Runtime Framework for Parallel and Adaptive Applications

Polykarpos Thomadakis, Christos Tsolakis, and Nikos Chrisochoides

Abstract—This paper presents the design and implementation of the new version of our software, the Parallel Runtime Environment for Multicomputer Applications. This framework provides large-scale applications with implicit load balancing, scheduling and latency hiding through a simple but powerful interface. The new design, the framework has been augmented with multithreading, separating communication and execution into different threads to provide asynchronous message reception and instant computation execution at the arrival of new work requests. Furthermore, it allows the application to run multiple computations in parallel, while monitoring the load of the system and performing migrations when desired. Scheduling and load balancing are also enhanced by introducing custom intra-node schedulers and the ability to perform multiple migrations that initiate from the same process further reducing the time spent in this procedure. The motivation for the development of the runtime system is to provide dynamic runtime support for parallel mesh refinement applications. Testing the system on such an adaptive application indicates an overall performance improvement of up to 40 percent with an overhead of less than one percent, in cases of up to 270 computing nodes, compared to no load balancing, by retaining a better work-load distribution among the execution units.

Index Terms—Runtime systems, dynamic load balancing, multithreading, hybrid model, message passing, parallel, distributed, scientific computing, parallel mesh generation

1 INTRODUCTION

Computational scientists have been working with large computational and data-intensive problems for decades. The only effective way to deal with such problems in a reasonable period of time is to utilize large computing clusters efficiently. However, achieving high performance on such machines can be challenging. The programmer would have to explicitly deal with overlapping data communication overhead with computations, load balancing, scheduling, fault-tolerance and power-aware use of both the computing nodes and the interconnection network. In this paper we focus on the new design of our runtime system, Parallel Runtime Environment for Multi-computer Applications (PREMA) [1] [2], that handles the former three in a more efficient way.

PREMA enhances the application with one-sided communication, remote method invocation, a globally addressable name-space, data migration and message forwarding, and implicit load balancing targeted to support adaptive and asynchronous applications. In order to perform its best, PREMA requires that the original data domain be broken into N sub-domains (with N >> #processors). This process is known as over-decomposition and increases the work available for the load balancing algorithm. Once the sub-domains have been created they are captured into an abstract structure the mobile object.

PREMA adopts a message-driven execution model where computations can be invoked asynchronously on each mobile object regardless of whether it is located on the local or a remote node. The runtime is responsible to locate where the respective data resides and invoke the computation requested. These requests called remote handlers constitute the work-load of a mobile object and the pending load of a computing node is thus represented by the sum of the load of all of its local mobile objects. When a computing node runs low in work-load it will initiate load balancing which is performed by migrating mobile objects from other nodes. Workload is thus migrated as a by-product of such migrations.

In the initial design and implementation [1], the runtime system would consist of a single hardware thread that would have to run all the system functionalities e.g. application’s computations, communication related operations, remote method invocation requests, load balancing, etc. In order for all those operation to achieve progress the application would have to explicitly switch control to the runtime system at regular intervals, using the respective interface.

This model would limit applications to only run a single handler at a time even for independent handlers. Moreover, long-running tasks would prevent the application from switching control to the runtime system in time, and consequently, message passing and other important for the system operations would be delayed instead of being overlapped. Furthermore, the runtime system was agnostic regarding the placement of its computing processors. As a result, processors that resided on the same hardware node had to go through the process of message passing and packing/unpacking, in order to move the work-load from one to the other.

Another issue is that the system operations are not thread-safe, i.e. an application may not spawn more threads and switch their control to the runtime system in order to execute more handlers at a time, which would help to minimize the problems mentioned above. To address those
issues, we decided to re-design and re-implement PREMA to adopt a native multi-threaded model.

In this new implementation, our runtime system has been enhanced with a thread that is dedicated to handle message passing operations, and a pool of threads that handles any local or remote method invocation request that targets a node or a mobile object. By having threads that asynchronously perform the requested operations, there is no need for the application to call any function to achieve progress, it is handled by the system internally. This multi-threaded scheme introduces the flexibility for concurrent invocations of remote methods and helps to efficiently overlap communications with computations.

The multi-threaded model has been incorporated in the lowest layer of PREMA - the Data Movement and Control Substrate- which is the infrastructure on top of which the rest of the layers are built. As a result, all of them have now access to multiple threads for their operations, increasing their flexibility but also requiring them to be modified accordingly in order to adapt to the new model. Synchronization functionalities have been introduced to allow applications to express requirements for data accesses (e.g. object specific locks) which are used from the runtime system to perform the scheduling. Finally, customizable schedulers are made available for the needs of the intra-node scheduling component while the inter-node scheduler of the original implementation is modified to conform with the new model.

Section 2 briefly describes the functionalities of each software layer of the system, the modifications they have undergone and how they are benefited from these changes. In Section 3 we evaluate the performance of PREMA first in terms of overhead on top of the communication library and then in terms of its performance in maintaining a fair workload distribution in an increasing number of cores. Section 4 presents some related work, while Section 5 summarizes our conclusions and presents some thought for the future of this project.

2 Design and Implementation
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 those three layers. The functionality that each layer provides is presented first, and then how they have been modified to better conform to the needs of our multi-threaded model and make the best use of it.

2.1 Data Movement and Control Substrate
The Data Movement and Control Substrate (DMCS) [3] acts as a thin layer that abstracts the underlying hardware, and the communication library (e.g. MPI) from the application or the higher level libraries that consist PREMA. By using this abstraction the runtime system can be easily ported to a different computing environment, by only making changes to 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 [4] paradigm. Methods that are supposed to be available for remote invocation are referred as remote handlers in the context of our runtime system, and they need to be registered before they can be used as such. This paradigm of asynchronous execution and communication allows the runtime system to hide remote data access latency behind computations which is utilized from both the system and the application.

To improve the performance of DMCS and face the limitations of the previous version we have extended its framework with multiple threads. This extension can increase the performance for both the communications and the computations that take place by making them more asynchronous and increasing the chances to overlap them. Specifically, DMCS is decomposed into three components, the handler-execution, communication and application components (fig.1). Each of them handles a different functionality that is crucial for the entirety of the system as described below.

The communication component is responsible for handling system and application operations destined for remote nodes. It consists of a dedicated thread that runs in a loop where it receives incoming messages, checks the local node for pending message requests destined to remote nodes, and finally, monitors the progress of message receptions/delivers that have already initiated and signals the other components about their completion when needed.

The communication component maintains a list of messages to be sent remotely for each thread. When a message needs to be sent by the application or the system it will be pushed to the respective list and the communication component will handle it appropriately. All MPI operations used by this component are asynchronous (MPI_Isend, MPI_Issend, MPI_Irecv, MPI_Ibarrier) to avoid deadlocks that could occur from their blocking counterparts. Therefore, all messaging requests need to be checked for completion. Those that have not completed instantaneously are moved to another structure with incomplete messaging requests and their progress is checked periodically. Once such a request is found to have been serviced, it is removed from the structure and those waiting on its completion are signaled.

Two types of messages are used (fixed or split-phase) based on whether their size will exceed a predefined threshold. When the size of the data to be sent is lower than the threshold, a fixed size message will be used regardless of the data size. This message will contain a header needed from runtime system, followed by the actual data that will be passed to the remote handler. For split-phase messages two messages need to be sent. The first one -at the size of the fixed message- contains only the header while the second holds the data of the remote handler. Thus, the receiving side can always expect to receive a message of specific size. When the size of the data is greater than the threshold, the header in the first message will inform the receiver about the existence of a second message and the actual size of its 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 allows us to preallocate a pool of such messages for each thread that can send or execute remote handlers. A new incoming message will be
received into a preallocated message of one of the threads’ pools, and will be pushed to its list of pending remote handlers in order to be scheduled from 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 when the pool runs out of preallocated messages, something that does not occur often.

The **handler-execution component** executes the remote method invocation requests. The user can decide the number of threads that it consists of 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 two which are reserved for the application and the communication components and they are not explicitly bound. This is the component where the bulk of computations are running and where the parallelism is exploited for both the application and the system related operations as described later in the paper. 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. Each thread is associated with such a pool and is responsible to service it. If the associated work pool is empty, a thread will employ work stealing with random victim selection [5] to avoid remaining idle and to balance the load in the node. When there is no work in the thread’s work pool and a steal attempt fails, the thread will backoff [6] for an exponentially increasing period of time (that is constrained to a maximum value) until it succeeds in finding some work, in which case the backoff period is set to zero. By enforcing this delay, the memory contention and the amount of power wasted is reduced. The scheduling part of this component can be modified through a provided interface which is mainly utilized by higher-level libraries of the system 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 here to produce work for the other components. When work starts at the other components, the application thread can be either put to sleep or its control can be passed to the runtime system in order to contribute to the handler-execution threads work until a condition (e.g. the current phase of the algorithm has finished) is met. However, this is not required to achieve progress if there is at least one thread at the handler-execution component.

### 2.2 Mobile Object Layer

The **Mobile Object Layer** (MOL) [7] builds on top of the 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 is achieved by extending the remote method invocation functionality of DMCS to target mobile objects directly, wherever they might reside.

A distributed directory maintains the last known locations of all the mobile objects in the local and the remote nodes. When a message is to be sent to a mobile object, its last known location is retrieved from the directory and the message is sent there. That node might not hold the aforementioned mobile object anymore in which case the message will be forwarded to its own last known location of the mobile object. Once the message arrives to its destination and the respective remote method is invoked, an update message is sent back to the original sender informing about the current location of the object. There are more location-management policies available, however, the *lazy updates*, described here, is the one that showed the best performance [8].

Running mainly inside DMCS remote handlers, the MOL leverages from 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 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, the directory must be thread safe.

To avoid the contention issues that would be created by using a mutex and a simple C++ STL map, a custom hash table with chaining (i.e. in case of collisions a list is used to keep the colliding elements in the same entry of the table) is used instead. This approach allows elements to be inserted in different entries of the table safely, but still leaves possible race conditions for colliding elements. Using a mutex per
table entry could solve this problem, however, it still is too much to require a mutex lock for each access to an entry since the majority of accesses are expected to be lookups and not insertions or deletions. Instead, for insertions the atomic operation compare and swap is used (CAS) while for deletions the element is marked as invalid and is reused instead of being removed from the list. The combination of the two results in a thread safe list that does not suffer from the ABA [9] problem. Furthermore, to improve cache locality, memory is allocated in chunks that are chopped and used as nodes of the list.

The MOL handlers are enhanced with an access type flag, exclusive or shared. Handlers flagged with shared access can run concurrently on the same object while a handler flagged as exclusive can be the only handler executing on a mobile object. The system will suspend the handlers that are not allowed to run and will only schedule them again once the mobile object is available for their type of access. 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 an mobile object has to be migrated; it first has to be locked exclusively, and then packed and sent to the new location, so that no handler tries to access it in an inconsistent state.

2.3 Implicit Load Balancing

PREMAs Implicit Load Balancing (ILB) [1] layer makes use of the functionalities of both DMCS and MOL to provide transparent and implicit data and load migration in the event of load imbalance detection. It provides all the necessary tools to develop efficient and transparent custom load balancing policies that can be switched without almost any modifications to the application, through an isolated scheduling module.

All mobile objects are associated with their pending computations which constitute the load of a mobile object, thus, migrating mobile objects will implicitly migrate work-load from one node to another resulting in load balancing. To allow the system to perform this operations automatically, each mobile object is registered with a set of user defined callback functions that allow the runtime system to pack, unpack, and retrieve its size, the difficulty to migrate it and its load based on its pending handlers. It is important to note here that all the book-keeping, load updates, balance requests and application computations are executing through a call to PREMA's polling function, so it is vital that it is called often enough to keep the system progressing and up to date.

When built on top of the new multi-threaded MOL and DMCS, the ILB can support concurrent execution of multiple handlers for mobile objects (ilb_message), load updates, balance requests, etc. Thus, migration of multiple objects can start at the same time and can easily overlap with each other and with application computations if there is enough workload. Furthermore, before initializing load balancing with a remote node the system is now able to try load balance looking for work in the same node first. This process is much faster since there is no need to send messages or perform data packing/unpacking.

In a previous attempt to extend PREMA with multiple threads ILB ran through MOL handlers that boil down to DMCS handlers. This meant that once a ILB handler was scheduled at the DMCS level, it would be executed right away. So in order to keep track of the pending handlers load we had to monitor it when the message was received. Thus, the communication thread was modified to update the number of handlers waiting for the mobile object that it just received a message for. Once we had this number it was easy to calculate the actual work-load of each rank. A system handler would spawn periodically to check the number of handlers, calculate the load per mobile object and update the scheduling module. If not enough load was found, the system would initiate the load balancing algorithm that was chosen by the user on startup.

In the current implementation, while MOL handlers are still running inside the DMCS handlers as before, the ILB ones now reside in a different work pool inside the scheduling module and are not running inside any other type of handler. Specifically, ILB handlers are routed to their destination through the MOL handlers which will calculate the load of the target mobile object, insert the handler to the pending handlers of the mobile object and then notify the scheduling module. As with MOL, ILB handlers are flagged for exclusive or shared access holding the same principles as discussed for the MOL. Another modification has been performed to the DMCS scheduler using the interface that it provides. It consists of two loops, the first will run all the handlers for the DMCS and the MOL (since they run inside DMCS handlers) and when there is no more work to be done there, it will switch to the second loop where it will execute the work that it pops from the scheduling module. This work can either be an ILB handler to execute or to initiate load balancing. As a result, the load of the runtime system will have been updated to the most up to date value before any computation handler or load balancing request starts running.

When the system initiates load balancing e.g. using a diffusive algorithm, it will request the load of some nodes that belong to the same neighborhood. It will pick one of them and send one or more DMCS handlers that when executed will try to find one or more mobile objects that have some load. If they succeed, they will try to lock these mobile objects in order to keep MOL/ILB handlers from executing on them. Once the objects are locked they can safely be packed and migrated together with their pending load to the new location. Any handler that had been pushed to the scheduling module will be invalidated at the point of packing and thus, any thread that pops one of them, after the mobile object has migrated, will abort its execution. For intra-node load balancing, the scheduling module has been extended to support customization so that a user could choose what type of load balancing to be used, the default is work-stealing.

One last contribution to the runtime system is the introduction of 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 to the same node. Its inputs are the mobile objects that are 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 it is called, it will collect the mobile objects on the same node, lock them so that they cannot be migrated by another handler and schedule the requested handler. This operation was mainly developed to better support applications that may require for different mobile objects to be fused back into a single one before they can start their computations like the one presented in [10]. An effort that uses this functionality can be found in [11] but is out of the scope of this paper.

3 PERFORMANCE EVALUATION

3.1 Experimental Setup

The following experiments have been conducted on a computing cluster consisting of 190 Intel(R) Xeon(R) E5-2660, E5-2660 v2, E5-2670 v2, E5-2698 v3, E5-2683 v4) computing nodes. Each node has two CPUs of 16-32 single-threaded (hyper-threading is off) cores in total, 128 GB of memory and runs Red Hat Linux (kernel 2.6.32-754.3.5.el6.x86_64). We used the MPICH 3.1.3 MPI implementation as the communication library and gcc 6.3.0 for compiling.

3.2 Communication

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

In Fig. 2a we can see that DMCS, MOL and ILB add a roughly fixed amount of overhead to the latency that is independent of the message size. Each layer is bound by the performance of the layers that it’s build upon, and all of them are bound by the performance of MPI as expected. Though, because the overhead is relatively fixed its effect is more noticeable in smaller messages where the MPI time is very low and the overhead is a significant percentage of the overall time as seen in Fig. 2b. The effects on the bandwidth seem to be less significant for both small (Fig. 2d) and large messages (Fig. 2c). ILB experiences the highest penalties since an ilb_message has to go through both DMCS and MOL and also be registered with the load balancing module before being scheduled. However, we believe that the overhead added is acceptable for the functionality that our system provides.

As mentioned in before, DMCS uses pools of preallocated, fixed-size messages which consist of the headers required by the remote handler requests. By keeping those pools we reduce the overhead accountable to memory allocations and also avoid querying for the size of incoming messages. As a result, the latency for initializing the message delivery process is reduced and turns to be relatively stable among different invocations. When the arguments of a handler are small enough to fit inside a fixed-size message, they are copied into it, otherwise, a separate message is sent for them. In the former case, the receiver has the preallocated messages ready to receive while for the latter the buffer can be allocated and the receiving call can be made as soon as the headers are received, even before the message with the actual data has been sent. This 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 3 shows the advantage of using preallocated, fixed-size messages. The small messages (smaller than 2KB) are highly benefited by 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). By using preallocated messages the latency (Fig. 3a) is almost half of the case where no preallocated messages are used (39 compared to 77µs). Furthermore, the bandwidth (Fig. 3b) 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 possibly because at this point the cost of sending the messages themselves becomes the dominant factor.

3.3 Load Balancing

3.3.1 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 as a way to test the system in a fully controlled and isolated environment. This is an attempt to avoid any behavior that a real application could demonstrate that could possibly affect the performance of the system.

The benchmark begins by creating mobile objects on process 0 and dispersing them to the available workers/cores, a computation is then invoked on each of the mobile objects via PREMA’s messaging mechanism. Once all computations on a mobile object have completed, a notification is sent back to process 0. When all completion notification have been received, the benchmark terminates. We assign 10 mobile objects to each available core and specify a weight (work load) from two categories, light and heavy, for each mobile object. The average execution time of a heavy weight mobile object is 2.5x the execution time of a lightweight one 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 can have 16 or 32 cores per node or two instances of 16 cores in the same node. We employ a diffusive load balancing algorithm to evaluate the performance of PREMA.

For the MPI version of the benchmark we replace the mobile objects with plain data objects and and do not perform any load balancing. Once a MPI rank has received all of its data it will execute the computations for all of them and then call MPI_Finalize. Note that even though PREMA uses one core exclusive for message handling, for fairness
Fig. 2. Ping-pong measurements for all three layers of PREMA compared to MPI. Comparison of latency (a), latency for small messages (b), bandwidth (c), and bandwidth for small messages (d).

Fig. 3. Ping-pong measurements for DMCS without preallocated messages. The effect of using preallocated messages in latency (a) and bandwidth (b). The performance for messages larger than 1KB is comparable to the optimized version and is not shown here.

this core is counted as an available core when we calculate 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 Figure 4. The work-load distribution achieved indicates the impact of using PREMA compared to using the plain MPI implementation. The first row compares the work-load distribution of 320 cores using plain MPI (320 ranks) and PREMA (10 ranks (nodes) of 32 threads each). In Fig. 4a the heaviest work-loads has been gathered roughly to the first 130 cores for an overall running time of 683 seconds. Fig. 4b shows the results after porting the benchmark on top of PREMA, where the work-load has been redistributed equally among the available cores decreasing the overall running to time to 495 seconds an improvement of 27.53 percent.

The second row shows the performance of the same benchmark when we quadruple the number of cores and the number work units. The pattern for the work-load distribution remains the same as and such we see the same performance breakdown as before. Fig. 4d shows that PREMA is not affected at all by the vast increase in number of cores 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 work-load of the problem. This is made more obvious if we look at the MPI case where the dashed line is further higher from the computations bars since there are many more ranks that need to be initialized. The third and fourth row 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 rank instead of 32 which increases the number of ranks as well as the number of cores that are reserved for communication. In other words, there are fewer cores for computations per hardware node than before (2 communication cores vs 1 2 communication core per node). Despite those modifications, PREMA still maintains a fair work-load distribution and also is not affected as much from the MPI initialization step. 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 the maximum refinement time. The effectiveness of the 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.

3.3.2 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. In order to evaluate its performance with such applications we used our in-house developed tetrahedral mesher CDT3D [12] 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, which 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 the purpose of this experiment the first two steps, ie. Delaunay Tetrahedralization and Boundary Recovery were...
executed sequentially as they are needed to bootstrap the mesh and they are less time consuming in comparison to the mesh refinement. The resulting mesh was then partitioned into \( N \) sub-domains with \( N >> \# \text{ cores} \) using an adaptive octree. The sub-domains are registered as mobile objects which are then serialized and distributed among the available processes by PREMA. This decomposition scheme was selected in an attempt to create sub-domains with similar number of tetrahedra as a way to create similar initial work-load 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 sub-domains. 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.

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 by this experiment are the following:

- Pack/Unpack Sub-domain callbacks, for serializing
and de-serializing the sub-domain in order for the ILB layer to be able to migrate the mobile objects when needed

- Initialize, for bootstrapping various data structures inside the mesher
- Refine, for the sub-domain refinement
- Callback for calculating the weight of a handler
- Minor handlers for printing the final mesh.

We ran the application with both ILB and plain MPI to evaluate the quality of load balancing. The preprocessing step is identical in both cases; once the sub-domains are created, they are assigned to the available worker cores. Fig. 5 depicts the performance comparison of ILB versus MPI using the mesh refinement application. The first and second rows show the results for 640 and 1280 cores allocated in the same manner as described in 3.3.1 using the same mesh size of 30 million elements, over-decomposed into 4.5 thousand sub-domains. The preprocessing and decomposition time are not included in the graph since these operations are performed by a single thread and are the same for all cases. Again a diffusive load balancing policy is used where each PREMA rank can only share its load with a specific subset of the ranks available. If no rank in the set has enough, it will randomly create a new set of other ranks. The performance improvement that ILB offers compared to the plain MPI implementation without 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 is reduced, and the difference between the maximum and minimum refinement time is reduced from 326 to 116 seconds while the overhead imposed from the load balancing is on average 0.48 seconds. For the 1280 cores case the difference between maximum and minimum refinement time decreases from 223 to 131 seconds. The third and fourth row show the results of running the application with an initial mesh of size 110 million elements decomposed into 27 thousand sub-domains. The performance gain is even larger in those cases with improvements of 56 and 43.6 percent respectively. Again, in table 2 we can see the big variation of refinement time for MPI which is smoothed down by PREMA’s implicit load balancing. More specifically, while in the plain MPI case the refinement time varies by up to 754 and 488 seconds, PREMA manages to take the variance down to 239 and 215 seconds. 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. 5 is that even though the load distribution has been improved it still not optimal. There are cores that have run for a much longer time than the average. The same observation can be made from Table 2 where the variance between minimum and maximum refinement time is much larger than the one noticed in the synthetic benchmark. This problem is caused by the quality of the mesh decomposition. Even though the adaptive octree tries to create sub-domains of similar size, the refinement time per sub-domain can differ dramatically.

For example, we have cases where the refinement of a single sub-domain could last as much as the refinement of hundred others, dominating the overall time of the refinement. And since concurrency is performed 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 the reason that, while PREMA significantly improves the performance of the application compared to MPI, it does not scale as expected when we increase the number of cores. To address this issue we plan to modify the application to incorporate data decomposition on top of domain decomposition that will allow parallelism inside a sub-domain.

### 4 Related Work

Systems like AC [13], Split-C [14] and UPC [15] are some of the first systems to provide a partitioned global address space (PGAS) environment for parallel computing as an extension to 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. The way that global access is provided is through arrays that are spread to among all computing nodes with each node having affinity to a specific subset of the array. However, data migration is not supported and the user needs to maintain the address space consistency if data migration is required.

Another issue with the aforementioned systems is that accessing remote and local data is done in a uniform way which makes it difficult for the user to understand which data accesses will cause inter-node communication and as such they rely heavily on compiler optimizations for performance. Other systems (e.g. Co-array Fortran [16], Global Arrays [17], Titanium [18]) that followed similar way of expressing global access but made inter-node communication explicit by using different interface between local and remote accesses in order to avoid such issues, however,
of enhancing it with multiple threads that are dedicated to specific operations like message passing and computation executions. Scheduling has also been exposed for customization in the shared memory level. We showed that this multithreaded model does not heavily impact 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 program. We have noted an improvement of up to 43.5 percent compared to a plain MPI implementation without load balancing in different amount of core allocations ranging from 640 to 5600 cores with a negligible overhead of less than one percent.

In the future we intend to replace our threading mechanism with that provided by Argobots in order to take advantage of their lightweight user-level threads and context-switching. On top of this library our system will be enhanced with task dependencies and out of core support based on our work in the past [24] which was utilizing the previous version of PREMA. Furthermore, support for efficient utilization of deep memory hierarchies that consist of many layers of memory (e.g. HBM, NVRAM, burst buffers etc.) will be implemented. Another aspect that we are interested, and becomes much more relevant as the size of computing clusters increases, is fault-tolerance which can be implemented using the mobile objects as the data that will be checkpointed instead of whole process states.

ACKNOWLEDGMENTS
This work is funded in part by the Dominion Fellowship, NSF grant no. CCF-1439079, NASA grant no. NNX15AU39A, NASA Transformational Tools and Technologies (TTT) Project of the Transformative Aeronautics Concepts Program under the Aeronautics Research Mission Directorate. This work was performed using the Turing computing cluster at Old Dominion University.

REFERENCES

5 Conclusion and Future Work
We have presented the new design and implementation of our runtime system PREMA. We showed the advantages


