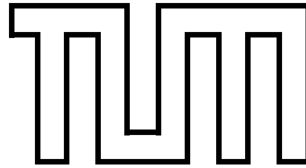



FAKULTÄT FÜR INFORMATIK

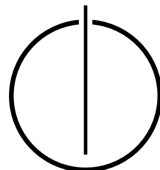
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN


Guided Research

# Implementation and Evaluation of Task-based Approaches for Molecular Dynamics Simulations

# Entwicklung und Evaluation Task-basierter Ansätze für Molekulardynamik Simulationen


| | |
|---|---|
| Author: | Fabio Alexander Gratl |
| Supervisor: | Univ.-Prof. Dr. Hans-Joachim Bungartz |
| Advisor: | Nikola Tchipev, M.Sc. |
| Date: | April 24, 2017 |

# Contents


**1 Introduction** **2**

**2 Quicksched** **2**
2.1 Overview . . . 2
2.2 Components . . . 3
2.2.1 Data Structures . . . 3
2.2.2 Important Functions . . . 5
2.3 Benchmark Performance . . . 6

**3 Molecular Dynamics via ls1-mardyn** **7**
3.1 Overview . . . 7
3.2 Linked-Cell Algorithm . . . 8
3.2.1 Force Calculation . . . 8
3.2.2 Segmentation of the Domain . . . 9
3.2.3 Established Approach *c08* . . . 10

**4 Task-based Approaches for Force Calculation** **11**
4.1 *qui8* . . . 11
4.2 *taskc08* . . . 11
4.3 *qui1* . . . 11
4.4 *qui_adj_n* . . . 12
4.5 Task Weighting . . . 13

**5 Results** **13**
5.1 Experimental Setup . . . 13
5.2 Strong-scaling . . . 14
5.3 Scheduling Timings . . . 16
5.4 Function Runtime Cost Distribution . . . 20

**6 Conclusions and Outlook** **21**

**Abstract**

*ls1-mardyn* is a molecular dynamics code specialized in simulations of very large scale of up to $4.125 \times 10^{12}$ particles [EHB+13]. Although the established approach for its force calculation, which is using coloring and *OpenMP* loop parallelization, proves to be very efficient, it still leaves potential for optimization because of its use of barriers. A promising method to replace these barriers are task-based scheduling concepts due to their higher flexibility and finer granularity. As such a library, the highly dynamic *Quicksched* code [GCS16] has here been used for implementing task-based shared-memory parallelization approaches for the force calculations in *ls1-mardyn.* Their performance has been tested on an *Intel Xeon Phi Knights Corner* accelerator and is compared to established approaches that use domain coloring and *OpenMP* loop parallelization. When directly comparing the results it is hard to beat the efficiency of the highly optimized established approaches. Yet, through an in-depth analysis of the scheduling, possible performance potentials can be identified and shown to be exploited by the task-based approaches. However, task-based approaches introduce new challenges of their own. These are analyzed and discussed and possible solutions are proposed.

# 1 Introduction

*Molecular Dynamics Simulations* are extremely useful tools in for example the fields of molecular biology [KDFB04], thermodynamics [DES+11], chemistry [vGB90], or material studies [Bin95] for analyzing and predicting properties and processes on a molecular level. Since molecular dynamics simulations are $N$-Body problems, which means that in theory each particle interacts with every other particle, the majority of their runtime is spent calculating these interactions, thus rendering them highly computational intensive. Fortunately, these interaction calculations are independent per time step and can therefore be parallelized. However, in order not to overwrite results, no two calculations involving the same particle can happen at the same time, which would introduce a *race condition*. This is why parallelization schemes or algorithms are needed to ensure maximum parallelism, while at the same time ruling out any race conditions. For this, two possible approaches are either classical loop parallelization via *OpenMP* or task-based approaches, which describe the program as a set of tasks, possibly with dependencies among them and then efficiently schedule them over the available computing resources.
Although comparatively easy to implement, the drawback of *OpenMP* loop parallelization is that to avoid race conditions, the domain needs to be partitioned into so called colors. Work within one color can be executed completely in parallel, the colors themselves however need to be executed sequentially. This introduces barriers at which some threads will be waiting, rendering this kind of approaches not optimal. A task-based approach can avoid these idle times by dynamically distributing small tasks with few dependencies over all available threads and reassigning them when idle threads are detected.
The goal of this paper is to introduce task-based parallelization approaches for the force calculation in the *large systems 1: molecular dynamics (ls1-mardyn)* code by using the *Quicksched* library. The approaches will be analyzed and compared to the established approach that uses *OpenMP* loop parallelization [TWG+15] and another version using *OpenMP 4.5*'s explicit tasks. We are aiming to determine the potential of such approaches and their pitfalls to decide on which aspects future work has to focus.

# 2 Quicksched

## 2.1 Overview

*Quicksched*[1] is a library for tasks-based parallelism for shared-memory applications.
It was developed by Pedro Gonnet[2,3], Aidan B.G. Chalk[2], and Matthieu Schaller[4]. They also work on *SPH With Inter-dependent Fine-grained Tasking (SWIFT)*[5], which is an open-source astrophysics simulation by the Institute for Computational Cosmology

[1] https://gitlab.cosma.dur.ac.uk/swift/quicksched
[2] School of Engineering and Computing Sciences, Durham University, United Kingdom.
[3] Google Switzerland GmbH, Zürich, Switzerland.
[4] Institute for Computational Cosmology, Durham University, United Kingdom.
[5] http://icc.dur.ac.uk/swift

(ICC)[6] and the Institute of Advanced Research Computing (IARC)[7] at the University of Durham.
The library is written in *C* and with roughly 3000 lines of code very compact. For the underlying threading mechanism either *OpenMP* or *POSIX Threads (Pthreads)* can be used. When compiled with support of both of them, this decision can even be made at runtime. For the experiments in this paper the *OpenMP* variant was used. *Quicksched* also supports tasks being dependent on or mutually exclusive to other tasks and tasks being dependent on a shared resource. Especially the possibility to create tasks that must not run concurrently, but whose order of execution is irrelevant, makes this library more flexible than for example tasks provided by the current *OpenMP* version 4.5[8].
Additionally, resources can be organized in a hierarchy to support multilevel locks.
For further details about *Quicksched* the interested reader can consult [GCS16].

## 2.2 Components

To establish a better understanding of the inner workings of *Quicksched* its core components, meaning the most important structs and functions, shall be discussed.

### 2.2.1 Data Structures

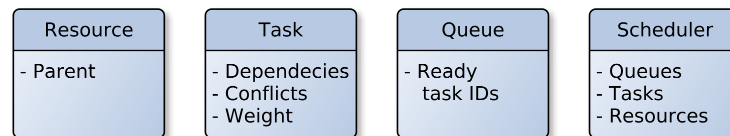



Figure 1: Main data structures of *Quicksched* and their essential informations.

**Resources**
Resources are used to grant exclusive access to certain objects. Therefore they have a `lock` field, indicating whether or not the resource is locked.
Since resources also support a hierarchical structure they also hold information on how to identify their parent resource.

**Tasks**
A task holds information on its dependencies. This is realized by every task knowing how many tasks it is dependent on and other tasks knowing which task they unlock. Similarly, each task has an array of pointers to resources it intends to lock. The task can only be executed when all dependencies are resolved and all resource locks could be obtained. This results in the tasks being able to be represented in a directed acyclic graph.
What a task actually does is determined by an integer `type` and the `runner()` function

[6] http://icc.dur.ac.uk
[7] https://www.dur.ac.uk/iarc
[8] http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

mentioned in paragraph runner in Section 2.2.2.
Furthermore, a task can be assigned a `cost`. From these costs the `weight` of a task is calculated taking dependencies into account:

$$weight_i = cost_i + \max_{j \in \text{unlocks}_i} \{weight_j\} \tag{1}$$

By adding the weight of the heaviest dependent task, this models the costs along the application's critical path of execution. Therefore, these weights are used as further information for optimizing the scheduling process, so that more critical tasks can be started earlier.

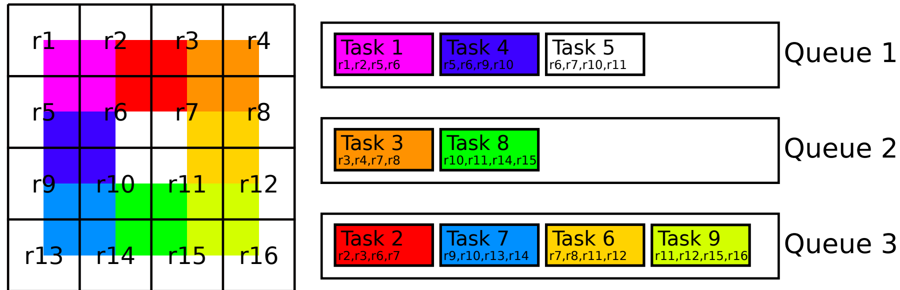



Figure 2: Left: Domain divided into resources. Colors indicate tasks and which resources they require.
Right: Tasks distributed over queues.

### Queues

Every thread has a queue that is filled with task IDs. Only so called *ready tasks* are put into queues. These are tasks whose task dependencies are resolved, but their required resources are not necessarily available yet. This is depicted in figure 2 where on the left side, is a set of resources $r$ that are required by tasks, which are represented as colored squares. The resources touched by the colored square are the required ones, like for example the red task requiring the resources $r2, r3, r6$ and $r7$. These tasks, since they depend on no other tasks, are then immediately distributed over the queues 1 to 3 depicted on the right side of the figure.
An important aspect is the ordering of tasks inside of a queue. Optimally, they would be sorted strictly in a descending order according to their weights to primarily follow the application's critical path of execution. However, for $n$ tasks this would require at least $O(n)$ operations for insertion and deletion, which are crucial for maintaining the structure. For *Quicksched* it was decided to organize the queue as a *max-heap*. Thereby, every $k$-th element's costs are larger than the $2k+1$-st and $2k+2$-nd. As a consequence, the costs for maintaining the structure are reduced to $O(\log n)$ at the expense of the strict ordering, since there is no efficient way of traversing the heap in exactly descending order. It can only be guaranteed that the $k$-th task has larger weight than at least $\lfloor \frac{n}{k} \rfloor - 1$ other tasks. This bound is not very good in theory, but according to [GCS16], it "[...] turns out to be quite sufficient in practice".

#### Scheduler

Tasks and resources need to be registered to the `scheduler`. Queues are created upon the instantiation of the scheduler, which makes it the user's primary point of interaction with *Quicksched*.
The general idea is to describe to the scheduler what to do by filling it with tasks, resources, and dependencies and then letting it execute everything by invoking a single function which is described in Section 2.2.2.

### 2.2.2 Important Functions

#### qsched_run()

`qsched_run()` is the aforementioned function to invoke the scheduler and let it execute everything. As can be seen in Figure 3 line 4 it starts an OpenMP region with the given number of threads. Every thread then goes into a loop fetching tasks until there are none left. The tasks are executed in line 9 through a `runner` function, that needs to be passed as an argument to `qsched_run()` (see line 3).

```
1  void qsched_run(qsched *s,
2                 int numThreads,
3                 void (*qsched_funtype)(int,void*) runner){
4  #pragma omp parallel num_threads( numThreads )
5      {
6          int qid = omp_get_thread_num() % s->nr_queues;
7          struct task *t;
8          while ((t = qsched_gettask(s, qid)) != NULL) {
9              runner(t->type, t->data);
10         }
11     }
12 }
```

Figure 3: Short excerpt of qsched_run()

#### runner()

The `runner()` function is not part of the library but needs to be implemented by the user, since it defines which tasks should exist and what they do. For compatibility, the signature obviously needs to be adhered to. The function takes an integer `type` used to indicate the type of the task and a `void` pointer `data` to pass arbitrary data. This second argument can be thought of as a way of passing arbitrary function arguments to the task itself.
In the `runner()` function itself typically a `switch` case statement is used to identify the current task according to the passed value of `type`, as can be seen in Figure 4.

```
void runner(int type, void *data){
    switch(type)
        case taskType1:
            // do stuff
        case taskType2:
            // do other stuff
}
```

Figure 4: General form of a typical runner() function

#### qsched_gettask()

The `qsched_gettask()` function is responsible for fetching tasks from the thread's queue. This is done through `queue_get()` in line 3 of Figure 5, which loops over the max-heap and tries to find the most expensive task for which all resource locks can be acquired. If no task could be fetched, a randomly chosen other queue is queried. This process is called *work-stealing*. For example in Figure 2 the thread working on queue 2 might finish first, and then tries to steal a task from the other queues since they could still have unprocessed tasks. It should be noted that choosing the queue randomly can lead to very bad performance on unbalanced queues and larger numbers of queues as will be seen in Section 5.3.
Should there be no task fetched, `qsched_gettask()` loops in a busy waiting manner through the `while` loop in line 2 until either a task is found or there are no waiting tasks left.

```
struct task *qsched_gettask(struct qsched *s, int qid) {
    while (s->waiting) {
        tid = queue_get(&s->queues[qid], s, 1);
        if (tid < 0)
            // steal from random other queue
        else
            return &s->tasks[tid];
    }
    return NULL;
}
```

Figure 5: Short excerpt of qsched_gettask()

## 2.3 Benchmark Performance

Bundled with the source code of the library there come two benchmarks, a tiled QR decomposition and a *Barnes-Hut* simulation [GCS16]. Since *ls1-mardyn* is a molecular dynamics simulation, the *Barnes-Hut* benchmark fits our scenario best because both concern $n$-body interactions.
A *Barnes-Hut* simulation approximates the solution of a $n$-body problem, which means computing all pairwise interactions of $n$ particles in a given domain. This naively requires $O(n^2)$ operations. By recursively subdividing the domain through an octree

decomposition and only looking at aggregates of distant cells instead of all, the number of computations can be reduced to be in $O(n \log n)$. A more in depth explanation and details of implementation can be found in [GCS16]. The benchmark was performed on a *Intel Xeon Phi* 5110P (*Knights Corner*) with 60 Cores at 1.05 GHz at the *SuperMIC* cluster at the LRZ[9]. The simulation contained 1.000.000 particles. Every marker in Figure 6 represents a measurement, the connecting lines are for increased visibility of trends.
Up until 60 threads a linear speedup can be observed with a parallel efficiency of more than 90%. As soon as the 60 thread mark is crossed and hyperthreading is used only marginal performance gains are achieved. This however is expected since all tasks use the same components of the CPU, namely the *floating-point unit* (FPU). Since the here used *Knights Corner* generation only has one FPU per core there are no further computing resources available for more than one thread.

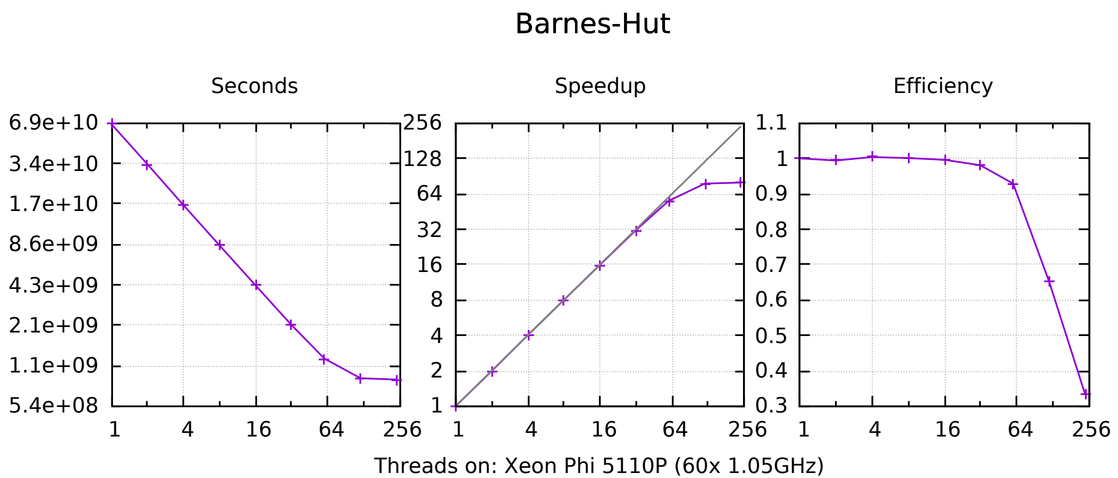



Figure 6: Quicksched Barnes-Hut benchmark with 1.000.000 particles on *SuperMIC*

# 3 Molecular Dynamics via ls1-mardyn

## 3.1 Overview

*Large systems 1: molecular dynamics (ls1-mardyn)*[10] is a molecular simulation specialized for very large numbers of rigid, multi-centered molecules of up to $4.125 \times 10^{12}$ [EHB+13]. The code is optimized for massively parallel execution on supercomputers, written in *C++*, and publicly available as free software under a BSD license [NBB+14]. It is currently developed by the High Performance Computing Center Stuttgart (HLRS)[11] at the University of Stuttgart, the Laboratory for Engineering Thermodynamics (LTD)[12] at

[9] https://www.lrz.de/services/compute/supermuc/supermic
[10] http://www.ls1-mardyn.de
[11] http://www.hlrs.de/home
[12] http://thermo.mv.uni-kl.de

the University of Kaiserslautern, the chair for Scientific Computing in Computer Science (SCCS)[13] at Technische Universität München and the chair for Thermodynamics and Energy Technology (ThEt)[14] at the University of Paderborn.

## 3.2 Linked-Cell Algorithm

### 3.2.1 Force Calculation

In every time step the interactions between all pairs of particles need to be calculated. These interactions are modeled by the *Lennard-Jones Potential*, which can be used to approximate forces between uncharged atoms or molecules [LJ31]. Particles that are located close to each other experience a strong repulsive force while further separated particles experience a weak attractive force. This is illustrated in the force profile depicted in Figure 7.

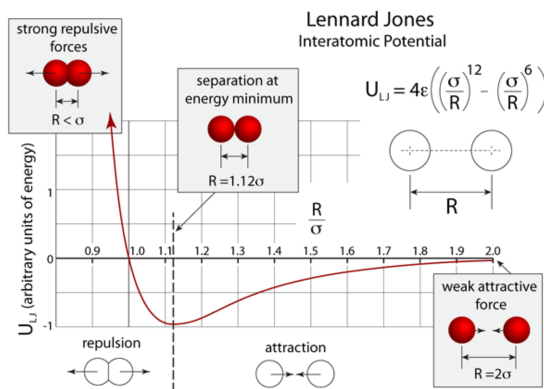



Figure 7: Profile of the Lennard-Jones Potential
Source: http://atomsinmotion.com/book/chapter5/md (accessed: 20.03.2017)

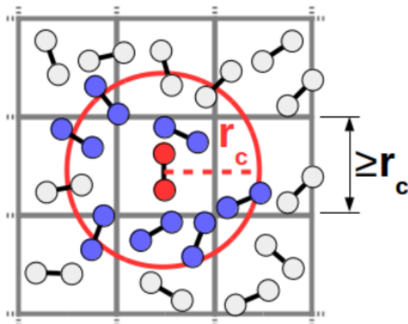



Figure 8: Cutoff radius around a molecule (red circle). Only blue molecules are considered for force calculation.
Source: [TWG+15]

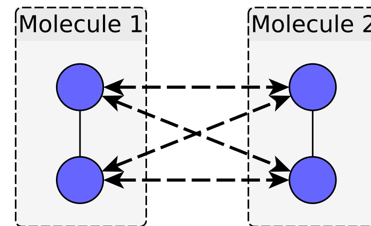



Figure 9: Interactions between multi-centered molecules. Blue discs represent Lennard-Jones centers, dashed arrows interactions.

[13] https://www5.in.tum.de
[14] http://thet.uni-paderborn.de

It can also be seen that with increasing distance between the particles the force converges to zero. Therefore it is reasonable to introduce a cutoff radius beyond which further particles are ignored as can be seen in Figure 8. As a result the number of computations is reduced significantly to $O(n)$. This technique is an essential part of the *Linked Cell* Algorithm. [GKZ07] [RBM+96]
Furthermore, molecules can possess multiple so called Lennard-Jones centers, which are the point in the molecule from where the Lennard-Jones potential is calculated. The interactions between molecules with two centers are depicted in Figure 9. Every center of the first molecule interacts with all centers of the second. For $m$ Lennard-Jones centers per molecule this raises the number of calculations to $O(n^m)$. Since *ls1-mardyn* is only applicable for rigid molecules, the Lennard-Jones centers within one molecule stay in the same place relative to the molecule and do not interact with each other.
Newton's third law of motion states that for the sum of all forces between two objects there exists an equal force in the opposite direction. Thereby, we know that the force exerted on particle $i$ through particle $j$ is the negative of the force on particle $j$ induced by particle $i$:

$$F_{i,j} = -F_{j,i} \tag{2}$$

This relation can be used to reduce the number of calculations by roughly a factor of $\frac{1}{2}$, and is hence a very important optimization. However, it introduces potential race conditions as two molecules in neighboring cells can be updated simultaneously.

#### 3.2.2 Segmentation of the Domain

To be able to parallelize a simulation it is necessary to discretize it and spread the different parts over the available processors. A very straightforward approach is to decompose the domain in blocks or cells of uniform size.
Cells at the edge of the domain which do not have all eight neighbors are called halo cells and are depicted blue in Figure 10 and 11. For these cells it is not relevant to calculate all interactions since they act as boundary of the domain and thereby their state is determined in a different manner, which is not of interest for this paper. Only interactions with non-halo cells are relevant.

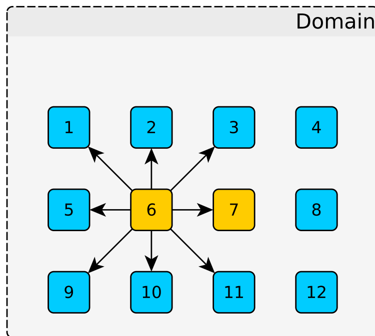



Figure 10: Dependencies of a single cell in 2D.

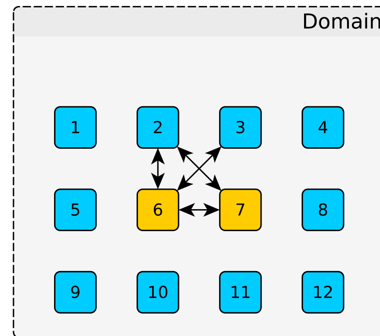



Figure 11: Compact solving incorporating Newton's third law of motion.

The edge length of cells should be at least as long as the cutoff radius introduced in Section 3.2.1. Thereby, a particle in one cell can at maximum interact with particles in all directly surrounding cells like in Figure 10. However, when using this pattern, the calculation step for each cell would block access to nine cells since overlapping computations could overwrite each other. Therefore, one can change the pattern for cell 6 to a more compact form as can be seen in Figure 11 and impose it on cells 5, 6, 7, 9, 10, 11 to cover the whole domain. This pattern only blocks four cells, and also covers interactions in both directions by using the optimization through Newton's third law of motion as seen in Equation 2. Through this, the number of computations can be reduced by a factor of $\frac{1}{2}$.
In a three-dimensional domain, the naive approach blocks 27 while the compact blocks only eight cells, which is visualized in Figure 12.

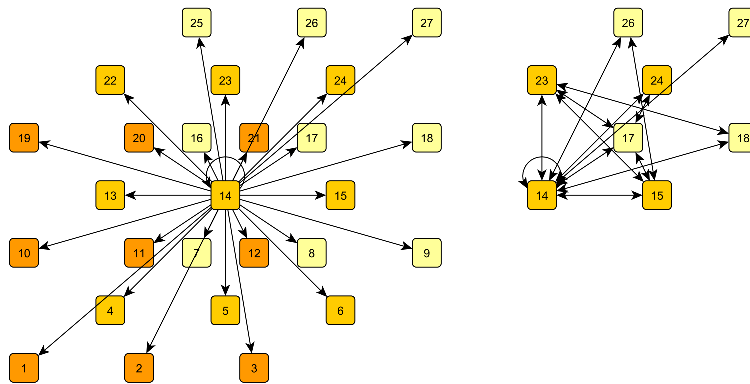



Figure 12: Left: dependencies of a single cell (cell 14) in 3D.
Right: compact pattern for cell 14 in 3D.

### 3.2.3 Established Approach c08

In [TWG+15] an approach was developed exploiting the schemes described in 3.2.2, and using *OpenMP* loop parallelization. Here the scheduling kind *dynamic*[15] is used with a chunk size of one. Since no two calculations on one cell must happen in parallel the calculations were colored in a checkerboard style manner as can be seen in Figure 13. Since this approach uses eight colors in 3D it is called *c08*.

[15] https://software.intel.com/en-us/articles/openmp-loop-scheduling

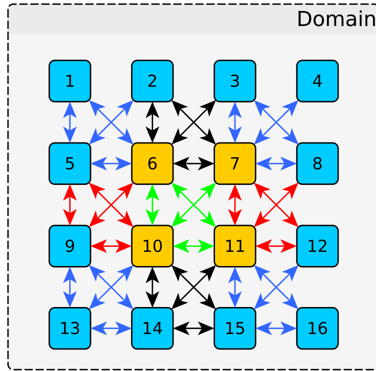



Figure 13: Coloring scheme *c08* for OpenMP loop parallelization.

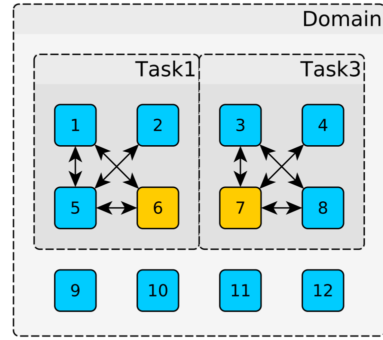



Figure 14: Scheme *qui8* for task-based parallelization.

# 4 Task-based Approaches for Force Calculation

## 4.1 qui8

The first tested scheme is based on the *c08* approach. Every closed color region will serve as one task. This results in tasks like in Figure 14. For a small domain like this six tasks would be generated. The tasks overlap, for example task number two would cover cells 2, 3, 6 and 7. This means it cannot be executed in parallel to other tasks using any of these cells. Nevertheless, the order in which the tasks are executed is irrelevant. Therefore, in *Quicksched* terms, every cell is a resource that needs to be locked.

## 4.2 taskc08

For comparison a scheme is used that is not using *Quicksched* but the explicit task functionalities of *OpenMP 4.5*. The grouping of calculations to tasks is realized in the same manner as *qui8*. However, since *OpenMP 4.5* does not support dynamic ordering of tasks that cannot be executed at the same time, the same coloring scheme as in *c08* is needed. Therefore this approach is called *taskc08*.
The tasks are ordered by a system of flag passing and order of generation. Tasks can be configured to wait to start until they can exclusively pick up predefined flags, or to hand out flags upon finishing. Here, every cell represents such a flag. Also in contrast to *Quicksched*, tasks are immediately executed as soon as they are generated.

## 4.3 qui1

The most fine-grained approach is to see every arrow from Figure 13 as a single task. Thereby only two cells per task are locked but 14 times as many tasks as *qui8* are produced in 3D. This approach shall be called *qui1*.

## 4.4 qui_adj_n

The last approach presented builds upon the *qui8* approach but adds more flexibility. A task is still a cube of cells but here the edge length, or number of cells per dimension, which will be called $n$, is variable. This approach is called *quicksched_adjustable_n* (short: *qui_adj_n*). Since tasks need to overlap by one cell to cover all computations of the domain there needs to be an offset of $n - 2$ between task $k$ and $k + 2$. Task $k + 1$ will stretch from the last column of task $k$ to the first of $k + 2$. These offsets are the same for each dimension. All calculations within a task are executed in serial, so no race conditions can occur. An example for $n = 3$ can be observed in Figure 15.

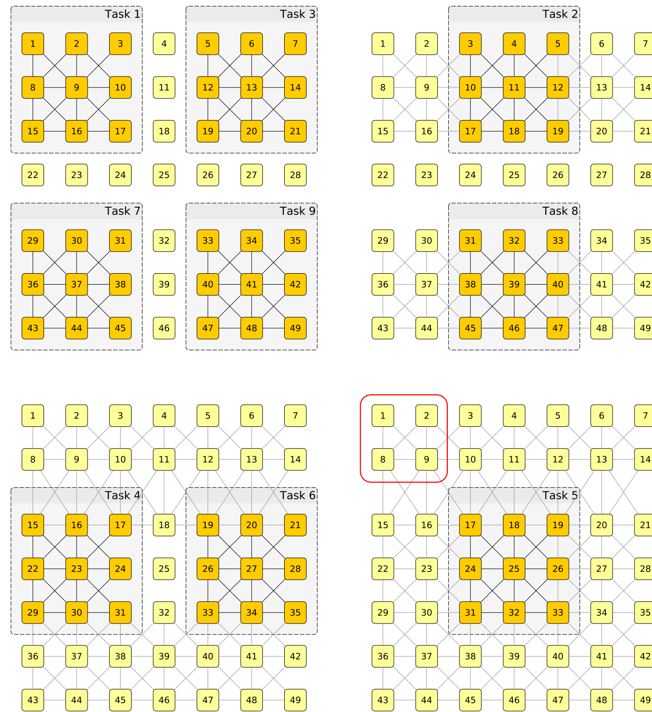



Figure 15: Scheme *qui_adj_n* for $n = 3$. The four areas of the graphic show which tasks can be executed in parallel.

This approach reduces the number of tasks, increases the computational costs, and thereby reduces the pressure on the scheduler. It can also be seen that locking becomes an increasing obstacle to parallelism. While in Figure 15 task 5 is running, no other task can be allowed to run, and thereby for example the calculations between cells 1, 2, 8 and 9 (circled in red) cannot be executed although they are not affected.

## 4.5 Task Weighting

As mentioned in Section 2.2.1, tasks can be weighted to improve their ordering in queues. It is reasonable to assume that a higher concentration of molecules correlates with higher computational costs per task.
*ls1-mardyn* includes a calculator for the number of floating point operations (FLOPs) needed to compute the Lennard-Jones potential between all particles of two cells. The number of operations depends on the number of particles in each cell as well as the number of Lennard-Jones Centers per molecule (= points in a molecule from which the potential needs to be calculated). The sum of the FLOP counts between all particle is used as the cost value of the task.
The resulting distribution of costs over the domain is shown in Figure 17 for a molecule distribution seen in Figure 16. The used tasking scheme here is *qui8*, which results in $20 \times 20 \times 20 = 8000$ tasks containing a total of 40.000 particles.

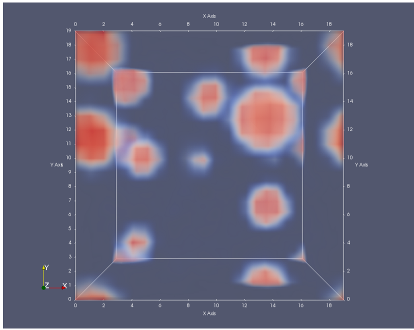


Figure 16: Distribution of molecules in the *lj40000-t300* experiment. Darker red means more molecules.

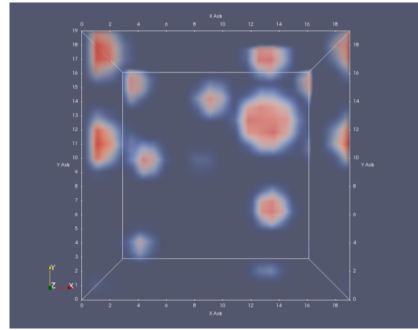


Figure 17: Calculated task costs. Darker red means higher costs.

It can be observed that the overall patterns between the figures are very similar, with the cost pattern appearing a little bit fainter. This can be explained by the cells lying on the outer region of an area of high particle count having reduced computational costs since they also interact with cells that contain barley any molecules. Since a task consists of eight cells, the task id and thereby its costs are saved in the frontal lower right cell. This results in a slight shift in between the two figures.

# 5 Results

## 5.1 Experimental Setup

In order to evaluate the performance of the approaches described in Section 4 , two scenarios are of particular interest.
The first one is a larger domain filled with 1.317.006 equally distributed particles with four Lennard-Jones centers. It is divided in $42 \times 42 \times 42 = 74088$ cells, including the

halo area. This scenario will be called *Aceton RC3*.
The second scenario is the in Section 4.5 shown heterogeneous scenario. Here the domain is smaller with $21 \times 21 \times 21 = 9261$ cells including halo filled with 40.000 particles with one Lennard-Jones center. This scenario will be called *LJ40000-T300*.
Its size makes the first scenario much more CPU-intensive since it contains almost 33 times as many particles in a domain that has only eight times as many cells. Also, the heterogeneous distribution in the second scenario introduces huge load imbalances. For this reason we focus primarily on *LJ40000-T300*, as it is more difficult to parallelize efficiently.

## 5.2 Strong-scaling

For both scenarios described in Section 5.1 strong-scaling experiments from one to 240 threads were conducted on the same 60 core *Knights Corner* cards that were described in Section 2.3. The results can be observed in figures 18 and 19, where the established approach *c08* is used as a reference base for the scaling.

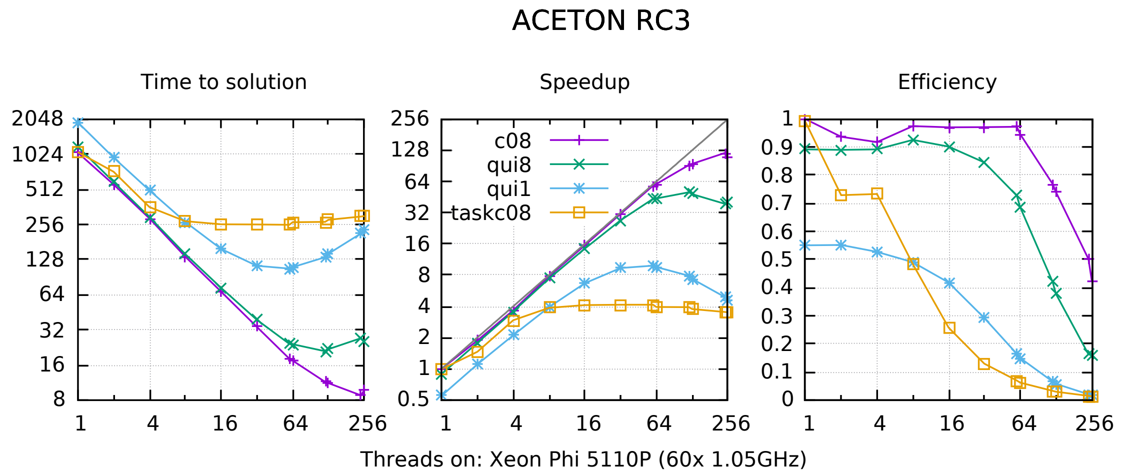



Figure 18: Strongscaling of homogeneous scenario on SuperMIC.

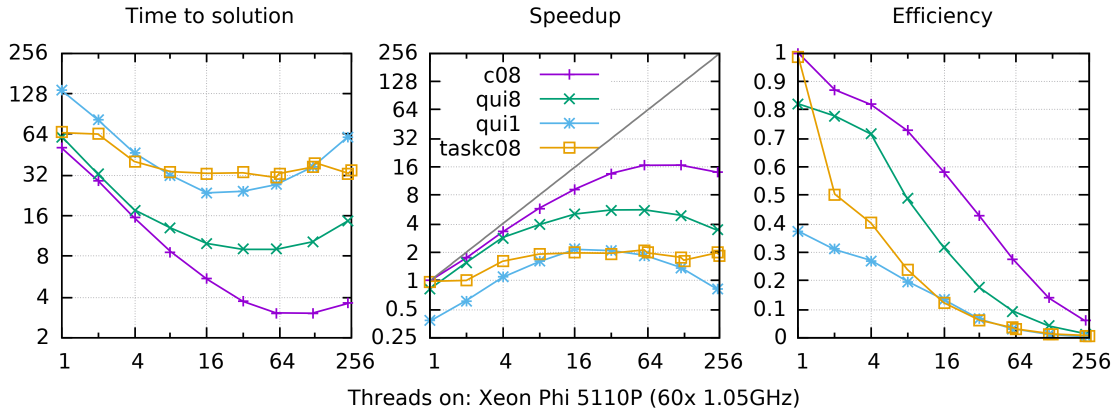



Figure 19: Strongscaling of heterogeneous scenario on SuperMIC.

When taking a look at the *OpenMP* explicit task approach *taskc08*, no speedup beyond 8 cores can be observed in neither scenario, probably due to the rigid structure of *OpenMP's* tasks as explained in Section 4.2. Not only are the *Quicksched* approaches faster but also demonstrate significantly better scaling capabilities in both scenarios.
Analyzing the first homogeneous scenario it can be seen that the approach *qui8* is only marginally slower than the almost perfectly scaling *c08* approach up to 60 cores. The very fine grained approach *qui1* is significantly slower on all thread counts and also scales much worse with serious efficiency drops starting already at eight threads. This might be due to the approach being too fine grained and thereby generating too many tasks to handle.
When using more threads than cores for both *Quicksched* approaches, no real performance gains can be achieved or even losses are measured. This, however, is in agreement with expectations drawn from the benchmark performance seen in Section 2.3 where hyperthreading also resulted in only minor performance gains.
Therefore, further analysis is dedicated to the heterogeneous scenario *LJ40000-T300* at 60 threads in order to understand where *Quicksched* loses performance.

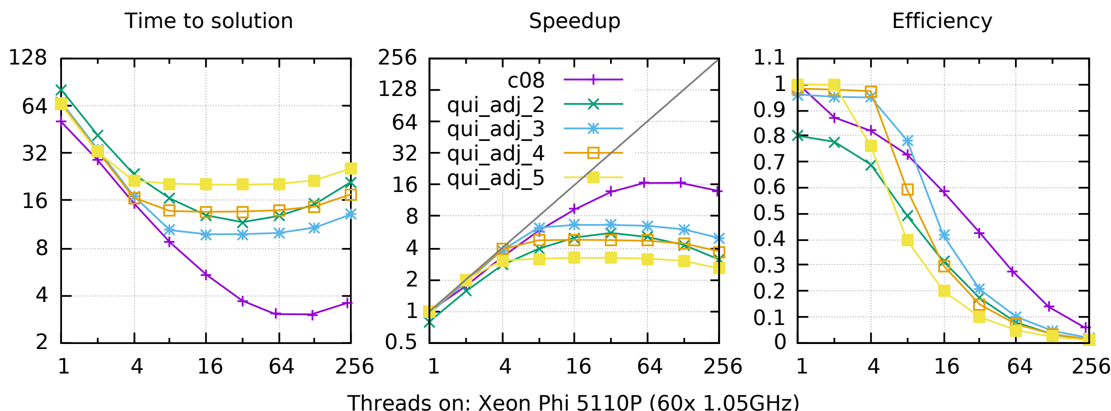



Figure 20: Strongscaling of heterogeneous scenario on SuperMIC with *qui_adj_n*.

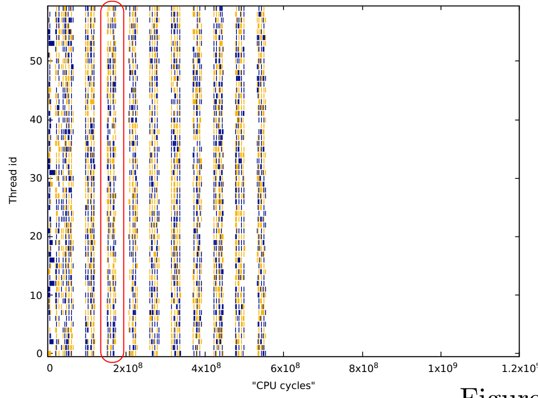
By applying the approach *qui_adj_n*, that was explained in Section 4.4, for different $n$, the influence of the size of a task can be evaluated. The measurements for $n$ from two to five are presented in Figure 20, where $n = 2$ is equivalent to the *qui8* approach. Performance with $n = 3$ is slightly better, which seems to indicate that the computational intensity of tasks previously was too small.
With further growing size of tasks once again the performance drops. This is probably due to a too coarse granularity where too many cells are blocked resulting in idle processor cores. For example, for $n = 5$ only 125 tasks are generated for 60 threads, so there are only approximately two tasks per core.

## 5.3 Scheduling Timings

In order to gain more insight into the results of the scheduling process a closer look shall be taken on how many cycles each core spends on force calculations. The horizontal bars in Figures 21 to 26 represent the cycles of each processor, with colored bars signifying cycles spent on force calculations. To increase distinguishability, the colors of succeeding tasks are alternating horizontally between blue and orange. The colors themselves have no special meaning. White space signifies anything but force calculation, mainly scheduling activity or idle time at barriers. Also, in the left graphs between every time step some cycles in the white space are used for inter-time step activities like for example updates to positions and speed or thermostats.

Ten time steps on Xeon Phi using different scheduling schemes.

Third Timestep of left graphic. (Marked in Red.)



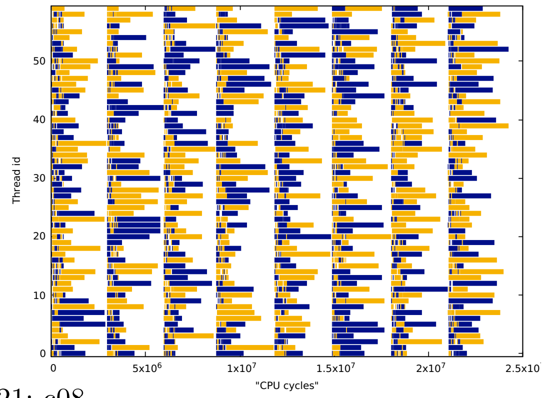
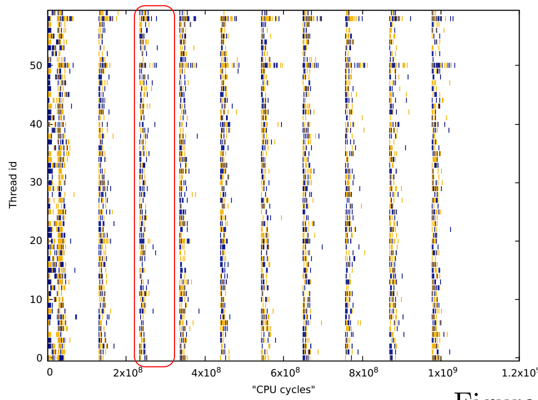




Figure 21: c08



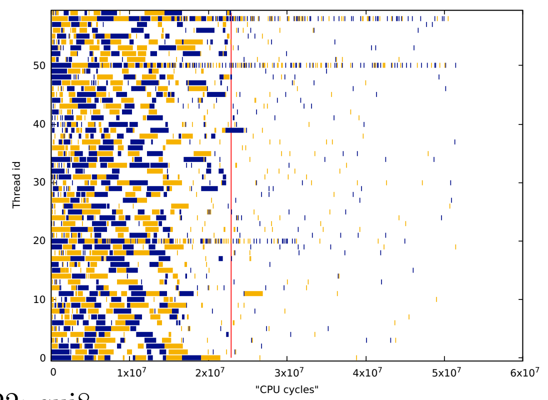
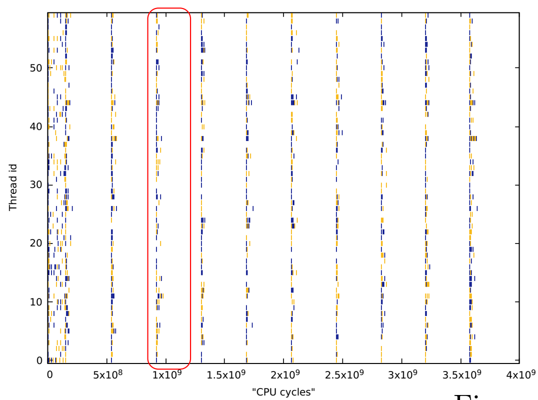




Figure 22: qui8



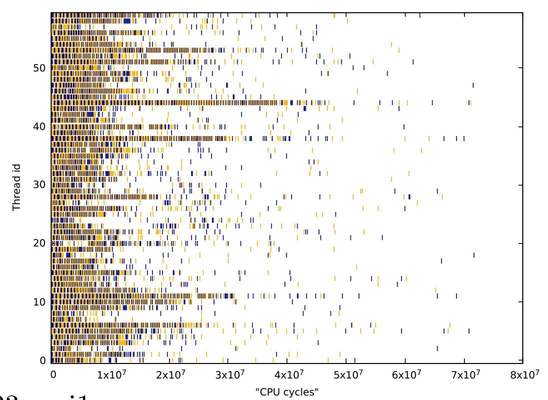




Figure 23: qui1

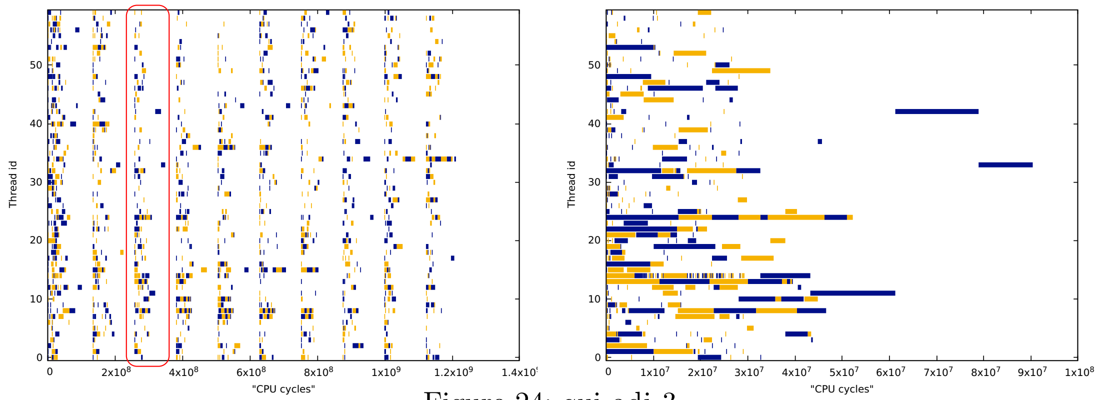


Figure 24: qui_adj_3

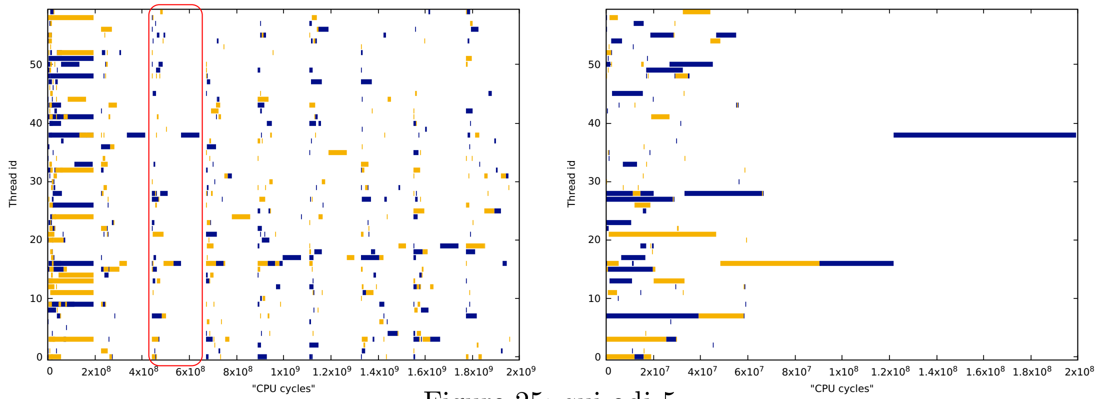


Figure 25: qui_adj_5

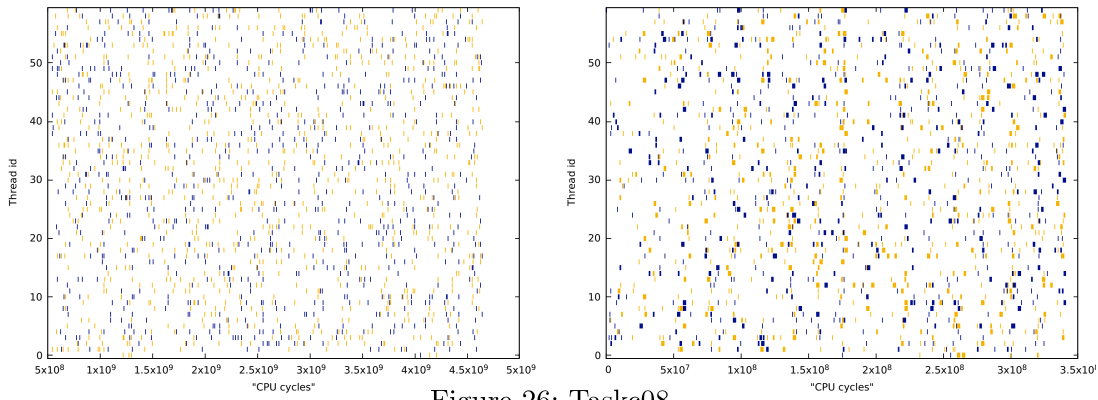


Figure 26: Taskc08

The left Figures 21 to 26 show the timings for ten time steps of the *LJ40000-T300* scenario. In Figure 22 it can be seen that the time steps in the *qui8* run are almost three times further apart than with the reference approach *c08*, which is shown in Figure 21. This observation matches the approximately three times slower execution times for 60 threads shown in Figure 19.

The right Figures 21 to 26 are zoomed-in on the third time step marked red in the

respective Figures to analyze the scheduling activity of both approaches on a more fine grained scale.
In Figure 21, which visualizes the *c08* run, the eight *OpenMP* barriers of the coloring algorithm are clearly visible. It can also be observed that larger tasks are not prioritized and thereby the distribution of smaller tasks leaves room for optimization.
The same can be said for *taskc08*, seen in Figure 26 however this approach even fails to actually use all processors at once. This could be due to the reason of how the tasks are generated or the scheduler using a non-optimal way of finding tasks that are ready to be executed.
Figure 22 shows that *Quicksched's* prioritization of more CPU-intensive tasks works. However, it appears that the threads 20, 50, and 58 received the majority of very cheap tasks. These are tasks covering the rather empty parts of the domain. As soon as any other thread is done with its tasks it starts the process of work-stealing, described in Section 2.2.2, by hitting the queues of a randomly selected queue. Yet at some point only the aforementioned three threads still have tasks. This results in a probability of 0.05 for finding a task, which is obviously too small to be efficient. The red line visualizes the point of time where the *c08* variant finishes. When looking at Figures 24 and 25 it can be observed that by enlarging the tasks the number of small and cheap tasks is reduced significantly, which reduces the work stealing problem. However, now the situation arises that the last tasks are mutually exclusive, since they operate on shared cells. This leads in the case of *qui_adj_3* to the sequential execution of roughly $\frac{1}{3}$ of the time step, or even $\frac{2}{3}$ for *qui_adj_5*. Such behavior indicates that these tasks are too big for this scenario, so that the parallel granularity is to coarse.
Table 1 shows for each parallelization scheme the number of tasks and how many of them can be executed in parallel on average due to blocking of shared cells. For all *c08* based approaches this is $\frac{1}{8}$ of the total number, analogue to the coloring. With *qui1* only $\frac{1}{27}$ of the total task count can run simultaneously because 27 tasks share the same cell, as can be seen in 12, and thus a coloring would require 27 colors.

| Parallelization Scheme | Total number of tasks | Avg. number of tasks that can run in parallel |
|---|---|---|
| qui8 | 8000 | 1000 |
| qui1 | 105435 | 3905 |
| qui_adj_3 | 1000 | 125 |
| qui_adj_4 | 343 | 42.875 |
| qui_adj_5 | 125 | 15.625 |
| taskc08 | 8000 | 1000 |

Table 1: Correlation between schemes and degree of parallelism for *LJ40000-T300*.

In Table 1 it can also be observed that already for *qui_adj_3* there are always only roughly two tasks per thread that can be executed. Since *Quicksched* implements no dedicated mechanism to order the tasks with respect to their resource locking, each processor starts

an exhaustive search, which also adds to the overhead in Figures 22 to 25. Finally it is shown that the *qui_adj_5* approach results here in only 125 tasks with only 15 of them being able to run in parallel, therefore the 60 processors can never all be used at the same time. This also explains the almost horizontal parts of the graphs for *qui_adj_3*, *qui_adj_4*, and *qui_adj_5* in Figure 20 because these are the parts where more processors cannot be put to use since there are no further tasks that can lock their required resources. For tasks of this size larger scenarios are needed.

### 5.4 Function Runtime Cost Distribution

Another way of analysis is to look at the time the code spends in each *Quicksched* function by reading the internal timers provided by the library itself. Figure 27 shows the percentages each *Quicksched* function takes of one run of `qsched_run()`, which is the whole force calculation of one time step again in the *LJ40000-T300* scenario on the *Xeon Phi* with the *qui8* tasking scheme.

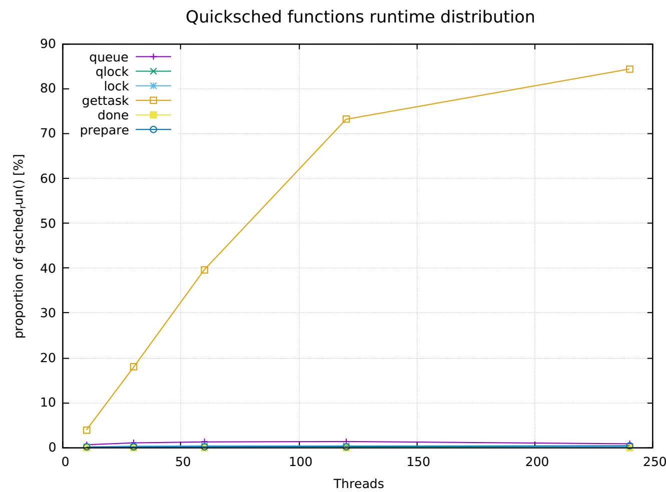



Figure 27: Percentages of *Quicksched* functions of the force calculation of one time step using *qui8*.

Only `qsched_gettask()` takes a significant share of time. This share however grows very quickly with the number of cores. At 60 threads about 40% of time is spent trying to fetch tasks, which is in agreement with the right chart in Figure 22 being approximately 40% white. This also underpins the explanation that the random selection of queues to steal from as described in Section 5.3 is a major problem, since this is part of `qsched_gettask()` as explained in Section 2.2.2.

# 6 Conclusions and Outlook

In the course of this paper, *qui1*, *qui8*, and *qui_adj_n* as *Quicksched*-based approaches of different granularity were presented as well as an approach to the weighting of tasks. These method's parallel performance relative to the established approach was analyzed and discussed.

The established *OpenMP*-based approach *c08* proves to be very efficient, but as can be seen in Figure 21 still leaves room for optimization. This potential is lost due to the missing concepts of prioritizing expensive blocks of computations, not following the critical path of execution, and existing color barriers blocking more dynamic use of compute resources. These are the points where the *Quicksched* task-based approaches gave promising results. The approach using *OpenMP 4.5*'s explicit tasks did not achieve this due to its rigidness.

The main problem, as shown in 5.3 and 5.4, appears to be that the work-stealing from a random queue can lead to immensely non-optimal behavior on imbalanced queues. Here, work on the *Quicksched* scheduler at how to choose the queue to steal from might be interesting.

It was demonstrated that a variation in granularity can have a severe impact on the speed of execution. A very high total number of tasks can be disadvantageous because it introduces a significant scheduling overhead. On the other hand, very few tasks that mutually block each other are also detrimental to parallel performance. Therefore, it is important to invest time in determining an optimal computational sizes of single tasks and granularity of parallelization over the whole domain. Yet, the *LJ40000-T300* scenario appears to be too small for task blocks of $3 \times 3 \times 3$ cells. Hence more adaptive task sizes that not only allow cubes but cuboids might be more helpful.

Furthermore, it is worth mentioning that these approaches made no use of the hierarchical aspect of *Quicksched's* conflict and dependencies model, which offers more generality than needed here.

Another interesting improvement would be to use tasks not only for the force calculation but for further procedures of the time step, like updates to speed and position, *MPI* communication, or thermostats. This would remove further barriers and allow for better utilization of idle processors.

Thereby, we conclude that task-based approaches for *ls1-mardyn* show potential and that it will prove worthwhile to put further work into improving the structuring of tasks and their scheduling.

# References


[Bin95] Kurt Binder. *Monte Carlo and molecular dynamics simulations in polymer science.* Oxford University Press, 1995.

[DES+11] Stephan Deublein, Bernhard Eckl, Jürgen Stoll, Sergey V Lishchuk, Gabriela Guevara-Carrion, Colin W Glass, Thorsten Merker, Martin Bernreuther, Hans Hasse, and Jadran Vrabec. ms2: A molecular simulation tool for

thermodynamic properties. *Computer Physics Communications*, 182(11):2350–2367, 2011.

[EHB+13] Wolfgang Eckhardt, Alexander Heinecke, Reinhold Bader, Matthias Brehm, Nicolay Hammer, Herbert Huber, Hans-Georg Kleinhenz, Jadran Vrabec, Hans Hasse, Martin Horsch, et al. 591 tflops multi-trillion particles simulation on supermuc. In *International Supercomputing Conference*, pages 1–12. Springer, 2013.

[GCS16] Pedro Gonnet, Aidan BG Chalk, and Matthieu Schaller. Quicksched: Task-based parallelism with dependencies and conflicts. *arXiv preprint arXiv:1601.05384*, 2016.

[GKZ07] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. *Numerical simulation in molecular dynamics: numerics, algorithms, parallelization, applications*, volume 5. Springer Science & Business Media, 2007.

[KDFB04] Douglas B Kitchen, Hélène Decornez, John R Furr, and Jürgen Bajorath. Docking and scoring in virtual screening for drug discovery: methods and applications. *Nature reviews Drug discovery*, 3(11):935–949, 2004.

[LJ31] John Edward Lennard-Jones. Cohesion. *Proceedings of the Physical Society*, 43(5):461, 1931.

[NBB+14] Christoph Niethammer, Stefan Becker, Martin Bernreuther, Martin Buchholz, Wolfgang Eckhardt, Alexander Heinecke, Stephan Werth, Hans-Joachim Bungartz, Colin W Glass, Hans Hasse, et al. ls1 mardyn: The massively parallel molecular dynamics code for large systems. *Journal of chemical theory and computation*, 10(10):4455–4464, 2014.

[RBM+96] Dennis C Rapaport, Robin L Blumberg, Susan R McKay, Wolfgang Christian, et al. The art of molecular dynamics simulation. *Computers in Physics*, 10(5):456–456, 1996.

[TWG+15] Nikola Tchipev, Amer Wafai, Colin W Glass, Wolfgang Eckhardt, Alexander Heinecke, Hans-Joachim Bungartz, and Philipp Neumann. Optimized force calculation in molecular dynamics simulations for the intel xeon phi. In *European Conference on Parallel Processing*, pages 774–785. Springer, 2015.

[vGB90] Wilfred F van Gunsteren and Herman JC Berendsen. Computer simulation of molecular dynamics: Methodology, applications, and perspectives in chemistry. *Angewandte Chemie International Edition in English*, 29(9):992–1023, 1990.