Locality-Aware Scheduling in OpenMP

Overview of thesis topics and deep dive into OpenMP related scheduling improvements

Jannis Klinkenberg

High Performance Computing | IT Center | RWTH AACHEN UNIVERSITY

About myself: Jannis Klinkenberg

- **2010: B.Sc. Scientific Programming (MATSE at RWTH / FH Aachen)**
  – Thesis: Pressure Calculation in Thermo-Dynamic Networks using Simulink and C++

- **2012: M.Sc. Artificial Intelligence at Maastricht University**
  – Focus: Machine Learning, Games and AI, Intelligent Search Techniques
  – Thesis: Strategy for Complex Structured Games Using Kernels and Nearest Neighbor Techniques

- **2012 – 2016: Gaining Experience in Industry**
  – Areas: Software architecture & development for automotive industry and power plant optimization, data management & processing solutions

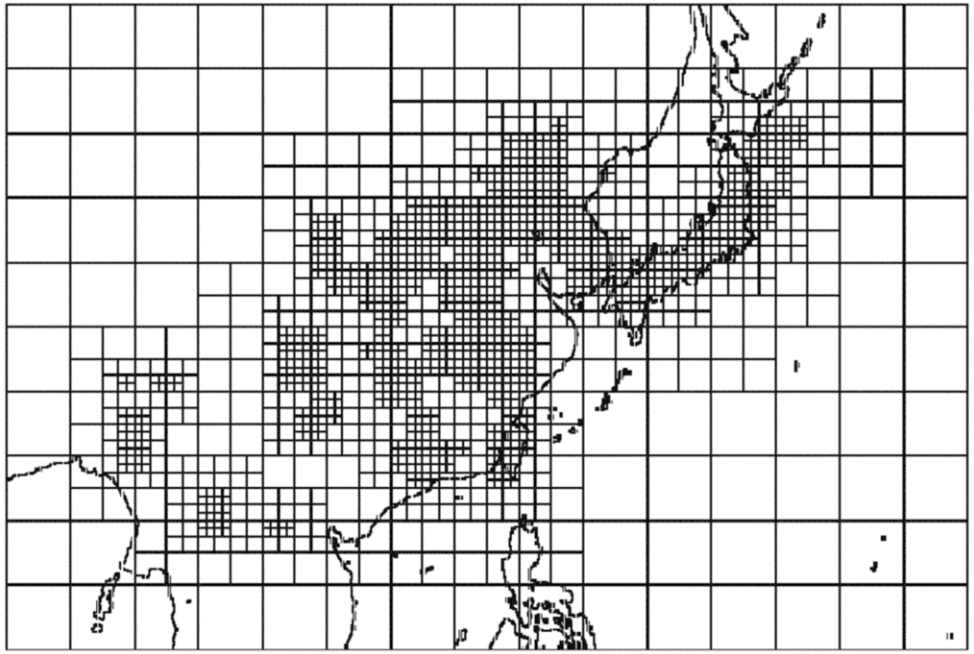
- **Since 2016: Research Assistant / PhD Student at Chair for High Performance Computing**

- **Research: Runtime Improvements for Dynamic, Complex and Heterogeneous Systems**
  – Chameleon: Dynamic load balancing in distributed memory for MPI + OpenMP task parallel programs
  – H2M: Heuristics for heterogeneous memory (together with Inria)
  – OpenMP Co-Chair of Affinity Subcommittee
  – ML / DL: Failure prediction and performance prediction

Motivation for Thesis Topics

- **Increasing complexity of today‘s HPC systems and software**
  - ➢ Performance variability
  - ➢ Load imbalance
  - ➢ Harder for users to exploit full potential

- **Examples: Software**
  - Dynamic scheduling
  - Adaptive mesh refinement (AMR)



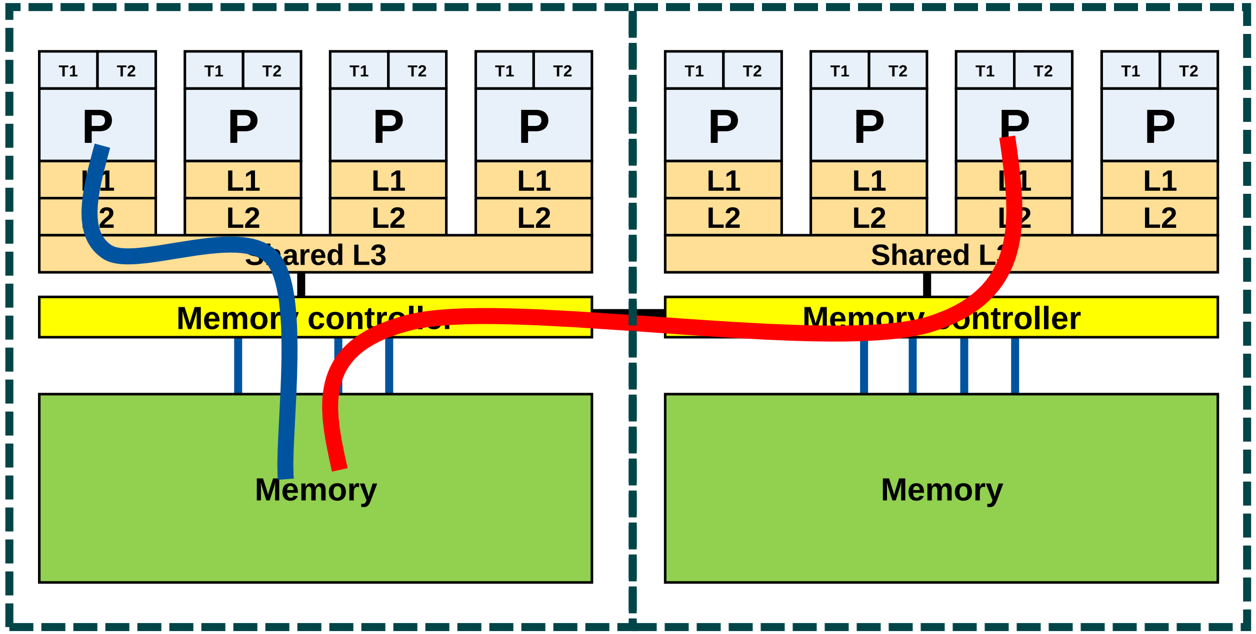
Source: https://doi.org/10.3390/atmos2030484

- Initial domain decomposition
- Depending on situation either refinement or coarsening of cells

Motivation for Thesis Topics

- **Increasing complexity of today‘s HPC systems and software**
  - Performance variability
  - Load imbalance
  - Harder for users to exploit full potential
- **Examples: Software**
  - Dynamic scheduling
  - Adaptive mesh refinement (AMR)
- **Examples: Hardware / Design**
  - NUMA architecture design

2-socket NUMA architecture (simplified)



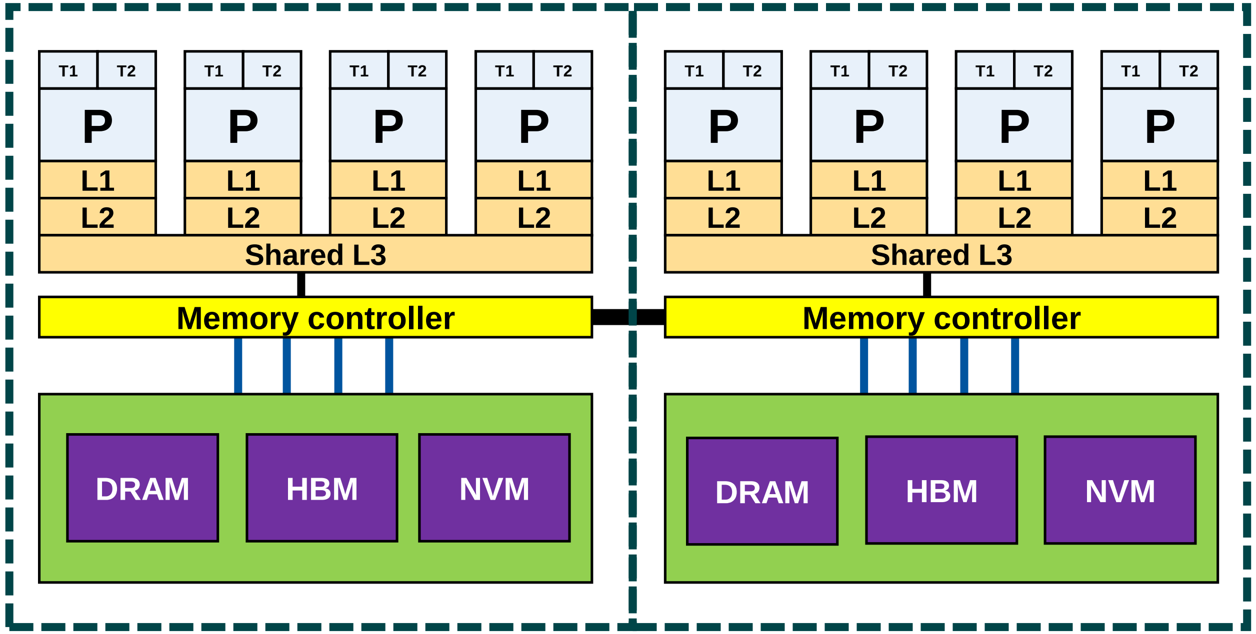
Accessing
local memory is fast

Accessing
remote memory is slower

Motivation for Thesis Topics

- **Increasing complexity of today‘s HPC systems and software**
  - ➢ Performance variability
  - ➢ Load imbalance
  - ➢ Harder for users to exploit full potential
- **Examples: Software**
  - Dynamic scheduling
  - Adaptive mesh refinement (AMR)
- **Examples: Hardware / Design**
  - NUMA architecture design
  - Complex memory hierarchies
    - HBM
    - Non-Volatile Memory (NVM)
    - DRAM

2-socket NUMA architecture (simplified)



- Very different memory characteristics (latency / bandwidth, …)
- **Q:** Where to place data items? When to move data items?
- **Q:** How to minimize overhead for data movement?

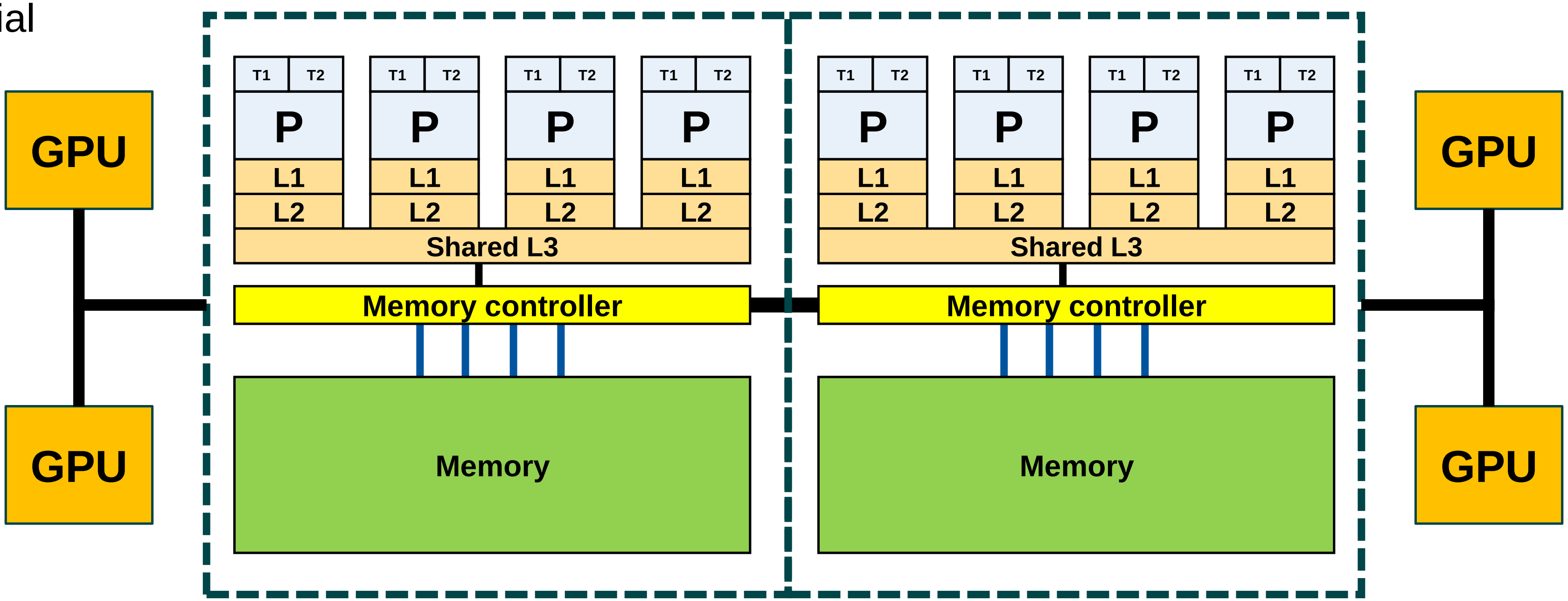
Motivation for Thesis Topics

- **Increasing complexity of today‘s HPC systems and software**
  - ➢ Performance variability
  - ➢ Load imbalance
  - ➢ Harder for users to exploit full potential

- **Examples: Software**
  - Dynamic scheduling
  - Adaptive mesh refinement (AMR)

- **Examples: Hardware / Design**
  - NUMA architecture design
  - Complex memory hierarchies
    - HBM
    - Non-Volatile Memory (NVM)
    - DRAM
  - Heterogeneous compute nodes
  - Dynamic adjustments of machines
    - Based on thermal conditions
    - Turbo-Boost in modern CPUs

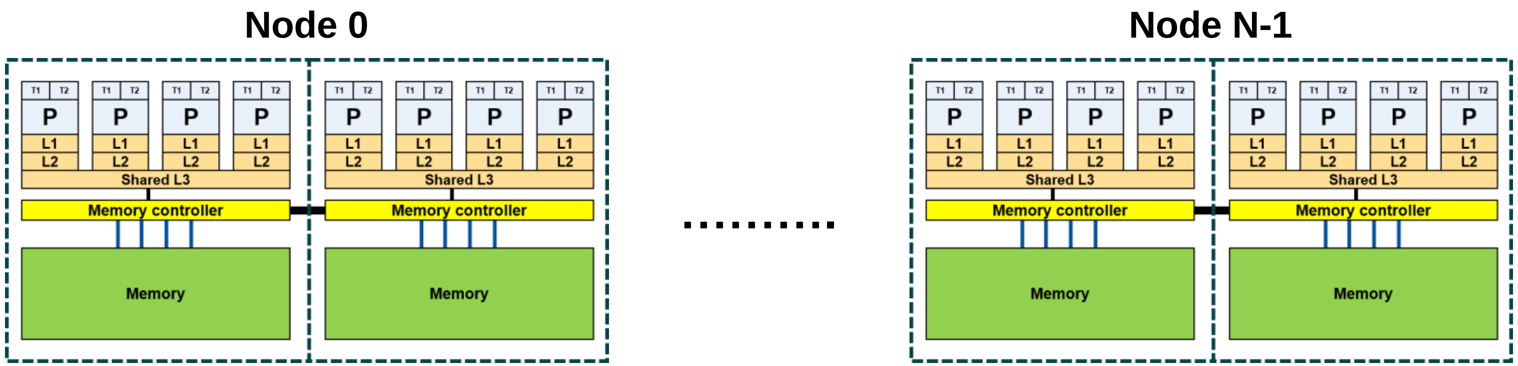
2-socket NUMA architecture (simplified)



- Location of threads accessing GPUs can affect performance
  - Offload latency
  - Transfer throughput

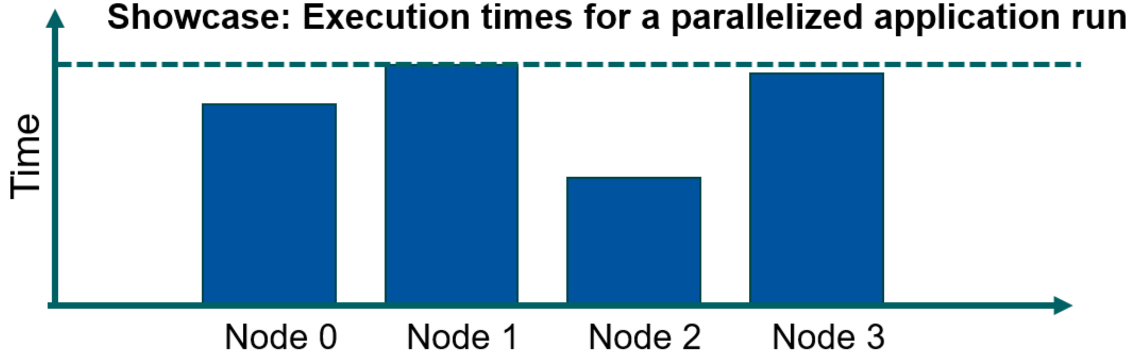
Motivation for Thesis Topics

- **Increasing complexity of today‘s HPC systems and software**
  - Performance variability
  - Load imbalance
  - Harder for users to exploit full potential



- **Examples: Software**
  - Dynamic scheduling
  - Adaptive mesh refinement (AMR)

- **Examples: Hardware / Design**
  - NUMA architecture design
  - Complex memory hierarchies
    - HBM
    - Non-Volatile Memory (NVM)
    - DRAM
  - Heterogeneous compute nodes
  - Dynamic adjustments of machines
    - Based on thermal conditions
    - Turbo-Boost in modern CPUs



- **Q:** How to balance the load between nodes without requiring extensive user code adaptions?

Thesis Outline Extraction: Core Chapters

- Variability of Application Runs
- **Locality-Aware Scheduling in OpenMP**
  - Task Affinity
  - Thread-to-Device Affinity

  } covered today
- Reactive Load Balancing for Hybrid Task-Parallel Applications
- Heuristics for Heterogeneous Memory

Locality-Aware Scheduling in OpenMP

Task Affinity

References:

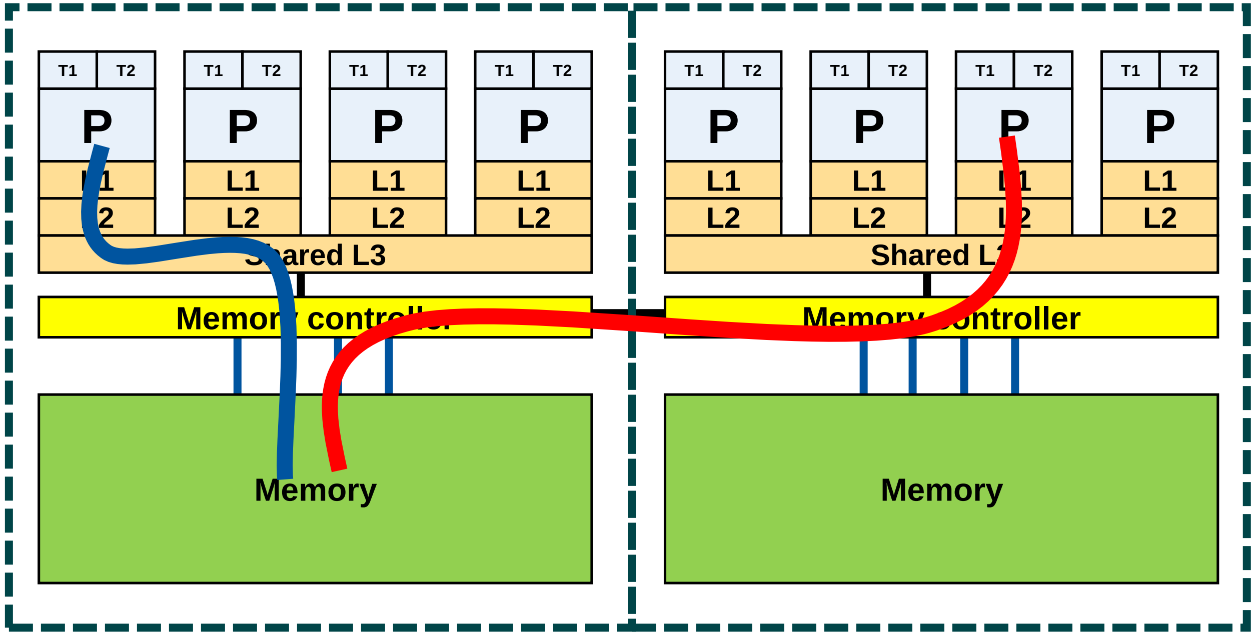
(1) Klinkenberg, J. *et al.* (2018). Assessing Task-to-Data Affinity in the LLVM OpenMP Runtime. In: de Supinski, B., Valero-Lara, P., Martorell, X., Mateo Bellido, S., Labarta, J. (eds) Evolving OpenMP for Evolving Architectures. IWOMP 2018. Lecture Notes in Computer Science(), vol 11128. Springer, Cham. https://doi.org/10.1007/978-3-319-98521-3_16

(2) Poster on COLOC Workshop (EuroPar 2018)

Motivation for Task Affinity

- **Execution of parallel programs**
  - Usually, OS can decide to migrate processes or threads between processing units
  - Existing techniques for process pinning & thread binding (`taskset`, `OMP_PROC_BIND`)
  - ➢ Avoid that process or threads are migrated
  - ➢ Best practice in HPC in most cases
- **OpenMP 3.0 introduced Tasking**
  - Allows parallelization of irregular and recursive algorithms
  - But: currently not much support for controlling / influencing placement of OpenMP tasks on OpenMP threads
  - **Generally:** Tasks can be executed by any thread in the task team

2-socket NUMA architecture (simplified)



Task accessing local memory is fast

Task accessing remote memory is slower

- Unpredictable remote memory accesses & execution times
- High runtime variability
- ➢ Data locality crucial to sustain performance
- ➢ **Need a way to specify affinity for tasks**

Proposal: Task-Affinity Extension for OpenMP 5.0

- `#pragma omp task affinity( list )`
  - Programmer specifies data used by task
  - Recommended to execute task closely to data location
  - Runtime identifies the location of the data and schedules task to a close thread
  - Clear separation between dependencies and affinity

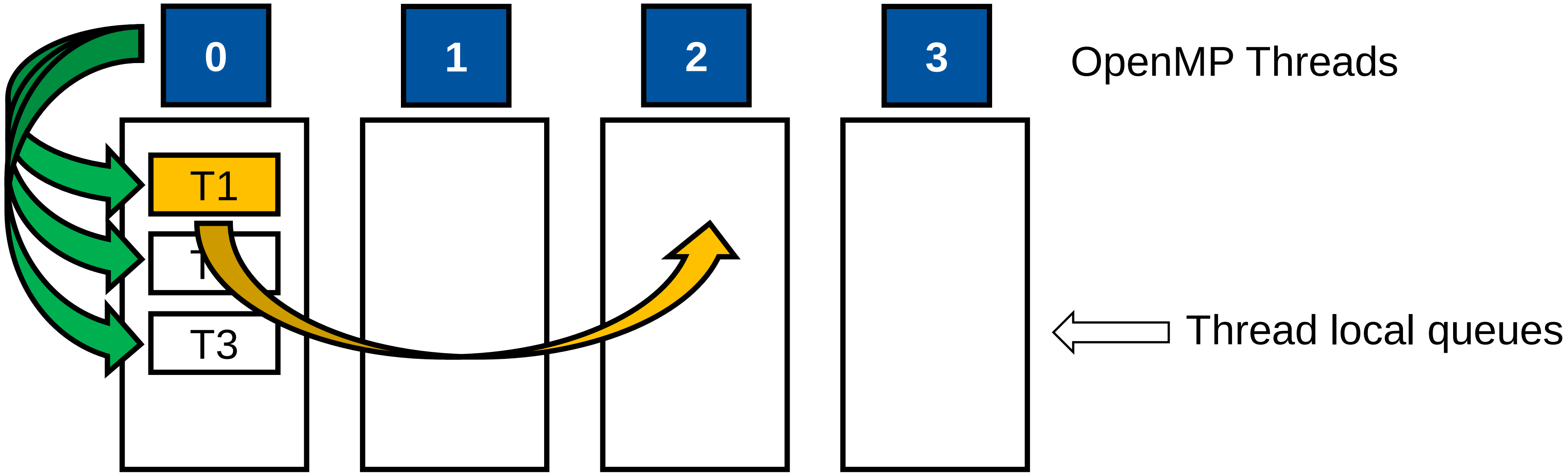
- **Important:** Non-prescriptive hint to the runtime
  - Reduce NUMA effects and improve overall performance
  - Do not prohibit task stealing & load balancing

- **Further Research Questions**
  - **Q1:** How does the location where tasks are created affect the performance?
  - **Q2:** Is task affinity able to improve performance and reduce the run time variability of task executions?

LLVM Reference Implementation

- **Implementation based on the LLVM OpenMP runtime**
  - Compatible with compilers like Intel, AMD and Clang (large community)
  - Simulating task affinity clause with API call right in front of the `task` construct
  - Currently limited to a single data reference (but extension available)

  - <u>Remember:</u> In LLVM, each OpenMP thread has a separate task queue
    - Tasks are usually pushed to local thread queues
    - Working on local tasks: remove at tail
    - Under-utilized threads steal from random victim

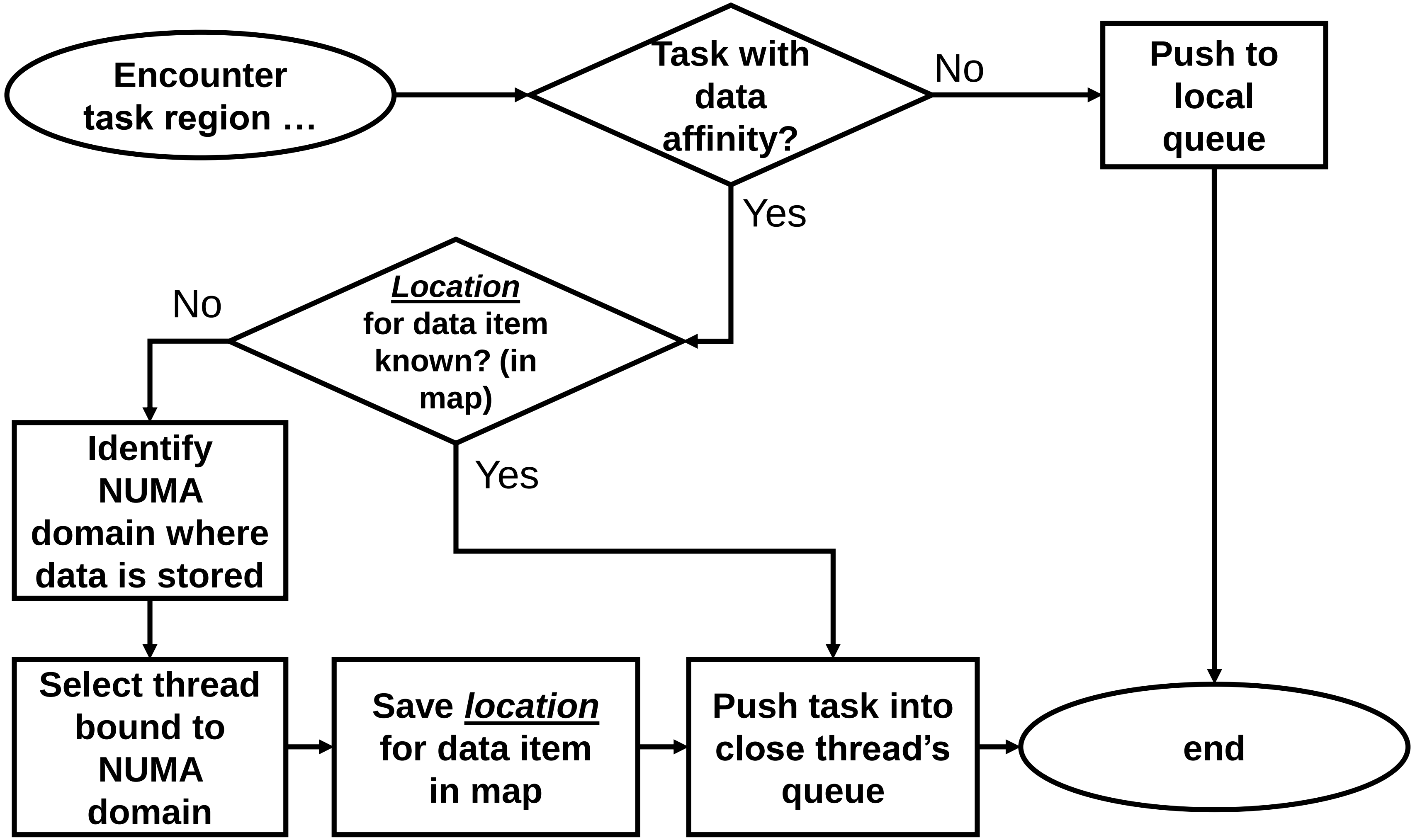


Directions & Approaches

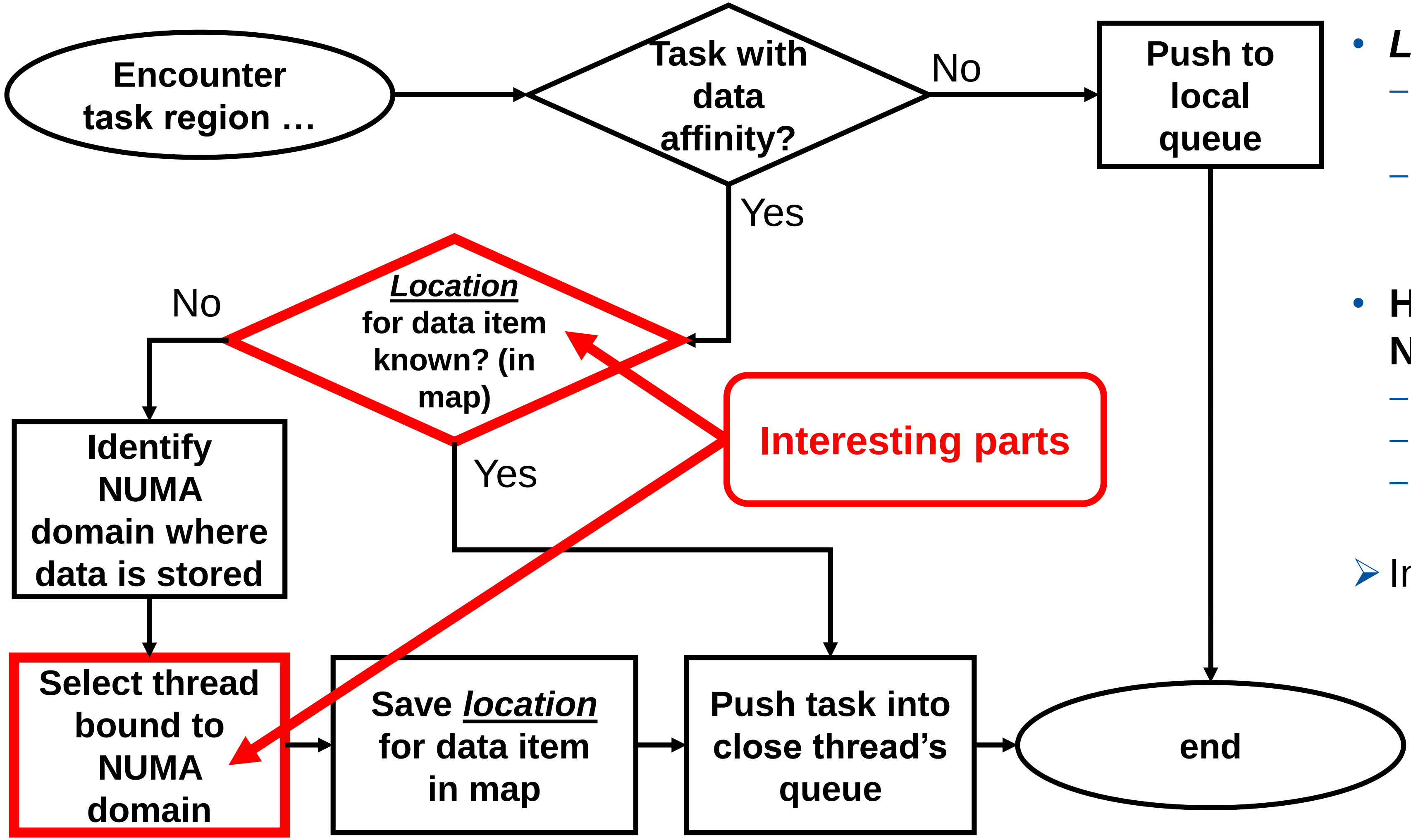
- **Fundamental directions / goals for task affinity**
  - *Domain Mode*
    - Execute task where data is physically stored / allocated
  - *Temporal Mode*
    - Execute task where last task has been executed that used same data
    - Reuse cached data and aim for temporal locality
  - ➢ Book keeping required! (using a lookup table or map) – Assumption thread binding is used

- **Fundamental approaches**
  - *NUMA-aware task distribution*
    - Identify NUMA domain for data reference & push task to a close location
  - *NUMA-aware task stealing*
    - Prefer stealing from thread in same NUMA domain

NUMA-aware Task Distribution



NUMA-aware Task Distribution



- ***Location* in the map?**
  - *Domain*: NUMA domain where data is physically allocated
  - *Temporal*: Thread where data was used the last time (by a task)
- **How to select a thread inside a NUMA domain?**
  - *Random*
  - *Round robin*
  - *Thread with lowest queue size*

➢ In total 6 versions

Evaluation – Benchmarks & Machines

1) Preliminary analysis with STREAM (tasking version)

- Address research questions
- Why STREAM?
  - Easy to understand and balanced
  - Simulate memory bound codes that use tasking
  - Determine upper bound for improvement

2) Overall performance & scalability

- STREAM → tasking version, balanced
- Parallel merge sort (BOTS) → recursive divide & conquer
- Sparse CG (SPMXV) → iterative, natural imbalances
- Health benchmark (BOTS) → divide & conquer, tree-based structure

- **Machines (with different NUMA characteristics)**
  - Intel® Xeon® E5-2650v4 (codename Broadwell)
    - 2 sockets, 12 cores per socket = 24 cores
    - 2.2 GHz base frequency
    - 128 GB memory
  - Intel® Xeon® E7-8860v4 (codename Broadwell)
    - 8 sockets, 18 cores per socket = 144 cores
    - 2.2 GHz base frequency
    - 1 TB memory

Evaluation – Compilation & Environment

- **Compiled all codes with –O3**

- **OpenMP thread binding**
  - `OMP_PLACES=cores`
  - `OMP_PROC_BIND=spread`

- **Data distribution across <u>all</u> NUMA domains**
  - Data initialized with first touch and
    `#pragma omp parallel for schedule(static)`

- **Additional settings**
  - Disabled automatic NUMA balancing (e.g. in RHEL)
  - Disabled Transparent Huge Pages (THP)
  - Set `KMP_TASK_STEALING_CONSTRAINT=0`

Preliminary analysis with STREAM (tasking version)

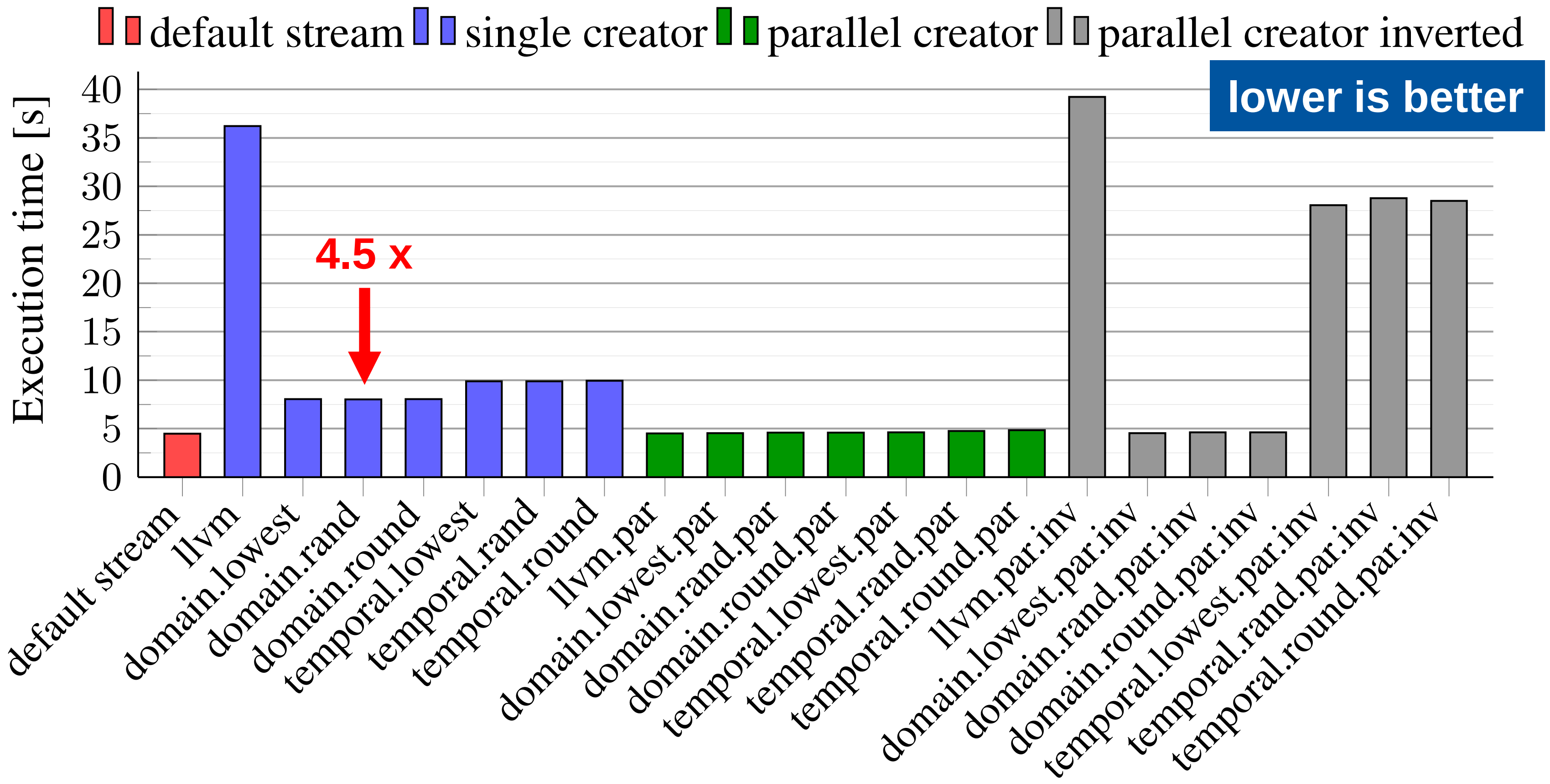
- **Q1: How does the location where tasks are created affect the performance?**
- Each kernel executed 10 times; large array split into `n_threads*factor` tasks
- Evaluate different task creation schemes
  - Single task creator (master)
  - Parallel task creators
  - Parallel task creators but invert chunks
- Parallel creators: Each thread creates tasks for its assigned chunk

Preliminary analysis with STREAM (tasking version)

- **Q1: How does the location <u>where</u> tasks are created affect the performance?**
- Each kernel executed 10 times; large array split into `n_threads*factor` tasks
- Evaluate different task creation schemes
  - Single task creator (master)
  - Parallel task creators
  - Parallel task creators but invert chunks
- Inverted: Each thread creates tasks for a **<u>different</u>** chunk

Preliminary analysis with STREAM (tasking version)

Sockets=8 Threads=64 N=2^{31} double=16 GB Median of 15 runs



- Not much improvement when task created where data is located
- Otherwise: LLVM baseline clearly suffering

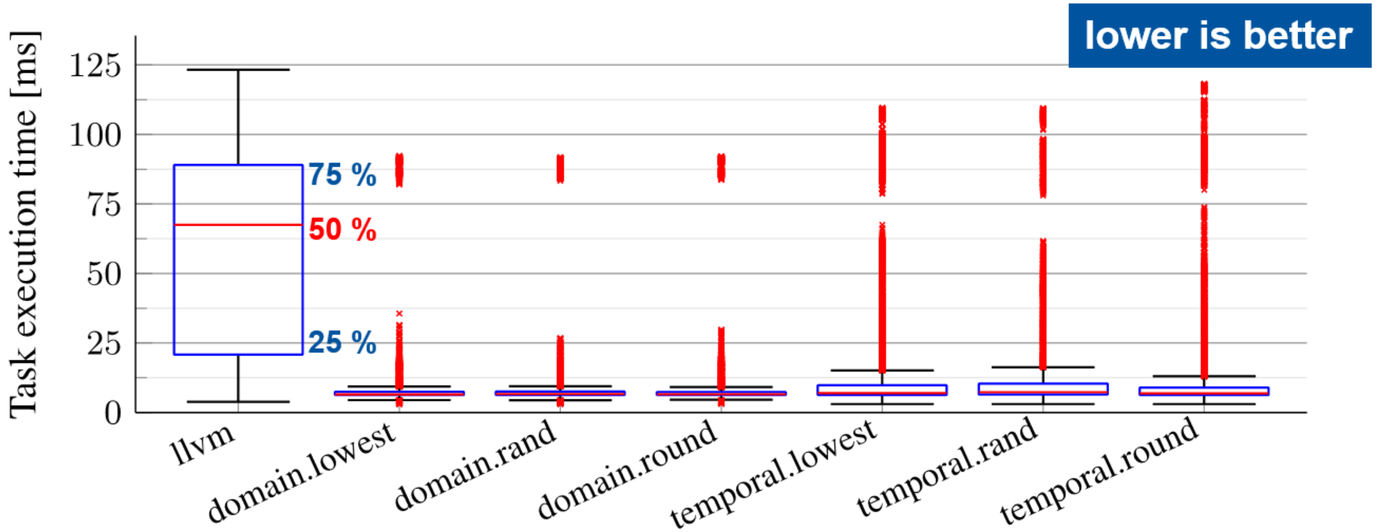
Preliminary analysis with STREAM (tasking version)

- **Q2: Is task affinity able to improve performance and reduce the run time variability of task executions?**

- Same setup with single task creator scheme
- Measure individual task execution times

- Problem: Complexity & exec. time of STREAM kernels varies
  - Hard to distinguish between real variations and those caused by different complexity
  - ➢ Just considering Triad kernel for this test

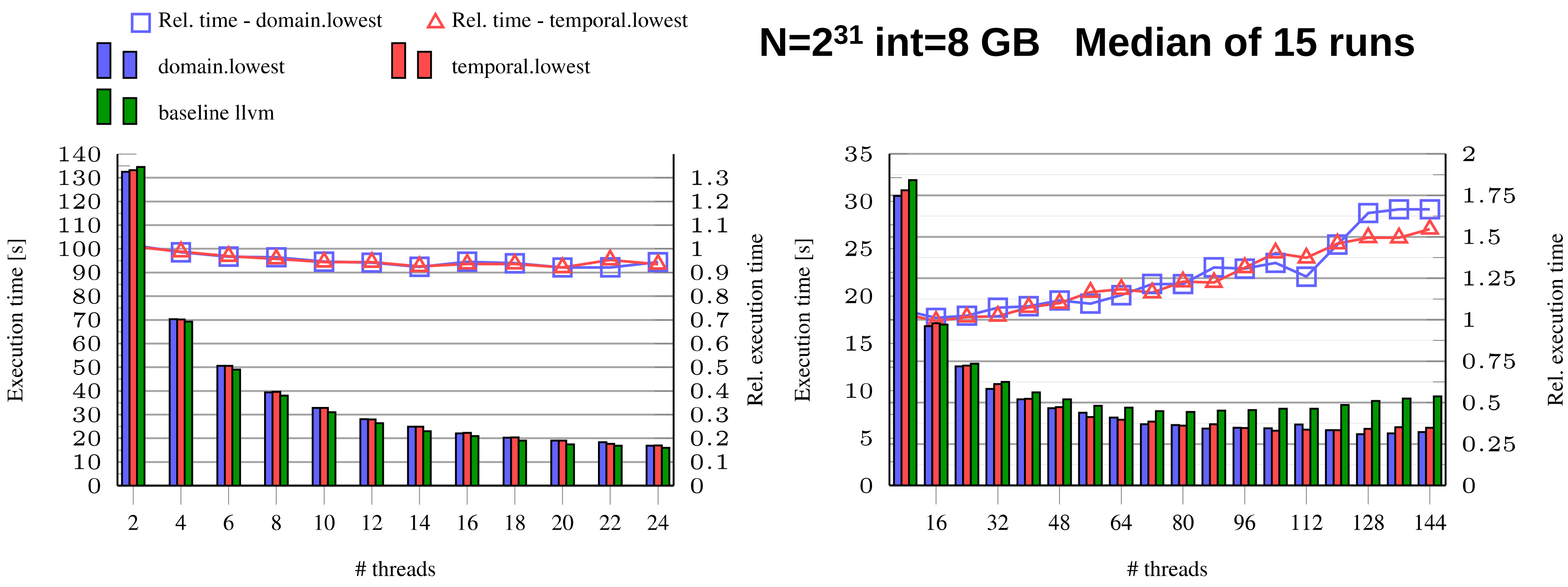
Sockets=8 Threads=64 N=2^{31} double=16 GB Tasks=10,240



- ➢ LLVM has much higher spread and median
- ➢ Significant reduction of runtime variability
- ➢ More reliable execution performance

Overall performance & scalability – Merge sort

N=2^{31} int=8 GB   Median of 15 runs



(a) Merge sort on 2-socket   (b) Merge sort on 8-socket

- Not much overhead but also not giving any speedup on 2 sockets
- Stronger NUMA effects → better improvements

Conclusion

- **Not much room for improvement when:**
  - Parallel task creator scenarios & tasks are already created in chucks where data is located + already pretty balanced workload

- **Works well when:**
  - Working with a lot of data (memory-bound)
  - Single task creator scenarios
  - Tasks created in parallel but not all created close to data
  - Suffering from load imbalances

- **What has been done since then?**
  - Extended prototype that lifted restriction to single data reference
    - Deal with array slices
    - Deal with multiple affinities
  - Affinity for tasks created by `taskloop` construct (ongoing)

Locality-Aware Scheduling in OpenMP

Thread-to-Device Affinity

References:

(1) Coming soon

Device Affinity: Potential Use Cases

1. **Bind threads so they get distributed appropriately for using devices**
   - e.g. `OMP_PLACES=devices`
   - Each place corresponds to the set of cores that are close to each device in the target machine
   - Decisions:
     - Not that easy. Could lead to ambiguous results for several devices/threads
     - On some systems `OMP_PLACES=sockets`

2. **Offload to devices that are close to the current thread**

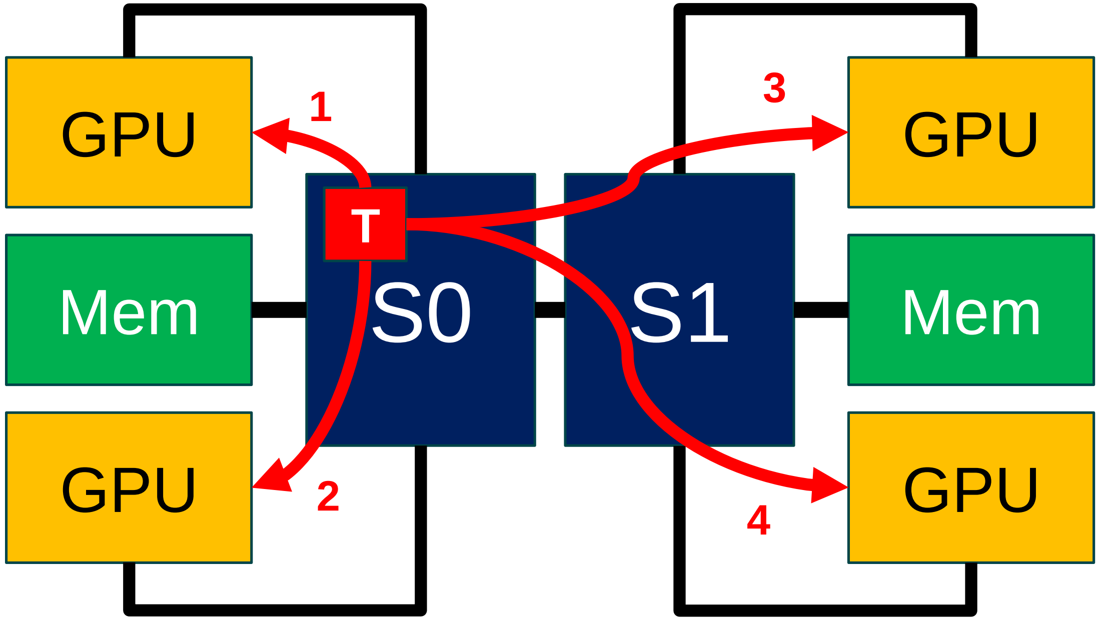
3. **Offload to devices that are close to data – or – that already hold the required data**
   - Turns out to be also more complicated
   - Might interfere with default device selection
   - How to work with sets of devices?
   - ➢ WiP

Thread-to-Device Affinity

- **Goal:** Find devices that are close to the current thread

- **Requirements:**
  - Result of call should be deterministic!
  - How to offer a general solution that is also extendable in future?

Sample Architecture 1



Thread-to-Device Affinity

- **Proposal**
  - `int omp_get_devices_in_order(int n_desired, int* dev_ids, double* val_order, <traits>)`
  - Traits could be used for filtering as well as ordering
  - Returns number of devices found
- **Example**

```
int n=20;              // desired number of devices
int n_dev_found;       // actual number of devices found for request (<= desired value)
int dev_ids[n];        // buffer with ids returned
double vals_order[n];  // buffer with values returned for ordering devices

n_dev_found = omp_get_devices_in_order(n, dev_ids, vals_order, <trait_lowest_distance>);

#pragma omp target device(dev_ids[0])             // use closest device
#pragma omp target device(dev_ids[n_dev_found-1]) // use remote device (max distance)
```

- **Questions**
  - What is distance or what does close mean? (could be implementation defined)
    - Currently considering NUMA latency distances
    - Could be more complex (respecting BW, PCI connection, …)
  - How should traits look like? (Similar solution as for allocator traits)

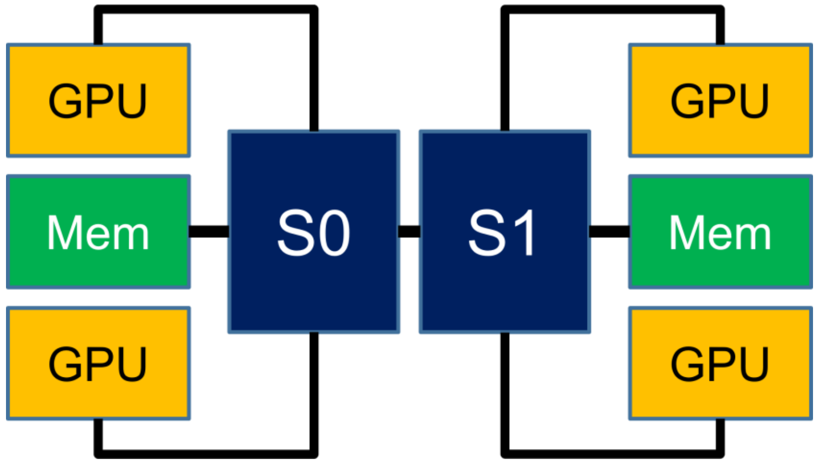
Prototypes & Concepts

- **Prototypes**
  - **Prio 1:** CUDA prototype for PoC
  - **Prio 2:** Prototype implementation in LLVM OpenMP runtime

- **Implemented Concept:**
  - Iterate over devices and NUMA domains (once during init)
    - Identify where devices are connected (e.g., using **hwloc**)
    - Currently: Use NUMA distances to order devices per NUMA domain
    - Save that lookup table
  - Reuse lookup table at run time when API routine is called by threads (avoids overhead)

- **Current Restrictions**
  - Only implemented for NVIDIA GPUs
  - Not traits → focus on distance

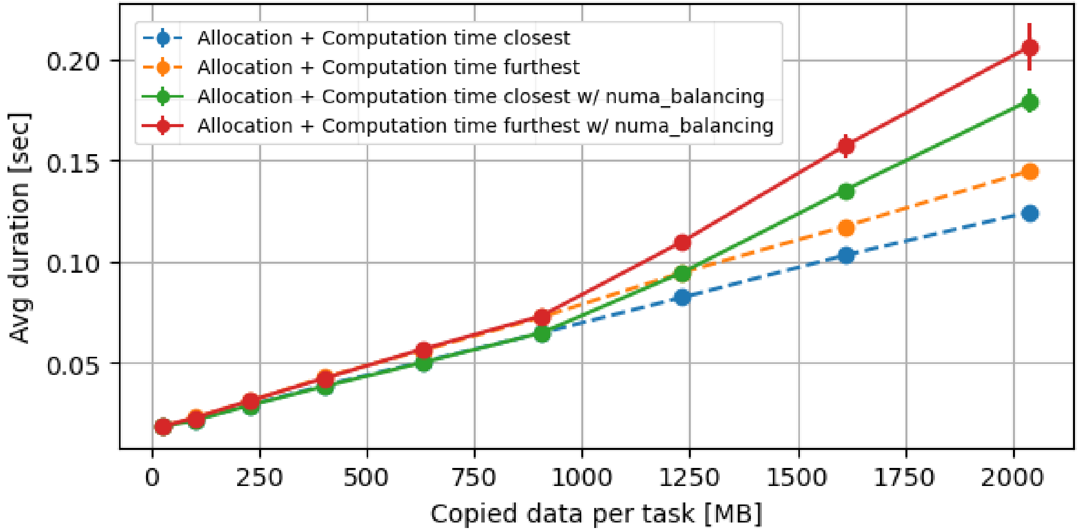
Sample Architecture 1



Preliminary Results

- **LLVM results on a 2 GPU system (1 Tesla P100-SXM2 per socket)**

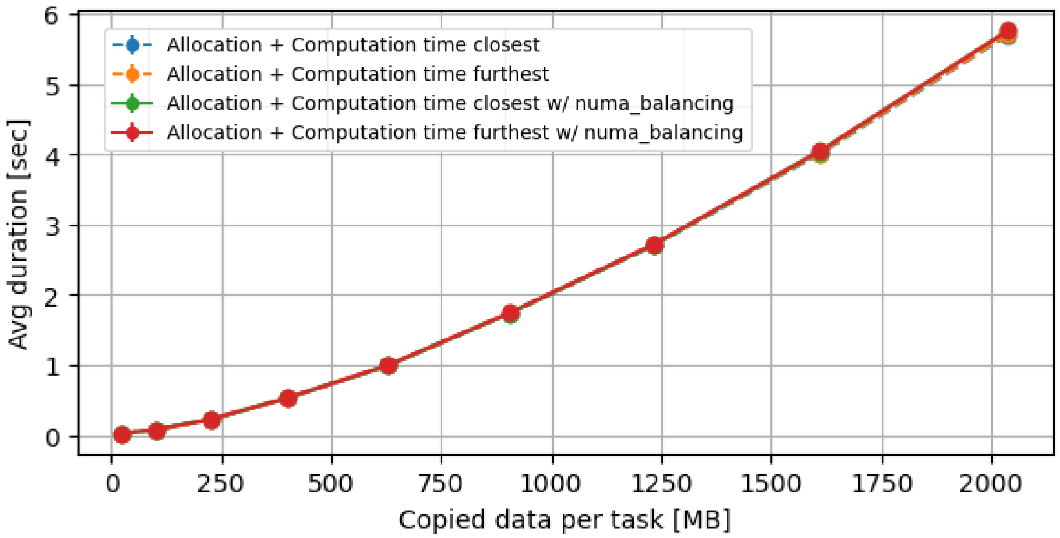
w/o computation – only invocation & transfer




```
Statistics (Computation): w/o numa_balancing
Min  relative difference    0.358 %
Mean relative difference   10.029 %
Max  relative difference   16.274 %

Statistics (Computation): w/  numa_balancing
Min  relative difference    0.828 %
Mean relative difference   10.861 %
Max  relative difference   16.194 %
```

w/ computation – incl. invocation & transfer



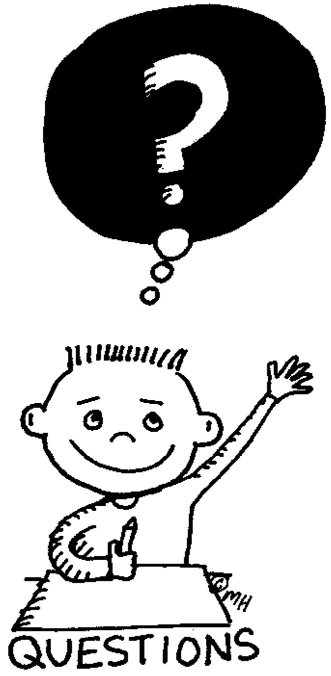

```
Statistics (Computation): w/o numa_balancing
Min  relative difference   0.132 %
Mean relative difference   0.549 %
Max  relative difference   2.398 %

Statistics (Computation): w/  numa_balancing
Min  relative difference  -0.250 %
Mean relative difference   0.262 %
Max  relative difference   0.898 %
```

Next Steps

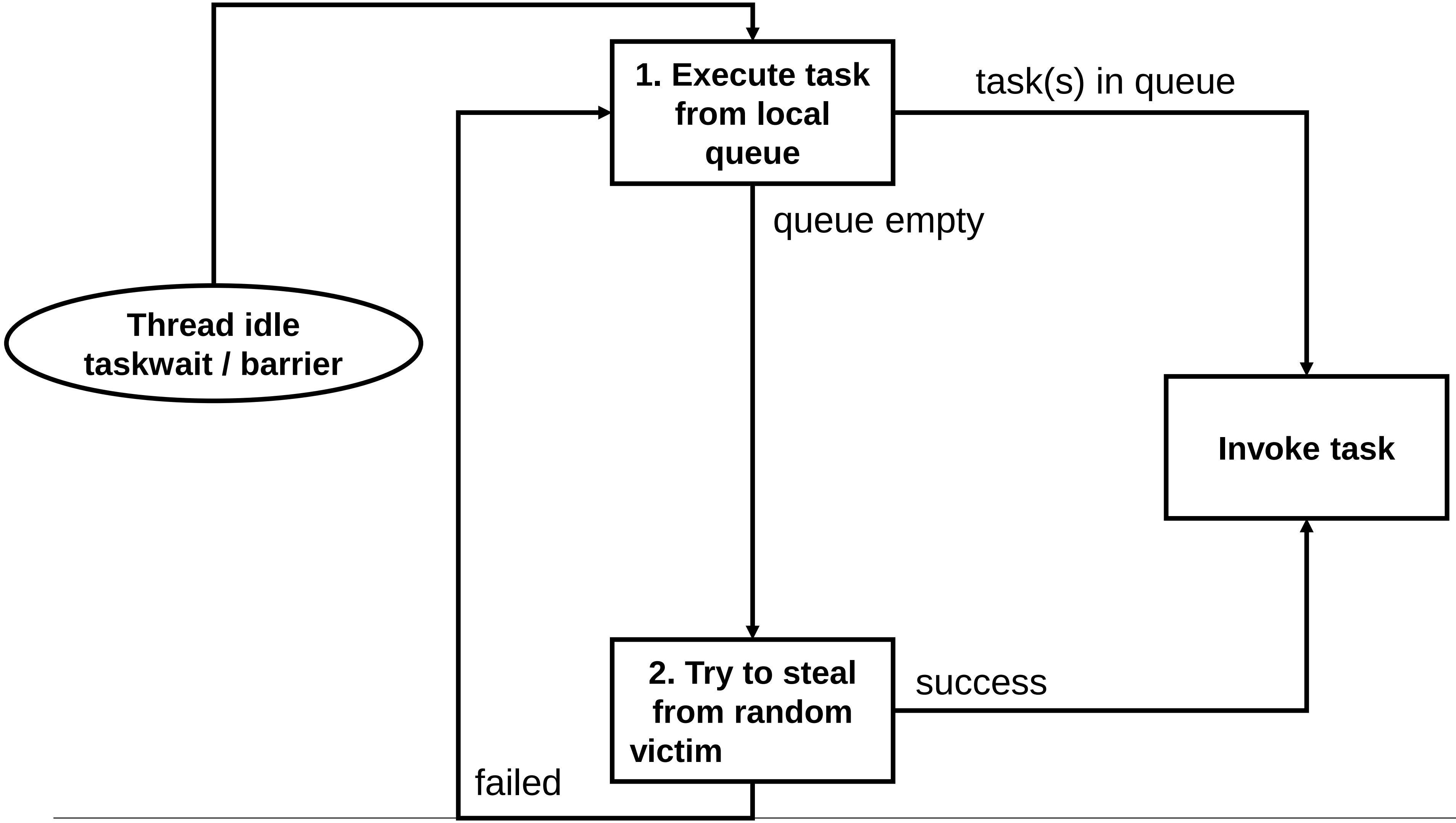
- **Extend support for AMD accelerators**
- **Extended experiments**
  - More architectures (NVIDIA DGX, Summit, Crusher, …)
  - Vary how much computation is actually done (find threshold)
  - Deeper look at GPU traces for more complex scenarios
- **Create a first set of traits for the API proposal**
  - Extend prototypes to return values used for ordering
- **Publication planned**

Thank you!

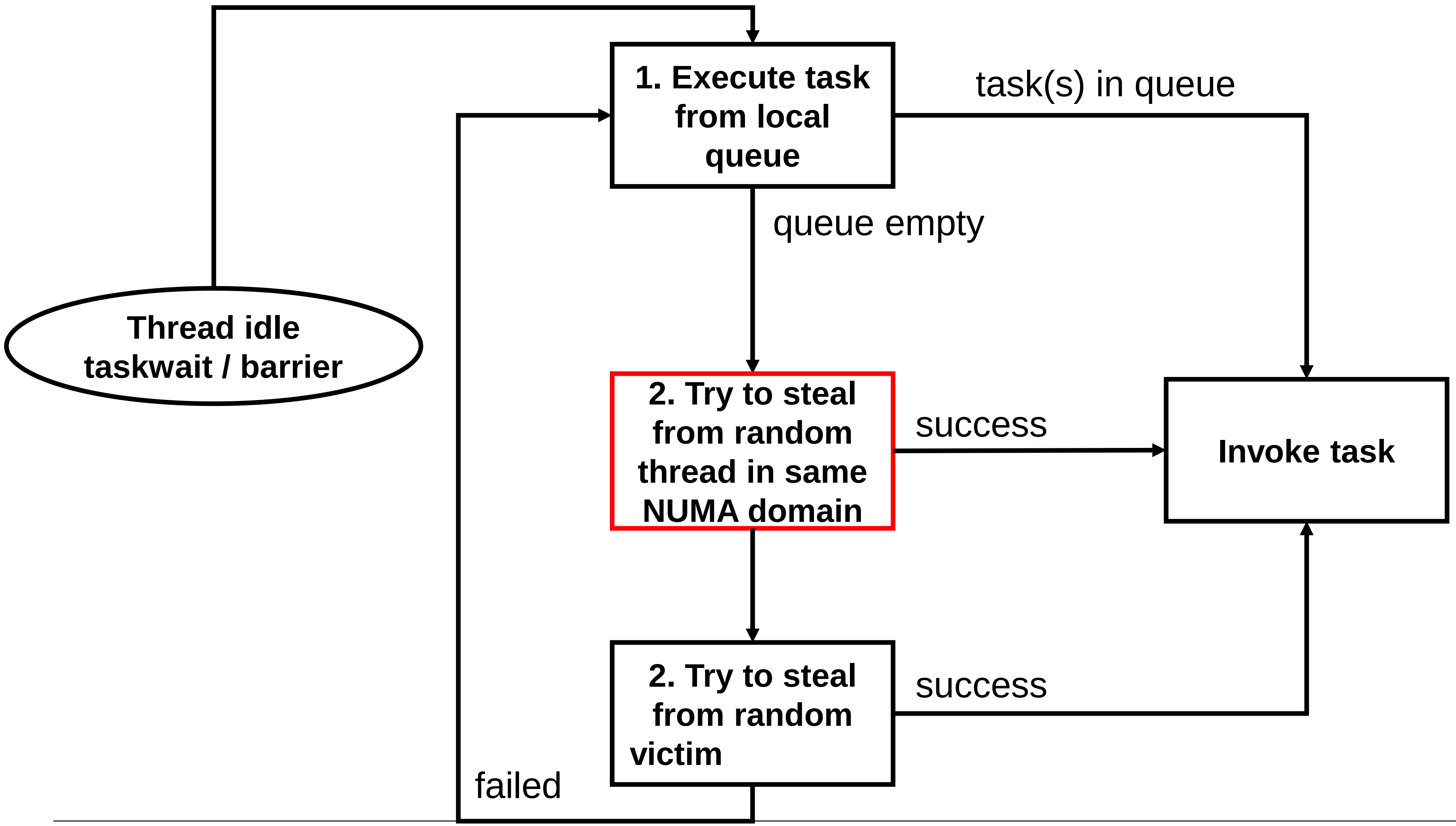


Backup Slides

NUMA-aware Task Stealing

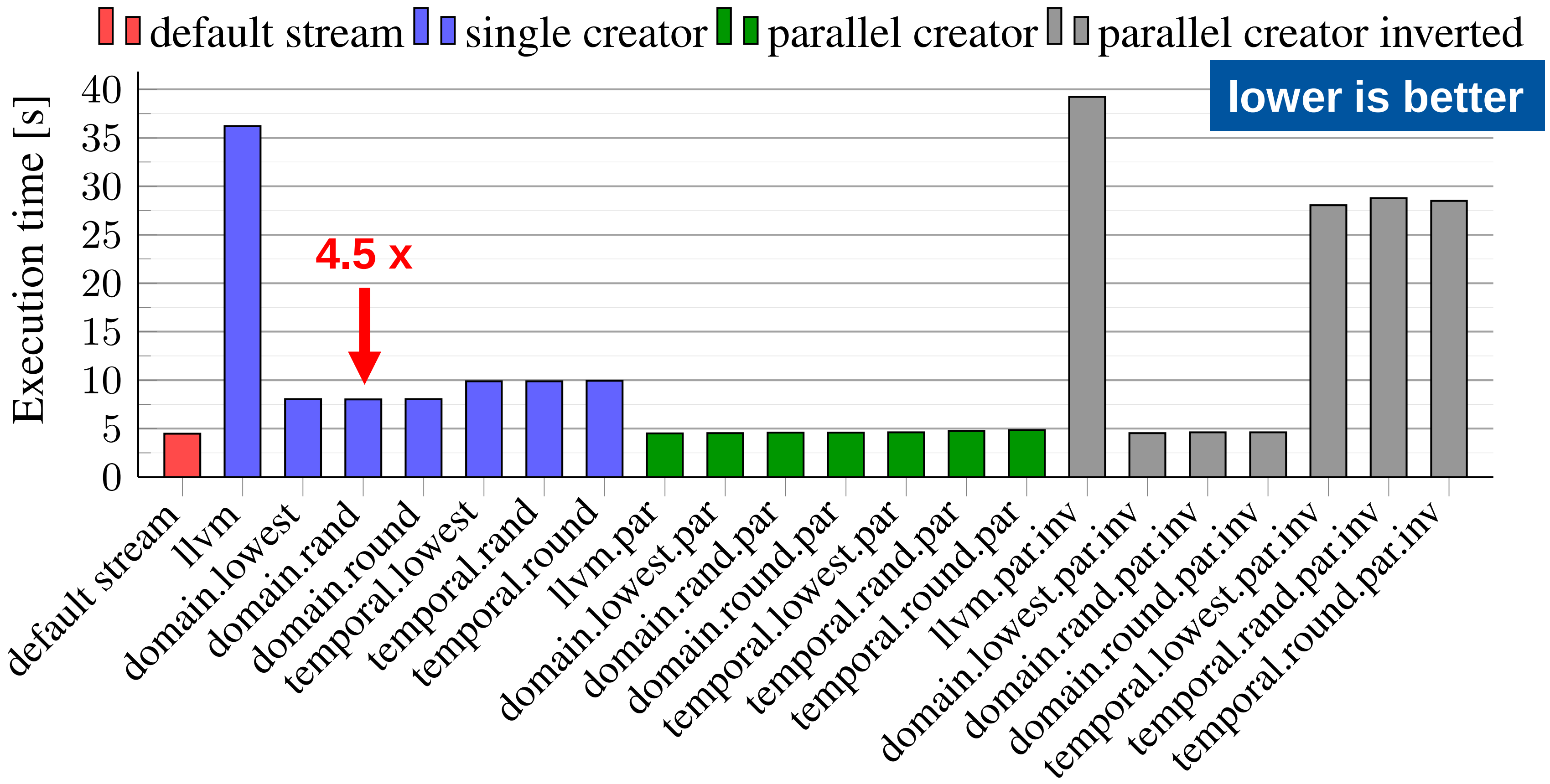


NUMA-aware Task Stealing



Preliminary analysis with STREAM (tasking version)

Sockets=8   Threads=64   N=2^{31} double=16 GB   Median of 15 runs

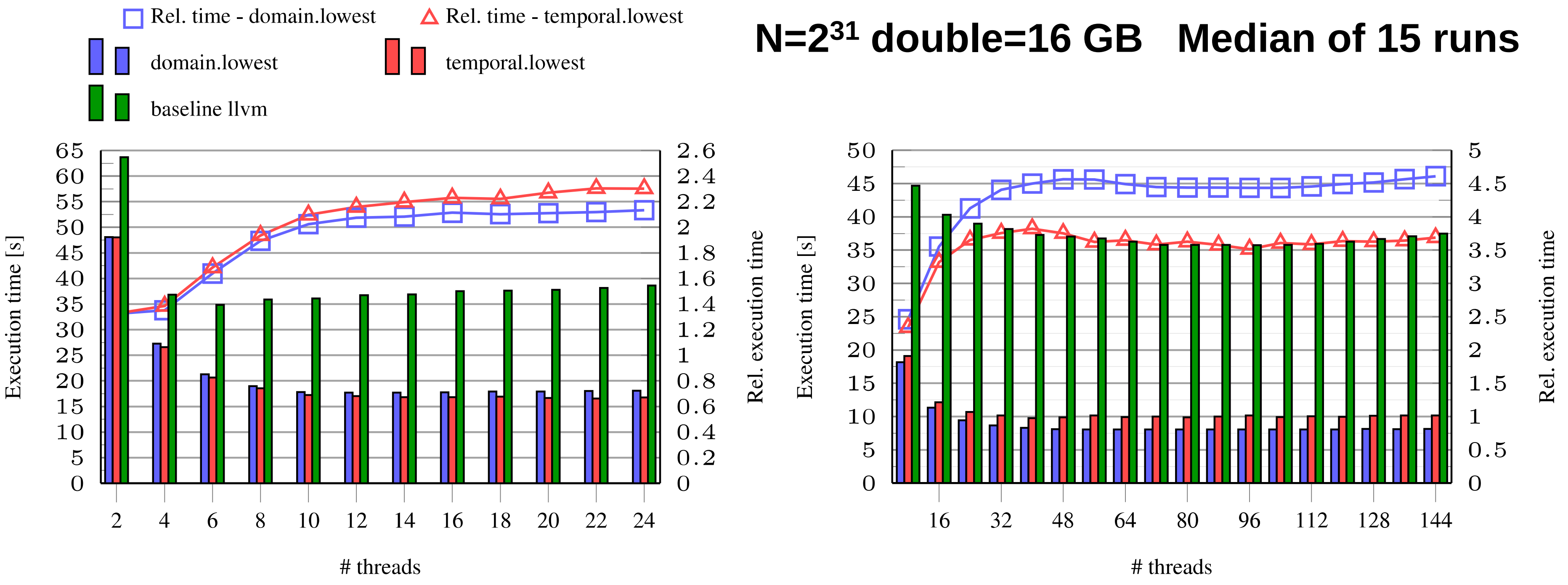


Compared to default STREAM:
***Parallel*: + 1-2 %**
***Single*:   + 80 %**

- Not much improvement when task created where data is located
- Otherwise: LLVM baseline clearly suffering

Overall performance & scalability – STREAM (single creator)

N=2^{31} double=16 GB Median of 15 runs



(a) STREAM on 2-socket

(b) STREAM on 8-socket

- Baseline stops scaling earlier
- Suffering from remote memory accesses
- Temporal mode more prone to stealing from foreign NUMA domain