Data Migration Substrate for the Load Balancing of Parallel Adaptive Unstructured Mesh Computations*

Nikos Chrisochoides,†Démian Nave,‡and Chris Hawblitzel§

Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556

Computer Science
Cornell University
Ithaca, NY 14853-3801

Technical Report TR-98-16
March 31, 1998

Abstract

Providing adequate software support for unstructured grid generation codes is challenging even for sequential implementations. In parallel implementations, program complexity increases by an order of magnitude due to dynamic, data-dependent, irregular computation and communication requirements. Even more challenging is the implementation of parallel methods that maintain the same quality guarantees as their sequential counterparts.

In this paper we present a data migration run-time system, the Mobile Object Layer, for implementing load balancing heuristics for parallel adaptive unstructured mesh computations. We also argue that the Mobile Object Layer (MOL) simplifies this formidable task. We present preliminary performance data from a parallel mesh generator, based on the Constrained Delaunay Triangulation (CDT), which suggest that the flexibility and general nature of our data-migration approach does not cause undue overhead.

---

†This work supported by NSF grant #9726388 and JPL award #961097
‡Supported by NSF grant #9726388
§Chris Hawblitzel was supported by an NSF fellowship for a portion of this work.
1 Introduction

High performance dominates the list of requirements in the development of large-scale scientific applications, but it is hard to achieve due to the increased complexity in implementing efficient parallel programs. This complexity arises in the context of three issues: data locality, communication cost, and load balancing. We introduce the Mobile Object Layer, a library which alleviates many of the difficulties of implementing solutions to the dynamic load balancing problem, while minimizing the overhead of maintaining data locality and efficient communication. The MOL is designed to be a lightweight and portable package for handling message forwarding between processors in the presence of data (object) or thread migration.

Much work has been done in the development of heuristics for load balancing parallel computations. All of these methods can be classified by two different data distribution schemes: static (ab initio) [1,2,3] distribution, and dynamic (run–time) [4,5,6,7] distribution. In this paper, we do not introduce new algorithms or heuristics for solving the dynamic load balancing problem. Instead, we present a software system for helping the application programmer in the following two tasks: (i) the movement of data as dictated by dynamic load balancing algorithms, and (ii) the automatic forwarding of messages to maintain distributed data structures after data migration.

The Need for the Mobile Object Layer. Although many distributed shared memory (DSM) systems [8,9,10,11] would be ideal platforms for the development of large-scale unstructured and adaptive applications, by and large, computational scientists and application writers choose message-passing libraries, like MPI, which do not provide any tools to support the migration of data or computation. Without these tools, the application programmer is forced to re-implement functionality similar to software systems that support shared address space over distributed memory machines. This is not a trivial task, and the result could be an inefficient implementation of this DSM functionality. Inefficiency could arise, for example, because the programmer chooses to maintain global information by using expensive update strategies, such as all-to-all communication or “eager” updates, which increase traffic in the network and reduce the network bandwidth available to the application.

There are several reasons why application programmers choose message–passing libraries over more sophisticated systems. The most apparent reasons are the lack of standards, the short life-span (i.e. diminishing support over time), and the complexity of current DSM software—problems which could lead to non-portable and high-maintenance applications on the next generation of machines. Existing DSM systems often require special hardware, operating system, or compiler support to enable or enhance their usefulness in a distributed application (CRL [11] is a notable exception). In contrast, the MOL has no such requirements; it is designed to be portable and lightweight, using only minimal resources to effect the functionality needed to enable a more dynamic style of programming.
2 The Mobile Object Layer

The MOL provides the tools to build migratable, distributed data structures consisting of mobile objects linked via mobile pointers to these objects. For example, a distributed graph could be constructed using a mobile object for each node, and, in each node, a list of mobile pointers to adjacent nodes. The MOL guarantees [12] that, if a node in such a structure were moved from one processor to another, any messages directed to the migrated node would be received by forwarding the messages from the sending processors to the node’s new location. This is accomplished by using directories to contain the locations of mobile objects; each time a mobile pointer is accessed, a lookup in a directory is performed to obtain the location of the corresponding object.

The MOL takes advantage of a decentralized directory structure, which allows fast local access to an object’s location. Each processor is assigned its own directory; this scheme reduces network traffic when a mobile pointer is accessed (as compared to a central directory method), but it introduces the problem of keeping directories consistent when objects are moved. The solution implemented in the MOL allows some directories to be out of date by updating “lazily,” as opposed to the expensive and non-scalable solution of broadcasting updates to all processors.

When a processor sends a message to a mobile object, it is sent to the location found in the local directory. If the object is not there, the MOL automatically forwards the message towards the real location, and notifies the processor that sent the message. In this way, a processor’s directory gets updated only when it explicitly shows interest in an object, and a message to that object “misses” the correct location.

The MOL directory structure and forwarding protocol are similar to those used in both ABC++ [9] and Amber [10]. In particular, the MOL uses sequence numbers to distinguish new updates from old in essentially the same way as that described by Fowler [13]. The forwarding approach is similar in spirit to cache-coherency protocols developed for distributed shared memory machines (e.g., FLASH [14]), which also maintain local directories without broadcasts. The MOL method is somewhat more aggressive due to forwarding; a message can be sent to an object without up-to-date information on the location of the object.

The MOL provides mechanisms to support mobile objects, but there are no policies specifying how the mobile objects must be used. It is up to the application (or application-specific library) to decide when and where to move an object. This allows the MOL to support a large number of systems, since no single policy could efficiently satisfy the needs of a broad range of irregular, parallel applications. Also, the MOL provides a simple, low-overhead interface so that application-specific libraries and high-level languages can be efficiently layered on top of the MOL. Finally, the MOL augments a low-level messaging layer without obscuring access to it; the application or library writer still has complete and direct access to these layers, as well as to the MOL. This is essential if the application is to obtain maximum performance. The current version of the MOL has been built and tested over Active Messages [15] and NEXUS [17].

In addition, the MOL supports both threaded and non-threaded models of execution; the threaded model is useful for tolerating communication latency—a thread that needs to wait for
an incoming message can yield to another thread instead of busy-waiting. Although the MOL is designed for use with non-preemptive threads packages in order to avoid the need for locks when updating directories, augmenting the implementation to use locks for preemptive threading would not be difficult.

3 A Parallel Unstructured Mesh Generator

The efficient implementation of an unstructured mesh generator on distributed memory multiprocessors requires the maintenance of complex and dynamic distributed data structures for tolerating latency, minimizing communication and balancing processor workload. We describe an implementation of the parallel Constrained Delaunay Triangulation (CDT) which uses the Mobile Object Layer to simplify data migration for load balancing the computation. We have chosen a primitive dynamic load balancing method to demonstrate both the functionality of the MOL and its usefulness as the bookkeeper for message-passing when data is migrated by the load balancing module.

The Constrained Delaunay Triangulation Method. The 2D mesh generator uses a Constrained Delaunay Triangulation method [18] to generate a guaranteed-quality [19,20] unstructured mesh. Given a precomputed domain decomposition, each subdomain (or region) is assumed to be independent of the others, except for at the interfaces between the regions. In two dimensions, the interfaces and boundaries of the domain are defined by “constrained” edges, which are split if they don’t satisfy certain criteria. Refinement of the mesh in one region does not affect another region, except for when constrained interface edges are split. When an interface edge is split, the change is sent to the processor containing the region which shares the split edge. The target processor updates the region as if it had split the edge itself, and continues processing. This particular method is simple to implement, and produces meshes close to the quality of a normal Delaunay triangulation.

The input to the parallel mesher is a decomposition of a domain into some number of regions, which are then assigned to processors in a way that maximizes data locality. Each processor is responsible for managing multiple regions, since, in general, there will be many regions in the domain of the computation. Subsequently, two types of imbalance arise due to both the decomposition of the domain and the nature of the computation:

- regions are assigned unequally by the initial decomposition, and
- processors may be assigned regions which require significantly more computation than other regions, such as regions where a high degree of accuracy is needed (e.g. in crack propagation simulations).

A Work–Stealing Load Balancing Method. We demonstrate the MOL’s functionality and ease of use with a simple and popular work–stealing load balancing strategy. In this model, a processor is responsible for requesting work; this is in contrast to work–sharing strategies (like diffusion [21]), or centralized methods.
The method we have chosen to implement maintains a counter of the number of work-units that are currently waiting to be processed, and consults a threshold of work to determine when work should be requested from other processors. When the number of work-units falls below the threshold, a processor requests as much work as is available to maintain its level of work.

A Dynamic Load Balancer Using the MOL. In the 2D mesher, the regions of the decomposition are the work-units to be completed by the processors. Furthermore, these regions are viewed as objects that can be migrated when a processor requests more work. Consequently, a work-stealing dynamic load balancer can be designed to use the topology of the regions [4]:

1. A region $r_i$ is adjacent to another region $r_j$ (and vice-versa) if there is a non-empty set of edges, $E$, such that $e \in r_i \cap r_j, \forall e \in E$.
2. Regions are either internal or interface regions; only interface regions are allowed to migrate.
3. Interface regions are defined as regions which are adjacent to one or more regions on other processors.
4. Internal regions are defined as regions which are adjacent only to regions on the same processor.

With this formulation, we can take advantage of the MOL, since each region is a mobile object; when there is an imbalance, interface regions can be migrated to rebalance the computation. When misdirected messages are sent to a migrated region, the MOL transparently forwards the messages and updates the directories of the requesting processors. The load balancing module can therefore migrate data without disrupting the message-passing in the computation.

Data Movement Using the MOL. The critical steps in the load balancing phase, and the events for which the MOL was designed, are the movement of objects to other processors, and the corresponding updates for the communication to the objects. In the case of the mesher, the object to be migrated is a region. To move a region, a programmer-supplied procedure is needed to pack the region’s data into a buffer. Then, a message-passing primitive is invoked to transport the buffer, and another user-supplied procedure handles the unpacking and rebuilding of the region in the new processor.

Before moving the region, the MOL requires that `mob_uninstallObj` be called to properly update the processor’s local directory to reflect the change in the region’s location. Both the region’s mobile pointer and the 4-byte MoveInfo structure returned by `mob_uninstallObj` must be sent with the packed data, since they are used to track the region’s migration. After the region has been unpacked, `mob_installObj` must be called to update the new processor’s directory. The region’s mobile pointer and the MoveInfo structure are used by `mob_installObj` to ensure that the region will receive messages forwarded from other processors, and to update these processors so that future messages are sent directly to the region.

It is important to note that, since the MOL is used in moving data, standard message-passing primitives, like `mpi_send`, cannot be used to communicate with migrated objects. Therefore, the MOL supplies a procedure, `mob_message`, which uses an object’s mobile pointer to send messages to
the last known location of the object. If needed, the MOL will automatically take care of forwarding messages and updating processors’ directories so that subsequent messages find the object without forwarding (unless the object is migrated again).

The MOL only requires the invocation of two procedures, `mob_uninstallObj`, which returns a MoveInfo structure that must be sent with a migrating object, and `mob_installObj`, which uses the mobile pointer and the MoveInfo to initialize the directory for the new processor. Regardless of which DSM system is used, it is up to the application to pack and unpack the object’s data; the MOL, however, provides the mechanisms to automatically and transparently update the application’s communication with a migrated object. As a result, the programmer is free to put more effort into the computational side of the application, as opposed to spending time implementing ways to update processors and to ensure messages reach migrated objects.

4 Performance Analysis

We present results for `mob_message`, which allows messages to be sent to a mobile object via a mobile pointer, and for `mob_request`, which directs messages of 1024 bytes or less (a parameterized value) to specific processors without explicitly requesting storage space on the target processor. In addition, we present data gathered from the parallel meshing application for both non-load balanced and load balanced runs at different percentages of imbalance in the computation.

All measurements for `mob_message` and `mob_request` were taken on an IBM RISC System/6000 SP, using the Active Messages implementation developed in [16]. The benchmarks measured the per-hop latency of messages ranging from 8 to 1024 bytes, as compared to the equivalent `am_store` calls. The performance is very reasonable; the latency of `mob_request` is within about 11% of the latency of `am_store`, while `mob_message`’s latency is about 12% to 14% higher than `am_store`’s latency.

![Graph 1a) Forwarding overhead](image1.png)

![Graph 1b) Handler overhead](image2.png)
To illustrate the importance of the MOL’s updates, Figure (1a) shows the latency of messages that were forwarded once each time they were sent versus messages that were not forwarded. Not surprisingly, the latency of the forwarded messages was about twice as high as that of the unforwarded messages. In a real application, the overall (amortized) cost of forwarding is determined by how often an object moves versus how often messages are sent to the object, since messages are forwarded immediately after an object moves but not after the updates have been received.

Figure (1b) shows the performance of the MOL’s three different types of handlers. A function handler executes immediately, and is restricted in the same fashion as an AM handler. Delayed handlers are executed from an internal queue, and are thus not restricted, except that context switching is not allowed. Threaded handlers are the least restricted, and the most flexible, class of handlers, since they are executed in a separate thread. As expected, the function handler is the fastest, while the delayed and threaded handlers are slightly slower. Since delayed handlers permit communication from within the handler, and threaded handlers permit both communication and context switching, the overheads from the more powerful handlers are fairly low relative to the functionality they add.

The next set of graphs represents data pertinent to a parallel mesh with between 100,000 and 170,000 elements, and for load imbalances of between 8 and 50 percent. Each processor in the system started with 16 regions, for a total of 64 regions on 4 processors. All measurements were taken on an IBM RISC System/6000 SP, using a NEXUS implementation of the MOL. Note that the coarse–grain decomposition leads to some imbalance in the final results for both the non–load balanced and the load balanced experiments.

The graph on the left represents the minimum and maximum computation times on the four processors in the non–load balanced experiments. Given above each bar is the number of elements

![Graph 1a](image1)

**Computation Time vs. Load Imbalance**

- **Minimum**
- **Maximum**

![Graph 1b](image2)

**Maximum Computation Time, Load Balanced Mesh**

- Calculation time
- Edge split initiation time
- Split edge with local refinement time
- Forwarding overhead

2a) Min/max computation time, no load balancing

2b) Maximum computation time, load balancing
generated in the mesh for that particular run. The graph on the right displays the maximum computation time for a series of load-balanced mesh computations which used the same initial mesh as the non-load balanced experiments. Each bar is broken down into the time spent triangulating regions, packing split edge requests, servicing split edge requests, and forwarding messages to migrated regions, in order to show the minimal overhead of using the MOL’s forwarding mechanism. Also, the tuple above each bar gives the number of split edge requests and the number of object migrations from the processor represented by the bar.

5 Summary and Conclusions

We have presented a run-time system, the Mobile Object Layer, for helping the developer in parallelizing adaptive mesh refinement codes. The MOL automatically maintains the validity of global pointers as data migrates from one processor to another, and implements a correct and efficient message forwarding and communication mechanism between the migrating objects. The flexibility of our approach combined with the low overhead make the MOL attractive for application developers, as well as compiler writers.

An important benefit of the Mobile Object Layer is that it allows the fast and efficient testing of many different load balancing strategies, without the bookkeeping hassle that is associated with the maintenance of global data structures. Efficiency of the whole application is also maintained, due to the lightweight, low-overhead nature of the MOL. The disadvantage of the MOL, specifically for Fortran 77 codes, is that the memory management must be rewritten in C (or C++) to take advantage of the facilities of the library.

6 Future Work

The current version of the MOL has been implemented on top of one-sided communication message passing libraries like Active Messages and NEXUS. The next version of the MOL will be interoperable with MPI i.e., parallel programs implemented on MPI will be able to use the MOL library in order to deal with data movement and automatic message forwarding. Also, we are investigating thread-safety issues for preemptive thread scheduling within the MOL.

Other current and future activities include the use of the MOL for the implementation and evaluation of a wide spectrum of well known dynamic load balancing methods for parallel crack propagation and molecular dynamics codes. Finally, with Paul Chew, we are in the process of designing and implementing a parallel 3D mesh generation code on top of the MOL for crack propagation simulations.

7 References


the IBM RISC System/6000 SP, In Proceedings of Supercomputing '96.


