Multithreaded Runtime Framework for Parallel and Adaptive Applications

Polykarpos Thomadakis, Christos Tsolakis and Nikos Chrisochoides

CRTC, Department of Computer Science, Old Dominion University, Norfolk, VA, USA.

Abstract

This paper presents a new design of the Parallel Runtime Environment for Multi-computer Applications (PREMA). This framework provides large-scale applications with one-sided communication, remote method invocations and a global namespace on top of transparent object migrations for implicit load balancing, scheduling, and latency hiding through an easy-to-use interface, for exascale-era platforms. The framework has been augmented with multi-threading, separating communication and execution into different threads to provide asynchronous message reception and instant computation execution. It allows for implicit parallel shared and distributed memory computations and guarantees correctness through an interface for assigning access privileges to parallel tasks while monitoring the load of the system and performing migrations. Scheduling and load balancing are enhanced by introducing custom intra-node schedulers and the ability to perform concurrent migrations. The motivation for the development of the runtime system is to provide a dynamic runtime for adaptive and irregular parallel applications like adaptive mesh refinement. Evaluating the system on such an application indicates an overall performance improvement of up to 50%, compared to static load balancing, with an overhead of less than 1% when using up to 270 computing nodes (i.e., 5600 cores); an improvement achieved by retaining a better work-load distribution among the execution units. Evaluations with a communication-intensive application with static load balancing reveals that no significant overhead is added despite the additional bookkeeping needed to monitor the load of each processing element.

1 Introduction

Irregular applications pose challenges in handling scheduling and load balancing of large, distributed computing platforms. Their unstructured, and dynamic computation and communication patterns make it very difficult to a priori predict their workflow and computational profile at compile time. An important characteristic is that the workload of individual work units may vary drastically throughout the application's execution, which makes it difficult to statically infer workload propagation. Another characteristic that impedes such tasks is the prohibited use of global synchronization points which could be used as points for information dissemination, decision-making, and data redistribution. Global synchronization hinders the performance of asynchronous applications due to the increased overheads they introduce for very large, massively scalable platforms.

The ever-increasing intra-node parallelism further exacerbates the problem. One can ignore the hardware structure and use message-passing to explore concurrency at both inter-/intra-node levels (by mapping one MPI rank per core), to simplify the programming model. A major issue in this approach is that cores on the same node execute on distinct virtual memory address spaces. Scheduling and load balancing require increased overheads for redistributing data and workload, due to data marshaling and copying, while decision-making implies expensive communication and coordination. Another approach is a hybrid programming model where inter-node concurrency uses message passing (e.g., MPI), and intra-node concurrency utilizes multi-threading (e.g., OpenMP). This approach can improve performance since it allows cores in the same node to share workload and data directly. However, it requires adapting the application to a new, more complex programming model. Applications need to be adjusted to handle synchronization between threads, maintain correctness, and avoid issues like race conditions and deadlocks.

In this paper, we present the Parallel Runtime Environment for Multicomputer Applications (PREMA) [1, 2]. PREMA implicitly handles inter- and intra-node scheduling and load balancing and facilitates an approach between the two (i.e., flat-MPI and hybrid programming paradigms), utilizing their advantages and overcoming their disadvantages. It implements one-sided communication and remote method invocations similar to Active Messages [3] which are enhanced with a globally addressable namespace, transparent object migrations, and message forwarding, as opposed to static global address-space implemented in languages like Split-C [4] and UPC [5]. On top of this infrastructure, it provides a set of load balancing policies and an API that allows for easy-to-use/implement custom inter/intra-node scheduling and load balancing without modifying the application. All of the above features are supported by a low-level multi-threading subsystem.

PREMA adheres to two principles. First, application data are encapsulated in *mobile objects*. Second, communication patters follow an object-oriented paradigm where mobile objects communicate with each other directly. Thus, the runtime system abstracts the underlying structure of hardware, processing

elements, and memory spaces. Following this Mobile Object Driven (MOD) programming model, it can utilize resources across shared and distributed memory and perform efficient scheduling and load balancing while presenting a uniform programming interface that implicitly handles issues related to concurrency. Messages that trigger remote or local method invocations are assigned access privileges. Access privileges express how method invocations may use a mobile object (exclusively, shared), providing information about data dependencies, which is used to maintain correctness and extract parallelism.

The runtime system offers implicit, 2-level (inter- and intra-node) scheduling and load balancing, as opposed to the previous version [1] that only provided support at the inter-node level. Shared memory scheduling and load balancing are provided through the parallel execution of computations on independent objects or non-conflicting computations on the same set of mobile objects by multiple threads. At the distributed level, load balancing is handled by migrating mobile objects between nodes transparently. The load balancing and scheduling strategies used are isolated in a separate module, allowing users to develop their application focusing only on correctness. At a later optimization step, a user might decide to experiment with different strategies to improve performance without needing to modify the application.

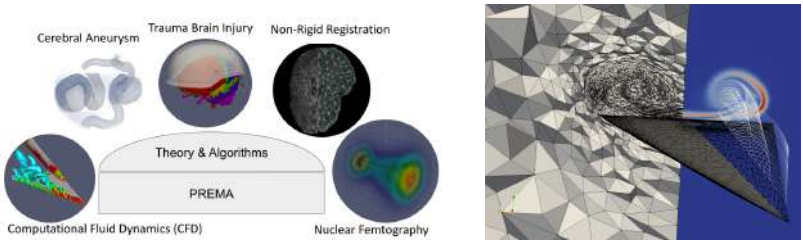
In summary, the contributions of this work include:

- An efficient and scalable runtime system for exascale-era platforms, providing one-sided communication, remote method invocations and a global namespace on top of transparent object migrations, for implicit shared and distributed memory load balancing and scheduling through a uniform interface. This interface allows developers to efficiently encode algorithms without the burden of capturing low-level architectural details, while allowing experimentation with different scheduling policies and the implementation of custom ones.
- An evolutionary methodology for porting legacy MPI systems to multi-core platforms based on the principle of separation of concerns and the lessons learned from this transition

2 Background

In recent years, the field of High-Performance Computing has expanded to include the scaling of virtually any computational process, ranging from engineering and simulations to data analytics and AI applications, with the new applications incorporating elements from the whole spectrum of these fields. The increasing irregularity of such applications, as well as the hardware that they utilize, pose a great challenge for scientists. As the complexity of the applications increases, the demand to efficiently explore more concurrency grows, leading to an extensive rewrite of scientific codes to remain scalable in the new settings. Moreover, the introduction of new technologies as a result of

the trends in modern computer architectures, such as dynamic power management, varying processor speeds, and deep memory hierarchies further increase the burden for developers that not only need to implement the algorithmic part of the application but also need to be up-to-date with the new hardware technologies.



(a) (b)

Fig. 1 Applications that use parallel mesh generation as part of their workflow, ranging from adaptive and irregular Computational Fluid Dynamics to more regular Nuclear femtography applications (a) and an example output of a CFD application taking advantage of the adaptive mesh generation application (b).

PREMA has been developed to separate the concerns of correctness and scalability for applications with an irregular workflow. The application that gave birth to such a need for our case is Parallel Mesh Generation, which is the basis for multiple other application fields. In Figure 1a we present some of the fields that take advantage of PREMA through the use of parallel mesh generation in their algorithm. Figure 1b shows an adaptive mesh resulting from our efforts towards "NASA 2030 CFD Vision"[6] for parallel error-based mesh adaptation for CFD simulations. Using PREMA as an abstraction to implement the algorithm, an application can scale from a single computer with a few processors to a supercomputing cluster with almost no modifications. PREMA offers abstractions for enabling automatic task scheduling and load balancing across shared and distributed memory platforms, while access to local and remote data is provided through the same interface. Moreover, optimizations for new hardware architectures are introduced at this level without affecting the application code. PREMA has been used with parallel mesh generation codes, utilized in CFD applications (see Section 5, and [7]), Nuclear Femtography and Bioinformatics, as well as other applications (see Section 3.3).

3 Software Stack and Programming Model

3.1 Software Stack

PREMA consists of three software layers following the principle of separation of concerns, namely the Data Movement and Control Substrate, the Mobile

Object Layer, and the Implicit Load Balancing layer. This section presents the functionality provided by the three layers.

The *Data Movement and Control Substrate* (DMCS) [8] acts as a thin layer that abstracts the underlying hardware and the communication library (MPI in this case) from the application or the higher-level libraries that constitute PREMA. By using this abstraction, PREMA can be easily ported to a different computing environment by only adapting the DMCS layer. Apart from that, DMCS offers a message-driven programming model by implementing one-sided communication and remote method invocations, similar to the *Active Messages* [3] paradigm. Methods that are supposed to be available for remote invocation are referred to as *remote handlers* in the context of the runtime system, and they need to be explicitly registered as such before being used. This paradigm allows the runtime system to hide remote data access latency behind computations performed by both the system and the application.

The *Mobile Object Layer* (MOL) [9] builds on top of *DMCS* and extends it with a globally addressable namespace while it introduces the construct of *mobile objects*. A mobile object is an abstract, location-independent container implemented by the runtime system to store application data. Data defined as mobile objects can be freely moved to remote destinations while remaining addressable from any node in the system through their unique identifiers, the *mobile pointers*. This functionality is made possible by extending the remote method invocation capabilities of DMCS to target mobile objects directly, wherever they might reside. Thus, applications are encouraged to follow a programming model oriented around mobile objects and invoke computations through messages without knowledge about their locations. We refer to this programming model as Mobile Object Driven (MOD).

A distributed directory maintains the last known locations of all the mobile objects in the local and remote nodes. When sending a message to a mobile object, the object's last known location is retrieved from the sending node's directory, and the message is transmitted there. The target node might not hold the mobile object anymore, in which case the message is forwarded based on that node's directory information. Once a forwarded message arrives at its destination, an update message is sent back to the original sender, updating the object's last known location. There are more location-updating policies available; the *lazy updates* presented in [10] is the one that exhibits the best performance.

The *Implicit Load Balancing Layer* (ILB)[1] makes use of of both DMCS and MOL to provide transparent and implicit data and load migration in the event of load imbalance detection. Based on the observation that no single load-balancing algorithm is optimal for all platforms and applications, the ILB's scheduler is implemented as a separate module that allows plugging-in new policies without any modifications to the application. This design allows developers to experiment with different methods without affecting their application code. Moreover, for more experienced users, ILB provides an API to develop custom load balancing policies.

In ILB, all mobile objects are associated with their pending computations, which constitute their load. Thus, migrating mobile objects will implicitly migrate work-load from one node to another, resulting in load balancing. Since mobile objects contain user-defined data, the application needs to provide callback functions that let the runtime system pack, unpack, and retrieve their sizes to migrate them between nodes. Another set of callbacks may also be provided to pass information about the cost of migrating a mobile object and dynamically calculate its load with respect to its pending handlers.

3.2 Programming Model

PREMA encourages the use of the MOD programming model. It exposes a data-centric design where communications and computations happen between mobile objects rather than processors. Parallelism is achieved by executing handlers simultaneously on multiple nodes, cores, and mobile objects. Applications have their data broken into N chunks, encapsulated into mobile objects, where $N \gg P$ and P is the number of processing elements. This technique is known as over-decomposition and allows greater flexibility in intra/inter-node scheduling and load balancing.

A typical application starts by performing some pre-processing steps and then over-decomposes its data. Then it creates mobile objects for each chunk of data, defines (de)serialization functions, and optionally distributes them to available nodes. The core of the application logic is then expressed through messaging among the mobile objects. Message handlers include computational tasks, the creation of new mobile objects, and data transfers. By adhering to this programming model, task executions are addressed to individual mobile objects, whether they reside on the local or a remote node, abstracting the actual data/hardware mapping from the application. Data encapsulated in a single mobile object are guaranteed to be local, allowing the application to access them directly in the context of handler execution.

Throughout the application's execution, PREMA handles scheduling and maintains the location of mobile objects. Moreover, it monitors nodes' workload, initiating load balancing when necessary, and guarantees that messages are delivered to their destination whether they are local, remote, or in the process of migration. The application is not aware of any of these operations and is developed as if all mobile objects were locally accessed through a unifrom API.

3.3 Adapting to Application Needs

Even though we use scalable parallel mesh generation applications as a case study for evaluation, PREMA is designed as a general-purpose runtime library. PREMA supports different execution models [11, 7] and has been tested with different applications. In addition to several parallel mesh generation methods [7], PREMA is used to implement a seismic wave simulation application (section 5.3.3). In earlier efforts, we used DMCS and MOL (PREMA's low-level

communication substrates) to develop an N-body simulation code [12]. These use-cases include both legacy applications following a black-box approach [7], and new applications developed with PREMA in mind (presented in this work). In both cases, PREMA was successfully used to scale the applications using different scheduling models like the master-worker model and diffusive load balancing.

PREMA offers multiple abstractions at all levels of the runtime system software stack. One can choose to use a subset of them depending on the specific needs of their application as well as their familiarity with the runtime system and expertise with load balancing policies. PREMA's design allows using a subset of its capabilities when there is no need for its extra features. For example, a user that only needs an active messages library can just utilize DMCS without building and studying the other software layers. At a later stage, one might need the global namespace and messages provided by mobile objects. If later implicit load balancing is desired, the application can be easily extended to utilize ILB without much effort. Moreover, a user can choose how the workload of a mobile object is calculated by providing a callback function or a static value and explicitly disable or re-enable distributed load balancing by calling the respective function *ilb::enable_dist_balancing(bool)*.

Another feature that gives more flexibility to PREMA is the ability to use, adapt, and modify load balancing policies that ship with PREMA or implement new ones through the provided API. Since there is no holy grail for load balancing that fits all applications and platforms, this interface can be used to experiment with different methods without the need to touch the application's code. For example, an application expert may prefer to utilize existing load balancing strategies and choose the one that gives the best results for the specific needs. Such a user might feel more comfortable only experimenting with high-level parameters provided by existing strategies to improve performance. An advanced user might try to extend an existing policy to extract more performance, while an expert in load balancing might want to develop a new one. Thus, PREMA can accommodate users with different needs, skills and experiences.

Figure 2 shows the abstract class (API) that a user can extend to implement virtually any new policy. Four methods need to be implemented to allow the new policy to communicate with PREMA and receive updates about the system state. In summary, these methods provide: (1) the trigger for starting a new load balancing phase (*dist_balance()*), (2) updates from PREMA to the policy regarding the load of each mobile object (*notify()*), (3) abstractions to implement shared memory scheduling (*push(), pop()*). In the appendix we present the implementation of two simplified load balancing policies, a master worker and a diffusive scheme.

```
class ilb::scheduler
{
public:

   // Initialize required structures here
   scheduler() {};
   // Free structures here when application terminate
   virtual ~scheduler() {};

   // This method is periodically called by PREMA to check and start distributed load
        balancing when needed
   virtual void dist_balance() = 0;

   // Notify scheduler about a change to mo's load and update node level load
   virtual void notify(ilb::mobile_object mo) = 0;

   // Thread with ID thread_id tries to get some handler to execute
   virtual ilb::handler* pop(int thread_id) = 0;

   // Thread with ID thread_id tries pushes a new handler to the scheduler
   virtual void push(int thread_id, ilb::handler* hdlr, ilb::mobile_object mo) = 0;
}
```

Fig. 2 PREMA's minimal interface to declare a new load balancing policy.

4 Leveraging Multi-core Architectures

4.1 Multi-threaded Design

In order to efficiently handle multi-core platforms, PREMA is natively integrated with a low-level multi-threading library. Utilizing multi-core platforms improves PREMA's performance on both communication and computation aspects by making them truly asynchronous, increasing the opportunities to overlap them and hide latencies. Moreover, it allows applications to seamlessly utilize multi-core platforms without explicitly dealing with concerns raised by shared memory concurrency.

4.1.1 Data Movement and Control Substrate

We perform the integration with the low-level multi-threading library at the level of DMCS which is the basis of PREMA's software stack. DMCS consists of three components, the **handler-execution**, **communication** and **application** components (Fig.3). Note that this structure remains internal to DMCS and is not exposed to the application. An application implicitly utilizes these components by requesting local or remote handler invocations. The functionality handled by each component is described below.

The **communication component** is responsible for handling operations destined for remote nodes. It consists of a loop responsible for sending/receiving messages as well as signaling other components of PREMA regarding message requests related to/targeting them. We offer two configurations; either a dedicated thread is used to perform the communication, or any of the threads in the handler execution component is assigned this task periodically. When a thread of the two other components desires to send a message, it pushes

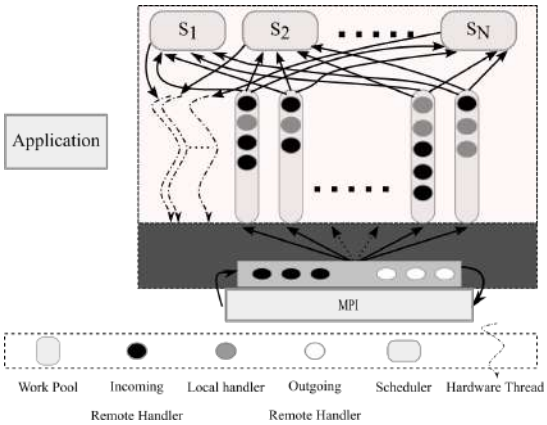


Fig. 3 The DMCS execution model. Each hardware thread is associated with a scheduler $S_1, ..., S_N$ that might use one or more private or shared work pools consisting of handlers invocation requests. A handler created locally can target the local or a remote node. The computation component (dark gray box) keeps a list of incoming handlers which it will assign to one of the work pools and outgoing handlers that will be delivered thought the communication library

a request to its respective list which is handled appropriately by the communication component. Ongoing messaging requests are maintained by this component and their progress is checked periodically. Once a completed request is found, it is freed and any thread waiting for its completion is signaled to resume its execution.

Two types of messages are used, *fixed* and *split-phase*, based on whether the size of the message exceeds a predefined threshold. When the size of the data to be sent is less than this threshold, a fixed size message will be used regardless of the data size. This message will contain a header needed from the runtime system, followed by the actual data that will be passed to the remote handler. When the size of the data is greater than the threshold, split-phase messages are used. Split-phase messages consist of two parts; a fixed-size message and a variable-sized message. The former contains the message header that will inform the receiver about the existence of a second message and the actual size of its data. The latter contains the actual message data. This information can then be used by the receiver to issue a second receiving operation, eliminating the need to query the network for the size of the next incoming message.

The use of fixed-size messages also allows to preallocate a pool of such messages for each thread that can be used to send or receive remote handlers. Incoming messages will be received into a preallocated message of one of the threads' pools and will be pushed to its list of pending remote handlers until it is scheduled by the handler-execution component. Once the handler is executed, the preallocated message will be pushed back to the pool for reuse. Likewise, a thread that desires to issue a remote method request will use one of the preallocated messages in its pool. Thus, new memory allocations are

avoided for both small and large messages except for the case where the pool runs out of preallocated messages and needs to be resized.

The **handler-execution component** executes the remote method invocation requests. This is the component where the bulk of computations are running and where the parallelism is exploited for application and system-related operations. When a new handler request is issued -either locally or from a remote node- an object containing this request is created by the issuer and is pushed, by the issuer if it's local or the communication thread if it is remote, to a pool of handler requests. The user can decide the number of threads that this component utilizes and whether or not to bind the threads to specific hardware cores. The default number of threads is that of the available cores of the node minus one reserved for the application and optionally one for the communication component, if such configuration is desired. Each thread is associated with a handler request pool and is responsible to service it. If the associated work pool is empty, a thread will employ work stealing with random victim selection [13] to avoid remaining idle and to balance the load in the node. When the thread's work pool is empty and a steal attempt fails, the thread will backoff [14] for an exponentially increasing amount of time (that is constrained to a maximum value) until it succeeds in finding some work, in which case the backoff period is reset to zero. By enforcing this delay both the memory contention and the amount of power wasted is reduced. The scheduling part of this component can be modified through an interface which is mainly utilized by higher-level libraries of PREMA as discussed later.

The **application component** consists of the main function of the application. In this component, the application can start the runtime system, register the methods to be run remotely, define the number of threads to be used in the handler-execution component, orchestrate the logic of the application, etc. Once all the preprocessing steps have been completed, the application can issue remote handler requests from this component to produce work for the other components. Once the former two components are active, the application thread can be released to the runtime system, contributing to the handler-execution threads work until a condition (e.g the current phase of the algorithm has finished) is met.

Before adopting DMCS to a multi-threaded design, each available core ran on a separate instance/process (i.e., similar to running one MPI rank per core). This design created the issue that each processing element (PE) could only work on its own tasks, having no access to the workload/handlers of others in the same node. Thus, being susceptible to resource under-utilization in cases where only a subset of the PEs is the target of handler executions. Moreover, because all of its operations (i.e., communication, handler execution, application workflow) ran on a single thread/core, it required explicitly calling a polling function that handled the progress of all operations. Even though this design provided a sequential execution model that simplified the process of writing applications on top of it, it could not take advantage of the benefits

provided by shared memory architectures. The new design addresses those issues and further improves its performance.

In summary, the multi-threaded design allows all processing elements in a single computing node to access any pending handler invocations targeting this node. Parallelism is explored by executing multiple handlers concurrently while resources are utilized efficiently by allowing individual threads to steal their peers' work. Moreover, no explicit polling operation is required since progress is handled implicitly by background threads. Implementing intra-node work-sharing/stealing in the previous single-threaded design would require expensive inter-process communication and data copies, increasing the overheads sustained by orders of magnitude. DMCS does not offer abstractions to handle issues related to concurrency implicitly; instead, those are handled by MOL/ILB. This choice was made to keep DMCS as lightweight as possible since the inclusion of such mechanisms would increase the critical path of one-sided communication and handler executions.

An important lesson learned from porting a low-level communication substrate, like DMCS, to a truly multi-threaded model supported by the hardware is that incorporating message passing with intra-node parallelism can significantly affect the latencies incurred. We found that maintaining per thread message pools and funneling communication operations to a single thread at a time can help to mitigate such effects. Moreover, implementing only the necessary functionality at this low level helps to avoid overheads that may arise when aiming for ease of use and forcing correctness (e.g., checking whether the arguments of a message are valid). Finally, this decision also adheres to the principle of separation of concerns that is maintained throughout the design of PREMA.

4.1.2 Mobile Object Layer

Running mainly inside DMCS remote handlers, the MOL leverages the multi-threaded DMCS layer and is amplified with the ability to perform its operations in parallel. This allows running multiple remote handlers that target the same or different mobile objects concurrently and even initiate parallel object migrations from a single node. However, these operations require access to the distributed directory and since they can run in parallel, they have to be performed in a thread-safe manner.

To avoid the contention issues that could be created by using a lock and a simple C++ STL map, a custom hash table with chaining[1] is used instead. This approach allows elements to be inserted in different entries of the table safely but still exhibits possible race conditions for colliding elements. Using a lock per table entry could solve this problem, however, it is too conservative to require a lock for each access since the majority of accesses are expected to be lookups that do not modify the directory. Instead, for insertions, the atomic operation compare and swap (CAS) is used, while for deletions the element is marked as invalid and is reused instead of being removed from the list. The

[1] in case of collisions a list is used to keep the colliding elements in the same entry of the table

combination of CAS and no deletions results in a thread-safe list that does not suffer from the ABA [15] problem.

MOL message handlers are assigned access privileges that express how they may use the target mobile object, exclusively (e.g., write) or shared (e.g., read). PREMA utilizes this information to extract parallelism and maintain correctness. Moreover, MOL guarantees the execution order of message handlers issued to a mobile object from within a single handler context. For example, a mobile object that issues multiple message handlers to the same target is guaranteed that all of its issued messages will execute in order. Handler invocations that target the same mobile object with exclusive access are serialized while handlers with shared access are allowed to run concurrently. When targeting different mobile objects, handler invocations are non-conflicting whether they desire shared or exclusive access since data enclosed in mobile objects are independent by definition.

To maintain ordering, messages are assigned an identifier representing the version ($lversion_i$) of the information in the local copy of the distributed directory and an increasing sequence number ($lseq_i$) maintained by each node i for each mobile object in the distributed directory. When a new message handler arrives from node i, $lversion_i$ and $lseq_i$ are checked against the receiver j information $lversion_j$, $lseq_j$ and are only considered for execution if they are consistent. If it is determined that preceding messages are still on the way, the out of order message are set aside until earlier messages have been received. Once messages are in order, their access privilege is finally examined to make decisions about the execution. For operations that do not run inside MOL handlers but may also target mobile objects, a set of locks is introduced that offers the same functionality for exclusive and shared access. An example of such need is when a mobile object has to be migrated; it has to be locked exclusively first, so that no handler tries to access it in an inconsistent state, and then packed and sent to its new location.

In summary, the multi-threaded design allows the same functionalities provided by the single-threaded model, while utilizing all PEs of a computing node. This enables implicit shared memory load balancing which would be even heavier to implement in MOL using distributed memory paradigms because of the use of over-decomposition that would require hundreds of mobile objects in a single node to communicate with each other through inter-process communications. Applications can take advantage of the added concurrency by simply annotating message handlers with access privileges. Thus, by utilizing MOL and the MOD programming model, an application can efficiently run on multi-core platforms without explicitly dealing with concerns like, maintaining consistency and avoiding critical sections.

The key to achieve efficiency in a data-flow-centric system, like MOL, is the implementation of the lock-free distributed directory. At the distributed memory level this problem was handled by using partial independent copies of the directory, holding possibly out of date information that was updated lazily in order to avoid excessive communications. At the shared memory level, we

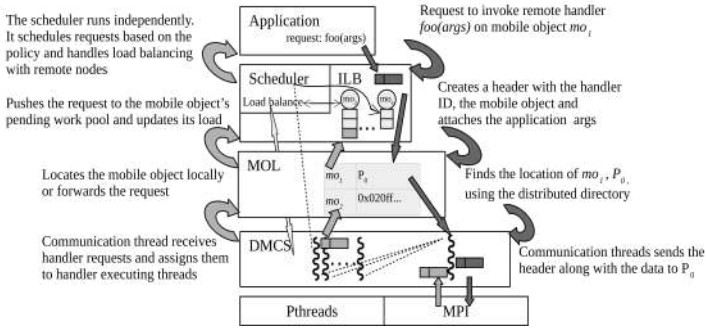


Fig. 4 High level representation of DMCS, MOL and ILB interactions.

used a lock-free hash table that allowed quick, concurrent access to different mobile objects. By allowing low-overhead concurrent access to different mobile objects and keeping information related to each mobile object in a per object structure, instead of a global structure, one avoids thread contention unless it becomes absolutely necessary (i.e., handlers targeting the same mobile object).

Also, because of over-decomposition, the chances of having too many requests for concurrent access to a single mobile object is expected to be relatively low, thus, contention is also expected to be low. Even when that is not the case, using atomic operations instead of locks (when possible) mitigates performance issues.

4.1.3 Implicit Load Balancing

ILB is responsible for scheduling and load balancing at both the shared and distributed memory levels. At the distributed memory level, ILB maintains the workload of each mobile object and, consequently, the workload of each node. Also, it encapsulates the operations required to maintain load balance across nodes, including information dissemination, bookkeeping, and decision making. The operations requiring inter-node communication are built utilizing the DMCS layer and can run in parallel, allowing concurrent mobile object migrations from the same or different nodes. Running such operations in parallel improves response time, makes it easier to overlap them, and enables more sophisticated load balancing schemes. On the other hand, it increases the complexity of developing new load balancing algorithms; however, this complexity is constrained to the scheduler module, which a typical user does not need to modify or adapt unless a new policy needs to be developed. Mobile object-specific locks are convenient in this context since policies can use them to guarantee that no handler is running on a mobile object before trying to migrate it. After exclusively locking an object, it is safe to pack and migrate it to another node along with its workload. By design, the scheduling interface does not need to check whether the pending handlers in its pools target mobile objects that might have migrated. Such handlers are invalidated during the packing process and are ignored when popped from the scheduling interface.

The ILB scheduler is periodically called by the handler-executing threads once they have finished pending work in the DMCS and MOL layers. Before initiating distributed load balancing, a scheduler would usually prefer to execute the local workload. The workload consists of pending ILB handlers residing in the local work pools of the scheduler (and internally in the respective list of individual mobile objects). ILB handlers are assigned exclusive or shared access, holding the same principles as discussed for the MOL. When a handler is safe to be executed (i.e., it is in order and does not conflict with others), it is pushed to the local scheduler work pools through the respective callback (see section 3.3). By enforcing all dependencies to be resolved before the local scheduler is informed about a new handler, we relieve the scheduling policy from having to maintain execution order correctness. In contrast, the load of local mobile objects is updated even before the respective handler is able to execute. This allows the scheduler to be informed about the whole workload of a node regardless of whether it has dependencies or not.

A high-level representation of the interactions between the different layers is presented in Fig. 4. When a new message is available, it is received from the communication component of DMCS and assigned to one of the handler-executing threads (light gray arrows). If it is a load balancing request, it is routed directly to ILB. If it is an application request targeting a mobile object mo_2, it will pass through the MOL to look up the object's location using the distributed directory. If the mobile object is not local, the request is forwarded to the object's last known location. Otherwise, the request is passed to ILB, which will update the load of the respective mobile object and push the request to its work pool. The ILB scheduler, which runs independently, can schedule some requests or initiate load balancing by sending such requests to other nodes using the DMCS interface. When the application needs to invoke a handler remotely to some mobile object mo_1 (dark gray arrows), a new request is created by ILB with relevant information attached. The request is passed to MOL, which finds the location of the mobile object and sends the request using the DMCS layer.

Last but not least, ILB offers the *ilb_multicast* operation. This handler execution request can be sent to multiple mobile objects and will only start running when all of those mobile objects reside in the same node. Its inputs are the mobile objects needed to be in the same node, a buffer for the arguments, and the index of the mobile object on which it should start running. Once called, it migrates (if needed) the mobile objects on the same node, locks them so that they cannot be moved by another handler, and schedules the requested handler. This operation was implemented to support applications that may require access to multiple mobile objects in a single handler before they can start their computations like the one presented in [16]. An effort that uses this functionality can be found in [7] but is out of the scope of this paper.

In summary, the multi-threaded design of ILB allows efficient utilization of multi-core nodes while monitoring their workload and providing load balancing. Using an easy-to-use API, one can quickly implement custom shared and

distributed memory (2-level) scheduling and load balancing policies without the need to handle dependencies or reason about mobile object locality. Work units that are available at the scheduling module are either safe to execute (i.e., have no dependency conflicts) or have already been invalidated by the runtime. As opposed to a single-threaded design, load balancing among cores in the same node does not need to go through the process of mobile object (de)serialization and expensive rounds of synchronization. Instead, just passing a pointer between threads is enough to share workload. Moreover, it enables concurrently migrating multiple mobile objects while computations are also in progress, thus, hiding and overlapping latencies in a single node.

By utilizing the mobile object-specific locks, one is able to safely and correctly maintain the load of individual objects and monitor their pending handlers in the respective lists by only serializing conflicting operations on the same mobile object. The lessons learned from implementing a 2-level implicit load balancing framework is to separate the concerns between local and global decision making as well as between correctness and performance. Monitoring the information for each level separately allows for easier, cleaner and more efficient implementation. Handling correctness at a lower level also improves performance and mitigates errors. Moreover, maintaining workload and pending handler information in a per-object fashion rather than a global one improves performance and avoids excessive contention.

4.2 Correctness

Formally proving correctness in a complex system like PREMA is a difficult task. Formal methods, Markov chains, discrete event simulations, or Petri nets could be some approaches to tackle this problem. Such a systematic study for correctness is left as future work and is outside the scope of this paper. As an initial step towards proving correctness, we try to demonstrate that our system is free of inherent deadlock problems based on its design.

Starting from the lowest layer, DMCS, the communication component is implemented using only non-blocking MPI calls to avoid deadlocks in the two-sided communication model. The message receiving part of the component uses *MPI_ Iprobe()* to check for messages from any source. Once an incoming message is found, the respective memory is allocated, and the receiving call is issued safely. The sending part also uses asynchronous sending operations. Progress for both sending and receiving is checked periodically between or in parallel to method invocations. The asynchronous messaging operations, along with the restriction that only one thread per MPI rank can send/receive messages at a time, guarantee that messaging will not lead to deadlocks.

In the threading component, threads are associated with thread-safe work pools that hold handler invocation requests and share their work through stealing. To guarantee correctness, handlers encapsulate tasks that do not rely on a future handler invocation for progression. Handlers can issue blocking messaging calls, which internally progress message-passing, and use mutual exclusion (locks) if they are acquired and released in the context of a single handler

(i.e., no inter-handler lock-unlock) but cannot block waiting for a handler that has not yet started executing. For example, it is safe for multiple handlers to compete for the same lock as long as they release it before completing their execution. This restriction guarantees that all handlers complete at some point, and there is no case of deadlocks.

The MOL builds on top of DMCS' handler threads and follows the same restrictions. In [9] it is formally proven that MOL, with a single thread, is always able to keep this information up to date and consistent no matter how many migrations take place in the distributed system. We build upon this for the multi-threaded version to maintain correctness. As presented in section 4, we use a thread-safe hash table to track mobile objects and maintain atomic access per mobile object entry. Atomic access guarantees that information like messaging sequence numbers and directory version identifiers are checked and updated safely in the presence of multiple threads. Moreover, handlers targeting mobile objects are associated with an access type that lets users define mutual exclusion at a higher level, leaving PREMA to handle the correct submission of conflicting handlers. The information about the access type of a handler executing on a mobile object is also maintained in the distributed directory entry. When a new handler is about to be executed on a mobile object, its access type is checked against the current mobile object state. If another handler is already running on the object with a conflicting access type, the new handler is suspended in a thread-safe list of waiting tasks, maintained per mobile object. When the current handler completes, it resubmits its dependent handlers back to the main work pool. By utilizing access types, PREMA can guarantee that no conflicting handlers will run concurrently, relieving the application from this burden and improving performance.

The ILB utilizes DMCS and MOL to route handler invocation requests to the appropriate mobile object and inject them into its thread-safe pool of pending work. Updates to the load balancing policy are also triggered in this process, constraining the policy implementations to the same restrictions as handler invocations. ILB utilizes the mobile object-specific locks of MOL to guarantee that a mobile object cannot migrate while there is at least one handler executing on it. Respectively, it also guarantees that while a mobile object is in the process of packing and migration, no handler can start running on it. Moreover, handlers targeting migrated mobile objects are automatically invalidated, removing this burden from the scheduling policy. Thus, given that the handler invocations and the load balancing policies are implemented under the above restrictions, ILB and PREMA are free of inherent deadlock situations.

In summary, arguments can be made for the absence of inherent deadlocks at the communication, handler execution, message ordering/forwarding and scheduling/workload-related bookkeeping. However, a complete formal proof of the correctness is beyond the scope of this work.

5 Performance Evaluation

5.1 Experimental Setup

The following experiments have been conducted on a computing cluster consisting of 190 Intel(R) Xeon(R) (E5-2660 v1-2, E5-2670 v2, E5-2698 v3, E5-2683 v4) computing nodes. Each node has two CPUs of 16-32 threads in total, 128 GB of memory, and runs Red Hat Linux. We used the MPICH 3.1.3 MPI implementation as the communication library and gcc 6.3.0 for compiling. The SW4lite proxy application was run on a newer cluster with Intel(R) Xeon(R) Gold 6148 @ 2.4 GHz CPUs of 40 cores each, using OpenMPI 3.1.4.

5.2 Communication

We first evaluate the performance of the communication-related functionalities of PREMA. Communication is an important part of the runtime system since PREMA adopts a message-driven execution model. We evaluate the performance of each layer of PREMA using a simple ping-pong benchmark and compare the results with those of an MPI implementation. For DMCS, we use two processes residing on two different nodes; process 0 sends a remote method invocation request of size X to process 1 and blocks waiting for the response. On arrival, the remote method executes and sends a request of the same size back to process 0. This request unblocks process 0 and triggers it to send the next message. This pattern is repeated 1000 times for each message size, and the average latency/bandwidth is reported. For MOL and ILB, the procedure is the same, but the messages are sent between two mobile objects, which reside in processes 0 and 1, respectively.

In Fig. 5a one can see that DMCS, MOL, and ILB add a roughly fixed amount of overhead to the latency that is independent of the message size. The performance of each layer is bound by the performance of lower layers and the performance of MPI. Since the overhead is stable, its effect is more noticeable in smaller messages where the MPI time is low, and the overhead is a significant percentage of the overall time, as seen in Fig. 5b. The effects on the bandwidth seem to be less significant for both small (Fig. 5d) and large messages (Fig. 5c). ILB experiences the highest penalties since an ILB message has to go through DMCS and MOL and be registered with the load balancing module before being scheduled. However, we believe the overhead added is acceptable given the provided functionality.

As mentioned before, DMCS uses pools of preallocated, fixed-size messages which consist of the headers required by the remote handler requests. By maintaining those pools, the overhead accountable to memory allocations is reduced, and querying for the size of incoming messages is avoided. As a result, the latency for initializing the message delivery process is reduced and turns out to be relatively stable among different invocations. When the arguments of a handler are small enough to fit inside a fixed-size message, they

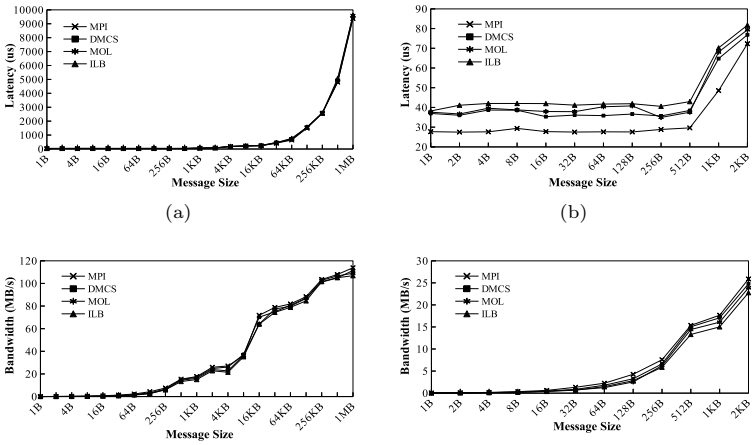


Fig. 5 Ping-pong measurements for all three layers of PREMA compared to MPI. Comparison of (a) latency, (b) latency for small messages, (c) bandwidth and (d) bandwidth for small messages.

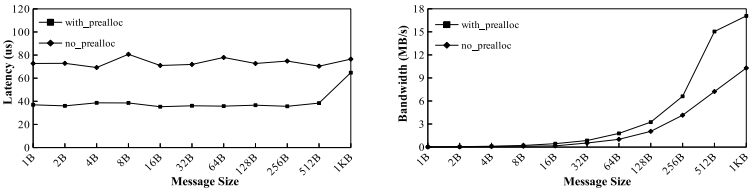
are copied into it; otherwise, they are sent as a separate message. In the former case, the receiver has the preallocated messages ready to receive, while in the latter, the buffer can be allocated, and the receiving call can be issued as soon as the headers are received, even before the message with the actual data has been sent. This implementation helps to overlap the time taken from the sender to send the second message with the time it takes for the receiver to prepare for the delivery.

Figure 6 shows the advantage of using preallocated, fixed-size messages. Small messages (smaller than 2KB) highly benefit from this optimization whether the handler arguments are copied into the header (up to size 512B for this case) or sent as a separate message (1KB message). Using preallocated messages reduces latency (Fig. 6a) by almost 50%, compared to not using them (39 compared to 77μs). Furthermore, the bandwidth (Fig. 6b) is also significantly affected for small messages larger than 64B with up to 100 percent improvement. For messages larger than 2KB, the performance of the two approaches is similar because the cost of sending the messages themselves becomes the dominant factor.

5.3 Load Balancing

5.3.1 Synthetic Benchmark

Next, we evaluate the performance of PREMA in terms of load balancing, overall application runtime, and the overhead imposed by the runtime system. We start with a simple synthetic benchmark to test the system in a fully controlled and isolated environment. Using such benchmarks allows for avoiding



(a) (b)

Fig. 6 Ping-pong measurements for DMCS without preallocated messages. The effect of using preallocated messages in (a) latency and (b) bandwidth. The performance for messages larger than 1KB is comparable to the optimized version and is not shown here.

any unexpected behavior that a real application could demonstrate that could affect the performance of the system.

The benchmark begins by creating mobile objects on process 0 and dispersing them to the available cores; computations are then invoked on each of the mobile objects via PREMA's messaging mechanism. When all computations on a mobile object have been completed, a notification is sent back to process 0. Once all completion notifications have been received, the benchmark terminates. We assign ten mobile objects to each available core and specify a weight (workload) from two categories, light and heavy, for each mobile object. The average execution time of a heavyweight mobile object is 2.5x the execution time of a lightweight, and 20 percent of the mobile objects are assigned to the heavy category. Each instance of PREMA consists of the same amount of cores, and we employ a diffusive load balancing algorithm to evaluate its performance.

For the MPI version of the benchmark, we replace the mobile objects with plain data objects and perform only static load balancing. Once an MPI rank has received all of its data, it will execute the computations for all of them and then terminate. Note that even though PREMA uses one core exclusive for message handling, for fairness, this core is counted as an available core when calculating the number of mobile objects to distribute to each node since the MPI version can utilize all cores of the node for computations.

The performance comparison for the benchmark is shown in Fig. 7. The workload distribution achieved indicates the impact of using PREMA compared to the MPI implementation. Figures 7a and 7b compare the workload distribution of 320 cores using plain MPI (320 ranks) and PREMA (10 ranks (nodes) of 32 threads each). In Fig. 7a the heaviest workloads have been gathered roughly to the first 130 cores for an overall running time of 683 seconds. Fig. 7b shows the results after porting the benchmark on top of PREMA, where the workload has been redistributed equally among the available cores decreasing the overall running time to 495 seconds, an improvement of 27.53 percent.

Figures 7c and 7d show the performance of the same benchmark when we quadruple the number of cores and work units. The pattern for the workload distribution remains the same, and as such, we see the same performance

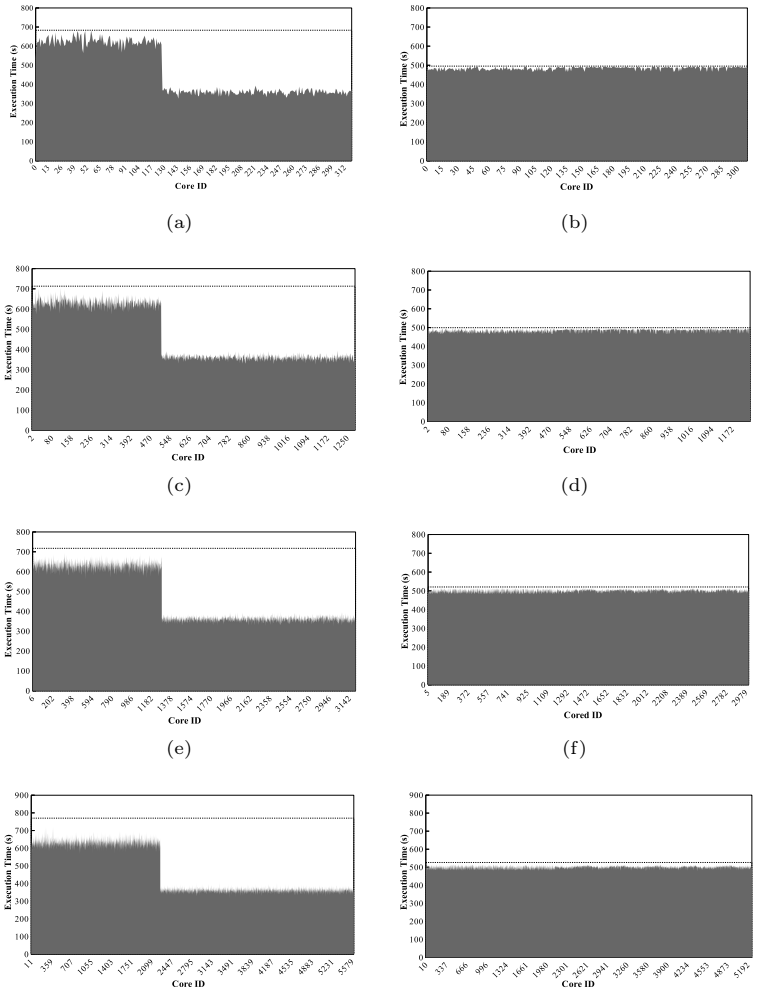


Fig. 7 Per core work-load breakdown running the synthetic benchmark, for MPI with (a) 320, (c) 1280, (e) 3200, (g) 5600 cores and PREMA with (b) 320, (d) 1280, (f) 3200 and (h) 5600 cores. Each core is a different MPI instance (rank) for the MPI cases. For the cases with PREMA one core per instance is reserved for communication, (b) and (d) have one instance per 32 cores while (f) and (h) have an instance per 16 cores. The black dashed line indicates the overall running time including initialization computations and termination time.

breakdown as before. Fig. 7d shows that PREMA is not affected at all by the vast increase in the number of cores and tasks that it needs to balance. Even though the overall runtime slightly increases, it is not a penalty from PREMA itself but by the initialization stage of MPI and the distribution of the initial workload of the problem. This is made more apparent by looking

at the MPI case, where the dashed line is further higher from the computations bars since many more ranks need to be initialized. Figures 7e, 7f and 7g, 7h present the results for 3200 and 5600 cores, where the size of the problem has increased accordingly. However, for these cases, PREMA uses 16 cores per instance instead of 32, which increases the number of PREMA instances, as well as the number of cores reserved for communication. In other words, there are fewer cores for computations per hardware node than before (2 communication cores vs 1 communication core per node). Despite these modifications, PREMA maintains a fair workload distribution and is not affected much by the MPI initialization step (mainly because far fewer MPI ranks are initialized). The overall improvement for the two cases is 27.42 and 31.63 percent, respectively. Table 1 presents the overhead of PREMA, the minimum and maximum refinement time, which is also the factor that impacts the overall running time the most. The minimum/maximum refinement time corresponds to the running time of the worker that finished all of its tasks at the earliest/latest, respectively. The effectiveness of the dynamic load balancing can be noticed by how the variance between these two values has been reduced compared to the variance in the MPI version while maintaining a very low overhead of, on average, 0.05 seconds.

5.3.2 Parallel Mesh Refinement Application

The initial motivation for the development of PREMA is to separate the concerns of performance and algorithmic correctness for parallel mesh refinement applications. To evaluate its performance with such applications, we used our in-house developed tetrahedral mesher CDT3D [17] as an application benchmark.

CDT3D uses as input a triangulation of a Piecewise Linear Complex (PLC) of the domain to be discretized. The basic steps involve: creating a Delaunay Tetrahedralization of the boundary points using Delaunay point insertion, recovering the boundary using topological transformations and edge/face partitioning, and finally refining the mesh. During the mesh refinement procedure, points are created using an Advancing Front type point placement and are then inserted by direct subdivision of the containing tetrahedra. The connectivity of the mesh is then optimized using a combination of topological transformations.

For this experiment, the first two steps (i.e., Delaunay Tetrahedralization and Boundary Recovery) were executed sequentially, as they are needed to bootstrap the mesh and are less time-consuming in comparison to mesh refinement. The resulting mesh was then partitioned into N sub-domains with $N >> \#cores$ using an octree adapted to the mesh density (see Figure 8). The sub-domains are registered as mobile objects and then serialized and distributed among the available processes by PREMA. This decomposition scheme was selected to create sub-domains with a similar number of tetrahedra for a balanced initial workload per sub-domain. The surface of each sub-domain is constrained (i.e., remains unchanged during the refinement), and thus, there is no need for communication between the subdomains. In the

future, we plan to relax this requirement by either allowing modifications on the boundary and consequently introducing a small amount of communication along the boundaries of the sub-domains or by pre-refining the sub-domain boundaries in a separate preprocessing stage.

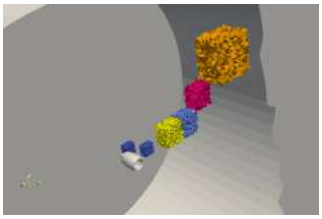


Fig. 8 Subdomains of the initial coarse mesh that is used to bootstrap the parallel refinement process. The input is a surface mesh of a nacelle inside a cylindrical domain. An adaptive octree is used to decompose the volume mesh to equal meshes based on the number of tetrahedra in each leaf. Subdomains are the unit of worked captured by mobile objects in this case and are used to perform dynamic load balancing in the distributed system.

Porting CDT3D on top of PREMA requires only writing the appropriate handlers and callbacks which will initialize and execute the CDT3D mesher. More specifically, the handlers and callbacks used in this experiment are the following:

- Pack/Unpack Sub-domain callbacks, for migrations
- Initialize, for bootstrapping mesher's data structures
- Refine, for the sub-domain refinement
- Callback for calculating the weight (computational cost) of a handler

The application is run with ILB and plain MPI to evaluate the load balancing quality. The preprocessing step is identical in both cases; once the sub-domains are created, they are assigned to the available worker cores. Fig. 9 depicts the performance comparison of ILB versus MPI using the mesh refinement application. Figures 9a, 9b and 9c, 9d show the results when MPI and PREMA are utilizing 640 and 1280 cores respectively. Cores are allocated in the same manner as described in section 5.3.1 using the same mesh size of 30 million elements, over-decomposed into 4.5 thousand sub-domains. The preprocessing and decomposition times are not included in the graph since they are identical for all cases. The same diffusive load balancing policy is used where each PREMA instance can only share its load with a specific subset of the instances available; if no instance in the set has enough load, a new set of instances is picked. The performance improvement that ILB offers compared to the MPI implementation with static load balancing for those two cases is 40.5 and 26 percent, respectively, by dynamically redistributing the available sub-domains to the starving workers. As can be seen from table 2 the maximum refinement time decreases from 340 to 197 seconds (40%), and the difference between the maximum and minimum refinement time is reduced from 326 to

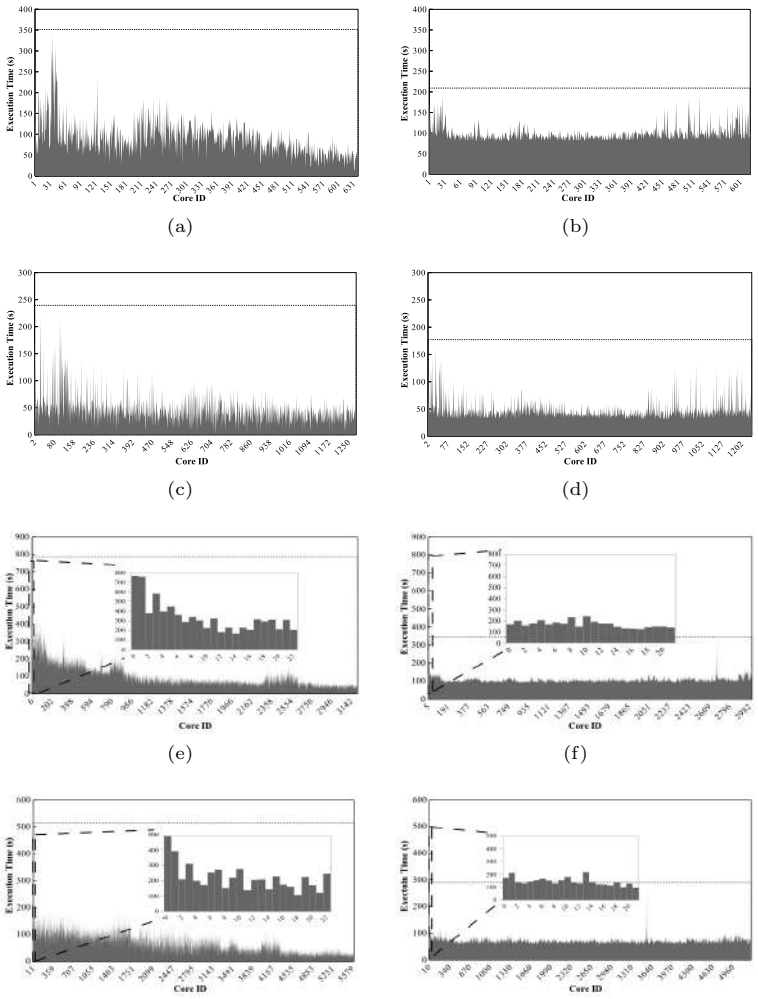


Fig. 9 Per core work-load breakdown running the CDT3D benchmark, for MPI with (a) 640, (c) 1280, (e) 3200, (g) 5600 cores and PREMA with (b) 640, (d) 1280, (f) 3200 and (h) 5600 cores. Each core is a different MPI instance for the MPI cases. For the cases with PREMA one core per instance is reserved for communication, (b) and (d) have one instance per 32 cores while (f) and (h) have an instance per 16 cores. (a), (b), (c) and (d) use an initial mesh of 30 million elements , decomposed into 4.5 thousand sub-domains while (e), (f), (g) and (h) use an initial mesh of size 110 million elements decomposed into 27 thousand sub-domains. The black dashed line indicates the overall running time including initialization, computations and termination time.

116 seconds while the overhead imposed from the load balancing is on average 0.48 seconds per core utilized.

For the 1280 cores case, the difference between the maximum and minimum refinement time decreases from 223 to 131 seconds, while the maximum time

Table 1 Time breakdown and comparison with the plain MPI version for different core allocations using the synthetic benchmark. The overhead presented is per thread utilized. The maximum and minimum refinement time correspond to the time spent by the most and least loaded thread in the lifetime of the benchmark. Maximum refinement time dominates the overall running time.

#cores	PREMA overhead (sec)			Min refinement (sec)		Max refinement (sec)	
	Max	Min	Avg	PREMA	MPI	PREMA	MPI
320	0.84	0.0001	0.07	464.8	324.9	494.4 (-27.4%)	681.8
1280	0.76	0.0001	0.05	464.9	324.7	497.6 (-29.3%)	704.1
3200	0.35	0.0001	0.04	480.7	322.6	515.2 (-25.5%)	691.9
5600	0.48	0.0003	0.04	479.2	310.6	515.9 (-28.1%)	717.8

decreases from 225 to 162 seconds (26%). Figures 9e, 9f and 9g, 9h show the results of running the application with an initial mesh of size 110 million elements decomposed into 27 thousand sub-domains on 3200 and 5600 cores. The performance gain is even larger in these cases since the increased mesh size caused a higher load imbalance. The improvement exhibited is 56 and 43.6 percent, respectively. In table 2 one can see the high variation of refinement time for MPI, which is mitigated by PREMA's implicit load balancing. More specifically, while for MPI the refinement time varies by up to 754 and 488 seconds, PREMA manages to take the variance down to 239 and 215 seconds, respectively. For all these cases, the time attributable to the runtime system is negligible. Table 2 shows the overhead of PREMA being less than one percent of the overall runtime and the decrease in variation between minimum and maximum refinement time.

An important observation from Fig. 9 is that even though the load distribution has been improved it is still not optimal; some cores ran for a much longer time than the average. The same observation is apparent from Table 2; the variance between the minimum and maximum refinement time is much larger than the one noticed in the synthetic benchmark, caused by the mesh decomposition quality. Even though the adaptive octree creates sub-domains of similar size, the refinement time per sub-domain can differ dramatically.

For example, the refinement of a single sub-domain could last as much as the refinement of a hundred others, dominating the overall time of the refinement. Since concurrency is exploited by running refinement on different sub-domains using one thread per sub-domain, adding more threads will not lower the refinement time of a single sub-domain. This is also why PREMA does not scale as expected when increasing the number of cores. To address this issue, we plan to modify the application to incorporate data decomposition on top of domain decomposition, which will allow parallelism inside a sub-domain. We will also look into more effective decomposition methods for the initial work distribution.

Table 2 Time breakdown and comparison with the plain MPI version for different core allocations using the CDT3D mesh refinement application. As in table 1, the overhead presented is per thread utilized. The maximum and minimum refinement time correspond to the time spent by the most and least loaded thread in the lifetime of the application. Maximum refinement time dominates the overall running time.

#cores	PREMA overhead (sec)			Min refinement (sec)		Max refinement (sec)	
	Max	Min	Avg	PREMA	MPI	PREMA	MPI
640	0.7	0.004	0.18	80.4	13.9	197.12 (-42.1%)	340.7
1280	0.82	0.001	0.05	31.12	2.1	162.2 (-27.9%)	225.1
3200	37.9	0.04	0.65	86.1	9.8	325.2 (-57.4%)	764.4
5600	29.5	0.02	0.48	51.2	1.6	266.8 (-45.5%)	489.9

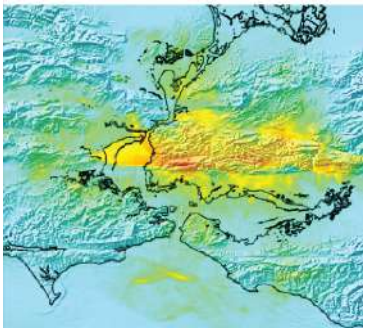


Fig. 10 Simulation produced by the SW4 seismic wave simulation application. Image adapted from the Computationl Infrastructure for Geodynamics [18].

5.3.3 Seismic Wave Simulations

SW4lite [19] is a proxy application that models the workflow of SW4 [20], an application that implements substantial capabilities for 3D seismic modeling (see Figure 10 for an example). Sw4lite is a simplified version of SW4 intended for testing performance optimizations in a few important numerical kernels of SW4 and was developed to support the Exascale Proxy Applications Project [21].

Even though this application is not impacted by load imbalance, it is communication-intensive and tightly coupled, a good candidate to showcase PREMA's low overhead and applicability even on applications that are not the main target of PREMA. We make the case here that even an application that consists of different kernels, where only a portion of them benefits from dynamic load balancing, can be ported to PREMA without negatively affecting the other kernels.

The proxy application starts by decomposing the original 2D grid into several partitions equal to the number of available processes. The processors are positioned into a logical 2D grid and are assigned a partition of the application grid. A preprocessing step follows before the main kernel of the proxy starts. The main computation kernel runs in an iterative fashion consisting

of computations intercepted by two cycles of neighbor-to-neighbor communication per iteration. The communication pattern for each cycle is as follows: Each processor sends some data of the partition it holds to its left neighbor and waits to receive the respective data from its right neighbor. Once the data are received, the same pairs of processors share data in the opposite direction. Next, the same process repeats for the y-axis; each processor sends another portion of its data to its bottom neighbor and waits to receive the respective data from its top neighbor. Then the communication continues in the opposite direction. Processors located on the edges of the grid only send/receive data to/from the available neighbors. Figure 11a shows the communication pattern schematically.

The following process was followed to port the application on top of PREMA :

- Each of the partitions of the decomposed 2D grid is registered as a mobile object.
- Each MPI rank holds partitions equal to the number of cores it utilizes.
- Preprocessing computations are performed by invoking remote handlers on each partition.
- Two-sided communication is replaced with one-sided asynchronous remote method invocations.

Specifically, the four-step communication pattern, part of the application's iterative process, has been modified to ensure correctness. In contrast with two-sided MPI, where the receiver can explicitly request for the data to be received, PREMA's message receiving is implicit; thus, we need to make sure that the receiver is ready to accept the data without corrupting its state. The communication in each direction begins with the receiving neighbor requesting the data. The sender will then send the respective data when it is ready. In this way, it is guaranteed that both neighbors' data are consistent after the first and third communication steps. The requests for the second and fourth steps are implicit as they are received as part of the actual data sent in the first and third steps. Figure 11b demonstrates the modified communication pattern.

For this paper, the problem LOH.1 presented in [22] is supplied as the input to SW4lite. PREMA runs using one MPI rank per available socket, consisting of ten hardware cores, while each computing node holds four sockets for a total of 40 cores per node. In this case, we use the configuration where PREMA does not have a dedicated thread for communication, but the handler execution threads run the communication functions periodically. The plain MPI version runs with one MPI rank to one core mapping. Figure 11c shows the performance comparison for the two approaches. We can see from the graphs that the performance of PREMA is equal and, in some cases, even better than the performance of the MPI implementation, even though the application would not benefit from load balancing and an extra step of message passing per iteration has been added. This is achieved by overlapping communications with computations (since both are asynchronous) and sharing threads available

in a socket to handle different requests. It is important to note that all load monitoring-related operations are run for PREMA, even though there is no need for load balancing, and there is no significant overhead.

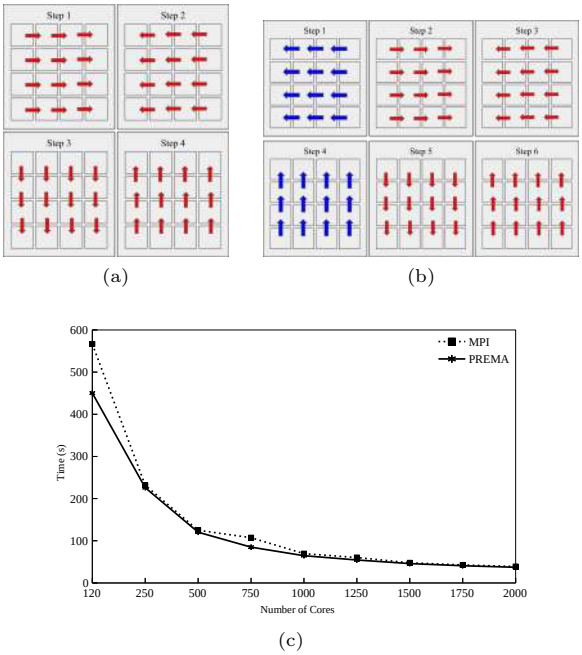


Fig. 11 The communication pattern of the (a) plain MPI SW4lite implementation, (b) modified implementation on top of PREMA. The blue arrows in (b) depict the data requests that have been added for PREMA implementation of SW4lite.(c) The performance comparison between the two implementations for different numbers of cores.

6 Related Work

Systems like AC [23], Split-C [24] and UPC [5] are some of the first systems to provide a partitioned global address space (PGAS) environment for parallel computing as an extension to the C language. They follow the same single-program multiple data (SPMD) model as MPI, where all computing nodes run the same code with different sets of data. Global access is provided through arrays spread among all computing nodes, with each node having affinity to a specific subset of the array. However, data migration is not supported natively, and the user needs to maintain the address space consistency if this is desired. Another issue with the aforementioned systems is that accessing remote and local data is done in a uniform way, making it difficult for the user to understand which data access will cause inter-node communication; thus, they rely heavily on compiler optimizations for performance.

Other systems (e.g. Co-array Fortran [25], Global Arrays [26], Titanium [27]) followed a similar way of expressing global access but made inter-node communication explicit by using a different interface between local and remote accesses to avoid such issues. However, the user will still need to maintain data consistency if data migrations are desired. CHAOS++[28] avoids this kind of limitation by natively allowing data migrations and implicitly maintaining consistency for them along with their pointers using mobile objects; however, it does not provide automatic load balancing.

Chapel [29] and X10 [30] are PGAS languages that use a more asynchronous approach based on the abstraction of locales or places, respectively. Places can be abstract or real machine locations where data or computations reside. Work and data are then assigned to them explicitly by the programmer using the language's interface. Work is assigned by creating asynchronous tasks that can run in any location and then synchronize to notify completion. High-level mechanisms to map data to locales also exist; however, this mapping is static and cannot be changed dynamically. As a result of the explicit assignment of data and the inability to change data mappings dynamically, dynamic load balancing has to be explicitly implemented by the application developer. Legacy systems like PREMA were designed to accommodate the needs of their time; to remain relevant, such systems need to evolve to serve the requirements of modern times.

HPX [31] is a distributed asynchronous many-task runtime system library that exposes an adaptive GAS model where objects can migrate between computing nodes, allowing for distributed load balancing. References and tasks to such possibly migrated objects are maintained and forwarded automatically. Even though there are efforts to integrate implicit load balancing with the system [32], to the best of our knowledge, HPX does not currently support such a feature nor an API to develop custom policies. Charm++ [33] is a system with similar characteristics to PREMA. Programs consist of medium-grained cooperating message-driven objects called *chares*. When a method is about to be invoked to an object, a message is sent to it implicitly. The execution of the code within a chare is then triggered asynchronously. Each chare is mapped to a physical processor by the runtime system transparently, allowing for dynamic change of the assignment of chares to processors during the execution. Thus, dynamic load balancing is provided implicitly to the application.

The Open Community Runtime (OCR) [34] is a fine-grained, asynchronous, task-based event-driven runtime. An OCR application consists of data blocks that encapsulate application data and can be referenced globally, event-driven tasks (EDTs) that perform the computations, and events that define the relationships between EDTs. A directed acyclic graph is composed of those components that resemble the application logic and is passed to the runtime, which manages data placement and schedules EDTs based on the dependencies defined in the graph. This approach makes porting a legacy MPI application on top of OCR quite complicated and error-prone. In contrast, PREMA supports a simple API similar to MPI, making this task simpler.

Legion [35] is a high-level parallel programming system for distributed heterogeneous architectures. Logical regions describe data organization and expose relationships used for locality and concurrency exploitation. Each task is explicitly associated with the regions it will access and the type of access needed (e.g., read-only, read-write). Legion can then map data to nodes in the distributed system and invoke the tasks accordingly. Because of the implicit distribution of data and work units, Legion requires applications to use regions as data structures, prohibiting them from allocating memory using C/C++ conventions for data that persist after a task completes. This approach induces an extended rewrite of legacy code, which may not always be feasible, particularly when external libraries are used.

Other systems provide implementations of activate messages for different use cases. Some target high performance at the expense of usability (e.g., DCMF [36], LAPI [37], GasNet [38]) and are designed mainly as low-level substrates for higher-level libraries. Others target ease of use but might suffer in performance (e.g. CORBA [39], Java RMI [40]) or try to get the best of the two (e.g. AM++ [41], ARMI [42]). Such systems implement some of the functionalities provided by DMCS and MOL but do not offer support for a global namespace of implicitly migrating objects in response to load balancing.

7 Conclusion and Future Work

We have presented the new design and implementation of our runtime system PREMA. We have shown the advantages of enhancing it with multiple threads dedicated to specific operations like message passing and computation and stated the lessons learned from this process. The new implementation is able to leverage both shared and distributed memory parallelism implicitly through a uniform interface by exploiting task access privileges. We presented the abstract interface that one can utilize to develop custom 2-level scheduling and load balancing policies. The multi-threaded model does not heavily impact the latency and bandwidth of the communication library while enabling more efficient communication and computation overlapping. Furthermore, we demonstrated the performance improvements of using PREMA with a parallel 3D advancing front mesh refinement application. We have noted an improvement of up to 56% compared to an MPI implementation without load balancing in varying sizes of core allocations ranging from 640 to 5600 cores with a negligible overhead of less than one percent. We have also demonstrated that the performance of tightly coupled and communication-intensive applications that do not need dynamic load balancing is not affected significantly by the additional load monitoring-related operations of PREMA.

In the future, we intend to replace our threading mechanism with that provided by Argobots [43] to take advantage of their lightweight user-level threads and context-switching, which will be enhanced with task dependencies. Furthermore, support for efficient utilization of deep memory hierarchies that consist of many layers of memory (e.g., High Bandwidth Memory, Non-Volatile

RAM, burst buffers) will be implemented. Leveraging from this extension, we will implement out of core support based on our work in the past [44] which was utilizing the previous version of PREMA. Another aspect that will leverage from this extension is the support of process fault-tolerance, which can be implemented using the mobile objects as the data that is checkpointed and recovered instead of whole process states. Finally, PREMA will be extended to support distributed heterogeneous systems that incorporate a mix of CPUs and GPUs. These additions will further reinforce the capabilities and features that enable efficient utilization of exascale-era platforms.

Acknowledgments

This work is funded in part by the Dominion Fellowship, the Richard T. Cheng Endowment at Old Dominion University and NSF grant no: CNS-1828593.

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, K. Vogiatzis, A. Kot, and N. Chrisochoides, "Parallel software framework for large-scale parallel mesh generation and adaptation for cfd solvers," in *AIAA Aviation Forum 2018*, (Atlanta, Georgia), June 2018.

[3] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," *SIGARCH Comput. Archit. News*, vol. 20, pp. 256–266, Apr. 1992.

[4] 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, Association for Computing Machinery, 1993.

[5] W. W Carlson, J. M Draper, D. Culler, K. Yelick, E. Brooks, K. Warren, and L. Livermore, "Introduction to upc and language specification," 04 1999.

[6] J. Slotnick, A. Khodadoust, J. Alonso, D. Darmofal, W. Gropp, E. Lurie, and D. Mavriplis, "CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences," Tech. Rep. CR-2014-218178, Langley Research Center, Mar. 2014.

[7] 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.

[8] K. Barker, N. Chrisochoides, D. Nave, J. Dobellaere, and K. Pingali, "Data movement and control substrate for parallel adaptive applications," *Concurrency and Computation:Practice and Experience*, pp. 77–105, February 2002.

[9] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel, "Mobile object layer: A runtime substrate for parallel adaptive and irregular computations," *Adv. Eng. Softw.*, vol. 31, pp. 621–637, Aug. 2000.

[10] A. Fedorov and N. Chrisochoides, "Location management in object-based distributed computing," in *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, pp. 299–308, Sept 2004.

[11] D. Nave, N. Chrisochoides, and L. Chew, "Guaranteed-quality parallel delaunay refinement for restricted polyhedral domains," *Computational Geometry*, vol. 28, no. 2, pp. 191–215, 2004. Special Issue on the 18th Annual Symposium on Computational Geometry - SoCG2002.

[12] 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.

[13] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, pp. 720–748, Sept. 1999.

[14] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Commun. ACM*, vol. 19, pp. 395–404, July 1976.

[15] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Understanding and effectively preventing the aba problem in descriptor-based lock-free designs," in *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pp. 185–192, May 2010.

[16] A. Chernikov and N. Chrisochoides, "Parallel guaranteed quality Delaunay uniform mesh refinement," *SIAM Journal on Scientific Computing*, vol. 28, no. 5, pp. 1907–1926, 2006.

[17] 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.

[18] "Computational Infrastructure for Geodynamics :: Software." https://geodynamics.org/cig/software/sw4/. [Accessed November 21, 2021].

[19] "Sw4lite." https://github.com/geodynamics/sw4lite, 2019. [Accessed January 23, 2021].

[20] N. Petersson and B. Sjögreen, "Sw4 v1.1 [software]," 2014.

[21] "Exascale project," 2019. [Accessed January 23, 2020].

[22] S. D. et al., "Tests of 3d elastodynamic codes: Final report for lifelines project 1a01," tech. rep., Pacific Eartquake Engineering Center, 2001.

[23] W. W. Carlson and J. M. Draper, "Distributed data access in ac," *SIGPLAN Not.*, vol. 30, pp. 39–47, Aug. 1995.

[24] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. S. Lumetta, T. von Eicken, and K. A. Yelick, "Parallel programming in split-c," *Supercomputing '93. Proceedings*, pp. 262–273, 1993.

[25] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, pp. 1–31, Aug. 1998.

[26] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, applications and performance of the global arrays shared memory programming toolkit," *International Journal of High Performance Computing Applications*, vol. 20, pp. 203–231, 06 2006.

[27] K. A. 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.

[28] C. Chang, J. Saltz, and A. Sussman, "Chaos++: A runtime library for supporting distributed dynamic data structures," in *Parallel Programming Using C++*, 1995.

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

[30] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, pp. 519–538, Oct. 2005.

[31] 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*, PGAS '14, (New York, NY, USA), pp. 6:1–6:11, ACM, 2014.

[32] P. Amini, *Adaptive Data Migration in Load-Imbalanced HPC Applications*. PhD thesis, Louisiana State University and Agricultural and Mechanical College, 2020.

[33] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," *SIGPLAN Not.*, vol. 28, pp. 91–108, Oct. 1993.

[34] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo, "The open community runtime: A runtime system for extreme scale computing," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, Sept 2016.

[35] 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.

[36] S. Kumar, G. Dózsa, G. Almási, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. E. Smith, and C. J. Archer, "The deep computing messaging framework: generalized scalable message passing on the Blue Gene/P supercomputer," in *ICS '08*, 2008.

[37] G. Shah, J. Nieplocha, H. Mirza, C. Kim, R. Harrison, R. Govindaraju, K. Gildea, P. DiNicola, and C. Bender, "Performance and experience with LAPI - a new high-performance communication library for the ibm rs/6000 sp," in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pp. 260 – 266, 01 1998.

[38] D. Bonachea and P. H. Hargrove, "Gasnet-ex: A high-performance, portable communication library for exascale," in *Languages and Compilers for Parallel Computing* (M. Hall and H. Sundar, eds.), (Cham), pp. 138–158, Springer International Publishing, 2019.

[39] A. L. Pope, *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[40] J. Waldo, "Remote procedure calls and java remote method invocation," *IEEE Concurrency*, vol. 6, no. 3, pp. 5–7, 1998.

[41] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "AM++: A generalized active message framework," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), p. 401–410, Association for Computing Machinery, 2010.

[42] N. Thomas, S. Saunders, T. Smith, G. Tanase, and L. Rauchwerger, "ARMI: A high level communication library for STAPL," *Parallel Processing Letters*, vol. 16, pp. 261–280, June 2006.

[43] 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.

[44] A. Kot, A. Chernikov, and N. Chrisochoides, "The evaluation of an effective out-of-core run-time system in the context of parallel mesh generation," in *IEEE International Parallel and Distributed Processing Symposium*, pp. 164–175, May 2011.

Appendix I

The goal of this appendix is to present the (simplified) implementation of two load balancing strategies, Master Worker and Diffusion, utilizing PREMA's scheduler API. The two strategies have been used to scale different irregular applications ([7], Section 5.3.2), while significantly reducing code complexity (removing load balancing-related code) and line count (e.g., 1200 vs 2500 LOC in the first application) compared to the respective MPI implementations.

I.1 Master Worker

Figure 12 presents the simplified master-worker implementation. The derived class assigns a single node as the master and defines a load threshold under which a worker node is considered underloaded (line 3). Each node keeps a custom map (provided by PREMA) that holds mobile objects along with their workload and tracks the overall node workload. When a worker finds its load under the threshold, it sends a remote request to the master for a new mobile object migration (lines 5-7). If the master holds enough load, it picks a mobile object, packs it, and sends it to the requesting worker (lines 27-30). Otherwise, it requests the worker to wait and pushes its rank to a list of waiting workers (lines 33-34). In the reception of the master's reply, the worker unpacks and installs the packed object to the local node, which updates PREMA about the migration(lines 38-41). If there is no mobile object to unpack, the worker sets a flag that it should wait from the master for a new workload when it becomes available (line 44). In this simplified case, we use a simple list to maintain handler invocation requests and support the *push()/pop()* operations; a more sophisticated implementation could use work pools per thread, per mobile object, or a combination of the two. Method *notify()* keeps the mobile objects - load map and node workload up to date and is called each time the node workload changes.

```
class simple_master_worker : public ilb::scheduler {
public:
   simple_master_worker(size_t low_limit, int master_rank) : m_low_limit (low_limit),
        m_master(master_rank), m_worker_waiting(0) {}
   void dist_balance() {
      if(prema::my_rank() != m_master && ! m_worker_waiting)
         if( m_mo_map.get_total_load() < m_low_limit) // Worker checks if load dropped below
              threshold
            dmcs::send(m_master, work_request); // request work from master
      else
         while(m_mo_map.get_total_load() > 0 && !m_waiting.empty()) { // master sends work to
              waiting nodes while there enough workload
            int dst = m_waiting.front();
            m_waiting.pop_front();
            work_request(dst);
         }
   }
   // update node's and mo's load
   void notify(ilb::mobile_object mo) {m_mo_map.insert(mo->get_load(), load);}

   ilb::handler* pop(int thread_id) {
      ilb::handler* hdlr = m_hldr_pool.front(); // pick the next handler in the pool
      return hdlr;
   }
   void push(int thread_id, ilb::handler* hdlr, ilb::mobile_object mo) {
      m_hldr_pool.push_back(hdlr); // insert a new handler in the work pool
   }
   void work_request(int src) {
      if(m_mo_map.get_total_load() > 0) { // if enough load
         ilb::mobile_object mo = m_mo_map.pop(); // master picks mo with largest workload
         void* buffer = mo->pack(); // packs it
         dmcs::send(src, work_request_reply, buffer); //and migrates it to requesting worker
      }
      else {               // not enough load
         m_waiting.push_back(src); // keep track of waiting workers
         dmcs::send(src, work_request_reply, NULL); // put worker to wait
      }
   }
   void work_request_reply(int src, void* buffer) {
      if(buffer != NULL) { // master replied with mo (workload)
         m_worker_waiting = 0;
         ilb::mobile_object mo(buffer); // unpack mo
         mo.install();          // and notify PREMA
      }
      else // wait until master has work to share
         m_worker_waiting = 1;
   }
private:
   int m_master, /* rank of the master node */, m_worker_waiting;
   size_t m_low_limit; // low threshold to request for load
   ilb::mo_work_map m_mo_map; // map wrt load for mobile_objects, also keeps total workload
   list<handler*> m_hldr_pool; // list of pending handlers
   list<int> m_waiting; // list of workers waiting for work from master
}
```

Fig. 12 Sample implementation of the Master-Worker model using PREMA's API.

I.2 Diffusion

Figure 13 presents the simplified diffusive scheme implementation. In this scheme, each node assigns a "neighborhood" of other nodes from which it can request workload. In each new load balancing phase, the underloaded node tries to steal from the node with the largest workload in the neighborhood. If no neighbor has enough workload, a new neighborhood is assigned for the next load balancing phase. In this implementation, *dist_ balance()* checks if the node is underloaded and initiates a new load balancing phase by requesting the workload levels of its neighborhood. Once the underloaded node receives all the responses, it chooses the neighbor with the highest load and requests for mobile object migration or assigns a new neighborhood if not enough workload exists(lines 25-36). The receiver of a migration request picks its mobile object with the largest workload and, if its workload is enough, packs and sends it to the underloaded node. Otherwise, it refuses to migrate any work (lines 39-44). Depending on this response, the requester will either unpack and install the received mobile object or replace the neighbor in the neighborhood set and prepare for a new load balancing phase.

```
class simple_diffusion : public ilb::scheduler {
public:
   simple_diffusion(size_t neighbors_cnt, size_t low_limit, bool replace, size_t min) :
         m_neighs_cnt(neighbors_cnt), m_low_limit (low_limit), m_replace_neighs(replace),
         m_pending(false){
      m_neighbors = assign_new_neighbors();// Pick a set of m_neighs_cnt neighbors
   }
   void dist_balance() {
      if(!m_pending && (m_mo_map.get_load() < m_low_limit)){ // low workload and no lb pending
         m_pending = true; m_levels_recv_cnt = 0; //new lb phase
         // Request neighbors' workload levels
         for(int n: m_neighbors) { dmcs::send(n, load_level_request); }
      }
   }
   // update node's and mo's load
   void notify(ilb::mobile_object mo) {m_mo_map.insert(mo->get_load(), load);}
   ilb::handler* pop(int thread_id) {
      return m_hldr_pool.front(); // pick the next handler in the pool
   }
   void push(int thread_id, ilb::handler* hdlr, ilb::mobile_object mo) {
      m_hldr_pool.push_back(hdlr); // insert a new handler in the work pool
   }
   void load_level_request(int src) {
      dmcs::send(src, load_level_request_reply, m_mo_map.get_load()); // Send my workload
   }
   void load_level_request_reply(int src, size_t load){
      m_levels_recv_cnt++; // Count how many neighbors replied
      m_load_to_neigh[src] = load; // Track their loads
      if(m_levels_recv_cnt == m_neighs_cnt){ // If received load levels from all
         int max_neigh = max_load_neigh(m_load_to_neigh); // neighbor with largest workload
         size_t max_load = m_load_to_neigh[max_neigh]; // Get max workload value
         if(max_load == 0){ // Found no neighbor with workload
            if(m_replace_neighs){ m_neighbors = assign_new_neighbors();} // set new neighbors
            m_pending = false; // End this lb phase
         }
         else{ dmcs::send(max_neigh, work_request);} // Found workload, request
         m_load_to_neigh.clear();
      }
   }
   void work_request(int src) {
      if(m_mo_map.get_total_load() > m_low_limit){ // If I have enough load
         ilb::mobile_object* mo = m_mo_map.top();
         if(mo->get_load() > m_min_to_send){ // and my mo's workload is large enough
            m_mo_map.pop();
            void* buffer = mo->pack(); // pack mo
            dmcs::send(src, work_request_reply, buffer); // and send it to source
         } else{ dmcs::send(src, work_request_reply, NULL);} // Otherwise, send NULL
      }
   }
   void work_request_reply(int src, void* buffer) {
      if(buffer != NULL) { // received a mo (workload)
         ilb::mobile_object mo(buffer); // unpack mo
         mo.install();          // and notify PREMA
      }else{ assign_one_new_neighbor(src); } // No mo (worload) received, replace neighbor
      m_pending = false; // Done with this lb phase
   }
private:
   size_t m_low_limit, m_level_recv_cnt;
   ilb::mo_work_map m_mo_map; // map wrt load for mobile_objects, also keeps total workload
   list<handler*> m_hldr_pool; // list of pending handlers
   list<int> m_neighobrs; // list of neighbors
   bool m_pending; // Indicates if in the process of load balancing
   bool m_replace; // are we replacing neighbors if they do not have workload?
}
```

Fig. 13 Sample implementation of a Diffusive model using PREMA's API.