**The Journal of Supercomputing manuscript No.**
(will be inserted by the editor)

# Tasking framework for Adaptive Speculative Parallel Mesh Generation

**Christos Tsolakis · Polykarpos Thomadakis · Nikos Chrisochoides**

Received: date / Accepted: date

**Abstract** Handling the ever-increasing complexity of mesh generation codes along with the intricacies of newer hardware often results in codes that are both difficult to comprehend and maintain. Different facets of codes such as thread management and load balancing are often intertwined, resulting in efficient but highly complex software. In this work, we present a framework which aids in establishing a core principle, deemed *separation of concerns*, where *functionality* is separated from *performance* aspects of various mesh operations. In particular, thread management and scheduling decisions are elevated into a generic and reusable tasking framework.

The results indicate that our approach can successfully abstract the load balancing aspects of two case studies, while providing access to a plethora of different execution back-ends. One would expect, this new flexibility to lead to some additional cost. However, for the configurations studied in this work, we observed up to 13% speedup for some meshing operations and up to 5.8% speedup over the entire application runtime compared to hand-optimized code. Moreover, we show that by using different task creation strategies, the overhead compared to straight-forward task execution models can be improved dramatically by as much as 1200% without compromises in portability and functionality.

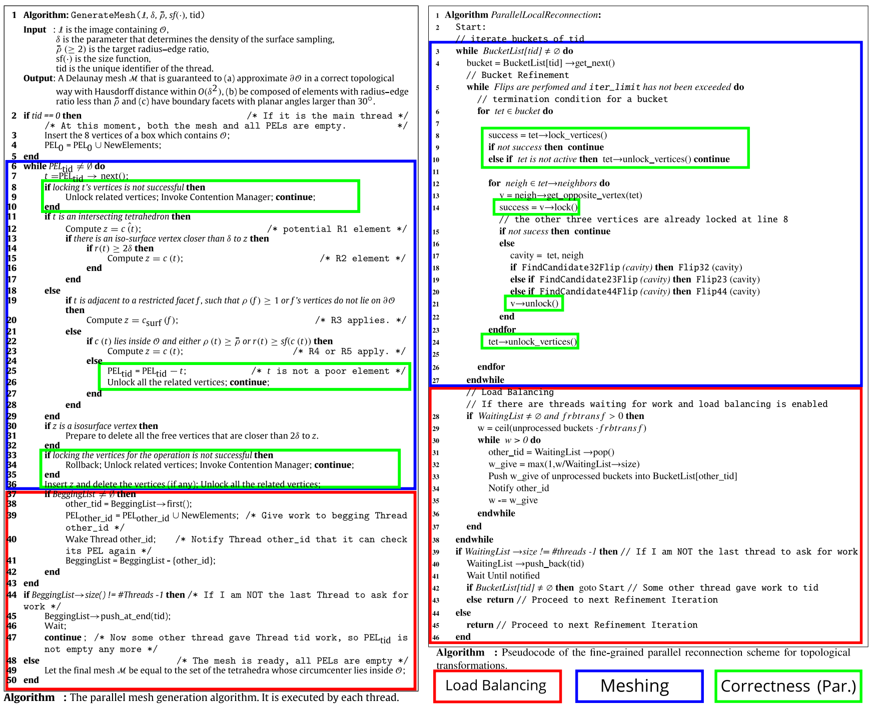
**Keywords** Parallel Computing · Tasking · Speculative Execution · Mesh Generation · Mesh Adaptation

## 1 Introduction

With the advent of multicore machines and new constantly evolving architectures in the form of accelerators, the need for abstracting parallelism in

Christos Tsolakis, Polykarpos Thomadakis, Nikos Chrisochoides
Center for Real-Time Computing, Old Dominion University, Virginia, USA
E-mail: {ctsolakis,pthomadakis,nikos}@cs.odu.edu

scientific applications becomes paramount. Although platform-specific and/or application-specific optimizations will always perform better than generic solutions, abstract interfaces can last longer and allow for better interoperability between applications. Choosing the right abstractions allows applications to build upon a generic framework while enabling low-level software substrates to offer implementations that take advantage of the underlying hardware. Moreover, abstractions provide space for future explorations and allow to future-proof [28] applications; as newer hardware (e.g., in the form of accelerators) becomes available, the application developer may need to perform minimal to no changes while the underlying runtime system can add new features opaquely.


```
1  Algorithm: GenerateMesh($\mathcal{I}$, $\delta$, $\bar{\rho}$, $sf(\cdot)$, tid)
   Input  : $\mathcal{I}$ is the image containing $\mathcal{O}$,
            $\delta$ is the parameter that determines the density of the surface sampling,
            $\bar{\rho}$ ($\geq 2$) is the target radius–edge ratio,
            sf($\cdot$) is the size function,
            tid is the unique identifier of the thread.
   Output: A Delaunay mesh $\mathcal{M}$ that is guaranteed to (a) approximate $\partial\mathcal{O}$ in a correct topological
            way with Hausdorff distance within $O(\delta^2)$, (b) be composed of elements with radius–edge
            ratio less than $\bar{\rho}$ and (c) have boundary facets with planar angles larger than $30^\circ$.
2  if tid == 0 then                                          /* If it is the main thread */
       /* At this moment, both the mesh and all PELs are empty.                        */
3      Insert the 8 vertices of a box which contains $\mathcal{O}$;
4      $PEL_0 = PEL_0 \cup$ NewElements;
5  end
6  while $PEL_{tid} \neq \emptyset$ do
7      $t = PEL_{tid} \rightarrow$ next();
8      if locking t's vertices is not successful then
9          Unlock related vertices; Invoke Contention Manager; continue;
10     end
11     if t is an intersecting tetrahedron then
12         Compute $z = c(\hat{t})$;                              /* potential R1 element */
13         if there is an iso-surface vertex closer than $\delta$ to z then
14             if $r(t) \geq 2\delta$ then
15                 Compute $z = c(t)$;                                /* R2 element */
16             end
17         end
18     else
19         if t is adjacent to a restricted facet f, such that $\rho(f) \geq 1$ or f's vertices do not lie on $\partial\mathcal{O}$
           then
20             Compute $z = c_{surf}(f)$;                             /* R3 applies. */
21         else
22             if $c(t)$ lies inside $\mathcal{O}$ and either $\rho(t) \geq \bar{\rho}$ or $r(t) \geq sf(c(t))$ then
23                 Compute $z = c(t)$;                              /* R4 or R5 apply. */
24             else
25                 $PEL_{tid} = PEL_{tid} - t$;                /* t is not a poor element */
26                 Unlock all the related vertices; continue;
27             end
28         end
29     end
30     if z is a isosurface vertex then
31         Prepare to delete all the free vertices that are closer than $2\delta$ to z.
32     end
33     if locking the vertices for the operation is not successful then
34         Rollback; Unlock related vertices; Invoke Contention Manager; continue;
35     end
36     Insert z and delete the vertices (if any); Unlock all the related vertices;
37     if $BeggingList \neq \emptyset$ then
38         other_tid = BeggingList→first();
39         $PEL_{other\_id} = PEL_{other\_id} \cup$ NewElements;  /* Give work to begging Thread
           other_id */
40         Wake Thread other_id;     /* Notify Thread other_id that it can check
           its PEL again */
41         BeggingList = BeggingList - {other_id};
42     end
43 end
44 if BeggingList→size() != #Threads -1 then /* If I am NOT the last Thread to ask for
   work */
45     BeggingList→push_at_end(tid);
46     Wait;
47     continue ;  /* Now some other thread gave Thread tid work, so $PEL_{tid}$ is
       not empty any more */
48 else                                  /* The mesh is ready, all PELs are empty */
49     Let the final mesh $\mathcal{M}$ be equal to the set of the tetrahedra whose circumcenter lies inside $\mathcal{O}$;
50 end
```
**Algorithm :** The parallel mesh generation algorithm. It is executed by each thread.

```
1  Algorithm ParallelLocalReconnection:
2    Start:
     // iterate buckets of tid
3    while BucketList[tid] $\neq \emptyset$ do
4      bucket = BucketList[tid] →get_next()
       // Bucket Refinement
5      while Flips are perfomed and iter_limit has not been exceeded do
         // termination condition for a bucket
6        for tet $\in$ bucket do
7
8          success = tet→lock_vertices()
9          if not success then continue
10         else if tet is not active then tet→unlock_vertices() continue
11
12         for neigh $\in$ tet→neighbors do
13           v = neigh→get_opposite_vertex(tet)
14           success = v→lock()
             // the other three vertices are already locked at line 8
15           if not sucess then continue
16           else
17             cavity = tet, neigh
18             if FindCandidate32Flip (cavity) then Flip32 (cavity)
19             else if FindCandidate23Flip (cavity) then Flip23 (cavity)
20             else if FindCandidate44Flip (cavity) then Flip44 (cavity)
21             v→unlock()
22           end
23         endfor
24         tet→unlock_vertices()
25
26       endfor
27     endwhile
       // Load Balancing
       // If there are threads waiting for work and load balancing is enabled
28     if WaitingList $\neq \emptyset$ and $frbtransf > 0$ then
29       w = ceil(unprocessed buckets $\cdot frbtransf$)
30       while $w > 0$ do
31         other_tid = WaitingList →pop()
32         w_give = max(1,w/WaitingList→size)
33         Push w_give of unprocessed buckets into BucketList[other_tid]
34         Notify other_id
35         w -= w_give
36       endwhile
37     end
38   endwhile
39   if WaitingList →size != #threads -1 then // If I am NOT the last thread to ask for work
40     WaitingList →push_back(tid)
41     Wait Until notified
42     if BucketList[tid] $\neq \emptyset$ then goto Start // Some other thread gave work to tid
43     else return // Proceed to next Refinement Iteration
44   else
45     return // Proceed to next Refinement Iteration
46   end
```
**Algorithm :** Pseudocode of the fine-grained parallel reconnection scheme for topological transformations.

(a) PODM pseudocode as presented in [27]. (b) CDT3D pseudocode as presented in [20].

Fig. 1: Pseudocodes of the speculative approach applied to a Delaunay-based algorithm (left) and a local reconnection operation (right) of an Advancing-Front method. Colored regions indicate the primal function of the enclosed steps.

The task management facilities presented in this work, are part of a longer-term project that aims to enable *separation of concerns* [18] with regard to *functionality* and *performance*, specifically for mesh operations. Figure 1 depicts the pseudocode of two fine-grained speculative meshing operations. Note that the application developer must manage and account for the meshing kernel, parallel correctness, and load balancing, all within a single algorithm. Developing and maintaining such an application becomes challenging since the developer has to keep all three parts in mind while modifying the code. Moreover, re-using hardware-specific optimizations among different applications can only be achieved by abstracting them outside of specific applications. Examples include data affinity-aware work schedulers, cache-optimized data structures, thread contention management policies, etc.

The separation of *functionality* from *performance* will contribute significantly towards the implementation of the *Telescopic Approach* [15] which lays down a design for achieving scalability for mesh generation on exascale machines. The *Telescopic Approach* spans across the multiple memory hierarchies of an exascale machine (shared, distributed-shared (DSM), distributed, out-of-core) and maps different algorithmic layers to the appropriate level of memory based on the intensity of communication between different meshing kernels. The lowest, closest to the hardware layer, that can sustain high volume of communication at low cost, is the Parallel Optimistic layer which is designed to explore concurrency at the CPU level within the limits of shared memory, using speculative/optimistic execution. Both meshing kernels of Figure 1 are designed to be used in the Parallel Optimistic layer of the *Telescopic Approach*. Abstracting the *performance* aspects from the *functionality* for these kernels will allow interoperability with lower level runtime systems like PREMA [41] and will speed up the development process by increasing the code-reuse among the applications that utilize the *Telescopic Approach*.

As case studies, we use the parallel meshing operations present in CDT3D [20, 19] that are common in most metric-based mesh adaptation codes [44] and the Delaunay-based kernel of PODM [27]. For these two applications, we explore ways a tasking environment can be used to express speculative mesh operations and describe approaches that enable to abstract the load balancing aspects of both case studies with no impact to functionality. The results in Sections 4.1 and 4.2 indicate not only low overhead, but even speedup with respect to the baseline hand-optimized applications for some mesh operations. In summary, the main contributions of this study are:

– Present a high-level front-end that abstracts and unifies task management for adaptive and irregular applications.
– Design and implement the front-end for three major back-ends: Intel®'s TBB, OpenMP, and Argobots.
– Illustrate how this front-end can be applied to two different speculative parallel unstructured mesh generation codes.
– Provide an in-depth analysis of the effect of task granularity on each back-end along with the advantages and disadvantages of different task creation strategies.

## 2 Related Work

There is a number of tasking systems that have emerged in the recent years targeting different layers of the application, from low-level assembly layer [35], and parallelizing compilers [49] to high level tasking frameworks [13]. A complete review of the current state-of-the-art tasking environments is outside the scope of this section. A comprehensive taxonomy based on architectural characteristics and user APIs appears in [42]. In the rest of this section, we focus on methods that exploit concurrency through speculative execution. *Speculative* execution (also known as *optimistic*) is a technique used in a number of applications, ranging from processors [43] to databases [32]. It allows for the exploitation of more concurrency out of a problem by executing steps of a procedure ahead of time, prior to resolving data dependencies between the steps themselves. In the case where steps are not needed, the precomputed results may either be disregarded or additional steps may be required to *roll back* to the previous state of the process. The correctness of this scheme, in the context of parallel algorithms, has been proven in [30] with the introduction of the notion of *Virtual Time* and validated in the context of Parallel Discrete Event Simulations within the *Time Warp* system.

There are several efforts in the literature that facilitate speculative execution utilizing higher level constructs. Among the many we list a few pertinent to this study such as the use of transactional memory at the software level [36], compiler-assisted methods [37,11,1] and libraries such as Galois [31], ParlayLib [6] and SPETABARU [10].

The Galois system [31] provides abstract set iterators, giving to the application developer the ability to extract parallelism out of the work-lists of a sequential application. Custom data structures and a runtime scheduler are responsible for detecting and recovering unsafe accesses to shared memory. The elegance of this approach is appealing but, for our use-case, it would require extensive modifications of the work lists maintained by each application. Also, to the best of our knowledge, its effectiveness for mesh generation has been demonstrated only on simple sequential mesh triangulation codes. In contrast, in this work, both use-cases build on top of an already-parallelized application that have demonstrated comparable performance to state-of-the-art methods [27,45].

In [7,6] the authors revisit the idea of expressing speculative execution as a combination of nested parallelism and commutative operations suggested in [40] and propose the use of *deterministic reservations* for dealing with a class of greedy algorithms. The main idea is to split the operation into two phases. One that attempts to reserve the data dependencies for a number of tasks speculatively, and then a commit phase that executes the tasks that successfully reserved all their dependencies. The two-phase approach is similar to the *inspector-executor* model [38]. However, an *inspector-executor* model is not suitable for our case due to the data-intensive nature of the targeted use-cases. In contrast, our approach re-uses the speculative approach step already present in both use-cases and merges the two steps in one. This approach acts directly upon touched data which improves cache utilization and allows tolerating more than 80% of system latencies [34]. Moreover, it avoids the synchronization required by a two-phase approach.

The SPETABARU tasking runtime system introduced in [10] can exploit concurrency of task graphs through speculation. The task graph is built based on the user-defined data dependencies between the tasks. The system manages the execution of tasks as well as disregarding data from failed speculative attempts upon runtime. The library was originally created for Parallel Monte Carlo Simulations, and it is primarily designed for parallel applications that utilize graphs of tasks. This tasking system generates the graph utilizing a single thread in a pre-processing step that generates all the tasks and evaluates the data dependencies among them. This approach is inadequate for our target data-intensive applications for two reasons. First, dependency discovery and resolution is the most expensive step, thus rendering the pre-processing step to a major bottleneck. Moreover, the continuous generation of new elements (and therefore tasks) would require additional synchronization points which would degrade performance significantly.

In [1] the authors incorporate Thread-Level Speculation in OpenMP. OpenMP is extended with the `speculative` directive which annotates a variable as the target of speculative execution. A thread-local version of such variables is created for each thread. The respective runtime monitors such variables and guarantees that all read accesses will return the most up-to-date value. When a thread consumes an outdated version of a `speculative` variable, it is stopped and restarted in order to consume the correct value of the variable. For both of our applications the speculative execution is already part of the application code and modifying it is outside the scope of this work. Moreover, the abstract front-end of our approach gives access to additional back-ends beyond OpenMP.

Although many of the above approaches are close to our goals, most of them will require nontrivial changes to the code required for the *Parallel Correctness* steps (see Figure 1) of the algorithm which for this study we chose to keep as part of the meshing task. Moreover, in contrast to all the presented approaches, the starting point in this work is applications that are already parallel instead of sequential. This comes with the benefit of having thread-safe mechanisms in place for memory allocation and speculative locking, but also with higher complexity due to their legacy nature. Also, the proposed approach can utilize a number of different back-ends, including Argobots [39]. Argobots provides lightweight User Level Threads (ULTs), capable of context switching with low overhead. Among others, ULTs are employed to tolerate latencies in cases such as failing to acquire a lock or calling synchronous MPI operations where regular tasks would block, along with the underlying hardware thread. Instead, a ULT will implicitly release the hardware thread it runs on, allowing other ULTs to use it. This interaction with MPI allows to build scalable runtime systems such as the PREMA runtime system [41], which in turn is a building block of the *Telescopic Approach* [15].

In the context of mesh generation, speculative execution was introduced in [34] where the authors execute the same meshing kernel across multiple processes without restricting them on their local data. Instead, the meshing kernel is launched *optimistically*, and the data dependencies are discovered and captured on the fly. If some dependency cannot be satisfied, the operation releases any captured dependencies, aborts its execution (rollback), and reenters the scheduler's pool. The speculative approach has already been utilized in Delaunay triangulation methods [5,4,25], Delaunay Refinement [27] and Advancing-Front [20] methods. However, in each case, the approach was application- and method-specific. In contrast, in this work we provide an abstract interface allowing the framework to be applied both across different mesh operations and between different mesh applications. Moreover, the approach in this work is method-agnostic, thus rendering the framework useful for other meshing methods as well as for adaptive and irregular applications in general.

## 3 Method

The proposed approach builds upon the observation that the speculative meshing operations of the two case studies can be decomposed into three components: *Meshing*, *Parallel Correctness* and *Load Balancing* (see Figure 1). The Load Balancing part is handled by the generic tasking framework presented in the following sections. *Parallel Correctness* is expressed through the use of atomic locks upon the cavity (data dependencies) of each operation. For this study, the parallel correctness steps will remain as part of the meshing task. The implementation of generic tasking framework is composed of an application-facing front-end which is agnostic to target hardware, and a back-end, which provides custom implementations for individual substrates.

### 3.1 Front-end: A Generic Tasking Framework

The general approach used in this work is to decompose any given operation into non-interruptible tasks that execute to completion. The framework supports both blocking calls for creating many tasks (see Listing 1) and a fine-grained API for single task creation (see Listing 2). This allows for the utilization of a range of tasking paradigms, from simple fork-join models to hierarchical or recursive task creation (see Figure 2).

```
/**
* @brief launch tasks and wait until they all finish
* @param user_task_args vector of task arguments
* @param user_func the user function to be executed,
* type should be void func(UserTaskArgs&),
* @param grainsize number of elements of user_task_args to group
* into a single task
*/

```

```
template<typename UserTaskArgs, typename FunctionType>
void task_for(std::vector<UserTaskArgs>& user_task_args,
              FunctionType user_func,
              int grainsize = 10)
```

Listing 1: Interface for launching several tasks at once.

```
/**
* @brief add a task to the internal queue
* @param user_task_args arguments of the task
* @param user_func the user function to be executed,
* type should be void func(UserTaskArgs&)
*/

template<typename UserTaskArgs,typename FunctionType>
void create_and_schedule(UserTaskArgs& user_task_args,
                         FunctionType user_func)

/**
* @brief wait until all generated tasks have completed.
*/
void wait_for_all()
```

Listing 2: Interface for creating a single task.

`task_for` in Listing 1 is similar to the parallel-for paradigm. It accepts a function `user_func` that implements the task operation, as well as a vector `user_task_args` of the different arguments for each task. Optionally, it can accept a grainsize which controls the number of terminal tasks that will be generated. The number of terminal tasks is `user_task_args.size()/grainsize`. Similar to `std::for_each`, `task_for` will apply `user_func` to each element of the `user_task_args` vector. However, in contrast to `std::for_each` not all invocations of `user_func` will be completed successfully. Some will abort due to rollbacks. In this study, re-applying the operation on aborted tasks is handled by the application logic, since it was already present before the introduction of this framework.

`create_and_schedule` in Listing 2 is a simple wrapper around the corresponding back-end that generates a task and places it in the internal queue of the framework. This call is not blocking, and the execution of the task may start immediately on a different thread. Finally, `wait_for_all` suspends the calling thread until the internal task queues are empty.

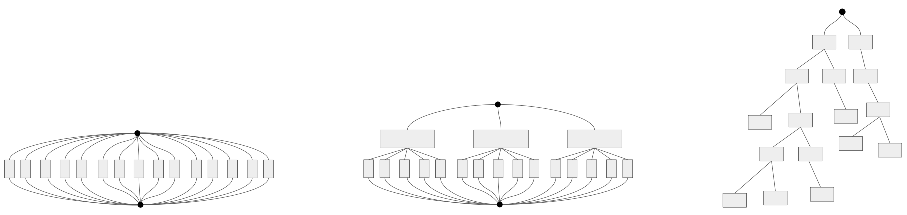


Fig. 2: Different tasking paradigms employed in this work. Left: flat model, Middle: two-level task creation, Right: hierarchical task creation.

The tasking framework also provides the user with a unique `thread_id` $\in [0, nthreads)$. This id is not pinned to any hardware thread, but it is guaranteed to stay fixed and unique for the duration of the task execution. The `thread_id` is required by both applications of this study for two main operations. First, data dependency acquisition is implemented utilizing atomic locks that hold the id of the owning thread [27,20]. Also, both pieces of software utilize the thread-aware memory management method described in [2] that uses a `thread_id` in order to access the appropriate thread-local memory pools that allow for the allocation and deallocation of elements in a thread-safe manner.

## 3.2 Task Generation Strategies

One of the considerations of explicitly creating tasks is the overhead of task creation. In the current implementation of `task_for`, there is support for all three task creation strategies of Figure 2. The `flat` model implements a basic fork-join paradigm [16]. It creates all tasks sequentially and waits for them to complete. As a first attempt to reduce the overhead, a `2level` task creation strategy was introduced. For this strategy, the application thread will spawn sequentially $2 \cdot nthreads$ tasks that partition the range of the `user_task_args` vector in equal parts. Each level-2 task will then iterate the assigned range of the task vector and spawn a task for each task-argument. Finally, the `hierarchical` model employs a divide-and-conquer scheme; it creates tasks recursively by creating two child tasks that bisect the task range up to the point where the assigned range is smaller or equal to the target grainsize. When the framework is used sequentially, no tasks are created independently of the chosen strategy. Instead, the application thread will apply `user_func` sequentially on each item of the `user_task_args` vector.

## 3.3 Implementation

The above framework is implemented with three different back-ends: the Argobots runtime system [39], Intel's TBB framework [48] and OpenMP [17]. For each of the three implementations we have incorporated the three task creation strategies of Figure 2 as well as high level constructs specific to each back-end such as `tbb::parallel_for`, `#pragma omp parallel for` and `#pragma omp taskloop` for a total of 12 different execution back-ends. We will use the notation `backend-strategy` to refer to tasking strategy `strategy` implemented on top of the back-end `backend`. Although interoperability in terms of mixing different backends is possible, for this work we restrict our attention to using one back-end at a time.

### *3.3.1 Argobots back-end implementation details*

Argobots is a low-level tasking framework developed to support higher level runtime systems, so it does not provide optimized schedulers for fork-join parallelism out of the box. To implement an optimized tasking framework for our needs, we developed custom scheduling mechanisms using the interfaces provided. For this work we opted for a work-stealing [8] scheduling mechanism even though any other mechanism could be incorporated in the future. In this scheduling mechanism, each thread (execution stream in Argobots' terminology) in the parallel environment is associated with a circular double-ended queue (deque) which is thread-safe and lock-free [12]. Every new task is pushed to the top of the deque of the thread that created it. When a thread finishes with the execution of a task, it first checks its own deque; if there are tasks available, it pops the one residing at the top of the deque and executes it. If its deque is empty, it will randomly pick one of the remaining threads and try to steal the task at the bottom of its deque. By picking the task at the top of the owned deque first, tasks that are hot in the cache are given priority. On the other hand, stealing the task at the bottom of other threads' deques: increases the chance of picking tasks that will create more child tasks, allows more work to become available for the stealing thread and results in a decreased number of steal attempts. We provide two tasking flavors for this implementation - User Level Threads (ULTs) that can yield explicitly and Tasklets that run to completion and can only block waiting for another tasklet created using this framework. In this work, both case studies use tasklets as we only need to wait for other tasks to complete and no other blocking operation is performed. Each task is created using a `abt::task_create` function call that asynchronously schedules a new task and immediately returns a task handle. The task handle can then be used to check or wait for the completion of the respective task's execution. Internally, the call to wait for a task completion will result in calling the scheduler and popping/stealing some other task.

### *3.3.2 TBB back-end implementation details*

Intel®Thread Building Blocks (TBB)[1] is a library that enables parallel programming across different applications and architectures. It provides high level constructs such as `tbb::parallel_for` in addition to giving access to the lower level tasking queues. TBB uses tasks to express parallelism, thus making it a suitable candidate for this study. Tasks are expected to be non-preemptive, which is the case for both applications of this study and for speculative operations in general. The scheduler switches the running thread only when a task is waiting for its spawned children. For the `hierarchical` and the `2level` task creation strategy, each level is enclosed in a `tbb::task_group` that allows to wait until all tasks of the group are completed. When using the lower level `create_and_schedule`, all generated tasks are added to the same global `tbb::task_group` thus allowing termination to be detected in a convenient manner while still enabling work stealing among all threads. Additionally, we implemented a wrapper that passes the arguments of `task_for` directly to

[1] Recently, Intel®Threading Building Blocks was renamed to Intel®oneAPI Threading Building Blocks (oneTBB) to highlight that the tool is part of the oneAPI ecosystem.

the higher level `tbb::parallel_for` function, in order to compare it with our framework.

### *3.3.3 OpenMP back-end implementation details*

OpenMP is an API that enables parallel shared-memory programming with the use of `#pragma`s making it easily accessible directly through the compiler. It is included in this study since it is often the first step towards introducing parallelism for many scientific applications. Tasks are created using `#pragma omp task` and they are declared as `untied` which gives them the opportunity to be scheduled on any available thread. For the `hierarchical` strategy, it was advantageous to prepend `#pragma omp taskyield` right before the recursive step. This created an extra scheduling opportunity for the back-end. Without it, it was noticed that a single thread would tend to run all the tasks it created, affecting performance and greatly increasing the recursion tree size. For comparison, we also implemented two more wrappers using higher level constructs. The first passes the arguments of `task_for` directly to `#pragma omp for` while the second passes to `#pragma omp taskloop`. For the `#pragma omp for`, we chose the `dynamic` scheduler because it performs on average better across the different mesh operations covered in this work.

## 4 Case Studies

As case studies we use the parallel meshing operations present in CDT3D [20,19] and the Delaunay-based kernel of PODM [27]. These two applications share a lot of common ideas when it comes to parallel execution, but they also have some differences. As mentioned in Figure 1, both applications utilize speculative execution for their meshing operations which they implement similarly; the meshing kernel (blue sections) uses atomic locks speculatively to guarantee correctness (green sections). This feature fits well with our approach since we assume that each task should be non-interruptible and should execute to completion. Also, they both integrate load balancing and thread management with the mesh application (red sections), that our framework can abstract away.

PODM is built around a single mesh operation for modifying the mesh, thus reducing the amount of code changes required. On the other hand, when it comes to parallel execution, it has a number of optimizations that complicate the use of the tasking framework. In particular, it uses a Hierarchical Load Balancing that ties worklists to specific threads in order to improve data affinity and takes into account the cost of memory access when moving load between threads. These optimizations result in a tight coupling between the mesh operations and the load balancing parts of the code.

In contrast, in CDT3D the coupling between the threads and their data is lower. At a high level it follows a fork-join pattern where sequential steps prepare global data structures for parallel execution. This structure matches well with the `task_for` API and reduces the places where the code needs to be modified. Moreover, it utilizes a set of different mesh operations that use both hand-optimized and generic work sharing methods, which results in varying impact on performance when transitioning to the tasking framework.

## 4.1 Case Study I : Parallel Mesh Adaptation Software (CDT3D)

CDT3D is composed of many modules depicted in Figure 3. With proper re-arrangement of the modules one could implement different meshing applications as described in [46]. The configuration chosen for this study is optimized for metric-based adaptation and has been already compared against state-of-the-art mesh adaptation codes in [45].

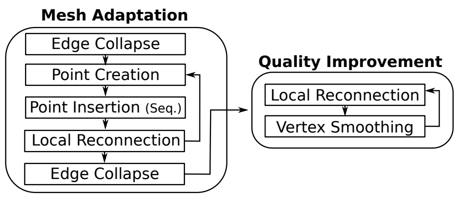


Fig. 3: Mesh operations in CDT3D.

For this case-study, the focus is on: Point Creation, Local Reconnection, Edge Collapse, and Vertex Smoothing. The common first step for porting the operations is to express them in a way that is compatible with the API of the front-end presented in Figure 1. The most natural choice is as an operation applied to an element. However, the baseline implementation of CDT3D already uses "buckets" (i.e, lists of elements) for some of its operations (see, for example, Figure 1b and references [20,19]). In the context of the presented mesh operations, "buckets" are used as simple strip partitioning method similar to the `chunk-size` parameter of the `#pragma omp parallel for` scheduler. In an effort to maximize code re-use of the application, we opted for the conventions of Table 1.

One important feature of the *Operator* in each case, is that it is built using the speculative/optimistic approach. In practice, it means that no data or domain decomposition is applied to the mesh, but the operator will attempt to acquire its dependencies through some exclusive locking mechanism upon execution. Failure to do so will result in unlocking any acquired resources and exiting.

### *4.1.1 Performance Evaluation*

For this evaluation, the code was recompiled picking the appropriate back-end implementation each time. The experiments were performed on the `wahab`

Table 1: Characteristics of the baseline implementation of the parallel mesh operations ported to the tasking framework.

| **Operation** | **Work-unit** | **Operator** | **Baseline implementation** |
|---|---|---|---|
| Local Reconnection | "Bucket" | Apply local reconnection between an element and its face neighbors for each element of a bucket | custom scheduling ([20]) |
| Point Creation | "Bucket" | Generate candidate points for each element of the bucket | custom scheduling ([20]) |
| Edge Collapse | Vertex | Collapse small edges attached to a mesh vertex | `omp for schedule(guided)` |
| Vertex Smoothing | Vertex | Improve the quality of the elements attached to a mesh vertex by smoothing | `omp for schedule(static)` |

cluster of Old Dominion University using dual socket nodes equipped with two Intel®Xeon®Gold 6148 CPU @ 2.40GHz (20 slots) and 368 GB of memory. The compiler is `gcc 7.5.0` and the compiler flags `-O3 -DNDEBUG -march=native`. `gcc 7.5.0` comes with support of OpenMP version `4.5`. For TBB, version `2021.1.1` was used. Each configuration was executed 10 times. All times are normalized based on the performance of the baseline application, unless it is stated otherwise. The graphs below use the geometric mean [23] to summarize the results for each configuration. The evaluation in the following paragraphs proceeds as follows: First, we compare higher-level constructs (`#pragma omp parallel for`, `#pragma omp taskloop` and `tbb::parallel_for`) to the `-flat` strategy implemented using the three back-ends. Next, we compare the `-flat`, `-2level`, and `-hierarchical` strategies as implemented in our framework. We then analyze and optimize the grainsize for each back-end and strategy in order to derive the optimal grainsize for each operation. Finally, we compare our framework using the optimal grainsizes with the baseline application.

*Higher-level parallel constructs and the `flat` model:* For the first benchmark, the `flat` tasking creation model is employed for each back-end and compared against higher-level constructs such as `#pragma omp parallel for`, `#pragma omp taskloop` and `tbb::parallel_for`. As expected, all back-ends exhibit an overhead when using the `flat` strategy compared to higher-level constructs due to the cost of sequentially creating all the tasks. Figure 4 presents the running time normalized with respect to the baseline implementation. `omp-flat` back-end suffers from the highest overhead, especially when more than 20 cores are used, which is the size of the socket for this machine. This trend is in part attributed to the fact that the naive creation of tasks in the `flat` model along with the `untied` specification allows any task to run on any core without any consideration about the affinity of data with respect to the cores.

The higher level constructs perform better than their `-flat` counterparts. `#pragma omp taskloop` improves significantly over `omp-flat` by merging multiple loop iterations into a single task, thus, decreasing the number of tasks that need to be created and scheduled. Moreover, by creating and scheduling fewer tasks, the number of context switches and cache-line invalidations is also reduced. `#pragma omp parallel for` equipped with the `dynamic` scheduler performs even better thanks to the absence of the overhead of task creation. `tbb::parallel_for` outperforms the rest by dynamically adjusting the loop ranges assigned to each task, based on the number of threads, the time for each task execution, and hardware occupancy.

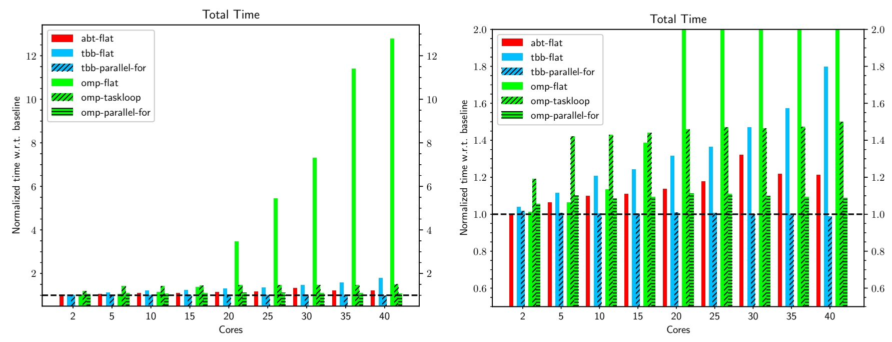


Fig. 4: Left: Normalized total running time of high level constructs and the `flat` model. Right: zoom-in at the range 0.5-2.0.

*Comparison between the `flat`, `2level` and `hierarchical` task creation strategies:* In Figure 5, the three task creation strategies are compared with each other. Both the `2level` and the `hierarchical` strategies reduce significantly the overhead in comparison to the `flat` strategy. In the `2level` strategy, $2 \cdot nthreads$ level-1 tasks that partition the `user_task_args` vector are created sequentially. Then, each level-1 task generates tasks that apply *Operator* to the appropriate unit of work based on Table 1. For this dataset, the `grainsize` is set to 1, which results in creating a level-2 task for each unit of work. Both the `2level` and the `hierarchical` strategies exhibit higher overhead at 2 cores due to the fact that more tasks are created in total. However, this overhead is amortized at a higher number of cores. The dual socket nature of the machine affects the system by a smaller amount, in comparison to the `flat` strategy, with the `omp` back-end suffering from the highest overhead at about 7% on 40 cores. On the other hand, the `abt` and `tbb` back-ends achieve a small improvement when using 40 cores.

The `hierarchical` task creation strategy creates tasks recursively by bisecting the `user_task_args` vector and creating two child tasks each time. The algorithm continues up until the target range reaches the grainsize, which is

1 in this dataset. `abt-hierarchical` and `tbb-hierarchical` exhibit a higher overhead at 2 cores, possibly due to the larger number of generated tasks. For more than 2 cores, the `hierarchical` strategy performs slightly better than the `2level`. This is attributed, in part, to the fact that the `hierarchical` strategy gives more flexibility in scheduling by having many smaller tasks running concurrently (versus the `2level` which combines them in larger ones). This also creates more work-steal opportunities for idle threads, while at the same time avoids the overhead of creating tasks sequentially (contrary to the `flat` strategy). Results of applying the `hierarchical` strategy with a grainsize of 1 using the `omp` back-end are omitted due to their high overhead, which reaches up to a 160x slowdown on 40 threads.

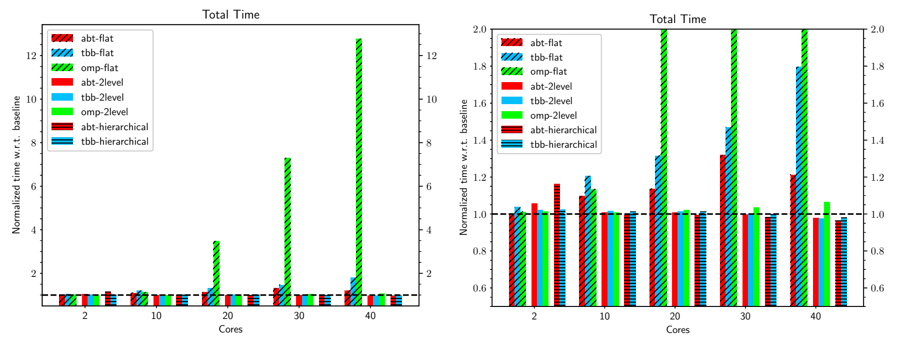


Fig. 5: Left: Normalized total running of the three task creation strategies implemented across the three different back-ends. The `grainsize` is fixed to 1. Right: zoom-in at the range 0.6-2.0.

*Effect of* `grainsize` *for each task creation strategy:* In the next dataset, we demonstrate that the tasking framework in addition contributes towards automating the process of performance tuning. Since the tasking framework uses the same scheduler across the four different operations of this case-study, running the application repeatedly while scanning through a set of different `grainsize` values and the available back-ends, we can obtain optimal values for each operation. The grainsize controls how many applications of *Operator* will be bundled into a single task. In general, creating a high number of tasks (smaller grainsize) gives more flexibility for load balancing by the scheduler. However, a high number of small tasks increases the cost of load balancing. Previous studies on CDT3D [20] revealed a significant dependence of the running time on the number of buckets created during local reconnection. In this study, instead of targeting a fixed number of buckets, we fix the size of each bucket to 150 tetrahedra which was found to be ideal for the baseline application.

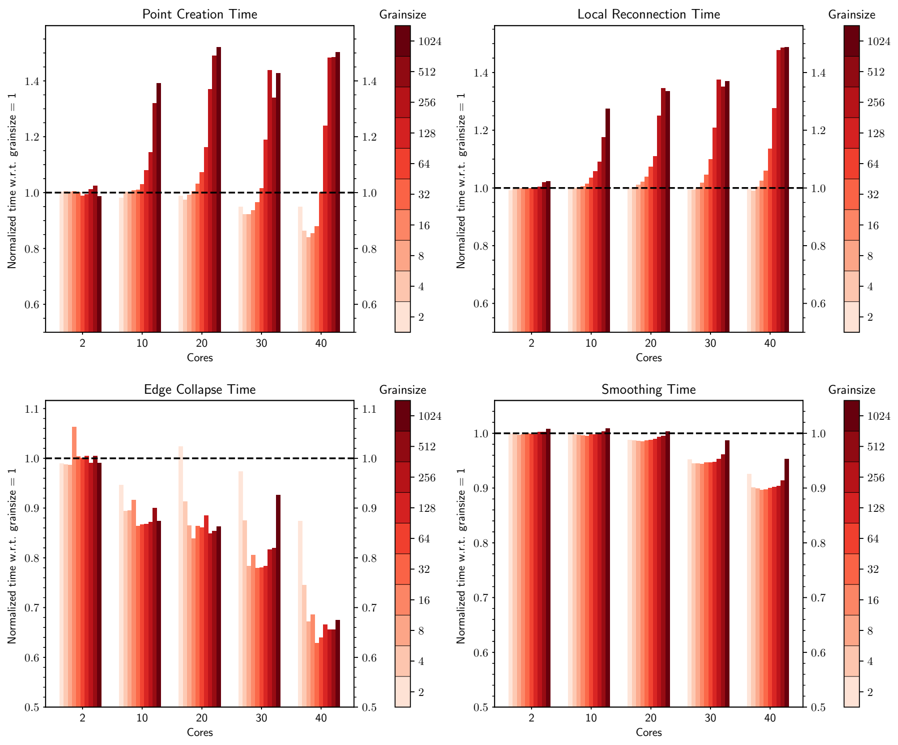


Fig. 6: Effect of grainsize for each operation for `omp-2level`. Times are normalized based on the time taken using grainsize = 1.

Figures 6, 7 and 8 compare the effect of different grainsizes for each operation using the `2level` task creation strategy. The running time in each case is normalized based on the time achieved using a fixed grainsize of 1. Overall, there are similar trends among the different back-ends. Point Creation and Local Reconnection perform better with a smaller grainsize. This is due to the fact that these operations already decompose their data into “buckets” (see Table 1) and each “bucket” offers enough workload to amortize the cost of creating and handling tasks. Using a higher grainsize creates fewer tasks, thus constraining the load balancer and causes a loss in performance. On the other hand, Edge Collapse and Vertex Smoothing, where the *Operator* is designed to accept a single vertex, benefit significantly from increasing the grainsize. In particular, a grainsize of 128 for the Edge Collapse offers more than 30% speedup in comparison to a value of 1 for the `omp` back-end and about 20% for the other two back-ends. The gains for Vertex Smoothing are lower, but they also appear in the middle of the range which we experimented. The same analysis was also performed for the `hierarchical` strategy. `abt-hierarchical` and `tbb-hierarchical` obtain optimal performance for the same grainsize values, while for `omp-hierarchical` the optimal values are 8192 for Edge Collapse and Vertex Smoothing, 32 for Vertex Creation and 1 for Local Reconnection. The graphs for the `hierarchical` strategy are omitted for brevity. It should be noted that a different set of values could be ideal for a different configuration (hardware, meshing problem, etc.).

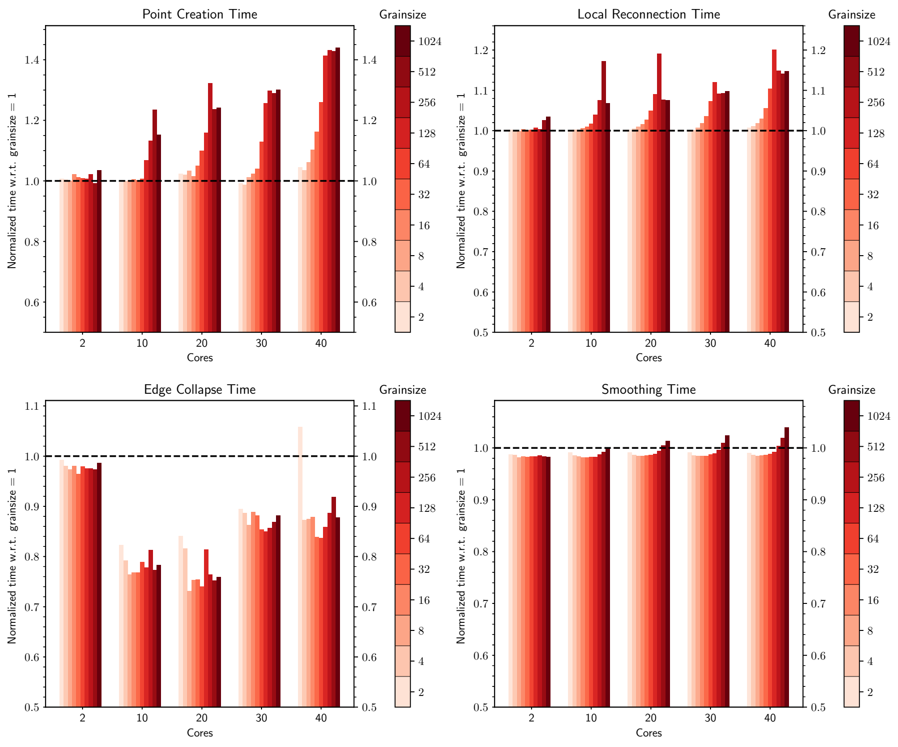


Fig. 7: Effect of grainsize for each operation for `tbb-2level`. Times are normalized based on the time taken using grainsize = 1.

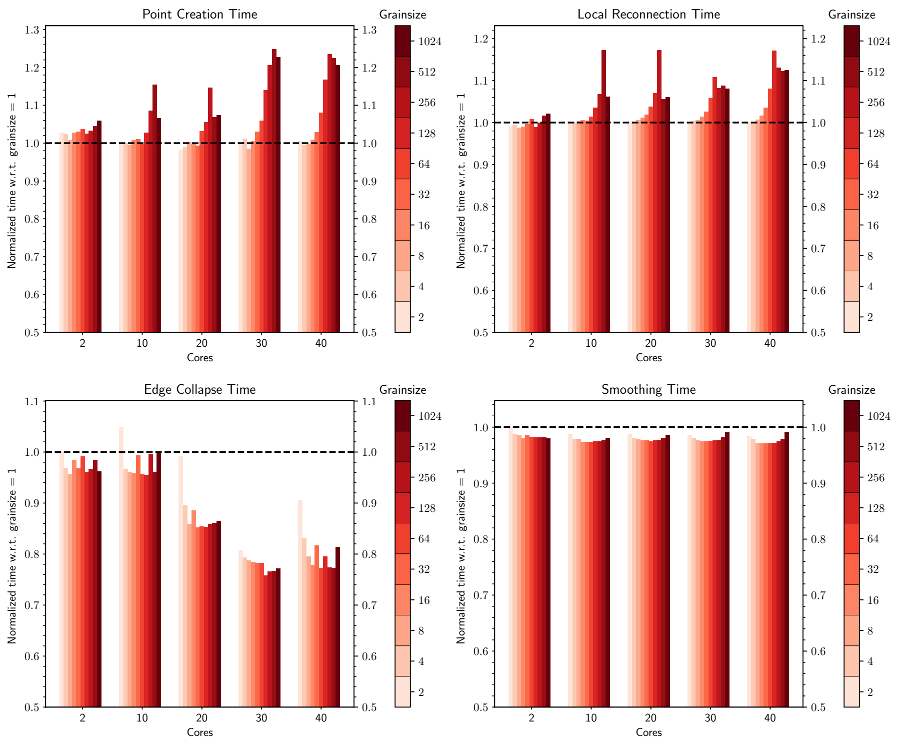


Fig. 8: Effect of grainsize for each operation for `abt-2level`. Times are normalized based on the time taken using grainsize = 1.

*Performance of `2level` and `hierarchical` task creation strategies utilizing optimal `grainsize`:* Finally, we compare the `2level` and `hierarchical` task creation strategies utilizing the three different back-ends and the optimal `grainsize` values derived in the previous paragraph. The performance data indicate significant improvements for some back-ends especially for the least optimized mesh operations (Edge Collapse, Vertex Smoothing). Figure 9 depicts the performance gains replacing the baseline implementation with the tasking framework for each of the operations. The grainsize is set to 1 for the Vertex Creation and Local Reconnection, 128 for Edge Collapse and 64 for Vertex Smoothing when utilizing `*-2level`, `abt-hierarchical` or `tbb-hierachical`. For `omp-hierarchical`, we used the grainsizes mentioned in the previous paragraph. Tables 2 and 3 present the percent (%) improvement over the baseline implementation.

The Point Creation and Local Reconnection operations benefit the least. The difference in performance gains between the two pairs of operations is related to the fact that not all operations use the same back-end in the baseline implementation (see Table 1). In particular, the Point Creation and Local Reconnection operations utilize a custom work-sharing approach described

Table 2: Percent (%) improvement of running time with respect to the baseline implementation for the Point Creation and Local Reconnection operations. Negative numbers signify percent (%) slowdown.

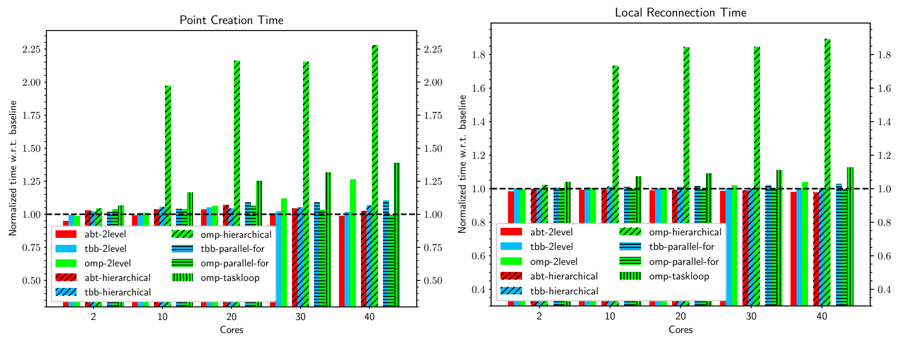
| | Point Creation | | | Local Reconnection | | |
|---|---|---|---|---|---|---|
| | Cores | | | Cores | | |
| | 2 | 10 | 40 | 2 | 10 | 40 |
| abt-2level | 5.03 | 0.82 | 1.47 | 1.45 | 0.79 | 2.01 |
| tbb-2level | 0.68 | -0.85 | -1.69 | -0.13 | -0.66 | -0.35 |
| omp-2level | 1.16 | -0.81 | -26.42 | 0.25 | -0.49 | -4.11 |
| abt-hierarchical | 2.69 | -3.54 | -2.47 | -0.02 | 0.67 | 2.04 |
| tbb-hierarchical | -2.22 | -5.30 | -6.65 | -0.40 | -1.20 | -0.32 |
| omp-hierarchical | -4.24 | -97.11 | -127.64 | -2.19 | -73.31 | -89.30 |
| tbb-parallel-for | -1.58 | -3.79 | -10.29 | -0.59 | -1.07 | -2.78 |
| omp-taskloop | -6.64 | -16.30 | -38.80 | -4.07 | -7.35 | -12.67 |
| omp-parallel-for | -3.39 | -3.65 | 1.01 | -0.22 | -0.27 | 0.53 |

Table 3: Percent (%) improvement of running time with respect to the baseline implementation of the Edge Collapse and Smoothing operations. Negative numbers signify percent (%) slowdown.

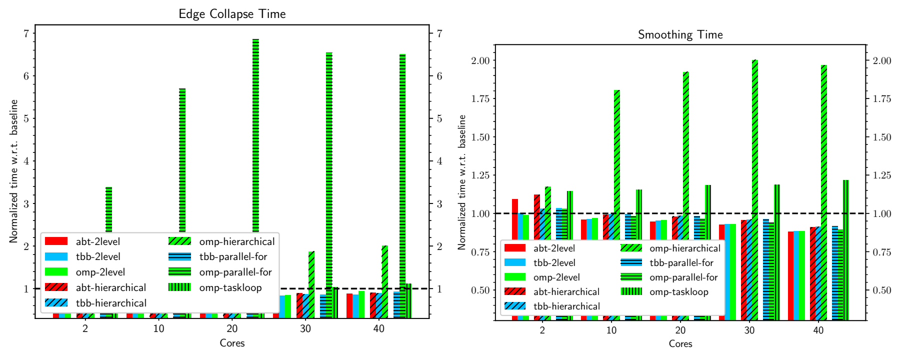
| | Edge Collapse | | | Smoothing | | |
|---|---|---|---|---|---|---|
| | Cores | | | Cores | | |
| | 2 | 10 | 40 | 2 | 10 | 40 |
| abt-2level | -0.46 | 10.88 | 12.05 | -9.18 | 3.98 | 12.04 |
| tbb-2level | 7.74 | 10.60 | 13.42 | -0.07 | 3.68 | 11.60 |
| omp-2level | 8.69 | 8.87 | 5.88 | 0.98 | 3.26 | 11.45 |
| abt-hierarchical | -5.44 | 5.75 | 9.67 | -12.32 | 0.77 | 8.92 |
| tbb-hierarchical | 3.02 | 5.26 | 10.58 | -3.17 | 0.49 | 8.62 |
| omp-hierarchical | -13.43 | -76.16 | -101.07 | -17.42 | -80.47 | -96.60 |
| tbb-parallel-for | 2.28 | 3.86 | 6.86 | -3.31 | 0.54 | 8.49 |
| omp-taskloop | -6.53 | -8.32 | -11.44 | -14.47 | -15.59 | -21.59 |
| omp-parallel-for | -238.23 | -469.65 | -550.96 | -3.16 | 1.89 | 10.67 |

in [20] which has been optimized for the these operations. On the other hand, the Edge Collapse and Vertex Smoothing operation were parallelized using simple OpenMP primitives. Also, the first two operations operate on "buckets" (i.e., lists of elements) instead of single elements, thus introducing *a-priori* data decomposition which may limit the effect of using different scheduling techniques.

`abt-2level` performs the best, offering up to 1.47% and 2.01% improvement on 40 cores for the Point Creation and Local Reconnection operations, respectively. The Edge Collapse and Smoothing operations benefit more. `tbb-2level` performs the best for the Edge Collapse operation delivering more than 13% improvement on 40 cores, while for Smoothing, the best performing is `abt-2level` with up to 12% improvement over the baseline implementation. For comparison, we also append data from the higher-level constructs (`tbb::parallel_for`, `#pragma omp parallel for`, `#pragma omp taskloop`) (i.e., from Figure 4) which should serve as a reference point, since they provide the simplest way to introduce tasks within an application. The higher-level constructs fail to improve the performance for the operations that use custom scheduling, and only some of them deliver small gains for Edge Collapse and Vertex Smoothing. In particular, `tbb::parallel_for` delivers improvements for Edge Collapse and Smoothing and `#pragma omp parallel for` exhibits some gains for Smoothing. However, the gains using the same back-ends within the proposed approach are higher.



(a) Percent (%) Improvement over `baseline` for the Vertex Creation and Local Reconnection.



(b) Percent (%) Improvement over `baseline` for the Edge Collapse and Smoothing Operations.

Fig. 9: Performance improvements over the baseline implementation for the different back-ends using the `2level` and `hierarchical` strategies and optimal grainsizes.

Figure 10 and Table 4 depict the effect of the tasking framework on the total running time of the application while utilizing the optimal grainsize for each back-end and task creation strategy. Overall, `abt-2level` performs the best with up to 5.81% improvement on 40 cores. `tbb-2level` offers a slightly smaller improvement (4.72%) while `omp-2level` adds a small overhead (−0.52%) on 40 cores. Although, the `hierarchical` strategy is able to exploit concurrency at an earlier stage, it does not perform as well as the `2level` strategy. This is attributed, in part, to the fact that the `2level` strategy generates almost half the number of tasks in comparison to the `hierarchical` strategy. In particular, the `2level` strategy generates $2 \cdot nthreads + n/grainsize$ tasks while the hierarchical generates $2^{\log_2(n/grainsize)+1} - 1 = 2(n/grainsize) - 1$ tasks where, $n$ number of work-units passed to `task_for` (i.e., the length of vector `user_task_args` in Listing 1)[2]. Since, in general, $2 \cdot nthreads \ll n$ and $grainsize \ll n$, the `2level` strategy produces about half the number of tasks.

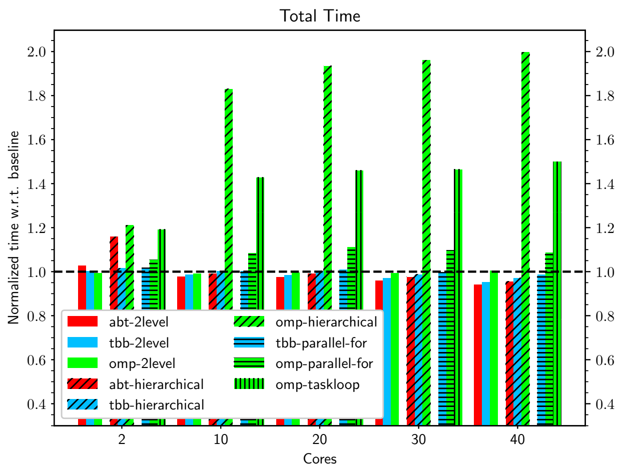


Fig. 10: Total running time of the entire application with optimal grainsizes for each back-end and task creation strategy.

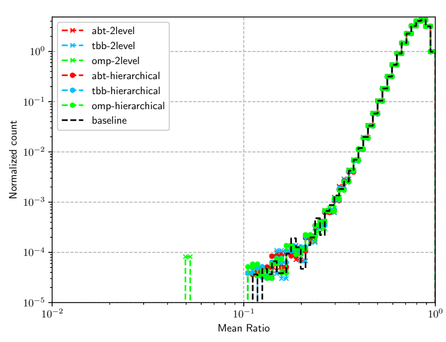
[2] $h = \log_2(n/grainsize)$ is the depth of a perfect binary tree with $n/grainsize$ terminal nodes. $2^{h+1} - 1$ is the number of nodes for a perfect binary tree with depth $h$.

Table 4: Percent (%) improvement of total running time with respect to the baseline implementation. Negative numbers signify percent (%) slowdown.

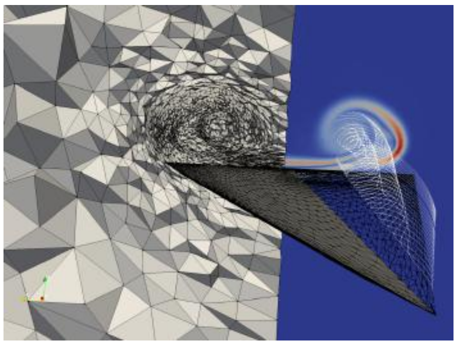
| | Total Time | | |
|---|---|---|---|
| | Cores | | |
| | 2 | 10 | 40 |
| abt-flat | 0.41 | -9.80 | -21.34 |
| tbb-flat | -3.76 | -20.62 | -79.67 |
| omp-flat | -1.21 | -13.46 | -1177.87 |
| | | | |
| abt-2level | -2.81 | 2.31 | 5.81 |
| tbb-2level | -0.31 | 1.25 | 4.72 |
| omp-2level | 0.62 | 0.91 | -0.52 |
| | | | |
| abt-hierarchical | -15.86 | 0.91 | 4.39 |
| tbb-hierarchical | -1.62 | -0.31 | 2.79 |
| omp-hierarchical | -21.0 | -83.01 | -99.71 |
| | | | |
| tbb-parallel-for | -1.81 | -0.30 | 1.40 |
| omp-taskloop | -19.17 | -42.88 | -49.97 |
| omp-parallel-for | -5.48 | -8.57 | -8.73 |

### *4.1.2 Stability of the Tasking Approach*

Among the requirements for a parallel mesh generation code as presented in [45] is the one of *stability* which requires that a mesh generated in parallel has comparable quality with one generated sequentially by the same application. The stability of the baseline application has been already demonstrated in [45]. In Figure 11a, we compare a mesh quality measure among the different back-ends and task creation strategies of the previous section. In particular, the histograms are built using the meshes generated at 40 cores in Figure 9 and averaging the data over the 10 runs of the experiment. Even when using a logarithmic scale, there is no significant difference between the different back-ends with the exception of the `omp-2level` back-end that produced a slightly lower minimum value. Still, the results are within the range ($> 0.01$) produced by other state-of-the-art approaches as presented in [45].



(a)



(b)

Fig. 11: Stability data and visualization of the generated mesh for this use-case. (a): Comparison of the mean-ratio quality metric for the different back-ends. (b): Visualization of the mesh generated by the experiments in this section: Metric-adapted mesh to a laminar flow over a delta wing.

## 4.2 Case Study II: Parallel Optimistic Delaunay Meshing (PODM)

The Parallel Optimistic Delaunay Meshing (PODM) method presented in [27] delivers good parallel performance on DSM machines and high mesh quality along with provable fidelity guarantees. In terms of meshing operations, PODM initializes the meshing procedure with only 6 elements that decompose the bounding box of the input image. The mesh is incrementally refined by inserting points generated based on rules that guarantee the quality and fidelity of the mesh with respect to the input image. For more details, see Figure 1a. The point insertion procedure is built around the Bowyer-Watson kernel [9,47] which introduces new points in the mesh while simultaneously preserving the invariant that after each point insertion, the mesh retains the Delaunay property. There are many ways to decompose the Bowyer-Watson kernel into tasks. In the past, it has been decomposed into *compute data dependencies (cavity)*, *collect data dependencies* and *update connectivity* tasks [14]. In higher dimensions ($> 3$), it is advantageous to decompose the data dependency evaluation (i.e., cavity expansion) into many tasks [26]. Other approaches [25,33], transform the problem of Delaunay Mesh Refinement into two tasks: one of generating the vertices to be added and one that updates the current triangulation by inserting the vertices. In this study, in an effort to keep the problem complexity low and introduce only a small amount of code changes, only two types of tasks will be used; one for scheduling an element and one for refining it.

PODM caries many years of optimization for DSM machines [24]. However, as it happens with highly optimized codes, viewing them from a new perspective may reveal new challenges. The optimizations and design decisions that made PODM very efficient put constraints on the tasking implementation. The most important one, is that threading is managed explicitly by the application and the Load Balancing section of Figure 1a is responsible for populating the work-queue of each thread. In other words, the workload distribution is explicit and tightly integrated with the application.

To overcome this issue, we use the `thread_id` obtained by the threading environment in order to access the appropriate queue in a thread-safe manner. Listing 3 presents a high level pseudocode of the tasking version of PODM. It implements the `flat` model of Figure 2 by decomposing the algorithm of Figure 1a into two tasks. `ScheduleTask` creates tasks for a number of elements from a thread queue. Notice that the task created in line 19 of Listing 3. can run with any `thread_id` which implicitly enables work distribution between different threads. Moreover, each thread will push the newly created elements into its private queue in line 31 of Listing 3. `RefineBadElement` encapsulates the blue section of Figure 1a and it will generate the point to be inserted, calculate and lock its cavity (i.e., data dependencies) and apply the Bowyer-Watson kernel as well as release any acquired locks in the end.

Figure 12 depicts a high level view of the execution flow of the tasking version. Initially, `ScheduleTask` will spawn a task for each of the 6 elements of the initial mesh. Since the initial mesh is very small, only some of the initial 6 tasks will be completed successfully due to rollbacks. In the second round, `ScheduleTask` will create a task for each of the newly created elements, and the process continues until all `thread_Queue`s are empty. Notice that this simple implementation has two major issues: Possibility of livelocks, occurring when two tasks lock themselves in an infinite cycle trying to acquire different parts of overlapping cavities, and the algorithm termination depending on empty thread-local queues. Thread-local queues are accessed based on the `thread_id` acquired from the tasking environment, which is, in general, random. Thus, there is a possibility that a non-empty `thread_Queue` may never get accessed. In these experiments, we didn't notice any of the aforementioned issues, but there is still a chance that they might occur. In a follow-up study, we could integrate our previous work on contention managers [27] that can treat both issues efficiently.

Line 16 of Listing 3 includes a limit on the number of elements to be scheduled at a time. This is necessary since many of the generated tasks will be invalid by the time they run because their corresponding element will have been deleted as part of an operation executed on another cavity. Therefore, scheduling all available elements at once will generate a high number of aborted tasks.

```
1  main()
2  {
3    while(not all thread_Queues are empty){
4      // Launch enough ScheduleTasks to keep all cores
5      // busy
6      for(tid : thread_ids)
7        task::create_and_schedule_task(ScheduleTask);
8    task::wait_for_all();
9    }
10 }
```

```
ScheduleTask()
{
  int tid = task::get_thread_id();
  int scheduled = 0;
  while(scheduled < schedule_limit &&
  !(thread_Queue[tid].empty()))
  {
    el = thread_Queue[tid].pop();
    task::create_and_schedule_task(el,RefineTask);
    scheduled++;
  }
}

RefineTask(el)
{
  int tid = task::get_thread_id();
  success = el.lock_vertices()
  if(success) {
    RefineBadElement(el);
    for(el : newly_created_elements)
      thread_Queue[tid].push(el)
  }
}
```

Listing 3: High level tasking-based pseudocode of PODM.

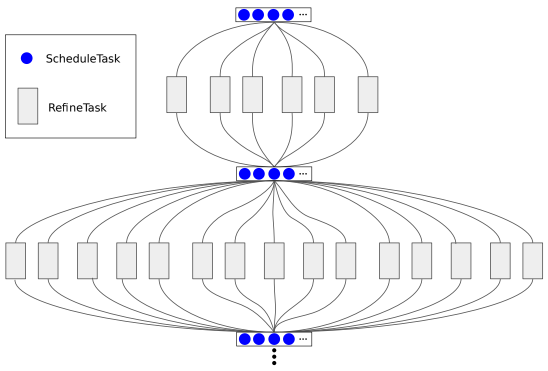


Fig. 12: Flowchart of the tasking version of PODM.

### *4.2.1 Performance Evaluation*

The hardware and compiler configuration is the same as in Section 4.1.1. Figure 13 depicts the effect of the different values of `schedule_limit` to the runtime with respect to the baseline application. Notice that for 1 thread, the ideal value is low. Any limit below 128 performs equally well, while for 40 a value of 512 performs better since it provides the system with more concurrency, albeit at the expense of more aborted tasks.

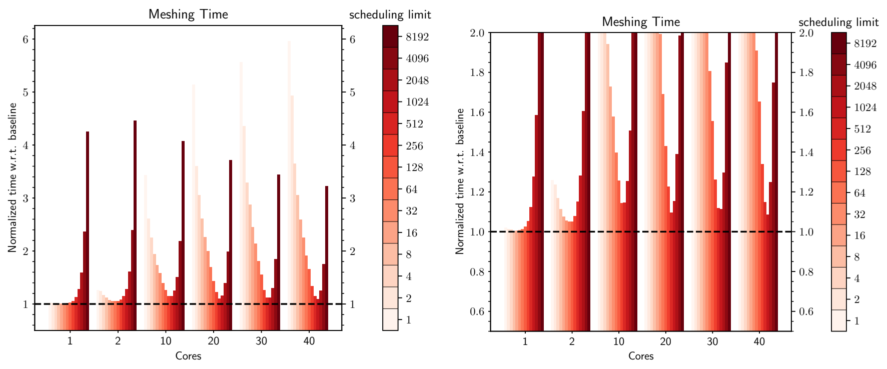


Fig. 13: Effect of scheduling limit for the second approach using the `tbb` back-end. Right zoom-in in range 0.5-2.0

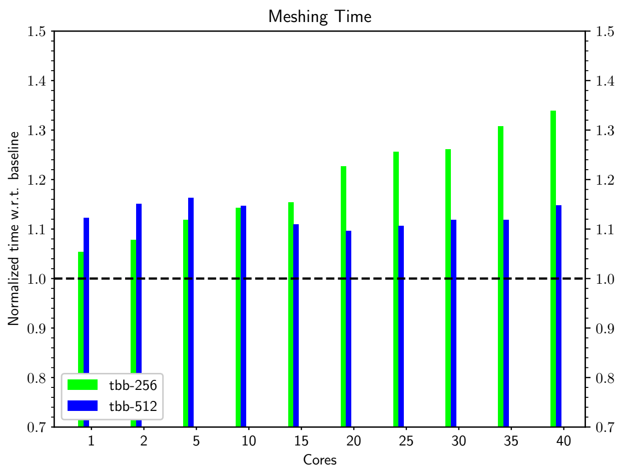


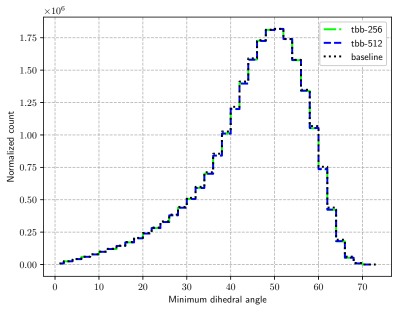
Fig. 14: Normalized meshing time of the tasking version of PODM for two different values of the `schedule_limit`.

Finally, Figure 14 presents the best values among our experiments. The use of the tasking framework adds only between 5% − 14% overhead with respect to the highly optimized baseline application across the different number of cores. Of course, these results come with the shortcomings mentioned above; but based on our previous experience, resolving them should not negatively impact the performance.

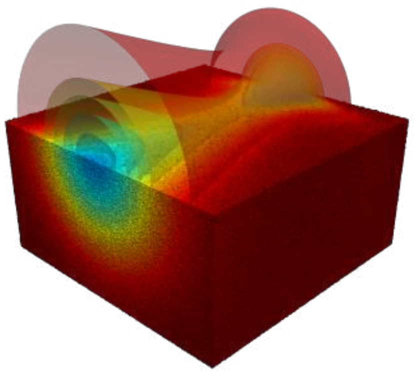
The `abt` and `omp` back-ends exhibit much higher overheads in this case. Scheduling decisions and the internal optimizations of `tbb` could be one of the reasons. Investigating the cause of this overhead and optimizing the `abt` and `omp` back-end implementations of the generalized framework could be investigated in the future.

### *4.2.2 Stability of the Tasking Approach*

Similar to Section 4.1.2, Figure 15a compares a mesh quality measure (minimum dihedral angle in this case) among the different tasking approaches of this case-study. In order to satisfy the *stability* requirement, the quality among the different execution back-ends should be comparable. The histograms of Figure 15a are built using the meshes generated at 40 cores in the previous section and averaging the data over the 10 runs of the experiment. The difference between the tasking approach and the baseline is marginal. The deviation from the baseline is lower that of the previous case-study due to the different meshing method used.



(a)



(b)

Fig. 15: Stability data and visualization of the generated mesh of this use-case. (a): Comparison of the minimum dihedral angle between the different back-ends. (b): Visualization of the generated mesh along with contours of the dataset.

## 5 Conclusion

In this work, we presented a high level tasking framework and applied it to a number of different meshing operations employing a speculative approach. A generic solution like the one presented in this work would be expected to introduce some overhead compared to hand-optimized code written for the specific needs of the application. However, such an overhead is only made visible in the second case study (PODM, Section 4.2), which is affected to a degree by the tighter integration of thread management with the application algorithm. In contrast, the first benchmark (CDT3D, Section 4.1) exhibits performance improvements thanks to the well-optimized scheduling algorithm of the tasking framework. These improvements are significantly higher when compared to the straight-forward use of tasks and higher than higher-level constructs that are already present in some of the back-ends (`#pragma omp parallel for`, `#pragma omp taskloop`, `tbb::parallel_for`). Table 4 indicates that the use of the provided higher-level constructs or the `flat` strategy which corresponds to the straight-forward use of tasks, introduces overheads when compared to the baseline application. The naive `flat` strategy performs the worst between the two since it produces too many tasks and in a sequential fashion. `omp-flat` for example, results in an almost 1200% slowdown. Introducing the `hierarchical` strategy that creates tasks recursively reduces the overhead of creating tasks by distributing it among different threads. This approach offers small gains in low number of cores and up to 4.39% improvement for the total running time when utilizing the Argobots back-end. The best results were obtained with the `2level` strategy, a hybrid of the other two, that manages improvements of up to 2.31% at 10 cores and 5.81% at 40 cores when compared to the baseline application. When it comes to individual operations, Tables 2 and 3 indicate performance improvements of up to 2.04% for the operations that use application-specific scheduler in the baseline implementation and up to 13% for the mesh operations that use generic OpenMP constructs in the baseline application.

Moreover, the abstract front-end gives a platform to explore multiple execution back-ends; Figures 4 and 5 show results over 12 different `strategy-backend` combinations, which are accessible to the application developer through a compile-time parameter. In terms of stability, Sections 4.1.2 and 4.2.2 indicate that the introduction of the tasking framework has no effect on the quality of the generated data.

Finally, separating the concerns of *functionality* and *performance* is a crucial step towards the implementation of the *Telescopic Approach* [15], which lays down a design for achieving scalability for mesh generation on exascale machines. The generalized tasking framework facilitates the integration with the PREMA runtime system [41] at the shared memory level by handing control of thread management and load balancing from the application to the runtime system. This decoupling is expected to speedup the implementation due to the improved encapsulation of the different methods and PREMA's more efficient management of hardware resources. For example, the application independent tasking pools that the tasking framework offers can provide load balancing across different instances of the same application occupying a common shared memory space. This scenario fits well with our previous work [22] and the Parallel Data Refinement layer of the *Telescopic Approach*.

The separation of concerns with regard to functionality and performance allows to create reusable modules for future applications. By extracting good solutions from previous implementations, we can reduce the implementation effort of future revisions of our applications but also provide templates for developing new speculative methods.

## 6 Future Work

Both case studies introduce a parameter (*grainsize* for CDT3D and *schedule_limit* for PODM) that controls the number of tasks created. In this study, we determined the optimal values by scanning over a predefined range. However, a more thorough study that includes different meshing inputs and parameters is needed in order to identify a set of optimal values that exhibits the best performance on average across different inputs. As with any parameter optimization study, this could be performed utilizing machine learning on pre-generated data. Moreover, the underlying tasking framework could be equipped with an online machine learning method [29] that can choose the optimal parameters based on runtime data. Based on previous experience [3], we believe that an online machine learning method is superior to formal load balancing descriptions which tend to oversimplify the problem when it comes to the effect of different hardware and problem configurations to the application. On the other hand, an online machine learning framework could be trained on multiple datasets of interest and reinforce its models based on runtime data.

Utilizing tasks in conjunction with the speculative approach could be further improved by abstracting the Parallel Correctness sections of Figure 1. This could be achieved with high level (but still application-specific) abstractions that lock and unlock the cavity of an operation automatically. The addition of contention managers and the ability to automatically reschedule aborted speculative tasks will make this framework more complete and improve the encapsulation by providing to the user a tasking framework that is able to decouple all three sections of Figure 1. Moreover, the decoupling of the three responsibilities will allow to refactor both our case-studies and extract a larger number of abstract parameters that control scheduling decisions. Studying the effect of these additional parameters is part of our future work.

The back-end of the `task_for` front-end of Listing 1 is currently a compile-time parameter. However, there is no technical constrain that would prevent it from being an extra argument of the front-end API. Making the back-end a parameter will enable interoperability among the different back-ends based on the needs of the user.

Taking into account the memory affinity layout when scheduling tasks could also improve the framework's performance. As part of our future work for the Argobots back-end, we intend to examine this aspect in more detail. Specifically, the back-end will utilize the Linux `numa` library to collect memory related information about the underlying hardware. This information will then be used by the back-end to coordinate task stealing attempts between threads, in a hierarchical way.

Throughout this study the default load balancing/work scheduler of each back-end was used. This work could be extended by evaluating the available work schedulers of OpenMP (`static, dynamic, guided`) and implementing *breadth-first* and *work-first* strategies [21] for the task-based methods. Moreover, we could customize the `tbb` scheduler and extend our Argobots back-end with custom load balancing methods. Having a variety of scheduling methods for each back-end will allow to produce a similar study based on the effect of the application-specific scheduling algorithms of our two case-studies versus generic load balancing methods.

In a follow-up study, we plan to explore more back-end systems that can utilize both homogeneous and heterogeneous platforms, including GPUs. Generic heterogeneous frameworks such as SYCL [3] and Kokkos [4] provide already support for launching and managing tasks on GPUs. Combining them with the tasking framework is expected to assist in hiding latencies related to data transfers to and from the device, as well as delays launching kernels. Evaluating such a framework would also require the addition or extension of current mesh operations for heterogeneous architectures.

**Acknowledgements** We would like to thank the reviewers for providing helpful comments on earlier drafts of the manuscript. This research was sponsored in part by the NASA Transformational Tools and Technologies Project (NNX15AU39A) of the Transformative Aeronautics Concepts Program under the Aeronautics Research Mission Directorate, NSF grant no. CCF-1439079, the Richard T. Cheng Endowment, the Modeling and Simulation fellowship of Old Dominion University and the Dominion Scholar fellowship of Old Dominion University. Experiments were supported by the Research Computing clusters at Old Dominion University. The authors would like to thank Kevin Garner for the corrections of the English text in the manuscript.

## References

1. Aldea, S., Estebanez, A., Llanos, D.R., Gonzalez-Escribano, A.: An OpenMP extension that supports thread-level speculation. IEEE Transactions on Parallel and Distributed Systems **27**(1), 78–91 (2016). DOI 10.1109/TPDS.2015.2393870
2. Antonopoulos, C.D., Ding, X., Chernikov, A., Blagojevic, F., Nikolopoulos, D.S., Chrisochoides, N.: Multigrain Parallel Delaunay Mesh Generation: Challenges and Opportunities for Multithreaded Architectures. In: Proceedings of the 19th Annual International Conference on Supercomputing, ICS '05, pp. 367–376. ACM, New York, NY, USA (2005). DOI 10.1145/1088149.1088198
3. Barker, K., Chrisochoides, N.: Practical Performance Model for Optimizing Dynamic Load Balancing of Adaptive Applications. IEEE (2005). DOI 10.1109/IPDPS.2005.352
4. Batista, V.H.F., Millman, D.L., Pion, S., Singler, J.: Parallel geometric algorithms for multi-core computers. Computational Geometry **43**(8), 663–677 (2010). DOI 10.1016/j.comgeo.2010.04.008
5. Blandford, D.K., Blelloch, G.E., Kadow, C.: Engineering a Compact Parallel Delaunay Algorithm in 3D. In: Proceedings of the Twenty-second Annual Symposium on Computational Geometry, SCG '06, pp. 292–300. ACM, New York, NY, USA (2006). DOI 10.1145/1137856.1137900
6. Blelloch, G.E., Anderson, D., Dhulipala, L.: ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In: Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '20, pp. 507–509. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3350755.3400254
7. Blelloch, G.E., Fineman, J.T., Gibbons, P.B., Shun, J.: Internally deterministic parallel algorithms can be fast. In: Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPoPP '12, pp. 181–192. Association for Computing Machinery, New York, NY, USA (2012). DOI 10.1145/2145816.2145840

[3] `https://www.khronos.org/sycl` (Accessed 27th April 2021)

[4] `https://kokkos.org` (Accessed 27th April 2021)

8. Blumofe, R.D., Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing. Journal of the ACM **46**(5), 720–748 (1999). DOI 10.1145/324133.324234
9. Bowyer, A.: Computing Dirichlet tessellations. The Computer Journal **24**(2), 162–166 (1981). DOI 10.1093/comjnl/24.2.162
10. Bramas, B.: Increasing the degree of parallelism using speculative execution in task-based runtime systems. PeerJ Computer Science **5**, e183 (2019). DOI 10.7717/peerj-cs.183. Publisher: PeerJ Inc.
11. Caamaño, J.M.M., Sukumaran-Rajam, A., Baloian, A., Selva, M., Clauss, P.: APOLLO: Automatic speculative POLyhedral Loop Optimizer. In: IMPACT 2017 - 7th International Workshop on Polyhedral Compilation Techniques, p. 8. Stockholm, Sweden (2017)
12. Chase, D., Lev, Y.: Dynamic circular work-stealing deque. In: Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '05, p. 21–28. Association for Computing Machinery, New York, NY, USA (2005). DOI 10.1145/1073970.1073974
13. Chi, Y., Guo, L., Choi, Y.k., Wang, J., Cong, J.: Extending high-level synthesis for task-parallel programs. In: The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '21, p. 225. Association for Computing Machinery, New York, NY, USA (2021). DOI 10.1145/3431920.3439470
14. Chrisochoides, N., Sukup, F.: Task Parallel Implementation of the Bowyer-Watson Algorithm. In: Proceedings of Fifth International Conference on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields, pp. 773–782 (1996)
15. Chrisochoides, N.P.: Telescopic Approach for Extreme-Scale Parallel Mesh Generation for CFD Applications. In: 46th AIAA Fluid Dynamics Conference. American Institute of Aeronautics and Astronautics (2016). DOI 10.2514/6.2016-3181
16. Conway, M.E.: A multiprocessor system design. In: Proceedings of the November 12-14, 1963, fall joint computer conference, AFIPS '63 (Fall), pp. 139–146. Association for Computing Machinery, New York, NY, USA (1963). DOI 10.1145/1463822.1463838
17. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Computational Science and Engineering **5**(1), 46–55 (1998). DOI 10.1109/99.660313. Conference Name: IEEE Computational Science and Engineering
18. Dijkstra, E.W.: On the role of scientific thought. In: Selected writings on computing: a personal perspective, pp. 60–66. Springer-Verlag, Berlin, Heidelberg (1982)
19. Drakopoulos, F.: Finite Element Modeling Driven by Health Care and Aerospace Applications. Ph.D. thesis, Computer Science, Old Dominion University, Virginia (2017). DOI 10.25777/p9kt-9c56. ISBN: 9780355362169
20. Drakopoulos, F., Tsolakis, C., Chrisochoides, N.P.: Fine-Grained Speculative Topological Transformation Scheme for Local Reconnection Methods. AIAA Journal **57**(9), 4007–4018 (2019). DOI 10.2514/1.J057657. Publisher: American Institute of Aeronautics and Astronautics
21. Duran, A., Corbalán, J., Ayguadé, E.: Evaluation of OpenMP Task Scheduling Strategies. In: D. Hutchison, T. Kanade, J. Kittler, J.M. Kleinberg, F. Mattern, J.C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M.Y. Vardi, G. Weikum, R. Eigenmann, B.R. de Supinski (eds.) OpenMP in a New Era of Parallelism, vol. 5004, pp. 100–110. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). DOI 10.1007/978-3-540-79561-2_9. Series Title: Lecture Notes in Computer Science
22. Feng, D., Tsolakis, C., Chernikov, A.N., Chrisochoides, N.P.: Scalable 3D Hybrid Parallel Delaunay Image-to-mesh Conversion Algorithm for Distributed Shared Memory Architectures. Comput. Aided Des. **85**(C), 10–19 (2017). DOI 10.1016/j.cad.2016.07.010
23. Fleming, P.J., Wallace, J.J.: How not to lie with statistics: the correct way to summarize benchmark results. Communications of the ACM **29**(3), 218–221 (1986). DOI 10.1145/5666.5673
24. Foteinos, P.: Real-Time High-Quality Image to Mesh Conversion for Finite Element Simulations. Ph.D., The College of William and Mary, United States – Virginia (2013)
25. Foteinos, P., Chrisochoides, N.: Dynamic Parallel 3D Delaunay Triangulation. In: W.R. Quadros (ed.) Proceedings of the 20th International Meshing Roundtable, pp. 3–20. Springer Berlin Heidelberg (2011). DOI 10.1007/978-3-642-24734-7_1
26. Foteinos, P., Chrisochoides, N.: 4D space–time Delaunay meshing for medical images. Engineering with Computers **31**(3), 499–511 (2014). DOI 10.1007/s00366-014-0380-z

27. Foteinos, P.A., Chrisochoides, N.P.: High quality real-time Image-to-Mesh conversion for finite element simulations. Journal of Parallel and Distributed Computing **74**(2), 2123–2140 (2014). DOI 10.1016/j.jpdc.2013.11.002
28. Furrer, F.J.: Future-Proof Software-Systems: A Sustainable Evolution Strategy. Springer Vieweg (2019). DOI 10.1007/978-3-658-19938-8
29. Hoi, S.C.H., Sahoo, D., Lu, J., Zhao, P.: Online Learning: A Comprehensive Survey. arXiv:1802.02871 [cs] (2018). ArXiv: 1802.02871
30. Jefferson, D.R.: Virtual time. ACM Transactions on Programming Languages and Systems **7**(3), 404–425 (1985). DOI 10.1145/3916.3988
31. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, pp. 211–222. Association for Computing Machinery, New York, NY, USA (2007). DOI 10.1145/1250734.1250759
32. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Transactions on Database Systems **6**(2), 213–226 (1981). DOI 10.1145/319566.319567
33. Marot, C., Pellerin, J., Remacle, J.F.: One machine, one minute, three billion tetrahedra. International Journal for Numerical Methods in Engineering **117**(9), 967–990 (2019). DOI 10.1002/nme.5987
34. Nave, D., Nikos Chrisochoides, Chew, L.P.: Guaranteed: Quality Parallel Delaunay Refinement for Restricted Polyhedral Domains. In: Proceedings of the Eighteenth Annual Symposium on Computational Geometry, SCG '02, pp. 135–144. ACM, New York, NY, USA (2002). DOI 10.1145/513400.513418
35. Rainey, M., Newton, R.R., Hale, K., Hardavellas, N., Campanoni, S., Dinda, P., Acar, U.A.: Task parallel assembly language for uncompromising parallelism. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, p. 1064–1079. Association for Computing Machinery, New York, NY, USA (2021). DOI 10.1145/3453483.3460969
36. Raman, A., Kim, H., Mason, T.R., Jablin, T.B., August, D.I.: Speculative parallelization using software multi-threaded transactions. In: Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems, ASPLOS XV, pp. 65–76. Association for Computing Machinery, New York, NY, USA (2010). DOI 10.1145/1736020.1736030
37. Rauchwerger, L., Padua, D.: The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. ACM SIGPLAN Notices **30**(6), 218–232 (1995). DOI 10.1145/223428.207148
38. Saltz, J., Mirchandaney, R., Crowley, K.: Run-time parallelization and scheduling of loops. IEEE Transactions on Computers **40**(5), 603–612 (1991). DOI 10.1109/12.88484. Conference Name: IEEE Transactions on Computers
39. Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., Carns, P., Castelló, A., Genet, D., Herault, T., Iwasaki, S., Jindal, P., Kalé, L.V., Krishnamoorthy, S., Lifflander, J., Lu, H., Meneses, E., Snir, M., Sun, Y., Taura, K., Beckman, P.: Argobots: A Lightweight Low-Level Threading and Tasking Framework. IEEE Transactions on Parallel and Distributed Systems **29**(3), 512–526 (2018). DOI 10.1109/TPDS.2017.2766062
40. Steele, G.L.: Making asynchronous parallelism safe for the world. In: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '90, pp. 218–231. Association for Computing Machinery, New York, NY, USA (1989). DOI 10.1145/96709.96731
41. Thomadakis, P., Tsolakis, C., Chrisochoides, N.: Multithreaded runtime framework for parallel and adaptive applications. IEEE Transactions on Parallel and Distributed Systems. (2021). URL `https://crtc.cs.odu.edu/pub/papers/journal_86.pdf`. (under review)
42. Thoman, P., Dichev, K., Heller, T., Iakymchuk, R., Aguilar, X., Hasanov, K., Gschwandtner, P., Lemarinier, P., Markidis, S., Jordan, H., Fahringer, T., Katrinis, K., Laure, E., Nikolopoulos, D.S.: A taxonomy of task-based parallel programming technologies for high-performance computing. The Journal of Supercomputing **74**(4), 1422–1434 (2018). DOI 10.1007/s11227-018-2238-4

43. Tomasulo, R.M.: An Efficient Algorithm for Exploiting Multiple Arithmetic Units. IBM Journal of Research and Development **11**(1), 25–33 (1967). DOI 10.1147/rd.111.0025. Conference Name: IBM Journal of Research and Development
44. Tsolakis, C., Chrisochoides, N., Park, M.A., Loseille, A., Michal, T.R.: Parallel Anisotropic Unstructured Grid Adaptation. In: AIAA Scitech 2019 Forum, AIAA SciTech Forum. American Institute of Aeronautics and Astronautics, San Diego, California (2019). DOI 10.2514/6.2019-1995
45. Tsolakis, C., Chrisochoides, N., Park, M.A., Loseille, A., Michal, T.R.: Parallel Anisotropic Unstructured Grid Adaptation. AIAA Journal (2021). DOI 10.2514/1.J060270
46. Tsolakis, C., Thomadakis, P., Chrisochoides, N.: Exascale-Era Parallel Adaptive Mesh Generation and Runtime Software System Activities at the Center for Real-Time Computing (2020). URL `https://epcced.github.io/ELEMENT/workshops.html`. (presentation), Accessed on 2021-03-08
47. Watson, D.F.: Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. The Computer Journal **24**(2), 167–172 (1981). DOI 10.1093/comjnl/24.2.167
48. Willhalm, T., Popovici, N.: Putting Intel® threading building blocks to work. In: Proceedings of the 1st international workshop on Multicore software engineering, IWMSE '08, pp. 3–4. Association for Computing Machinery, New York, NY, USA (2008). DOI 10.1145/1370082.1370085
49. Ying, V.A., Jeffrey, M.C., Sanchez, D.: T4: Compiling sequential code for effective speculative parallelization in hardware. In: Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture, ISCA '20, p. 159–172. IEEE Press (2020). DOI 10.1109/ISCA45697.2020.00024