Scalable Load Balancing

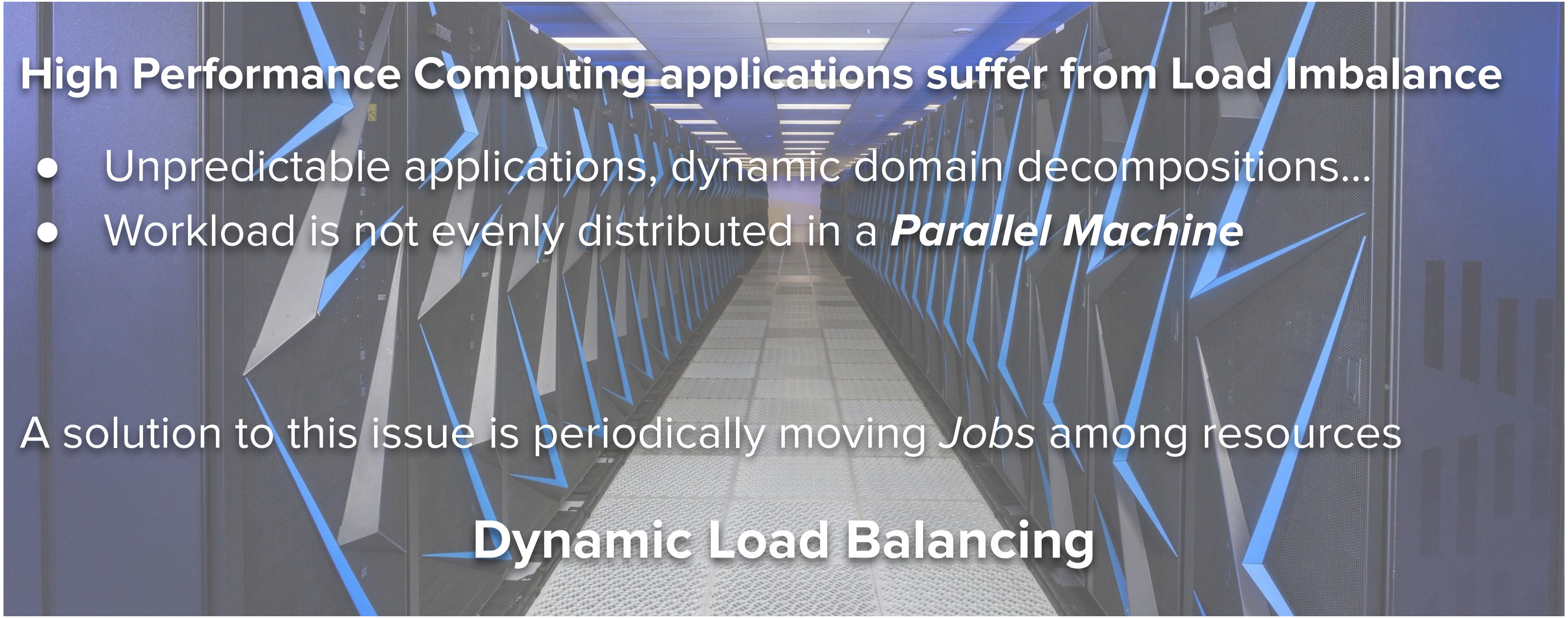
Distributed Algorithms & the Packing Model

Vinicius Freitas, Laércio L. Pilla, Johanne Cohen

vfreitas@lri.fr ou vinicius.mct.freitas@gmail.com

ECL LAPESD CNPq CAPES Programa de Pós-Graduação em Ciência da Computação UFSC UFSC cnrs UNIVERSITÉ PARIS SUD université PARIS-SACLAY

Introduction



High Performance Computing applications suffer from Load Imbalance

- Unpredictable applications, dynamic domain decompositions…
- Workload is not evenly distributed in a ***Parallel Machine***

A solution to this issue is periodically moving *Jobs* among resources

Dynamic Load Balancing

Sierra supercomputer in the Lawrence Livermore National Lab (US)

Introduction

Scalability is important!

As machines and applications **grow larger**, load balancing solutions must be able to **scale** along.

IBM *Summit* supercomputer at the Oak Ridge National Lab (US)

Presentation Agenda

Scalable Load Balancing: Distributed Algorithms & the Packing Model

1. Introduction
2. **Background (Algorithms & HPC)**
3. The Packing Model
4. Load Balancing Algorithms
5. Experimental Evaluation
6. Work in Progress

Scheduling for Parallel Machines

Let **M** be the set of machines available in a *Parallel Machine*.

Let **J** be the *ordered list* of jobs to be computed in the *Parallel Machine*.

Assume that each job j in J, is mapped to some machine M_i in M.

Scheduling for Parallel Machines

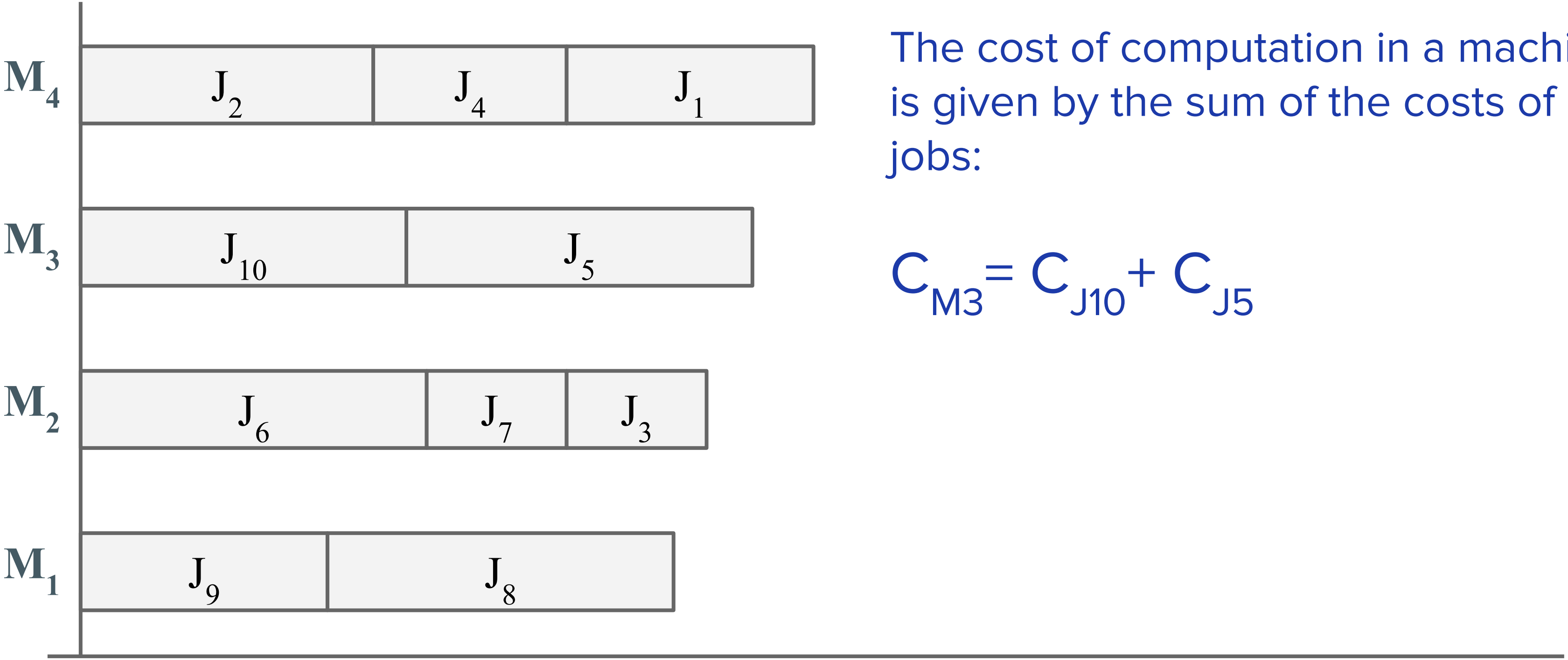
The cost of a job *j* is given by the time takes on CPU, noted by C_j.

The cost of computing all jobs in a machine M_i, is given by C_{Mi}.

Alas, the overall cost of a parallel computation is the maximum among all machines:

$$C_{max} = max(C_{Mi} \text{ for each } M_i \text{ in } M)$$

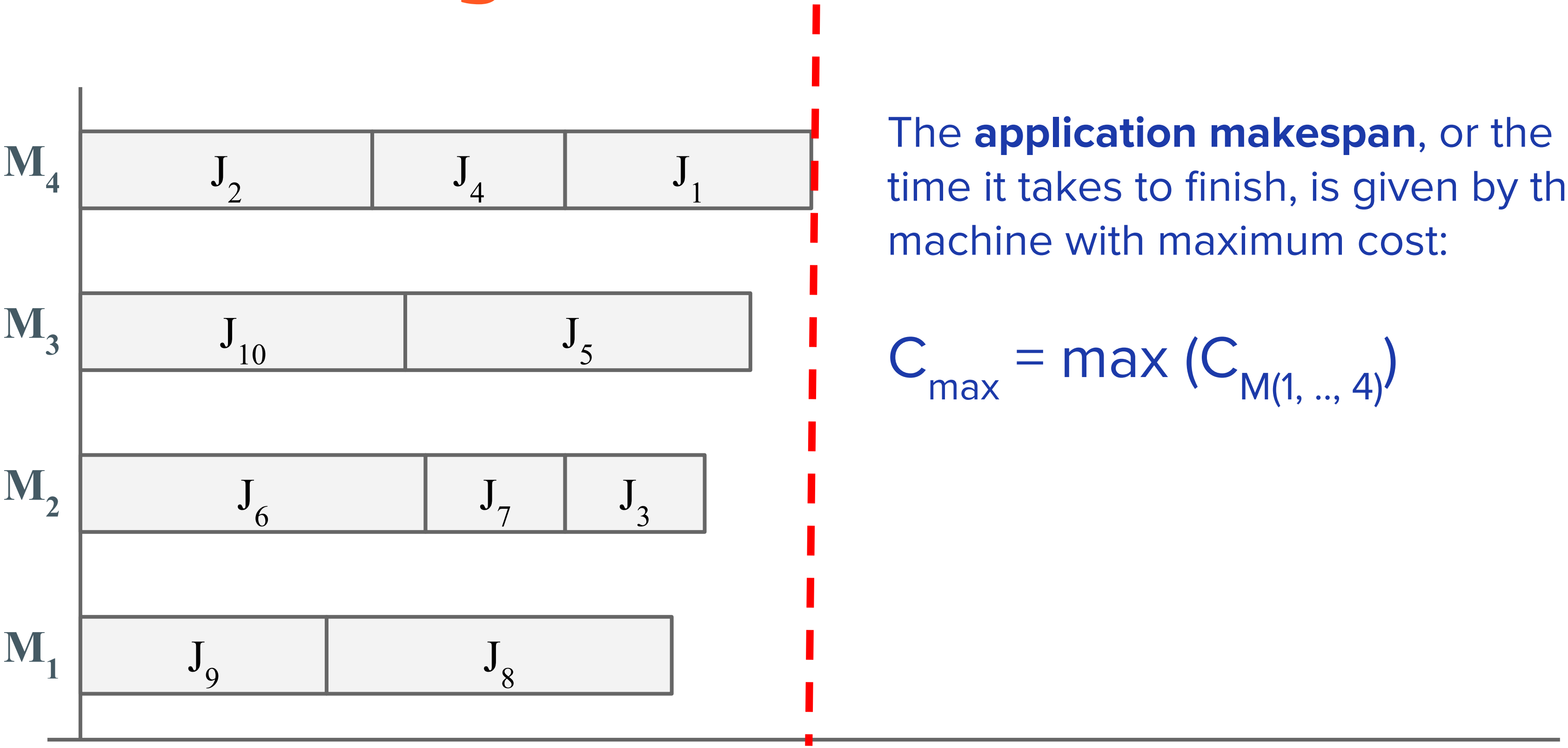
Scheduling for Parallel Machines



The cost of computation in a machine is given by the sum of the costs of its jobs:

$$C_{M3} = C_{J10} + C_{J5}$$

Scheduling for Parallel Machines



The **application makespan**, or the time it takes to finish, is given by the machine with maximum cost:

$$C_{max} = max\ (C_{M(1,\ ..,\ 4)})$$

Premises

The objective is to **minimize application makespan** (the List Scheduling problem):

$P \mid\mid C_{max}$: The burden of computation is divided, but the machine that finishes its work last defines the *overall cost of computation*.

Premises for Distributed Load Balancing

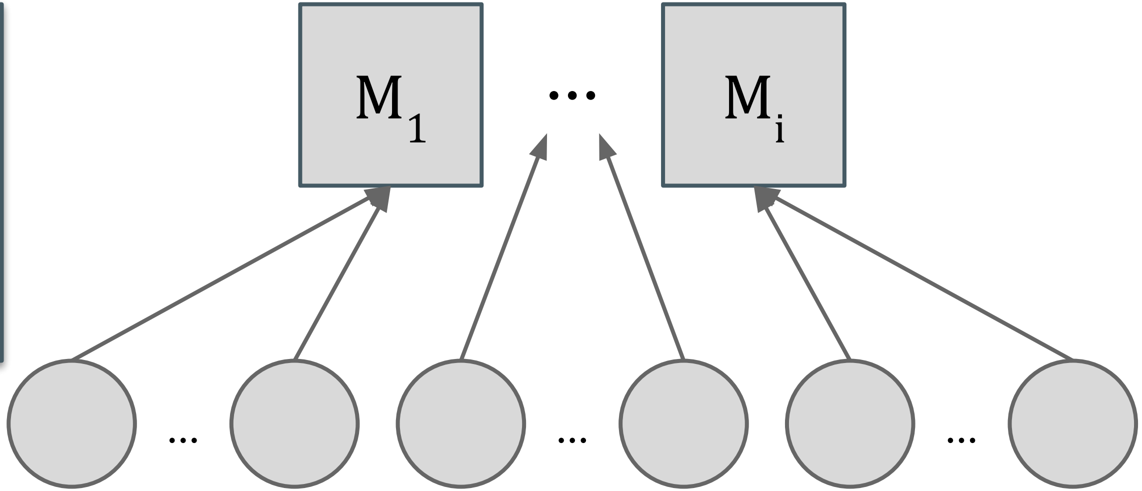
- We assume that jobs are already allocated to machines;
- We want to remap the jobs that make the machine *overloaded*;
- Choosing what jobs to migrate and where to should be done *in parallel*;
- We need fast and *useful* decisions.
  - They don't have to be greedy.

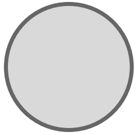
Scheduling notation from: BRUCKER, P., "Scheduling Algorithms", 5th Ed., Springer.

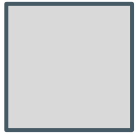
Distributed Load Balancing

Assume that each job is mapped to some machine.

Each machine decides which jobs they want to move.

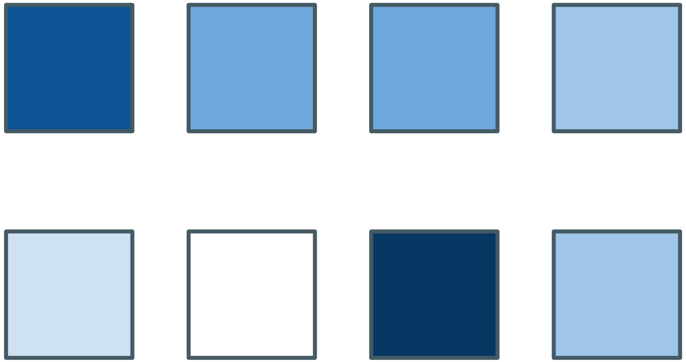


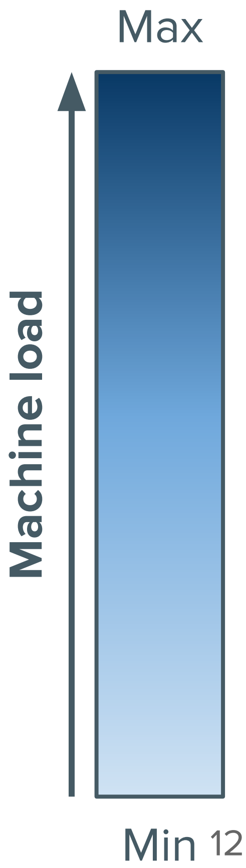
 Jobs

 Machines

Distributed Scheduling

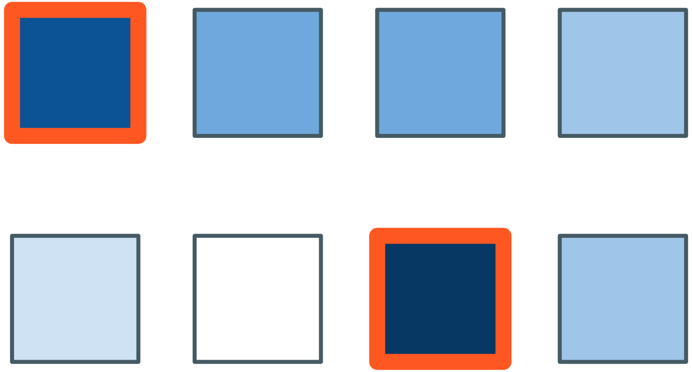
The load of a machine is given by the sum of the loads of its jobs.

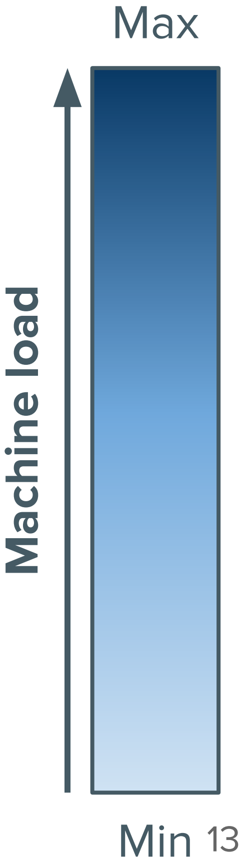




Distributed Scheduling

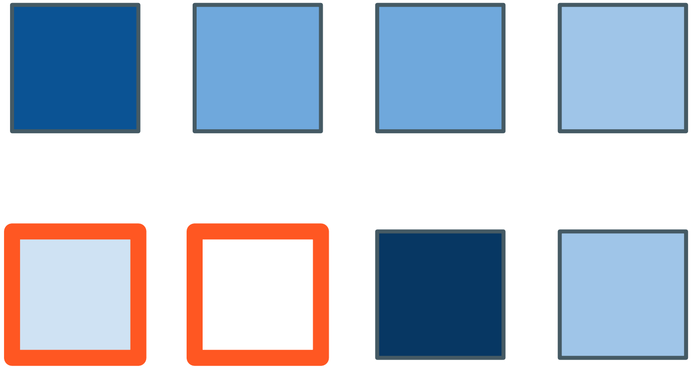
Overloaded machines will have stimulus to **migrate** their jobs!

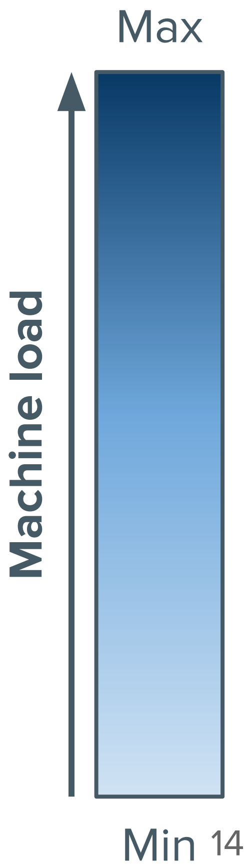




Distributed Scheduling

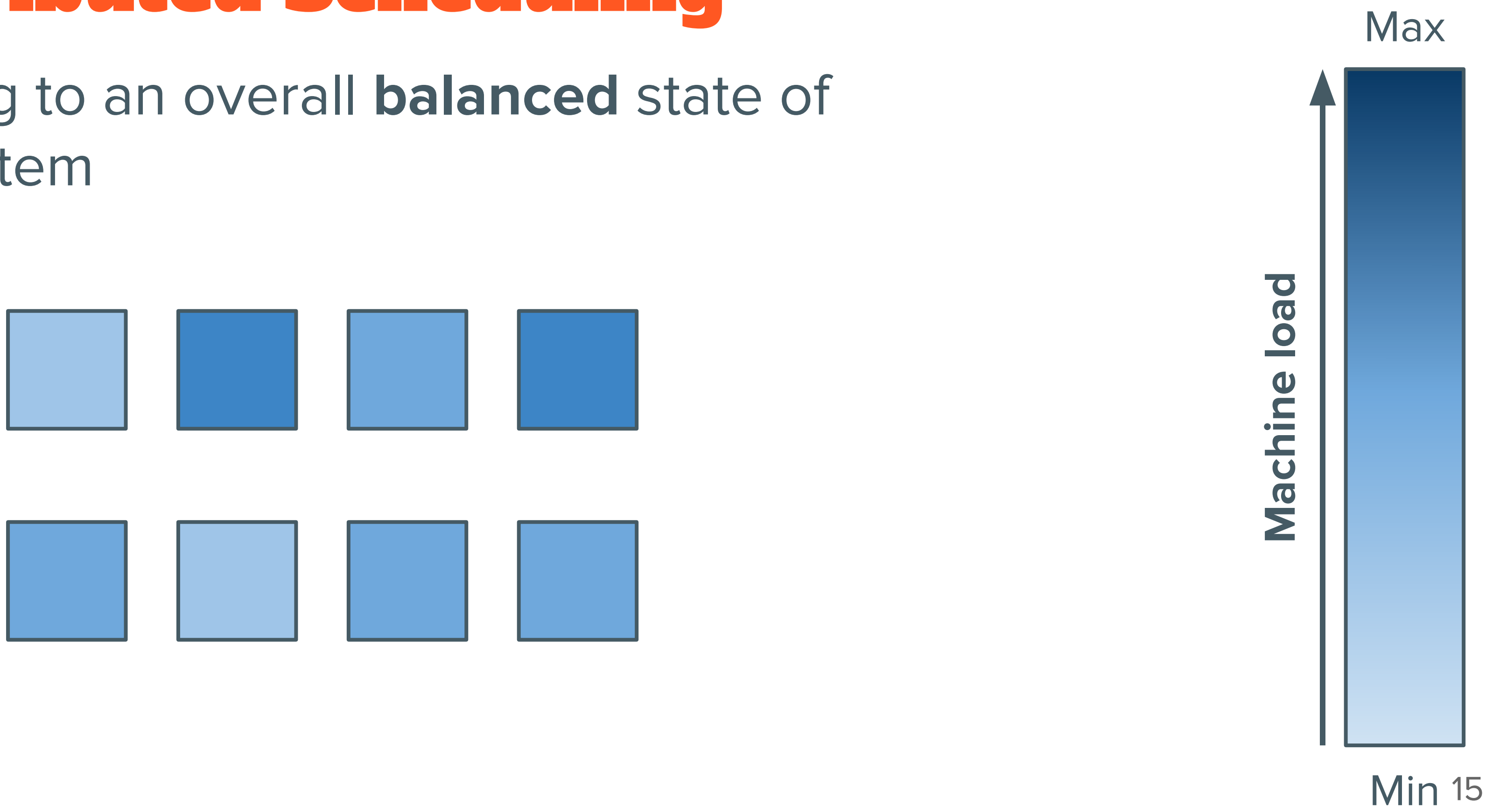
Underloaded machines will have stimulus to **receive** jobs!





Distributed Scheduling

Leading to an overall **balanced** state of the system



Presentation Agenda

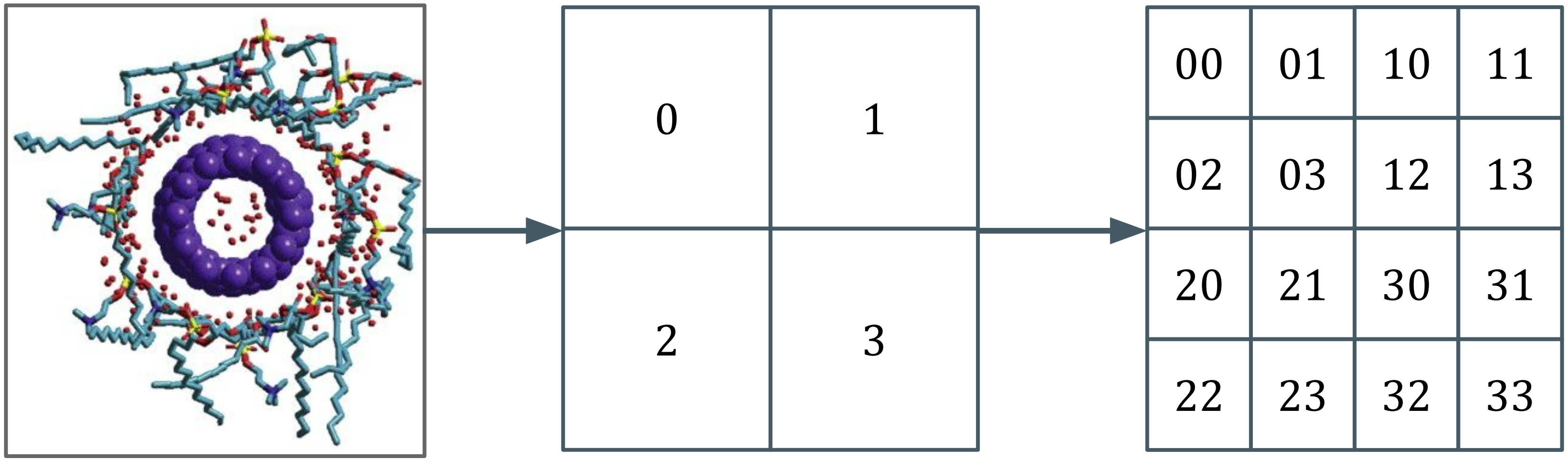
Scalable Load Balancing: Distributed Algorithms & the Packing Model

Complex Decision Making

Parallel applications are overdecomposed to overlap computing and communication as well as having more scheduling options.

This means that $|\mathbf{J}| >> |\mathbf{M}|$; which leads to a high complexity when we have to account for every job in $\mathbf{J}$ in our algorithms.

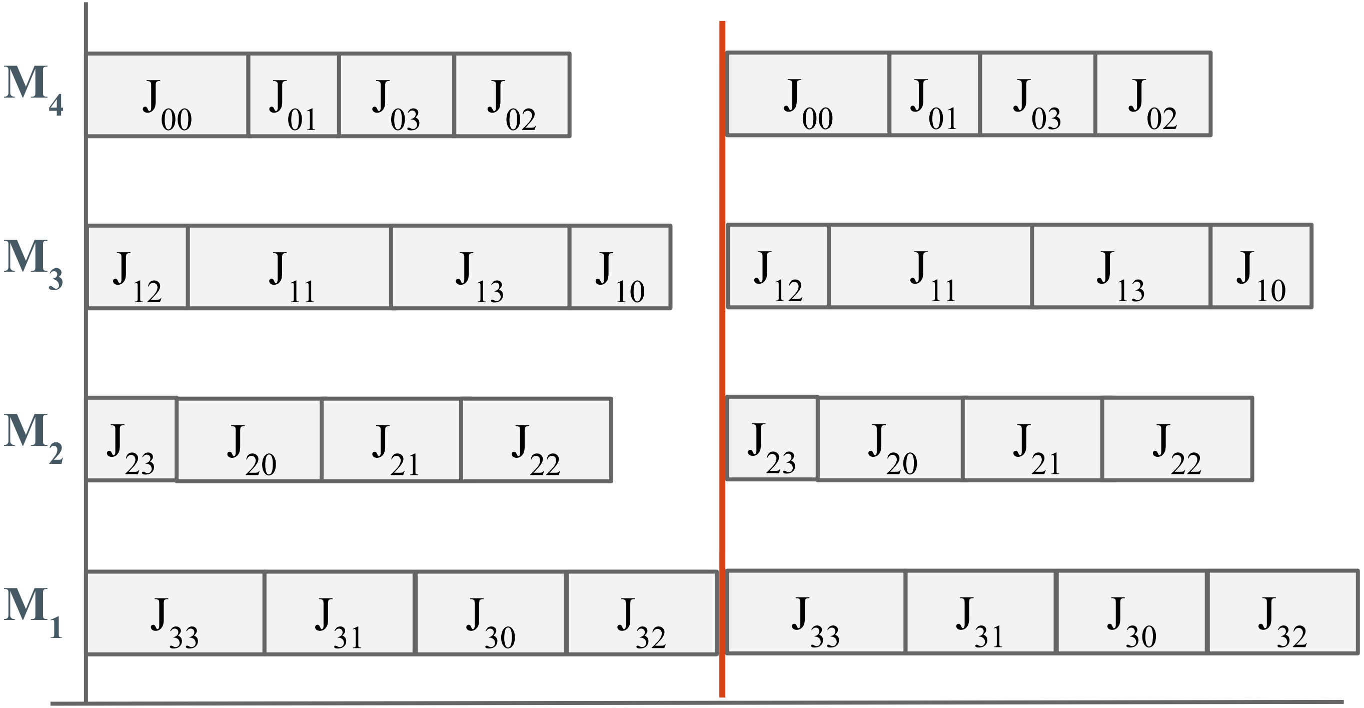
Domain (Over-) Decomposition



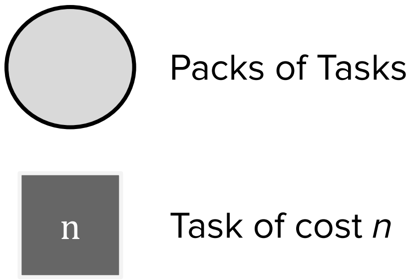
Applications may be spatially decomposed into multiple cells, which may be executed in parallel with periodical synchronizations

Parallel Iterative Applications

00	01	10	11
02	03	12	13
20	21	30	31
22	23	32	33

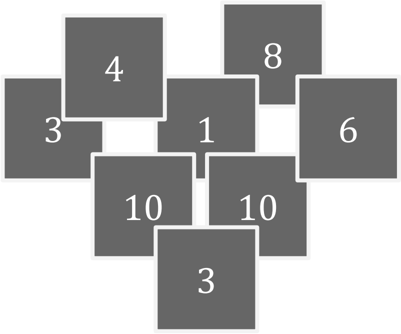


The Packing Model

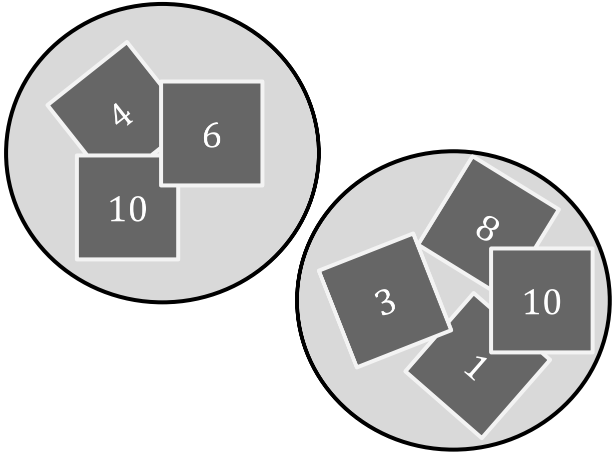


Discretize the problem of load balancing.

Make it into a balls into bins problem.



* non-uniform tasks



* “uniform” packs of tasks

Presentation Agenda

Scalable Load Balancing: Distributed Algorithms & the Packing Model

PackDrop:

Sender Initiated Load Balancing

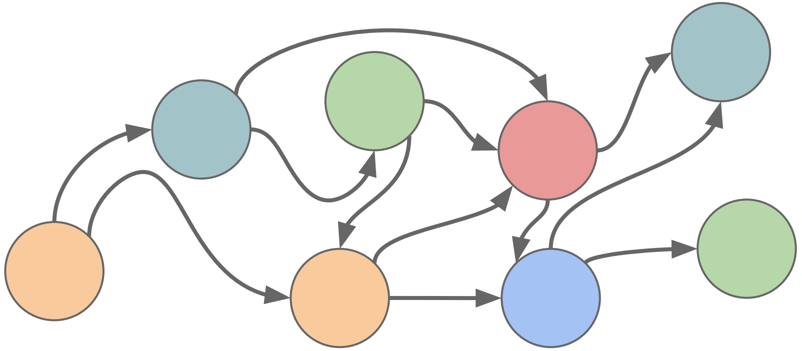
PackDrop - Sender Initiated

Simplified algorithm

1. **Gossip load information**
2. **If overloaded:**
   a. **Until balanced: ...**
   b. **...**

Initially, our algorithm uses a *Gossip Protocol* to spread load information.

This way **overloaded** and **underloaded** machines have a broad view of the state of other machines.



PackDrop - Sender Initiated

Simplified algorithm

1. **Gossip load information**
2. **If overloaded:**
    a. **Until balanced: …**
    b. **…**

Overloaded machines will try to send their workload away their *underloaded* counterparts.

1) Information on who is **overloaded** or **underloaded** is spread by a *Gossip Protocol*.
2) **Overloaded** and **underloaded** machines will portray different behaviors.

PackDrop - Sender Initiated

Simplified algorithm

1. **Gossip load information**
2. **If overloaded:**
   a. **Until balanced:**
      i. **Remove tasks in increasing order of load**
      ii. **Create uniform packs with removed tasks**
   b. **...**

i) Initially, overloaded machines will remove the tasks that make themselves **overloaded** following a *Shortest Processing Time* policy (increasing order of load).

ii) These tasks will be divided into approximately uniform packs, which will be migrated to other machines.

PackDrop - Sender Initiated

Simplified algorithm

1. **Gossip load information**
2. **If overloaded:**
   a. **Until balanced ...**
   b. **Send packs uniformly at random to underloaded machines**
3. **Else ...**

b) Then, machines will randomly choose **underloaded** targets to receive these packs.

PackDrop - Sender Initiated

Simplified algorithm

1. **Gossip load information**
2. **If overloaded …**
3. **Else: When receive a pack:**
   a. **Check if accepting the pack will make me overloaded.**
   b. **No: receive the pack**
   c. **Yes: …**

3) Receiving or not a pack is decided with a *three-way handshake* protocol.

a) **Underloaded** or **balanced** resources will only accept a pack if this pack does not lead them to an **overloaded** state.

b) If everything is ok, the pack will be received and its local load updated.

PackDrop - Sender Initiated

Simplified algorithm

1. **Gossip load information**
2. **If overloaded …**
3. **Else: When receive a pack:**
   a. **Check …**
   b. **No: …**
   c. **Yes: Reject pack.**
   d. **Its owner will look for another receiver**

c) Otherwise, the pack will be ***rejected***

d) At this time the original owner of the pack will choose (uniformly at random) another target for its remaining load.

PackSteal:

Receiver Initiated Load Balancing

PackSteal - Receiver Initiated

Simplified algorithm

1. **Reduce Average Load**
2. **If overloaded:**
   a. **Send a HINT message to a local neighbor**
3. **...**

PackSteal uses a “piggybacking” message exchanging protocol.

Information on Machine load is passed along on every message.

PackSteal - Receiver Initiated

Simplified algorithm

1. **Reduce Average Load**
2. **If overloaded:**
   a. **Send a HINT message to a local neighbor**
3. **...**

2 a) The HINT message will stimulate other Machines to STEAL its load.

PackSteal - Receiver Initiated

Simplified algorithm

1. **Reduce Average Load**
2. **If overloaded: ...**
3. **If underloaded:**
    a. **Send a STEAL msg to a random known machine***

* Target chosen at random if there is no known machine, OR if known machines are denying steal attempts

3 a) The STEAL message will require the Machine to send load to a remote Machine.

When a machine cannot send load back to a STEAL, it will forward the message, sending a STEAL to another machine as if it was sent by the original thief.

Presentation Agenda

Scalable Load Balancing: Distributed Algorithms & the Packing Model

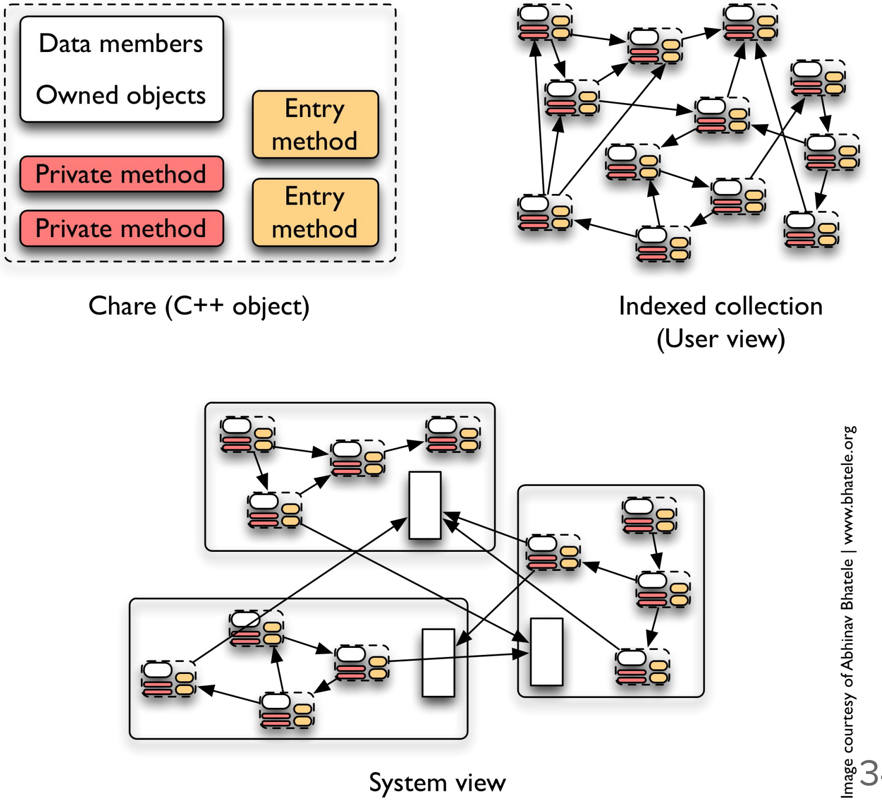
Experimental Evaluation

LB Test - Synthetic Benchmark for Load Balancing evaluation in Charm++.

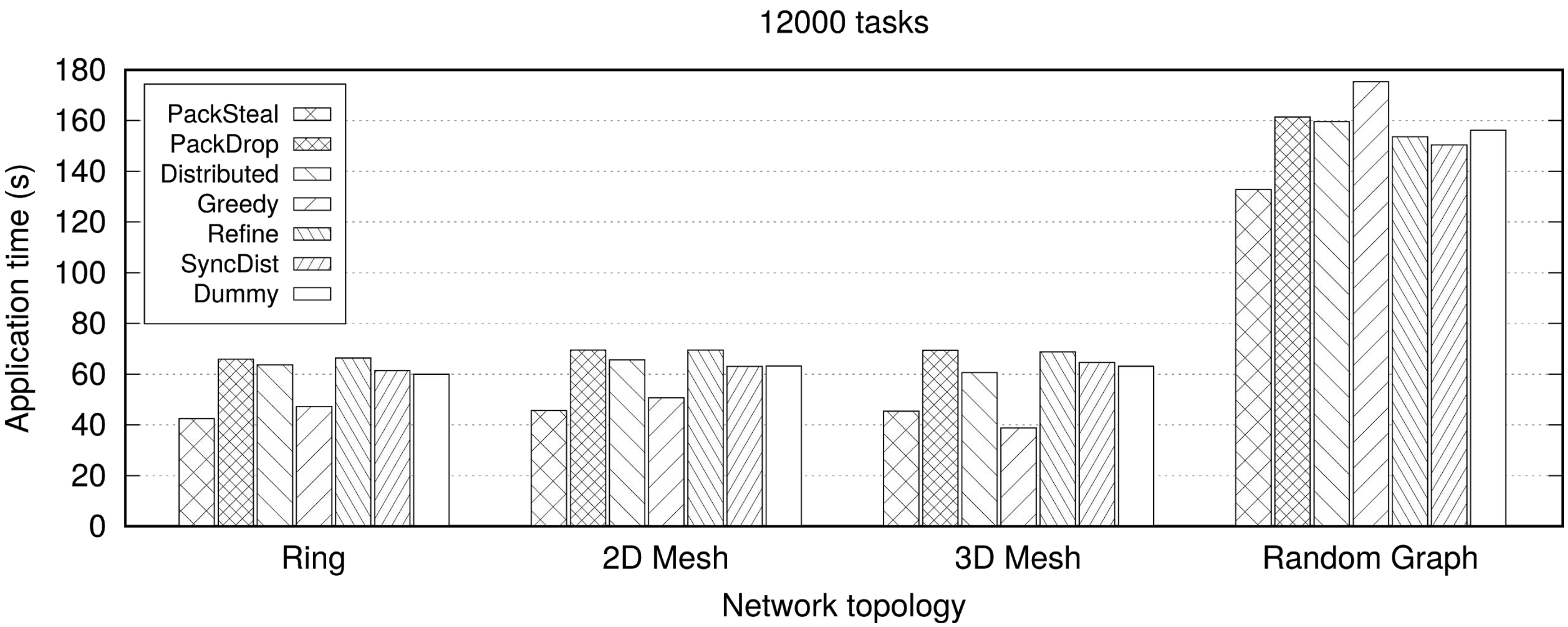
Measuring impacts of:

- Communication patterns
- LB Frequency
- Number of *Chares*

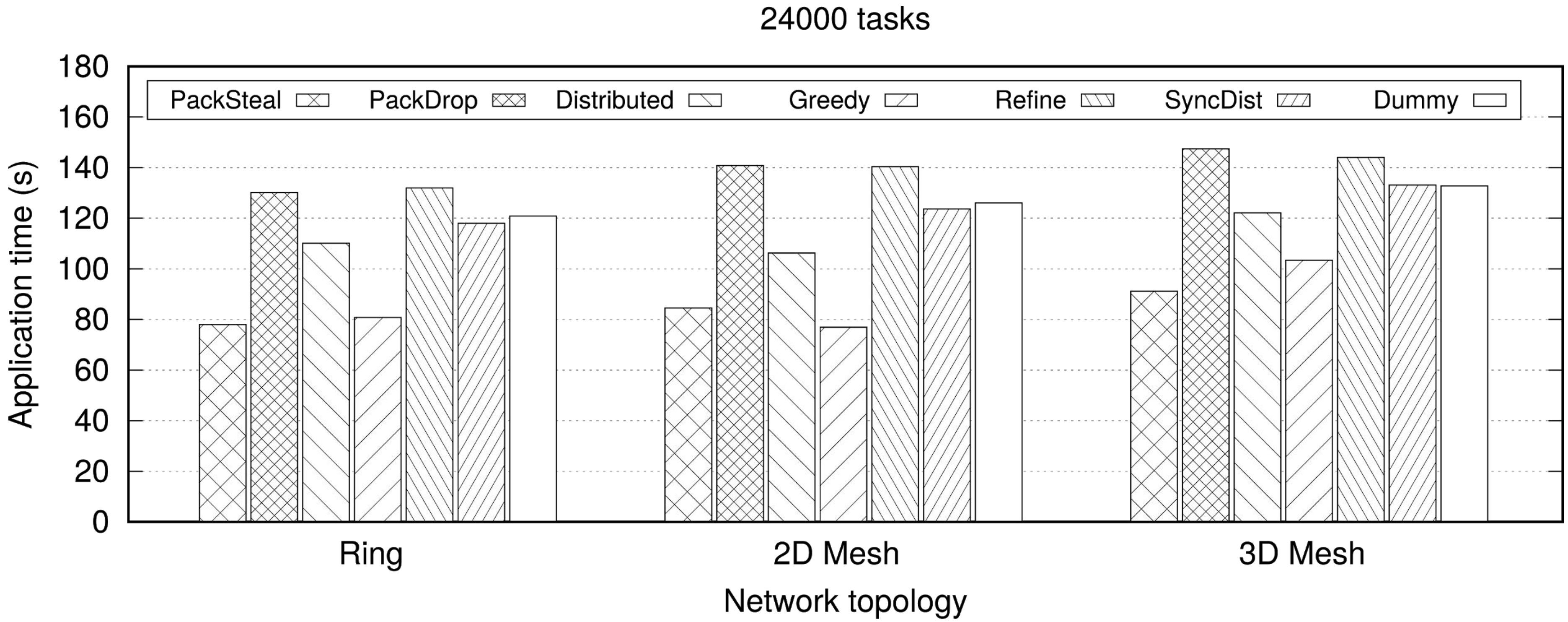
On a NUMA machine with 40 cores and 128GB of RAM.



300 Iterations of Synthetic Load



300 Iterations of Synthetic Load



Experimental Evaluation

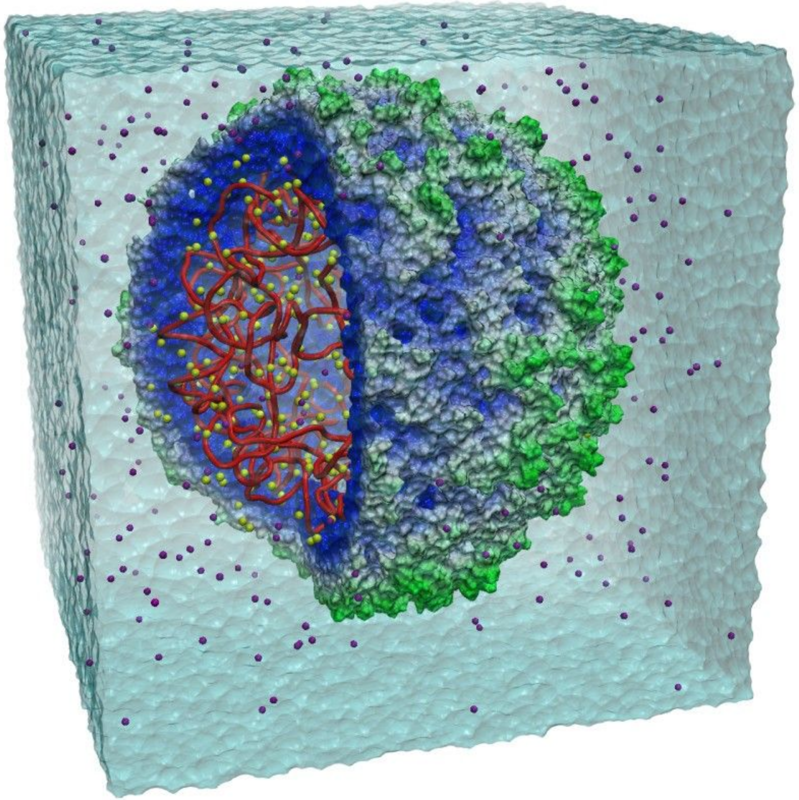
LeanMD - Molecular Dynamics Benchmark for Performance Evaluation in Charm++.

Measuring impacts of:

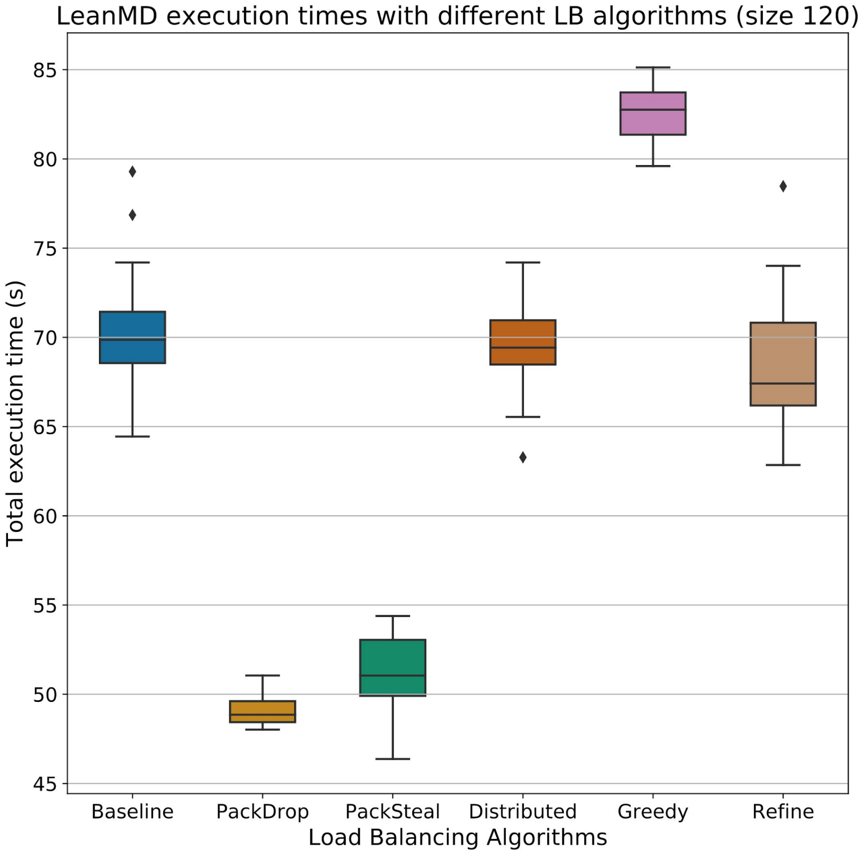
- Simulation size

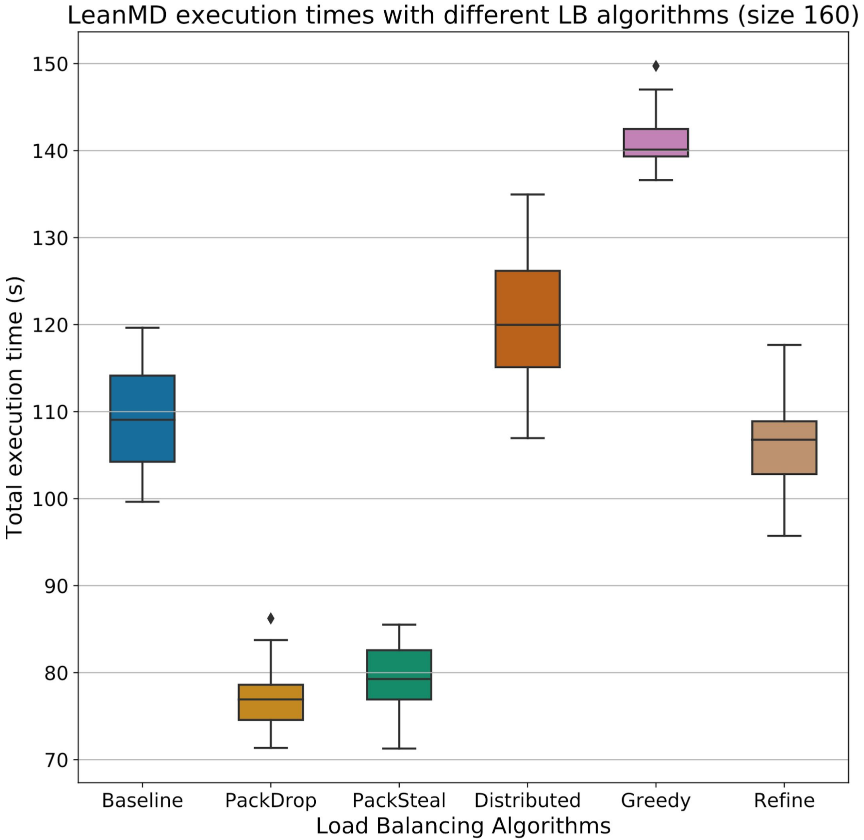
On *Irene* supercomputer with 960 cores in total.

Communication between nodes using MPI.



300 Iterations of MD Load





Presentation Agenda

Scalable Load Balancing: Distributed Algorithms & the Packing Model

Work in Progress

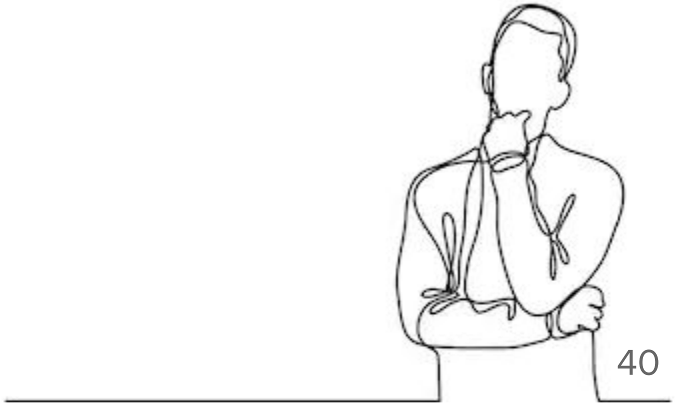
Complete discretization of application workload

Implementation of well-behaved distributed load balancing algorithms for discrete workloads in the HPC context

- Selfish Load Balancing Games
- Random Matching Algorithms
- Other reinforcement learning algorithms

Communication-aware discretization of application workload

- Graph partitioning
- Migration cost estimation



Scalable Load Balancing

Distributed Algorithms & the Packing Model

Vinicius Freitas, Laércio L. Pilla, Johanne Cohen

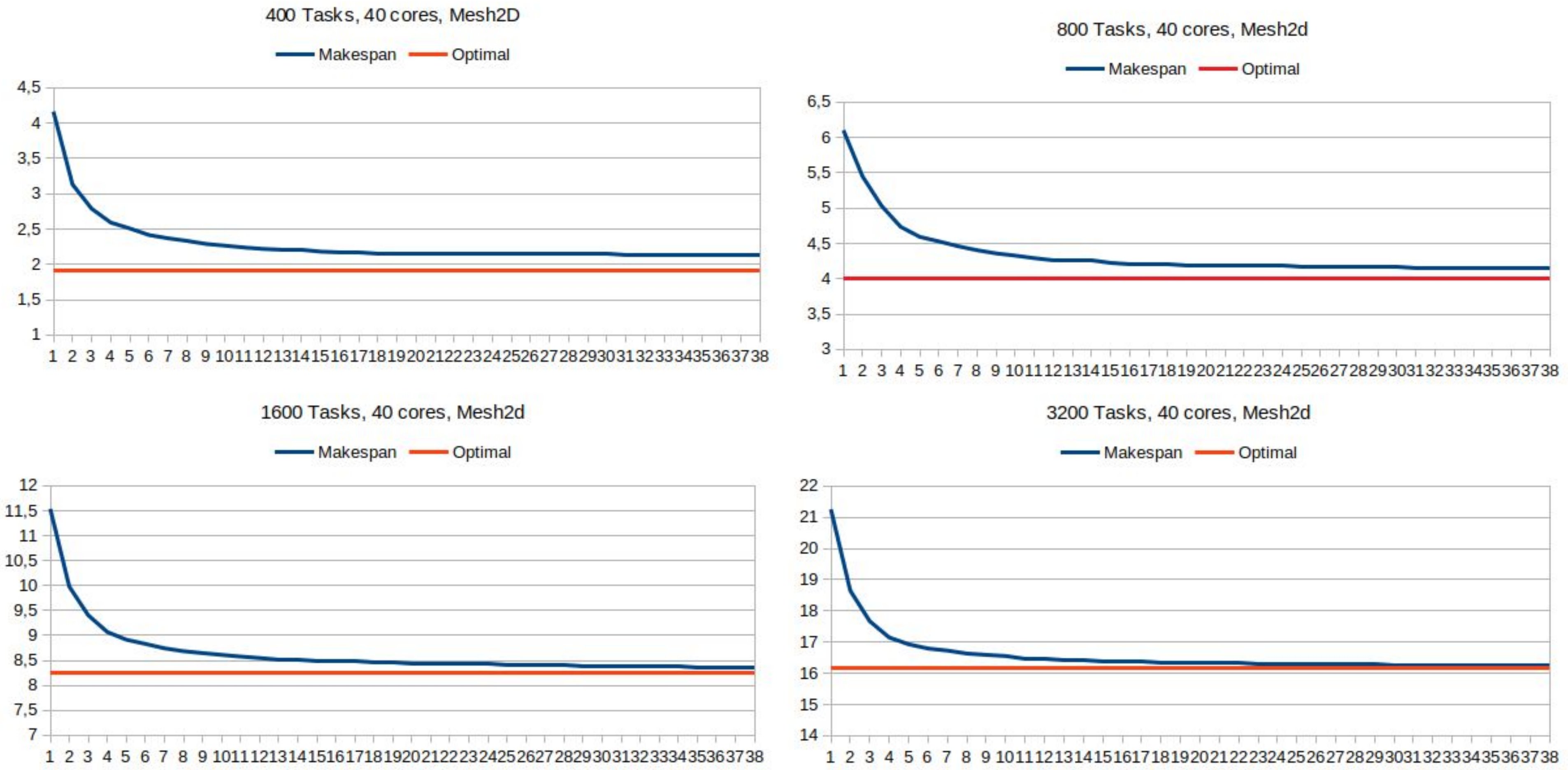
vfreitas@lri.fr ou vinicius.mct.freitas@gmail.com

Extra Turns

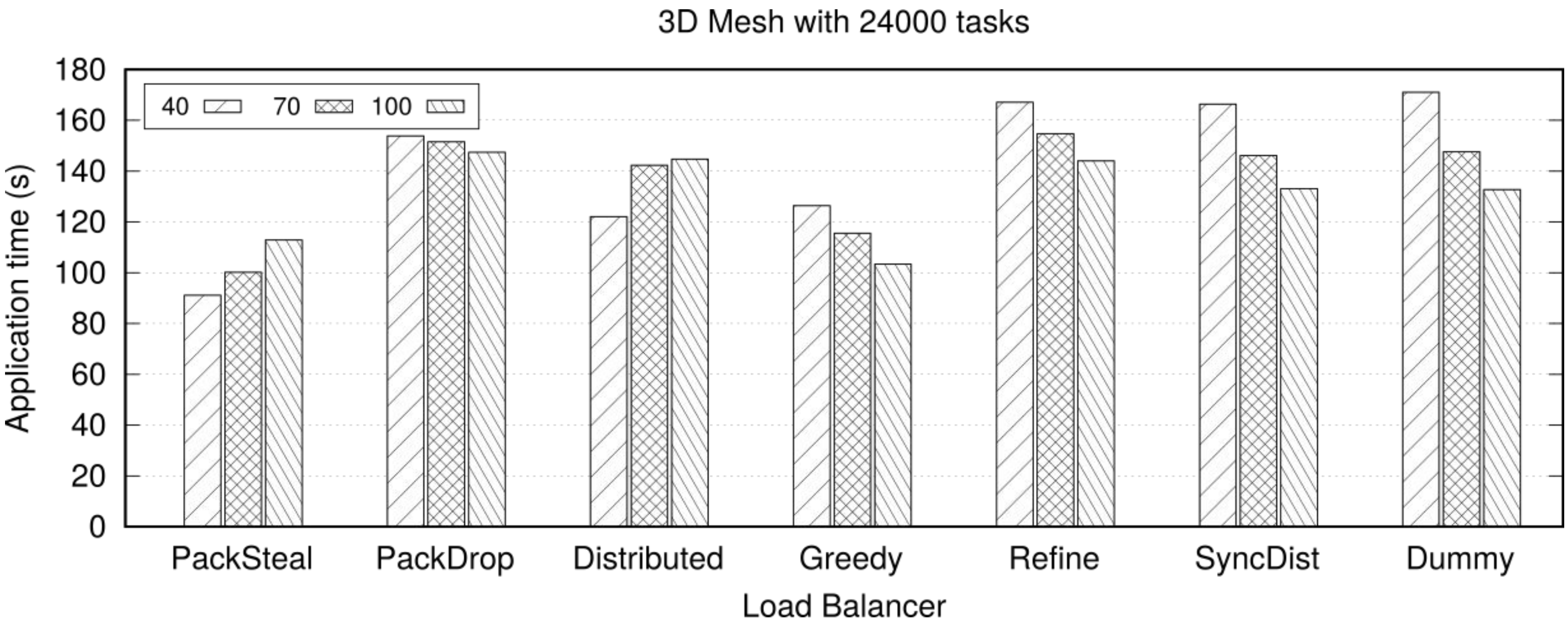
More Graphs for Curious Minds

Vinicius Freitas, Laércio L. Pilla, Johanne Cohen
vfreitas@lri.fr ou vinicius.mct.freitas@gmail.com

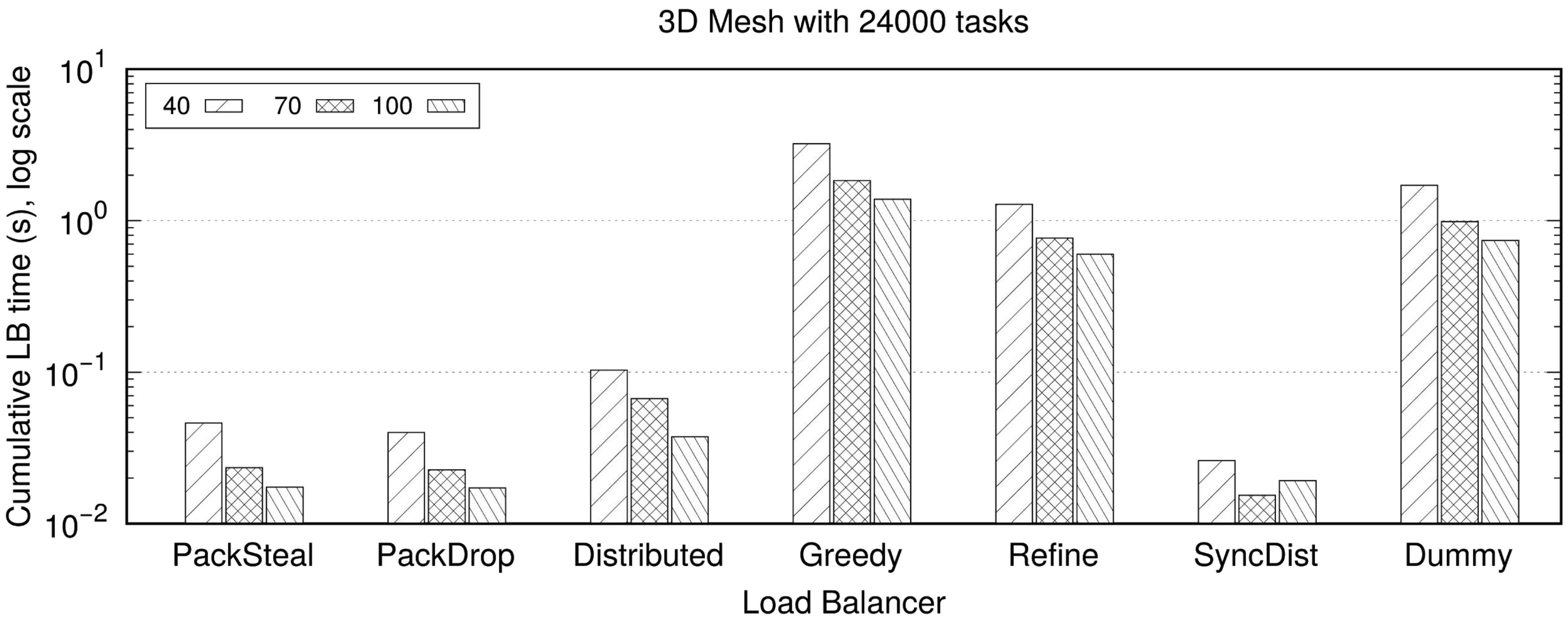
Preliminary observation of convergence time in Distributed Selfish Load Balancing



Varying LB frequency



Cumulative LB Time



300 Iterations of MD Load