W&M ScholarWorks

Dissertations, Theses, and Masters Projects

Theses, Dissertations, & Master Projects

2003

Location Management in a Mobile Object Runtime Environment

andriy Fedorov
College of William & Mary - Arts & Sciences

Follow this and additional works at: https://scholarworks.wm.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
Fedorov, andriy, "Location Management in a Mobile Object Runtime Environment" (2003). *Dissertations, Theses, and Masters Projects.* Paper 1539626829.
https://dx.doi.org/doi:10.21220/s2-7x0m-dp11

This Thesis is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

LOCATION MANAGEMENT IN A MOBILE OBJECT RUNTIME ENVIRONMENT

A Thesis

Presented to

The Faculty of the Department of Computer Science

The College of William & Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Master of Science

by

Andriy Fedorov

2003

APPROVAL SHEET

This thesis is submitted in partial fulfillment of

the requirements for the degree of

Master of Science

[illegible]

Andriy Fedorov

Approved by the Committee, December 2003

[illegible]

Nikos P. Chrisochoides, Chair

[illegible]

Phil Kearns

[illegible]

Bruce B. Lowekamp

[illegible]

Dimitrios S. Nikolopoulos

[illegible]

Xiaodong Zhang

Table of Contents

ACKNOWLEDGMENTS

I owe a lot to my advisor Nikos Chrisochoides for invaluable discussions, guidance and continuous support. I would like to thank professor Phil Kearns for his eye-opening and inspiring lectures and assignments. This thesis would never be written without Allen and Meg Tucker, who showed me the way to William and Mary, and without those who helped me during my first months in the USA: Ken Blakely, Vanessa Godwin, Denis Kenzior, Brian Lamprecht, Bob and Debby Noonan, and many others. Thanks to Kevin Barker, Andrey Chernikov, Chaman Singh Verma and Tom Crockett for sharing their experience and for being very patient answering my numerous questions. Thanks to The United States of America for the opportunities I have here in advancing myself.

This work was partially supported by the National Science Foundation grants ITR-0312980, NGS-0203974, ACI-0085969, EIA-9972853 and CCR-0049068.

I thank my parents, who always believe in me, for their support and encouragements.

List of Figures

ABSTRACT

This thesis focuses on two important aspects of runtime support for parallel and distributed applications. First, we present the design and implementation of Clam, a runtime system which provides one-sided communication with the support for global name-space, object migration, and transparent routing of messages to objects. Second, we perform a comparative study of techniques used for managing location information of mobile objects within Clam. The performance evaluation of the runtime system justifies the design decisions we have made and shows the advantages of the implementation over similar libraries. The study of location management reveals that for some distributed applications an intelligent choice of location management policy is a crucial contributing factor to the application performance.

LOCATION MANAGEMENT IN A MOBILE OBJECT RUNTIME ENVIRONMENT

Chapter 1

Introduction

In this thesis we focus on software runtime support system for asynchronous adaptive and irregular applications, like Adaptive Mesh Generation and Refinement (AMR). Specifically, we describe an efficient implementation of one-sided communication and global address space, which is crucial for these applications. Existing runtime environments either do not have these capabilities or have a number of caveats, like poor portability, that complicate their wide-spread usage.

There are two major contributions of this thesis. First, we design and implement an efficient portable runtime system that addresses the computation and communication requirements of applications like parallel AMR. Our approach is based on careful balance of three important issues: correctness, performance, and ease-of-use. The preliminary results show that the implementation is portable, easy to use, and introduces low overheads over the underlying low-level communication substrate. Second, we present an evaluation of the location management mechanisms implemented within the runtime system. To the best of our knowledge, location management has not been studied previously in this context. Our results indicate that location management is critical for certain applications and thus must be carefully considered during the application design.

The runtime system we present here provides customized runtime support for asynchronous adaptive and irregular applications. The computations in such applications can be tightly or partially coupled, or decoupled. The level of coupling is determined by the intensity of the communication and by the level of dependency between communicating tasks. Communication intensive applications generate large amounts of messages in short time periods. For AMR applications this means from tens to hundreds of thousands of messages per second. Some applications can postpone processing of the incoming messages without delaying the computation within a task. Others have to wait for incoming communication, or suspend until the previously posted communication is acknowledged by the partner task. We call computations tightly coupled if they have strong dependencies (i.e., require synchronous communication) and are communication intensive. If the communication is not intensive, but weak dependencies are present (i.e., communication can be asynchronous), the computation is called partially (or loosely) coupled. Applications which do not have any communication and/or dependencies are called decoupled.

The Portable Runtime Environment for Mobile Applications (PREMA) is a framework created to support the development of AMR-like applications. The Communication Layer for Asynchronous Mobile Computations (Clam) we describe in this thesis serves as a communication component of the PREMA framework and is superior to the previously used implementation, as we show in our performance evaluation.

The problem of location management within Clam emerges from the adaptivity of the applications we aim to support. Because of this adaptivity, dynamic load-balancing is critical. Mobile object abstraction is provided by Clam for the purposes of balancing work-units among the processors. The data dependencies often present in an application require mechanisms to “communicate” with the non-local work-units (objects). A Location Management Policy (LMP) implementation enables this communication in the context of object migration, or mobility.

In this thesis we survey existing approaches for location management. We identify a set of diverse LMPs, describe their strengths and weaknesses, perform an experimental study to evaluate their properties and their impact on the performance of the selected benchmark applications.

The rest of the thesis is structured as follows. In Chapter 2 we concentrate on the details of design and implementation of Clam. Chapter 3 introduces the problem of location management, surveys existing approaches in related areas, and describes in detail the LMPs we selected for the evaluation. Chapter 4 presents our performance data from the evaluation of the runtime system and the selected LMPs. We conclude with the summary of contributions and directions for future work in Chapter 5.

Chapter 2

Runtime System

In this chapter we describe the design and implementation of the light Communication Layer for Asynchronous Mobile Computations (Clam)[1]. Our design and implementation are based on the balance of three important issues: correctness, performance, and ease-of-use. Our preliminary experience with Clam as a component of the Portable Runtime Environment for Mobile Applications (PREMA) shows an improvement in the overall quality of PREMA software in terms of portability, performance, and effectiveness.

2.1 Functionality

The computations associated with parallel adaptive and irregular applications, like mesh generation and refinement, are either tightly coupled or partially coupled. Computation and communication patterns for these applications are variable and unpredictable. One-sided communication paradigm substantially simplifies code development and maintainability for such applications. Clam supports one-sided communication in the context of data/object migration. Its functionality can be grouped as follows:

[1]The name, Clam, reflects the desired features of the implementation: it should be small, strong and viable.

- **remote memory operations**: *put* and *get*;
- **remote service request (RSR)**: invocation of an application-defined function on a remote processor;
- **mobile object functionality**: creation, migration, and messaging for mobile objects.

Each processor is assigned a unique identifier. A user-defined set of functions, which should correspond to the predefined prototypes, is registered with the runtime system. These functions are called *handlers*. A handler can be invoked on a remote processor using the Clam API. There are four types of handlers: (1) memory operation, (2) RSR (fixed number of arguments), (3) RSRN (takes buffer as an argument) and (4) *mobile object* message handlers.

The targeted applications require efficient asynchronous communication support. Communication primitives provided by Clam are non-blocking. Clam can communicate directly using user buffers and calling user-specified callback function when the communication is complete. If no callback function is provided, Clam will copy user data into a new buffer, so that the buffer can be reused upon the function return.

The mobile object functionality provides support for application adaptivity. Load-balancing is crucial in AMR applications. The application workload cannot be statically distributed because it is changing throughout the execution. Thus, work-units should *migrate* among the processors. The workload local to a processor can be represented by the set of data objects in the memory of that processor (this is a particularly useful abstraction for AMR applications). During load-balancing, local work-units (objects) can migrate to the address space of remote processors. However, the computation is not decoupled in the general case: there may be dependencies between work-units located on different processors. Hence the requirement for communication support in the context of mobile objects.

Applications which use Clam can associate a *mobile pointer* with any data object local to the processor's memory. This procedure makes that object mobile in the context of Clam, and thus in the context of the application. Given a mobile pointer to an object, the application can send a *message* to that object. When the application sends a message to a mobile pointer, it specifies the target mobile pointer, the message handler, and the arguments to be passed to the handler. The runtime system is responsible for delivery of this message to the object. A message will result in an invocation of the message handler on the processor, where the recipient object is currently located. When an object has to be migrate from one processor to another, it should be *uninstalled* using Clam API and after migration *installed* at the new processor.

Clam is using single-threaded execution model. A separate thread may be used for communication purposes (although no additional threads are used in the current implementation). The application should explicitly call *poll* function of Clam in order for pending handlers to be executed; handlers are executed synchronously.

In addition to the described functionality, Clam provides barrier synchronization and quiescence detection primitives. Quiescence detection implements Safra's termination detection algorithm described by Dijkstra in [24]. The system is quiescent when no handlers are awaiting execution and there is no pending communication. Such functionality proved to be essential for application-level termination detection. Efficient termination detection is crucial for asynchronous application and for parallel mesh generation in particular.

2.2 Design Considerations

There are three important issues to be considered in the design of a runtime system: correctness, performance and ease-of-use. The design of Clam attempts to balance these aspects.

The importance of correctness is specifically emphasized because of the one-sided nature of communication. In binary send/receive communication the application is responsible for avoiding communication deadlocks. One-sided communication is usually implemented on top of binary send/receive provided by the operating system. Thus, the burden of deadlock prevention is the responsibility of the runtime system. The one-sided communication functionality of Clam is similar to this of Active Messages (AM) [48]. However, the limitations imposed by AM are too strict for the AMR applications. Clam relaxes these requirements: the only limitation for user handlers is that polling cannot be performed within the handler. The effect is twofold. The relaxing of the model gives application developers more flexibility. At the same time, it introduces the possibility for deadlock: unrestricted communication can eventually lead to memory exhaustion, which cannot be prevented by the runtime system.

Performance of the runtime system is determined by a number of components. First, the overheads introduced by the runtime system over the underlying communication should be small. Second, the use of the runtime system should not diminish the scalability of the application. Finally, the runtime system should not restrict the capability of application to use otherwise available system functionality. Minimum number of intermediate layers within the system results in fewer memory copies and faster message processing. Although it is appealing to implement mobile object messaging on top of the Remote Service Request functionality, this would inevitably lead to additional overheads. The mobile object message functionality is implemented on the same level as RSR in Clam. Our performance evaluation shows the advantages of this design decision.

The Clam design addresses the ease-of-use requirement by defining small, nonetheless powerful, API. Discussions with applications developers made it possible to identify the core functionality required from the runtime system, define its clear semantics and avoid redundancy.

Portability of the runtime system is yet another design concern. Clam as a component of PREMA is designed to be highly portable and interoperable with existing systems. The primary interoperability problem we encountered is concerned with the MPI implementation (here and throughout this thesis, the MPI implementation used in Clam and discussed in the examples is LAM MPI [2]). There are two main reasons why MPI interoperability is important:

- the functionality provided by MPI cannot and should not be duplicated in Clam;
- applications which already use MPI may require support from Clam too.

The first implementation of Clam was based on MPI point-to-point communication, because of wide popularity and portability of MPI. The interoperability problems arise from the fact, that in LAM MPI there is only one TCP communication channel which handles interprocessor point-to-point communication. The MPI standard [3] suggests that MPI implementation may use multiple channels. However, no efficient publicly available stable MPI implementation exist which would have this feature. Myrinet and out-of-band UDP LAM MPI implementations do not have single channel limitation [1], but Myrinet is not

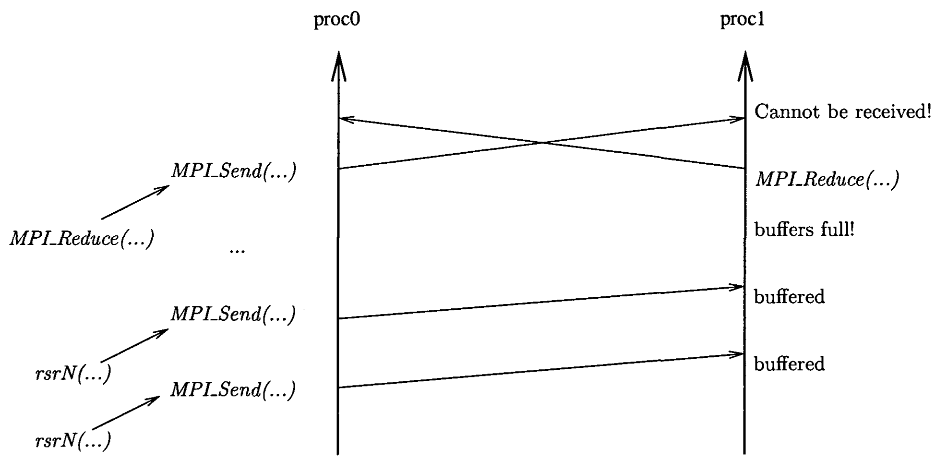


Figure 2.1: Potential deadlock induced by buffering.

widely available, and out-of-band UDP is very slow and designed primarily for debugging purposes. The lack for multi-channel TCP support within MPI can lead to problems with buffering in a runtime library which uses MPI for communication.

Figure 2.1 depicts one of the possible deadlock scenarios in a runtime system which uses MPI for communication. *proc0* issues a series of one-sided communication operations, which eventually result in *MPI_Send*s. *proc1*, however, does not issue poll operation, and the posted sends are buffered by MPI or by the operating system. The MPI collective operation, *MPI_Reduce*, invoked later requires communication over the TCP connection which was used previously by *MPI_Send*s. The buffer space available in LAM MPI may not suffice at that point to buffer all pending *sends* initiated by the runtime system and receive *MPI_Reduce send*. Hence, communication required by *MPI_Reduce* cannot complete. The communication channels can be freed only after matching *receive*s are posted for the buffered *send*s. This can be done only during polling, since we consider single-threaded implementation.

The Clam design addresses the issue of interoperability in two ways. First, the Clam API includes functionality, which allows application to determine the completion of all pending communication. Second, the design is taking into account the communication layer portability. We define the Abstract Communication Interface (ACI) as a component of Clam. ACI is the only part of Clam which interacts directly with OS-provided communication primitives (MPI, TCP etc). The ACI API provides a small set of operations to enable posting of communication requests. It is sufficient to re-implement ACI in order to port Clam on a new communication substrate. Any implementation of the ACI which does not use MPI performs all communication via non-MPI communication channels, and thus eliminates the possibility of the previously described buffering problem.

2.3 Implementation

The architecture of Clam is presented in figure 2.2. Clam is implemented as a set of modules. Some of the architecture components are tightly incorporated within the system. Other components are interfaced through a set of functions so that they can be easily substituted.

Clam is implemented in C. This decision has been made for better portability and to achieve better performance. C++ lacks portability because of differences in implementation of the language and STL across different vendors and platforms. It is also quite problematic to use a runtime system written in C++ with applications implemented in C. Clam data structures (list and hashtable) are based on implementations from Linux kernel [18]. These data structures are used by ACI implementation and in the main module of Clam.

The memory manager is another shared component of the system. It allows for strict

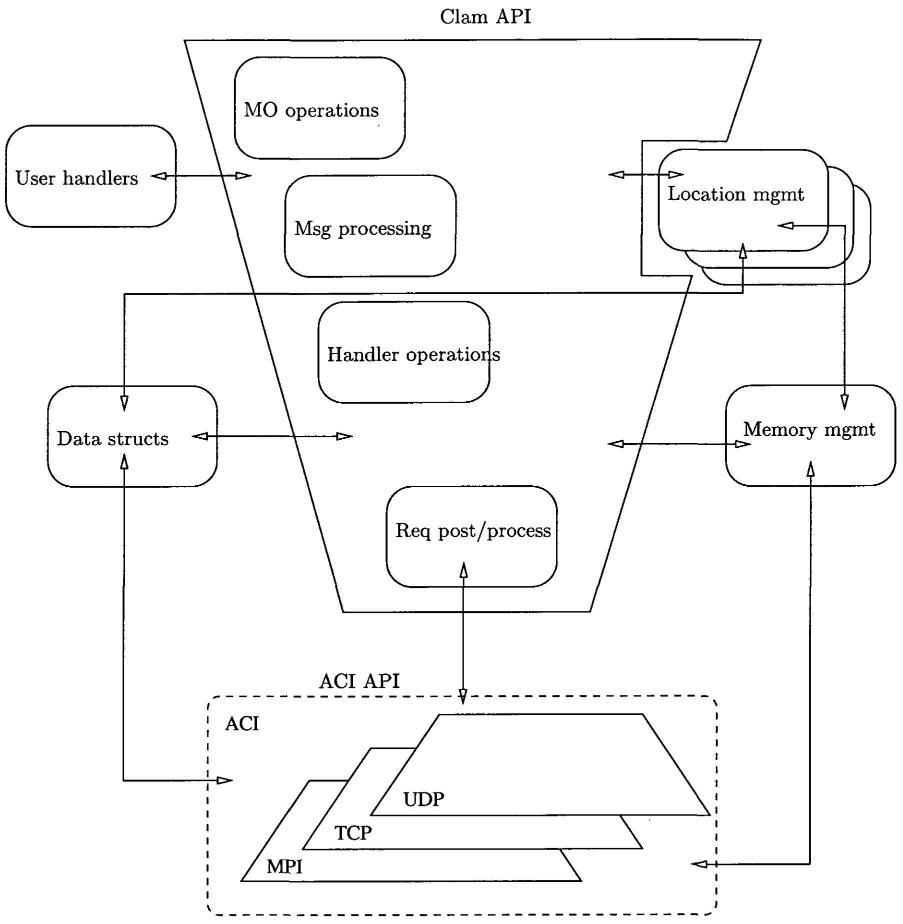


Figure 2.2: The architecture of Clam.

checking of memory operations in the debug mode and enables caching of frequently used data structures (greatly simplified version of slab caching [17]). Uncoordinated memory management complicates debugging and does not improve performance. Clam memory manager significantly simplifies the process of development and gives slight performance gains, which are to be evaluated later.

The communication-dependent part of Clam is hidden within the ACI implementation. Again, this addresses the issue of Clam portability. The ACI API can be implemented with virtually any subsystem which provides point-to-point communication primitives. All communication operations result in posting asynchronous communication requests to the ACI. The communication is initiated by passing processor ID, communication buffer, and request status object to the ACI. When the requested communication operation is complete, the status object is updated to reflect the completion. The two available tested implementations of ACI are based on MPI and TCP. In the current implementation, TCP ACI module is still using MPI for startup, processor ranking, and internal TCP channel setup. These procedures can be implemented without MPI. This has not been done in Clam mostly because of the convenient startup and termination functionality provided by LAM MPI.

There are additional benefits from having clear separation of communication-specific part of Clam. Some of the applications from the targeted domain communicate large amounts of small messages. Such applications can possibly achieve better network utilization communicating through UDP instead of TCP. With the separation of ACI, such implementation has become possible. However, the complexity of UDP ACI implementation is much more sophisticated than TCP ACI. Also, experience of developers in the area shows, that very slight performance improvements of communication over UDP are not justified by the implementation complexity [6, 7]. The implementation of UDP ACI is left as future work.

The functions of the main Clam module provide support for:

- *handler registration*: mechanisms for address-independent cross-processor handler invocation;
- *mobile object operations*: functions for creation, migration and processing of messages for mobile objects, FIFO ordering of messages;

- *synchronization*: guarantees that only one thread is inside Clam at a time, and provides mobile object lock during execution of a handler directed to that object (while locked mobile object cannot migrate);
- *ACI request management*: maintains queues of incomplete ACI requests and processes completed requests.

The mobile pointer functionality has been discussed earlier. Such functionality requires support for transparent location-independent message handler invocation. A specific module of Clam, the Location Management Module, is performing this function. The location management module implements a *Location Management Policy* (LMP), an algorithm, which provides location-independent message routing. Location management is the central point of research for the second part of this thesis. It was very important to make this module highly "pluggable". The location management module is interfacing Clam through the set of functions and can be easily swapped (this idea is similar to the way specific filesystem is implemented within the Linux kernel VFS [18], but in the current Clam implementation the location management module cannot be changed at runtime).

2.4 Related Work

Clam has been designed and implemented to serve as a new communication layer for the PREMA framework. In this section, the previously used implementation of the communication substrate within PREMA is discussed. For the comprehensive survey of related work in the context of other runtime systems the reader is referred to [13].

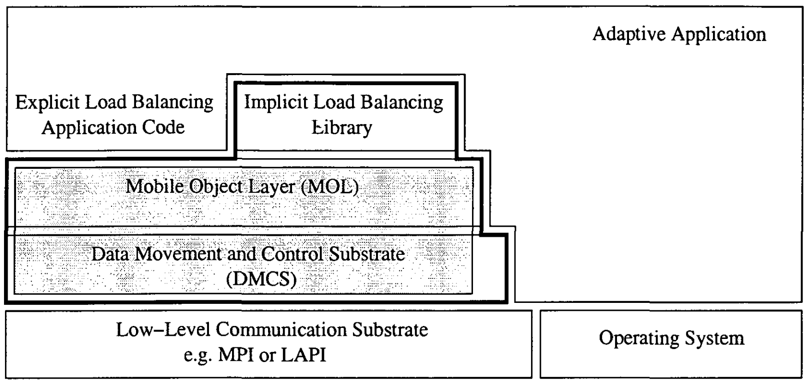

Adaptive Application
Explicit Load Balancing Application Code
Implicit Load Balancing Library
Mobile Object Layer (MOL)
Data Movement and Control Substrate (DMCS)
Low–Level Communication Substrate e.g. MPI or LAPI
Operating System

Figure 2.3: The initial architecture of PREMA.

The PREMA programming model employs SPMD approach, similar to conventional MPI-1 [3] applications. The initial architecture of PREMA is shown in figure 2.3. The framework included three layers. The Data Movement and Control Substrate (DMCS) was interfacing the low-level communication primitives (MPI or LAPI) and provided one-sided communication and RSR functionality. DMCS is described in detail in [15]. The Mobile Object Layer (MOL) [22] was implemented on top of the DMCS API. MOL provided global address space for the *mobile object* abstraction (object mobility was previously implemented in Smalltalk [16] and Emerald [31]). Mobile object functionality serves as the basis for the Implicit Load Balancing Library (ILB), which uses mobile object abstraction to implement *schedulable objects* (SO). A SO is the smallest unit of granularity managed by the ILB. Balancing workload among the processors is done by associating SOs with the workload units and migrating them among the processors.

It has been shown, that the functionality provided by PREMA helps in achieving good application performance [12]. It is also the case that application development is greatly simplified when PREMA is used. However, we discovered a number of problems in the initial design and implementation of PREMA. Most of the problems were identified from our experience with applications, using the framework depicted in figure 2.3.

With Clam we addressed the problems discovered within the DMCS/MOL implementation:

- lack of true interoperability with MPI (Clam: communication quiescence procedure added, TCP-based implementation available);
- complicated and overloaded API (Clam: API reduced about two times, duplication of DMCS functionality within MOL eliminated, simplified semantics of the API);
- "separation of concerns" between DMCS and MOL (Clam: monolithic design);
- use of C++ reduces portability and interoperability with applications (Clam: implemented in C);
- poor portability due to tight dependency on MPI (Clam: clear separation of communication module; implementations based both on TCP and MPI are available).

Clam has become a new communication layer of the PREMA framework. The ILB module and existing applications, which use it, have been ported on Clam in three days. This fact supports the for high ease-of-use and stability of Clam. Performance results presented in section 4.2 show, that Clam has a number of advantages over the DMCS/MOL implementation.

2.5 Discussion and Future Work

Despite the obvious benefits of the Clam design and implementation, it cannot be considered the last step in the evolution of the communication subsystem of PREMA. A number of outstanding issues still have to be studied. First, experimental data show, that for some application configurations, Clam adds higher overhead compared to DMCS/MOL (about 10% more). It is the case, that DMCS communication uses blocking *MPI_Send* calls for communication. For small messages (less than 64Kb) blocking communication achieves the best performance [1]. However, it is known that blocking on communication can lead to a deadlock [48]. It remains to be seen whether this difference in communication mechanisms is the cause of the observed overhead. A thorough profiling has to be completed.

Current Clam implementation does not use additional threads for communication purposes. Advantages of adding such threads to the implementation have to be considered.

The experience with the ILB showed, that it is absolutely necessary for load-balancer to be able to receive load-balancing utility messages independently and concurrently with the application execution [12]. In DMCS/MOL, and currently within Clam, this is achieved by synchronizing the runtime system and performing periodic poll in a separate load-balancer thread. Load-balancing messages are distinguished from the application messages using *tags*. However, this turned out to be an unsatisfactory solution for applications which generate high network traffic. Congestion of the single processor-to-processor communication channel leads to late arrival of load-balancer messages and thus poor load-balancing decisions. In future, we plan to address this issue by providing an API for creation of additional communication networks (not possible in MPI ACI, but feasible in TCP). This will eliminate the channel congestion problem and the existing requirement for message tags.

Garbage collection of destroyed and unused mobile pointers has not been addressed in the current implementation. The reason for this is that the implementation of distributed garbage collection would add a lot to the complexity of the code and to the runtime overheads (most of the distributed garbage collection algorithms require additional communication [41]). For the existing AMR applications which use PREMA, there is no requirement for dynamic object destruction. Distributed garbage collection in Clam is left as a future work.

Finally, Clam is an open-source project. We plan to prepare release of the source code and make this runtime system available to the community in the near future.

Chapter 3

Location Management Policies

The problem of location management (LM) is relevant in parallel and distributed systems, where objects dynamically relocate. Techniques for managing location, i.e., Location Management Policies (LMP), describe the rules which are used to find objects and the actions to be taken when objects migrate to the new locations in the network.

A LMP should provide efficient implementations for *move* and *find* operations on objects. Efficiency in the context of LMP is defined in terms of (1) communication, (2) computation overheads, and (3) response time. However, for the same LMP, it is possible that optimizations of one operation will deteriorate the performance of another and vise versa. This is illustrated by the following two extreme strategies, described in [11]. The *"full-information"* strategy requires up-to-date information about all objects for efficient *find* operations, but then the cost for performing *move* is high (all nodes have to be updated). On the other hand the *"no-information"* strategy does not require location updates. Consequently, the *find* operation is very expensive, its cost is almost equivalent to a global search.

In this thesis we evaluate the impact of location management policies on performance of parallel and distributed applications that require object migration. Specifically, we evaluate a number of diverse location management policies within Clam, the runtime system presented in Chapter 2. The LMPs we evaluate combine existing experience of location management in Parallel Distributed Computations (PDC), mobile communication networks, and mobile agents computing. This evaluation is the first (to the best of our knowledge) comprehensive evaluation of LMPs in PDC. One of the objectives of this study is to classify existing location management approaches in terms of their impact on the overall performance of parallel and distributed computation applications.

In this chapter we first overview location management in parallel and distributed computations, mobile communication networks, and mobile agents computing. We do not overview location management in the areas, which have significant differences in the model or problem statement compared to location management in Clam (e.g., Distributed Shared Memory concerns with replication and consistency models [33]; in Peer-to-Peer systems assignment of objects is almost static and the search procedure uses different assumptions [38]). Next we describe location management in Clam, the design choices for LMP development, and the policies we have implemented and evaluated.

3.1 Overview

3.1.1 Parallel Distributed Computations

We study location management in the context of the runtime system described in the previous chapter. The Clam mobile object model assumes that mobile objects represent user-defined objects, which correspond to user data. These mobile objects are distributed among the processors by the ILB library to balance the processors' workload. The task for

Clam is to provide efficient routing of messages sent to mobile objects.

Our survey of the related work on location management within those runtime systems that provide support for object mobility showed, that most of such systems use the same LM technique. The *forwarding addresses* technique was first used in DEMOS/MP operating systems in the context of process migration [42]. Later it was extended and evaluated by Fowler in [27]. In forwarding technique each time an object migrates, it leaves a pointer to the new location. Messages sent to the object follow the trail of pointers to reach the object. A number of modifications described later in this chapter allow to keep the forwarding chain short.

Following are some of the systems, which employ forwarding technique for location management: Emerald [31] (object-oriented system with fine-grained object mobility support); Thor [37] (implements object-oriented database management system); Amber [20] (provides simplified model for multiprocessor applications). SSP chains [44] (distributed technique for garbage-collection), MOL [22] (mobile object functionality for load-balancing), Charm++ [36] (framework for dynamic load-balancing).

Another technique for mobile object location management is the centralized directory. It is used in ABC++ which extends object-oriented features of C++ and provides object migration support using centralized location database [10]. Few other systems use uncommon methods for location management. These systems are usually developed for specific applications and and hence are not universal. For example the *arrow* protocol implemented in the Aleph toolkit [28] supports exclusive access to objects using directory based on spanning tree.

Following are our assumptions about the PDC model relevant to the development and evaluation of the LMPs within this thesis:

- number of nodes involved in the computation is in the order of hundred;
- overdecomposition of the problem: number of objects is in the order of thousand;
- fixed resource allocation; resources are non-faulty;
- all-to-all overlay network;
- possible geographical distribution of the computational resources (network partitioning);
- object migration and communication patterns are unpredictable in the general case.

3.1.2 Mobile Communication Networks

During the last years we have been observing constant development of wireless and cellular communication technologies. Different kinds of Public Land Mobile Networks are becoming more and more ubiquitous. Cellular phones, palm-top computers, laptops with wireless network cards, i.e., *mobile terminals* (MT), are not attached to a single stable physical location, but roam around. This creates the need for special techniques to handle such movement in order to guarantee communication between MTs. In this section we describe standard location management procedures in cellular communication networks and survey some of the proposed modifications.

Typical structure of a network infrastructure supporting cellular wireless communication is depicted on figure 3.1. The geographical area is divided into location areas (LA) [8]. Each LA can contain one or more *cells*. A mobile support station (MSS) is assigned to

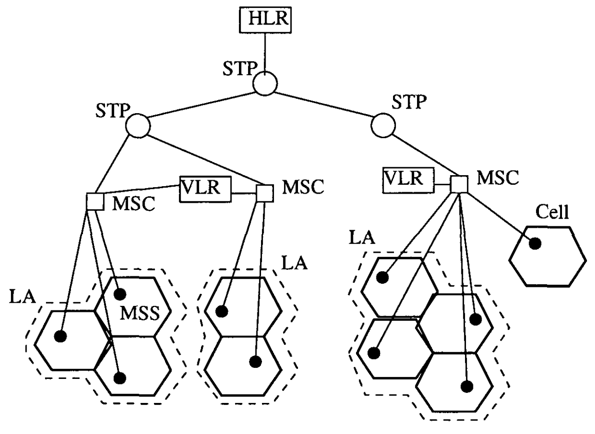


Figure 3.1: Example architecture of a mobile communication network.

every cell to handle all network traffic directed from a MT located within the cell. Up in the hierarchy a *mobile switching center* (MSC) governs one or more LAs and maintains a database with MTs locations. Multiple MSCs are connected together by a fixed backbone and/or intelligent network through a number of *signal transfer points* (STPs).

In cellular networks mobile users are tracked using *two-tier scheme* [8, 40] (as defined in IS-41 [25] and GSM [26] standards). A location database, called *Home Location Register* (HLR), is predefined for each MT. Another database, a *visitor location register* (VLR), is associated with one or more LAs (see figure 3.1). Two procedures, governed by the standard, define what happens when MT moves from one LA to a different one, and how a call recipient can be found. Following is the brief description of location procedures as implemented in the current mobile networks.

There are two possible scenarios when a MT moves from one coverage area to another. If the new area shares the local database (VLR) with the original one, that VLR is simply updated with the new location of a user. If VLRs are different, home registry has to be updated with the user's new location. The MT requests to remove its record from the old VLR and registers with the new one.

If there is a need to locate a particular MT when another MT makes a call from some cell, the request is first sent to the local VLR of the caller. No further actions are required if that VLR possesses information about the recipient's location. Otherwise, a query is propagated to the callee's HLR. The up-to-date location information (which HLR always has) is sent back to the caller's support station. The support station covers the whole location area consisting of multiple cells. The actual cell where the recipient is located is determined by polling, or *paging*, within the LA. The search request is broadcast to all cells of the LA, and the recipient reports its location cell upon receiving this request. At that point connection between the two MTs is finally established.

Most of the research about location management in Mobile Communication Networks has been concerned with the costs of updating the HLR. Some studies were trying to keep and improve the centralized nature of the scheme, while others were attempting to distribute the process of location. Interestingly enough, all of the described techniques are just proposals. They have been evaluated using theoretical analysis, simulations and traces, but none is a part of the existing standards.

The nature of mobile network communication is usually unpredictable, but the infrastructure should support any particular pattern [49]. Schemes that can adapt to the communication and migration characteristics of MTs are advantageous. In [30] Jain proposes to keep a cache at each VLR. When a home database is queried for a specific MT, the response is stored locally, so that the subsequent call to the same MT may not require communication with the HLR. It has been shown, that if CMR is high, caching performs very well.

Another proposed improvement is based on user *profile replication*. A *profile* represents the set of mobile users, whose location information is always kept up-to-date at the local VLR. This enables quick location of the most "popular" users. Different approaches to profile replication are discussed in [43, 46].

Forwarding technique eliminates the update operation by keeping a pointer to the new location of a migrated MT at the source LA VLR [29]. When a request to locate that MT arrives, it will be forwarded to that new location. Forwarding techniques decrease the load on HLR, but have high overheads if forwarding chains become long. The study described in [29] shows, that if Call-to-Mobility Ratio (CMR, the number of calls issued to the user over the number of times it changes location) is lower than 0.5 and forwarding chains are at most 5 hops long, forwarding reduces user location costs network overheads by 20-60%.

A conceptually different approach uses *distributed database architecture* [8] instead of a centralized HLR. This technique takes advantage of the fact, that in most cases backbone/intelligent network architecture has hierarchical tree structure. This allows to distribute the load of location management among the non-leaf nodes of the tree. Different hierarchical approaches are described in [34, 40].

Partitioning of the coverage area into zones, among which MT moves infrequently, is yet another modification which reduces the number of LSs and query time for certain call/migration patterns. A partition consists of location areas, which are represented by the dedicated location server. That *representative* LS is not aware of the exact MT location, but knows its current partition. This technique reduces update-induced communication.

Summarizing, hierarchical location schemes eliminate the need for centralized HLR at the cost of increased general complexity of location management and increased storage requirements at the intermediate LSs. Hierarchical techniques support locality of communication and migration of MTs.

Compared with LM in PDC, mobile network systems have a number of distinct properties. In PDC a local location directory is associated with each processor. This directory is analogous to VLR in combination with supporting stations, which communicate with MTs local to a location area. However, in PDC location directory is always aware of all the objects local to the process address space. This eliminates requirement for paging. In PDC applications, similarly to MNC, communication and migration patterns are not predictable in general case. At the same time, in MNC the migration options for a MT are limited by neighboring areas, while in PDC object can migrate to any of the processors regardless of their geographical location. Of course, PDC application do not include unpredictability of human character, present in cellular phone networks. Another difference of PDC is the time required for a mobile object to change its location. In MNC there are strict limitations on maximum travel speed for cellular phone users; the sizes of communication cells are also predefined [39]. PDC applications can possibly move hundreds of objects in few seconds between geographically distant locations.

There are also differences in the system architecture. Mobile networks in most cases have hierarchical structure. There can be dedicated location servers on non-leaf nodes of the hierarchy. In PDC applications computation is done either on a COW, or a collection of clusters. All nodes have equal functions, and the application can rarely take advantage of the underlying network routing, as it is handled by the low-level protocols.

The PDC model assumes fine-grained object mobility. The number of objects may be large. We argue though, that the number of objects in mobile communication networks, e.g., in cellular networks, is generally much higher (hundreds of millions). This puts certain size and memory limitations on MNC LM algorithms.

3.1.3 Mobile Agents Computing

Mobile agent computing is a relatively new area of distributed computing, which is gaining more popularity with the development and growth of the Internet. Mobile agent (MA) is an independent piece of code and data. It can be taken from the execution context on one host, migrated to a different machine, and continue execution there after migration completes. Recent progress in developing platform-independent environments (e.g., Java Virtual Machine) addressed many technical difficulties, inherent to the implementation of mobile agents, which also contributed to the growing popularity of the model. The spectrum of applications, which can take advantage of mobile agents, includes e-commerce, distributed collaboration environments, information search and dissemination, network management and monitoring [35]. Some of the MA applications require support for communication between the agents [19]. In such cases location management techniques play very important role [49]. Nevertheless, being important, location management is not the major research issue in mobile agent computing: the main challenges in MA systems are support for mobility, security, naming services and fault-tolerance [32].

Home server location algorithm associates a specific host with each mobile agent. Every time a mobile agent changes its location, home server is updated with the new location. A message addressed to the mobile agent is sent to its home server, which forwards that message to the agent. Ajanta [47] implements two-tier home-based location management scheme, similar to the one used in cellular communication. A number of modifications of the centralized scheme exist [49]. For example, the server can be queried for the current location of the destination agent. The answer to the query can then be used to send the message directly. This scheme is called *query server*. It is also possible to have a single dedicated host, *central forwarding server*, to keep location information about *all* agents.

The disadvantages of the centralized scheme are high load on the database host and its network links, possibly high storage requirements, single point of failure, poor scalability. Nevertheless, it gives good average for the number of hops a message has to travel to reach the object. In applications with the small number of agents or low communication rate this approach can be beneficial.

The *forwarding* technique can also be used for locating mobile agents. Voyager platform [5] uses forwarding pointers in combination with home server for agent location. Problems with forwarding pointers include possibly long chains of forwarders and low resistance to failure.

Home-server and forwarding are the most popular location management methods in mobile agent computing. Alouf et al [9] compared these two schemes. Markovian analysis and experimental results show that centralized server performs better than forwarders on a LAN, but not on a wide-area network. Other approaches to locating agents in MA systems include *broadcast* and *hierarchy* of the location servers.

Cao et al summarize experience with mobile agents location management by introducing the concept of *mailbox* in [19]. The authors generalize most of the existing approaches and introduce new classes of location protocols. A mailbox which buffers incoming messages is associated with each mobile agent. The mailbox serves as a mediator in communication

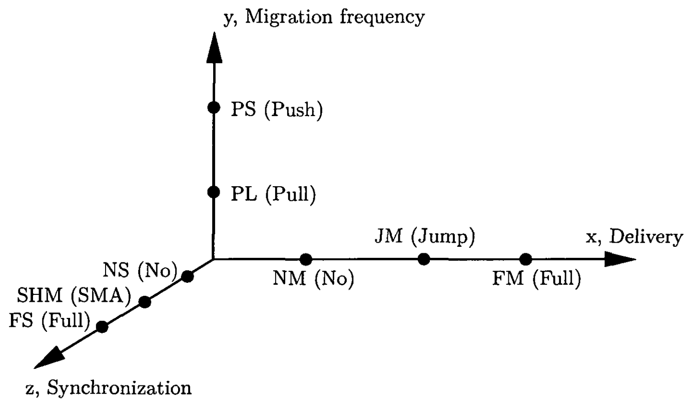


Figure 3.2: Design space for MA communication protocols (Cao et al, [19]).

with the mobile agent. Logically, the mailbox is a part of the mobile agent, but it can be detached from the agent. Therefore, it is not required that the agent and its mailbox are located on the same host. Using this idea, classification of the inter-agent communication algorithms is done along the three dimensions, as shown in figure 3.2.

Dimension x defines the frequency of mailbox migration. Decisions about migrating the mailbox are made dynamically with Jump Migration, and with Full Migration mailbox always migrates together with the agent. Axis y determines the agent–mailbox interaction. Either mailbox is responsible for forwarding incoming messages to the agent (push), or the agent periodically queries its mailbox for available "mail" (pull). Finally, dimension z identifies options in synchronization in order to achieve better communication reliability. If synchronization takes place, the moving object (mobile agent or mailbox) synchronizes with the stationary object (mailbox or host) to prevent message loss during the process of migration. Synchronization can be partial (host with migrating mailbox (SHM) or migrating agent with its mailbox (SMA)) or full (combination of SHM and SMA).

Combinations of the classification parameters creates a variety of protocols. In such a taxonomy each protocol is defined by the string of format *XX–YY–ZZ* with its components chosen from each of the axes. A particular algorithm should be chosen according to the requirements of an application. *NM–PS–** class of protocols corresponds to the home-based technique. Essentially, stationary mailbox represents forwarding server. *JM–*–** and *FM–*–** generalize forwarding pointer protocols with and without forwarding chain shortcutting respectively.

Mobile agents application model has certain properties, which make it quite different from mobile network communication model and from PDC. For certain applications, migration and communication patterns of a mobile agent can be known in advance [49]. This almost never holds for mobile networks. Mobile agents are usually designed to operate on a wide area network, most likely, on the Internet. Such environments consist of millions of possible locations for an agent. Moreover, mobile agents applications are much more dynamic than, for example, cellular phone networks. The number of mobile agents for an application may vary and change during its runtime. Hundreds of agents can possibly be created and destroyed in seconds. Mobile agents do not operate on a fixed predefined set of hosts. The number of locations mobile agents operating on the Internet can visit is bounded only by the number of on-line hosts supporting the platform of the mobile agent.

One more important difference of PDC model is in the assumptions about communication. The reliability of communication provided to an application in PDC is usually implemented by the lower levels of the system. One-sided communication abstraction provided by Clam is reliable. Messages sent to mobile objects are also guaranteed to arrive in FIFO order. The MA communication may not be reliable. This is why synchronization is considered as a part of MA location management.

3.2 Location Management in Clam

A Location Management Policy (LMP) in Clam defines rules for performing three operations: *update*, *search*, and *search-update*. The *update* operation takes place when an object migrates from one processor to another. The *search* operation specifies how a message to a non-local mobile object can be delivered. The *search-update* defines the procedure of updating location information of selected processors after the message was delivered to an object.

The *location directory* is a distributed data structure managed by a LMP. It maps an object onto the possible or exact location of that object. Some techniques may use a *set* of possible locations [40], but they are outside the scope of this thesis. Such approaches can be beneficial in applications where object migration is localized within group of nodes. This is not the case in general for load-balancing. The development of customized LMPs for a specific load-balancing algorithm has not been addressed in this work.

The location management design space is shown in figure 3.3. Each axis represents a LMP operation and the corresponding options arranged in the order of increasing complexity. Migration of an object to a new location may result in one of the four possible *update* scenarios:

- no update (no communication);
- update of the local directory (no communication);
- update of location directories at selected sites;
- update of directories at all processors (broadcast).

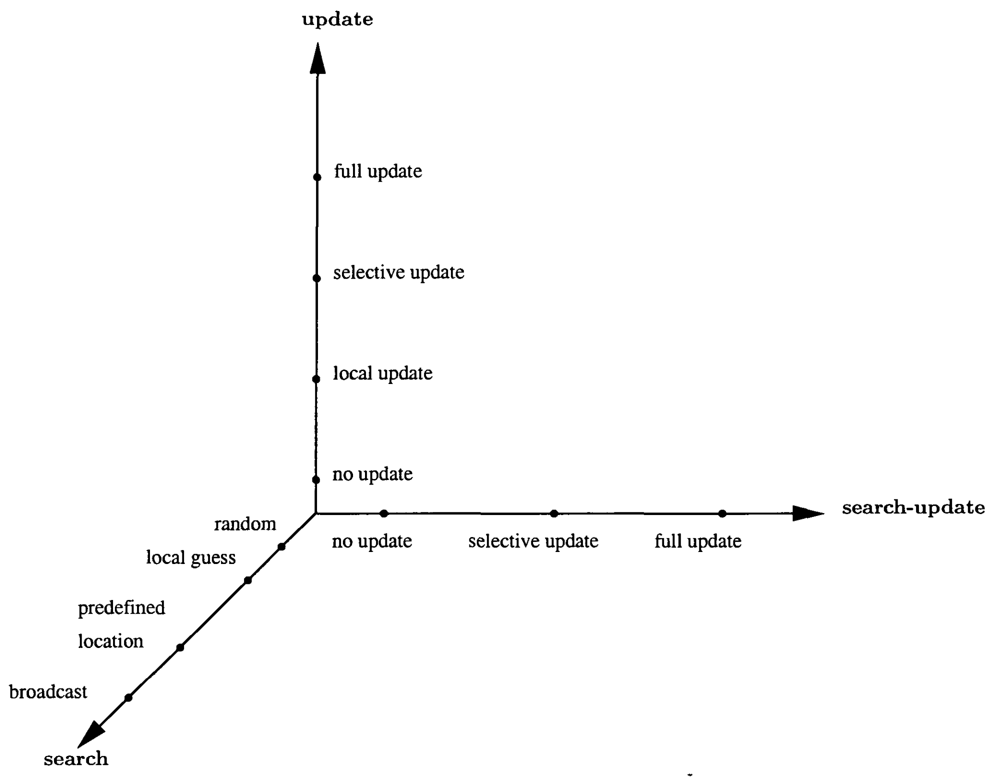


Figure 3.3: Design space for location management policies development.

It is not feasible to maintain exact location of an object at all processor directories. This would require global synchronization of the system before each object migration. We do not consider this as a possible solution because of high overheads. That is why the location information stored at a local directory can often be outdated, regardless of the update technique used. Therefore, the local directory information is nothing more than a *location guess* which was correct at some point in the past. A message directed to an object results in a point-to-point message sent to a processor, where that object can *possibly* reside. If the guess was wrong, that message will trigger another point-to-point message, i.e., the message will be forwarded. Thus, the *search* operation is essentially a process of routing the message toward the processor where the searched object is located. We distinguish four general options for implementing the *search* operation. They differ by the choice of the recipient(s) for the initial *search* message:

- the recipient is chosen randomly;
- the recipient is chosen using the guess from the local directory;
- the message is sent to a predefined processor;
- the message is sent to all of the processors (broadcast).

All of the *search* methods can also send queries to the corresponding locations instead of sending actual messages. This could decrease forwarding traffic in some applications, but the pending outgoing messages would have to be queued on the source processor until the reply arrives. However, when the reply does arrive, it may already be outdated, i.e., querying decreases forwarding overheads at the cost of increasing the likelihood of location information being invalid. Query-based search strategies are not evaluated in this study. Querying is likely to perform badly for applications with intensive communication and object migration while introducing significant complications into the implementation of the LMP.

Search-update is an optional procedure in a LMP. Its purpose is to reduce the length of the forwarding pointers chain. If *search-update* procedure is present in a LMP, it can update either some of the processors in the system or update all of the processors. The idea behind the *search-update* operation is similar to caching in cellular networks. It is based on the assumption, that if a message was sent to an object, it is likely another message will be sent to it again. *Search-update* attempts to reduce the cost of subsequent object *searches*.

In PDC the cost of subsequent messages to an object would be reduced if those message travel shorter path than the previously sent messages to the same object from the same processor. This improvement is achieved by updating the location directory of the initiator processor with the newer object location guess. The update procedure will result in shortening of the forwarding pointers chain. The difference from cellular networks is that in distributed computing it is possible to have a directory entry for each object in the system (unlimited cache). In MNC model this may not be feasible because of the large number of MTs. Different *search-update* strategies have been presented and evaluated in [27] by Fowler.

A location management policy implemented within Clam should:

- minimize the length of the path for forwarded messages;
- minimize additional communication;
- balance location management "duties" among the processors;
- minimize computation overheads of the LMP operations.

Different LMPs pursue different trade-offs of the listed requirements. As it is shown later in this thesis, the application performance may heavily depend on the choice of LMP based on the application properties (intensity of the object communication and migration, in particular).

Seven location management policies have been implemented within Clam for the purposes of this evaluation. The selection of policies was affected by a number of factors. First, the policies which are commonly used in PDC had to be evaluated. Second, the selected

LMP	*Update*	*Search*	*Search-update*
Lazy Forwarding (LF)	no	local guess	no
Jump Update (JU)	no	local guess	yes, selective
Path Compression (PC)	no	local guess	yes, selective
Broadcast Update (BU)	yes, all	local guess	no
Partition Update (PU)	yes, selective	local guess	no
Eager Update (EU)	yes, selective	local guess	no
Home-Based (HB)	yes, selective	home processor of the object	no

Figure 3.4: Summary of the implemented location management policies.

policies should be appropriate within the PDC model. Third, we attempted to use some of the ideas collected from surveying location management methods in the relevant areas. Following is the description of the implemented LMPs, which is also summarized in table 3.4 with respect to previously discussed LMP operations.

It is important to note, that each mobile pointer in Clam contains a processor ID where that pointer was originally created. In the absence of any information about the mobile pointer in the directory of a processor, the origin processor of a mobile pointer is used as the best location guess.

Lazy Forwarding (LF) is the simplest forwarding protocol. Messages to objects are routed following forwarding pointer addresses stored in the local directory. When an object moves, only the local directory is updated. Forwarding has low migration cost. The main disadvantage of LF LMP is that the length of forwarding chain is bounded by the number of processors only.

Jump Update (JU) is similar to LF, but when a message reaches the object, an update message is sent back to the processor from where the message is originated. This is similar to caching in mobile networks. The cost of subsequent message to the object is reduced by at least one hop.

Path Compression (PC) differs from JU that the update message is propagated to all processor in the forwarding pointers trail.

Broadcast Update (BU): the new location of an object is broadcast each time the object migrates. *Search* proceeds the same way as in LF, by forwarding.

Partitioned Update (PU): differs from BU by updating only selected locations when an object moves. The idea of partition-based location management originates from MNC, although we implemented a different algorithm not based on tree hierarchy. The motivation behind this protocol is to address the issue of slow network links. We assume the partitioning of nodes in sub-clusters. Nodes within a sub-cluster are connected with high capacity links, while inter-subcluster connections are slow. When an object moves, a new location is broadcast to all the nodes within the sub-cluster. If the new location node is in a different sub-cluster, the object location is broadcast within that sub-cluster upon object arrival. Search procedure is done using forwarding. PU LMP attempts to minimize *update* traffic over the slow connection. *Search-update* updates the source processor with the new location, if the message was forwarded. The update is broadcast within the partition of that processor.‘

Eager Update (EU) uses the idea of profile replication, briefly described in Section 3.1.2. Each time a message arrives to the object, the sender processor is added to the profile of that object. This profile accumulates information about processors "interested" in communication with the object. When an object migrates, all of the processors from the list are updated with the new location, and the list is reset. Updates in EU LMP are more "intelligent" than in BU LMP.

Home-Based (HB): each mobile pointer in Clam contains an ID of the processor where that pointer was created. We call this processor "home" of the mobile object. When a message is issued to an object and the object is not local, the message will be sent to the "home" node. The directory of the "home" node is updated with the new object location after every migration, so that it can route incoming messages to the object.

Figures 3.5-3.10 illustrate how the selected LMPs manage the distributed location directory. We consider the system consisting of five processors and one mobile object. We assume that the object has been created on processor 0. Solid arrows show the location information on each of the processors for that object. The figures depict snapshots of directories after each of the actions in the sequence: (1) object moves from 0 to 1; (2) object moves from 1 to 2; (3) processor 4 sends a message to the object; (4) object moves from 2 to 3. In these examples we do not take into account concurrency: all messaging activity from the previous step is completed before proceeding to the next step.

JU LMP is the most common technique for location management in PDC run-time systems listed in Section 3.1.1. HB LMP is also used in some of the implementations. Complexity of LF, JU and PC LMPs was studied in [27]. To the best of our knowledge, there is no mobile object run-time environment which would provide a choice of LMP to the application. We are also not aware of any work which would evaluate and compare these LMPs.

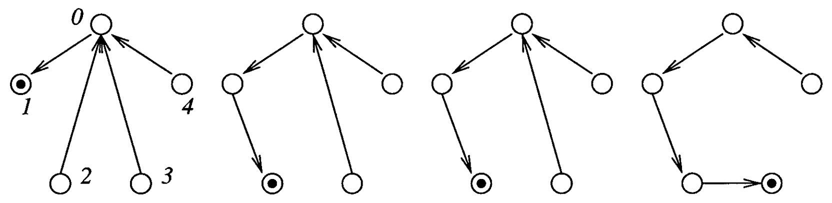


Figure 3.5: Lazy Forwarding LMP.

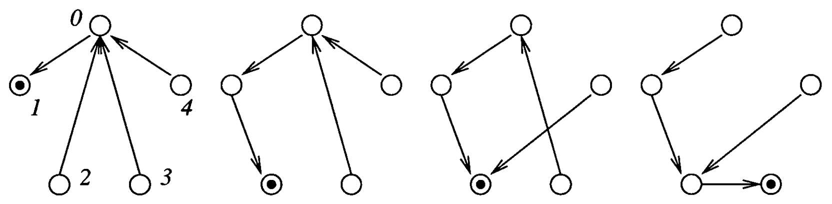


Figure 3.6: Jump Update LMP.

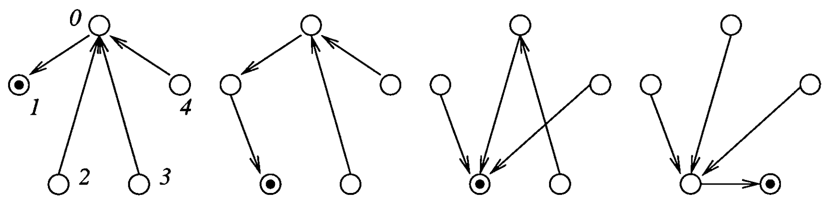


Figure 3.7: Path Compression LMP.

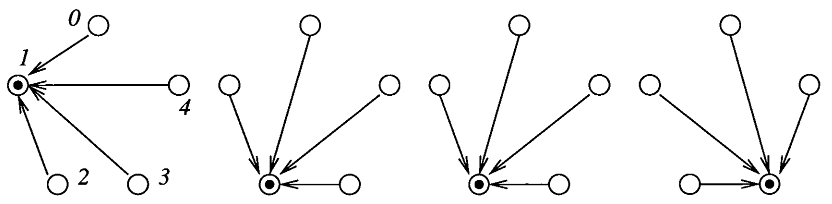


Figure 3.8: Broadcast Update LMP.

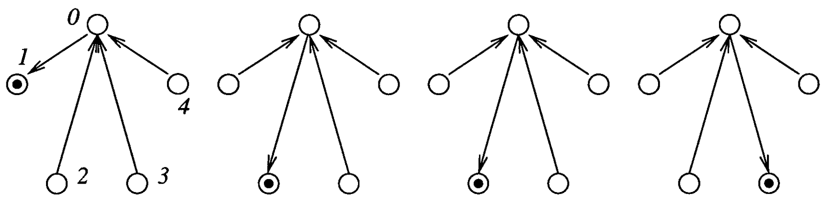


Figure 3.9: Home-Based LMP.

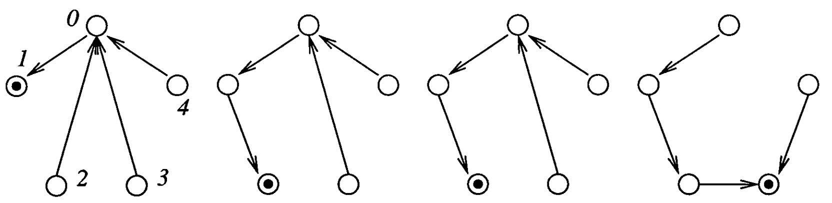


Figure 3.10: Eager Update LMP.

Chapter 4

Evaluation

This chapter is structured as follows. First we evaluate the performance of Clam in terms of absolute overheads it introduces and how it compares with the previous implementation of PREMA communication layer. The second part of the evaluation describes the series of tests which compare the performance of the selected LMPs with respect to their impact on the performance of the two benchmarks we describe in this chapter. We conclude with the analysis of the collected performance data.

4.1 Experimental Environment

4.1.1 Hardware Platforms

The primary testing environment we used in our experiments was SciClone Cluster of The College of William and Mary [4][1]. The architecture of SciClone is heterogeneous. It features different types of processor configurations and various networks (Fast Ethernet, Gigabit Ethernet, Myrinet). The detailed description of SciClone can be found in [4]. Most

[1]Computational facilities of SciClone Cluster were enabled by grants from Sun Microsystems, the National Science Foundation, and Virginia's Commonwealth Technology Research Fund.

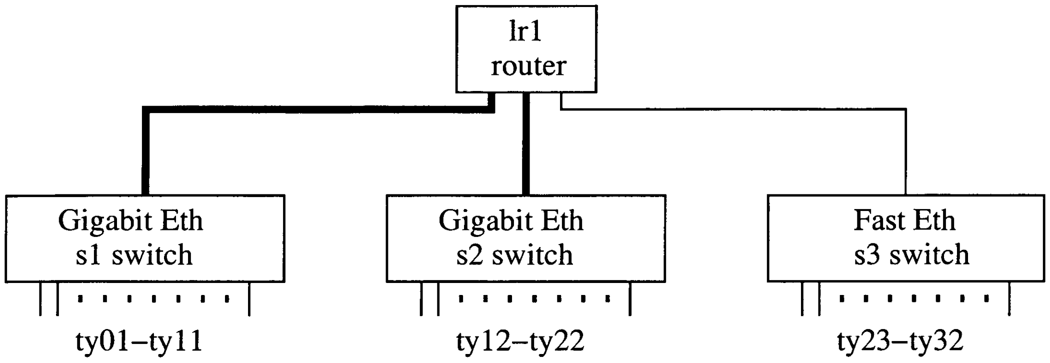


Figure 4.1: Simplified configuration of the CS Network Testbed.

of the SciClone nodes support more than one network interface. In our experiments all communication was done via Fast Ethernet interconnect.

The Computer Science Network Testbed[2] is a subsystem of switches connecting 32 lower nodes of SciClone Typhoon subcluster through a separate network interface. The testbed has been designed for experiments with various network parameters. WAN can be simulated by adjusting hardware settings on the network switches and changing parameters of the routing software. The simplified architecture of the testbed is depicted in figure 4.1. We simulated wide-area network environment by locking bandwidth of the *s1–lr1* and *s2–lr1* (see figure 4.1) links to 10 Mbps on the *s1* and *s2* switches and keeping 100 Mbps bandwidth for the communication between nodes connected to the same switch (hereon we call this configuration 10/100 configuration).

[2]The Network Testbed is designed and maintained by the group of Dr. Bruce Lowekamp at the College of William and Mary.

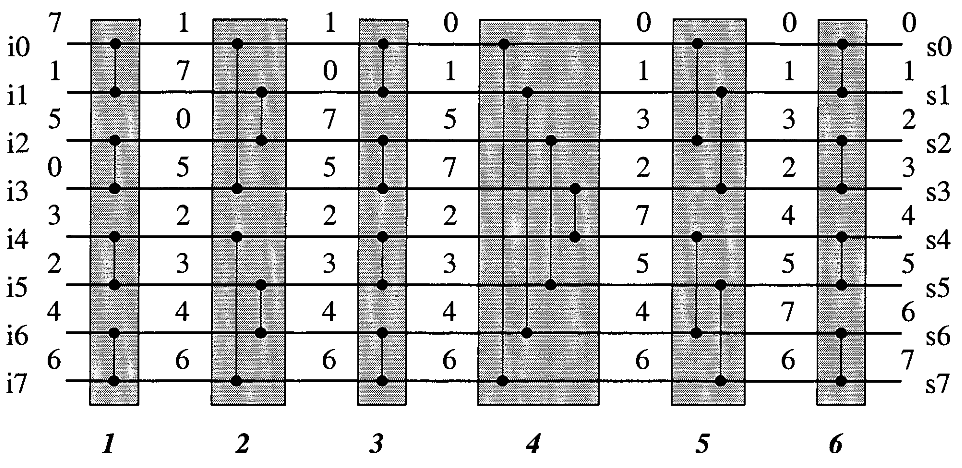


Figure 4.2: Sorting network for eight inputs.

4.1.2 Benchmarks

4.1.2.1 Synthetic Microbenchmark

Parallel network sort benchmark, which we call *netsort4* for historical reasons, implements a sorting network[3]. Sorting network is a comparison network which specifies a sequence of comparisons for its inputs to produce a sorted sequence. The details behind sorting networks are discussed in detail in [23]. The process of sorting a sequence of eight numbers using sorting network is illustrated in figure 4.2. Through a series of comparisons and exchanges, the input sequence i transforms into the sorted sequence s. For input line *i0* this results in comparison with lines 1, 3, 1, 7, 2 and 1. Each of the shaded regions corresponds to a stage. All comparisons within the stage can be done concurrently. A sorting network of such structure can sort an input sequence of n numbers in $O(\lg^2 n)$ time [23]. In the rest of this section we concentrate on the details of the *netsort* implementation.

It is important to note, that *netsort4* benchmark has been developed with the purpose

[3]The *netsort4* benchmark was originally implemented by Chris Hawblitzel.

```
typedef struct sortnode_t {
  int          id;
  int          stage;
  int          value[DEPTH+1];
  mobile_ptr_t partner[DEPTH];
  int          partner_value[DEPTH+1];
  int          partner_ready[DEPTH+1];
  int          partner_id[DEPTH];
} sortnode_t;
```

Figure 4.3: *netsort* sortnode structure.

of simulating communication intensive tightly-coupled application. The benchmark was not designed to achieve high performance and speedups of sorting.

Description of the *netsort4* benchmark follows. A *sortnode* is created for each element of the input sequence of size n. Sortnode is described by `struct` presented in figure 4.3. A Clam mobile pointer is created for each sortnode. During the setup procedure, sortnodes are created and initialized. Sortnodes are assigned `ids` from 0 to n, which do not change throughout the execution. The `stage` field is initialized to 0, and corresponds to the current stage of the algorithm for that sortnode. `DEPTH` is defined as the maximum stage for a problem instance. The `value` array contains values assigned to a sortnode on each of the stages. Initially the `value[0]` field is assigned an input sequence element. The array of Clam mobile pointers `partner` describes comparison sortnodes at each stage. The `partner_value` contains values of partner sortnodes, and the non-zero value of the ith element in `partner_ready` tells that the ith partner sortnode has reached the stage i. The `partner_id` keeps the id of the ith partner sortnode.

The benchmark is initiated by sending a *start* message to each of the sortnodes from processor 0. All other processors except 0 are polling for incoming messages until the *finish_handler* signals completion. Upon the arrival of *start* message a sortnode sends *receive_value* message to its partner sortnode on the first stage. That message includes the stage of the sortnode and its value at that stage. The *receive_value* handler checks whether the recipient sortnode reached the same stage as the stage specified in the message. If it did, the current sortnode value is compared with the value included in the message and is modified if necessary to *min* or *max* of the two depending on the `id` of the current partner, i.e., if it is numerically greater or smaller than its own `id`. The `stage` value is incremented, and the sortnode sends *receive_value* message to the next partner as described above.

The *receive_value* messages can reach a sortnode out-of-order. If the *receive_value* message from the *i*th partner arrives before the message from the *(i-1)*th partner, the `stage` value included in the message is greater than the stage of the recipient sortnode. In this case corresponding `partner_val` is assigned the sent value, and `partner_ready` is set to 1. The *receive_value* handler will make comparisons for stages *i* and *(i-1)* when *receive_value* message from partner *(i-1)* arrives. The algorithm finishes when all sortnodes reach the stage value of `DEPTH`.

A slightly modified version of the *netsort4* benchmark, *netsort5*, works the same way as *netsort4*, but the creation of sortnodes, and thus mobile pointers, is evenly distributed among the processors. It has been described in the previous chapter, that in Clam the processor where a mobile pointer was created is designated as *home* of the mobile pointer.

In order to study the effects of LMPs, after the completion of each stage a sortnode is migrated to a randomly assigned processor. The benchmark allows to increase access-to-mobility ratio by changing the frequency of object migrations. Another parameter of the benchmark is the message payload size.

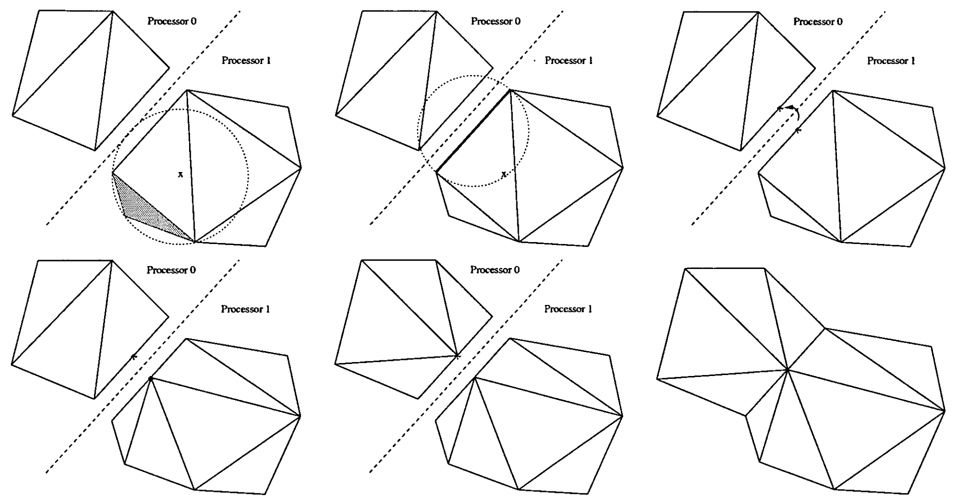


Figure 4.4: Example of inserting a boundary point in PCDT.

netsort is a tightly coupled and communication intensive benchmark. We believe, that some of the applications from AMR domain have similar communication properties.

4.1.2.2 PCDT End-to-End Application

Parallel Constrained Delaunay Triangulation (PCDT) is a parallel *mesh generation* algorithm based on Delaunay triangulation [45]. The reader is referred to [21] for the detailed description of the algorithm and for the definition of related terms.

The main difference of the PCDT algorithm from Delaunay triangulation is that the *point cavity* cannot expand across the predefined boundary. At the preprocessing stage of PCDT, the problem is divided into a number of subdomains satisfying certain boundary properties. Each subdomain can then be triangulated almost independently on separate processors. The process of subdomain triangulation consists of selecting and changing "bad" triangles (i.e., those, which do not satisfy certain geometric requirements) from the initial triangulation. The recalculation of the subdomain mesh can lead to modifications of the edges located on the subdomain boundary. When a new point has to be inserted on the boundary, a "split" message is sent to the neighboring subdomain located on some remote processor. The described process is depicted in figure 4.4[4].

An implementation of PCDT decomposes the initial domain and distributes resulting subdomains among the processors, which mesh the subdomains concurrently. We consider PCDT as a part of the end-to-end iterative application, where the requirements for a particular subdomain (and thus complexity of its processing) can dynamically change. In such application, meshing is just one of the steps in the computation. The resulting mesh is used by parallel Finite Element Methods (FEM) solvers. The subsequent mesh refinement or coarsening depends on the error estimators in the case of parallel FEM solvers. Static assignment of subdomains to processors would lead to unequal load, thus *dynamic load-balancing* is required.

The end-to-end PCDT benchmark simulates a real end-to-end application. It approximates the mesh generation phase of an adaptive FEM solver. The data collected from the single-iteration PCDT application[5] was used to estimate times for subdomain refinement, number and size of "split"-initiated messages, and sizes of the subdomains before and after refinement. The input parameters for the benchmark are number of iterations I, percentage of subdomains to be refined or derefined on each iteration R, and the level of aggregation of "split" messages A (aggregating multiple "split" points into a single message improves network utilization). The number of subdomains is fixed to 512.

The ILB layer of PREMA provides dynamic load-balancing within the PCDT bench-

[4]Figure 4.4 is a courtesy of Brian Holinka.

[5]We used single-iteration PCDT application developed by Andrey Chernikov.

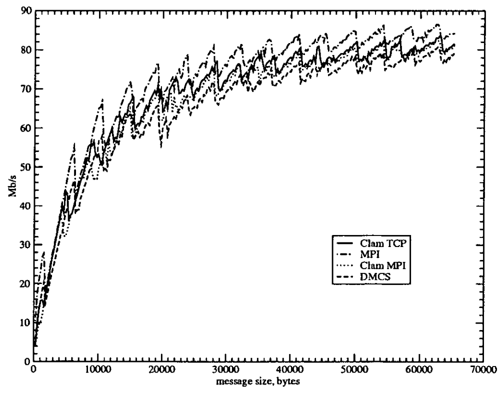
mark. The complete coverage of the algorithms used in ILB and its implementation can be found in [12–14]. We used work-stealing load-balancer for all the experiments.

$R \times 512$ subdomains are chosen at each iteration of the benchmark. These subdomains are refined or derefined (these operations are assumed to have similar complexity). The load-balancer has to redistribute subdomains according to dynamically changing load. Other subdomains are not involved in the refinement. The processed subdomains may however result in "split" messages sent to neighboring subdomains. These are the mobile object messages handled and routed by the Clam location management module.

4.2 Performance Evaluation of the Runtime System

We conducted a series of tests both to compare Clam with the DMCS/MOL implementation and to measure the absolute overheads. The first test measures maximum achieved bandwidth over the 100 Mbps Fast Ethernet link using ping-pong method. In this test Clam remote service request functionality is used to invoke remote function with the buffer of variable size as an argument. The Clam-achieved bandwidth is measured for TCP and MPI implementations of ACI. Similar test is done for DMCS, MOL and pure MPI. The results for small and large message sizes are plotted in figure 4.5.

The results of the ping-pong test show, that in almost all cases Clam achieves better bandwidth than DMCS and MOL. For small message sizes, LAM MPI performs best. In the case of large messages, Clam implemented on top of TCP ACI gives the best performance. This is happening mostly because for large messages LAM MPI is using three-way handshake protocol. The performance gain of Clam over the similar MOL functionality is over 20%.



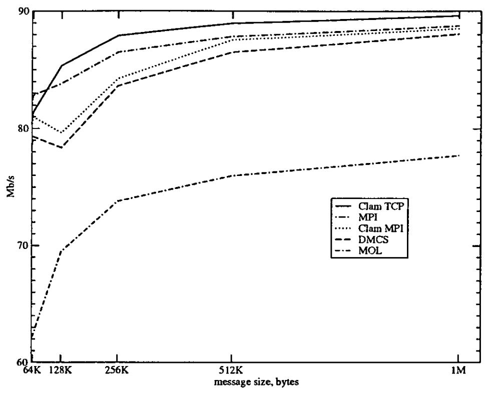


Figure 4.5: Maximum achieved bandwidth for small and large message sizes.

The poor performance of the MOL in this test is explained by the DMCS/MOL separation, which requires an additional memory copy. Both MPI and TCP implementations of Clam give better performance results than DMCS.

The second test was designed to compare the performance of the mobile message functionality of Clam vs MOL. In this test a single object is created on processor 0. This object is first migrated to processor 1, and a mobile message is sent to that object from processor 0. When the message is received, a *reply* RSR is invoked on processor 0 from the processor where the object is located. The latency measured in this test is defined as the time from sending a message to receiving the reply. Next the object migrates to processor 2, and the procedure is repeated. Lazy Forwarding location management policy is used, so when the object is located on processor 2, each message sent to it from 0 traverses through processor 1 to 2. The results of the test are presented in figure 4.6.

The mobile object message latency test demonstrates that (1) Clam has better overall performance, and (2) the per-hop overhead is about constant when Clam is used while it is increasing for MOL.

The last two performance tests evaluate the overall effectiveness of Clam. Figure 4.7

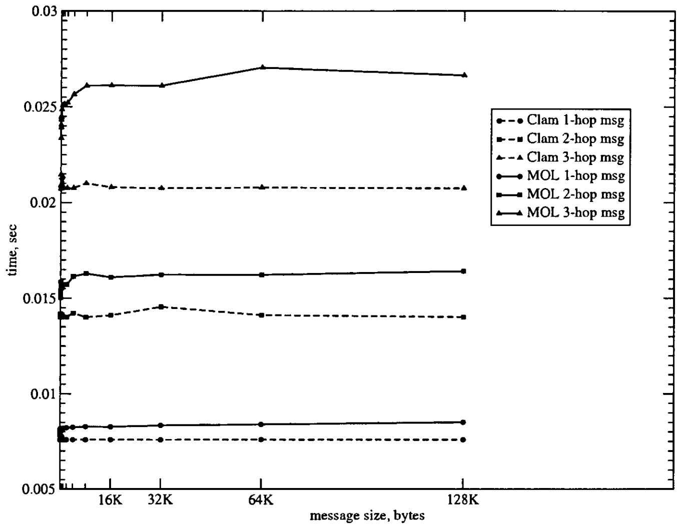


Figure 4.6: Mobile object message latency test.

plots run times of the *netsort* benchmark for Clam and MOL implementations. For small processor configurations MOL-based implementation performs better. However, Clam outperforms MOL when more than 8 processors are used, and the difference is increasing when we scale the size further. Clam is more scalable because of the non-blocking communication algorithms used.

Finally, figure 4.8 plots runtime breakdown for the Parallel Constrained Delaunay Triangulation (PCDT) application which is using PREMA load-balancing functionality implemented with Clam (in this test we had 512 subdomains of the 2-D pipe model, the algorithm generated about 35 million triangles, the subdomains were assigned area bounds between 1.92e-2 and 0.26e-2; the test ran on 32 nodes of the Whirlwind subcluster of SciClone). The plot shows, that the overhead introduced by Clam is within 5% of the total execution time. The communication component of the execution time for no-balancing test is caused by continuous polling in absence of work.

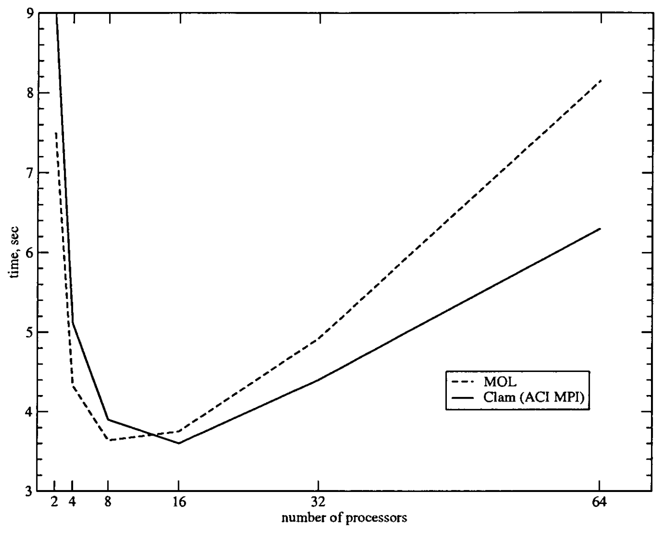


Figure 4.7: *netsort4* benchmark performance.

4.3 Evaluation of the Location Management Policies

In order to evaluate the LMPs described in section 3.2, we first ran *netsort4* and *netsort5* benchmarks on a different number of nodes within the SciClone cluster using different location policies. For configurations of up to 64 processors we used the Typhoon subcluster with one process running on each node. 128-processor experiment was ran using all the nodes of the Whirlwind, Typhoon and Tornado subclusters. We ran the benchmark with 4096 random numbers to sort (78 stages). All message and sortnodes were appended with the payload of 10 Kbytes. We used MPI ACI for all of the experiments since MPI ACI showed better performance for small messages. The directory updates use small messages and overall most of the messages in our benchmarks are less than 64 Kbytes.

The total execution times of the *netsort4* and *netsort5* benchmarks using different LMPs are plotted in figures 4.9 and 4.10 respectively. PU LMP is not present on the plots, because it was designed specifically for the partitioned testbed configuration of 32 processors only.

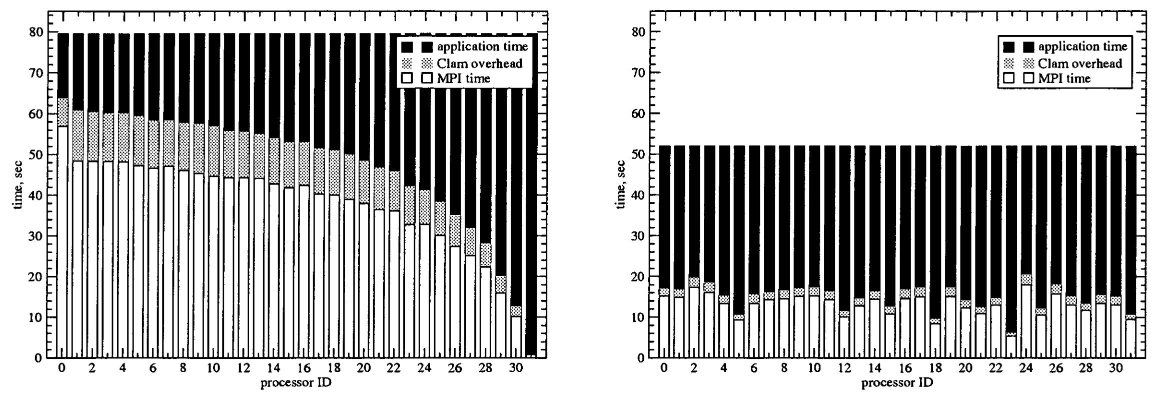


Figure 4.8: PCDT runtime breakdown without load-balancing and with PREMA diffusion load-balancing.

This experiment evaluates the impact of LMP on the application performance. Results show, that the total runtime grows when we increase the configuration for all of the policies. JU and EU LMPs give the best results, while HB and BU LMPs are the worst.

There is a number of LMP properties which affect the overall performance of a communication intensive mobile object application. The requirements for an efficient LMP are listed in section 3.2. Figures 4.11, 4.12 and 4.13 help explaining the performance results.

Figures 4.11 and 4.12 show the distribution of the number of hops application messages had to travel for the *netsort4* test. BU LMP for this application guarantees the shortest path. Apparently, most of the directory updates arrive in time so that the subsequent messages are delivered in one hop. HB LMP also gives very good distribution: most of the messages reach destination in two hops, and almost no messages take more than four hops. Messages travel longer paths as intensity of updates decreases from PC to JU and to no updates in LF LMP. While for LF the maximum number of hops is 26, for PC it is 13.

However, the data from the message hop distribution (i.e., figures 4.11 and 4.12) are not sufficient to explain the relative performance of the LMPs. For example, although HB LMP

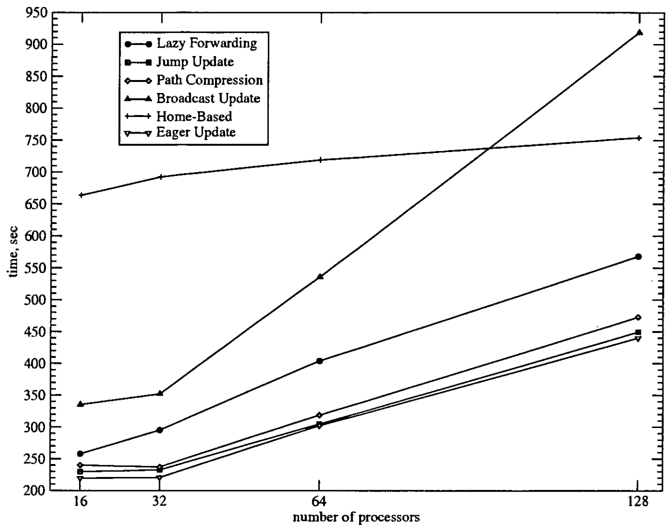


Figure 4.9: Execution times of *netsort4*.

provides the shortest travel distance, it has very bad performance overall. Figure 4.13 gives the breakdown of internal Clam messages into three types. *Application messages* are initial messages sent to an object as a result of mobile object message function invocation. These messages may result in a sequence of forwardings, until they finally reach the processor where the targeted object is located (*forwarding messages*). After that an LMP may send one or more *update messages*, which are also included into the breakdown.

It can be seen from figure 4.13 that processor 0 performs more communication than any other processor regardless of the LMP used. Application can always send a message to an object using its mobile pointer. At the time a processor posts a message, its local directory may not have any information about the target object. In this case, any LMP within Clam will use the internal mobile pointer information to determine the target of the initial message. Each mobile pointer in Clam includes the ID of the processor, where the mobile pointer/object was created. This is the "home" processor of the mobile pointer,

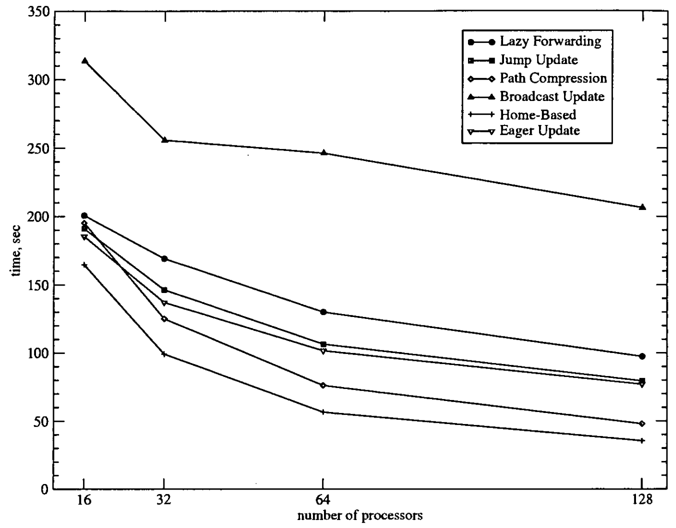


Figure 4.10: Execution times of *netsort5*.

and it will be used as the "best location guess". In *netsort4* we have 4096 objects evenly distributed among the processors during the initialization stage. All objects have the same "home" – processor 0, where they were created. Obviously, when a message is sent to an object for the first time from a specific processor, it will be sent to processor 0. If the total number of processors is n, $\frac{4069}{n}$ objects are local to each of the processors. The rest $(4096 - \frac{4096}{n})$ objects are non-local, and the best guess for those objects is the "home" processor 0. Hence, the probability of a message to be sent to processor 0 during the first stage P_1 can be defined as

$$P_1 = \frac{4096 - \frac{4096}{n}}{4096} = 1 - \frac{1}{n}.$$

During each of the subsequent stages each of the sortnodes will be assigned randomly to some new processor. Thus, the total number of the objects, which are known to the directory of a specific processor can increase at most by $\frac{4096}{n}$ (if all of the newly arrived

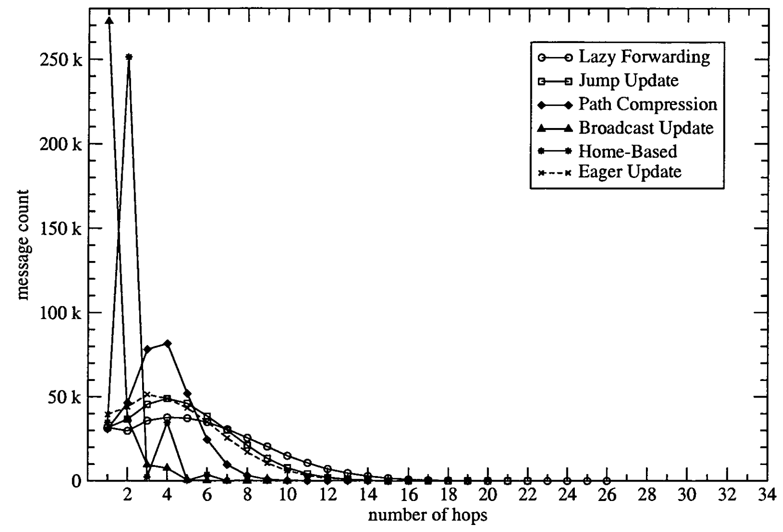


Figure 4.11: Number of hops for a message to reach the object; *netsort4*, 32 processors.

objects were not known at the previous stage). Given an algorithm stage i, the probability P_i of a message being sent to "home" processor 0 can be defined as

$$P_i = \begin{cases} 0 & \text{if } i \geq n \\ 1 - \frac{i}{n} & \text{if } i < n \end{cases}$$

Apparently, for fixed problem size, the probability of sending a message to processor 0 is growing as we increase the number of processors.

Figure 4.13 supports the observation that communication on processor 0 is the limiting factor of the benchmark performance. The forwarding messages are significantly more expensive than the update messages (10 Kbytes vs about 100 bytes). That is why HB and LF with the high amount of forwarded messages do not perform well. We also see that although JU and PC decrease the amount of forwarded messages, they fail to improve the application performance much because of the existing bottleneck.

A sortnode is migrated to a different processor after each comparison in *netsort4*. Also, each comparison is a result of *receive_val* message arriving to the object. Most of the time

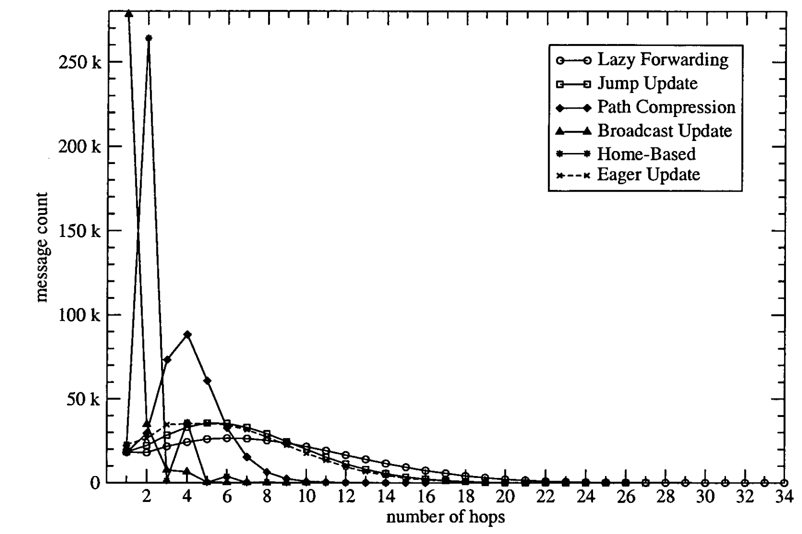


Figure 4.12: Number of hops for a message to reach the object; *netsort4*, 64 processors.

an object migrates after it receives a message. Because of this, JU and EU LMPs have roughly the same performance, and their effect of update is the same.

The performance of HB LMP does not change very much with the increase in the number of processors. Messages to non-local objects are always routed through the "home" processor 0. The rate of arrival for those messages increases however, that explains the slight increase in the runtime for HB LMP.

The execution time of the benchmark is increasing almost linearly as we increase the number of processors for BU LMP. The reason for this is that the time spent during the initialization stage dominates the execution time. During the initialization all objects are distributed among the processors, and for each object processor 0 has to send n updates. The performance of BU LMP for *netsort* is determined by the *update* traffic. In *netsort4* the initialization stage is becoming more expensive as we add more processors, while the runtime update costs per processor decrease.

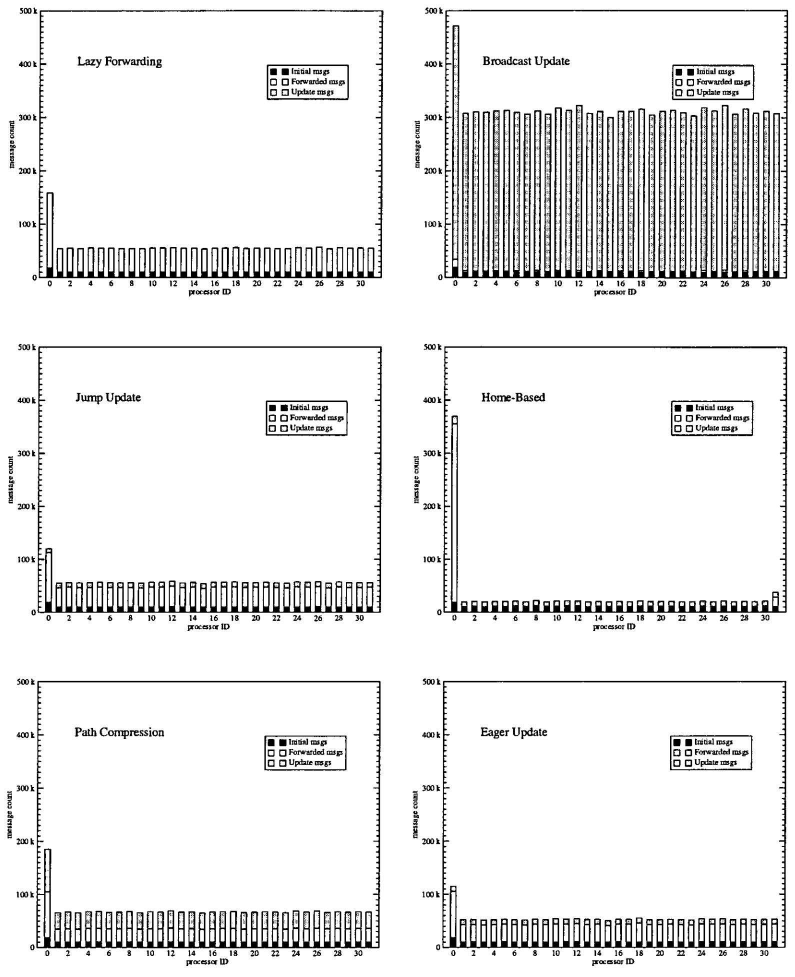


Figure 4.13: Breakdown of point-to-point message types for *netsort4*, 32 processors.

The evaluated LMPs have totally different relative performance for the *netsort5* benchmark. Each processor in this benchmark creates equal number of mobile pointers, i.e., the LM overheads at "home" processors are balanced. Figure 4.14 shows, that the communication on processor 0 is no longer the determining factor in the application performance. Although it performs slightly more communication (compared to other processors) during the initial stage of the algorithm, the application performance depends more on the average load of processors. We observe that out of the three forwarding LMPs, PC LMP gives the best results, because it provides the shortest message path. Behavior of HB LMP is very dif-

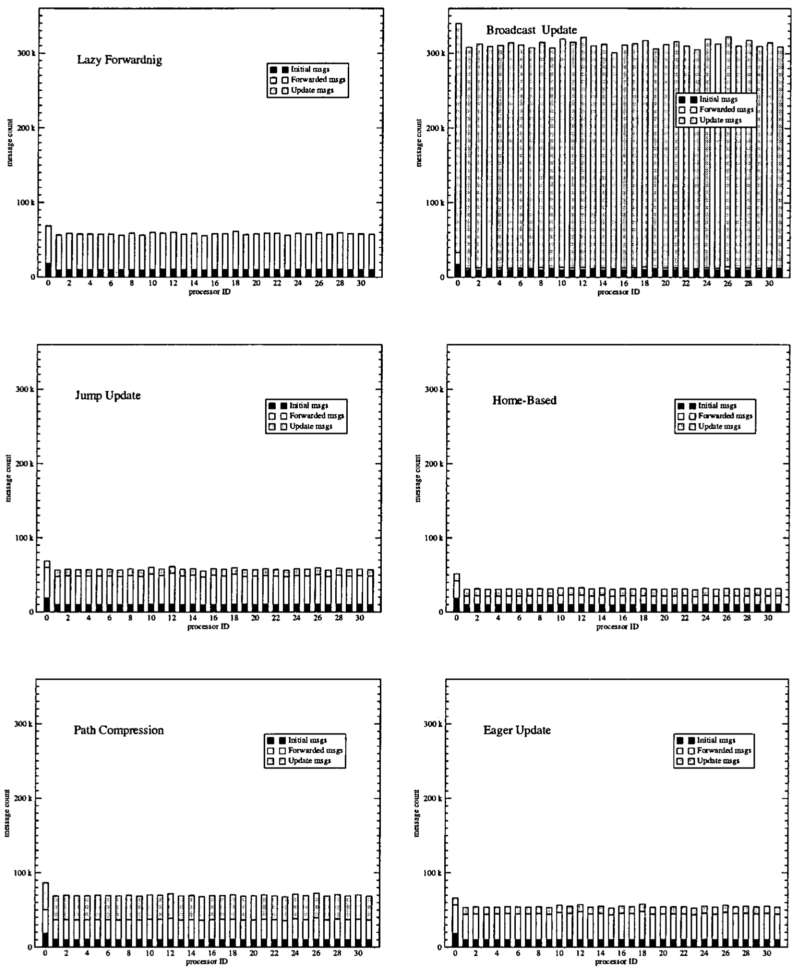


Figure 4.14: Breakdown of point-to-point message types for *netsort5*, 32 processors.

ferent. The bottleneck is eliminated and all the processor on average receive equal number of messages. HB LMP also has very short forwarding path. This explains good scalability of HB LMP for the *netsort5* benchmark. The performance of BU LMP improves as we increase the number of processors: the initialization costs and update costs per processor decrease (fewer objects are created per processor).

In the next series of experiments we studied the effect of different application parameters on the performance of the LMPs. One of such parameters is the mobile message payload. Figure 4.15 shows that the difference in message size affects the execution time, but not the

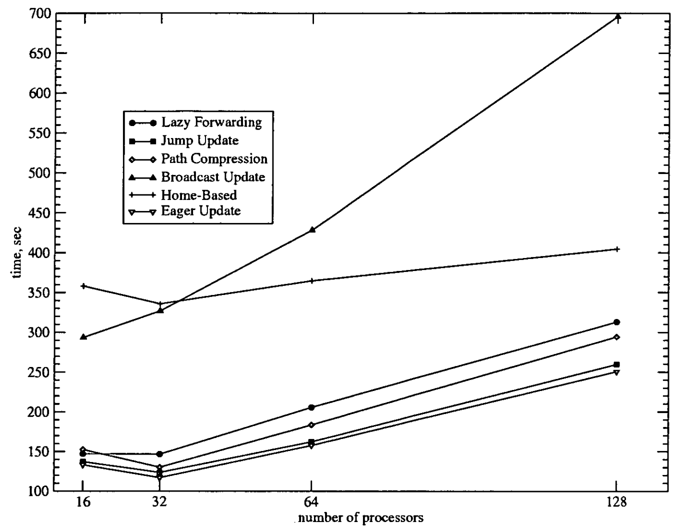


Figure 4.15: *netsort4* benchmark with 1 Kbyte message payload.

relative performance of different LMPs (the message is 10 times smaller compared to the previous test with *netsort4*).

Access-to-mobility ratio λ in PDC applications is analogous to CMR in cellular networks. For a given application, $\lambda = \frac{a}{m}$, where a is total number of accesses (search operations) to objects and m is the total number of object migrations. λ has been used as a parameter in the study of forwarding techniques performed in [27]. The *netsort4* and *netsort5* benchmarks have been modified to experiment with different values of λ and see the impact of those changes on the application performance. Figures 4.16 and 4.17 show performance results of running *netsort4* and *netsort5* benchmarks with $\lambda \approx 20$ (a sortnode is migrated once every 20 stages).

Our results show, that the change in access-to-mobility ratio can change the performance of an application when using different LMPs. BU LMP achieves the best execution times for *netsort4* when $\lambda \approx 20$, while for the same test with $\lambda \approx 1$ it had poor performance.

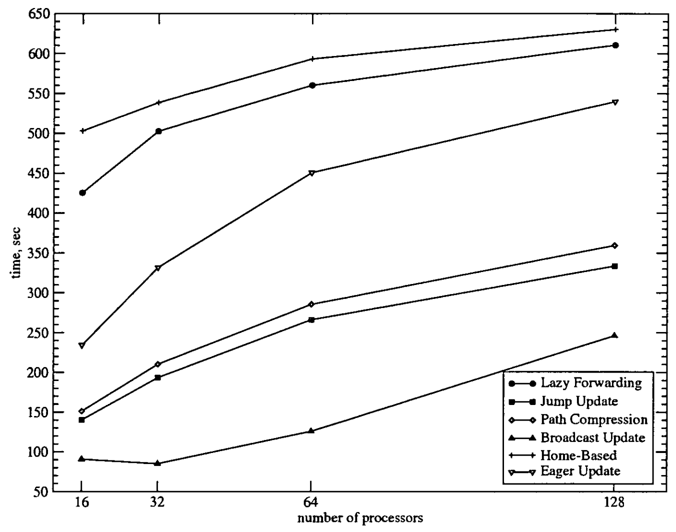


Figure 4.16: Execution times of *netsort4* with $\lambda \approx 20$.

The total number of movements decreased. The *update* operation is very expensive for BU LMP, and higher values of λ allow for better amortization of the update costs.

The significant difference between EU and JU LMP can be observed from figure 4.17. EU LMP sends updates when an object moves, while JU updates the sender immediately after the message arrives. The distribution hops for mobile messages is shown in figure 4.18. Apparently, the average length of message path is shorter for JU LMP than for EU LMP, thus JU is more effective overall.

The *netsort4* experiment shows, that the change in λ does not impact the performance of HB LMP. All messages are still routed through the single "home" node, and the total number of messages is the same as in the previous test. The slight improvement in runtime is due to the reduction of communication on processor 0 (less updates are sent). The level of concurrency is lower, and hence the information in 0's directory is correct more often.

The same experiment with different λ for *netsort5* shows the reduction in the execution

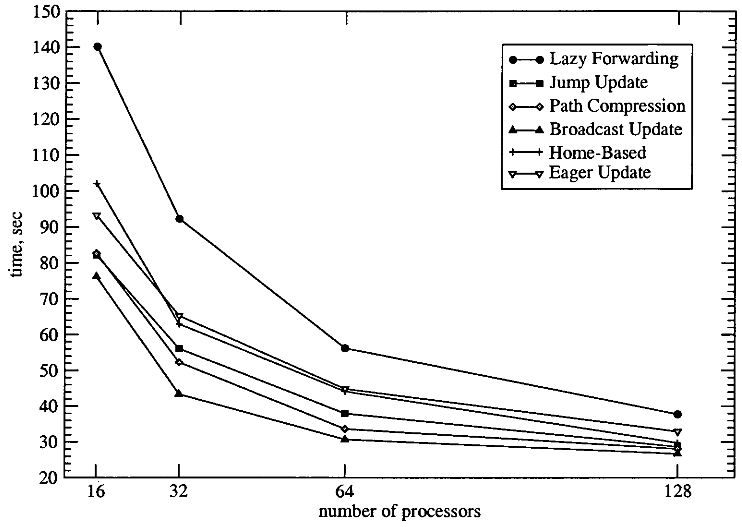


Figure 4.17: Execution times of *netsort5* with $\lambda \approx 20$.

time for all LMPs. BU LMP again performs much better than in the *netsort5* with $\lambda \approx 1$ because of significantly fewer object migrations and thus fewer updates. LF LMP has very bad performance compared with other algorithms. The absence of update mechanisms leads to significant performance degradation in applications with high access-to-mobility ratio.

The last set of *netsort* experiments was aimed to evaluate the impact of the network on the performance of the studied policies. PU LMP has been designed specifically to address the issue of network partitioning. For this test we configured PU LMP implementation so that the processors are partitioned according to the switch of the network testbed they are attached. Link bandwidth is 10 Mbps between partitions and 100 Mbps within each partition.

We ran *netsort4* benchmark with $\lambda \approx 1$ and message payload 1 Kbyte on the experimental testbed with 10/100 Mbps configuration. We discovered, that for that experiment PU and HB LMPs outperform all other algorithms. The total execution time of the appli-

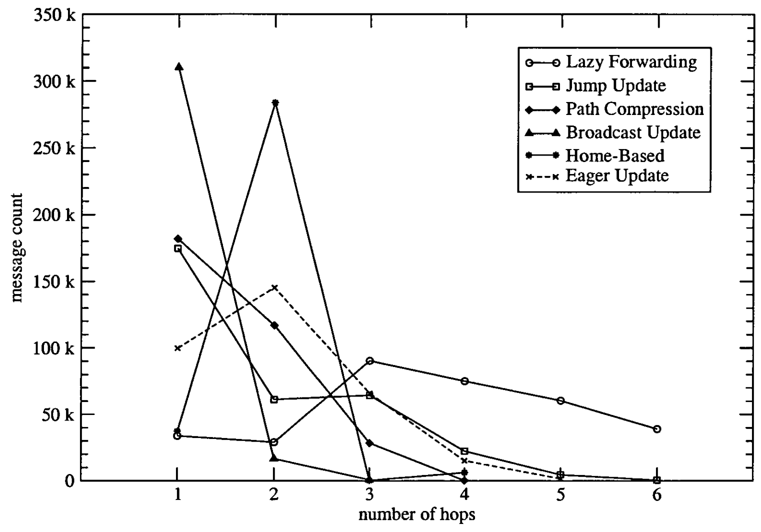


Figure 4.18: Number of hops for a message to reach the object, $\lambda \approx 20$; *netsort4*, 32 processors.

cation on the regular network configuration and on the experimental testbed is shown in figure 4.19. The computing nodes used in this experiment are the same as in the previous case, where we also used 32 nodes of Typhoon subcluster.

The results of HB LMP are explained by the previously shown data: most of the messages reach the target in two hops. When the number of hops increases, so does the probability of a message to be forwarded through the inter-partition network link. PU LMP achieves high efficiency for the two reasons: (1) the location information is updated with lower network costs compared to BU LMP; (2) it provides good average for the forwarding chain length.

We used PCDT end-to-end benchmark to perform analogous study of the evaluated LMPs. On each iteration of the benchmark we choose 20% of the subdomains which require processing by the PCDT algorithm. These "heavy" subdomains have to be balanced among the processors. The test case consisted of 20 iterations. Aggregation level was set to

50. Unequal load distribution was balanced by the ILB layer of PREMA using work-stealing method [13]. The execution times of this benchmark for different LMPs is given in figure 4.20, and the hop distribution in figure 4.21.

The difference in hop distribution caused by difference in LMP is similar to the one we observed in the *netsort* benchmarks. However, the execution time is not affected as much as it was in our previous experiments. All of the LMPs have about the same performance (the difference is within 10% except for the BU LMP). We identified a number of possible reasons for such behavior. The significant difference between *netsort* and PCDT benchmarks is that the latter is using ILB module. The ILB implementation is multi-threaded: the polling thread is running concurrently with the main application thread [12]. Another difference of the PCDT test is the significant increase of the computational complexity compared to communication-dependent *netsort*. As a side-effect of polling, forwarding and update messages can be processed on the background simultaneously with the application handler execution. This would definitely hide the latencies introduced by LMPs.

Another interesting observation we have made is that the choice of LMP is somehow affecting the ILB work migration decisions. Figure 4.22 shows, that there is significant difference in the total number of objects migrated for different LMPs (the difference is especially large between JU and HB LMPs). The reasons for such a difference are not clear to us and have to be studied further.

4.4 Discussion

The performance study described in this chapter shows, that location management policy may have significant impact on the application performance. The important conclusion of the evaluation is that the choice of mechanisms used for location management for a particular application cannot be disregarded.

The comparison of results we obtained from *netsort4* and *netsort5* benchmarks shows how important it is to have a balanced assignment of the mobile objects to "home" processors. This assignment can be handled by the runtime system and should be evaluated in the future. Another implementation detail which should be considered is multithreaded runtime system implementation. In the context of location management, multithreading together with the idea of multiple communication channels can eliminate interference of application with LMP.

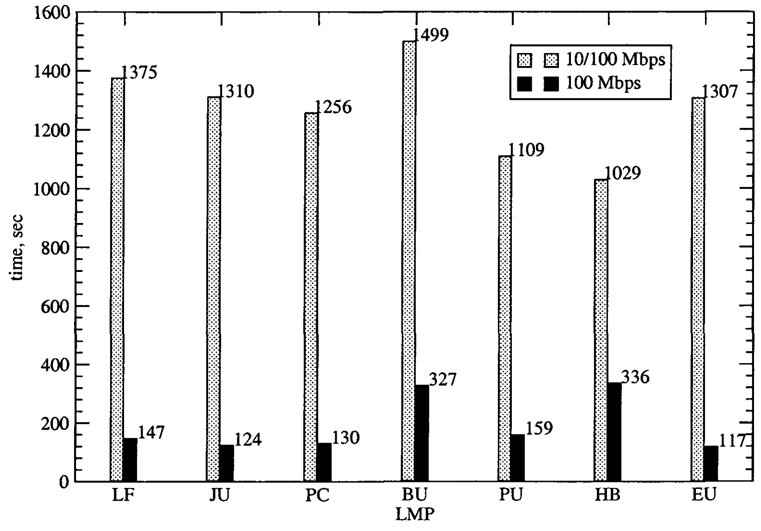


Figure 4.19: *netsort4* execution time on 100 Mbps and 10/100 Mbps configurations.

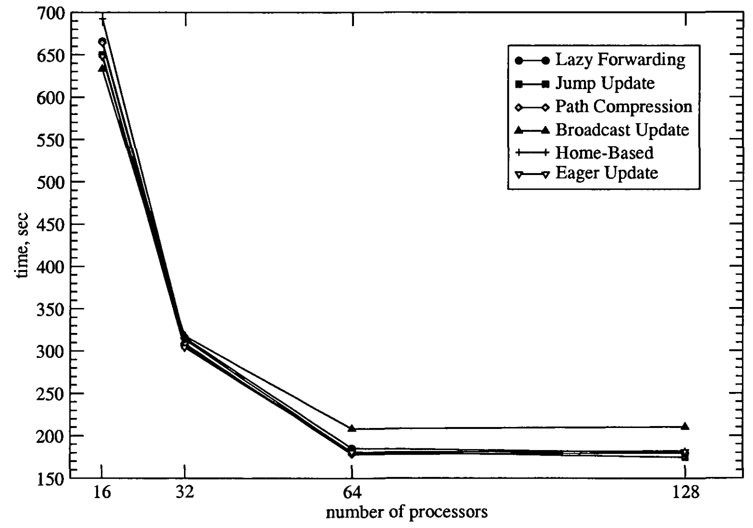


Figure 4.20: PCDT benchmark execution time.

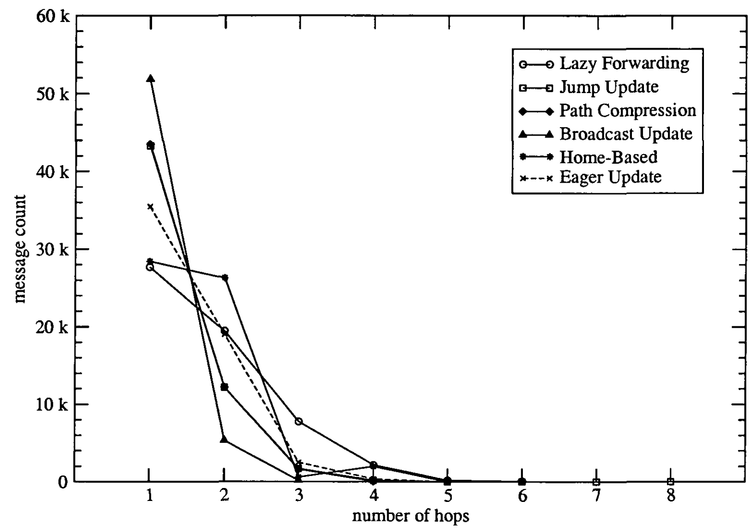


Figure 4.21: Number of hops for a message to reach the object; PCDT benchmark, 32 processors.

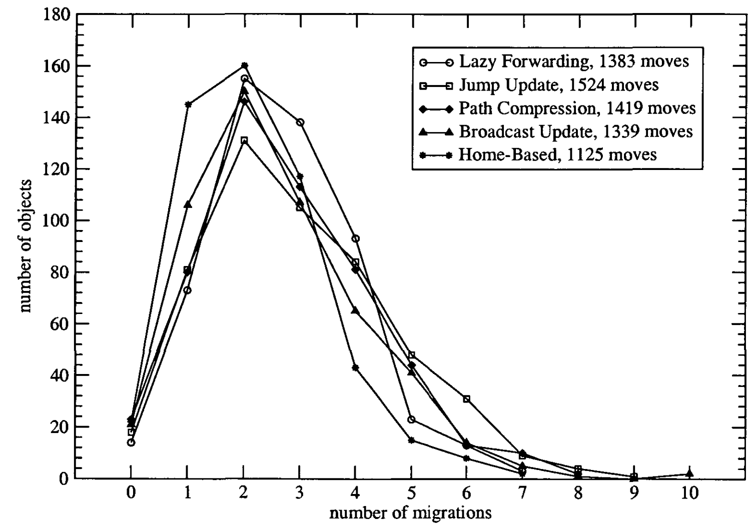


Figure 4.22: Difference in the object migration intensity as a side-effect of changing LMP.

Chapter 5

Conclusions and Future Work

Efficiency, performance, and the costs of development and maintenance for scientific computing applications are directly dependent on the quality and capabilities of the runtime software. Clam, the runtime system we present in this thesis, addresses these issues along with a number of previously unresolved runtime support problems within the PREMA framework. We made PREMA more stable, improved its portability, performance, and usability by rebalancing the three fundamental issues: correctness, performance, and ease-of-use.

The second major contribution of our work is in the survey, comparison and evaluation of location management techniques for parallel distributed computation applications. The problem of location management in PDC has not been carefully examined previously within the described model. Our results show, that location management is extremely important for some PDC applications. Moreover, we show that the optimal choice of a LMP for certain highly coupled communication intensive applications depends on multiple factors: number of nodes involved in the computation, properties of the communication network, migration and communication patterns of the application. At the same time, our preliminary data also show that for loosely coupled applications location management does not play a significant role in the application performance (PCDT benchmark). The conclusion from our preliminary evaluation that location management must be considered during the application development. It may be crucial for an application to have the ability to choose the most appropriate LMP. Clam has been designed to provide this choice to the application.

The study of location management we have done answered the main question we asked: "Is location management relevant?" There is a number of issues we want to investigate next. A separate study has to be done on how to choose the optimal LMP for a specific application and platform configuration. In this thesis we have identified and implemented only the most intuitive location management techniques. With the results we have collected in this evaluation, new LMPs can be designed and implemented within Clam which combine features of the evaluated LMPs and thus meet application needs better. We plan to investigate the feasibility and relevance of dynamic LM, where the most appropriate LMP would be chosen based on the current properties of the application and/or environment (network). We also need to study how different load-balancing algorithms are affected by LM.

Finally, our results show that network bandwidth impacts the relative performance of LMPs. Location management techniques which take network properties into account have better performance than LMPs which do not take this into consideration. Further, we need to explore in detail how other parameters of the interconnect (network latency, packet loss etc.) affect location management.

Bibliography

[1] LAM General User's Mailing List Archives. <http://www.lam-mpi.org/MailArchives/lam/msg07032.php> (8 November 2003).

[2] LAM MPI. <http://www.lam-mpi.org> (3 November 2003).

[3] MPI: A Message-Passing Interface Standard. <http://parallel.ru/docs/Parallel/mpi1.1/mpi-report.html> (27 August 2003).

[4] SciClone Cluster Project. <http://www.compsci.wm.edu/SciClone> (27 August 2003).

[5] Voyager. <http://www.recursionsw.com/products/voyager/voyager.asp> (23 October 2003).

[6] Jeff Skuyres (LAM MPI developer). Personal communication, October 2003.

[7] Laxmikant Kalé, Raveen Kumar, Orion Lawlor (Charm++ developers). Personal communication, October 2003.

[8] I. Akyildiz, J. McNair, J. Ho, H. Uzunalioglu, and W. Wang. Mobility Management in Next-Generation Wireless Systems. In *Proceedings of the IEEE*, volume 87, pages 1347–1384, August 1999.

[9] Sara Alouf, Fabrice Huet, and Philippe Nain. Forwarders vs. Centralized Servers: An Evaluation of Two Approaches of Locating Mobile Agents. *Performance Evaluation*, 49(1–4):299–319, 2002.

[10] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. Eigler, and G. Gao. ABC++: Concurrency by Inheritance in C++. *IBM Systems Journal*, 34(1):120–137, 1995.

[11] B. Awerbuch and D. Peleg. Online Tracking of Mobile Users. *Journal of the Association for Computing Machinery*, 42(5):1021–1058, September 1995.

[12] K. Barker and N. Chrisochoides. An Evaluation of a Framework for the Dynamic Load-Balancing of Highly Adaptive and Irregular Parallel Applications. In *Proceedings of SuperComputing'03*, 2003.

[13] K. Barker, N. Chrisochoides, A. Chernikov, and K. Pingali. A Load Balancing Framework for Adaptive and Asynchronous Applications. *IEEE Transactions on Parallel and Distributed Computing*, 14(12), December 2003.

[14] Kevin Barker. *Load-balancing Support for Adaptive and Irregular Applications on Distributed Memory Parallel Machines.* PhD thesis, The College of William and Mary. Expected to be completed in 2004.

[15] Kevin Barker, Nikos Chrisochoides, Jeffrey Dobellaere, Demian Nave, and Keshav Pingali. Data Movement and Control Substrate for Parallel Adaptive Applicatios. *Concurrency: Practice and Experience*, 14(2):77–101, 2002.

[16] J. K. Bennett. The Design and Implementation of Distributed Smalltalk. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Norman Meyrowitz, editor, volume 22, pages 318–330, New York, NY, 1987. ACM Press.

[17] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer*, pages 87–98, 1994.

[18] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly, 2nd edition, 2003.

[19] J. Cao, X. Feng, J. Lu, and S. K. Das. Mailbox-Based Scheme for Mobile Agent Communications. *Computer*, 35(9):54–60, September 2002.

[20] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, Litchfield Park AZ USA, 1989.

[21] P. Chew, N. Chrisochoides, and F. Sukup. Parallel constrained delaunay meshing. In *Proceedings of 1997 Joint ASME/ASCE/SES Summer Meeting, Special Symposium on Trends in Unstructured Mesh Generation*, 1997.

[22] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel. Mobile Object Layer: A Runtime Substrate for Parallel Adaptive and Irregular Computations. *Advances in Engineering Software*, 31(8–9):621–637, 1998.

[23] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2nd edition, 2001.

[24] Edsger W. Dijkstra. Shmuel Safra's version of termination detection, 1987. Manuscript EWD998-7, <`http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD998.PDF`> (2 November 2003).

[25] EIA/TIA. Cellular Radio-telecommunications Intersystem Operations. Technical Report IS-41 revision C, 1995.

[26] ETSI/TC. Mobile Application Part (MAP) Specification, version 4.8.0. Technical Report recommendation GSM 09.02, 1994.

[27] ROBERT J. FOWLER. *Decentralized Object Finding Using Forwarding Addresses.* PhD thesis, University of Washington, 1985.

[28] MAURICE HERLIHY AND MICHAEL P. WARRES. A Tale of Two Directories: Implementing Distributed Shared Objects in Java. *Concurrency: Practice and Experience*, 12(7):555–572, 2000.

[29] R. JAIN AND Y-B. LIN. An Auxiliary User Location Strategy Employing Forwarding Pointers to Reduce Network Impacts of PCS. *Wireless Networks*, 1(2):197–210, July 1995.

[30] R. JAIN, Y-B. LIN, C. LO, AND S. MOHAN. A Caching Strategy to Reduce Network Impacts of PCS. *IEEE Journal on Selected Areas in Communication*, 12(8):1434–1444, October 1994.

[31] ERIC JUL, HENRY LEVY, NORMAN HUTCHINSON, AND ANDREW BLACK. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[32] N. KARNIK AND A. TRIPATHI. Design Issues in Mobile-Agent Programming Systems. *IEEE Concurrency*, 6(3):52–61, 1998.

[33] P. KELEHER. Decentralized replicated-object protocols. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)*, 1999.

[34] P. KRISHNA, N. H. VAIDYA, AND D. K. PRADHAN. Static and Dynamic Location Management in a Distributed Mobile Environment. Technical Report 94-030, Dept. of Computer Science, Texas A&M University, 1994.

[35] D. B. LANGE AND M. OSHIMA. Seven Good Reasons for Mobile Agents. *Communications of the ACM*, 42(3):88–89, March 1999.

[36] O. LAWLOR AND L. V. KALÉ. Supporting Dynamic Parallel Object Arrays. *Concurrency and Computation: Practice and Experience*, 15:371–393, 2003.

[37] BARBARA LISKOV, MARK DAY, AND LIUBA SHRIRA. Distributed Object Management in Thor. In *Proceedings of International Workshop on Distributed Object Management*, pages 79–91, 1992.

[38] D. MILOJICIC, V. KALOGERAKI, R. LUKOSE, K. NAGARAJA, J. PRUYNE, B. RICHARD, S. ROLLINS, AND Z. XU. Peer-to-Peer Computing. Technical Report HPL-2002-57, HP Laboratories Palo Alto, 2002.

[39] P. MÜLLER, F. VAN MEGEN, AND T. KLEINBERGER. A New Approach for Locating Moving Programs Based on Experiences from the PLMN Domain. <`http://www.icsy.de/~vanmegen/paper/Reasons\%20for\%20a\%20specialize%d\%20Location\%20Service.pdf`> (23 October 2003), 2001.

[40] EVAGGELIA PITOURA AND GEORGE SAMARAS. Locating Objects in Mobile Computing. *Transactions on Knowledge and Data Engineering*, 13(4):571–592, July/August 2001.

[41] D. PLAINFOSSÉ AND M. SHAPIRO. A Survey of Distributed Garbage Collection Techniques. In *Proceedings of International Workshop on Memory Management*, pages 211–249, 1995.

[42] M. POWELL AND B. MULLER. Process Migration in DEMOS/MP. *Operating Systems Review*, 17(5), October 1983.

[43] S. RAJAGOPALAN AND B. R. BADRINATH. An Adaptive Location Management Strategy for Mobile IP. In *Proceedings of the 1st ACM International Conference on Mobile Computing and Networking*, 1995.

[44] M. SHAPIRO, P. DICKMAN, AND D. PLAINFOSSÉ. SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection. Technical Report 1799, 1992.

[45] J. R. SHEWCHUK. Tetrahedral mesh generation by delaunay refinement. In *Proceedings of Symposium on Computational Geometry*, pages 86–95, 1998.

[46] N. SHIVAKUMAR AND J. WIDOM. User Profile Replication for Faster Location Lookup in Mobile Environments. In *Proceedings of the 1st ACM International Conference on Mobile Computing and Networking*, pages 161–169, 1995.

[47] A. TRIPATHI, N. KARNIK, T. AHMED, R. SINGH, A. PRAKASH, V. KAKANI, M. VORA, AND M. PATHAK. Design of the Ajanta System for Mobile Agent Programming. *Journal of Systems and Software*, 62(2):123–140, May 2002.

[48] T. VON EICKEN, D.E. CULLER, S.C. GOLDSTEIN, AND K.E. SCHAUSER. Active Messages: A Mechanism for Integrated Communication and Computation. In *19th Annual Symposium on Computer Architecture*, 1992.

[49] P. WOJCIECHOWSKI. Algorithms for Location-Independent Communication between Mobile Agents. Technical Report DSC-2001/13, Département Systèmes de Communication, EFPL, 2001.

VITA

Andriy Fedorov

Andriy Fedorov was born in Berdyansk, Ukraine on May 10, 1980. He graduated from Ternopil High School No.10 in 1997. In 2001 he completed requirements for BS degree in Computer Science at Institute of Computer Information Technologies of Ternopil Academy of National Economy, Ukraine. Starting from 2001 he is pursuing Ph.D. degree at the College of William and Mary.