

**UNIVERSIDADE FEDERAL DE SANTA CATARINA  
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA**

Vinicius Marino Calvo Torres de Freitas

**BALANCEAMENTO DE CARGA DISTRIBUÍDO:  
UMA ABORDAGEM ORIENTADA A PACOTES**

Florianópolis

2017



Vinicius Marino Calvo Torres de Freitas

**BALANCEAMENTO DE CARGA DISTRIBUÍDO: UMA  
ABORDAGEM ORIENTADA A PACOTES**

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.

Orientador: Prof. Dr. Laércio Lima Pilla

Florianópolis

2017

Ficha de identificação da obra elaborada pelo autor,  
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Freitas, Vinicius Marino Calvo Torres de  
Balanceamento de carga distribuído : uma  
abordagem orientada a pacotes / Vinicius Marino  
Calvo Torres de Freitas ; orientador, Laércio Lima  
Pilla, 2017.  
94 p.

Trabalho de Conclusão de Curso (graduação) -  
Universidade Federal de Santa Catarina, Centro  
Tecnológico, Graduação em Ciências da Computação,  
Florianópolis, 2017.

Inclui referências.

1. Ciências da Computação. 2. Balanceamento de  
carga. 3. Análise de desempenho. 4. Computação de  
alto desempenho. 5. Sistemas paralelos. I. Pilla,  
Laércio Lima. II. Universidade Federal de Santa  
Catarina. Graduação em Ciências da Computação. III.  
Título.

Vinicius Marino Calvo Torres de Freitas

**BALANCEAMENTO DE CARGA DISTRIBUÍDO: UMA  
ABORDAGEM ORIENTADA A PACOTES**

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de “Bacharel em Ciências da Computação”, e aprovado em sua forma final pela Curso de Bacharelado em Ciências da Computação.

Florianópolis, 13 de Dezembro 2017.

---

Prof. Dr. Renato Cislighi  
Coordenador do Curso

**Banca Examinadora:**

---

Prof. Dr. Laércio Lima Pilla  
Orientador

---

Prof. Dr. Márcio Bastos Castro

---

Prof. Dr. Mário A. R. Dantas



Dedico este trabalho aos meus pais, que me deram educação e companhia. Dedico também a Amanda Almeida, cuja compreensão e carinho me levaram para frente durante o desenvolvimento deste trabalho.





## AGRADECIMENTOS

Aos meus pais, que desde minha infância incentivaram minha educação, me estimularam a sempre buscar conhecimento e mostraram-me a vida acadêmica do ponto de vista de dois professores universitários.

Ao meu orientador, Laércio, por me apresentar esta área de pesquisa que hoje sigo e por todas as revisões atenciosas deste e de outros trabalhos.

À Universidade Federal de Santa Catarina, seu corpo docente e administrativo, pela oportunidade de aprender e crescer como aluno de nível superior nos últimos anos.

Ao meu irmão, que vi crescer nos últimos anos e com isso me incentivou a dedicar-me cada vez mais nesse trabalho.

Aos meus amigos e colegas de curso, dos quais sem o apoio eu não conseguiria manter-me na Universidade.

A todos os meus professores, mestres sem os quais eu não seria capaz de absorver tudo que aprendi neste curso.

Aos meus colegas do *Embedded Computing Laboratory (ECL)*, na UFSC, que alimentaram minha paixão por pesquisa desde que me juntei ao grupo.

À minha namorada, Amanda, cuja companhia e incentivo me permitiram dar o meu melhor para finalizar este trabalho.

E por último, mas não menos importante, às pessoas que estiveram mais próximas de mim na reta final do desenvolvimento deste Trabalho de Conclusão; incentivando-me até nos momentos mais difíceis.



*Remember that night  
White steps in the moonlight  
They walked here too  
Through empty playground, this ghosts'  
town  
Children again, on rusting swings getting  
higher  
Sharing a dream, on an island, it felt right*

*We lay side by side  
Between the moon and the tide  
Mapping the stars for awhile*

*Let the night surround you  
We're halfway to the stars  
Ebb and flow  
Let it go  
Feel her warmth beside you*

*Remember that night  
The warmth and the laughter  
Candles burned  
Though the church was deserted  
At dawn we went down through empty streets  
to the harbour  
Dreamers may leave, but they're here ever  
after*

David Gilmour & Polly Samson



## RESUMO

O desbalanceamento de carga é um problema decorrente do crescimento de sistemas paralelos em busca de desempenho, devido ao grau de imprevisibilidade de simulações computacionais. O reescalonamento global de tarefas é uma das possíveis soluções para mitigar esse problema. Através de estratégias centralizadas e hierárquicas, aplicações em plataformas paralelas ganham em desempenho. Contudo, com o crescimento dessas plataformas, novas abordagens de escalonamento devem ser tomadas. Neste trabalho, proponho uma nova estratégia de balanceamento de carga distribuída, que busca se aproveitar do paralelismo inerente das maiores plataformas da atualidade, reduzindo custos de comunicação através da abordagem orientada a pacotes. Resultados mostram ganhos com *speedups* observados de até 1,33 e 1,05 quando comparado a execuções sem balanceamento e com o estado da arte, respectivamente. São esperados resultados ainda mais promissores em escalas maiores, não acessíveis durante a produção deste trabalho.

**Palavras-chave:** Balanceamento de carga. Análise de desempenho. Computação de alto desempenho. Sistemas paralelos



## ABSTRACT

Load imbalance is a problem caused by the growth of parallel systems seeking performance, due to the unpredictability of computational simulations. Global task rescheduling is one of the possible solutions to mitigate this problem. Through centralized and hierarchical strategies, applications in parallel systems enhance their performance. Although, as this platforms grow, new scheduling approaches must be taken. In this work, I propose a new distributed load balancing strategy, which seeks to take advantage of the parallelism in modern large-scale platforms, reducing communication costs through the package oriented approach. Results show speedups of up to 1.33 and 1.05 when compared to executions with no rescheduling and to the state of the art, respectively. Even better results are expected in larger scales of test, not accessible during the production of this dissertation.

**Keywords:** Load balancing. Performance evaluation. High performance computing. Parallel systems.





## LISTA DE FIGURAS

Figura 1	Variação da carga em sistemas paralelos (PILLA, 2014).	32
Figura 2	Variação da comunicação em sistemas paralelos (PILLA, 2014).	33
Figura 3	Comunicação entre <i>chares</i> / divisão dos <i>chares</i> entre diferentes CPU's (CHARM++, acessado em 29 de setembro de 2017).	34
Figura 4	Visão do sistema de uma aplicação Charm++.	35
Figura 5	Representação visual geral de balanceadores de carga centralizados (à esquerda) e hierárquicos (à direita).	37
Figura 6	Fluxo de funcionamento do <i>PackDropLB</i> , seguido individualmente por cada PE.	42
Figura 7	Um sistema desbalanceado, com núcleos divididos em grupos (Balanceado, Subcarregado e Sobrecarregado) por limites relacionados à carga média.	43
Figura 8	Três passos de execução do Protocolo <i>Gossip</i> , com <i>throughput</i> de 2. À esquerda o primeiro passo, no meio, o segundo e à direita, o terceiro.	45
Figura 9	Tempos de execução do <i>benchmark lb test</i> para 18990 tarefas (em cima) e 36990 tarefas (embaixo) com diferentes balanceadores de carga.	53
Figura 10	Tempos de execução do <i>benchmark Stencil 3D</i> para dimensões de bloco 540 e de <i>array</i> 12 com diferentes balanceadores de carga.	54
Figura 11	Tempos de execução do <i>LULESH</i> para dimensões de bloco 140 e de <i>array</i> 20 com diferentes balanceadores de carga.	55
Figura 12	Tempos de execução do <i>LeanMD</i> para dimensões $10 \times 15 \times 10$ com diferentes balanceadores de carga.	55
Figura 13	Tempos de execução do <i>LeanMD</i> para dimensões $15 \times 20 \times 15$ com diferentes balanceadores de carga.	56
Figura 14	Speedups ( $\mathcal{S} - 1$ ) de todos os <i>benchmarks</i> e aplicações testadas com cada estratégia de balanceamento de carga.	57



## LISTA DE TABELAS

Tabela 1	Número de execuções dos experimentos para averiguação dos resultados com <i>benchmarks</i> menores ( <b>M</b> ) e maiores ( <b>G</b> ). . . . .	51
Tabela 2	Tempo médio de execução obtidos do <i>benchmark lb test</i> para as diferentes estratégias. . . . .	52



## LISTA DE ABREVIATURAS E SIGLAS

PE	<i>Processing element</i> , unidade de processamento . . . . .	27
LB	<i>Load balancer</i> - balanceador de carga . . . . .	28
MPI	<i>Message Passing Interface</i> . . . . .	34
WS	<i>Workstealing</i> , roubo de tarefas . . . . .	34
CPU	<i>Central Processing Unit</i> . . . . .	34
PUP	<i>Pack/Unpack, framework</i> de serialização de objetos . . . .	35
LULESH	Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics . . . . .	50
LeanMD	<i>Lennard-Jones Molecular Dynamics</i> . . . . .	50
HDD	<i>Hard Disk Drive</i> - Disco Rígido . . . . .	51
SATA	<i>Serial Advanced Technology Attachment</i> . . . . .	51



## LISTA DE SÍMBOLOS

$C$	Número de unidades de processamento .....	31
$s$	Porção serial de programa .....	31
$S$	<i>Speedup</i> .....	31
$T_0$	Tempo de execução original de uma aplicação .....	32
$T_1$	Tempo de execução sobre o qual deseja-se observar o <i>speedup</i> .....	32
$\mathcal{M}$	Mapeamento de tarefas .....	32
$\mathcal{T}$	Conjunto de tarefas .....	32
$\mathcal{P}$	Conjunto de unidades de processamento .....	32
$T$	Número de tarefas no sistema .....	38
$ps$	<i>Pack Size</i> , o tamanho de cada pacote .....	42
$ts$	<i>Task Size</i> , a carga de uma tarefa .....	42
$nC$	Número de núcleos no sistema .....	42
$nT$	Número de tarefas no sistema .....	42
$t$	<i>Throughput</i> do Protocolo <i>Gossip</i> .....	44
$NT$	Conjunto de tarefas que devem tentar ser transmitidas no- vamente .....	46
$lt$	Limite de tentativas de migração para um pacote .....	46





## LISTA DE ALGORITMOS

1	Criação de pacotes .....	44
2	Envio de pacotes .....	45
3	Remapeamento das tarefas .....	46



## SUMÁRIO

<b>1 INTRODUÇÃO</b> .....	27
1.1 OBJETIVOS .....	28
1.2 MÉTODO DE PESQUISA .....	28
1.3 ORGANIZAÇÃO DO TRABALHO .....	29
<b>2 CONCEITOS IMPORTANTES</b> .....	31
2.1 O ARGUMENTO DE AMDAHL E DESEMPENHO DO PARALELISMO .....	31
2.2 BALANCEAMENTO DE CARGA INDEPENDENTE DE APLICAÇÃO .....	32
<b>2.2.1 Problemas atrelados ao balanceamento de carga</b> ...	33
2.3 AMBIENTE PARALELO: CHARM++ .....	34
<b>2.3.1 Objetos migráveis e troca de mensagens assíncronas</b>	34
<b>2.3.2 Framework de balanceamento de carga</b> .....	35
2.4 BALANCEAMENTO DE CARGA CENTRALIZADO E HI- ERÁRQUICO .....	36
<b>2.4.1 GreedyLB</b> .....	36
<b>2.4.2 RefineLB</b> .....	37
<b>2.4.3 HybridLB</b> .....	37
2.5 BALANCEAMENTO DE CARGA DISTRIBUÍDO .....	38
<b>2.5.1 GrapevineLB (MENON; KALÉ, 2013)</b> .....	38
2.6 CONCLUSÃO .....	39
<b>3 BALANCEAMENTO DE CARGA DISTRIBUÍDO ORI- ENTADO A PACOTES: PACKDROPLB</b> .....	41
3.1 ALGORITMO PROPOSTO: <i>PACKDROPLB</i> .....	41
<b>3.1.1 Criação de pacotes</b> .....	42
<b>3.1.2 Protocolo Gossip (DEMERS et al., 1987)</b> .....	44
<b>3.1.3 Envio de pacotes</b> .....	44
3.2 CONCLUSÃO .....	46
<b>4 AVALIAÇÃO DE DESEMPENHO</b> .....	49
4.1 <i>BENCHMARKS</i> E APLICAÇÕES .....	49
<b>4.1.1 lb test</b> .....	49
<b>4.1.2 Stencil 3D</b> .....	50
<b>4.1.3 LULESH</b> .....	50
<b>4.1.4 LeanMD</b> .....	50
4.2 AMBIENTES DE TESTES .....	51
4.3 RESULTADOS OBTIDOS .....	52
<b>4.3.1 lb test</b> .....	52

<b>4.3.2 Stencil 3D</b> .....	53
<b>4.3.3 LULESH</b> .....	54
<b>4.3.4 LeanMD</b> .....	55
<b>4.4 CONCLUSÃO</b> .....	56
<b>5 CONCLUSÃO</b> .....	59
5.1 TRABALHOS FUTUROS .....	59
<b>REFERÊNCIAS</b> .....	61
<b>APÊNDICE A - Códigos do <i>PackDropLB</i></b> .....	67
<b>APÊNDICE B - Artigo baseado no trabalho de conclusão</b>	83

## 1 INTRODUÇÃO

Com o avanço das aplicações científicas e industriais, estas necessitam de cada vez mais tempo de processamento e poder computacional; isso acontece por muitas delas serem simulações muito detalhadas, precisas e/ou complexas, como simulações de ondas sísmicas (DUPROS et al., 2010; BOITO et al., 2017), cósmicas (JETLEY et al., 2010; GIOACHIN et al., 2007), previsão do tempo (OSTHOFF et al., 2012) e de dinâmica molecular (BHATELE et al., 2009; MEI et al., 2011). Para prover tais recursos às aplicações, são usados sistemas computacionais paralelos, capazes de prover uma redução do tempo de execução com aumento do número de unidades de processamento (PE's), para que estas terminem em tempo hábil. Por exemplo, de nada adianta realizar uma previsão do tempo para daqui a três dias que leva uma semana para ficar pronta, é necessário acelerar esse tipo de aplicação.

O principal objetivo da computação paralela é fazer com que a solução de problemas computacionais se torne mais fácil e rápida (KUCK, 2011). Contudo, nem sempre a solução mais veloz é também a mais fácil. Ainda assim, a maioria dos programas possuem uma parcela inerentemente serial. Portanto, para fazer uma predição de seu desempenho paralelo, isso deve ser levado em conta (GUSTAFSON, 2011), o que será mais detalhado na Seção 2.1.

A solução para os trabalhos que necessitam de muito poder computacional é usar mais nós de processamento. Aplicações de memória distribuída usualmente são executadas em diversas máquinas comunicando-se em rede. Isso torna a infraestrutura mais barata, em contra partida tornando a sua programação mais complexa (FLYNN, 1972; TANENBAUM; GOODMAN, 1998). Esse custo elevado no desenvolvimento das aplicações se paga evitando complicações no desenvolvimento das comunicações de memória compartilhada e permitindo maior escalabilidade com adição de mais máquinas, sem grandes mudanças no sistema.

Aplicações de memória distribuída sofrem de uma série de problemas, como maior complexidade de desenvolvimento, comunicação como gargalo, dependência e distribuição de dados, sincronização, áreas críticas e desbalanceamento de carga (PILLA, 2014; NAVAU; PILLA, 2011).

A distribuição de trabalho através de recursos computacionais é um grande problema para a computação paralela (PEARCE et al., 2012; LIU et al., 2007). Diversas aplicações possuem comportamento previsível

e suas cargas (tempo de execução das tarefas) conhecidas, sendo assim possível estimar um bom mapeamento de tarefas para recursos (mais detalhado na Seção 2.2). Contudo, outras aplicações tendem a ter mudanças dinâmicas em seu comportamento, fazendo com que mesmo um mapeamento inicial aparentemente ótimo chegue a um estado desbalanceado.

Uma possível solução para esse tipo de problema é o uso de balanceadores de carga dinâmicos. Esses balanceadores devem usar um estado do sistema como base para fazer um remapeamento das tarefas entre as unidades de processamento, buscando um estado mais balanceado.

Ao se desenvolver um balanceador de carga dinâmico (LB) é necessário entender que seu tempo de execução deve ser curto o suficiente para que ele não torne a aplicação mais lenta, ao mesmo tempo que fornece um remapeamento mais equilibrado das tarefas (DEVECI et al., 2015).

## 1.1 OBJETIVOS

Propor uma nova estratégia de balanceamento de carga distribuído, aproveitando do paralelismo disponível nas grandes plataformas da atualidade. Para tal, foram planejados os seguintes objetivos específicos:

- Implementar um novo algoritmo de balanceamento de carga distribuído.
- Realizar análise de desempenho da estratégia, comparando-a com o estado da arte.
- Identificar os benefícios da estratégia proposta.

## 1.2 MÉTODO DE PESQUISA

Balanceamento de carga distribuído é um assunto ainda pouco explorado quando comparado ao balanceamento de carga centralizado. Estudos iniciais já mostram resultados promissores para larga escala (ME-NON; KALÉ, 2013), então estender ainda mais esse campo de pesquisa, tentando novas estratégias de comunicação e contabilizando diferentes informações para realizar o balanceamento de carga, são formas de buscar entender melhor essa área em expansão.

Este trabalho apresenta uma nova estratégia de balanceamento de carga distribuída, que foi implementada e testada no sistema paralelo **Charm++** (KALÉ; KRISHNAN, 1993), o qual provê um *framework* para o desenvolvimento de algoritmos de balanceamento de carga, permitindo que ele seja feito independente de aplicação. Essa estratégia se aproveita tanto dos núcleos sobrecarregados (que buscam enviar suas tarefas para outros núcleos o mais rápido possível) quanto dos núcleos subcarregados (que buscam enviar informações sobre sua situação) simultaneamente, buscando entender a viabilidade de informações diferentes sendo transmitidas e aplicadas, bem como tentando agilizar o processo de balanceamento.

### 1.3 ORGANIZAÇÃO DO TRABALHO

O restante do trabalho será organizado em quatro capítulos. O Capítulo 2 é uma revisão bibliográfica mostrando os principais temas do estado da arte estudados para o desenvolvimento deste. O Capítulo 3, explica o que foi desenvolvido, dificuldades encontradas e caminhos de implementação não aproveitados, bem como a descrição final do algoritmo. O Capítulo 4 descreve o processo de avaliação de desempenho, as técnicas, plataformas computacionais, aplicações e *benchmarks* utilizados, bem como a avaliação de desempenho em si, com os resultados obtidos e conclusões que se pode tirar deles. Por último, o Capítulo 5 descreve as conclusões obtidas a partir do que se estudou, além de traçar possíveis rumos futuros da pesquisa e possíveis campos de aplicação para os resultados.





## 2 CONCEITOS IMPORTANTES

Este capítulo é focado em explicar e detalhar alguns conceitos importantes para melhor compreensão das propostas deste trabalho. São apresentados conceitos relacionados a paralelismo, balanceamento de carga (centralizado, hierárquico e distribuído) e a principal linguagem de programação utilizada nesta pesquisa, o **Charm++**.

### 2.1 O ARGUMENTO DE AMDAHL E DESEMPENHO DO PARALELISMO

O Argumento de Amdahl (Equação 2.1) é uma fórmula usada para prever o desempenho da paralelização de uma aplicação. Ela leva em conta a porção da aplicação que é inerentemente serial ( $s$ ) e a porção da aplicação que pode ser paralelizada ( $1 - s$ ), assim como o número de unidades de processamento disponíveis para acelerá-la ( $C$ ). O Argumento de Amdahl não considera o sobrecusto de criação de *threads* ou de comunicação, por isso é apenas uma estimativa de quanto uma implementação paralela pode ser mais rápida que uma serial ( $S$ ) (GUSTAFSON, 2011).

$$S = \frac{1}{s + \frac{1-s}{C}} \quad (2.1)$$

Mesmo depois de contabilizado o sobrecusto de comunicação inicial e da criação de *threads*, ainda há outros fatores que afetam o tempo total de algumas aplicações. Como mencionado anteriormente, algumas simulações têm um comportamento imprevisível e, entre suas iterações (ou depois de algum tempo em sua execução), acabam por gerar muita demanda em alguns PE's e pouca em outros, ocasionando desbalanceamento de carga, como mostrado na Figura 1.

Esse mesmo problema pode ocorrer com a comunicação. Mesmo que o escalonamento original posicione as tarefas que se comunicam de forma ótima, em simulações seus padrões de comunicação podem mudar através do tempo, criando uma demanda por mais comunicação inter-nó, um sobrecusto que também prejudica o desempenho, como mostrado na Figura 2.

Após contabilizados todos os fatores e observados os tempos de execução reais obtidos após otimizações e paralelismo, tem-se o *Speedup* real de uma aplicação, que pode ser descrito pela Equação 2.2, onde

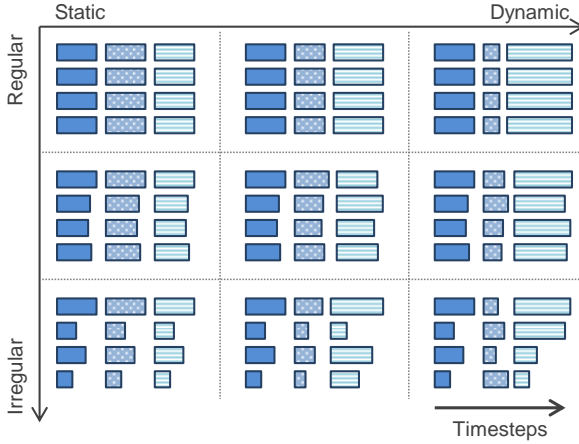


Figura 1 – Variação da carga em sistemas paralelos (PILLA, 2014).

$T_0$  é o tempo original da aplicação e  $T_1$  é o tempo observado após as mudanças.

$$S = \frac{T_0}{T_1} \quad (2.2)$$

## 2.2 BALANCEAMENTO DE CARGA INDEPENDENTE DE APLICAÇÃO

Balaceamento de carga em sistemas de memória distribuída é o processo de redistribuir trabalho entre diferentes recursos em *hardware* (BECKER; ZHENG; KALÉ, 2011). Pode-se então definir uma função de mapeamento na Equação 2.3, onde  $\mathcal{T}$  representa o conjunto de tarefas da aplicação e  $\mathcal{P}$  o conjunto de PE's da plataforma (PILLA, 2014), criando um mapa, ou estado, do sistema. Algoritmos de balanceamento de carga manipulam estes mapas, criando novas relações entre PE's e tarefas.

$$\mathcal{M} : \mathcal{T} \rightarrow \mathcal{P} \quad (2.3)$$

Existem diversas formas de se tomar decisões para migração de tarefas em balanceamento de carga. Em aplicações ricas em comunicação, tende-se a manter as tarefas que se comunicam mais próximas, distanciando-as das que se comunicam menos, assim evitando gargalos

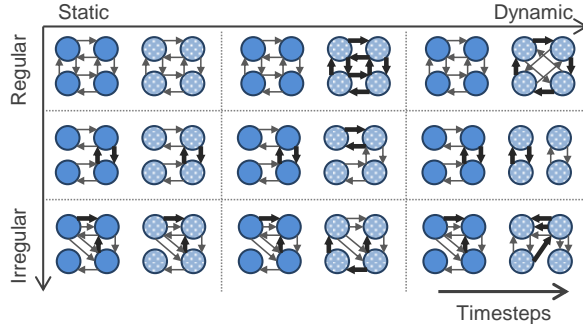


Figura 2 – Variação da comunicação em sistemas paralelos (PILLA, 2014).

de comunicação. Em aplicações com carga muito irregular (Figura 1), modelos de comportamento podem ser levados em conta para o balanceamento (BECKER; ZHENG; KALÉ, 2011), mas isso deve ser muito específico para cada domínio de aplicação.

Usualmente, estratégias de balanceamento de carga são fortemente acopladas à aplicação, devido a necessidade de preparar tarefas para migração. Um *framework* de serialização (apresentado na Seção 2.3) pode ser utilizado para facilitar esse processo, se este for desacoplado da aplicação, o balanceador também pode ser desacoplado com maior simplicidade. Também é possível realizar balanceamento de carga movendo processos inteiros, simplificando o trabalho do programador.

Quando desenvolvendo uma nova estratégia de balanceamento de propósito geral, é importante que ela seja independente de aplicação, ou seja, ela não deve se basear em dados específicos, que só seriam encontrados em um pequeno grupo de aplicações alvo.

### 2.2.1 Problemas atrelados ao balanceamento de carga

Ao se desenvolver uma nova estratégia de balanceamento, uma série de fatores devem ser considerados. Balanceamento de carga dinâmico não se aplica a todas as aplicações paralelas; por exemplo, aplicações bem comportadas, ou com comportamento conhecido, poderiam se beneficiar mais de um bom escalonamento inicial sem todos os sobrecustos do balanceamento dinâmico.

Os sobrecustos do balanceamento de carga vão além do tempo

de execução de uma estratégia. Sincronizar a aplicação, salvar o estado atual do sistema para que o balanceador o compreenda, realizar o remapeamento abstraído pela estratégia, serializar toda a informação que será remapeada, enviar as tarefas para os seus novos hospedeiros, sincronizar todas as *threads* novamente e retomar a aplicação; são todos passos que adicionam sobrecustos ao reescalonamento global.

Contudo, aplicações de comportamento imprevisível e com muita dinamicidade (como simulações) ainda podem usufruir de balanceamento de carga dinâmico, justificando seu uso principalmente em sistemas massivamente paralelos.

### 2.3 AMBIENTE PARALELO: CHARM++

**Charm++** (ACUN et al., 2016) é um sistema de programação paralela que implementa um modelo de objetos migráveis orientados a troca de mensagens (KALÉ, 2011). O ambiente paralelo coleta informações sobre o programa em execução, permitindo que o mesmo tenha informação sobre a carga das tarefas; esse conhecimento é usado pelo ambiente para executar estratégias de balanceamento de carga e roubo de tarefas (WS), que servem para melhorar o seu desempenho.

#### 2.3.1 Objetos migráveis e troca de mensagens assíncronas

O paralelismo em **Charm++** é feito através de troca de mensagens assíncronas. Os objetos, chamados pela linguagem de *chares*, possuem métodos de entrada (*entry methods*) que podem ser invocados por enti-

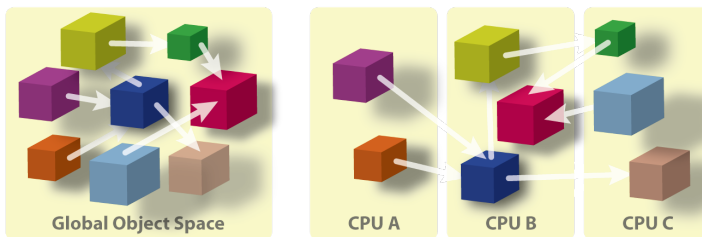


Figura 3 – Comunicação entre *chares* / divisão dos *chares* entre diferentes CPU's (CHARM++, acessado em 29 de setembro de 2017).

dades remotas de forma assíncrona. Isso significa que uma tarefa, após realizar uma chamada remota como essa, continuará seu fluxo normal de execução, sem esperar uma resposta (SCOTT, 2011). Esses métodos não possuem retorno, mas caso o *chare* que recebeu precise retornar uma informação, esta também pode ser enviada através de outra mensagem assíncrona.

Objetos migráveis são implementados usando o *framework* PUP (*Pack/Unpack*), que lida com a serialização dos objetos, permitindo que estes sejam migrados de um PE para outro (CHARM, 2016).

Aplicações em **Charm++** são compiladas tendo em mente um tipo de processador e interconexão específicos. Durante a execução, todos estes recursos são transparentes à aplicação, quem cuida do escalonamento dos recursos é o próprio sistema de execução. A visão do sistema de uma aplicação pode ser vista na Figura 4<sup>1</sup>.

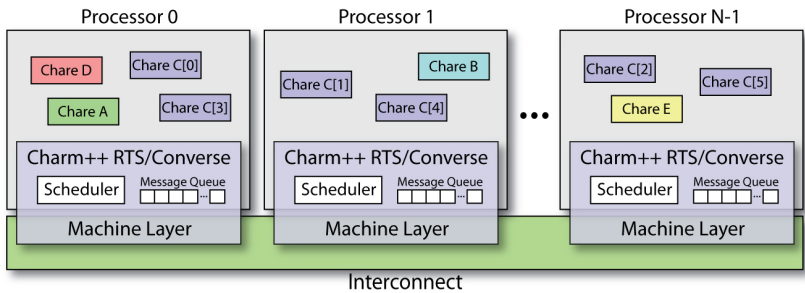


Figura 4 – Visão do sistema de uma aplicação Charm++.

### 2.3.2 *Framework* de balanceamento de carga

O **Charm++** provê um *framework* de balanceamento de carga baseado na abstrações das relações entre PE's e *chares* em um mapa (Equação 2.3). O sistema paralelo é responsável por abstrair as informações reais, passando para o programador só aquilo que ele precisa para realizar o remapeamento. No caso de um balanceador centralizado, é passada uma estrutura pronta, que pode ser modificada até chegar no mapeamento desejado e então, reprocessada pelo sistema (CHARM, 2016). Quando se trata de um balanceador distribuído, cada PE deve

<sup>1</sup> *Charm Runtime System*:  
<http://charmplusplus.org/tutorial/CharmRuntimeSystem.html>.

gerar mensagens únicas que são enviadas ao sistema no fim da execução da estratégia, para então efetuar o remapeamento.

## 2.4 BALANCEAMENTO DE CARGA CENTRALIZADO E HIERÁRQUICO

Existem diversas formas de se realizar balanceamento de carga. O balanceamento de carga global, ou centralizado, é o que faz uso de toda a informação disponível no sistema para executar suas estratégias. Esse tipo de implementação tem a vantagem de poder garantir maior precisão de balanceamento, sem criar nós mal balanceados devido à falta de informação.

No entanto, nem sempre essa é a forma mais recomendada, uma vez que em sistemas muito grandes agregar toda a informação e enviá-la a um único ponto cria um gargalo, que pode tornar a estratégia muito lenta e portanto diminuindo sua eficiência. Para sistemas de média e pequena escala, essa ainda é a forma mais recomendada de se realizar balanceamento de carga dinâmico, já que o sobrecusto de agregação é mais baixo.

Algoritmos centralizados podem também criar um gargalo de processamento. Em muitos casos eles não são paralelizáveis e, quando tem que lidar com uma quantidade massiva de dados e processamento, para máquinas de *petascale* ou *exascale*, por exemplo (ZHENG et al., 2010), acabam levando mais tempo do que seria tolerável. Estratégias hierárquicas buscam solucionar parte deste problema, fazendo com que o sistema se balanceie em diferentes níveis. Uma visão geral desse tipo de estratégia pode ser visto na Figura 5. Alguns exemplos de estratégias centralizadas e hierárquica (CHARM, 2016) são apresentados a seguir.

### 2.4.1 *GreedyLB*

O *GreedyLB* é uma clássica estratégia gulosa, que tenta remapear todas as tarefas entre unidades de processamento, começando com as mais pesadas, buscando cargas mais uniformes entre os PE's. Como ela tem um processamento inerentemente sequencial, essa estratégia normalmente não se beneficia de paralelismo (se o fizesse poderia resultar num balanceamento impreciso). Outra desvantagem dessa estratégia é que frequentemente ela move muitas tarefas entre PE's, portanto em sistemas de tarefas pesadas, o *GreedyLB* pode ter um sobrecusto de

remapeamento muito alto.

### 2.4.2 *RefineLB*

O *RefineLB* é uma estratégia baseada em refinamento, que busca tirar dos PE's apenas aquelas tarefas que fazem eles ficarem sobrecarregados, enviando-as para PE's subcarregados. Ela leva vantagem em relação ao *GreedyLB* por limitar o número de migrações, não sofrendo tanto com aplicações de tarefas pesadas quanto a estratégia gulosa. Contudo, em situações de muito desbalanceamento, uma estratégia que prioriza migrar poucas tarefas não terá um desempenho tão satisfatório quanto uma gulosa.

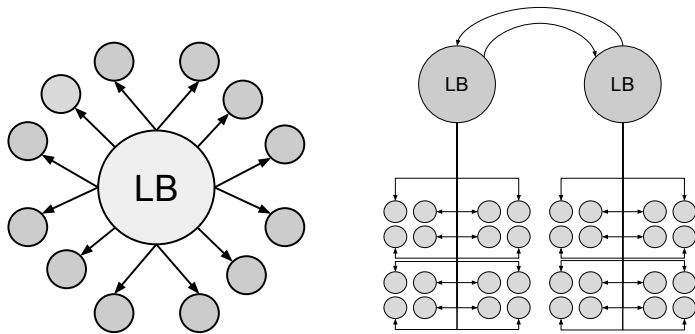


Figura 5 – Representação visual geral de balanceadores de carga centralizados (à esquerda) e hierárquicos (à direita).

### 2.4.3 *HybridLB*

*HybridLB* é uma estratégia hierárquica que separa o sistema em dois graus de granularidade. Em granularidade mais grossa, ele executa a estratégia de refinamento, enquanto em granularidade mais fina, ele executa uma estratégia gulosa. Isso limita as migrações entre nós de balanceamento e limita a quantidade de dados com os quais a estratégia gulosa tem que lidar. Essa estratégia também tem a vantagem de, uma vez que a granularidade grossa é resolvida, a fina pode ser paralelizada.

## 2.5 BALANCEAMENTO DE CARGA DISTRIBUÍDO

Diferente do balanceamento de carga global, o balanceamento de carga distribuído não se beneficia de ter toda a informação sobre o estado do sistema. Ao invés disso, cada núcleo, ou nó de processamento, tem apenas as suas informações locais. Isso faz com que o balanceamento não possa ser tão preciso, uma vez que não é viável obter toda a informação global, mas ao mesmo tempo não sofre com o sobrecusto dessa agregação. O balanceamento de carga distribuído é muito mais viável em sistemas de larga escala, com altíssimo grau de paralelismo, que podem se beneficiar de balanceamento, mesmo que não possam fazê-lo da forma mais precisa.

Por terem de início somente informação local, é importante também que nessas estratégias os PE's divulguem alguma informação sobre seus estados. Isso precisa ser feito para determinar uma carga média ou para os núcleos sobrecarregados terem conhecimento de quais são os subcarregados no sistema.

Propagar informação de forma absoluta, de todos para todos, ainda seria muito custoso, portanto algoritmos distribuídos devem fazer uma propagação mais inteligente de seus dados.

### 2.5.1 *GrapevineLB* (MENON; KALÉ, 2013)

O *GrapevineLB* é uma estratégia de balanceamento de carga distribuído baseada em refinamento dividido em duas etapas. Na primeira, os núcleos subcarregados informam todo o sistema sobre a sua situação através de um Protocolo *Gossip* (DEMERS et al., 1987), mais detalhado na Seção 3.1.2. Na segunda, os núcleos sobrecarregados buscam enviar suas tarefas para os subcarregados que os contactaram. Isso é feito até que os sobrecarregados atinjam um limite de tentativas ou estejam balanceados.

Os resultados obtidos em sua publicação original mostram o potencial de escalabilidade do algoritmo, uma vez que, para um mesmo conjunto de tarefas, mesmo com o aumento do sistema, o algoritmo é capaz de prover balanceamento num tempo semelhante. Isto se dá porque suas duas etapas escalam de forma diferente. A etapa de comunicação tem custo crescente de acordo com o número de PE's ( $C$ ), enquanto a etapa de troca de tarefas tem custo crescente em função da razão de tarefas ( $T$ ) por PE's ( $C$ ), temos uma relação  $C + \frac{T}{C}$ , dando a estratégia mais potencial de escalabilidade, uma vez que parte do seu



custo cresce com  $C$ , enquanto a outra parte, decresce.

## 2.6 CONCLUSÃO

Apesar de diversas aplicações terem um alto potencial de balanceamento, diversos fatores podem interferir em seu desempenho. Desbalanceamento de carga é um deles, este problema pode ser mitigado com o uso do reescalonamento global das tarefas, ou balanceamento de carga dinâmico. O Charm++ provê um ambiente paralelo que permite e facilita esse balanceamento, com um *framework* pronto para o desenvolvimento de balanceadores.

Existem diversos tipos de estratégias de balanceamento de carga dinâmico, para diferentes tipos de sistemas e aplicações. Este trabalho foca-se nas estratégias distribuídas, para sistemas de mais larga escala, com uso de múltiplos nós computacionais.



### 3 BALANCEAMENTO DE CARGA DISTRIBUÍDO ORIENTADO A PACOTES: *PACKDROPLB*

Este capítulo é focado em descrever o funcionamento do algoritmo proposto neste trabalho, o *PackDropLB*. Ele será descrito em detalhes, porém não serão abordados detalhes da sua implementação em *Charm++*, disponibilizada por completo nos apêndices deste trabalho e para livre acesso *online*<sup>1</sup>.

#### 3.1 ALGORITMO PROPOSTO: *PACKDROPLB*

A estratégia proposta é o *PackDropLB*, que se baseia na ideia de criar pequenos conjuntos de tarefas, chamados de pacotes. Ao transportar grupos de tarefas com um tamanho uniforme são necessárias menos mensagens para se enviar mais carga, dessa forma diminuindo custos de comunicação.

Os pacotes são pequenos conjuntos de tarefas de carga variável, estimada para facilitar a movimentação de tarefas sem a necessidade de processar informação relacionada a sua carga específica (Equação 3.1). Nós não têm conhecimento de quanta carga estarão recebendo, essa informação é estimada com base na quantidade de pacotes recebidos. Uma vez que não se tem informação global, tenta-se melhorar a situação com menos recursos, evitando grandes custos de comunicação para obter toda a informação do sistema.

A Figura 6 mostra o fluxo de funcionamento do algoritmo principal, descrito a seguir:

1. Os dois primeiros passos, executados por todos os PE's, são realizar duas reduções, a primeira servindo para agregar a informação sobre a carga média e a segunda sobre o número de tarefas no sistema;
2. Em seguida, cada um dos núcleos se divide entre sobrecarregados e subcarregados;
3. Aqueles que são sobrecarregados irão começar a criar seus pacotes para transferência, enquanto os demais irão começar a propagar seus identificadores pela rede através de um protocolo *Gossip*

---

<sup>1</sup>As implementações podem ser acessadas no repositório git: <https://github.com/viniciusmctf/PackDropLB-bachthesis-research/>

(mais detalhado posteriormente na Seção 3.1.2), com o objetivo de informar os sobrecarregados a quem eles devem enviar suas tarefas;

4. Todos os processadores são sincronizados para se iniciar a troca de tarefas;
5. Os PE's sobrecarregados tentam enviar seus pacotes aos subcarregados.

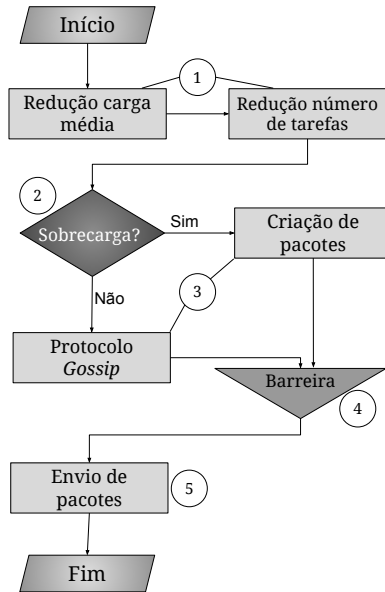


Figura 6 – Fluxo de funcionamento do *PackDropLB*, seguido individualmente por cada PE.

### 3.1.1 Criação de pacotes

O tamanho de cada pacote,  $\overline{ps}$ , deveria ser baseado no tamanho médio de uma tarefa no sistema,  $\overline{ts}$ , no número de núcleos disponíveis,  $nC$ , e no número total de tarefas (em **Charm++**, *chares*) no sistema,  $nT$ . As proporções estão descritas na Equação 3.1.

$$ps = \bar{ts} \times \left(2 - \frac{nC}{nT}\right) \quad (3.1)$$

Essa proporção faz com que a quantidade de tarefas em um pacote, baseado na carga média, seja algo próximo a 2. Ainda se permite uma variação dentro do tamanho do pacote, uma vez que não é possível atingir sempre o tamanho exato estimado. A variação usada foi de 5% para mais e para menos, criando dois limites, superior e inferior, como evidenciado na Figura 7. A taxa de precisão no balanceamento pode ser mudada dependendo da aplicação alvo, uma vez que isso depende muito da característica das cargas de cada aplicação (aplicações com muitas cargas leves irão se beneficiar mais de um balanceamento de alta precisão nesse caso).

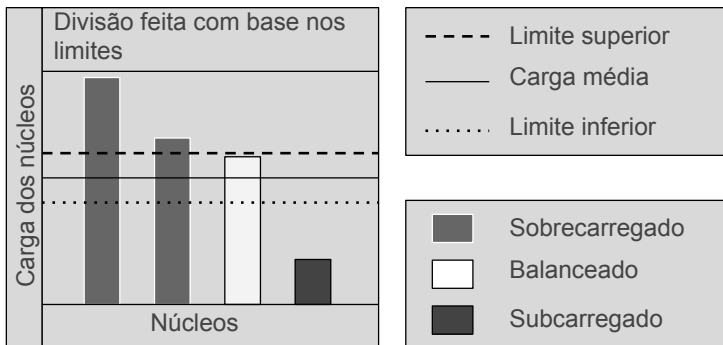


Figura 7 – Um sistema desbalanceado, com núcleos divididos em grupos (Balanceado, Subcarregado e Sobrecarregado) por limites relacionados à carga média.

No Algoritmo 1, o procedimento é detalhado.  $\mathcal{T}$  é o conjunto de tarefas de um núcleo,  $carga$  é a carga total do núcleo que está criando os pacotes,  $limite$  é o limiar de sobrecarga e  $Pacotes$  é o conjunto de pacotes gerado.

É importante notar que os elementos são retirados da lista de tarefas de um PE sempre na mesma ordem, priorizando as tarefas de menor carga, evitando criar pacotes que deixariam os núcleos subcarregados, sobrecarregados.

---

**Algoritmo 1:** Criação de pacotes
 

---

**Entrada:**  $ps, \mathcal{T}, carga, limite$   
**Saída:** *Pacotes*

- 1  $P \leftarrow \emptyset, Pacotes \leftarrow \emptyset$
- 2 **enquanto**  $carga > limite$  **faça**
- 3      $t \leftarrow a \in \mathcal{T} \mid a$  é o menor elemento de  $T$
- 4      $\mathcal{T} \leftarrow \mathcal{T} \setminus \{t\}$
- 5      $P \leftarrow P \cup \{t\}$
- 6      $carga \leftarrow carga - t$
- 7     **se** (*tamanho de P*)  $> ps$  **então**
- 8          $Pacotes \leftarrow Pacotes \cup P$
- 9          $P \leftarrow \emptyset$
- 10    **fim**
- 11 **fim**
- 12  $Pacotes \leftarrow Pacotes \cup P$

---

### 3.1.2 Protocolo *Gossip* (DEMERS et al., 1987)

A propagação de informação com Protocolo *Gossip* funciona como uma rede de fofocas. Criando uma abstração para um grafo, como visto na Figura 8, podemos dizer que os membros da rede são os vértices e que as arestas são a propagação de informação. O funcionamento se baseia em nós que possuem uma informação que deve ser propagada passando essa informação para número  $t$  de nós, chamamos essa variável de *throughput*. Cada um desses nós unirá a informação obtida à sua e enviará esta para mais  $t$  nós do grafo. Isso se repete até que se atinja um limite predeterminado de iterações da propagação. O número de iterações varia de acordo com o número de vértices no grafo, a Figura 8 mostra três iterações do protocolo.

Por ser uma estratégia de propagação que funciona bem em sistemas distribuídos, já mostrando um desempenho escalável no *GrapevineLB*, decidiu-se adotá-la na implementação do *PackDropLB*.

### 3.1.3 Envio de pacotes

Na Figura 6, etapa de *Envio de Pacotes*, há quatro momentos envolvidos. O primeiro é descrito no Algoritmo 2. Nesta etapa, os PE's sobrecarregados do sistema irão enviar todos os seus pacotes aos possí-

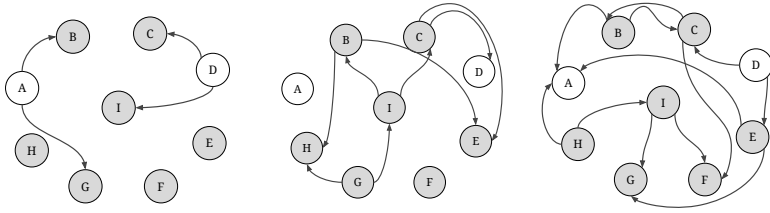


Figura 8 – Três passos de execução do Protocolo *Gossip*, com *throughput* de 2. À esquerda o primeiro passo, no meio, o segundo e à direita, o terceiro.

veis alvos determinados durante a etapa de propagação de informação, tendo uma saída no formato de um conjunto de pares Pacote/Alvo ( $R$ ), com o qual serão executadas as requisições que levam à segunda etapa.

---

**Algoritmo 2:** Envio de pacotes

---

**Entrada:** *Pacotes*, *Alvos*

**Saída:** relação entre pacotes e alvos esperando confirmação  
 $R$

```

1  $R \leftarrow \emptyset$ 
2 enquanto  $Pacotes \neq \emptyset$  faça
3   pacote  $\leftarrow p \mid p \in Pacotes$ 
4   alvo  $\leftarrow a \mid a \in Alvos \wedge a$  é um elemento aleatório de
     Alvos
5    $Pacotes \leftarrow Pacotes \setminus \{pacote\}$ 
6    $R \leftarrow R \cup \{(pacote, alvo)\}$ 
7 fim
```

---

Na segunda etapa, os PE's que estão recebendo tarefas devem decidir se a **carga de um novo pacote** ( $ps$ ) quando somada a sua **carga local** ( $carga$ ) irá se manter abaixo do *limiar de sobrecarga*, condição necessária para aceitar a tarefa. Tarefas também podem ser enviadas de forma forçada, quando não estão sendo enviadas pela primeira vez, para acelerar a convergência do balanceador de carga.

Na terceira etapa, descrita no Algoritmo 3, os nós subcarregados enviam mensagens de volta àqueles que tentaram lhes enviar tarefas. Essas mensagens são no formato de uma **tripla alvo/pacote/sucesso** (APS), onde alvo é quem enviou a mensagem, pacote é o conjunto de tarefas que ele deve receber e sucesso é um booleano que indica se

---

**Algoritmo 3:** Remapeamento das tarefas
 

---

**Entrada:**  $APSSs$ , mapeamento original  $\mathcal{M}$ ,  $local$

**Saída:** novo mapeamento  $\mathcal{M}'$ ,  
conjunto de novas tentativas  $NT$

```

1  $\mathcal{M}' \leftarrow \mathcal{M}$ 
2 enquanto  $APSSs \neq \emptyset$  faça
3    $(a, p, s) \leftarrow e \mid e \in APSSs$ 
4    $APSSs \leftarrow APSSs \setminus \{(a, p, s)\}$ 
5   se  $s$  então
6      $\mathcal{M}' \leftarrow \mathcal{M}' \setminus \{(local, p)\}$ 
7      $\mathcal{M}' \leftarrow \mathcal{M}' \cup \{(a, p)\}$ 
8   senão
9      $NT \leftarrow p$ 
10  fim
11 fim

```

---

a migração foi bem sucedida. A saída dessa terceira etapa é o novo mapeamento ( $\mathcal{M}'$ ) das tarefas que originalmente estavam no nó sobrecarregado ( $local$ ).

A última etapa ocorre apenas no caso de haverem migrações que falhem em auxiliar um PE subcarregado na segunda etapa, tendo um valor de *sucesso* falso na terceira. Nesse caso, os pacotes que foram parar no conjunto  $NT$  terão mais tentativas de migração, até que, atingido um certo limite ( $lt$ ), ele terá uma migração forçada para o próximo alvo escolhido, ou seja, ele não pode retornar um valor de  $s$  falso.

### 3.2 CONCLUSÃO

O principal algoritmo estudado para o desenvolvimento da nova estratégia, foi o *GrapevineLB* (Seção 2.5.1). Como ele é um algoritmo baseado em agregação de informação e balanceamento, seguir por este mesmo caminho pareceu uma abordagem natural. No entanto, nessa pesquisa decidiu-se por mudar a migração tarefa a tarefa, que pode causar muita comunicação desnecessária, já que a maioria das tarefas migradas é muito pequena (FREITAS; PILLA, 2017); para isso foi usada uma técnica de criação de grupos de tarefas a serem migradas, mais detalhada anteriormente na Seção 3.1.1.



O principal ganho esperado com essa mudança é diminuição no volume de comunicação, ao custo de um balanceamento menos preciso. Isso acontece pois um núcleo remoto tem apenas uma estimativa da carga que irá receber quando a aceita, contudo, isso garante muito menos comunicação, uma vez que são necessárias menos mensagens para se aceitar mais tarefas.



## 4 AVALIAÇÃO DE DESEMPENHO

Este capítulo tem como principal intuito explicar e descrever os *benchmarks* (Seção 4.1) e o ambiente de testes utilizado (Seção 4.2) em cada uma das execuções. Ao seu final, na Seção 4.3, são expostos os resultados obtidos e em seguida, as conclusões que podem ser tiradas a partir deles.

### 4.1 BENCHMARKS E APLICAÇÕES

Foram utilizados dois perfis paralelos sintéticos (*lb test* e *Stencil 3D*) e duas aplicações de mundo real (LULESH e *LeanMD*) para avaliação da nova estratégia de balanceamento de carga. Estes foram escolhidos por apresentarem uma série de diferenças, sendo todas capazes de aproveitar-se do balanceamento de carga dinâmico em tempo de execução, além de todas proverem implementações prontas no ambiente do Charm++.

#### 4.1.1 lb test

O **lb test** é um *benchmark* sintético usado para testes de balanceadores de carga. Por ser especificamente desenvolvido para o uso com balanceadores de carga, é possível ajustar seus parâmetros para gerar diversos tipos de distribuição de trabalho e comunicação. Para a avaliação de desempenho dos algoritmos, decidiu-se por executar duas observações deste perfil paralelo, variando a quantidade de tarefas. A primeira com 18990 (148 ~ tarefas por PU) e a segunda com 36990 (288 ~ tarefas por PU) tarefas, com carga variando entre 300 e 90000ms em 150 iterações do *benchmark*, com balanceamento de carga a cada 40 iterações e topologia de comunicação em anel.

A grande diferença entre carga máxima e mínima entre as tarefas é usada para gerar mais desbalanceamento, tendo tarefas de carga bem variadas. O número de tarefas foi definido de forma a gerar uma escala comparativa (uma maior que a outra) e não utilizar potências de 2, já que estas aumentam a tendência do sistema a se manter balanceado.

### 4.1.2 *Stencil 3D*

*Stencils* são padrões de computação paralela, em que determinada célula de um conjunto precisa se comunicar com todas as suas vizinhas para realizar uma computação e é paralelizado dividindo seu conjunto em pedaços menores (*tiling*). O **Charm++** possui um *benchmark* de *Stencil 3D*, uma malha tridimensional cujas bordas têm uma temperatura alta e o centro é frio. O processamento é feito permitindo que a temperatura seja recalculada por toda a malha, fazendo o calor se espalhar das bordas para o centro. Nesse caso, escolheu-se o tamanho de bloco 540 e de *array* 12, estes tamanhos foram definidos para gerar um tamanho relevante suficiente para o balanceamento se justificar; além disso os tamanhos não são potências de 2, uma vez que isso potencializa o desbalanceamento. Ele foi processado ao longo de 100 iterações, com balanceamento de carga realizado a cada 15 iterações.

### 4.1.3 LULESH

O **LULESH** (*Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics*) (KARLIN et al., 2013) é um simulador de hidrodinâmica altamente simplificado, usado para resolver uma simples *Sedov blast wave* (SEDOV, 1946) com respostas analíticas, contudo, esta aplicação serve como base para implementação de outros problemas desta área (KARLIN et al., 2012). O **LULESH** cria uma malha tridimensional para realizar suas simulações e tem sua paralelização feita num padrão estêncil, conforme explicado na seção 4.1.2. Os parâmetros para dimensões de bloco e elementos do **LULESH** foram de 140 e 7. Os parâmetros desta aplicação seguem as mesmas justificativas do *Stencil 3D*.

### 4.1.4 *LeanMD*

*LeanMD* (MEHTA, 2004) é uma aplicação de dinâmica molecular, que simula o comportamento de átomos baseados no *potencial Lennard-Jones*, que descreve a interação entre moléculas ou átomos<sup>1</sup>. Sua computação é paralelizada com a decomposição espacial e de forças, realizando cálculos de força entre dois átomos a uma certa distância (PILLA, 2014). *LeanMD* foi executado por 500 iterações, com balanceamento de carga iniciando na iteração de número 20 e ocor-

---

<sup>1</sup>Lista de *Mini-Apps* do **Charm++**: <https://charmplusplus.org/benchmarks/>

rendo de novo a cada 100 iterações. As dimensões utilizadas para os conjuntos de células foram de  $15 \times 20 \times 15$  e de  $10 \times 15 \times 10$ , respectivamente. O primeiro conjunto de parâmetros gera um experimento mais pesado, com mais tarefas de carga maior, enquanto o segundo é mais leve, gerando menos tarefas de carga menor.

## 4.2 AMBIENTES DE TESTES

Todos os testes preliminares foram rodados em 16 nós do *cluster Genepi*<sup>2</sup>, cada um contando com 2 processadores *Intel Xeon E5420 QC @ 2.5GHz* com 4 *cores* cada, totalizando 128 PU's. Estas máquinas estão equipadas com 8GB de memória RAM e 160GB de armazenamento SATA (*Serial Advanced Technology Attachment*) HDD (*Hard Disk Drive* - Disco Rígido). Os nós estão interconectados utilizando a tecnologia *InfiniBand 20G*, sendo que as placas contidas em cada um são *Mellanox ConnectX IB 4X DDR MT26418*. O sistema operacional utilizado foi o Ubuntu 14.04 em versão *min*, a versão do *Charm++* adotada foi a 6.7.0 e o GCC foi utilizado na versão 5.4.0.

A Tabela 1 mostra quantas execuções de cada *benchmark* foram realizadas para cada balanceador na colheita dos resultados, de forma que o *lb test M* se refere àquele com 18990 tarefas e o *lb test G* àquele com 36990 tarefas; ao mesmo tempo que o *LeanMD M* se refere àquele de dimensões  $10 \times 15 \times 10$  e o *LeanMD G* àquele de dimensões  $15 \times 20 \times 15$ .

<i>Benchmark</i>	Número de execuções
<i>lb test M</i>	12
<i>lb test G</i>	18
<i>Stencil 3D</i>	12
<i>LULESH</i>	20
<i>LeanMD M</i>	14
<i>LeanMD G</i>	5

Tabela 1 – Número de execuções dos experimentos para averiguação dos resultados com *benchmarks* menores (**M**) e maiores (**G**).

<sup>2</sup>Experimentos apresentados neste trabalho foram realizados no Grid5000, mantido pelo grupo de propósitos científicos hospedado pela Inria e incluindo CNRS, RENATER e diversas Universidades e outras organizações (ver <https://www.grid5000.fr>).

### 4.3 RESULTADOS OBTIDOS

Esta seção irá apresentar os resultados obtidos em cada um dos cenários detalhados na Seção 4.1, bem como uma análise destes, buscando entender o que foi observado em cada situação e os frutos que podem ser colhidos do desenvolvimento da nova estratégia, o *PackDropLB*, e da sua abordagem, o balanceamento orientado a pacotes.

Diversos resultados nas seções a seguir foram representados através de gráficos de velas (*boxplot*), nestes casos a base dos tempos de execução não foi 0s, uma vez que os resultados nesta representação seriam muito deformados desta forma, perdendo um pouco de seu significado. Da forma abordada, eles podem ser analisados com mais precisão.

#### 4.3.1 lb test

Os resultados observados para o *lb test* mostram que o ambiente e o número de tarefas estudados já são muito grandes para usar estratégias como *Greedy* ou *Hybrid*, degradando seu desempenho em relação a estratégias que priorizam a migração de um número menor de tarefas. A Figura 9 mostra que o desempenho da estratégia proposta foi, em ambos os casos, melhor do que àquela que mais se assemelha, o *Grapevine*.

Os resultados também se mostraram sempre muito semelhantes aos do *Refine* (Tabela 2), uma estratégia centralizada que busca minimizar migração de tarefas, mostrando que a estratégia proposta realiza balanceamento satisfatório, mesmo quando comparada a uma estratégia centralizada.

Estratégia	18990 tarefas	36990 tarefas
PackDrop	46,5709s	88,7720s
Grapevine	47,7737s	92,5051s
Refine	46,5048s	89,2286s
Greedy	48,6943s	93,2736s
Hybrid	48,1052s	96,1135s
Sem LB	53,1991s	100,3843s

Tabela 2 – Tempo médio de execução obtidos do *benchmark lb test* para as diferentes estratégias.

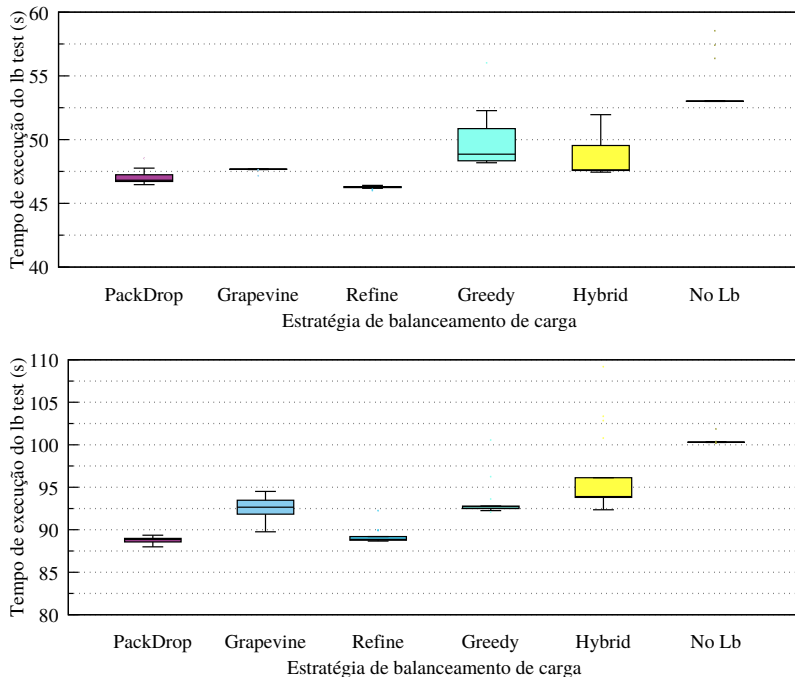


Figura 9 – Tempos de execução do *benchmark lb test* para 18990 tarefas (em cima) e 36990 tarefas (embaixo) com diferentes balanceadores de carga.

### 4.3.2 Stencil 3D

O *Stencil 3D* foi um caso diferente do *lb test*, uma vez que, neste *benchmark*, não foi observado qualquer ganho ao se realizar balanceamento de carga, apenas foi um sobrecurso decorrente da migração das tarefas e tempo de decisão dos balanceadores de carga, além de um aumento no tempo de execução de cada iteração, provavelmente decorrente de uma perda de localidade espacial da informação dos *arrays*.

Na Figura 10 pode-se observar os tempos de algumas estratégias. Nota-se que apesar do *PackDrop* ter o melhor desempenho quando comparado ao *Refine* e ao *Grapevine*, ele ainda não se equipara ao resultado sem balanceamento. Isso pode ocorrer decorrente da pequena *ghost zone* deste *stencil* (de tamanho 1), que aumenta o custo de comunicação, mas estabiliza a carga entre iterações, impossibilitando que

as diferenças de carga se acumulem. Os tempos do *Greedy* e do *Hybrid* excederam os maiores resultados mostrados, tornando-se irrelevantes para esta análise, com médias de 104,45s e 134,30s, respectivamente.

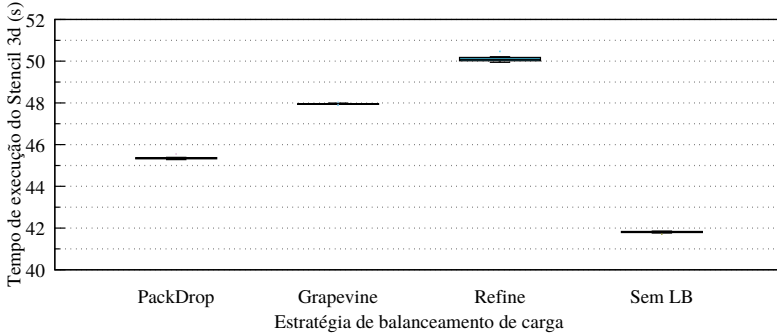


Figura 10 – Tempos de execução do *benchmark Stencil 3D* para dimensões de bloco 540 e de *array* 12 com diferentes balanceadores de carga.

### 4.3.3 LULESH

O *LULESH* apresentou um comportamento semelhante ao *Stencil 3D*, também sendo incapaz de ter ganhos no balanceamento de carga, como pode ser observado na Figura 11. Praticamente todas as observações feitas a respeito do *Stencil 3D* na Seção 4.3.2 se repetem aqui, tendo também um caso em que as *ghost zones* de tamanho 1 podem ter prejudicado o paralelismo pela grande intensidade da comunicação.

Como isso não foi observado anteriormente, por limitações de tempo esse parâmetro não foi modificado para mais resultados, contudo, é algo a se observar no futuro.

Como no caso anterior, tanto o *Greedy* quanto o *Hybrid* mostraram resultados muito piores que as demais estratégias e com isso foram omitidos, tendo tempos totais de execução da aplicação de 94,94s e 93,34s, respectivamente. Da mesma forma que se observou que a estratégia proposta foi a que gerou menos perda, devido ao número limitado de migrações realizadas, causando menos aumento na comunicação após realizar o remapeamento das tarefas.



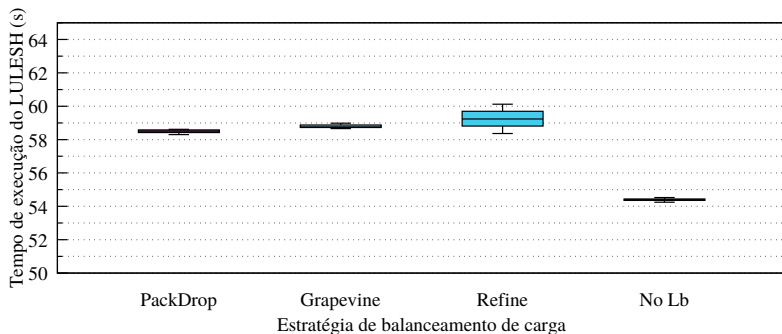


Figura 11 – Tempos de execução do *LULESH* para dimensões de bloco 140 e de *array* 20 com diferentes balanceadores de carga.

#### 4.3.4 LeanMD

Os resultados do *LeanMD* (exibidos nas Figuras 12 e 13) mostram que tanto o *PackDrop* como o *Grapevine* e o *Refine* são capazes de melhorar o tempo de execução desta aplicação, enquanto novamente o desempenho do *Greedy* e do *Hybrid* se mostram mais uma vez insatisfatórios. Isso mostra mais uma vez como o tamanho da plataforma cobra dessas duas últimas estratégias de balanceamento, tornando mais eficaz o uso de estratégias distribuídas (como o *PackDrop* e o *Grapevine*) ou que limitam a quantidade de migrações (como o *Refine*).

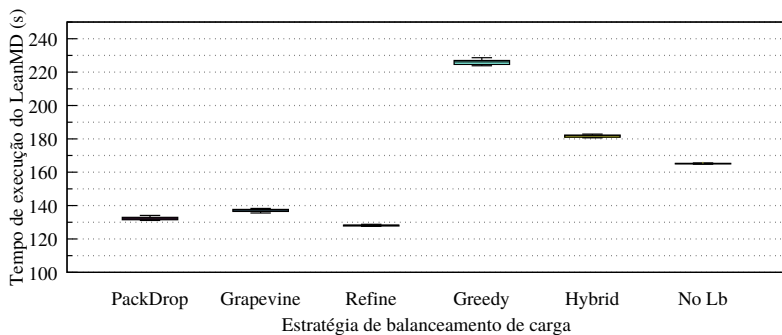


Figura 12 – Tempos de execução do *LeanMD* para dimensões  $10 \times 15 \times 10$  com diferentes balanceadores de carga.

Comparando as diferentes dimensões utilizadas no *LeanMD*, percebemos que os ganhos foram semelhantes com as três estratégias de melhor desempenho, com o *Refine* e o *PackDrop* sempre se equiparando, tendo o *Grapevine* um pouco atrás.

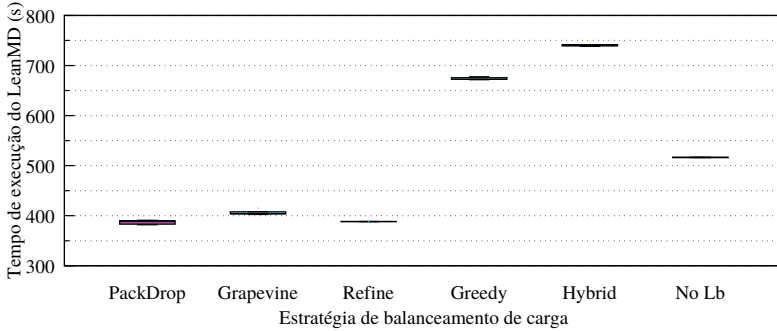


Figura 13 – Tempos de execução do *LeanMD* para dimensões  $15 \times 20 \times 15$  com diferentes balanceadores de carga.

Não foi possível testar a estratégia em ambientes maiores por restrições de tempo, contudo, os resultados indicam que com o aumento do número de nós no sistema, as duas estratégias distribuídas tendem a manter seu custo e eficiência de balanceamento, enquanto a centralizada (o *Refine*) tende a ter o custo de agregação de informação e remapeamento aumentado com o aumento da plataforma.

#### 4.4 CONCLUSÃO

Observando todos os resultados, notamos que a estratégia apresentada (*PackDrop*) é capaz de entregar um ganho de desempenho considerável nos casos observados, principalmente quando comparado à estratégia mais semelhante a ele, o *Grapevine*.

A Figura 14 mostra os *speedups* (Equação 2.2,  $\mathcal{S} - 1$ ) relativos à execuções sem balanceamento de carga. Esses experimentos foram todos realizados utilizando 128 PUs, distribuídas em 16 nós diferentes e mostram que o *speedup* atingido pela nova abordagem é mais eficiente que o *Grapevine*, comparando-se mais ao *Refine*. Contudo, são necessárias mais investigações, especialmente testes de escalabilidade, para garantir que mesmo em ambientes maiores, o *PackDrop* continua sendo tão eficiente quanto outras estratégias distribuídas.

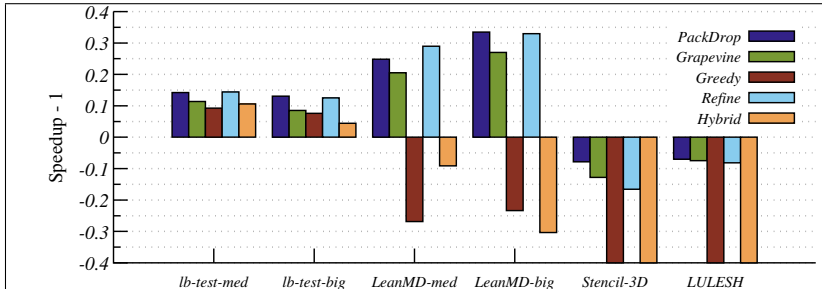


Figura 14 – Speedups ( $S - 1$ ) de todos os *benchmarks* e aplicações testadas com cada estratégia de balanceamento de carga.

Tanto o *Stencil 3D* como o *LULESH* mostraram resultados insatisfatórios para todos os balanceadores de carga, devido a forma como o paralelismo delas é feito (com comunicação entre cada iteração e *ghost zones* de tamanho 1) a migração das tarefas acaba gerando perda de desempenho.



## 5 CONCLUSÃO

Este trabalho apresentou *PackDropLB*, uma nova estratégia de balanceamento de carga, seguindo uma nova abordagem orientada à pacotes. Esta foi implementada no ambiente paralelo **Charm++** (Seção 2.3). O *framework* de balanceamento de carga deste ambiente permitiu que ela fosse comparada a outras estratégias nativas através de *benchmarks* e aplicações reais.

O principal diferencial da estratégia em relação à outras abordagens distribuídas é a criação de pacotes, que fazem com que não seja necessário trocar informações sobre a carga das tarefas antes das migrações, assim como possibilita a migração de múltiplas tarefas de uma única vez.

Essa nova abordagem mostrou resultados promissores, tendo ganho de desempenho em todos os casos de teste experimentados em relação à outra estratégia distribuída observada, o *GrapevineLB*. Contudo, os ganhos não puderam ser completamente explorados neste trabalho por restrições da plataforma de testes, muito pequena para mostrar o ganho real de uma estratégia de reescalonamento distribuída.

### 5.1 TRABALHOS FUTUROS

Os testes realizados neste trabalho ainda não tem escala grande o suficiente para mostrar o desempenho das estratégias distribuídas, como foi visto no Capítulo 4. Uma continuação importante para este trabalho é a realização de mais experimentos em ambientes maiores, onde existe a possibilidade para ainda maior ganho de desempenho, principalmente quando comparados aos algoritmos centralizados.

Durante o desenvolvimento do *PackDropLB*, pensou-se numa nova estratégia de balanceamento de carga distribuída. Esta seguiria a abordagem orientada a pacotes, usando princípios de roubo de tarefas (JANJIC; HAMMOND, 2013) para fazer o remapeamento de trabalho. Isso tira a responsabilidade de entregar pacotes dos PE's sobrecarregados e passando-a para os subcarregados.

Em ambientes massivamente paralelos, comunicação é um sobre-custo que aparece frequentemente. As tarefas e a comunicação entre elas podem ser representadas como um grafo, como vértices e arestas, respectivamente. Tendo isso em mente, algoritmos de particionamento de grafos são muito utilizados no reescalonamento global de tarefas.

Exemplos disso são a biblioteca *Zoltan* (DEVINE et al., 2009) e as ferramentas SCOTCH (PELLEGRINI, 2009). Contudo, a maioria das estratégias apresentadas não aproveita o paralelismo dos sistemas modernos. É possível desenvolver uma nova estratégia sensível à comunicação com o uso da abordagem orientada à pacotes, buscando o aumento da escalabilidade desse tipo de abordagem.

## REFERÊNCIAS

- ACUN, B. et al. Power, reliability, and performance: One system to rule them all. *Computer*, IEEE, v. 49, n. 10, p. 30–37, 2016.
- BECKER, A.; ZHENG, G.; KALÉ, L. V. Load balancing, distributed memory. In: PADUA, D. (Ed.). *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011. p. 1043–1051.
- BHATELE, A. et al. *NAMD: A Portable and Highly Scalable Program for Biomolecular Simulations*. [S.l.], February 2009.
- BOITO, F. Z. et al. High performance I/O for seismic wave propagation simulations. In: *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. [S.l.: s.n.], 2017. p. 31–38.
- CHARM. *Charm++ Manual*. 6.7.1. ed. 201 North Goodwin Avenue, Urbana, IL, 4 2016. Chapter I, section 6.3. Chapter II, section 7. Chapter III, section 24.
- CHARM++. acessado em 29 de setembro de 2017. Charm++ web page in the Parallel Programming Lab from the University of Illinois. <<http://charm.cs.illinois.edu/research/charm>>.
- DEMERS, A. et al. Epidemic algorithms for replicated database maintenance. In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 1987. (PODC '87), p. 1–12.
- DEVECI, M. et al. Fast and high quality topology-aware task mapping. In: *2015 IEEE International Parallel and Distributed Processing Symposium*. [S.l.: s.n.], 2015. p. 197–206.
- DEVINE, K. et al. Getting started with zoltan: A short tutorial. In: *Proc. of 2009 Dagstuhl Seminar on Combinatorial Scientific Computing*. [S.l.: s.n.], 2009. Also available as Sandia National Labs Tech Report SAND2009-0578C.
- DUPROS, F. et al. High-performance finite-element simulations of seismic wave propagation in three-dimensional nonlinear inelastic geological media. *Parallel Computing*, v. 36, n. 5–6, p. 308 – 325, 2010. Parallel Matrix Algorithms and Applications.

- FLYNN, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21, n. 9, p. 948–960, Sept 1972.
- FREITAS, V. M. C. T. de; PILLA, L. L. Comparando diferentes ordenações de tarefas em balanceamento de carga distribuído. In: *Anais da Escola Regional de Alto Desempenho do Rio Grande do Sul 2017*. [S.l.: s.n.], 2017.
- GIOACHIN, F. et al. Scalable cosmology simulations on parallel machines. In: *VECPAR 2006, LNCS 4395*, pp. 476–489. [S.l.: s.n.], 2007.
- GUSTAFSON, J. L. Amdahl's law. In: PADUA, D. (Ed.). *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011. p. 53–59.
- JANJIC, V.; HAMMOND, K. How to be a successful thief. In: WOLF, F.; MOHR, B.; MEY, D. an (Ed.). *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 114–125.
- JETLEY, P. et al. Scaling hierarchical n-body simulations on gpu clusters. In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. Washington, DC, USA: IEEE Computer Society, 2010. (SC '10).
- KALÉ, L.; KRISHNAN, S. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In: PAEPCKE, A. (Ed.). *Proceedings of OOPSLA'93*. [S.l.]: ACM Press, 1993. p. 91–108.
- KALÉ, L. V. Charm++. In: PADUA, D. (Ed.). *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011. p. 256–264.
- KARLIN, I. et al. *Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory*. [S.l.], 2012. 1-17 p.
- KARLIN, I. et al. Exploring traditional and emerging parallel programming models using a proxy application. In: *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*. Boston, USA: [s.n.], 2013.
- KUCK, D. J. Parallel computing. In: PADUA, D. (Ed.). *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011. p. 1409–1416.



- LIU, L. et al. A throughput-driven task creation and mapping for network processors. In: SPRINGER. *International Conference on High-Performance Embedded Architectures and Compilers*. [S.l.], 2007. p. 227–241.
- MEHTA, V. *LeanMD: A Charm++ framework for high performance molecular dynamics simulation on large parallel machines*. Dissertação (Mestrado) — University of Illinois at Urbana-Champaign, 2004.
- MEI, C. et al. Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime. In: *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*. Seattle, WA: [s.n.], 2011.
- MENON, H.; KALÉ, L. A distributed dynamic load balancer for iterative applications. In: *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: ACM, 2013. (SC '13), p. 15:1–15:11.
- NAVAUX, P. O. A.; PILLA, L. L. Fundamentos das arquiteturas para processamento paralelo e distribuído. In: *Anais da Escola Regional de Alto Desempenho do Rio Grande do Sul*. [S.l.: s.n.], 2011.
- OSTHOFF, C. et al. Atmospheric model cluster performance evaluation on hybrid mpi/openmp/cuda programming model platform. In: *2012 31st International Conference of the Chilean Computer Science Society*. [S.l.: s.n.], 2012. p. 216–222.
- PEARCE, O. et al. Quantifying the effectiveness of load balance algorithms. In: *Proceedings of the 26th ACM International Conference on Supercomputing*. New York, NY, USA: ACM, 2012. (ICS '12), p. 185–194.
- PELLEGRINI, F. Distillating knowledge about scotch. In: NAUMANN, U. et al. (Ed.). *Combinatorial Scientific Computing*. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009. (Dagstuhl Seminar Proceedings, 09061). <<http://drops.dagstuhl.de/opus/volltexte/2009/2091>>.
- PILLA, L. L. *Topology-Aware Load Balancing for Performance Portability over Parallel High Performance Systems*. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul, Porto Alegre, Rio Grande do Sul, 4 2014.

SCOTT, M. L. Synchronization. In: PADUA, D. (Ed.). *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011. p. 1989–1996.

SEDOV, I. L. Propagation of strong shock waves. p. 241–250, 1946.

TANENBAUM, A. S.; GOODMAN, J. R. *Structured Computer Organization*. 4th. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998.

ZHENG, G. et al. Hierarchical load balancing for charm++ applications on large supercomputers. In: *2010 39th International Conference on Parallel Processing Workshops*. [S.l.: s.n.], 2010. p. 436–444.

**APÊNDICE A - Códigos do *PackDropLB***



## A.1 ORDEREDELEMENT.H

```

/* Copyright Vinicius F. @ UFSC [2016]

Contact: vinicius.mctf@grad.ufsc.br

*/

#ifndef _ORDERED_ELEMENT_H_
#define _ORDERED_ELEMENT_H_

/* This is a load balancing entity, used for
load balancing heaps and ordered queues.
*/
struct Element {
    Element(int id, double load) : id(id), load(load) {}
    // Element(Element& e) : id(e.id), load(e.load) {}
    Element() {}
    int id;
    double load;
};

inline bool operator<(const Element& lhs, const Element&
rhs) {
    return lhs.load < rhs.load;
}

inline bool operator>(const Element& lhs, const Element&
rhs) {
    return lhs.load > rhs.load;
}

#endif /* _ORDERED_ELEMENT_H_ */

```

## A.2 PACKDROPLB.H

```

#ifndef PACK_DROP_LB
#define PACK_DROP_LB

// Charm includes
#include "DistBaseLB.h"
#include "PackDropLB.decl.h"

// Data structures includes
#include "OrderedElement.h"
#include <list>
#include <vector>
#include <queue>
#include <unordered_map>
#include <deque>

// Algorithm includes

// Class definition
void CreatePackDropLB();

class PackDropLB : public CBase_PackDropLB {
public:
    PackDropLB(const CkLBOptions&);
    PackDropLB(CkMigrateMessage *m);
    void Load_Setup(double total_load);
    void Chare_Setup(int count);
    void PackAck(int pack_id, int from, int psize, bool
        force);
    void RecvAck(int pack_id, int to, bool success);
    void EndStep();
    void First_Barrier();
    void GossipLoadInfo(int, int, int, int [], double []);
    void DoneGossip();
    void Final_Barrier();

private:
    // Private functions
    void InitLB(const CkLBOptions &);
    void Setup();
    void PackSizeDef();
    void LoadBalance();
    void SendLoadInfo();
    void ForcedPackSend(int pack_id, bool force);
    void PackSend(int pack_id = 0, int one_time = 0);
    void Strategy(const DistBaseLB::LDStats* const stats);
    bool QueryBalanceNow(int step) { return true; };
    void CalculateReceivers();
    int FindReceiver();

```

```

// Attributes
bool lb_end;
bool lb_started;
int acks_needed;
int gossip_msg_count;
int kMaxGossipMsgCount;
int kPartialInfoCount;
double avg_load;
int chare_count;
double tries;
size_t total_migrates; // as in dist_lb
int my_chares;
double pack_load;
double my_load;
double threshold;
bool is_receiving;
int done;
int pack_count;
int req_hop;
int info_send_count;
int rec_count;
LBMigrateMsg* msg;
const DistBaseLB::LDStats* my_stats;
std::vector<MigrateInfo*> migrateInfo;
std::list<int> non_receiving_chares;
std::vector<int> receivers;
std::vector<int> pe_no;
std::vector<double> loads;
std::priority_queue<Element, std::deque<Element>>
    local_tasks;
std::unordered_map<int, std::vector<int>> packs;
CProxy_PackDropLB thisProxy;

};

#endif /* PACK_DROP_LB */

```

## A.3 PACKDROPLB.CI

```

module PackDropLB {

extern module DistBaseLB;
initnode void lbinit(void);

group [migratable] PackDropLB : DistBaseLB {
  entry void PackDropLB(const CkLBOptions &);
  entry [expedited] void PackAck(int pack_id, int from, int
    psize, bool force);
  entry [expedited] void RecvAck(int pack_id, int to, bool
    success);
  entry void EndStep();
  entry void GossipLoadInfo(int req_hop, int pe, int n, int
    pe_no[n], double load[n]);
  entry void First_Barrier();
  entry [reductiontarget] void Load_Setup(double
    total_load);
  entry [reductiontarget] void Chare_Setup(int count);
  entry [reductiontarget] void Final_Barrier();
};

};

```



## A.4 PACKDROPLB.C

```

/**
 * Author: Vinicius Freitas
 * contact: vinicius.mct.freitas@gmail.com OR
 *          vinicius.mctf@grad.ufsc.br
 * Produced @ ECL - UFSC
 * Newly developed strategy based on Harshita Menon's
 * implementation of GrapevineLB
 */

#include "PackDropLB.h"
#include "OrderedElement.h"
#include "elements.h"
#include <stdlib.h>
#include <cstring>
#include <algorithm>

CreateLBFunc_Def(PackDropLB, "The_distributed_load_
balancer");

PackDropLB::PackDropLB(CkMigrateMessage *m) :
    CBase_PackDropLB(m) {
}

PackDropLB::PackDropLB(const CkLBOptions &opt) :
    CBase_PackDropLB(opt) {
    lbname = "PackDropLB";
    if (CkMyPe() == 0)
        CkPrintf("[%d]_PackDropLB_created\n", CkMyPe());
    InitLB(opt);
}

void PackDropLB::InitLB(const CkLBOptions &opt) {
    thisProxy = CProxy_PackDropLB(thisgroup);
}

void PackDropLB::Strategy(const DistBaseLB::LDStats* const
stats) {
    if (CkMyPe() == 0) {
        CkPrintf("In_PackDrop_Strategy\n");
    }
    lb_started = false;
    my_stats = stats;
    threshold = 0.05;
    lb_end = false;
    tries = 0;
    info_send_count = 0;
    packs.clear();
    pack_count = 0;
    total_migrates = 0;
}

```

```

acks_needed = 0;
kMaxGossipMsgCount = 2 * CmiLog2(CkNumPes());
kPartialInfoCount = -1;
receivers.clear();
local_tasks = std::priority_queue<Element,
    std::deque<Element>>();
srand((unsigned)CmiWallTimer()*CkMyPe()/CkNumPes());

//local_tasks.reserve(my_stats->n_objs);
my_load = 0;
for (int i = 0; i < my_stats->n_objs; ++i) {
    if (my_stats->objData[i].migratable) {
        local_tasks.emplace(i,
            my_stats->objData[i].wallTime);
        my_load += my_stats->objData[i].wallTime;
    }
}
CkCallback cb(CkReductionTarget(PackDropLB,
    Load_Setup), thisProxy);
contribute(sizeof(double), &my_load,
    CkReduction::sum_double, cb);
}

void PackDropLB::Load_Setup(double total_load) {
    // Calculating the average load of a pe, based on the
    // total system load.
    avg_load = total_load/CkNumPes();
    int chares = my_stats->n_objs;

    CkCallback cb(CkReductionTarget(PackDropLB,
        Chare_Setup), thisProxy);
    contribute(sizeof(int), &chares, CkReduction::sum_int,
        cb);
}

void PackDropLB::Chare_Setup(int count) {
    chare_count = count;
    // packs.reserve(count/CkNumPes());

    // Calc Pack Size
    double avg_task_size =
        (CkNumPes()*avg_load)/chare_count;
    pack_load = avg_task_size*(2 - CkNumPes()/chare_count);

    // Mount Packs
    double ceil = avg_load*(1+threshold);
    double pack_floor = pack_load*(1-threshold);
    double pack_load_now = 0;
    if (my_load > ceil) {
        int pack_id = 0;
        packs[pack_id] = std::vector<int>();
    }
}

```

```

    while (my_load > ceil) {
        Element t = local_tasks.top();
        packs[pack_id].push_back(t.id);
        pack_load_now += t.load;
        my_load -= t.load;
        local_tasks.pop();
        if (pack_load_now > pack_floor) {
            pack_id++;
            packs[pack_id] = std::vector<int>();
        }
    }
    pack_count = packs.size();
    //PackSend();
} else {
    // Scatter my pe_id
    double r_loads[1];
    int r_pe_no[1];
    r_loads[0] = my_load;
    r_pe_no[0] = CkMyPe();
    // Initialize the req_hop of the message to 0
    req_hop = 0;
    GossipLoadInfo(req_hop, CkMyPe(), 1, r_pe_no,
        r_loads);
    //EndStep();
}
if (CkMyPe() == 0) {
    //CkPrintf("Starting QD\n");
    CkCallback cb(CkIndex_PackDropLB::First_Barrier(),
        thisProxy);
    CkStartQD(cb);
}
}

void PackDropLB::First_Barrier() {
    //if (CkMyPe() == 0) CkPrintf("Done gossiping\n");
    LoadBalance();
}

void PackDropLB::LoadBalance() {
    // End of the communication stage
    // if (CkMyPe() == 0)
        CkPrintf("—————barrier—————\n");
    lb_started = true;
    if (packs.size() == 0) {
        msg = new(total_migrates, CkNumPes(), CkNumPes(), 0)
            LBMigrateMsg;
        msg->n_moves = total_migrates;
        //CkPrintf("[%d] I have contributed // UNDERLOADED
            //\n", CkMyPe());
        contribute(CkCallback(CkReductionTarget(PackDropLB,
            Final_Barrier), thisProxy));
    }
    return;
}

```

```

    }
    CalculateReceivers();
    PackSend();
}

void PackDropLB::CalculateReceivers() {
    double pack_ceil = pack_load*(1+threshold);
    double ceil = avg_load*(1+threshold);
    for (size_t i = 0; i < pe_no.size(); ++i) {
        if (loads[i] + pack_ceil < ceil) {
            receivers.push_back(pe_no[i]);
        }
    }
}

int PackDropLB::FindReceiver() {
    int rec;
    if (receivers.size() < CkNumPes()/4) {
        rec = rand()%CkNumPes();
        while (rec == CkMyPe()) {
            rec = rand()%CkNumPes();
        }
    } else {
        rec = receivers[rand()%receivers.size()];
        while (rec == CkMyPe()) {
            rec = receivers[rand()%receivers.size()];
        }
    }
    return rec;
}

void PackDropLB::PackSend(int pack_id, int one_time) {
    tries++;
    if (tries >= 8) {
        EndStep();
        return;
    }
    int idp = pack_id;
    while (idp < packs.size()) {
        // Determine the pack to be sent
        if (packs[idp].empty()) {
            // In this case, the pack meant to be sent is
            // not in this PE anymore
            ++idp;
            continue;
        }
        // Determine a semi_random receiving end, based on
        // gossip
        int rand_rec = FindReceiver();

        // Send pack to receiver, incrementing the acks
        // needed

```

```

        acks_needed++;
        thisProxy[rand_rec].PackAck(idp, CkMyPe(),
            packs[idp].size(), false);

        if (one_time) {
            // If the pack is being resent, the function
            // will stop
            break;
        }
        ++idp;
    }
}

void PackDropLB::PackAck(int id, int from, int psize, bool
force) {
    // Sends back the info, with the appropriate bool
    // related to wheter it's accepted or not
    // If the migration is forced, it will be accepted.
    bool ack = ((my_load + pack_load*psize <
        avg_load*(1+threshold)) || force);
    if (ack) {
        migrates_expected+=psize;
        my_load += pack_load*psize;
    }
    thisProxy[from].RecvAck(id, CkMyPe(), ack);
}

void PackDropLB::RecvAck(int id, int to, bool success) {
    if (success) {
        const std::vector<int> this_pack = packs.at(id);
        for (size_t i = 0; i < this_pack.size(); ++i) {
            int task = this_pack.at(i);
            MigrateInfo* inf = new MigrateInfo();
            inf->obj = my_stats->objData[task].handle;
            inf->from_pe = CkMyPe();
            inf->to_pe = to;
            migrateInfo.push_back(inf);
        }
        packs[id] = std::vector<int>();
        total_migrates++;
        acks_needed--;
        pack_count--;
        if (acks_needed == 0) {
            //CkPrintf("[%d] Creating migration message of
            // size %d:\n", CkMyPe(), total_migrates);
            msg = new(total_migrates, CkNumPes(),
                CkNumPes(), 0) LBMigrateMsg;
            msg->n_moves = total_migrates;
            //CkPrintf("[%d] Creating migration
            // message:\n", CkMyPe());
            for (size_t i = 0; i < total_migrates; ++i) {
                MigrateInfo* inf = (MigrateInfo*)

```

```

        migrateInfo[i];
        msg->moves[i] = *inf;
        delete inf;
    }
    migrateInfo.clear();
    lb_end = true;
    //CkPrintf("[%d] I have contributed\n",
    CkMyPe());
    contribute(CkCallback(CkReductionTarget(PackDropLB,
        Final_Barrier), thisProxy));
}
} else {
    acks_needed--;
    if (tries >= 2) {
        ForcedPackSend(id, true);
    } else {
        ForcedPackSend(id, false);
    }
}
}

void PackDropLB::ForcedPackSend(int id, bool force) {
    //if (force) CkPrintf("[%d] Forcelly balancing load\n",
    CkMyPe());
    int rand_rec = FindReceiver();
    //CkPrintf("[%d] Chosen receiver was %d\n", CkMyPe(),
    rand_rec);
    tries++;
    acks_needed++;
    thisProxy[rand_rec].PackAck(id, CkMyPe(),
    packs.at(id).size(), force);
}

void PackDropLB::EndStep() {
    if (total_migrates < pack_count && tries < 8) {
        CkPrintf("[%d]_Gotta_migrate_more:%d\n", CkMyPe(),
        tries);
        PackSend();
    } else {
        msg = new(total_migrates, CkNumPes(), CkNumPes(),
        0) LBMigrateMsg;
        msg->n_moves = total_migrates;
        //CkPrintf("[%d] Creating migration message:\n",
        CkMyPe());
        for (size_t i = 0; i < total_migrates; ++i) {
            MigrateInfo* inf = (MigrateInfo*)
            migrateInfo[i];
            //std::memcpy(msg->moves+i, inf,
            sizeof(MigrateInfo));
            msg->moves[i] = *inf;
            //CkPrintf(":%d:", inf->from_pe);
            delete inf;
        }
    }
}

```

```

    }
    migrateInfo.clear();
    lb_end = true;
    contribute(CkCallback(CkReductionTarget(PackDropLB,
        Final_Barrier), thisProxy));
}
}

void PackDropLB::Final_Barrier() {
    //CkPrintf("[%d] This core has reach the end of
    program\n", CkMyPe());
    ProcessMigrationDecision(msg);
}

/*
 * Gossip load information between peers. Receive the gossip
 * message.
 */
void PackDropLB::GossipLoadInfo(int req_h, int from_pe, int
    n,
    int remote_pe_no[], double remote_loads[]) {
    // Placeholder temp vectors for the sorted pe and their
    load
    std::vector<int> p_no;
    std::vector<double> l;

    int i = 0;
    int j = 0;
    int m = pe_no.size();

    // Merge (using merge sort) information received with the
    information at hand
    // Since the initial list is sorted, the merging is
    linear in the size of the
    // list.
    while (i < m && j < n) {
        if (pe_no[i] < remote_pe_no[j]) {
            p_no.push_back(pe_no[i]);
            l.push_back(loads[i]);
            i++;
        } else {
            p_no.push_back(remote_pe_no[j]);
            l.push_back(remote_loads[j]);
            if (pe_no[i] == remote_pe_no[j]) {
                i++;
            }
            j++;
        }
    }
}

if (i == m && j != n) {
    while (j < n) {

```

```

    p_no.push_back(remote_pe_no[j]);
    l.push_back(remote_loads[j]);
    j++;
}
else if (j == n && i != m) {
    while (i < m) {
        p_no.push_back(pe_no[i]);
        l.push_back(loads[i]);
        i++;
    }
}

// After the merge sort, swap. Now pe_no and loads have
// updated information
pe_no.swap(p_no);
loads.swap(l);
req_hop = req_h + 1;

SendLoadInfo();
}

/*
 * Construct the gossip message and send to peers
 */
void PackDropLB::SendLoadInfo() {
    // TODO: Keep it 0.8*log
    // This PE has already sent the Randimum set threshold
    // for gossip messages.
    // Hence don't send out any more messages. This is to
    // prevent flooding.
    if (gossip_msg_count > kMaxGossipMsgCount) {
        return;
    }

    // Pick two random neighbors to send the message to
    int rand_nbor1;
    int rand_nbor2 = -1;
    do {
        rand_nbor1 = rand() % CkNumPes();
    } while (rand_nbor1 == CkMyPe());
    // Pick the second neighbor which is not the same as the
    // first one.
    do {
        rand_nbor2 = rand() % CkNumPes();
    } while ((rand_nbor2 == CkMyPe()) || (rand_nbor2 ==
        rand_nbor1));

    // kPartialInfoCount indicates how much information is
    // send in gossip. If it
    // is set to -1, it means use all the information
    // available.
    int info_count = (kPartialInfoCount >= 0) ?

```



```
    kPartialInfoCount : pe_no.size();
    int* p = new int[info_count];
    double* l = new double[info_count];
    for (int i = 0; i < info_count; i++) {
        p[i] = pe_no[i];
        l[i] = loads[i];
    }

    thisProxy[rand_nbor1].GossipLoadInfo(req_hop, CkMyPe(),
        info_count, p, l);
    thisProxy[rand_nbor2].GossipLoadInfo(req_hop, CkMyPe(),
        info_count, p, l);

    // Increment the outgoing msg count
    gossip_msg_count++;

    delete [] p;
    delete [] l;
}

#include "PackDropLB.def.h"
```



## **APÊNDICE B - Artigo baseado no trabalho de conclusão**



# Balanceamento de carga distribuído: Uma abordagem orientada a pacotes

Vinicius M. C. T. de Freitas<sup>1</sup>

<sup>1</sup>Departamento de Informática e Estatística  
Universidade Federal de Santa Catarina (UFSC) – Florianópolis – SC – Brazil

vinicius.mct.freitas@gmail.com

**Abstract.** *Load imbalance becomes a recurrent problem as parallel platforms grow and with the imprevisibility of computational simulations. Global rescheduling strategies are presented as a partial solution for this matter. In this work, I present a new distributed load balancer, which seeks to use the available parallelism on nowadays largest platforms, while trying to mitigate strategy execution time and lack of information problems. Results show speedups of up to 1.33 and 1.05 when compared to executions with no rescheduling and to the state of the art, respectively.*

**Resumo.** *O desbalanceamento de carga torna-se um problema recorrente com o crescimento das plataformas paralelas e com a imprevisibilidade das simulações computacionais. Estratégias de reescalonamento global são apresentadas como parte da solução para este problema. Neste trabalho, proponho um novo balanceador de carga distribuído, que busca usar do paralelismo disponível nas maiores plataformas da atualidade, enquanto tenta mitigar problemas atrelados ao tempo de execução da estratégia e falta de informação global. Resultados mostram ganhos com speedups observados de até 1,33 e 1,05 quando comparados a execuções sem balanceamento e com o estado da arte, respectivamente.*

## 1. Introdução

Com o avanço das aplicações científicas e industriais, estas necessitam de cada vez mais tempo de processamento e poder computacional; isso acontece por muitas delas serem simulações muito detalhadas ou complexas de carga imprevisível, como simulações de ondas sísmicas [Boito et al. 2017] e de dinâmica molecular [Mei et al. 2011].

Para prover tais recursos às aplicações, são usados sistemas computacionais paralelos, capazes de prover uma redução do tempo de execução com um aumento do número de unidades de processamento (PUs). Aplicações de memória distribuída usualmente são executadas em diversas máquinas comunicando-se em rede. Isso torna a infraestrutura mais barata, em contra partida tornando a sua programação mais complexa [Flynn 1972]. Aplicações de memória distribuída sofrem de uma série de problemas, como maior complexidade de desenvolvimento, comunicação como gargalo, dependência e distribuição de dados, sincronização, áreas críticas e desbalanceamento de carga [Pilla 2014].

A distribuição de trabalho através de recursos computacionais é um grande problema para a computação paralela [Pearce et al. 2012]. Como algumas aplicações tendem

a ter mudanças dinâmicas em seu comportamento, mesmo um mapeamento inicial aparentemente ótimo pode chegar a um estado **desbalanceado**.

Uma possível solução para esse tipo de problema é o uso de balanceadores de carga dinâmicos (LB). Esses balanceadores devem usar um estado do sistema como base para fazer um remapeamento das tarefas entre as PUs, buscando uma distribuição mais equilibrada da carga. Estes, contudo, devem executar de forma ágil para que não tornem o tempo total de execução da aplicação mais lento [Deveci et al. 2015].

Este trabalho apresenta **PackDrop**, uma nova estratégia de balanceamento de carga distribuído. Ela se baseia no uso de informação local para criar e migrar grupos de tarefas, buscando menores custos de comunicação e reduzindo o número de migrações, enquanto torna a distribuição de carga mais homogênea e aproveita o paralelismo dos sistemas.

## 2. Trabalhos Correlatos

O reescalonamento global de tarefas pode ser definido em uma Função de Mapeamento (1), onde  $\mathcal{T}$  representa o conjunto de tarefas da aplicação e  $\mathcal{P}$  o conjunto de PUs da plataforma [Pilla 2014], criando um mapa, ou estado, do sistema. Algoritmos de balanceamento de carga manipulam estes mapas, criando novas relações entre PUs e tarefas.

$$\mathcal{M} : \mathcal{T} \rightarrow \mathcal{P} \quad (1)$$

Existem diversos fatores a se considerar ao desenvolver uma estratégia de balanceamento de carga. Este trabalho foi comparado com duas estratégias centralizadas, uma hierárquica e uma distribuída, detalhadas a seguir:

- **Greedy**: uma estratégia de balanceamento centralizada. Um algoritmo guloso que busca homogeneizar a carga, independente do número de migrações.
- **Refine**: uma estratégia centralizada baseada em refinamento. Tenta migrar o mínimo de tarefas possíveis dos núcleos sobrecarregados para os subcarregados.
- **Hybrid**: uma abordagem hierárquica. Aplica em granularidade grossa, entre grupos de PUs, o *Refine* e em granularidade fina, entre PUs de um grupo, o *Greedy* [Zheng et al. 2010].
- **Grapevine**: uma estratégia distribuída. Usa um método probabilístico para tentar parear PUs sobrecarregados e subcarregados [Menon and Kalé 2013].

### 2.1. Charm++

O sistema paralelo Charm++ é escalável e usado em diversas aplicações científicas, além de apresentar um *framework* de balanceamento de carga independente de aplicação [Acun et al. 2016]. Este sistema apresenta uma série de *benchmarks* e implementações dos balanceadores de carga supracitados.

Sendo assim, este sistema foi escolhido para a implementação do *PackDrop* e sua avaliação de desempenho, apresentada posteriormente na Seção 4.

## 3. Estratégia Proposta: *PackDrop*

A estratégia proposta é o **PackDrop**. Ela se baseia em criar conjuntos de tarefas para migração, diminuindo a quantidade total de mensagens trocadas entre os PUs para realizar

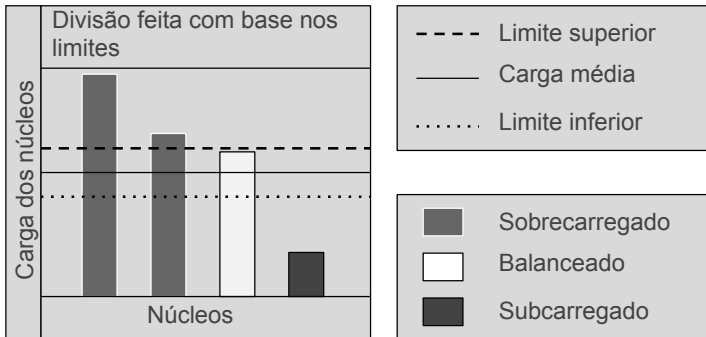
o balanceamento. Diversas ferramentas e métodos usados em sua implementação serão apresentados em detalhes a seguir. Por fim, o algoritmo será reapresentado como um todo, assim como seu fluxo de execução, na Seção 3.4.

### 3.1. Criação de Pacotes

Definir pacotes de tamanhos fixo é perigoso, uma vez que a carga das tarefas depende da aplicação, da plataforma e dos parâmetros de execução. Sendo assim, uma fórmula variável pode ser definida para calcular este tamanho. O tamanho de cada pacote,  $ps$ , deveria ser baseado no tamanho médio de uma tarefa no sistema,  $\bar{ts}$ , no número de núcleos disponíveis,  $nC$ , e no número total de tarefas (em Charm++,  $chares$ ) no sistema,  $nT$ . As proporções estão descritas na Equação 2.

$$ps = \bar{ts} \times \left(2 - \frac{nC}{nT}\right) \quad (2)$$

Essa proporção faz com que a quantidade de tarefas em um pacote, baseado na carga média, seja algo próximo a 2. Ainda permite-se uma variação dentro do tamanho do pacote, uma vez que não é possível atingir sempre o tamanho exato estimado. A variação usada foi de 5% para mais e para menos, criando dois limites, superior e inferior, como evidenciado na Figura 1. A taxa de precisão no balanceamento pode ser mudada dependendo da aplicação alvo, uma vez que isso depende muito da característica das cargas de cada aplicação (aplicações com muitas cargas leves irão se beneficiar mais de um balanceamento de alta precisão nesse caso).



**Figura 1. Um sistema desbalanceado, com núcleos divididos em grupos (Balanceado, Subcarregado e Sobrecarregado) por limites relacionados à carga média.**

No Algoritmo 1, o procedimento é detalhado.  $\mathcal{T}$  é o conjunto de tarefas de um núcleo,  $carga$  é a carga total do núcleo que está criando os pacotes,  $limite$  é o limiar de sobrecarga e  $Pacotes$  é o conjunto de pacotes gerado. É importante notar que os elementos são retirados da lista de tarefas de um PU sempre na mesma ordem, priorizando as tarefas de menor carga, evitando criar pacotes que tornariam núcleos subcarregados em núcleos sobrecarregados.

---

**Algoritmo 1:** Criação de pacotes

---

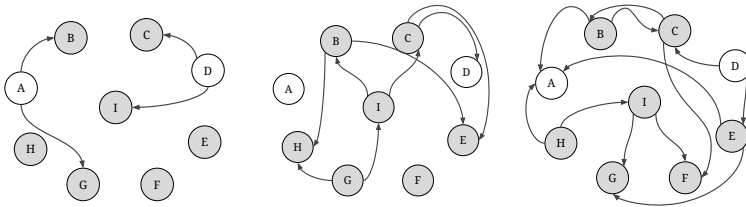
**Entrada:**  $ps, \mathcal{T}, carga, limite$ **Saída:** Pacotes

```
1  $P \leftarrow \emptyset, Pacotes \leftarrow \emptyset$ 
2 enquanto  $carga > limite$  faça
3    $t \leftarrow a \in \mathcal{T} \mid a \text{ é o menor elemento de } \mathcal{T}$ 
4    $\mathcal{T} \leftarrow \mathcal{T} \setminus \{t\}$ 
5    $P \leftarrow P \cup \{t\}$ 
6    $carga \leftarrow carga - t$ 
7   se ( $\text{tamanho de } P$ )  $> ps$  então
8      $Pacotes \leftarrow Pacotes \cup P$ 
9      $P \leftarrow \emptyset$ 
10  fim
11 fim
12  $Pacotes \leftarrow Pacotes \cup P$ 
```

---

### 3.2. Protocolo Gossip

A propagação de informação com Protocolo *Gossip* [Demers et al. 1987] funciona como uma rede de fofocas. Criando uma abstração para um grafo, como visto na Figura 2, podemos dizer que os membros da rede são os vértices e que as arestas são a propagação de informação. O funcionamento se baseia em nós que possuem uma informação que deve ser propagada passando essa informação para um número  $t$  de nós, chamamos essa variável de *throughput*. Cada um desses nós unirá a informação obtida à sua e enviará esta para mais  $t$  nós do grafo. Isso se repete até que se atinja um limite predeterminado de iterações da propagação. O número de iterações varia de acordo com o número de vértices no grafo, a Figura 2 mostra três iterações do protocolo.



**Figura 2.** Três passos de execução do Protocolo *Gossip*, com *throughput* de 2. À esquerda o primeiro passo, no meio, o segundo e à direita, o terceiro.

Por ser uma estratégia de propagação que funciona bem em sistemas distribuídos, já mostrando um desempenho escalável no *GrapevineLB*, decidiu-se adotá-la na implementação do *PackDropLB*.

### 3.3. Envio de Pacotes

Na etapa de **Envio de Pacotes**, há quatro momentos envolvidos. O primeiro é descrito no Algoritmo 2. Nesta etapa, os PUs sobrecarregados do sistema irão enviar todos os seus



pacotes aos possíveis alvos determinados durante a etapa de propagação de informação, tendo uma saída no formato de um conjunto de pares Pacote/Alvo ( $R$ ), com o qual serão executadas as requisições que levam à segunda etapa.

---

**Algoritmo 2:** Envio de pacotes

---

**Entrada:**  $Pacotes, Alvos$   
**Saída:** relação entre pacotes e alvos esperando confirmação  $R$

```

1  $R \leftarrow \emptyset$ 
2 enquanto  $Pacotes \neq \emptyset$  faça
3    $pacote \leftarrow p \mid p \in Pacotes$ 
4    $alvo \leftarrow a \mid a \in Alvos \wedge a$  é um elemento aleatório de  $Alvos$ 
5    $Pacotes \leftarrow Pacotes \setminus \{pacote\}$ 
6    $R \leftarrow R \cup \{(pacote, alvo)\}$ 
7 fim

```

---

Na segunda etapa, os PUs que estão recebendo tarefas devem decidir se a **carga de um novo pacote** ( $ps$ ) quando somada a sua **carga local** ( $carga$ ) irá se manter abaixo do *limiar de sobrecarga*, condição necessária para aceitar o pacote. Tarefas também podem ser enviadas de forma forçada, quando não estão sendo enviadas pela primeira vez, para acelerar a convergência do balanceador de carga.

Na terceira etapa, descrita no Algoritmo 3, os nós subcarregados enviam mensagens de volta àqueles que tentaram lhes enviar tarefas. Essas mensagens são no formato de uma **tripla**  $alvo/pacote/sucesso$  (APS), onde alvo é quem enviou a mensagem, pacote é o conjunto de tarefas que ele deve receber e sucesso é um booleano que indica se a migração foi bem sucedida. A saída dessa terceira etapa é o novo mapeamento ( $\mathcal{M}'$ ) das tarefas que originalmente estavam no nó sobrecarregado (*local*).

---

**Algoritmo 3:** Remapeamento das tarefas

---

**Entrada:**  $APSSs$ , mapeamento original  $\mathcal{M}$ , *local*  
**Saída:** novo mapeamento  $\mathcal{M}'$ , conjunto de novas tentativas  $NT$

```

1  $\mathcal{M}' \leftarrow \mathcal{M}$ 
2 enquanto  $APSSs \neq \emptyset$  faça
3    $(a, p, s) \leftarrow e \mid e \in APSSs$ 
4    $APSSs \leftarrow APSSs \setminus \{(a, p, s)\}$ 
5   se  $s$  então
6      $\mathcal{M}' \leftarrow \mathcal{M}' \setminus \{(local, p)\}$ 
7      $\mathcal{M}' \leftarrow \mathcal{M}' \cup \{(a, p)\}$ 
8   senão
9      $NT \leftarrow p$ 
10  fim
11 fim

```

---

A última etapa ocorre apenas no caso de haverem migrações que falhem em auxiliar um PE subcarregado na segunda etapa, tendo um valor de *sucesso* falso na terceira. Nesse caso, os pacotes que foram parar no conjunto  $NT$  terão mais tentativas de migração,

até que, atingido um certo limite ( $lt$ ), ele terá uma migração forçada para o próximo alvo escolhido, ou seja, ele não pode retornar um valor de  $s$  falso.

### 3.4. Visão Geral

A Figura 3 mostra o fluxo de funcionamento do algoritmo principal, descrito a seguir:

1. Os dois primeiros passos, executados por todos os PUs, são realizar duas reduções, a primeira servindo para agregar a informação sobre a carga média e a segunda sobre o número de tarefas no sistema;
2. Em seguida, cada um dos núcleos se divide entre sobrecarregados e subcarregados;
3. Aqueles que são sobrecarregados irão começar a criar seus pacotes para transferência, enquanto os demais irão começar a propagar seus identificadores pela rede através de um protocolo *Gossip*, com o objetivo de informar os sobrecarregados a quem eles devem enviar suas tarefas;
4. Todos os processadores são sincronizados para se iniciar a troca de tarefas;
5. Os PUs sobrecarregados tentam enviar seus pacotes aos subcarregados.

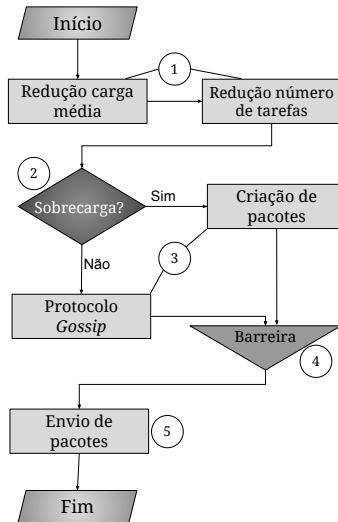


Figura 3. Fluxo de funcionamento do *PackDrop*, seguido individualmente por cada PE.

### 4. Avaliação de Desempenho

O principal ganho esperado com a nova abordagem aqui apresentada é a **diminuição no volume de comunicação**, ao custo de um balanceamento menos preciso. Isso acontece pois um núcleo remoto tem apenas uma estimativa da carga que irá receber quando a aceita, contudo isso garante um menor número de troca de mensagens.

A seguir serão apresentados os *benchmarks* utilizados para a avaliação de desempenho do *PackDrop* e o ambiente de testes.

#### 4.1. Benchmarks

Dentre os *benchmarks* utilizados para teste e avaliação, dois são perfis paralelos sintéticos (*lb test* e *Stencil 3D*)<sup>1</sup> e dois são *mini-apps* de aplicações de mundo real (LULESH<sup>2</sup> e *LeanMD*<sup>3</sup>).

1. **lb test**: Um perfil paralelo sintético, criado para testar balanceadores de carga. Foi executado com uma topologia de comunicação de anel, por 150 iterações, com balanceamento de carga a cada 40 iterações e carga das tarefas variando entre 300ms e 90000ms. Além disso, quanto a quantidade de tarefas ele foi separado em dois conjuntos. O primeiro (*med*) com 18990 tarefas e o segundo (*big*) 36990 tarefas. Estes parâmetros foram escolhidos para testar diferentes escalas de tamanho de aplicação.
2. **Stencil 3D**: Um perfil paralelo sintético seguindo o padrão estêncil tridimensional. Esse *benchmark* cria uma malha tridimensional de bordas quentes e centro frio (temperaturas de 255 e 0, respectivamente), e realiza o cálculo de propagação desta. Nesse caso, escolheu-se o tamanho de bloco 540 e de *array* 12, estes tamanhos foram definidos para gerar um tamanho relevante o suficiente para o balanceamento se justificar; além disso os tamanhos não são potências de 2, uma vez que isso potencializa o desbalanceamento. Ele foi processado ao longo de 100 iterações, com balanceamento de carga realizado a cada 15 iterações.
3. **LULESH**: Um modelo para simulação de hidrodinâmica. Usado para resolver uma simples *Sedov blast wave* [Sedov 1946] com respostas analíticas, contudo, esta aplicação serve como base para implementação de outros problemas desta área [Karlin et al. 2012]. O LULESH cria uma malha tridimensional para realizar suas simulações e tem sua paralelização feita num padrão estêncil. Os parâmetros para dimensões de bloco e elementos do LULESH foram de 140 e 7. Os parâmetros desta aplicação seguem as mesmas justificativas do *Stencil 3D*.
4. **LeanMD**: Um modelo simulação para dinâmica molecular. Simula o comportamento de átomos baseados no *potencial Lennard-Jones*, que descreve a interação entre moléculas ou átomos. Sua computação é paralelizada com a decomposição espacial e de forças, realizando cálculos de força entre dois átomos a uma certa distância [Pilla 2014]. *LeanMD* foi executado por 500 iterações, com balanceamento de carga iniciando na iteração de número 20 e ocorrendo de novo a cada 100 iterações. As dimensões utilizadas para os conjuntos de células foram de  $10 \times 15 \times 10$  (*med*) e de  $15 \times 20 \times 15$  (*big*), respectivamente. O primeiro conjunto de parâmetros gera um experimento mais pesado, com mais tarefas de carga maior, enquanto o segundo é mais leve, gerando menos tarefas de carga menor.

A Tabela 1 mostra quantas execuções de cada *benchmark* foram realizadas para cada balanceador na colheita dos resultados.

---

<sup>1</sup>Disponibilizados com o Charm++ através de: <http://charmplusplus.org/download/>

<sup>2</sup>Disponível em: <https://codesign.llnl.gov/lulesh.php>

<sup>3</sup>Disponível em: <http://charmplusplus.org/miniApps/>

<i>Benchmark</i>	Número de execuções
<i>lb test med</i>	12
<i>lb test big</i>	18
<i>Stencil 3D</i>	12
<i>LULESH</i>	20
<i>LeanMD med</i>	14
<i>LeanMD big</i>	5

**Tabela 1. Número de execuções dos experimentos para averiguação dos resultados com *benchmarks* menores (M) e maiores (G).**

## 4.2. Ambiente de Testes

Todos os testes preliminares foram rodados em 16 nós do *cluster Genepi*<sup>4</sup>, cada um contando com 2 processadores *Intel Xeon E5420 QC @ 2.5GHz* com 4 *cores* cada, totalizando **128 PUs**. Estas máquinas estão equipadas com 8GB de memória RAM e 160GB de armazenamento SATA HDD. Os nós estão interconectados em um *switch* utilizando a tecnologia *InfiniBand 20G*, sendo que as placas contidas em cada um são *Mellanox ConnectX IB 4X DDR MT26418*. O sistema operacional utilizado foi o *Ubuntu 14.04* em versão *min*, a versão do *Charm++* adotada foi a 6.7.0 e o *GCC* foi utilizado na versão 5.4.0.

## 4.3. Resultados

Observando todos os resultados, notamos que a estratégia apresentada (*PackDrop*) é capaz de entregar um ganho de desempenho considerável nos casos observados, principalmente quando comparado à estratégia mais semelhante a ele, o *Grapevine*.

A Figura 4 mostra os *speedups* (Equação 3,  $\mathcal{S} - 1$ ) relativos a execuções sem balanceamento de carga. Esses experimentos foram todos realizados utilizando 128 PUs, distribuídas em 16 nós diferentes e mostram que o *speedup* atingido pela nova abordagem é mais eficiente que o *Grapevine*, comparando-se mais ao *Refine*.

$$\mathcal{S} = \frac{\text{tempo\_original}}{\text{tempo\_acelerado}} \quad (3)$$

Tanto o *Stencil 3D* como o *LULESH* mostraram resultados insatisfatórios para todos os balanceadores de carga. A migração das tarefas acaba gerando perda de desempenho devido a forma como o paralelismo delas é realizado (com comunicação entre cada iteração e *ghost zones* de tamanho 1).

Apesar dos resultados positivos são necessárias mais investigações, especialmente testes de escalabilidade, para garantir que mesmo em ambientes maiores, o *PackDrop* continua sendo tão eficiente quanto outras estratégias distribuídas. Balanceadores distribuídos destacam-se em comparação aos centralizados e hierárquicos conforme o tamanho do sistema cresce [Menon and Kalé 2013].

<sup>4</sup>Experimentos apresentados neste trabalho foram realizados no Grid5000, mantido pelo grupo de propósitos científicos hospedado pela Inria e incluindo CNRS, RENATER e diversas Universidades e outras organizações (ver <https://www.grid5000.fr>).

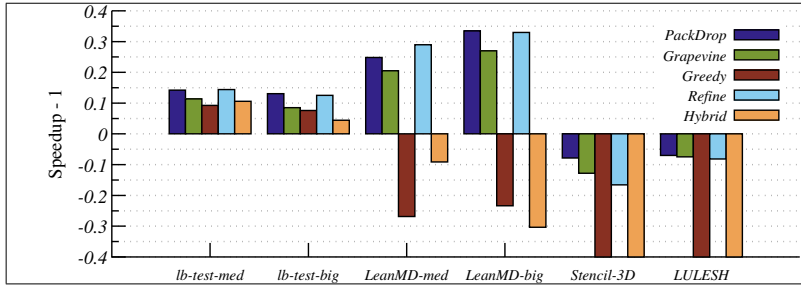


Figura 4. Speedups ( $S - 1$ ) de todos os *benchmarks* e aplicações testadas com cada estratégia de balanceamento de carga.

## 5. Conclusão

Este trabalho apresentou *PackDropLB*, uma nova estratégia de balanceamento de carga, seguindo uma nova abordagem orientada à pacotes. O principal diferencial da estratégia em relação a outras abordagens distribuídas é a criação de pacotes, que fazem com que não seja necessário trocar informações sobre a carga das tarefas antes das migrações, assim como possibilita a migração de múltiplas tarefas de uma única vez.

Essa nova abordagem mostrou resultados promissores, tendo ganho de desempenho em todos os casos de teste experimentados em relação à outra estratégia distribuída observada, o *Grapevine*. Contudo, os ganhos não puderam ser completamente explorados neste trabalho por restrições da plataforma de testes, muito pequena para mostrar o ganho real de uma estratégia de reescalonamento distribuída.

### 5.1. Trabalhos Futuros

Os testes realizados neste trabalho ainda não tem escala grande o suficiente para mostrar o desempenho das estratégias distribuídas. Uma continuação importante para este trabalho é a realização de mais experimentos em ambientes maiores, onde existe a possibilidade para ainda maior ganho de desempenho, principalmente quando comparados aos algoritmos centralizados.

Uma nova estratégia de escalonamento global ainda pode seguir a abordagem orientada a pacotes e usar princípios de roubo de tarefas [Janjic and Hammond 2013] para fazer o remapeamento de trabalho. Isso tira a responsabilidade de entregar pacotes dos PUs sobrecarregados e a passa para os subcarregados, que devem pedi-las. Essa abordagem pode ter ainda mais ganhos, uma vez que não precisa de um protocolo de propagação de informação para iniciar as trocas de tarefas.

## Referências

- Acun, B., Langer, A., Meneses, E., Menon, H., Sarood, O., Totoni, E., and Kalé, L. V. (2016). Power, reliability, and performance: One system to rule them all. *Computer*, 49(10):30–37.
- Boito, F. Z., Bez, J. L., Dupros, F., Dantas, M. A. R., Navaux, P. O. A., and Aochi, H. (2017). High performance I/O for seismic wave propagation simulations. In *2017*

- 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 31–38.
- Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. (1987). Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA. ACM.
- Deveci, M., Kaya, K., Uçar, B., and Çatalyürek, V. (2015). Fast and high quality topology-aware task mapping. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 197–206.
- Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960.
- Janjic, V. and Hammond, K. (2013). How to be a successful thief. In Wolf, F., Mohr, B., and an Mey, D., editors, *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, pages 114–125, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Karlin, I., Keasler, J., Haque, R., Wand, F., McGraw, J., and Cohen, J. (2012). Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.
- Mei, C., Sun, Y., Zheng, G., Bohm, E. J., Kalé, L. V., C. Phillips, J., and Harrison, C. (2011). Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime. In *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA.
- Menon, H. and Kalé, L. (2013). A distributed dynamic load balancer for iterative applications. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 15:1–15:11, New York, NY, USA. ACM.
- Pearce, O., Gamblin, T., de Supinski, B. R., Schulz, M., and Amato, N. M. (2012). Quantifying the effectiveness of load balance algorithms. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 185–194, New York, NY, USA. ACM.
- Pilla, L. L. (2014). *Topology-Aware Load Balancing for Performance Portability over Parallel High Performance Systems*. PhD thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, Rio Grande do Sul.
- Sedov, I. L. (1946). Propagation of strong shock waves. pages 241–250.
- Zheng, G., Meneses, E., Bhatele, A., and Kale, L. V. (2010). Hierarchical load balancing for charm++ applications on large supercomputers. In *2010 39th International Conference on Parallel Processing Workshops*, pages 436–444.