Towards Supporting Heterogeneous Hardware in GROMACS

L. Morgenstern[1,2], A. Beckmann[2], I. Kabadshow[2]

[1]Operating Systems Group, Faculty of Computer Science, Chemnitz University of Technology
[2]Institute for Advanced Simulation, Jülich Supercomputing Centre, Research Centre Jülich

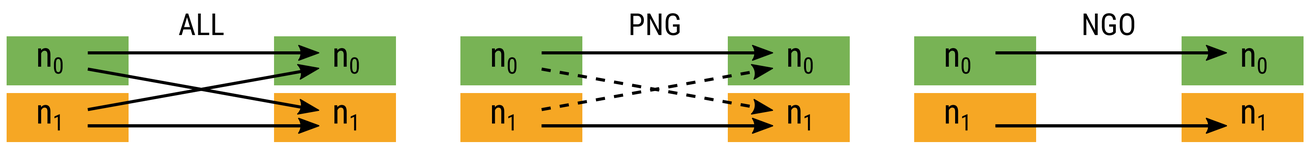
Project – GROMEX (2016 - 2019)

Molecular dynamics (MD) has become a vital research method in biochemistry and materials science. As part of SPPEXA, GROMEX (GROMACS on the Exascale) aims to develop a flexible and unified tool-box in the field of MD simulations on the exascale. In MD, the fast multipole method (FMM) is used to reduce the complexity of the computation of pairwise long-range interactions. Since the problem size is typically fixed by the physical size of molecules, MD applications target strong scaling. Thus, the computational effort per compute node is very low and MD applications tend to be latency- and synchronization-critical. To meet the requirements of latency-critical applications, we need to consider hardware properties such as non-uniform memory access (NUMA).

Approach 1 – Work Stealing

Work stealing is a load-balancing approach for task-based applications. As soon as it runs out of work, a thread may steal and execute tasks from other threads. Provided that the overhead for stealing tasks is sufficiently low, this leads to a shorter time to solution. In NUMA-systems this overhead depends on the location of the thread relative to the location of the task it attempts to steal. To analyze the tradeoff between load-balancing and NUMA-awareness, we present the following work stealing policies:

- Steal from arbitrary NUMA-nodes (ALL)
- Preferably steal from local NUMA-node (PNG)
- Steal from local NUMA-node only (NGO)

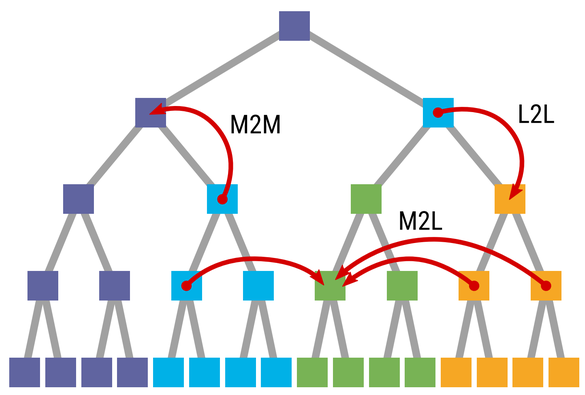


Approach 2 – Data and Thread Placement

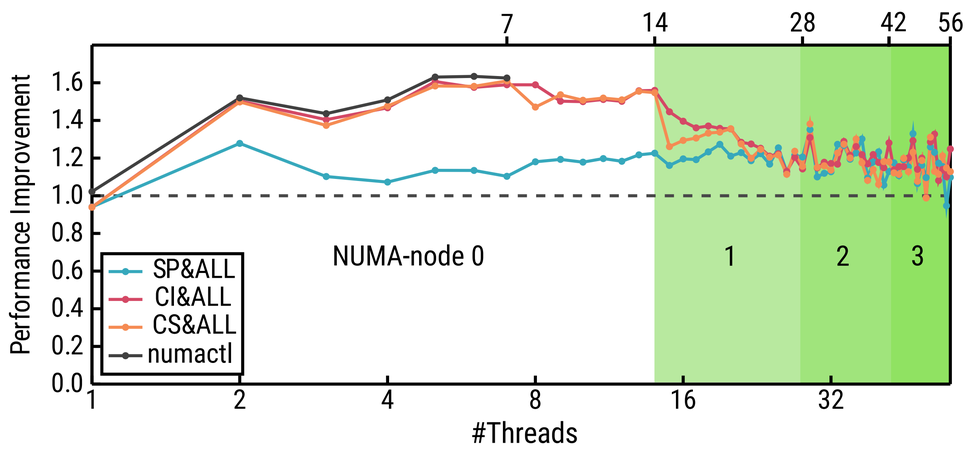
To distribute the workload as equally as possible, the assignment of data to threads is realized through equal partitioning of the FMM-tree levels. Based thereon, data locality is assured by placing a thread and its data on the same NUMA-node. Sticking to these principles, we present the data and thread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

When using SP, a classical load-balancing approach is applied. In general case, this means that all NUMA-nodes are used, with an equal amount of threads being assigned to each NUMA-node. Using CI means in turn that as few as possible NUMA-nodes are used. This is implemented by assigning threads to a single NUMA-node as long as the NUMA-node has got idle cores. Not before all non-SMT (Simultaneous Multi-Threading) cores of a NUMA-node are busy, the next NUMA-node is filled up with threads. CS is an experimental combination of SP and CI that serves the analysis of NUMA dependent on SMT.

Besides modifying the source code to implement NUMA-awareness, the command line tool numactl has been used with options cpunodebind=0 and membind=0 to place threads and data on NUMA-node 0. However, this is only reasonable, if the number of threads is smaller than the number of cores on NUMA-node 0.

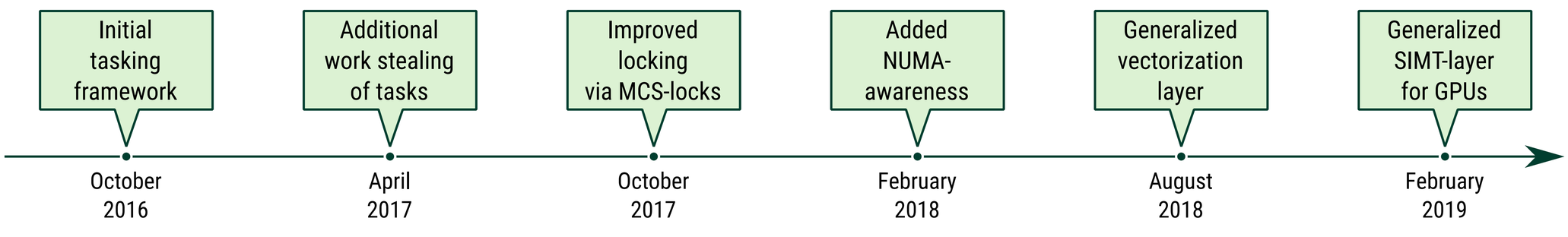


Preliminary Results



- 1000 particles, multipoleorder p=3, depth d=3 on 2 Intel Xeon E5-2680 v4 CPUs with 14 cores and 2-way SMT; 4 (Cluster-on-Die) NUMA-nodes
- Baseline for performance improvement is our FMM implementation without any adjustment to NUMA-architectures, even without numactl
- NUMA-aware data placement leads to an improvement in performance of upto 60%
- Work stealing only pays off between threads running on the same NUMA-node
- Peaks in performance improvement are due to SMT
- Performance improvement through NUMA-awareness increases with increasing number of NUMA-nodes

Roadmap



JÜLICH Forschungszentrum

MAX-PLANCK-GESELLSCHAFT

TECHNISCHE UNIVERSITÄT CHEMNITZ