Mobile Object Layer: A Runtime Substrate for Parallel Adaptive and Irregular Computations

Nikos Chrisochoides*,† Kevin Barker,‡ Démian Nave§ Chris Hawblitzel¶

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

Computer Science
Cornell University
Ithaca, NY 14853

Abstract

In this paper we present a parallel runtime substrate, the Mobile Object Layer (MOL), that supports data or object mobility and automatic message forwarding in order to ease the implementation of adaptive and irregular applications on distributed memory machines. The MOL implements a global logical name space for message passing and distributed directories to assist in the translation of logical to physical addresses. Our data show that the latency of the MOL primitives is within 10% to 14% of the the latency of the underlying communication substrate. The MOL is a light-weight, portable library designed to minimize maintenance costs for very large-scale parallel adaptive applications.

Keywords: Parallel, message passing, load balancing, runtime software, adaptive mesh generation.

1 Introduction

The time required to complete computationally intensive simulations, such as crack growth in complex geometries (e.g. gear assemblies and airframes), can have a detrimental impact on the effectiveness of the engineers studying the simulated problem. If, for example, a crack growth problem could be completed in under an hour on a 100 node (or larger) parallel machine, then the effectiveness of the engineers would substantially increase [1]. With current processor and network performance, field simulations of 1 to 10 million degrees of freedom can be realized in less than two hours [2] on massively parallel machines. Unfortunately, this technology is not widely available; however, it is expected that, within several years, the ever-decreasing

*To whom correspondence should be addressed at Computer Science and Engineering, University of Notre Dame: E-mail address: nikos@cse.nd.edu and Fax: (219) 631-9260.

†This work partially supported by NSF CAREER Award grant # CCR98-76179, NSF grant # CCISE-9726388, and JPL award #961097.

‡Partially supported by the Arthur J. Schmidt Fellowship and by NSF grant # CCISE-9726388

§Supported by NSF grant # CCISE-9726388

¶Partially supported by an NSF fellowship.

cost of both symmetric multiprocessor (SMP) machines and low-latency, high-bandwidth interconnects will help alleviate this problem.

While hardware advancements are leading to more powerful computing platforms, the software to utilize these new systems is nearly non-existent. The building blocks needed to implement adaptive and irregular applications are missing, and development of these components takes years. An example of this is the lack of parallel software to efficiently handle the discretization (i.e. mesh generation) of 3-dimensional (3D) complex domains with rapidly changing geometry and/or topology. The complexity of efficient parallel codes, for adaptive applications such as 3D unstructured mesh generation, increases dramatically compared to the corresponding sequential code, due to dynamic, data-dependent, and irregular computation and communcation requirements of the applications. This inherent complexity makes development using existing parallel programming paradigms (e.g. message passing) both time-consuming and error-prone, especially without the aid of parallel languages, software tools, and libraries.

In this paper we present a software system that greatly eases the burden placed on application developers to implement and maintain building blocks for parallel adaptive applications. Parallel software tools in general and our software system specifically should assist application programmers to address the following fundamental issues in parallel computing:

- *Data Locality:* While processors can quickly access data stored in their local memories, the performance penalty for accessing non-local data is too large to be ignored. Furthermore, adaptive, irregular applications are not amenable to compile-time analysis; therefore, a parallel run-time system should be used to exploit data-locality.

- *Communication Overhead:* The necessarily large physical size of massively parallel machines implies large message-passing latencies, the effects of which are exacerbated by increasing processor speeds. Compile time analysis is of little or no help in hiding these latences, because the communication patterns of adaptive, irregular applications are variable and unpredictable. Therefore, new techniques, such as multithreading, are required to hide message-passing latencies at run-time.

- *Load Balancing:* Large-scale, high-performance parallel machines generally consist of many nodes (often SMP nodes), which coordinate in a loosely synchronous fashion. In order to better utilize available resources, it is important to avoid overloading some nodes while leaving others idle. The efficient implementation of adaptive applications which dynamically balance processor loads at run-time requires a great deal of effort above what is required to implment a non-load balancing application. The software support system should therefore facilitate the implementation of dynamic load-balancing strategies.

With these issues in mind, we have developed the Mobile Object Layer, a software system which is designed to simplify the implementation of bulding-blocks for parallel adaptive applications. Our system

is designed to tolerate communication and synchronization costs, and to be flexible enough to allow application developers to easily exploit data locality in their parallel codes. The MOL was built especially to simplify the implementation of dynamic load-balancing libraries, an important component for parallel adaptive applications.

Existing load balancing heuristics are classified into two categories: (A) global (direct) [3, 4] and (B) local (incremental) [5, 8]. Global algorithms are based on grouping mesh components (points or elements) into clusters such that the components within a cluster have a high degree of natural association, while the clusters are "relatively distinct" from each other. Most of these algorithms are very successful in solving the load-balancing problem for static PDE computations. However, these methods are computationally expensive and hard to parallelize—they require a global knowledge of the topology of the mesh (eg. the element-dual graph), and therefore are not suitable for adaptive methods in which the topology and geometry of the mesh can change any time h-refinement is performed. In addition, some of these methods, at least those based on spectral techniques like Recursive Spectral Bisection (RSB), are sensitive to small perturbations in the graph (characteristic of h-refinement methods), and often lead to heavy data migration [10, 11]. Some of these concerns, though, have been addressed [12].

Incremental methods start with an initial partition or distribution of work, and then iteratively minimize the imbalance among the nodes by using profit functions [13] that guide the load balancing process. These methods are not as computationally expensive as global methods, and are easier to parallelize. In the early 90's, it was shown that incremental methods are very successful in load-balancing the computation of parallel adaptive PDE methods [6, 7]. Both global methods, like RSB, and incremental methods, like *Geometry Graph Partitioning* [5] and *PartGraphKway* (PGK) [8], have been studied extensively during the last 10 years. Today, in software packages such as Chaco [4] and Metis [8], one can find very efficient implementations of algorithms for the solution of the graph partitioning problem —the solution of the graph partitioning problem is in the core of traditional load balancing methods for many parallel applications. Table 1 depicts the performance of the global RSB method implemented in Chaco, and the incremental PGK method implemented in Metis. These algorithms are evaluated in terms of the surface to volume ratio, the size of the separators, and the connectivity of the resulting subdomains. Sequential implementations of both global and incremental methods yield partitions with very good balance with respect to the number of elements per subdomain. However, the incremental methods, independent of the implementation, are an order of magnitude faster than the global methods.

In the scientific computing community, incremental methods, like those implemented in Chaco and Metis, are used explicitly; i.e., at certain points of the simulation, load-balancing routines are called in order to equi-distribute processors' work-loads. These points are usually choosen to be placed before computation phases in which load imbalance reduces efficiency (e.g. before a solution step), or after computation phases in which load imbalance is introduced (e.g. after mesh refinement). Although this explicit load-balancing approach is conceptually clean for the parallelization of existing sequential PDE solvers, it has three main

disadvantages:

- It is very difficult to hide the overheads associated with the migration of data and the translation of the mesh data structure into a suitable format for the partitioning libraries like Chaco and Metis (see Table 2).
- Explicit calls to the load-balancing routines require synchronization, since, in general, all processors have to cooperate. In certain cases—for example, load balancing the mesh generation subphase—the explicit approach causes large overheads due to unnecessary synchronization (see Table 2 and Section 5).
- The efficient implementation of adaptive applications is challenging even for sequential machines. In parallel implementations with data re-distribution, program complexity increases by many orders of magnitude due to additional bookkeeping required becuase of the data re-distribution. This bookkeeping is an error prone task that makes maintainability of parallel adaptive codes even more difficult—especially when the codes are optimized to reduce communication overheads.

An alternative to a traditional *explicit* load balancing approach is an *implicit* approach [14], based upon work stealing [15, 17] or work sharing [18] methods, which have been implemented in runtime systems like Cilk [16] and Charm++ [19]. The rest of this paper describes the Mobile Object Layer, a lean, language-independent, and easy to understand, port, and maintain runtime library for the efficient implementation of implicit dynamic load balancing methods for adaptive and irregular applications on distributed memory parallel platforms. The MOL supports a global addressing scheme designed for object mobility, and provides a correct and efficient protocol for message forwarding and communication between migrating objects. We demonstrate the effectiveness and the usability of our system for the implicit dynamic load balancing of a parallel 3-D Delaunay mesh generation code.

In Section 2, we provide background information and related work on message passing and object (data) migration. Next, in Section 3, we describe the MOL, which is the main contribution of this paper. Then, in Section 4, we describe a parallel 3D unstructured mesh generation code, which is a building block for a problem specific environment for 3D fracture mechanics simulations [20]. The performance data in Section 5 suggest that the flexibility and general nature of the MOL's approach for data migration do not cause undue overhead. We conclude with our closing remarks and future work in Sections 6 and 7, respectively.

2 Background and Related Work

The MOL is a *lean, language-independent* and *easy to port and maintain* runtime system for implementing adaptive applications on current and non-traditional parallel platforms [21]. Our design philosophy is based on the principle of *separation of concerns.* Figure 1 depicts the architecture of the overall system and the layers which address the different requirements (concerns) of parallel applications.

In the remainder of this section, we will examine some background and research related to message passing and object migration.

2.1 Background

The Mobile Object Layer is constructed on top of DMCS [22], which provides thread-safe one-sided communication. DMCS implements an API very similar to that proposed by the PORTS consortium [23], and those implemented by Nexus [24] and Tulip [25]. The current version of DMCS is ported on top of MPI [26] and LAPI [27]. The second layer, the MOL, supports a global addressing scheme in the context of object mobility. Other runtime systems developed for parallel, object-oriented programming languages such as Amber [28], COOL [17], and Charm++ [19], and distributed shared memory systems like ABC++ [29], TreadMarks [30], and CRL [31] provide similar functionality.

Although software DSM systems may provide ideal platforms for the development of parallel adaptive applications, by and large, computational scientists and application programmers choose message–passing libraries, such as MPI. The most apparent reasons for this are the lack of standards, the short life-span of software DSM systems due to diminishing support over time, and the complexity of current DSM software — problems which could lead to high-maintenance applications on the next generation of parallel machines. Also, existing DSM systems often require special hardware, operating system, or compiler support to enable or enhance their functionality and performance. In contrast, the MOL has no such requirements; it is designed to be easy to understand, and thus easy to maintain and port on the next generation of parallel machines. The MOL is a lightweight system which supports only the minimal functionality needed to enable a style of parallel programming which is more dynamic than either MPI or vendor-specific communication systems.

The reasons for choosing a software system like MOL that supports a rather familiar programming model (thread-safe message passing), but limited functionality (global name space without remote caching), over complete software DSM systems are: (1) familiarity with the programming model, (2) maintainability of large and expensive applications, (3) portability, and (4) performance. Communication libraries like MPI are alternative choices for developing adaptive applications, but such libraries do not provide the tools necessary for object migration. For example, using plain MPI, application programmers need to implement some of the functionality of software DSM systems explicitly in their codes. This is not a trivial task, and the result could lead into inefficient and sometimes incorrect codes. Inefficiencies could arise, for example, because programmers might choose to maintain global information by using expensive remote "caching" 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.

2.2 Related Work

Languages like Split-C [32] and CC++ [33] have integrated global pointers at the language level and have shown that global pointers can be used successfully to build efficient distributed data structures. However, neither language provides direct support for object migration, nor does either language transparently support automatic updates of global pointers to migrated objects. The MOL, though, is not a high level language like Split-C or CC++, but instead is a runtime library suitable for implementing high-level languages supporting object migration.

The MOL supports *mobile objects*, similar to the *globally addressable objects* of Chaos++ [34]. However, in Chaos++, global objects are owned by a single processor and all other processors with data-dependencies to a global object possess shadow copies. The MOL does not use shadow objects because of the complexity of the code required to maintain consistency between the original and shadow objects. Instead, MOL relies on an efficient message-forwarding mechanism to locate and fetch data from "mobile objects" through "mobile pointers" (see Section 3).

Like ABC++ [29], the MOL implements object migration mechanisms that allow an object to move away from its original home node. The migration mechanisms in ABC++ require communication with the "home-node" each time a message is sent to a mobile object. The MOL eliminates this additional communication by automatically updating its directories to keep track of objects' locations. Furthermore, MOL updates are not broadcast to all processors in the system, but are "lazily" sent to individual processors as needed. The forwarding protocol used to deal with updates correctly and efficiently is nontrivial (see Section 3.3), and goes beyond those proposed in [29].

The MOL, like the C Region Library (CRL) [31], is light-weight software. However, the CRL (like ABC++) implements a shared memory model through accesses to shared "regions" of memory, while the MOL implements explicit message passing, which is a more familiar paradigm to parallel application programmers. Although the MOL requires programmers to send explicit messages to objects, it shares the same minimalist philosophy with the CRL: they both are designed as light-weight, portable libraries, and do not rely on special language, operating system, or hardware support.

In contrast, Emerald [35] and Amber [28] are comprehensive, object-oriented, high-level languages which support object mobility. Both systems use a combination of specialized languages, compilers, and preprocessors in order to make function invocation on objects as transparent as possible. Emerald relies on its specialized language to translate operations on mobile objects into potential remote accesses, and to automatically pack and unpack objects as they move from one node to another. Amber's language, a dialect of C++, is more standard, and relies on virtual memory organization to make object operations transparent. In particular, Amber facilitates object mobility by assigning each object a virtual memory address that is unique across all nodes, which is maintained when an object moves from one node to another.

These systems are designed to make mobile pointers look nearly identical to local pointers, whereas, in

the MOL, mobile pointers are explicit, and cannot be used in the same way that local pointers can. However, the transparency that Emerald and Amber provide must be weighed against the difficulty in implementing this transparency. For instance, both systems migrate stack frames as well as objects; [35] describes a number of complications involved in moving stack frames, such as deciding which frames to move for a given object, and dealing with callee-saves-registers. Solutions to these complications are machine and operating system dependent, which makes porting these systems more difficult.

Finally, hardware systems like FLASH [36] integrate both message passing and global shared memory into a single architecture. The key feature of the FLASH architecture is the MAGIC programmable node controller which connects processor, memory, and network components at each node. MAGIC is an embedded processor which can be programmed to implement both cache coherence and message passing protocols. MOL's forwarding approach is similar in spirit to cache-coherency protocols developed for distributed shared memory machines, like FLASH, which also maintain local directories without broadcasts; because of forwarding, however, the MOL's protocol is somewhat more aggressive. A message can be sent to an object without waiting to find its true location. In addition, hardware supported implementations of cache-coherent protocols multicast explicit "invalidation" messages, while invalidations in MOL are implicit — when an object moves, then directory entries in other processors become invalid, but the invalid entries are detected only when they are used.

3 Mobile Object Layer

The MOL provides the tools to build distributed, mobile data structures. In the MOL, such a data structure consists of a number of mobile objects held together with mobile pointers. For example, a directed graph might be built using one mobile object for each node, where each node holds a list of mobile pointers to other nodes. Once such a data structure is built, the mobile objects can be moved from processor to processor and all of the mobile pointers will remain valid. The MOL keeps track of the locations of objects with directories, so that each use of a mobile pointer requires a lookup in the directory to determine the current location of the mobile object. A simple way to do this might be to have one central directory located on a single processor. However, the performance of this would be unacceptable — each use of a mobile pointer would require communication with the central processor. Instead, the MOL completely decentralizes the directory structure. Each processor has its own quickly accessible local directory, creating the problem of keeping all of the local directories up to date when an object moves. Broadcasting updates to all processors each time an object moves would be too expensive and would not scale to a large number of processors. Instead, the MOL updates directories lazily, allowing some local directories to be out of date.

When a processor sends a message to a mobile object, it sends the message to the (possibly incorrect) location given by its local directory. If the location turns out to be incorrect, the MOL forwards the message towards the real location, and sends an update back to the processor that sent the message. In this (lazy)

way, updates to a processor's local directory occur only when one of the processor's messages "misses" the correct location of the target object. This avoids the need to broadcast updates to all processors each time an object moves.

The MOL provides the mechanisms to support mobile objects and mobile pointers, but it does not specify the policies that say how mobile objects are moved. In particular, it is up to an application (or application-specific library) to decide when and where to move an object. In this way, the MOL can support a large number of systems requiring varying object migration policies, since no single policy could hope to efficiently satisfy the needs of a broad range of irregular, parallel applications. In addition, the MOL provides a simple and 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 such as MPI or Active Messages [37] without obscuring access to it. The current version of the MOL is built over DMCS [22], Active Messages [38], LAPI[27], and NEXUS [24], but the application or library writer still has complete and direct access to the underlying communication substrate. This is essential if the application is to obtain maximal performance.

The MOL currently supports a threaded and a non-threaded model of execution. A threaded model is usefeul because it eases the scheduling of computation — a thread that needs to wait for an incoming message can yield to another thread instead of busy waiting, thus effectively overlapping computation with communication. The MOL does not specify its own thread interface, but it can be easily adapted to work with different thread libraries. For example, we have used an implementation of the MOL in conjunction with a simple implementation of the PORTS thread interface [39] running over an SP-2 Quickthreads implementation [40]. However, the MOL could also be used with pthreads. Because the MOL does not provide a threads interface, mechanisms for locking and scheduling must be provided by the threads packages, and not by the MOL.

3.1 Example Using MOL

This section presents a simple example to illustrate the MOL programming model. It does not discuss the MOL interface in detail; the details of the MOL interface and its implementation are discussed in the following sections.

Figure 2 shows a simple C++ class for a tree data structure, and a single method "setAll" that operates on the tree. A parallel version of the class based on MOL is shown in Figure 3. The local pointers between tree nodes have been replaced with mobile pointers, and the method invocation has been replaced with a mobile object message. The *mob_message* function sends a message to another mobile object that is forwarded until it reaches the object. In this case, the message is sent to the object pointed to by the mobile pointer "children[i]", the argument "f" is passed as data in the message, and the user-specified handler "remoteSetAll" is invoked when the message reaches the object. This example demonstrates that there is a

clear progression from a serial algorithm to an efficient parallel algorithm when using MOL.

As an optimization, the programmer can use the function *mob_deref* to check if an object resides on the local processor before sending a message to it; if the object is local, then the object can be accessed directly without sending a message (see Figure 4). Otherwise, the programmer sends a message to the object on a remote processor.

The power of MOL is that the code shown will work even if the tree nodes are migrating from processor to processor while the call to "setAll" is taking place. A third version of the code (Figure 5) demonstrates the use of the MOL interface for migrating objects. To migrate an object in MOL, the code uninstalls the object from the original processor, sends the object data to another processor (along with a MoveInfo structure that MOL uses to track the object's state), and then installs the object on the new processor. The MOL function *mob_request* is used in this example to transfer the data, but other functions (such as an Active Messages *am_store*) could be used as well. For simplicity, this code allocates each mobile object in a contiguous chunk of memory to make it easy to move the object data, although this is not strictly necessary in MOL. In addition, a "myself" and "moveInfo" field are included in the object itself for convenience, although again MOL does not require this.

These examples show some of the flavor of MOL. The interface is simple but still powerful, and it makes very few restrictions on the application program. In the next few sections, the interface and its implementation are explored in detail, and we demonstrate that MOL is efficient as well as powerful.

3.2 Mobile Pointers and Distributed Directories

The basic building block provided by MOL is the *mobile pointer*. A mobile pointer consists of the number (*id*) of the processor where the object was originally allocated (the "home node") and an index number which is unique on that processor. A (home node, index number) pair forms a name for the mobile object that is unique over the whole system. Mobile pointers are valid on all processors, and they can be passed as data in messages without any extra help from the system. To allocate a new mobile pointer, the user calls *mob_createMobilePointer*, passing in a pointer to the local object that defines the mobile object data (fig. 6). The MOL makes no requirements on the structure or size of a mobile object, so a mobile object may be as simple as a single C structure or C++ object, or it may consist of many structures, objects, and arrays linked together with local pointers.

Each processor maintains a directory that helps to find the location of a mobile object.[1] Each directory entry contains the processor number of the current best guess of the object's location, a sequence number indicating how up to date the guess is, and a pointer containing the address of the object if the object resides on the local processor.

Whenever a processor wants to send a message to a mobile object identified by some mobile pointer, it

[1] Because directories are sparse, we have implemented them as hash tables, which gives us potentially constant lookup times without the undue memory overhead required to store a large array of directory entries.

looks up the mobile pointer in its directory. There are three possible results of this lookup. First, the object may reside on the current processor, in which case the message can be handled locally. Second, there may be an entry in the directory indicating that an object resides or at least used to reside at some other processor. In this case, the message is sent to the processor indicated by the directory entry. If the object does not actually reside at the processor indicated by the directory entry, then the message will be forwarded. Third, the directory may have no entry for the mobile pointer. In this case, the mobile pointer's home node entry is used as a default "best guess" processor, and the message is sent to the home node.

The MOL allows directory entries to be out of date in order to minimize the cost of moving an object. When the user moves an object from some source processor to some target processor, only the source and target processors are aware of the change. The target processor sets the directory entry for the mobile pointer to point to the local address of the object, and the source processor sets its directory entry to point to the target processor. Other processors' directories are updated lazily, when they send a message to the object's old location and the message gets forwarded. Once a processor receives an update for an object, all further messages from the processor to the object will go directly to the object's new location.

Each mobile object has a movement sequence number associated with it, which is incremented each time the object moves. The MOL does not assume any special ordering properties in the network (such as FIFO or causal ordering), and allows the network to delay or slow down messages for arbitrary lengths of time. It is therefore possible for a processor to receive updates out of order. To prevent an older update from overwriting a more recent one, each update is tagged with the movement sequence number of the object that the update represents. The processor receiving updates can then tell which update is the most recent.

Because the user must be allowed to specify the object migration policy, the MOL does not move objects automatically, nor does the MOL make any special requirements on how objects must be packed and unpacked. To move an object, the user first calls *mob_uninstallObj*, which updates the processor's directory entry to reflect the mobile object's next location. The object's data is then moved to the new processor using, for example, *mpi_send* or *am_store*. The MoveInfo handle returned by *mob_uninstallObj* must also be moved so that MOL can track the state of the object. The object is then installed on the new processor by calling *mob_installObj*.

To free a mobile object, *mob_freeMobileObj* is called with the object's mobile pointer and a user handler as parameters. The MOL invokes the user handler on the processor where the object resides, so that the handler can free the object data. For instance, if multiple C++ objects are allocated to hold the mobile object's data, then the user handler can free each of those C++ objects. Then, the MOL sets this processor's directory entry for the mobile object to point back to the mobile pointer's home node. The home node can later reuse the mobile pointer for a new object when the user calls *mob_newMobilePtr*.

Processors can access local objects through the *mob_deref* function, which returns the local address of a mobile object if the object is currently on the local processor. If the object does not currently reside on the local processor, then *mob_deref* returns NULL. Remote objects are accessed via mobile messages, which are

discussed in detail in Section 3.4.

3.3 Analysis of the Directory Protocol

This section presents more details about the directory protocol and outlines a proof of the protocol's correctness. Correctness means that a message destined for some mobile object will always make progress towards that object as it is forwarded from processor to processor, in a way that this section will make precise.

A directory entry on some processor p for some object O contains a "best guess" of the location of O, and a movement sequence number indicating how up to date the directory entry is. To describe directory entries for an object O, let $G(O)$ be processor p's guess of the location of O, and $S(p)$ be the corresponding sequence number in p's directory entry for O. The value of $G(p)$ may be either a processor (which means that the object O does not currently reside on p, but may be found on the processor $G(p)$), or the value "*local*" (which means that O currently resides on processor p). $G(p) = p$ is not synonymous with $G(p) = local$ (the distinction between these will be clarified shortly). Finally, let s^* be the movement sequence number of the object O itself. s^* will be incremented each time O is moved. To aid the analysis, let $L(s)$ be the location that O resided at when O's sequence number was equal to s.

When the object O is allocated on some processor p_o, $G(p_o) = local$ and $S(p_o) = s^* = 1$. No other processors have yet heard of O; for all $p \neq p_o$, the assignment $S(p) = 0$ is used to indicate that processor p knows nothing about O, and $G(p)$ is set to p_o (the default guess of an object's location is the home node of that object). After the initial allocation of O, there are three operations that affect the directories:

1) O may be uninstalled from its current locationm, $p_{current}$, in preparation for moving it to some new node p_{new}. This updates the directories as follows:

$$G(p_{current}) := p_{new}; \qquad s^* := s^* + 1;$$

At this point, $L(s^*)$ is equal to p_{new}. Note that $S(p_current)$ is unchanged—it is still equal to the old s^*, not the newly incremented s^*, so that after the uninstall operation s^* equals $S(p_current) + 1$. If $p_{current}$ sends out an update at this point, it will have the form, "when O had sequence number $S(p_current) + 1$, it was located at processor p_{new}". An update of this form may be sent out even before O reaches p_{new}—this allows MOL to forward messages as eagerly as possible, without waiting for an acknowledgment that O has reached p_{new}. As described below, MOL is able to handle messages that arrive at a processor before the object itself arrives.

2) After O has been uninstalled from $p_{current}$, O may be installed on p_{new}. This updates the directories as follows:

$$G(p_{new}) := local; \qquad S(p_{new}) := s^*;$$

3) At any time, an update may show up at processor p. The update contains the information "when O had sequence number $s + 1$, it was located at processor $L(s + 1)$". If $s > S(p)$, then the update is placed in p's directory:

$$G(p) := L(s + 1); \qquad S(p) := s;$$

If $s \leq S(p)$, then p ignores the update (this ensures that old information never overwrites new information).

Each operation affects the state of only a single processor, so an operation can be thought of as an atomic action, and the evolution of the state of the whole system can be described by a sequence of atomic operations.

Figure 7 shows an example state, where the object O was allocated on processor u_1, then moved to u_2, then finally moved to u_3. In addition, processor u_4 received an update and knows that the object moved to u_3. The arrows in the graph indicate the guesses $G(u)$ of each processor. Note that the arrows out of each processor u point to a $G(u)$ with a higher sequence number than u's sequence number; in other words, $S(G(u)) > S(u)$. Here is a more rigorous statement of this observation: at any point in the sequence of operations on the system, the following invariant holds:

- If an object O is currently installed on some processor $p_{current}$, then $S(G(p)) > S(p)$ for all $p \neq p_{current}$.

- If an object O is currently uninstalled (it is in transit from processor $p_{current}$ to processor p_{new}), then $S(G(p)) > S(p)$ for all p such that $G(p) \neq p_{new}$.

This invariant is the crux of the correctness of the protocol. Roughly, it means that if a message follows the "best guesses" in the processors' directories, than it will only get forwarded from a processor with a lower sequence number to a processor with a higher sequence number. With each step, the message sees a higher sequence number, and the message therefore makes progress towards the destination object. When the message reaches a node with the sequence number s^*, it has found the object. Note that s^* may also be increasing if O is moving around, so it is possible for O to outrace the message so that the message never catches up, even though the message makes progress towards O. However, if O eventually settles down, then the message will eventually catch up with it; the message will not get stuck on a processor or lost in a loop.

The rough argument above leaves out some details, because the invariant contains some cases where $S(G(p)) > S(p)$ does not hold. First, if O is installed on processor $p_{current}$, then $S(p_{current}) = local$, so $S(G(p_{current}))$ does not exist. This is not a problem, though – if a message for O arrives at $p_{current}$ then it has reached its final destination and does not need to be forwarded. Second, if O is in transit from $p_{current}$ to p_{new}, then $S(G(p))$ might be less than or equal to $S(p)$ if $G(p) = p_{new}$. When this happens, a message destined for O that is sent from p to p_{new} may arrive at p_{new} before O arrives at p_{new}. In the MOL system, a message detects this condition by watching sequence numbers as it moves from processor to processor. If it steps to a processor whose sequence number is not greater than the previous processor, then the message knows that it has stepped to the processor p_{new} to which the object is moving, and it simply waits for the object on that processor. Thus, the message reaches O even when the message sees a decrease in the sequence numbers. As a special case of this situation, it is possible for $G(p_{new}) = p_{new}$ to occur (p_{new} may receive an update saying that O is moving to p_{new} before O actually arrives at p_{new}). This is why $G(p_{new}) = local$ is not equivalent to $G(p_{new}) = p_{new}$ – the first equation says that O resides on p_{new}, while the second says

that O does not reside on p_{new} but is in transit towards p_{new}.

The full proof of the invariant is omitted for brevity. The key point in the proof is as follows: when an object O with sequence number s^* is installed on a processor $p_{new} = L(s^*)$, then $S(L(s^*)) = s^*$, and sequence numbers $S(p)$ only increase over time (as s^* grows), so for all sequence numbers $s < s^*$, $S(L(s))$ is always always greater than or equal to s. For an update "when O had sequence number $s+1$, it was located at processor $L(s+1)$" to change a directory entry at processor p, the condition $s > S(p)$ must initially hold. The update sets $G(p)$ to $L(s+1)$, so $S(G(p))$ is then equal to $S(L(s+1))$, which must be greater than or equal to $s+1$, which in turn is greater than s.[2] The update sets $S(p)$ to s, so $S(G(p)) > S(p)$ is true after the update.

Interestingly, the correctness of the protocol does not depend on any ordering properties of the network. The protocol will work, for instance, even if the communication channels do not provide FIFO ordering.

One last note: the correctness proof assumed that sequence numbers could grow arbitrarily large, but the hardware representation of an integral number is bounded. As a result, it is possible for sequence numbers to overflow after an object has moved many times. If this happens, the MOL broadcasts an update to all processors before an overflow occurs. Overflows happen rarely, so these broadcasts do not impose a significant performance penalty.

3.4 Message Layer

There are two types of MOL messages: *mobile requests* which are sent to specific processors and are not forwarded, and *mobile messages* which are sent to mobile objects and may be forwarded from processor to processor as they make progress toward their target object. Both requests and messages transfer data to their target processors and invoke a user-specified handler upon receipt.

In order to avoid the overhead of dynamically allocating memory to hold incoming and outgoing messages, the MOL maintains incoming and outgoing message pools on each processor. This allows senders of a message to have memory preallocated on a remote node to hold that message, thus avoiding the handshaking and address passing that would otherwise be necessary. The sizes of the entries in the message pools, the number of message pools, and the initial number of entries in each pool can be determined by the application at the time of MOL initialization. This number of pool entries is not fixed throughout the life of the application; pools are allowed to grow dynamically during runtime in order to avoid deadlock.

Whenever a message is sent, the sending processor reserves space in an outgoing pool for the message, and it copies the message into this space. After the outgoing message is constructed, space is reserved in the remote nodes incoming message pool, and the message is sent via an asynchronous store mechanism provided by the underlying communication substrate. Once this store completes, the message's space in the outgoing pool is released.

[2] This argument does not hold if $G(p)$ is set to p_{new} while O is still in transit to p_{new}, because $S(L(s^*))$ may be less than s^* before the installation; this is part of the reason for the qualification $G(p) \neq p_{new}$ in the invariant.

When the message arrives at the destination processor, a DMCS handler is invoked to handle the incoming message. If the message needs to be forwarded, then space is reserved in the next destination processor's incoming message pool and the message is sent on with another asynchronous store. However, this can not be done from within the original DMCS handler due to restrictions placed on communications from within handlers. Instead, the original handler is placed into a queue of delayed handlers, to be executed at a later time after the original handler exits. Some communication substrates, in order to guarantee deadlock-free execution, prevent communication from within a handler. In order to ease application portability, we maintain this semantic rule in the MOL, even though it does not apply to DMCS. For this reason, we provide three different types of user handlers, with varying degrees of performance and functionality.

The application is able to specify which type of handler should execute on the target node. The first type is a *function* handler, and these execute directly from within the DMCS handler running on the destination processor. This is the fastest approach and the one with the lowest overhead. However, it is the least flexible; a functional handler is not able to perform any communication operations or to context switch to another thread. The reasons for this are deadlock prevention and to ensure that progress is being made. The second type of handler is a *delayed* handler. Delayed handlers are not executed directly from within the DMCS handler, but instead are enqueued, and executed when the delayed queue is flushed (which comes during a *mob_poll()* call). This is slightly slower, but the user is allowed to do an arbitrary amount of communication. The third type of handler is the most flexible, but also has the most overhead: *threaded* handlers. Threaded handlers allow communication and context switching

4 Application: Guaranteed-Quality 3D Delaunay Mesh Generation

Mesh generation is the procedure of discretizing a geometric domain into small and simpler cells (or elements), such as triangles for two-dimensional domains, and tetrahedra for three-dimensional domains. Mesh generation is a necessary step for the discretization of continuous partial differential equations (PDE's) into discrete systems of algebraic equations using finite element analysis.

Delaunay triangulation algorithms have been used very successfully for guaranteed-quality unstructured grid generation on sequential machines. Delaunay-based algorithms generate unstructured grids by iteratively adding new points and modifying the existing triangulation by means of purely local operations. These algorithms can be viewed as iterative procedures performing four basic operations per iteration: (i) *point creation*, where a new point is created using an appropriate spatial distribution technique; (ii) *point location*, where an element that contains the new point is identified; (iii) *cavity computation*, where existing elements that violate the Delaunay property [41] are removed; and (iv) *element creation*, where new triangles are built by connecting the new point with old points such that the resulting triangulation satisfies certain geometric

properties. This type of incremental construction of a Delaunay triangulation is sometimes referred to as the Bowyer-Watson (BW) algorithm [43, 44].

In our parallel implementation of the BW algorithm, an initial Delaunay tetrahedralization, T_0, of a set of points is overdecomposed into $N \gg P$ subdomains (or *regions*), where P is the number of processors. Regions are assigned to processors in a way that maximizes data locality, and each processor is responsible for managing multiple regions. As a result of this decomposition, imbalance can arise due to unequal distribution of regions over the processors, and due to large differences in computation in each processor (e.g. because of regridding changing topology and geometry, or because of h-refinement).

By decomposing T_0 into many regions, new points can be inserted concurrently into many areas of the mesh. However, two new points cannot be added concurrently if their corresponding cavities overlap; retriangulating the cavities will result in edges of new elements crossing other elements in the triangulation. To prevent the construction of an invalid triangulation, either the processors involved must be synchronized, or the computation of one of the cavities must be restarted at a later time. In the latter case, there is a *setback* in the progress of the mesh generation, in that a halted cavity must release its elements, and the work done to compute the cavity must be done again. Both synchronization and setbacks introduce two additional sources of imbalance, the effects of which are exacerbated by the variable and unpredictable computation and communication patterns of the parallel implementation of the BW algorithm. We implement an implicit work–stealing load balancing algorithm to deal with this imbalance at run–time.

4.1 Load-balancing with a Work-stealing Method.

The work-stealing load balancing method we 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 a sufficient amount of work to maintain consistent resource utilization. The regions can be viewed as the work-units or objects which the load balancer can migrate to re-distribute the computation. Each region can be viewed as a mobile object; by associating a mobile pointer with each region, messages sent to migrated regions will be forwarded by the MOL to the regions' new locations.

The MOL allows the load-balancing library to migrate data without interfering with the message-passing of the code that performs the computation. This suggests an incremental development of parallel adaptive applications. Initially, the developer implements an efficient and correct code while dealing only with data-locality and communication, not dynamic load-balancing. Once this phase is completed, load-balancing algorithms which maintain data-locality and minimize communication can be developed with minimum modifications to the original code. The developer is spared the problems of concurrently developing two very complex, but conceptually independent components of an application.

4.2 Data Movement Using the MOL.

The critical steps in the load balancing phase of the mesh generator are the region migration and the updates of the mesh near the interfaces of the regions. To move a region, the MOL requires that *mob_uninstallObj* be called to update the sending processor's local directory to reflect the pending change in the region's location. Next, a programmer-supplied procedure is used to pack the region's data into a buffer, which must also contain the region's mobile pointer and the MoveInfo structure returned by *mob_uninstallObj* to track the region's migration.[3] Then, a message-passing primitive (e.g. *mpi_send or am_store*) is invoked to transport the buffer. Upon receiving the buffer on the target processor, another user-supplied procedure unpacks and rebuilds the region in the new processor. After the region has been unpacked, *mob_installObj* must be called with the region's mobile pointer and the MoveInfo structure to update the new processor's directory.

Since the MOL is used to move data, standard message-passing primitives, like *mpi_send or am_put*, will not work to send mesh updates from one region to another, since regions can be migrated. The user can use MOL's *mob_message* to send messages related to mesh updates; this guarantees that the message will be forwarded in a correct (see Section 3.3) and efficient way to the appropriate processor. *mob_message* also updates the sending processor's directory so that, unless the target region moves again, subsequent messages will be sent directly to the new processor, instead of forwarded.

The MOL is an integral part of simplifying the mesh generation process, since it allows regions, and thus in-progress cavities, to be migrated without blocking or synchronization. More importantly, the MOL eliminates problems with moving a region which is a target of a cavity expansion or region update message. Since the MOL forwards messages from a migrated region's original processor to the region's new location, messages bound for the migrated region are received and enqueued on the new processor to await the reconstruction and installation of the region. Otherwise, it would be up to the application to forward the misdirected messages; the implementation of such a forwarding mechanism is non-trivial, and at best error-prone.

5 Performance Data

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 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 LAPI implementation developed in [27]. The benchmarks measured the per-hop latency of messages ranging from 1 to 8192 bytes, as compared to the equivalent *LAPI_Amsend* calls. The performance is

[3] This requirement will be removed from the next version of the MOL.

very reasonable; the latency of *mob_request* is within about 11% of the latency of *LAPI_Amsend*, while *mob_message*'s latency is about 10% to 14% higher than *LAPI_Amsend*'s latency.

To illustrate the importance of the MOL's updates, fig. 9 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. In the case of the mesh generator, a large number of cavity expansion messages are sent, relative to the number of times a mesh region is migrated (see fig. 13), resulting in a low amortized cost for forwarding cavity expansion messages. Figure 10 shows the performance of the MOL's three types of handlers. The graph clearly shows that the overheads caused by the delayed and threaded handlers are fairly low relative to the functionality they add.

The next set of graphs represents data for a parallel mesh with between $1,000,000$ and $4,000,000$ tetrahedra, and for load imbalances of between 10% and 45%. Each of the 16 processors in the system started with 16 regions. All measurements were taken on an SP2, using a LAPI implementation of the MOL.

Figure 11 shows the computation times for no load balancing, for explicit load-balancing using Parallel METIS [9], and for implicit load-balancing using a work-stealing method for 10%, 15%, 20%, and 45% imbalance for a 2 million tetrahedral mesh. The time to load balance using Parallel METIS consists of the synchronization time, the CSR translation time, and the data movement time. Table 2 shows these times for a single invocation of Parallel METIS, at the end of the mesh generation phase. Multiple invocations of Parallel METIS would incur all three costs for each invocation. Therefore, the best case for this particular application using Parallel METIS is load-balancing at the end of the mesh generation stage.

Figure 12 shows the same data, but for $1,000,000$, $2,000,000$ and $4,000,000$ tetrahedral meshes at 10% imbalance. Finally, fig. 13 details the minimal time spent in the MOL for implicit load-balancing of 1, 2, and 4 million tetrahedral meshes at 10% imbalance. The average number of cavity expansions per processor and the maximum number of region migrations is given iside each bar. The MOL overhead is indeed small, because the forwarding time is amortized over the number of messages per region migration.

6 Summary and Conclusions

We have presented a run-time substrate, the Mobile Object Layer (MOL), which supports a global logical name space and data migration for the efficient implementation of dynamic load-balancing strategies for adaptive, irregular parallel applications. The MOL uses a global logical namespace to assist in message passing between objects, and implements distributed directories to translate between logical and physical addresses when an object is migrated, or when a message is sent or forwarded to an object. A message may be forwarded when the message is destined for an object which has been migrated from the sending

processor's "best guess" location. In this case, as the message is forwarded, each processor in the path uses the mobile pointer of the target object to determine the next "best guess" location of the object, and forwards the message to the next location. When the target object is located, the processor containing the object updates the directory of the processor sending the message, at which point subsequent messages from the originating processor are no longer forwarded. The amortized costs of translation from logical to physical addresses and forwarding messages are 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 updates have been received.

The MOL offers a substantial improvement over explicit message passing systems through the added functionality of communication between *mobile objects* without knowledge of an object's location. The MOL eases the burden placed on developers of mobile, adaptive applications, and hides the complexity involved in maintaining the validity of mobile pointers. The "separation of concerns" philosophy is adhered to in the system; the DMCS layer ensures message ordering and implements an efficient one-sided message passing interface, while the MOL maintains the causality of messages between objects. Application code would responsible for the migration of objects and for the validity of references to those objects, if the application relied solely upon a message-passing library, like MPI or LAPI.

The MOL is lightweight, in that its latency is very close to that of the message layer it is built upon, even for forwarded messages. The actual latency of the MOL primitives is within 10% to 14% of the the latency of the underlying communication substrate. Thus, little is lost in the efficiency of an application relying upon the MOL to effect object migration. This is accomplished by only doing a minimal amount of extra computation to maintain the distributed directories and the incoming and outgoing message pools. The results of this minimalism show up in the MOL interface; the MOL, in contrast to existing object-oriented high-level languages like Amber [28], and software DSM systems like Treadmarks [30], does not provide a comprehensive solution to the scalable shared memory paradigm. Instead, it is designed to reduce the amount of effort needed to efficiently and easily implement mobile, adaptive applications using a global name space on distributed memory parallel machines.

7 Future Work

Our future plans include augmentations to address needs like I/O for computational steering, and out-of-core computations for traditional parallel platforms and for multi-layer [21] architectures, like the HTMT Petaflops design. Also, further support for dynamic load-balancing will be added via dynamic resource allocation and deallocation. We also plan on porting DMCS to the Virtual Interface Architecture [45] communication subsystem for NT-SMP servers.

Enhancements to the MOL will include pointer arithmetic and application-specific distributed shared memory management schemes, like space-filling curves, for both structured and unstructured applications.

Finally, we will be developing a new open framework that will allow easier and more effective development of run-time systems on commercially available communications substrates for networks and clusters of NT workstations. All of these enhancements will be implemented such that the application developer will benefit from and trust these systems, while at the same time improving the ease-of-use and the maintainability of complex software systems, even for non-experts.

8 Acknowledgments

We are grateful to our colleagues at Cornell Theory Center for making available to us their 128 node SP machine. The code was developed on our two, 2-way SMP node SP that was donated to this PI by IBM's Shared University Research program. Also, we thank Horst Simon, Keshav Pingali, Thorsten von Eicken, Chi-Chao Chang and Greg Czajkowski for their very helpful comments on earlier and this draft of the paper.

References

[1] Ingrafea, T, Personal Communication, Cornell Univeristy, 1998.

[2] Mavriplis D., and Pirzadeh S., Large-scale parallel unstructured mesh computations for 3D high-lift analysis, NASA/CR-1999-208999 ICASE Report No. 99-9, Institute for Computer Applications in Science and Engineering Mail Stop 403, NASA Langley Research Center Hampton, VA 23681-2199, February 1999.

[3] Horst D. Simon. Partitioning of unstructured problems for parallel processing. Technical Report RNR-91-008, NASA Ames Research Center, Moffet Field, CA, 94035, 1990.

[4] Hendrickson B. and Robert L. The Chaco User's Guide, version 1.0. Technical Report SAND93-2339, Sandia National Laboratories, 1993.

[5] Nikos Chrisochoides, Elias Houstis and John Rice. Mapping Algorithms and Software Environment for Data Parallel PDE Iterative Solvers Special Issue of the Journal of Parallel and Distributed Computing on Data-Parallel Algorithms and Programming, 1994;21(1):75–95.

[6] H. L. de Cougny, K.D. Devine, J.E. Flahery, R.M. Loy, C. Ozturan, and M. Shephard. Load balancing for parallel adaptive solutions of partial differential equations. Appl. Numer. Math. 1994;16(1-2):157-182.

[7] A. Vidwans, Y. Kallinderis and V. Venkatakrishnan Parallel Dynamic Load-Balancing Algorithm for Three-Dimensional Adaptive Unstructured Grids, AIAA Journal, 1995;32(3):497-505.

[8] George Karypis and V. Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing, to appear.

[9] Kirk Schloegel, George Karypis, and Vipin Kuma. Parallel Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes. Technical Report 97-014, University of Minnesota, 1997.

[10] Roy D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. Concurrency Practice and Experience, 1991;3(5):457-481.

[11] C. Walshaw and M. Berzins, Dynamic load balancing for PDE solvers an adaptive unstructured meshes. University of Leeds, School of Computer Studies, Report 92.32, 1992.

[12] Horst D. Simon, A. Shon, R. Biswas. HARP: A dynamic spectral partitioner. J. Parallel Distrib. Comput. 1998;50(1-2):83-103.

[13] Christos H. Papadimitriou and Kenneth Steiglitz. Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Englewood Cliffs, NJ 07632, 1982.

[14] Chrisochoides N, Multithreaded Model For Load Balancing Parallel Adaptive Computations On Multicomputers, Journal of Applied Numerical Mathematics, 1996;20:1–17.

[15] Blumofe R. and Leiserson C. Scheduling Multithreaded Computations by Work Stealing Proceedings of the 35th Annual IEEE Conference on Foundations of Computer Science, Santa Fe, NM, Novemeber 20-22, 1994.

[16] Christopher F. J. The Cilk system for Parallel Multithreaded Computing. Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January, 1996.

[17] R. Chandra and A. Gupta and J.L. Hennessy Data Locality and Load Balancing in COOL PPOPP4, pp 249–259, San Diego, CA, May 19–22, 1993.

[18] Casavat T, and Khul Hon G. A taxonomy of Scheduling in Gneral-Purpose Distributed Computing Systems IEEE Transactions on Software Engineering, 1988;14(2):141–154.

[19] L. Kale, S. Krishnan, Charm++, Parallel Programming Using C++ (eds. Wilson, G. and Lu, P.), The MIT Press, 1998.

[20] Bruce Carter, Chuin-Shan Chen, L. Paul Chew, Nikos Chrisochoides, Guang R. Gao, Gerd Heber, Antony R. Ingraffea, Roland Krause, Chris Myers, Demian Nave, Keshav Pingali, Paul Stodghill, Stephen Vavasis, Paul A. Wawrzynek. Parallel FEM Simulation of Crack Propagation – Challenges, Status, and Perspectives. To appear in Irregular 2000.

[21] Chrisochoides N, Application-driven Approach for Prototyping Runtime Systems for Future Teraflops and Petaflops Architectures NSF Proposal Report (unpublished), Computer Science and Engineering, University of Notre Dame, July 1998.

[22] Chrisochoides N, Pingali, K, Kodukula I, Data Movement and Control Substrate for Parallel Scientific Computing Lecture Notes in Computer Science (LNCS), Springer-Verlag 1997;1199:256.

[23] Portable Run-Time Systems Consortium, http://www.cs.uoregon.edu/research/paracomp/ports

[24] Foster, I, Kesselamn C, Tuecke S, The Nexus Task-parallel Runtime System, Proc. 1st Intl Workshop on Parallel Processing, 1994.

[25] Pete Beckman and Dennis Gannon, Tulip: Parallel Run-time Support System for pC++, http://www.extreme.indiana.edu

[26] MPI Forum (1997), Message-Passing Interface Standard 1.0 and 2.0, http://www.mcs.anl.gov/mpi/index.html

[27] DiNicola P, Gildea K, Govindaraju R, Mirza J, Shah G; LAPI Architecture Definition: Low Level API Draft, IBM Confidential Report, December 1996.

[28] J.S. Chase , F.G. Amador, E.D. Lazowska, H.M. Levy and R.J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors SOSP12, pp 147–158, December, 1989.

[29] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. Ch. Eigler, and G. G. Gao, ABC++: Concurrency by Inheritance in C++, IBM Systems Journal, Vol. 34, No.1, 1995, pp. 120-137.

[30] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel TreadMarks: Shared Memory Computing on Networks of Workstations IEEE Computer, 1996;29(2):18-28.

[31] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach CRL: High-Performance All-Software Distributed Shared Memory SOSP15, pp 213–228, Copper Mountain, CO, December, 1995.

[32] D. E. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumeta, T. von Eicken, and K. Yelick, Parallel Programming in Split-C, In Proceedings of Supercomputing '93.

[33] K. M. Chandy and C. Kesselman, CC++: A Declarative Concurrent Object-Oriented Programming Notation, In Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993.

[34] Chang, C.; Sussman A.; and Saltz, J.; Chaos++, Parallel Programming Using C++ (eds. Wilson, G. and Lu, P.) The MIT Press, 1998.

[35] Jul E., Levy H, Hutchison N, Black A; Fine-Grained Mobility in the Emerald System, TOCS, 1988; 6(1):109-133.

[36] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, The Stanford FLASH Multiprocessor , In Proceedings of the 21st International Symposium on Computer Architecture, pages 302-313, Chicago, IL, April 1994.

[37] D. E. Culler, K. Keeton, L. T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, and K. Wright, Generic Active Messages Interface Specification, UC Berkeley, November 1994.

[38] C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken, Low-Latency Communication on the IBM RISC System/6000 SP, In Proceedings of Supercomputing '96.

[39] PORTS Level 0 Thread Modules from Argonne/CalTech, ftp://ftp.mcs.anl.gov/pub/ports/

[40] David Keppel, Tools and Techniques for Building Fast Portable Threads Package, UW-CSE-93-05-06, Technical Report, University of Washington at Seattle, 1993.

[41] Preparata F, and Shamos M, Computational Geometry, AnIntroduction, Springer-Verlag, pp 398, 1985.

[42] L. Paul Chew, Nikos Chrisochoides, and Florian Sukup, "Parallel Constrained Delaunay Meshing," In *Proceedings of 1997 Joint ASME/ASCE/SES Summer Meeting, Special Symposium on Trends in Unstructured Mesh Generation*, June 29-July 2, 1997, Nortwestern Univeersity, Evanston, IL.

[43] Bowyer A, Computing Dirichlet Tessellations, The Computer Journal, 1981; 24(2):162–166.

[44] Watson D, Computing the n-dimensional Delaunay tessellation with applications to Vornoi polytopes, The Computer Journal, 1981;24(2):167–172.

[45] Virtual Interface Architecture Specification, Version 1.0, Compaq Corporation, Intel Corporation, and Microsoft Corporation, 1997. http://www.viarch.org.

Biographies of Authors

Nikos P. Chrisochoides is an Assistant Professor in the Department of Computer Science and Engineering at University of Notre Dame. His research focuses on the integration of algorithmic and applications-driven research in parallel computing with research in parallel runtime software systems for high-end architectures. Chrisochoides received his Ph.D in Computer Science from Purdue University in 1992. He has been the first Alex Nason Fellow at the Northeast Parallel Architectures Center, Syracuse University from 1992 to 1995 and a Research Associate at Cornell from 1995 to 1997. He received an NSF Career Award on: "Application-driven Approach for Prototyping Runtime Systems for Future Teraflops and Petaflops Architectures".

Kevin Barker is a graduate student in the Department of Computer Science and Engineering at University of Notre Dame. His research interests include the design of application-driven parallel run-time systems. He received a B.S. in Computer Science in 1997 from North Carolina State University.

Demian Nave is a graduate student in the Department of Computer Science and Engineering at University of Notre Dame. His research includes the formulation of a theoretical framework for guaranteed-quality parallel Delaunay mesh generation, with applications to parallel finite-element simulations. Nave received a B.S. in Computer Science and a B.S. in Aerospace Engineering from the University of Notre Dame in 1997.

Chris Hawblitzel is a graduate student in Computer Science at Cornell University. His research interests include parallel processing and operating systems. His current research is in language based security.

9 Figures, Tables, and Captions

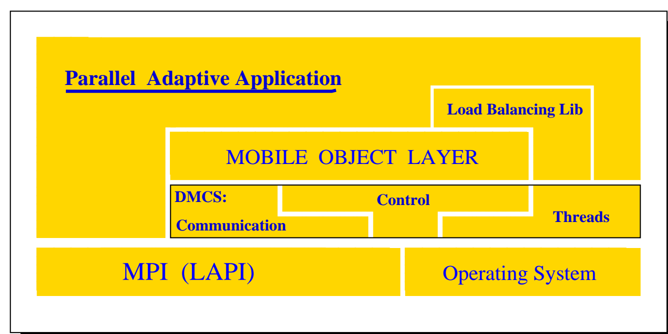

Figure 1: Parallel run-time system: Architecture.

```
class TreeNode {
public:
    float data;
    TreeNode *parent;
    TreeNode **children;
    int numChildren;

    void setAll(float f) {
        int i;
        data = f;
        for(i = 0; i < numChildren; i++) {
            children[i]->setAll(f);
        }
    }
};
```

Figure 2: Serial Implementation of a Tree Structure.

```
class TreeNode {
public:
    float data;
    MobilePtr parent;
    MobilePtr *children;
    int numChildren;

    void setAll(float f) {
        int i;
        data = f;
        for(i = 0; i < numChildren; i++) {
            mob_message(children[i], remoteSetAll, &f, sizeof(f),
                        MOB_DELAYED_HANDLER, NULL);
        }
    }
};

void remoteSetAll(int srcProc, MobilePtr mp, TreeNode *treeNode,
                  float *f, int nbytes, void *arg) {
    treeNode->setAll(*f);
}
```

Figure 3: Parallel Implementation of a Tree Structure based on MOL.

```
for(i = 0; i < numChildren; i++) {
    TreeNode *child = (TreeNode *) mob_deref(children[i]);
    if(child != NULL) child->setAll(f);
    else mob_message(children[i], remoteSetAll, &f, sizeof(f),
                     MOB_DELAYED_HANDLER, NULL);
}
```

Figure 4: Optimizing code with *mob_deref*.

Mesh	RSB				PGK			
Size	Time	Sep. Size	S/V	Max Conn.	Time	Sep. Size	S/V	Max Conn.
200K Tets	65	1836	.036	10	4	1629	.032	11
500K Tets	164	3047	.024	9	11	3246	.026	10
1M Tets	389	3859	.017	9	25	4044	.022	9

Table 1: Time in seconds and the quality of separators measured in terms of size of the maximum separator (in number of faces) and surface to volume ratio, for a 16-way partition of $200,000$, $500,000$ and 1 million tetrahedron meshes using RSB from Chaco and PGK from Metis.

Size	Execution	Synchronization	CSR Translation	Data-Movement	Total, 1 Invocation
1M Tets	47s	5s (11%)	1s (2%)	3s (6%)	9s (19%)
2M Tets	94s	6s (6%)	2s (2%)	7s (7%)	15s (15%)

Table 2: Total execution time in seconds for generating 1 million and 2 million tetrahedron meshes with 10% imbalance on 16 nodes of an SP/2. Parallel METIS is invoked once at the end to explicitly load-balance the mesh.

```
class TreeNode {
public:
    MobilePtr myself;
    MoveInfo moveInfo;
    float data;
    MobilePtr parent;
    MobilePtr *children;
    int numChildren;

    void setAll(float f) {...}
};

void migrateNode(TreeNode *treeNode, int newProc) {
// uninstall the object:
    treeNode->moveInfo = mob_uninstallObj(treeNode->myself, newProc);
// send the object data to the new processor:
    int nbytes = sizeof(TreeNode) + treeNode->numChildren * sizeof(MobilePtr);
    mob_request(newProc, migrateHandler, treeNode, nbytes,
                MOB_DELAYED_HANDLER, NULL);
// free the local object:
    free(treeNode);
}

void migrateHandler(int srcProc, char *treeNodeData, int nbytes, void *arg) {
// set up the new object:
    TreeNode *treeNode = (TreeNode *) malloc(nbytes);
    memcpy(treeNode, treeNodeData, nbytes);
    treeNode->children = (MobilePtr *) (treeNodeData + sizeof(TreeNode));
// install the new object:
    mob_installObj(treeNode->myself, treeNode, treeNode->moveInfo);
}
```

Figure 5: Implementation of Object Migration using MOL.

```
MobilePtr mob_newMobilePtr(void *localObjPtr,
                           MobileObjData *mobileObjData);

void mob_freeMobileObj(MobilePtr mp,
                       mob_handler freeHandle);

MoveInfo mob_uninstallObj(MobilePtr mp,
                          int newProc);

void mob_installObj(MobilePtr mp,
                    void *localPtr,
                    MoveInfo moveInfo);

void *mob_deref(MobilePtr mp);
```

Figure 6: MOL Mobile Object Interface.

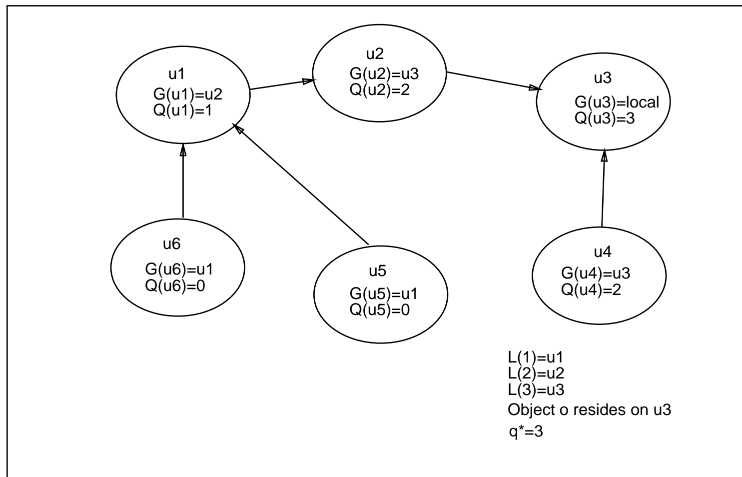

Figure 7: Example System State

```
#define MOB_FUNC_HANDLER 1
#define MOB_DELAYED_HANDLER 2
#define MOB_THREAD_HANDLER 3

void mob_poll();

void mob_message(MobilePtr destObjPtr,
                 mob_handler handler,
                 void *src_addr,
                 int nbytes,
                 int flags,
                 void *handler_arg);

void mob_request(int destProc,
                 mob_handler handler,
                 void *src_addr,
                 int nbytes,
                 int flags,
                 void *handler_arg);

/* message handler prototype: */
void handler(int srcProc,
             MobilePtr mp,
             void *local_obj_ptr,
             void *msg_data,
             int nbytes,
             void *handler_arg)

/* request handler prototype: */
void handler(int srcProc,
             void *msg_data,
             int nbytes,
             void *handler_arg)
```

Figure 8: MOL Message Interface

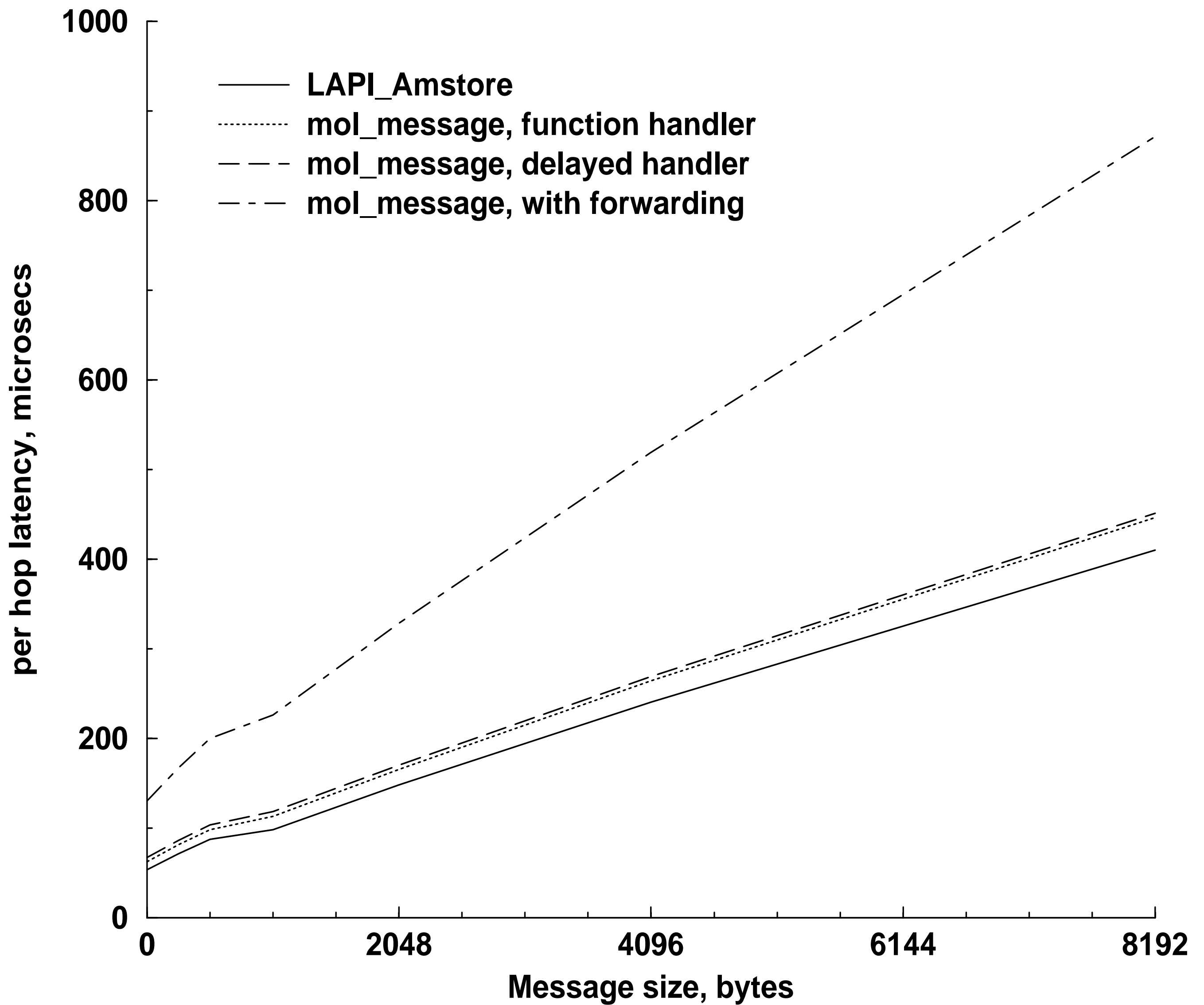

Figure 9: MOL forwarding overhead versus message size, per hop. When amortized over the number of messages sent to a migrated object, forwarding overhead approaches the *mol_message* time, since most messages are never forwarded (due to directory updating).

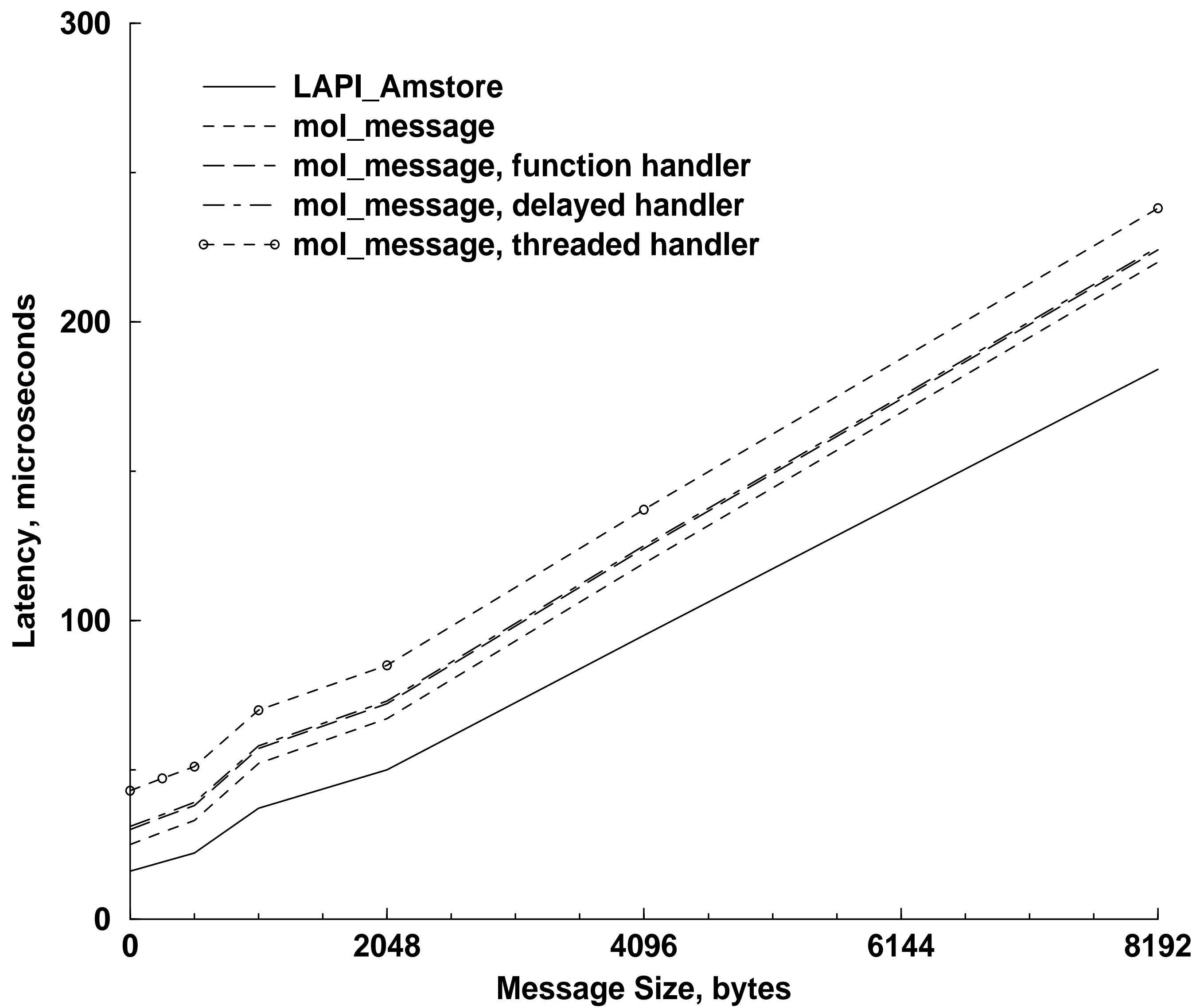

Figure 10: MOL handler overhead versus message size.

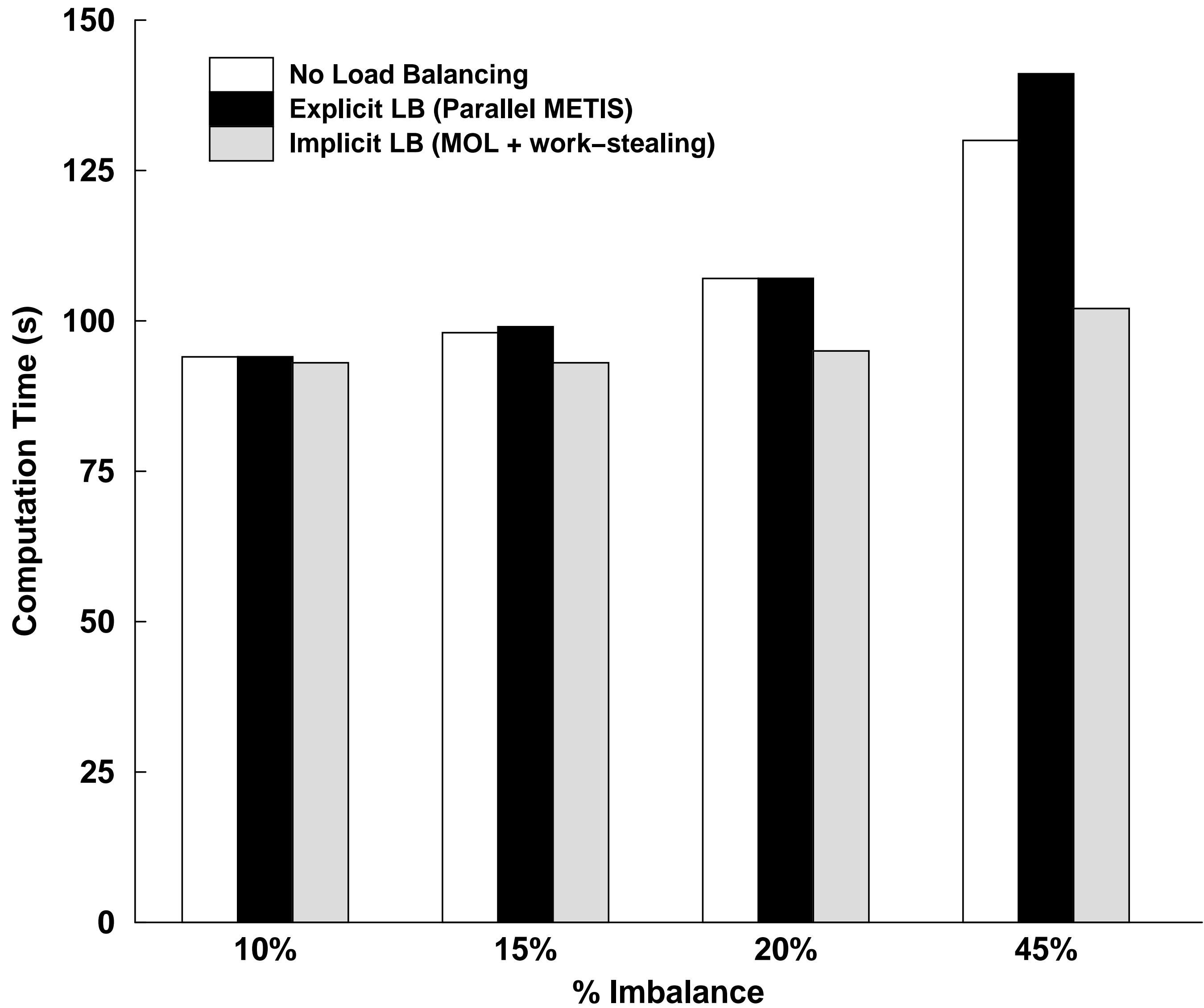

Figure 11: Computation times for no load-balancing, explicit load-balancing, and implicit load-balancing versus percent imbalance for a 2 million tetrahedron mesh, on 16 processors of an IBM SP/2.

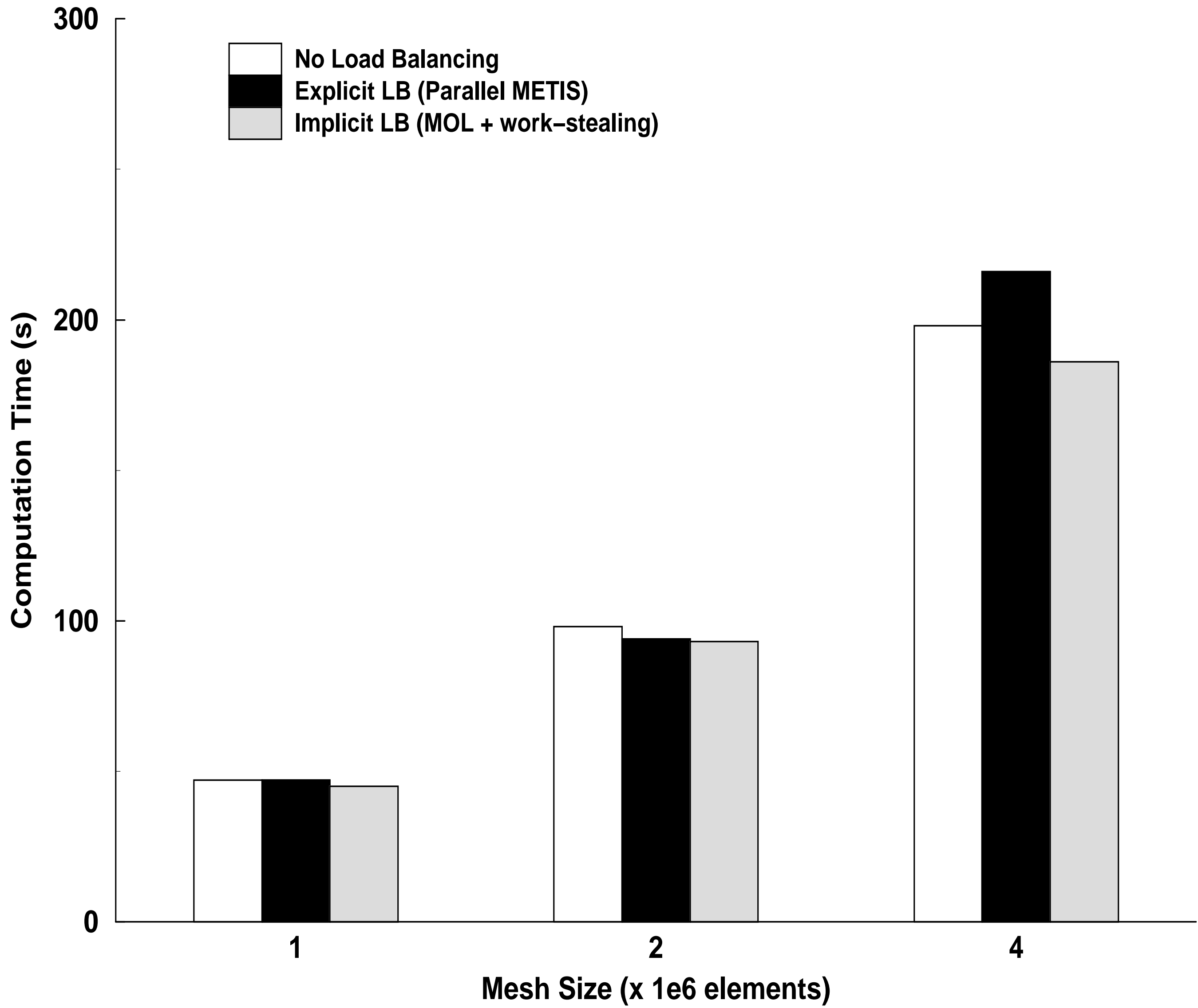

Figure 12: Computation times for no load-balancing, explicit load-balancing, and implicit load-balancing for 1, 2, and 4 million tetrahedra at 10% imbalance, on 16 processors of an IBM SP/2.

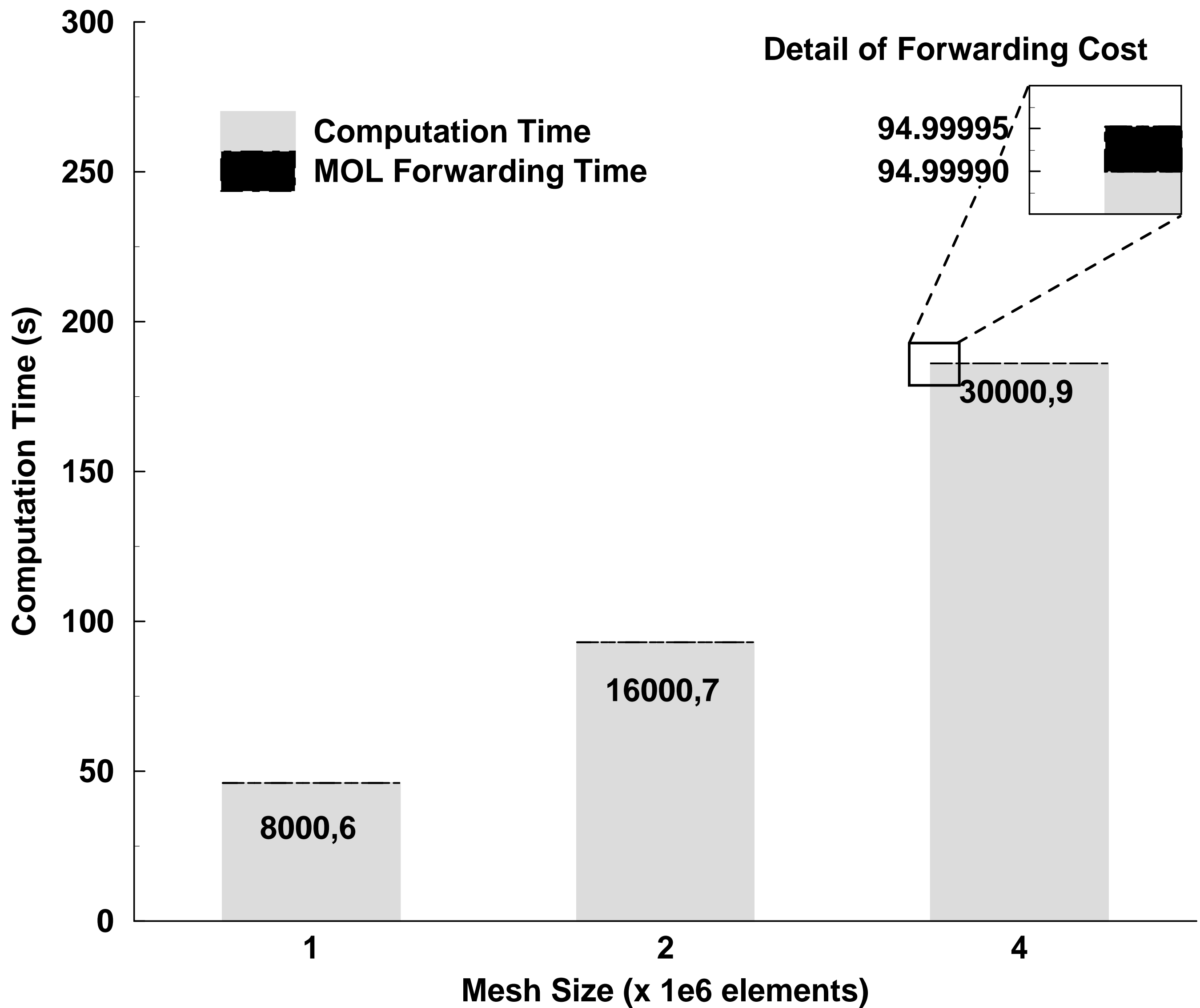

Figure 13: Detail of MOL forwarding in the computation times for implicit load-balancing at 10% for 1, 2, and 4 million tetrahedron meshes on 16 processors of an IBM SP/2. The average number of cavity expansions and the maximum number of region migrations is given inside each bar.