NUMA-Awareness as a Plug-In for an Eventify-based Fast Multipole Method

Laura Morgenstern[1,2], David Haensel[1], Andreas Beckmann[1], and Ivo Kabadshow[1]

[1] Forschungszentrum Jülich, Jülich Supercomputing Centre, 52425 Jülich, Germany
{l.morgenstern,d.haensel,a.beckmann,i.kabadshow}@fz-juelich.de
[2] Technische Universität Chemnitz, Faculty of Computer Science, 09111 Chemnitz, Germany

Abstract. Following the trend towards Exascale, today's supercomputers consist of increasingly complex and heterogeneous compute nodes. To exploit the performance of these systems, research software in HPC needs to keep up with the rapid development of hardware architectures. Since manual tuning of software to each and every architecture is neither sustainable nor viable, we aim to tackle this challenge through appropriate software design. In this article, we aim to improve the performance and sustainability of FMSolvr, a parallel Fast Multipole Method for Molecular Dynamics, by adapting it to Non-Uniform Memory Access architectures in a portable and maintainable way. The parallelization of FMSolvr is based on Eventify, an event-based tasking framework we co-developed with FMSolvr. We describe a layered software architecture that enables the separation of the Fast Multipole Method from its parallelization. The focus of this article is on the development and analysis of a reusable NUMA module that improves performance while keeping both layers separated to preserve maintainability and extensibility. By means of the NUMA module we introduce diverse NUMA-aware data distribution, thread pinning and work stealing policies for FMSolvr. During the performance analysis the modular design of the NUMA module was advantageous since it facilitates combination, interchange and redesign of the developed policies. The performance analysis reveals that the runtime of FMSolvr is reduced by 21% from 1.48 ms to 1.16 ms through these policies.

Keywords: non-uniform memory access · multicore programming · software architecture · fast multipole method

1 Introduction

1.1 Challenge

The trend towards higher clock rates stagnates and heralds the start of the exascale era. The resulting supply of higher core counts leads to the rise of Non-Uniform Memory Access (NUMA) systems for reasons of scalability. This

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_31

requires not only the exploitation of fine-grained parallelism, but also the handling of hierarchical memory architectures in a sustainable and thus portable way. We aim to tackle this challenge through suitable software design since manual adjustment of research software to each and every architecture is neither sustainable nor viable for reasons of development time and staff expenses.

Our use case is FMSolvr, a parallel Fast Multipole Method (FMM) for Molecular Dynamics (MD). The parallelization of FMSolvr is based on Eventify, a tailor-made tasking library that allows for the description of fine-grained task graphs through events [7, 8]. Eventify and FMSolvr are published as open source under LGPL v2.1 and available at www.fmsolvr.org. In this article, we aim to improve the performance and sustainability of FMSolvr by adapting it to NUMA architectures through the following contributions:

1. A layered software design that separates the algorithm from its parallelization and thus facilitates the development of new features and the support of new hardware architectures.
2. A reusable NUMA module for Eventify that models hierarchical memory architectures in software and enables rapid development of algorithm-dependent NUMA policies.
3. Diverse NUMA-aware data distribution, thread pinning and work stealing policies for FMSolvr based on the NUMA module.
4. A detailed comparative performance analysis of the developed NUMA policies for the FMM on two different NUMA machines.

1.2 State of the Art

MD has become a vital research method in biochemistry, pharmacy and materials science. MD simulations target strong scaling since their problem size is typically small. Thus, the computational effort per compute node is very low and MD applications tend to be latency- and synchronization-critical. To exploit the performance of NUMA systems, MD applications need to adapt to the differences in latency and throughput caused by hierarchical memory.

In this work, we focus on the FMM [6] with computational complexity $\mathcal{O}(N)$. We consider the hierarchical structure of the FMM as a good fit for hierarchical memory architectures. We focus on the analysis of NUMA effects on FMSolvr as a demonstrator for latency- and synchronization-critical applications.

We aim at a NUMA module for FMSolvr since various research works [1, 4, 3] prove the positive impact of NUMA awareness on performance and scalability of the FMM. Subsequently, we summarize the efforts to support NUMA in current FMM implementations.

ScalFMM [3] is a parallel, C++ FMM-library. The main objectives of its software architecture are maintainability and understandability. A lot of research about task-based and data-driven FMMs is based on ScalFMM. The authors devise the parallel data-flow of the FMM for shared memory architectures in [3] and for distributed memory architectures in [2]. In ScalFMM NUMA policies can be set by the user via the OpenMP environment variables `OMP_PROC_BIND`

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_31

and `OMP_PLACES`. However, to the best of our knowledge, there is no performance analysis regarding these policies for ScalFMM.

KIFMM [15] is a kernel-independent FMM. In [4] NUMA awareness is analyzed dependent on work unit granularity and a speed-up of 4 on 48 cores is gained.

However, none of the considered works provides an implementation and comparison of different NUMA-aware thread pinning and work stealing policies. From our point of view, the implementation and comparison of different policies is of interest since NUMA awareness is dependent on the properties of the input data set, the FMM operators and the hardware. Therefore, the focus of the current work is to provide a software design that enables the rapid development and testing of multiple NUMA policies for the FMM.

2 Fundamentals

2.1 Sustainability

We follow the definition of sustainability provided in [11]:

Definition 1. ***Sustainability****. A long-living software system is sustainable if it can be cost-efficiently maintained and evolved over its entire life-cycle.*

According to [11] a software system is further on *long-living* if it must be operated for more than 15 years. Due to a relatively stable problem set, a large user base and the great performance optimization efforts, HPC software is typically long-living. This holds e.g. for the molecular dynamics software GROMACS or Coulomb-solvers such as ScaFaCos. Fortran FMSolvr (included in ScaFaCos), the predecessor of C++ FMSolvr, is roughly 20 years old. Hence, sustainability is our main concern regarding C++ FMSolvr.

According to [11], sustainability comprises non-functional requirements such as maintainability, modifiability, portability and evolvability. We add performance and performance portability to the properties of sustainability, to adjust the term to software development in HPC.

2.2 Performance Portability

Regarding performance portability, we follow the definition provided in [14]:

Definition 2. ***Performance Portability****. For a given set of platforms H, the performance portability Φ of an application a solving problem p is:*

$$\Phi(a,p,H) = \begin{cases} \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases}$$

where $e_i(a,p)$ is the performance efficiency of application a solving problem p on platform i.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_31

2.3 Non-Uniform Memory Access

NUMA is a shared memory architecture for modern multi-processor and multi-core systems. A NUMA system consists of several NUMA nodes. Each NUMA node is a set of cores together with their local memory. NUMA nodes are connected via a NUMA interconnect such as Intel's Quick Path Interconnect (QPI) or AMD's HyperTransport.

The cores of each NUMA node can access the memory of remote NUMA nodes only by traversing the NUMA interconnect. However, this exhibits notably higher memory access latencies and lower bandwidths than accessing local memory. According to [12], remote memory access latencies are about 50% higher than local memory access latencies. Hence, memory-bound applications have to take data locality into account to exploit the performance and scalability of NUMA architectures.

2.4 Fast Multipole Method

Sequential Algorithm. The fast multipole method for MD is a hierarchical fast summation method (HFSM) for the evaluation of Coulombic interactions in N-body simulations. The FMM computes the Coulomb force $\mathbf{F}_i$ acting on each particle i, the electrostatic field $\mathbf{E}$ and the Coulomb potential Φ in each time step of the simulation. The FMM reduces the computational complexity of classical Coulomb solvers from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ by use of hierarchical multipole expansions for the computation of long-range interactions.

The algorithm starts out with a hierarchical space decomposition to group particles. This is done by recursively bisecting the simulation box in each of its three dimensions. We refer to the developing octree as FMM tree. The input data set to create the FMM tree consists of location $\mathbf{x_i}$ and charge q_i of each particle i in the system as well as the algorithmic parameters multipole order p, maximal tree depth d_{max} and well-separateness criterion ws. All three algorithmic parameters influence the time to solution and the precision of the results.

We subsequently introduce the relations between the boxes of the FMM tree referring to [5], since the data dependencies between the steps of the algorithm are based on those:

- **Parent-child relation**: We refer to box x as parent box of box y if x and y are directly connected when moving towards the root of the tree.
- **Near Neighbor**: We refer to two boxes as near neighbors if they are at the same refinement level and share a boundary point.
- **Interaction Set**: We refer to the interaction set of box i as the set consisting of the children of the near neighbors of i's parent box which are well separated from i.

Based on the FMM tree, the sequential workflow of the FMM referring to [10] is stated in Algorithm 1, with steps 1. to 5. computing far field interactions and step 6. computing near field interactions.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_31

Algorithm 1 Fast Multipole Method

Input: Positions and charges of particles
Output: Electrostatic field **E**, Coulomb forces **F**, Coulomb potential Φ

0. Create FMM Tree:
Hierarchical space decomposition
1. Particle to Multipole (P2M):
Expansion of particles in each box on the lowest level of the FMM tree into multipole moments ω relative to the center of the box.
2. Multipole to Multipole (M2M):
Accumulative upwards-shift of the multipole moments ω to the centers of the parent boxes.
3. Multipole to Local (M2L):
Translation of the multipole moments ω of the boxes covered by the interaction set of box i into a local moment μ for i.
4. Local to Local (L2L):
Accumulative downwards-shift of the local moments μ to the centers of the child boxes.
5. Local to Particle (L2P):
Transformation of the local moment μ of each box i on the lowest level into far field force for each particle in i.
6. Particle to Particle (P2P):
Evaluation of the near field forces between the particles contained by box i and its near neighbors for each box on the lowest level by computing the direct interactions.

FMSolvr: An Eventify-based FMM. FMSolvr is a task-parallel implementation of the FMM. According to the tasking approach for FMSolvr with Eventify [7, 8], the steps of the sequential algorithm do not have to be computed completely sequentially, but may overlap. Based on the sequential workflow, we differentiate six types of tasks (P2M, M2M, M2L, L2L, L2P and P2P) that span a tree-structured task graph. Since HFSMs such as the FMM are based on a hierarchical decomposition, they exhibit tree-structured, acyclic task graphs and use tree-based data structures. Figure 1 provides a schematic overview of the horizontal and vertical task dependencies of the FMM implied by this parallelization scheme. For reasons of comprehensible illustration, the dependencies are depicted for a binary tree and thus a one-dimensional system, even though the FMM simulates three-dimensional systems and thus works with octrees. In this work, we focus on the tree-based properties of the FMM since these are decisive for its adaption to NUMA architectures. For further details on the functioning of Eventify and its usage for FMSolvr please refer to [7, 8].

3 Software Architecture

3.1 Layering: Separation of Algorithm and Parallelization

Figure 2 provides an excerpt of the software architecture of FMSolvr by means of UML. FMSolvr is divided into two main layers: the algorithm layer and the

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_31

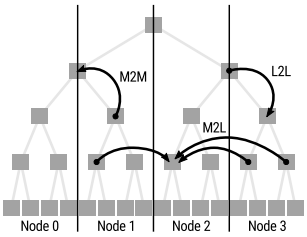

Fig. 1. Exemplary horizontal and vertical data dependencies that lead to inter-node data transfer.

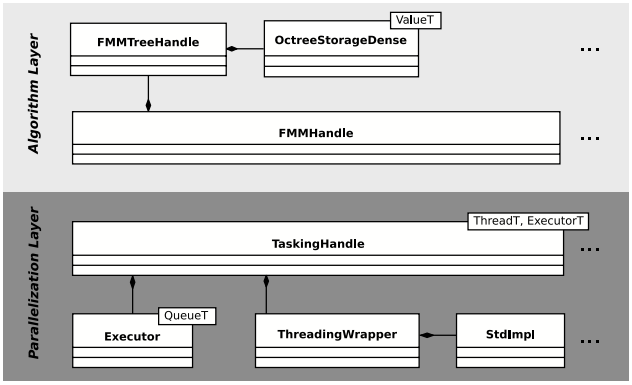

Fig. 2. Layer-based software architecture of FMSolvr. The light gray layer shows an excerpt of the UML-diagram of the algorithm layer that encapsulates the algorithmic details of the FMM. The dark gray layer provides an excerpt of the UML-diagram of the parallelization layer that encapsulates hardware and parallelization details. Both layers are coupled by using the interfaces `FMMHandle` and `TaskingHandle`.

parallelization layer. The algorithm layer encapsulates the mathematical details of the FMM, e. g. exchangeable FMM operators with different time complexities and memory footprints. The parallelization layer hides the details of parallel hardware from the algorithm developer. It contains modules for threading and vectorization and is designed to be extended with specialized modules, e. g. for locking policy, NUMA-, and GPU-support. In this article, we focus on the description of the NUMA module. Hence, Figure 2 contains only that part of the software architecture which is relevant to NUMA. We continuously work on further decoupling of the parallelization layer as independent parallelization library, namely *Eventify*, to increase its reusability and the reusability of its modules, e. g. the NUMA module described in Section 4.

3.2 NUMA-Module: Modeling NUMA in Software

Figure 3 shows the software architecture of the NUMA module and its integration in the software architecture of FMSolvr. Regarding the integration, we aim to

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_31

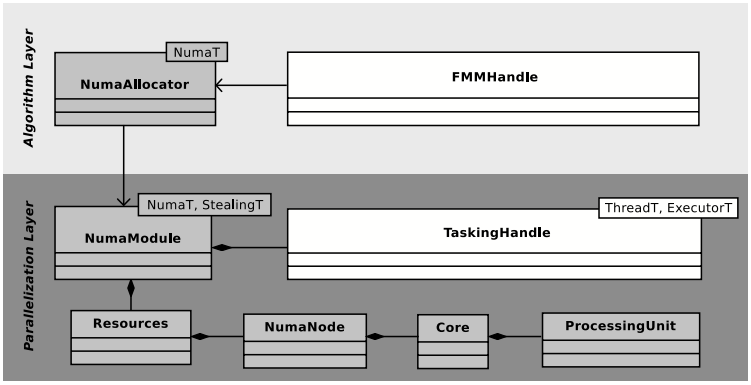

Fig. 3. Software architecture of the NUMA module and its connection to `FMMHandle` and `TaskingHandle`.

preserve the layering and the according separation of concerns as effectively as possible. However, effective NUMA-awareness requires information from the data layout, which is part of the algorithm layer, and from the hardware, which is part of the parallelization layer. Hence, a slight blurring of both layers is unfortunately unavoidable in this case. After all, we preserve modularization by providing an interface for the algorithm layer, namely `NumaAllocator`, and an interface for the parallelization layer, namely `NumaModule`.

Based on the hardware information provided by the NUMA distance table, we model the NUMA architecture of the compute node in software. Hence, a compute node is a `Resource` that consists of `NumaNode`s, which consist of `Core`s, which in turn consist of `ProcessingUnit`s. To reuse the NUMA module for other parallel applications, developers are required to redevelop or adjust the NUMA policies (see Section 4) contained in class `NumaModule` only.

4 Applying the NUMA-Module

4.1 Data Distribution

As described in Section 2.4, the FMM exhibits a tree-structured task graph. As shown in Figure 1, this task graph and its dedicated data are distributed to NUMA nodes through an equal partitioning of the tasks on each tree level. To assure data locality, a thread and the data it works on are assigned to the same NUMA node. Even though this is an important step to improve data locality, there are still task dependencies that lead to data transfer between NUMA nodes. In order to reduce this inter-node data transfer, we present different thread pinning policies.

4.2 Thread Pinning Policies

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_31

Algorithm 2 Equal Pinning

```
r = t mod n
for i = 0; i < n; i++ do
    if i < r then
        // number of threads per node for nodes 0, ..., r − 1
        tpn = ⌊t/n⌋ + 1
        Assign threads i · tpn, ..., (i · tpn) + tpn − 1 to node i
    else
        // number of threads per node for nodes r, ..., n − 1
        tpn = ⌊t/n⌋
        Assign threads i · tpn + r, ..., (i · tpn + r) + tpn − 1 to node i
    end if
end for
```

Equal Pinning. With *Equal Pinning* (EP), we pursue a classical load balancing approach (cf. *Scatter Principally* [13]). This means that the threads are equally distributed among the NUMA nodes. Hence, this policy is suitable for NUMA systems with homogeneous NUMA nodes only.

Algorithm 2 shows the pseudocode for NUMA-aware thread pinning via policy EP. Let t be the number of threads, n be the number of NUMA nodes and tpn be the number of threads per NUMA node.

The determined number of threads is mapped to the cores of each NUMA node in an ascending order of physical core-ids. This means that threads with successive logical ids are pinned to neighboring cores. This is reasonable with regard to architectures where neighboring cores share a cache-module. In addition, strict pinning to cores serves the avoidance of side-effects due to the behavior of the process scheduler, which may vary dependent on the systems state and complicate an analysis of NUMA-effects.

Compact Pinning. The thread pinning policy *Compact Pinning* (CP) combines the advantages of the NUMA-aware thread pinning policies *Equal Pinning* and *Compact Ideally* (cf. [13]). The aim of CP is to avoid data transfer via the NUMA interconnect by using as few NUMA nodes as possible while avoiding the use of SMT.

Algorithm 3 shows the pseudocode for the NUMA-aware thread pinning policy CP. Let c be the total number of cores of the NUMA system and c_n be the number of cores per NUMA node excl. SMT-threads. Considering CP, threads are assigned to a single NUMA node as long as the NUMA node has got cores to which no thread is assigned to. Only if a thread is assigned to each core of a NUMA node, the next NUMA node is filled up with threads. This means especially that data transfer via the NUMA interconnect is fully avoided if $t \leq c_n$ holds. If $t \geq c$ holds, thread pinning policy EP becomes effective to reduce the usage of SMT. For this policy we apply strict pinning based on the neighboring cores principle as described in Section 4.2, too.

CP is tailor-made for the FMM as well as for tree-structured task graphs with horizontal and vertical task dependencies in general. CP aims to keep the

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_31

Algorithm 3 Compact Pinning

```
if t < c then
    for i = 0; i < n; i++ do
        Assign threads i · c_n, ..., (i · c_n) + cn − 1 to node i
    end for
else
    Use policy Equal Pinning
end if
```

vertical cut through the task graph as short as possible. Hence, as few as possible task dependencies require data transfer via the NUMA interconnect.

4.3 Work Stealing Policies

Local and Remote Node. *Local and Remote Node* (LR) is the default work stealing policy that does not consider the NUMA architecture of the system at all.

Prefer Local Node. Applying the NUMA-aware work stealing policy *Prefer Local Node* (PL) means that threads preferably steal tasks from threads located on the same NUMA node. However, threads are allowed to steal tasks from threads located on remote NUMA nodes if no tasks are available on the local NUMA node.

Local Node Only. If the NUMA-aware work stealing policy *Local Node Only* (LO) is applied, threads are allowed to steal tasks from threads that are located on the same NUMA node only. According to [13], we would expect this policy to improve performance if stealing across NUMA nodes is more expensive than idling. This means that stealing a task, e. g. transferring its data, takes too long in comparison to the execution time of the task.

5 Results

5.1 Measurement Approach

The runtime measurements for the performance analysis were conducted on a 2-NUMA-node system and a 4-NUMA-node system. The 2-NUMA-node system is a single compute node of Jureca. The dual-socket system is equipped with two Intel Xeon E5-2680 v3 CPUs (Haswell) which are connected via QPI. Each CPU consists of 12 two-way SMT-cores, meaning, 24 processing units. Hence, the system provides 48 processing units overall. Each core owns an L1 data and instruction cache with 32 kB each. Furthermore, it owns an L2 cache with 256 kB. Hence, each two processing units share an L1 and an L2 cache. L3 cache and main memory are shared between all cores of a CPU.

The 4-NUMA-node system is a quad-socket system equipped with four Intel Xeon E7-4830 v4 CPUs (Haswell) which are connected via QPI. Each CPU

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_31

consists of 14 two-way SMT-cores, meaning, 28 SMT-threads. Hence, the system provides 112 SMT-threads overall. Each core owns an L1 data and instruction cache with 32 kB each. Furthermore, it owns an L2 cache with 256 kB. L3 cache and main memory are shared between all cores of a CPU.

During the measurements Intel's Turbo Boost was disabled. Turbo Boost is a hardware feature that accelerates applications by varying clock frequencies dependent on the number of active cores and the workload. Even though Turbo Boost is a valuable, runtime-saving feature in production runs, it distorts scaling measurements by boosting the sequential run through a higher clock frequency.

The runtime measurements are performed with `high_resolution_clock` from `std::chrono`. FMSolvr was executed 1000× for each measuring point, with each execution covering the whole workflow of the FMM for a single time step of the simulation. Afterwards, the 75%-quantile of these measuring points was computed. This means that 75% of the measured runtimes were below the plotted value. This procedure leads to stable timing results since the influence of clock frequency variations during the starting phase of the measurements is eliminated.

As input data set a homogeneous particle ensemble with only a thousand particles is used. The values of positions and charges of the particles are defined by random numbers in range $[0, 1)$. The measurements are performed with multipole order $p = 0$ and tree depth $d = 3$. Due to the small input data set and the chosen FMM parameters, the computational effort is very low. With this setup, FMSolvr tends to be latency- and synchronization-critical. Hence, this setup is most suitable to analyze the influence of NUMA-effects on applications that aim for strong scaling and small computational effort per compute node.

5.2 Performance

NUMA-aware Thread Pinning We consider the runtime plot of FMSolvr in Figure 4 (top) to analyze the impact of the NUMA-aware thread pinning policies EP and CP on the 2-NUMA-node system without applying a NUMA-aware work stealing. It can be seen that both policies lead to an increase in runtime in comparison to the non-NUMA-aware base implementation for the vast majority of cases. The most considerable runtime improvement occurs for pinning policy CP at #Threads = 12 with a speed-up of 1.6. The reason for this is that data transfer via the NUMA interconnect is fully avoided by CP since all threads fit on a single NUMA node for #Threads $\leq$ 12. Nevertheless, the best runtime is not reached for 12, but for 47 threads with policy CP. Hence, the practically relevant speed-up in comparison to the best runtime with the base implementation is 1.19.

Figure 4 (bottom) shows the runtime plot of FMSolvr with the thread pinning policies EP and CP on the 4-NUMA-node system without applying NUMA-aware work stealing. Here, too, the most considerable runtime improvement is reached for pinning policy CP if all cores of a single NUMA node are in use. Accordingly, the highest speed-up on the 4-NUMA-node system is reached at #Threads = 14 with a value of 2.1. In this case, none of the NUMA-aware implementations of FMSolvr outperforms the base implementation. Even though

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_31

the best runtime is reached by the base implementation at #Threads = 97 with 1.77 ms, we get close to this runtime with 1.83 ms using only 14 threads. Hence, we can save compute resources and energy by applying CP.

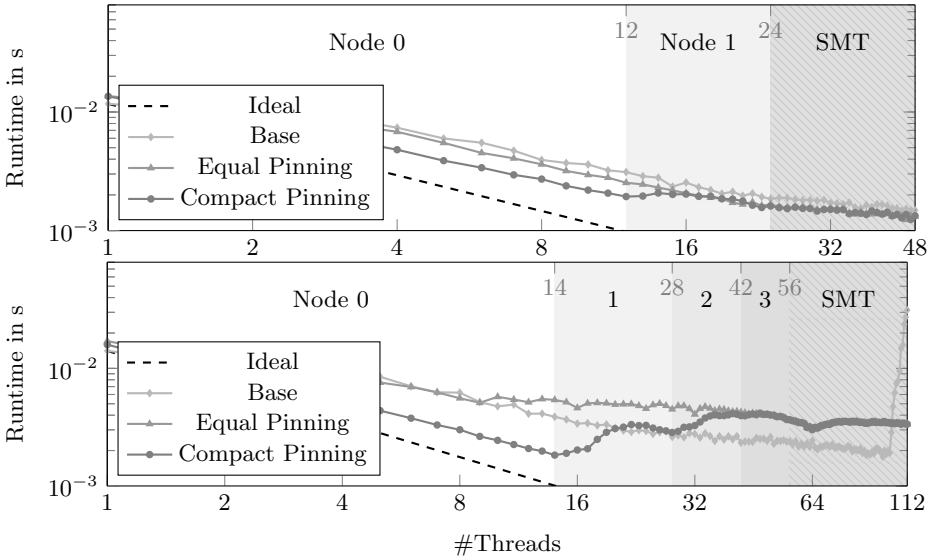

Fig. 4. Comparison of NUMA-aware thread pinning policies EP and CP with work stealing policy LR on 2-NUMA-node and 4-NUMA-node system.

NUMA-aware Work Stealing Figure 5 (top) shows the runtime of FMSolvr for CP in combination with the work stealing policies LR, PL and LO on the 2-NUMA-node system dependent on the number of threads. The minimal runtime of FMSolvr is reached with 1.16 ms for 48 threads if CP is applied in combination with LO. The practically relevant speed-up in comparison with the minimal runtime of the base implementation is 21%.

Figure 5 (bottom) shows the runtime plot of FMSolvr on the 4-NUMA-node for CP in combination with the work stealing policies LR, PL and LO. As already observed for the NUMA-aware thread pinning policies in Section 5.2, none of the implemented NUMA-awareness policies outperforms the minimal runtime of the base implementation with 1.76 ms. Hence, supplementing the NUMA-aware thread pinning policies with NUMA-aware work stealing policies is not sufficient and there is still room for the improvements described in Section 7.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_31

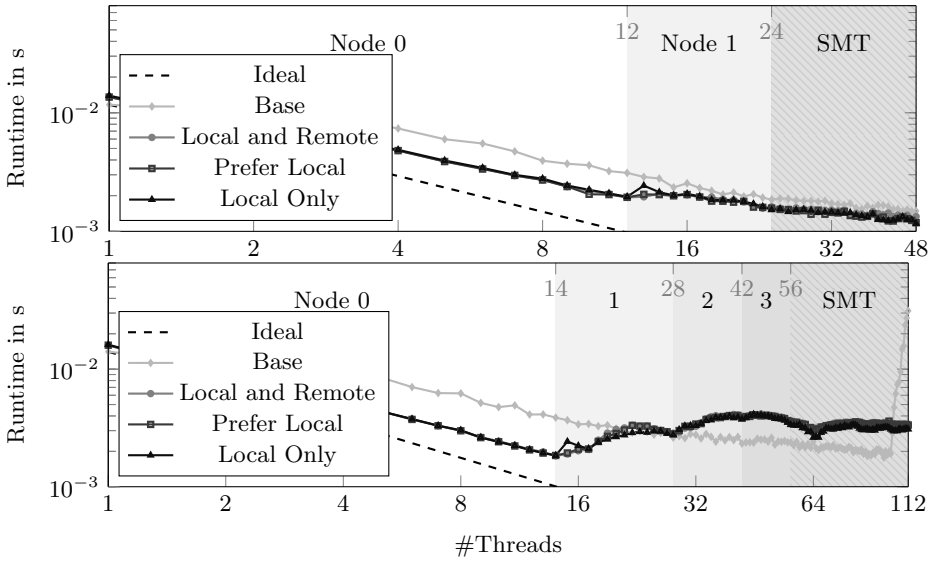

Fig. 5. Comparison of NUMA-aware work stealing policies LR, PL and LO based on NUMA-aware thread pinning policy CP on a 2-NUMA-node and 4-NUMA-node system

5.3 Performance Portability

We evaluate performance portability based on Definition 2 with two different performance efficiency metrics e: architectural efficiency for Φ_{arch} and strong scaling efficiency for Φ_{scal}.

Assumptions regarding Φ_{arch}: The theoretical double precision peak performance P_{2NN} of the 2-NUMA-node system is 960 GFLOPS (2 sockets with 480 GFLOPS [9] each), while the theoretical peak performance P_{4NN} of the 4-NUMA-node system is 1792 GFLOPS (4 sockets with 448 GFLOPS [9] each). With the considered input data set, FMSolvr executes 2.4 Million floating point operations per time step.

Assumptions regarding Φ_{scal}: The considered strong scaling efficiency is for each platform and application determined for the lowest runtime and the according amount of threads for which this runtime is reached.

As can be seen from Table 1 the determined performance portability varies greatly dependent on the applied performance efficiency metric. However, the NUMA Plug-In increases performance portability in both cases.

6 Threats to Validity

Even though we aimed at a well-defined description of our main research objective – a sustainable support of NUMA architectures for FMSolvr, the evaluation of this objective is subject to several limitations.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_31

Table 1. Performance portabilities Φ_{arch} and Φ_{scal} for Eventify FMSolvr with and without NUMA Plug-In.

	Φ_{arch}	Φ_{scal}
FMSolvr without NUMA Plug-In	0.10%	11.22%
FMSolvr with NUMA Plug-In	0.11%	30.41%

We did not fully prove that FMSolvr and the presented NUMA module are sustainable since we analyzed only performance and performance portability in a quantitative way. In extending this work, we should quantitatively analyze maintainability, modifiability and evolvability as the remaining properties of sustainability according to Definition 1.

Our results regarding the evaluation of the NUMA module are not generalizable to its use in other applications or on other NUMA architectures since we so far tested it for FMSolvr on a limited set of platforms only. Hence, the NUMA module should be applied and evaluated within further applications and on further platforms.

The chosen input data set is very small and exhibits a very low amount of FLOPs to analyze the parallelization overhead of Eventify and the occurring NUMA effects. For lack of a performance metric for latency- and synchronization-critical applications, we applied double precision peak performance as reference value to preserve comparability with compute-bound inputs and applications. However, Eventify FMSolvr is not driven by FLOPs, e.g. it does not yet make explicit use of vectorization. In extending this work, we should reconsider the performance efficiency metrics applied to evaluate performance portability.

7 Conclusion and Future Work

Based on the properties of NUMA systems and the FMM, we described NUMA-aware data distribution and work stealing policies with respect to [13]. Furthermore, we present the NUMA-aware thread pinning policy CP based on the performance analysis provided in [13].

We found that the minimal runtime of FMSolvr is reached on the 2-NUMA-node system when thread pinning policy CP is applied in combination with work stealing policy LO. The minimal runtime is then 1.16 ms and is reached for 48 threads. However, none of the described NUMA-awareness policies outperforms the non-NUMA-aware base implementation on the 4-NUMA-node system. This is unexpected and needs further investigation since the performance analysis on another NUMA-node system provided in [13] revealed that NUMA-awareness leads to increasing speed-ups with an increasing number of NUMA nodes. Nevertheless, we can save compute resources and energy by applying CP since the policy leads to a runtime close to the minimal one with considerably less cores.

Next up on our agenda is the implementation of a NUMA-aware pool allocator, the determination of more accurate NUMA distance information and the implementation of a more balanced task graph partitioning. In this article,

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_31

suitable software design paid off regarding development time and application performance. Hence, we aim at further decoupling of the parallelization layer *Eventify* and its modules to be reusable by other research software engineers.

References

1. AbdulJabbar, M.A., Al Farhan, M., Yokota, R., Keyes, D.E.: Performance evaluation of computation and communication kernels of the fast multipole method on intel manycore architecture (2017). https://doi.org/10.1007/978-3-319-64203-1_40
2. Agullo, E., Bramas, B., Coulaud, O., Khannouz, M., Stanisic, L.: Task-based fast multipole method for clusters of multicore processors. Research Report RR-8970, Inria Bordeaux Sud-Ouest (Mar 2017), https://hal.inria.fr/hal-01387482
3. Agullo, E., Bramas, B., Coulaud, O., Darve, E., Messner, M., Takahashi, T.: Task-based fmm for multicore architectures. SIAM Journal on Scientific Computing **36**(1), C66–C93 (2014). https://doi.org/10.1137/130915662
4. Amer, A., Matsuoka, S., Pericàs, M., Maruyama, N., Taura, K., Yokota, R., Balaji, P.: Scaling fmm with data-driven openmp tasks on multicore architectures. In: IWOMP (2016)
5. Beatson, R., Greengard, L.: A short course on fast multipole methods. Wavelets, multilevel methods and elliptic PDEs **1**, 1–37 (1997)
6. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. J. Comput. Phys. **73**(2), 325–348 (Dec 1987). https://doi.org/10.1016/0021-9991(87)90140-9
7. Haensel, D.: A C++-based MPI-enabled Tasking Framework to Efficiently Parallelize Fast Multipole Methods for Molecular. Ph.D. thesis, TU Dresden (2018)
8. Haensel, D., Morgenstern, L., Beckmann, A., Kabadshow, I., Dachsel, H.: Eventify: Event-Based Task Parallelism for Strong Scaling. Accepted at PASC (2020)
9. Intel: APP Metrics for Intel Microprocessors (2020)
10. Kabadshow, I.: Periodic Boundary Conditions and the Error-Controlled Fast Multipole Method. Ph.D. thesis, Bergische Universität Wuppertal (2012)
11. Koziolek, H.: Sustainability evaluation of software architectures: A systematic review. In: Proceedings of the Joint ACM SIGSOFT Conference (2011). https://doi.org/10.1145/2000259.2000263
12. Lameter, C.: NUMA (Non-Uniform Memory Access): An Overview. Queue **11**(7), 40:40–40:51 (Jul 2013). https://doi.org/10.1145/2508834.2513149
13. Morgenstern, L.: A NUMA-Aware Task-Based Load-Balancing Scheme for the Fast Multipole Method. Master's thesis, TU Chemnitz (2017)
14. Pennycook, S.J., Sewall, J.D., Lee, V.W.: A metric for performance portability. CoRR (2016), http://arxiv.org/abs/1611.07409
15. Ying, L., Biros, G., Zorin, D.: A kernel-independent adaptive fast multipole algorithm in two and three dimensions. Journal of Computational Physics (2004). https://doi.org/10.1016/j.jcp.2003.11.021

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_31