Eventify: Event-Based Task Parallelism for Strong Scaling

David Haensel
d.haensel@fz-juelich.de
Jülich Supercomputing Centre
Jülich, Germany

Laura Morgenstern
lmor@hrz.tu-chemnitz.de
Chemnitz University of Technology
Chemnitz, Germany

Andreas Beckmann
a.beckmann@fz-juelich.de
Jülich Supercomputing Centre
Jülich, Germany

Ivo Kabadshow
i.kabadshow@fz-juelich.de
Jülich Supercomputing Centre
Jülich, Germany

Holger Dachsel
h.dachsel@fz-juelich.de
Jülich Supercomputing Centre
Jülich, Germany

ABSTRACT

Today's processors become fatter, not faster. However, the exploitation of these massively parallel compute resources remains a challenge for many traditional HPC applications regarding scalability, portability and programmability. To tackle this challenge, several parallel programming approaches such as loop parallelism and task parallelism are researched in form of languages, libraries and frameworks. Task parallelism as provided by OpenMP, HPX, StarPU, Charm++ and Kokkos is the most promising approach to overcome the challenges of ever increasing parallelism. The aforementioned parallel programming technologies enable scalability for a broad range of algorithms with coarse-grained tasks, e. g. in linear algebra and classical N-body simulation. However, they do not fully address the performance bottlenecks of algorithms with fine-grained tasks and the resultant large task graphs. Additionally, we experienced the description of large task graphs to be cumbersome with the common approach of providing in-, out- and inout-dependencies. We introduce event-based task parallelism to solve the performance and programmability issues for algorithms that exhibit fine-grained task parallelism and contain repetitive task patterns. With user-defined event lists, the approach provides a more convenient and compact way to describe large task graphs. Furthermore, we show how these event lists are processed by a task engine that reuses user-defined, algorithmic data structures. As use case, we describe the implementation of a fast multipole method for molecular dynamics with event-based task parallelism. The performance analysis reveals that the event-based implementation is 52 % faster than a classical loop-parallel implementation with OpenMP.

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**.

KEYWORDS

task parallelism, shared memory, multi-core, strong scaling

PASC '20, June 29-July 1, 2020, Geneva, Switzerland

ACM ISBN 978-1-4503-7993-9/20/06…$15.00
https://doi.org/10.1145/3394277.3401858

ACM Reference Format:
David Haensel, Laura Morgenstern, Andreas Beckmann, Ivo Kabadshow, and Holger Dachsel. 2020. Eventify: Event-Based Task Parallelism for Strong Scaling. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '20), June 29-July 1, 2020, Geneva, Switzerland*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3394277.3401858

1 INTRODUCTION

1.1 Challenge

Today's processors gain their performance through increasing core counts, instead of increasing clock rates. The utilization of these massively parallel compute resources remains a challenge for many traditional HPC applications. While aiming at the computation of larger and more complex scientific problems, non-functional software properties such as scalability, portability and programmability were rarely taken into account. However, sustainable software that runs efficiently on diverse, massively parallel architectures is vital in the upcoming exascale era.

Our goal is to solve the performance portability challenge for HPC-applications that target strong scaling. Our use case is FMSolvr, a state-of-the-art C++-implementation of the Fast Multipole Method (FMM) for molecular dynamics (MD) that can be integrated in MD-codes such as GROMACS [1] or Coulomb-solvers such as ScaFaCoS [2]. FMSolvr is published as open source under LGPL v2.1 and available at www.fmsolvr.org. The FMM is a fast summation technique that computes all pairwise long-range interactions, e. g. coulombic or gravitational interactions, in particle-based simulations. Since the computation of Coulomb interactions is the most expensive part of MD-simulations, its performance and scalability are decisive for the whole simulation [3]. Biochemical MD-simulations performed by GROMACS usually encompass only a few hundred thousand particles. Current supercomputers such as Summit and Sierra provide an equivalent amount of compute cores. Hence, the computational effort per compute core is very low. In contrast to typical HPC-applications, FMSolvr is not compute-bound, but synchronization-critical. Therefore, a parallelization approach must exhibit extremely low runtime overheads, otherwise the task overhead exceeds the task computation time and renders the parallelization ineffective. MD-simulations usually require millions or billions of timesteps. This demands a runtime of a few milliseconds per timestep to reach a feasible total simulation time. Due to the small problem size of typical MD-simulations and the demanded

millisecond-runtime, we cannot rely on scaling the particle system but have to reach sufficient strong scaling efficiency.

To be applicable in large-scale software projects such as GROMACS or ScaFaCoS, a parallel programming approach for FMSolvr needs to fulfill the following requirements:

- **Sustainability and Compatibility**: Provided as a library in standard C++ for reasons of sustainability and compatibility with the software stack on current and future HPC-systems.
- **Maintainability and Extensibility**: Designed with encapsulation, separation of concerns and further object oriented programming concepts in mind to enable maintenance, extension and exchange of components like scheduling policies and synchronization mechanisms.
- **Performance and Scalability**: Reach strong scaling on modern multi-core architectures for fine-grained, dependent tasks.

1.2 State of the Art

This leads to the question to what extent current parallel programming approaches fulfill the aforementioned requirements. The defacto standard for shared memory parallelization in HPC is the exploitation of loop-level parallelism with OpenMP [4]. Depending on the algorithm, this may introduce load imbalances and leave sequential regions; especially in tree-based algorithms like the FMM. Additionally, this approach requires a synthetic way of expressing parallelism in loops instead of algorithmic dependencies. That these limitations hold in particular for the FMM is shown in [5], [6] and [7]. The present work confirms these findings through the performance analysis of a loop-parallel OpenMP-implementation of FMSolvr.

Task-based parallelism emerged as an alternative to tackle the limitations of loop-level parallelism. Task-based approaches aim to reduce sequential regions and synchronization phases by expressing the parallelization of an algorithm along its tasks and task dependencies. Common task-based approaches are provided by Chapel [8], Charm++ [9], Intel TBB [10], HPX [11], Kokkos [12], OpenMP, StarPU [13], PaRSEC [14] and X10 [15]; they are applied to a broad range of applications from data mining via machine learning through linear algebra and molecular dynamics. A comprehensive taxonomy of current task-parallel programming technologies in HPC is provided by Thoman et al. [16]. The authors differentiate three types of technologies: languages, language-extensions and libraries. In contrast to languages (e. g. Chapel, X10) and language-extensions (e. g. OpenMP, Charm++, StarPU), libraries (e. g. HPX, Intel TBB, Kokkos) are dependent on the availability of a regular C++ compiler only. Hence, library-based approaches are beneficial for our use case since they fulfill the sustainability and compatibility requirement. Subsequently, we take a closer look at library-based approaches.

HPX is a parallel runtime system that strictly adheres to the C++ standard and provides a user-friendly API for fine-grained task-parallel programming on shared and distributed memory systems. However, HPX employs a global address space (GAS). While the employed global address space improves programmability, it does not allow for custom communication algorithms which hurts our extensibility requirement. According to [17], the increased amount of thread state information due to GAS leads to decreased application performance. HPX-based FMMs are described in [17] and [18].

Intel TBB is a C++-library that supports data and task parallel programming on shared memory systems. As the successor of deprecated Cilk and Cilk Plus, Intel TBB continues to employ the approach of recursive task spawning to generate parallelism. Since this means that tasks are created even if they are not yet executable, it introduces additional scheduling overhead and increases memory usage.

Kokkos is a C++-library that provides a performance portable, user-friendly programming model for parallel programming on CPUs and GPUs. According to [19], Kokkos reaches 90% of the performance of application-specific parallelization approaches. Due to its general applicability, Kokkos cannot take highly application-specific knowledge such as critical paths or customized task priorities into account. We aim to take such knowledge into account for performance optimization even if this comes at the expense of general applicability.

Due to the application of the FMM beyond MD, e. g. in plasma physics and astrophysics, its parallelization with task- and data-flow based approaches is heavily researched on shared and distributed memory systems, multi-core, many-core and GPU-architectures. However, a comparison of this broad range of FMM applications, especially regarding performance and scalability, remains an open research questions. This is due to the fact that diverse mathematical and technical variants of the FMM operators exist, which lead to different accuracy and performance behaviour. For classification of the present work, we nevertheless provide a short outline on parallel FMM implementations.

ExaFMM [20] is an open source FMM-library that supports shared memory systems via OpenMP, distributed memory systems via MPI as well as GPUs via CUDA. It aims at scaling large particle simulations with billions of particles to exascale systems. For their strong scaling test with 10^8 particles the group reports 93% parallel efficiency on 2048 processes. This result is highly promising for the FMM in particular, as well as hierarchical algorithms in general, to reach excellent scalability on exascale systems. In [21] the task-parallel programming approaches Intel Cilk Plus, Intel TBB and OpenMP tasks are applied to the DTT-part (Dual Tree Traversal) of ExaFMM. The performance analysis reveals that Intel TBB perfectly scales up to 64 cores on KNL for 10^8 particles. Hence, approaches with recursive task spawning are efficient for compute-bound applications; however, no analysis is provided for synchronization-critical applications.

In [22] a data-driven FMM with the runtime system QUARK [23] is described. For particle ensembles with 10^7 particles the approach leads to linear speed-up on 16 cores. The idea of breaking the stages of the FMM into smaller tasks to improve load balancing is furthermore applied and analyzed in [17], [5] and [24]. Since the mentioned works consistently report advantages in scalability, the present work extends the data-driven approach.

ScalFMM [24] is a parallel, C++ FMM-library. It is a kernel independent FMM, while FMSolvr is specialized on kernels with spherical harmonic expansions. Since this may have an impact on the parallelization approach and its performance, this complicates a direct comparison of both implementations. The main objectives of

its software architecture are maintainability and understandability. We consider these properties vital for sustainable HPC software. Hence, FMSolvr pursues similar non-functional software requirements. 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 [24] and for distributed memory architectures in [25]. For the StarPU-based FMM in [24] strong scaling with a parallel efficiency of 91% is reached through choosing a sufficiently large particle ensemble with $2 \cdot 10^8$ particles. Since the number of particles in biochemical simulations can be below 10^6 particles, we cannot follow this approach and increase the amount of particles to reach strong scaling. Instead, the objective of the current work is to find a parallelization approach that delivers sufficiently low overheads.

However, these works focus on the efficient computation of large, often inhomogeneous, particle ensembles with millions of particles. This leads to particular challenges regarding memory footprint, scheduling policies and communication patterns for the applied parallelization approaches. The focus of this work, however, is on the efficient computation of small, homogeneous particle ensembles. Hence, the overhead of the applied parallelization approaches cannot be hidden through computational work, but must be reduced to a minimum.

1.3 Unique Features of Eventify

With *Eventify*, we introduce a low-overhead, library-based, task-parallel programming technology that targets strong scaling and performance portability of applications with many, tiny, dependent tasks. Eventify is published as open source under LGPL v2.1 and available at www.fmsolvr.org. The focus of the present work is on shared memory systems. However, the extension of Eventify to distributed memory systems and GPUs is work in progress.

In comparison to other task-parallel programming technologies the unique features of Eventify are:

- **Event Lists**: A convenient way to describe recurring task dependency patterns in large task graphs.
- **Static Event Dispatcher**: Component of the task engine that efficiently handles event lists.
- **Reuse of Data Structures**: The reuse of user-tuned algorithmic data structures enables efficient dependency description and resolution.
- **Bottom-Up Task Creation**: A task is created as soon as all of its input dependencies are resolved. In comparison to recursive task spawning this reduces scheduling overhead and memory footprint.
- **Type-Driven Priority Queue**: A data structure that stores and priorities tasks of different types. This improves programmability and debuggability since no casting of function pointers and translation of task types into priority values is required.

Bringing together these features enables Eventify to strong scale FMSolvr with its many, tiny, dependent tasks. Eventify is currently applied to FMSolvr only. However, we consider the concept of event-based task-parallelism to be directly applicable to further tree-based algorithms with recurring dependency patterns such as Barnes-Hut tree codes and multigrid methods.

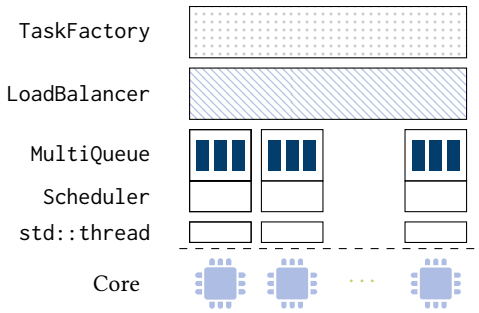

Figure 1: An overview of the task engine.

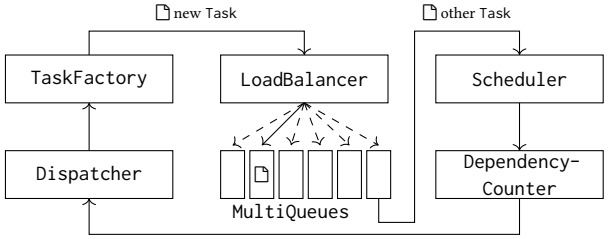

Figure 2: The life-cycle of a task using the task engine.

1.4 Outline

The rest of the article is organized as follows. In Section 2 we describe the software architecture and functionality of Eventify. To keep the article self-contained, we describe the FMM for MD and its parallelization potential in Section 3. Based thereon, its implementation with OpenMP on the one hand, and Eventify on the other hand is provided. Subsequently, a comparative performance analysis is provided in Section 4. Finally, we summarize our results and outline future work in Section 5.

2 EVENTIFY

In this Section, we describe the design and implementation of the API and task engine of Eventify. The focus is on the two main components of the task engine, the type-driven priority scheduler as well as the static event dispatcher.

When aiming for sustainable high performance computing (HPC) software it is vital to stick to certain design goals and guidelines. For the development of the presented task engine we stick to three main software requirements: correctness, maintainability and performance portability. It is obvious that the implementation has to produce correct results and needs to be maintainable. However, performance portability is an extensive goal on its own and needs further explanation. We advocate that it is neither possible nor desirable to hand-tune code for every platform. Due to the fast development of HPC hardware, a generic hardware independent implementation is preferable. A high level of abstraction and encapsulation comes in handy as one tool to achieve performance portable. This means that, library or system calls are only allowed within certain specialized wrappers or interfaces. This leads to exchangeable components by design.

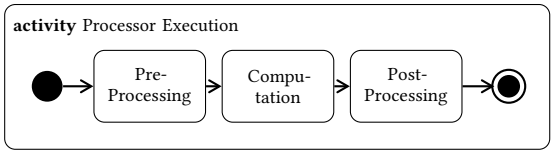

Figure 3: Three phase execution of the processor enabling pre- and post-processing.

Figure 1 shows an overview of the main components of the presented task engine. For each CPU core it consists of three private components:

- a `std::thread`,
- a `Scheduler` and
- a `MultiQueue`.

At the beginning of the program the threads are forked once and assigned to a private scheduler. During the program execution no further fork and join is required since the schedulers reuse the assigned thread. Whenever new tasks are available, the scheduler executes them depending on the scheduling strategy. Every scheduler owns a `MultiQueue`. Depending on the scheduling and load-balancing strategy, the queue can be accessed by other threads to remove or insert tasks. Tasks can be created by any other task running on any thread using the `TaskFactory`. The factory creates new tasks which are always "ready-to-execute" and hands them over to the `LoadBalancer`. The `LoadBalancer` distributes the tasks among the threads and enqueues them into the corresponding `MultiQueue`. The life-cycle of a task is also shown in Figure 2. The high abstraction in the implementation of the task engine leads to a single responsibility of each component, which makes it easy to exchange components whenever necessary. For the scheduling we use a work-stealing scheduler, which starts stealing work from remote queues only if its own queue is empty. Since the tasks do not involve any dependency checks before the execution, all dependencies have to be fulfilled before the task is created and enqueued as "ready-to-execute".

The task itself consists of two parts. The first is a unique identifier used for identifying the work unit of this task. Regarding the Fast Multipole Method (FMM), this is a box index in the octree. The second is a so called `Processor` reflecting the actual task executable. This `Processor` is similar to a function callback encapsulated in a class and shared among the threads. In the presented FMM implementation, each `Processor` reflects a certain FMM operation. The application of the unique identifier onto the execution method of the `Processor` resembles the actual execution of the task. The `Processor` itself is user-defined and needs to be implemented for each task type. The default implementation of the `Processor` is split into three phases to enable pre-processing and post-processing besides the actual computation (see Figure 3).

2.1 Static Event Dispatcher

Data-flow-based parallelization uses a data-centric view and executes operations with respect to the algorithm. For this purpose the data-flow graph needs to be mapped to the program. This can be done with an event dispatcher. Since the data-flow is known at compile-time, it should be possible to configure it at compile-time as well. This leads to a flexible and configurable dispatcher without runtime overhead because all dispatch methods are resolved at compile-time.

2.1.1 Modeling Algorithmic Dependencies. There are two common approaches to model algorithmic dependencies. The first is the dependency graph which models the algorithm in a backward view. The second is the data-flow graph modeling the dependencies in a forward view. Both use the mathematical structure of a graph.

Unfortunately, there is no uniform definition of a dependency graph in literature. Therefore, we call a directed acyclic graph (DAG) a *dependency graph*, if it represents the dependencies of every piece of data. This means, every vertex represent one and only one piece of data and every edge represents one and only one manipulation of it.

Insights into the parallelization can be obtained by the structure of such a dependency graph. There exist several scientific publications showing how to partition those graphs and distribute the work accordingly. Nevertheless, those approaches are limited by the complexity of the optimal partitioning being NP-complete [26]. This might introduce a significant overhead for synchronization-critical applications.

For the parallel execution of an algorithm, the dependencies need to be fulfilled before the next task can be executed. This can be checked using the dependency graph. The drawback of this approach is, that it requires the setup and traversal of the complete graph within the program. Even if it is not required to store the complete graph in memory, this introduces runtime overhead.

Additionally, it requires to create a significant number of tasks upfront. Afterwards, the task engine needs to iterate over all created tasks and check if their dependencies are met. This causes a constant polling on the created tasks for checking the dependencies without computation.

In contrast to the dependency graph, the data-flow graph does not model every single piece of data. The data-flow graph is data-centric and only models pipelines of data manipulations. In the data-flow graph a vertex denotes an abstract operation in the algorithm, whereas an edge denotes the input and output data of an operation. It is important to mention, that edges and vertices only represent abstract data and operations and not a concrete piece of data or operation as in the dependency graph. The start node of the graph is used for the algorithms input data. The input data is used and manipulated by the following operations in the graph until the end of the graph is reached. The last node in the graph represents the output data of the algorithm.

Figure 11 shows the data-flow graph of the FMM. All operations are represented by a single node in the graph and their corresponding input and output data is denoted on the edges. As seen in this example, the data-flow graph may contain cycles and is therefore not acyclic.

2.1.2 Event-based Parallelization. The data-flow model can be implemented using an event-based approach [27, p. 3]. This approach comprises event-sources, that can trigger certain events. These events are dispatched by an event dispatcher that calls the corresponding event listeners. The specialization for the data-flow model uses data events instead of more general event-sources. Those data events reflect the progress of the operations achieved on the data.

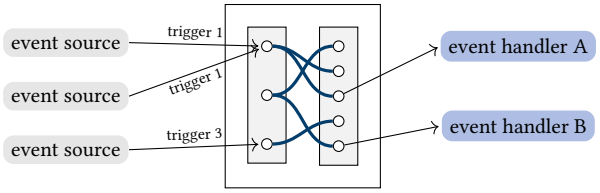

Figure 4: Schematic view of the static event dispatcher.

Conventional data-flow frameworks [28] offer a dynamic interface for creating parts of the dependency graph and for registering event listeners. This part has to be adapted depending on the algorithmic flow. The main contribution of our static event dispatcher is to allow configuration and resolving of the dispatch at compile-time. Resolving the dispatch itself at compile-time leads to higher performance as well as to higher robustness of the actual implementation.

2.1.3 Static Data-Flow Dispatcher. In contrast to run-time based dispatcher implementations like the one available in Intel TBB [28], we want to register the event-listener and resolve the dispatch at compile-time. Therefore, we use template metaprogramming for the configuration of the static event dispatcher. The concrete event dispatcher is thereby only a type definition and not a statefull object. Figure 4 shows a schematic view of the static event dispatcher. In the presented implementation, the wiring inside the dispatcher is evaluated at compile-time.

Listing 1: This listing shows the original `StaticEventDispatcher` used in the FMM implementation. The corresponding data-flow graph can be found in Figure 11. The first `EventListener` defines the following: When the event `OMEGA` is triggered, a new `M2M_Task` is created. The other `EventListeners` work accordingly.

```
using DataFlowDispatcher = EventListenerContainer<
  EventListener<OMEGA, Listener<M2M_Create>>,
  EventListener<OMEGA, Listener<M2L_Create>>,
  EventListener<MU, Listener<L2L_Create>>,
  EventListener<MU_Lowest, Listener<L2P_Create>>>;
```

Listing 2: Call of the dispatch method using the `DataFlowDispatcher`. In this example an omega was computed and the event `OMEGA` is dispatched for this box. The actual follow-up task is determined by the event dispatcher statically. With the concrete event dispatcher from Listing 1 this results in the creation of an `M2M_Task` and an `M2L_Task`.

```
// Run the computation for an omega
compute_omega(box_id);
// Resolve dependencies
EventDispatcher::dispatch<OMEGA>(box_id);
```

To simplify the traversal of the data-flow graph we want to discuss the details using a concrete example. Listing 1 shows the static event dispatcher configuration corresponding to the data-flow of the FMM as shown in Figure 11. An exemplary call to the dispatch method, for dispatching an ω, is shown in Listing 2. In this example, we register four event listeners with the data events called `OMEGA`, `MU` and `MU_Lowest`. When the event is triggered, the corresponding event handler registered with the `Listener` is called.

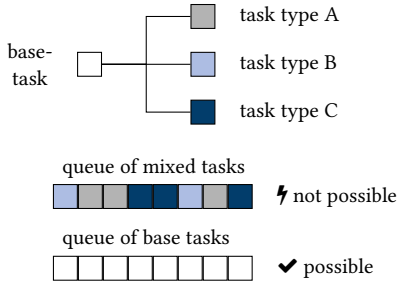

Figure 5: Classical queue involving virtual inheritance.

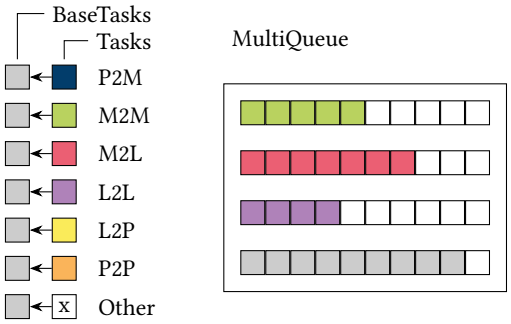

Figure 6: MultiQueue using three priority sub-queues and a backfill sub-queue.

All event listeners are registered in the `EventListenerContainer` via a variadic template pack. For the registration of an additional event listener a new template parameter is simply appended to this pack. The call of the dispatch method on the `EventListenerContainer` forwards the call to all registered event listeners. Afterwards, each event listener filters the triggered event using compile-time branching (substitution failure is not an error (SFINAE)). If the triggered event and the registered event match, the corresponding event handler is called. Otherwise, the dispatch call is empty. We can safely assume that modern C++ compilers will discard all empty functions during compilation. Again, we would like to emphasize that calling a dispatch method at run-time results in the direct call to the corresponding listeners, without any resolving overhead.

Additionally, the presented event dispatcher in Listing 1 looks comparable to a domain-specific-language, while being written in pure C++ via template metaprogramming.

2.2 Type-Driven Priority Scheduler

The second component of the task engine is the type-driven priority scheduler. That is used to execute tasks in a given order reflecting the critical path of the parallelization. Compared to conventional priority queues, we propose a lightweight implementation with less instructions required for inserting new tasks and the same amount required for dequeuing tasks. Additionally, our implementation is highly flexible and could also be used for other applications that require prioritization.

2.2.1 Task Prioritization. To maintain enough parallelism within the FMM it is required to execute the critical path prioritized. Tasks

initially created from the input provide enough work for all threads. However, due to the tree-based structure of the FMM, with every step upwards in the tree less tasks are created than consumed. Therefore, the upwards operations need to be prioritized, while new tasks outside the critical path are kept as backfill. This guarantees enough parallelism once the root node of the tree with only one task is reached. With every step downwards from the root node more tasks are created than consumed and enough parallelism is available again.

Within FMSolvr the FMM operators are reflected by different task types. Therefore we can infer the priority only from the task type. For the implementation of the queue this poses two problems. The first problem is how to implement the prioritizing of tasks. The second problem refers to the storage of tasks of different types in the same queue.

For the required prioritization there are two conventional approaches. The first is to utilize min-max heaps [29, 30] for the underlying queue container and the second one is to create a fixed number of sub-queues representing the different priorities. Min-max heaps cause logarithmic complexity for inserting and extraction. Having multiple sub-queues allows constant complexity for inserting and extraction.

The common solution for the second problem of different types of tasks involves virtual inheritance. Instead of working with the concrete type of a class a common virtual base class is used to store the tasks in a single queue. However, by using a common virtual base class it becomes too expensive to distinguish tasks depending on the concrete type. As described earlier, the type of the task is required for inferring the priority.

To solve both problems, we propose the multi-queue concept.

2.2.2 Multi-Queue. Figure 6 shows the multi-queue concept. The multi-queue consists of several sub-queues depending on the required number of priorities. These sub-queues store concrete tasks with their concrete types and are called *typed sub-queues* in the following sections. The ordering of the sub-queues in the definition represents the different priorities. Additionally, a backfill sub-queue using virtual inheritance for tasks without priority is available.

Due to the fact that the actual prioritization is quite specific to the algorithm, we offer an interface to easily change the collection of typed sub-queues and hence the prioritization. The available typed sub-queues as well as their ordering is expressed in a template parameter list in a type definition. Listing 3 shows the concrete definition for the multi-queue used in the FMM implementation. In this specific example, the types of the `M2MTasks`, `M2LTasks` and `L2LTasks` are prioritized in this order and all other tasks are enqueued in the backfill queue.

Listing 3: Definition of the multi-queue used in the FMM implementation. Here, we create typed sub-queues for M2M, M2L and L2L tasks with priorities in this order.

```
using multi_queue =
    MultiQueue<M2MTask, M2LTask, L2LTask>;
```

The implementation of the collection of sub-queues utilizes `std::tuple`. All sub-queues are elements of the tuple. Since we can iterate over the tuple it is possible to select the correct sub-queue during compile-time. This can be implemented by recursively iterating over the tuple and choosing the correct sub-queue by comparing the types using constant expressions [31]. If a C++14 compiler is available, the `std::get` method for retrieving tuple elements by type can be used. As any other queue our multi-queue needs to provide an enqueue and dequeue method. The enqueue method of the multi-queue uses a template parameter to deduce the concrete type of an inserted task. With this type the corresponding sub-queue can be selected and the task can be inserted correspondingly. The selection of the correct sub-queue depends only on the type of the task and can therefore be set up at compile-time. It is important to note that the enqueue method directly uses the sub-queue for inserting new tasks. This results in the same number of instructions required for inserting into a single queue.

The dequeuing from the multi-queue has the same constant complexity as dequeuing from an enumerated priority queue. The procedure is as follows:

(1) All sub-queues are checked for available tasks.
(2) Whenever a sub-queue is not empty a task is dequeued from this sub-queue.

Up until now we did not discuss the backfill sub-queue. An algorithm might have dozens of different task types, but only a few need to be prioritized. Using typed sub-queues for each of these task types would unnecessarily increase the number of sub-queues. That is why we added a backfill sub-queue that is based on virtual inheritance. All task types that are not explicity named in the multi-queue definition (see Listing 3) are non-prioritized and thus processed by the backfill sub-queue. Considering our use case FMSolvr, tasks of the types `P2M`, `L2P` and `P2P` are non-prioritized and thus added to the backfill sub-queue (cf. Listing 3 and Figure 6). This can be done with all tasks which are not critical and hence do not need to preserve their type.

3 USE CASE

3.1 Fast Multipole Method

The FMM is an algorithm for solving the N-body problem. The N-body problem is about the calculation of pairwise interactions of particles or objects resulting from electrostatic or gravitational potentials. The naïve computation of all these interactions in such N-body problems exhibits a computational complexity of $O(N^2)$, with N denoting the number of particles. With the FMM this can be reduced to linear complexity. For further details on the FMM we refer to [32, 33]. The idea behind the algorithm is as follows:

(1) The simulation space is subdivided in order to group the particles into a set of 8^d boxes.
(2) Particles in the near field interact directly.
(3) All other boxes interact in the far field via expansions.

The sequential flow of the FMM is shown in Figure 7. First, the particles with their positions $\mathbf{x}$ and their charges q are binned into a tree. The desired accuracy of the resulting energy $\mathbf{E}$, the forces $\mathbf{F}$ and the potential Φ is given by the user. According to this accuracy, the FMM specific parameters depth d of the tree, well-separateness criteria ws and multipole order p are determined to achieve minimal run-time.

Figure 7: The sequential workflow of the FMM. The computation is split into far field and near field computation. The input parameters are the coordinates x and the charges q of the particles. The accuracy of the algorithm can be influenced by the depth d of the tree, the well-separation criteria ws and the multipole order p. After the sequential execution of all operators (e.g. P2M, M2M, ...) the energy E, the forces F and the potential Φ are computed.

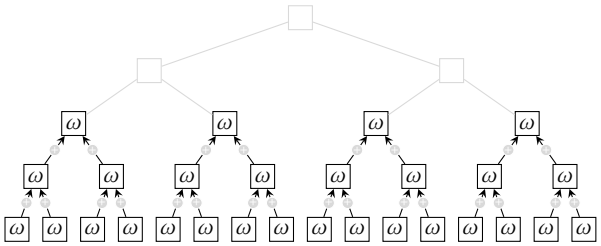

Figure 8: Multipole to Multipole operation

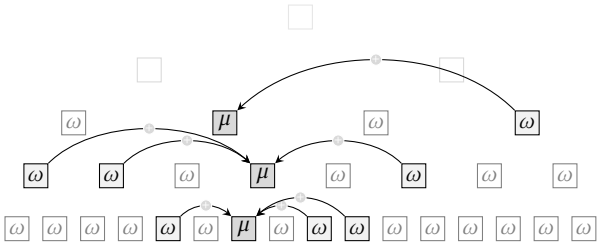

Figure 9: Multipole to Local operation

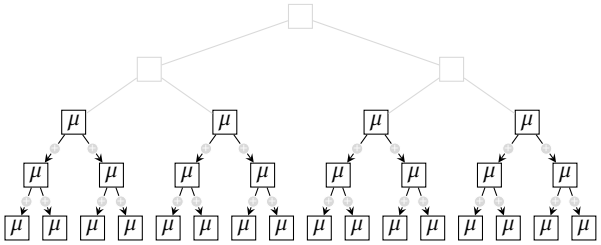

Figure 10: Local to Local operation

For simplicity we discuss binary trees only, although the implementation is done in 3D requiring an octree. For the computation of the far field, the FMM uses multipole expansions ω and local expansions μ. These expansions can be transformed with the help of three operators. These operators, namely M2M, M2L and L2L have a complexity of $O(p^3)$ or $O(p^4)$ depending on the implementation. The far field computation starts with the expansion of particles into multipoles, the so-called P2M operation. This operation

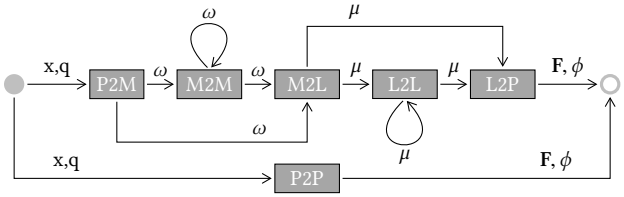

Figure 11: Dependencies between the operators of the FMM. The edges denote the data whereas the vertices denote the operators.

has a computational complexity of $O(Np^2)$. After the construction of all multipoles for each box on the lowest level, the multipoles are shifted towards the root node and are accumulated with the M2M operator (see Figure 8). These multipole expansions are then translated into local expansions with the multipole to local (M2L) operator (see Figure 9). After all local expansions on each level are computed, the local to local (L2L) operator is used to shift the local expansions downwards (see Figure 10). After the local expansions are shifted down to the lowest level, the far field potential and forces for the particles in each box are computed with the local to particle (L2P) operator in $O(Np^2)$ complexity. After the far field computation, the near field potential and forces are computed with a direct solver called particle to particle (P2P) in $O(M^2)$ complexity, where M denotes the number of particles interacting in the near field.

3.2 OpenMP FMSolvr

In the OpenMP version of FMSolvr each step of the FMM, namely P2M, M2M, M2L, L2L, L2P and P2P, is parallelized by means of `#pragma omp parallel for`. For M2M, M2L, L2L and P2P we apply `collapse(3)` in order to generate sufficient parallelism in these steps. In addition, we apply `schedule(runtime)` to define a scheduling policy, e.g. `static`, `guided` or `dynamic`, via the environment variable `OMP_SCHEDULE` at runtime. Since loop-parallel implementations of the FMM with OpenMP are already well studied, i. a. in [7], [5] and [20], we do not elaborate on yet another loop-parallel FMM implementation in this work.

3.3 Eventify FMSolvr

To dissolve the synchronizations induced by the sequential algorithm and preserved in the OpenMP version of FMSolvr, we need to model the data dependencies between the operations. In Eventify, we do so with event lists. Figure 11 shows the dependency graph of the FMM. In the graph, the edges denote the data and the vertices denote the operators performed on the data. The data-flow graph shows that the next operation on a certain expansion ω or μ can be started as soon as the expansion is computed. By sticking to those data dependencies only, we can get rid of the sequential workflow in our implementation completely and can still guarantee a correct result. For example, an M2L operation in one box can be started as soon as the multipole of this box is computed. This is independent from P2M operations in other boxes.

To provide a reasonable task granularity we combine tasks into the following groups: We combine eight M2M operations of all child

boxes towards the parent box into a single M2M task. For the M2L operation we group all 189 operations from a single box towards all interacting target boxes together. The L2L task encompasses eight shift operations from a parent box towards its eight child boxes. Finally, we group all L2P operations moving the expansion from the center of a box towards the particle positions into a single L2P task.

Beside the non-trivial inter-task dependencies shown in Figure 11 we have another bottleneck coming in unfavorably. Due to the tree structure and the dependencies between the operations, we are losing parallelism in the upward M2M operations. Eventually, this results in a single M2M operation at the root node. All following downward operations (L2L) depend on this single operation. Fortunately, this can be overlapped due to the availability of M2L operations on lower levels, which are independent from the downwards operation (L2L) on higher levels. Hence, it is very important to execute the algorithm along the critical path to maintain enough parallelism. This means that we should compute all available M2M tasks towards the root node, before computing further M2L tasks on lower levels. After this operation, enough tasks are generated on the lower levels (e.g. M2L) to avoid starvation during the upwards phase.

4 PERFORMANCE ANALYSIS

In this section, we analyze the performance of the parallel FMM-implementations described in Section 3.

4.1 Hardware

All measurements were performed on a 4-socket system equipped with four Intel Xeon E7-4830 v4 CPUs (Broadwell). Each CPU provides 14 physical two-way SMT-cores. Hence, the full compute node covers 56 physical cores and 112 logical cores, respectively.

4.2 Input Data Set

Our input data set is an amorphous silicon dioxide (SiO_2) melt consisting of 103680 particles. As common in biochemistry, it exhibits a relatively homogeneous charge distribution. As a realistic input data set, it was applied for the comparison of several algorithms that solve the N-body problem in [2]. Since we aim at the evaluation of Eventify for synchronization-critical input data sets with low computational effort, the accuracy settings are low, but realistic. Accordingly, we set the multipole order $p = 4$ and the tree depth $d = 5$.

4.3 Measurement Method

To obtain reliable test results each measuring point represents the mean value of ten equal runtime measurements. For each experiment, the runtime was measured for a number of threads $\#Threads = 1, \ldots, 56$, since this covers measurements for the single-threaded version as well as the utilization of all physical cores.

Intel's Turbo Boost was disabled during all measurements. This hardware feature adjusts core clock frequencies depending on work load and the number of running threads. Since the clock frequency for the single-threaded implementation is automatically increased, this distorts scaling plots.

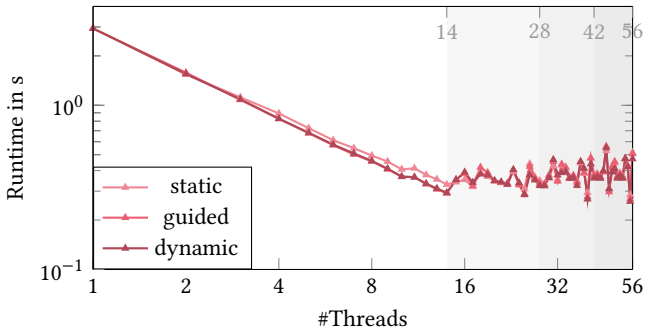

Figure 12: Runtime of FMSolvr with OpenMP scheduling policies static, guided and dynamic.

For Eventify, we make use of the environment variable LD_PRELOAD to load jemalloc, which is *a general purpose malloc implementation that emphasizes fragmentation avoidance and scalable concurrency support* [34]. In comparison to the default glibc allocator, jemalloc improves the performance of the large amount of concurrent allocations required by task creation. For OpenMP, we do not employ jemalloc since it does not lead to any performance gain.

For efficiency plots, we apply the definition of parallel efficiency provided by Rauber et al. [35]: $E_p(n) = \frac{T^*(n)}{p \cdot T_p(n)}$, where $T^*(n)$ is the sequential execution time of the best sequential algorithm and $T_p(n)$ is the parallel execution time on p processors. However, in practice it is not feasible to determine the best sequential algorithm and its implementation for the FMM since there are diverse options to implement the workflow and the operators of the method. As a feasible solution, we execute OpenMP FMSolvr with a single thread to determine the sequential runtime $T^*(n)$. This is reasonable since OpenMP FMSolvr does not contain any parallelization overhead when executed with a single thread and furthermore provides the fastest sequential runtime of the FMM-implementations studied in this work.

4.4 OpenMP FMSolvr: Scheduling Policies

Figure 12 shows the runtime of FMSolvr with the OpenMP scheduling policies [4] static, guided and dynamic. For $\#Threads \leq 14$ the policies guided and dynamic provide the smallest runtimes. The smallest runtime of OpenMP FMSolvr is 260 ms and is reached with the scheduling policy dynamic at $\#Threads = 55$. However, it is hard to determine the best scheduling policy for $\#Threads > 14$ since the runtime exhibits heavy variations. One reason for these variations is thread migration between cores and thus varying memory access times. We observed thread migration for $\#Threads > 14$ even though we pinned threads to cores with the thread affinity settings `OMP_PROC_BIND=close` and `OMP_PLACES=cores`. This may be due to the fact that *the determination of whether the affinity request can be fulfilled is implementation defined* according to the OpenMP standard [4]. Nevertheless, the reasons for the runtime variations need further research.

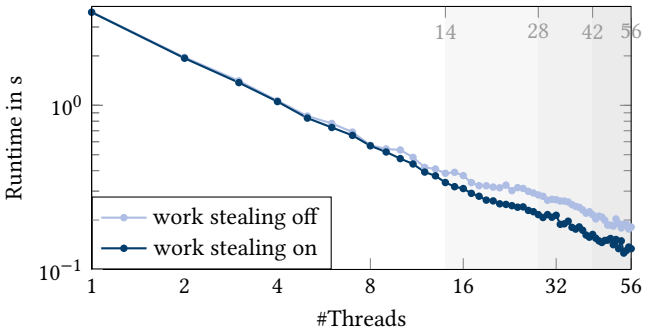

Figure 13: Runtime of Eventify FMSolvr with and without work stealing.

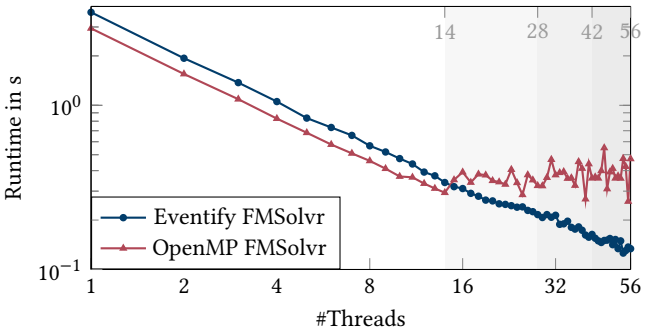

Figure 14: Runtime comparison of Eventify FMSolvr (work stealing on) and OpenMP FMSolvr (dynamic) with the respective optimal scheduling policy.

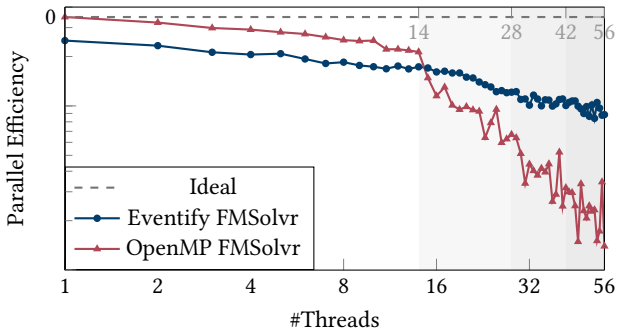

Figure 15: Parallel efficiency of Eventify FMSolvr (work stealing on) and OpenMP FMSolvr (dynamic) with the respective optimal scheduling policy.

4.5 Eventify FMSolvr: Scheduling Policies

The plot in Figure 13 provides a runtime comparison for Eventify FMSolvr with and without work stealing. As the plot shows, the version with work stealing is clearly beneficial compared to the version without work stealing. The smallest runtime is 126 ms and accordingly reached with work stealing at $\#Threads = 53$. In comparison to OpenMP FMSolvr we do not apply any thread affinity settings for Eventify FMSolvr since the focus of this work is only on the essential concepts of Eventify. Detailed studies on thread affinity and data locality for FMSolvr are provided in [36].

4.6 Eventify vs. OpenMP

Figure 14 and Figure 15 provide a comparison of the OpenMP and Eventify version that performed best in the previous experiments. Expectedly, OpenMP Fmsolvr is beneficial for low amounts of threads $\#Threads \leq 14$. First, this is because OpenMP loop-parallelism introduces nearly no parallelization overhead since it does neither create nor manage tasks. Second, this is because OpenMP sticks to our thread affinity settings for $\#Threads \leq 14$ and thus can benefit from improved data locality. However, for $\#Threads > 14$ Eventify FMSolvr is beneficial in terms of runtime and scalability. The overall smallest runtime is 126 ms and is reached by Eventify FMSolvr at $\#Threads = 53$. In comparison to the smallest runtime reached by OpenMP FMSolvr this is a runtime improvement of 52 %.

5 CONCLUSION AND FUTURE WORK

We presented a task engine specialized for synchronization-critical applications. This task engine encompasses two main components, a type-driven priority scheduler as well as a static event dispatcher.

The type-driven priority scheduler fulfills the requirement of prioritizing tasks along the critical path while preserving important type information. This increases the robustness tremendously without imposing new overheads or performance bottlenecks usually introduced by priority queues. On the contrary, we even improved the performance due to compile-time resolution of the correct subqueue via template metaprogramming. This leads to the situation where inserting into our multi-queue is as lightweight as inserting into a single queue.

The static event dispatch framework is highly configurable at compile-time, while providing the best possible performance of dispatching due to the compile-time branching. The definition of a concrete event dispatcher resembles a domain specific language, although it is pure C++ code. This opens the possibility for other application developers to adapt and apply this concept for their needs.

The results prove the readiness of modern software development approaches to sufficiently strong scale even synchronization-critical HPC simulations. This will be the key for utilizing supercomputers in the upcoming exascale era. Nevertheless, there is still potential for optimization.

The first bottleneck is caused by Non-uniform memory access (NUMA). NUMA effects occur when data is located or transferred between memory nodes of different CPUs. Since the latency difference between local and remote memory accesses leads to performance loss, we have to develop NUMA-aware data distribution, thread pinning and work-stealing policies. First results of our NUMA-aware FMM implementation can be found in [36].

The second bottleneck is caused by the use of `std::mutex` as locking mechanism. The implementation of these locks causes performance loss due to cache coherency. Cache coherency becomes even more expensive when crossing NUMA-borders. A strategy to reduce the pressure on the cache coherence protocol is the usage of

scalable locks like the MCS-lock [37]. Additionally, it is conceivable to implement lock-free strategies for the multi-queue.

The third bottleneck are the fine-grained allocations due to task creation and deletion. Even with the use of *jemalloc*, this bottleneck poses still a considerable overhead. Nevertheless, this can be solved by reusing task objects from a memory pool.

In conclusion, this paper opens a perspective to overcome future quantitative changes in hardware with the help of good software design and with an implementation with a high level of abstraction. From our point of view, the importance of proper software engineering in high performance computing will increase tremendously.

6 ACKNOWLEDGMENTS

This project was partly supported by the DFG priority programme Software for Exascale Computing (SPP 1648).

REFERENCES

[1] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1-2, pp. 19 – 25, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352711015000059

[2] A. Arnold, F. Fahrenberger, C. Holm, O. Lenz, M. Bolten, H. Dachsel, R. Halver, I. Kabadshow, F. Gähler, F. Heber, J. Iseringhausen, M. Hofmann, M. Pippig, D. Potts, and G. Sutmann, "Comparison of scalable fast methods for long-range interactions," *Phys. Rev. E*, vol. 88, p. 063308, Dec 2013. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.88.063308

[3] I. Kabadshow, H. Dachsel, C. Kutzner, and T. Ullmann, "GROMEX – Unified Long-range Electrostatics and Flexible Ionization," http://www.mpibpc.mpg.de/15304826/inSiDE_autumn2013.pdf, 2013 (accessed April 27, 2017).

[4] OpenMP Architecture Review Board, "OpenMP application programming interface version 5.0," 2018. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[5] P. Atkinson and S. McIntosh-Smith, "On the Performance of Parallel Tasking Runtimes for an Irregular Fast Multipole Method Applicatio," in *Scaling OpenMP for Exascale Performance and Portability*, B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, and M. S. Müller, Eds. Cham: Springer International Publishing, 2017, pp. 92–106.

[6] K. Taura, J. Nakashima, R. Yokota, and N. Maruyama, "A Task Parallel Implementation of Fast Multipole Methods," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov 2012, pp. 617–625.

[7] E. Agullo, O. Aumage, B. Bramas, O. Coulaud, and S. Pitoiset, "Bridging the Gap Between OpenMP and Task-Based Runtime Systems for the Fast Multipole Method," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2794–2807, Oct 2017.

[8] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1177/1094342007078442

[9] L. V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. New York, NY, USA: ACM, 1993, pp. 91–108. [Online]. Available: http://doi.acm.org/10.1145/165854.165874

[10] C. Pheatt, "Intel Threading Building Blocks," *J. Comput. Sci. Coll.*, vol. 23, no. 4, pp. 298–298, Apr. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1352079.1352134

[11] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A Task Based Programming Model in a Global Address Space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: ACM, 2014, pp. 6:1–6:11. [Online]. Available: http://doi.acm.org/10.1145/2676870.2676883

[12] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[13] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011. [Online]. Available: http://hal.inria.fr/inria-00550877

[14] R. Bagrodia, R. Meyer, M. Takai, Yu-An Chen, Xiang Zeng, J. Martin, and Ha Yoon Song, "Parsec: a parallel simulation environment for complex systems," *Computer*, vol. 31, no. 10, pp. 77–85, Oct 1998.

[15] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-oriented Approach to Non-uniform Cluster Computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094852

[16] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos, "A taxonomy of task-based parallel programming technologies for high-performance computing," *The Journal of Supercomputing*, vol. 74, no. 4, pp. 1422–1434, Apr 2018. [Online]. Available: https://doi.org/10.1007/s11227-018-2238-4

[17] B. Zhang, "Asynchronous Task Scheduling of the Fast Multipole Method Using Various Runtime Systems," in *Proceedings of the 2014 Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, ser. DFM '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 9–16. [Online]. Available: https://doi.org/10.1109/DFM.2014.14

[18] Z. Khatami, H. Kaiser, P. Grubel, A. Serio, and J. Ramanujam, "A Massively Parallel Distributed N-body Application Implemented with HPX," in *Proceedings of the 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 57–64. [Online]. Available: https://doi.org/10.1109/ScalA.2016.12

[19] H. C. Edwards and C. R. Trott, "Kokkos: Enabling Performance Portability Across Manycore Architectures," in *2013 Extreme Scaling Workshop (xsw 2013)*, Aug 2013, pp. 18–24.

[20] R. Yokota and L. A. Barba, "A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems," *The International Journal of High Performance Computing Applications*, vol. 26, no. 4, pp. 337–346, 2012. [Online]. Available: https://doi.org/10.1177/1094342011429952

[21] M. Abduljabbar, M. Al Farhan, R. Yokota, and D. Keyes, "Performance Evaluation of Computation and Communication Kernels of the Fast Multipole Method on Intel Manycore Architecture," in *Euro-Par 2017: Parallel Processing*, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds. Cham: Springer International Publishing, 2017, pp. 553–564.

[22] H. Ltaief and R. Yokota, "Data-Driven Execution of Fast Multipole Methods," *CoRR*, vol. abs/1203.0889, 2012. [Online]. Available: http://arxiv.org/abs/1203.0889

[23] A. YarKhan, J. Kurzak, and J. Dongarra, "QUARK Users' Guide: QUeueing And Runtime for Kernels," 2011.

[24] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-Based FMM for Multicore Architectures," *SIAM Journal on Scientific Computing*, vol. 36, no. 1, pp. C66–C93, 2014. [Online]. Available: https://doi.org/10.1137/130915662

[25] E. Agullo, B. Bramas, O. Coulaud, M. Khannouz, and L. Stanisic, "Task-based fast multipole method for clusters of multicore processors," Inria Bordeaux Sud-Ouest, Research Report RR-8970, Mar. 2017. [Online]. Available: https://hal.inria.fr/hal-01387482

[26] L. Hyafil and R. Rivest, *Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems*, ser. Laboratoire de Recherche: Rapport de recherche. IRIA, 1973.

[27] N. Goodspeed, "A proposal to add coroutines to the C++ standard library (Revision 1)," 2014.

[28] (2017) Intel TBB Data Flow and Dependence Graphs. [Online]. Available: https://software.intel.com/en-us/node/517340

[29] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte, "Min-max heaps and generalized priority queues," *Communications of the ACM*, vol. 29, no. 10, pp. 996–1000, 1986.

[30] N. M. Josuttis, *The C++ standard library: a tutorial and reference*. Addison-Wesley, 2012.

[31] (2011) Constant Expression. [Online]. Available: http://en.cppreference.com/w/cpp/language/constant_expression

[32] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of computational physics*, vol. 73, no. 2, pp. 325–348, 1987.

[33] I. Kabadshow, *Periodic boundary conditions and the error-controlled fast multipole method*. Forschungszentrum Jülich, 2012, vol. 11.

[34] J. Evans, "jemalloc memory allocator," http://jemalloc.net/, 2017 (accessed April 11, 2017).

[35] T. Rauber and G. Rünger, *Parallel Programming for Multicore and Cluster Systems*. Springer Berlin Heidelberg, 2010.

[36] L. Morgenstern, "A NUMA-Aware Task-Based Load-Balancing Scheme for the Fast Multipole Method," Master Thesis, TU Chemnitz, 2017.

[37] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2011.