

# Location Management in Object-based Distributed Computing\*

Andriy Fedorov and Nikos Chrisochoides  
Department of Computer Science  
The College of William and Mary  
Williamsburg, VA 23185-8795  
{fedorov,nikos}@cs.wm.edu

## Abstract

*Distributed systems which use non-stationary communicating objects have to address the problem of managing locations of these objects. We study location management (LM) and its impact on application performance in the context of dynamic load-balancing for parallel distributed computing. We summarize our experience with LM in distributed object-based systems by comparing six location management policies (LMPs). The LMPs are studied within the PREMA framework used mainly for parallel mesh generation and refinement applications. Our experimental study is using a synthetic tunable micro-benchmark and two mesh generation applications. We explain why the commonly adopted in practice Jump Update LMP is efficient for most of our applications and we compare it with the other location management techniques. We show how the performance of a particular LMP can be affected by the application properties and its data layout. Finally, we identify the conditions under which certain LMPs are more beneficial than Jump Update.*

## 1. Introduction

The problem of location management (LM) is relevant in parallel and distributed systems, which operate on dynamically relocating objects. Techniques for managing location, i.e., Location Management Policies (LMPs), describe the rules which are used to find objects and the actions to be performed when objects migrate within the network.

A LMP should provide efficient implementations for *move* and *find* operations on objects. Efficiency of a LMP can be measured in terms of communication, computation

overheads and response time. There is a trade-off between the complexities of *move* and *find* operations. This trade-off can be illustrated by the following two extreme strategies for location management [5]. The “*full-information*” strategy requires the up-to-date information about all objects for efficient *find*, but then the cost of *move* is high (all nodes have to be updated when objects migrate). On the other hand, the “*no-information*” strategy does not require location updates. Consequently, the *find* operation is very expensive.

In this paper we evaluate the impact of location management policies on performance of parallel and distributed applications, which perform computations using the *mobile object* abstraction [16]. Specifically, we evaluate a number of diverse location management policies within Portable Runtime Environment for Mobile Applications (PREMA). PREMA provides implicit load-balancing support [7] for computation- and communication-intensive adaptive irregular distributed applications (primarily parallel mesh generation applications). The load-balancing framework implemented within PREMA operates on user-defined *schedulable* objects [6] representing the workload. The distributed and coupled nature of the targeted applications requires communication with the non-local data, which brings up the issue of efficient location management within PREMA or any similar runtime system.

The LMPs we evaluate combine existing experience of location management in Parallel Distributed Computing (PDC) and other relevant areas, like mobile cellular networks. This evaluation is the first, to the best of our knowledge, comprehensive evaluation of location management in PDC. We use a synthetic micro-benchmark and two real applications to evaluate the impact of location management techniques on their performance. The main question we attempt to answer in this study is “*Under what conditions and how location management can affect the application performance?*”

---

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

The paper is structured as follows. We first give an overview of the related areas, where location management problem has been addressed: DSM systems, cellular networks and mobile agents systems. In Section 3 we describe the object-based computation approach as it is implemented within PREMA, our assumptions, and the specifics of our model. Section 4 presents the design space for location management and the implemented LMP. The evaluation section describes the applications, experiments and analysis of the obtained results. We conclude with the discussion on our results and future work.

## 2. Related Work

We studied a number of runtime systems and languages, which support mobility of user-defined objects. Our survey showed, that most of them use the same Jump Update (JU) LM mechanism, which is based on message forwarding. The *forwarding addresses* technique was first used in DEMOS/MP operating system in the context of process migration [30]. Later it was extended and evaluated by Fowler in [17]. When the forwarding technique is used, every time an object migrates, the processor it leaves keeps information about the new location of the object. Any communication directed to the object will follow the sequence of pointers and eventually reach the object, given it does not constantly relocate. The JU forwarding technique, which we describe further, is used in Emerald [21] (object-oriented system with fine-grained object mobility support), Thor [26] (object-oriented database management system), Amber [9] (provides simplified model for multiprocessor applications), SSP chains [31] (distributed technique for garbage-collection), MOL [11] (mobile object functionality for load-balancing), Charm++ [23] (dynamic load-balancing).

Another technique for mobile object location management uses *centralized directory*. The up-to-date location of an object is always kept at some node(s) in the system. This approach is implemented in ABC++ [4]. Few other systems use uncommon methods for location management, usually developed for specific applications (e.g., the *arrow* protocol implemented in the Aleph toolkit [18] supports exclusive access to objects using the spanning tree based directory).

Distributed Shared Memory (DSM) systems also have to address the issue of LM. Li and Hudak [25] describe *page ownership* in the context of sequentially-consistent shared virtual memory. The two main approaches they suggest are *centralized* and *distributed* memory managers. The centralized manager is analogous to the centralized directory mentioned earlier. Distributed page ownership algorithms are further divided into fixed and dynamic. With the fixed distributed management each processor is assigned a subset of pages (similar to DASH [24] and Memnet [13]). The up-to-

date location of a given page is always known by the predefined page owner. Dynamic distributed page management is similar to forwarding.

Location management is very important in wireless networks, especially in cellular phone networks. In cellular networks mobile users are tracked using the *two-tier scheme* [29]. A location database, called Home Location Register (HLR), is predefined for each mobile terminal. HLR is always aware of the exact locations of the users assigned to it. Considering frequent movements of users, at a given time the recipient's HLR may be distant from the user initiating communication. A Visitor Location Register (VLR) is maintained for each geographical area of a predefined size. The purpose of VLR is to provide location services, thus, communication, among users inside this area without querying HLR.

A number of improvements to the existing standard have been suggested. The proposed modifications attempt to reduce the location costs by either reducing communication with HLR or by minimizing the paging costs [2]. Existing proposals include caching [19], profile replication [33], taking advantage of the communication infrastructure hierarchy [22, 29]. The reader is referred to [2, 29] for comprehensive surveys of location management approaches in mobile networks.

The last of the reviewed areas in which location management is considered important is *mobile agents computing*. Mobile agent is an independent piece of code and/or data. It can be taken from the execution context on one host, migrated to a different machine, and continue execution there after migration completes. Some applications of mobile agents require communication between the agents, which makes location management an important issue [8, 35]. The previously mentioned centralized and forwarding location management techniques are often used in locating mobile agents. Ajanta [34] implements a two-tier home-based location management scheme, similar to the one used in cellular communication. A number of modifications of the centralized scheme are described in [35]. Voyager platform [1] uses forwarding pointers in combination with home server for agent location. Alouf et al. [3] showed that centralized server performs better than forwarders on a LAN, but not on a wide-area network. Cao et al. summarize experience with mobile agents location management by introducing the concept of a *mailbox* in [8]. The authors generalize most of the existing approaches and introduce new classes of location protocols based on the communication requirements for different classes of mobile agents applications.

## 3. Object-based Computations

We study location management in the context of object-based parallel distributed computing (PDC). We assume

that objects contain data defined by the application and represent its workload. This computation model proved to be effective in implementations of dynamic load-balancing for irregular problems in adaptive asynchronous applications. Our research is motivated by one such implementation, PREMA, which we use for dynamic load-balancing in mesh generation applications [7, 10, 28].

PREMA consists of two components. First component, *Clam* [14, 16], provides one-sided communication (remote service request and remote memory operations) and mobile object abstractions (the terminology was previously introduced in [11]). A mobile object is an application-defined data structure properly registered with PREMA (or with *Clam*) in order to make it “mobile”. There is only one instance of each mobile object within the system. A *mobile pointer* is a data structure created by PREMA for a given mobile object, which can be used on any of the processors to reference the corresponding object. As with regular pointers, there is no limit on the number of mobile pointers for a given mobile object. Mobile pointers are used for communication with mobile objects by sending *messages* to those objects. The location management module of *Clam* provides location-independent message delivery.

The second component of PREMA, Implicit Load-Balancing library (ILB) [6, 7], implements a variety of dynamic load-balancing algorithms. ILB uses the mobile object abstraction provided by *Clam* to create *schedulable objects*. A schedulable object represents application-defined unit of work. A typical application which is using ILB would over-decompose the initial problem, create mobile objects, define handlers which perform computation on those objects and assign weights to those handlers according to their expected execution time. Computation proceeds as a sequence of invocations of message handlers on user-defined objects, while ILB evens the computation load by migrating schedulable objects among processors.

Throughout this paper our assumptions are that the number of objects  $N$  is higher than the number of nodes  $P$  ( $N \gg P$ ); we do not have object replication; we have fixed non-faulty resources allocation; all nodes can communicate over all-to-all overlay network; the patterns of object migration and communication are unpredictable in the general case; each mobile pointer contains the ID of the “home” processor, where the referenced mobile object was created.

The problem of location management in object-based distributed computing is distinct from the one in the areas surveyed in the previous section. In DSM systems location management concerns with finding the owner of a shared page. However, DSM systems allow page replication, which should be considered in the design of a location mechanism.

Mobile networks have significant differences in the model: it is possible that the exact location of a given termi-

nal is not known at either of the location registers (paging is required). The migration options for a terminal are limited in the general case by the neighboring geographical regions, cells. Moreover, there are certain limitations on movement speed and cell size [27], which restrict migration to some extent. This is not the case for distributed computing, where communication and migration limits are imposed only by the dynamic properties of the communication links. Mobile networks usually have hierarchical structure and there are certain nodes dedicated to location management. In distributed computing all nodes of a cluster, or a collection of clusters, usually have equal roles. Applications can rarely have access to the intermediate routers in the hierarchy.

Mobile agents also operate on a different from the typical PDC kind of environment: in most cases it is WAN with dynamic number of participating nodes. Such applications should be able to handle faulty communication and hosts, other dynamic properties of the Internet, which impose specialized requirements on applicable location management techniques.

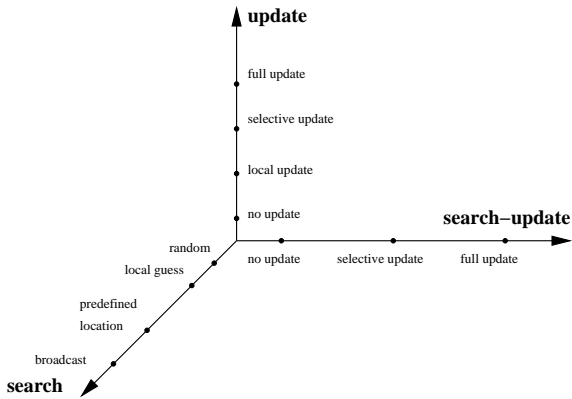
#### 4. Location Management Policies

A Location Management Policy (LMP) defines rules for 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. *Search-update* defines the procedure of updating the location directory after the message delivery. *Location directory* is a distributed data structure managed by a LMP. It maps each mobile object onto its possible or exact location processor. The local to a specific processor part of the distributed location directory is called a *local directory*.

The location management design space is shown in Figure 1. Each axis represents a LMP operation and the corresponding options arranged in the order of increasing complexity. Migration of an object may result in one of the four possible *update* scenarios:

- no update (no communication);
- update of the local directory (no communication);
- update of the local directories at selected sites (limited communication);
- global update of the distributed directory (broadcast).

Procedures which maintain *exact* location information at all sites are prohibitively expensive. We allow location information stored at a local directory to be outdated. Therefore, in our case the local directory information is just a *location guess* which was correct at some point in the past. A message directed to an object results in a point-to-point



**Figure 1. Design space for location management policies development.**

message sent to a processor, where that object can *possibly* reside. If the guess was wrong, that message triggers another point-to-point message, i.e., the message is forwarded. Thus, the *search* operation is essentially a process of routing a message toward the processor where the searched object is located along some *forwarding path*. We guarantee that any given forwarding path does not create cycles by keeping logical timestamps for each directory entry, as proposed by Fowler in [17]. Hence, each message will always make progress toward the target object. We distinguish four general options for implementing the *search* operation. They differ by the choice of the recipient(s) for the *search* message:

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

*Search-update* is an optional procedure in a LMP. Its purpose is to reduce the costs of subsequent accesses to the object. 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 that another message will be sent to it again. It should be clear from the earlier discussion, that a message may have to be forwarded regardless of the *update* technique used. The purpose of the *search-update* operation is to minimize the forwarding path length. This can be achieved by updating some of the local directories with a newer location guess upon the successful delivery of the message. This procedure will result in shortening of the forwarding path. Two *search-update* strategies have been proposed and evaluated in [17]. The choices for the *search-update* are similar to those for the *update* operation.

**Table 1. Summary of the implemented location management policies.**

LMP	Update	Search	Search-update
LF	yes, local	local guess	no
JU	yes, local	local guess	yes, sender
PC	yes, local	local guess	yes, selective
BU	yes, all	local guess	no
EU	yes, local and "interested"	local guess	no
HB	yes, local and "home"	forward to "home"	no

In general, a location management policy 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, 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).

Six 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 considered. Second, the selected 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 summary of the implemented LMPs, which is also summarized in Table 1 with respect to the previously discussed LMP operations. The reader is referred to [14] for a more comprehensive description.

**Lazy Forwarding (LF):** the simplest forwarding protocol. Messages are routed following forwarding pointers stored in local directories. When an object moves, only the local directory is updated. LF has minimal migration cost. The significant disadvantage is that the length of the forwarding path is bounded by the total number of processors only.

**Jump Update (JU):** similar to LF, but if a message had to be forwarded to reach the object, an update is sent back to the processor from where the message originated (similar to caching in mobile networks). The cost of subsequent

message to the object from the same processor is reduced by at least one hop.

**Path Compression (PC):** similar to JU, but the update message is propagated to all of the processors which the original message passed during forwarding.

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

**Eager Update (EU):** uses the idea of profile replication, originally proposed for cellular phone networks [33]. 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 the object migrates, all of the processors from the profile are updated with the new location, and the profile is reset.

**Home-Based (HB):** each mobile pointer contains an ID of the processor where that pointer was created (“home” ID). 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.

JU LMP is the most popular technique for location management in PDC runtime systems. HB LMP is also used in some of the implementations. LF, JU and PC LMPs were studied in [17] using theoretical analysis and simulations. To the best of our knowledge, there is no mobile object runtime environment which would provide a choice of LMP to the application. We are also not aware of any work which would concentrate on the evaluation and comparison of the selected LMPs, or otherwise evaluate the importance of location management.

## 5. Experimental Evaluation

We performed the experimental study on SciClone Cluster of The College of William and Mary. We used Typhoon (64 nodes, Sun Ultra 5, single UltraSPARC Iii 333/2MB, 256MB mem) and Whirlwind (64 nodes, Sun Fire V120, single UltraSPARC Iie 650/512KB, 1GB mem) subclusters. Typhoon nodes are divided into two groups of 32 nodes, with FastEthernet switch connecting the nodes within each group. Whirlwind nodes are connected to the FastEthernet switch. Typhoon and Whirlwind FastEthernet switches are connected by GigabitEthernet link. All nodes are running Solaris 9. All applications were implemented within the PREMA framework with communication over LAM MPI 7.0 with `tcp` RPI.

### 5.1. Applications

**Parallel sorting micro-benchmark (*netsort*)** Parallel network sort benchmark (*netsort*) implements a bitonic

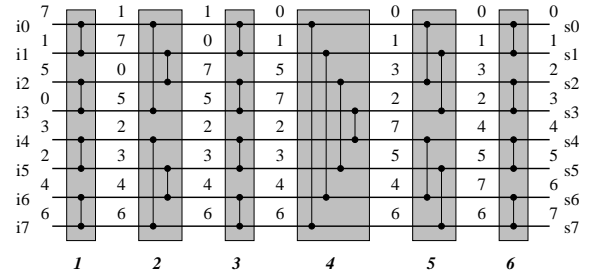


Figure 2. Sorting network for eight inputs.

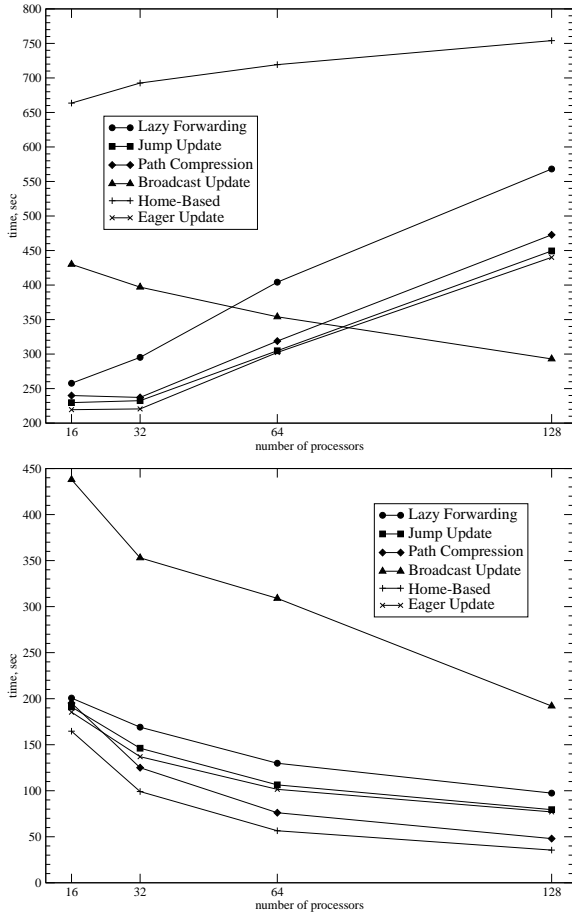
sorting network [12]. The process of sorting a sequence of eight numbers using sorting network is illustrated in Figure 2. Through a series of comparisons and exchanges, the input sequence  $i$  transforms into the sorted sequence  $s$ . Each of the shaded regions corresponds to a stage. The comparisons within each stage can proceed concurrently. The reader is referred to [14] for the detailed description of the benchmark implementation. Two versions of *netsort* have been implemented: in *netsortC* (centralized) all mobile objects are created on the same processor, while in *netsortD* (decentralized) each processor creates an equal share of mobile objects.

It is important to note, that *netsort* benchmark has been developed with the purpose of simulating *communication-intensive tightly-coupled application*. The benchmark was not designed to achieve high performance and speedups for parallel sorting, but rather to have the ability to vary the mobility parameters. *netsort* benchmarks do not use ILB. They are implemented directly on top of the mobile object functionality provided by *Clam*.

#### Parallel Constrained Delaunay Triangulation (*pcdt*)

The *pcdt* code simulates a parallel 2-D *mesh generation* algorithm based on Delaunay triangulation [32]. The reader is referred to [10] for the detailed description of the algorithm and for the definitions of the related terms. The main difference of this algorithm from Delaunay triangulation is that the point cavity cannot expand across the predefined boundary. At the preprocessing stage 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 is inserted on the boundary, a “split” message is sent to the neighboring subdomain located on some remote processor.

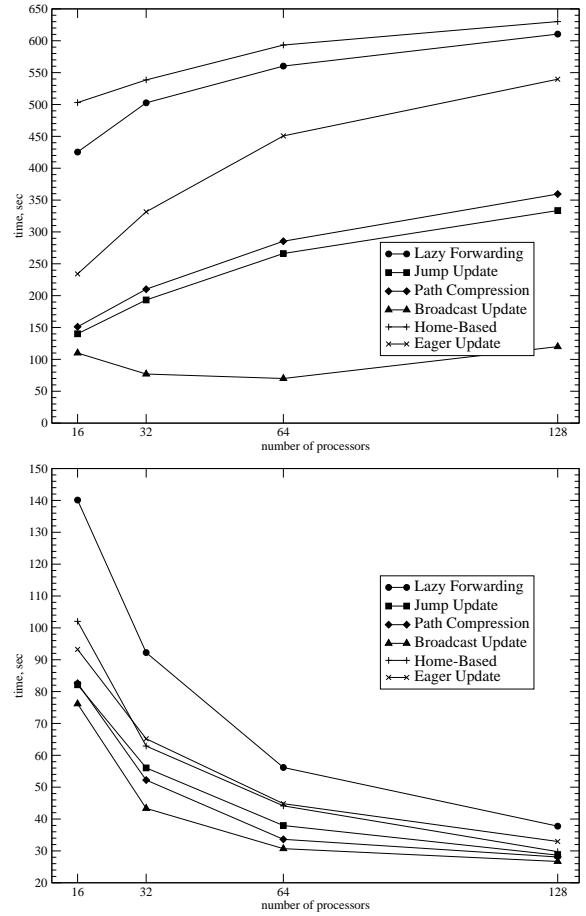
*pcdt* decomposes the initial domain and distributes resulting subdomains among the processors, which mesh the



**Figure 3.** *netsortC* (up) and *netsortD* (down) execution time,  $\lambda \approx 1$ ;  $P = 16, 32, 64, 128$ .

subdomains concurrently. A mobile pointer is associated with each subdomain. The benchmark operates on 512 subdomains. At each of the 20 iterations we randomly select 30% of the subdomains to be refined. The split messages are aggregated in groups of 50. The sources of load-imbalance in this application is the unpredictable level of refinement and communication required for a specific subdomain.

**Parallel Optimistic Delaunay Meshing (*podm*)** The *podm* benchmark simulates the process of 3-D mesh generation for restricted polyhedral domains [28]. Similarly to *pcdt*, this method is based on Delaunay refinement. The main difference is that the subdomain boundary can change during the process of refinement. During cavity expansion, certain mesh elements required for the cavity may be located on neighboring non-local subdomains. In such case, request for the element is sent. This request may fail, if the element is already acquired by some other cavity, which leads to a roll-back [20] of the expanding cavity. The cavity



**Figure 4.** *netsortC* (up) and *netsortD* (down) execution time,  $\lambda \approx 20$ ;  $P = 16, 32, 64, 128$ .

can complete only if all the non-local elements are successfully acquired. After the cavity is complete, updates to the non-local neighboring subdomains are sent. The non-trivial amount of communication with mobile objects in *podm* is tolerated by concurrent expansion of multiple cavities. In *podm* we have total of 343 subdomains, 30% of which will be refined two times more than the rest of subdomains subdomains. A subdomain “approaches” the required level of refinement with each successful completion of a cavity. The *podm* benchmark is described in detail in [15].

*pcdt* and *podm* use ILB for dynamic diffusion load-balancing [7], which creates object migration.

## 5.2. Experimental Results

We first ran *netsortC* and *netsortD* on different number of nodes using each of the implemented LMPs. For the configurations of up to 64 processors we used the Ty-

**Table 2. Message forwarding path length (ave/max) for netsort (64 processors).**

LMP	netsortC		netsortD	
	$\lambda \approx 1$	$\lambda \approx 20$	$\lambda \approx 1$	$\lambda \approx 20$
LF	8.1/34	3.2/13	8.3/34	2.5/13
JU	6.6/27	1.7/13	6.7/29	1.5/10
PC	4.0/16	1.4/7	4.1/17	1.3/8
BU	1.2/10	1.0/7	1.2/8	1.04/10
EU	6.2/26	2.1/13	6.3/26	1.7/11
HB	2.2/16	1.9/12	2.2/22	1.6/16

phoon sub-cluster with one process running on each node. 128-processor experiments used all of the nodes of the Whirlwind and Typhoon subclusters. We ran the benchmark with 4096 random numbers to sort (78 stages). All the messages and sortnodes were appended with the payload of 10Kb.

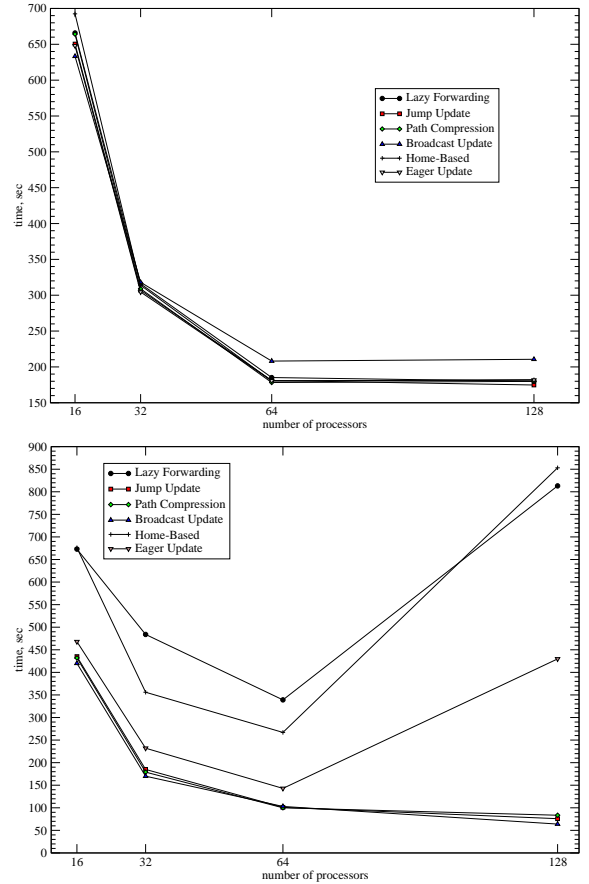
The total execution times of the `netsortC` and `netsortD` benchmarks using different LMPs are plotted in Figure 3. This experiment evaluates the impact of LM on the overall application performance. Results show, that for the centralized `netsortC` the total runtime grows when we increase the configuration for all of the policies. JU and EU LMPs give the best results, while HB LMP is the worst. In the case of `netsortD`, best performance is attributed to the HB policy, while BU is the worst.

Access-to-mobility ratio  $\lambda = \frac{a}{m}$  [17], where  $a$  is the total number of accesses (search operations) to objects and  $m$  is the total number of object migrations, is one of the parameters of an object-based distributed application. The `netsortC` and `netsortD` benchmarks have been modified to experiment with different values of  $\lambda$  and study the impact of those changes on the application performance. Figure 4 shows performance results of running `netsortC` and `netsortD` benchmarks with  $\lambda \approx 20$  (we achieve this by decreasing the number of object migrations).

In the last set of experiments we used `pcdt` and `podm` for the analogous evaluation of LMPs. The results are presented in Figure 5. We observe that no significant performance penalty is introduced by differences in LM in `pcdt`. At the same time, `podm` is very sensitive to the LMP used. JU, PC and BU give the best performance. LF and HB are the worst.

### 5.3. Analysis of the Results

In order to explain the behavior of LMPs for different applications, we need to understand the communication patterns of the applications and the parameters of the LMPs which affect the performance of a particular application.



**Figure 5. `pcdt` (top) and `podm` (down) execution time;  $P = 16, 32, 64, 128$ .**

The `netsort` benchmark is a tightly coupled (the subsequent sorting stage cannot proceed until the previous stage is complete), communication-bound application. Intuitively, LMPs which introduce minimum additional communication will suit `netsort` best. The message size in `netsort` is large relative to the size of the LMP *update* messages, thus short forwarding path length is crucial. Table 2 gives collected data on average and maximum forwarding path length for `netsort` on 64 nodes. The number of messages sent to objects is 327680, and it is equal to the number of object movements when  $\lambda \approx 1$ . When  $\lambda \approx 20$  the total number of moves is about 16384.

The first set of experiments shows the impact of the initial data distribution on the application performance. In `netsortC` all mobile objects are created on the same processor. Mobile pointers corresponding to those objects are then distributed among other processors. In the absence of the initial location data for a given mobile pointer, messages sent to mobile objects will always be directed to the “home” processor of those objects. The single for all ob-

jects “home” has to do significantly more communication than the other processors, which leads to bad scalability of most of the LMPs [14]. However, this is not the issue for BU LMP, as after the objects are distributed, all processors are aware of the objects’ locations (this makes the initialization stage of BU in `netsortC` expensive though). Although the cost of the *update* operation grows with the number of processors, the total number of object movements per processor decreases proportionally. We operate on the same problem size, thus the total number of messages sent from each processor decreases too, this makes BU LMP scale for `netsortC`. The performance of HB LMP does not change significantly, because all the communication is always serialized through the single “home” processor. Summarizing, the performance of `netsortC` with  $\lambda \approx 1$  is influenced by the initial data layout for most of the studied LMPs.

The data distribution of `netsortD` is balanced, as each processor acts as the “home” for an equal share of mobile objects. Because of this, HB LMP gives the best results when  $\lambda \approx 1$  (see Figure 3). HB LMP guarantees relatively short forwarding path, and it also has low update costs. The performance of BU LMP is slightly better compared to `netsortC`, as the initial distribution stage and corresponding updates are absent. The performance of LF, JU, PC and EU LMPs is defined by the average length of the forwarding path, shown in Table 2. HB LMP gives the best performance, as the load of location management is perfectly balanced among the processors, and HB provides next-to-the-shortest average forwarding path length (see Table 2).

Our data from Figure 4 show the impact of the access-to-mobility ratio on the application performance. BU LMP achieves the best execution times for `netsortC` when  $\lambda \approx 20$ , while for the same test with  $\lambda \approx 1$  it had poor performance. The total number of movements decreased. The *update* operation is very expensive for BU LMP, and higher values of  $\lambda$  allow for better amortization of the *update* costs. These costs become dominant with large number of processors, which leads to performance degradation with 128 nodes. The significant difference between EU and JU LMP can also be observed. EU LMP sends updates when object moves, while JU updates the sender immediately after the message arrives. This results in a shorter message forwarding path for JU than for EU LMP, as Table 2 suggests. EU LMP profile is based on *processors* which send messages, while communication is induced by *mobile objects* which do not have fixed processor location. The change in  $\lambda$  does not significantly affect 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 same experiment with  $\lambda \approx 20$  for `netsortD` shows the reduction in the execution time for all LMPs. BU LMP again performs much better than in `netsortD` with  $\lambda \approx 1$  for the same reason as in `netsortC`: the rela-

**Table 3. Mobile object migration and communication data (ave/max) for `pcdt` and `podm` (64 processors); *fw distr*: forwarding path length, *mv distr*: object movements, *mv tot*: total movements, *ms tot*: total messages.**

App	LMP	<i>fw distr</i>	<i>mv distr</i>	<i>ms tot</i>	<i>mv tot</i>
<code>pcdt</code>	LF	1.99/8	1.83/7	57382	940
	JU	1.36/7	1.91/6	57382	980
	PC	1.35/5	1.95/7	57382	1002
	BU	1.10/3	1.64/6	57382	840
	EU	1.53/6	1.88/6	57382	966
	HB	1.67/6	1.79/7	57382	920
<code>podm</code>	LF	2.85/15	5.8/19	6987898	2003
	JU	1.10/17	15.1/36	6999244	5173
	PC	1.07/9	15.7/40	6993058	5391
	BU	1.05/5	16.8/41	6992392	5783
	EU	1.52/12	11.8/38	6993824	4064
	HB	1.61/21	14.4/47	6997755	4942

tive cost of *update* becomes negligible. The performance of LMPs is defined by the average forwarding path length.

The performance data in Figure 5 shows that the differences in LM do not impact the performance of `pcdt`. From Table 3 we can conclude that  $\lambda \approx 57$  for `pcdt`. Most of the messages directed to mobile objects are *split* messages. No messages in `pcdt` require replies. Communication costs and computation costs of handling *split* messages are insignificant compared to the costs of processing *refine* messages (one per subdomain). Moreover, the algorithm correctness does not change if all of the *splits* are aggregated and sent at the end of the computation as a single message. `pcdt` is a computation-defined, partially coupled application [14]. Because of this we observe low migration rate, short forwarding paths (see Table 3) and consequently insignificant impacts of location management.

`podm` is different from `pcdt`: it is communication-intensive and tightly coupled. It also has higher access-to-mobility ratio: between 1209 and 3488. Only a limited number of concurrent cavities is allowed within a subdomain. The cavity completion time depends on how timely the responses for remote elements are received. Although the message size never exceeds 5Kb, `podm` generates about 7M of those messages. The `podm` message handlers have similar computational complexity. For a comparison, 512 of `pcdt` handler invocations (*refines*) have complexity of 1000 relative to 1 of 56870 *splits*. This leads to more objects being migrated by the load-balancer (see Table 3). Another interesting observation is that the total number of messages sent in `podm` varies for different LMPs and is the same for `pcdt`. The explanation is that there is no non-determinism

in `pcdt`, while for `podm` the network conditions influence the probability of success for a distributed cavity element request.

The value of  $\lambda$  is high for `podm`, which makes LMPs like BU and PC with relatively expensive *update* operation more beneficial even with high number of nodes. Overall, the LMPs which provide shorter forwarding path give better performance. Interestingly, EU performs worse than JU LMP, while from its description it is clear that EU updates all message senders with the new location during migration. However, the updates sent by EU are delivered to *processors*' local directories. Similarly to `netstort`, this is not effective as communication is induced by migrating mobile objects. While HB LMP provides short average forwarding path, it is still significantly higher than for JU and BU (most of the messages will travel two hops: to and from "home"). HB LMP also gives the highest maximum forwarding path value because of the high communication and migration rate of the application (a message forwarded from "home" is likely to be sent to the out-of-date location).

The location mechanism used can significantly affect the intensity of object movements during load-balancing, as it becomes apparent from Table 3. The reason for this is that the studied LMPs create different additional communication induced mostly by forwarding. This prevents timely delivery of the load-balancing messages and results in poor decision making of ILB when the network traffic is high. The reason for this is not only the network congestion, but also the serialization of all application, ILB and LMP communication over the same MPI point-to-point communication channel [14].

The LMPs we analyzed pursue different trade-offs between the *search* and *update* implementations. Applications which exhibit high  $\lambda$  values require minimal costs of the *search* operation. Because of this they benefit from the LMPs with effective expensive *update* and *search-update* mechanisms, like Broadcast Update LMP.

## 6. Conclusions and Future Work

The study we described in this paper shows the importance of location management in tightly coupled communication-intensive applications. The performance of the `netstort` and `podm` benchmarks can significantly be affected by the LMP used. At the same time, partially coupled applications like `pcdt` are not susceptible to the differences in location management technique.

Our study has broader impact as our findings are not limited to PREMA and mesh generation applications and can be applied to larger class of applications with similar properties. The study confirms that Jump Update LMP conventionally used in distributed object runtime systems gives good performance in most of the experiments: it guarantees

short forwarding path, is relatively not expensive for a wide range of applications with various  $\lambda$ , and has good scalability. However, we can also conclude that techniques with expensive *update* implementation, like BU and PC, are more beneficial than JU in tightly coupled applications with high access-to-mobility ratio, which is often the case in parallel distributed computing. We also establish that the processor-based profiling similar to EU LMP is not effective for parallel distributed applications even with low mobility rate.

The LMPs we studied generate different amount of messages, which affects the efficiency of the ILB decision-making. It will be interesting to study how this impact can be tolerated within PREMA. We are also interested in developing location management techniques which would address object-based vs processor-based profiling. We believe such techniques can be effective in applications where each object communicates with the limited number of other objects. One of the limitations of our study is that the `podm` benchmark is a scaled-down model of the real application. The impact of over-decomposition and load-balancing have not been studied for Optimistic Delaunay mesh generation yet. We need to revisit the issue of location management within the real application, where its impacts are likely to be even more drastic.

## Acknowledgments

Many thanks to Kevin Barker and Andrey Chernikov for their help with ILB and `pcdt`, Demian Nave and Chaman Singh Verma for the application data and discussions about `podm`, Chris Hawblitzel for the initial version of `netstort`, and Tom Crockett for his invaluable assistance with Sci-Clone.

This work was performed using computational facilities at the College of William and Mary which were enabled by grants from Sun Microsystems, the National Science Foundation, and Virginia's Commonwealth Technology Research Fund.

The comments from anonymous referees helped us to improve the presentation quality of the paper.

## References

- [1] Voyager. <http://www.recursionsw.com/products/voyager> (10/23/2003).
- [2] I. Akyildiz, J. McNair, J. Ho, H. Uzunalioglu, and W. Wang. Mobility Management in Next-Generation Wireless Systems. *Proceedings of the IEEE*, 87(8):1347–1384, 1999.
- [3] S. Alouf, F. Huet, and P. Nain. Forwarders vs. Centralized Servers: An Evaluation of Two Approaches of Locating Mobile Agents. *Performance Evaluation*, 49(1–4):299–319, 2002.

- [4] 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.
- [5] B. Awerbuch and D. Peleg. Online Tracking of Mobile Users. *Journal of the ACM*, 42(5):1021–1058, 1995.
- [6] K. Barker. *Runtime Support for Load Balancing of Parallel Adaptive and Irregular Applications*. PhD thesis, The College of William and Mary, 2004.
- [7] 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):183–192, 2004.
- [8] J. Cao, X. Feng, J. Lu, and S. K. Das. Mailbox-Based Scheme for Mobile Agent Communications. *Computer*, 35(9):54–60, 2002.
- [9] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. 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, 1989.
- [10] P. Chew, N. Chrisochoides, and F. Sukup. Parallel Constrained Delaunay Meshing. *AMD-Vol. 220 Trends in Unstructured Mesh Generation*, pages 89–96, 1997.
- [11] 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.
- [12] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.
- [13] G. S. Delp. *The Architecture and Implementation of Memnet: A High-speed Memory Computer Communication Network*. PhD thesis, University of Delaware, Newark, 1988.
- [14] A. Fedorov. *Location Management in a Distributed Object Runtime Environment*. MS thesis, The College of William and Mary, 2003.
- [15] A. Fedorov. podm: Parallel Optimistic Delaunay Meshing Benchmark. PES Memo #04-01, <<http://www.cs.wm.edu/~fedorov/pdf/memo04-01.pdf>> (04/23/2004), 2004.
- [16] A. Fedorov and N. Chrisochoides. Communication Support for Dynamic Load Balancing of Irregular Adaptive Applications. Accepted for publication in *Proceedings of the 2004 International Conference on Parallel Processing Workshops (CRTPC04)*, 2004.
- [17] R. J. Fowler. *Decentralized Object Finding Using Forwarding Addresses*. PhD thesis, University of Washington, 1985.
- [18] M. Herlihy and M. Warres. A Tale of Two Directories: Implementing Distributed Shared Objects in Java. *Concurrency and Computation: Practice and Experience*, 12(7):555–572, 2000.
- [19] 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, 1994.
- [20] D. A. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.
- [21] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.
- [22] 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.
- [23] O. Lawlor and L. V. Kalé. Supporting Dynamic Parallel Object Arrays. *Concurrency and Computation: Practice and Experience*, 15(3–5):371–393, 2003.
- [24] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 148–159, 1990.
- [25] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.
- [26] B. Liskov, M. Day, and L. Shrira. Distributed Object Management in Thor. *Distributed Object Management*, pages 79–91, 1993.
- [27] P. Müller, F. van Meegen, 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%20Location%20Service.pdf>> (10/23/2003), 2001.
- [28] D. Nave, N. Chrisochoides, and P. Chew. Guaranteed-Quality Parallel Delaunay Refinement for Restricted Polyhedral Domains. In *Proceedings of the 18th Annual ACM Symposium on Computational Geometry*, pages 135–144, 2002.
- [29] E. Pitoura and G. Samaras. Locating Objects in Mobile Computing. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):571–592, 2001.
- [30] M. Powell and B. Muller. Process Migration in DEMOS/MP. *ACM SIGOPS Operating Systems Review*, 17(5):110–119, 1983.
- [31] M. Shapiro, P. Dickman, and D. Plainfossé. SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection. Technical Report 1799, INRIA, 1992.
- [32] J. R. Shewchuk. Tetrahedral Mesh Generation by Delaunay Refinement. In *Proceedings of the 14th Annual ACM Symposium on Computational Geometry*, pages 86–95, 1998.
- [33] 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.
- [34] 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, 2002.
- [35] P. Wojciechowski. Algorithms for Location-Independent Communication between Mobile Agents. Technical Report DSC-2001/13, Département Systèmes de Communication, EFPL, 2001.