

Balanceamento Dinâmico de Carga em Sistemas Peta/Exaflop

Celso L. Mendes & Eduardo R. Rodrigues

cmendes@illinois.edu

eduardo.rocha@cptec.inpe.br

Department of Computer Science
Univ. Illinois at Urbana-Champaign

Centro de Prev. Tempo e Estudos Climáticos
Inst. Nacional de Pesquisas Espaciais

Lab. Assoc. Comput. Matemática Aplicada
Inst. Nacional de Pesquisas Espaciais



Tópicos do MiniCurso

- Parte I: Virtualização de Programas Paralelos
 - Introdução ao Charm++
 - Características Básicas do Adaptive-MPI
 - Virtualização em Programas MPI
 - Migração entre Processadores
 - Instrumentação e Visualização de Desempenho
- Parte II: Balanceamento Dinâmico de Carga
 - Balanceamento Centralizado × Distribuído
 - Balanceamento por Carga Computacional ou Comunicação
 - Heurísticas para Balanceamento
 - Exemplos em Aplicações Científicas

Material de Apoio ao MiniCurso

- Slides:

- Disponíveis para download em <http://charm.cs.uiuc.edu/~cmendes/ERADSP2011/Slides/>

- Exemplos:

- Disponíveis para download em <http://charm.cs.uiuc.edu/~cmendes/ERADSP2011/Exemplos/>
- Cada exemplo está num sub-diretório - mostrados ao longo do MiniCurso

- Obs:

- O MiniCurso assume acesso a uma instalação atual de Charm++/AMPI

Introdução

- Sistemas **Peta/Exaflop** são/serão sistemas com **muitas unidades de processamento**.
 - Por exemplo, o computador K do Japão tem 548.352 cores (TOP 500 *list*, Junho-2011);
- O grande desafio é fazer as aplicações **escalarem** a esse volume de processamento;
- Os principais razões pelas quais escalabilidade não é atingida são basicamente três (Dongarra *et al*, 2003):
 1. Trechos sequenciais;
 2. Comunicação e sincronização;
 - 3. Desbalanceamento de Carga**

Motivação

- **Balanceamento de carga** tipicamente significa que os processadores têm idealmente **a mesma quantidade de trabalho**, de forma que nenhum processador segure o avanço da computação.
- Para se balancear carga em uma máquina cujos processadores têm o mesmo poder computacional, o programador deve **dividir o trabalho** e a **comunicação** de forma equitativa.
- Isso pode ser um desafio em aplicações cujos tamanhos são desconhecidos antes do tempo de execução.
(Dongarra *et al*, 2003)

Exemplo

"Because atmospheric processes occur nonuniformly within the computational domain, e. g., **active thunderstorms** may occur within only a **few sub-domains of the decomposed domain**, the **load imbalance** across processors can be significant."

(Xue, M.; Droegemeier, K.K.; Weber, D. Numerical Prediction of High-Impact Local Weather: A Driver for Petascale Computing. In: **Petascale Computing: Algorithms and Applications**. 2007.)

Virtualização de Processador

O conceito chave aqui é o de **virtualização de processador**

O programador **divide a computação** em um **grande número de entidades**, que são **mapeadas para os processadores disponíveis** por um **software inteligente**.

Nesse MiniCurso iremos abordar o ambiente **Charm++** e o ***Adaptive MPI***.

Infraestrutura de Software: Charm++

- Origem do Charm++:
 - PPL: Parallel Programming Lab., Univ.Illinois, EUA
 - Liderança: Prof. Laxmikant (Sanjay) Kalé
 - <http://charm.cs.illinois.edu>



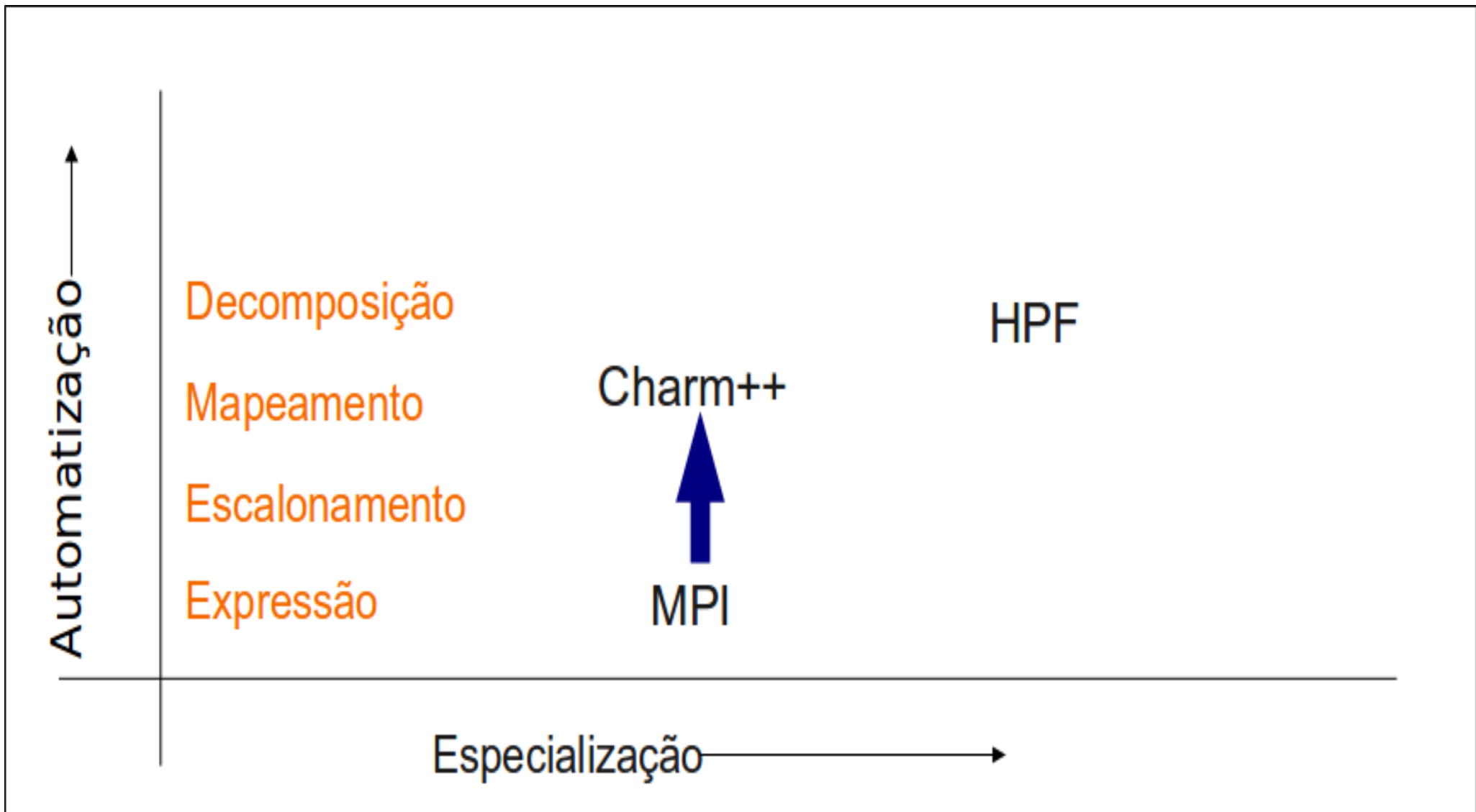
Charm++

O que Charm++ **não** é

- Não é uma ferramenta de paralelização automática
- Não é uma linguagem
 - Pode ser usado com C, C++ e Fortran
- Não é um compilador
- Não é um modelo SPMD
- Não é um modelo centrado em processador
- Não é um modelo de *thread*
- Não é *Bulk Synchronous*

Charm++

Decomposição é feita pelo programador, todo o resto pelo ambiente



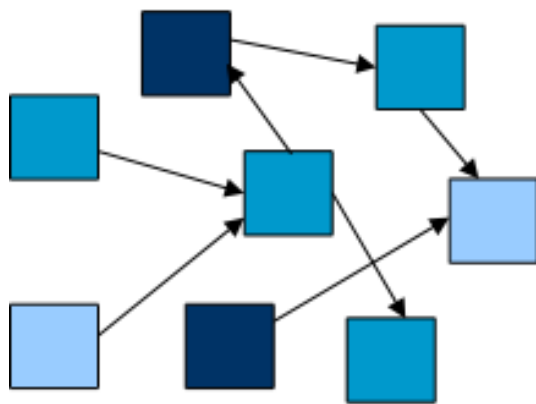
Virtualização:

Decomposição baseada em Objetos (Chares)

- Divide-se a computação em um grande número de objetos (Chares)
- Independente do número de processadores
- Tipicamente maior que o número de processadores
- Os Chares se comunicam via chamadas de métodos (mensagens) assíncronas
- Deixa o sistema (biblioteca) mapear os objetos nos processadores
- Similar a mensagens ativas e atores

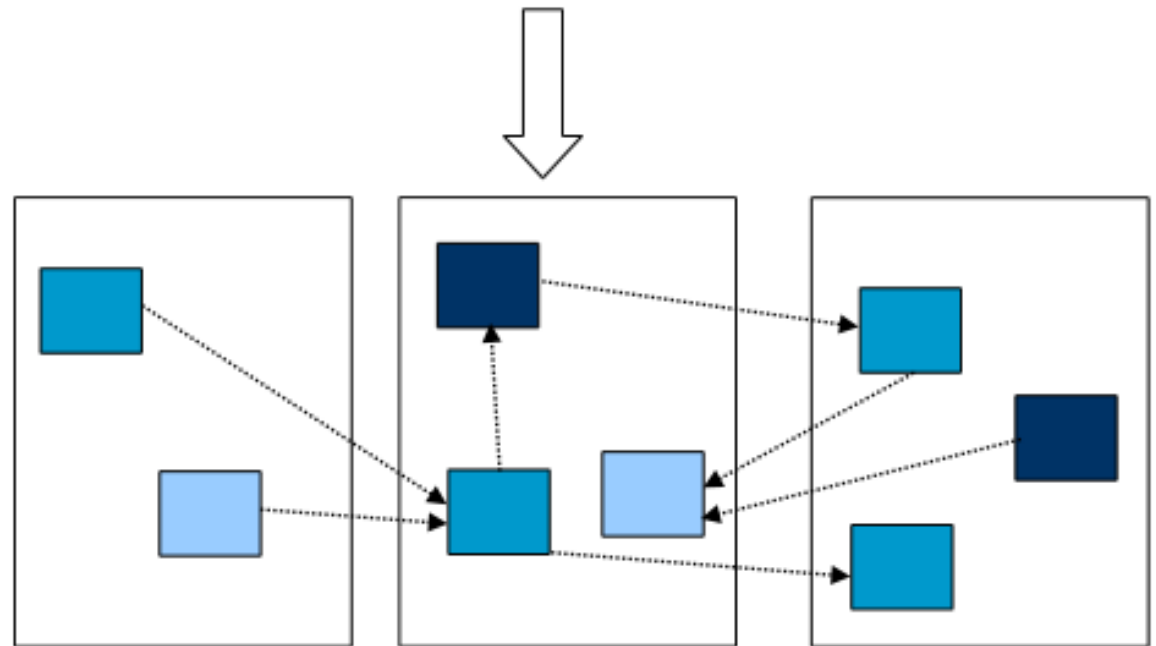
Paralelização baseada em Objetos

O usuário está preocupado apenas com a interação entre objetos



*Visão do
Usuário*

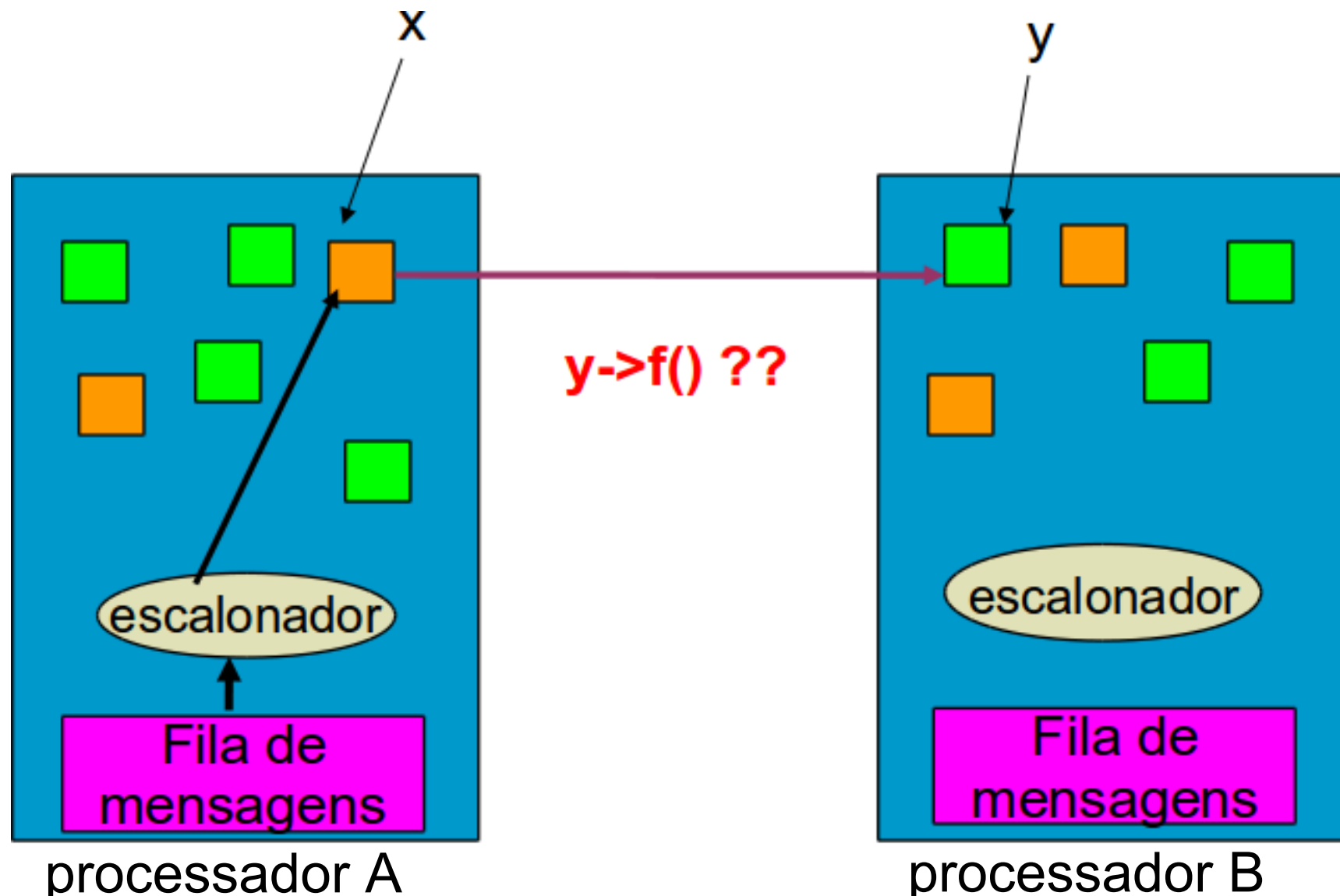
Implementação do sistema



Execução Orientada a Mensagens

- Objetos comunicam-se **assincronamente** através de **chamadas remotas de métodos**
- A ordem de execução é não-determinística
- Benefícios:
 - **Comunicação tolerante a latência**
 - **Maior sobreposição com computação**

Execução Orientada a Mensagens (o método *f* deve ser um método de entrada)



Construindo o Charm++ na sua Máquina

- Baixar o Charm:

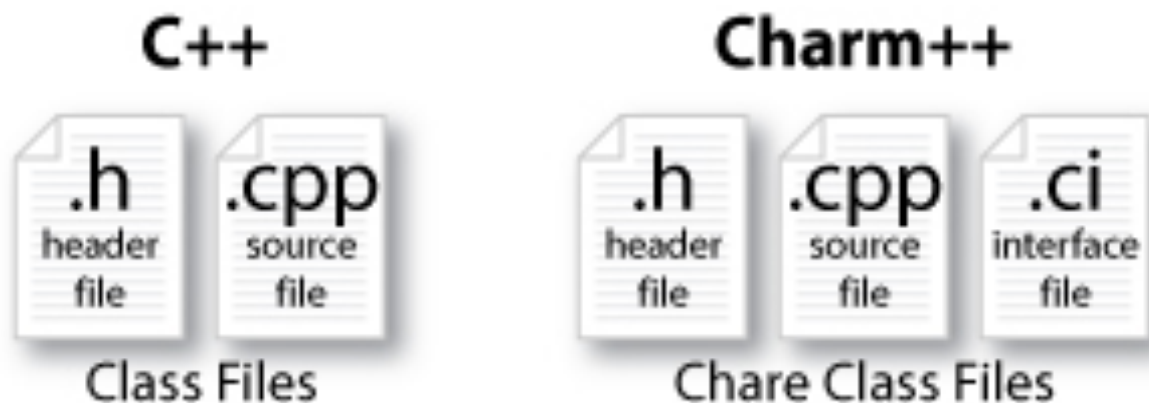
<http://charm.cs.uiuc.edu/software>

- Construir:

- `./build <target> <version> <options> [charmc-options]`
- Ver README para detalhes
- `./build charm++ net-linux-x86_64`
- `./build charm++ mpi-crayxt gfortran`

Estrutura de Arquivo do Charm++

- Objetos C++ (sejam eles Chares ou não) residem em **arquivos .cpp e .h convencionais**
- Chares e métodos de entrada são **declarados em um arquivo de interface .ci** e são implementados em um arquivo .cpp.



Exemplo de Programação em Charm++

Olá Mundo: arquivo .ci

- .ci: Charm Interface
- Define que tipo de Chares estão presentes na aplicação
- Pelo menos um **mainchare** dever ser definido
- Cada definição é feita dentro de um **módulo**

```
mainmodule hello {  
  
    mainchare Main {  
        entry Main(CkArgMsg* msg);  
    };  
  
};
```

Exemplo de Programação em Charm++

Olá Mundo: código

main.h

```
#include "hello.decl.h"

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

main.C

```
#include "main.h"

// Entry point of Charm++ application
Main::Main(CkArgMsg* msg) {

    CkPrintf("Hello World!\n");

    CkExit();
}

Main::Main(CkMigrateMessage* msg) { }

#include "hello.def.h"
```

Compilação e Execução

Compilar

- `charmc <opções> <código-fonte>`
 - `-o`, `-g`, `-module`, `-tracemode`

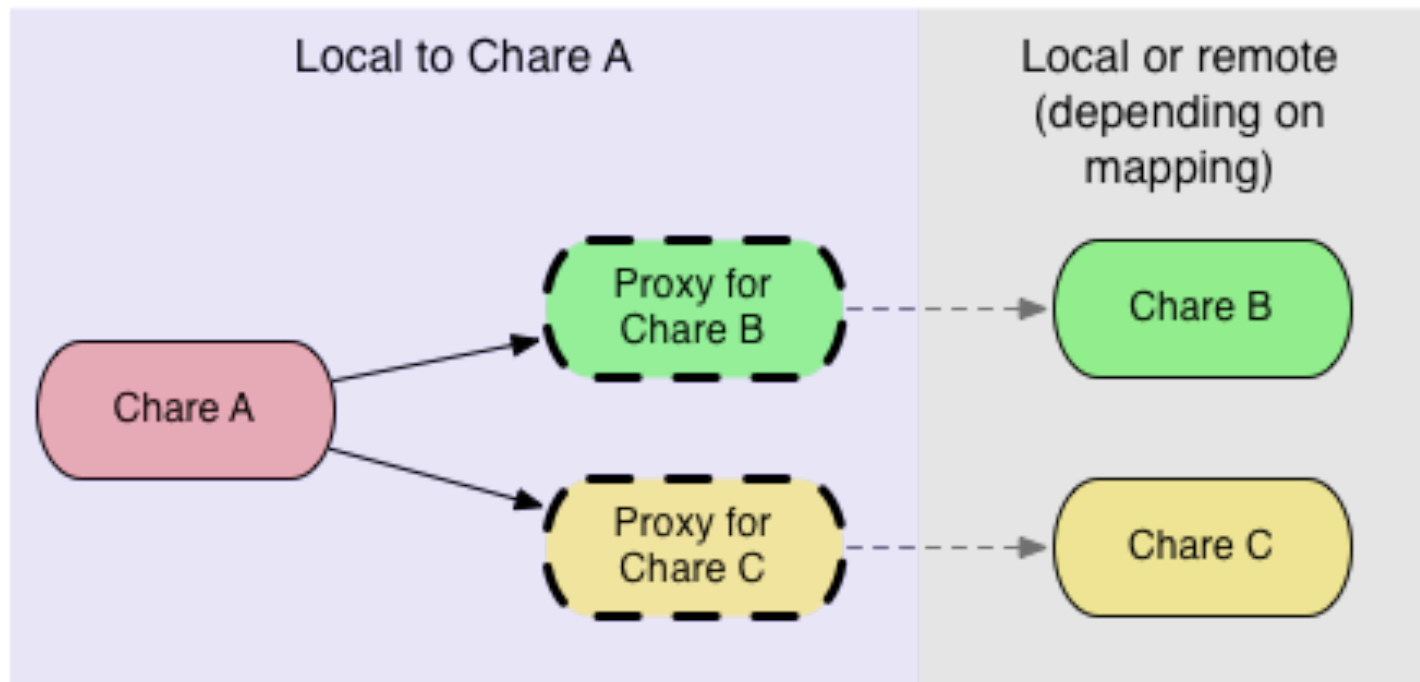
```
pgm: pgm.ci pgm.h pgm.C
      charmc pgm.ci
      charmc pgm.C
      charmc -o pgm pgm.o
```

Para rodar um programa CHARM++ em 4 processadores:

```
charmrun pgm +p4 <params>
```

Como Chares se Comunicam?

- Chares ficam espalhados em múltiplos processadores
- Não é possível invocar diretamente métodos desse Chares
- Para comunicação, usa-se **proxies**



Chamada Remota a Métodos: classes proxy

- **Classes proxy** são geradas para cada **classe** **chare**
 - Por exemplo, **CProxy_Y** é o proxy gerado para a classe **Y**
 - Objetos proxy **sabem onde o objeto real está**
 - **Chamadas aos métodos do proxy** apenas implicam no **empacotamento e envio de dados**
- Dado um proxy p, é possível chamar um método como:
 - `p.method(msg);`

Um exemplo um pouco mais complexo

- O Mainchare envia uma mensagem para o objeto Hello;
- O objeto Hello imprime "Hello World!";
- O objeto Hello envia uma mensagem de volta ao mainchare;
- Mainchare termina o programa.

Código

hello.ci

```
mainmodule hello {
  readonly CProxy_Main mainProxy;

  mainchare Main {
    entry Main(CkArgMsg*);
    entry void end(void);
  };

  chare Hello {
    entry Hello();
    entry void PrintHello(void);
  };
}
```

hello.cpp

```
#include "hello.decl.h"

/*readonly*/ CProxy_Main mainProxy;

class Main : public CBase_Main {
public:
  Main(CkArgMsg* m) {
    delete m;
    mainProxy = thishandle;

    CProxy_Hello h =
      CProxy_Hello::ckNew();
    h.PrintHello();
  }

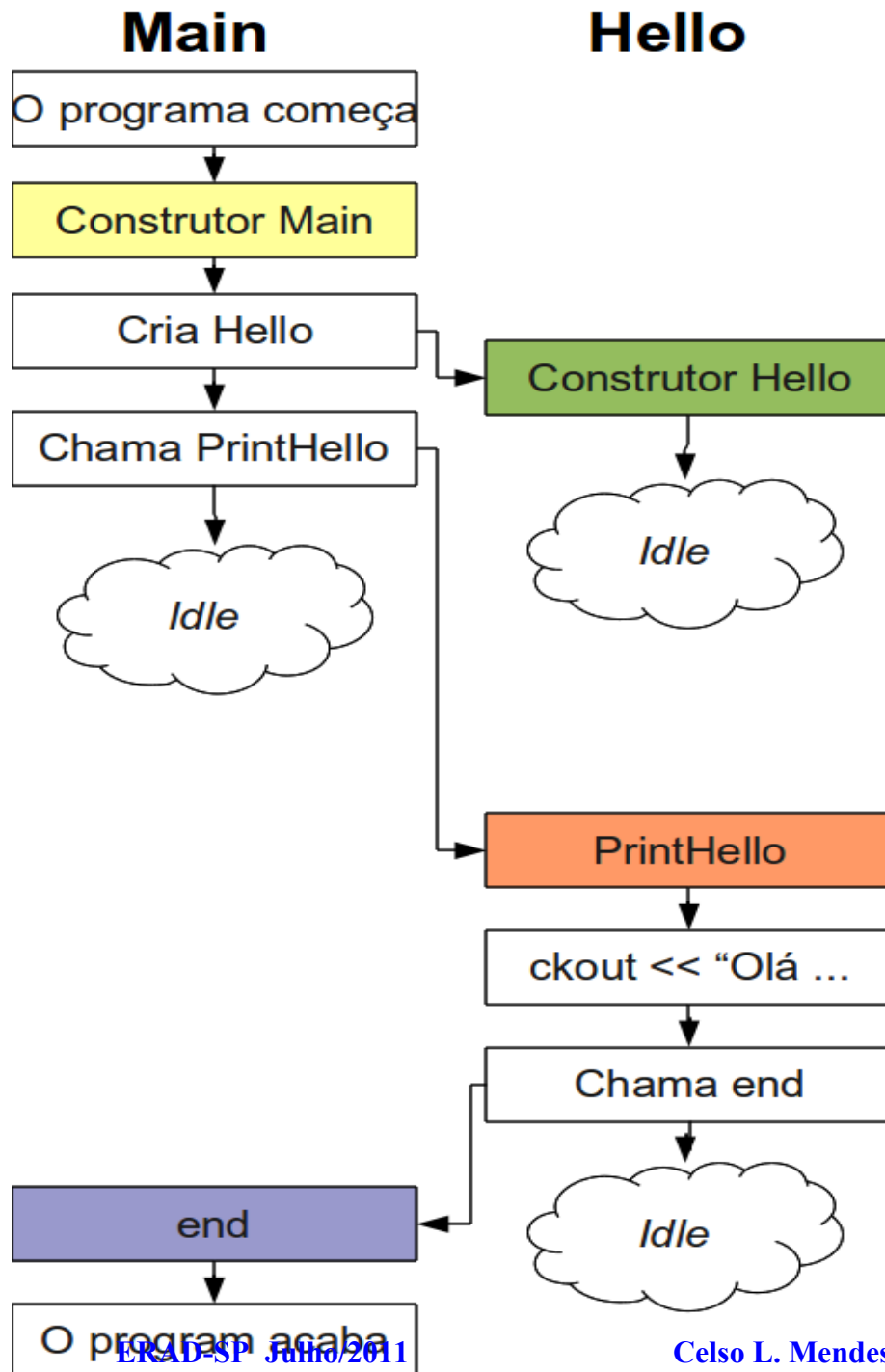
  void end() {
    CkExit();
  }
};

class Hello : public CBase_Hello {
public:
  Hello() {}

  void PrintHello(void) {
    ckout << "Olá Mundo!" <<
      endl;
    mainProxy.end();
  }
};

#include "hello.def.h"
```

Fluxo de Execução



hello.cpp

```
#include "hello.decl.h"

/*readonly*/ CProxy_Main mainProxy;

class Main : public CBase_Main {
public:
    Main(CkArgMsg* m) {
        delete m;
        mainProxy = thishandle;

        CProxy_Hello h =
            CProxy_Hello::ckNew();
        h.PrintHello();
    }

    void end() {
        CkExit();
    }
};

class Hello : public CBase_Hello {
public:
    Hello() {}

    void PrintHello(void) {
        ckout << "Olá Mundo!" <<
            endl;
        mainProxy.end();
    }
};

#include "hello.def.h"
```

Mais Sobre o Charm++

- **URL** principal:
 - <http://charm.cs.uiuc.edu>
- Documentação, **manuais**, etc.:
 - <http://charm.cs.uiuc.edu/help>
- Publicações
 - <http://charm.cs.uiuc.edu/papers>
- Tutorial **online** sobre sobre programação com Charm++:
 - <http://charm.cs.uiuc.edu/tutorial/>
- Testes diários em **diversas máquinas/arquiteturas**:
 - <http://charm.cs.illinois.edu/autobuild/cur/>
 - mostra exemplos de como construir, resultados, etc
- Para obter ajuda:
 - <mailto:charm@cs.uiuc.edu>

Motivação para o AMPI

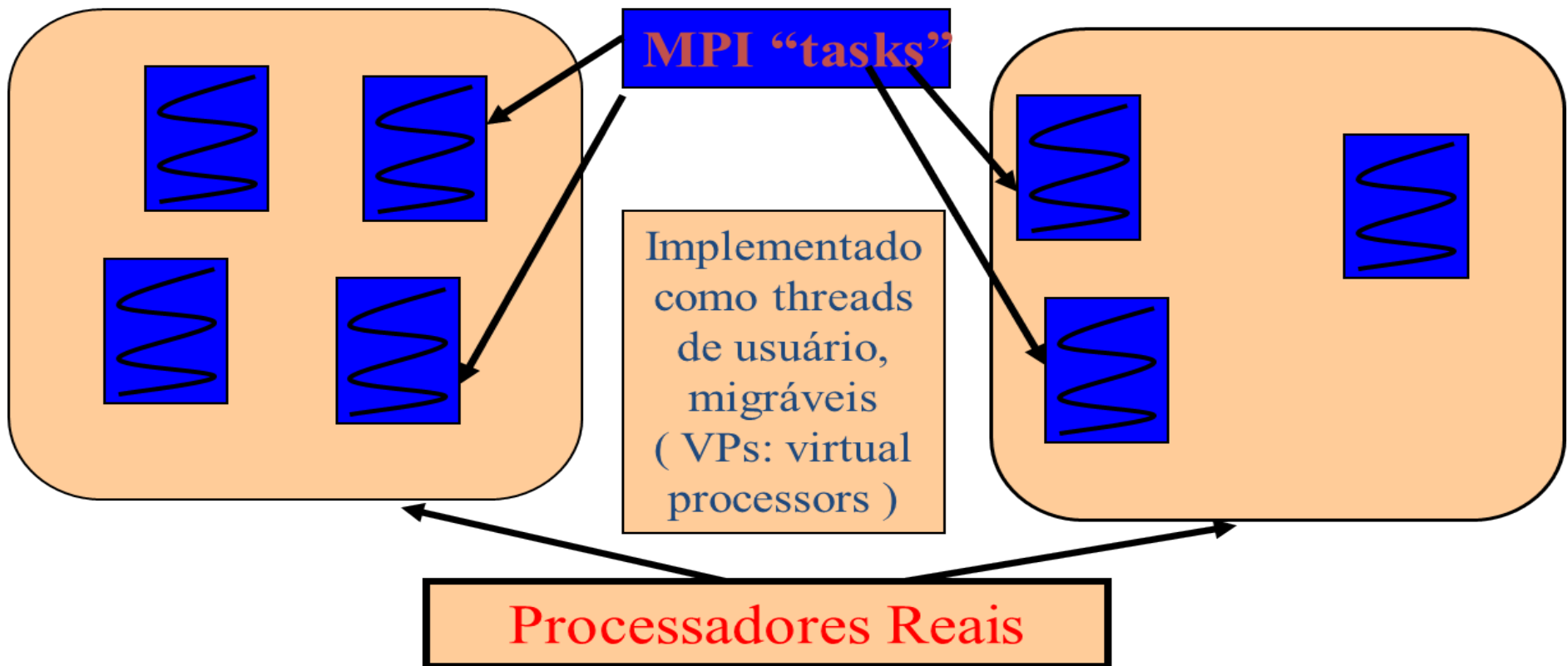
- MPI é um padrão *de fato* para programação paralela
- Porém, aplicações modernas podem ter:
 - distribuição de carga pelos processadores **variável** ao longo da simulação;
 - refinamentos **adaptativos** de grades;
 - múltiplos módulos relativos a **diferentes componentes físicos** combinados na mesma simulação;
 - exigências do algoritmo quanto ao **número de processadores** a serem utilizados.
- Várias destas características não combinam bem com implementações convencionais de MPI

Alternativa: Adaptive MPI

- **Adaptive MPI** (AMPI) é uma implementação do padrão MPI **baseada em Charm++**
- Com AMPI, é possível utilizar aplicações MPI já existentes, através de poucas modificações no código original
- Tais aplicações MPI podem então usufruir dos **benefícios** da virtualização de processadores, conforme será visto neste MiniCurso
- AMPI está disponível e é portátil para diversas arquiteturas (na verdade, para todas as arquiteturas suportadas em Charm++)

Adaptive MPI: Princípios Gerais

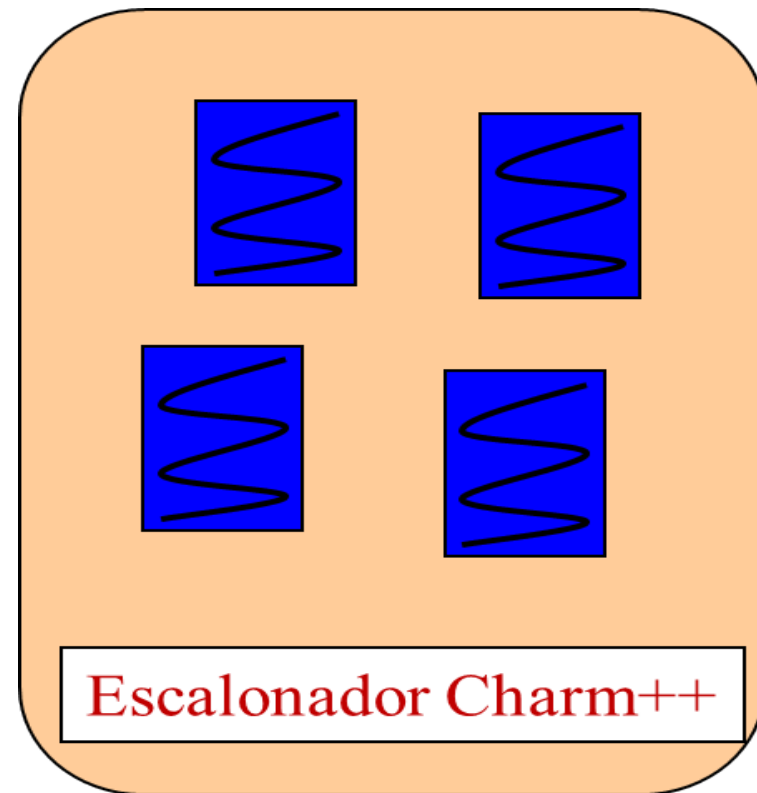
- Em AMPI, cada tarefa MPI é embutida em um **objeto** (elemento de vetor, ou *thread* de usuário) Charm++
- Como todo objeto Charm++, as tarefas AMPI (*threads*) são **migráveis** entre processadores



Adaptive MPI: Características

- Em AMPI, cada **tarefa MPI** é implementada como uma **thread de usuário** (*threads* cooperativas)
- Diferentemente de *threads* de *kernel*, *threads* de usuário são leves e resultam em **trocas de contexto mais rápidas**
- Apenas **uma thread** de usuário **executa em um dado momento**, sob comando do Escalonador Charm++
- Tipicamente, todas as *threads* em um processador residem em **um único processo**
- Número total de processadores: P
- Número total de *threads*: VP ($> P$)
- Razão de virtualização: VP / P

Na prática, a virtualização equivale a uma *sobre-decomposição* dos objetos pelos processadores

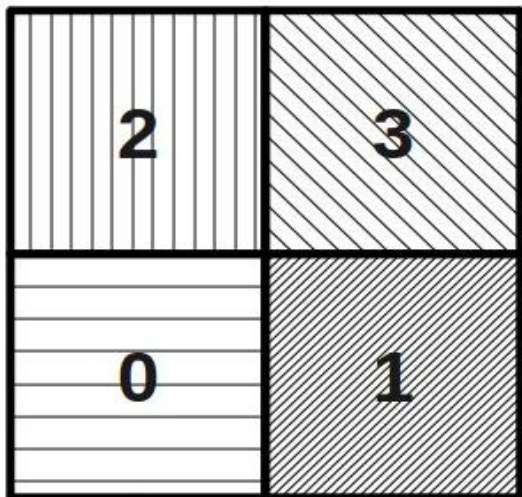


Adaptive MPI e Virtualização

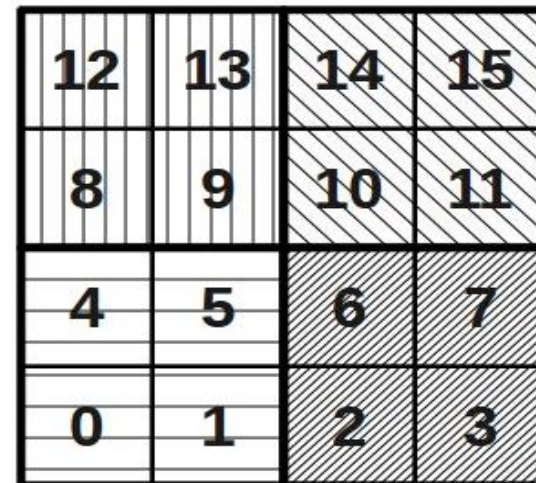
- Benefícios da virtualização:
 - **Sobreposição** automática entre **computação e comunicação**
 - Melhor **uso de cache**
 - **Flexibilidade** para se fazer **balanceamento de carga**

(Huang, C.; Zheng, G.; Kumar, S.; Kale, L. *Performance Evaluation of Adaptive MPI*. **PPoPP**, 2006.)

(Rodrigues, E.R.; Navaux, P.O.A.; Panetta, J.; Mendes, C.L.; Kale, L.V. *Optimizing an MPI Weather Forecasting Model via Processor Virtualization*. **HiPC**, 2010.)



P=4 , 4 tarefas MPI



P=4 , 16 threads AMPI

Ganhos com a Virtualização

- Exemplo: BRAMS, previsão de tempo
 - Máquina: Cray-XT5, 64 processadores

| Número de Threads AMPI | Razão de Virtualização | Número de cache-misses em L3 | Tempo de Execução (segundos) |
|------------------------|------------------------|------------------------------|------------------------------|
| 64 | 1 | $8,5 \times 10^6$ | 4.971 |
| 256 | 4 | $4,4 \times 10^6$ | 3.858 |
| 1024 | 16 | $3,9 \times 10^6$ | 3.713 |

Status Atual do AMPI

- Compatibilidade com o padrão **MPI-1.1**
 - Excessão: tratamento de erros, interface de *profiling*
- Suporte parcial a **MPI-2**:
 - Comunicação unilateral (*one-sided*) disponível
 - ROMIO integrado, para I/O paralelo
 - Principais ausências: gerenciamento dinâmico de processos, interfaces para algumas linguagens
- Boa documentação e testes
 - Testado periodicamente com o benchmark IMB

Construindo o AMPI na sua Máquina

- Todo o código do AMPI está contido na distribuição pública do Charm++
- Construir o AMPI:
 - `./build <target> <version> <options> [charmc-options]`
 - Ver README para detalhes
 - `./build AMPI net-linux-x86_64`
 - `./build AMPI mpi-crayxt gfortran`

Escrever e Executar um Programa AMPI

- Escrever um programa MPI convencional:

...

```
int main(int argc, char** argv) {  
    int pid, rank;  
    char hostnm[HNSIZE];
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    gethostname(hostnm, HNSIZE);
```

```
    pid = getpid();
```

```
    printf("Olá mundo - rank %d - "\n  
           "hostname %s - pid %d\n", rank,\n           hostnm, pid);
```

```
    MPI_Finalize();
```

```
}
```

- Compilar:

```
ampicc prog.c -o prog
```

- Executar:

```
./charmrun +p 2 ./prog +vp 4
```

Escrever e Executar um Programa AMPI em Fortran

- Escrever um programa MPI convencional:
 - Substituir a palavra reservada **program <nome>** por **subroutine mpi_main**
- Compilar:
ampif90 prog.f90 -o prog
- Executar:
./charmrun +p 2 ./prog +vp 4

Mapeamento Inicial dos VPs

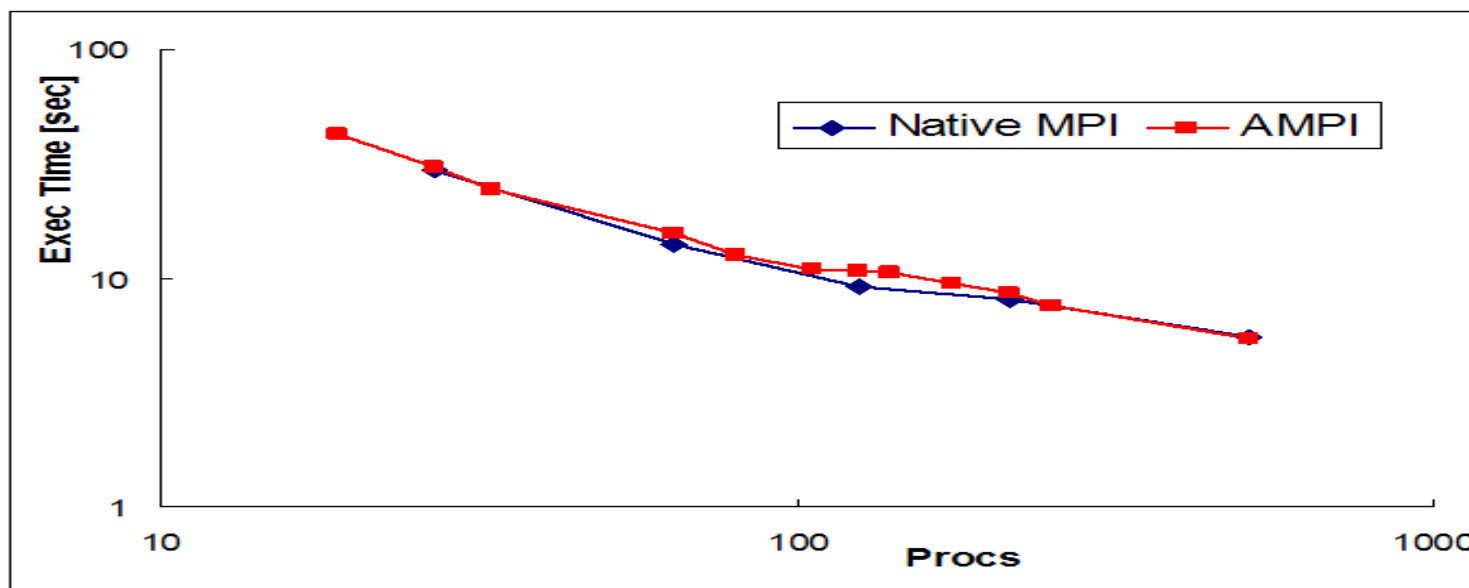
- Vários mapeamentos iniciais dos VPs aos processadores estão disponíveis
 - Seleção: *charmrun +p 2 prog +vp 8 +mapping <map>*
 - RR_MAP: round-robin (cíclico)
 - BLOCK_MAP: bloqueado (default)
 - Outros mapeamentos podem ser implantados
- Exemplo: P=2, VP=8, <map>=RR_MAP
threads (VPs) 0,2,4,6 estão no Proc.0
threads (VPs) 1,3,5,7 estão no Proc.1
- Notar que este é apenas o mapeamento inicial; com a capacidade de migração, cada VP poderá mover-se ao longo da execução para outro processador

Restrições de Memória por Thread

- Em princípio, cada *thread* AMPI pode ter no máximo 1 MBytes de variáveis na **pilha** (variáveis locais)
- Alternativa: opção `+tcharm_stacksize`
- Exemplo: aumentando a pilha por thread para ~20MB
`charmrun +p 2 prog +vp 8 +tcharm_stacksize 20000000`
- Notar que é possível aumentar ou diminuir o valor de 1MB, dependendo de cada caso particular -- por exemplo, com um número muito alto de *threads*, pode ser preciso diminuir o tamanho de cada pilha. Ver o **exemplo** em *Exemplos/Bigstack/*

Desacoplamento entre Números de Processadores Reais (P) e Virtuais (VP)

- Contanto que $VP \geq P$, é possível rodar um programa AMPI com qualquer número de processadores reais
- Exemplo: suponha um stencil 3D, que deve ser rodado em K^3 processadores com MPI nativo; com AMPI, qualquer número de processadores é útil



Variáveis Globais e Estáticas

- Ao utilizar AMPI, **variáveis globais e estáticas devem ser eliminadas da aplicação**, ou **privatizadas** em cada *thread*
- Esses tipos de variáveis são perigosas para *threads*
 - Razão: **variáveis globais e estáticas** são **compartilhadas** entre *threads* de um processo
 - Exemplo: (supondo que *count* seja global ou estática)

Valor Errado

| Thread 1 | Thread 2 |
|---|------------------------------|
| count=1 block in MPI_Recv b=count | count=2 block in MPI_Recv |

tempo

Privatização de Variáveis Globais e Estáticas

- Existem várias formas de se privatizar variáveis globais e estáticas:
 - Remover **manualmente** esses tipos de variáveis e **alocá-las no *heap***. Exemplo:

Código original

```
PROGRAM MAIN
  USE shareddata
  include 'mpif.h'
  INTEGER :: i, ierr
  CALL MPI_Init(ierr)
  CALL MPI_Comm_rank(
    MPI_COMM_WORLD,
    myrank, ierr)
  DO i = 1, 100
    xyz(i) = i + myrank
  END DO
  CALL subA
  CALL MPI_Finalize(ierr)
END PROGRAM
```

Código AMPI

```
SUBROUTINE MPI_Main
  USE shareddata
  USE AMPI
  INTEGER :: i, ierr
  TYPE(chunk), pointer :: c
  CALL MPI_Init(ierr)
  ALLOCATE(c)
  CALL MPI_Comm_rank(
    MPI_COMM_WORLD,
    c%myrank, ierr)
  DO i = 1, 100
    c%xyz(i) = i + c%myrank
  END DO
  CALL subA(c)
  CALL MPI_Finalize(ierr)
```

END SUBROUTINE

Privatização de Variáveis Globais e Estáticas

- Mais formas de se privatizar variáveis globais e estáticas:

- Remover semi-automaticamente esses tipos de variáveis
 - Ferramenta AMPlizer, do Photran (em testes)

- Uso da opção **-swapglobals** na compilação:

`ampicc prog.c -o prog -swapglobals`

- Uso de **TLS** (disponível na versão CVS do Charm):

`ampicc prog.c -o prog -tlsglobals`

(Rodrigues, E. R.; Navaux, P. O. A.; Panetta, J.; Mendes, C. L. *A New Technique for Data Privatization in User-level Threads and its Use in Parallel Applications*. In: **25th ACM Symposium On Applied Computing - SAC**, Sierre, Switzerland, 2010.)

Privatização de Variáveis Globais e Estáticas

- Vantagens e desvantagens dos vários métodos:

| Método | Vantagens | Desvantagens |
|---------------------------------|---|---|
| Reestruturação do código | a) independe de compilador b) aumenta localidade entre variáveis do programa | a) trabalhosa de implementar manualmente b) há ferramenta automática apenas para uma linguagem (Fortran) |
| -swapglobals | a) simples de utilizar b) totalmente automática | a) trata apenas variáveis globais b) só funciona em sistemas ELF c) custo de execução é proporcional ao número de variáveis globais |
| -tlsglobals | a) trata globais e estáticas b) custo de execução independe do número de variáveis | a) requer anotação extra em C (__thread) b) atualmente, para Fortran, depende de mudanças no compilador |

Balanceamento de Carga em AMPI

- Desbalanceamento de carga afeta o desempenho de aplicações paralelas dinâmicas
- Balanceamento automático: **MPI_Migrate()**
 - **MPI_Migrate** é uma **chamada coletiva** que informa ao ambiente de execução que a **thread** está pronta para **migrar**, caso seja necessário para balancear carga.
 - Ao se usar um balanceador de carga, este deverá:
 - Decidir que *thread* vai para onde;
 - Enviar a decisão para o processador que tem a *thread*;
 - Empacotar dados de *heap* e pilha da thread;
 - Enviar *thread* e dados ao processador-destino;
 - Desempacotar dados de *heap* e pilha;
 - Reiniciar a execução.
 - Tudo isto é feito automaticamente pelos balanceadores do Charm++/AMPI

Balanceamento Automático

Compilar e executar

- Para usar um balanceador de carga você deve ligar sua aplicação com os **módulos de balanceamento**:

- `ampicc prog.c -o prog -module EveryLB -memory isomalloc`

-memory isomalloc: necessário p/ migração do *heap*

- Executar com a opção **+balancer**:

- `./charmrun +p4 ./prog +vp8 +balancer GreedyLB`

Você pode precisar de um `"-fno-stack-protector"` na compilação

Exemplo

```
...  
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
gethostname(hostnm, HNSIZE);  
pid = getpid();  
printf("(1) Olá mundo - rank %d - \"\  
    \"hostname %s - pid %d\\n\", rank,\  
    hostnm, pid);
```

MPI_Migrate();

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
gethostname(hostnm, HNSIZE);  
pid = getpid();  
printf("(2) Olá mundo - rank %d - \"\  
    \"hostname %s - pid %d\\n\", rank,\  
    hostnm, pid);
```

...

Balanceadores Disponíveis no Charm++

- **MetisLB**: Usa o particionador Metis(tm);
- **GreedyLB**: Um algoritmo guloso;
- **GreedyCommLB**: Um algoritmo guloso que considera comunicação além de carga pura;
- **TopoCentLB**: Um algoritmo guloso que leva em consideração a topologia da máquina;
- **RefineLB**: Move *threads* de processadores carregados, mas limita o número de migrações;
- **RefineCommLB**: Mesmo que o RefineLB, mas considera comunicação;
- **RefineTopoLB**: Mesmo que o RefineLB, mas considera a topologia;
- **ComboCentLB**: Balanceador especial que permite combinar dois ou mais balanceadores.

Balanceadores Disponíveis no Charm++

- MetisLB
 - GreedyLB
 - GreedyC
 - TopoCe
 - RefineLB
 - RefineC
 - RefineT
 - ComboCentLB: Balanceador especial que permite combinar dois ou mais balanceadores.
- Na segunda parte desse MiniCurso, veremos como escrever balanceadores de carga.

Princípios Gerais dos Balanceadores em AMPI

- Os balanceadores são **dinâmicos**, isto é, podem ser invocados a qualquer instante da execução, através da chamada a *MPI_Migrate()*
- Os balanceadores consideram as **cargas de computação** e a **comunicação** medidas em cada *thread*
- É possível desligar tais medições, se desejável:
 - Ao início da execução:
 - opções **+LBOff**, **+LBCommOff** na linha de comando
 - Durante a execução, pelo próprio programa:
 - Rotinas **LBTurnInstrumentOn()**, **LBTurnInstrumentOff()**
 - Por default, a instrumentação está sempre ativa

Exemplo de Uso dos Balanceadores em AMPI

- Possível cenário de uso de um balanceador em um código **iterativo**, com AMPI:
 - Iniciar a execução **sem instrumentação** de carga
 - execução ativada com a opção *+LBOff*
 - Após K iterações, **ligar** a instrumentação
 - chamada a *LBTurnInstrumentOn()*
 - Após $K+M$ iterações, **invocar o balanceamento**
 - chamada a *MPI_Migrate()*
 - após retorno, chamar *LBTurnInstrumentOff()*
 - reiniciar ciclo para as próximas $K+M$ iterações
 - Neste caso, o balanceador irá considerar apenas a carga e a comunicação medidas nas **últimas M iterações** executadas, ou seja, apenas o passado recente!

Visualização de Desempenho

Projections

- *Projections* é a ferramenta de análise/visualização de desempenho do Charm++;
- Ela pode ser usada para identificar desbalanceamento de carga na aplicação;
- Análise de desempenho com *Projections* normalmente envolve 3 passos:
 - Ligar a aplicação ao gerador de traços;
 - Executar a aplicação para geração dos traços;
 - Usar a ferramenta gráfica para visualizar os resultados, após a execução.

Visualização de Desempenho

Projections

- *Projections* é uma ferramenta tipo **GUI**, baseada em Java
- Pode ser usada em qualquer máquina com **Java**
- Não é distribuída junto com o Charm++ padrão
 - Deve ser **baixada e construída separadamente**

Ligando a Aplicação ao Gerador de Traços e Executando

- Compilar:
 - `ampif90 prog.f90 -o prog -tracemode projections`
- Executar a aplicação normalmente:
 - `./charmrun +p2 ./prog`
 - Essa execução gera arquivos contendo os **traços da execução**

Projections - Exemplo 1

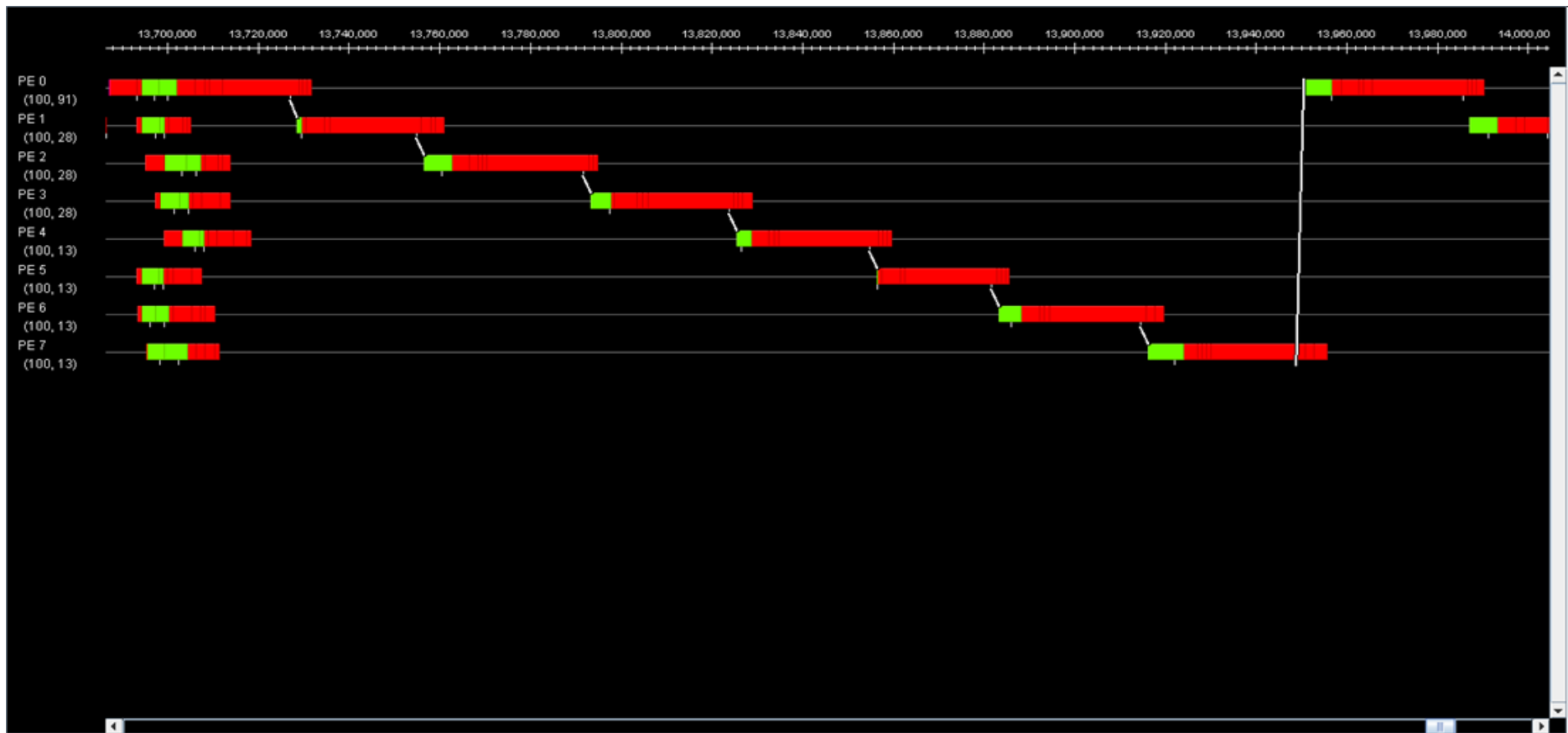
Troca de mensagens num anel

- Idéia: Circular mensagem entre os VPs, dispostos na forma de um anel (*ring*)
- Código:
 - Diretório *Exemplos/Ring/*, arquivos *fring.f90*, *fring-ampi.f90*
- Compilar:
 - `ampif90 fring-ampi.f90 -o fring-ampi -tracemode projections`
- Executar a aplicação normalmente:
 - `./charmrun +p2 ./fring-ampi +vp 4`
 - Essa execução gera arquivos contendo os **traços da execução**: arquivos `*.sts` e `*.log`
 - Tais arquivos podem ser visualizados com *Projections*, após a execução

Projections - Exemplo 1

Troca de mensagens num anel

- Visualização de parte da execução de *fring-ampi*:



Projections - Exemplo 2

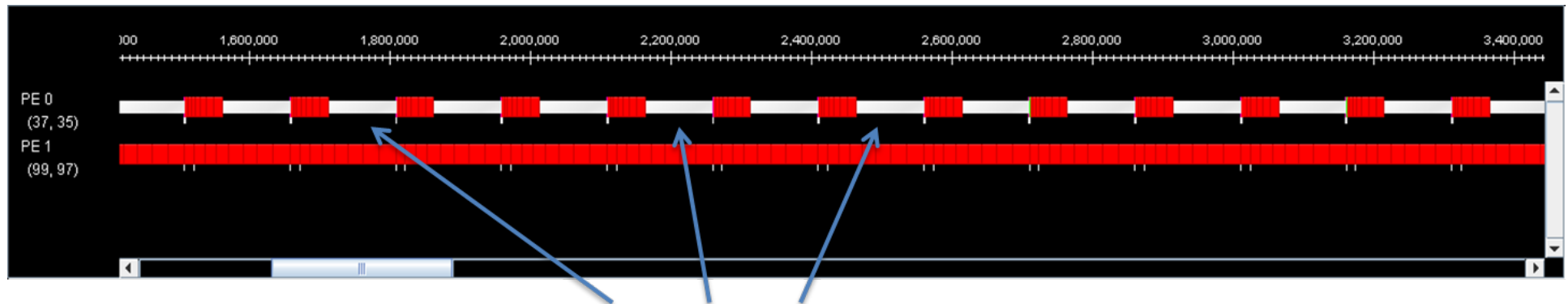
Desbalanceamento de Carga Simples

- Suponha um código MPI iterativo, no qual a cada iteração **cada elemento realiza uma quantidade de trabalho proporcional ao seu rank**. Logo, há um claro desbalanceamento de carga. Usando virtualização e balanceamento, isto pode ser corrigido.
- Código:
 - Diretório *Exemplos/LB-Basico/*, arquivo `work.c`
- Compilar:
 - ajustar o Makefile, apontando-o para o local do Charm++
 - dois executáveis AMPI são gerados: normal e com balanceamento (arquivos `work.AMPI` e `work.AMPI-LB`)

Projections - Exemplo 2

Desbalanceamento de Carga Simples

- Primeira execução: sem balanceamento
- Executar:
 - `./charmrun +p2 ./work.AMPI +vp 16`
 - Notar surgimento dos arquivos de traço de execução
 - Tempo reportado (Cray-XT6): 17.7 s



Branco= tempo ocioso, Vermelho=computando

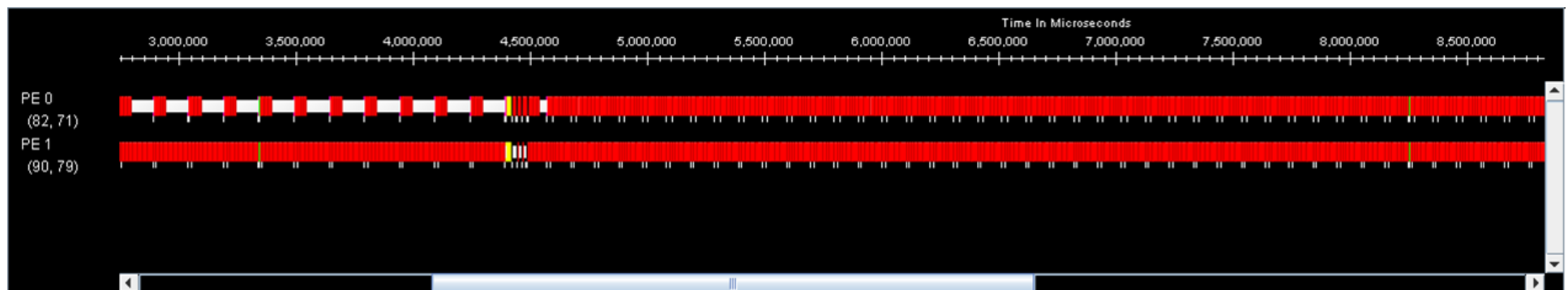
Carga do Proc.0: Ranks $0+1+2+\dots+7=28$

Carga do Proc.1: Ranks $8+9+10+\dots+15=92$

Projections - Exemplo 2

Desbalanceamento de Carga Simples

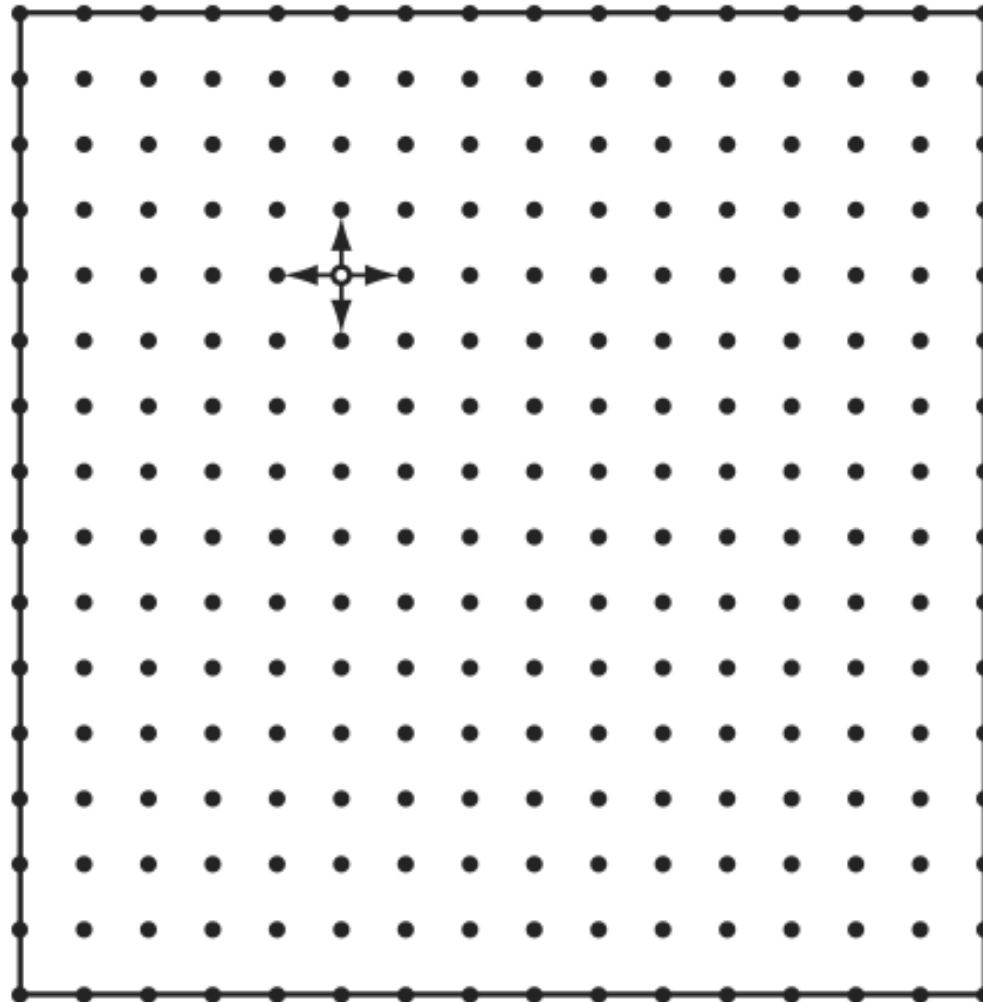
- Segunda execução: com **balanceamento** após 11 iterações (ver chamada a MPI_Migrate no código)
- Executar:
 - `./charmrun +p2 ./work.AMPI-LB +vp 16 +balancer GreedyLB`
 - Notar surgimento dos arquivos de traço de execução
 - Novo tempo reportado (Cray-XT6): 12.4 s
 - Ganho de Desempenho: 30%



- Após a iteração 11, as cargas ficam equilibradas
- Pode-se ver (arq. out_ampi-lb) quais threads migraram!

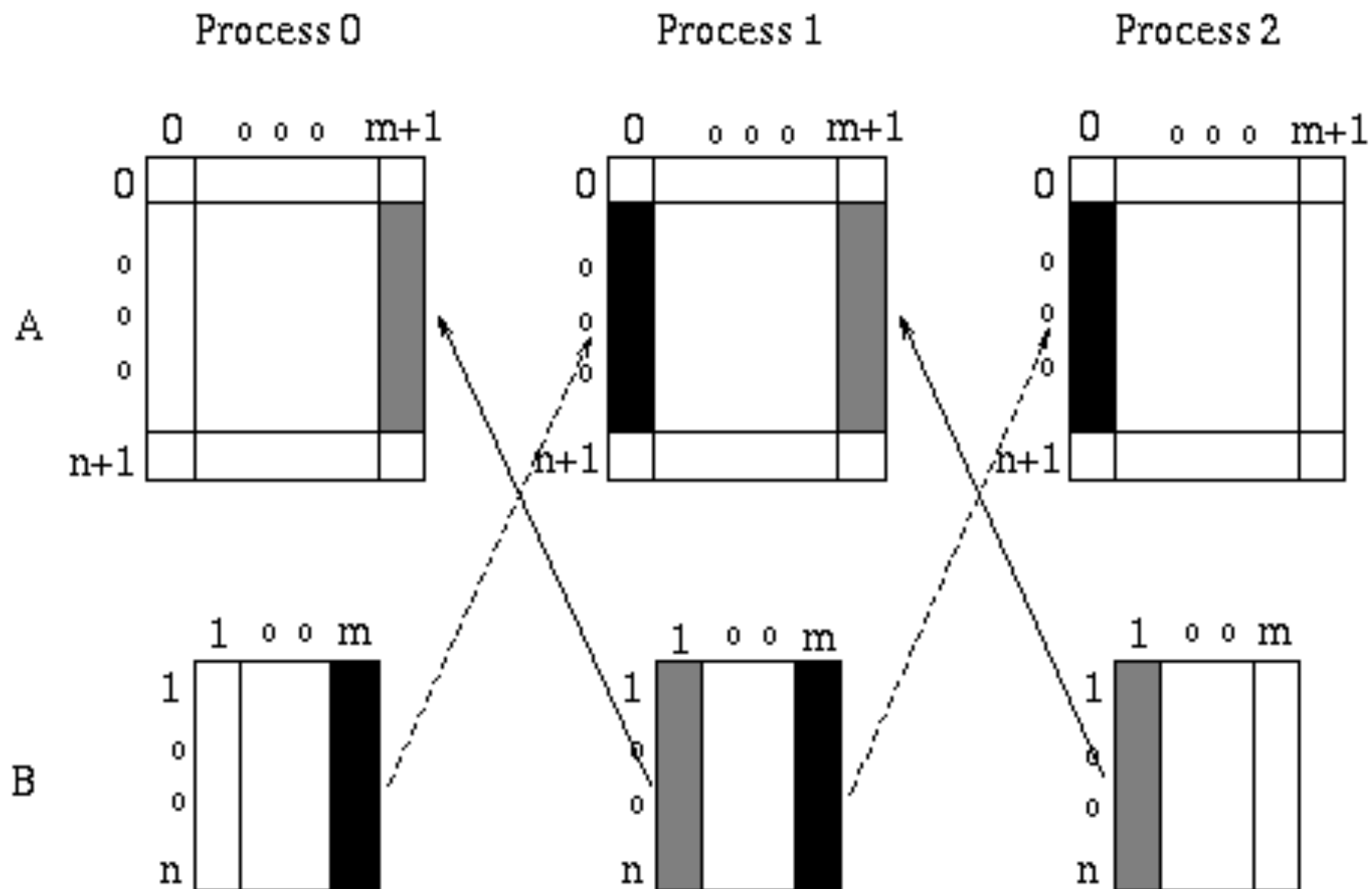
Projections - Exemplo 3

Jacobi com Partição 1-D



Projections - Exemplo 3

Jacobi com Partição 1-D



(figura do Netlib)

Projections - Exemplo 3

Jacobi com Partição 1-D

- Exemplo: 8 processadores;
- Exemplo: 8 processadores e 64 processadores virtuais (VPs ou *threads*);
- Exemplo: 8 processadores e 64 processadores virtuais (*threads*) com desbalanceamento de carga.

Tópicos do MiniCurso

- Parte I: Virtualização de Programas Paralelos
 - Introdução ao Charm++
 - Características Básicas do Adaptive-MPI
 - Virtualização em Programas MPI
 - Migração entre Processadores
 - Instrumentação e Visualização de Desempenho
- Parte II: Balanceamento Dinâmico de Carga
 - Balanceamento Centralizado × Distribuído
 - Balanceamento por Carga Computacional ou Comunicação
 - Heurísticas para Balanceamento
 - Exemplos em Aplicações Científicas

Que processador(es) estima(m) a carga do sistema e toma(m) a decisão do que migrar?

- Balanceador Centralizado
- Balanceador Distribuído
- Balanceador Híbrido

Que processador(es) estima(m) a carga do sistema e toma(m) a decisão do que migrar?

● Balanceador Centralizado

- Balanceadores de carga centralizados têm a vantagem de ter acesso à carga global e a toda a informação de comunicação do sistema. Isso permite que esse tipo de balanceador produza uma melhor decisão de balanceamento de carga.
- Por outro lado, balanceadores de carga centralizados são inerentemente não-escaláveis.

● Balanceador Distribuído

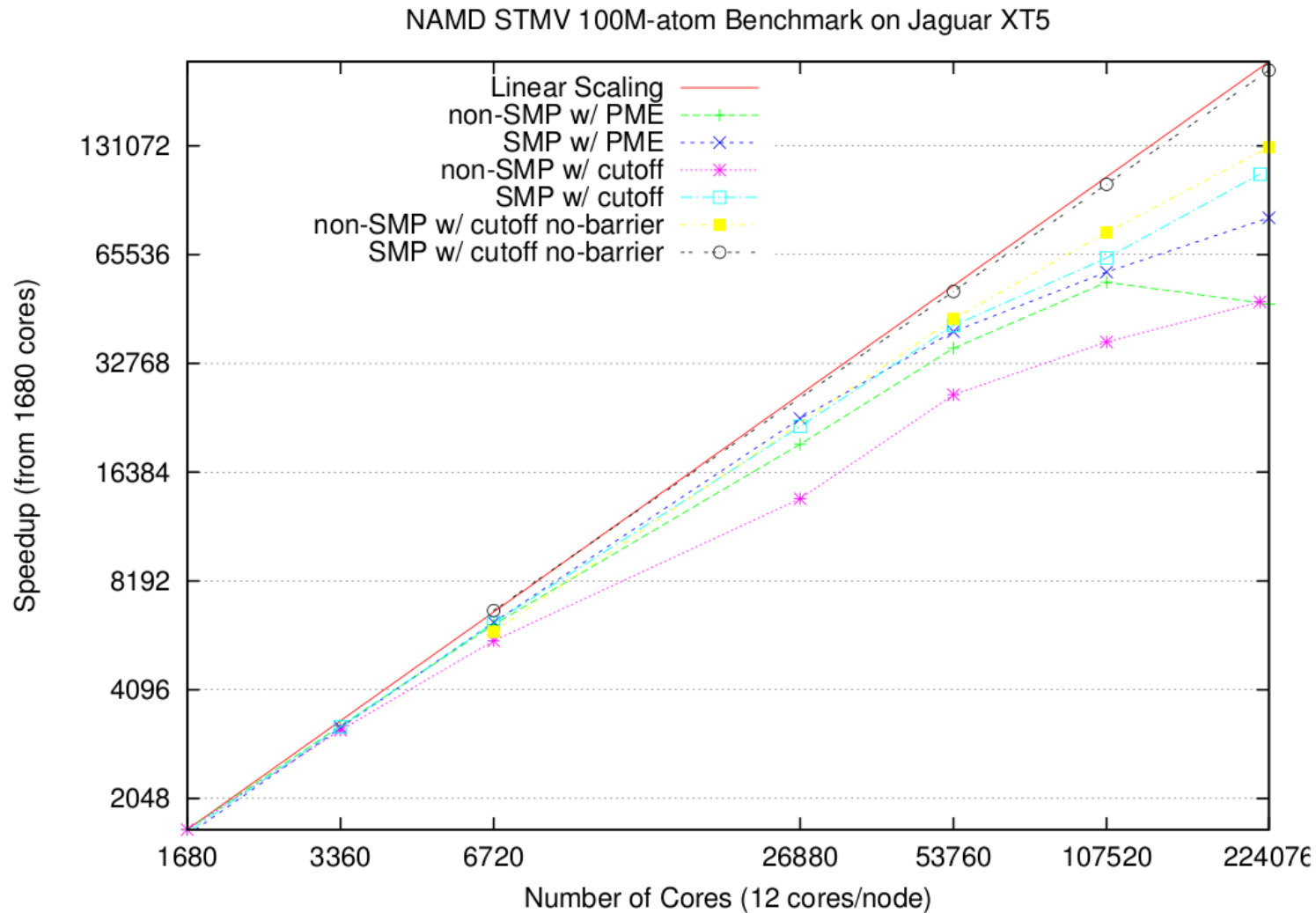
- □ Balanceadores distribuídos são escaláveis
- Entretanto, a decisão de balanceamento não leva mais em conta a informação global de carga e comunicação.

Que processador(es) estima(m) a carga do sistema e toma(m) a decisão do que migrar?

● Balanceador Híbrido

- **Balanceadores de carga híbridos** combinam características de balanceadores centralizados e distribuídos, tentando extrair as **vantagens** de cada um deles e **evitar** as suas limitações
- ☐ **Balanceadores híbridos são escaláveis** para sistemas PetaFlop atuais
- Entretanto, **ainda não se sabe se eles serão escaláveis para sistemas Exaflop futuros**
- Como são mais complexos, e como podem ser desenvolvidos a partir de balanceadores distribuídos, por razões didáticas não serão abordados neste MiniCurso
- Exemplo: HybridLB do Charm++, com NAMD

Balanceador de Carga Híbrido: Exemplo de Resultado com NAMD num Cray-XT5



(Mei, C. *et al*, "Enabling and Scaling Biomolecular Simulations of 100~Million Atoms on Petascale Machines with a Multicore-optimized Message-driven Runtime", accepted for Supercomputing'2011, Seattle, Nov.2011.)

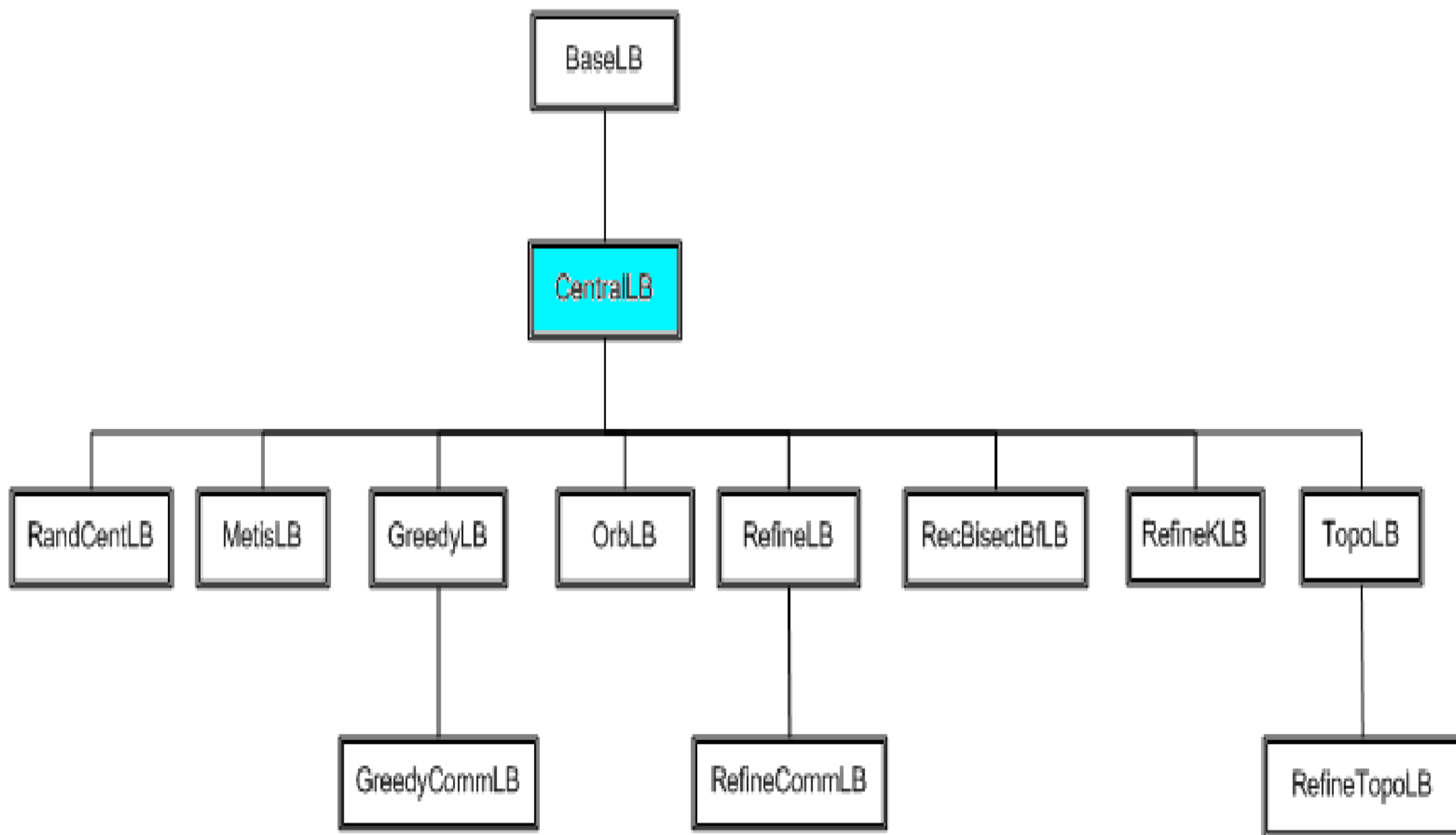
O que vamos ver nessa segunda parte do MiniCurso

- Como escrever um balanceador de carga centralizado em Charm++;
- Como escrever um balanceador distribuído em Charm++;
- Exemplos de balanceadores de carga;
- Resultados experimentais em uma aplicação real.

Balancedores de Carga Centralizados: Princípios Gerais

- Forma típica de operação:
 - Cada elemento mantém medida de sua **carga computacional** ou de seu **volume de comunicação** com outros elementos
 - No momento do balanceamento, as informações individuais são **agrupadas em um processador central**
 - O processador central **toma as decisões** para balanceamento
 - Decisões tomadas são **comunicadas aos processadores**
 - Cada processador **envia ou recebe carga extra**, de acordo com a decisão central, e o processamento **reinicia**
- Características marcantes:
 - O balanceamento é uma operação coletiva, **sincronizada**
 - Exige **agrupamento** de informações e **distribuição** de decisões
 - Tomada de decisão requer juntar informações **num único** processador central

Hierarquia dos Balanceadores de Carga Centralizados no Charm++



Criação de uma Nova Estratégia de Balanceamento

- Implementar uma sub-classe de **CentralLB** que sobrescreve o método **work (...)**.

```
class FooLB : public CentralLB {  
    public:  
    ...  
    void work (CentralLB::LDStats* stats);  
    ...  
};
```


Banco de Dados de Balanceamento de Carga

- Objetivo: armazenar dados relativos às cargas medidas

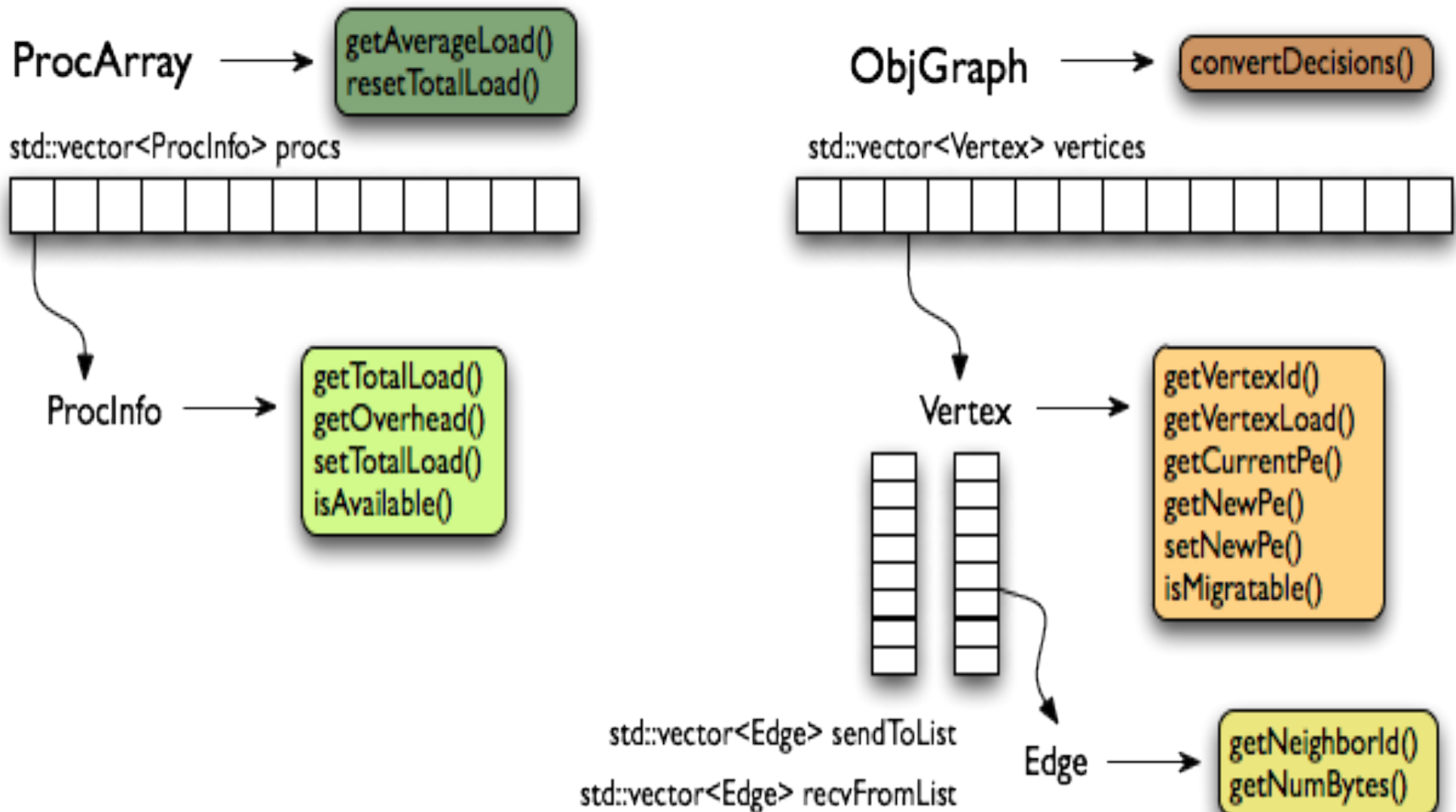
```
struct LDStats {  
    ProcStats*   procs;  
    LDObjData*   objData;  
    LDCommData*  commData;  
    int *to_proc;  
    ...  
}
```

Banco de Dados de Balanceamento de Carga

Atribui todas as threads ao processador 0

```
void fooLB::work(CentralLB::LDStats* \
                stats,int count){
    for(int i=0;i < stats->n_objs; i++)
        stats->to_proc[i] = 0;
}
```

Banco de Dados de Balanceamento de Carga



Compilação e Integração ao Charm++

No sub-diretório **tmp** onde foi instalado o Charm++:

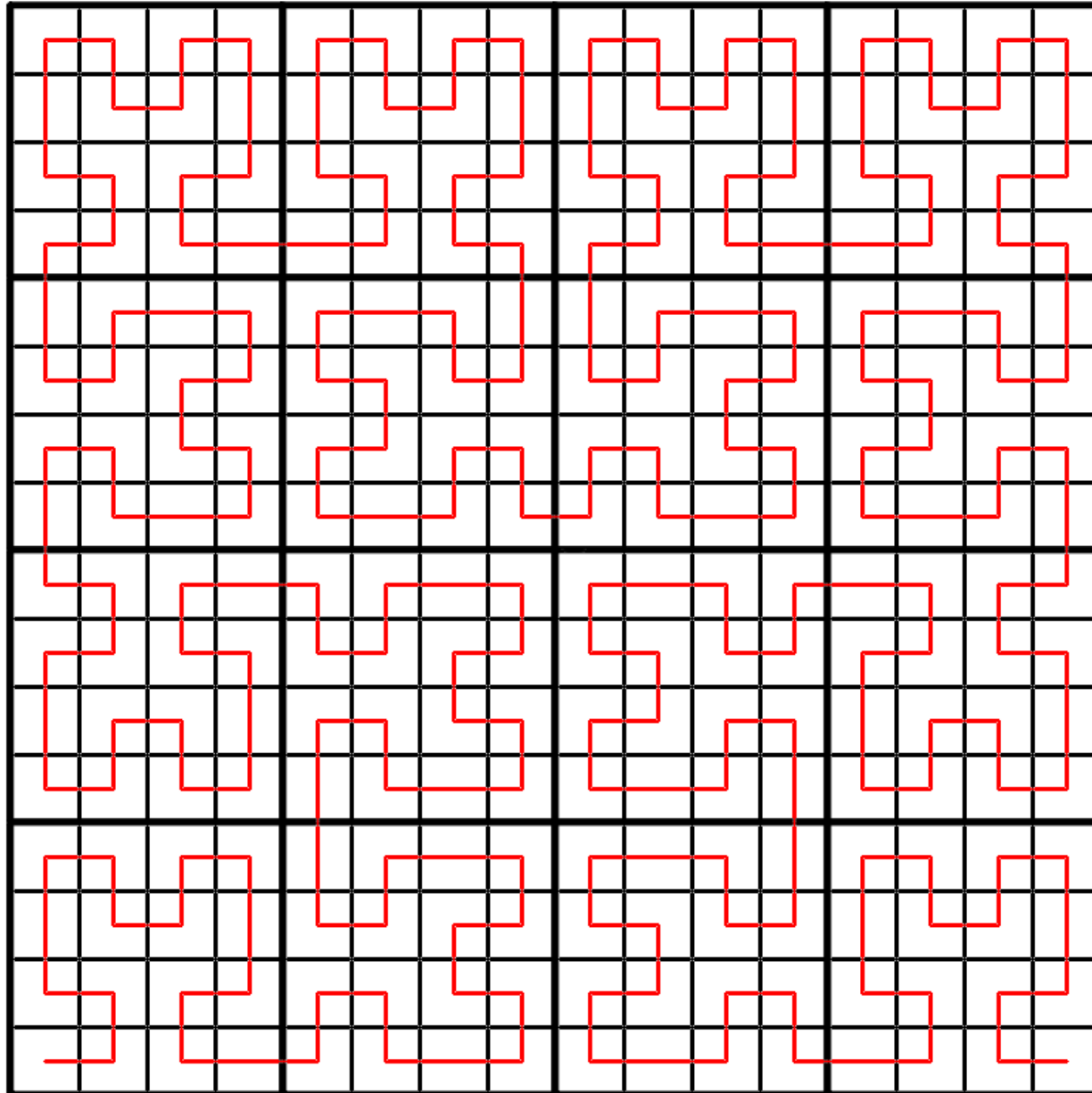
- Editar e rodar **Makefile_lb.sh**
 - Isso cria o arquivo Make.lb que é incluído no Makefile principal
- Gerar a interface do balanceador:
 - `charmcc FooLB.ci`
- Rodar **make depends** para corrigir dependências
- Reconstruir o Charm++

Agora o balanceador FooLB está disponível

(mostrar esse exemplo)

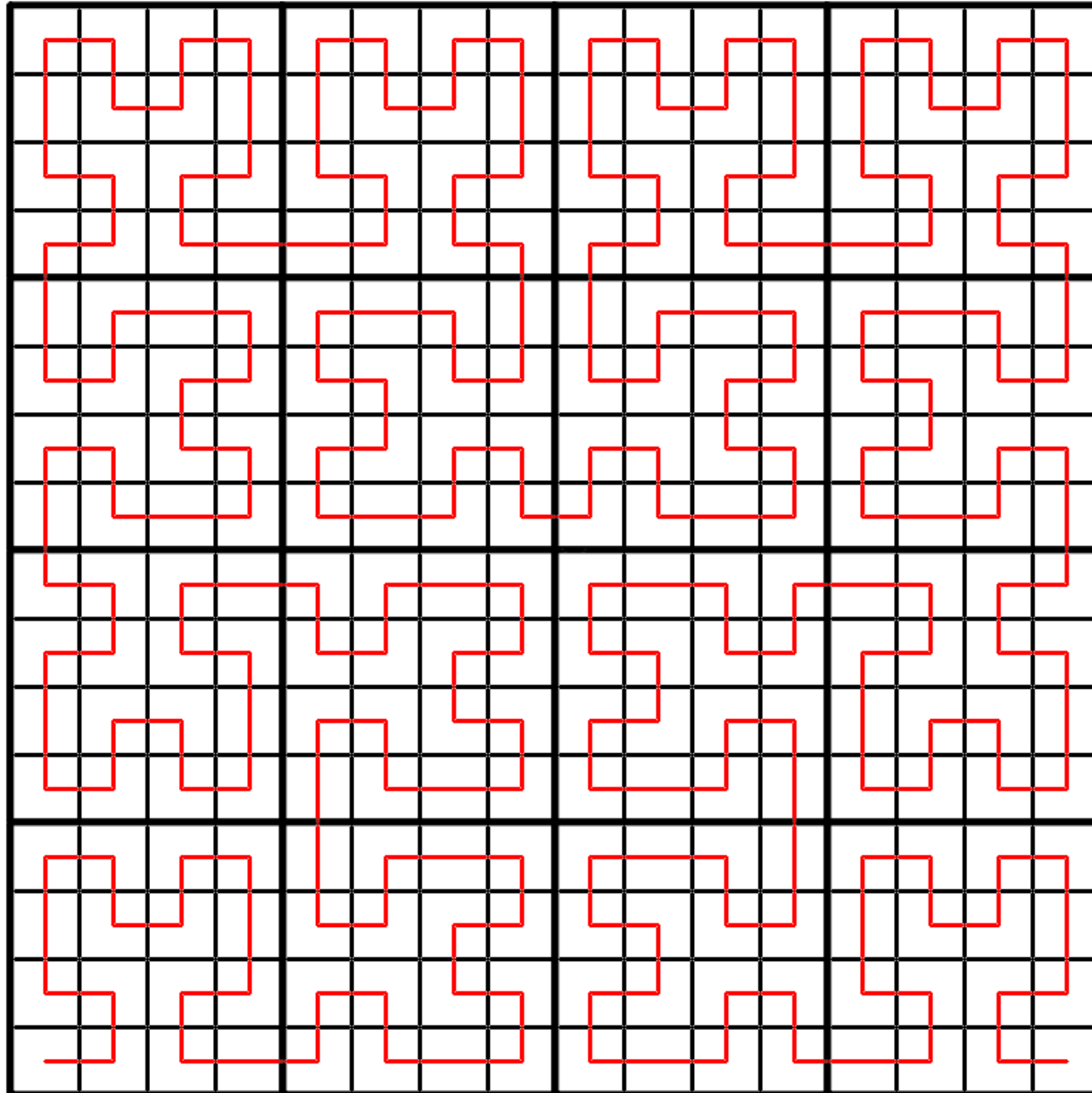
Exemplo de Balanceador de Carga Centralizado

Curva de Hilbert: mapeia um espaço N-dimensional em um espaço 1D



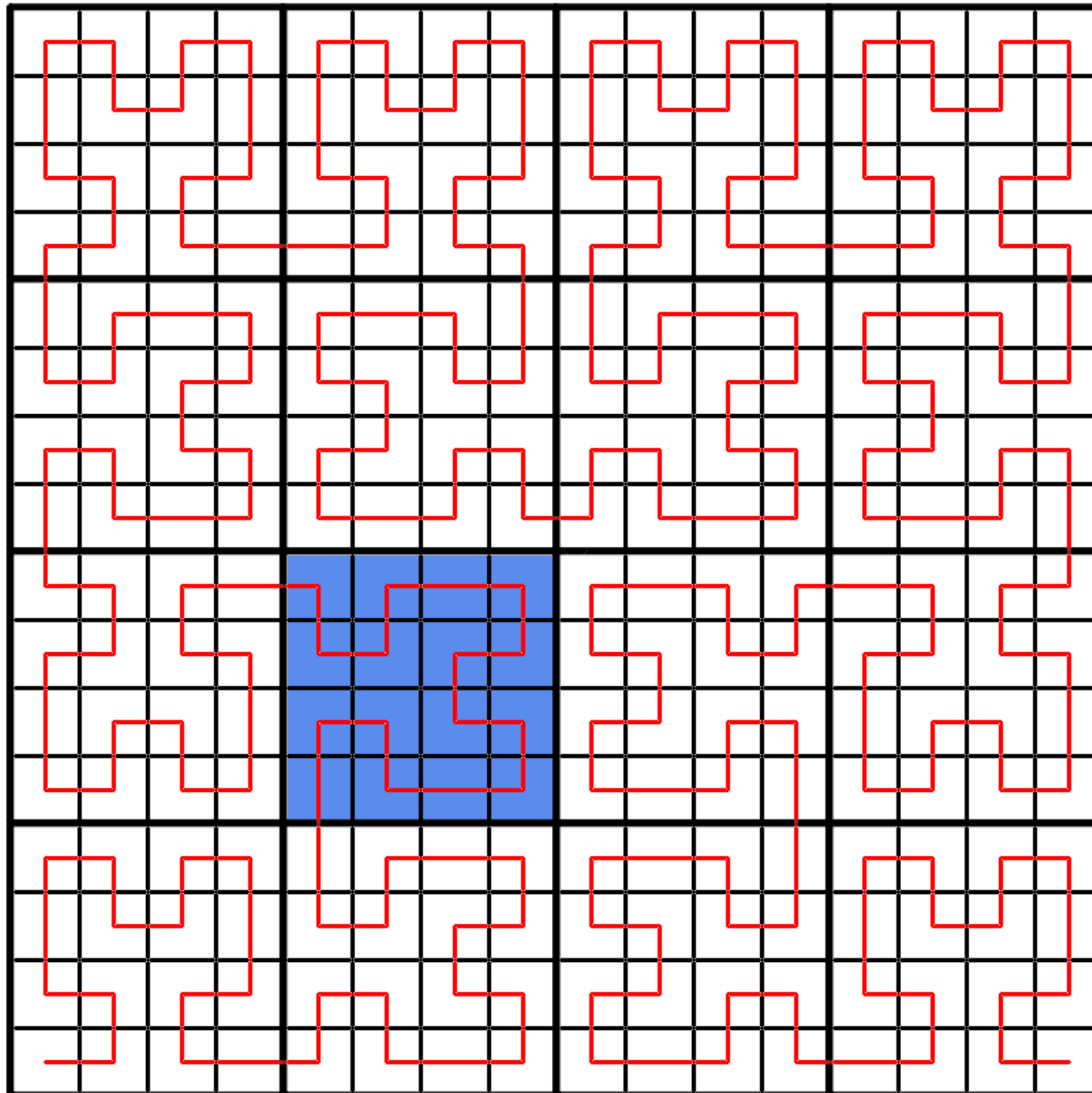
Curva de Hilbert

pontos próximos na curva estão também próximos no espaço N-D



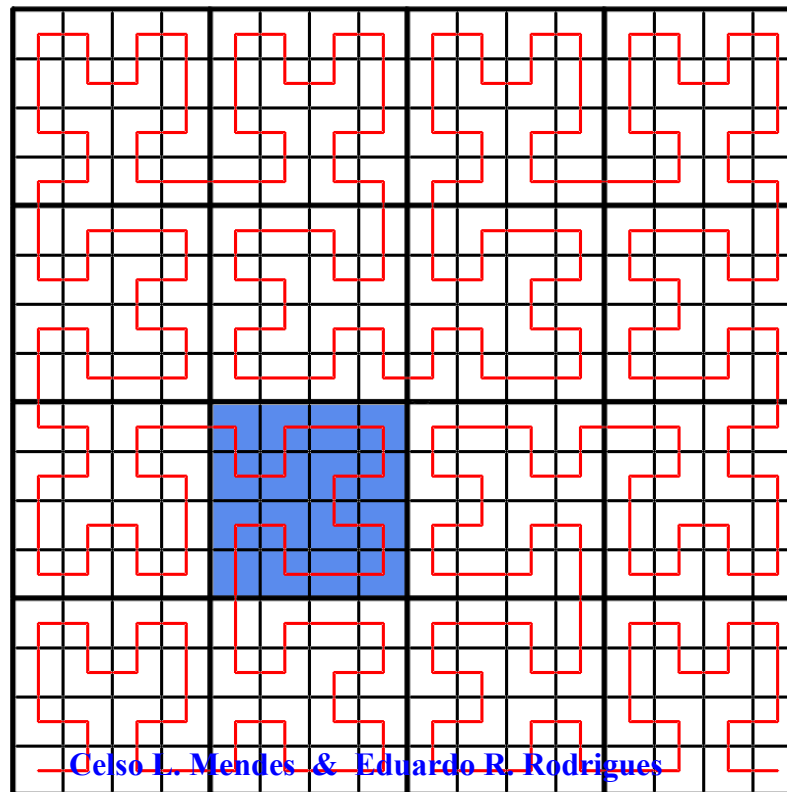
Curva de Hilbert

nessa figura há 256 elementos, sendo 16 elementos no quadrado azul



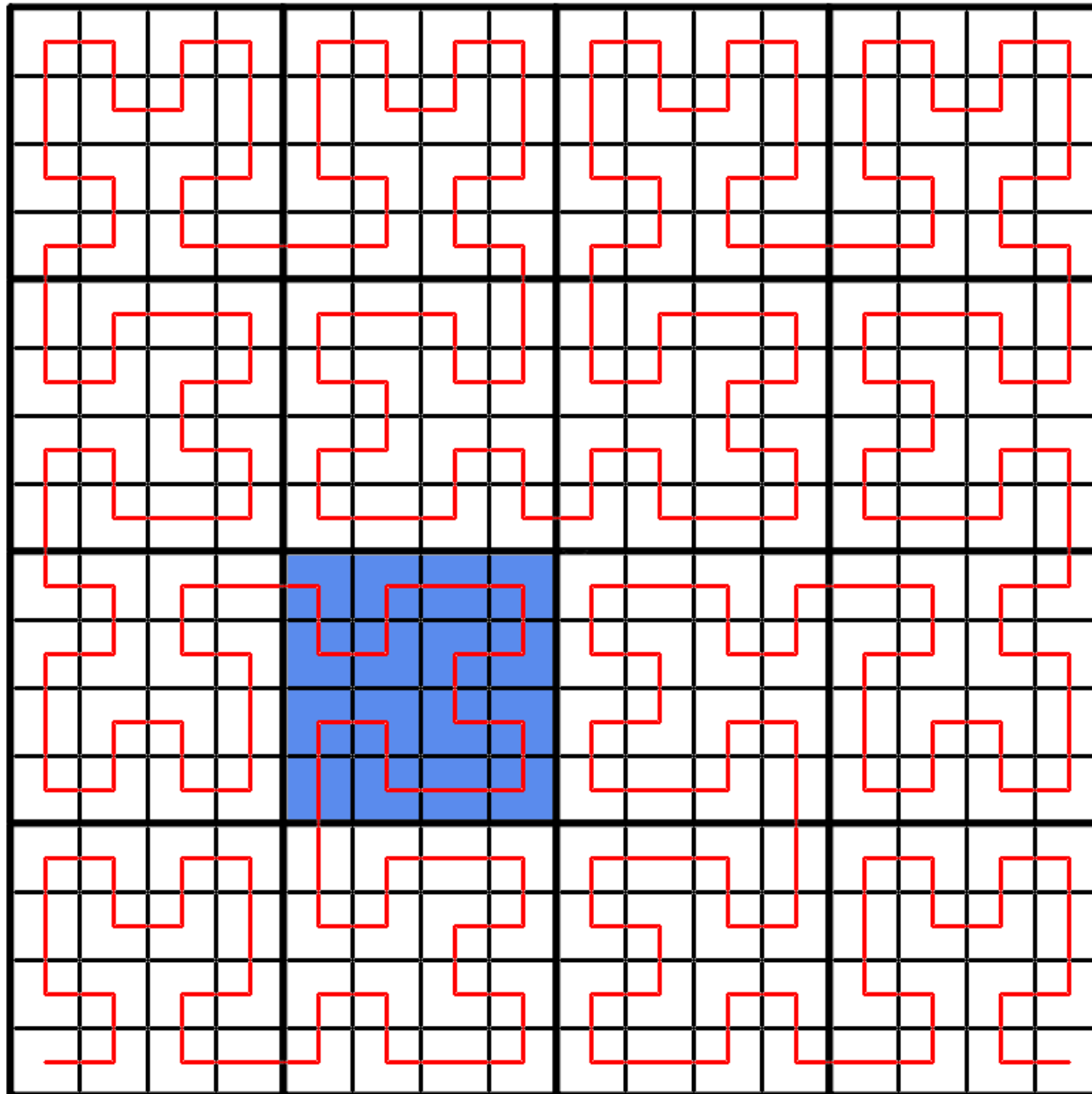
Curva de Hilbert

- Numa aplicação paralela em que o **domínio é regular** e a **decomposição é espacial**, podemos usar a curva de **Hilbert** para mapear *threads* para processadores;
- Nessa figura, por exemplo, temos **256 threads** e **16 processadores** (o quadrado azul corresponde a um processador contendo 16 *threads*)
- Dessa forma, *threads* que se comunicam ficaram **naturalmente próximas umas das outras**.



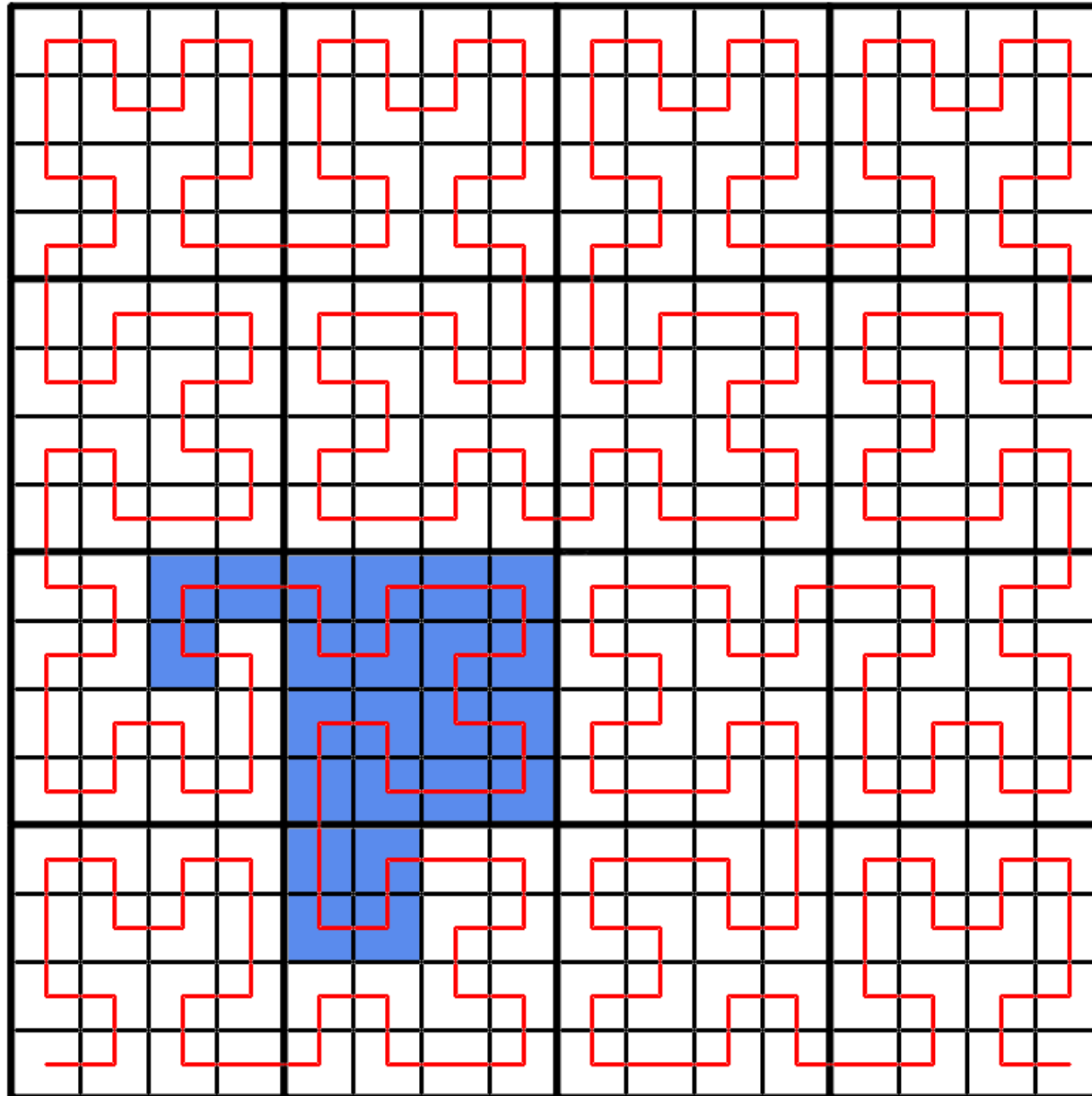
Para Balancear Carga

termos que cortar essa curva



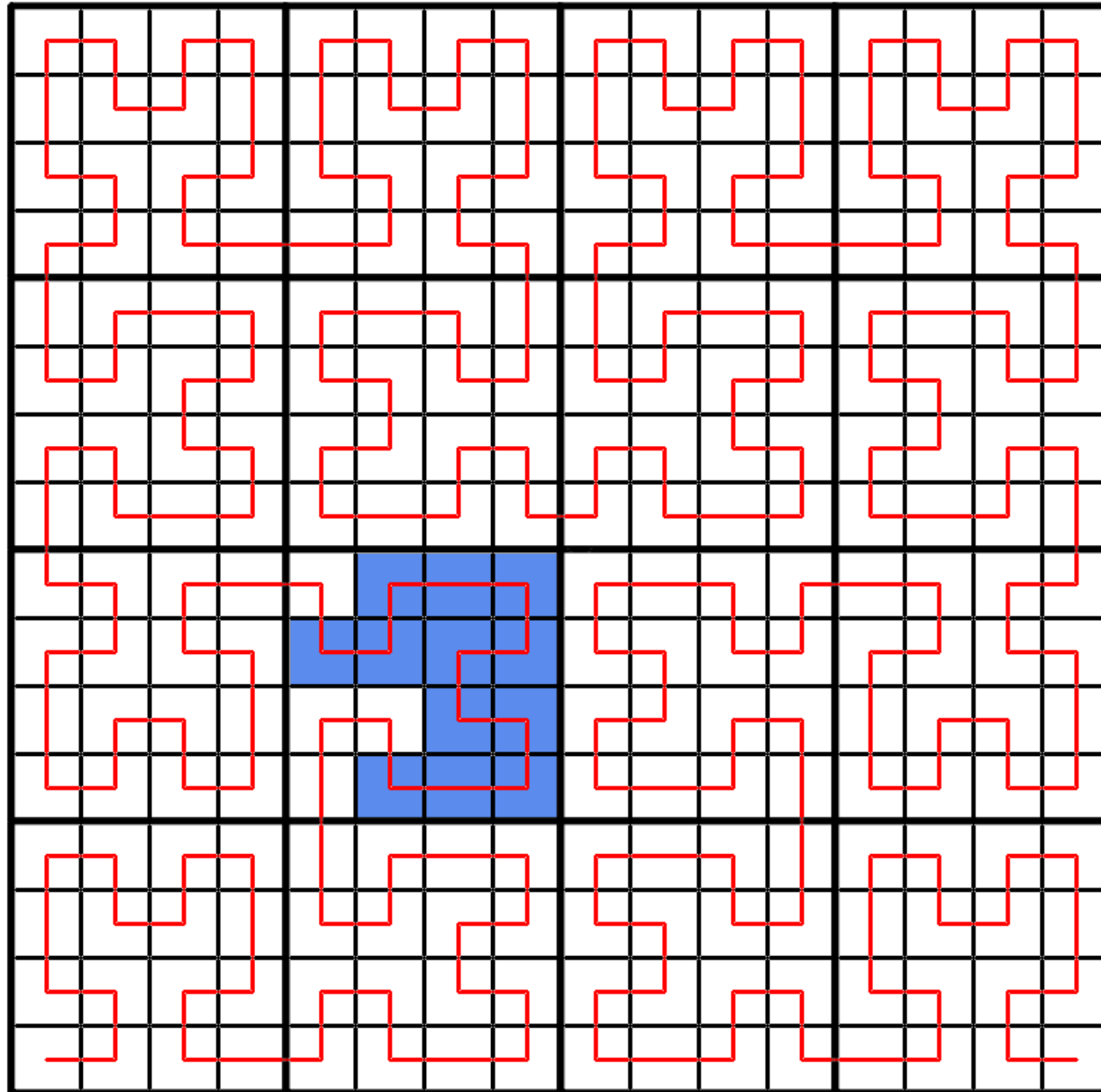
Para Balancear Carga

podemos expandir...

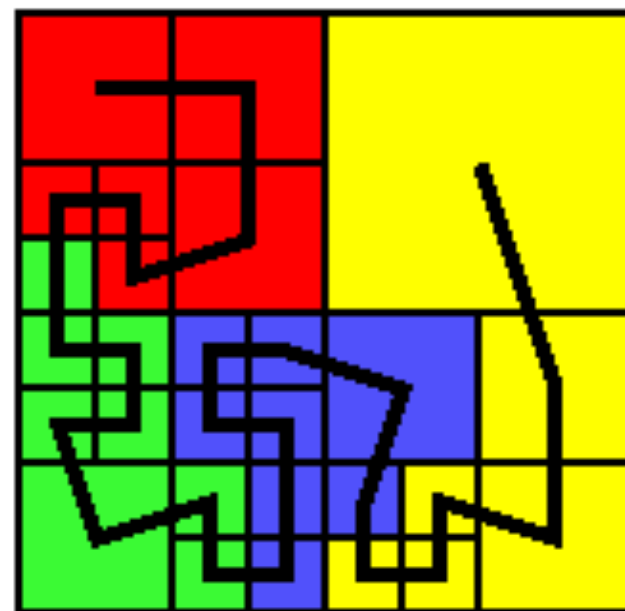
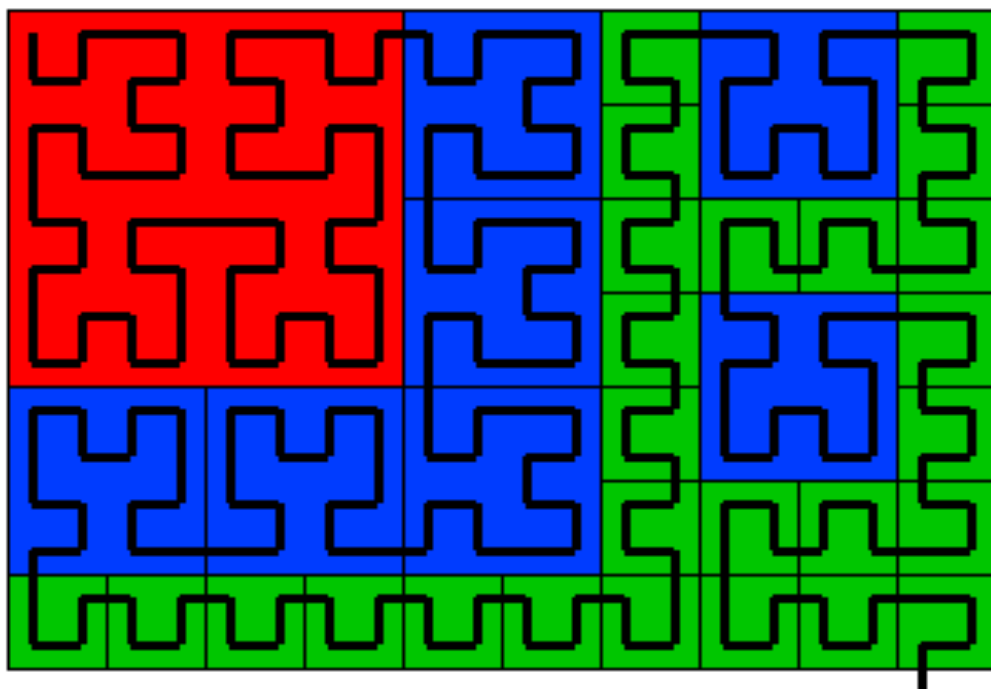


Para Balancear Carga

... ou retrain o segmento azul de acordo com a carga das threads ao longo da curva

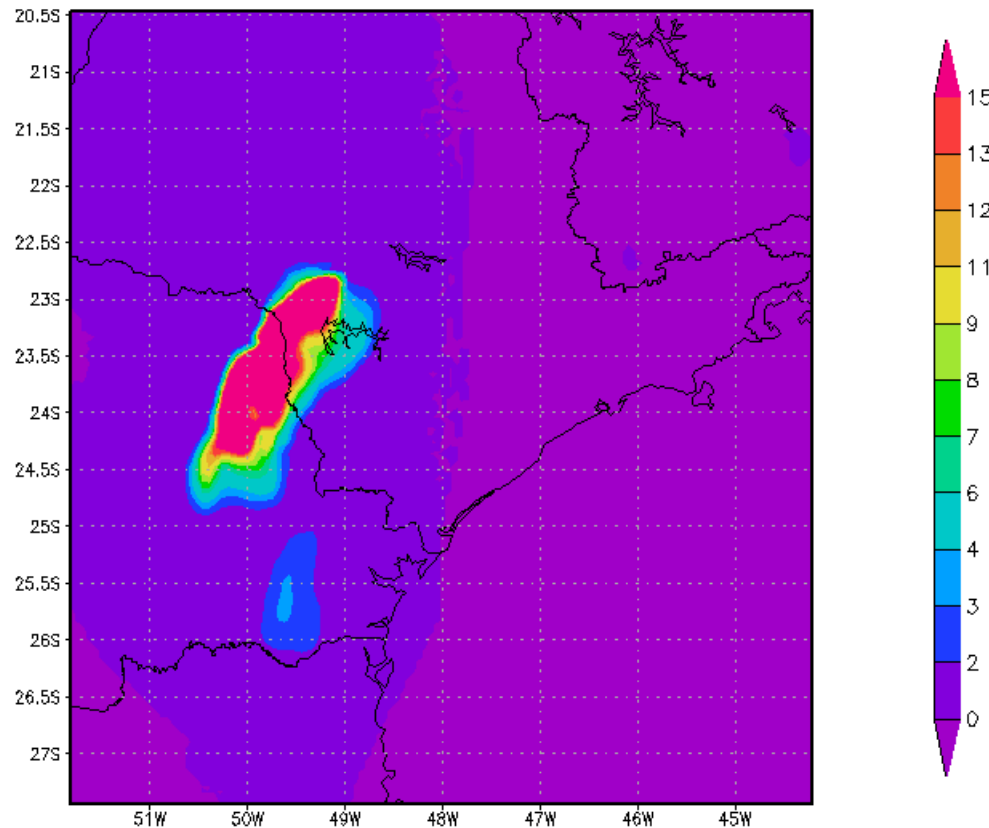


Limitações do Formato do Domínio podem ser Contornadas



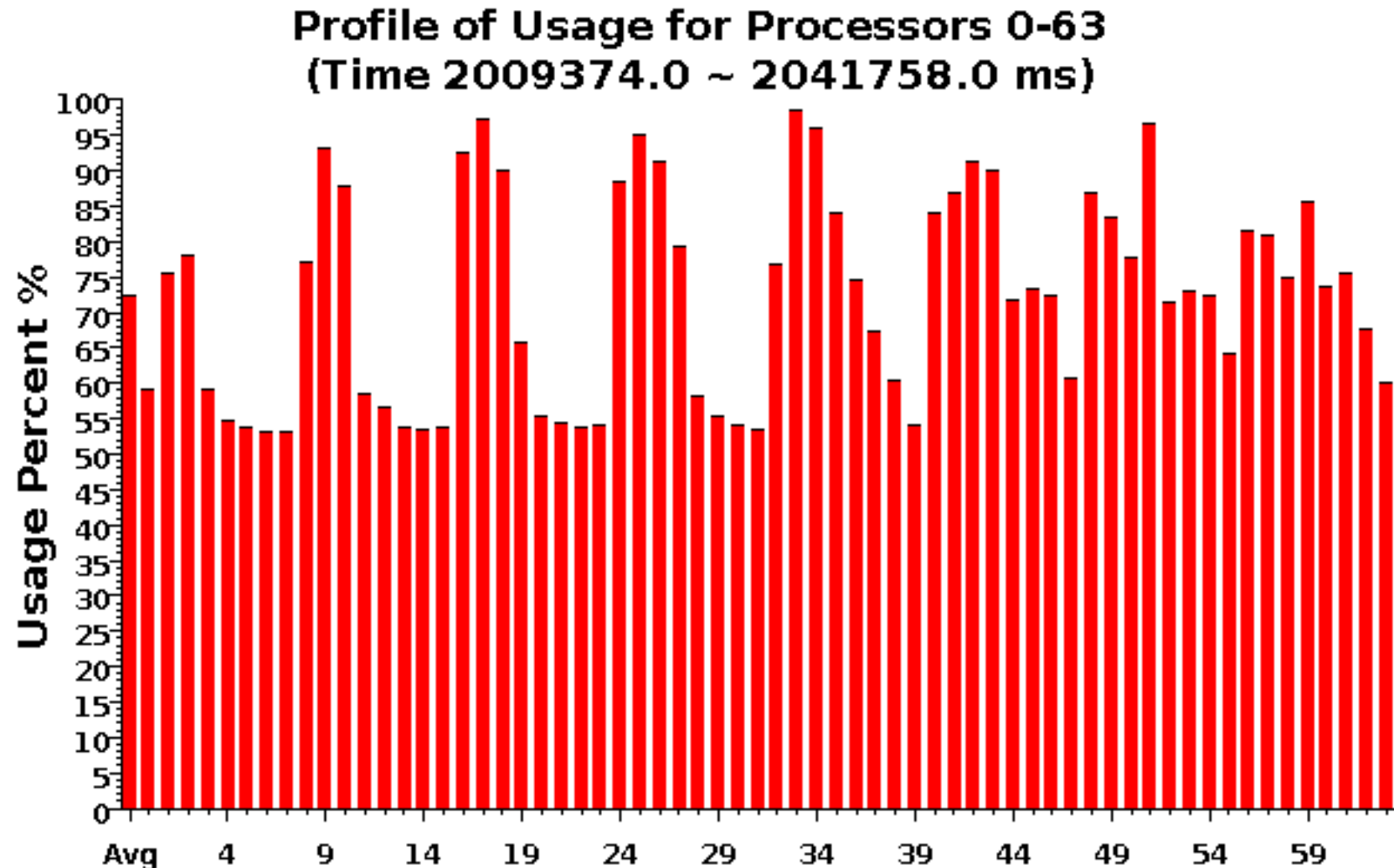
Exemplos em Aplicações Científicas

- ☐ Aqui usamos como exemplo o modelo meteorológico **BRAMS**;



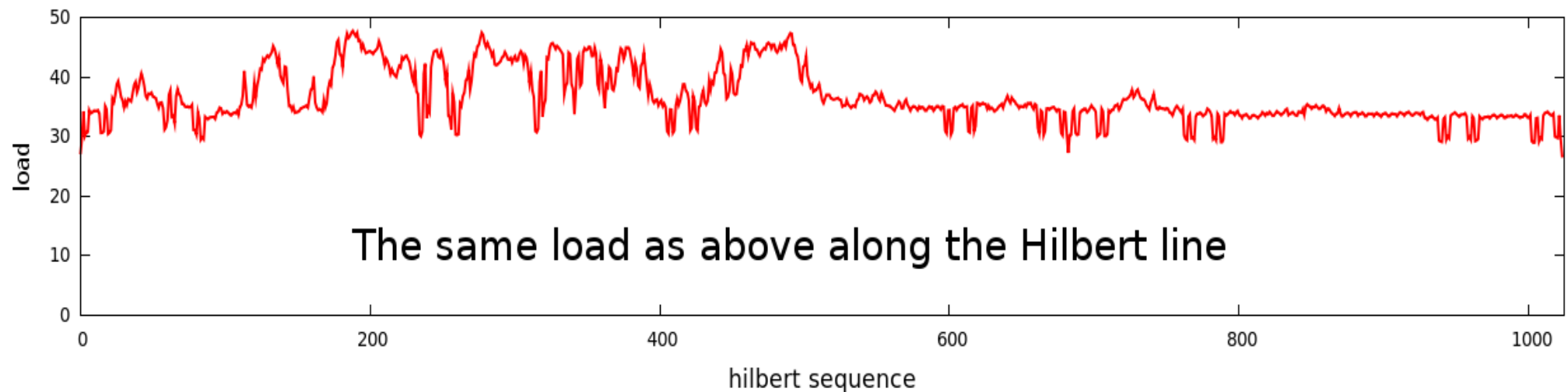
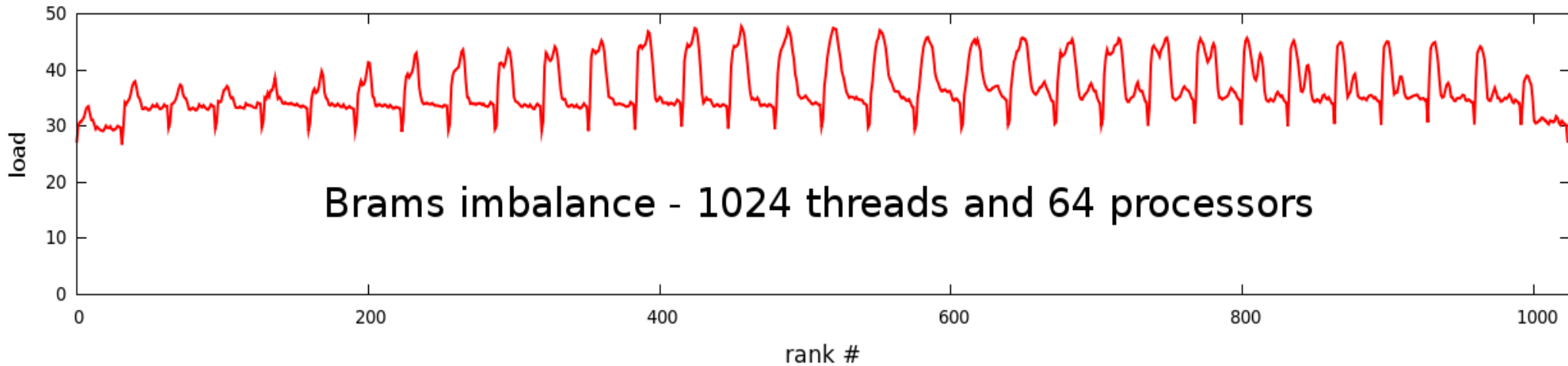
Exemplos em Aplicações Científicas

- 64 processadores e 1024 *threads*



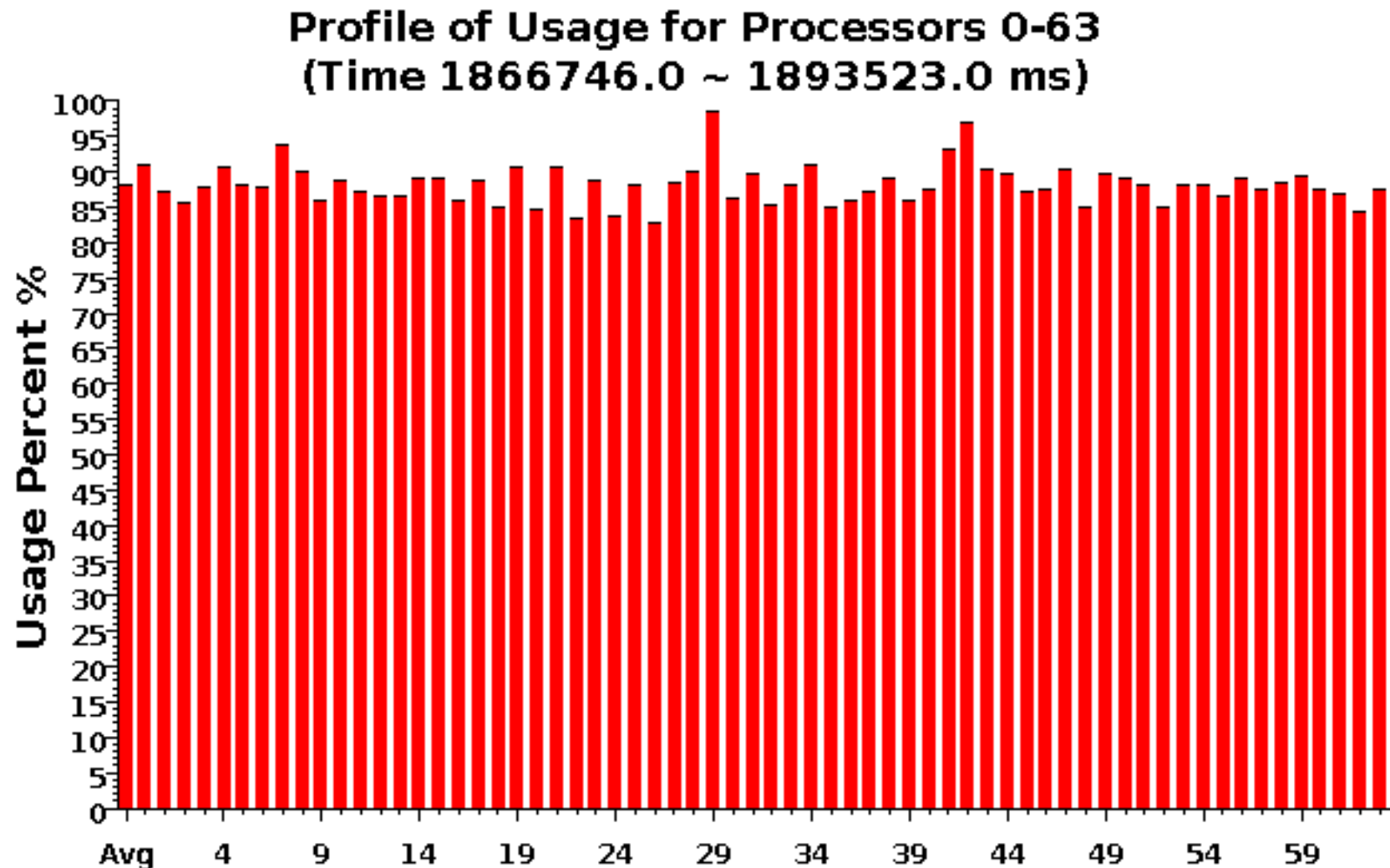
Exemplos em Aplicações Científicas

- 64 processadores e 1024 *threads*



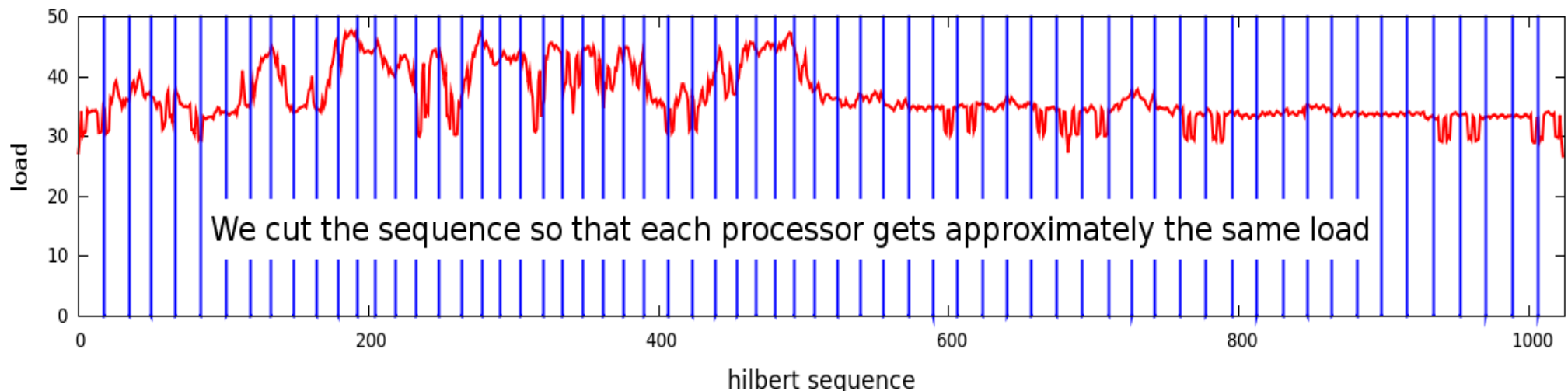
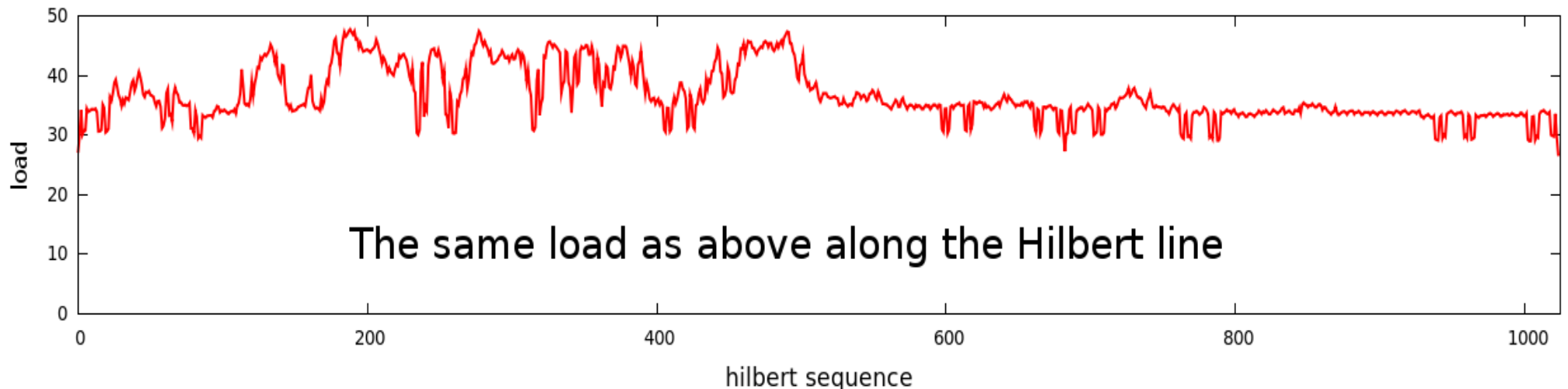
Exemplos em Aplicações Científicas

- Usando-se o balanceador de carga baseado na Curva de Hilbert (*HilbertLB*)

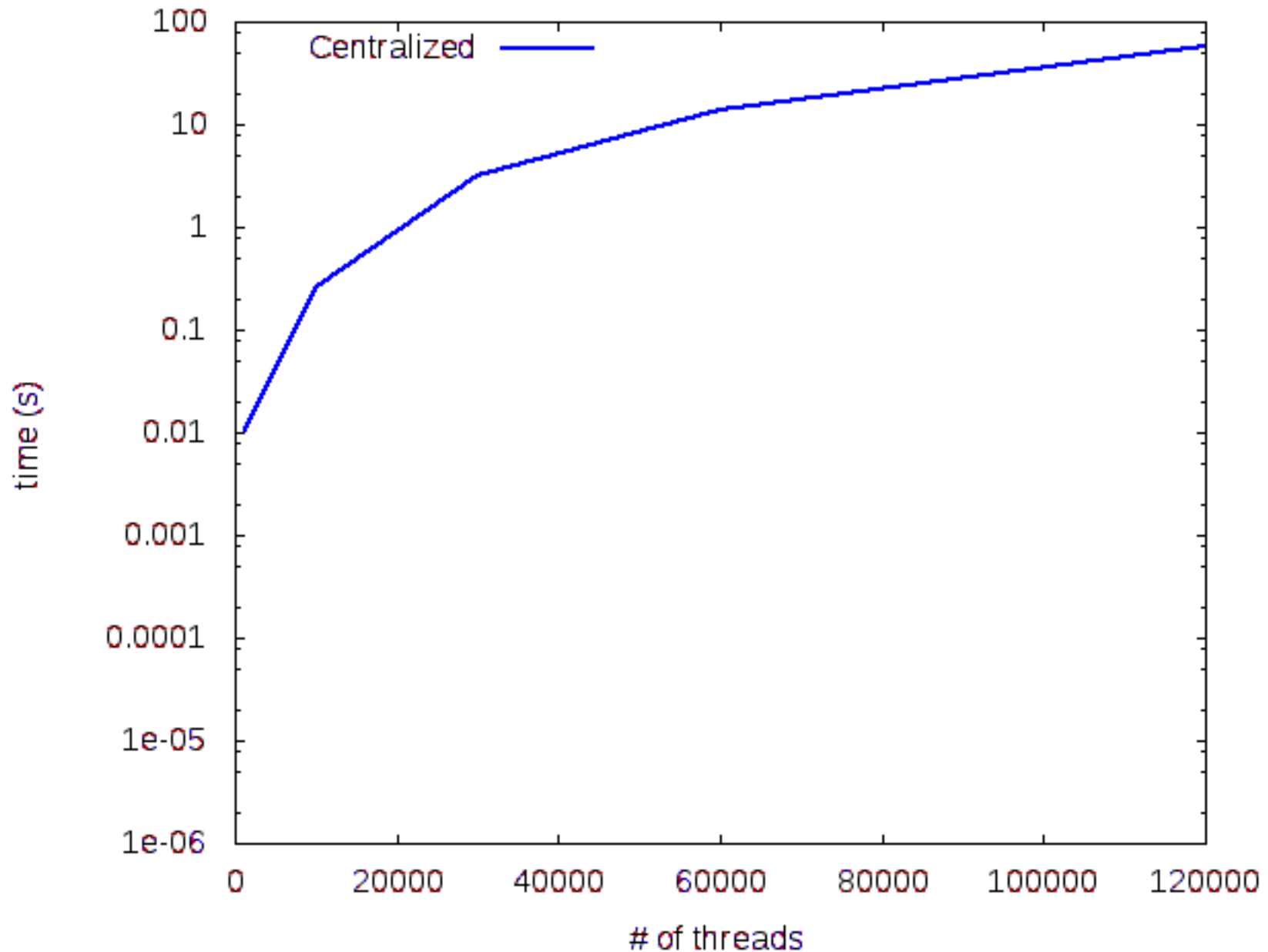


Exemplos em Aplicações Científicas

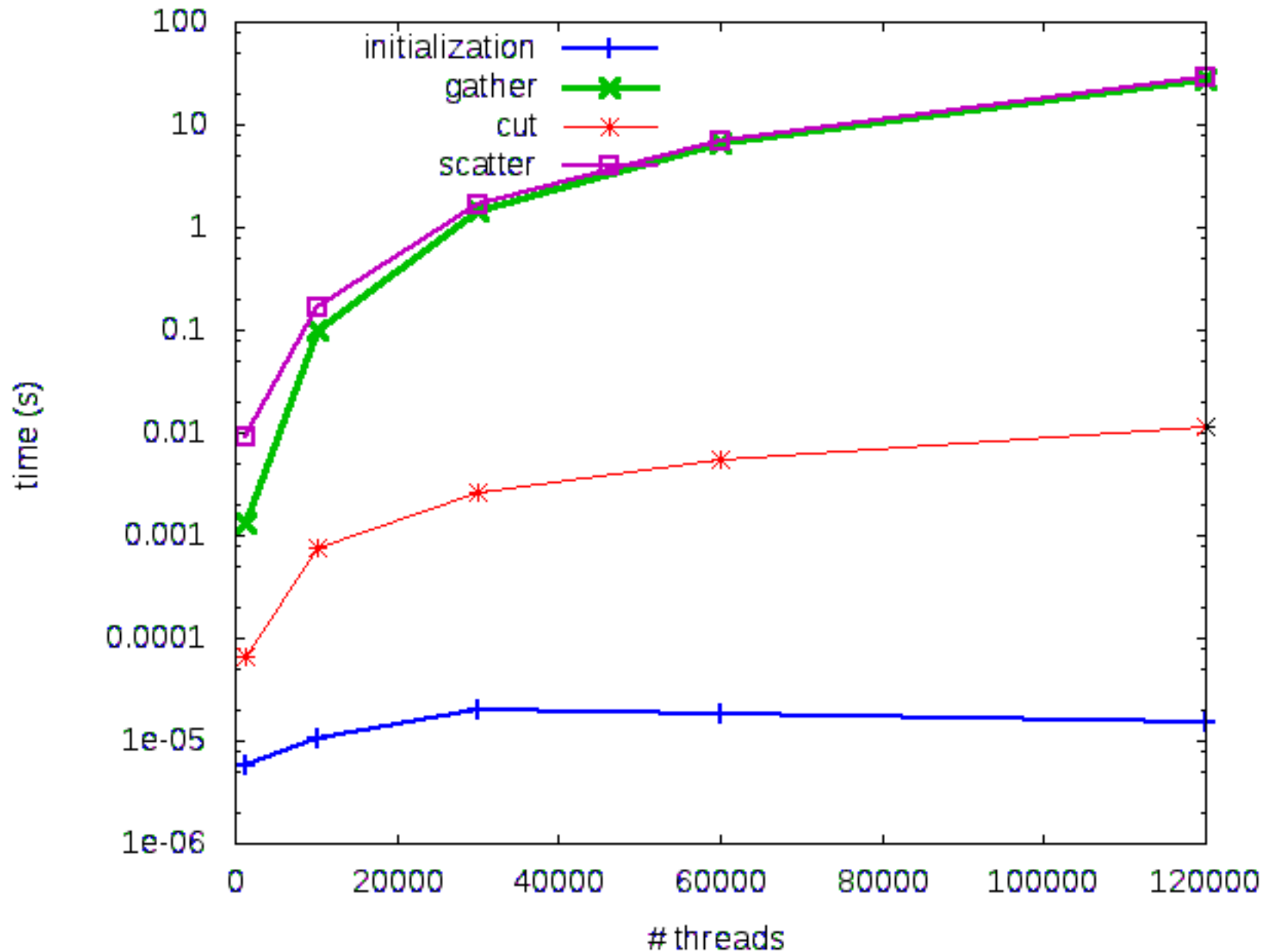
- Usando-se o balanceador de carga baseado na Curva de Hilbert (*HilbertLB*)



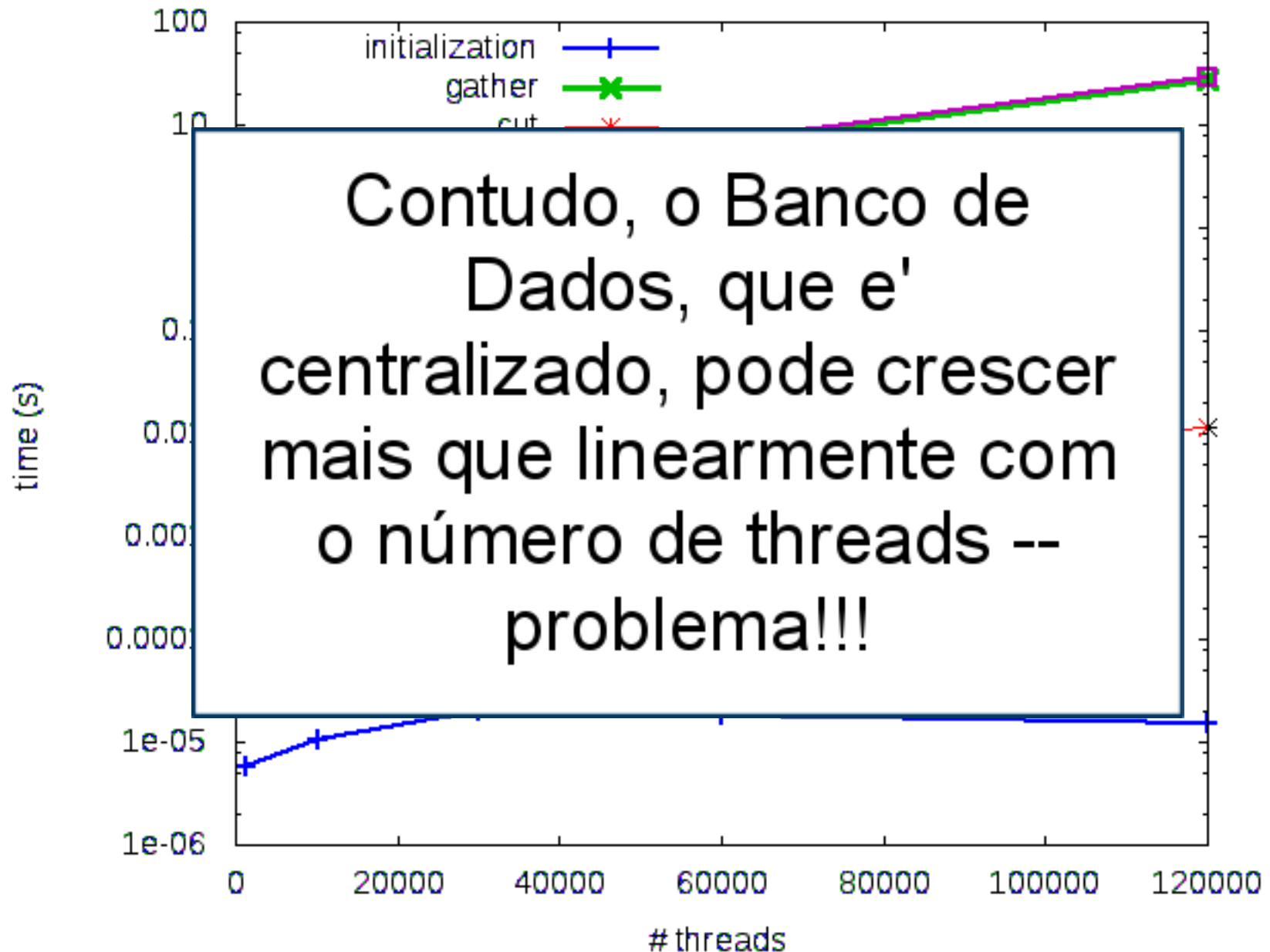
Escalabilidade do Balanceador de Carga Centralizado



Componentes do Balanceador de Carga Centralizado



Componentes do Balanceador de Carga Centralizado



Balancedador de Carga Distribuido

- Como no balanceador centralizado, o distribuído também é **escrito em Charm++**;
- Mas agora, nós temos que considerar alguns detalhes que antes estavam escondidos.

```
module DistribLB {  
  
    initnode void lbinit(void);  
  
    group DistribLB: BaseLB {  
        entry void DistribLB(const CkLBOptions &);  
        entry void recvMsgLoad(int pe, double load);  
    };  
  
};
```

Balanceador de Carga Distribuído

- Dessa vez, não iremos usar a superclasse CentralLB, portanto temos de **registrar nosso balanceador de carga** nós mesmos;
- Isso é feito com uma função declarada como **initnode**, que significa que ela é executada em cada um dos *nodes* antes da computação começar.

```
module DistribLB {  
  
    initnode void lbinit(void);  
  
    group DistribLB: BaseLB {  
        entry void DistribLB(const CkLBOptions &);  
        entry void recvMsgLoad(int pe, double load);  
    };  
  
};
```

Balanceador de Carga Distribuído

- A classe do Balanceador de Carga é declarada como **group**, isso significa que haverá um objeto desse tipo em cada *node* da máquina paralela;
- Essa classe também é declarada como subclasse de **BaseLB**, que provê alguns serviços básicos para o Balanceador de Carga.

```
module DistribLB {  
  
    initnode void lbinit(void);  
  
    group DistribLB: BaseLB {  
        entry void DistribLB(const CkLBOptions &);  
        entry void recvMsgLoad(int pe, double load);  
    };  
  
};
```

Balanceador de Carga Distribuído

- Nesse exemplo, nós apenas temos dois **métodos de entrada**; um construtor e um método que envia informação de carga para outros processadores;
- Essa **informação de carga** pode ser usada para tomada de decisão pelo Balanceador de carga.

```
module DistribLB {  
  
    initnode void lbinit(void);  
  
    group DistribLB: BaseLB {  
        entry void DistribLB(const CkLBOptions &);  
        entry void recvMsgLoad(int pe, double load);  
    };  
  
};
```


Código do Balanceador de Carga Distribuído

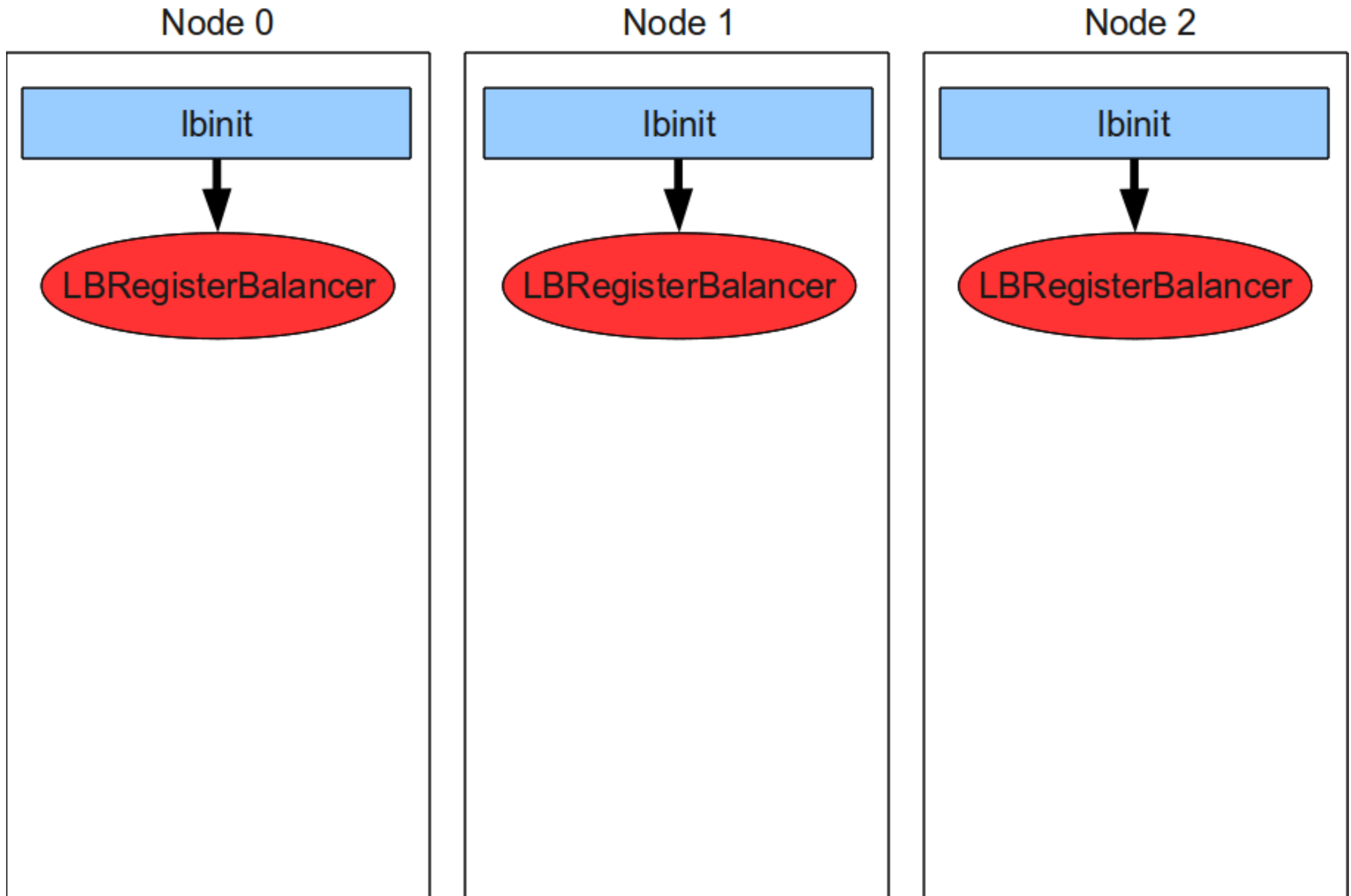
- Nessa declaração de classe, temos uma (bem simples) estrutura para um balanceador de carga distribuído.

...

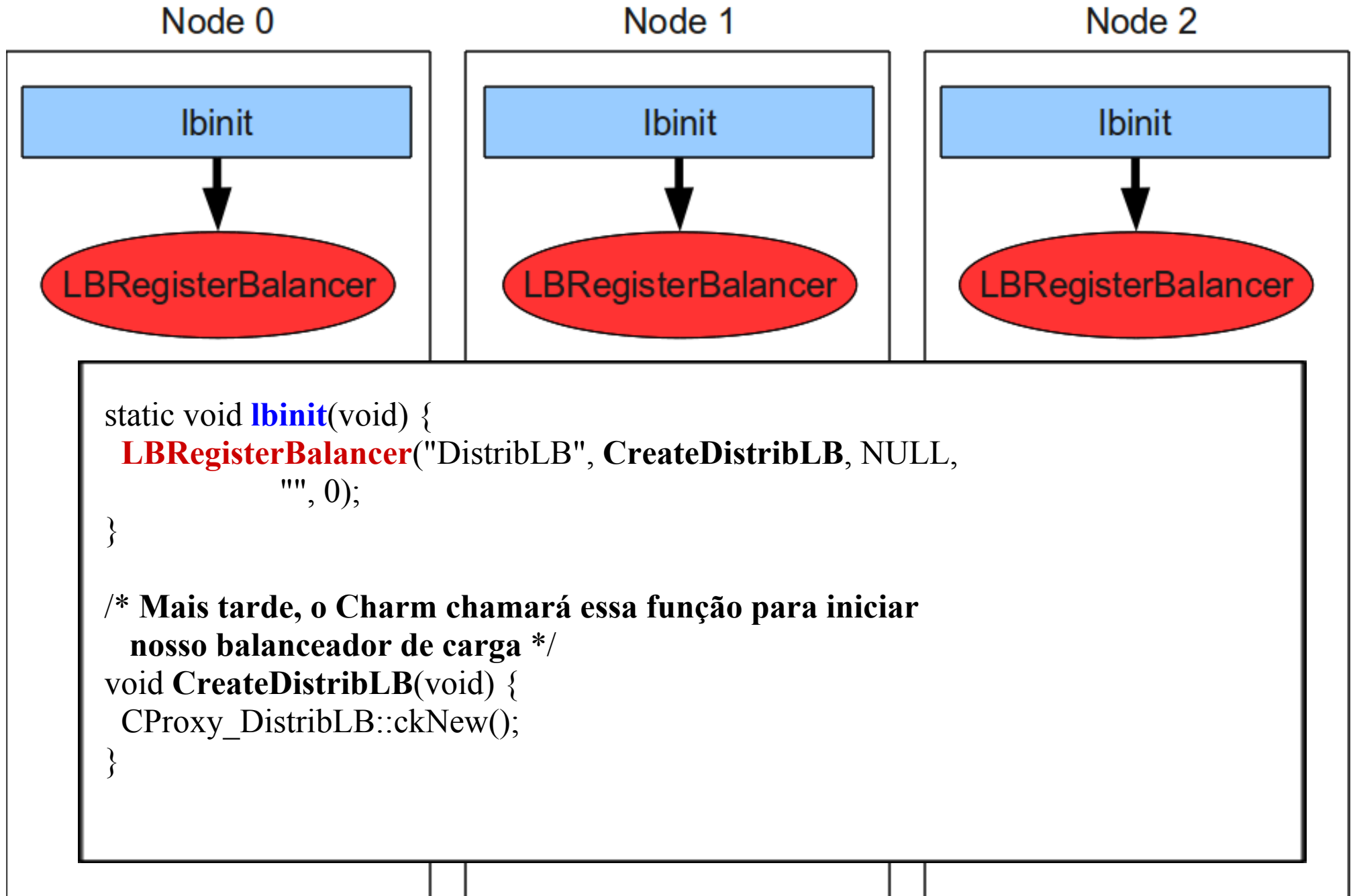
```
class DistribLB : public BaseLB {  
private:  
    int msgTotal;  
    CProxy_DistribLB thisProxy;  
public:  
    DistribLB(const CkLBOptions &opt);  
    ~DistribLB();  
  
    static void staticAtSync(void*);  
    static void staticMigrated(void*, LDObjHandle h,  
                               int waitBarrier);  
  
    void AtSync(void);  
    void Migrated(LDObjHandle h, int waitBarrier);  
    void recvMsgLoad(int, double);  
};
```

Vejamos o ciclo de vida dessa classe...

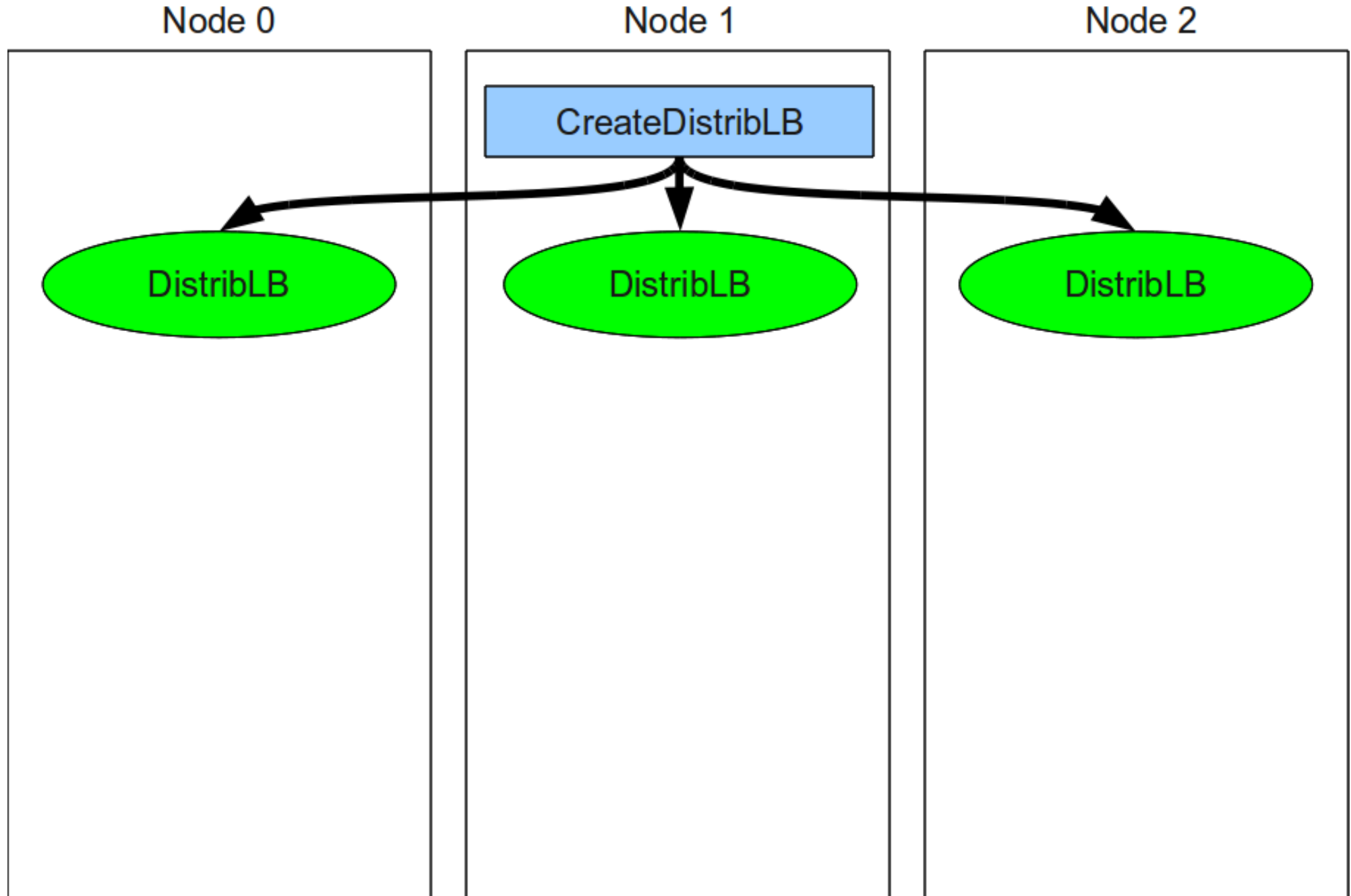
Ciclo de Vida do Balanceador de Carga



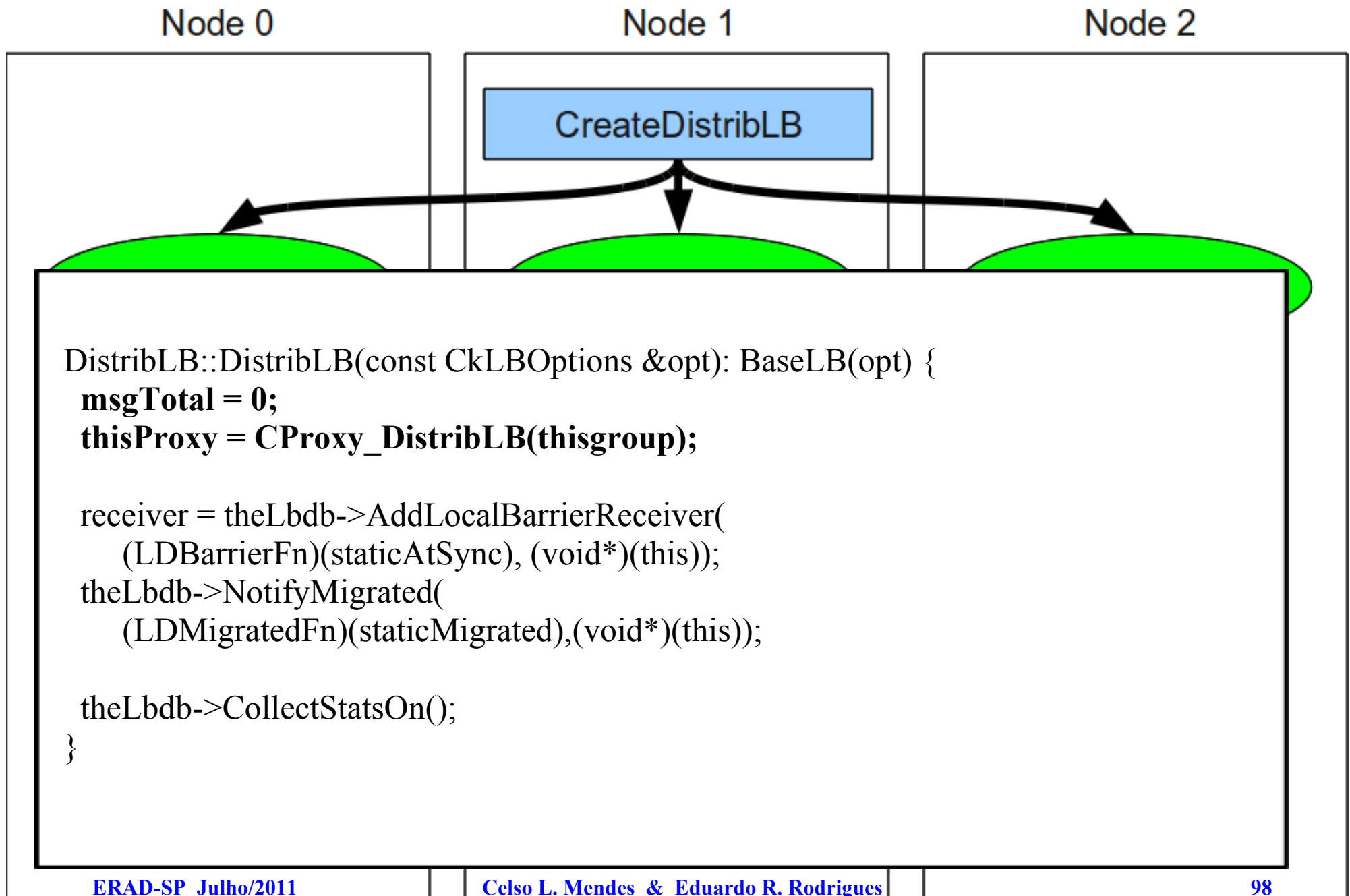
Ciclo de Vida do Balanceador de Carga



O ambiente Charm++ chama o nosso construtor



O que o construtor faz: Iniciar propriedades da classe



Inicializar Tratadores de Evento

Esse tratador
é chamado
toda vez que o
balanceador
de carga é
invocado

Esse tratador
é chamado
quando um
processador
virtual migra

```
DistribLB::DistribLB(const CkLBOptions &opt): BaseLB(opt) {  
    msgTotal = 0;  
    thisProxy = CProxy_DistribLB(thisgroup);  
  
    receiver = theLbdb->AddLocalBarrierReceiver(  
        (LDBarrierFn)(staticAtSync), (void*)(this));  
    theLbdb->NotifyMigrated(  
        (LDMigratedFn)(staticMigrated), (void*)(this));  
  
    theLbdb->CollectStatsOn();  
}
```

Iniciar Instrumentação de Carga e Comunicação

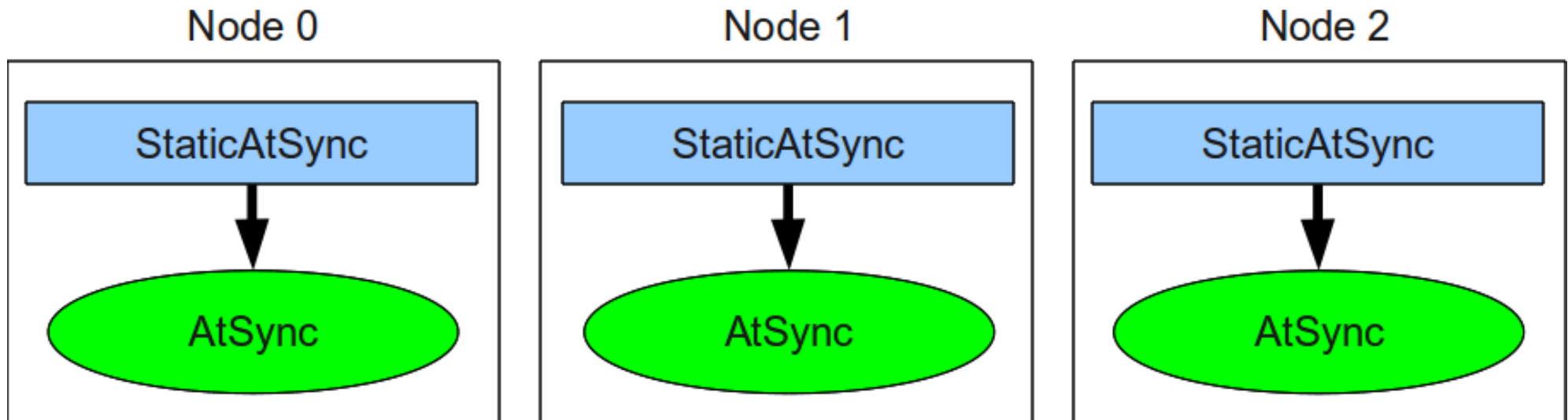
Ativa instrumentação

```
DistribLB::DistribLB(const C
msgTotal = 0;
thisProxy = CProxy_DistribLB(thisgroup);

receiver = theLbdb->AddLocalBarrierReceiver(
    (LDBarrierFn)(staticAtSync), (void*)(this));
theLbdb->NotifyMigrated(
    (LDMigratedFn)(staticMigrated), (void*)(this));

theLbdb->CollectStatsOn();
}
```

O Balanceador de Carga é Invocado



```
void DistribLB::staticAtSync(void* data) {  
    DistribLB *me = (DistribLB*)(data);  
    me->AtSync();  
}
```

```
void DistribLB::AtSync() {  
    double total_walltime, total_cputime;  
    theLbdb->TotalTime(&total_walltime, &total_cputime);  
    thisProxy[(CkMyPe()+1) % CkNumPes()].recvMsgLoad(CkMyPe(), total_walltime);  
    thisProxy[(CkMyPe()+CkNumPes()-1) % CkNumPes()].recvMsgLoad(CkMyPe(), total_walltime);  
    theLbdb->ClearLoads();  
}
```


O Balanceador de Carga é Invocado

Obter
informação de
carga local

Enviar a
informação de
carga para
algum outro
processador

```
void DistribLB::staticAtSync(void* data) {  
    DistribLB *me = (DistribLB*)(data);  
    me->AtSync();  
}
```

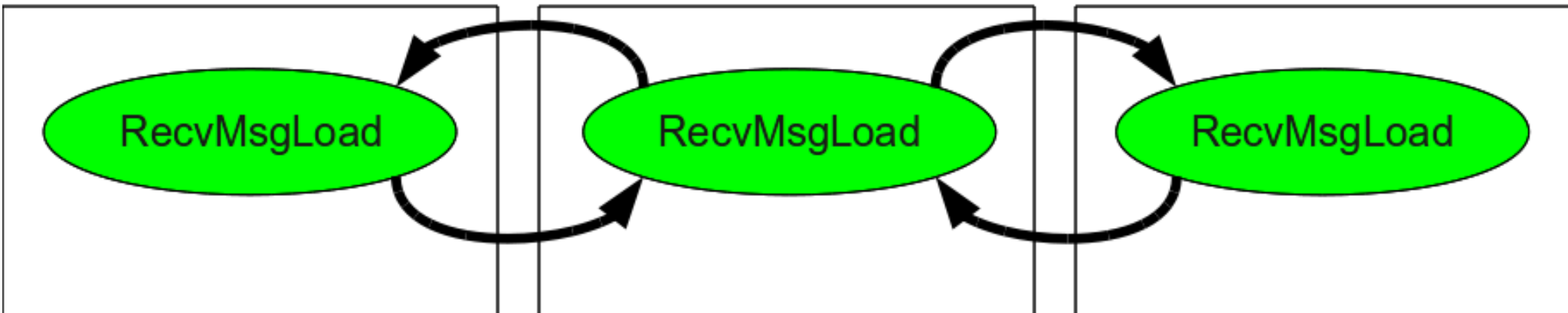
```
void DistribLB::AtSync() {  
    double total_walltime, total_cputime;  
    theLbdb->TotalTime(&total_walltime, &total_cputime);  
    thisProxy[(CkMyPe()+1) % CkNumPes()].recvMsgLoad(CkMyPe(), total_walltime);  
    thisProxy[(CkMyPe()+CkNumPes()-1) % CkNumPes()].recvMsgLoad(CkMyPe(), total_walltime);  
    theLbdb->ClearLoads();  
}
```

Troca Distribuída de Informação de Carga

Node 0

Node 1

Node 2



```
void DistribLB::recvMsgLoad(int pe, double load) {  
    msgTotal++;  
  
    if (msgTotal == 2) {  
        msgTotal = 0;  
        LDObjData *objs = new LDObjData[theLbdb->GetObjDataSz()];  
        theLbdb->GetObjData(objs);  
        theLbdb->Migrate(objs[0].handle , (CkMyPe()+1) % CkNumPes());  
    }  
}
```

Troca Distribuída de Informação de Carga

Node 0

Node 1

Node 2

RecvMsgLoad

Decisão de balanceamento de carga pode ser tomada aqui

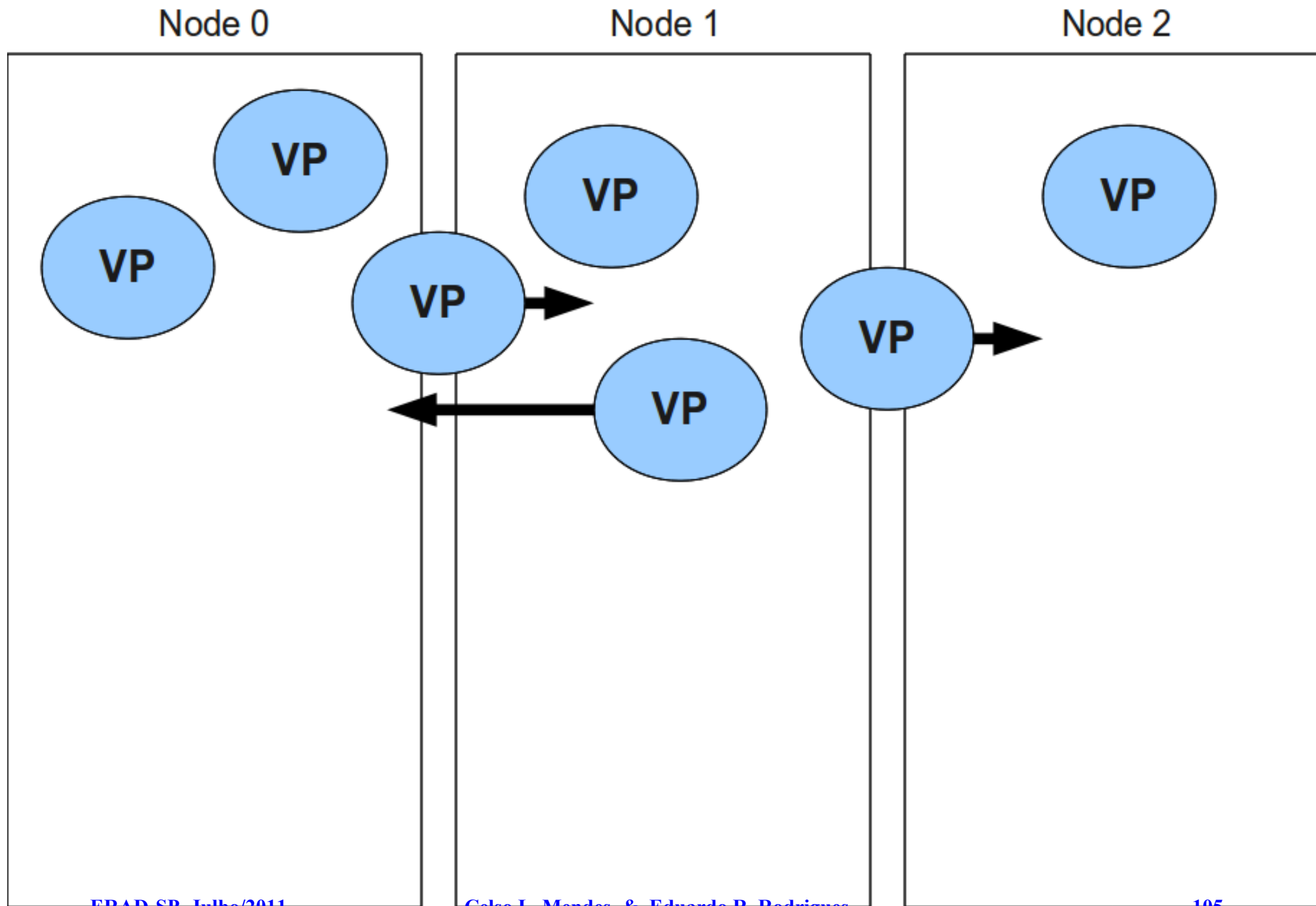
Migrações são ordenadas

MsgLoad

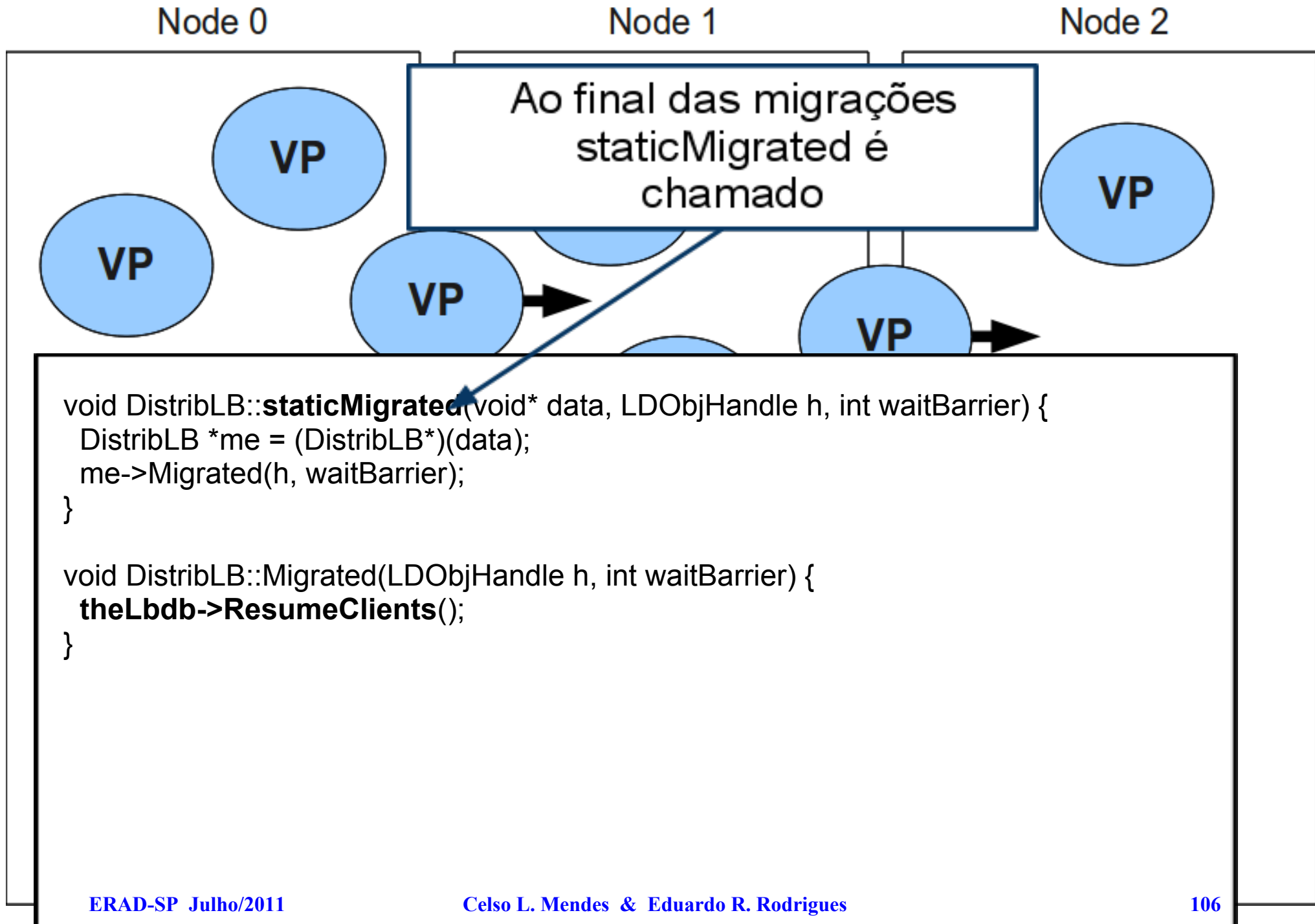
```
void DistribLB::recvMsgLoad()  
{  
    msgTotal++;
```

```
    if (msgTotal == 2) {  
        msgTotal = 0;  
        LDObjData *objs = new LDObjData[theLbdb->GetObjDataSz()];  
        theLbdb->GetObjData(objs);  
        theLbdb->Migrate(objs[0].handle , (CkMyPe()+1) % CkNumPes());  
    }  
}
```

Migrações de Threads Ocorrem



Continuação da Execução



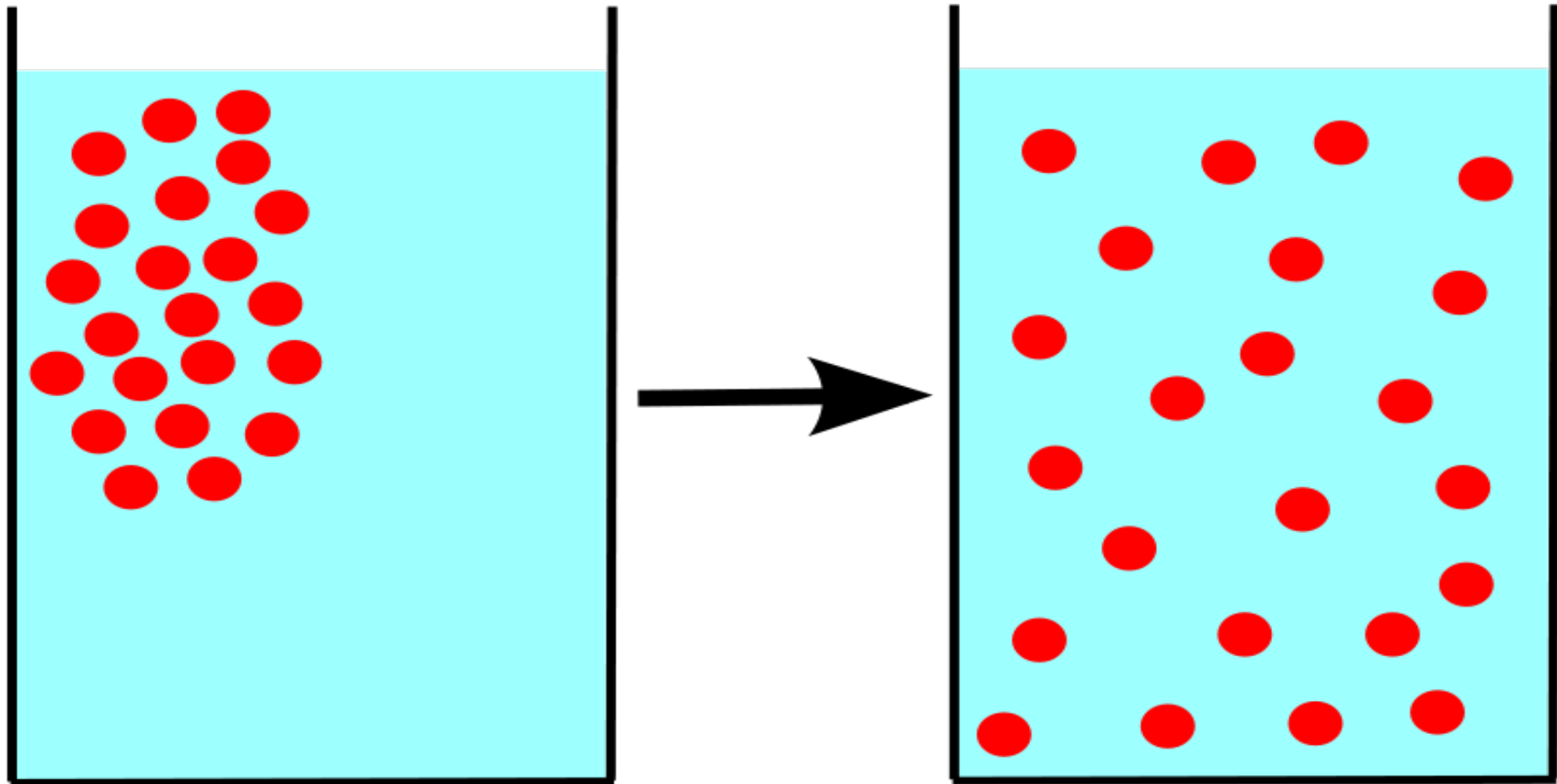
Balanceadores de Carga Distribuídos

- Nesse exemplo vimos a "mecânica" de um balanceador de carga distribuído;
- A comunicação de informação de carga determina escalabilidade;
- Um balanceador de carga "inteligente" armazenaria a informação recebida e tomaria uma decisão baseada nessa informação;
- Um balanceador híbrido pode ser construído usando o mesmo princípio.

Exemplos de Balanceadores de Carga Distribuídos

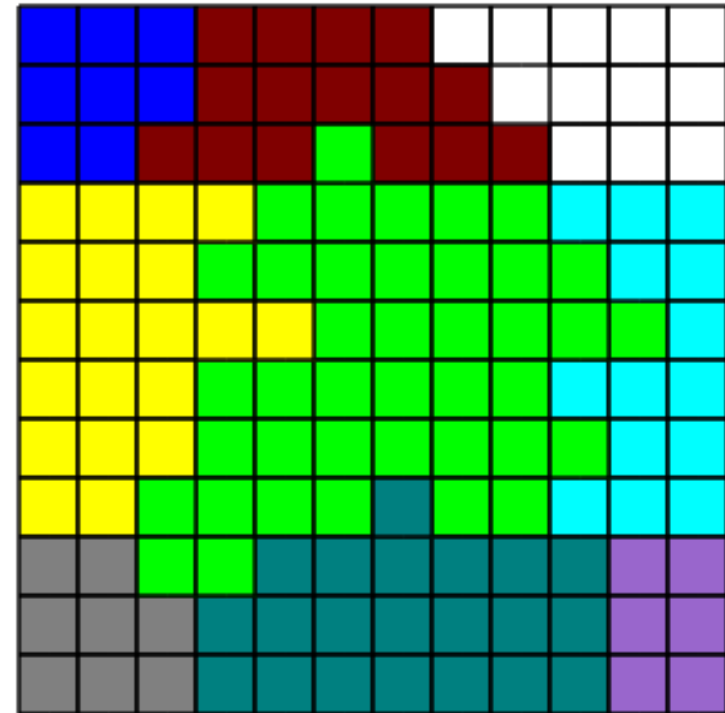
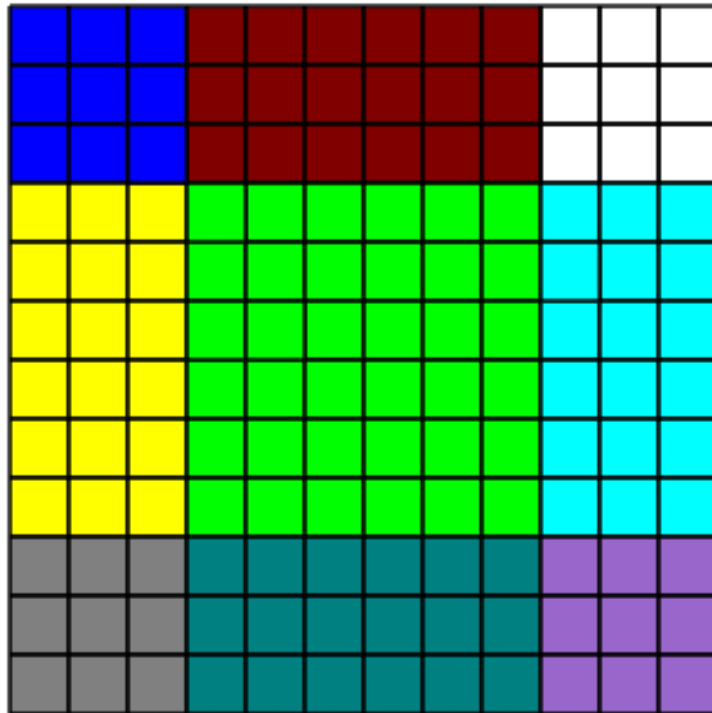
- Balanceador de carga por Difusão
- Balanceador de carga distribuído baseado na curva de Hilbert

Balanceador de Carga por Difusão



- Energia ou matéria flui de regiões de alta concentração para baixa concentração;
- Difusão ocorre de uma sub-região para outra que é adjacente.

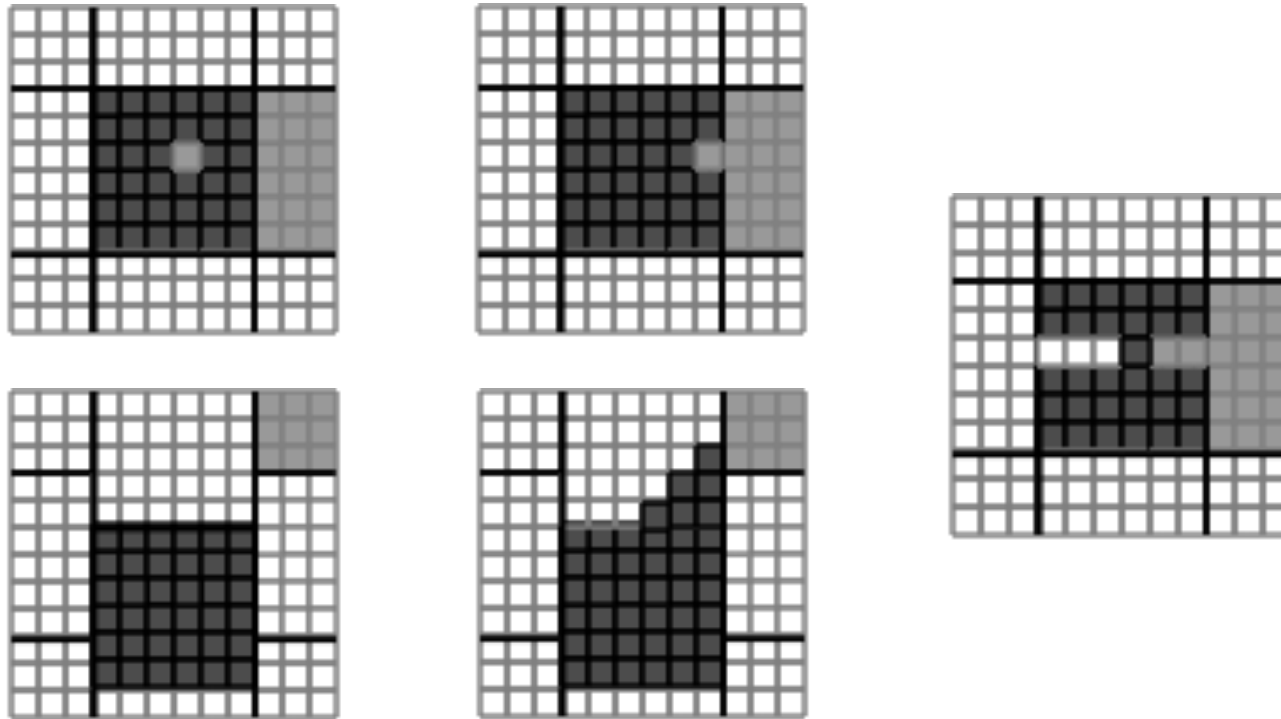
Balanceador de Carga por Difusão



- Comunicação ocorre apenas entre vizinhos;
- Para cada invocação, a carga é balanceada localmente;
- Com o tempo, toda a carga fica balanceada.

Balanceador de Carga por Difusão

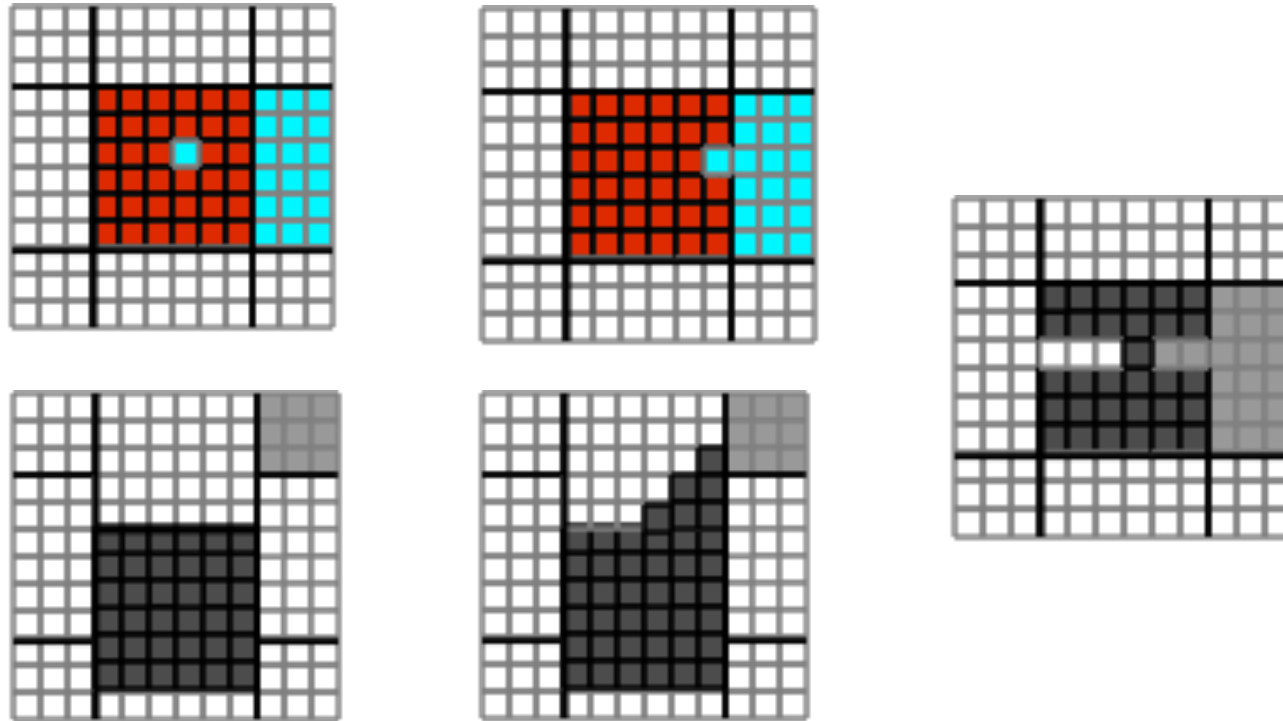
Implementação



- Vejamos o caso em que a decomposição é regular

Balanceador de Carga por Difusão

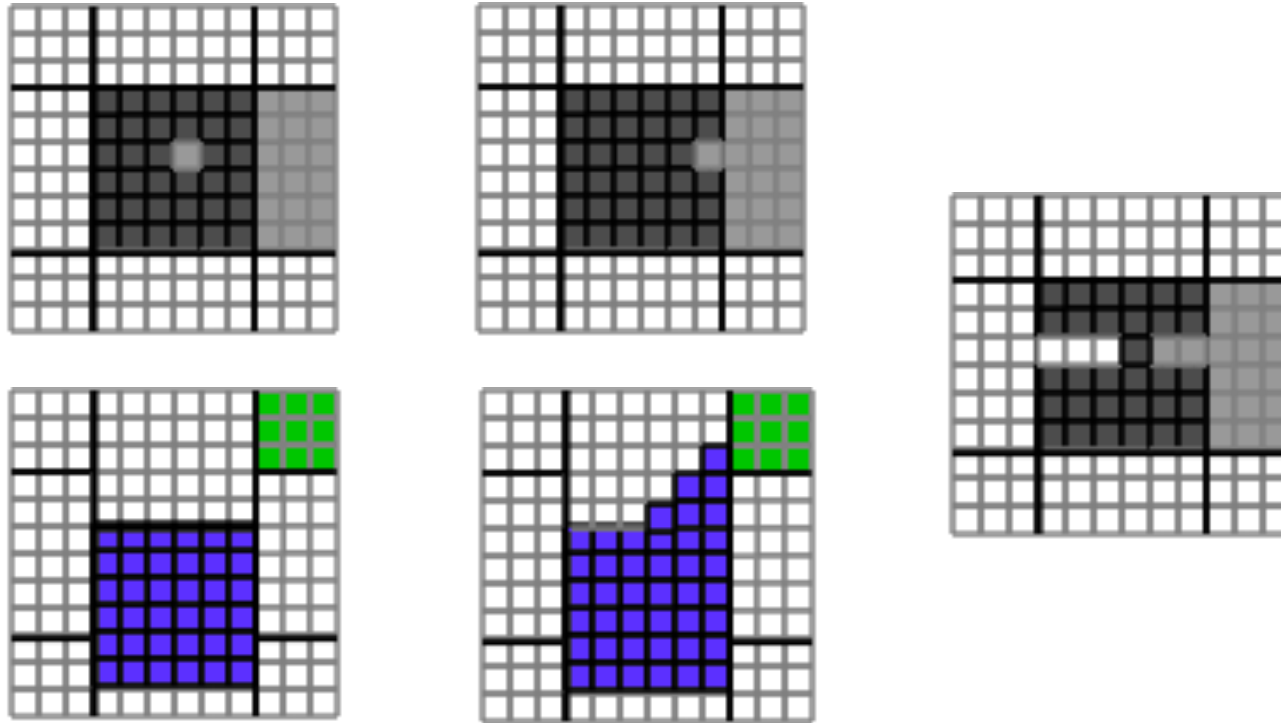
Implementação



- Temos que considerar a **comunicação natural** da aplicação.

Balanceador de Carga por Difusão

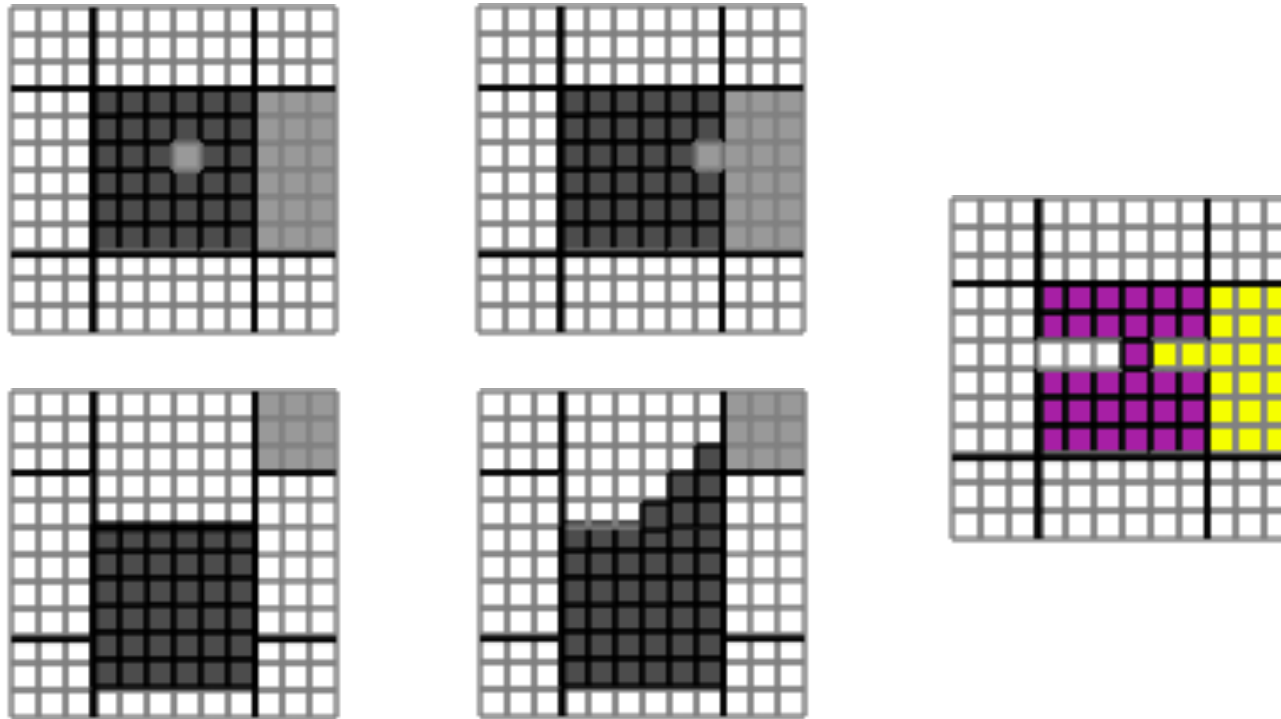
Implementação



- **Novos processadores** podem **surgir/desparecer** da vizinhança.

Balancedador de Carga por Difusão

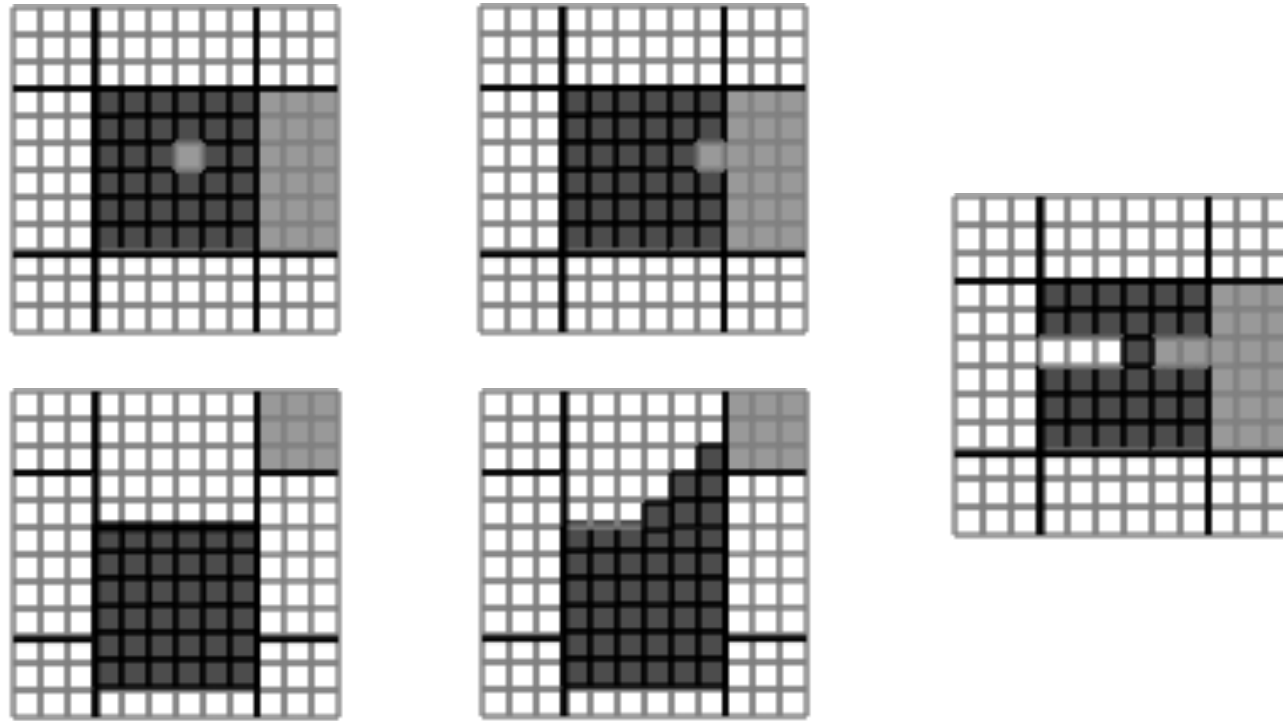
Implementação



- Temos que evitar **fragmentação**.

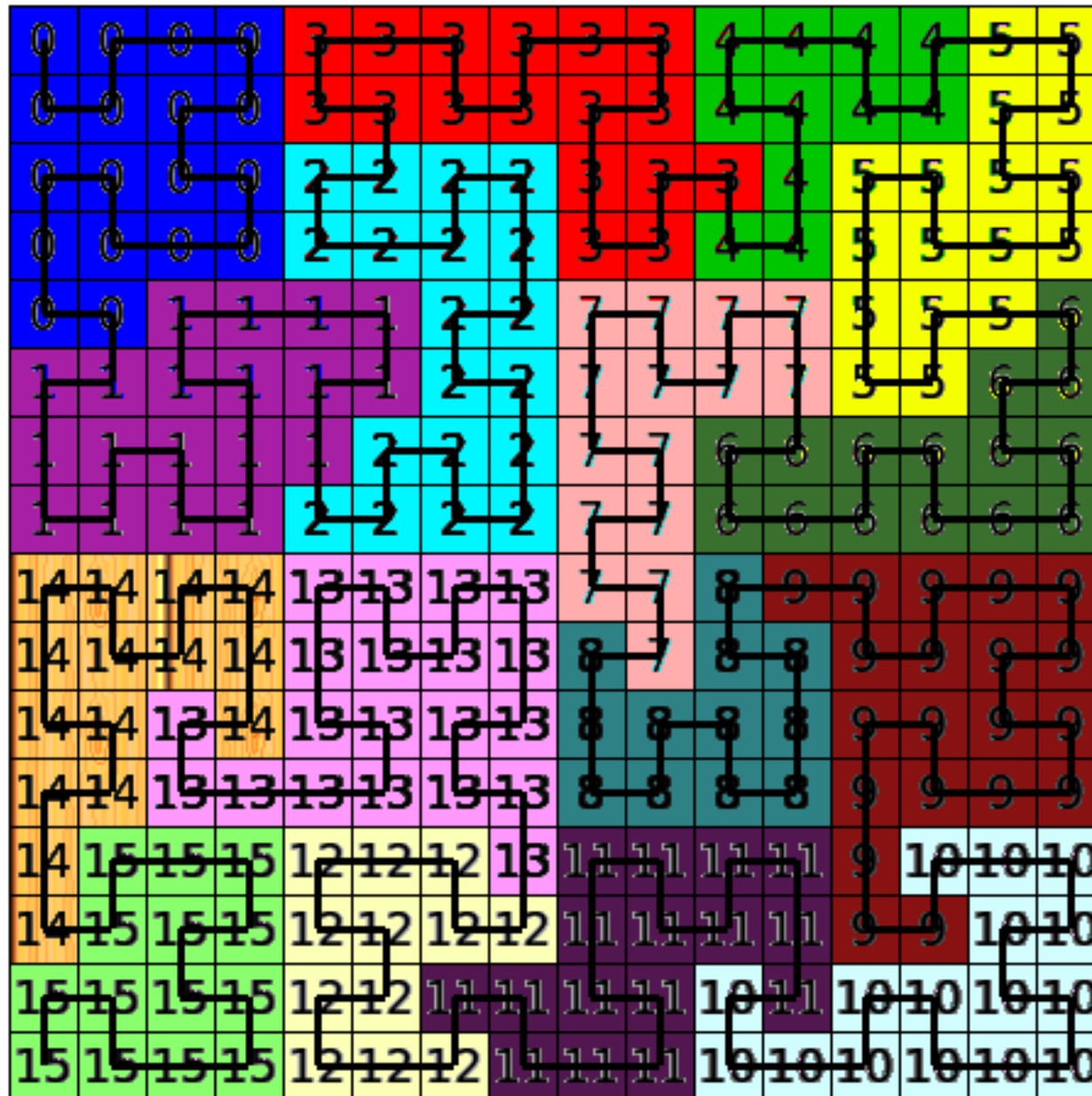
Balanceador de Carga por Difusão

Implementação



- A Conclusão Final é: implementar um balanceador de carga por difusão pode ser uma **tarefa bastante complexa**.
- Mas, ele é **escalável**.

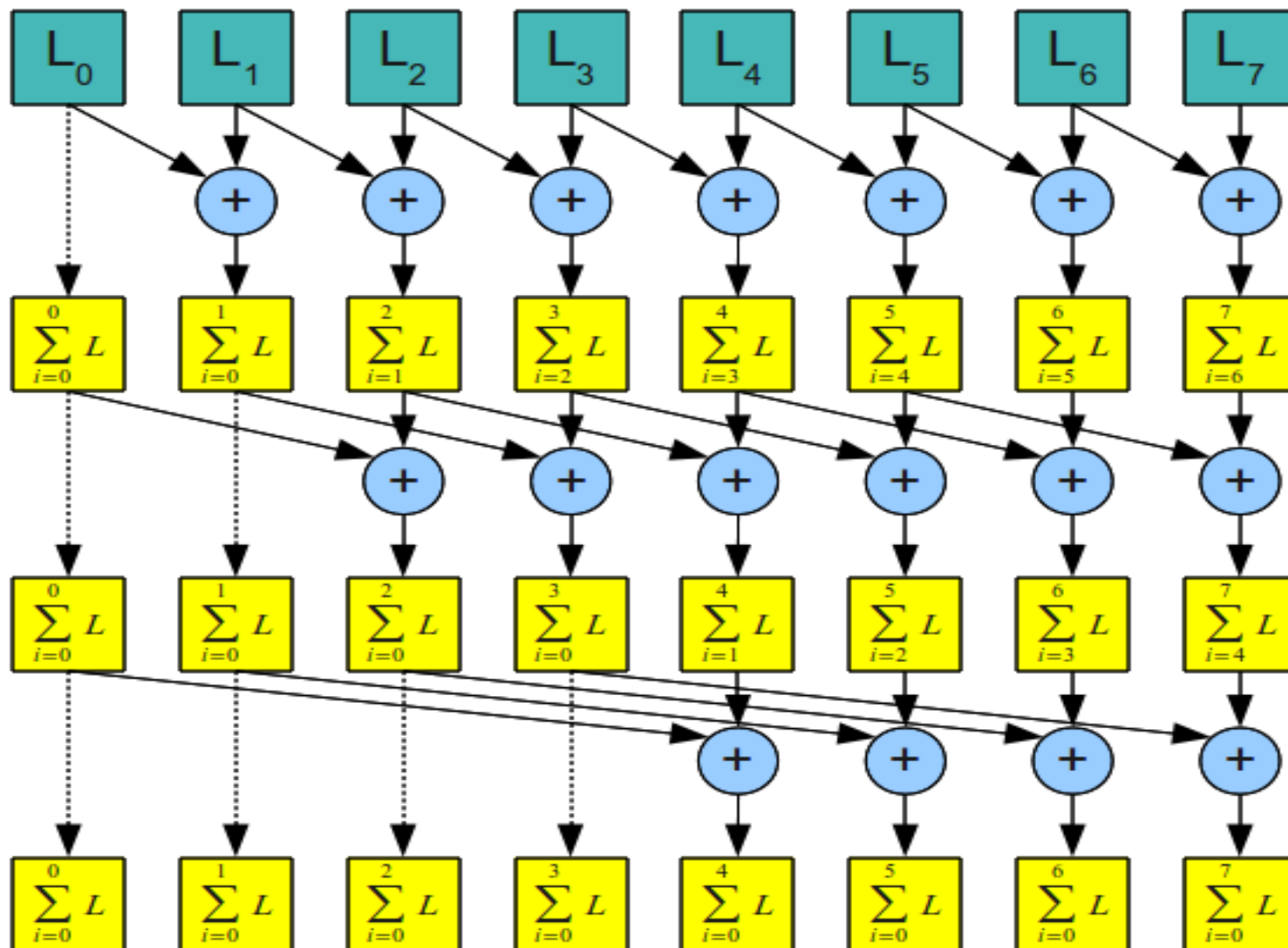
Tornando Distribuído o Balanceador de Carga Baseado na Curva de Hilbert



Balanceador de Carga Distribuído Baseado na Curva de Hilbert

- Computar a soma prefixa das cargas (pode ser feita em paralelo em $\log(N)$ passos);
 - valores 4 3 5 2 4
 - soma 4 7 12 14 18
- *Broadcast* da carga total, feita pelo último processador, que tem esse valor como resultado do passo anterior (também $\log(N)$);
- Algoritmo de corte (sem comunicação).

Soma Prefixa



Algoritmo de Corte

Cada processador executa independentemente

input : **float** myPrefixSum

float myLoad

float totalLoad

int numPEs

output: **int** destPE

idealLoad = totalLoad / numPEs ;

destPE = **floor**(myPrefixSum / idealLoad) ;

destPELeftNeighbor = (myPrefixSum - myLoad) / idealLoad

if destPE = destPELeftNeighbor **then**

if (idealLoad * destPE - myPrefixSum) $^2 \leq$

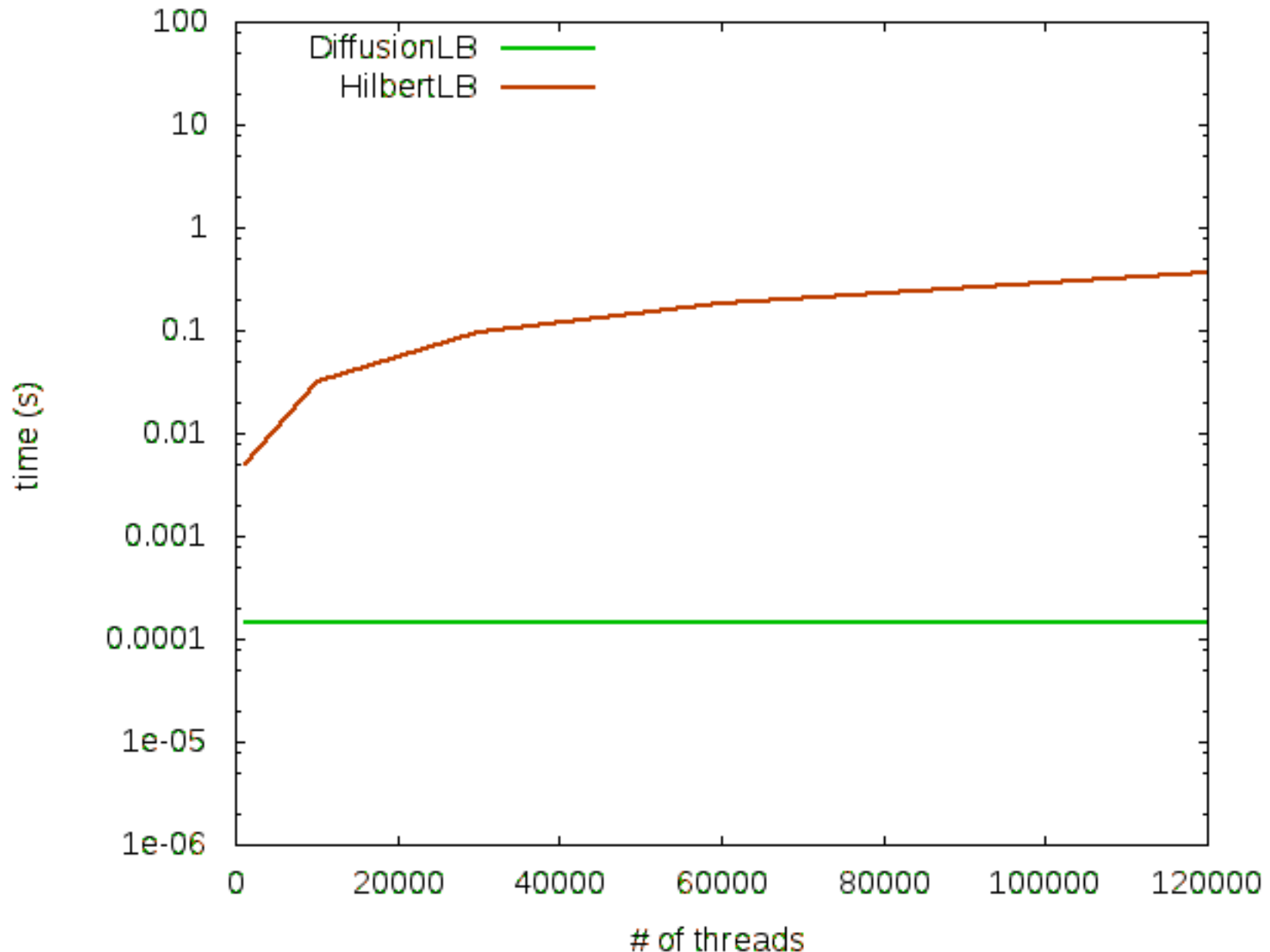
(idealLoad * destPE - (myPrefixSum - myLoad)) 2 **then**

destPE = destPE - 1;

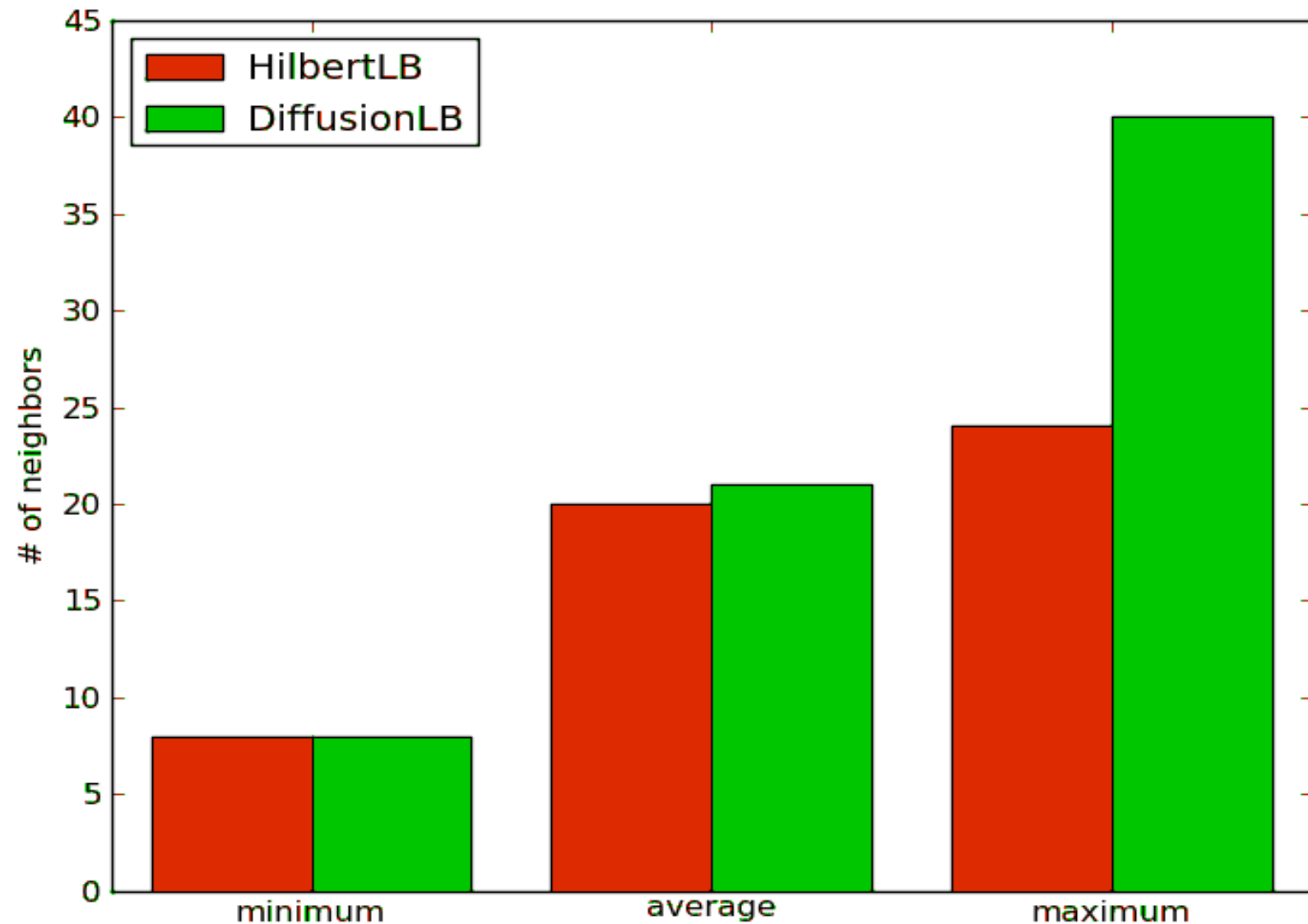
end

end

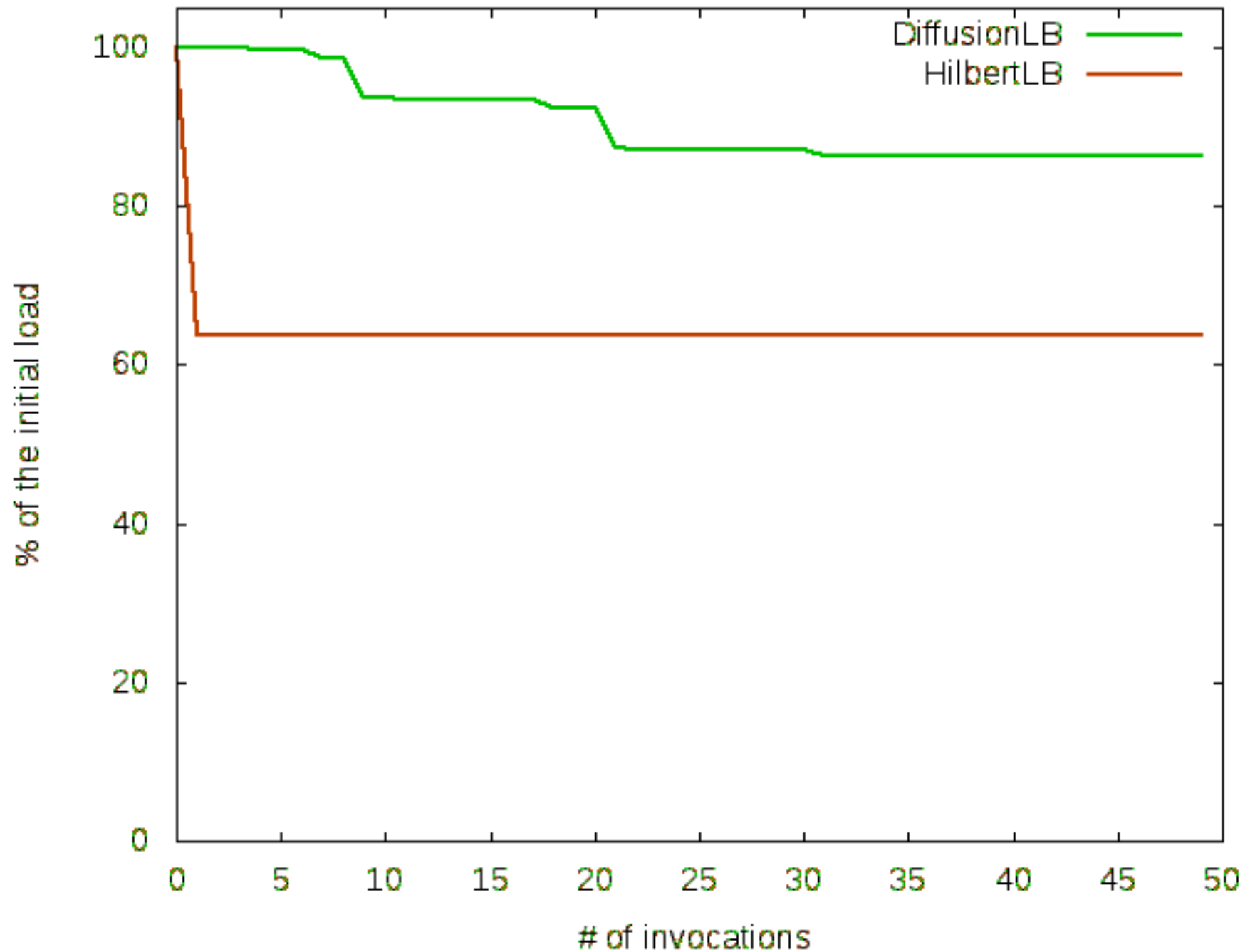
Tempo de Execução do Balanceador Apenas (em função do número de threads)



Número de Vizinhos após 50 Invocações

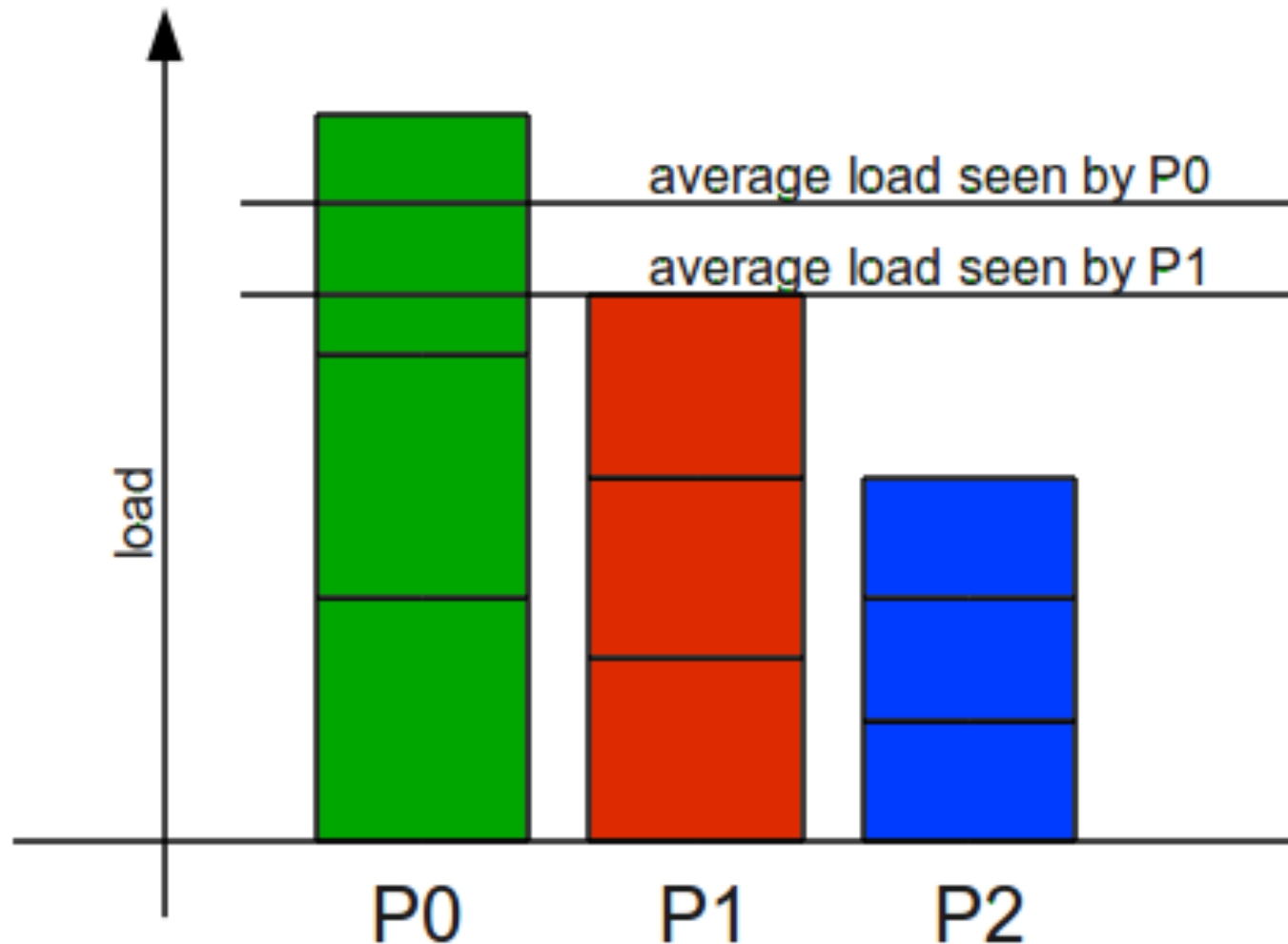


Velocidade de Balanceamento



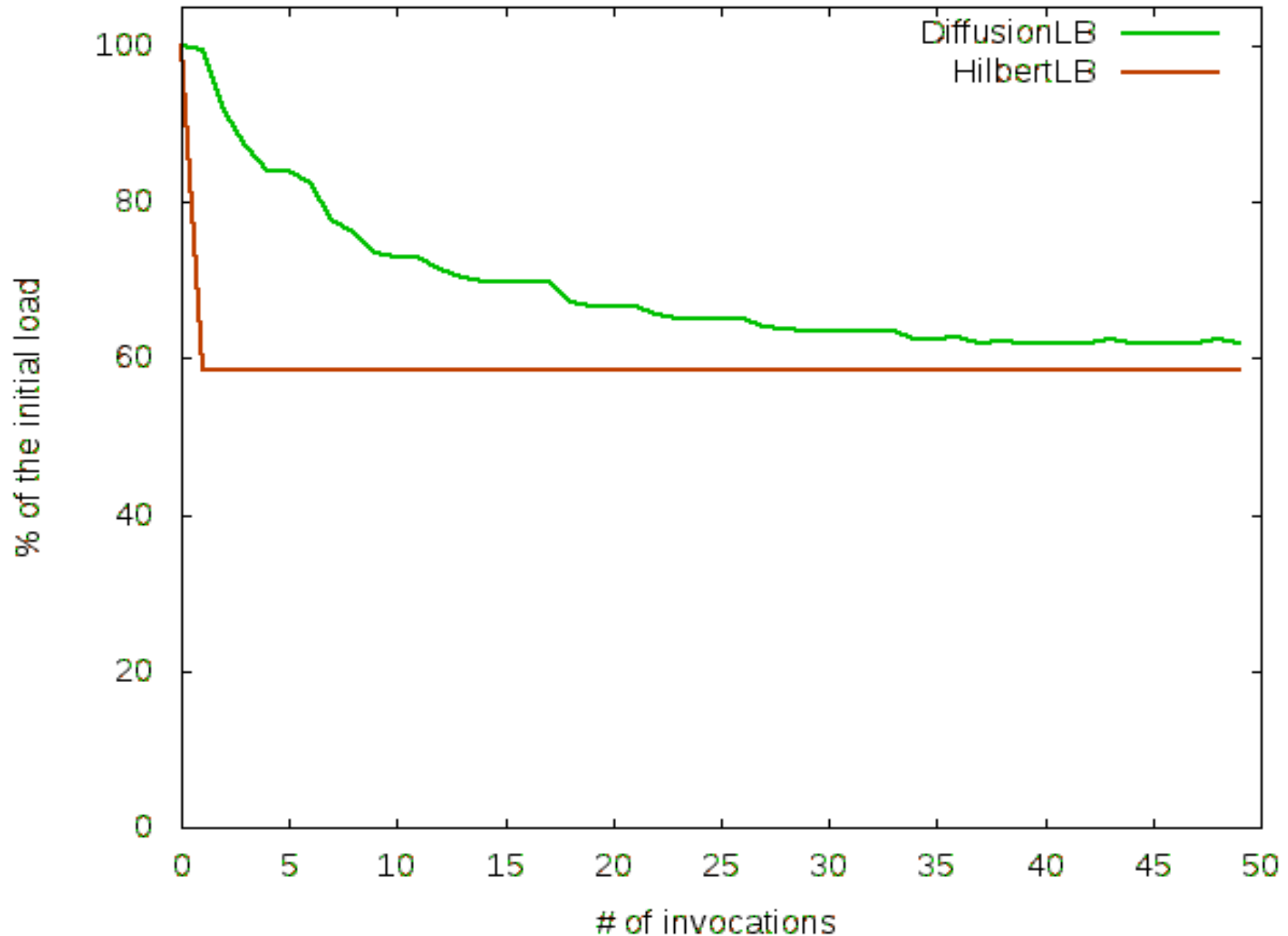
A carga fica "travada" com o balanceador de carga por Difusão

Eis um exemplo simples que ilustra esse fato



Aumento do Grau de Virtualização

Com o aumento do grau de virtualização a carga se torna mais "fluida"



Conclusões

- Balanceadores de carga distribuídos são **escaláveis**
- Entretanto, a **falta de informação global** pode tornar sua ação lenta
- O balanceador de carga distribuído baseado na curva de Hilbert retém alguma informação global (carga total e soma prefixa) a fim de melhorar a qualidade do balanceamento; contudo, o custo de comunicação é $\log(N)$
- Os balanceadores de carga distribuídos serão essenciais para **sistemas Exaflop** (com **milhões** de processadores)

Obrigado!