On data-locality issues of task-based programming models

Dana Akhmetova
Supervisor: Professor Örjan Ekeberg
KTH Royal Institute of Technology
Stockholm/Sweden

Outline

1. Introduction into task-based programming models (TBPMs)

2. Interplay between task granularity and scheduling overhead in many-core shared-memory systems

3. Data locality issues in TBPMs

Task-based programming models

- A task-based program is formulated in terms of "tasks" (work to be done).
- A task is the set of instructions that occur between two interactions with the runtime system.
- Tasks are dynamically created, and then - assigned by a task scheduler to CPUs for their execution.
- A promising way to program Exascale applications:
  - applications are divided into a myriad of small tasks;
  - the system is overprovisioned with tasks (Ntasks >> Ncores).
- TBPMs: Intel TBB, Cilk/Cilk++/Intel Cilk Plus, OpenMP 3.0 and others.

Example of task-based code

```
fib(int n)
{
   if (n < 2)
      return n;

   a = spawn fib(n-1);

   b = spawn fib(n-2);

   sync;

   return a + b;
}
```

Example of task-based code

```
fib(int n)
{
   if (n < 2)
      return n;

   a = spawn fib(n-1);

   b = spawn fib(n-2);

  sync;

   return a + b;
}
```

Example of task-based code

```
fib(int n)
{
   if (n < 2)
      return n;

   a = spawn fib(n-1);

   b = spawn fib(n-2);

   sync;

   return a + b;
}
```

Example of task-based code

```
fib(int n)
{
   if (n < 2)
      return n;

   a = spawn fib(n-1);

   b = spawn fib(n-2);

   sync;

   return a + b;
}
```

Example of task-based code

```
fib(int n)
{
   if (n < 2)
      return n;

   a = spawn fib(n-1);

   b = spawn fib(n-2);

   sync;

   return a + b;
}
```

DAG representation

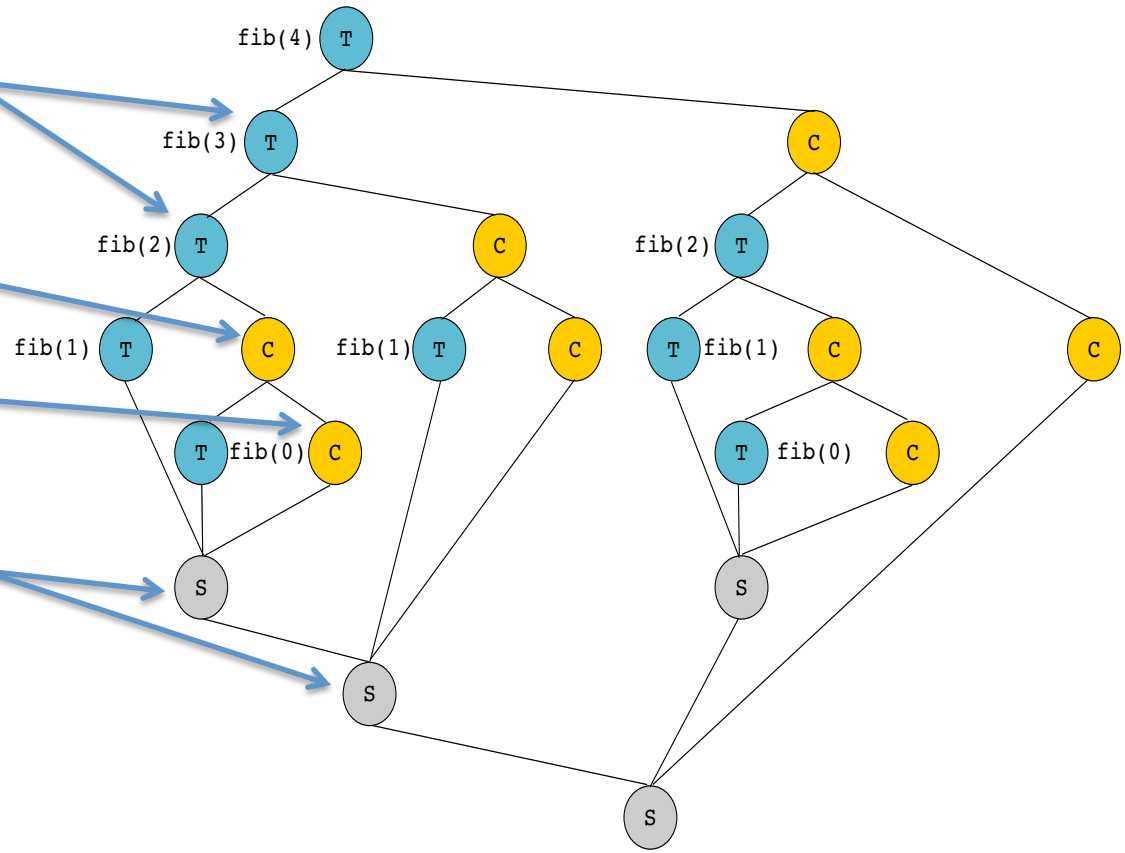
- Execution of a task-based program is a sequence of tasks being relayed:
  - task A is completed -> task B is triggered;
  - both task B and task C are done -> task D is triggered.

- It can be viewed as a directed acyclic graph (DAG).

- Each node in the DAG represents a specific task while edges represent dependencies between tasks.

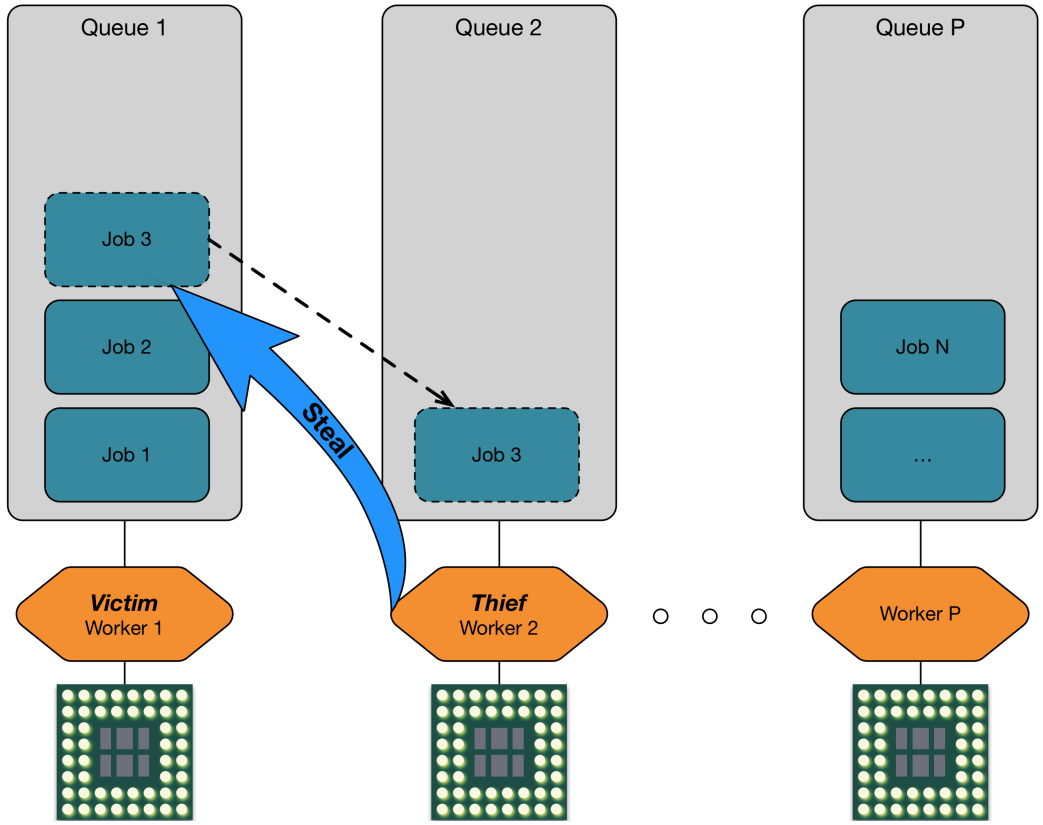
DAG for Fib(4)

```
fib(int n)
{
    if (n < 2)
        return n;

    a = spawn fib(n-1);

    b = spawn fib(n-2);

    sync;

    return a + b;
}
```



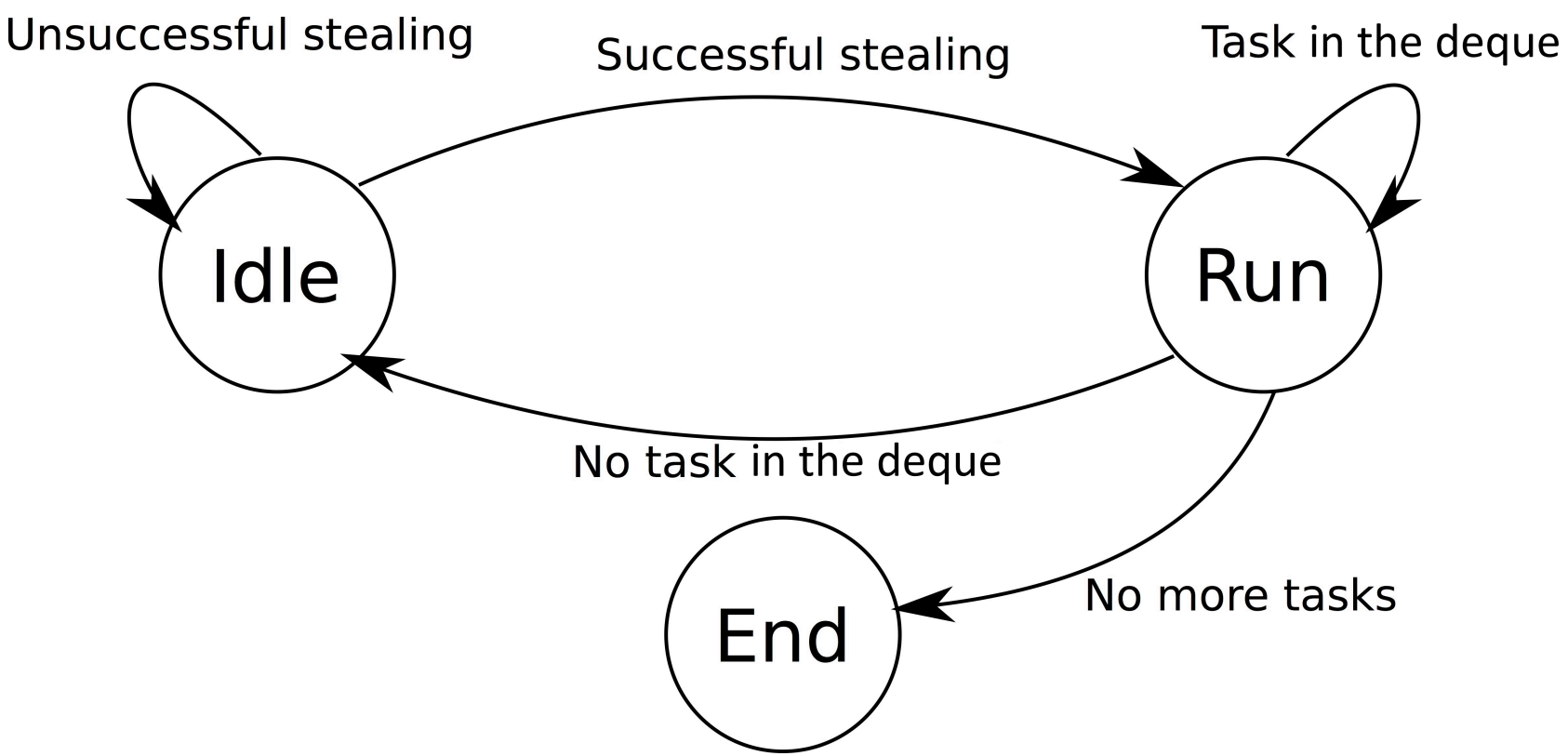
Work-stealing scheduling

- There are different scheduling algorithms on how to map tasks to CPUs.
- We focused on the work-stealing approach (which is implemented in the Cilk runtime system).



http://actor-framework.readthedocs.io/en/stable/Scheduler.html

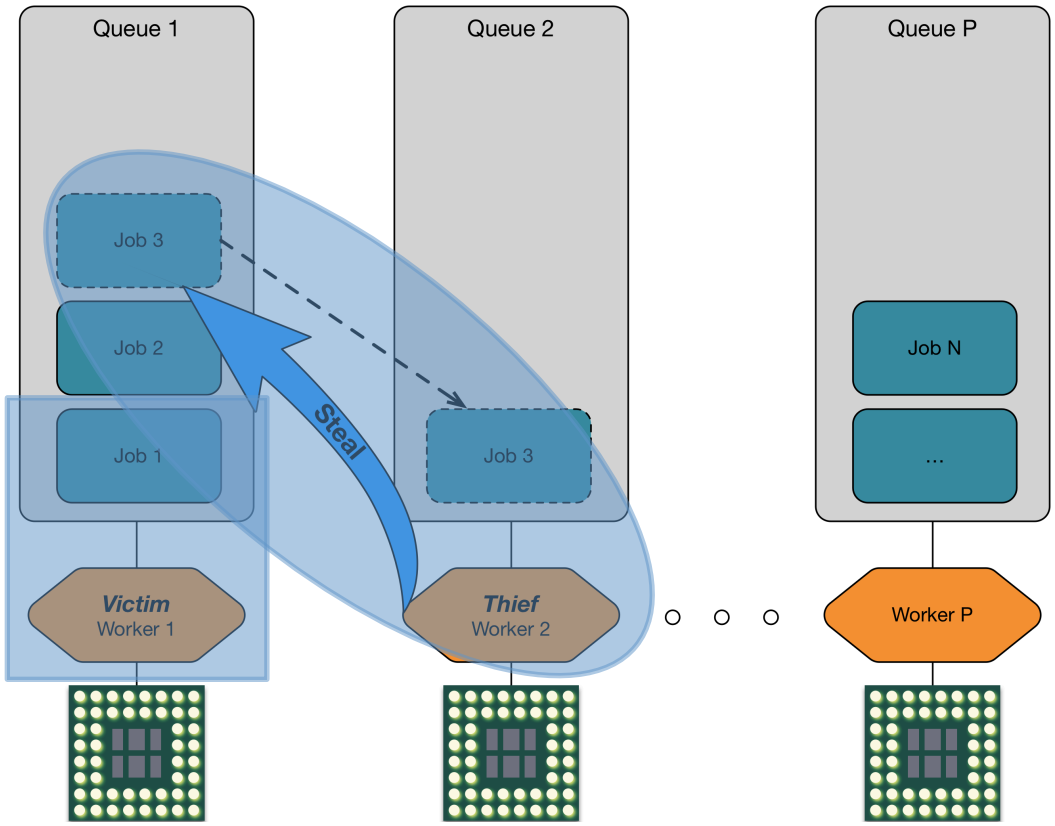
Work-stealing scheduling



Scheduling overhead

- Local + remote

- Remote overhead:
  - Unsucc. stealing cost (the latency of checking if a victim thread has a task that can be stolen);
  - Succ. stealing cost (+ the cost of moving a continuation task from the victim to the local thread).



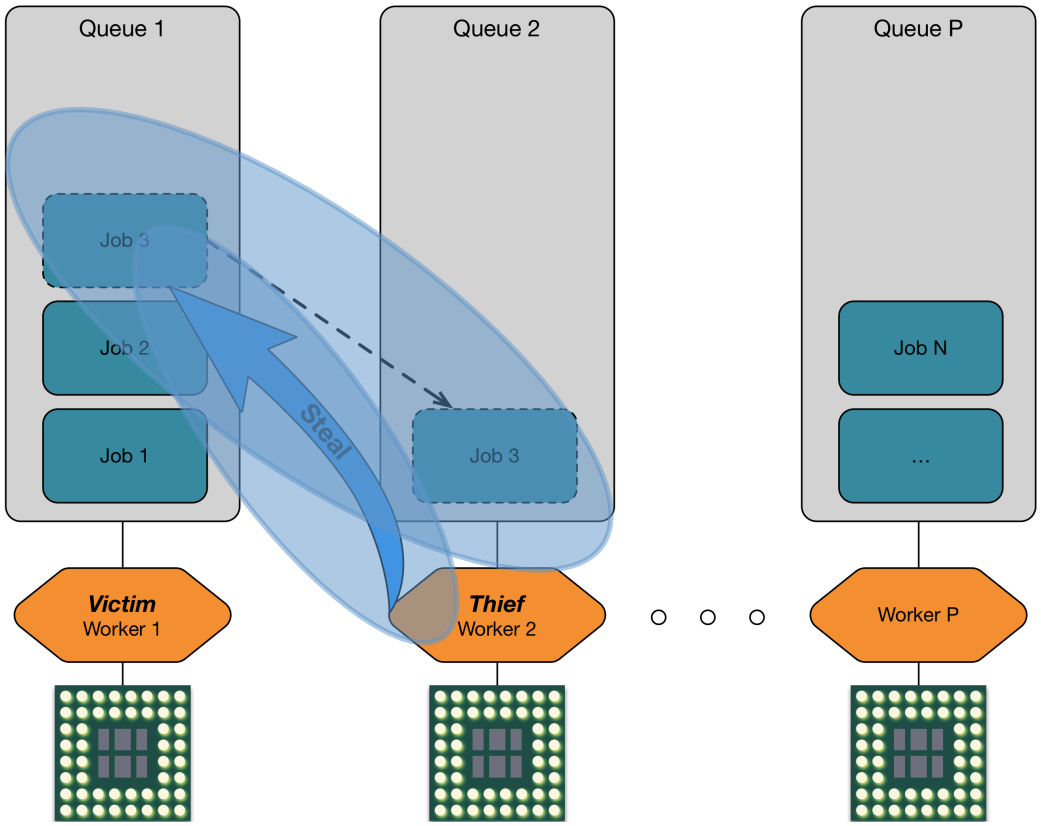
http://actor-framework.readthedocs.io/en/stable/Scheduler.html

Scheduling overhead

- Local + remote
- Remote overhead:
  - Unsucc. stealing cost

    (the latency of checking if a victim thread has a task that can be stolen);
  - Succ. stealing cost

    (+ the cost of moving a continuation task from the victim to the local thread).



http://actor-framework.readthedocs.io/en/stable/Scheduler.html

Task granularity

- Execution time of a single task between interactions with the runtime system (e.g., spawning a new task).
- Size of leaf tasks (inner most tasks) = granularity of computation.

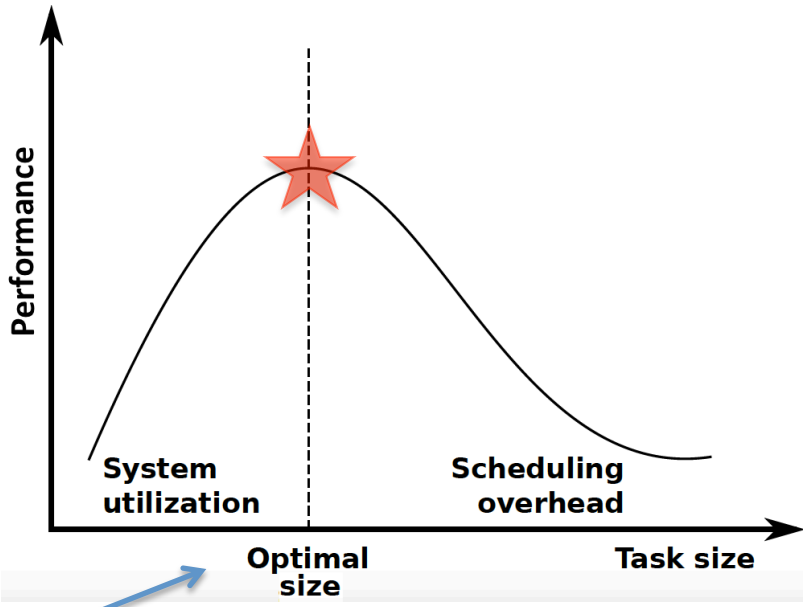
- ave_task_size = sum_exec_time_tasks / total_num_tasks

- Fine-grain -> large number of tasks
- Coarse-grain -> small number of tasks

- Example (dense matrix-vector multiplication Ab = y)
  - fine-grain: each task represents an element in y;
  - coarse-grain: each task represents 3 elements in y.

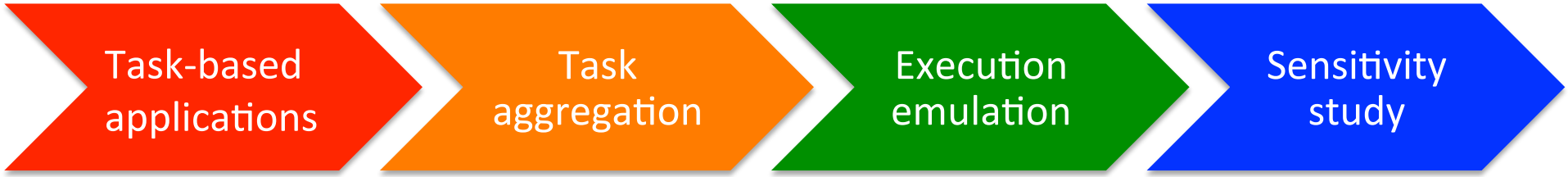
Performance of task-based applications

- **Performance of task-based applications** = **f** (# available tasks, task granularity, runtime scheduler algorithm and overhead).
  - **Fine-grained tasks:** increase of system load balance and utilization, but larger runtime overhead.
  - **Coarse-grained tasks:** smaller runtime overhead, but system becomes imbalanced, utilization decreases.
- To achieve high efficiency and scalability, one needs to find the **balance between task granularity, system utilization and task scheduler**.



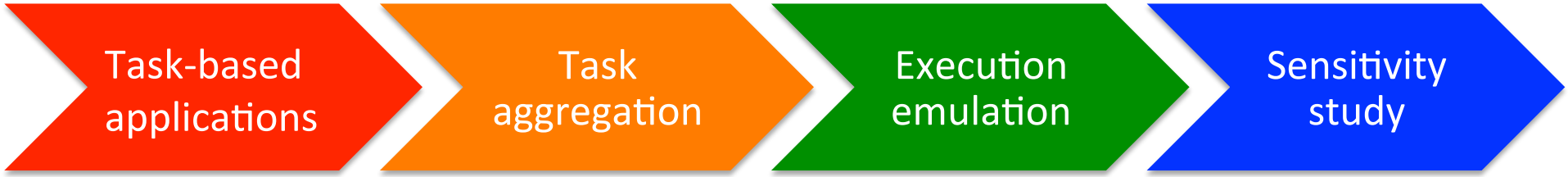
There is **an optimal task size** at which the relative impact of the scheduling overhead is minimal while the system utilization and balance are still high.

Methodology



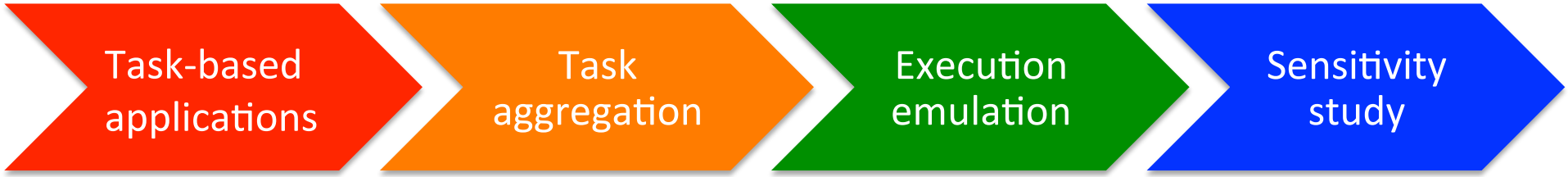
- **We focused on "spawn/sync" task-based applications on systems that do not yet exist (exascale machines – nodes with 1000s cores).**
- **We worked with DAGs of the Fib, Integrate, Heat and Jacobi applications from the Cilk benchmark suite.**

Methodology



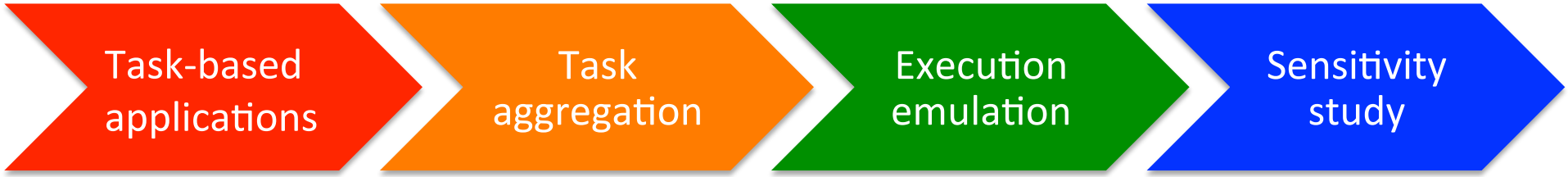
- **We aggregated tasks into coarser-grained with our algorithm that analyzes an application's DAG and automatically aggregates tasks without re-writing the application:**
  - **no complex code transformation;**
  - **no dealing with different task prog. model implementations and keywords;**
  - **study of parallel characteristics of apps;**
  - **maintains the application semantics and workload.**

Methodology



- **We emulated the execution of the applications with different task granularities with Prometheus, a system emulator for task-based applications.**
- **We emulated the work of 9 different task schedulers.**

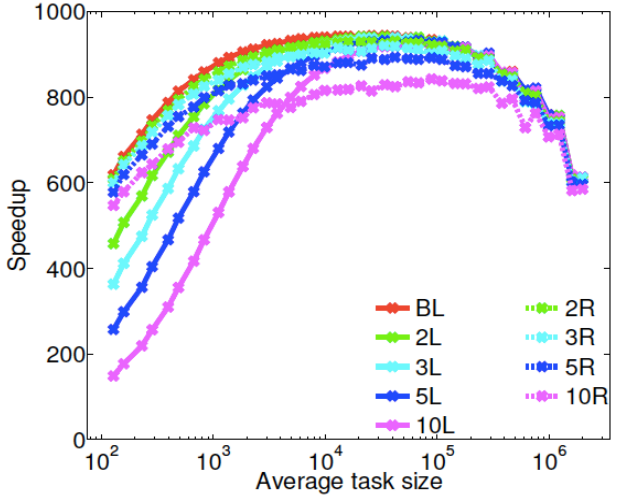
Methodology



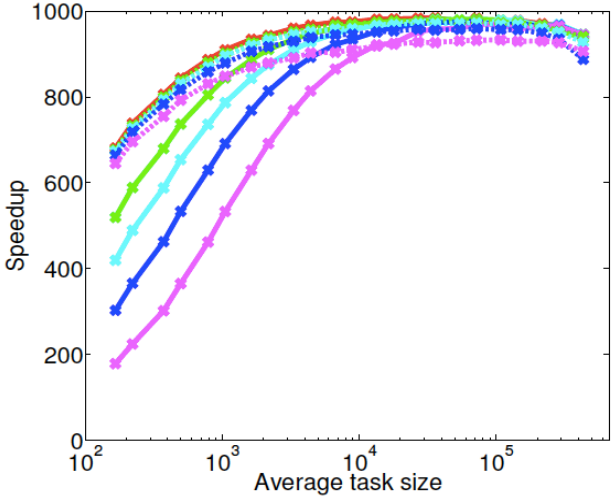
- **We analyzed the impact of changing task granularity and scheduling algorithm on performance of task-based applications.**

Task granularity analysis

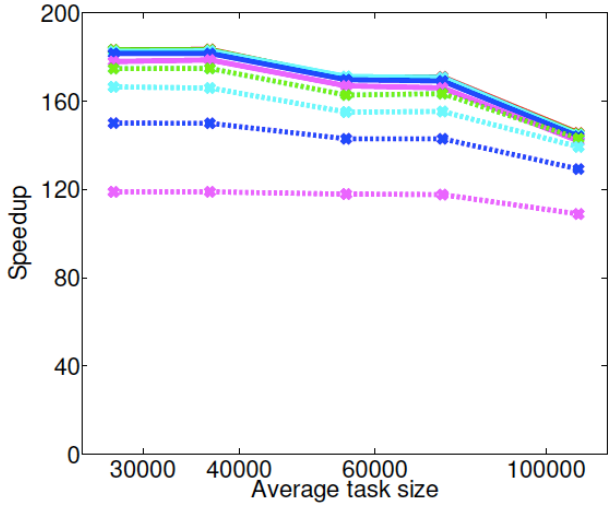
Speedup as function of ave. task size (logarithmic scale on the X-axis):



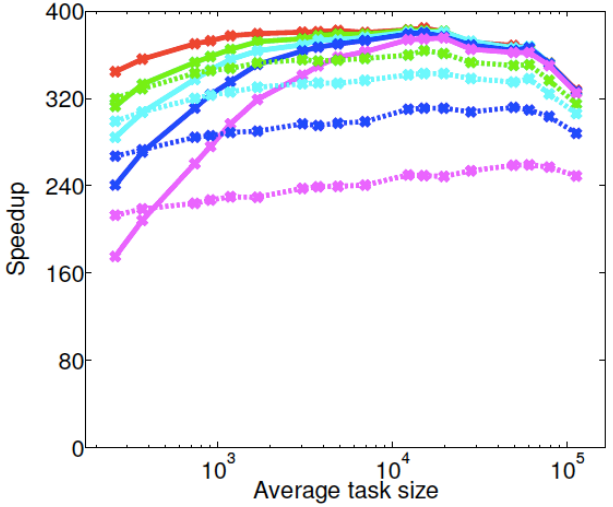
(a) Fib



(b) Integrate



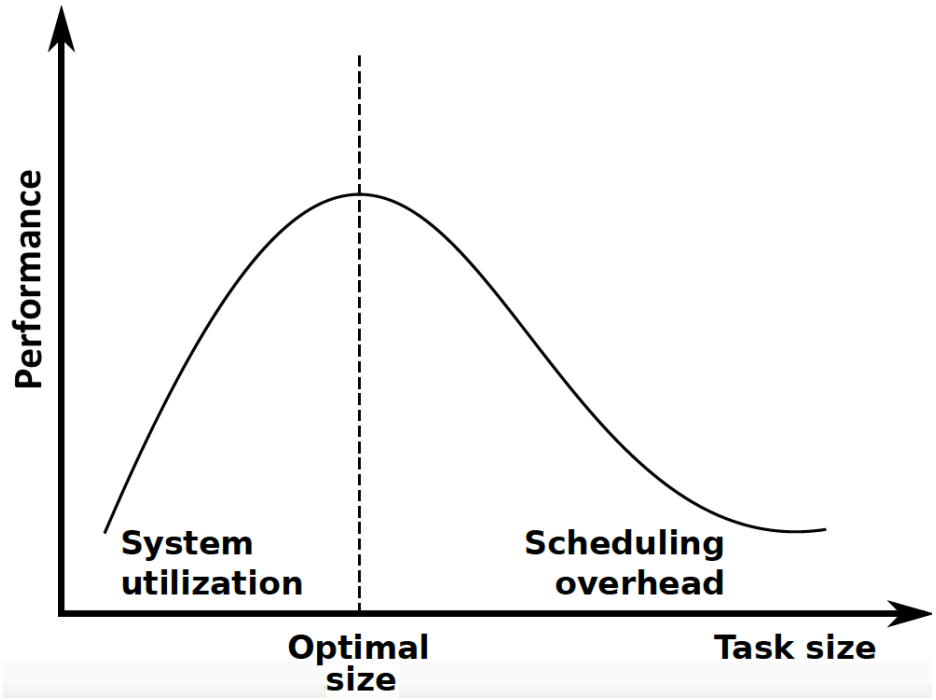
(c) Heat

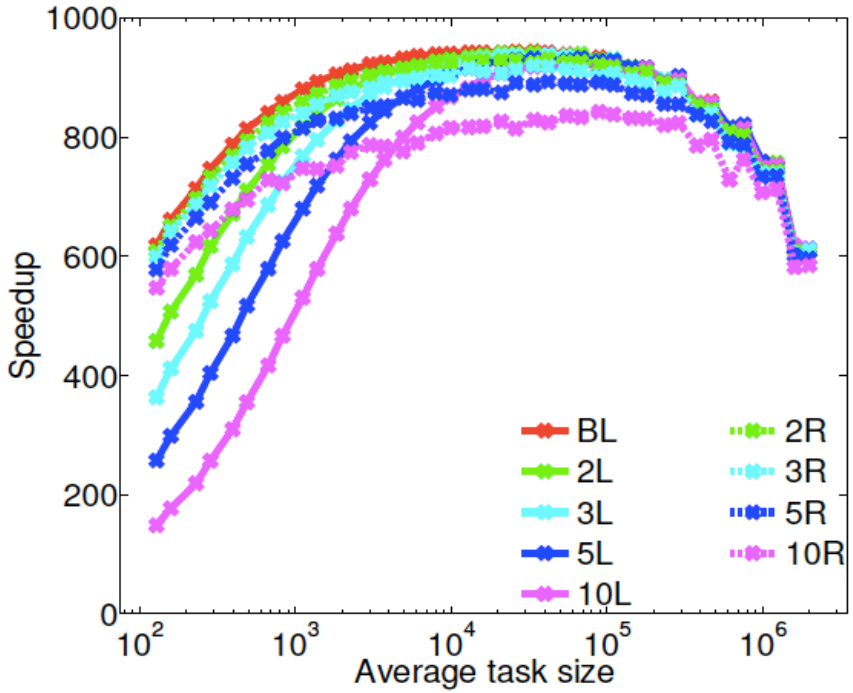


(d) Jacobi

Task granularity analysis

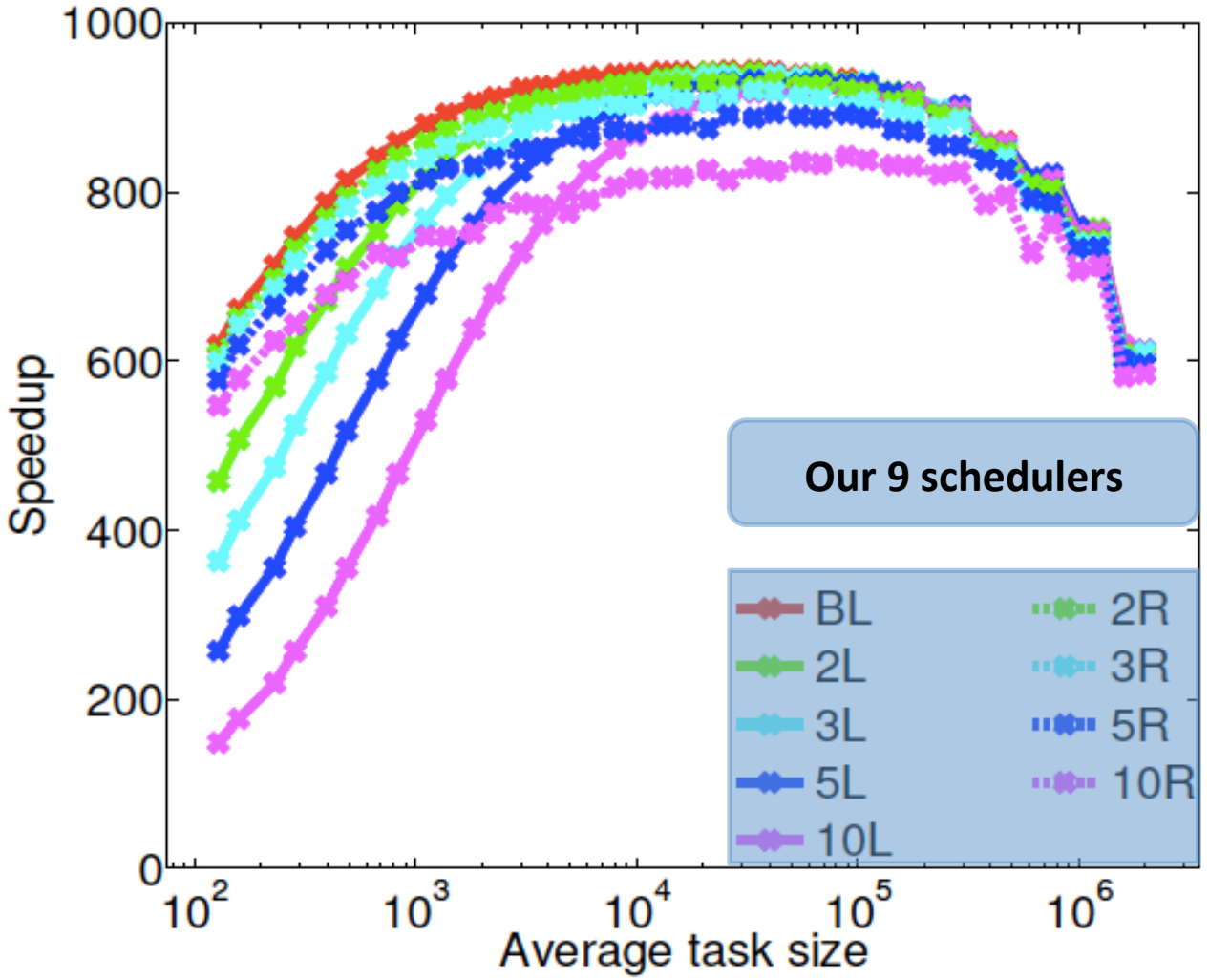
Speedup as function of ave. task size (logarithmic scale on the X-axis):





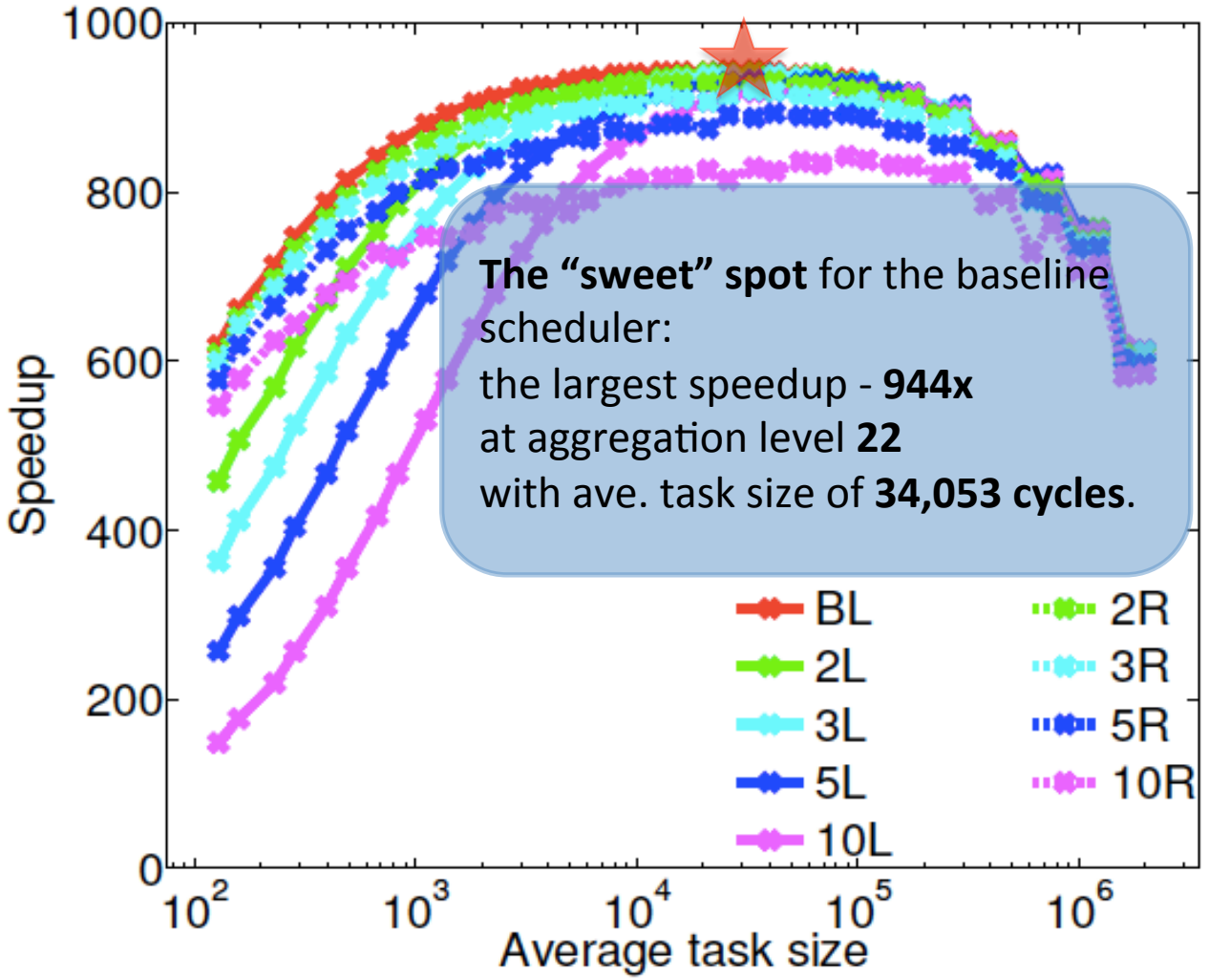
(a) Fib

Task granularity analysis



(a) Fib

Task granularity analysis



(a) Fib

Results

The interplay between the task granularity and the scheduling overhead has **considerable consequences** on the application scalability and efficiency.

Optimal task granularity is **between 12,000 and 100,000 cycles** for representative schedulers.

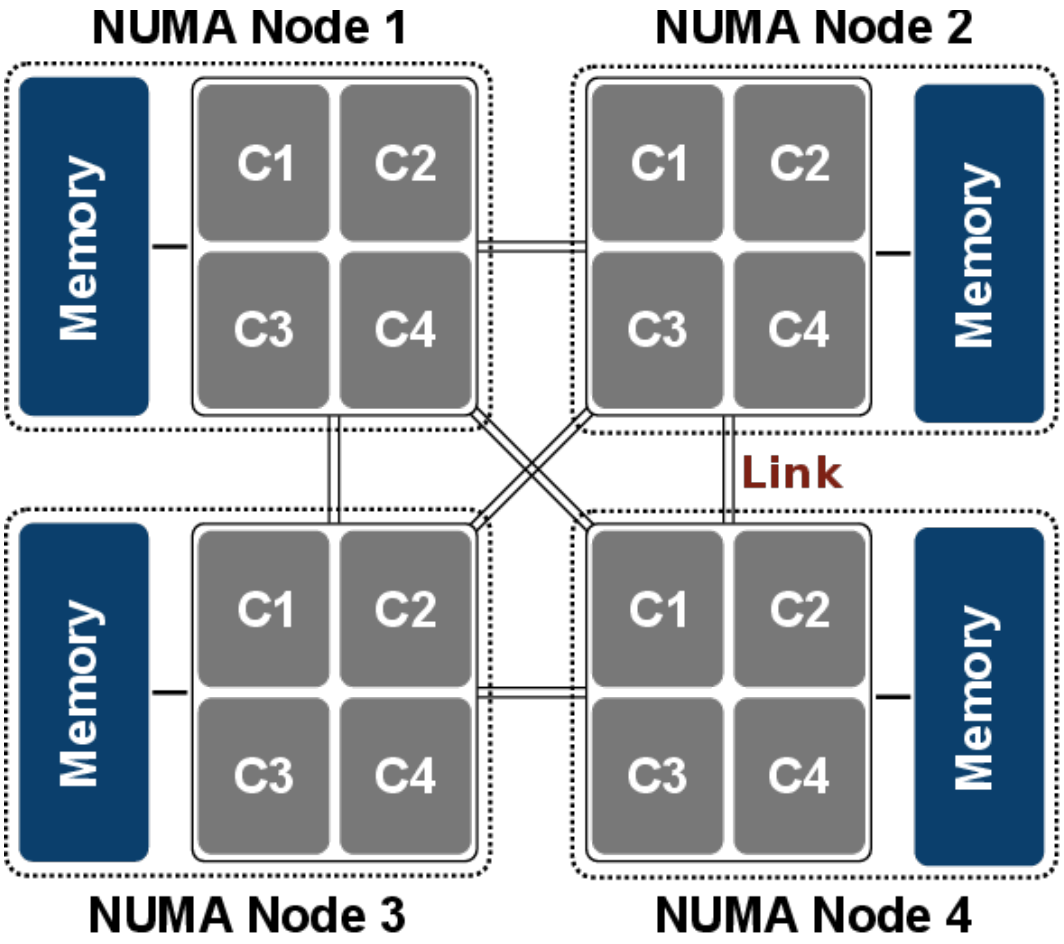
The speedup with 1,024 cores can improve **from 681x to 984x** (for Integrate) when decomposing an application into tasks of optimal granularity.

A promising approach for future Exascale programming models:
to employ a best-effort local scheduler and
a sophisticated (data-locality/workload-aware) remote scheduler.

NUMA architecture

- A NUMA architecture connects different NUMA nodes (Nodes 1-4) - typically multi-core CPUs (C1-C4) - via interconnect links (Link), to enable a single logically shared global memory.

- However, memory access times (latency) - and possibly throughput as well - are reduced when, for instance, Node 1 accesses memory associated with Node 3, due to overhead introduced by the link. This effect is typically referred to as NUMA effect.



http://www.iue.tuwien.ac.at/phd/weinbub/dissertationsu16.html

Data locality

- The probability of a memory reference being "local" to prior memory accesses.
- Task stealing often results in data migration: when threads steal tasks, they also often take a working set of these tasks.
- Task size affects locality as larger tasks also have larger memory footprints, so task granularity should be tuned for efficient use of the memory hierarchy.

Data-locality aware TBPMs

- Classical random work-stealing TBPMs (Cilk and OpenMP) don't support it.

- Hierarchical work-stealing policies: checking a task to be stolen firstly among cores in a local NUMA domain and only then among remote domains (Qthreads).

Data-locality sensitivity study

- Cilk, OpenMP, Qthreads + serial versions
- Same algorithmic structure
- Hardware performance counters (approx. 20)

Task-based applications

- FFT: an implementation of the basic Cooley-Tukey algorithm of a recursive complex Fast Fourier Transformation on the n complex components of the array.

- HPCCG: a conjugate gradient solver approximation to an unstructured implicit finite element or finite volume application.

- LUD: a divide-and-conquer LU decomposition of NxN matrix, where N is at least 16 and a power of 2.

- Madness: a real application kernel from the computational chemistry domain, it projects a 3D analytic function into its numerical representation through adaptive spatial decomposition over the [0..1] interval.

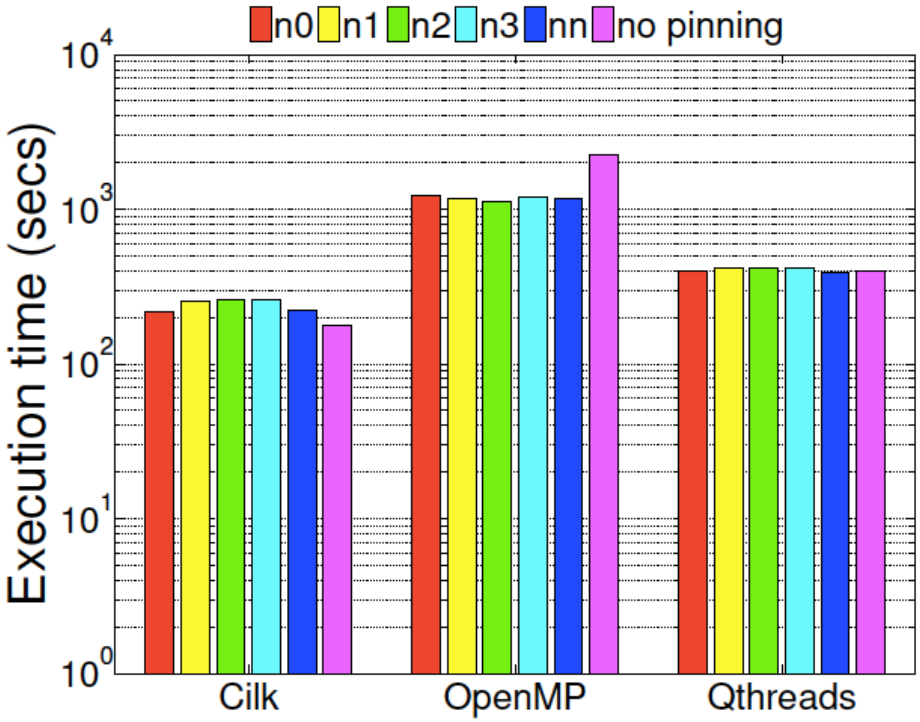
- MxM: a matrix-matrix multiplication kernel.

Methodology

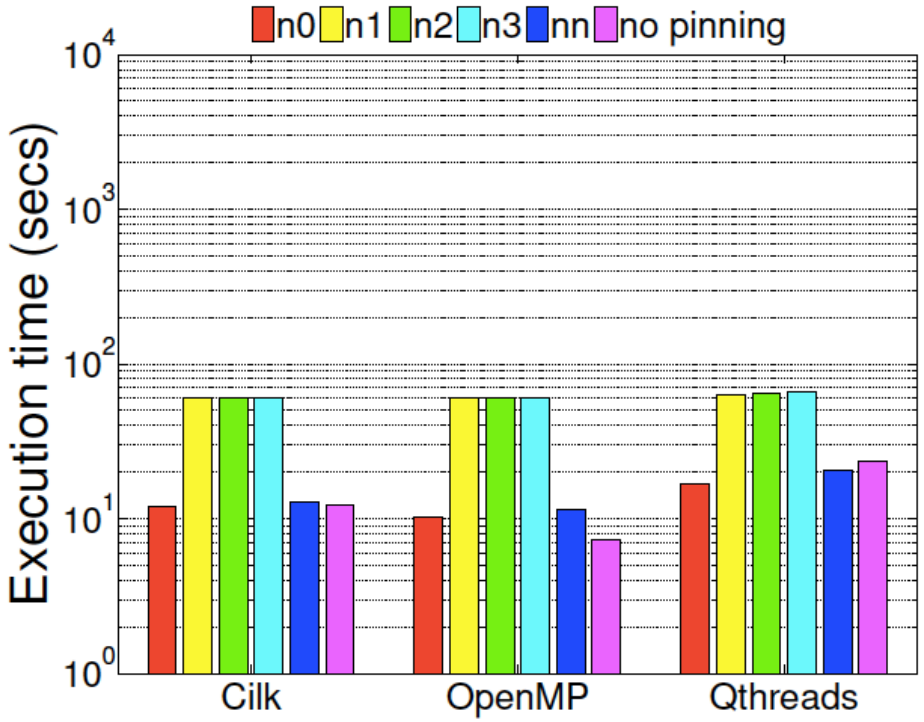
- 6 scenarios to run applications on different NUMA domains of a Tegnér node.
- Binding execution and memory on different NUMA nodes.

Test case	Computation	Memory
n0	NUMA 0 (CPUs 0-11)	NUMA 0
n1	NUMA 0 (CPUs 0-11)	NUMA 1
n2	NUMA 0 (CPUs 0-11)	NUMA 2
n3	NUMA 0 (CPUs 0-11)	NUMA 3
nn	NUMA 0 (CPUs 0-11)	no pinning
no pinning	no pinning	no pinning

Experimental results: execution time

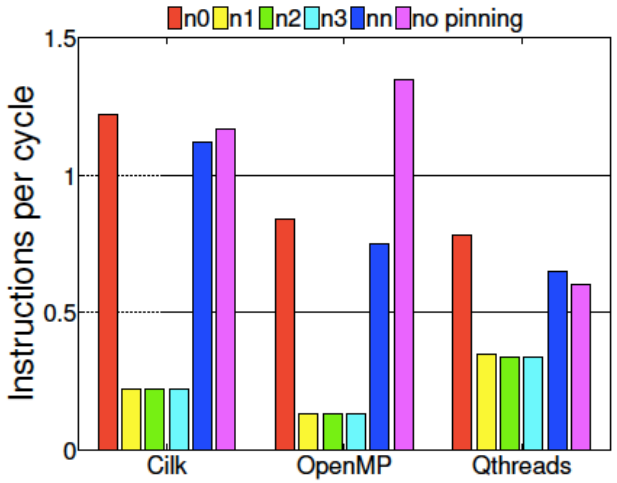


(a) FFT

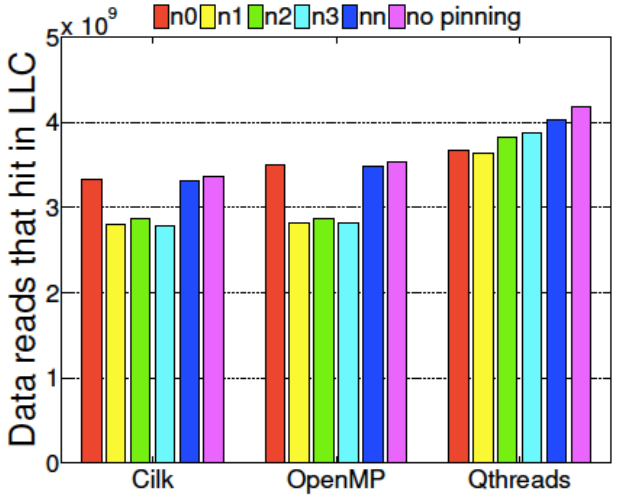


(b) HPCCG

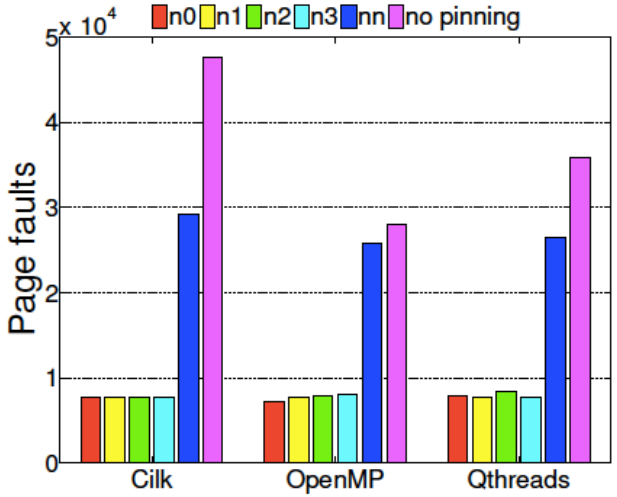
Experimental results for HPCCG



Instructions per cycle for HPCCG



All demand data reads that hit in the LLC



Page faults

Results

- There are data-locality sensitive and data-locality non-sensitive applications.
- An application written in one programming model can be slower than in other models.
- For our test applications, the Cilk versions have the best performance rather than the Qthreads and the OpenMP versions.

Conclusions and future steps

- The studied models should be aware of the data placement, e.g. by providing mechanism to schedule tasks in a way that minimizes data movement.

- Other applications will be studied on data-locality sensitivity.

- Energy and power measurements studies by accessing the RAPL counters as data movement induces energy costs.

Thank you!