TECHNISCHE UNIVERSITÄT CHEMNITZ

JÜLICH Forschungszentrum

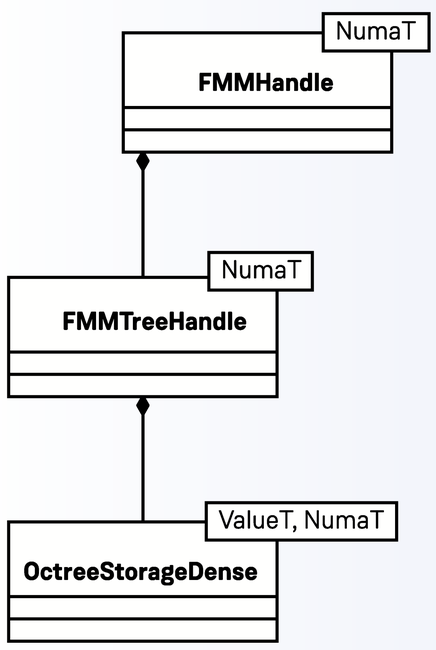
NUMA-Awareness as a Plug-In

L. Morgenstern[1,2], D. Haensel[1], A. Beckmann[1], I. Kabadshow[1]
[1]Jülich Supercomputing Centre, Forschungszentrum Jülich
[2]Operating Systems Group, Chemnitz University of Technology

Motivation

Molecular dynamics (MD) has become a vital research method in biochemistry and materials science. GROMACS aims do develop a flexible and unified tool-box in the field of MD simulations. In MD, the **fast multipole method** (FMM) is used to compute all pairwise long-range interactions between N particles in time O(N). To tackle exascale, MD applications – as well as several other HPC applications – have to target strong scaling. To meet the according requirements such as synchronization- and latency-awareness, software needs to adopt to specific hardware properties such as caching and non-uniform memory access (NUMA). This poster shows how we added **NUMA-awareness** to our **C++ tasking framework for fine-grained parallelism** with an FMM as use case. However, the poster has an emphasis on separation of concerns through software architecture since the representation of NUMA in software is not only relevant for the FMM but should be reusable by similar applications.

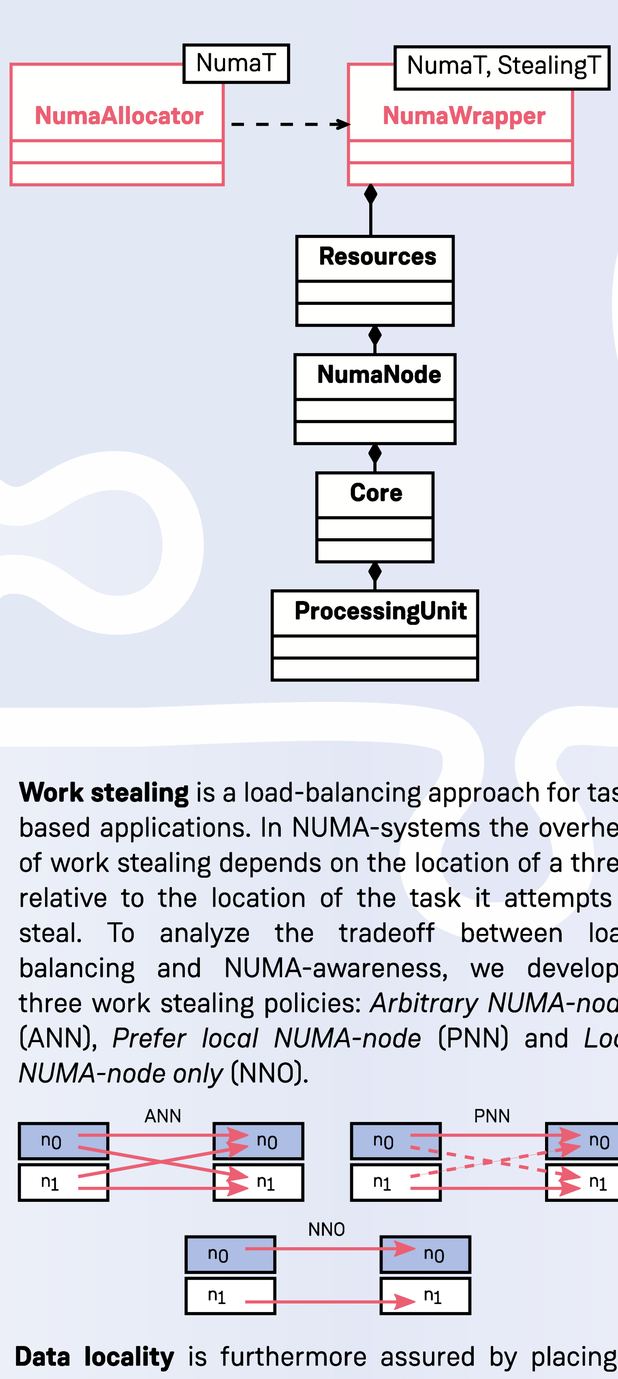
Fast Multipole Method



Workflow:

1. Generate octree through hierarchical space decomposition
2. Expand particles on lowest level into multipole moments
3. M2M: Transfer multipole moments upwards
4. M2L: Translate multipole moments into local moments
5. L2L: Transfer local moments downwards
6. Translate local moments to particles on lowest level
7. Evaluate near field interactions

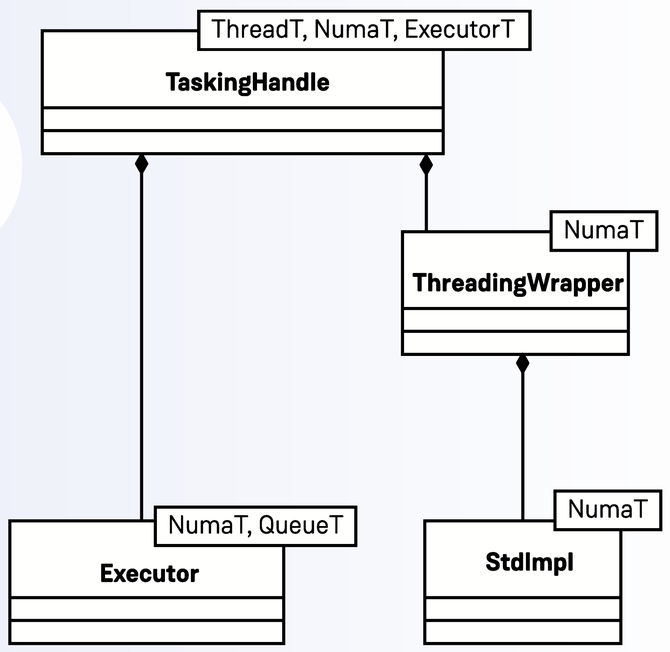
NUMA-Awareness



Work stealing is a load-balancing approach for task-based applications. In NUMA-systems the overhead of work stealing depends on the location of a thread relative to the location of the task it attempts to steal. To analyze the tradeoff between load-balancing and NUMA-awareness, we developed three work stealing policies: *Arbitrary NUMA-nodes* (ANN), *Prefer local NUMA-node* (PNN) and *Local NUMA-node only* (NNO).

Data locality is furthermore assured by placing a thread and its data on the same NUMA-node.
To distribute the workload as equally as possible, the assignment of data to threads is realized through equal partitioning of the FMM-tree levels. Based thereon, we developed the following data and thread placement policies: *Scatter Principally* (SP), *Compact Ideally* (CI) and *Compact Scatter* (CS).

Tasking Framework



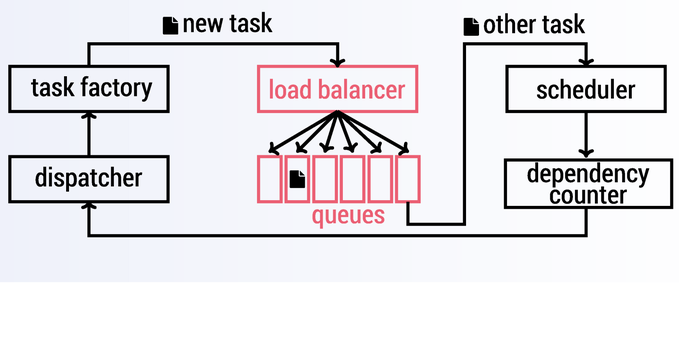
Goal: support synchronization- and latency-critical applications via a low-overhead tasking framework.
Requirements:

- **Sustainability**. Beyond the project's lifetime.
- **Separation of concerns**. Don't bother chemists and biologists with dirty hardware details. And, don't bother computer scientists with chemistry;-)

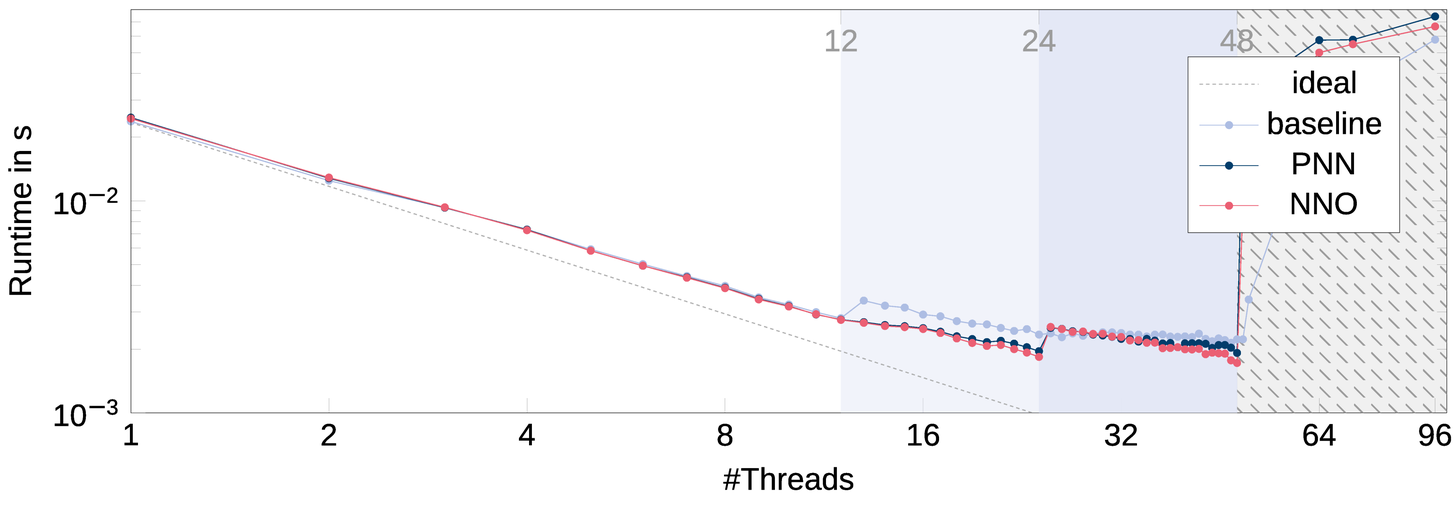
Main ideas:

- Static data-flow dispatching
- Ready-to-execute-tasks
- Typed priority queues

Implementation based on language-inherent C++ features such as std::thread and template meta programming only. No OpenMP, no OpenACC, no #pragma at all.



Results



- **Input data set**: 1000 particles, multipole order p=3, tree depth d=3
- **Hardware**: single JURECA compute node with 2 E5-2680 v3 CPUs with 12 cores and 2-way SMT each (48 SMT threads overall)
- **Method**: each run covers a single time step of the FMM with 1000 repetitions for averaging
- **Ideal**: runtime of single-threaded FMM divided by #Threads
- **Baseline**: runtime without inherent NUMA-awareness; use of numactl to emulate UMA-system for #Threads ≤ 12
- **PNN**: NUMA-aware thread placement CI and work stealing policy PNN
- **NNO**: NUMA-aware thread placement CI and work stealing policy NNO
- **Outcome**: separation of algorithm and hardware pays off; NUMA-Plug-In leads to effective performance improvement of 24%.