The Evaluation of an Effective Out-of-core Run-Time System in the Context of Parallel Mesh Generation

Andriy Kot
Computer Science Department
The College of William and Mary
Williamsburg, VA
kot@cs.wm.edu

Andrey N. Chernikov, Nikos P. Chrisochoides
Computer Science Department
Old Dominion University
Norfolk, VA
{*achernik, nikos*}*@cs.odu.edu*

***Abstract*—We present an out-of-core run-time system that supports effective parallel computation of large irregular and adaptive problems, in particular unstructured mesh generation (PUMG). PUMG is a highly challenging application due to intensive memory accesses, unpredictable communication patterns, and variable and irregular data dependencies reflecting the unstructured spatial connectivity of mesh elements.**

Our runtime system allows to transform the footprint of parallel applications from wide and shallow into narrow and deep by extending the memory utilization to the out-of-core level. It simplifies and streamlines the development of otherwise highly time consuming out-of-core applications as well as the converting of existing applications. It utilizes disk, network and memory hierarchy to achieve high utilization of computing resources without sacrificing performance with PUMG. The runtime system combines different programming paradigms: multi-threading within the nodes using industrial strength software framework, one-sided active messages among the nodes, and an out-of-core subsystem for managing large datasets.

We performed an evaluation on traditional parallel platforms to stress test all layers of the run-time system using three different PUMG methods with significantly varying communication and synchronization patterns. We demonstrated high overlap in computation, communication, and disk I/O which results in good performance when computing large out-of-core problems. The runtime system adds very small overhead (up to 18% on most configurations) when computing in-core which means performance is not compromised.

I. Introduction

With the increasing computational demands of scientific applications many existing parallel codes need to be scaled by several orders of magnitude. At the same time many of those codes are memory bound and do not take full advantage of added computing power. In fact, it is not unusual for such applications to require hundreds of nodes to achieve sufficient aggregate memory and to run only several minutes. Also, not all applications are suited for such high degree of parallelism which results in many processors/cores idling most of the time.

This work is supported in part by NSF grants CCF-0833081, CSR-0719929, CCS-0750901 and CCF-0916526.

Our solution is to use computational resources effectively by employing less computing power (i.e., fewer nodes) and using out-of-core approach for augmenting the memory with disk storage. We developed several out-of-core applications and demonstrated in [1], [2] the effectiveness of this approach. Unfortunately, a task to adapt an existing parallel scientific code or develop an out-of-core code from scratch is both challenging and time consuming.

To simplify and streamline this process we designed and implemented a practical out-of-core runtime system that supports the execution of large scale parallel applications on a fraction of the nodes that otherwise would be normally required. As such the parallel codes can utilize computing resources effectively and provide end users with added benefits such as abilities to: (1) increase the problem size using the same hardware setup or (2) keep the same problem size but use less of hardware resources (e.g., use fewer cores/nodes in a cluster). Our contribution is an efficient runtime C library that features an easy-to-use API for overlapping I/O and communication with computation, in addition to communication and load balancing functionality supported by its predecessor [3].

We evaluated the runtime system using parallel unstructured mesh generation, a challenging adaptive and irregular application. Mesh generation is memory-intensive, as opposed to computation-intensive, and therefore all data transmission latencies, such as RAM, disc, and network, critically influence the overall performance. Moreover, the number of elements changes (grows) throughout the execution of the application, and the structure of the mesh heavily depends on the input geometry, hence memory allocation needs, for each subproblem, are different.

It may seem counter intuitive but using our out-of-core paradigm with fewer nodes on a shared computing resource (as in multiple users using a job scheduler) could lead to a shorter time between a user submitting his job and getting the results. For example, Parallel Constrained Delaunay Meshing requires about 64GB of memory to generate a mesh of 238 million elements which requires 32 nodes (2GB per node) on our university cluster. The

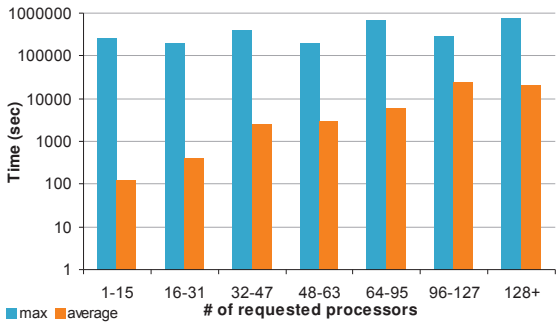

Figure 1. Wait-in-queue time statistics for parallel jobs collected from the last four and a half years from a 300+ processor cluster at the College of William and Mary.

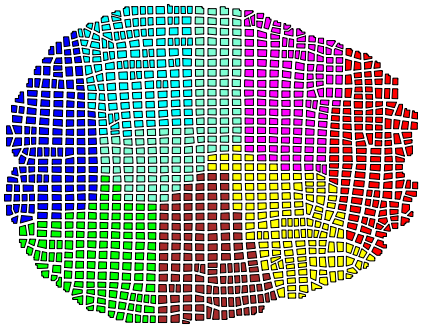

Figure 2. Decomposition of a 2D slice of a human brain MRI into 1024 subdomains mapped onto 8 processors.

execution time is 310 seconds. When ported to our runtime system, the same computation can be done with 16 nodes or less and it takes 731 seconds (16 nodes). Figure 1 shows an example of how long jobs have to wait before they start executing depending on how many nodes they request. On this specific small cluster, requests for less than 16 nodes are scheduled within a couple of minutes while request for 32 nodes wait on average for half an hour, and requests for over a hundred nodes take several hours to start. On average, the out-of-core job in our example will finish in about 14 minutes while the in-core job needs about 35 minutes!

Our work provides additional options to researchers who make use of long running, large scale computing codes. The obvious benefit is to be able to run much larger problems than otherwise possible with limited computing resources. For example, using the proposed system one can utilize a dedicated small workgroup cluster rather than a larger more powerful but shared supercomputer. Another benefit is the ability to use fewer nodes on a shared computing resources and get the results faster by shortening the wall-clock time.

Contribution Design, implementation and evaluation of an out-of-core runtime system aimed at large problems and effective utilization of computational resources. Relatively simple and streamlined process to transform an in-core application into an out-of-core one. Design and implementation of an out-of-core mesh generation application based on state-of-the-art in-core version.

A. Mesh Generation Applications

Parallel mesh generation procedures decompose the original mesh generation problem into smaller subproblems that can be solved (meshed) in parallel, see Figure 2. The subproblems can be formulated as either tightly [4] or partially [5]–[7] coupled or even decoupled [8]. The coupling of the subproblems (i.e., the degree of dependency) determines the intensity of the communication and of the synchronization between processing elements working on separate subproblems. Our results with the tightly coupled approach [4], [9] indicate that it has high overheads for efficient parallel execution, while the domain decomposition for the 3D decoupled method is still an open problem [10]. Therefore, we focus on the partially coupled methods. We stress test the key components of the runtime system (I/O, SMT, and network) using the following three different coarse grain partially coupled approaches to parallel mesh generation.

Uniform Parallel Delaunay Refinement (UPDR) The UPDR method/software presented in [7], [11] is based on the following key idea: for any two given subdomains A and B and their common buffer zone Z between A and B the mesher would mesh concurrently A union Z and B union Z and then remesh Z; the process would stop, as Z is designed to not require any further refinement. This approach balances trade-offs between the costs of data decomposition and of communication, i.e., it utilizes a simple data decomposition method at the cost of some communication and global synchronization. The communication is structured such that during each phase of the algorithm all processes know the recipient and/or the sender of the data that they work with.

Non-Uniform Parallel Delaunay Refinement (NUPDR) The NUPDR method/software [5] extends the UPDR for the case when the application requires that the final mesh have graded (non-uniform) spatial element sizes in different areas of the domain. This method utilizes a quad-tree data structure that distributes the data into blocks corresponding to the leaves of the quad-tree. Due to the variable sizes of the leaves the communicating processes also cannot be known in advance, although there is some regularity in the communication pattern because of the structured way of constructing the quad-tree.

Parallel Constrained Delaunay Meshing (PCDM) The PCDM method/software [6] utilizes the domain decomposition, as opposed to the data distribution, approach to parallelization. I.e., the elements of the resulting mesh strictly conform to the subdomain boundaries, while with the data distribution approach they only overlap in a more loosely defined way. However, the spatial relationship among the subdomains looses its regularity, and the communication

graph becomes completely unstructured. At the same time, this method sends only asynchronous small messages which can be aggregated to minimize startup overheads, and exhibits low overall communication costs. This makes the software suitable to exploit concurrency at the level of an SMP node and a cluster of nodes. In addition, the single-node performance is comparable to that of the fastest to our knowledge sequential guaranteed quality Delaunay meshing library (Triangle). Our experimental results show very good (sub-linear) scalability on traditional parallel architectures.

B. Related Work

Distributed shared memory systems such as C Region Library (CRL) [12] provide region-based shared address space programming model on distributed architectures. CRL hides message passing and instead achieves parallelism through accesses to shared regions of virtual memory. In [13], [14] the authors propose to define a common data model and data-structure neutral interfaces for mesh generation and adaptive mesh refinement among other services for scientific applications on future petascale computer architectures. Zoltan [15] provides graph-based partitioning algorithms as well as geometric load balancing algorithms. Zoltan requires synchronization during load balancing and behaves similarly to other stop-and-repartition libraries we reviewed [16]. Charm++ [17] is a parallel dialect of C++ and an adaptive runtime system which provides load balancing, fault tolerance and automatic checkpointing. Parallel programming languages (e.g., Chapel [18], Co-array Fortran [19], X10 [20]) try to improve the programmability of parallel computers by supporting partitioned global address space and abstractions for various forms of parallelism. None of the above provide explicit support for out-of-core computing. In [21] authors proposed to use parallel octrees and space-filling curves to generate and adapt massive octree meshes. They implemented a parallel octree meshing tool and used it for terascale finite element simulations. The Adaptive Large-scale Parallel Simulations (ALPS) [22] is a library providing dynamic mesh adaptivity and redistribution. ALPS uses parallel octree-based hexahedral finite element meshes and dynamic load balancing based on space-filling curves. In [23] the authors present a parallel octree-based adaptive mesh finite element library for petascale computing. However, this library targets semi-structured hexahedral finite element meshes, while we generate unstructured (i.e., triangular and tetrahedral) meshes.

II. Multi-layered Run-Time System

A. Requirements

The three PUMG methods we describe in this paper have the following common characteristics:

1) spatial locality — each processing element (PE) works with a subset of mesh elements that cover a certain geometrically defined area, and most of the computation is performed on data that does not have outside dependencies;
2) although the communication patterns vary among the methods, the common property is that the size of the data that the PEs need to exchange is relatively small compared to the sizes of the subdomains;
3) local synchronization — changes in a subdomain usually affect only neighbors of that subdomain, and global synchronization is not required;
4) irregular access pattern — it is not possible to predict the exact mesh elements and memory locations that are accessed;
5) SPMD data model — single program is used to process portions of the dataset in parallel;
6) interoperability — to simplify the porting process we should not obstruct the MPI or any other form of communication used by the rest of the application (i.e., FE solver).

B. Background

We adopt the *mobile object* which is defined in [3] as a location-independent container implemented by the run-time system to store application data. The decision to define mobile objects is left to an application programmer, but it is recommended to be used for representing larger semi-isolated fragments of a dataset (e.g., subdomains). A mobile object can be freely moved by the run-time system between nodes and is globally addressable.

A *message* is an amalgamation of data transfer and a remote procedure call [24]. It is one-sided, that means the receiving node does not have to post an explicit receive and is not interrupted when a message arrives.

A *message handler* is a function defined by an application and registered with a mobile object. A message is delivered to a mobile object by invocation of a corresponding message handler on a node where the mobile object is located. Message handlers, messages and mobile objects allow encapsulation of data represented by mobile objects.

A *mobile pointer* is a global identifier and is used to reference a mobile object. Because a mobile object is not restricted to any specific node a message is addressed to the mobile pointer and the run-time system routes the message appropriately. Order of messages is preserved only between two endpoints.

In the course of out-of-core computing mobile objects can be unloaded to and re-loaded from disk. Mobile objects support *serialization*[1] by implementing serialization interfaces provided by the run-time system.

[1]Serialization is the process of transforming the memory representation of an object to a data format suitable for storage or transmission.

C. Programming Model

The programming model is centered around the mobile object concept. The run-time system is designed for data centric computation where most of communication happens between mobile objects rather than between processors. Parallelism is achieved by executing message handlers simultaneously on multiple nodes and multiple tasks within each message handler. The MRTS tries to achieve maximum utilization by executing as many tasks as available yet not oversubscribing the PEs which can lead to unnecessary context switches and performance degradation.

The usual application for the run-time system has its dataset broken into a collection of mobile objects. We encourage *overdecomposition*, that is the problem is broken into N subproblems and $N \gg P$, where P is the number of PEs. It allows greater flexibility for dynamic load balancing [25] and is even more important for out-of-core computing where the number of objects allowed in memory simultaneously is limited by available physical memory.

At the beginning, an application performs initial preprocessing (if necessary), creates mobile objects, defines serialization interfaces, registers message handlers, distributes the mobile objects between nodes (optional), initiates the parallel phase by posting the initial messages (e.g., main/driver function) and then passes control to the run-time system.

The execution progresses by executing messages handlers, posting messages and dynamically creating new mobile objects. A message is posted to perform an operation on the data of a particular mobile object. Messages can be addressed to local (including self), out-of-core and remote mobile objects. In fact, we strongly recommend to use messages rather than function calls or other means of communication outside of the context of the mobile object. Otherwise, the application is responsible for load balancing and to check and ensure availability of the data it tries to access.

A message addressed to a local mobile object is inserted into its message queue. If the object is local but out-of-core the message is queued and the object is scheduled to be loaded in-core. If the object is remote the message is routed to the corresponding node and processed there. The processing of a message from a remote node is the same as for a local message.

The bulk of parallel computations are performed inside message handlers. When no message handlers are executing and no messages are being delivered the run-time system detects a termination condition. At this point the control is passed back to the application. Usually, at this point the application performs post-processing (if necessary) and terminates. Although, it is possible to start another phase of computing with the run-time system.

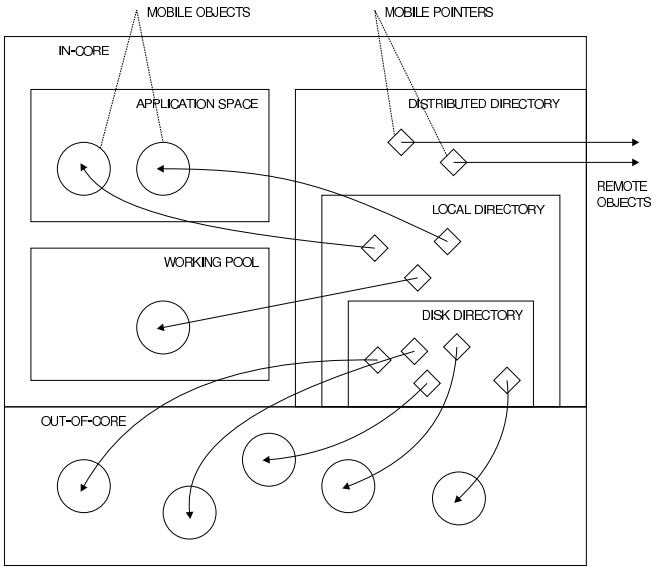

Figure 3. Memory organization and global addressing of the MRTS

D. Organization

The run-time system is organized into layers according to the principle of separation of concerns (see Fig. 3). Parallelism is exploited via multi-threading on a node level and via message passing between nodes. The memory space available to an application consists of local, disk and remote memory. Hence, we call our run-time system the Multi-layered Run-Time System (MRTS). The MRTS is organized into the following layers: the storage layer, the out-of-core layer, the control layer and the computing layer.

The storage layer is used for managing mobile objects stored out-of-core. The underlying storage facility is hidden from the application and can utilize regular files, block devices and databases[2]. Blocking and non-blocking operations for loading and storing a mobile object are provided. This functionality is primarily used by the MRTS internally and is not exposed to an application.

The out-of-core layer is responsible to keep track of mobile objects and control swapping (i.e., make decision when and which objects should be un-/loaded from/to memory. The out-of-core layer also maintains a cache to prefetch mobile objects depending on swapping scheme and input from application.

The control layer is responsible for delivering messages either locally or remotely and controlling migration of objects between nodes. Object location is determined by querying the mobile object distributed directory. Depending on the location of the object the message can be routed to a remote node or queued for local execution. The control layer

[2]The evaluation of different storage subsystems is out of scope of this paper and will be submitted elsewhere. Out-of-core objects are stored in a single large file and meta-data is kept in memory at all times for all experiments presented in this paper.

decides the order in which message queues of local mobile objects are processed. The input from the control layer influences the swapping decisions of the out-of-core layer. In addition, the control layer provides memory management primitives to an application [26].

The computing layer is used to provide uniform interface to various multi-threading technologies employed in the MRTS. We encourage the use of *tasks* – fragments of code that can run in parallel and are expected to complete without blocking. Each message handler function viewed as a task once it is scheduled to be executed and can spawn new tasks during the execution. Unlike messages tasks can only access data of the corresponding mobile object. However, tasks are lightweight and can be used to exploit fine-grain parallelism without much overhead. The computing layer manages the execution of message handlers and tasks, it is responsible for memory allocation, synchronization and load balancing the tasks between PEs (i.e., cores, nodes, racks).

E. Implementation

Software layers The storage layer implements several swapping schemes which are based on popular cache algorithms. In addition to the least recently used (LRU) scheme we implemented the least frequently used (LFU), the most recently used (MRU), the most used (MU) and the least used (LU) schemes. While the LRU scheme enjoys highest performance most of the time, for some applications (e.g., PCDM) the LFU can be up to 7% faster.

A set of swapping thresholds is used to influence as well as to force swapping in extreme cases. The hard swapping threshold is defined to be a multiple of the size of the largest mobile object currently stored on disk. The actual value can be set at the initialization of the MRTS, the default is two. This threshold is checked whenever the application wants to allocate additional memory. If the amount of memory after allocation is less than the threshold unused objects are forcefully unloaded to free memory. The soft swapping threshold is defined as a fraction of the total available memory and is used to influence caching of the out-of-core mobile objects. When the amount of free memory drops below the soft threshold the storage layer is "advised" to start swapping. The soft threshold can be set at the initialization of the MRTS, the default is one half.

Additionally, the out-of-core layer provides an API to assign swapping priorities to mobile objects[3] and directly lock/unlock mobile objects. The locking is straightforward, a locked object cannot be unloaded from memory before it is unlocked. The priorities are used to provide hints to the run-time system regarding the importance of keeping an object "in-core" but still allow it to make final decision.

The control layer uses preemptive communication internally. A preemptive message interrupts whatever is executing and only returns control when it finishes. Executing potentially long running mobile messages can lead to high overheads. Therefore, application messages are queued upon arrival and executed when appropriate. When a message is removed from the queue it is "delivered" by executing its respective message handler. When the message handler terminates the control layer makes a decision whether to continue to process the message queue of the current object or switch to another object or serve systems aspects like information dissemination and/or decision making for load-balancing or swapping. The control layer keeps track of all messages, including the messages of out-of-core mobile objects, and assigns swapping priorities depending on the number of messages and the order they were delivered. Depending on the amount of work (i.e., number of messages) in-core the control layer can "advise" the out-of-core layer to initiate swapping.

Mobile Objects and Threads The mobile object directory that stores mobile pointers is a distributed directory with lazy updates [27], for a mobile object that resides on a remote node its last known location is stored. When a messages is sent to that location it is not guaranteed that the destination mobile object will be there. If not, the message is forwarded to the last known location of the object on that node. When the message finally arrives to the object's current location an update service message is sent back to all nodes through which the message was routed. We experimented with different location management policies and determined that lazy updates provides good compromise between accuracy and update overhead [27].

The computing layer provides a lightweight mostly-wrapper interface to multi-threading libraries. We encourage and support multi-threading within a message handler. Each message handler is a task and can be further broken into child tasks and some of those tasks can be executed in parallel. We utilize two different but similar industrial-strength multi-threading programming technologies (only one can be active): (1) Intel Threading Building Blocks (TBB) [28] is a C++ template library designed to simplify and streamline parallel programming for C++ developers. It provides high-level abstraction, is based on generic programming and is designed to hide low level details of managing threads and supports nested parallelism; (2) Grand Central Dispatch (GCD) [29] is an Apple technology used to optimize application support for systems with multiple and/or multi-core processors. GCD implements task parallelism based on the thread pool pattern. In both cases we use provided functionality to achieve task level parallelism within a message handler, a task can be implemented as a *block* in case of GCD or as a method of the *task* class or a lambda function in case of TBB.

A user defined mobile object must implement initialization, un-/registration and de-/serialization methods. Initialization is performed when the object is first created; the

[3]The swapping priority assigned to a mobile object is stored inside the corresponding mobile pointer data-structure.

object is unregistered when it has to be moved to another node and is registered when it is installed on a new node; the object is de-/serialized when it is transferred from/to disk.

Whenever a mobile object is created a mobile pointer is generated. Each mobile pointer contains either a reference to its object if that object is local and in-core or its location otherwise. Additionally, a mobile pointer of a local mobile object is associated with a queue of messages that were delivered to the mobile object. When an object is loaded in-core the message queue is processed. The size of a message queue influences scheduling and swapping.

Message Passing A message is composed of a destination mobile pointer, a message handler and optional arguments. A message handler is implemented as a function. When it is called it is provided with a reference to the corresponding mobile object (not the mobile pointer) and optional arguments. Messages that are delivered to their destination nodes are stored together with the respective mobile objects. This means that if an object is out-of-core its messages are also stored out-of-core. The number of messages in a message queue is stored in the respective mobile pointer.

To send a message to a mobile object the following should be supplied: a mobile pointer that identifies the destination mobile object, a message handler and optional arguments. In case of a local mobile objects the message is queued in the respective queue. Alternatively, the message is delivered through a one-sided communication mechanism to a last known node where the object might be located. A remote procedure call is performed to both deliver the message as well as to notify remote node of the delivery. We are using the Aggregate Remote Memory Copy Interface (ARMCI) [30] library for such low-level inter-node communications. The ARMCI library is a portable one-sided communication library that can be used in MPI applications and offers an extensive set of functionality in the area of RMA communication: (1) data transfer operations (2) atomic operations (3) memory management and synchronization operations, and (4) locks. Additionally, the ARMCI library is part of the Global Arrays [31] which is popular in scientific computing and widely supported on existing and upcoming supercomputers. In turn, this ensures the MRTS portability.

Object Migration When an object is to be migrated to another node or stored out-of-core it must be appropriately serialized, i.e., packed. Then again, when an object is installed on a node or is loaded in-core it has to be de-serialized, i.e., unpacked. Due to a potentially complex internal structure of a mobile object the serialization operation must be defined by the application. Not all mobile objects designated as out-of-core are actually unloaded to disk, some are cached in memory. To allow a high degree of flexibility for the out-of-core computing we provide several instruments of control. An application can choose not to influence the system altogether, in such case the decision to load/store mobile objects is made based on their access

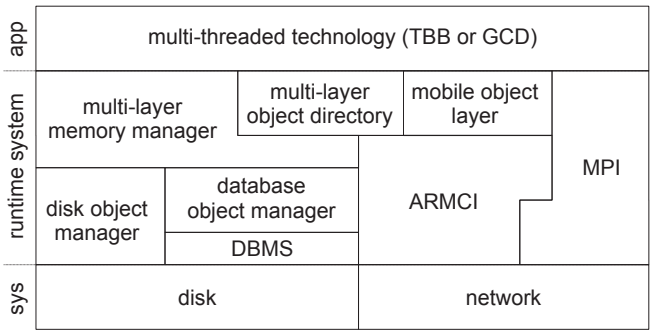

Figure 4. Software organization of the MRTS

pattern (i.e., message pattern). Alternatively, an application can assign priorities which makes high priority objects to be cached more often. Finally, an application can force loading an object as well as locking an object which means the object is loaded or stays in memory regardless of its access pattern and priority respectively. Note, an application should be very careful with locking too many objects since it can result in running out of memory.

Figure 4 shows the software organization of the MRTS.

III. Out-of-core Non-Uniform Parallel Delaunay Refinement

Out-of-core PDR and out-of-core PCDM methods appeared in [1] and [2], respectively. In this section, we describe in more detail the out-of-core NUPDR method and its implementation with MRTS. Its in-core versions appeared in [32] for 2D and in [5] for 3D.

The NUPDR is using a master-worker model. The master starts by constructing a quad-tree which initially contains a single leaf enclosing the entire geometry, and an initial triangulation. Next, a queue of leaves containing poor quality triangles is generated (we will refer to it as refinement queue). At this point the master enters a loop which will only terminate when the refinement queue is empty and no workers are computing. Termination of the loop indicates that the mesh is refined and the algorithm terminates.

Inside the loop, if the refinement queue is not empty and there is an available worker a leaf is removed from the queue, additionally a buffer zone BUF of the leaf (i.e., other neighboring leaves) is also removed from the queue. A leaf is then passed to an available worker for refinement.

If the queue is empty or no workers are available, the master waits for a worker to finish refining. When this happens the leaves that compose the buffer BUF of the refined leaf are checked for poor quality triangles. All leaves that have bad triangles are reinserted into the refinement queue.

Poor quality triangles are stored as several structures based on a ratio between the side length of the enclosing leaf and their circumradius. A worker refines a leaf by processing

poor quality triangle structures in a loop starting with the lowest ratio (largest triangles). In that loop a queue of poor triangles with specific ratio is processed until it is empty.

For each poor triangle, a point is computed using a deterministic function and is inserted into the mesh. Then the mesh is updated which could lead to a propagation of changes into buffer leaves BUF and the creation of poor triangles for the current leaf and for the buffer leaves. As a result, the poor quality triangles are inserted into the corresponding data structures.

When both loops complete, the leaf is recursively split while a relation for constructing the quad-tree holds [5]. The locally refined mesh and quad-tree leaf are returned to the master.

Out-of-core Non-Uniform Parallel Delaunay Refinement The MRTS programming model does not support master-worker pattern directly and as such some restructuring of the algorithm is required. First, for each leaf of the quad-tree we create a mobile object which holds a portion of the mesh that is enclosed by this leaf. The refinement queue is also a mobile object. Additionally, the refinement queue mobile object holds and updates the quad-tree structure internally.

At the start a single thread creates the first top leaf mobile object and generates the initial mesh. In the process of mesh generation the top leaf could be split and in such cases new mobile objects are constructed. Each leaf stores its list of poor quality triangles independently of the rest.

Next, a list of leaves that contain poor triangles is generated. A message designated `update` is sent to the refinement queue mobile object and the control is passed to the MRTS. When the control is returned to the application the mesh is fully refined.

The `update` message takes the following arguments: a list of changes to the quad-tree, which is a list of mobile pointers to the newly created leaves and their relation to the existing leaves; a list of mobile pointers of the leaves with bad triangles.

When an `update` message is received by the refinement queue mobile object, its handler performs the following. The quad-tree and the refinement queue are updated with the new leaves. If the refinement queue is empty (a list of leaves with bad triangles could be empty) the message handler exits. If not, a leaf is removed from the queue, its buffer BUF is computed, and the respective leaves are also removed from the queue. A message designated as `construct buffer` is sent to the leaf and its BUF buffer. The only arguments of the message are the mobile pointer of the leaf and the number of leaves in the buffer.

The message handler of `construct buffer` will do the following depending on the receiver. If the message is received by the leaf object, a counter is created with the number of leaves in the buffer. If the message is received by one of the leaves in the buffer, it sends a message `add to buffer` to the leaf being refined and frees the memory it used for storing its portion of the mesh.

The `add to buffer` message is used to deliver a portion of the mesh to another leaf. When an `add to buffer` message is received by a leaf, the counter of the buffer leaves is decremented and the argument mesh is integrated into the mesh of receiving mobile object. When the counter reaches zero, a message designated as `refine` is sent to the leaf object (i.e., itself). The `refine` message takes no arguments.

The message handler of a `refine` message performs the same step as a worker in the NUPDR algorithm. The only difference is the following. Instead of updating a global list of leaves with poor triangles, a local structure is created and updated through the refinement. After the refinement completes, an `update` message is sent to the refinement queue object. The local list of leaves with poor triangles as well as any changes made to quad-tree are passed as arguments to the `update` message. Then, new mobile objects are created as needed (for every new leaf) and the corresponding portions of the mesh are distributed among them. Finally, the portions of the mesh that correspond to the leaves other than the current leaf are returned to their owners via `recreate` messages.

In the end, when no message handlers are executing and no messages are traveling, we reach the termination condition. At this point the control is returned to the application and the algorithm completes.

Optimization While the algorithm described above works correctly, it is not as efficient as it can be. Following are the number of changes we introduced to considerably improve the performance.

The refinement queue object is relatively small and receives and sends many messages. Therefore, we locked it in memory meaning it will never be unloaded out-of-core.

Since we operate in a shared memory environment, we try to minimize the use of `add to buffer` messages. Instead, we check whether the receiving leaf object is in-core, and in such a case call the message handler directly. When the handler is called directly the sender's mesh fragment is made available to the receiver and does not have to be copied. Consequently, the memory occupied by the mesh fragment is not freed and a `recreate` message is unnecessary.

The leaves that are part of the buffer are locked in memory after they send the `add to buffer` messages or call the respective handlers directly. They do not occupy a significant amount of memory at this point and do not require a `recreate` message anymore. Instead, a `recreate` message handler is called directly and afterwords the objects are unlocked (i.e., can be unloaded from memory).

Instead of sending a `refine` message, we call the message handler directly, thus eliminating the possibility it will be forced out of memory before the message is

delivered.

We change the order of the leaves in the refinement queue based on how many leaves are in their buffers. This way we try to have as many leaves as possible present together in-core and available for refining. We also check which leaves are in-core and try to refine the leaves with the most buffer leaves loaded.

Additional improvements come from managing the priorities of the out-of-core subsystem. When we remove a leaf from the refinement queue we check if it is currently loaded, if it is, we assign it a very high priority to minimize the possibility it will be unloaded before a `construct buffer` message arrives. Also, we assign different priorities to the leaves of the buffers depending on the order they were removed from the refinement queue.

Findings The NUPDR algorithm requires access to several leaves of the quad-tree to refine a single leaf. To accommodate this we either have to collect all leaves in one mobile object dynamically on demand or store a single leaf in each object but then ensure that when the message is delivered all related objects are local and in-core. Since the MRTS discourages direct control over mobile objects we used the first approach. With optimization the ONUPDR using this approach performs similarly to the NUPDR. However, this discovery lead us to believe that the ability to collect several mobile objects during the execution of a mobile message can simplify the development and provide additional space for optimization.

We introduced a multicast mobile message to the MRTS. A multicast mobile message is similar to a mobile message except it can be sent to multiple mobile objects and ensure that specific mobile objects are loaded into memory when the message is delivered. Note, this is still experimental and requires further research and evaluation.

Instead of a destination mobile pointer, a vector of mobile pointers is supplied. Additionally, a counter specifies which objects will receive the message. In the example of the ONUPDR, we would provide a vector containing mobile pointers of a leaf and its buffer as the first argument and 1 as the second argument, meaning the message should be delivered only to the leaf mobile object.

Internally, the MRTS must first collect all mobile objects from the vector on the same node and in-core, and only after that the mobile message is delivered. The message is then delivered to one or more mobile objects in the vector (depending on second argument), order is not important, can be simultaneously.

IV. Performance Evaluation

We conducted our evaluation using resources from (1) SciClone cluster at the College of William and Mary[4] (64 single-cpu Sun Fire V120 servers at 650 MHz with 1 GB memory and 32 dual-cpu Sun Fire 280R servers at 900 MHz with 2 GB memory) and STEMS cluster which is part of Center for Real-time Computing[5] (four, four-way SMP IBM OpenPower 720 compute nodes, with IBM Power5 processors clocked at 1.62 GHz and 8 GB memory).

We start by evaluating the performance of the control layer of the MRTS. We tested small problems sizes on STEMS for all three methods and very large problems were tested on SciClone for in-core methods.

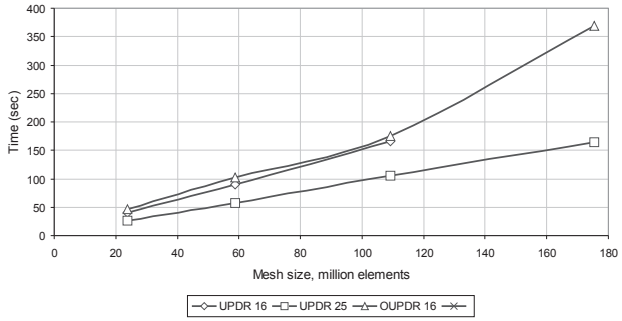

Figure 5. Execution times for UPDR and OUPDR for in-core problem sizes

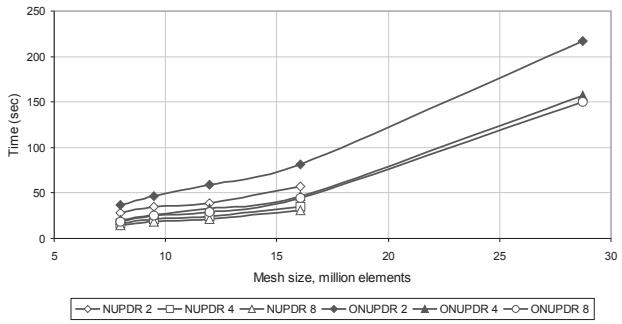

Figure 6. Execution times for NUPDR and ONUPDR for in-core problem sizes

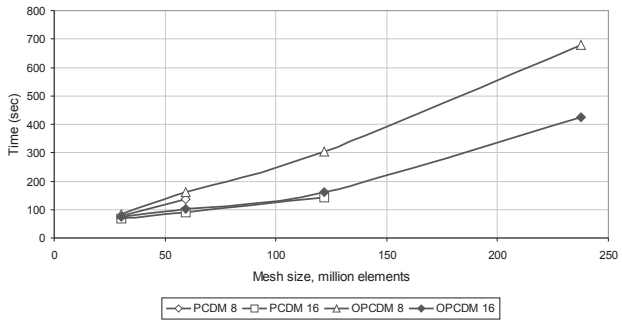

Figure 7. Execution times for PCDM and OPCDM for in-core problem sizes

[4]http://compsci.wm.edu/SciClone

[5]http://crtc.wm.edu

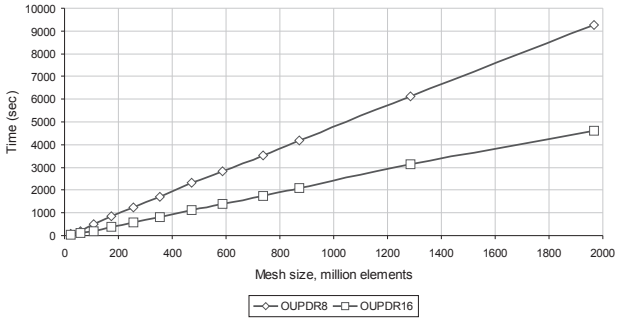

Figure 8. Execution times for OUPDR for out-of-core problem sizes

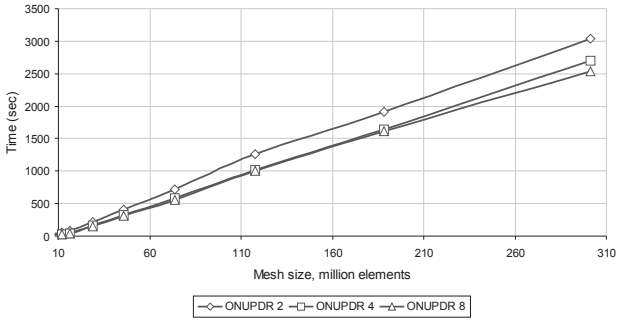

Figure 9. Execution times for ONUPDR for out-of-core problem sizes

Figure 5 shows the execution times of the UPDR (16 and 25 PEs) and the OUPDR (16 PEs). The largest problem size on the chart, 175 million elements is too large for UPDR running on 16 processors. We can see that the performance of the UPDR and the OUPDR is very similar (the OUPDR is up to 12% slower) for in-core problem sizes which means that the overhead introduced by the MRTS is small. Figure 6 shows the execution times of the NUPDR and the ONUPDR for 2, 4, and 8 PEs[6]. For 4 and 8 PEs the overhead can be as high as 18% which is acceptable. For 2 PEs case the ONUPDR is up to 41% slower. This is explained by the fact that the NUPDR uses custom memory allocator that shows much lower overhead than the MRTS memory manager in 2 PEs case. Figure 7 shows the execution times of the PCDM (16 and 25 PEs) and the OPCDM [2] for 8 and 16 processors. As is the case with the UPDR and OUPDR the performance of the OPCDM is very similar to that of the PCDM (up to 13% overhead).

Figures 8, 9 and 10 demonstrate the performance of the out-of-core and storage layers of the MRTS. They show the execution times of the OUPDR (8 and 16 PEs), ONUPDR (2, 4 and 8 PEs) and OPCDM (8 and 16 PEs) for very large problems. These charts demonstrate that the size of very large problems do not degrade the performance of the

[6]The NUPDR and current implementation of the ONUPDR are shared memory applications and as such are restricted to a single node

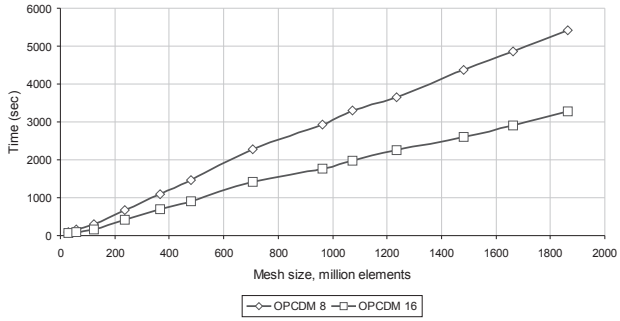

Figure 10. Execution times for OPCDM for out-of-core problem sizes

Table I
SINGLE PE PERFORMANCE OF UPDR AND OUPDR METHODS.

Size $\times 10^6$	PEs	Time (sec)		Speed ($\times 10^3$/sec)	
		UPDR	OUPDR	UPDR	OUPDR
24	4	294	46	20	33
59	9	295	102	22	36
109	16	295	176	23	39
175	25	297	368	24	30
255	36	293	576	24	28
353	49	295	802	24	27
471	64	300	1133	25	26
588	81	296	1386	24	27
739	100	300	1745	25	26
874	121	294	2111	25	26
1284	n/a	n/a	3122	0	26
1967	n/a	n/a	4599	0	27

Table II
SINGLE PE PERFORMANCE OF NUPDR AND ONUPDR METHODS.

Size, $\times 10^6$	Time (sec)		Speed ($\times 10^3$/sec)	
	NUPDR	ONUPDR	NUPDR	ONUPDR
8	17	20	119	100
9	21	27	114	89
12	24	33	124	90
16	35	46	115	86
29	n/a	157	n/a	46
46	n/a	322	n/a	36
74	n/a	589	n/a	31
118	n/a	1016	n/a	29
188	n/a	1638	n/a	29
301	n/a	2702	n/a	28

methods (time increases almost linearly) on MRTS.

Tables I, II and III reflect the performance of the out-of-core layer as well as the performance of the control layer. We are interested in the performance of a single PE and use $Speed$ metric which is computed as $Speed = \frac{S}{T \times N}$, where S is the size of the problem (i.e., number of mesh elements), T is the total execution time and N is the number of PEs. Note, the execution time of the original application is from older SciClone cluster since they need the aggregate memory of over a hundred processors. The MRTS applications run on the newer faster STEMS cluster and have faster per PE

Table III
SINGLE PE PERFORMANCE OF PCDM AND OPCDM METHODS.

Size $\times 10^6$	PEs	Time (sec)		Speed ($\times 10^3$/sec)	
		PCDM	OPCDM	PCDM	OPCDM
30	4	308	73	24	26
59	8	296	101	25	37
122	16	319	163	24	47
238	32	310	425	24	35
366	48	327	707	23	32
480	64	304	918	25	33
706	96	324	1408	23	31
963	128	299	1772	25	34
1074	n/a	n/a	1986	n/a	34
1235	n/a	n/a	2256	n/a	34
1480	n/a	n/a	2614	n/a	35
1662	n/a	n/a	2900	n/a	36
1864	n/a	n/a	3285	n/a	35

Table IV
OVERLAP OF COMPUTATION, COMMUNICATION AND OUT-OF-CORE DISK IO IN THE OUPDR.

Size $\times 10^6$	Time (sec)	Comp (%)	Comm (%)	Disk (%)	Overlap (%)		
					min	max	avg
24	46	88	18	0	1	7	6
59	102	85	16	0	0	2	1
109	176	86	21	0	2	8	7
175	368	65	15	36	4	19	16
255	576	61	12	51	8	29	24
353	802	58	11	61	6	35	30
471	1133	57	13	64	11	38	33
588	1386	55	13	70	5	46	38
739	1745	54	14	73	5	48	41
874	2111	51	18	73	6	54	42
1284	3122	52	18	76	5	57	46
1967	4599	53	16	82	20	63	50

speed in most cases. Rather than compare the actual speeds in those tables we want to see the trend as we increase the problem size. We can see that the original applications as well as the MRTS implementations seem to maintain more or less constant speed. This means that as we increase the problem size the MRTS is able to sustain the performance level. Additionally, for the original applications this means they scale rather well [5]–[7], [11].

Tables IV, V and VI are presented to demonstrate the out-of-core performance of the MRTS applications. These tables show computation, communication (or synchronization for ONUPDR) and disk I/O as a percentage of total execution time. The last three columns show overlap of computation, communication/synchronization and disk I/O which we compute as $Overlap = \frac{Comp+Comm+Disk-Total}{Total} \times 100\%$, where $Comp$ is the computation time, $Comm$ is the communication/synchronization time, $Disk$ is the disk I/O time and $Total$ is the total execution time. MRTS is designed to promote overlapping of communication and I/O and our data show we have been very successful at it. The overlap is over 50% for large problems and can be as high as 62%. This means the MRTS is capable of tolerating high latencies

Table V
OVERLAP OF COMPUTATION, SYNCHRONIZATION AND OUT-OF-CORE DISK IO IN THE ONUPDR.

Size $\times 10^6$	Time (sec)	Comp avg (%)	Sync avg (%)	Disk avg (%)	Overlap (%)		
					min	max	avg
8	20	98	2	0	0	0	0
9	27	99	1	0	0	0	0
12	33	98	2	0	0	0	0
16	46	98	2	0	0	0	0
29	157	51	1	81	5	38	33
46	322	40	1	103	7	52	43
74	589	36	1	112	7	56	48
118	1016	35	1	116	17	58	52
188	1638	32	1	123	18	64	56
301	2702	33	0	124	17	64	58

Table VI
OVERLAP OF COMPUTATION, COMMUNICATION AND OUT-OF-CORE DISK IO IN THE OPCDM.

Size $\times 10^6$	Time (sec)	Comp avg (%)	Comm avg (%)	Disk avg (%)	Overlap (%)		
					min	max	avg
30	73	49	53	0	0	2	2
59	101	64	36	0	0	0	0
122	163	94	12	0	2	7	5
238	425	66	7	50	4	27	23
366	707	62	5	64	8	36	30
480	918	60	4	72	6	43	36
706	1408	61	3	76	10	50	40
963	1772	57	3	87	6	56	47
1074	1986	58	3	88	8	63	49
1235	2256	59	3	91	9	65	53
1480	2614	58	3	95	14	67	57
1662	2900	59	4	98	10	73	60
1864	3285	60	4	97	7	74	62

Table VII
THE COMPARISON OF PERFORMANCE OF THE COMPUTING LAYER IMPLEMENTATIONS.

Size, $\times 10^6$	Threading Building Blocks			Grand Central Dispatch		
	T1(sec)	T4(sec)	Spdup	T1(sec)	T4(sec)	Spdup
7.97	49.20	24.94	1.97	46.29	27.54	1.68
9.49	60.98	31.88	1.91	61.89	34.05	1.82
11.98	70.38	32.93	2.14	71.17	37.84	1.88
16.04	114.59	56.66	2.02	115.31	60.11	1.92

rather well and accommodate data-intensive application.

The MRTS can use and supports either GCD or TBB multi-threading libraries to utilize shared-memory computing. Since GCD availability on non-Apple systems is very limited yet we had to use an older system running an experimental version of FreeBSD: Dell PowerEdge 6600 with 4 Intel Xeon MP 1.47 GHz processor and 16 GB of memory.

Table VII shows sequential time (T1), parallel time with 4 PEs (T4) and relative speedup (Spdup) for the ONUPDR with TBB and GCD implementations of the computing layer. Size is the number of elements in the resulting mesh, a pipe cross-section geometry was used for all experiments. The

speedup is comparable to the speedup of the NUPDR, We can see that GCD implementation is slightly slower yet we can see similar trends for both implementations.

V. Conclusion

We presented the Multi-layered Run-Time System, a practical parallel out-of-core runtime system designed for effective utilization of computing resources without sacrificing performance. We used traditional CoWs and out-of-core computing paradigm to perform an evaluation of our implementation using three parallel unstructured mesh generation methods with a wide spectrum of memory access patterns and communication/synchronization requirements to stress test the MRTS. In particular, the NUPDR was used to test multi-threaded performance, the UPDR was used to test structured communication with some synchronization and the PCDM was used to test fully asynchronous communication. Furthermore, each application tested the out-of-core subsystem.

The MRTS allows for shared memory, distributed memory and out-of-core computing to enable effective computing on a wide range of systems with varying capabilities and potentially leverage memory and network hierarchies of emerging supercomputers. It is implemented on top of established software libraries and standards like TBB/GCD for multi-threading and ARMCI/MPI for both one- and two-sided message passing. This permits incremental application development for multi-layered parallel architectures. Moreover, it allows for an evolutionary approach to application migration of complex applications like parallel mesh generation from traditional parallel platforms to emerging massively parallel platforms.

The task of porting parallel unstructured mesh generation codes onto MRTS is relatively straightforward. Generally, little effort is required to port an application that utilizes a similar programming model. Extra effort is required to optimize the application to take full advantage of the functionality provided by the MRTS but it is far less work than implementing and optimizing an out-of-core application from scratch.

MRTS provides global address space, data mobility and active messages parallel programming models. Additionally, check and restore functionality for fault tolerance can be implemented with little effort on top of the out-of-core subsystem which is important for large scale applications. Moreover, not all applications with access to large supercomputers have enough parallelism to exploit. At the same time many of those applications require very large memory. The MRTS can be modified to use the memory of remote nodes as out-of-core media [33]. This would allow such applications to utilize large memory without major changes to the algorithm.

Contribution The runtime system we presented supports computation of large problems with limited hardware resources as well as effective utilization of those resources. The runtime system introduces small overhead (up to 18% on most configurations) and achieves high overlap of communication and disk I/O (up to 62%). As such it does not sacrifice performance and is capable of tolerating high communication and I/O latencies. The API and and the programming model of the runtime system make the task of transforming an in-core application into an out-of-core one much simpler than developing it from scratch. Design and implementation of ONUPDR which is based on state of the art in-core NUPDR is an example of such transformation.

Acknowledgment

This work was performed in part using computational facilities at the College of William and Mary which were enabled by grants from Sun Microsystems, the National Science Foundation, and Virginia's Commonwealth Technology Research Fund. The first author wants to thank Old Dominion University and Computer Science Department for their hospitality.

References

[1] A. Kot, A. Chernikov, and N. Chrisochoides, "Effective out-of-core parallel delaunay mesh refinement using off-the-shelf software," in *20th IEEE International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, April 2006.

[2] ——, "Parallel out-of-core delaunay refinement," in *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, September 2005, p. 183.

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

[4] D. Nave, N. Chrisochoides, and L. P. Chew, "Guaranteed–quality parallel Delaunay refinement for restricted polyhedral domains," *Computational Geometry: Theory and Applications*, vol. 28, pp. 191–215, 2004.

[5] A. N. Chernikov and N. P. Chrisochoides, "Three-dimensional Delaunay refinement for multi-core processors," in *Proceedings of the 22nd Annual International Conference on Supercomputing*. Island of Kos, Greece: ACM Press, 2008, pp. 214–224.

[6] ——, "Algorithm 872: Parallel 2D constrained Delaunay mesh generation," *ACM Transactions on Mathematical Software*, vol. 34, no. 1, pp. 1–20, Jan. 2008.

[7] ——, "Parallel guaranteed quality Delaunay uniform mesh refinement," *SIAM Journal on Scientific Computing*, vol. 28, pp. 1907–1926, 2006.

[8] L. Linardakis and N. Chrisochoides, "Graded Delaunay decoupling method for parallel guaranteed quality planar mesh generation," *SIAM Journal on Scientific Computing*, vol. 30, no. 4, pp. 1875–1891, Mar. 2008.

[9] D. Nave, N. Chrisochoides, and L. P. Chew, "Guaranteed–quality parallel Delaunay refinement for restricted polyhedral domains," in *Proceedings of the 18th ACM Symposium on Computational Geometry*, Barcelona, Spain, 2002, pp. 135–144.

[10] A. N. Chernikov and N. P. Chrisochoides, "Generalized Delaunay mesh refinement: From scalar to parallel," in *Proceedings of the 15th International Meshing Roundtable*. Birmingham, AL: Springer, Sep. 2006, pp. 563–580.

[11] ——, "Practical and efficient point insertion scheduling method for parallel guaranteed quality Delaunay refinement," in *Proceedings of the 18th Annual International Conference on Supercomputing*. Malo, France: ACM Press, 2004, pp. 48–57.

[12] K. Johnson, M. Kaashoek, and D. Wallach, "CRL: High-performance all-software distributed shared memory," in *15th Symp. on OS Prin. (COSP15)*, December 1995, pp. 213–228.

[13] L. Diachin, A. Bauer, B. Fix, J. Kraftcheck, K. Jansen, X. Luo, M. Miller, C. Ollivier-Gooch, M. Shephard, T. Tautges, and H. Trease, "Interoperable mesh and geometry tools for advanced petascale simulations," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012015, 2007.

[14] A. To, W. Liu, G. Olson, T. Belytschko, W. Chen, M. Shephard, Y. Chung, R. Ghanem, P. Voorhees, D. Seidman, C. Wolverton, J. Chen, B. Moran, A. Freeman, R. Tian, X. Luo, E. Lautenschlager, and A. Challoner, "Materials integrity in microsystems: a framework for a petascale predictive-science-based multiscale modeling and simulation system," *Computational Mechanics*, vol. 42, pp. 485–510, 2008.

[15] K. Devine, B. Hendrickson, E. Boman, M. S. John, and C. Vaughan, "Design of dynamic load-balancing tools for parallel applications," in *Proc. of the Int. Conf. on Supercomputing*, Santa Fe, May 2000.

[16] K. Barker and N. Chrisochoides, "An evalaution of a framework for the dynamic load balancing of highly adaptive and irregular applications," in *Supercomputing Conference*. ACM, Nov. 2003.

[17] L. Kalé and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proceedings of OOPSLA '93*, 1993, pp. 91–108. [Online]. Available: citeseer.nj.nec.com/95307.html

[18] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, vol. 21, pp. 291–312, 2007.

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

[20] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538.

[21] T. Tu, D. R. O'Hallaron, and O. Ghattas, "Scalable parallel octree meshing for terascale applications," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Seattle, WA: IEEE Computer Society, 2005.

[22] C. Burstedde, O. Ghattas, G. Stadler, T. Tu, and L. C. Wilcox, "Towards adaptive mesh PDE simulations on petascale computers," in *Proceedings of Teragrid*, 2008.

[23] C. Burstedde, O. Ghattas, M. Gurnis, G. Stadler, E. Tan, T. Tu, L. C. Wilcox, and S. Zhong, "Scalable adaptive mantle convection simulation on petascale supercomputers," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–15.

[24] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active messages: A mechanism for integrated communication and computation," in *Proceedings of the 19th Int. Symp. on Comp. Arch.* ACM Press, May 1992, pp. 256–266.

[25] K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali, "A load balancing framework for adaptive and asynchronous applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 2, pp. 183–192, Feb. 2004.

[26] C. D. Antonopoulos, F. Blagojevic, A. N. Chernikov, N. P. Chrisochoides, and D. S. Nikolopoulos, "A multigrain Delaunay mesh generation method for multicore SMT-based architectures," *Journal on Parallel and Distributed Computing*, vol. 69, pp. 589–600, 2009.

[27] A. Fedorov and N. Chrisochoides, "Location management in object-based distributed computing," in *Proceedings of Cluster*, San Diego, California, 2004.

[28] Intel Corporation. (2008) Intel(R) Threading Building Blocks Reference Manual. [Online]. Available: software.intel.com

[29] Apple Inc. (2009) Grand Central Dispatch (GCD) Reference. [Online]. Available: developer.apple.com

[30] J. Nieplocha, J. Nieplocha, M. Krishnan, and M. Krishnan, "High performance remote memory access comunications: The armci approach," *International Journal of High Performance Computing and Applications*, vol. 20, p. 2006, 2005.

[31] J. Nieplocha, B. Palmer, M. Krishnan, H. Trease, and E. Apr, "Advances, applications and performance of the global arrays shared memory programming toolkit," *Intern. J. High Perf. Comp. Applications*, vol. 20, 2005.

[32] A. N. Chernikov and N. P. Chrisochoides, "Parallel 2D graded guaranteed quality Delaunay mesh refinement," in *Proceedings of the 14th International Meshing Roundtable*. San Diego, CA: Springer, Sep. 2005, pp. 505–517.

[33] A. Kot and N. Chrisochoides, ""Green" multi-layered "smart" memory management system," in *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, September 2003, p. 372.