Towards Chapel-based Exascale Tree Search Algorithms: dealing with multiple GPU accelerators

Tiago Carneiro, Nouredine Melab, Akihiro Hayashi, Vivek Sarkar

University of Luxembourg (Luxembourg), INRIA Lille/Université de Lille (France), Georgia Tech – (USA)

March 22–27, 2021
(virtual event)

Agenda

Introduction

Algorithms for **solving combinatorial optimization problems (COPs)** can be divided into **exact** (complete) or **approximate** strategies.

Approximate: feasible time.

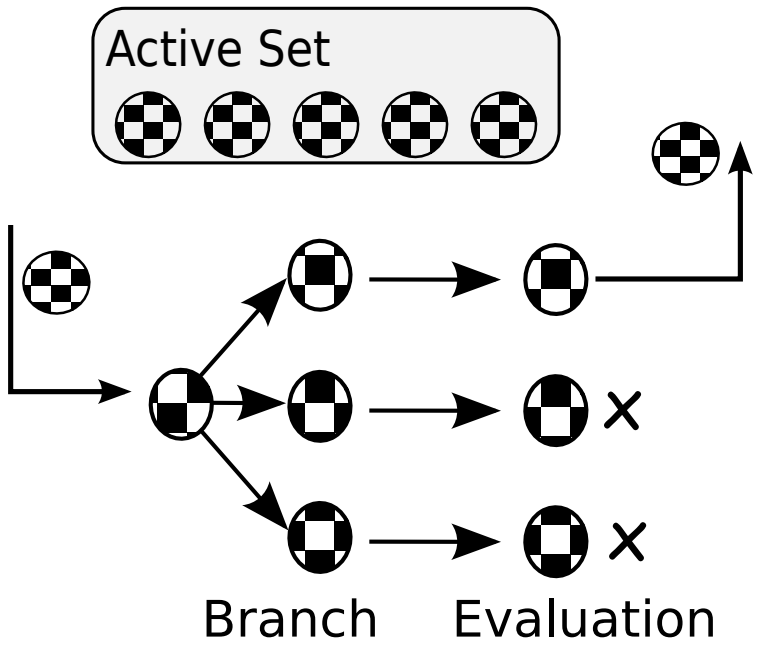
- Local Search (k-opt)
- Meta-heuristics (GRASP, Tabu Search)
- Hybrid Methods, etc.

Exact: exponential execution time.

- Tree-based search: backtracking, B&B (and their variations).
- Dynamic Programming, etc.

Introduction

Tree Search Algorithms implicitly enumerate a solution space, dynamically building a tree.



- Nodes are removed from the **active set** according to the search strategy.
- The search continues until the active set is empty.
- (Crainic, Le Cun, Roucairol (2006)).

Introduction

Tree search algorithms...

- are highly irregular,
- demand hand-optimized data structures,
- demand load balancing schemes,
- **exponential execution time**.

Implementation is performance-oriented:

- Usually in C/C++
- Low-level features $\rightarrow$ performance
- Parallel computing libraries

GPUs are crucial in exact optimization: they enable solving to the optimality instances having prohibitive execution times on CPUs.

Introduction

In this way, it is expected that exascale systems will decrease the time required to solve instances of COPs to the optimality.

- several programming models, languages, runtimes combined to efficiently exploit all levels of parallelism of such heterogeneous systems.

High-productivity languages:

- Exascale systems are going to be complex to program.
- **Efforts towards productivity are crucial for better exploiting the future generation of supercomputers**.

Introduction

Chapel plays a special role: it provides different parallel and distributed iterators that implement load balancing among processes, besides other high-level features.

Chapel's parallel iterators:

- Iterators in Chapel are similar to procedures that can be used to isolate iterations from the loop body.
- Each value yielded by the iterator corresponds to an iteration of the loop.

Introduction

Chapel distributed iterators are a key feature for achieving a trade-off between productivity and parallel efficiency/performance in distributed tree-based optimization.

Distributed iterators implement:

- Master/worker model for pool management;
- Distributed load balancing/work distribution;
- Complex distributed metrics reduction;
- Termination criteria.

Thus, using Chapel can be up to 8× **more productive** than MPI+C in the context of this work (Carneiro et al. (2020)).

Agenda

Initial Premises

GPUs are required in exact optimization: they enable solving to the optimality instances having prohibitive execution times on CPUs.

GPU support in Chapel:

- A couple of approaches try to mitigate the gap between Chapel and GPUs (Chu et al. (2017) and Sidelnik et al. (2012)).
- **They do not allow lower-level GPU programming features.**
- We do need such features in exact optimization.

Initial Premises

Hayashi et al. (2019): https://github.com/ahayashi/chapel-gpu

- GPU/CPU oriented iterator for Chapel.
- Allows the use of pre-compiled CUDA-C/C++ kernels.
- Allows concurrent heterogeneous distributed CPU/GPU execution.
- **However, does not provide load balancing between GPUs/CPU tasks**.

Tree-based search are highly irregular applications
Load balancing is therefore crucial.

Initial Premises

Hayashi's iterator is not suitable for irregular GPU-based applications.

So, how to use both Chapel's high-level features for distributed programming and GPUs?

Agenda

The Proposed Algorithm

We revisit:

- Master-worker distributed tree-search,
- Using Chapel's distributed iterators for **work distribution**,
- Distributed backtracking,
- Enumerates *all* valid configurations of the N-Queens.

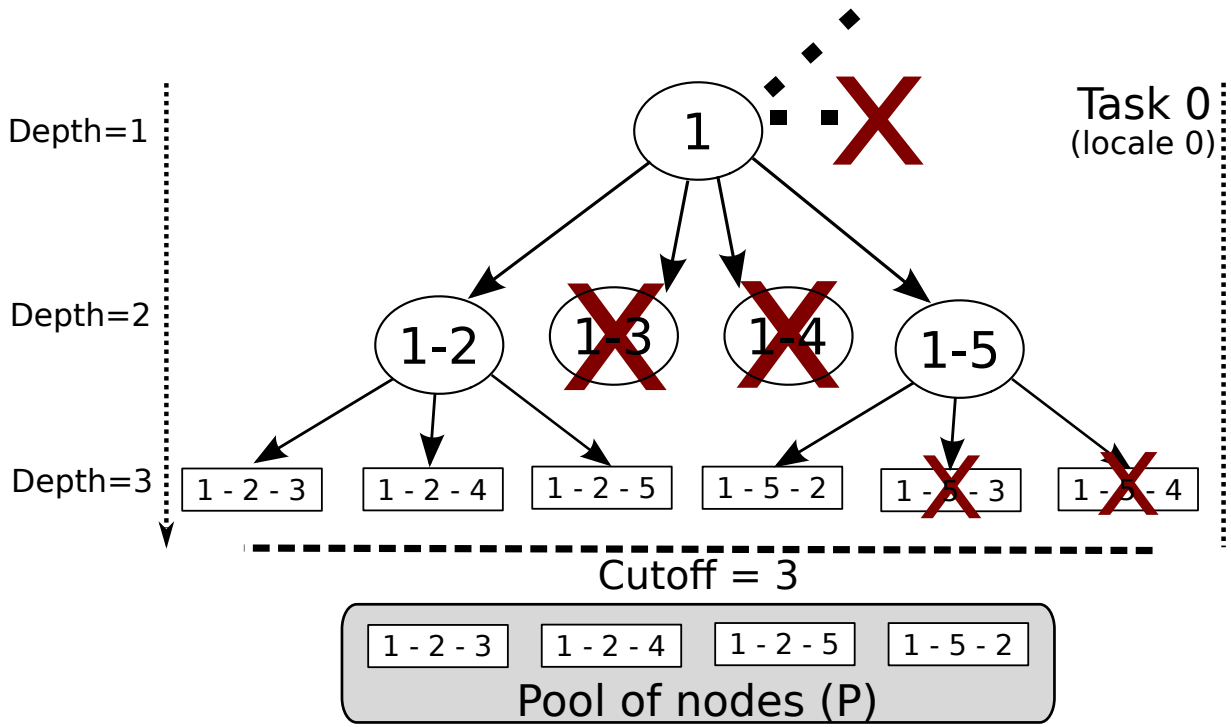
But now:

- Using C-Interoperability layer along with Chapel's high-level features,
- Pre-compiled CUDA-C kernels,
- CPU-GPU scheme.

the N-Queens problem is a proof-of-concept that motivates further improvements in solving related COPs.

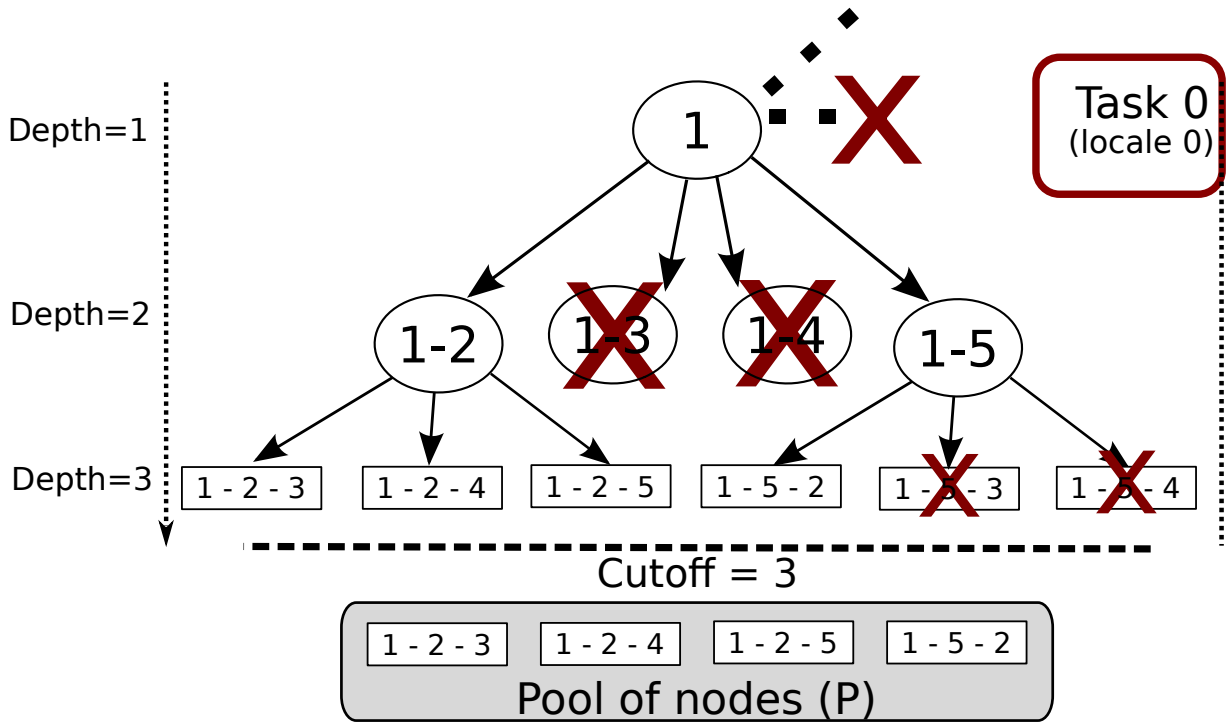
The Master Locale

Generating the initial pool of subproblems: partial search on CPU.



The Master Locale

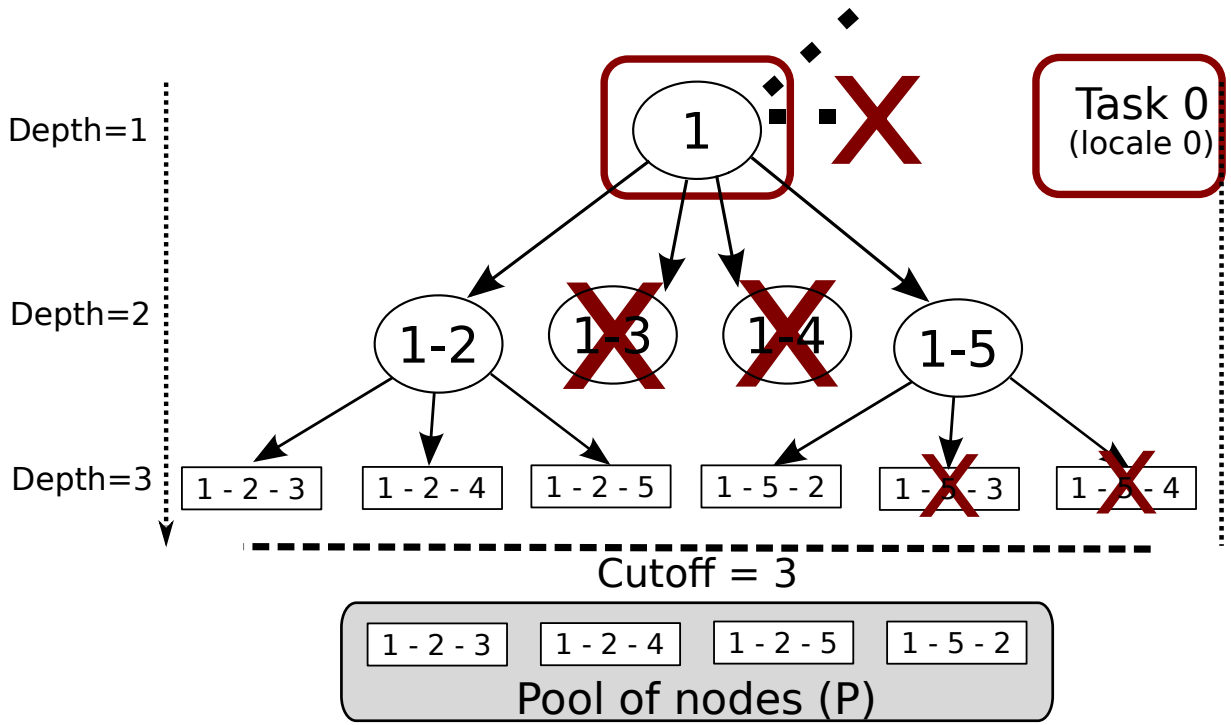
Generating the initial pool of subproblems: partial search on CPU.



- On Locale 0 (master) - Task 0.
- Serial search.

The Master Locale

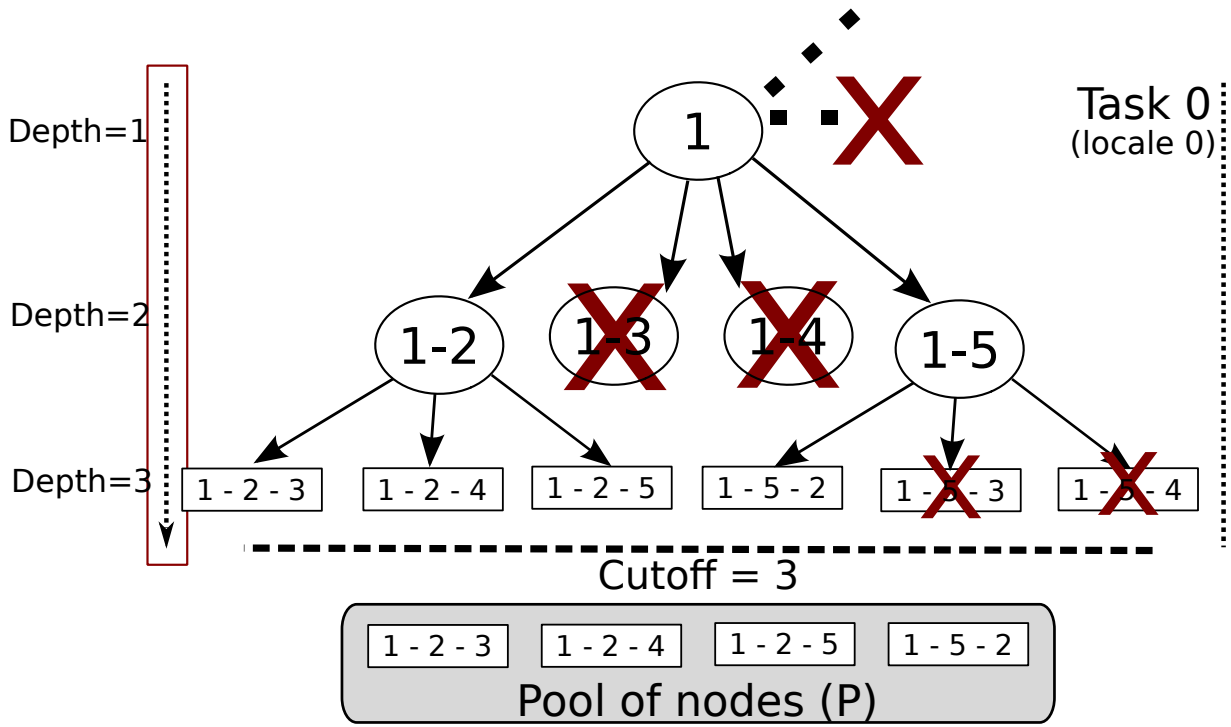
Generating the initial pool of subproblems: partial search on CPU.



- On Locale 0 (master) - Task 0.
- Serial search.
- Starts with the initial configuration of the problem.

The Master Locale

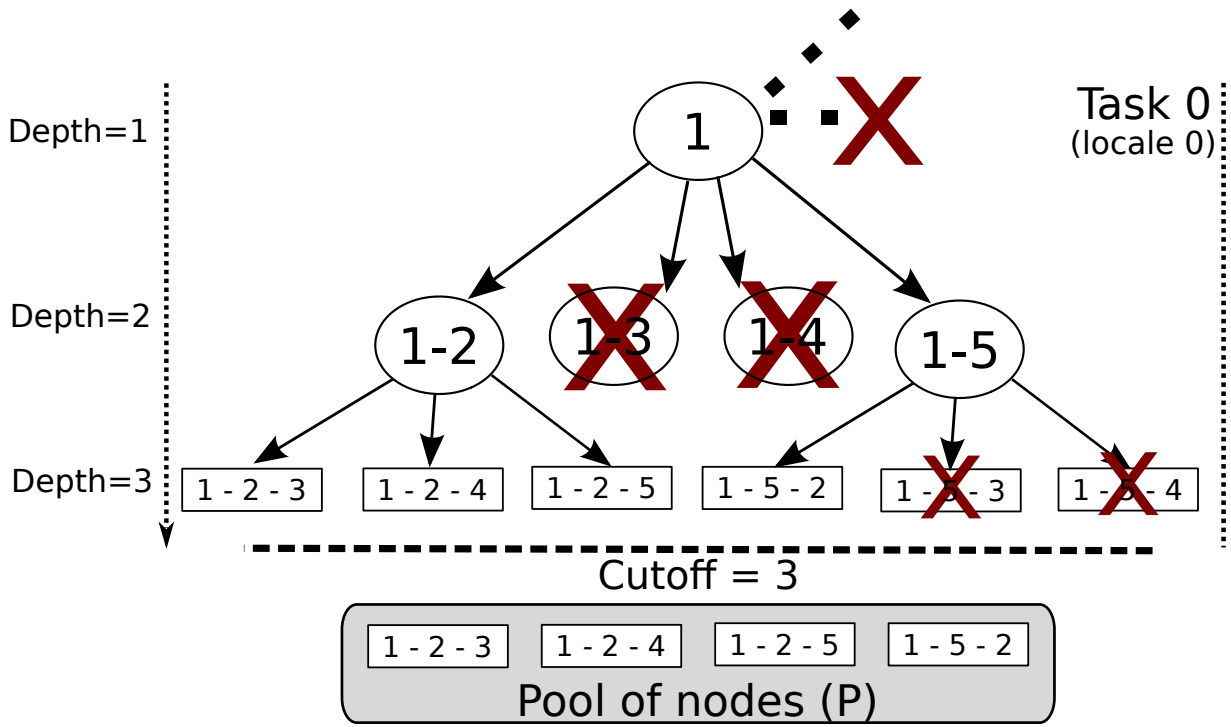
Generating the initial pool of subproblems: partial search on CPU.



- **Partial search** until a cutoff depth.

The Master Locale

Generating the initial pool of subproblems: partial search on CPU.



- Pool keeps **all valid and feasible** partial solutions with **cutoff** elements of the permutation.

The Master Locale

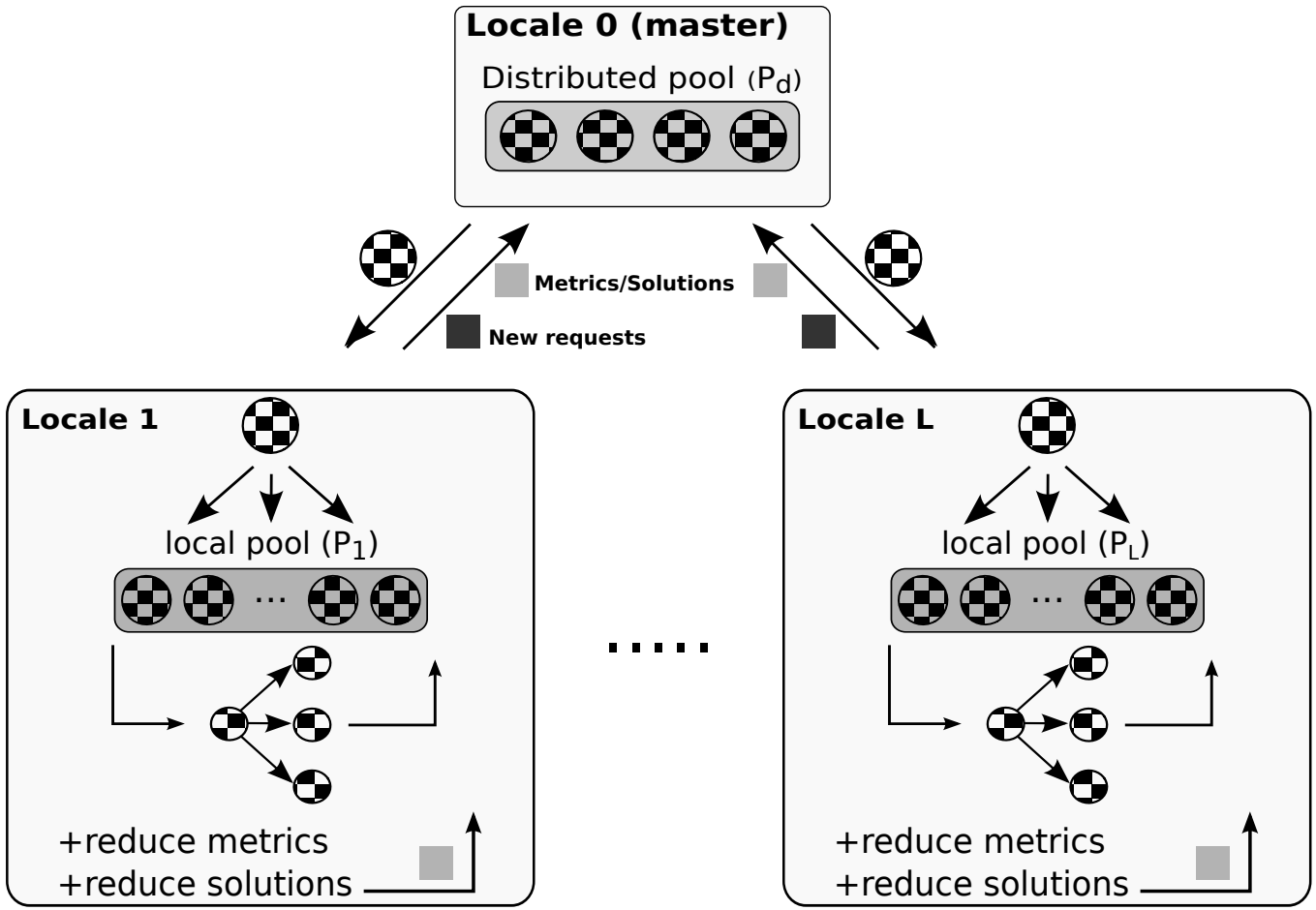
Master-worker: distributed load balancing using iterators.

Algorithm 1: The Master locale.

```
1  N ← get_problem( )
2  cutoff ← get_cutoff_depth( )
3  second_cutoff ← get_scnd_cutoff_depth( )
4  P ← {} Node
5  metrics ← (0, 0)
6  metrics += initial_search(N, cutoff, P)
7  Size ← {0..(|P| − 1)} // Domain
8  D ← Size mapped onto locales to a standard distribution
9  P_d ← [D] : Node
10 P_d = P // Using implicit bulk-transfer
11 forall node in P_d following a distributed iterator with(+ reduce
       metrics) do
12 |   metrics += Algorithm_2(N, node, cutoff,
13 |       second_cutoff)
14 end
15 present_results(metrics)
```

The Master Locale:

Master-worker: distributed load balancing using iterators.



Exploiting Intra-node Parallelism

From one node received via parallel iterator, how to...

- Generate load enough for all GPUs of the system?
- How to divide the load among GPUs?

Exploiting Intra-node Parallelism

From one node received via parallel iterator, how to...

1. Generate load enough for all GPUs of the system?
   - Nested parallelism generating a local pool via partial search.
   - From depth **cutoff** until depth **second_cutoff**.
   - Thus, it is another partial search on CPU.
   - Task-local pool.
2. How to divide the subproblems of the local pool among GPUs?

Exploiting Intra-node Parallelism

From one node received via parallel iterator, how to...

1. Generate load enough for all GPUs of the system?
   - **Nested parallelism** for generating a local pool via partial search.
   - From depth **cutoff** until depth **second_cutoff**.
   - Thus, it is another partial search on CPU.
   - Task-local pool.
2. From the task-local pool...
   - How to divide the subproblems of the local pool among GPUs?

Exploiting Intra-node Parallelism

How to divide the local pool among GPUs?

Algorithm 3: Exploiting multiple GPUs.

Input: N, P, the second cutoff $depth$
Output: A *tuple* containing the explored tree size and the number of complete and valid solutions found on GPU.

```
1  tree_h ← [0..|P| − 1] int
2  sols_h ← [0..|P| − 1] int
3  γ ← cuda_get_num_devices( )
4  GPU_load ← |P|
5  forall gpu_id in 0..γ − 1 do
6      cuda_set_gpu(gpu_id)
7      device_load ← get_load(gpu_id, GPU_load, γ)
8      stride ← get_starting_point(GPU_load, gpu_id, γ)
9      sols_ptr ← sols_h + stride
10     tree_ptr ← tree_h + stride
11     pool_ptr ← P + stride
12     call_GPU_search(N, depth, device_load, pool_ptr,
13         tree_ptr, sols_ptr)
14 end
15 redTree ← (+ reduce tree_size_h)
16 redSols ← (+ reduce sols_h)
17 metrics+ = (redTree, redSol)
18 return (redTree, redSols)
```

- Variables for metrics reduction.
- Getting the number of GPUs via CUDA.

Exploiting Intra-node Parallelism

How to divide the local pool among GPUs?

Algorithm 3: Exploiting multiple GPUs.

Input: N, P, the second cutoff $depth$
Output: A *tuple* containing the explored tree size and the number of complete and valid solutions found on GPU.

```
1  tree_h ← [0..|P| − 1] int
2  sols_h ← [0..|P| − 1] int
3  γ ← cuda_get_num_devices( )
4  GPU_load ← |P|
5  forall gpu_id in 0..γ − 1 do
6      cuda_set_gpu(gpu_id)
7      device_load ← get_load(gpu_id, GPU_load, γ)
8      stride ← get_starting_point(GPU_load, gpu_id, γ)
9      sols_ptr ← sols_h + stride
10     tree_ptr ← tree_h + stride
11     pool_ptr ← P + stride
12     call_GPU_search(N, depth, device_load, pool_ptr,
13         tree_ptr, sols_ptr)
14 end
15 redTree ← (+ reduce tree_size_h)
16 redSols ← (+ reduce sols_h)
17 metrics+ = (redTree, redSol)
18 return (redTree, redSols)
```

- For each GPU...
- Get the GPU load.

Exploiting Intra-node Parallelism

How to divide the local pool among GPUs?

Algorithm 3: Exploiting multiple GPUs.

Input: N, P, the second cutoff $depth$
Output: A *tuple* containing the explored tree size and the number of complete and valid solutions found on GPU.

```
1  tree_h ← [0..|P| − 1] int
2  sols_h ← [0..|P| − 1] int
3  γ ← cuda_get_num_devices( )
4  GPU_load ← |P|
5  forall gpu_id in 0..γ − 1 do
6      cuda_set_gpu(gpu_id)
7      device_load ← get_load(gpu_id, GPU_load, γ)
8      stride ← get_starting_point(GPU_load, gpu_id, γ)
9      sols_ptr ← sols_h + stride
10     tree_ptr ← tree_h + stride
11     pool_ptr ← P + stride
12     call_GPU_search(N, depth, device_load, pool_ptr,
13         tree_ptr, sols_ptr)
14 end
15 redTree ← (+ reduce tree_size_h)
16 redSols ← (+ reduce sols_h)
17 metrics+ = (redTree, redSol)
18 return (redTree, redSols)
```

- For each GPU...
- Calculate its load.
- Calculate the pointers for calling the CUDA kernel.
- Strides on the memory.

Exploiting Intra-node Parallelism

How to divide the local pool among GPUs?

Algorithm 3: Exploiting multiple GPUs.

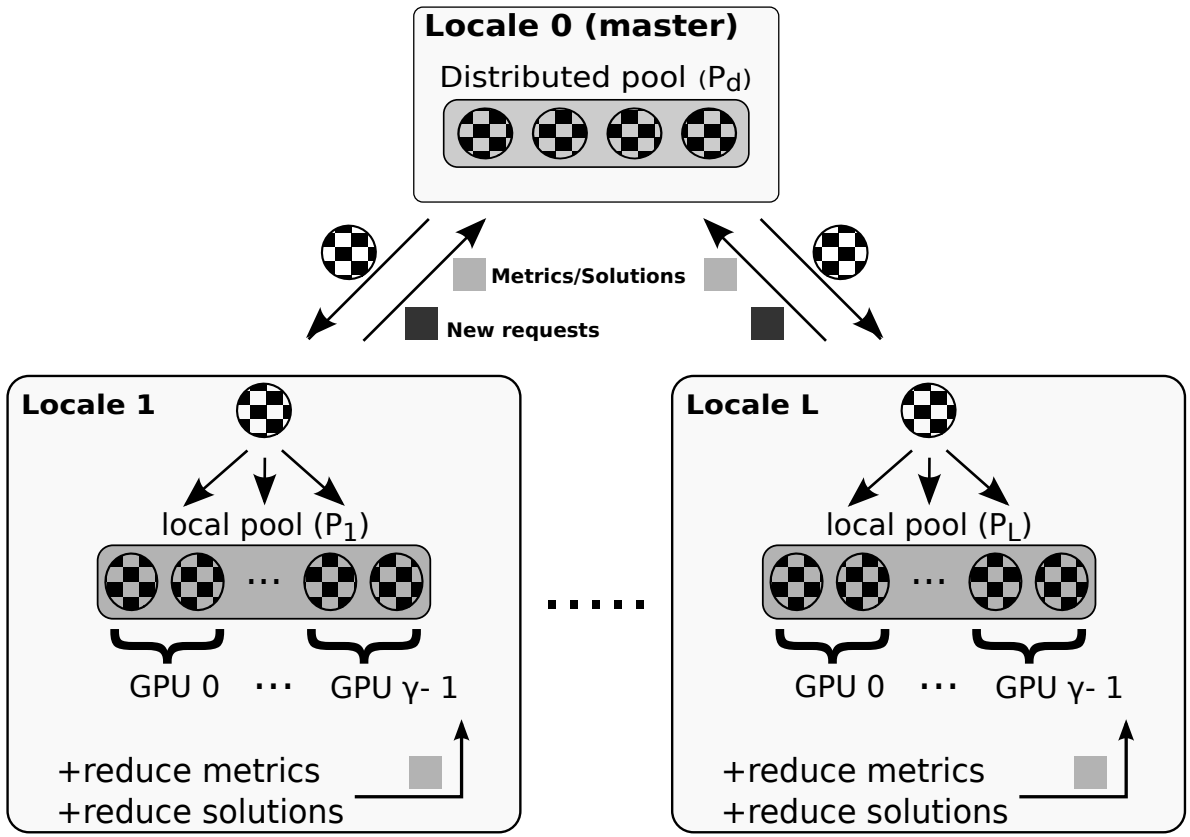
Input: N, P, the second cutoff $depth$
Output: A *tuple* containing the explored tree size and the number of complete and valid solutions found on GPU.

```
1  tree_h ← [0..|P| − 1] int
2  sols_h ← [0..|P| − 1] int
3  γ ← cuda_get_num_devices( )
4  GPU_load ← |P|
5  forall gpu_id in 0..γ − 1 do
6      cuda_set_gpu(gpu_id)
7      device_load ← get_load(gpu_id, GPU_load, γ)
8      stride ← get_starting_point(GPU_load, gpu_id, γ)
9      sols_ptr ← sols_h + stride
10     tree_ptr ← tree_h + stride
11     pool_ptr ← P + stride
12     call_GPU_search(N, depth, device_load, pool_ptr,
13         tree_ptr, sols_ptr)
14 end
15 redTree ← (+ reduce tree_size_h)
16 redSols ← (+ reduce sols_h)
17 metrics+ = (redTree, redSol)
18 return (redTree, redSols)
```

- Parallel reduction using Chapel's high-level features.

Overview of the algorithm

Two levels of parallelism: intra-node and inter-node.



Agenda

Performance Evaluation

Evaluation: The following programs were conceived for enumerating all valid and complete configurations of the N-Queens problem.

- **Baseline**: single-node multi-GPU implementation optimized for single-locale execution written in CUDA-C.
- **GPUIterator**: distributed version of the baseline implementation written in Chapel. Uses Hayashi et al. (2019). No load balancing.
- **ChplGPU**: implementation previously detailed.

Obs. : All implementations employ the same CUDA-C kernel code.

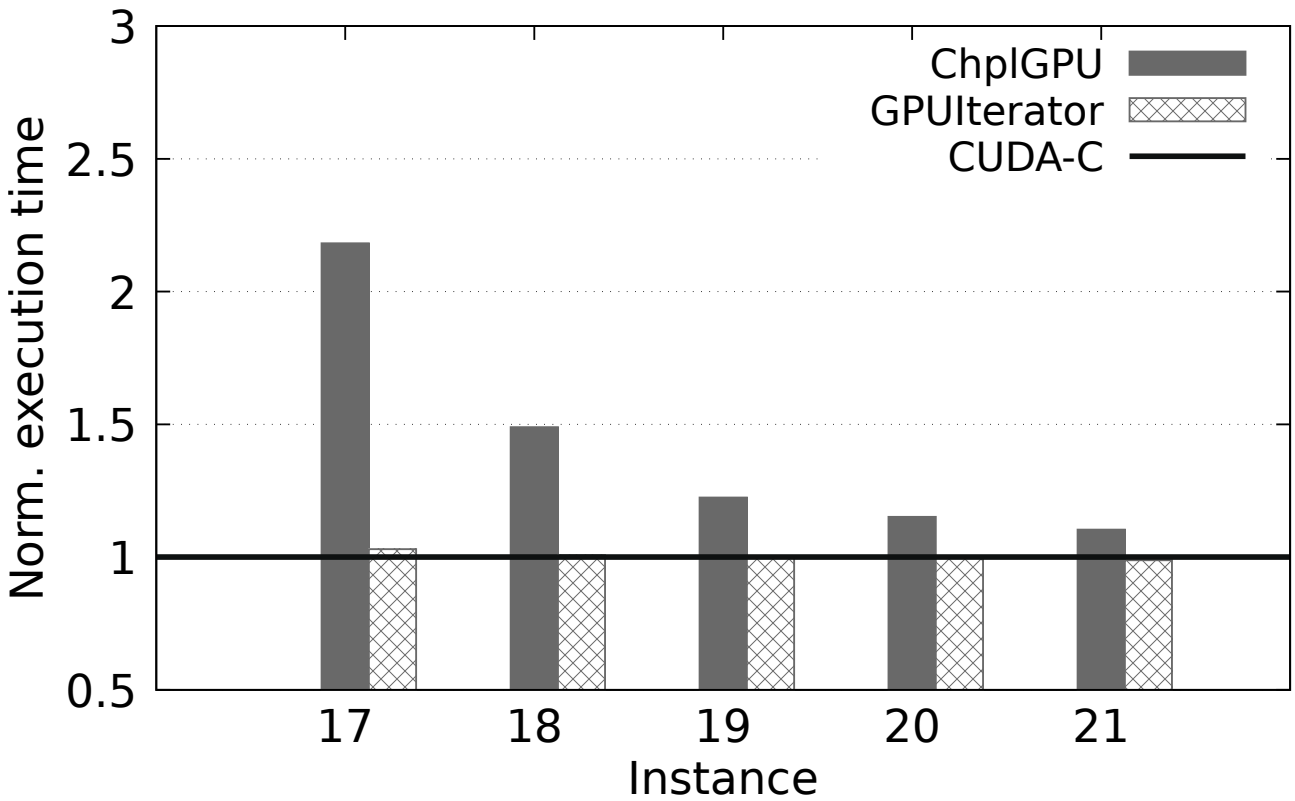
Performance Evaluation

Parameters settings:

- 12 computer nodes.
- *Two* Intel Xeon E5-2650 v4@ 2.00GHz (a total of 24 cores/ 48 threads per node) and 128 GB RAM.
- *Two* NVIDIA GeForce GTX 1080 Ti – Pascal generation (11GB RAM and 3584 CUDA cores @ 1582Mhz).
- Maximum 24 GPUs (86,016 CUDA cores) used in the experiments.
- 100 Gbps Intel Omni-Path network.

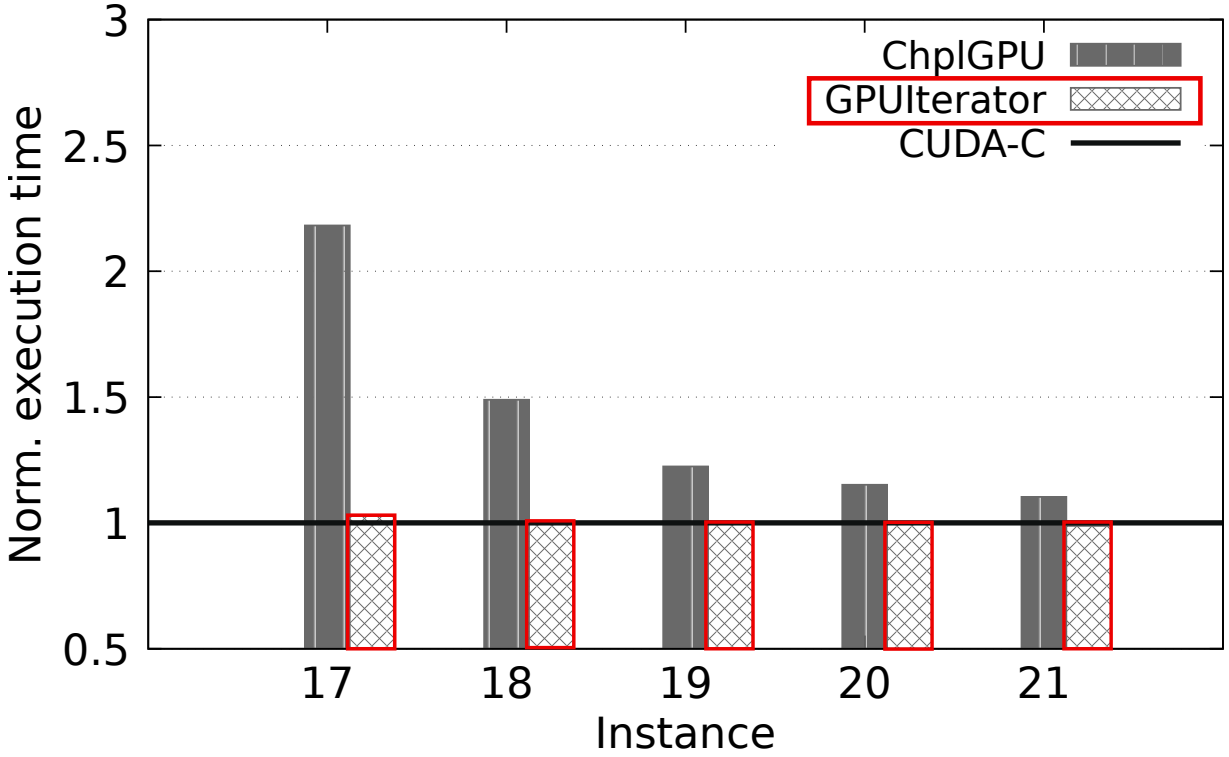
Performance Evaluation

Single-node performance: *one* computer node – *two* GPUs.



Performance Evaluation

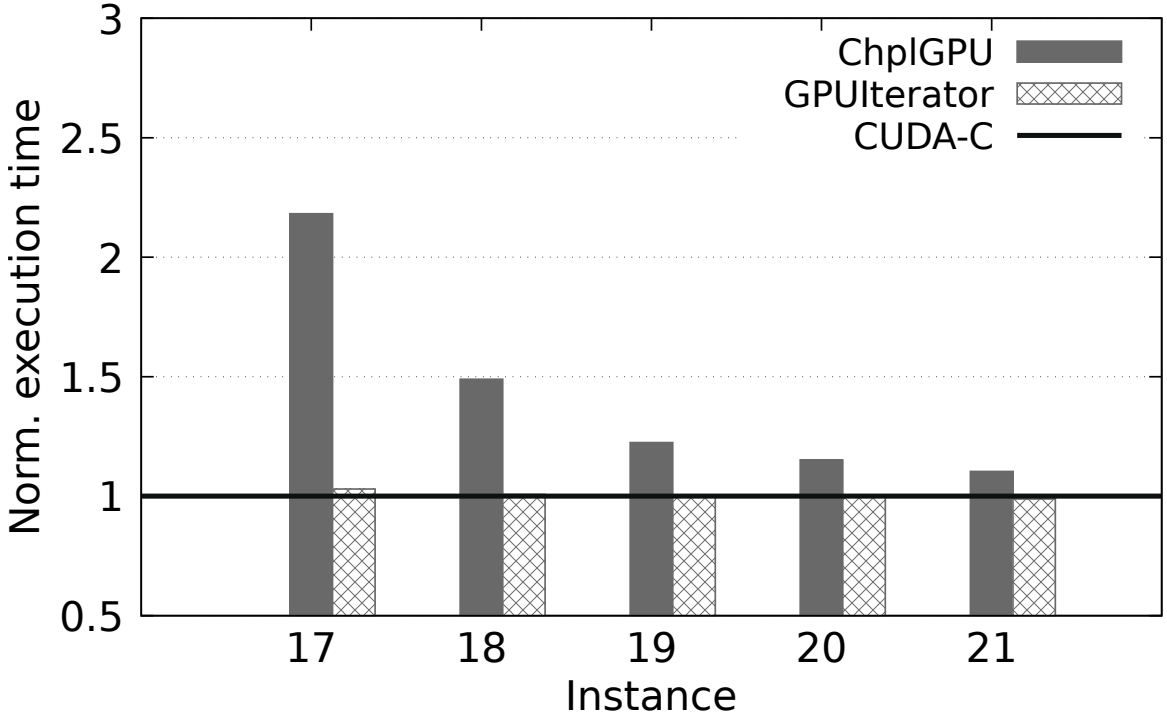
Single-node performance: *one* computer node – *two* GPUs.



- Chpl-based GPUIterator equivalent to the CUDA-C baseline.
- Using the iterator does **not mean significant overhead**.

Performance Evaluation

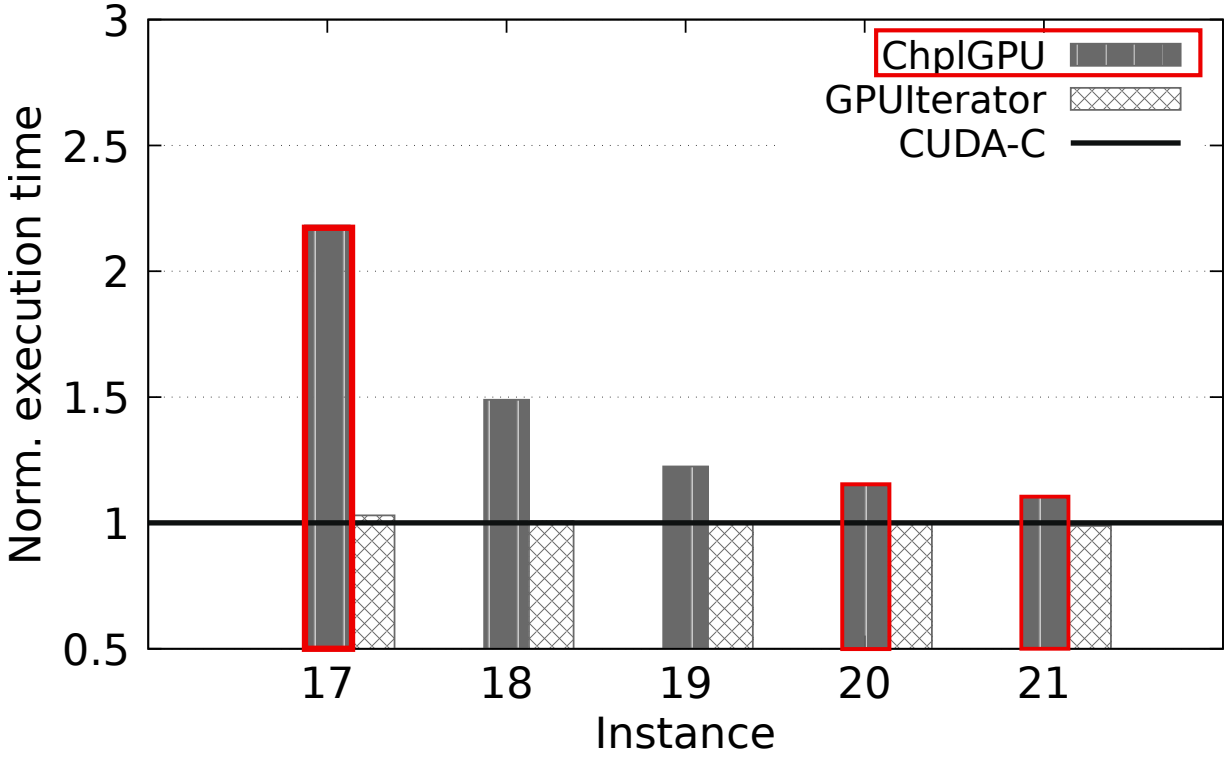
Single-node performance: *one* computer node – *two* GPUs.



- The ChplGPU implementation **is not single-node oriented**.
- Nested parallelism and other features **for distributed execution.**

Performance Evaluation

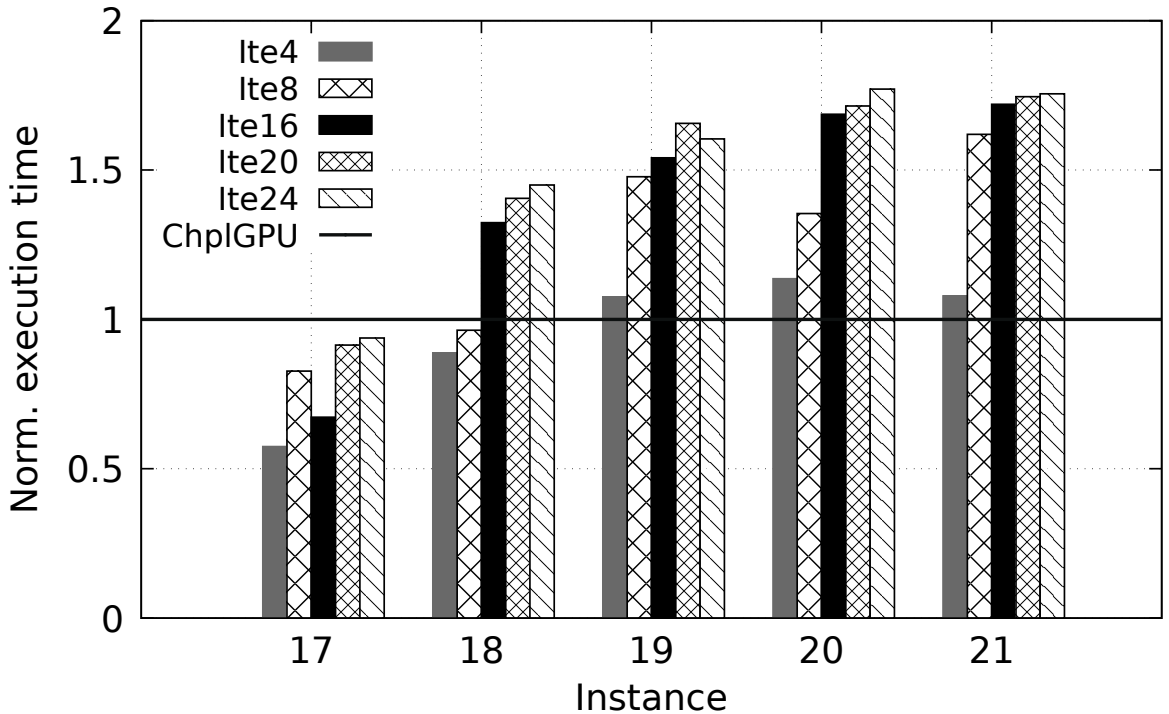
Single-node performance: *one* computer node – *two* GPUs.



- **The overhead becomes less significant** as N increases.
- It is from 2.15× to 10% slower than the baseline.

Performance Evaluation

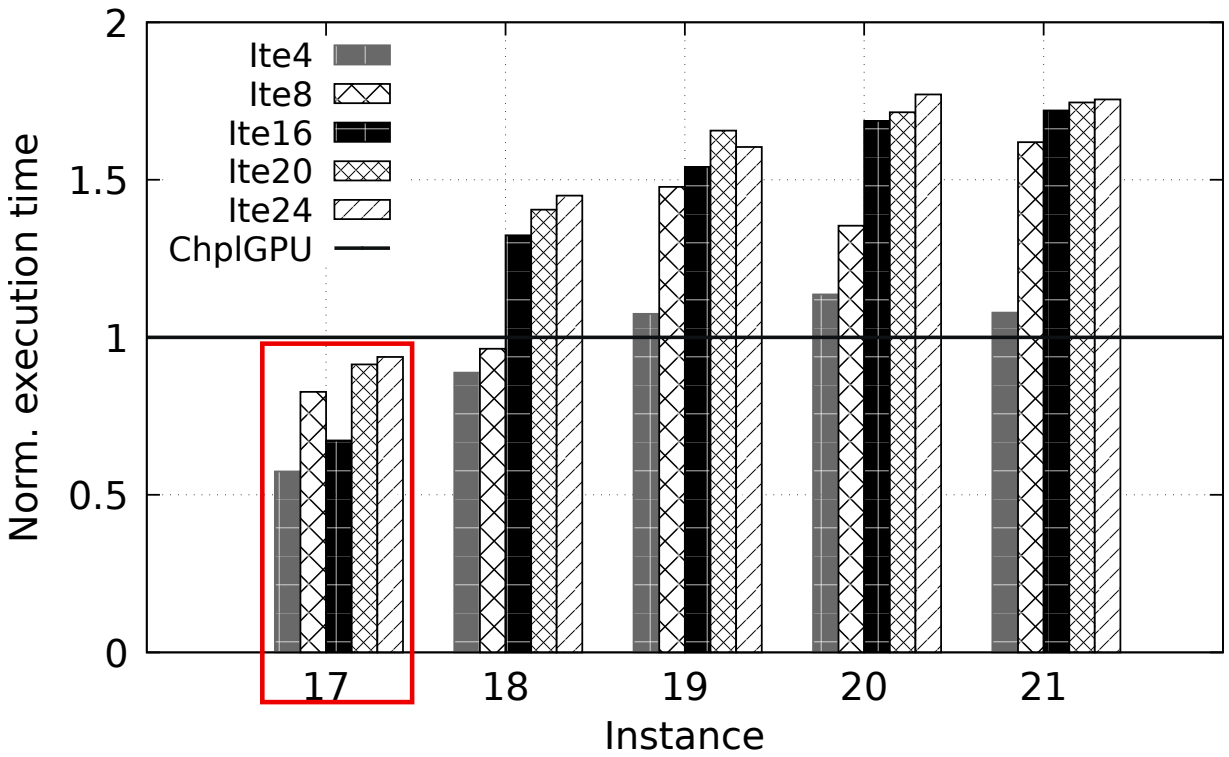
Distributed performance: for 4 GPUs (2 computer nodes) to 24 GPUs (12 computer nodes).



- **For the smallest instance**, that take only a few seconds, GPUIterator is faster than ChplGPU.

Performance Evaluation

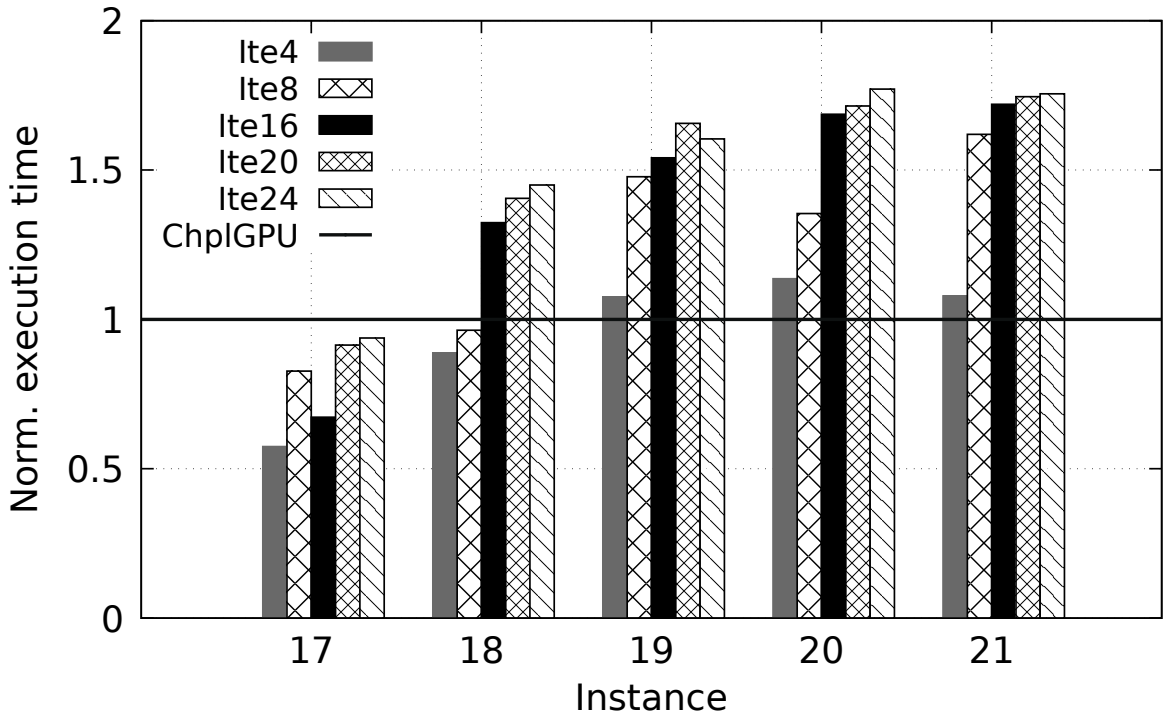
Distributed performance: for 4 GPUs (2 computer nodes) to 24 GPUs (12 computer nodes).



- **For the smallest instance**, that take only a few seconds, GPUIterator is faster than ChplGPU.
- **As more GPUs (nodes) are added...**

Performance Evaluation

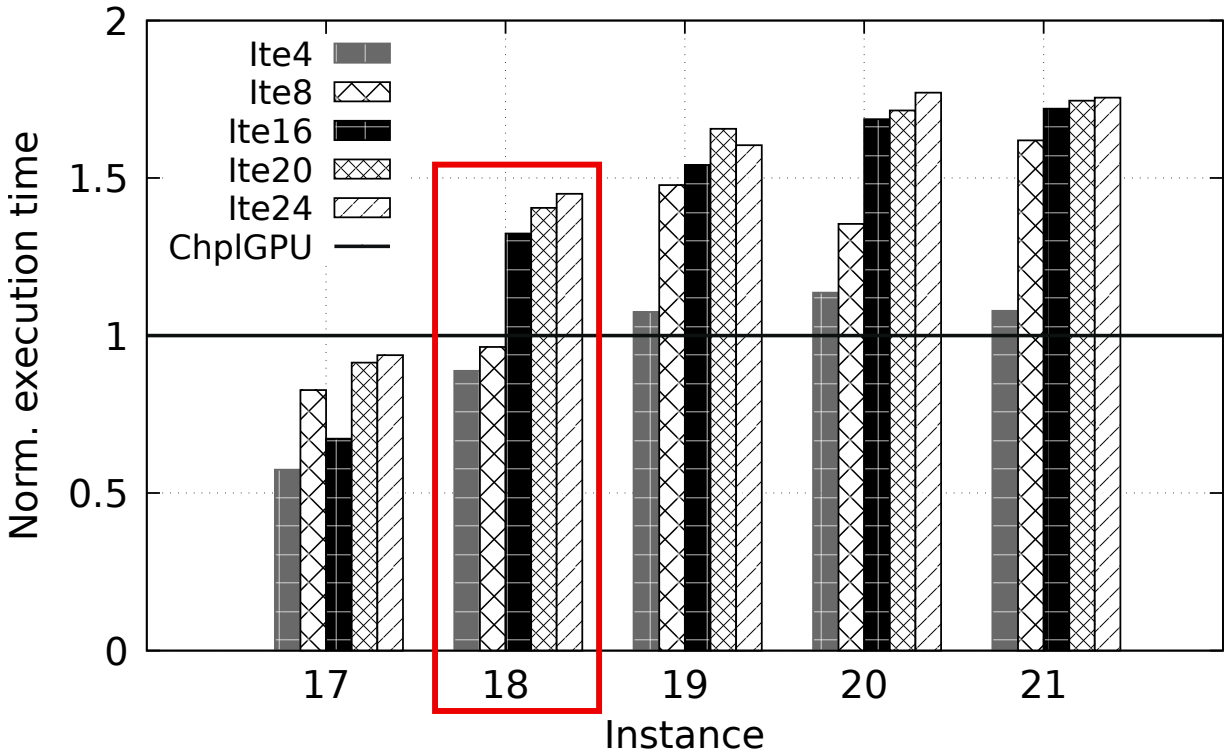
Distributed performance: for 4 GPUs (2 computer nodes) to 24 GPUs (12 computer nodes).



- GPUIterator performs poorly due to the **lack of distributed load balancing**.

Performance Evaluation

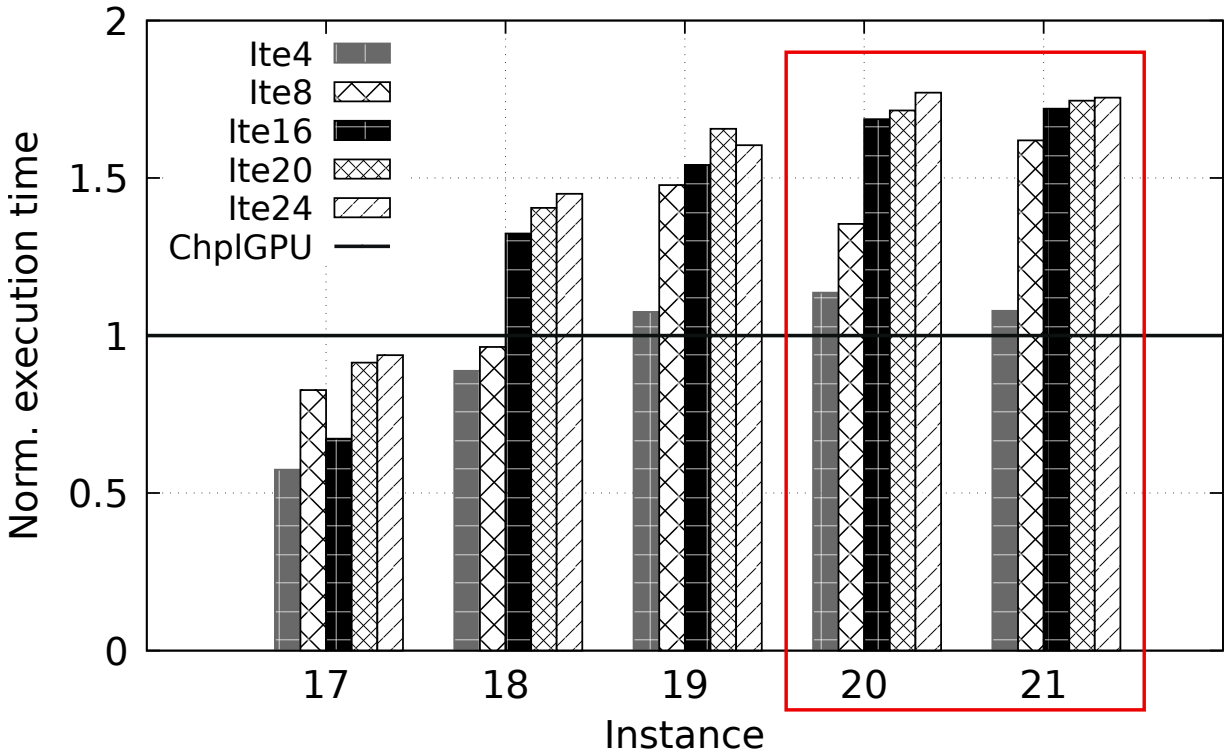
Distributed performance: for 4 GPUs (2 computer nodes) to 24 GPUs (12 computer nodes).



- For $N = 18$ and 16–24 GPUs, ChplGPU is from 1.32×–1.45× faster.

Performance Evaluation

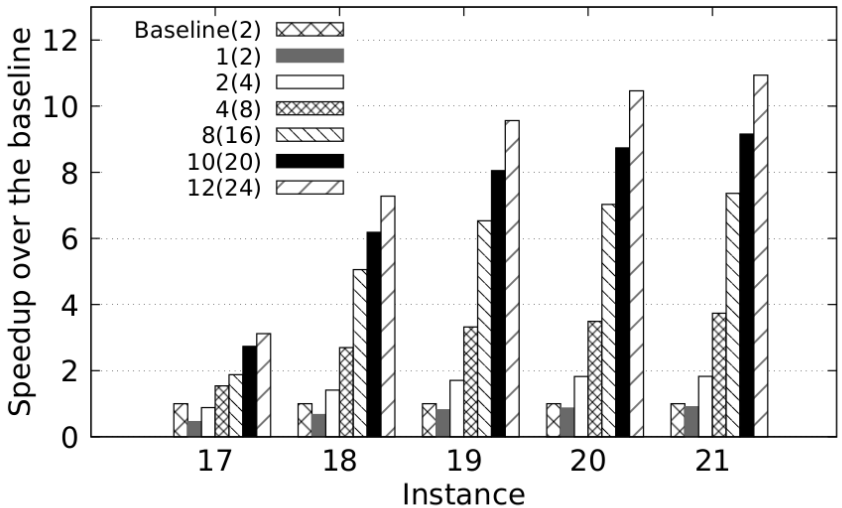
Distributed performance: for 4 GPUs (2 computer nodes) to 24 GPUs (12 computer nodes).



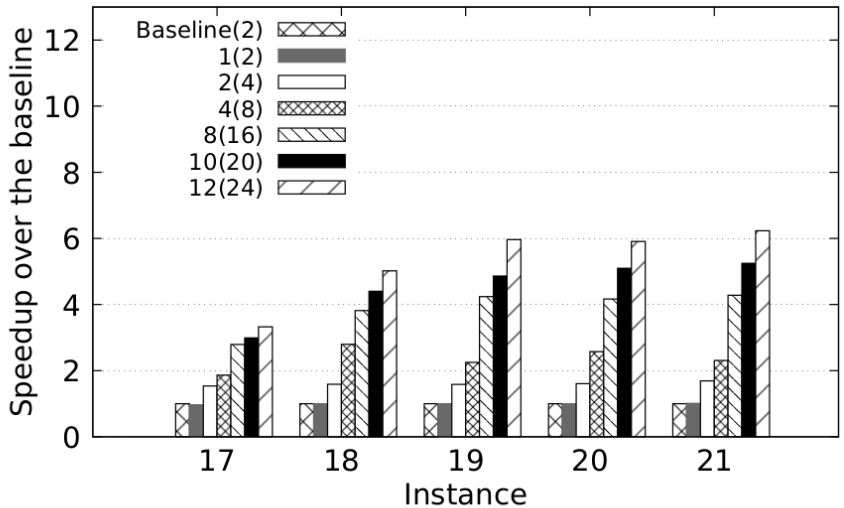
- For $N \geq 20$, ChplGPU is **from** 1.13× **(4 GPUs) to** 1.77× **(24 GPUs)** faster than its counterpart.

Performance Evaluation

Efficiency: speedup compared to the CUDA-C baseline.



(a) ChplGPU *vs.* Baseline (CUDA-C)

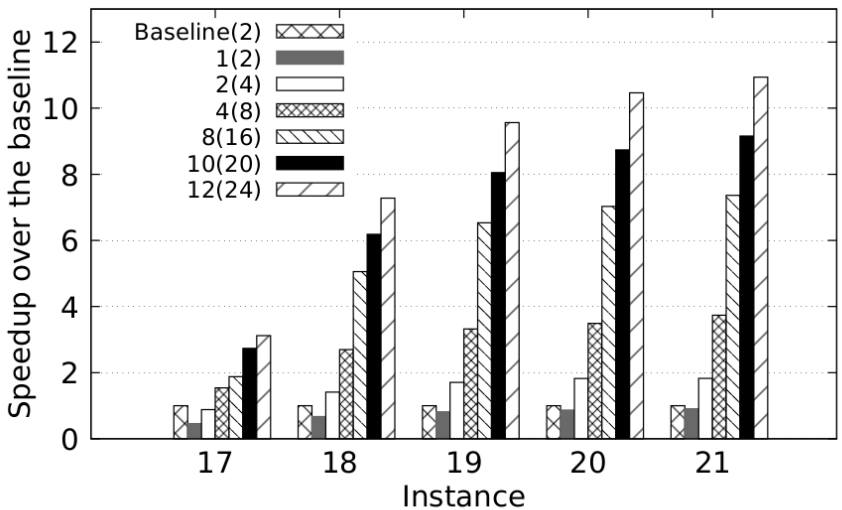


(b) GPUiterator *vs.* Baseline (CUDA-C)

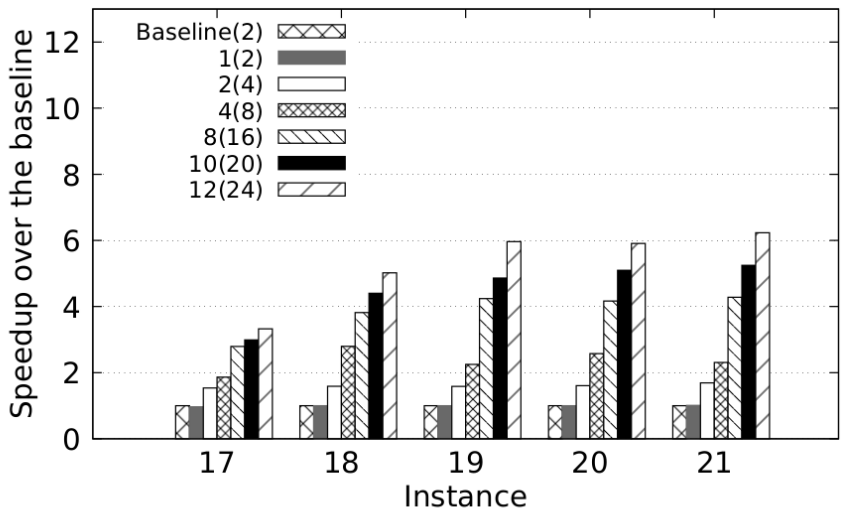
- As a consequence... For $N \geq 18$, the **speedups** achieved by the **GPUIterator** implementation are around 50% **of the linear** speedup.

Performance Evaluation

Efficiency: speedup compared to the CUDA-C baseline.



(a) ChplGPU *vs.* Baseline (CUDA-C)



(b) GPUiterator *vs.* Baseline (CUDA-C)

- In turn, for the ChplGPU implementation, when $N \geq 19$, the speedups achieved for all $< problem, \#GPUS >$ configurations are at least 80% of the linear speedup. For $N \geq 20$ rangd from 87% to 91%.

Agenda

Discussion

GPU Iterator:

- The best option for a single-locale implementation.
- Best option in terms of time to a first implementation.
- Low programming effort to get a distributed version.

Error-prone details are hidden:

- Pointer arithmetics
- Load distribution (CPU-GPU, GPUs, and locales)
- Small SLOC count (8 lines+)

However, poor scalability in irregular tree search.
Good solution for programming distributed heterogeneous and regular applications (*).

Discussion

ChplGPU:

- Nested parallelism.
- GPU-related load distribution by hand.
- Much more complex.

Error-prone details explicitly programmed:

- 1.5× longer than using the iterator.
- GPU load distribution.
- Pointer arithmetics.

Pays off: scales much better than its counterpart.

Discussion

However, some challenges concerning the use of GPUS remain, **even using a high-productivity language**:

Challenges:

- CPU-GPU heterogeneity.
- How to generate on CPU load enough for all CPU/GPUs?
- How to perform load balancing between all CPU/GPUs using iterators?

First research direction : incorporate work-stealing mechanism into Hayashi's GPUIterator module.

Future Research Directions

Research directions:

- Extend the current implementation to a distributed B&B.
- Solving challenging COPs (FSP, Q3AP, etc.).

Road Towards Exascale:

- The scalability should be increased
- CPU-GPU heterogeneity?
- Fault tolerance. Checkpointing?

Thank you!

Questions ?

https://github.com/tcarneirop/ChOp
https://github.com/ahayashi/chapel-gpu

tiago.carneiropessoa@uni.lu