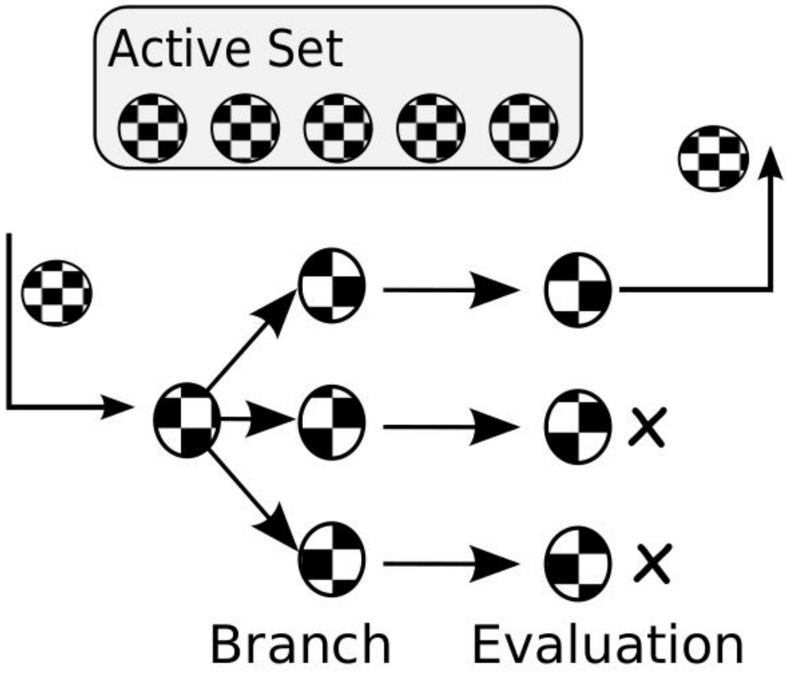
Towards Ultra-scale Exact Optimization Using Chapel

T. Carneiro, N. Melab

Inria Lille – Nord Europe, CNRS - CRIStAL
Parallel Computing & Optimisation Group (PCOG) – University of Luxembourg

Context: tree-based search algorithms

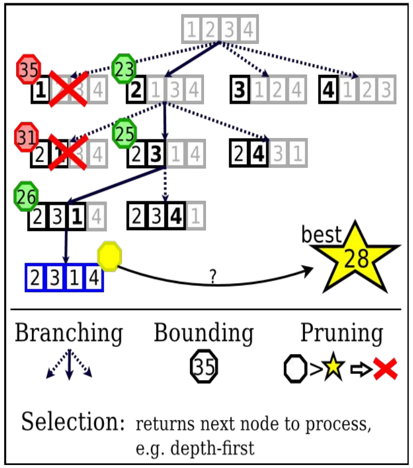


- **e.g., Backtracking and B&B**
- 4 operators
  - Branching, Bounding, Pruning, and Selection (DFS, BFS, …)
- Major properties of the search tree
  - **Very large**
    - **Billions** of tree nodes
  - **Highly dynamic** (pruning+branching)
    - Unpredictable
  - **Highly irregular** tree (pruning)
    - … in shape and size
    - 95% of tree nodes are pruned

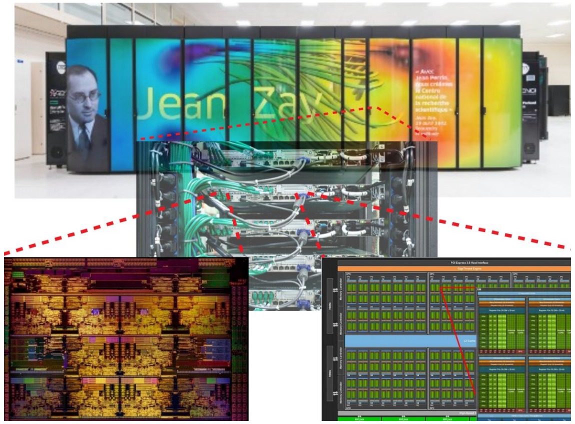
Overall objectives

- Revisit the design and implementation of parallel tree-based search for solving big permutation-based COP to optimality on “ultra-scale” supercomputers dealing with …
  - both **scalability** and **heterogeneity** …
  - … with **productivity-awareness**.

B&B applied to BOPs
e.g. FSP (50j,20m)
10^{64} sub-problems



Supercomputer **(e.g. Jean-Zay (IDRIS)**
85.000+ CPU cores, 2.696 V100 GPUs



Research questions

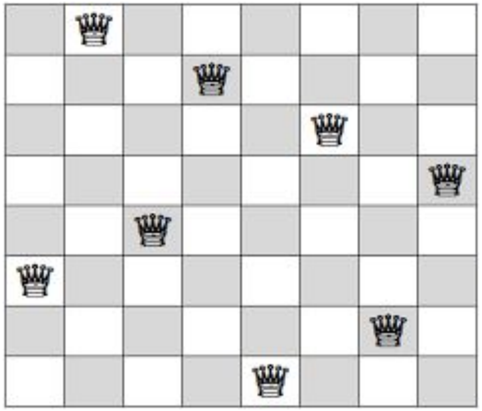
- **Research questions:**
  - Which HPC **programing language/environment favors both productivity and performance?**
  - How to address **scalability** and **heterogeneity** while keeping productivity?

What do we expect from high-productivity lang.?

- Performance
  - Competitive to both C-OpenMP and MPI+X
- Interoperability with C
  - Legacy code (e.g, instance generator)
  - Complex code (e.g., bounding function)
  - Using accelerators (e.g., CUDA)
- Distributed programming features
  - One-sided communication
  - Hide the communication aspects (PGAS)
  - Work distribution

Prototype multi-locale tree search in Chapel

- **Is Chapel feasible for irregular tree search?**
  - Prototype application.
  - Incrementally conceived from a multicore one
  - Chapel high-level features for distributed programming
  - Load balancing, using ***distributed iterators***
  - **The simplest permutation-based:** N-Queens problem

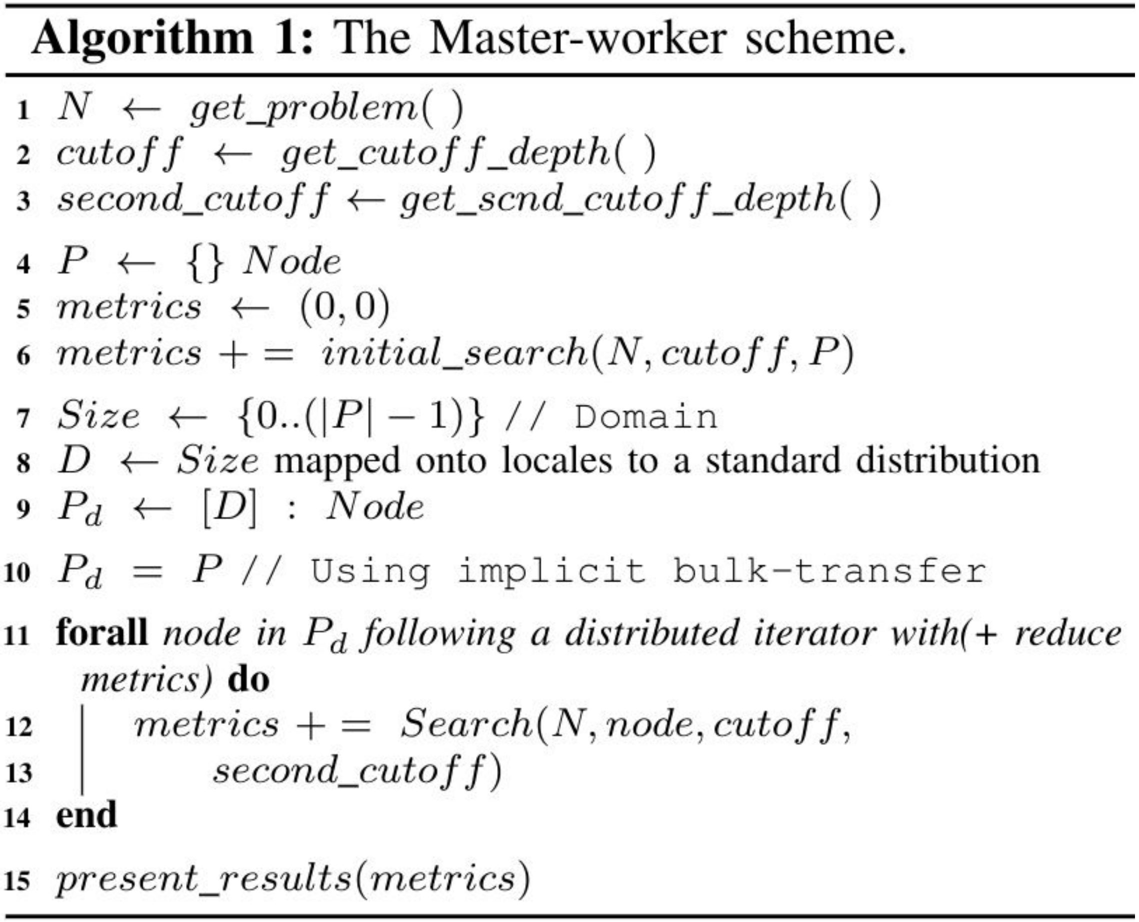


- **Objectives:**
  - Performance *vs.* MPI+OpenMP
  - Programming cost *vs.* MPI+OpenMP
  - Scalability *vs.* MPI+OpenMP
  - **Extend it** for solving a more difficult problem

A PGAS-based tree search algorithm

Partial search generates an initial load (pool data structure)

- Then, the parallel search takes place



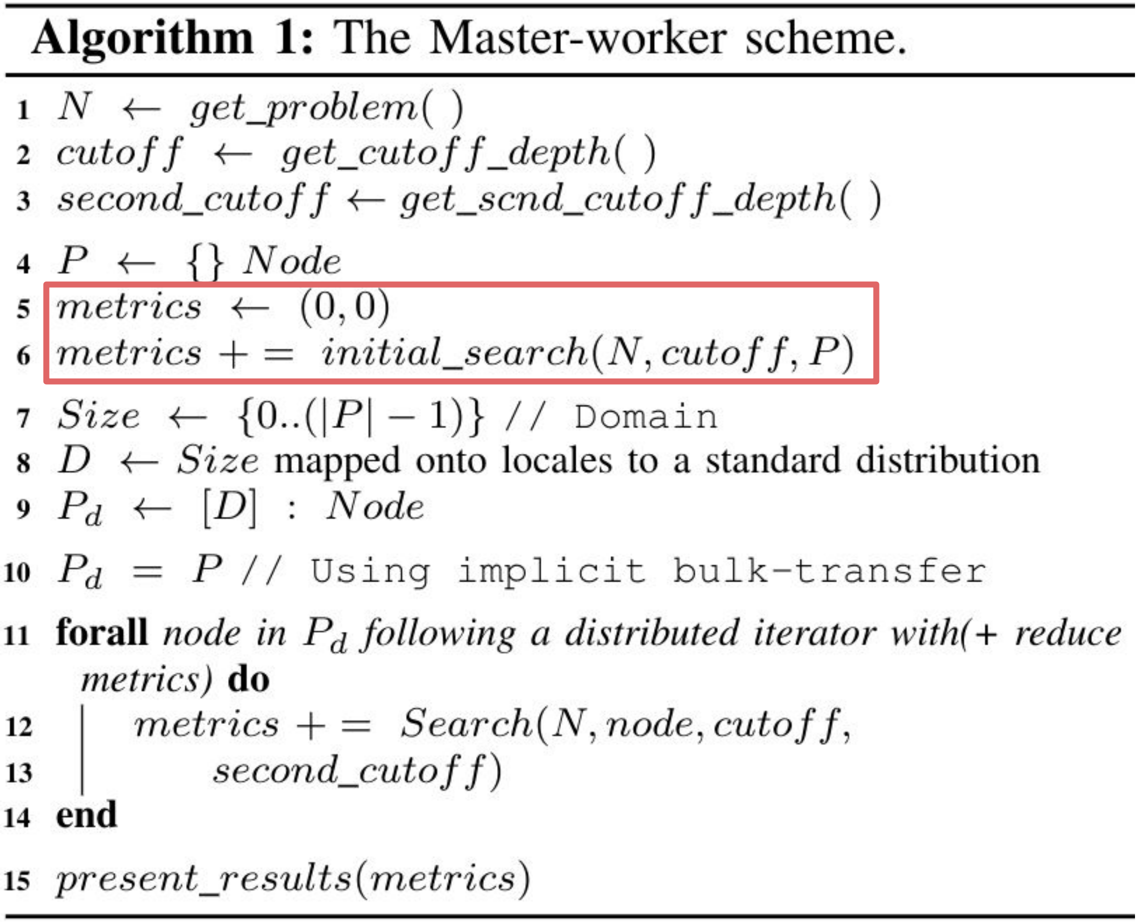
Algorithm 1: The Master-worker scheme.

```
$N \leftarrow get\_problem(\,)$
$cutoff \leftarrow get\_cutoff\_depth(\,)$
$second\_cutoff \leftarrow get\_scnd\_cutoff\_depth(\,)$
$P \leftarrow \{\}\ Node$
$metrics \leftarrow (0, 0)$
$metrics\ += \ initial\_search(N, cutoff, P)$
$Size \leftarrow \{0..(|P| - 1)\}$ // Domain
$D \leftarrow Size$ mapped onto locales to a standard distribution
$P_d \leftarrow [D]\ :\ Node$
$P_d = P$ // Using implicit bulk-transfer
forall $node$ in $P_d$ following a distributed iterator with(+ reduce metrics) do
    $metrics\ += \ Search(N, node, cutoff, second\_cutoff)$
end
$present\_results(metrics)$
```

A PGAS-based tree search algorithm

Partial search generates an initial load (pool data structure)

- Then, the parallel search takes place



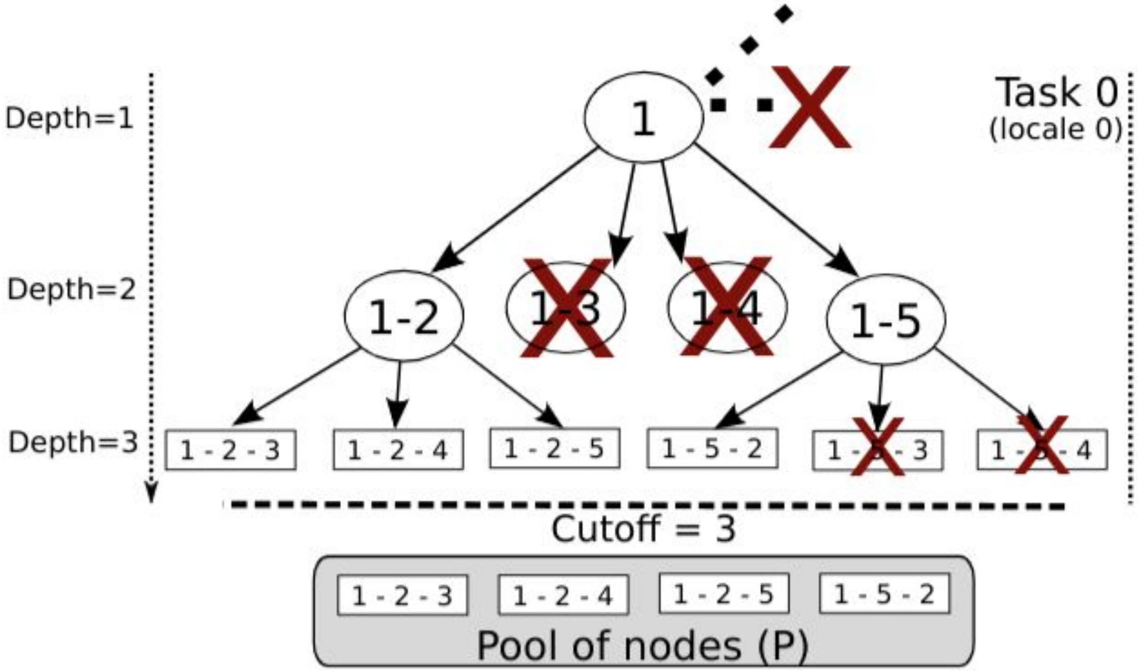
Algorithm 1: The Master-worker scheme.

```
$N \leftarrow get\_problem(\,)$
$cutoff \leftarrow get\_cutoff\_depth(\,)$
$second\_cutoff \leftarrow get\_scnd\_cutoff\_depth(\,)$
$P \leftarrow \{\}\ Node$
$metrics \leftarrow (0,0)$
$metrics \mathrel{+}= initial\_search(N, cutoff, P)$
$Size \leftarrow \{0..(|P|-1)\}$ // Domain
$D \leftarrow Size$ mapped onto locales to a standard distribution
$P_d \leftarrow [D] : Node$
$P_d = P$ // Using implicit bulk-transfer
forall node in $P_d$ following a distributed iterator with(+ reduce metrics) do
    $metrics \mathrel{+}= Search(N, node, cutoff, second\_cutoff)$
end
$present\_results(metrics)$
```

A PGAS-based tree search algorithm

Partial (initial) search:

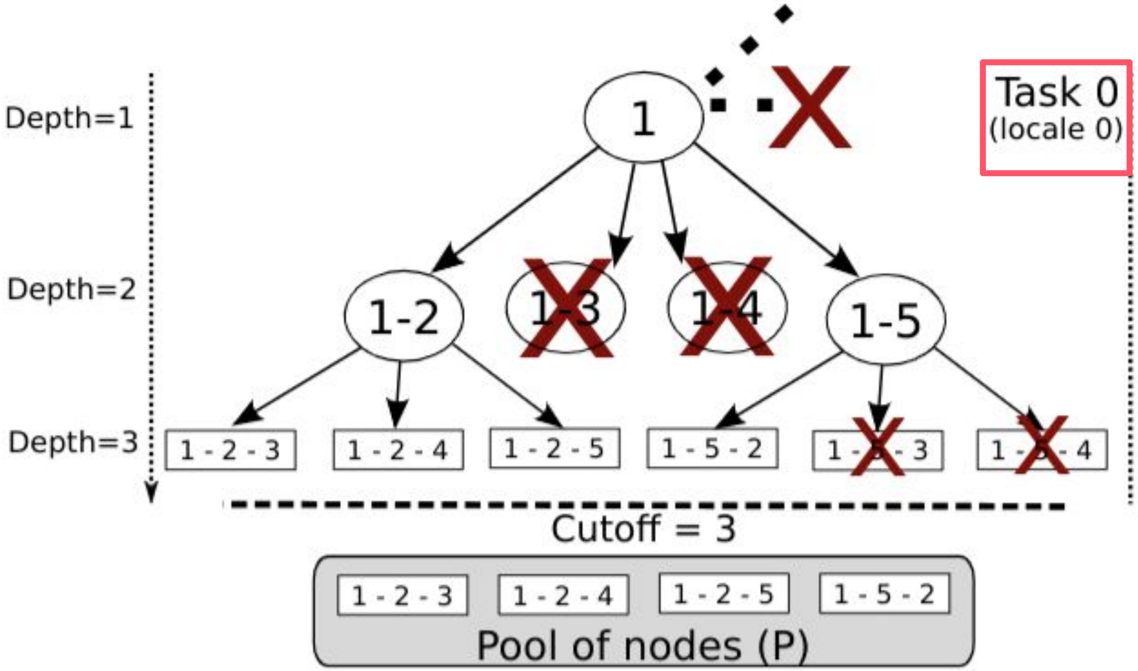
- From depth 1 until the **cutoff** depth (*cutoff* <= *N*)



•A PGAS-based tree search algorithm

Partial (initial) search:

- From depth 1 until the **cutoff** depth (*cutoff* <= *N*)

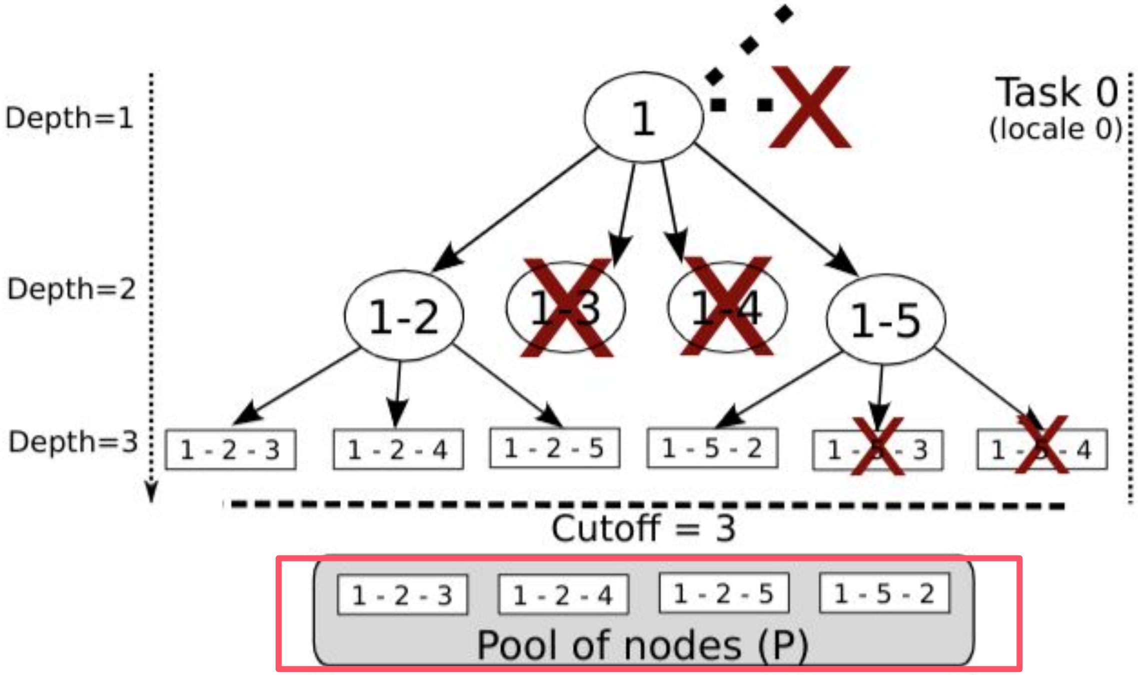


Serial, on *locale* 0 - *task* 0

A PGAS-based tree search algorithm

Partial (initial) search:

- From depth 1 until the **cutoff** depth (*cutoff* <= *N*)



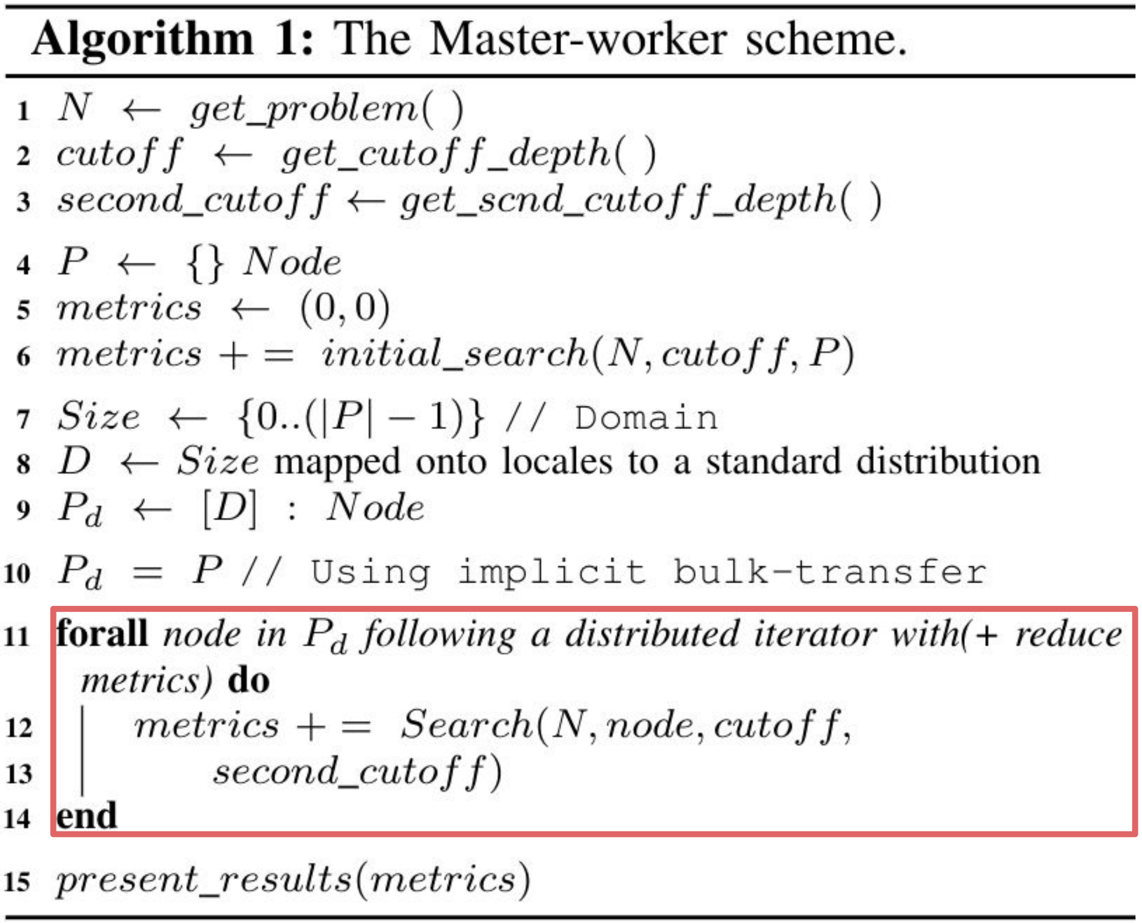
Serial, on *locale* 0 - *task* 0

- Stores *all* feasible, valid and incomplete solutions of size *cutoff*.

A PGAS-based tree search algorithm

Then, parallelism is added though a forall statement

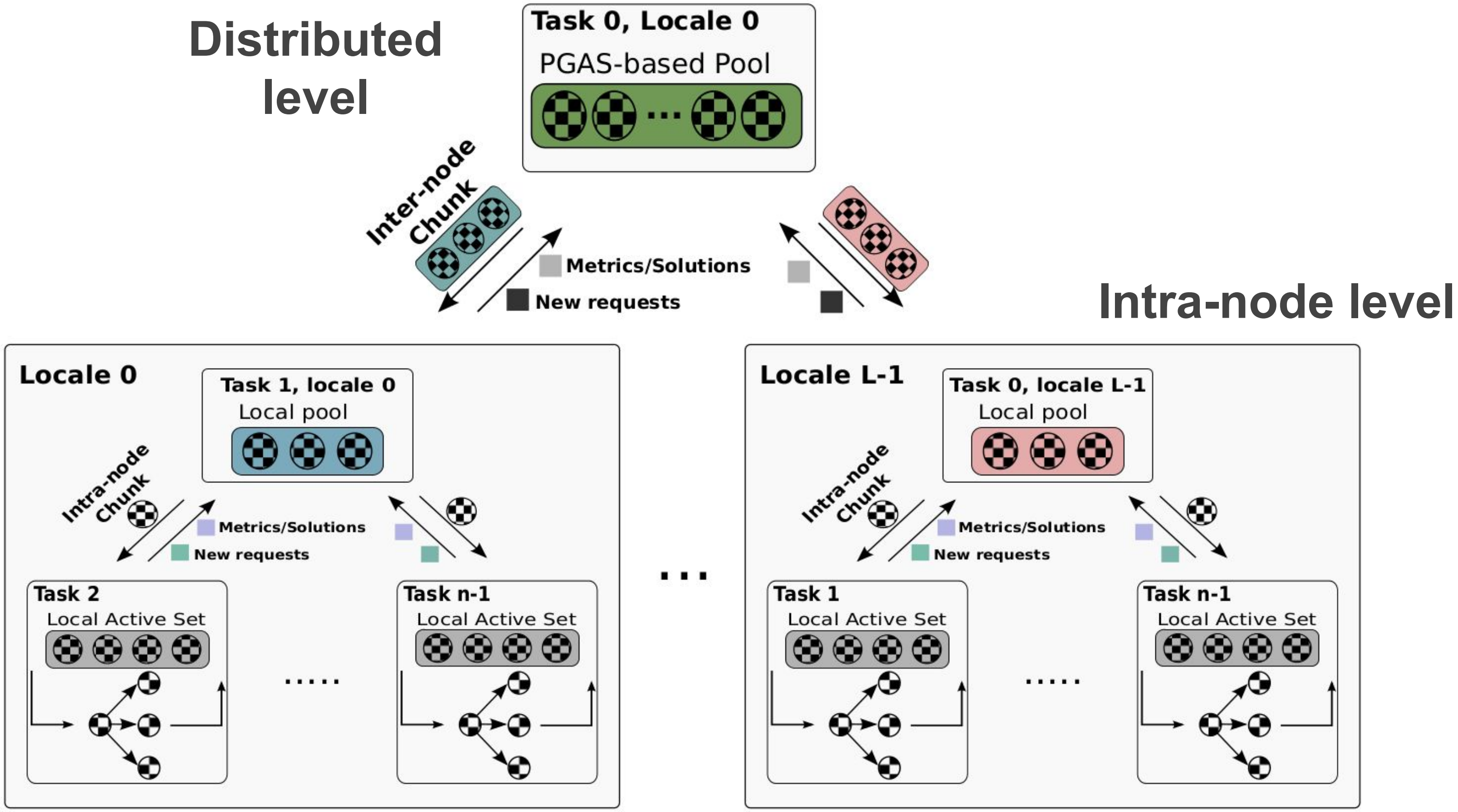
- No need for explicit communication for work distribution and metrics reduction.



Algorithm 1: The Master-worker scheme.

```
$N \leftarrow get\_problem(\ )$
$cutoff \leftarrow get\_cutoff\_depth(\ )$
$second\_cutoff \leftarrow get\_scnd\_cutoff\_depth(\ )$
$P \leftarrow \{\}\ Node$
$metrics \leftarrow (0,0)$
$metrics\ += initial\_search(N, cutoff, P)$
$Size \leftarrow \{0..(|P|-1)\}$ // Domain
$D \leftarrow Size$ mapped onto locales to a standard distribution
$P_d \leftarrow [D]\ :\ Node$
$P_d = P$ // Using implicit bulk-transfer
forall node in $P_d$ following a distributed iterator with(+ reduce metrics) do
    $metrics\ += Search(N, node, cutoff,$
        $second\_cutoff)$
end
$present\_results(metrics)$
```

A PGAS-based tree search algorithm



Centralized pool of nodes

First multi-locale implementation: N-Queens

PGAS approach is close to its high-level representation

```
...
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
MPI_Get_processor_name(processor_name, &name_len);
...
int r_start = range_start(proc_id,survivors,num_procs);
int r_end = range_end(proc_id,survivors, num_procs);
int chunk = get_mpi_chunk(proc_id,survivors,num_procs);
...
local_metrics += queens_initial_search(....);
...
#pragma omp parallel for ... schedule(dynamic) reduction(+...)
for(int idx = r_start; idx<r_end ;++idx)
    ...
...
MPI_Reduce(...);
MPI_Reduce(...);
MPI_Finalize();
```

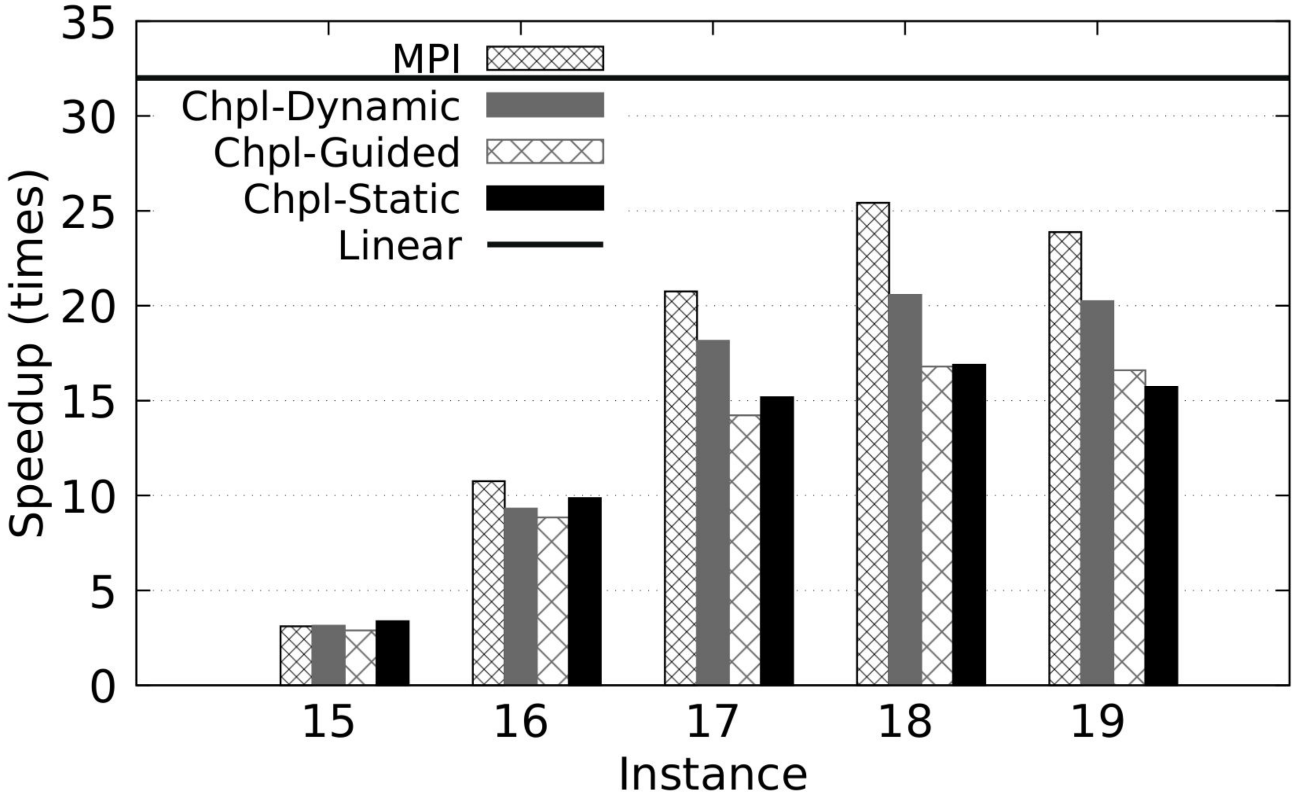
```
const Space = {0..(number_nodes-1)};
const D: domain(1) dmapped Block(boundingBox=Space) = Space;
var A_d: [D] queens_node;

metrics += queens_initial_search(size,initial_depth,A);

forall idx in distributedDynamic(c=Space, chunkSize=chunk) with (+
     ↪ reduce metrics) do
    metrics += queens_node_exporer(size,initial_depth,A_d[idx]);
```

PGAS Model (Chpl)

Distributed memory
(MPI+OpenMP)

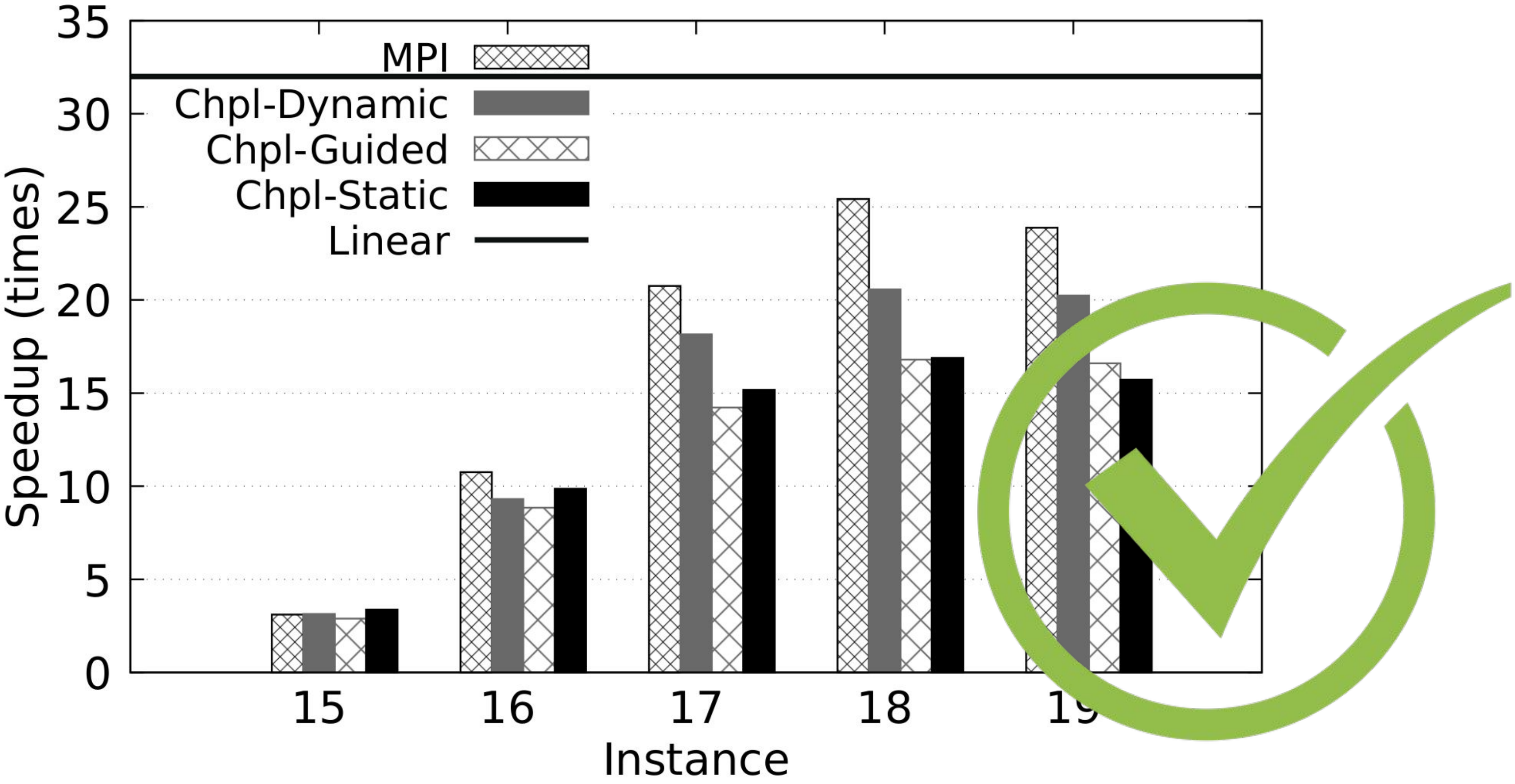
First multi-locale implementation: N-Queens

32 locales: 384 cores/768 threads. two Intel Xeon X5670 @ 2.93 GHz (total of 12 cores/24 threads). Infiniband network.



First multi-locale implementation: N-Queens

32 locales: 384 cores/768 threads. two Intel Xeon X5670 @ 2.93 GHz (total of 12 cores/24 threads). Infiniband network.



Improving intra-node parallelism

- Compiler-generated intra-node code is efficient for regular/weakly irregular applications.
  - e.g. Backtracking applied to NQueens [*Carneiro and Melab, HPCS'2019*]

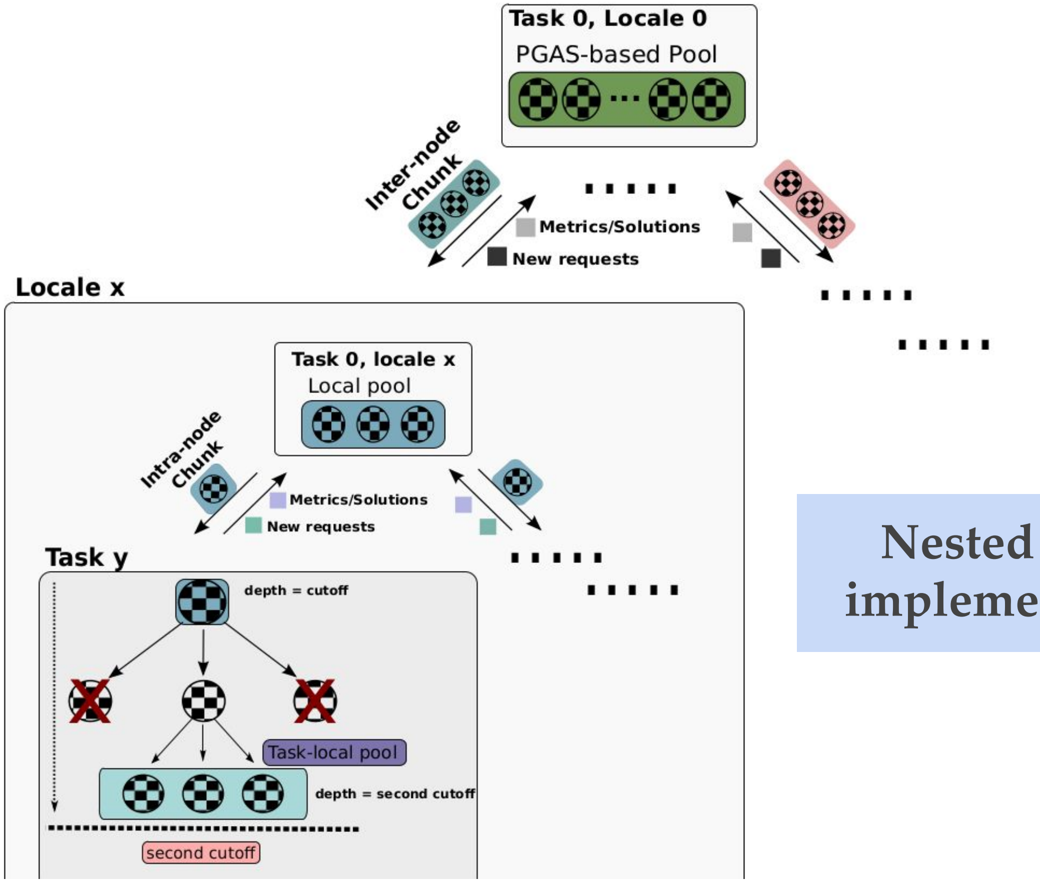
- … but not for highly irregular applications (e.g. B&B applied to FSP)

  □ Work units are coarse-grained (highly irregular)

  □ Intra-node parallelism should be hand-defined

Improving intra-node parallelism

- Bi-level intra-node parallelism
  - The task chunk is decomposed (2nd cutoff depth)
    - Local task pool distributed according to Dynamic WP



Nested parallelism implemented by hand

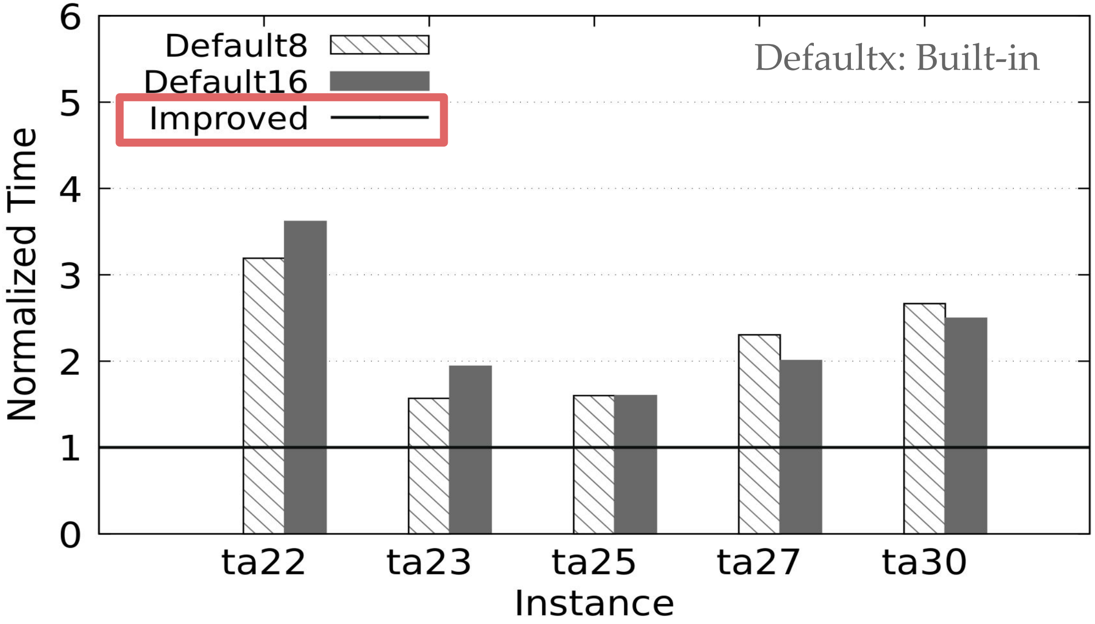
Improving intra-node parallelism

- Compiler-generated intra-node code is efficient for regular/weakly irregular applications.
  - e.g. Backtracking applied to NQueens [*Carneiro and Melab, HPCS'2019*]

- … but not for highly irregular applications (e.g. B&B applied to FSP)

□ Work units are coarse-grained (highly irregular)

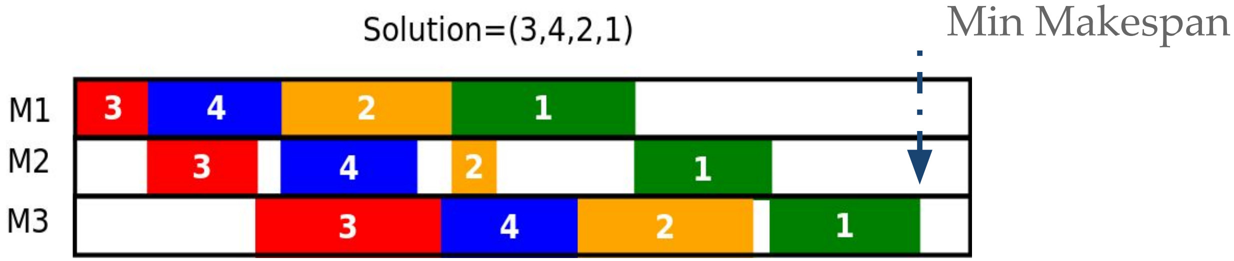
□ Intra-node parallelism should be hand-defined



Defaultx: Built-in

Problem Instances

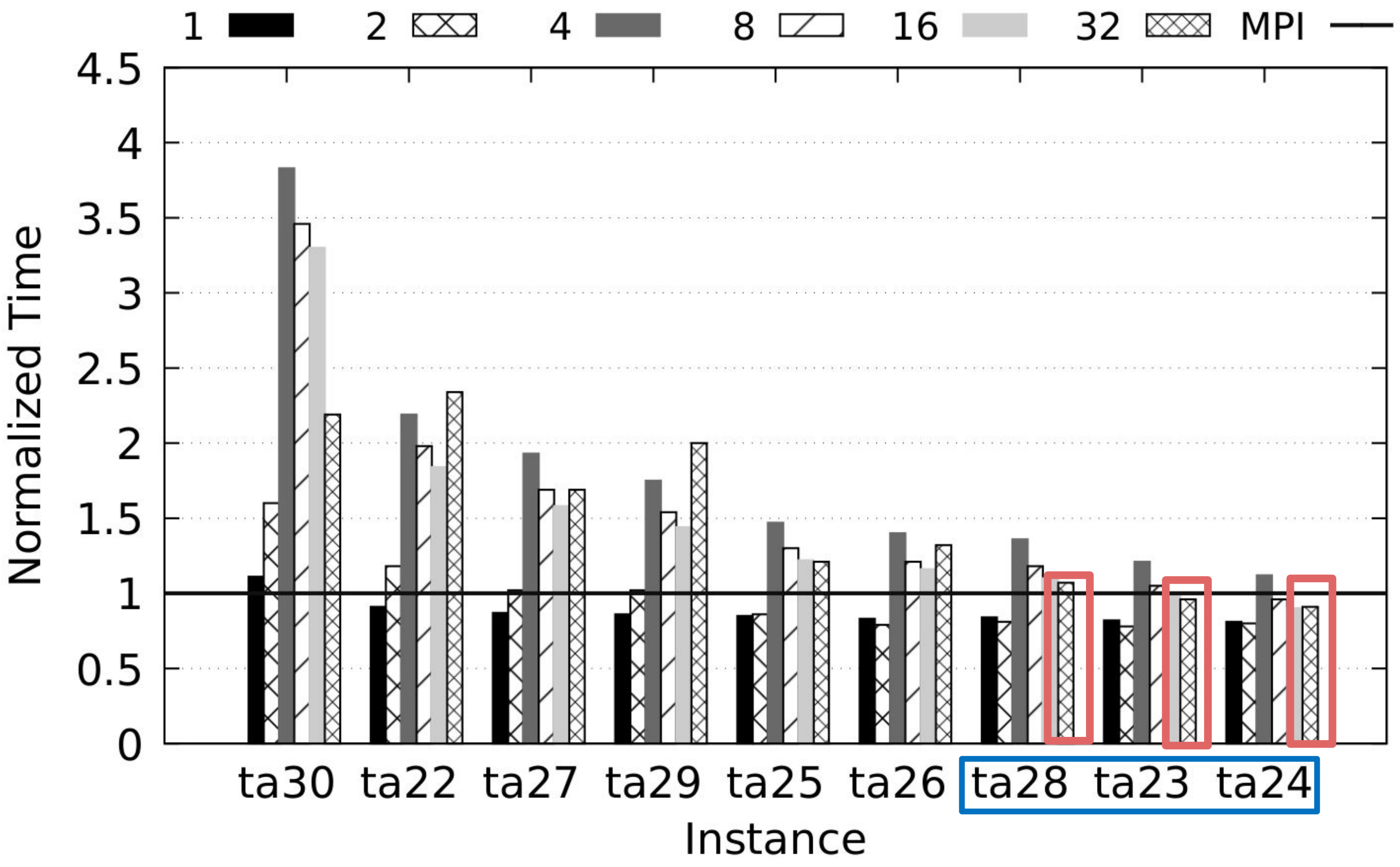
- FSP Instances
  - 9 *Taillard*'s instances, N=20 jobs on M=20 machines
  - Ranked according to their complexity (*#decomposed sub-problems*)
  - *Vs.* an MPI+Pthreads **state of the art** B&B *[Gmys et al. 2019]*



Instance-#	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
$\mathbf{NN}_{LB1}$ (10^6) $\mathbf{T}_{LB1}$ (sec)	711 120	37 200 6400	71 876 11 460	5208 970	11 392 1750	1854 320	12 285 2100	3018 490	111 20

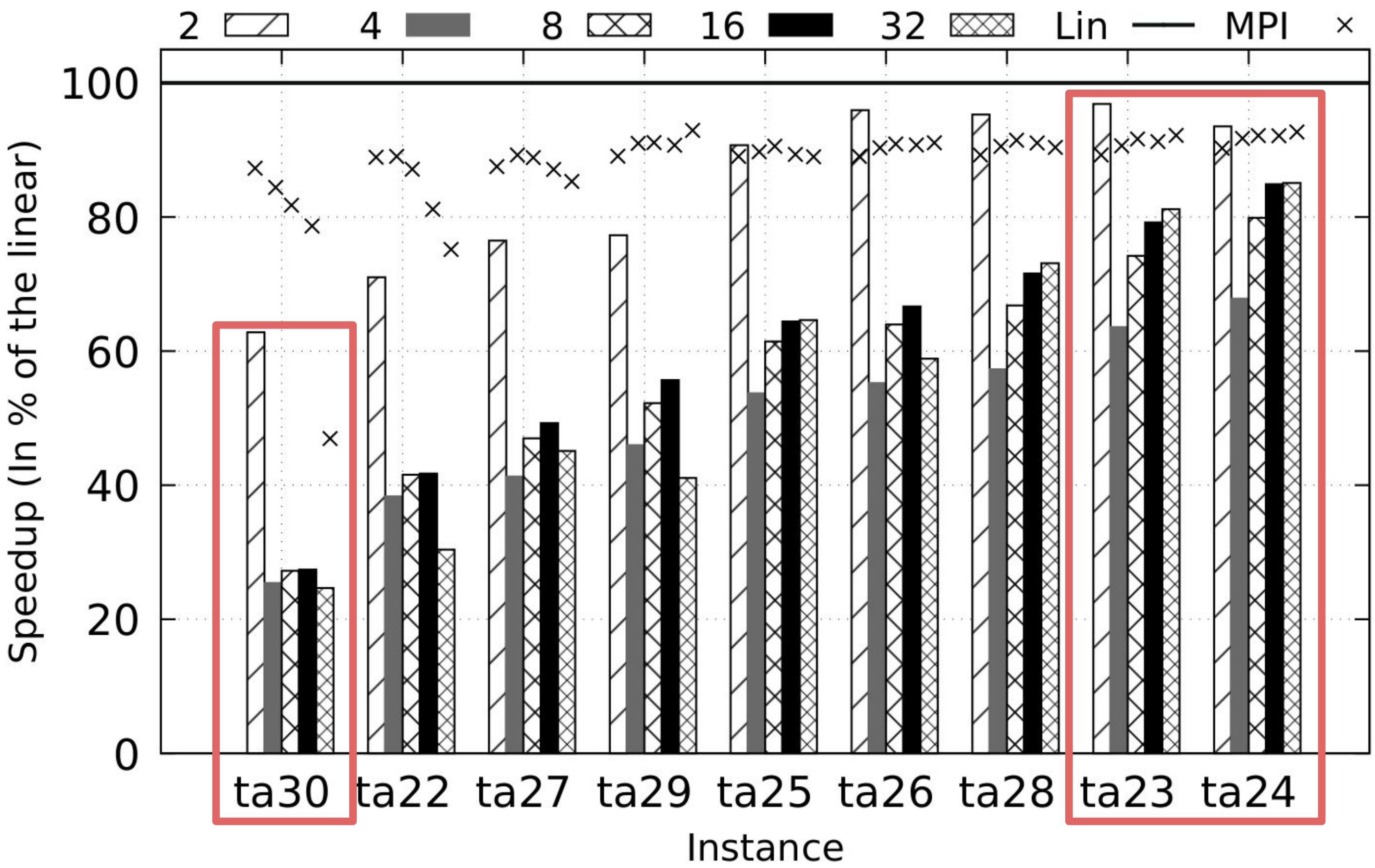
Chapel-BB *vs.* MPI-PBB: execution time

- For big instances, Chapel-BB is slightly faster/equivalent than/to MPI-PBB with 32 locales (1024 cores)



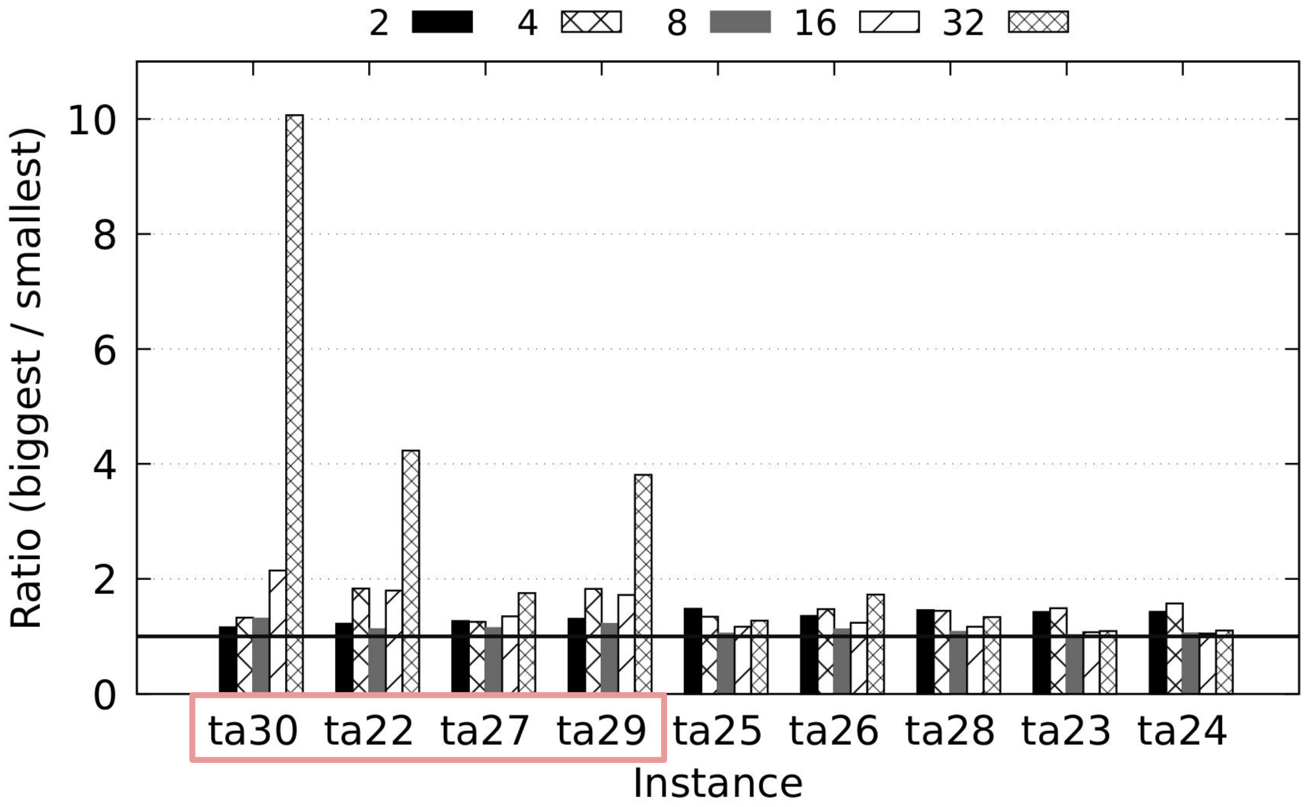
Chapel-BB *vs.* MPI-PBB: scalability

- Speed-ups from 24.5% to 85% of the linear one on 32 locales
- For small instances, not enough work to feed the locales



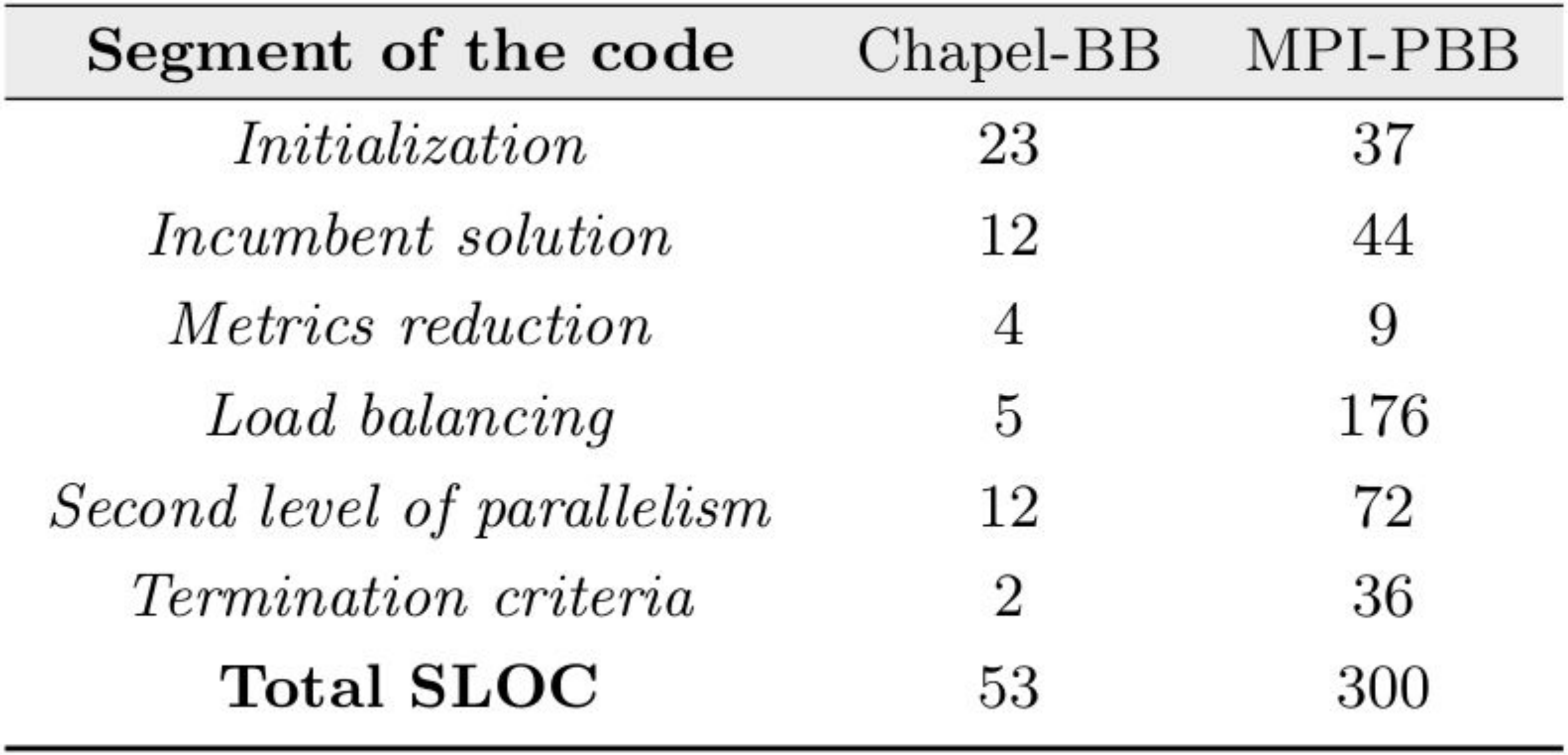
Built-in load balancing should be improved

- Small instances are highly irregular
  - … in decomposition activity (#decomposed tree nodes)
  - WS implemented in MPI-PBB (***state-of-the-art***) but not in Chapel-BB



A Productivity-oriented evaluation: cost

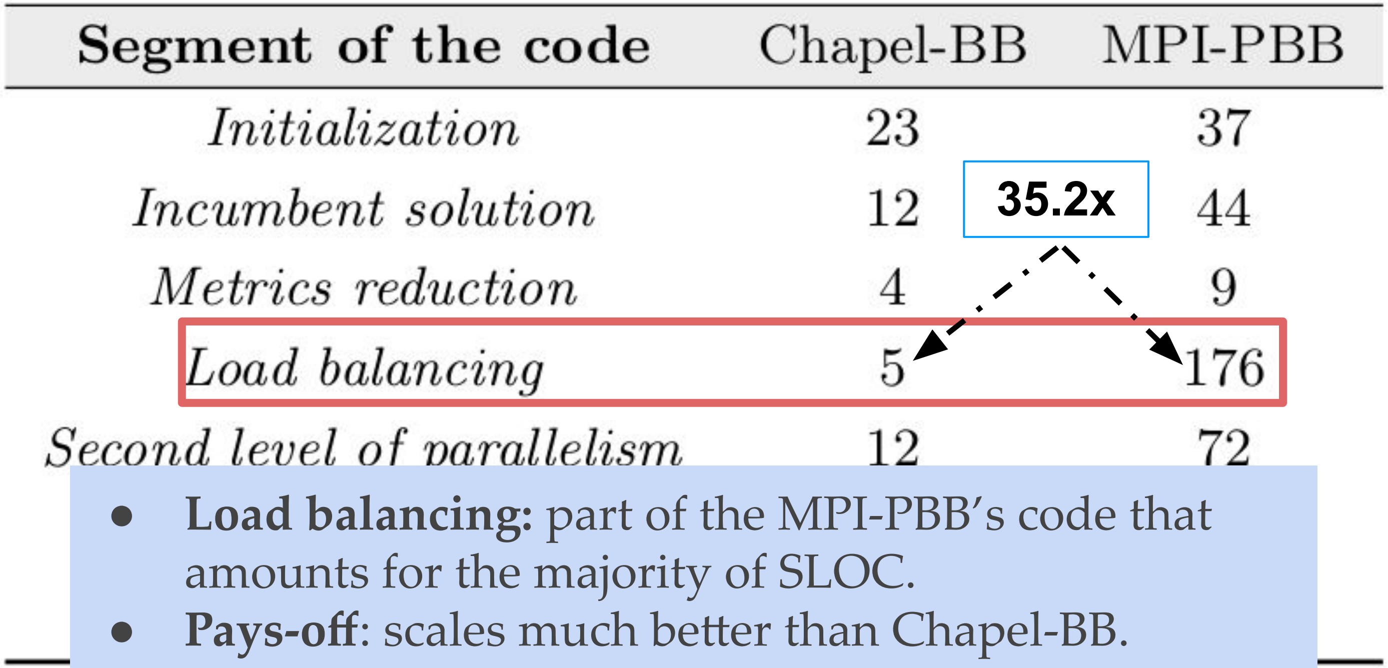
- **Implementation cost**:



Segment of the code	Chapel-BB	MPI-PBB
Initialization	23	37
Incumbent solution	12	44
Metrics reduction	4	9
Load balancing	5	176
Second level of parallelism	12	72
Termination criteria	2	36
Total SLOC	53	300

A Productivity-oriented evaluation: cost

- **Implementation cost:**



Segment of the code	Chapel-BB	MPI-PBB
Initialization	23	37
Incumbent solution	12	44
Metrics reduction	4	9
Load balancing	5	176
Second level of parallelism	12	72

35.2x

- **Load balancing:** part of the MPI-PBB's code that amounts for the majority of SLOC.
- **Pays-off**: scales much better than Chapel-BB.
- Chapel-BB uses built-in iterators.

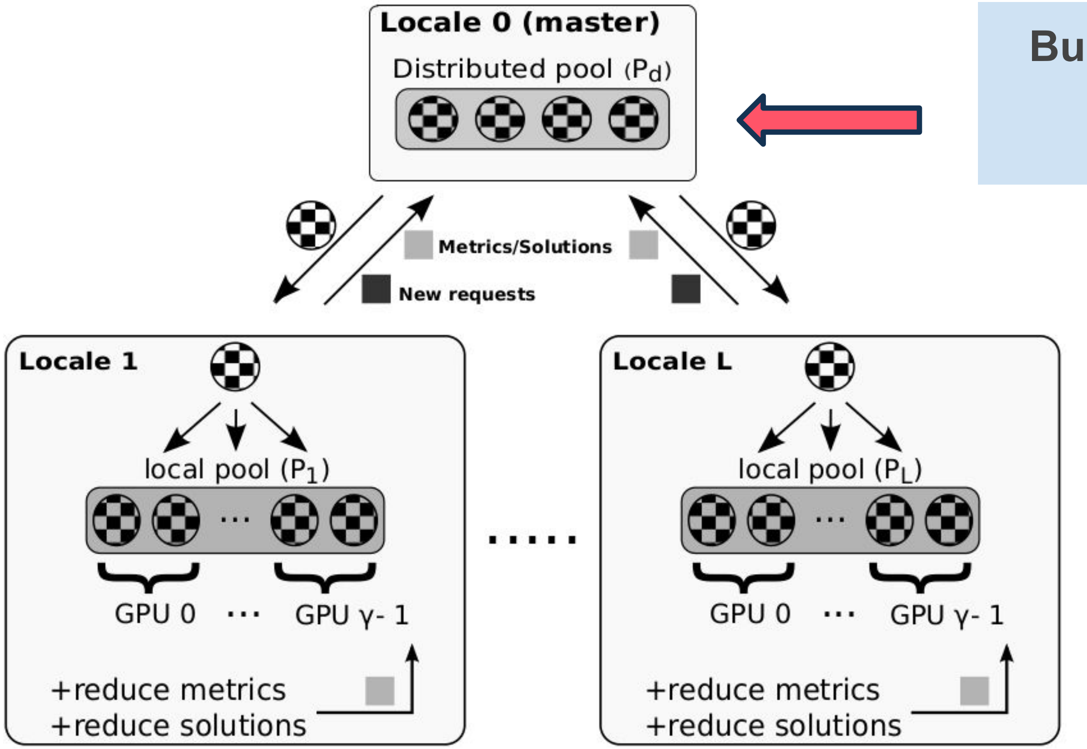
Extending the implementation for GPUs

- **GPUs:**
  - **Crucial** nowadays in exact optimization
  - Allow one to solve instances with prohibitive execution time on CPUs [*Gmys et al. 2020, 2021*]
  - Energy-efficient → power wall
  - Chapel does not officially support GPUs

- **Implementation:**
  - **We can not use the *GPUIterator* module**: lack of load balancing
  - Adapted the improved intra-node scheme for GPUs
  - Communication in Chapel + intra-node in CUDA + Chpl
  - **Prototype:** N-Queens

Extending the implementation for GPUs

- **Extension for GPUs:** combining high-level and CUDA kernels
  - Collaboration with Habanero Extreme Scale Software Research Lab, **Georgia Tech** (*A. Hayashi and V. Sarkar*).



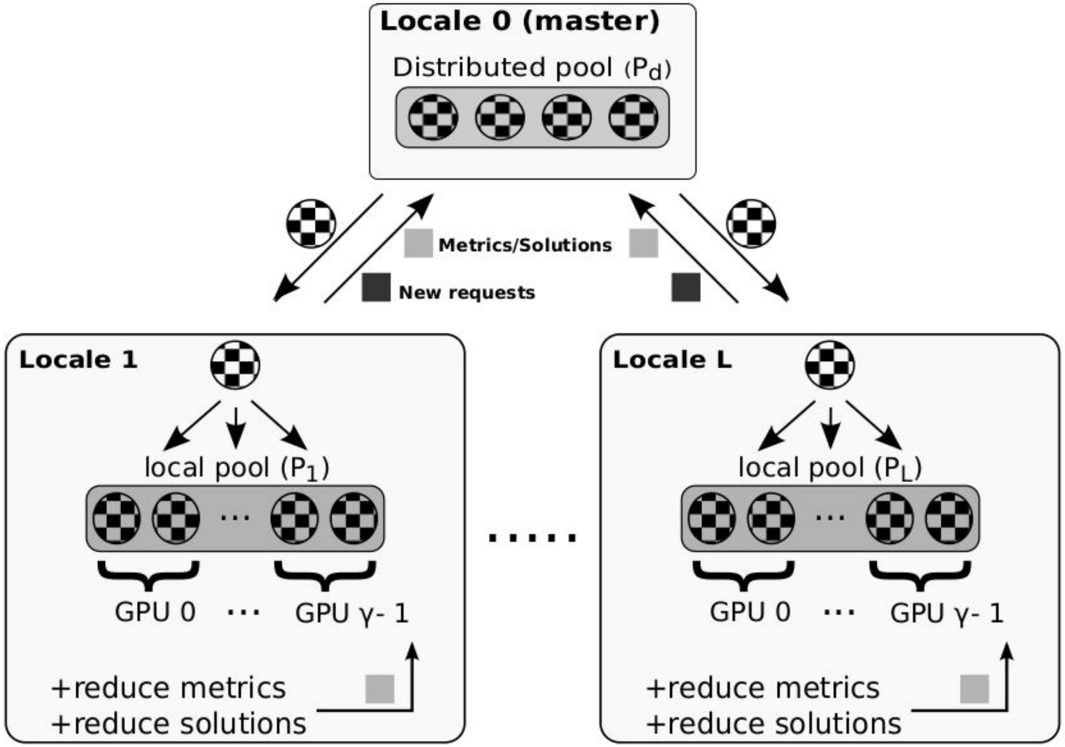
Built in distributed load balancing/work distribution

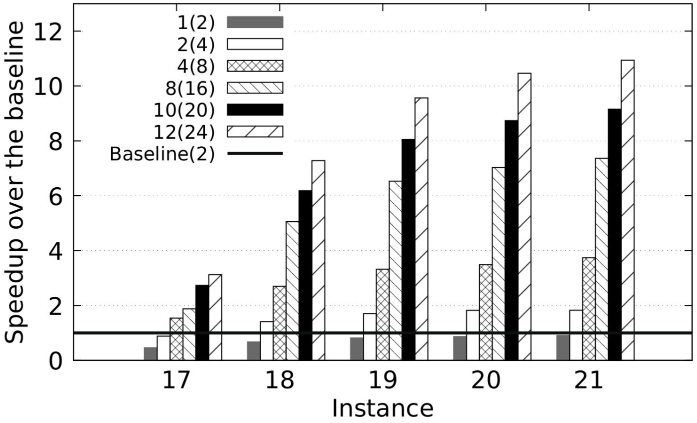
Intra-node: Chapel + CUDA

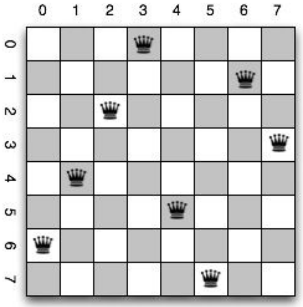
Work distribution by hand → intra-node productivity lost

Extending the implementation for GPUs

- **Extension for GPUs:** combining high-level and CUDA kernels
  - Collaboration with Habanero Extreme Scale Software Research Lab, **Georgia Tech** (*A. Hayashi and V. Sarkar*).



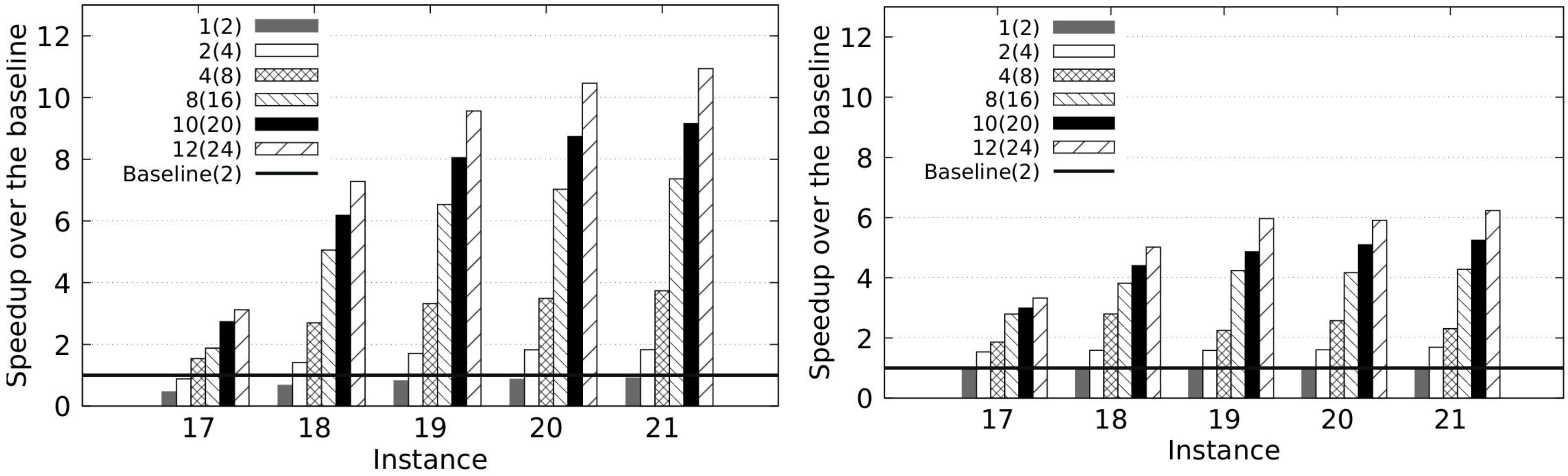




T. Carneiro, N. Melab, A. Hayashi. V. Sarkar, *Towards Chapel-based Exascale Tree Search Algorithms: dealing with multiple GPU accelerators* HPCS 2020 (2021).

Extending the implementation for GPUs

- **Proposed implementation vs. *GPUIterator*-based**
    - The *GPUIterator*-based implementation cannot scale due to its lack of load balancing.



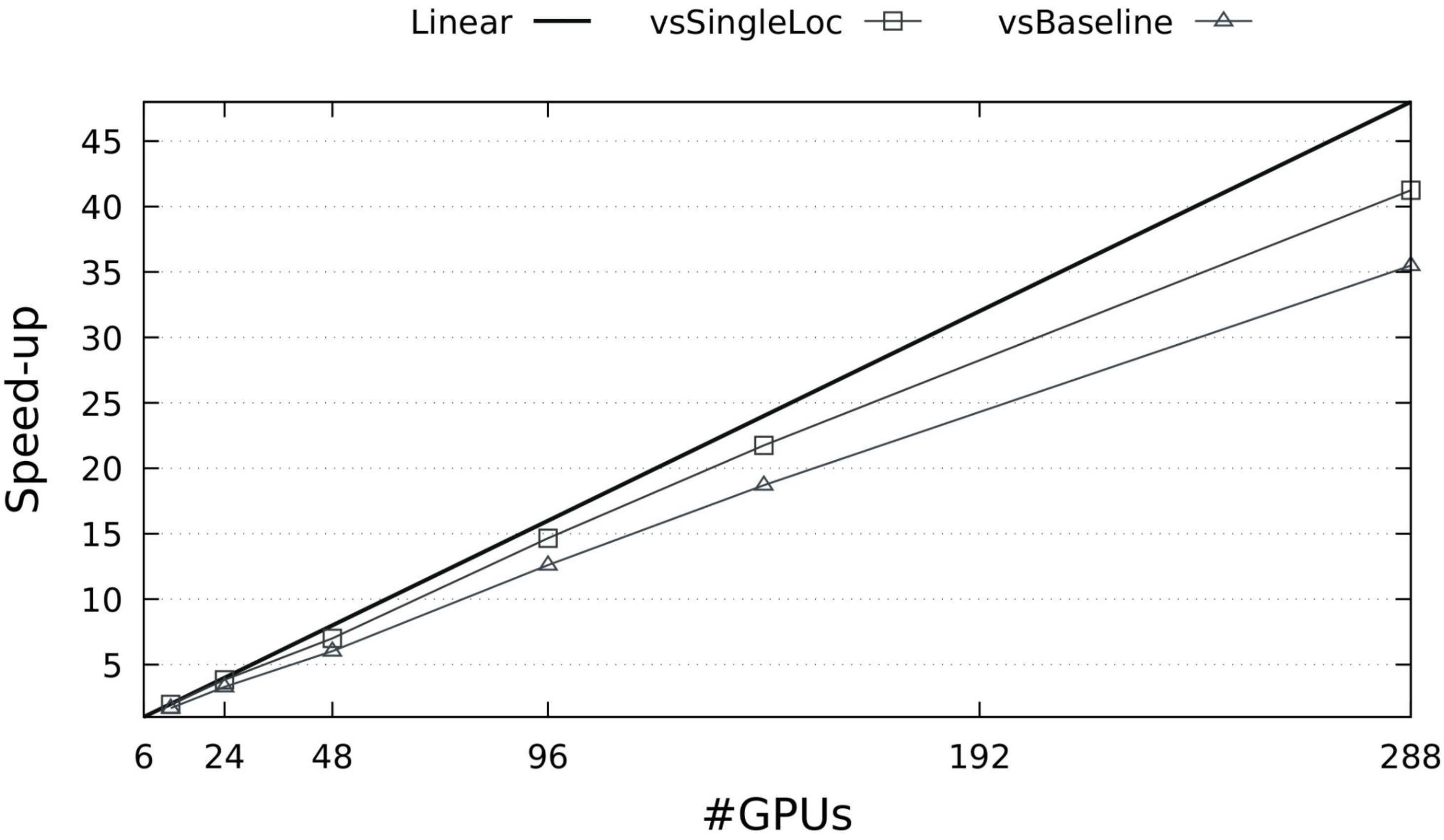
(a) ChplGPU *vs.* Baseline (CUDA-C)

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

T. Carneiro, N. Melab, A. Hayashi. V. Sarkar, *Towards Chapel-based Exascale Tree Search Algorithms: dealing with multiple GPU accelerators* HPCS 2020 (2021).

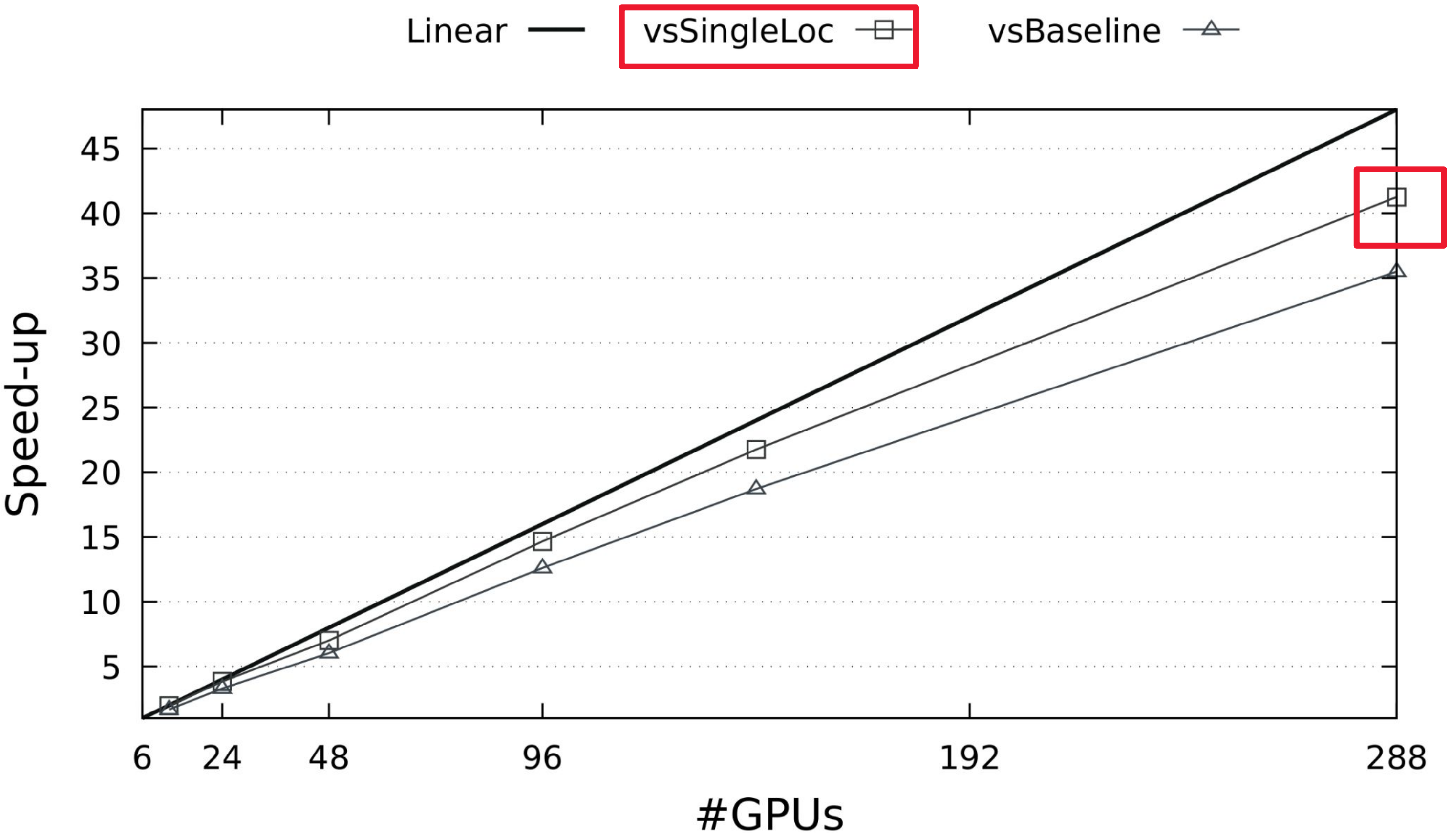
Extending the implementation for GPUs

- **First large-scale experiments: 20-Queens** ***(39,029,188,884 solutions)***
  - Up to 288 GPUs
  - 6 GPUs per node, 48 nodes used



Extending the implementation for GPUs

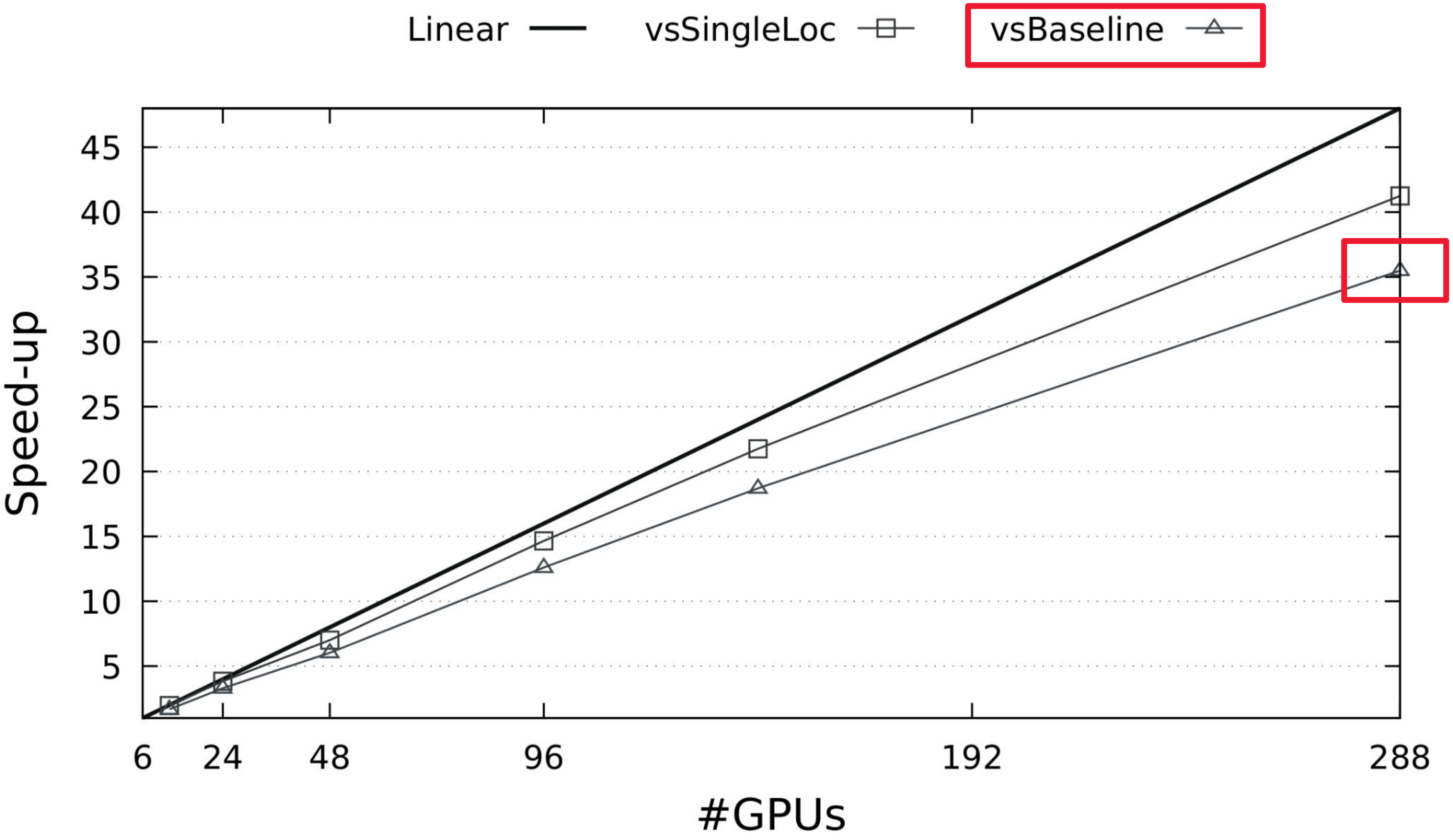
- **First large-scale experiments: 20-Queens** *(39,029,188,884 solutions)*
  - Up to 288 GPUs
  - 6 GPUs per node, 48 nodes used



86% of the linear speedup

Extending the implementation for GPUs

- **First large-scale experiments: 20-Queens** *(39,029,188,884 solutions)*
  - Up to 288 GPUs
  - 6 GPUs per node, 48 nodes used



74% of the linear speedup

Conclusions

- Chapel for the design and implementation of heterogeneous distributed tree search for solving BOPs
  - **Need to hand-redefine some features (*hierarchical parallelism*)**
  - **Use C-Interoperability layer**

- Programming "cost"
  - 5.7x "less costly" than MPI+X (*X=PThreads*)
  - Built-in load balancing
  - **Thanks to the global view:** implicit termination and reduction, no additional library, transparent communication, etc.

- Efficiency and scalability
  - Competitive efficiency and scalability compared to MPI+X for big instances on 1.024 cores **… but can be up to 3.8x slower**
  - **Limitations:** PGAS-based data distribution, communication, LB, etc.

Future Works

- Investigating the **Work Stealing**-based load balancing
  - Inspired by the WS of the state-of-the-art of MPI-PBB
  - Provide it as an iterator
- Heterogeneity and productivity: the *GPUIterator* module
  - How to harness both the CPUs and GPUs of the system?
  - Error-prone details implemented by hand (CUDA + Chpl)
  - Incorporate WS into the *GPUIterator* module
- Fault tolerance using checkpointing
  - Rarely addressed in parallel optimization although critical (Mean Time Between Failures - MTBF < 1h)
  - **Issues:** recovery strategy (what, when and where?), restart strategy (with consistent global state)? GPU?

Thank you!

https://github.com/tcarneirop/ChOp