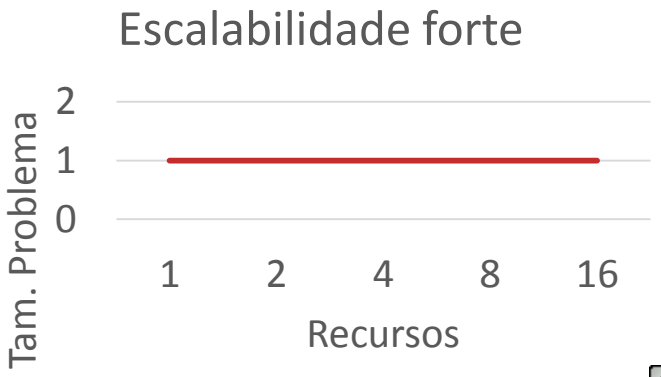
1. Introdução a Charm++

- **Escalabilidade**
  - **Fraca**: maior problema para maior plataforma
  - **Forte**: mesmo problema para maior plataforma









1. Introdução a Charm++

- **Como programar para essas plataformas?**
  - **OpenMP? MPI?**
    - Bons para aplicações regulares
    - Bons para ambientes homogêneos
    - Sofrem com
      - **Ambientes voláteis**
      - **Ambientes heterogêneos**
      - **Aplicações irregulares ou dinâmicas**

1. Introdução a Charm++

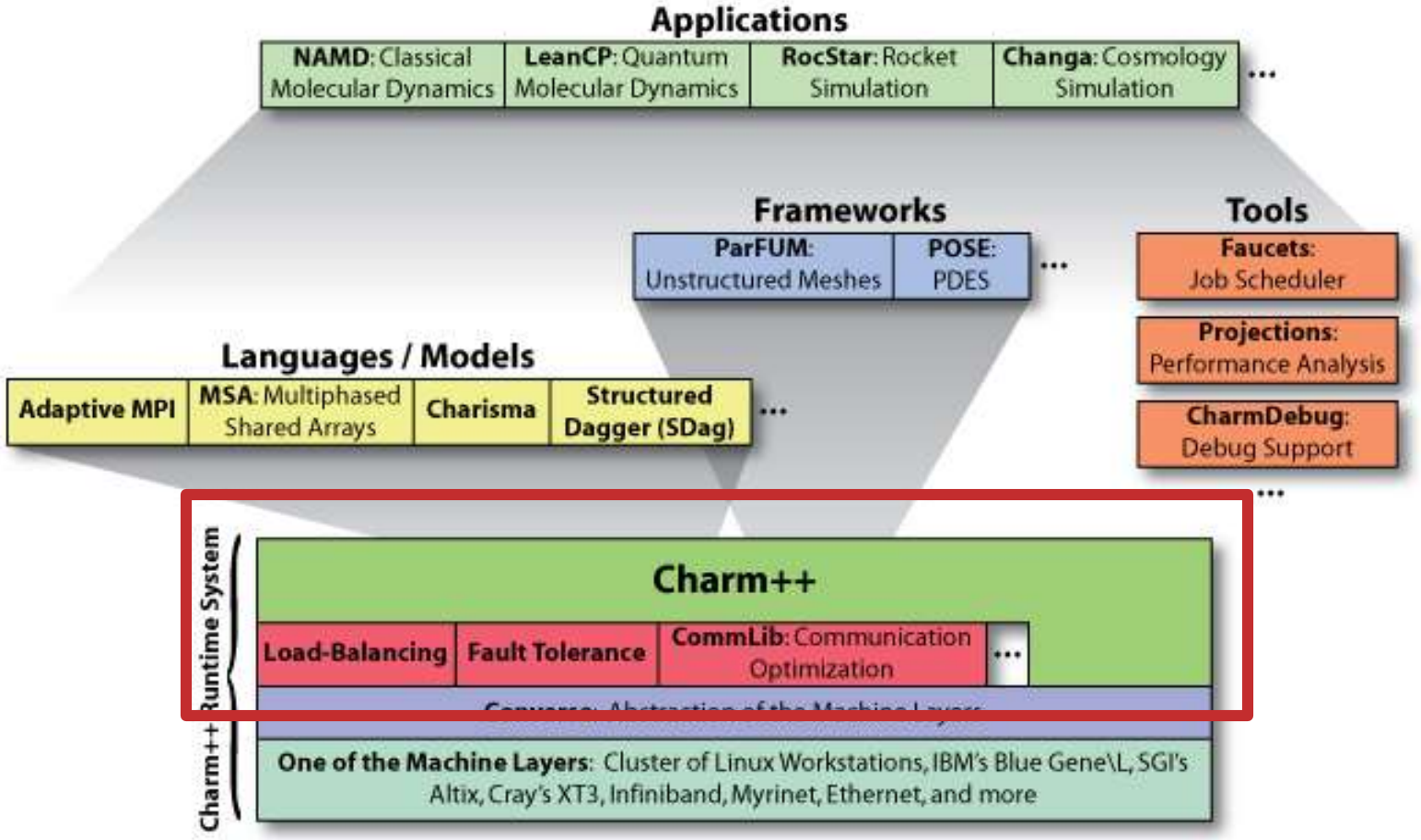
- **Charm++**
  - Paradigma de programação paralela
  - Desenvolvido desde 1993 pelo PPL – UIUC
  - Construído sobre a ideia de **objetos**
    - Identificam as entidades sendo simuladas
    - Definem o trabalho a ser feito
    - Usam classes em C++
    - Interação através de chamadas de métodos

1. Introdução a Charm++

- **Charm++**
  - **Sistema de execução (RTS)** elaborado
    - Balanceamento de carga
    - *Checkpointing*
    - Ambientes elásticos
    - Bibliotecas de otimização de comunicação

1. Introdução a Charm++

- **Ambiente de Charm++**



Fonte: http://charm.cs.illinois.edu/tutorial/CharmRuntimeSystem.htm

1. Introdução a Charm++

- Exemplo de uso
  - **Dinâmica molecular**
    - Simulação do cápside do vírus HIV
    - Possibilidade de desenvolvimento de remédios

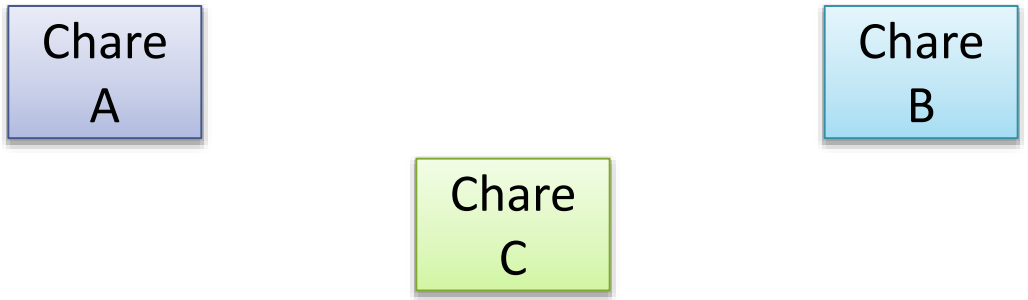
1. Introdução a Charm++

- **Objetivo do curso**
  - **Visão introdutória de Charm++**
    - Permitir o desenvolvimento e entendimento de códigos simples
    - Possibilitar o uso de funcionalidades mais elaboradas

2. MODELO DE PROGRAMAÇÃO

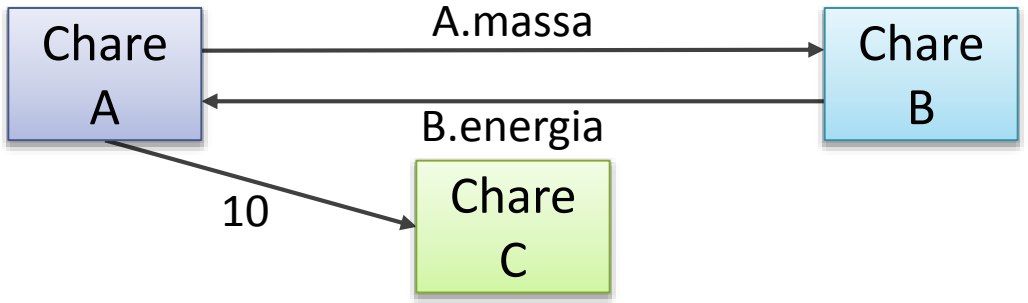

2. Modelo de Programação

- ***Chares***
  - Objetos básicos de Charm++
    - **Encapsulam dados**
    - **Métodos representam computação sobre os dados**
      - Execução de métodos é sequencial
    - ***Espaço global de objetos***: conjunto de todos chares de uma aplicação



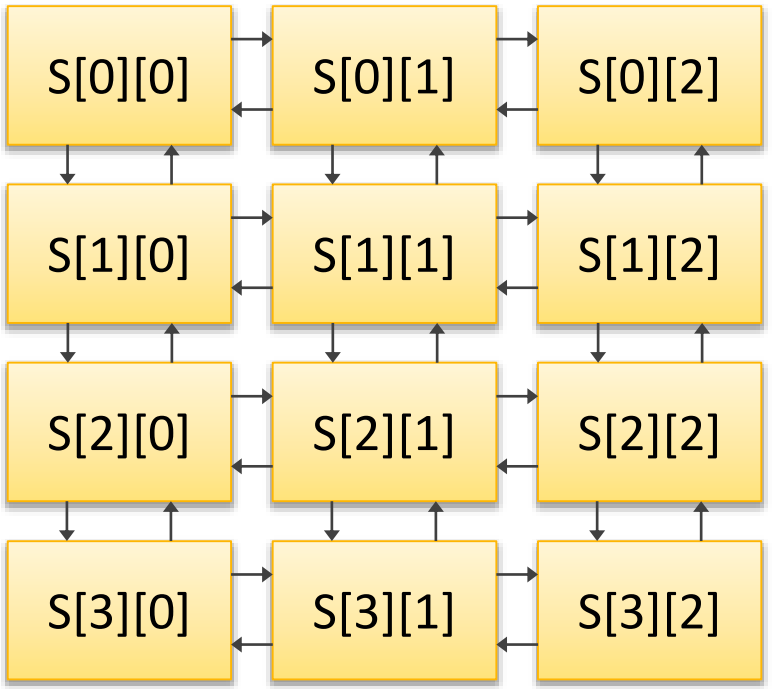
2. Modelo de Programação

- ***Chares***
  - Dados e estados privados
  - **Comunicação através de troca de mensagens**
    - Comunicação com qualquer chare conhecido no espaço global de objetos
      - **Inclusive consigo mesmo**



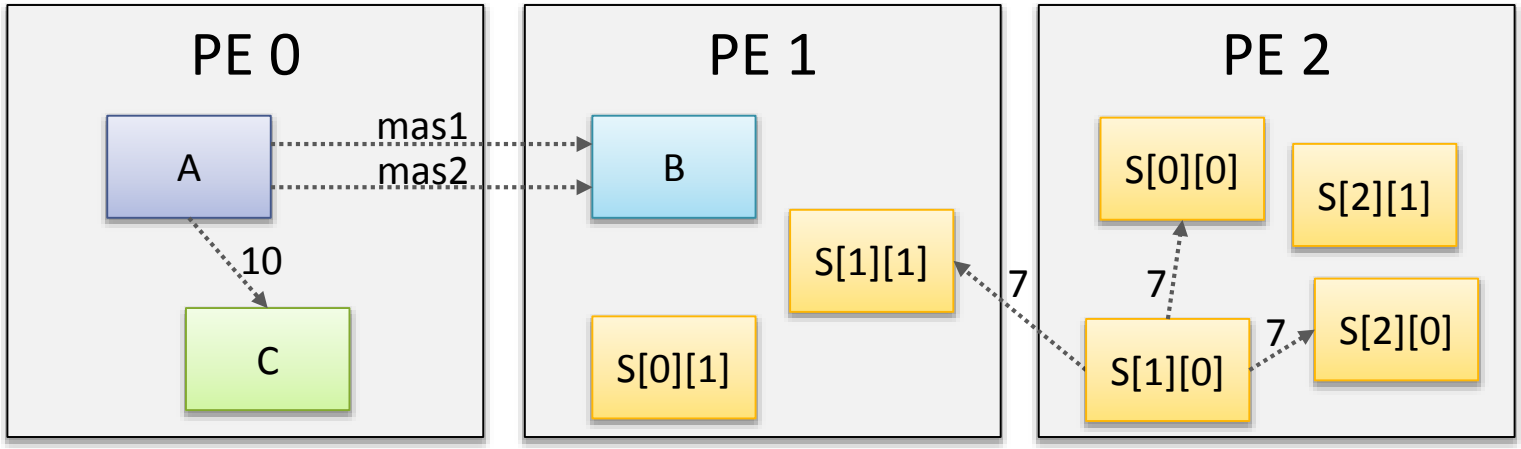
2. Modelo de Programação

- ***Vetores de chares***
  - Organização de chares através de índices lógicos
    - **Como vetores simples em C++**
    - Uma a seis dimensões
    - Conhecimento da própria posição
      - `thisIndex.{x,y}` p/ 2D



2. Modelo de Programação

- **Execução**
  - Chares executam sobre ***Elementos de Processamento (PEs)***
    - Mapeamento e quantidades independentes
  - ***Main chare*** organiza a execução da aplicação



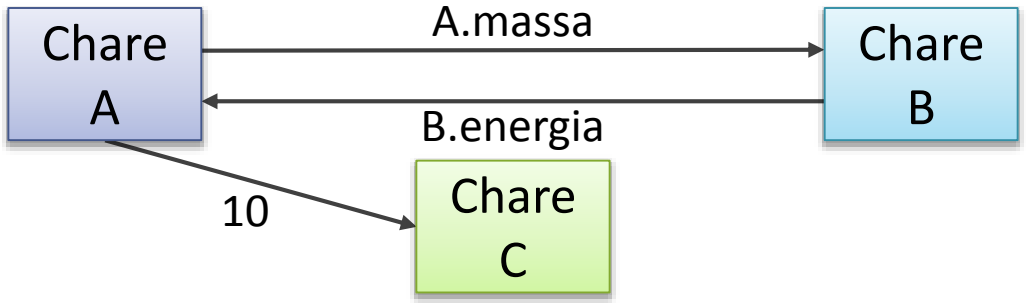
2. Modelo de Programação

- **Troca de mensagens**
  - ***Chamadas remotas de métodos***

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

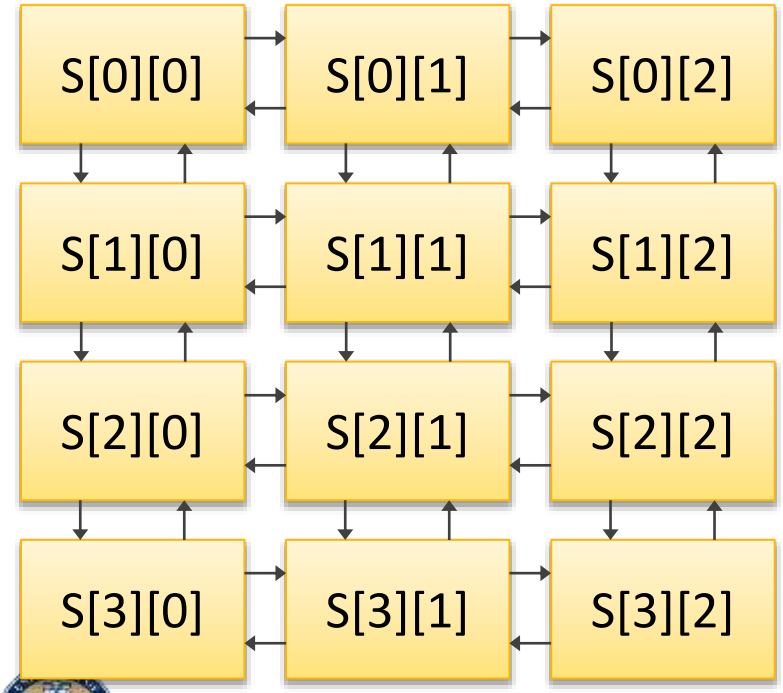
```
Chare B{
  int energia;
  void recebeDado (int dado) { ...
    A.recebeResposta (energia);
  ... } }
```

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



2. Modelo de Programação

- **Troca de mensagens em vetores de chares**
  - `thisProxy(thisIndex.x, thisIndex.y+1).recebeOeste(…);`




```
class Superficie{ ...
  void recebeNorte (...) { ... }
  void recebeSul (...) { ... }
  void recebeLeste (...) { ... }
  void recebeOeste (...) { ... }
}
```

```
void main (...) { ...
  CProxy_Superficie S =
CProxy_Superficie::ckNew(3,4);
... }
```

2. Modelo de Programação

- ***Métodos de entrada***
  - Métodos que podem ser chamados por outros chares
- Troca de mensagens ***assíncrona***
  - Chamada de método não espera por resposta

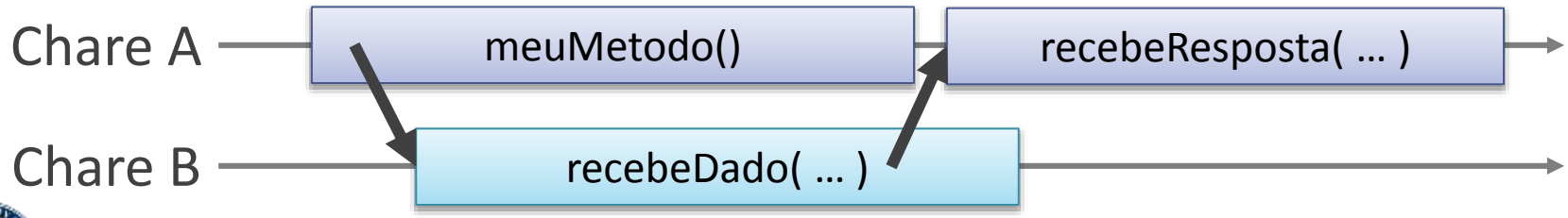
3. MODELO DE EXECUÇÃO

3. Modelo de Execução

- Três preceitos que guiam Charm++
  - **Execução direcionada por mensagens assíncronas**
  - **Sobredecomposição**
  - **Migrabilidade**

3. Modelo de Execução

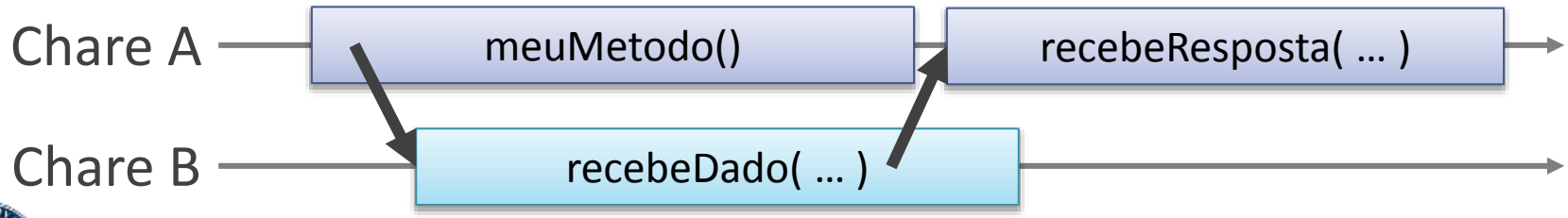
- **Execução direcionada por mensagens assíncronas**
  - **Chares somente são escalonados ao receberem mensagens**
  - Fluxo de controle da aplicação não é predeterminado
    - **Maior concorrência**



3. Modelo de Execução

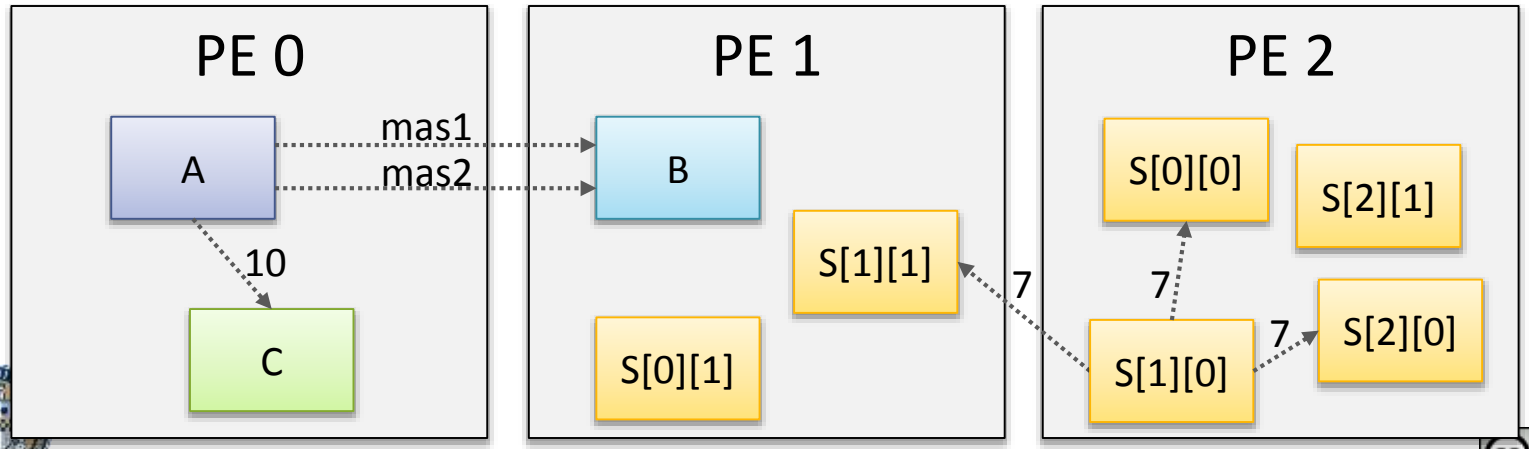
Execução direcionada por mensagens assíncronas

- **Vantagens**
  - Remove sincronização para comunicação
  - **Esconde latência de comunicação**
    - Escalona-se outro chare enquanto se espera
    - Organização de uma fila de mensagens p/ escalonamento
    - **Sobreposição de comunicação e computação**



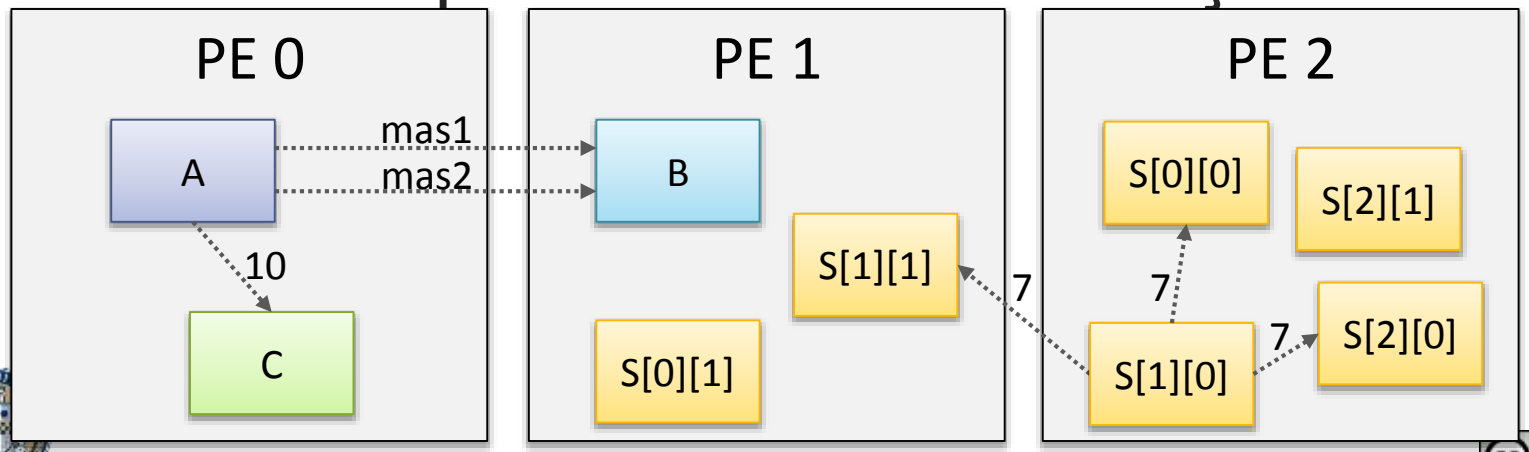
3. Modelo de Execução

- **Sobredecomposição**
  - Particionamento da aplicação em **mais chares do que núcleos disponíveis**
    - Números independentes
  - Definição de aplicações em termos de entidades lógicas



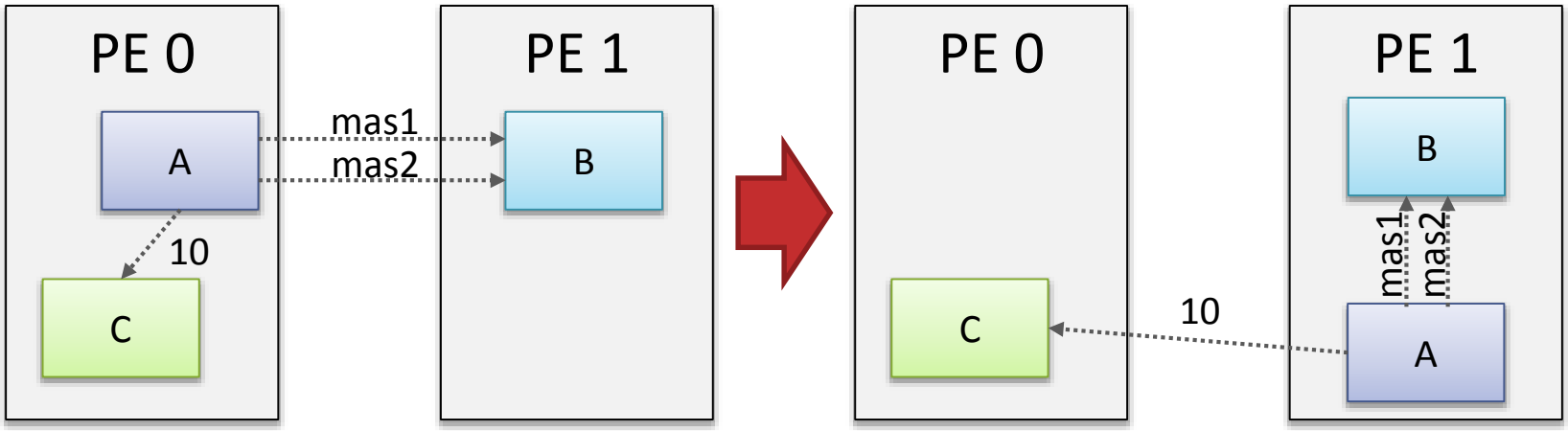
3. Modelo de Execução

- **Vantagens**
  - Maior modularidade
  - Esconde latência de comunicação
    - **Escalonam-se chares prontos para execução**
  - Possibilita **gerenciamento do mapeamento de chares** pelo sistema de execução



3. Modelo de Execução

- **Migrabilidade**
  - **Capacidade de mover chares entre núcleos**
    - Necessita de métodos de empacotamento/desempacotamento
      - **Interface disponibilizada por Charm++**



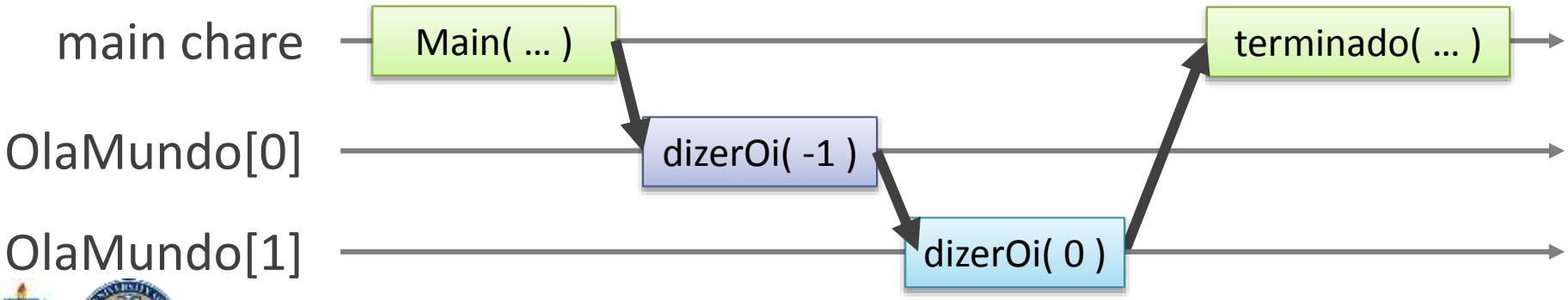
3. Modelo de Execução

- **Vantagens**
  - Possibilita **balanceamento de carga**
    - Otimização de comunicação, carga, energia…
  - Possibilita **checkpoint/restart**
    - Recuperação da execução colocando chares em outros recursos
  - Possibilita **uso de recursos elásticos**
    - Migração de chares quando número de recursos muda

4. EXEMPLO 1: OLÁ MUNDO


4. Exemplo 1: Olá Mundo

- ***Hello world* com vetor de chares**
  - Código organizado em módulos
    - **Módulo OlaMundo**
      - Chares responsáveis pela impressão de mensagens
    - **Módulo Main**
      - Main chare responsável pela inicialização e finalização do programa



4. Exemplo 1: Olá Mundo

Olá Mundo!

- **Tipos de arquivos em um módulo**
  - **.C**: implementação de classes
  - **.h**: cabeçalhos e definições
  - **.ci**: definição das classes chare, métodos de entrada e variáveis globais

4. Exemplo 1: Olá Mundo

Módulo OlaMundo

- **OlaMundo.ci**
  - **module**: define o módulo
  - **array[1D]**: classe de vetor de chares 1D
  - **entry**: métodos de entrada

```
module OlaMundo {
  array [1D] OlaMundo {
    entry OlaMundo();
    entry void dizerOi(int);
  };
};
```


4. Exemplo 1: Olá Mundo

Módulo OlaMundo

- **OlaMundo.h**
  - **CBase_Classe**: classe gerada durante a compilação que encapsula a classe chare

```
class OlaMundo : public CBase_OlaMundo {
  public:
    OlaMundo ();
    void dizerOi(int emissor);
};
```

4. Exemplo 1: Olá Mundo

Módulo OlaMundo

- **OlaMundo.C (1/3)**
  - **Extern /* readonly */**: variáveis globais definidas em outro modulo
  - **Classe.decl.h**: gerado durante a compilação

```
#include "OlaMundo.decl.h"
#include "OlaMundo.h"
#include "main.decl.h"

extern /* readonly */ CProxy_Main mainProxy;
extern /* readonly */ int numChares;

OlaMundo::OlaMundo () { }
```

4. Exemplo 1: Olá Mundo

Módulo OlaMundo

- **OlaMundo.C (2/3)**
  - **Proxies**: manequins de chares p/ comunicação
  - **thisIndex**: índice do chare no vetor de chares

```
void OlaMundo::dizerOi (int emissor) {
  CkPrintf("\"Ola mundo\" do chare %d no PE %d
(pedido pelo chare %d).\n", thisIndex, CkMyPe(),
emissor);
  if (thisIndex < (numChares - 1))
    thisProxy[thisIndex + 1].dizerOi(thisIndex);
  else mainProxy.terminado();
}
#include "OlaMundo.def.h"
```

4. Exemplo 1: Olá Mundo

- **OlaMundo.C (3/3)**
  - **Classe.def.h**: gerado durante a compilação
  - **CkMyPe()**: retorna PE onde o chare está

```
void OlaMundo::dizerOi (int emissor) {
  CkPrintf("\"Ola mundo\" do chare %d no PE %d
(pedido pelo chare %d).\n", thisIndex, CkMyPe(),
emissor);
  if (thisIndex < (numChares - 1))
    thisProxy[thisIndex + 1].dizerOi(thisIndex);
  else mainProxy.terminado();
}
#include "OlaMundo.def.h"
```

4. Exemplo 1: Olá Mundo

Módulo main

- **main.ci**
  - **Mainmodule, mainchare**
  - **Readonly**: variáveis globais

```
mainmodule main {
  readonly CProxy_Main mainProxy;
  readonly int numChares;
  extern module OlaMundo;

  mainchare Main {
    entry Main (CkArgMsg* msg);
    entry void terminado ();
  };
};
```

4. Exemplo 1: Olá Mundo

Módulo main

- **main.h**
  - Todos métodos de entrada são públicos
  - **CkArgMsg***: parâmetros passados para a execução do programa

```
class Main : public CBase_Main {
  public:
    Main (CkArgMsg* msg);
    void terminado();
};
```

4. Exemplo 1: Olá Mundo

Módulo main

- **main.C (1/3)**
  - **CkExit**: finalização de programa Charm++

```
#include "main.decl.h"
#include "main.h"
#include "OlaMundo.decl.h"

/* readonly */ CProxy_Main mainProxy;
/* readonly */ int numChares;
[…]
void Main::terminado () {
        CkExit ();
}

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

4. Exemplo 1: Olá Mundo

- **main.C (2/3)**
  - **mainProxy**: comunicação dos chares c/ main

```
Main::Main (CkArgMsg* msg) {
  numChares = 7;
  if (msg->argc > 1) numChares = atoi (msg->argv[1]);
  delete msg;
  CkPrintf ("Rodando o programa \"Ola Mundo\" com %d
chares em %d unidades de processamento.\n", numChares,
CkNumPes());
  mainProxy = thisProxy;
  CProxy_OlaMundo OlaMundoArray = CProxy_OlaMundo::ckNew
(numChares);
  OlaMundoArray[0].dizerOi (-1);
}
```

4. Exemplo 1: Olá Mundo

Módulo main

- **main.C (3/3)**
  - **CProxy_Classe::ckNew**: instanciação de chares

```
Main::Main (CkArgMsg* msg) {
  numChares = 7;
  if (msg->argc > 1) numChares = atoi (msg->argv[1]);
  delete msg;
  CkPrintf ("Rodando o programa \"Ola Mundo\" com %d
chares em %d unidades de processamento.\n", numChares,
CkNumPes());
  mainProxy = thisProxy;
  CProxy_OlaMundo OlaMundoArray = CProxy_OlaMundo::ckNew
(numChares);
  OlaMundoArray[0].dizerOi (-1);
}
```

4. Exemplo 1: Olá Mundo

- **Compilação**
  - **charmc**: abstração do compilador nativo
  - Primeiro passo: arquivos .ci
    - Geração de .def.h e .decl.h

```
charmc main.ci
charmc OlaMundo.ci
charmc -o main.o main.C
charmc -o OlaMundo.o OlaMundo.C
charmc -language charm++ -o OlaMundo main.o OlaMundo.o
```

4. Exemplo 1: Olá Mundo

- **Exemplo de execução**
  - **./OlaMundo +p 2 5**
    - **+p 2**: dois elementos de processamento
    - **5**: Número de chares OlaMundo

```
Rodando o programa "Ola Mundo" com 5 chares em 2 unidades
de processamento.
"Ola mundo" do chare 0 no PE 0 (pedido pelo chare -1).
"Ola mundo" do chare 1 no PE 0 (pedido pelo chare 0).
"Ola mundo" do chare 2 no PE 0 (pedido pelo chare 1).
"Ola mundo" do chare 3 no PE 1 (pedido pelo chare 2).
"Ola mundo" do chare 4 no PE 1 (pedido pelo chare 3).
Program finished.
```

5. SISTEMA DE EXECUÇÃO

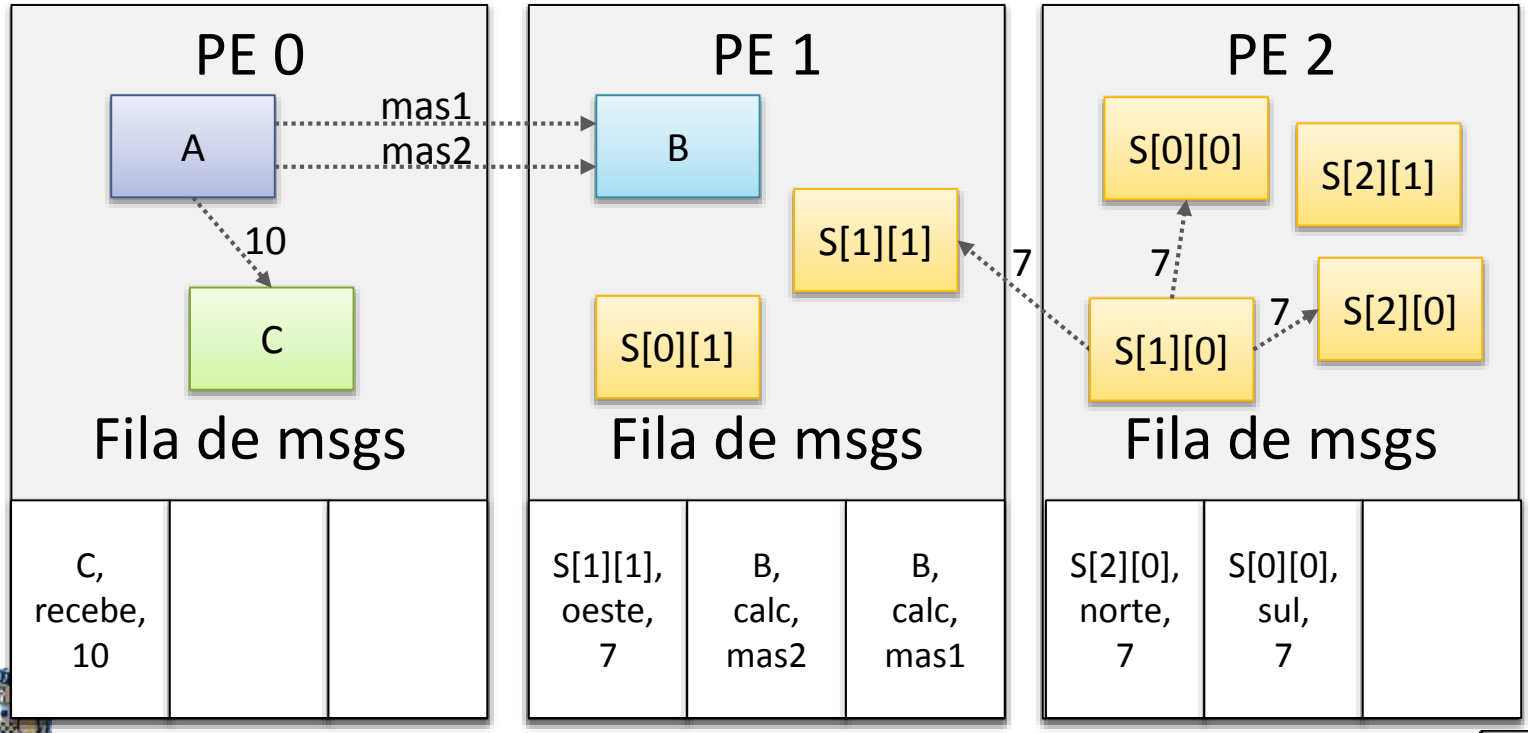
5. Sistema de Execução

Tópicos

- **Sistema de execução**
  - Filas de mensagem e escalonamento
  - Roteamento de mensagens
  - Serialização de objetos
  - Balanceamento de carga
  - *Checkpointing*

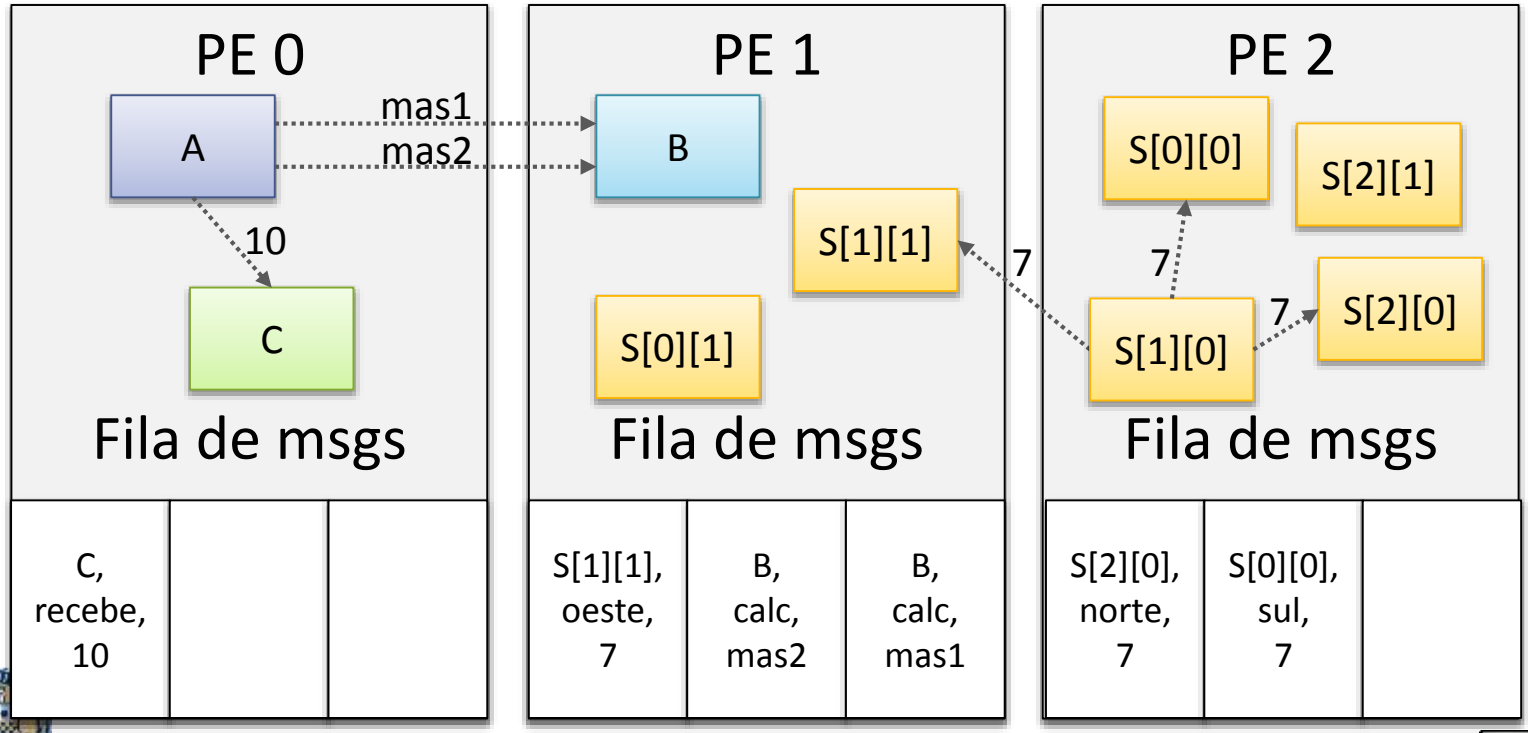
5. Sistema de Execução

- **Filas de mensagens e escalonamento**
  - **Chares executam apenas após** algum de seus métodos de entrada **terem sido invocados**



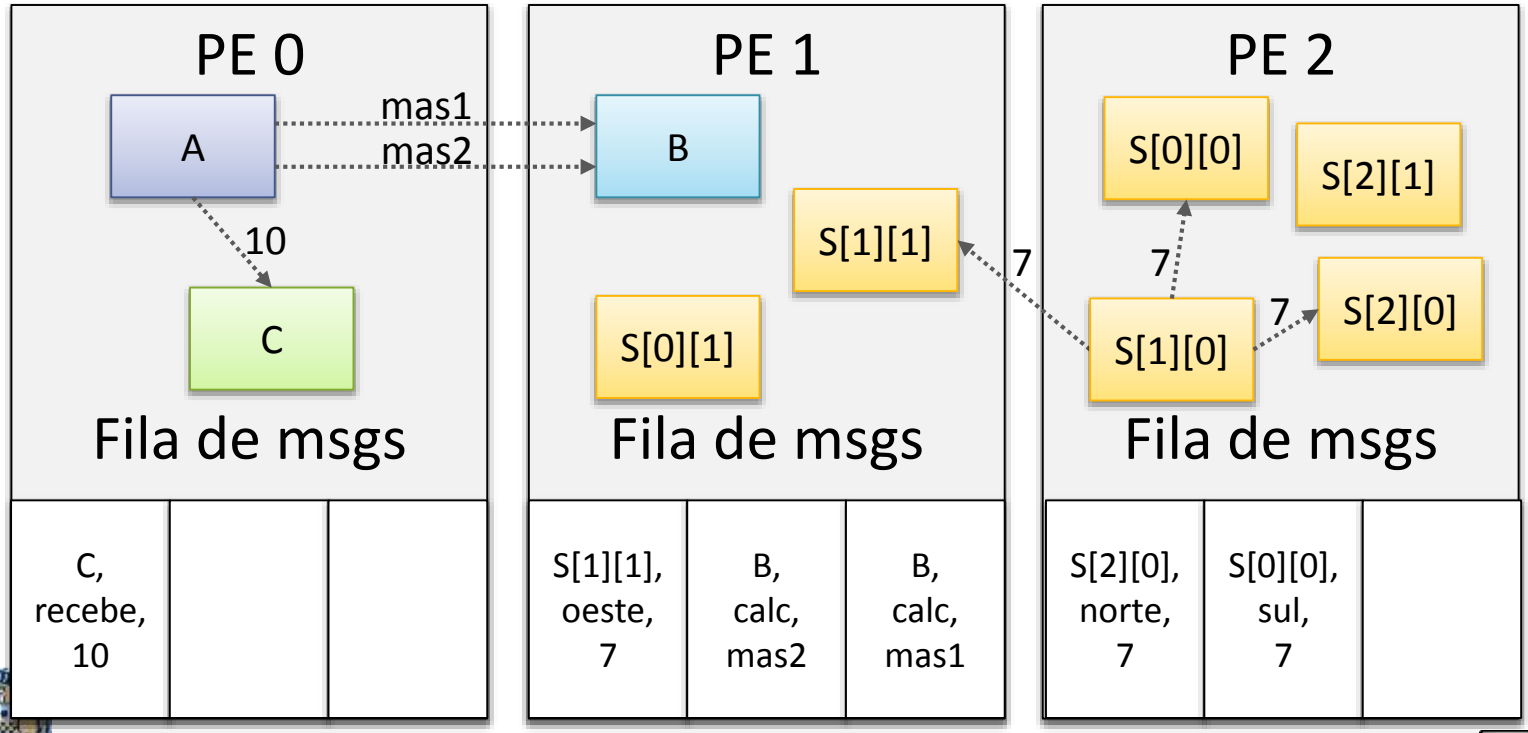
5. Sistema de Execução

- **Escalonamento local**
  - **Apenas um chare executa em um PE** e sua execução não pode ser preemptada



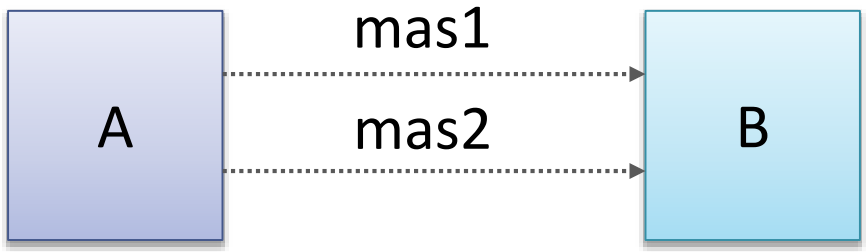
5. Sistema de Execução

- **Escalonamento local**
  - Fila de mensagens guia qual será o próximo chare a executar



5. Sistema de Execução

- **Chamadas de métodos**
  - **Não há garantia de ordem de execução**
    - Possíveis execuções
      - B.calc(mas1), B.calc(mas2)
      - B.calc(mas2), B.calc(mas1)



5. Sistema de Execução

- **Roteamento de mensagens**
  - Comunicação na plataforma paralela é abstraída
    - **Programador não precisa saber onde um chare será mapeado**
    - Chares podem vir a ser migrados
  - **Sistema de execução é responsável pelo encaminhamento das mensagens**
    - Núcleo original sabe onde o chare se encontra

5. Sistema de Execução

- **Roteamento de mensagens**
  - Chares não enxergam uns aos outros diretamente
    - Comunicação feita sempre através de **proxies**

```
CProxy_Main mainProxy;
CProxy_OlaMundo OlaMundoArray =
CProxy_OlaMundo::ckNew (numChares);
thisProxy[thisIndex+1].dizerOi(thisIndex);
```

5. Sistema de Execução

- **Serialização de objetos**
  - Arcabouço com múltiplos usos
    - **Serialização de mensagens**
    - **Migração de chares**
    - **Checkpointing**
  - **Métodos PUP**
    - *Pack and unpack*
    - Usados tanto no empacotamento quanto no desempacotamento de objetos

5. Sistema de Execução

- **Método pup**
  - Comando |
  - PUParray

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

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

5. Sistema de Execução

- **Balanceamento de carga**
  - **Arcabouço externo a aplicações** para o balanceamento de carga
    - **Definição de algoritmos para o remapeamento de chares dado um objetivo**
      - Melhorar comunicação
      - Reduzir o consumo de energia ou temperatura de núcleos
      - Equilibrar o tempo de execução nos vários núcleos

5. Sistema de Execução

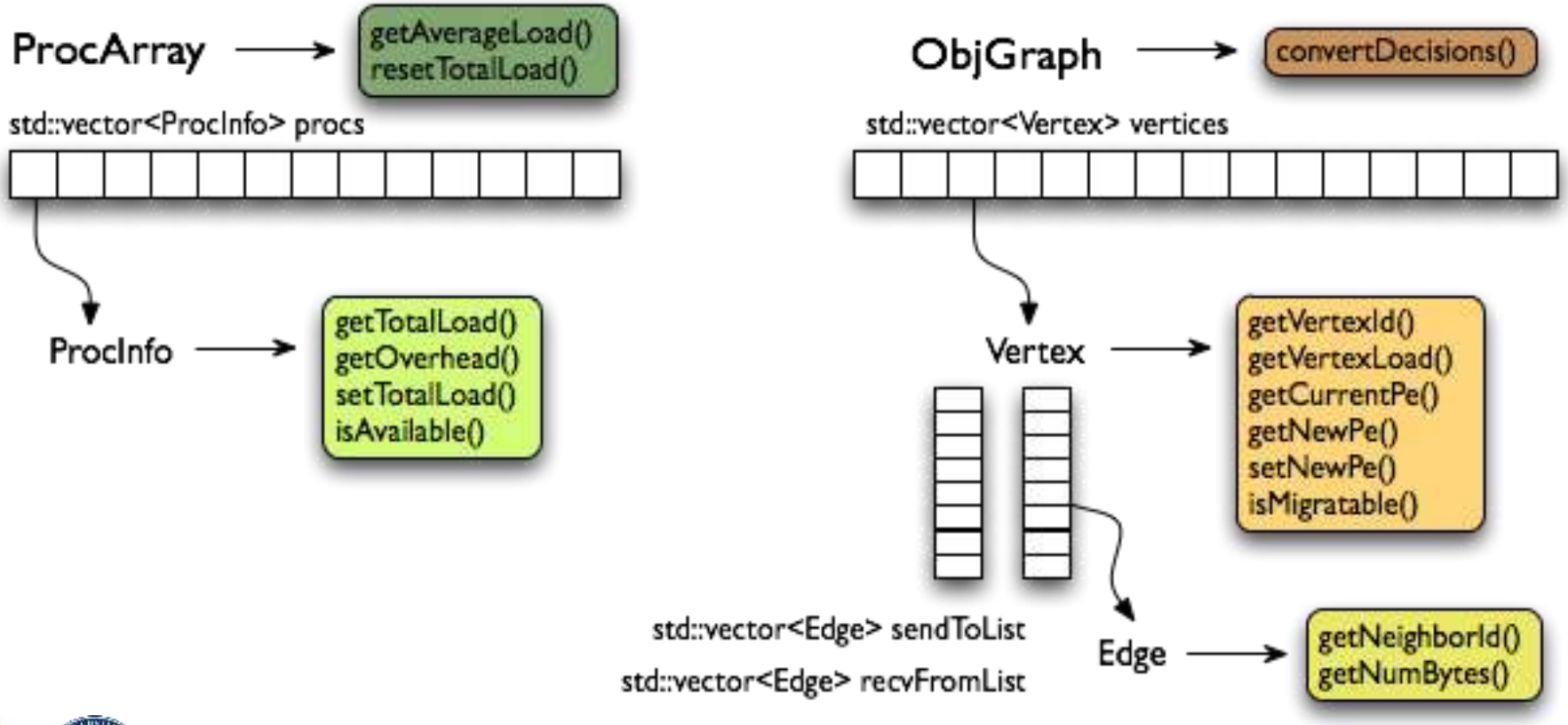
- ***Princípio da persistência***
  - A carga computacional e padrões de comunicação de aplicações científicas tendem a persistir ao longo do tempo
  - **O passado recente é uma boa estimativa do futuro próximo**
    - Informações das aplicações são coletadas e usadas para a tomada de decisões de escalonamento global

5. Sistema de Execução

- **Mecanismos de balanceamento de carga**
  - **Balanceamento de carga periódico**
    - Temporal: +LBPeriod 10.0
    - Iterações
      - Chamada de método `AtSync()`
      - Implementação de método `ResumeFromSync()`
  - ***Seed load balancers***
    - Decidem onde chares serão mapeados quando criados

5. Sistema de Execução

- **Estruturas de dados p/ balanceamento**
  - Espera novo mapeamento como resposta



Fonte: http://charm.cs.illinois.edu/manuals/html/charm++/20.html

5. Sistema de Execução

- ***Checkpoint***
  - **Imagem do estado atual da aplicação**
    - Estados internos dos chares
    - Comunicação em trânsito
  - Usos
    - **Análise post mortem**
    - **Particionamento da execução**
    - **Tolerância a falhas**


5. Sistema de Execução

- **Execução particionada**
  - **Uso de armazenamento em rede (e.g., nfs)**
  - *Checkpointing*
    - `void CkStartCheckpoint(char* dirname, const CkCallback& cb);`
    - Callback: método a ser chamado após o término do checkpoint ou ao reinício
  - Recomeço da execução
    - `./meu_programa +p20 +restart meu_log`

5. Sistema de Execução

- **Tolerância a falhas online**
  - **Checkpoint em memória ou disco local**
    - `void CkStartMemCheckpoint(CkCallback &cb)`
  - No caso de falha, o protocolo de recuperação reinicia automaticamente

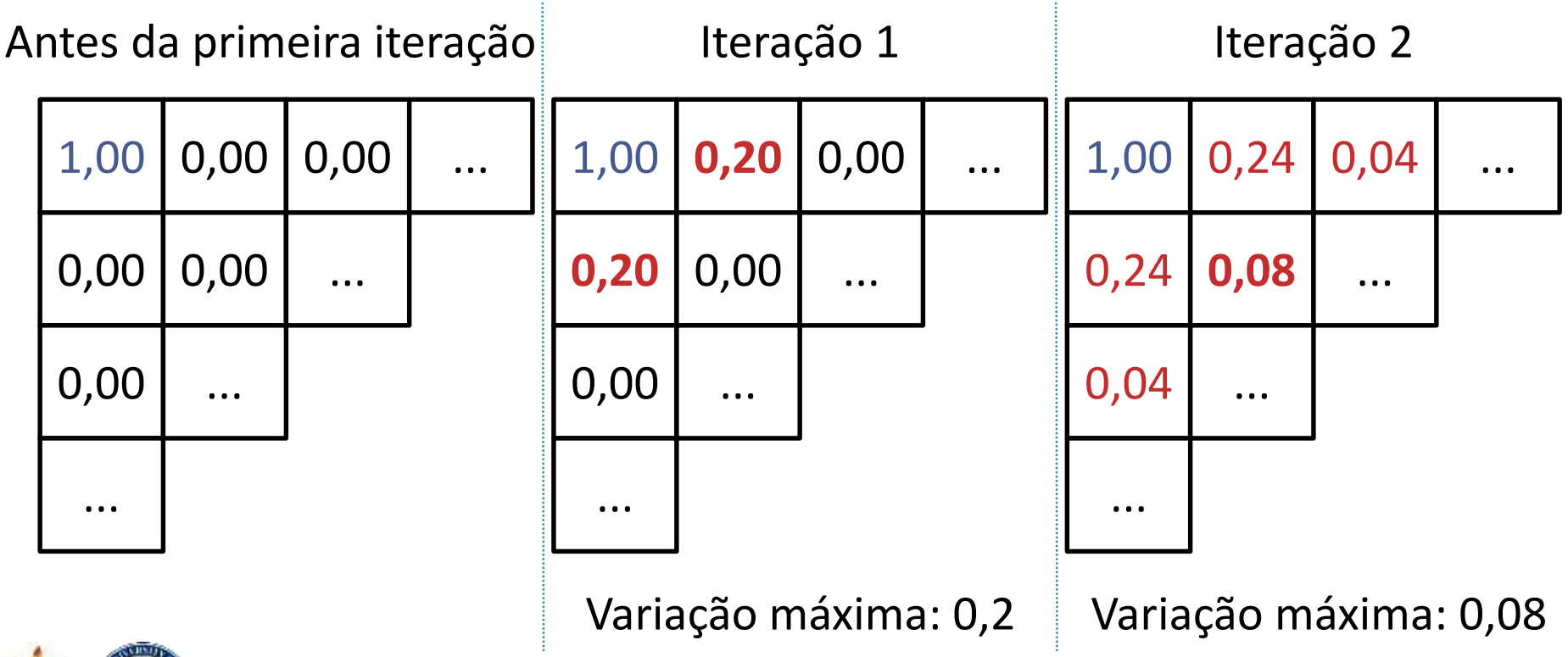
6. EXEMPLO 2: JACOBI2D

6. Exemplo 2: Jacobi2D

- **Jacobi2D**
  - **Método iterativo sobre matriz 2D**
    - **Valor das células é atualizado a cada iteração**
    - Novo valor igual média do valor atual e dos vizinhos
    - Aplicação para quando variação for menor que um limiar

6. Exemplo 2: Jacobi2D

- **Jacobi2D**
  - Primeira célula com valor fixo em 1

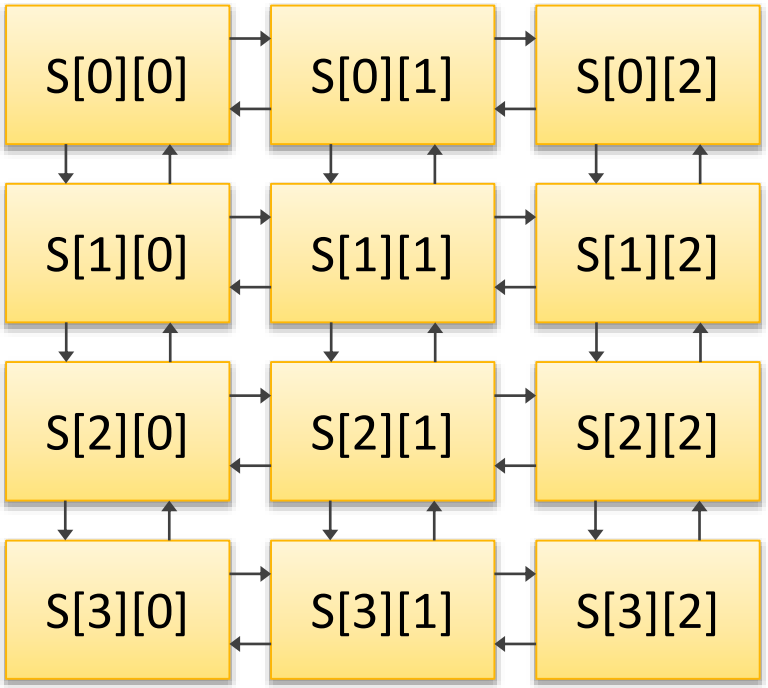


6. Exemplo 2: Jacobi2D

- **Jacobi2D**
  - **Cada chare contém uma parte da matriz**
  - **Comunicação é feita a cada iteração**
    - Transferência das fronteiras atualizadas
  - Controle de granularidade
    - Uma célula por chare = comunicação excessiva
    - Todas células em um chare = sem paralelismo

6. Exemplo 2: Jacobi2D

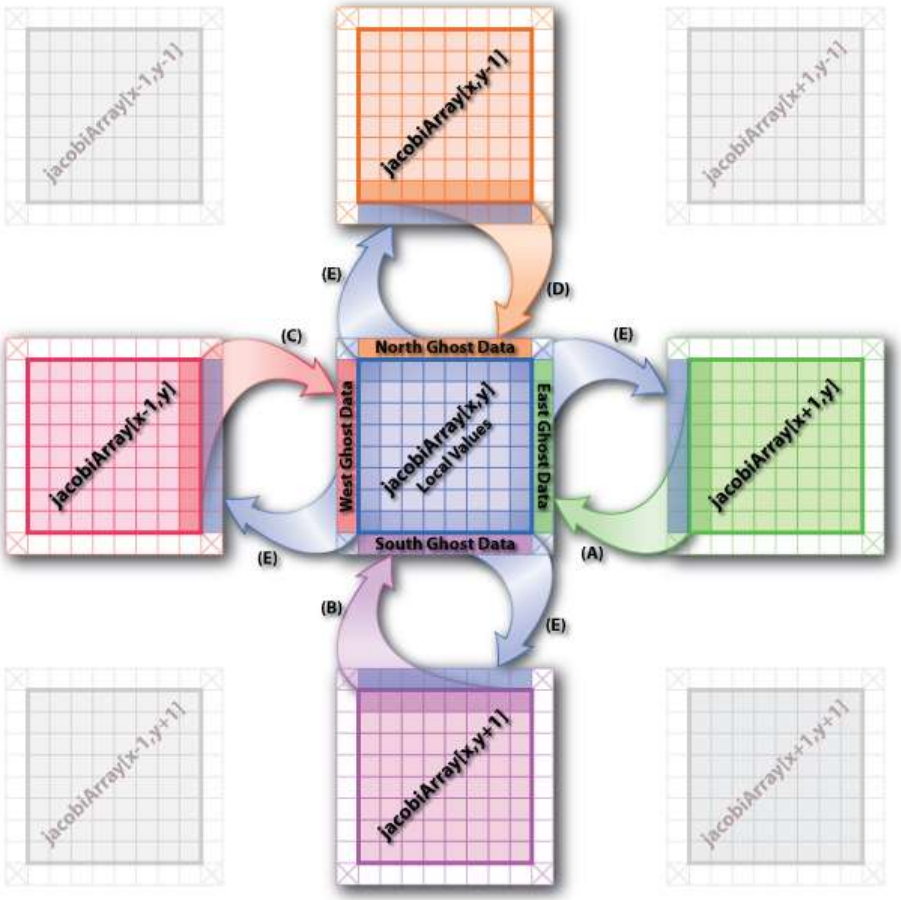
- **Padrão de comunicação**
  - Bordas trocam menos mensagens




```
class Superficie{ ...
  void recebeNorte (...) { ... }
  void recebeSul (...) { ... }
  void recebeLeste (...) { ... }
  void recebeOeste (...) { ... }
}
```

6. Exemplo 2: Jacobi2D

- **Comunicação com vizinhos**



Fonte: http://charm.cs.illinois.edu/tutorial/Basic2DJacobi.htm

6. Exemplo 2: Jacobi2D

Módulo Jacobi

- **Jacobi.ci**
  - Tamanho das mensagens especificado

```
module jacobi {
  array [2D] Jacobi {
    entry Jacobi();
    entry void iniciaEtapa();
    entry void dadosLeste(int tam, double vals[tam]);
    entry void dadosOeste(int tam, double vals[tam]);
    entry void dadosSul(int tam, double vals[tam]);
    entry void dadosNorte(int tam, double vals[tam]);
  };
};
```

6. Exemplo 2: Jacobi2D

Módulo Jacobi

- **Jacobi.h (1/2)**
  - **temp**: envio de fronteiras, valores em iteração
  - **alvoChamadas**: quantas mensagens deve receber antes de começar a calcular

```
class Jacobi : public CBase_Jacobi {
 private:
  double* valores;
  double* temp;
  int alvoChamadas;
  int chamadas;
  void tentaCalcular();
  double calcula();
```

6. Exemplo 2: Jacobi2D

Módulo Jacobi

- **Jacobi.h (2/2)**
  - **~Jacobi()**: destrutor
  - **iniciaEtapa()**: método inicial de cada iteração, envia mensagens aos vizinhos

```
 public:
  Jacobi();
  ~Jacobi();
  void iniciaEtapa();
  void dadosLeste(int tam, double* vals);
  void dadosOeste(int tam, double* vals);
  void dadosSul(int tam, double* vals);
  void dadosNorte(int tam, double* vals);
};
```


6. Exemplo 2: Jacobi2D

Módulo Jacobi

- **Jacobi.C (1/4)**
  - **linhas, colunas**: dimensões das submatrizes em cada chare
  - **linhasChares, colunasChares**: dimensões do vetor de chares

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

extern /* readonly */ CProxy_Main mainProxy;
extern /* readonly */ int linhas;
extern /* readonly */ int colunas;
extern /* readonly */ int linhasChares;
extern /* readonly */ int colunasChares;
```

6. Exemplo 2: Jacobi2D

- **Jacobi.C (2/4)**

```
Jacobi::Jacobi() {
  int numElementos = (linhas + 2) * (colunas + 2);
  valores = new double[numElementos];
  memset(valores, 0, sizeof(double) * numElementos);
  temp = new double[numElementos];
  […]
  alvoChamadas = 1;
  if (thisIndex.x > 0) alvoChamadas++;
  if (thisIndex.x < colunasChares - 1)
    alvoChamadas++;
  if (thisIndex.y > 0) alvoChamadas++;
  if (thisIndex.y < linhasChares - 1)
    alvoChamadas++;
  chamadas = 0;
}
```

6. Exemplo 2: Jacobi2D

Módulo Jacobi

- **Jacobi.C (3/4)**

```
void Jacobi::iniciaEtapa() {
  if (thisIndex.x > 0) {
    for (int i = 0; i < linhas; i++)
      temp[i] = valores[1 + ((colunas + 2) * (i + 1))];
    thisProxy(thisIndex.x - 1,
thisIndex.y).dadosLeste(linhas, temp);
  }
  [...]
  if (thisIndex.y < linhasChares - 1) {
    double* bufferPtr = &(valores[((colunas + 2) * colunas) +
1]);
    thisProxy(thisIndex.x, thisIndex.y +
1).dadosNorte(colunas, bufferPtr);
  }
  tentaCalcular();
}
```

6. Exemplo 2: Jacobi2D

- **Jacobi.C (4/4)**
  - Após cada mensagem, tenta calcular

```
[...] void Jacobi::dadosSul(int tam, double* vals) {
  memcpy(&(valores[((colunas + 2) * (linhas + 1)) + 1]),
vals, sizeof(double) * tam);
  tentaCalcular();
}
void Jacobi::tentaCalcular() {
  chamadas++;
  if (chamadas >= alvoChamadas) {
    chamadas = 0;
    double difMax = calcula();
    mainProxy.recebeDiferenca(difMax);
  }
}
```

6. Exemplo 2: Jacobi2D

- **Main.ci**
  - **recebeDiferenca()**: controle de iterações

```
mainmodule main {
  readonly CProxy_Main mainProxy;
  readonly int linhas;
  readonly int colunas;
  readonly int linhasChares;
  readonly int colunasChares;
  extern module jacobi;

  mainchare Main {
    entry Main(CkArgMsg* msg);
    entry void recebeDiferenca(double);
  };
};
```

6. Exemplo 2: Jacobi2D

Módulo Main

- **Main.h**
  - **limiar**: define quando parar a aplicação
  - **recebimentos**: contador de respostas

```
class Main : public CBase_Main {
  private:
    double limiar;
    int recebimentos;
    double difMax;
    CProxy_Jacobi vetorJacobi;
    void iniciaEtapa();

  public:
    Main(CkArgMsg* msg);
void recebeDiferenca(double valorEnviado);
};
```

6. Exemplo 2: Jacobi2D

Módulo Main

- **Main.C (1/2)**

```
void Main::iniciaEtapa() {
  difMax = 0.0;
  recebimentos = 0;
  vetorJacobi.iniciaEtapa();
}
void Main::recebeDiferenca(double valorEnviado) {
  difMax = ((valorEnviado > difMax) ? (valorEnviado) : (difMax));
  recebimentos++;
  if (recebimentos >= (linhasChares * colunasChares)) {
    CkPrintf("Main::recebeDiferenca() -> difMax = %lf\n", difMax);
    if (difMax > limiar)
      iniciaEtapa();
    else
      CkExit();
  }
}
```

6. Exemplo 2: Jacobi2D

- **Main.C (2/2)**

```
Main::Main(CkArgMsg* msg) {
  linhas = 10; colunas = 10;
  linhasChares = 5; colunasChares = 5;
  limiar = 0.005;

  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);
  mainProxy = thisProxy;
  vetorJacobi = CProxy_Jacobi::ckNew(linhasChares,
colunasChares);
  iniciaEtapa();
}
```

7. CONCLUSÕES

Conclusões

- **Conceitos principais de Charm++**
  - **Execução direcionada por mensagens assíncronas**
  - **Sobredecomposição**
  - **Migrabilidade**
    - **Mecanismos para maior produtividade**

Conclusões

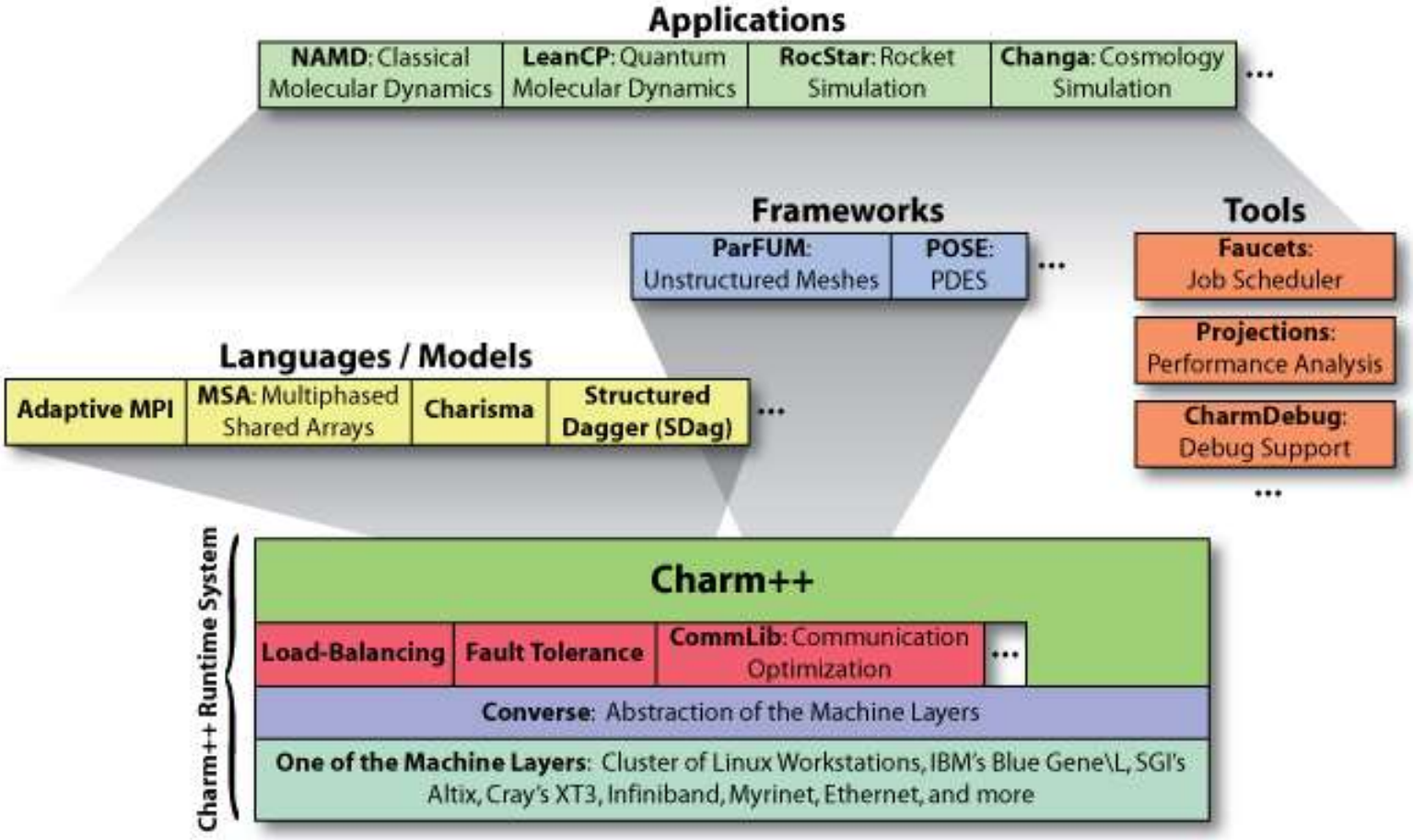
- **Uso de Charm++ para pesquisa**
  - Paralelização de aplicações
  - Balanceamento de carga
  - Tolerância a falhas
  - Algoritmos cientes de energia
  - Entre outros

Conclusões

- **Mecanismos não abordados no curso**
  - **Adaptive MPI**
    - Interface MPI que se beneficia do ambiente de Charm++
  - **BigSim**
    - Simulador e emulador de ambientes paralelos de grande escala

Conclusões

- **Mecanismos não abordados no curso**



Fonte: http://charm.cs.illinois.edu/tutorial/CharmRuntimeSystem.htm

ERAD 2015

Programação Paralela em Charm++

Muito obrigado.

Contato: laercio.pilla@ufsc.br

Dúvidas?