Towards Runtime Support for Unstructured and Dynamic Exascale-era Applications

Polykarpos Thomadakis[1*] and Nikos Chrisochoides[1]

[1]Department of Computer Science, Old Dominion University, Norfolk, 23529, Virginia, USA.

*Corresponding author(s). E-mail(s): pthom001@odu.edu;
Contributing authors: nikos@cs.odu.edu;

Abstract

This paper presents an effort to mitigate overheads, latencies and limitations observed in message-driven runtime frameworks, by utilizing lightweight threads tightly integrated with message-passing. It also introduces new abstractions and features for group communication as well as fine-grained concurrency on top of remote method invocations to improve workload balancing in shared and distributed memory. We observe up to 100% difference in performance behavior for task creation and handling message-passing. Evaluations on 1000 cores (25 nodes) of a distributed memory machine showed that the integration of fine-grained concurrency with the runtime achieves performance improvements of 12% on a seismic wave simulation benchmark, as opposed to 50% degradation with OpenMP. Moreover, a 3D mesh refinement application showed 50% improvement, exploiting multi-grain parallelism at data and task level.

1 Introduction

Developing efficient applications for modern, large scale, heterogeneous computing clusters requires a lot of effort and expertise in both the application and the systems domain. The required expertise becomes even more relevant for unstructured applications due to their sparse, data-intensive, and dynamic (or adaptive) nature. The workflow of such applications is not statically predictable and heavily depends on the specific given input; thus, raising

the complexity of scheduling and load balancing algorithms required to efficiently utilize modern platforms. Our solution to simplify the development of efficient adaptive and irregular applications is the introduction of a high-level Domain Specific Language (DSL). The DSL hides the additional effort required for maintaining correctness in the context of concurrency as well as the idiosyncrasies of low-level hardware, and transparently scales applications to large computing clusters. It includes selective domain-specific language constructs, a compilation toolchain, and a runtime framework that provides tasking, scheduling, and load balancing across and within tightly integrated heterogeneous nodes. In this work, we focus on the runtime framework of the DSL, namely the Parallel Runtime Environment for Multicore Applications (PREMA) [1, 2].

An effective runtime should address the following fundamental issues in parallel computing: a) global namespace, b) scheduling and load balancing, c) latency hiding, d) fault resilience, e) heterogeneity and performance portability. Currently, PREMA addresses the former three while work is in progress for the latter two. In the process of designing and implementing high-level DSL constructs on top of PREMA, we realized some of its limitations, also found in other similar systems, that inhibit its performance and ease of use. Some of these limitations include: 1) requiring remote method invocations (Active Messages) or tasks to run to completion to avoid delaying the progress engine, 2) inability to preempt task execution, 3) limited high-level constructs to check remote task completion and task dependencies, and 4) inability to balance a coarse-grained work unit's load once it has been scheduled.

Restricting applications to use run-to-completion tasks is a common requirement among shared and distributed memory runtime systems, like PREMA. Tasks are usually expected not to synchronize with each other, except from "parent" tasks synchronizing with their "children", and should not be involved in a (busy)-waiting routine, e.g., waiting for an MPI message reception or for an interrupt to fire. Moreover, such systems lack the ability for tasks to voluntarily release control of a CPU thread, when busy-waiting is unavoidable, in order to allow other tasks to execute. These restrictions are put in place in an effort to prevent tasks from delaying the execution flow or even causing deadlocks; however, they increase the complexity of developing applications on top of such systems, especially when porting legacy codes. In section 3, we present an attempt to loosen these constraints by introducing the *wait_until()* construct that allows PREMA to preempt a task execution until a given condition is satisfied. We show that this feature can quickly lead to live-lock scenarios when using tasks that do not use dedicated stacks and cannot context switch.

Unstructured and adaptive applications are usually hard to scale across multiple computing cores and nodes due to the difficulty in statically dividing them into uniform chunks of work. One way to handle the irregular workload of such applications is to apply dynamic load balancing in an attempt to fix workload imbalance when it arises at run-time. PREMA implements implicit

load balancing through a uniform object-oriented, message-driven execution model (see section 1.1) that virtualizes memory spaces and processing elements, and allows transparent data and workload migrations. Applications are designed as if those objects are located in independent process that do not share common hardware and can only interact with each other through messages that trigger a function execution (called handlers). Internally, the objects share the resources of a computing node, the work units of an object (received handlers) can be executed, if possible in parallel, by any idle thread in the node, and objects, along with their workload, can be migrated among nodes. Thus, workload can be redistributed both in distributed and shared memory, while using a uniform interface to create work units (handlers). Even though this approach is elegant, it can be sub-optimal when the there is a large disparity among handlers' workload since a handler is a single unit of work that cannot be shared among threads.

In this paper, we present an effort to alleviate the issues presented above. We show that by integrating PREMA with lightweight threads (in our case, Argobots [3]) one can easily avoid the limitations imposed by using run-to-completion tasks, substantially improving end-user productivity without significant cost in performance. Moreover, by utilizing the capabilities of this new integration, we implement high-level group-commnuication primitives that further increase ease-of-use. Finally, to mitigate the load imbalance imposed in cases of large workload disparity between handlers, we diverge from the uniform message-driven execution model and integrate PREMA with a tasking framework that enables applications to express handlers as a set of partially independent tasklets. We show that by sacrificing a little ease-of-use to utilize a hybrid execution model, runtime systems can achieve significant improvements. Specifically, by compartmentalizing handlers, PREMA is able to redistribute workload outliers among the idle cores of a computing node.

1.1 Parallel Runtime Environment for Multicore Applications

The Parallel Runtime Environment for Multicore Applications (PREMA) is a system that provides runtime support for applications targeting large-scale computing clusters. PREMA allows for seamless and efficient utilization of the available computing power of a computing platform, both in shared and distributed memory, by offering 2-level parallelism that utilizes the Message Passing Interface (MPI) for inter-node communication and Pthreads for intra-node coordination. Its ultimate goal is to alleviate applications from the burden of dealing with work scheduling, load balancing, and overlapping the communication overhead with computations with minimum involvement of the user. PREMA introduces the abstraction of mobile objects, a globally addressable, location-independent container implemented by the runtime system to store application data. Mobile objects are indented to represent semi-isolated, coarse-grained data, similar to these belonging to an MPI rank after appropriately over-decomposing [4] the application domain; however, users are free

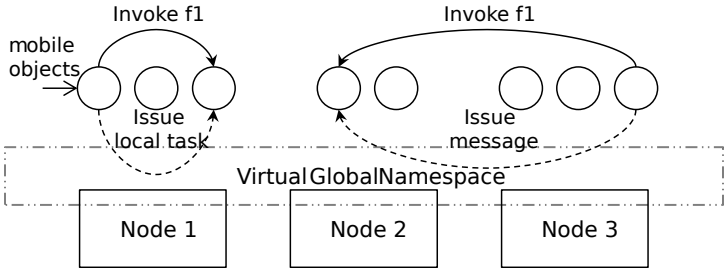

Fig. 1 PREMA's mobile object driven (MOD) model. Applications are expressed as method invocations between local or remote mobile objects. A virtual global namespace is used to provide a uniform high-level interface to issue local or remote work. The runtime system is responsible to run a task locally or send an active message depending on the location of the target mobile object.

to encapsulate any type or size of desired data. Utilizing the abstraction of mobile objects, PREMA introduces a mobile object-driven (MOD) programming model where interactions are expressed as (remote) method invocations (called handlers in PREMA) between mobile objects rather than processes or threads. Handlers can be invoked on mobile objects uniformly, regardless of whether their data are local or remote. An example of the MOD model is shown in Figure 1.

By utilizing the MOD programming model and associating remote handlers with access privileges, an application is able to transparently run on multi-core distributed platforms without explicitly handling concurrency. PREMA is able to extract shared-memory parallelism by running non-conflicting handlers concurrently and allowing threads to share their workload. On the distributed memory level, it can migrate mobile objects between different computing nodes in order to provide distributed-memory load balancing. Thus, the hardware memory spaces and processing elements are virtualized, allowing inter-handler parallelism (multiple handlers running in parallel) and sharing mobile object workload for threads in the same node. To increase flexibility, the framework exposes a simple and isolated module that allows easy experimentation/development of new 2-level load balancing/scheduling policies without affecting the application code.

1.2 Contributions

This paper presents an effort to address tasking and communication challenges related to message-driven runtime frameworks, like PREMA, by using lightweight threads tightly integrated with message-passing. We use Argobots to demonstrate a number of optimizations, including lightweight threads with distinct stack, capable of preemption. We also use a high-level tasking framework, introduced in in [5], and present a detailed overview of its design and implementation on top of Argobots. In this context, the tasking framework is used to facilitate even load distribution by employing multi-level task-parallelism. In summary, this work:

- provides the means for efficient task creation and message-handling, using low-latency, preemptable, lightweight threads as opposed to run-to-completion tasks with the purpose of mitigating overheads and latencies stemming from blocking operations
- encourages the utilization of multiple levels of data and task over-decomposition to improve the quality of workload distribution by increasing the flexibility of the runtime system to dynamically assign work units across both shared and distributed memory
- facilitates ease-of-use in the context of message-driven, global address-space and data migrations, introducing high-level group communication constructs
- evaluates methods for integrating remote method invocations and message-handling, where we observe up to 100% difference in performance behavior, and presents significant improvement of 50% through exhibiting multi-grain task and data over-decomposition.

This work has a broad impact on scientific computing use cases; some examples of applications that use PREMA and can leverage from its new features include mesh generation applications ([6], section 6.7), solvers (section 6.6) , N-body simulations [7], as well as other applications of irregular and adaptive computation and communication nature. Subsequently, it can be used as the backend of a domain specific language for unstructured and dynamic applications.

2 Related Work

2.1 Lightweight Threading Systems

Some of the most popular and highly optimized general-purpose systems include MassiveThreads [8], QThreads [9], StackThreads/MP [10], Cilk [11], Intel Threading Building Blocks [12], and Argobots [3]. All of these libraries offer large-scale lightweight thread support, allowing a large number of limited-resource threads to coexist in a system, managed by a set of heavy-weight OS threads. Threads generated from these libraries can block and wait for other threads to complete and can also migrate between OS threads before completing their execution. QThreads, StackThreads/MP, MassiveThreads and Argobots offer (in different extends) sets of synchronization primitives (mutexes, conditional variable, etc.) that allow OS threads to context-switch between lightweight threads instead of blocking when the waiting condition is not immediately satisfied. StackThreads/MP and Cilk require compiler support while the rest of the systems are provided as C language libraries which makes them much easier to use in applications and higher-level runtimes. MassiveThreads and Argobots are enhanced with additional support for efficient interaction with with I/O operations. Threads created by the two systems are able to automatically detect blocking IO calls and context switch at user-level, simplifying the interaction between computation-heavy and IO-heavy tasks in a single threading model. Finally, while all other libraries

adopt a predefined scheduling policy, i.e. work-first LIFO scheduling within a single OS thread and FIFO randomized work-stealing between OS threads, Argobots does not restrict users to a specific policy and provides the tools to implement their own.

Based on our review, Argobots was chosen as the best option for a low-level threading system, since it achieves high performance [3, 5], and incorporates the features provided by all the other systems. It also exposes abstractions that allow low-level customization and optimization by the user while maintaining a portable and broadly applicable interface. Other libraries facilitate higher usability, but this comes at the cost of less flexibility and lack of low-level control. For instance, all other libraries implement transparent scheduling decisions, hide work pools and provide no control over stack and context-switching. Having access to these implementation decisions and being able to customize them plays a crucial role in optimizing a complicated runtime system such as PREMA to achieve high performance. Argobots allows that while providing two types of tasks to optimize performance, exposing an explicit task-yield operation, and integrating with MPI and power management systems.

2.2 Distributed Memory Systems

Distributed memory runtime systems have been used since the beginning of the field of High-Performance Computing [13]. Systems like Split-C [14] and Unified Parallel C [15] introduced the partitioned global address space (PGAS) environment for parallel computing as an extension to the C language, using globally accessible arrays distributed among the computing nodes. Hiding message-passing into global array accesses makes it feasible to achieve functional programs by sharing a common virtual address space; however, it is difficult for developers to optimize remote accesses since they are implicit. In addition, data allocation is static and cannot change based on dynamic load. Titanium [16] made inter-node communication explicit; however, data migrations are still not supported.

Chapel [17] extends the PGAS languages model with an asynchronous approach and the abstraction of locales. Locales can be either an abstracted or real machine component where data or computations can reside; work and data are then assigned to them explicitly. Even though Chapel supports shared memory tasking, distributed load balancing has to be explicitly implemented by the application developer. HPX [18] is another task-based runtime using the asynchronous PGAS approach. HPX does not support distributed load balancing, although it does so in the shared memory. Legion [19] is a data-centric parallel programming framework that targets distributed heterogeneous systems. Its abstraction of logical regions to represent user data allows it to efficiently map data to computing memories and devices, but makes it difficult to develop unstructured and irregular applications. Unicorn [20] and StarPU [21] are systems closely related to Legion, exposing concepts similar to Legion's logical regions to represent their data. Hence, they also lack an efficient way to develop unstructured and irregular applications. D-Galois [22] is

an object-based optimistic parallelization system for irregular applications. It provides a set of concurrency-aware data structures as well as mechanisms to rollback conflicting operations in order to optimize optimistic parallel execution. Galois' abstractions mainly focus on graph analytics, thus, applications need to be expressed in an appropriate fashion. Thus, none of these systems are suitable for developing irregular and adaptive applications that require dynamic data redistributions.

3 Integration of PREMA and Argobots

In this section, we present how each of PREMA's software layers, namely the Data Movement and Control Substrate (DMCS), the Mobile Object Layer (MOL), and the Implicit Load Balancing (ILB) are adjusted to be integrated with Argobots.

DMCS incorporates MPI to utilize multiple computing nodes in a high-performance computing cluster, as well as Argobots to take advantage of the hardware cores of each node. The programming model of DMCS is similar to that of Active Messages (i.e., each message sent is associated with a function call to be invoked on the receiver side once the message is received). This layer is divided into three components: the application, the communication, and the handler-execution component. The application component runs on the default, implicitly-created Argobots Execution Stream. The handler-execution component consists of multiple Execution Streams (ES), usually one less than the number of cores residing on a computing node. The communication component is either handled by a dedicated stream or any stream whose work pools are empty. The streams of the application and handler-execution components run customized Argobots schedulers, each assigned with a primary work pool while accessing each others' pools for the needs of work-stealing. On top of these work pools, each scheduler is assigned a private work pool, used to trigger a change of the active scheduler, a feature used by the higher-level layers. When a scheduler's work pools are empty, it attempts to perform work-stealing on a randomly selected work pool of a local peer. If this fails, it performs exponential backoff to minimize the cycles wasted while waiting for new work units. The communication component encloses all message-passing related operations and is either handled by a dedicated stream or the streams in the handler-executing and application components when the rest of their work has been completed. When a remote method invocation request is received in the communication component, a new User Lightweight Thread (ULT) is created that is pushed to a randomly selected pool of the handler execution component (if any) or to the application component's work pool. In section 6.2, we evaluate different approaches for handler/task creation using Argobots.

By encapsulating remote handler invocations in ULTs their execution can be interrupted at any point allowing others handlers to execute on the same hardware thread. Furthermore, the interrupted handler will continue its execution from where it stopped when there is an available core in the same node.

This functionality suits the needs of parallel applications that otherwise need to use a (busy) waiting mechanism for some resources to become available. Using the yielding function, an application can interrupt the running ULT that needs to wait so that another one can run. In contrast, in the PThreads implementation, using busy waiting inside a handler for reasons other than for sending a message was discouraged as it could cause starvation and even deadlocks in some cases. The use of ULTs and their yielding capabilities allow to overcome these constraints. Some scenarios that can leverage the yielding functionality include blocking on a lock and message acknowledgment operations. Figure 2 shows an example of how using ULTs can avoid deadlock cases induced when blocking in the PThreads implementation. The issue arises by the implementation of the *wait_until([condition])* operation, which blocks the running thread until the given condition evaluates to true. To avoid wasting cycles while waiting, PREMA tries to find another task to run by popping the next task available in the pools; however, when used from inside a handler, it can lead a live-lock scenario like the following. Let us assume a scenario of three tasks, namely T1, T2, and T3 (see Figure 2 left) where T1 needs to wait for some acknowledgment A3 from T3, T2 waits for acknowledgment A1 from T1, and T3 does not wait for any acknowledgments. PREMA starts running T1 until it blocks waiting for acknowledgment A3, then switches to T2 until it blocks waiting for acknowledgment A1, and finally switches to T3, which acknowledges A3. Even though A3 has been acknowledged, the control will never return to T1 to unblock it. Once T3 finishes its execution, the control returns to T2's *wait_until()* operation, which will keep checking the task pool but will never run T1 since T1 has already been popped and is running T2 from its *wait_until()* operation. The Argobots implementation avoids such a scenario by using separate stacks for each task, saving their states before switching control of the execution stream, and resubmitting them to the task pool when unblocked. This also allows blocked tasks to be stolen in case the currently running thread starts a long-running process.

The ILB layer is implemented as an Argobots' stackable scheduler that is pushed to the dedicated pool of each available execution stream to change the DMCS scheduling policy. It inherits the pools created by DMCS to continue executing remote handlers of the lower layers while also handling pools dedicated to the ILB. Handlers need to be issued through the ILB messaging operation for their loads to be monitored. Their execution consists of two steps. In the first step, the requests need to be routed to the current location of the target mobile objects and have their load evaluated. In the second step, handlers are scheduled for execution. For the first step, MOL finds the location of the mobile object and guarantees that it is in a valid state. Next, the ILB is notified about the new handler, the handler's load is calculated, and it is pushed to the list of pending handlers of the mobile object. The second step is executed later from the stacked Argobots scheduler created for the ILB. This scheduler maintains a list of all local mobile objects in the computing node to monitor the load of the whole node. When there is no other ongoing

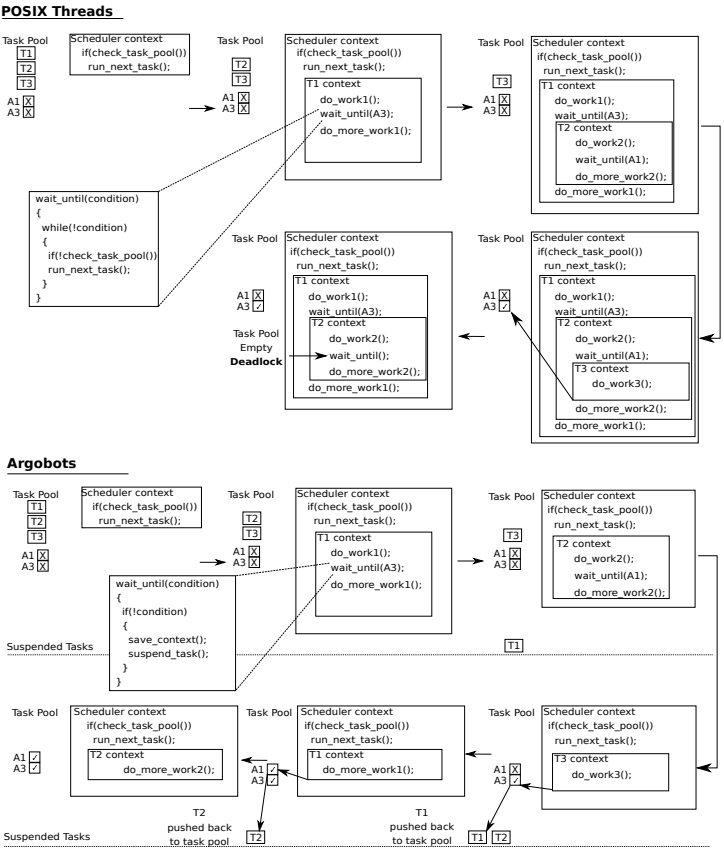

Fig. 2 An example of how blocking inside a handler can cause deadlock, and how Argobots helps to solve this issue.

work, it picks one handler from the list of pending handlers of the next available mobile object and creates a ULT out of it. ULTs created from this part of the system are then pushed to the fine-grained tasking module, presented in section 5. If no pending handlers are available, ILB starts a new cycle of distributed load balancing to find a remote mobile object with enough workload. The distributed memory load-balancing scheduler is also implemented inside a ULT as part of the ILB to allow the load balancing policies to use operations that could block. Since this operation includes sending messages and checking containers that might require some synchronization, ILB should provide applications with this feature.

4 Constructs for Group Communication

```
void run_important_computation(...);
bool computation_done = false;

DEFINE_HANDLER(computation_hdler_done) {
    // Signal waiting thread
    computation_done = true;
}

DEFINE_HANDLER(computation_hdler) {
    // Execute computations as needed
    run_computation(dmcs_get_msg_args());

    // Execution is done, notify source
    dmcs_send(dmcs_get_msg_source(),
          computation_hdler_done);
}

int main() {
    double values[4];
    size_t v_size = sizeof(data);
    int target = 1;

    // Execute run_computation_hdler on
         target
    // args: values[4]
    dmcs_send(target,
          run_computation_handler,data,v_size);

    // Wait until notified completion
    wait_until(computation_done == true);
}
```

```
void run_important_computation(...);

DEFINE_HANDLER(computation_hdler) {
    // Execute computations as needed
    run_computation(dmcs_get_msg_args());
}

int main() {
    double values[4];
    size_t v_size = sizeof(data);
    int target = 1;
    dmcs_future done;

    // Execute run_computation_hdler on
         target
    // args: values[4]
    dmcs_send(&done,target,
          computation_hdler, data,
          v_size);

    // Suspend ULT until future is set
    done.wait();
}
```

Fig. 3 Minimal example of remote handler completion acknowledgment in PREMA without (left) and with (right) futures.

4.1 Futures

A feature of Argobots that takes advantage of the ULT's ability to yield is the *eventual* data type. An eventual corresponds to the concept of futures found in many programming languages. A future is a mechanism for safely passing values between threads that run concurrently. Argobots implement this mechanism and enhance it with the yielding power of ULTs. A running ULT may designate an eventual where it will store a value once it executes; other ULTs that need this value for their computations can then attempt to retrieve it by accessing the respective future. Trying to access an eventual, which has not been assigned a value yet, will cause the accessing ULTs to block. In such a case, the ULT is suspended by the Argobots framework and removed from the pool of active work units. It becomes available again only when the respective value of the event has been set. If the future has already been assigned a value, it will allow the ULTs accessing it to retrieve the value without blocking. PREMA utilizes this feature of Argobots and exposes it as a high-level construct while also extending it to distributed memory.

In the old design, an application using PREMA needed to manually develop acknowledgments for remote handlers execution when such functionality was desired (Figure 3 left). In the new version of PREMA, when the application needs to invoke a function on a remote processor or mobile object, the

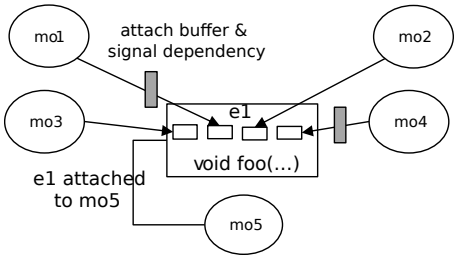

Fig. 4 A visualization of PREMA's event primitive. Event e_1 is attached to mobile object mo_5 and expects four dependencies to be signaled in order to invoke foo on mo_5. Once foo is invoked, mo_5 has access to all buffers attached as dependent data.

messaging interface used can optionally return a future that is signaled once the remote function call completes(Figure 3 right). Internally, PREMA creates a new Argobots eventual and attaches its identifier to the header of the remote message handler. Once the handler invocation completes on the target, PREMA checks whether the respective section of the message header is set. If it is, it will invoke a new remote handler back to the sender with the value related to the future as an argument. When the remote handler executes, it sets the future to a ready state on the original sender and thus, unblocks all ULTs waiting for it. With this mechanism, the synchronization of the application logic becomes easier to handle and can be better optimized by PREMA.

4.2 Events

Distributed futures manage to remove most of the boilerplate code users had to write to acknowledge handler completion. They, also, improve concurrency and eliminate some of the constraints of the original implementations of PREMA. However, they can only signal one dependency at a time, either acknowledging the completion of a handler or used in a user-defined functionality (e.g., blocking on a future that is signaled from a remote message manually). To handle cases where multiple dependencies need to be satisfied, we have introduced the primitive of distributed events. An event is a synchronization tool associated with a mobile object, a handler task, a predefined number of dependencies, and optionally data buffers. It is uniquely identifiable in the whole distributed system and can be used by the application to satisfy dependencies remotely (see Figure 4). For example, an event can be triggered as the last step of a handler to signal its completion to dependent tasks and pass required data to them. The data can even consist of a mobile object that needs to be transferred to the location of the associated mobile object. In this case, the runtime system will handle the migration process implicitly, even if the object resides remotely. PREMA transfers the signals from different dependencies along with their data or mobile objects and triggers the associated handler task when the predefined number of dependencies has been fulfilled.

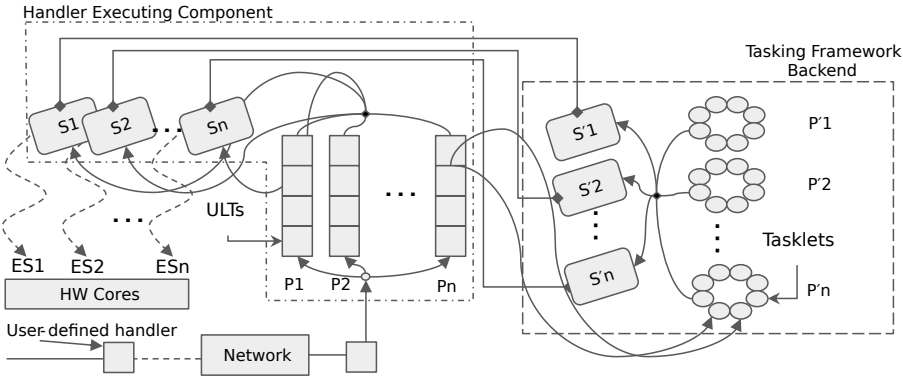

Fig. 5 From left to right: a user-defined handler, implemented as an Argobots ULT, comes through the network and is assigned to one of the task pools $\{P_1, P_2, ..., P_n\}$, managed by the schedulers $\{S_1, S_2, ..., S_n\}$; tasks are mapped to Argobots execution streams $\{ES_1, ES_2, ..., ES_n\}$ which in turn execute on hardware cores. A user-defined handler can spawn one or more fine-grained tasklets, which are handled by the tasking framework backend. The tasking framework (right) has its own task pools $\{P'_1, P'_2, ..., P'_n\}$ and schedulers $\{S'_1, S'_2, ..., S'_n\}$.

5 Fine-grained Recursive Task Parallelism

The need for finer-grained parallelism inside a handler arises from the large workload disparity witnessed among handlers of irregular and adaptive applications [23]. Despite the use of over-decomposition to diffuse workload[24, 4], the workload disparity among handlers targeting different mobile objects can still be large. This is an effect of the decomposition being computed at the initialization stage of the application. To bridge this gap, we developed a standalone tasking module on top of Argobots. It is designed to help utilize multiple hardware threads in the context of a single handler execution. In [5] we study the standalone version of the module as a black box and experiment with different models to spawn parallel tasks in shared memory. In this work, we delve into its low-level implementation and integration with PREMA, as well as the issues raised from introducing preemption on a tasking framework utilizing lock-free task pools. A tight integration of the two systems is a requirement for an efficient, low latency hybrid tasking model. When used as an integrated part of PREMA, this module can utilize the existing execution streams, avoiding the creation of new threads and the possible over-subscription overheads. To distinguish between ULTs used in other parts of PREMA and tasks created by this module, the latter will be called tasklets for the rest of the paper.

The interaction between the shared and distributed memory modules within PREMA is depicted in Figure 5; namely the handler executing component and the fine-grained tasking framework backend. Each of the two modules consists of a set of schedulers $\{S_1, S_2, ..., S_n\}$, $\{S'_1, S'_2, ..., S'_n\}$ and task pools $\{P_1, P_2, ..., P_n\}$, $\{P'_1, P'_2, ..., P'_n\}$. Handlers are issued either locally or remotely, through the network, and are implemented as Argobots User Level Threads (ULTs). Tasklets can be spawned in the context of a handler execution and

are managed by the tasking framework backend. The schedulers of the tasking framework backend are attached as plugins to PREMA's schedulers, allowing them to utilize the existing Argobots Execution Streams, avoiding resource over-subscription.

5.1 Low-level Implementation

The scheduler of this module attempts to maximize parallelism and minimize memory use by using a hybrid of depth-first and breadth-first execution policy and recursive creation of work units. Each processing element (PE) is associated with a list of tasklets. Each time a new tasklet is created by a PE, it is pushed to the bottom of its list. To pick a new tasklet to execute, the PE pops a tasklet from the bottom of its list; if the list is empty, it will try to steal a tasklet from the top of another PE's list. Provided that the tasklets creation is performed recursively, this scheduling algorithm prioritizes tasklets that are hot in the cache (latest created) when there is pending work in the pool, while maximizing the amount of stolen work when stealing is attempted, by targeting tasks that were created early in the recursion steps. To implement this scheduling algorithm, the Argobots abstraction of custom pools was used to encapsulate a lock-free implementation of a circular double-ended queue (deque)[25]. The interface of the abstract pools only provides push and pop (and remove but is not needed in our case) operations to manipulate the contents of a data structure. Thus, stealing is implemented as part of the pop operation, and each abstract pool is implemented as an array of pointers to all available deques, instead of associating it with a single deque. When an ES is ready to pick a new task, it calls the pop function, which, in turn, checks its deque; if empty, it will randomly steal a tasklet from another deque.

Tasklet dependencies are also provided in this module; a "parent" work unit can create several "children" tasklets and wait for their completion. The "children" can then create their own "children", constructing a tasklet dependency tree. The function that creates a tasklet returns a handle that can be joined for completion, causing the caller to yield if not complete. The root tasklet of a tree is implemented as a ULT to enable yielding; however, children tasks are implemented using common list structures for performance. To enable switching between children tasklets when waiting for dependents completion, we explicitly push and pop children tasks from the lists and run them inside the parent or a separate ULT, which allows yielding from any node in the tree. Once a tasklet execution completes, the control returns to the parent tasklet, which, in turn, evaluates whether it will continue waiting for other children to complete or not. Waiting in this context will cause popping/stealing another tasklet.

5.2 Safely yielding tasklets

An important incentive to build such tasking frameworks on top of lighweight threads (LWTs), is the ability of LWTs to yield, allowing the creation of tasks

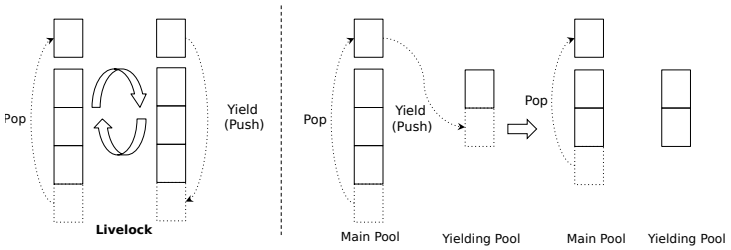

Fig. 6 An example of a live-lock scenario during the yield operation in the tasking framework (left) and how it is avoided by using a secondary pool to temporarily store yielding tasklets (right).

that might not run to completion. This enables tasks to run blocking operations such as messaging and acquiring locks. The requirements to maintain this ability while using the aforementioned approach to creating tasks yields some issues that can cause the framework to not operate correctly. In this section, we identify the problems raised by this task creation strategy and present our approach to handle them.

5.2.1 Avoiding live-locks

The yielding operation in Argobots saves the context of the yielding ULT, pushes the ULT back into its respective work pool, and switches control to the execution stream's scheduler; however, following the same steps in the case of our fine-grained tasking module could cause a livelock. In our work-pool implementation, both push and pop operations target the bottom of the deque. Pushing a yielding ULT back to the same pool will insert it at the bottom of the deque, making it the first ULT that will be popped in the next pop request. In a scenario where a ULT yields while busy waiting for some resource, the yielding ULT would be constantly popped and pushed to the bottom of the deque, allowing no other ULTs to be executed (Figure 6 left). To avoid such a scenario, we distinguish between new and yielding ULTs by keeping track of the last ULT executed on each ES and comparing it with ULTs that are about to be pushed to the respective work pool. If they are the same, we can infer that the ULT about to be pushed is yielding. Yielding ULTs are pushed to a secondary work pool, maintained per ES, to unblock the main work pools and allow other ULTs to execute (Figure 6 right). Once all ULTs in the main pools have executed, the ULTs in the secondary/yielding pool are run; ULTs running in the secondary pool and yield are pushed back to the main pool to avoid the same live-lock scenario from occurring there.

5.2.2 Signaling blocked tasks

Another issue arises from the default implementation of signaling blocked ULTs in Argobots. Blocked ULTs remain in a private structure of Argobots; when an active ULT signals a blocked ULT, it changes its state from blocked to active and pushes it back to the last work pool where it resided before being

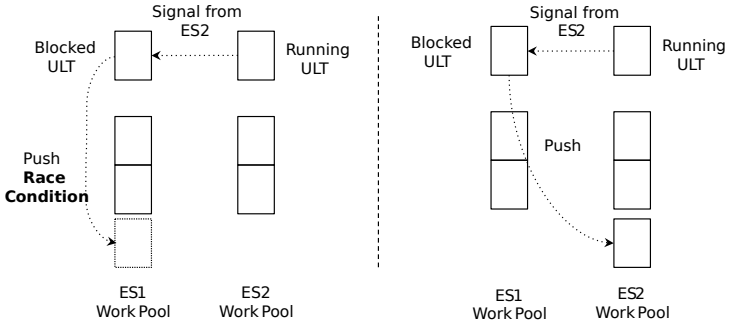

Fig. 7 By default, when a blocked ULT is signaled to unblock it is pushed back to the last pool it executed. The push operation is executed by the ES of the ULT that signals the blocked ULT; as a result, an ES may push the just unblocked ULT to the work pool of another ES which causes a race condition in our lock-free implementation (left). To overcome this issue, our implementation ignores the pool that Argobots chooses to push the blocked ULT and always pushes it to the pool of the signaling ULT's ES. This modification conforms to the requirements of the custom deque and avoids race conditions.

blocked. This work pool might belong to an ES different than the one running the signaling ULT, which breaks the requirement that each execution stream can only push and pop to/from its work pool and only steal from other work pools (see Figure 7, left). To overcome this issue, the push operation ignores the target pool that Argobots chooses and assigns signaled ULTs to the work pool of the execution stream running the signaling ULT, as shown in Figure 7, right. Thus, pushing to a work pool can only be performed by the execution stream that owns the pool, conforming to the deque requirements.

6 Performance Evaluation

In this section, we measure the performance of PREMA, taking advantage of the contributions presented so far. Due to the difficulty in isolating minor overheads derived from different approaches for handler task creations and message-handling using full scale, real-world applications, in 6.2 and 6.3 we evaluate PREMA on benchmarks derived from the widely-used OSU Microbenchmarks [26]. In section 6.4, we derive a synthetic microbenchmark that stresses PREMA's handler execution component, presenting an extreme condition of highly contended mutexes, and present the impact on its performance before and after utilizing lightweight threads. Section 6.6 presents an evaluation on SW4lite; an application designed to reflect the workflow of a few important kernels of SW4. It is part of DoE's exascale project (ECP) initiative for benchmarks that accurately represent a wide range of scientific applications while avoiding dealing with large and complex code bases [27]. Sections 6.5, 6.7 present evaluations on real-world applications in shared and distributed memory, respectively.

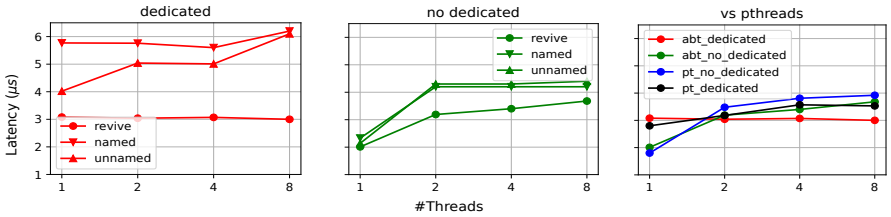

Fig. 8 Latency observed on ping pong benchmark for different task creation approaches using dedicated streams for communication (a) or not (b) and comparison of the best approaches with the PThreads implementation (c).

6.1 Experimental Setup

Two computing clusters are used to run the performance benchmarks. The first one, namely Turing, is a 250-node cluster consisting of Intel(R) Xeon(R) (E5-2660, E5-2660 v2, E5-2670 v2, E5-2698 v3,E5-2683 v4) 128 GB CPUs ranging between 16 to 32 cores spread among two sockets (2 NUMA nodes). The second platform, namely Wahab, is a 200-node cluster which utilizes Intel(R) Xeon(R) Gold 6148 @ 2.4 GHz CPUs of 40 cores each in two sockets (4 NUMA nodes).

6.2 Handler Task Creation

In this subsection, three approaches to creating handler tasks as lightweight threads are examined. Performance is measured as the latency in handler creation, with and without a dedicated communication stream, in a ping pong benchmark. Two nodes exchange 20000 64B-sized messages in total, where the sender sends a message and then waits for an acknowledgment. The three approaches are described below:

- Unnamed: The Argobots runtime is responsible for monitoring and releasing the memory of ULTs when they complete.
- Named: PREMA checks ULTs for completion and frees their resources explicitly. An array of handles is maintained per handler executing ES.
- Revive: A variation of the second approach where the completed ULTs in the preallocated arrays are reused through the *ABT_revive()* function.

Figure 8 shows the performance observed in terms of latency for the three different approaches when using a dedicated stream for communication (8a) or not (8b) and compares the best for the two with their PThreads counterparts. In all cases, a single message is on-the-fly at any time. Two nodes exchange 20000 64B-sized messages in total, where the sender sends a message and then waits for an acknowledgment before sending the next.

Figure 8a shows the latency observed when a dedicated stream is in use to handle communication and assign the respective tasks to other streams. The approach used plays a significant role in this case, where the “revive” approach

of reusing previously completed ULTs benefits the overall performance, maintaining a very stable latency of $3\mu s$. In contrast, using the "named" or "unnamed" approach not only achieves a lower performance (of 6 and 4 μs, respectively) when a single handler executing stream is used but also faces increasing overheads when the number of streams also increases. The performance degradation observed in this case comes from the fact that ULTs use their own stack. Each time a stream creates a new ULT, it needs to allocate its stack memory, which increases the critical path of the ULT creation. Reviving ULTs, on the other hand, does not need to allocate new memory as the memory of previously completed ULT is reused.

Figure 8b shows the latency observed when no dedicated stream is used. In this case, the significance of the three different approaches is not apparent when only a single stream is in use. The reason is that Argobots use internal memory pools for the stacks of completed ULTs per ES. When a stream completes the execution of a ULT, it stores the memory used for the stack in its memory pool to reuse it later and avoid reallocations. However, when one ES creates most ULTs (producer) and other ESs execute them (consumers), the producer pool is depleted without being reused. The consumer ESs keep the stack memory in their memory pools, forcing the producer to steal from them or allocate more memory. This extra overhead is observed for "named" and "unnamed" approaches when two or more streams are used in figure 8b. In this case, the overheads are lower compared to Figure 8a because all streams have the same chance to produce or consume ULTs. Thus, they all have more opportunities to refill their pools and avoid allocations or steals.

Figure 8c shows the comparison of the revive approach when using dedicated streams or not. We also compare the performance of the two with their PThreads implementation counterparts. Using no dedicated thread for communication achieves the least overhead when only a single thread is utilized, and PThreads exhibits the lower overhead in this case. When more than one handler executing stream/thread is available, using a dedicated stream showed lower latency for both the Argobots and PThreads implementation, with a difference between them of about 15%. Using no dedicated stream falls a little behind with the PThreads implementation achieving up to 10% worse performance than the Argobots counterpart.

6.3 Message Handling

Message handling is part of the DMCS and utilizes the MPI as a communication library. In this section, four ways to handle message passing are evaluated.

- ULT-Per-Message (UPM): A new ULT is created to run the MPI operation, yielding if the operation did not complete immediately or exiting otherwise. If a separate stream is used for communication, message reception is handled by a separate ULT that runs in a loop and yields; otherwise, one of the handler-executing threads directly polls the network.

- Communication-In-Pool (CIP): Two ULTs are spawned that run in a loop, one for receiving and one for sending messages. Send requests are passed through regular C++ queues. Each ULT serves the respective operations and yields its execution.
- Combining the two (CIP-UPM): A new ULT is created for each outgoing message (like UPM), while incoming messages are handled through a separate ULT running in a loop (like CIP).
- Queue: Outgoing message requests are passed through a C++ queue (like CIP). When no dedicated stream is used, this queue is served directly from the handler-executing streams that also serve the incoming messages. Otherwise, a single ULT is spawned in the dedicated stream that continuously polls the network and serves message requests in the queue.

The performance of each approach is presented in Figure 9 in combination with the different task creation approaches on the same benchmark. We present results for both cases where a dedicated stream is used for message handling (bottom) or not (top). As can be seen from the graph, the best performance in terms of latency observed is achieved through the combination of the "queue" message handling and the "revive" handler task creation approach with $3\mu s$ latency achieved on average. It is interesting to note that depending on the existence of a dedicated stream or not, different combinations provide the best performance, except for the best case (queue-revive).

When no dedicated stream is present (top), the "queue" approach is the best performing, regardless of the task creation approach. CIP and UPM perform similarly but with an increasing overhead compared to queue as the number of handler-executing streams increases (up to $5\mu s$ overhead). This is expected since the two approaches require the creation of ULTs and context switching in comparison to the queue approach, which only needs to check a queue and poll the network. The combination of CIP and UPM (CIP-UPM) adds the largest overhead in all cases as it constitutes the longest critical path before handling message passing (up to $6\mu s$ overhead). On the other hand, when a dedicated stream is in use for message passing (bottom), we see that UPM and CIP-UPM perform better when the named or unnamed task creation approach is used (on average $3.5\mu s$ overhead). In these cases, the overhead of the other two message-handling ways is experiencing almost double the overhead as the number of handler-executing streams increases(up to $6\mu s$). Thread stack creation and reuse is probably the explanation for this effect as in the task creation benchmark. In this benchmark, it has been shown that the queue + revive combination of message handling and handler task creation is the best in terms of latency, regardless of whether a dedicated stream is used for communication.

6.4 Blocking Operations in Handler Execution

An important feature stemming from the integrating with Argobots is the ability to yield handler execution either explicitly by calling the respective

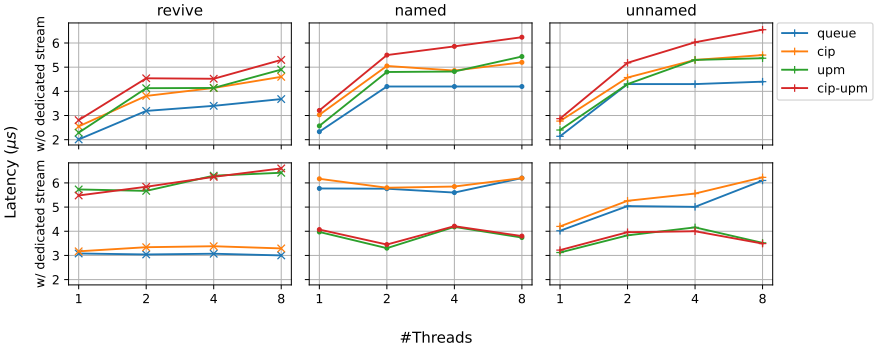

Fig. 9 Latency observed on ping pong benchmark for different task creation and message passing handling approaches using dedicated streams for communication (top) or not (bottom).

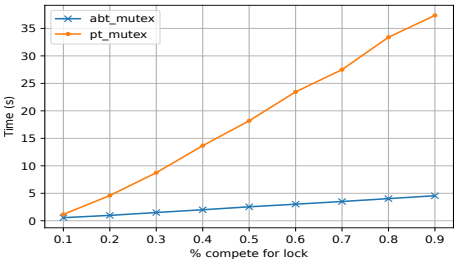

Fig. 10 Execution time with respect to percentage of tasks of a mobile object competing for its mutex. Mutex is implemented as an Argobots mutex (abt_mutex) or PThreads mutex (pt_mutex)

function or implicitly when a blocking call is detected. The potential benefit provided is presented through a synthetic benchmark. An allocation of ten cores is used along with ten mobile objects, each using an exclusive mutex to provide access to its data. For each mobile object, 100 handler invocations are issued where a specific percentage of them needs to take control of the mutex before executing. We set the time that the mutex is held to 50ms at a time and experiment with standard PThread mutexes and ULT-aware mutexes that can suspend handlers when locked. Figure 10 shows an evaluation of the two implementations with different percentages of handlers acquiring the mutexes, ranging from 0.1 to 0.9 (10 - 90 handlers/mobile object). One can observe the tremendous difference observed in the two implementations (up to 1000%). The issue with the PThread implementation is that a handler that tries to access a taken mutex will block the hardware thread it executes on, preventing handlers targeting a different mobile object/mutex from running. In contrast, the Argobots implementation will suspend the running handler ULT, allowing another handler to run on the same thread. In similar cases, the user might know that acquiring a resource exclusively could cause delays and, thus, may explicitly yield competing handlers to mitigate the propagation of such effects.

6.5 Fine-grained Recursive Tasking

In this section, we evaluate the performance of the fine-grained tasking framework on the Barcelona OpenMP tasks suite of benchmarks[28] and compare it with OpenMP and TBB. Specifically, we focus on three benchmarks: Protein Sequence Alignment, Fast Fourier Transformation (FFT) on one billion points, and Sort on one billion elements. All three benchmarks are run on both computing infrastructures available to us and are implemented following a recursive parallelization model, suitable for PREMA's tasklets; their results are presented in Figure 11. Another thorough comparison study for the three libraries is presented in [5] in the context of parallel mesh generation.

6.5.1 Protein Sequence Alignment

The performance achieved by PREMA tasklets in both systems is depicted in the left of Figure 11, achieving a speedup of 26 on the 32-core system and 37.4 on the 40-core. The difference in performance between PREMA tasklets and the two industrial-strength frameworks is negligible; PREMA tasklets outperform OpenMP in all thread variations on the Wahab cluster and fall behind on the Turing cluster but only by a small amount. A similar trend is noted when performance is compared to TBB on the two machines.

6.5.2 FFT

An interesting observation in this benchmark is the declining performance of OpenMP (Figure 11, middle). We see that its performance suffers in the specific application as it is affected by the large number of tasks generated. On the other hand, PREMA's and TBB's performance scales well on Turing for the first 16 cores ($\approx$ 13 speedup) and saturates to approximately 19 on 32 cores. For the Wahab system, PREMA continues to perform on par with TBB, achieving a speedup of 11 on 16 cores, with both frameworks' performance declining after this point, having no increase in speedup for 32-40 cores. We see that in this benchmark the speedup achieved by any framework is much lower than what was observed in the previous one. The difference between the two benchmarks is that first, each task of the alignment case runs much longer than a single task in FFT, and second, FFT creates about a thousand times more tasks. Thus, the overheads related to task creation are much more difficult to hide behind computations and also, they add-up quickly due to their large number. The decline in performance on higher core counts can also be attributed to the decrease in the amount of work per thread, which limits the amount of possible concurrency while increasing the number of failed steal attempts, and to the use of more CPU sockets which leads to NUMA effects.

6.5.3 Sort

The declining performance of OpenMP is also exhibited in this benchmark; however, it does so once it reaches the utilization of 16 cores. The other two

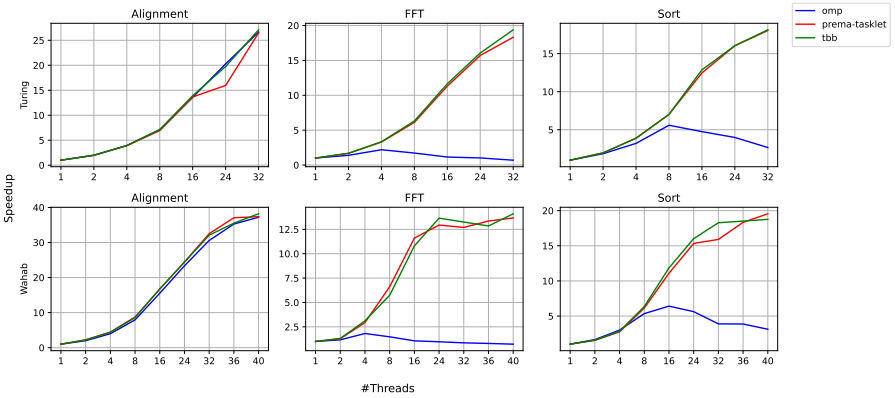

Fig. 11 Speedup achieved on two different platforms, Turing (top) and Wahab (bottom), from fine-grained tasking using OpenMP (omp), Intel TBB (tbb) and PREMA tasklets, for Protein Sequence Alignment, FFT and Sort applications.

tasking frameworks continue to perform well on both platforms. The performance of the Turing machine continues to be superior both for the sequential case and parallel ones, even though it utilizes fewer threads. Like in FFT, both frameworks' performance scales well up to the first 16 cores with a speedup of approximately 13 on Turing and 11 on Wahab. In larger counts of cores, the performance of both frameworks declines on both platforms (Figure 11, right). This decline stems from the same factors as the FFT case, i.e., large number of small tasks creating overheads difficult to overlap. Moreover, this benchmark also introduces a significant amount of accesses to memory shared among multiple threads, leading to cache-line false sharing effects.

6.6 Seismic Wave Simulation Benchmark

In this section, we evaluate the new features of PREMA on the SW4lite benchmark [29]. We study its performance on different work unit allocations (mobile objects, PREMA tasklets) for single-node and multi-node experiments.

6.6.1 Single Node Performance

We evaluate the performance of SW4lite on a single node, utilizing different approaches to concurrency, utilizing domain decomposition and message passing, or tasking. For the task decomposition approach, we evaluate OpenMP (omp for), PREMA tasklets as a stand-alone library, and PREMA tasklets as part of a single PREMA instance. For the domain decomposition approach, we evaluate MPI, one instance of PREMA for the whole node using one mobile object per thread, and using one instance of PREMA (utilizing a single mobile object) for each core. Figure 12 shows the results of this evaluation for a small input running the point-source functionality of SW4lite. It is clear that both versions of PREMA tasklets outperform the OpenMP version of the code (lines with triangle markers) by about 20% but only attain a speedup of 10 at 40

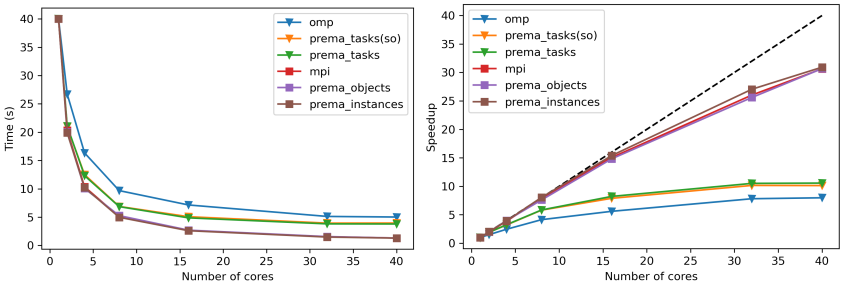

Fig. 12 SW4 performance of threading (triangles) and message passing (squares) parallelism on a single node for a small input. omp: OpenMP, prema_tasks: tasklets, prema_tasks(so): tasklets as a standalone library. mpi: 1 MPI rank/core, prema_instances: 1 PREMA rank/core, prema_objects: 1 mobile object,thread/core.

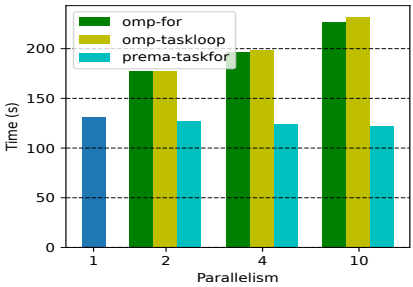

Fig. 13 Intra-handler parallelism with OpenMP for , taskloop and PREMA fine-grained tasklets.

cores. On the other hand, the domain decomposition approach performs much better, achieving a speedup of 32 at 40 cores for all versions. This observation raised a concern about whether the performance disparity will always be in favor of just data decomposition or there is an optimal combination of the two approaches that gives the best results. In other words, whether it is always the best choice to decompose the data into more "pieces" as the number of cores increases or there is a point where it's more efficient to apply a two-level approach with a coarse-grained data decomposition and finer-grained task decomposition on top of it. In the next section, we experiment with different approaches for combinations of data and tasking over-decomposition.

6.6.2 Multi-Node Performance

Our observations on the single-node evaluations lead us to run more experiments on larger inputs and different work unit allocations to find out how different combinations of data and task decomposition can affect the performance of an application like SW4lite. The addition of tasklets on top of PREMA makes it enables running such a study simplifying the experimentation with data and task decomposition independent of the available

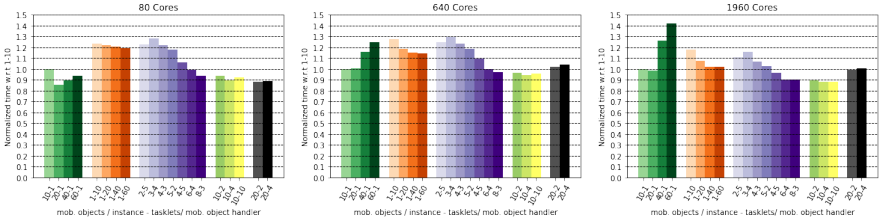

Fig. 14 Normalized performance of the SW4 benchmark for different PE allocations with respect to the performance achieved when mapping one mobile object per PE and one tasklet per handler (10-1).

hardware resources. In other words, one can decompose the application data/-tasks domain in many more "pieces" than the available nodes/cores without needing to provide extra code to handle them or taking care of resource over-subscription. Trying to run such a study without tasklets, e.g., using OpenMP, results in a bad performance, as shown in Figure 13. In this benchmark, we exploit distributed and shared memory parallelism concurrently, using one mobile object per core for 640 cores and also try intra-handler parallelism using OpenMP for, OpenMP task loop, and PREMA tasklets. Using OpenMP causes over-subscription of hardware resources (PEs) since it is unaware of the existing threads running from PREMA, resulting in overheads related to constant context-switching and multiple threads competing for the same hardware core. In contrast, PREMA tasklets are aware of the threads already running and utilize them efficiently, achieving up to 10% improvement over the base case or 60% over OpenMP.

Using PREMA tasklets, we evaluate different approaches to (over-)decompose only at the data domain level, only at the task level by creating tasklets and with a combination of the two. The different work unit allocations allow different levels of freedom for PREMA to take advantage of, like load balancing and latency hiding, since both data subdomains and tasklets in a PREMA instance are shared among its threads. Figure 14 presents the overall running time on an increasing number of cores (strong scaling), normalized by the running time of the base case. The base case for this experiment is using one mobile object per hardware thread and one tasklet per handler, using one instance of PREMA per socket of 10 hardware cores to avoid NUMA effects. The x-axis shows the work-unit allocations, where the first number represents the number of mobile objects per PREMA instance and the second one shows the number of tasklets spawned per mobile object handler. As can be seen from the figure, the application benefits from increasing the number of mobile objects per instance (green bars) when the number of cores is low and the work enclosed in each mobile object is substantial to overlap the overheads of message-passing and context-switching. However, it benefits less as the number of cores increases and the work per mobile object thins out , and thus, the overheads related to the increased number of messages and context-switching can no longer be tolerated. Utilizing task decomposition only (orange bars)

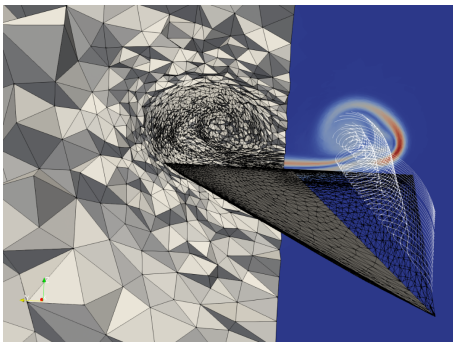

Fig. 15 Visualization of a mesh generated by the parallel mesh refinement application of this section: Metric-adapted mesh to a laminar flow over a delta wing. Figure adapted from [5].

cannot surpass the performance of domain decomposition in any case, but it gets better as the number of tasks increases and the size of the data per PREMA instance decreases. This can be explained from the data observed in 6.6.1 where the task parallelism approach achieves by far worse performance than the approach utilizing data decomposition. Thus, when the are fewer but larger data domains, task parallelism performance suffers due to the inherent implementation of the applications. However, when there more but smaller data domains, the inherent sub-par performance of task parallelism is less important and task-overdecomposition can help to evenly balance workload and more efficiently utilize the available cores.

Combining the two approaches seems to have better results when the number of mobile objects is close to the number of threads, and task decomposition is performed on top of that (purple bars). Our best results (12% improvements) in most cases were achieved when using a combination of 10 mobile objects -10 tasklets (yellow bars). Using 20 mobile objects per instance (2 per thread) and two to four times data decomposition per object (gray bars) also showed some improvement in lower counts of cores which diminished in larger allocations without, however, hurting performance as other cases did.

These experiments show that a reasonable level of over-decomposition in both levels can substantially benefit the performance of an application by allowing an overall increased flexibility for the underlying scheduler/load balancer. The specific values of over-decomposition are application-dependent and might need some experimentation to optimize. Adjusting the decomposition can be difficult and error prone when done by hand but PREMA completely hides this transition once the standard case has been implemented

6.7 Parallel Mesh Refinement

Our final benchmark is a parallel mesh refinement application, namely CDT3D [30]. A visualization of a mesh generated by CDT3D is presented in Figure 15. From our previous experience integrating this highly irregular and

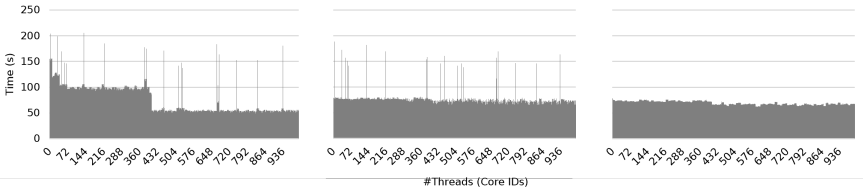

Fig. 16 Per core execution time for the mesh refinement application with inter-handler shared memory load balancing (left), inter-handler shared & distributed memory load balancing (middle), and intra-handler tasklet parallelism on top of inter-handler shared & distributed memory load balancing (right)

adaptive application with PREMA, we detected some serious limitations of PREMA that led us to enhance it with the lightweight threads and the fine-grained tasklets. To run the application on top of PREMA, we decomposed the initial mesh into sub-meshes and assigned them to the available cores, running a sequential refinement handler on each. We over-decompose the mesh so that each core is initially assigned ten sub-meshes to allow more flexibility for inter-handler shared and distributed memory load balancing. The issue is depicted in the first two graphs of Figure 16 that present the execution time per core when running CDT3D on 1000 cores (25 nodes) of the Wahab distributed memory machine. The left graph shows the execution time of each core when only shared memory inter-handler parallelism is used, while in the middle, distributed load balancing is also utilized. While most of the load balancing issues are handled, a few "spikes" in execution time remain, which constitute refinement processes that took too long to execute compared to the average, stemming from the adaptive nature of the application. By invoking PREMA tasklets to explore intra-handler parallelism, we are finally able to achieve correct load balancing that reduces the overall running time by 50% compared to the distributed load balancing case and 60% compared to the base case. The spikes in execution time are removed by overdecomposing a single handler into multiple tasklets. The tasklets that constitute this handler can be shared across the available cores of a single node, effectively parallelizing the workload that would otherwise map to only a single core. Thus, mitigating the effects of any handler that executes substantially longer than others.

7 Conclusion and Future Work

We have presented the integration of PREMA with lightweight threads utilizing Argobots. The product of this effort overcomes the limitations previously exhibited by PREMA while incorporating features for effortlessly handling control flow in shared and distributed memory and a tasking framework that allows for fine-grained parallelism inside the execution of remote method invocations. We have experimented with multiple design choices for task creation and message handling, where we observed up to 100% difference in latency. Moreover, we have shown that the lightweight threads remove constraints

forced by previous implementations, allowing blocking in remote method invocations while avoiding the overheads stemming from waiting semantics like mutexes. The fine-grained tasking framework achieves performance on par with industrial-strength systems like TBB while outperforming OpenMP. Our experimentation with different combinations of workload decomposition both on the data and task level on the SW4lite benchmark showed up to 12% improvements. Finally, we show that the integration of tasking on top of remote method invocations can have tremendous effects on irregular applications, like 3D mesh refinement, achieving up to 50% improvement through exhibiting multi-grain over-decomposition both on the data and task level.

In the future, we intend to use the current work as an intermediate layer to scale the high level abstractions of the shared memory tasking framework presented in [5] over distributed memory systems. Using this product as a vehicle, will study the performance impact of parameters regarding the optimal number of PREMA instances per node and the number of threads per PREMA instance, as well as different approaches to distribute the available workload. This aspect can be divided into two sub-problems: (A) How to distribute the mobile objects themselves and how to invoke the respective tasks on each of them. Approaches to try are: (1) do nothing at the application-level and let the load balancing algorithm distribute the mobile objects reactively, (2) proactively distribute the mobile objects one at a time using a round robin policy, (3) proactively distribute the mobile objects in groups, and (4) use the same approach as (3) but consider the expected load of each mobile object to form potential groups of tasks. (B) How to handle task submission after mobile objects have been distributed? Similar to the task submission approaches for the shared memory implementation, we will evaluate approaches like: (1) a single process submits all (remote) tasks (flat model), (2) statically assigning task submission to all available processes, and (3) hierarchically assigning task submission in a tree-like fashion.

As a next step for our work in PREMA, we are working on integrating support for heterogeneous distributed systems since they are becoming the norm in high-performance computing. Our preliminary results show that our implementation can substantially decrease the effort required to develop applications on such hardware while optimizing performance [31]. In the future, we intend to incorporate machine learning (ML) into the runtime system to automatically infer the data and task over-decompositions that optimize the performance of the DSL. A bridge to practice and understand the requirements of such a process while collecting data from the current implementation of the runtime will be our work on leveraging ML in the field of nuclear physics accelerators[32, 33].

Declarations

Ethical Approval

Not applicable.

Competing Interests

The authors declare no competing interests.

Authors' Contributions

P.T wrote to the main manuscript text. Both authors contributed to the conception and design of the work. P.T developed the software and benchmarks of this work. Both authors reviewed the manuscript.

Funding

This work is funded in part by the Dominion Fellowship, the Richard T. Cheng Endowment at Old Dominion University and NSF grants: CCF-1439079, CNS-1828593.

Availability of Data and Materials

The software developed as part of this work is not currently publicly available but will be in the future.

References

[1] K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali, "A load balancing framework for adaptive and asynchronous applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, pp. 183–192, February 2004.

[2] P. Thomadakis, C. Tsolakis, and N. Chrisochoides, "Multithreaded runtime framework for parallel and adaptive applications," *Engineering with Computers*, Jul 2022.

[3] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2018.

[4] N. Chrisochoides, "Multithreaded model for the dynamic load-balancing of parallel adaptive pde computations," *Applied Numerical Mathematics*, vol. 20, no. 4, pp. 349–365, 1996.

[5] C. Tsolakis, P. Thomadakis, and N. Chrisochoides, "Tasking framework for adaptive speculative parallel mesh generation," *The Journal of Supercomputing*, vol. 78, pp. 1–32, 2022.

[6] K. Garner, P. Thomadakis, T. Kennedy, C. Tsolakis, and N. Chrisochoides, "On the end-user productivity of a pseudo-constrained parallel data refinement method for the advancing front local reconnection mesh generation software," in *AIAA Aviation Forum 2019*, (Dallas,Texas), June 2019.

[7] M. Balasubramaniam, K. Barker, I. Banicescu, N. Chrisochoides, J. Pabico, and R. Carino, "A novel dynamic load balancing library for cluster computing," in *Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pp. 346–353, 2004.

[8] J. Nakashima and K. Taura, *MassiveThreads: A Thread Library for High Productivity Languages*, pp. 222–238. Berlin, Heidelberg: Springer, 2014.

[9] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, 2008.

[10] K. Taura, K. Tabata, and A. Yonezawa, "Stackthreads/mp: Integrating futures into calling standards," in *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, (New York, NY, USA), p. 60–71, Association for Computing Machinery, 1999.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[12] "Advanced hpc threading: Intel® oneapi threading building blocks."

[13] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos, "A taxonomy of task-based parallel programming technologies for high-performance computing," *J. Supercomput.*, vol. 74, p. 1422–1434, apr 2018.

[14] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel programming in split-c," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, (New York, NY, USA), p. 262–273, ACM, 1993.

[15] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to upc and language specification," tech. rep., UC Berkeley, 1999.

[16] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high performance java dialect," *Concurrency - Practice and Experience*, vol. 10, pp. 825–836, 1998.

[17] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perf. Comp. Appl.*, vol. 21, pp. 291–312, Aug. 2007.

[18] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, (New York, NY, USA), pp. 6:1–6:11, ACM, 2014.

[19] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 66:1–66:11, IEEE Computer Society Press, 2012.

[20] T. Beri, S. Bansal, and S. Kumar, "The unicorn runtime: Efficient distributed shared memory programming for hybrid cpu-gpu clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1518–1534, 2017.

[21] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput.: Pract. Exper.*, vol. 23, p. 187–198, feb 2011.

[22] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), p. 456–471, Association for Computing Machinery, 2013.

[23] P. Thomadakis, C. Tsolakis, and N. Chrisochoides, "Multithreaded runtime framework for parallel and adaptive applications," *Engineering with Computers*, vol. 38, pp. 4675 – 4695, 2022.

[24] N. Chrisochoides, "Parallel run-time system for adaptive mesh refinement," in *Solving Irregularly Structured Problems in Parallel* (A. Ferreira, J. Rolim, H. Simon, and S.-H. Teng, eds.), (Berlin, Heidelberg), pp. 396–405, Springer Berlin Heidelberg, 1998.

[25] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, (New York, NY, USA), p. 21–28, Association for Computing Machinery, 2005.

[26] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The mvapich project: Transforming research into high-performance mpi library for hpc community," *Journal of Computational Science*, vol. 52, p. 101208, 2021. Case Studies in Translational Computer Science.

[27] "Ecp proxy applications." https://proxyapps.exascaleproject.org/, 2019. [Accessed: 2022-11-28].

[28] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *2009 International Conference on Parallel Processing*, pp. 124–131, 2009.

[29] "Sw4lite." https://github.com/geodynamics/sw4lite, 2019. [Accessed: 2022-02-10].

[30] F. Drakopoulos, C. Tsolakis, and N. P. Chrisochoides, "Fine-Grained Speculative Topological Transformation Scheme for Local Reconnection Methods," *AIAA Journal*, vol. 57, pp. 4007–4018, July 2019. Publisher: American Institute of Aeronautics and Astronautics.

[31] P. Thomadakis and N. Chrisochoides, "Towards performance portable

programming for distributed heterogeneous systems." arXiv:2210.01238, 2022.

[32] P. Thomadakis, A. Angelopoulos, G. Gavalian, and N. Chrisochoides, "Using machine learning for particle track identification in the clas12 detector," *Computer Physics Communications*, p. 108360, 2022.

[33] P. Thomadakis, A. Angelopoulos, G. Gavalian, and N. Chrisochoides, "De-noising drift chambers in clas12 using convolutional auto encoders," *Computer Physics Communications*, vol. 271, p. 108201, 2022.