Object Distribution with Local Information

Bujor D. Silaghi
Department of Computer Science
University of Maryland
College Park, MD 20742
bujor@cs.umd.edu

Peter J. Keleher
Department of Computer Science
University of Maryland
College Park, MD 20742
keleher@cs.umd.edu

Abstract

We investigate the problem of distributing communicating objects across wide-area environments. Our goals are to balance load, minimize network communication, and use resources efficiently. However, applications running in such environments are often dynamic and highly unpredictable. Furthermore, synchronous communication is usually too expensive to be used in disseminating load information. We therefore investigate policies that use local information to approximate desired global behaviors. Our results with Java applications show that simple, local approaches are surprisingly effective in capturing load information and object relationships, and in making migration and clustering decisions based on profiled information.

1. Introduction

Long-lived or persistent distributed applications must be dynamically reconfigurable in order to run efficiently in metacomputing environments. Metacomputer environments are often characterized by distribution, heterogeneity, and varying resource capacities. Reconfigurability can be explicit in an application's structure. However, this approach is unlikely to be portable, and places a large burden on application developers. A more general approach is for the runtime system to implement reconfiguration transparently to the application.

This paper presents a general inquiry into the problem of object placement in such environments. Large-scale object systems with hundreds of thousands of communicating objects are considered, a paradigm that is likely to become increasingly common in the future. We focused our experimental evaluation on Java application traces, yet any testbed environment exposing such features could provide us with similar insights.

Creating a good mapping of objects to nodes requires several distinct steps. First, we must be able to evaluate the load distribution of a given mapping. This generally requires a way of estimating objects' computational needs and nodes' computational capacities. Such an evaluation must account for both parallelism and load balance. Second, we must be able to evaluate and minimize a mapping's communication cost. This problem reduces to that of co-locating communicating objects. Neither parallelism maximization nor communication minimization can proceed in isolation.

Object placement in a network of processors such that both the processors and the network are efficiently used is a hard problem. The optimal distribution of computational and communicative entities on a network of processing elements, usually referred to as the graph-embedding problem, is known to be NP-complete [4]. Researchers have taken two approaches in dealing with this complexity [3]: designing sub-optimal distribution policies (but with certain proven properties) for fairly general configurations, and using optimal policies for particular application/processor configurations. The latter approach is well-suited for applications with known behavior and which are designed to run on specific processor configurations (trees, grids, hypercubes, etc.). The first approach is less studied in the literature, and most solutions rely on restrictions to either the applications or environment.

This paper presents a broad inquiry into object profiling schemes, and policies for deciding when and where to migrate or cluster objects in systems with many objects. More specifically, we evaluate a broad range of approaches in the context of the following goals:

- *accommodating dynamic applications and environments* - Both the applications and the set of available processors may change during an execution.
- *no reliance on a priori application information* - Our policies have no a priori information about application communication patterns.
- *scalability* - We seek to avoid bottlenecks by only considering decentralized algorithms.

No previous work, to our knowledge, has taken a similar approach in trying to solve the general distribution problem. The main contribution of this paper is in showing that all the above requirements can be met by employing fairly simple decision making policies, and that the implementation of these policies in real systems is feasible.

2. The simulation framework

The main actors of the simulation are objects and a set of connected processors on which objects execute. The processor configuration and the network topology are specified in an environment configuration file.

Time is divided into discrete ticks. Each tick is simulated by performing a tick's worth of work on each node in the system in round-robin order. A simple language allows network topology customization: links between processor can be added, removed or their latency changed. Our default network topology is a switch, with contention modeled on destination links. We also investigate the impact of a broadcast interconnect, with contention modeled in all phases.

The cost to send a message between processors p_1 and p_2 is given by:

$$cost = \frac{os\_cost}{speed(p_1)} + lat(p_1, p_2) + \frac{os\_cost}{speed(p_2)} \quad (1)$$

where $lat(p_1, p_2)$ is the link latency from p_1 to p_2. The first and last terms are intended to model middleware and operating system occupancy. The cost of sending a local message is always 1. An invocation message may trigger the migration of the sender object, and in such cases we scale $cost$ by a constant factor.

2.1. The object model

Our object model includes three basic assumptions. First, we assume globally-unique object identifiers. Second, we assume that every object can be executed on any processor. Finally, all of our objects are *active* [1], i.e. they each have an associated local thread.

Each object performs a sequence of alternating computational and communication phases. A computational phase only specifies the length of the computation, whereas a communication phase involves a local or remote object. By default, sends are asynchronous in that the local thread is allowed to proceed immediately after the message is copied into a system communication buffer. The effects of *blocking sends* are discussed as well. Receiving messages is a blocking operation.

Since the Java object model is different than the one used by the simulator, part of preparing an application trace is mapping between the two models. This is done according to the following invariant: a simulator object thread will execute all the instructions executed in the context of the corresponding Java object by any Java thread, and only those.

2.2. Application Traces

The input data to the simulator are application traces obtained by running various applications on an instrumented Java Virtual Machine 1.2.2. The following *events* are relevant for an object trace record: *compute* (records the amount of computation performed on behalf of the object), *send* (consists of the object identifier of the invoked object), *receive* (records the object identifier of the invoker), *new* (specifies the identifier of the newly created object).

A *compute* event takes a number of clock ticks equal to the specified amount of computation divided by the executing processor's speed. All other events take one clock tick to execute on any processor.

2.3. Applications and processor configurations

We used six applications which we believe are representative for the Java environment. Following is a short description of the application suite.

$\mathbf{A_{jmark}}$ Is the JMark 2.0 benchmark for Java virtual machines.

$\mathbf{A_{elite}}$ e-Lite is a multi-threaded internet browser from the ICE Browser family.

$\mathbf{A_{seqconc}}$ Denotes a mixture of mathematical algorithms, each implemented using the fork-join parallel decomposition paradigm [10].

$\mathbf{A_{parconc}}$ Same as above but with the individual algorithms launched in parallel as Java threads.

$\mathbf{A_{vchat}}$ Is the server side of the VolanoMark 2.1 benchmark.

$\mathbf{A_{vmark}}$ Denotes the multi-client side of the VolanoMark 2.1 benchmark.

Some of the applications are highly communicative while others exhibit a more computational behavior. Table 1 gives a quantitative overview of this aspect. We show the number of objects created during the run (Objects), the total number of events over all objects (E), the amount of computation performed by the application (C), and the number of messages exchanged by its objects (M).

Table 2 shows the processor configuration parameters used in our experiments. *Max spd* is the maximal speedup by using all processors over the first processor in the lot.

Table 1. Application parameters. Large numbers are rounded.

App	Objects	E $[10^6]$	C $[10^6]$	M $[10^6]$
A_{jmark}	$524,256$	61	$2,600$	15
A_{elite}	$1,043,580$	38	540	9
$A_{seqconc}$	$1,336,873$	23	$8,500$	5
$A_{parconc}$	$1,337,260$	23	$8,500$	5
A_{vchat}	$65,832$	98	660	24
A_{vmark}	$433,906$	46	475	11

Table 2. Processor configurations.

Proc	No procs	Max spd	Applications
P_{13}	13	12	A_{jmark}, A_{elite}
P_{13}	13	12	A_{vchat}, A_{vmark}
P_{30}	30	30	$A_{seqconq}, A_{parconc}$

3. Algorithms and policies

Object-placement algorithms need to address three issues: parallelism, load balance, and communication. There are three situations in which object placement policies are invoked. First, we evaluate the target of each message as a potential new host for the message's source. If the migration is deemed acceptable, the source object moves to the target prior to sending the message. Second, entire underloaded machines are candidates for eviction, and overloaded machines are candidates to be split among multiple machines. Finally, we consider placement of new objects.

We divide the overall problem into several sub-problems, as follows: load estimation, load information dissemination, target suitability with respect to communication, target suitability with respect to load balance and finally, object placement and clustering.

3.1. Estimating processor loads

Load estimates are used for determining whether individual migrations upon method invocation are acceptable, for determining when to "evict" objects in group from underloaded or overloaded hosts, and for determining initial object placement. The following are the three basic load measures used in this paper:

$\mathbf{L_{res}}$ The number of resident objects divided by the processor speed.

$\mathbf{L_{ready}}$ The number of ready, unblocked objects.

$\mathbf{L_{prof}}$ Define t^o_{comp} as the number of ticks that object o has spent computing. Define t^o_{comm} as the sum of all clock ticks that o has spent between being blocked on receives and the subsequent arrival of messages for o. Let T_{comp} and T_{comm} be given by

$$T_{comp} = \frac{1}{s_{target}} * \sum_{o \in p} s_{source} * t^o_{comp} \quad (2)$$

$$T_{comm} = \sum_{o \in p} t^o_{comm} \quad (3)$$

where s_{source} is the speed of the processor where the data is profiled, and s_{target} is the speed of the processor which uses the collected data. Furthermore let l be given by

$$l = n_{obj} * \frac{T_{comp}}{T_{comp} + T_{comm}} \quad (4)$$

where n_{obj} is the number objects residing on the processor. Finally, we express the processor load as follows:

$$L_{prof} = \begin{cases} l, & l \geq 1 \\ -\frac{1}{l}, & l < 1 \end{cases} \quad (5)$$

We say that the machine is *overloaded* if $L_{prof} > 1$, and *underloaded* if $L_{prof} < 0$. If the machine is overloaded, L_{prof} gives the number of processors with the same speed that could efficiently execute n_{obj} objects. If the machine is underloaded the absolute value of L_{prof} gives how many times more objects than n_{obj} the processor would need in order to be efficiently used.

We implemented two variants of this profiling method: L^O_{prof} maintains counts for each object instance, whereas L^P_{prof} shares counts across all objects on the same processor.

3.2. Disseminating processor loads

Load information is only useful once disseminated to other processors. We evaluated the following alternatives:

$\mathbf{D_{oracle}}$ The oracle approach assumes that every processor has instantaneous and current access to the local load estimation of every other processor.

$\mathbf{D_{append}}$ This approach appends all local information about processor loads to outgoing messages. In a communicative system, this approach disseminates current load estimates to all nodes with little latency. A few variants of D_{append} have been examined.

3.3. Target suitability with respect to communication

The decision of whether to migrate an object to another host requires determining whether the prospective target is a *better* host than the current one. This question should ideally be answered from a global perspective, considering the effect of the migration on overall application execution time and network utilization. However, we constrain our decision policies to operate in the local domain because global synchronization is expensive in large-scale systems. We only consider objects referenced from the current object (the migration candidate), and their respective host processors.

Let $r_j(i)$ be the number of invocations by object j on object i. We denote object i being resident on node p as $i \in p$, and retrieve i's host processor as $h(i)$. For an invocation from object s to target t, we investigate the following policies (δ is an additive constant used to handle pathological conditions):

$\mathbf{M_{nomig}}$ No migration is allowed.

$\mathbf{M_{obj}}$ Migrates the source if the most invoked reference happens to be the current target object.

$$r_s(t) \geq \delta + r_s(i) \qquad \text{for all } i \neq t \tag{6}$$

$\mathbf{M_{all}}$ Migrates the source if the current target object has been invoked more than all other objects, cumulatively.

$$r_s(t) \geq \delta + \sum_{i \neq t} r_s(i) \tag{7}$$

$\mathbf{M_{ideal}}$ Migrates the source if the cumulative invocations to objects residing on the target processor is greater than the cumulative invocations to objects residing on all other processors.

$$\sum_{i \in \mathrm{h}(t)} r_s(i) \geq \delta + \sum_{j \notin \mathrm{h}(t)} r_s(j) \tag{8}$$

$\mathbf{M_{best}}$ Migrates the source if the cumulative invocations to objects residing on the target processor is greater than the cumulative invocations to objects residing on each of the other processors.

$$\sum_{i \in \mathrm{h}(t)} r_s(i) \geq \delta + \sum_{j \in p} r_s(j) \qquad \text{for all } p \neq \mathrm{h}(t) \tag{9}$$

$\mathbf{M_{bett}}$ Migrates the source if the cumulative invocations to objects residing on the target processor is greater than the cumulative invocations to objects residing on the source processor.

$$\sum_{i \in \mathrm{h}(t)} r_s(i) \geq \delta + \sum_{j \in \mathrm{h}(s)} r_s(j) \tag{10}$$

$\mathbf{M_{always}}$ Always migrates on invocations.

3.4. Target suitability with respect to load balance

The above policies migrate objects only if doing so does not violate load balance invariants. We distinguish four cases for potential migrations. Let l_s and l_t denote processor load estimations of the source and target. Then:

$l_s \geq 0 \wedge l_t < 0$. Migration occurs.

$l_s < 0 \wedge l_t \geq 0$. Migration does not occur.

$l_s \geq 0 \wedge l_t \geq 0$. Neither machine is underloaded. We migrate the invoker only if additional conditions regarding the load ratio of the source and target processors are met.

$l_s < 0 \wedge l_t < 0$. Both processors are underloaded and migration is questionable because either or both processors might be evicted in the near future. We favor the faster processor in this case.

3.5. Initial object placement

The initial object of an application is always created on processor zero. Other objects are created according to one of the following policies:

$\mathbf{H_{creator}}$ Newly created objects are placed on the same host as the creator.

$\mathbf{H_{least}}$ Newly created objects are placed on the least loaded processor.

$\mathbf{H_{nfree}}$ As above, but the least loaded processor is chosen given the following priorities: underloaded processors, used (but not underloaded) processors, and free processors.

3.6. Object clustering and eviction from underloaded or overloaded hosts

The eviction mechanism migrates objects in groups, either because the host processor must be abandoned due to inefficiency, or because the host processor is overloaded and some of its load must be transferred to some other processor.

- *underloaded processors* - We identify underloaded processors by profiling the average fraction of lost quanta during some past period or by using L_{prof}. All the objects residing on an underloaded machine have to be expelled. For each object, we select the machine with the greatest affinity (defined by M_{best}) and move the object to that processor.

- *overloaded processors* - We use L_{prof} to identify overloaded processors, compute the number of objects that have to be evicted, and determine the speed of the target processor. The target processor is chosen to be a free processor whose speed does not exceed the value thus computed.

  Objects to be evicted are selected based on a clustering policy: two partitions are created over the set of all objects, and one of them is moved in group on the target processor. We experimented with two clustering heuristics, both approximations of the minimal cut.

Processors are checked periodically for an underloading or overloading condition. These checks are more expensive than checks for single-object migration, yet they are performed much less frequently.

4. Experimental evaluation

We performed a number of experiments to assess the individual contributions of the various policies. All tests were performed in the simulated environment presented at the beginning of the paper, using Java application traces collected a priori.

The metric used to evaluate a policy configuration is the speedup relative to the speedup of alternative configurations, or to the maximal speedup theoretically achievable on the given architecture (see Table 2).

4.1. Load estimation

We evaluate load estimation methods in terms of initial object placement policies. As long as object creation rates do not dramatically drop below object destruction rates, load imbalance is a good indicator of the actual goodness of a load estimation policy.

The number of resident objects, L_{res}, performs well. This agrees with the results of other studies, who suggest that simple counts of active processes provide the best estimates [5]. In fact, L_{res} performs close to optimal ($10.5/12$) when applications exhibit the necessary degree of parallelism ($A_{seqconc}$ and $A_{parconc}$ run on P_{13}).

On the other hand, the number of ready objects, L_{ready}, performs poorly. We argue that this policy is not a reliable method of load estimation. The reason is that a combination of computationally-intensive objects and differing machine speeds can result in poor performance with this metric. Choosing highly communicative applications or relatively uniform execution environments gives L_{ready} a better chance for estimating the actual load. In our case, L_{ready} performs worst for $A_{seqconc}$ and $A_{parconc}$, both of which are computationally intense (see Table 1).

The L_{prof} metrics lead to poor performance due to their inability to collect enough data about the behavior of objects by the time most of the objects are created. This is very apparent for A_{vchat} and A_{vmark}, where most objects have been created before adequate profiling information has been collected (see Figure 2). For this reason we do not employ L_{prof} policies as direct load estimators. Instead, we use them to estimate processor overloading. L^{P}_{prof} performs slightly worse than L^{O}_{prof} in some of the cases, but on the average it proves to be a good approximation of L^{O}_{prof}.

The quality of load estimators is critical to good load balancing schemes. We contrasted our policies with an algorithm that is blind to processor workload, and performs a random initial placement of newly created objects. The degradation in performance varies from $1/2$ to as much as $1/4$ in some cases.

4.2. Load dissemination

We experimentally compared the two approaches for disseminating load information with a variety of policy configurations. In all tests the performance of D_{append} is remarkably close to that with perfect, global knowledge. In fact, sometimes it outperformed the oracle approach, and in one case it did so by 30%. There is therefore ample opportunity to spread information transitively, and this is true regardless of whether the application is relatively more or less dynamic.

It appears that immediate knowledge of actual load is less beneficial than finding processor loads with a slight inertia. We attribute this effect to the locality of reference principle from an object creation / communication perspective: related objects, as given by communication patterns, tend to be created closely in time. Knowing the exact load could hurt performance as newly created objects will be distributed on different processors. On the other hand, slight delays will allow these objects to be placed on the same processor, and help towards communication minimization. This side effect is less apparent for A_{vchat} and A_{vmark} which have larger messages/object ratios and more apparent for $A_{seqconc}$ and $A_{parconc}$.

To verify this claim we repeated all dissemination experiments with load dissemination disabled at object creation time. With a few exceptions, we indeed obtained additional performance improvements.

4.3. Target suitability with respect to communication and load balance

We investigate the policies used to evaluate targets in terms of communication suitability and load balance.

Unlike load estimation, complex policies for migration buy extra performance improvement over simpler ones. Ex-

cessively greedy policies like M_{obj} and M_{all} have a better ratio of local to network messages but incur too many migrations. Overall performance is thus hurt due to additional migration costs. In the extreme is M_{always} which generates an order of magnitude more migrations and pays the higher price. Our current migration cost is very conservative. Larger migration costs would disproportionately hurt the performance of this approach.

The other policies, M_{ideal}, M_{bett} and M_{best}, which try to capture object-processor and not object-object affinities, perform better, with M_{bett} performing best in a majority of cases. M_{bett} manages to improve performance up to 2.5 times over a non-migration setting with A_{jmark}, but no more than 1.3 with all other applications. We believe this is so because the iterative nature of A_{jmark} defines more clearly communication patterns than any other application. Object relationships are hard to infer since most of the objects are very short lived and they are gone by the time relevant data has been profiled.

We restrict migrations in trying to preserve load balance. For instance we are not willing to migrate from an underloaded processor to an overloaded one. We experimented with different combinations of acceptable load ratio for the source and target processor and found that performance is relatively insensitive to this aspect. The break-even point between allowing more migrations or preserving the load balance is not clearly defined for the applications considered.

4.4. Object eviction and clustering

The approaches described in previous sections distribute load over all available processors. When all nodes can be efficiently used, initial object placement combined with one of the migration policies can yield the expected results. A more enlightened policy might dynamically choose to use only those nodes that can be exploited efficiently.

This section investigates techniques that detect and eliminate both underloaded and overloaded processors. Underloaded processors are released by evicting all objects. Overloaded processors are de-populated by migrating objects to other nodes. Object clustering based on profiled data is performed in both cases.

With eviction we intend to model the number of used computational nodes after the application's degree of parallelism, and achieve load balance at the same time. Object distribution as given by initial object placement, fine-grained object migration, and object clustering will hopefully reduce communication constraints and help exploit inherent application parallelism. Figure 1 shows typical performance indices of eviction combined with migration and clustering for the test suite.

Combining object clustering with migration does in most of the cases further improve performance over eviction alone, most noticeably the case of $A_{seqconc}$ and $A_{parconc}$. Spectacular improvements based on online profiling of object communication are not to be expected since most of the objects in a typical Java application die young [12], before relevant clustering information can be collected. Exceptions are relatively static applications and A_{vmark} is one of them (see Figure 2). Indeed, for A_{jmark} migration and clustering can yield up to 75% improvement over eviction alone.

Parallelism is dependent upon synchronization constraints and the number of objects in the system. Profiles of the number of objects and used processors for two applications are shown in Figure 2. The number of used processors adapts quite well to the growing and shrinking of dynamic applications, and stabilizes with static applications.

Throughout the experiments we used L^P_{prof} to detect overloaded processors. Similar results were obtained with L^O_{prof}; we focused on the former since it is more appealing from an implementation point of view. A host was considered to be underloaded if it was in the idle state for more than 33% of some past interval.

4.5. Varying processor speed

In metacomputing environments resources are often non-dedicated. A runtime system similar to ours may have to contend with other processes for resources. We model contention by changing the effective processor speed during application execution. Due to the profiling nature of our policies no noticeable degradation in performance has been observed throughout the tests.

5. Related work

There is a large body of research on distributed systems based on object or component models. We concentrate on the ability of systems to support automatic object distribution.

Legion [6] is a metacomputer project designed to support wide-area distributed computing, based on earlier work on the Mentat object system. Rather than implementing system-wide policies, Legion provides a framework through which programmers can supply custom object-placement policies.

SOS [11] is an object-oriented operating system based on fragmented objects — each object consists of a provider and possibly multiple proxies. The system does not include automatic migration mechanisms, but custom policies for communication minimization can be specified by defining communication protocols between proxies and corresponding providers.

Emerald [2] is an object-oriented system (compiler and run-time system) with support for fine-grain, medium-grain

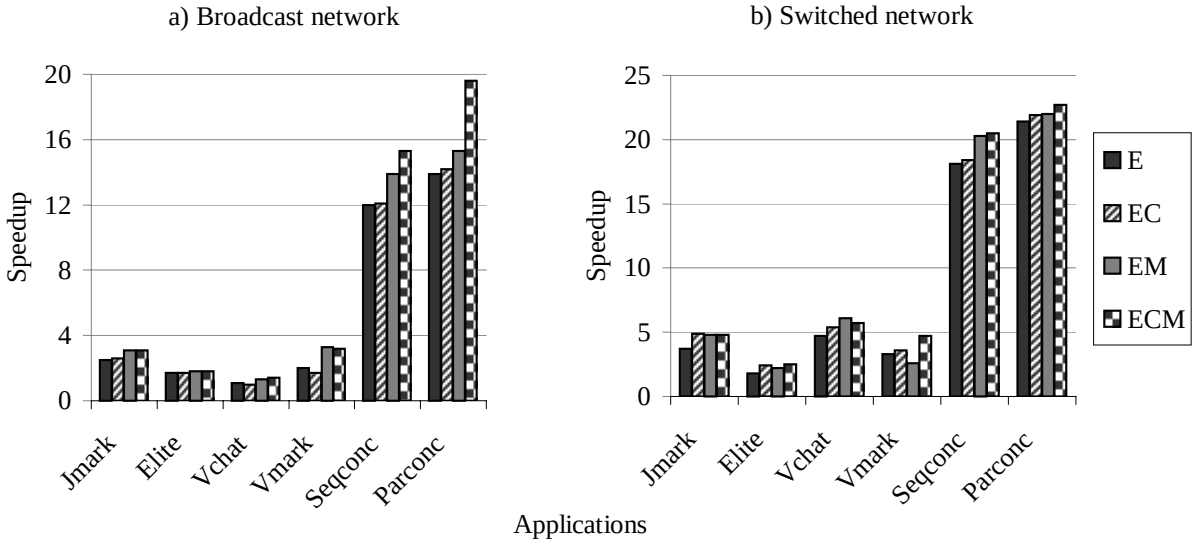

Figure 1. Example of speedups with different combinations of eviction (E), clustering (C), and migration (M) policies, for two types of networks.

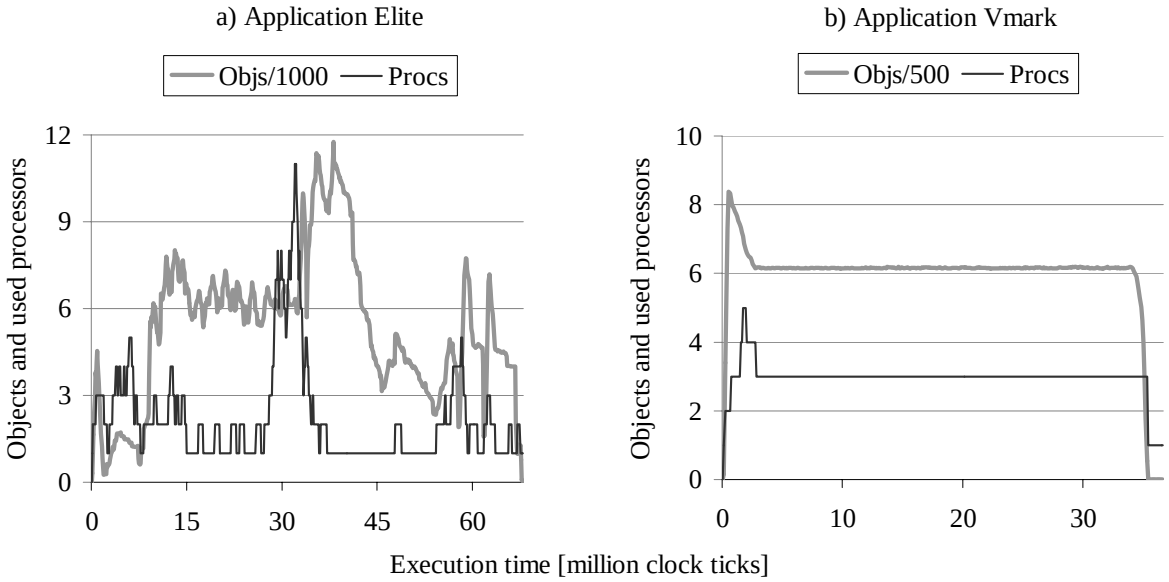

Figure 2. Number of objects as an indication of application parallelism for two applications. We depicted case E from the figure above.

and large-grain objects. Distribution is defined at the language level, and allows the programmer to control the location of objects through language operators [9]. However, Emerald does not include automatic migration mechanisms; the runtime-system distributes objects only as directed by applications.

Globe [13] is specifically aimed at wide-area distributed applications. Objects in Globe are similar to the fragmented objects found in SOS and location transparency is provided by the system. It is too early to tell whether and how Globe will feature automatic distribution mechanisms.

The Coign system automatically partitions existing applications built using Microsoft's COM standard [7]. The Coign approach differs substantially from ours in that it

assumes static environments and an off-line training stage prior to the actual running of the application.

6. Discussion and Conclusions

This paper has presented a simulation-based study of several approaches to mapping communicative objects to nodes in a wide-area system. The central theme of our study is investigating the efficacy of simple, local policies whose overhead does not scale with the number of objects in the system.

Our study revealed a number of interesting trends. For example, the most sophisticated load estimation policies did not perform well because they took too long to accumulate state. Instead, the best policies were simple counts of resident objects. If there is a lesson to be learned, it is that complicated measures have more pathological cases than simple measures, and a few pathological cases can dominate overall performance.

Second, appending incremental load information to outgoing messages performed at least as well as an oracular approach that propagated all current load estimates instantly, and with zero cost. We found that perfect up-to-date information may hurt performance because of "desired side effects" that manifest due to dissemination inertia. Furthermore, we learned that for systems supporting fine-grained patterns with complex communication behavior, piggy-backed information quickly disseminates information across the system.

Third, communication-based migration decisions significantly improve if we employ more complex discriminators based on object-processor affinity as opposed to object-object affinity. This places a premium on methods that strive to accurately approximate this information in large dynamic systems.

Fourth, object behavior profiling based on blocking/ready time is a successful candidate in estimating processor underloading and overloading conditions. An eviction mechanism can automatically grab and release processors based on application needs, thus efficiently using only as many resources as required. Combining eviction with migration and object clustering can achieve further performance improvements by reducing interprocessor traffic and synchronization.

Finally, we showed that a combination of simple, cheap approaches to each of these areas can result in performance nearly as good as the best approaches, including those based on oracular information.

There is more than one direction in which our work could be extended. First, the object model can be enriched by adding support for passive objects, read-only objects, shared objects, stationary objects and hard-links, which are a means of specifying that a group of objects should always be co-located. The key question is how can we exploit the characteristics of each new object class in order to develop better distribution policies.

Second, investigating more heuristics based on cluster analysis [8] might reveal surprising results both in the context of eviction as well as object cluster migration at invocation, a feature we have not implemented yet.

Finally, implementation of these techniques in a real system would provide us with more accurate feedback as to their performance.

References

[1] A. Bakker, I. Kuz, and M. van Steen. Towards a taxonomy of distributed-object models. In *Proceedings of the Third Annual ASCI Conference*, pages 22–27, Heijen, The Netherlands, Jun 1997.

[2] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, Jan 1987.

[3] R. Diekmann, B. Monien, and R. Preis. Load balancing strategies for distributed memory machines. In F. Karsch, B. Monien, and H. Satz, editors, *Multi-Scale Phenomena and Their Simulation*, pages 255–266. World Scientific, 1997.

[4] H. El-Rewini, T. Lewis, and H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall, Englewood Cliffs, NJ, 1994.

[5] D. Ferrari and S. Zhou. An empirical investigation of load indices for load balancing applications. In *Proceedings of the 12th International Symposium on Computer Performance Modeling, Measurement, and Evaluation*, pages 515–528, 1987.

[6] A. Grimshaw and W. Wulf. Legion: A view from 50,000 feet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, CA, Aug 1996. IEEE Computer Society Press.

[7] G. Hunt. *Automatic Distributed Partitioning of Component-Based Applications*. PhD thesis, Univ. of Rochester, 1998.

[8] R. A. Jarvis and E. A. Patrick. Clustering using a similarity based on shared near neighbors. *IEEE Transactions on Computers*, Nov 1973.

[9] 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, Feb 1988.

[10] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, second edition, 1999. I.S.B.N. 0–201–31009–0.

[11] M. Shapiro. Prototyping a distributed object-oriented OS on Unix. In E. Spafford, editor, *Proceedings of the Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 311–331, Fort Lauderdale, FL, Oct 1989.

[12] D. Stefanovic, K. McKinley, and J. Moss. Age-based garbage collection. In *Proceedings of OOPSLA*, pages 370–381, Denver, CO, Oct 1999.

[13] M. van Steen, P. Homburg, and A. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, pages 70–78, Jan–Mar 1999.