# Parallel Software Framework for Large-Scale Parallel Mesh Generation and Adaptation for CFD Solvers

Polykarpos Thomadakis and Christos Tsolakis
*Old Dominion University, Department of Computer Science, Norfolk, VA, 23529, USA*

Konstantinos Vogiatzis
*Engility Inc, Chantilly, VA, 20151, USA*

Andriy Kot
*UIUC, Urbana–Champaign, IL, 61820, USA*

Nikos Chrisochoides
*Old Dominion University, Department of Computer Science, Norfolk, VA, 23529, USA*

**This paper presents the design and implementation of the new version of our software, the Parallel Runtime Environment for Multi-node Applications (PREMA 2.0). The motivation for the development of the runtime system is to provide a software framework that efficiently handles irregular parallel applications whose performance suffers from work-load imbalances. The runtime system provides the minimum set of tools for developing applications that do not explicitly handle load balancing, by monitoring the system load and making dynamic load balancing decisions implicitly, based on user directives. PREMA 2.0 supports multiple execution units allowing the application to run multiple computations in parallel, while it monitors the load of the system and performs migration when desired without interrupting execution. Concurrent migrations from the same process are also made available, giving flexibility to the load balancing algorithm design. Furthermore, the communication and the execution have been separated into different threads to provide asynchronous message reception and instant computation execution at the arrival of new work requests. Testing the system on a synthetic benchmark with load imbalance indicates an overall performance improvement of almost 30 percent, by retaining a better work-load distribution among the execution units.**

## I. Introduction

Large-scale parallel applications require the use of high performance machines that consist of many processing nodes working in a loosely synchronous fashion, in order to achieve good performance. The most efficient use of multi-node computer platforms is a challenging problem due to work-load imbalances of the computations on individual nodes, fault-tolerance and power-aware use of both nodes and the interconnection network. This paper's focus is on load balancing. To accomplish the best possible results in terms of load balancing, it is important to evenly distribute the work-load among the nodes to avoid leaving some of them idle waiting for others to complete their work and wasting resources. Statically distributing the load is not efficient since many of the factors that affect the load distribution cannot be predicted before running an application. To achieve the best results, a dynamic balancing algorithm is needed that will monitor the system in runtime and redistribute the load accordingly.

This paper presents the design and implementation of the new version of our software, the Parallel Runtime Environment for Multi-computer Applications (PREMA). The motivation for the development of the original runtime system is to provide a framework that efficiently handles irregular parallel applications that suffer from load imbalances, by providing the minimum set of tools needed to develop applications that do not explicitly handle load balancing. This burden is left to the Runtime System instead. In order to maintain consistent execution and load monitoring, the runtime system requires that the application makes use of its own execution mechanisms. Since both the computations and the load balancing routines are executed from a single thread, multi-threaded applications that exploit two-level parallelism in an inter/intra node fashion are not well supported from this system. Also, an explicit call of a runtime system function is required to achieve message reception and computation invocation, thus the progress of the system is heavily dependent on the application.

To address these issues, this paper presents a new effort to extend the design of Parallel Runtime Environment for Multi-node (multi-core) Applications (PREMA 2.0) to support multiple execution units allowing the application to run multiple computations in parallel, while the runtime system is still able to monitor the load of the system and perform migration when desired without interrupting execution. Concurrent migrations from the same process are also made available, giving even more flexibility to the load balancing algorithm designer. Furthermore, the communication and the execution have been separated into different threads to provide asynchronous message reception and instant computation execution at the arrival of new work requests. Progress is now guaranteed on the system level without any involvement of the application code; once a request for execution is issued, it will be handled at some point. Because the design between the two versions differs, to enable easy-to-follow comparisons we use the term of MPI rank or simply "rank" to distinguish between sets of threads that share the same address space. For the original version it is a single thread, but for the new design it is a group of threads using the same MPI process.

### A. PREMA 1.0

The Parallel Runtime Environment for Multicomputer Applications (PREMA) is an environment that provides one-sided communication, remote method invocation, globally addressable name-space, data migration and message forwarding, and implicit load balancing targeted to support adaptive and asynchronous applications. The original data domain is broken into N sub-domains (N>> number of processors) to allow more flexibility for the load balancing algorithm. This is known as over-decomposition. Computation is executed asynchronously on each sub-domain by sending a message to the sub-domain regardless of whether the data is located on the local or a remote processor.

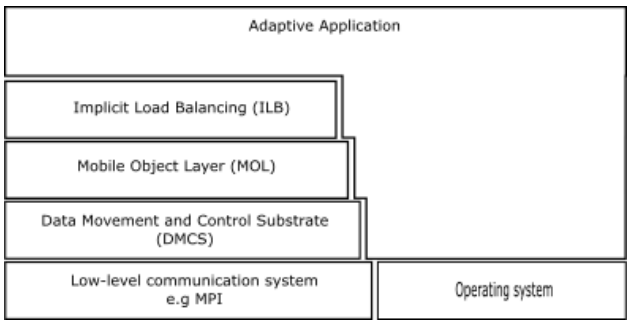


**Fig. 1 PREMA architecture**

These messages called remote handlers constitute the load of a sub-domain. Thus, the load of a processor is represented by the sum of the load from all of its local sub-domains. In order to balance the load of the system, the sub-domains are moved from one processor to another, moving their associated computation messages with them. However, to achieve this asynchronous execution the application needs to explicitly make calls to the appropriate system functions, which will make progress on the message reception, the execution of the user-defined computation functions and also update the structures that are used to initiate the load balancing algorithm. Thereby, calling this function in an appropriate interval is essential for the efficient completion of an application. Furthermore, the environment supports a framework that allows easy and efficient development of custom load balancing policies. PREMA is composed of three separate layers according to the principle of separation of concerns as demonstrated in fig. 1. The layers are built incrementally i.e. the higher level layers make use of the lower ones' functionality, giving the ability to utilize only some of the layers if desired. However, higher-level layers can be added with minimal changes to the application code. Namely, the three layers are:

1) The Data Movement and Control Substrate (DMCS) [1]
2) The Mobile Object Layer (MOL) [2]
3) The Implicit Load Balancing (ILB) [3]

Each layer provides different functionality as described in section III.

### B. Contributions of PREMA 2.0

The new PREMA design provides the same powerful functionality of its predecessor and extends it to a multi-threaded execution model.

1) DMCS has been modified thoroughly to conform to a multi-threaded scheme, separating the communication sublayer, the execution units and the application logic. The multi-threaded scheme, allows for parallel remote method invocations, asynchronous message reception and overlap between computations and communication. Furthermore, remote methods are invoked, without any involvement of the application as opposed to the explicit calls to a system function that was required before.
2) Building on top of this new design, MOL adopts the multi-threading capabilities. As a result, messages that target sub-domains on the same processor can be executed in parallel allowing for better utilization of the system

and reducing latency. Also, communication between sub-domains in the same node is now cheaper since they share the same memory and a push to a queue can replace a message transmission.

3) The runtime system can only monitor work-load that will be executed through its own work pool, thus, supporting multiple parallel computations is essential for the correct and efficient handling of multi-threaded applications. Such an application can map tasks that can execute in parallel to different work units of the system to benefit from the implicit load balancing functionality without sacrificing intra-node concurrency.
4) Since there are multiple workers per node sharing the same memory instead of each worker having its own address space, load balancing for workers residing in the same hardware node is now more efficient.
5) Migrations are not restricted to a single pair of processes at a time, since the system allows concurrent migrations to/from the same target/source. The user can benefit from this using the isolated framework provided by PREMA.

## II. Related Work

There are several ongoing research projects related to parallel runtime system designs for the exascale computing era.

The Open Community Runtime (OCR) [4] is a fine-grained, asynchronous and event-driven runtime. An OCR application consists of data blocks that encapsulate application data, event driven tasks, which perform the computations on the data blocks, and events that define the relationship. Events coordinate the execution of OCR tasks. The OCR program is defined in terms of a collection of tasks organized into a directed acyclic graph. The runtime manages the movement of the data and concurrent execution of the tasks according to the relationships defined in the graph. User data are stored as a collection of independent blocks that can be referenced from anywhere on the system using OCR Events and Event Driven Tasks. To accomplish this, OCR defines a new programming model with a steep learning curve. This approach makes the effort of porting a legacy MPI application on top of OCR much more complicated and error prone. In contrast, PREMA supports a simple API very similar to the one that MPI offers, making this task much simpler.

HPX [5] is another task-based runtime and programming model targeting exascale systems. The execution model of HPX is based on Active Messages called parcels, which encapsulate remote method calls to an object using its global address in the system. Data is communicated in form of arguments bound to the parcel and parcels are transferred between work executing entities called localities. A locality is a set of hardware resources that faces bound latencies e.g. a socket in a compute node. A global address space is provided where objects in each locality can be referenced from the entire system. Execution dependencies and synchronization are made available through the entities of local control objects (LCOs). HPX does not support distributed load balancing although it does so in the shared memory. However, HPX 5 [6], the HPX implementation from the Indiana University, has introduced distributed data balancing, but it's still experimental and requires explicit invocation by the user.

Legion [7] is a data-centric parallel programming system that supports the development of applications targeting distributed heterogeneous architectures. Logical regions are used to describe the organization of data and to make relationships used for locality and independence. Tasks are the execution units, which can access the data in those regions using explicit privileges defined beforehand. 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 prohibits applications from allocating memory using C/C++ conventions for data that are needed after the completion of a task. In such cases, all memory related operations shall be converted to use the Legion's logical regions. This induces extended rewrite of legacy code, which may not always be feasible, particularly when external libraries are being used.

Charm++ [8] is a system with similar characteristics to PREMA. Programs consist of medium grained cooperating message-driven objects called chares. At the invocation of a method of an object an implicit message is sent to the invoked object, which may reside anywhere in the system. The execution of the code within a chare is 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.

## III. PREMA: Software Design

### A. Data Movement and Control Substrate

*1. Original Implementation*

The *Data Movement and Control Substrate (DMCS)* [1] is designed to provide one sided-communication and remote method invocation, similar to Active Messages [9] a key property for the implementation of irregular parallel applications and the implicit load balancing algorithm. It serves as the underlying low-level middle-ware between the application (or the higher-level layers of PREMA) and the communication subsystem. This allows for easy portability of the whole runtime system to a different communication subsystem only by porting this layer, currently, DMCS is built on top of MPI [10]. Its functionalities can be grouped in three main categories, the environmental, memory manipulation and remote service request functions:

1) The environmental functions are used to initialize and shutdown the system, functions that give information about the system such as the rank of a process and the total number of processes in system and a collective that is used to register the remote handlers with the system.
2) The memory manipulation functions provide the ability to allocate and de-allocate memory on a remote process, read/write to the memory of a remote process and also offer an extended set of remote read/write operations which will also invoke the execution of a user-defined remote handler either only to the remote process or both to the remote and the local process after the memory manipulation operation has completed.
3) Remote service requests are user-defined methods that are executed on the recipient of an appropriate message, much like the RPC [11] scheme but without the ability to return a value. However, if a value is needed as a reply to the execution of a request, it can easily be returned through a new reply message.

After initializing the system, the application needs to register the user-defined remote methods, called remote handlers, with the runtime system in all running ranks. In this step, a handler table is created that converts user handlers i.e. function addresses to integer indexes. Using this mapping, a remote service request can designate the handler to be executed using its index, which will then be converted to the corresponding function address on the remote rank. For the sending rank the application only needs to designate the function to be executed by its name, the target, the data and their size if any and an optional tag. Since DMCS provides one-sided communication the receiver will automatically receive the message and start executing the corresponding handler.

PREMA 1.0 adheres to a single threaded model and as such the same thread will execute the application operations and the system operations. This means that message reception, remote handler execution and application defined operations need to be handled by the same thread. To be able to run all of these the application needs to explicitly call a method that will poll the network for new messages and execute any pending remote handler requests. It is obvious that the frequency with which the polling function is called can be crucial for the performance of the application since it triggers the execution of remote handlers and thus, guarantees the application's progress and termination.

*2. Multithreaded Implementation*

For the new multi-threaded design the main principles of DMCS still hold, providing the same functionality over a communication subsystem but extending it with multiple threads available to handle its various operations using POSIX threads [12]. A new operation has been added providing the functionality of sending data that are not contiguous in memory but can be referenced with offsets based on a memory address (e.g. sending specific elements of an array). A problem that an application could encounter in the single-threaded implementation is the need for explicit polling so that new messages can be received and remote handlers can be executed. In cases where the application needed to work on something else while waiting for the remote handlers to be executed (e.g. prepare the data for the next round of computations), it would not be able to poll the network and pass control to the remote handler invocations frequently enough and as a result the communication and computation servings could be

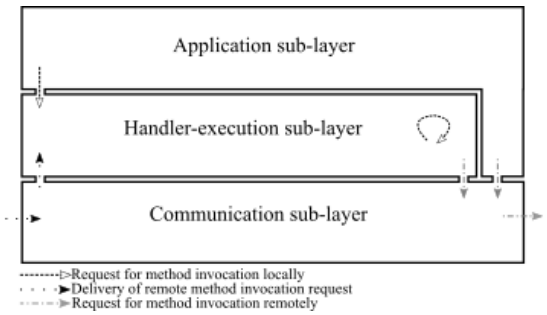


**Fig. 2 A high level view of the multi-threaded DMCS**

delayed for the whole system. Another potential issue could be the sequential execution of the remote handlers. Many applications exploit parallelism in both shared and distributed schemes running multi-threaded applications on top of MPI. The sequential execution of remote requests might limit this kind of applications to a sequential model while they could make use of concurrent execution of remote service requests. In addition to that, using multiple workers per MPI rank instead of a single worker per rank reduces the amount message sending operations by substituting them with pointer passing.

To face these potential concerns the new DMCS has been separated into three independent sub-layers each of which handles different functionalities of the system. Each sub-layer runs on a dedicated thread assigned to a different core, providing concurrent execution of remote methods and also overlap between communication and computation. The three sub-layers that constitute DMCS are the *application*, *handler execution* and *communication sub-layers* working closely together as shown in fig. 2.

The *application sub-layer* runs the application operations defined by the user and is the only layer visible to the user. The initialization and termination of the system, the registration of the remote handlers and the orchestration of the application logic is part of this sub-layer. When the application wants to send a remote request to another rank (local worker) via a DMCS send operation, a message header is created and passed to the communication (handler-execution) sub-layer along with the arguments, if any. Depending on the send method (blocking, non-blocking or synchronous) it might block until the data are safe to be modified or return immediately. A single thread is used on this layer, but the application designer is free to spawn more threads for his application if required.

The *communication sub-layer* is a single thread dedicated to handle the communication demands of the system. Its main operations consist of:

1) Polling the network for incoming messages.
2) Checking for pending outgoing messages.
3) Keeping track of message completion in order to inform the application and unblock blocking sends.

It maintains two lists, one containing pending outgoing messages produced by the local rank and need to be sent to a remote one and one containing remote service requests that have been received from the network and are passed to the handler execution layer to be handled appropriately. The communication thread runs in a loop in which it first polls the network for new messages and receives them until there is none to receive. All the messages received are then pushed into the list containing the remote service requests to be picked up from the handler execution sublayer. Next, it will check the pending outgoing message list and send any pending messages. All sends are asynchronous in the MPI level, this guarantees the absence of deadlocks in a condition where two processes send to each other at the same time. Once each message of this list is sent asynchronously, it is checked for completion. In case it is not completed yet, it is included in a list of non-completed messages. This list of messages is periodically checked to locate any potential completed send in which case, the sender of the message is informed and any blocking operation is revoked.

The *handler-execution sub-layer* consists of a pool of threads that are dedicated to handle remote service requests. Those pre-allocated threads are inactive until a new remote service request is available in the corresponding list maintained by the communication layer. Once a new request is pushed to the list, a thread of the pool will turn active, pick it and execute it. At completion, the thread will go back to check for a new requests in the list. If one is found, it will follow the same procedure, if not it will suspend its execution again until the communication thread pushes a new request and signals it. The existence of only one thread for the reception and sending of messages results in guaranteed message ordering per sender. However, since there are many threads that execute the requests it is possible for the execution of a remote handler that was received at time $t_1$, to complete before execution of the handler that was received at time $t_0$ with $t_0 < t_1$.

A high level view of the operations performed by each thread is presented in fig 3

### B. Mobile Object Layer

*1. Original Implementation*

The Mobile Object Layer (MOL) [2] is the second level layer of PREMA built on top of DMCS extending its functionalities by providing a globally addressable name-space, message forwarding and data migration capabilities. It introduces the constructs of mobile objects and mobile pointers to provide this functionality. Mobile objects are user defined data objects that can be migrated on demand and may encapsulate data not residing in contiguous memory. A mobile pointer is a pair of the rank that created this mobile object, and an index number. This pair is unique for the whole system and is used to identify each mobile object in the system even if it has migrated to different ranks. Making

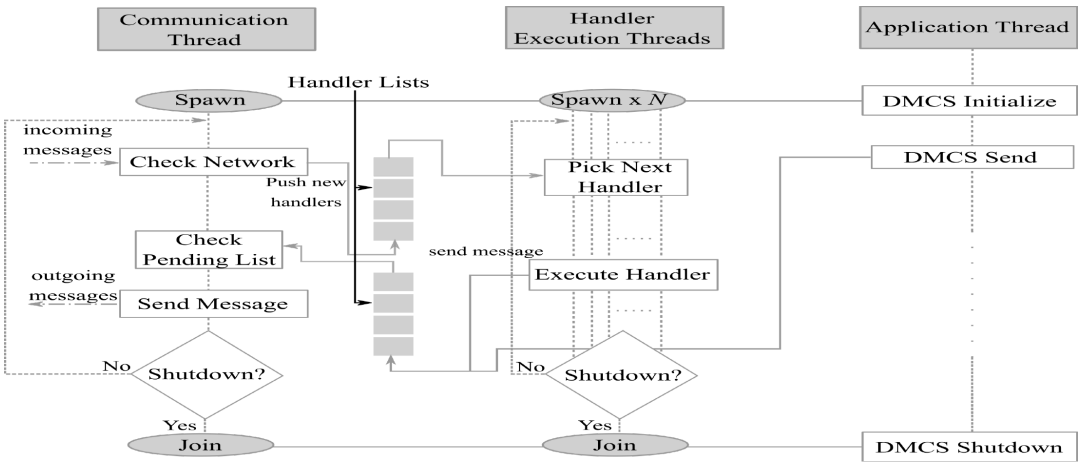


**Fig. 3 The multi-threaded DMCS architecture**

use of this feature, a rank can send a message to a specific mobile object regardless of its location and, by making use of the remote service request of DMCS, execute a user-defined handler on it.

The MOL maintains the location of each mobile object in a directory. In order to send a message to a mobile object a look up to this directory is performed, the location of the mobile object is retrieved and then the message can be sent. To reduce the cost of this lookup, the directory is distributed between the ranks each of them holding its own copy instead of a global directory being kept in a single rank. However, this decentralization introduces the problem of keeping the local directories up-to-date and consistent in the event of an object migration. To face this problem several updating protocols have been implemented [13], namely the broadcasting, lazy and eager update protocols:

1) Broadcasting update. Each time an object migrates to another rank, its new location is broadcasted to all ranks updating their directories.
2) Lazy update. In this protocol only the local directory of the migrating object is updated. When a message for this object comes, it is forwarded to its new location and once it reaches its destination an update is sent back to the source rank of this message.
3) Eager update. For each object the local rank keeps an array of interested ranks. When a message arrives for this object, the sender rank is added to this array and when the object migrates, an update is sent to all ranks in the array and the array is cleared.

Each directory entry contains the rank in which the mobile object was last known to reside and may be out-of-date, a pointer to the local data if the mobile object is local and a sequence number, which is used to check the consistency of an entry. Each time a mobile object moves, this sequence number is increased to indicate this change. Any update to remote ranks regarding this move will be associated with this sequence number. By tagging update notifications with a sequence number, we are sure that a delayed update message will never be able to overwrite the most recent information regarding the location of a mobile object. Two types of messages are available, the *mol_request*, which is a message to be sent to a rank, thus the directory is not used, and the *mol_message*, which is a message to be sent to a mobile object on the system which requires the use of the directory. The *mol_request* simply takes advantage of the DMCS to send the message to the desired rank. The *mol_message* on the other hand will first need to look up the mobile pointer of the mobile object in the directory where it might have one of the following results:

1) The object is local. In this case the message can be handled locally.
2) There is an entry on the directory that points to another rank. In this case, the object is forwarded to this rank; even if the entry is out-of-date the receiving rank will further forward the message to the location where the mobile object migrated.
3) Finally, the directory may have no entry for this mobile object; in this case the message is sent to the rank that originally created the mobile object. This information is embedded in the mobile pointer itself.

A demonstration of the MOL messaging and forwarding mechanism is presented in fig. 4. Because of the forwarding mechanism of MOL, messages can be received out-of-order since MPI only guarantees ordering in messages between pairs of ranks. To address this issue MOL needs to guarantee message ordering between pairs of ranks and mobile

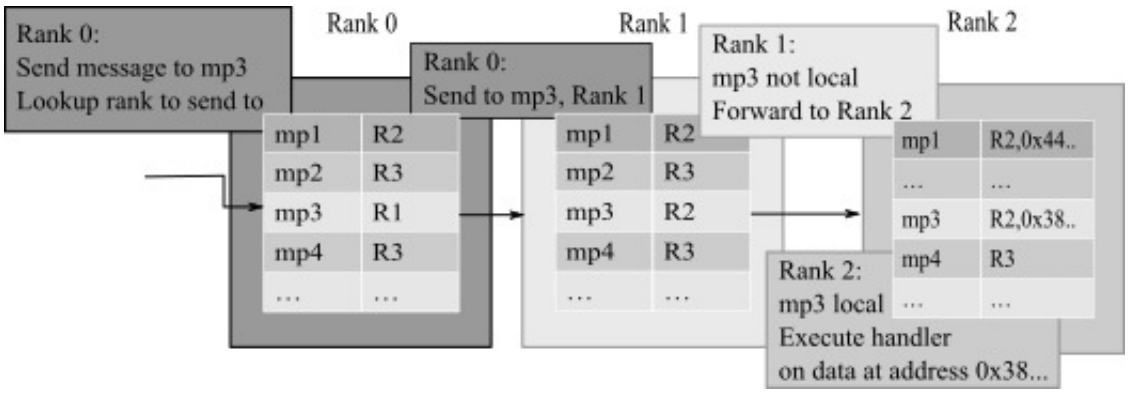


**Fig. 4 A demonstration of the MOL distributed directory and message forwarding functionality**

objects, two messages sent by the same rank must always be handled in the correct order whether they have been received via forwarding or sending directly. Each rank keeps an outgoing message sequence number for each known mobile object and a set of incoming message sequence numbers (one incoming sequence per rank) for each local mobile object it currently holds. When a rank sends a message to a mobile object the outgoing message sequence number for that object is attached to the message and then increased. The receiving rank can check if the message is in order by looking at the sequence number it has for the sender rank referring to the local object. If the message is in order it is executed, otherwise, it is delayed until its sequence number is the current one. The message sending and receiving of the *mol_request* procedure is the same as the one of the DMCS with the difference that it uses the DMCS as the communication substrate instead of the MPI and the handlers are put in an execution queue to be handled later. For the *mol_message*, the sending operation involves tagging the message with the current sequence number and then passing the message to DMCS. The receiving part involves executing a DMCS handler that makes the required checks with the distributed directory to decide whether or not the message can be enqueued to the execution right away or not based on the cases mentioned earlier. As in DMCS, the MOL handlers stored in the execution queue are executed when the application calls an explicit polling function and not inside the DMCS handler that triggered in reception of the message; this function will also call the DMCS polling function to guarantee progress. A *mol_message* handler will first check the directory again to make sure that the object is still present and the message is in order, before being executed.

The MOL provides two functions that allow mobile objects to migrate to other ranks, namely *mol_uninstall* and *mol_install*. The first is used to update the local directory about the new location of the mobile object and to collect important information in a structure that needs to be transferred with the mobile object to its new location. After manually packing the data along with the structure returned from the mol_uninstall the user can move the mobile object to another rank. The receiving rank shall then unpack the data and call the mol_install function that will update the local directory about the new mobile object and make it available for remote handler invocations.

*2. Multi-threaded Implementation*

The new multi-threaded design of MOL makes use of the improvements in DMCS supporting the same functionalities as the single threaded version with the added feature of concurrent execution. The addition of concurrency inserts new capabilities but also new challenges that had to be faced in order to make the MOL functional. For the *mol_request* there is no difference from the DMCS remote service requests and the concurrency is handled correctly by the DMCS. However, *mol_message* needs to make use of the distributed directory in order to operate. These handler execution requests are no longer stored in an execution queue to be executed later, but they are instantly executed from within the DMCS handler by one of the available handler threads.

Concurrent accesses and modifications of the directory structure could trigger race conditions among the handler executions. Having a mutex that would only allow one thread to access the directory each time, would solve the problem, however, it would create contention problems limiting performance scaling. At this point only the lazy update protocol has been implemented for the multi-threaded MOL. The directory has been implemented as a static hash table with chaining, i.e. collisions are solved using linked list per hash table slot. By static in this context we mean that the hash table does not expand or shrink depending on the number of its elements. This allows concurrent access to different

entries without any synchronization, for example a new mobile pointer can be inserted into the directory while another is being updated at the same time.

The entries of the directory are not removed until the termination of the application and their memory is reused when the mobile object is destroyed by the application. It is up to the application to avoid referencing a mobile pointer whose data have been de-allocated, exactly like it shall not reference a C pointer to memory that has been freed. By guaranteeing that there are no deletions in the hash table, we can provide multiple concurrent insertions even if they target the same slot of the hash table using the atomic operation, *compare and swap* (CAS) that guarantees that the insertion will happen only if there has been no other thread that touched the slot at the same time. This technique is very well known, but it suffers from the *ABA* problem [14]. However, when there is no deletion, the ABA problem does not occur and thus concurrent accesses to the structure are safe.

To allow concurrent access to the outgoing and incoming message sequence numbers, the sets regarding those sequences have been moved into the directory structure for each mobile pointer available, instead of keeping them separately as it was before. The hash table is thread-safe and so are those set if we take only handler executions into account and not migration for now. Since many *mol_messages* can be sent concurrently, the sequence numbers for the outgoing messages are increased atomically to guarantee that each number will be unique. For the receiver side, there is no need to use an atomic operation to check the current expected sequence or to increase it to the next one, since only the handler with the message that holds the current sequence number will be able to update it. After this check and update have completed, multiple remote handlers can execute concurrently on the same or different mobile objects. The MOL provides object-specific mutexes that can be used by the application in order to achieve mutual exclusion between handler executions that target the same mobile object. The application can use those mutexes inside the handler functions via the *mol_lock* and *mol_unlock* operations provided by MOL. External mutual exclusion mechanisms can still be used if needed.

Let us now consider the possible race conditions that could occur, taking migrations into consideration. The following operations can happen concurrently:

1) Two or more installations
2) Two or more uninstallations
3) One or more installations with one or more uninstallations
4) One or more installations with one or more handler executions
5) One or more uninstallations with one or more handler executions.

*Two or more installations*: They can target the same or different slots of the hash table. For different slots there is no race condition; they access different memory addresses. In the case of targeting the same slot and at least two previously unknown (for this rank) mobile objects are installed there is a race condition that is handled by the CAS method. If previously known objects (for this rank) are being installed, there is no new insertion in the hash table, just updates to different entries, and thus no race condition.

*Two or more uninstallations*: Those can only occur on different mobile objects which are also known to the current rank. Different entries will be modified to reflect the effect of uninstallation and since no entry is removed or added from the uninstallation operation, there are no race conditions.

*One or more installations with one or more uninstallations*: When they target different mobile objects, there is no race condition, since different entries in the hash table are updated and no deletion or insertion takes place. However, when they target the same mobile object that is a race condition. To deal with this problem, a system level lock and its counterpart try_lock are provided. These functions shall be used when a mobile object is to be migrated to another rank. They will check if a remote handler is executing on the mobile object and if one is, the lock function will block until all currently running handlers on this object have completed and the try_lock will check if some remote handler is running and return the result. On success, the mobile object will be unavailable for handler execution and so it will be safe to migrate to another rank.

*One or more installations with one or more handler executions*: Again, when they target different mobile objects there is no race condition. Even when they target the same mobile object, both executions will eventually complete successfully but we may have a number of messages forwarded in a circle and back to the rank.
Suppose that a rank $R1$ moves a mobile object to another rank $R2$ and directly after that, it sends some *mol_message(s)* to this mobile object. The messages will be received in order, but the execution of the MOL handler(s) might start before the installation of the object has completed. In such a scenario, the MOL will detect that the object is not local and will forward the *mol_message(s)* to the rank last known to have this mobile object by $R1$. After some forwarding, the *mol_message(s)* will eventually be forwarded back to $R1$. We overcome this issue by tagging each *mol_message* with the directory sequence number, which is increased each time the mobile object is uninstalled. Because the rank has

not yet properly installed the mobile object, it will have a directory sequence number that is less or equal to the directory sequence number of the *mol_message(s)*. This will imply that the object is on its way to move locally and that it should delay the execution of the message(s) until the object is installed properly.
Finally, if a MOL handler requires the uninstallation of the mobile object it is executing on. As mentioned before, the only way to uninstall a mobile object is to lock it first. Trying to lock the mobile object, however, will fail because there is already a handler executing, the same that tries to uninstall the mobile object. To overcome this issue the *mol_message* function now has a flag parameter that can distinguish the message as a handler that requires exclusive access to the mobile object it runs on. Those handlers will start executing only when there is no other handler running on the same mobile object, and they will lock the object implicitly until execution is completed. This allows the handler to uninstall the object without explicitly calling *mol_lock*.

### C. Implicit Load Balancing

*1. Original Implementation*

PREMA's Implicit Load Balancing (ILB) [3] 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. ILB also provides the flexibility of developing and using custom load balancing policies by supporting an isolated scheduling module available to the user. Those two components work tightly together to achieve a data aware implicit load-balancing framework requiring minimal changes to the application.

After the creation of a mobile object, a function provided by ILB is used to register the mobile object with the system. Along with the mobile object, a set of user defined call-back functions are registered that allow the runtime system to gather information about the system and make decisions regarding load balancing. The information that can be retrieved from these callbacks is briefly: The effort needed to migrate the object to a remote rank (e.g. because of the high cost of serializing its data), an affinity map that specifies the rank that will benefit more from the migration (e.g. by reducing communication overhead with objects that reside in the other rank) and the load that this mobile object has, based on the handlers that are pending for this object. Through these call-backs the application can influence the load balancing decisions by minimizing migrations, maximizing smoothness of the work-load distribution and maintaining co-locality of objects that share data dependencies. By associating each mobile object with its pending work-load, expressed as pending handlers, the total work-load of a rank is evaluated as the sum of the work-loads of every mobile object residing on it.

As mentioned before, a mobile object can consist of any data structure stored in contiguous or non-contiguous memory. Therefore, some extra callbacks are required from the ILB to be able to transparently migrate an object. Those are: a packing, an unpacking and a size returning routine. Each of the callback routines is registered with the runtime system only once at the initialization step and is associated with each mobile object on its creation. Once each mobile object has been registered with its callback routines, it can migrate to different ranks when needed.

One function, *ilb_message*, is used to send messages that will trigger the execution of a remote handler. This function uses the *mol_message* operation to send a message to a specific mobile object. The ILB message received will first pass from the two underlying layers, through system handler executions, that will run the operations needed to pass the message to the next layer. The DMCS handler passes the message to the MOL, then the MOL executes a system handler that performs all the checks described in section III.B and puts the handler in the list of handlers of the particular mobile object. After pushing the handler to the list, it notifies the load-balancing layer about the new work-load available and the new load of the rank is computed based on the user-defined callback routines. Remote handlers are executed on the next call to the polling function of ILB. Each mobile object is picked up and has its handlers executed until no mobile object has pending handlers for execution. The ILB polling function calls the respective functions of all the underlying layers, in order for the remote handlers to be executed and the system to progress.

On each call to the polling function, and before the execution of any ILB handler, the current load of he rank is checked, and if it has fallen under a user-specified threshold the load-balancing algorithm will start. Information collected from the ILB is passed to the scheduling module that needs to conform to basic template supported by ILB and makes the balancing decisions. Once this template is overridden, virtually any load balancing policy can be used depending on the needs of the application. Three algorithms are provided by the framework: the Receiver Diffusion, the Master-Worker, and the Prioritized Multi List scheduler, but the user is free to create his own. Once the migration candidates are found, the mobile object is packed using the appropriate call-back function and uninstalled from the rank. Once received from the new owner, it is unpacked using again the appropriate function and then installed. After the

balancing completes, the rank will switch back to execute computations until the work-load falls below the threshold again. It is important to note here that the computations continue for the ranks that have enough work-load while load balancing messages are exchanged by periodically spawning a thread that polls the network for load balancing messages only and reply to them.

*2. Multi-threaded Implementation*

Built on top of the new multi-threaded MOL and DMCS, the ILB now supports concurrent execution of multiple handlers for arbitrary mobile objects. Also, concurrent migration of multiple objects at a time is supported, and in addition to that, computations and migrations can overlap if there is enough work-load on the rank that transfers a mobile object. Furthermore, load balancing is also provided implicitly in the shared memory level.

The functionality of the communication thread has been expanded for the load-balancing layer. A map of the local objects and the number of each handler waiting to execute on it is maintained. When a new ilb_message arrives, the communication thread will update this map to represent the number of handlers waiting for this mobile object. When a handler is about to execute, the corresponding value of the map is decreased. Since we track the number of handlers per object, it is easy to calculate the actual work-load of each rank .A system handler is spawned periodically to check the number of handlers per object and if it has changed since the last time it was checked, calculate the load per mobile object. When the load of an object has been updated for all its pending handlers, the scheduler is notified about this change. Only one system handler can execute this operation at a time to ensure that the handler threads are not kept busy with book-keeping and also to prevent inconsistency in the calculation of the overall load for the rank.

After calculating the total load of the rank, the system handler will check if the work-load is low enough to initiate the load-balancing algorithm based on the scheduler policy defined by the user. For now, only the Receiver diffusion algorithm has been implemented in the multi-threaded model. When the conditions are met, the handler will send messages to all of neighbors asking for their loads. Each rank can only agree to transfer or receive work units from one rank at a time, so that it will be able to keep track of the load it has and does not migrate more than needed. Because, any handler thread can answer load-balancing messages there is a possibility that a rank that has some work-load will be asked for migration by many of its neighbors. Each of its threads will find out that its work-load is enough for some of it to migrate, resulting in eventually migrating all its work-load. To prevent this from happening, only the first rank that requests for migration will be served at a time. The others will be refused and they will pick another neighbor to communicate. Each message between these two ranks can be handled by different threads, thus allowing multiple mobile objects to be transferred by different handlers. Even though the transfer from the network can be handled by only the communication thread, the others can concurrently pack/unpack the requested mobile objects and uninstall/install them. The multi-threaded model makes the design of the scheduler a little more complex, since the designer needs to have in mind that those handlers will execute in parallel. As mentioned in section III.B before migrating a mobile object the system needs to make sure that no handler is executed on it at the same time, using the lock/try_lock functions. The user can decide whether to wait for a specific mobile object to be available for migration or pick the first one that is instantly available.

Because all handler-executing threads could be busy executing long computations, replying to load balancing messages can be delayed when those are stuck in the end. To face this problem, a thread is spawned when all others are busy, in order to answer those messages in time. It will also execute the system handler that calculates the load of the rank to keep such information up-to-date, but will never initiate the load balancing process.

In addition to the distributed memory load balancing, the multi-threaded ILB provides implicit shared memory load balancing. The work pool that each of the handler execution threads is picking work from is shared between them, and as a result, every handler thread that belongs to the same rank is able to pick handler requests that target mobile objects that are local to the rank. Having this two-layer load balancing scheme is very beneficial to the application. At the first level, the shared memory one, balancing the load is provided much more easily and efficiently by just sharing pointers to common work pool. When there is not enough work remaining for the current rank, the thread that is the first to find that out will initiate the load balancing algorithm to retrieve some work units from a different rank. To further improve the efficiency of the load balancing, it can also request more work units in order to give them to the rest of the local threads that will soon run out of work, minimizing the time spent in communication.

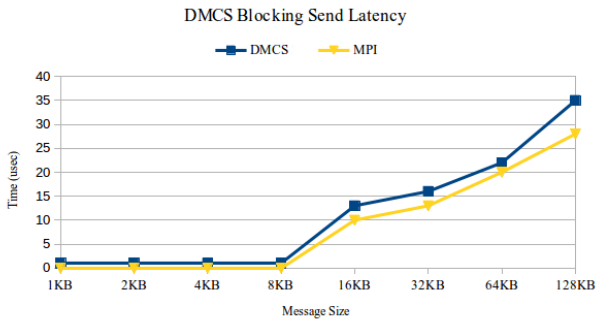


**Fig. 5 Latency comparison between DMCS 2.0 and MPI for different message sizes**

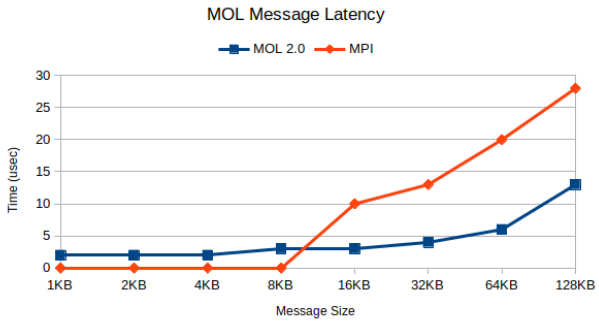


**Fig. 6 Latency comparison between MOL 2.0 and MPI for different message sizes**

## IV. Preliminary Results

### A. Experimental Setup

A series of benchmarks have been conducted to evaluate the performance and overhead of the three layers of the runtime system. The experimental platform that we use is the Turing computing cluster at the High Performance Computing Center of Old Dominion University. The main cluster of the HPC Center contains 190 multi-core Intel(R) Xeon(R) (E5-2660, E5-2660 v2, E5-2670 v2, E5-2698 v3, E5-2683 v4) compute nodes, each containing between 16 and 32 single-threaded cores and 128 GB of RAM. For the purposes of the following experiments, only nodes with E5-2698 v3 and E5-2683 v4 processors are used each of them having 32 single-threaded cores. The fast cluster inter-communication is provided by an FDR based Infiband infrastructure. We used the MPICH 3.1.3 MPI implementation and the 6.3.0 version of GCC compiler for the requirements of these experiments.

### B. Experimental Results

*1. DMCS*

The performance of the DMCS layer is presented first. The blocking variation of the message passing function provided is tested. The time reported is the time conceived from the user as waiting time until the next command starts executing. We compare the time needed for the MPI and DMCS 2.0 to complete such a transaction using a ping-pong benchmark. One rank sends an X size message and waits for a response, then it sends another one and waits again. Each round is executed 100 times and the average is reported for each message size. Latency is considered as the time needed to unblock with the buffer being safe to be used again. Since the new DMCS sends all messages asynchronously it is difficult to calculate the exact overhead added, so we use the same benchmark with the MPI to compare their latencies.

Figure 5 shows the latency of DMCS blocking send operation compared to the MPI-only counterpart. The increase in the latency is accounted to the following factors. First of all, the message sending operation is split in two, one message is sent containing the header needed by the DMCS to handle the message properly and a second one contains the actual data. This means that two messages have to be sent for each sending operation. Second, in the MPI-only version, there is an explicit blocking receive for each blocking send performed, since there is nothing else that needs to be done in the meantime. On the other hand, DMCS handles receives implicitly without knowing when it has to check for new messages and, in addition to that, must be able to service multiple sending operations. This requires having to periodically check for new incoming messages, outgoing message requests and also query in-process sending operations for completion. All this book-keeping adds up to this overhead on top of MPI. Last and least significant is the fact that messages have to be pushed to list before being sent with MPI, which also increases the critical path for messaging.

*2. MOL*

To evaluate the performance of the new MOL layer we executed the same ping-pong test as for the DMCS layer. We should note here that a mol_message operation does not immediately send the message to its destination but it makes a copy to allow the application to continue with the next operation without waiting for the communication to

finish and the message is sent asynchronously later in time. This is the time conceived as a mol_message operation and this is reported in the results in fig. 6. Again, we compare this time with the MPI blocking send operation to see the time difference. The MPI only guarantees that the buffer to be sent is safe to be modified and not that it has reached the destination, the same as the mol_message operation. We see that for small message sizes the latency to copy the message is actually higher than what MPI does. However, for medium sized and big messages copying is much less significant, and since this offers no blocking send capabilities, it is a reasonable overhead to pay in order to continue with execution of computations right away.

*3. ILB*

The performance of the ILB layer is evaluated through a synthetic benchmark that we developed for this reason. With such a benchmark it is easier to examine the load balancing component itself, having full control of the way the application behaves, without unexpected side-effects, and as such have more certainty about the results. The benchmark begins by dispersing work units to available processors, computation is the invoked via PREMA's messaging mechanism. Once computations of a data object are complete, a notification is sent back to the root processor. The application terminates once all notifications have been received. The number of work units per available core is set to 10, and the weights of the individual units are assigned to two categories, light and heavy. The average time of a heavyweight unit is 2.5 times the time of a lightweight one and 20 percent of the work units are assigned to the heavy category, while the diffusion algorithm is used to provide load balancing. Note that even though PREMA uses one core exclusive for message handling, this core is also counted as an available core when we calculate the number of work units to distribute to each rank (i.e. the same number of work units is given in both cases).

We evaluate the effectiveness of ILB by comparing the work-load distribution and the total runtime with the same benchmark run on plain MPI. It is important to note that in the case of MPI each worker is a separate MPI rank, however, in the case of PREMA, each hardware node uses one core dedicated for communication and the rest (31 in this case) are assigned as handler execution threads(workers) in the same MPI rank. Cores in nodes are allocated so that all of the cores of a node will be allocated before moving to the next node. All nodes contain 32 cores and as a result 32 MPI ranks reside in each node for the MPI-only case and one MPI rank per node in the case of PREMA. Figures 7 and 8 show the work-load per worker and execution time with 320 cores and 3200 work units for ILB and MPI respectively. The colored segments represent application work units; this helps to make the heavy tasks easily recognizable and to show how the work units have migrated among the workers in the case of PREMA. Only the time spent in computations is presented here, and thus only for the cores that execute them, since this is the heavyweight task and the only one that PREMA can mitigate through load balancing. Table 1 shows the effect of PREMA in evenly distributing the work-load across the workers by decreasing the time difference between the worker that finished first and the one that finished last from 380 seconds to 33. Furthermore, from the table, we can see that the overhead of ILB, consisting of packing/unpacking and migrating mobile objects and information dissemination, is negligible compared to the overall runtime of the application. As a result, the total runtime of the benchmark is reduced by roughly 28 percent.

We also test the performance of PREMA against MPI-only in a higher core count. We conduct the same experiment described before increasing the number of work units as we increase the number of cores. Figures 9 and 10 show the work distribution when we use 640 cores in total for each case. In fig. 11 and 12 we show the work distribution for 1280 cores and 12800 work units in total. We can see that PREMA is not influenced by the number of cores or nodes in the system; it will derive a nice load distribution if it has the required flexibility which is provided by an efficient over-decomposition. The overall runtime improvement remains approximately the same for all core configurations and it even increases slightly due to the increased number of MPI ranks that are spawned in the MPI-only case. Table 2 also proves this claim. The overhead accounted to the runtime system remains at the same levels for each of the core configurations, while the overall time slightly increases since there are more nodes in the system that need to be reached adding up to the time spent in communication. Of course, this overhead is also depending on the load balancing algorithm that is used, but as we see the diffusive one we use does a good job keeping the costs down. This is achieved by separating the ranks in "neighborhoods" and allowing each rank to request work from ranks that reside in the same neighborhood. If there is no neighbor able to contribute some work, then the rank will randomly choose new neighbors and start requesting again. By following this scheme, the time spent in querying for information is kept constant and does not heavily depend on the total number of ranks in the system.

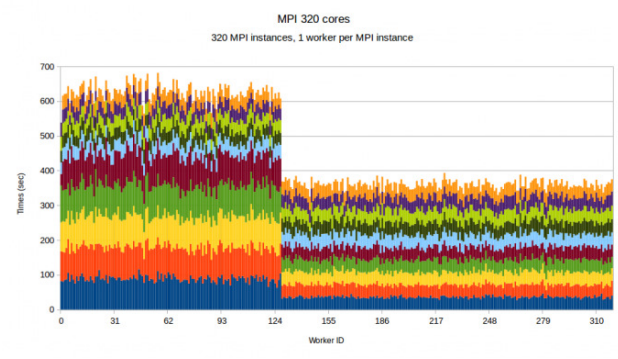


**Fig. 7 Work-load per worker of MPI using 320 workers**

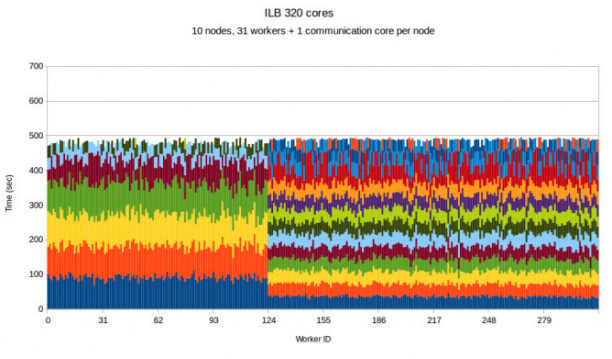


**Fig. 8 Work-load per worker of ILB using 310 workers + 10 communication cores**

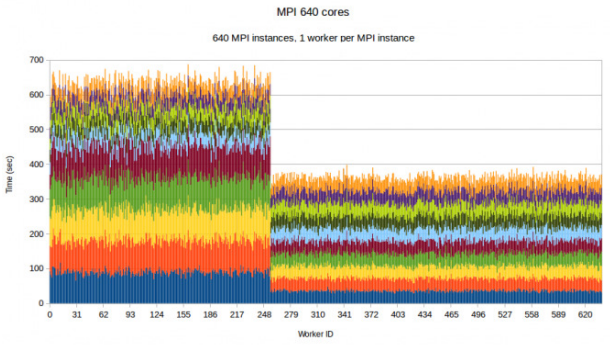


**Fig. 9 Work-load per worker of MPI using 640 workers**

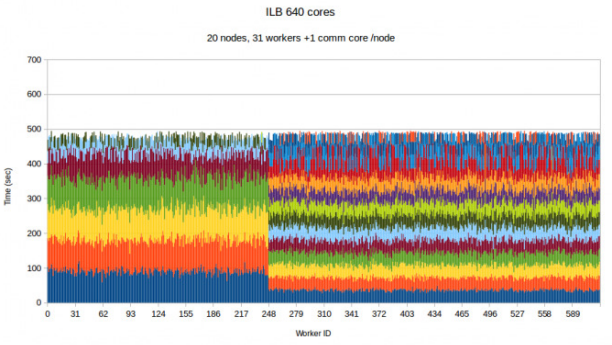


**Fig. 10 Work-load per worker of ILB using 620 workers + 20 communication cores**


**Fig. 11 Work-load per worker of MPI using 1280 workers**

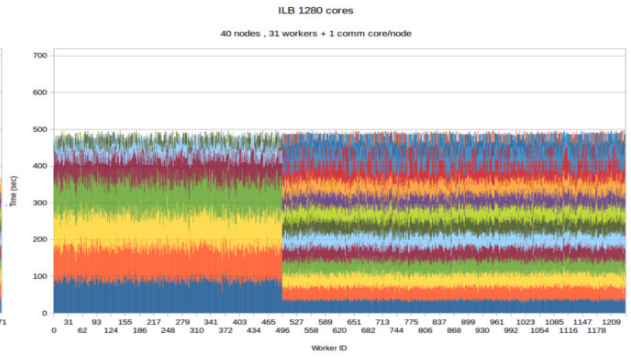


**Fig. 12 Work-load per worker of ILB using 1240 workers + 40 communication cores**

**Table 1 Time breakdown and comparison with the MPI-only version using the synthetic benchmark**

| | Load balance overhead (sec) | | | Min computation time (sec) | Max computation time (sec) | Total time (sec) |
|---|---|---|---|---|---|---|
| | Max | Min | Avg | | | |
| MPI-only | - | - | - | 324.9 | 681.8 | 683.2 |
| ILB | 0.84 | 0.0001 | 0.07 | 464.8 | 494.4 | 495.5 |

**Table 2 Time breakdown and comparison with the MPI-only version for different core allocations using the synthetic benchmark**

| #cores | Load balance overhead (sec) | | | Min computation time (sec) | | Max computation time (sec) | | Total time (sec) | |
|---|---|---|---|---|---|---|---|---|---|
| | Max | Min | Avg | ILB | MPI-only | ILB | MPI-only | ILB | MPI-only |
| 320 | 0.84 | 0.0001 | 0.07 | 464.8 | 324.9 | 494.4 | 681.8 | 495.5 | 683.2 |
| 640 | 0.82 | 0.0001 | 0.05 | 464.1 | 321.4 | 495.1 | 687.1 | 496.8 | 692.7 |
| 1280 | 0.76 | 0.0001 | 0.05 | 464.9 | 324.7 | 497.6 | 704.1 | 499.3 | 713.1 |

## V. Conclusion and Future Work

We have presented the new design for the PREMA 2.0 software and demonstrated its effectiveness We discussed the potential performance gain by using such software for load balancing and for any irregular parallel application that can make use of its message driven execution model and the global name-space that it provides. We keep the lessons learned from the redesign of such a system and intend to investigate in depth the reasons that provoke the problems that we face in our attempt to improve the system. This was just the beginning in the road of implementing a robust and powerful parallel runtime system that will meet the expectations of the extreme-scale computing era. We also note the importance of having an effective decomposing algorithm that will be able to provide a decomposition that will enable us to have a well-distributed work-load among the available ranks.

For our future work, we intend to try different ways of improving the performance of DMCS and MOL to achieve a lower overhead on top of MPI or another communication substrate. In addition to that, our group plans to invest in the redesign of our Multi-layered Runtime System (MRTS) [15] building it on top of the redesigned PREMA software. The MRTS provides out-of-core execution for applications that need to manipulate data much bigger than those that can fit in the main memories of the nodes in the parallel system of cluster. To accomplish this, it makes use of the lower level memory (i.e. NVRAM, burst SSD storage) to temporally store data there and swap them in-memory in time when they are needed again for computations. The important factor in this process is to do it while overlapping the access latency to lower layers with computations. The data are assigned to mobile objects that reside either on the higher or the lower layers and once work is available in the sense of user-defined handlers, the mobile objects migrate from the lower layer to the higher ones in order to start execution. We are also planning to integrate this system with the Argobots [16] framework. Argobots is a low-level infrastructure developed as part of the Argo [17] project of Argonne National Laboratory. It supports a lightweight thread and task model to provide massive concurrency. Its execution model consists of two levels of parallelism: Execution Streams and Work Units. The Execution Stream (ES) is a kernel-level thread that contains several Work Units (WUs). WUs are user-level work units associated with an ES. The concurrency is achieved in the granularity of ESs, so two WUs can execute concurrently only if they are associated with different ESs. WUs are further distinguished in User-level Threads (ULTs) and Tasklets. ULTs provide fast context switch and Tasklets are lightweight work units that run to completion.

Argobots support interoperability with MPI, overlapping computation with communication implicitly. Multiple blocking MPI calls can be made concurrently by multiple ULTs of the same ES. When a ULT is blocked, Argobots will detect it and context switch to other ULTs available, this allows us to keep the system busy with useful work instead of waiting. DMCS also supports calling communication operations from different handlers at the same time but a thread will block until its blocking operation finishes delaying other handlers that could execute at the same time on the same thread. PREMA could be ported to make use of the functionalities of Argobots to provide a better integration between computation threads, communication and fast context switch in user-level.

## Acknowledgments

This work is funded in part by the Dominion Fellowship, NSF grant no. CCF-1439079, NASA grant no. NNX15AU39A, NASA's Transformational Tools and Technologies (TTT) Project of the Transformative Aeronautics Concepts Program under the Aeronautics Research Mission Directorate. We would like to thank the systems group in the High Performance Computing Center of Old Dominion University for their great help and support. We would also like to thank Pete Beckman and Halim Amer from ANL for the valuable discussion we had about their system Argobots and the possible integration of our system on top of it. This material is based upon work supported by, or in part by, the Department of Defence (DoD) High Performance Computing Modernization Program (HPCMP), under the User Productivity Enhancement, Technology Transfer and Training (PETTT) Program, contract number GS04T09DBC0017. We would like to thank the DoD Supercomputing Resource Centers for making their computational resources available to us.

## References

[1] Barker, K., Chrisochoides, N., Nave, D., Dobellaere, J., and Pingali, K., "Data Movement and Control Substrate for Parallel Adaptive Applications," *Concurrency and Computation:Practice and Experience*, 2002, pp. 77–105.

[2] Chrisochoides, N., Barker, K., Nave, D., and Hawblitzel, C., "Mobile Object Layer: A Runtime Substrate for Parallel Adaptive and Irregular Computations," *Adv. Eng. Softw.*, Vol. 31, No. 8-9, 2000, pp. 621–637. doi:10.1016/S0965-9978(00)00032-6, URL http://dx.doi.org/10.1016/S0965-9978(00)00032-6.

[3] Barker, K., Chernikov, A., Chrisochoides, N., and Pingali, K., "A Load Balancing Framework for Adaptive and Asynchronous Applications," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, 2004, pp. 183–192.

[4] Mattson, T. G., Cledat, R., Cavé, V., Sarkar, V., Budimlić, Z., Chatterjee, S., Fryman, J., Ganev, I., Knauerhase, R., Lee, M., Meister, B., Nickerson, B., Pepperling, N., Seshasayee, B., Tasirlar, S., Teller, J., and Vrvilo, N., "The Open Community Runtime: A runtime system for extreme scale computing," *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–7. doi:10.1109/HPEC.2016.7761580.

[5] Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., and Fey, D., "HPX: A Task Based Programming Model in a Global Address Space," *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ACM, New York, NY, USA, 2014, pp. 6:1–6:11. doi:10.1145/2676870.2676883, URL http://doi.acm.org/10.1145/2676870.2676883.

[6] "HPX5," , 2018. URL https://crest.soic.indiana.edu/hpx-5/, [Accessed April 23, 2018].

[7] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A., "Legion: Expressing Locality and Independence with Logical Regions," *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012, pp. 66:1–66:11. URL http://dl.acm.org/citation.cfm?id=2388996.2389086.

[8] Kale, L. V., and Krishnan, S., "CHARM++: A Portable Concurrent Object Oriented System Based on C++," *SIGPLAN Not.*, Vol. 28, No. 10, 1993, pp. 91–108. doi:10.1145/167962.165874, URL http://doi.acm.org/10.1145/167962.165874.

[9] von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E., "Active Messages: A Mechanism for Integrated Communication and Computation," *SIGARCH Comput. Archit. News*, Vol. 20, No. 2, 1992, pp. 256–266. doi:10.1145/146628.140382, URL http://doi.acm.org/10.1145/146628.140382.

[10] "MPI forum," , 2018. URL http://mpi-forum.org/, [Accessed April 23, 2018].

[11] Birrell, A. D., and Nelson, B. J., "Implementing Remote Procedure Calls," *ACM Trans. Comput. Syst.*, Vol. 2, No. 1, 1984, pp. 39–59. doi:10.1145/2080.357392, URL http://doi.acm.org/10.1145/2080.357392.

[12] Nichols, B., Buttlar, D., and Farrell, J. P., *Pthreads Programming*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[13] Fedorov, A., and Chrisochoides, N., "Location management in object-based distributed computing," *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, 2004, pp. 299–308. doi:10.1109/CLUSTR.2004.1392627.

[14] Dechev, D., Pirkelbauer, P., and Stroustrup, B., "Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs," *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2010, pp. 185–192. doi:10.1109/ISORC.2010.10.

[15] Kot, A., Chernikov, A., and Chrisochoides, N., "The Evaluation of an Effective Out-of-core Run-Time System in the Context of Parallel Mesh Generation," *IEEE International Parallel and Distributed Processing Symposium*, 2011, pp. 164–175.

[16] Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., Castello, A., Genet, D., Herault, T., Jindal, P., Kale, L., Krishnamoorthy, S., Lifflander, J., Lu, H., Meneses, E., Snir, M., Sun, Y., and Beckman, P. H., "Argobots: A Lightweight Threading/Tasking Framework," *IEEE Transactions on Parallel and Distributed Systems*, 2016. URL https://scholar.google.com/citations?hl=en&user=-abUTFQAAAAJ&view_op=list_works&sortby=pubdate.

[17] "Argo," , 2018. URL http://www.argo-osr.org/, [Accessed April 23, 2018].