Studying Object Behavior Patterns for Implementing Efficient Object Distribution Mechanisms

Bujor D. Silaghi*
Department of Computer Science
University of Maryland, College Park

April 1, 1999

Abstract

Distributing objects 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. We are interested in developing techniques for achieving a reasonably well distribution with minimal system overhead being incurred. We implemented an object simulator along with a set of methods and policies for solving this problem and performed an initial evaluation of these on a number of well-known object behavior patterns.

1 Introduction

This paper describes a model used to evaluate the behavior of objects in a distributed environment and an implementation of this model as a simulator. We are primarily concerned with the understanding of different object behavior patterns and the way an object-based distributed system could benefit from *being aware* of these patterns and improve the performance of applications. The final goal of our study is to design and implement algorithms and policies that could exploit this kind of information.

We implemented an object simulator, called OPSIM hereafter, which can run scripts written to model the behavior of distributed applications based on objects.

The remainder of this paper describes these efforts and is structured as follows. Section 2 describes research directions and work related to our study. Section 3 introduces the assumptions and the environment in which the simulations took place. In Section 4 we present the patterns run on the simulator

*bujor@cs.umd.edu

and in Section 5 we explore in more depth how object distribution is achieved in our system. Finally, in Section 6 we describe the evaluation tests carried out and interpret our first results.

2 Related Work

[ADD]

3 The Simulation Framework

The main actors of a simulation are the objects and the processors running the objects. This section briefly outlines the object abstraction implemented and the way the simulation is carried out by the system.

3.1 The Object Model

Three basic assumptions are made about the object model. We assume a global object referencing scheme: each object has a unique identifier throughout the whole system which can be used to reference the object transparently, no matter the location of the object. We also assume that every object can be executed on any processor. Finally, we require that each object has associated a logical thread, that is we currently only support *active objects*.

Objects are grouped in classes which describe the behavior of their instances. Basically, each object will perform 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.

In order to exchange information, each object has a set of references to other objects, any of which can be the destination of messages. These references would correspond to references kept in variables on the stack and in the fields of the source object for an object-oriented system. The set of references may shrink when the object loses some of them, or grow when new objects are created or references are exchanged through messages. Besides references, a message may contain *raw* data which corresponds to instances of primitive types in actual object systems; this type of data in not interpreted by the simulator.

Every simulation commences with the creation of a *main object* which is subsequently run. This object may in turn create other objects or delete references, and so on until the application finishes. The simulation ends when the main object reaches a certain state. If none of the objects is able to proceed, we have a deadlock and the simulation is aborted. The simulation can also be terminated by enforcing a certain execution time.

In any state of the simulation, the set of reachable objects is given by the set reachable from the main object or the network (i.e., the only *root* objects are the main object and the network). We needed to include the network in the root set because it may happen that references to objects not reachable from the

main object are packed in network messages which could expose these objects to the main object later.[1]

Since the reference graph is dynamic certain objects may become unreachable during the run. In order to accurately simulate real systems, these objects have to be identified and destroyed, otherwise we would waste processor time by running them. The garbage collection is performed using reference counting and periodically a mark-and-sweep phase is applied to the reference graph to collect unreachable cycles.

Following is the set of instructions interpreted by the simulator together with a short description for each of them. All of these instructions apply to a *current object*.

COMP specifies a computational phase. It has as parameter the length of computation expressed in number of instructions.

NEW results in the creation of a new object. After creation, the new object will be referenced solely by its creator. The only parameter specifies the class to be instantiated. Where the newly created object is placed is dealt with in Section 5.6.

DEL instructs the simulator to delete a reference from the current object's set of references. The current object will not be able to refer again to the deleted object unless it gets the reference again by a message receive.

SND sends a message to the object specified as one of the parameters. An object cannot send a message to itself. Other parameters include a set of references to be handed over to the destination and some amount of raw data. The send is unblocking and therefore there is no reply value associated with it.

RCV receives the first message enqueued. If no messages are enqueued, the object is blocked until the first arrives. We implemented solely the asynchronous communication model without any loss of generality. If synchronous communication is desired, it can be simulated by a pair of succesive SND-RCV on the sender side, and a pair of succesive RCV-SND on the receiver side. The references packed in the message are usually automatically added to the set of references held by the receiver. There is a special case when the sender can actually instruct the receiver to delete some of its references, achieving the same effect as executing a DEL by the receiver.

SLEEP puts the current object to sleep for the specified number of clock ticks.

GOTO implements loops and conditional statements.

[1] There is one situation in which we are currently not able to collect unreachable objects from the main object. This is given by a set of objects with circular references which keep sending messages to each other that contain references to object in the set.

DONE finishes the execution of the current object. The object is not destroyed as there may be other objects holding references to it. If the current object is the main object, the simulation is considered terminated.

DBG is used to implement auxiliary services needed for debugging purposes.

The flow of the program is not entirely determined by the type of the instruction, i.e., GOTO or non-GOTO. For instance, it is possible to specify that a certain instruction be executed only when some other instruction gets executed, and also to specify a probability with which an instruction gets executed once reached.

3.2 The Processor Network

The objects are run over a network of processors. Currently this has to be a fixed set but we plan to account for a variable number of processors, since that would better model a real system in which hosts can go down or up on the fly. We also make another unrealistic assumption: processors have unlimited amount of memory, though each object has a size given by the class it belongs to. Again, this will be dealt with in future research.

Each processor has a running speed, expressed in clock ticks as a constant between 1 and 10. The only instruction that eats processor time is COMP and the speed of executing COMPs is proportional to the speed of the host of the object. This is a safe simplification since the time eaten by other instructions can be accounted for by inserting additional COMP instructions. Currently the speed of processors is kept constant but we could remove this assumption if, for example, we wanted to simulate virtual machines running as mere applications on physical processors. In this case the speed of the virtual machine would vary in proportion to its allocated quanta on the physical processor.

The simulation is clock based. For every processor in the network we simulate a clock tick then we move on to the next clock tick and so on until the application finishes. Scheduling of objects on processors is done using round-robin, with a quanta of approximately 100 clock ticks. When an object gets blocked (blocking RCV or SLEEP) the unused quanta is halved. This way, at any time an object will have less than twice the running quanta of scheduling left. Since the scheduling quanta is approximately the same on all processors, the amount of instructions executed while scheduled will increase when moving the object on a faster processor, and decrease when moving it on a slower processor.

OPSIM applications are scripts which describe patterns of object behavior. These scripts are generic and include probabilistic parameters. There is no language in which the scripts can be written; they are encoded as data structures within the simulator. To get an instance of an application, a script must be run on the simulator and instruction traces will be generated for each object. The traces must then be run either on one processor or on a set of processors. We are interested in using the processing time as efficiently as possible i.e., obtaining

a speedup close to the optimal speedup[2], and also in minimizing the network costs incurred by inter-processor message exchange.

As we already said, the simulation finishes when the main object executes a DONE instruction. However, after generating the traces on one processor and running them on a set of processors, that may not be the case. For instance, due to execution speedup, all objects may have finished their traces while the main object still didn't get the chance to execute the DONE instruction. In this case, and *only if* the main object does not create any more objects, we consider the application terminated.

4 Applications

We have experimented with seven synthetic applications which will be described subsequently. Each of them tries to capture a certain pattern of behavior which we think is representative for object-based systems.

The direction of arrows indicates how invocation messages flow throughout the system and their thick suggests the degree of interaction between objects.

4.1 The Most-Preferred-Object Pattern

The master object creates and deletes objects in increments of two in each round. After it creates two objects, it pairs them together and then the objects in the pair communicate asynchronously with each other until they are destroyed or the application finishes. Two objects in a pair act as a server-client with the client sending a request, doing some computations, and then waiting for a reply. The server waits for a request, procesess the request, and then replies. The amount of computation performed by the client and the server in a round is usually imbalanced with the server taking more time to compute than the client.

This application is dynamic, pairs of objects being created and deleted on the fly. It ends when after a specified number of rounds.

A simplified diagram of the most-preferred-object pattern in presented in Figure 1.

The reason for this pattern is to trial scenarios in which objects have a most-preferred communication partner or communicate exclusively with another object. The ideal distribution in this case would arrange the paired objects on the same processor.

4.2 The Pipeline Pattern

In this pattern the master object creates a pipeline of objects and then it plays both the role of the input to and of the output from the pipeline. After the

[2]Assuming the set of processors $[s_1, s_2, \ldots, s_n]$, given by their speeds, the speedup is equal to $S = \frac{1}{s_t} \sum_{k=1}^{n} s_k$ for some t between 1 and n, since with distribution disabled the simulator will run the application on a specified processor. Unless otherwise specified t will always be 1.

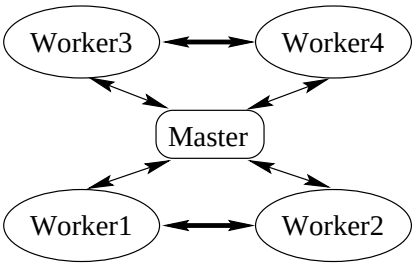

Figure 1: The most-preferred-object pattern.

pipeline is set up, no more objects are created and the simulation ends after a number of rounds. The pipeline may not be full at all during the simulation; for instance, the master object may insert a number of messages equal to 2/3 of the length of the pipeline at one end, and then retrieve the same number of messages from the other end (per round).

The pipeline pattern is depicted in Figure 2.

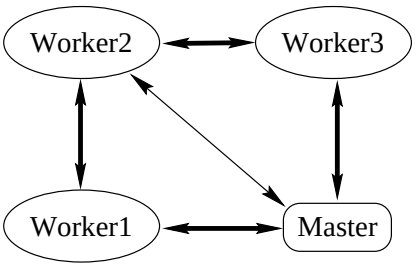

Figure 2: The pipeline pattern.

Unlike the previous pattern, workers simulate a synchronous communication because we want to ensure that messages from the same source arrive in order[3]. This explain why the thicker arrows are bi-directional when normally they should be uni-directional.

The pipeline pattern is implemented in many parallel applications and we think of it as being quite representative for parallel processing. The ideal distribution in this case would arrange contiguous segments of the pipeline on the same processor.

[3]Out-of-order message delivery from the same source could occur if when sending the first message the source and the destination are located on different processor, and then the destination migrates on the source's processor and gets the second message sooner than the first.

4.3 The Salesman Pattern

The master object creates and deletes objects all along the simulation. Asynchronous communication takes place between the master object and the worker objects and it is bursty. That is, the master object exchanges tens of messages with one object than it does the same with another one, and so on until the simulation finishes. Objects not currently communicating with the main object execute computational instructions.

The salesman pattern is presented in Figure 3.

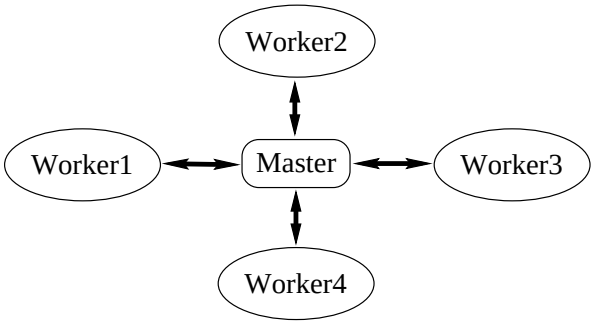

Figure 3: The salesman pattern.

We consider this pattern important because it exposes one of the current trends in distributed computing: agents deployed by the client to collect information from remote sources. The ideal distribution in this case would arrange the master object and the current partner object of type worker on the same processor.

It's highly unlikely that all the worker objects will reside on the same processor and therefore the master object should migrate to the host of the next partner if different from the host of the current one. It should be avoided at all costs the opposite effect —workers migrating in turn to the master's host— since that could lead to a situation when the master's host becomes overloaded and remote invocations are forced with the other workers.

4.4 The Divide-and-Conquer Pattern

The master object creates and deletes worker objects and communicates with each of them to solve a problem. In turn, each child may create (or delete) a number of objects trying to solve its subproblem, and so forth. Therefore the reference graph, ignoring the references to parents, has a tree-like structure with a variable branch factor. The computational phases descend down the tree until they reach the leaves and then ascend up the tree, everything being controlled by exchanging messages. On every path from the master object to a leaf there is a single object performing computations at any time. The communication is asynchronous.

The divide-and-conquer pattern is depicted in Figure 4.

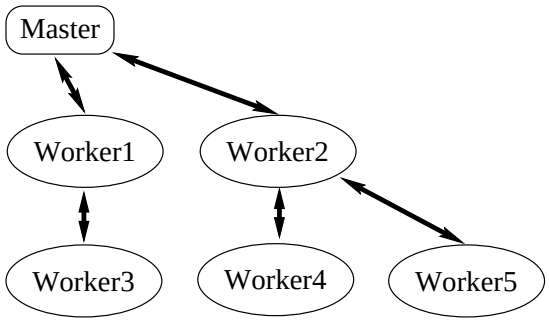

Figure 4: The divide-and-conquer pattern.

This pattern occurs in many problems for which the work can be decomposed and then the solution can be composed from the component solutions. An ideal distribution in this case it's harder to grasp intuitively; generally speaking, it should maximize the number of objects in parent-child relationships on each processor.

4.5 The Grid Pattern

The master objects creates a matrix of worker objects after which no objects may be created or deleted. Each worker exchanges asynchronously messages with its four neighbors —send to all of them and the receive from all of them— and performs a computational phase in each round. After each round all the workers apply with the master for passing a barrier; when all workers reach the barrier the master informs all workers of the next round. The applications ends after a number of rounds.

The grid pattern is presented in Figure 5.

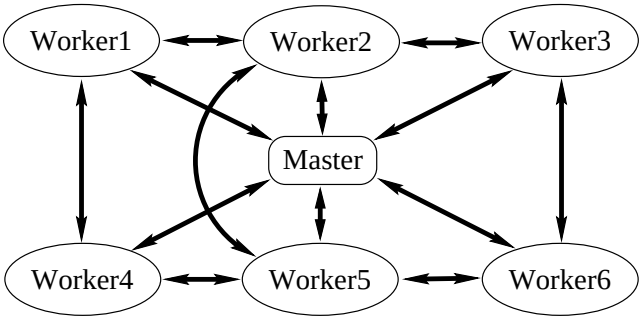

Figure 5: The grid pattern.

This pattern occurs frequently in scientific problems (especially simulations of natural processes) where a continuous space is discretized based on a grid and an iterative algorithm is applied that includes both communication with neighboring grid elements and a global synchronization, usually with a barrier. An ideal distribution can be pictured by drawing tangent circles over the grid such that the number of grid element in each circle is proportional to the processor speed associated with that circle.

4.6 The Client-Server Pattern

The master object initially creates a number of servers objects and about an order of magnitude more client objects. It then introduces a limited number of servers to each client. The clients randomly select one of the servers they know about, issue requests and then wait for replies. The servers wait for requests, process these requests and send replies back to clients.

This simulation is dynamic in nature with the master object creating client objects all along the simulation. Furthermore, although client objects are not deleted by the master they may reach the DONE state before the simulation ends, providing therefore for the situation when useless objects have to be managed by the system throughout the run. In order to avoid message ?cluttering? at the servers, they may skip the processing phase for certain requests which would correspond to refusing to process those requests.

The client-server pattern is depicted in Figure 6.

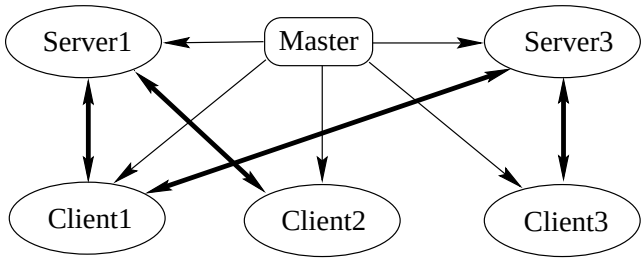

Figure 6: The client-server pattern.

This is also a classic pattern and can be found in database applications. The ideal distribution in this case would arrange !!!!!!!!!!!!!!

4.7 The Random Pattern

The random pattern helped us test the simulator in single-processor mode, more exactly the probabilistic behavior of it. Because no synchronation exists between objects, there is no way to impose ordering constraints between events.

Impossible situations[4] can arise when running the traces in a multi-processor configuration and hence we will not discuss about this pattern furthermore.

4.8 Combining Patterns

This pattern is merely a mix of the above patterns, except the random pattern. The master object creates six objects, each of them being responsible for starting up one of the patterns. Then it goes to sleep, and the application is ended when it wakes up. We will concentrate on this pattern for most of our simulations.

5 Algorithms and Policies

We present the algorithms and policies currently implemented in OPSIM. Section 5.1 describes the network model. Section 5.2 and Section 5.3 deal with ways of estimating the processor load and disseminating this information to the interested processors. In Section 5.4 we give a general view of the migration policies and then we expand on object affinity in Section 5.5. We end with a short discussion on remote object creation in Section 5.6.

5.1 Modeling the Network

The network abstraction used by the simulator is quite simple. We assume uniform network costs, that is, every pair of processors can communicate with the same parameters. Furthermore, we ignore the network traffic and real-life events such as network congestion. The cost to send a message of size msg_size is given by

$$net\_delay = net\_overhead + msg\_size/net\_bandwidth$$

In contrast, the cost of sending a message locally is always 1, irrespective of the size of the message.

Note that the above cost reflects the delay as experienced by the receiver of the message. In real life, sending a remote message may cause the sender to waste some time actually delivering the message to the network (system calls overhead, etc.). Therefore, by reducing the number of remote messages we get a triple effect: less synchronization delays, less overhead, both of which contribute to the execution time, and better utilization of the network. Our simulator is not able to exploit the second one.

5.2 Estimating Processor Load

For mostly-computational applications, reasonable speedup can be achieved by guaranteeing load-balancing. Therefore we need a way to estimate the dynamic load of each processor in the network.

[4] One example is trying to delete a reference that is not hold by the object, possibly one that will be sent to the object in the near future.

Furthermore, application parallelism limits the speedup that can be achieved and therefore the number of processor that can be used efficiently. Consequently, we need a way to identify processors which are underloaded (i.e., wasting processing cycles) and use them as primary targets for newly created objects or migrations and use completely free processors only when the application can expand efficiently — when all processors currently used are overloaded and new objects are being created, for instance. We achieve this effect by a combination of policies which include: processor load estimation, object placement at creation and migration.

We implemented four different ways for estimating processor load. The accuracy of these estimations and the efficiency of disseminating them will ultimately determine what each processor knows about the load of the others. Following is a description of the four methods. The first three of them are oblivious to the degree of overloading/underloading of processors, while the last one takes this issue into account.

1. The number of resident objects divided by the processor's speed. This is supposed to have little overhead and it turns out to be a good approximation of the actual load.

2. The number of *live* (i.e., not DONE) resident objects divided by the processor's speed. This is a refining of the above method and has about the same overhead.

3. The number of ready objects divided by the processor's speed. The overhead associated with this method matches the previous ones.

4. Actual load as given by the cumulative execution cost of resident objects divided by the processor's speed. This method, not only has bigger overhead but also it assumes some way of computing this execution cost. For off-line simulations this works fine since we have the *future history* in the form of traces. For real systems we need to come up with this information online, or use some previously gathered profile data. For instance, profile data regarding the average lifetime of objects based on their class could be used to infer this kind of information.

5. This method is more complex and involves determining whether the processor is underloaded or overloaded. To do this, each processor records the fraction of lost clock ticks, lct, during some past period, rt. It then derives $load$, proportional to the number of extra objects it could run to get out of the underloaded state but without entering the overloaded state by the following formula:

$$load = -no * sp * sf * \frac{lct}{(rt - lct)}$$

where no is the number of objects residing on the processor, sp is the processor's speed, and sf is a scaling factor used for better accuracy. If

lct amounts to zero than we assume that the processor is overloaded and we estimate it quantitatively by using method one of the methods (1)-(3) above.

This method has reasonable overhead and it resumes to performing some constant and very small amount of bookeeping by each processor from time to time.

5.3 Disseminating Processor Load Information

The load information needs not only be estimated but it also needs to be disseminated to all the interested processors. We took three approaches as follows.

1. The oracle approach is an unrealistic approach and assumes that every processor has instant access to the current load estimation of any other processor. Nonetheless, this method is useful because it gives an optimal policy and therefore a yardstick for more realistic approaches.

2. Piggybacking processor load information on messages is another method we implemented in OPSIM. Every time a remote method invocation is performed or an object migration occurs, the source processor will pack some of its knowledge of other processors' loads in the message. The source will also pack in its own load, as given by a direct estimation. The target processor will confront this information with its own view and keep the one most up to date.

   Staleness of load information is expressed as version control numbers. Each time a processor releases its load to other processors it generates a new version number in increasing sequence. Whenever some other processor knows of two different variants of this load, it will keep as the one most up to date that which has a larger version number, and discard the other.

3. This method extends the previous one with *ping messages*, which is the only type of message that does not involve objects. Each time a processor updates the load information of some other processor, it associates a time stamp with it. When the time stamp expires it is considered that the respective load information is meaningless and a special request is sent to the processor to query it for the load.

4. We are trying to employ ping messages as rarely as possible. There are two deviations from the above method. First, each time a ping message is sent, the source will piggyback some of its processor load knowledge and also will ask the target to send back, beside its own load, some extra load information. More exactly, it will ask the target for load information about processors that have the highest degree of staleness at the source. Second, only one ping message is sent —the target is chosen to be the processor with the highest degree of staleness at the source— every time we check for information staleness. In contrast, the previous method forces ping

messages be sent to all the processors whose load information is considered to be out of date.

Piggybacking load information on messages involves a constant amount of effort since no more than a certain number of instances —currently 10— is packed in. This also prevents adding too much extra overhead in terms of data size but could turn out to be an unacceptable limitation if, for instance, we dealt with a very large pool of processors.

5.4 Migration and Eviction

We are trying to accomplish two things simultaneously. First, we are striving for load balancing by placing on each processor a load proportional to its speed. This will ensure efficient utilization of computational resources. Second, we would like to obtain load balancing with minimal network costs. One way to do this is to group related objects that tend to communicate a lot with each other on the same host.

The issue of combining these two goals is not as simple as it may appear at first sight. Consider for example the extreme scenarios. On the one hand, an application whose objects are computationally intensive and barely communicate with each other. In this case the role played by the network is minimal and almost any object placement arrangement will be satisfying, as long as load balancing is maintained. On the other hand, consider an application whose objects do nothing but communicate. In this case, presumably the best arrangement would be to place all the objects on the fastest processor, since the possible gain obtained by using multiple processors would be overcome by the communication costs.

There is another key factor in designing successful distribution policies, the associated overhead. Since we may be talking about dynamic environments with thousands of processors and millions of (potentially fine-grained) objects, it becomes clear that the algorithms cannot take into account the whole world. Ideally, they should be independent of the number of objects or processors. Since the number of references an objects holds depends primarily on the type of the object (or application, for that matter) it shouldn't vary much with the number of objects or processors in the system. Therefore, policies that take into account only the set of referenced objects in deciding the object location, should have acceptable overhead.

We want to capture the notion of affinity between objects as given by communication partnerships. The only events that count in this evaluation are the message sends [5], and therefore we monitor these events for each object in part. Every time a message send occurs we increment a counter associated with the reference. There is basically no extra overhead associated with this operation.

We continue by presenting the basic decisions when migrating an object.

[5] Another approach would be to consider RCV events but it is more convenient to do it in terms of sends. This is because we are already holding references to all the possible destinations from the current object.

When Since only communication events count, we try to perform migrations when sending messages. Each time a message send occurs we check whether the source object is a candidate for migration. We initially experimented with periodic migrations policies using fixed time intervals, but they turned out to perform worse than those based on send events. How exactly is determined whether an invocation will result in an object migration is dealt with in more detail in Section 5.5.

Which This question is automatically answered by choosing send-based migration policies: the only objects that are to be migrated, if acceptable, are objects that have just executed SND intructions. The migration messages are packed into the corresponding invocation messages that triggered the migration.

Where Similarly, the only possible destination of a migration is the destination processor of the invocation that triggered the migration. This makes sense since the only potential hosts for each object should be hosts to which it sends messages. The evaluation is made at each invocation so it suffices to check only the current destination as a potential future host.

New invocations may describe more accurately the communication preferences of objects than old ones. We implemented a decaying policy for invocation counters which divides them by a constant every time the object holding the references is being migrated. We miss the opportunity to decay the counters if no migration occurs, but the fact that the object is not migrated may signify that it doesn't have a clear pattern of behavior and hence it's all the same. Performing decaying at migration is convenient because the state of the object needs to be marshaled and sent over the network, and the overhead to search the references (on the stack, for instance) can be combined with these operations.

Since the application is launched on a certain processor, we need a way to populate the other processors. This is not handled by the above policy because processors with no objects cannot be destinations for migration. Therefore, we need some kind of eviction policy which would throw out objects that are only slightly related to other objects on the same processor. In effect, this will also decrease the load on the processor, and will make room for related objects to be migrated in. Figuring out the bad objects, using the invocation reference scheme, is however much harder because we would need some kind of minimization whereas the sequence of invocation counts is monotonically increasing. We implemented such a policy but it turned out to perform bad in conjunction with migration: objects were experiencing the ping-pong effect and ended up being trashed between processors. We will not expand on this subject anymore.

5.5 Object Affinity

The most important question when migrating an object is whether the destination host is a *better* host than the current one. This question should be answered

from a global perspective, that is, considering the effect of the migration on the application execution time and the network utilization. As we mentioned before, this is practically impossible when considering large-scale systems. Therefore we will constrain our decision policies to operate on a local domain; more exactly, we will only consider objects referenced from the current object (the migration candidate) and their respective host processors.

The first set of conditions deals with the affinity between objects: it rates a processor from an object's perspective based on the objects hosted by that processor which are referenced by the object. Only one such condition is used at a time to evaluate this affinity. They are presented in increasing order of complexity.

Let r_i denote the number of invocations performed on reference i from the current object and n the number of references. The host processor of the object referred by i is denoted by p_i. Furthermore, let i_0 be the reference corresponding to the target object.

The first condition is a greedy and unsafe decision: it migrates the source if the most invoked reference happens to be the current target object and it overcomes the second most invoked object by at least δ. Also, it can be implemented efficiently: for each object we have to keep the current maximum and compare it to the current target's invocation count on every remote invocation.

$$\gamma(r_{i_0}) \geq \delta + \gamma(r_i) \qquad \text{for all } i \neq i_0$$

The second condition is more on the safe side: it migrates the source only if the current target object has been invoked more than the cumulative invocations of all the other references, offsetted by δ. If the same pattern of invocation is kept, this guarantees that the target processor is the best host for the source object. This one too has a simple implementation: for each object we have to keep the cumulative invocation count and compare it to the target's invocation count on every remote invocation.

$$\gamma(r_{i_0}) \geq \delta + \sum_{\substack{i=1 \\ i \neq i_0}}^{n} \gamma(r_i)$$

The third condition relaxes the previous condition but still requires that the target processor be the best host for the source object: it migrates the object if the cumulative invocations to objects residing on the target processor is greater than the cumulative invocations to objects residing on any other processor, offsetted by δ.

$$\sum_{p_i = p_{i_0}} \gamma(r_i) \geq \delta + \sum_{p_j = p} \gamma(r_j) \qquad \text{for all } p \in P$$

Finally, the last condition tries at the same time to remain on the safe side and to relax the previous condition as much as possible. This time the only

requirement is that the target processor be a better host than the current one: it migrates the source object if the cumulative invocations to objects residing on the target processor is greater than the cumulative invocations to objects residing on the source processor, offsetted by δ.

$$\sum_{p_i=p_{i_0}} \gamma(r_i) \geq \delta + \sum_{p_j=p_s} \gamma(r_j)$$

The last two conditions are more complicated to evaluate for two reasons. First, they rely on the fact that the host processor of every object referenced from the source object is known. There is a discussion on this issue in Section 7. Second, caching temporary results for each object is not a very convenient way in this case because the pool of processors could be fairly large. The other option left is evaluating the condition from scratch every time a remote method invocation occurs. This implies scanning the set of references but it's relatively hard to estimate the overhead incurred from an abstract point of view.

In all cases the additive constant, δ, is a correction factor used to rule out pathological behavior. One such example is given by the first invocation of an object that will trigger a migration of the source object with any of the above methods. Function γ is usually just the identity function but we also experimented with quadratic and exponential functions.

The second set of conditions deal with checking whether the migration can occur given the loads of the source and target processors. We distinguish four different cases. For a migration to be performed the condition corresponding to the case at hand, and one of the above have to be simultaneously satisfied — we assume that this is the case. Let l_p and l_q denote processor load estimations of the source p and target q, as given in Section 5.2.

$l_p \geq 0 \wedge l_q < 0$. Migration will always occur.

$l_p < 0 \wedge l_q \geq 0$. Migration cannot occur.

$l_p \geq 0 \wedge l_q \geq 0$. We need another condition of the form $c_p * l_p \geq c_q * l_q$ be satisfied in order to perform migration, where c_q/c_p is the minimum acceptable load ratio of the source and target processors.

$l_p < 0 \wedge l_q < 0$. Migration will occur only if the application is not in an expansion phase. That is, the following condition must be satisfied too

$$l_r < 0 \qquad \text{for all } r \in P$$

This allows for the application to abandon the most underloaded processors when the application shrinks, i.e., a large number of objects die off without a commensurate number of objects being created. If the application is expanding than presumably processors p and q will get some objects because of the first condition.

5.6 Remote Object Creation

When using migration and eviction objects newly created were placed on the same processor as the creator. This seems reasonable because the parent and child objects are expected to communicate and therefore co-locating them is expected to reduce communication costs.

Since we abandoned eviction we had implement a replacement mechanism for populating free and underloaded processors. Currently we have a single policy for choosing the initial host processor for newly created objets: the least loaded processor. The effect of this decision depends also on the method used for evaluating processor load. For instance, if choosing the last method presented in Section 5.2, than priority is given to underloaded processors over free processors and therefore the application tends to grab free processors only as needed.

Load balance can thus be achieved by choosing the least loaded processor. At the same time, there is hope to obtain good object locality since communicating objects are often created at the same time and it's likely that they will be placed by this policy on the same (least loaded) processor.

6 Experimental Evaluation

We performed a number of experiments to assess the individual contributions of the various policies and algorithms. Since it is virtually impossible to test all the possible parameter combinations[6] we only present results for combinations that would seem likeable in a real system. The results presented apply to the combined pattern as described in Section 4.8; we experimented with the individual patterns too but we consider this pattern more relevant due to its complexity and the fact that it embeds all the others.

Two of the processor configurations used are as follows:

1. 12 processors: [5, 3, 7, 1, 1, 10, 5, 5, 4, 3, 8, 3]. The optimal speedup associated with this configuration is 11. An average number of 1641 objects were created in this setting and 661458 invocation messages were exchanged. The parameter settings guarantee that all the processors can be efficiently used.

2. 37 processors: [5, 2, 1, 2, 4, 7, 10, 8, 6, 3, 3, 3, 2, 9, 10, 9, 5, 5, 5, 7, 6, 2, 6, 4, 10, 8, 8, 5, 5, 9, 1, 4, 6, 3, 7, 2, 8]. The optimal speedup in this case is 40. 3106 objects were created and 1028675 messages were exchanged on the average. The parameter settings were such that only a subset of the whole set of processors can be efficiently used. Figure 6 pictures the number of objects as a function of the execution time in a uni-processor setting.

The first set of experiments evaluates the load estimation methods. One way to do it is measuring how well object placement at creation is achieved using the

[6] A simulation is controlled by tens of parameters, not taking into account those that describe the application templates.

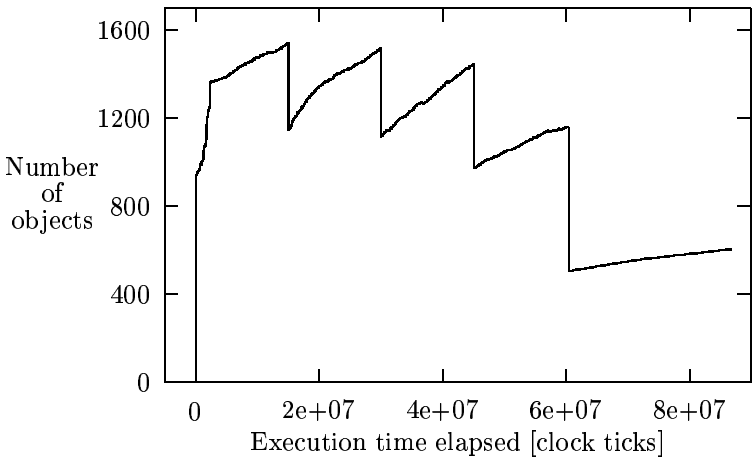

Figure 7: Number of objects during execution.

various estimations in terms of the execution time. We used the first processor configuration.

Using the first four methods for estimating the processor load we obtained average speedups of 9.17, 9.76, 6.64 and 10.5. The application scripts were intentionally designed such that the number of dying objects exceedes the number of created objects after the initial setup phase[7]. This will create a load imbalance that cannot be resolved by the first three methods because they have no means of estimating the expected lifetime of objects. However, the last method accounts for this and so we are close to obtaining optimal speedup. The results yielded by the fifth method match those given by method (2), since it is based on this one an no migration is allowed.

Method (2) gives slightly better results than (1). This is partially because done but not deleted objects are taken into account by method (1) when they shouldn't be. Also, the number of ready objects on a processor doesn't seem to an appropriare measure for estimating processor load since the speedup obtained, 6.64, is almost as small as half the optimal speedup.

The second set of experiments evaluates the load dissemination policies. Again, we do it in terms of the execution times and we use method (2) above for estimating processor load. The four policies yield speedups of 9.76, 9.87, 10.1 (with 844 ping messages) and 9.97 (with 588 ping messages) respectively. All the policies perform at least as well as the oracle approach for two reasons. First, there is a delay in getting the load estimation which in this case acts to our advantage since objects created in sequence have a greater chance of being placed on the same processor. Second, the only load variation is given by

[7]We also tested with application scripts that conserve or increase the number of objects and obtained for the first two methods results similar to method (4).

creating new objects; once migration in enabled the dynamicity increases and it becomes harder to track the processor loads.

As we've seen, reasonable speedup can be achieved using a combination of techniques for estimating processor load, disseminating these estimations and remote object creation. In doing so, we ignore the efficient utilization of the network resources. For instance, the ratio of network to local invocation messages can be as high as 16.95/1, when ignoring migration, or even higher if we increase the number of processors, but keep the average number of objects hosted by a processor approximately the same.

The last set of experiments using processor configuration (1) evaluates the migration policies. We used method (2) for estimating processor load and method (1) to disseminate the load. Using a value of δ of 5 we obtained the following speedups, along with the ratio of network to local invocation messages and the number of migrations performed: 8.49 (2.32/1, 10075), 9.4 (2.63/1, 391), 8.43 (1.04/1, 563) and 5.59 (1/5.27, 873). The pattern is clear: the smaller the ratio of network to local invocation messages, the less speedup is obtained. One extreme is given by the last method which yields half the optimal speedup and an improvement of 89 in the ratio of network to local messages.

The balance between speedup and network utilization is also influenced by the minimum acceptable load ratio of the source and target processors (c_q/c_p in Section 5.5 and below) and also by the cutting factor of the invocation counts, cf below. We used migration policy (4) for evaluating this influence which is expressed quantitatively in the following table.

	$cf = 1$	$cf = 2$	$cf = 4$	$cf = 8$
$c_q/c_p = 3/2$	5.59, 1/5.27	4.97, 1/9.16	4.83, 1/8.53	4.95, 1/9.16
$c_q/c_p = 1/1$	7.20, 1/2.78	6.79, 1/2.87	6.77, 1/3.26	6.75, 1/3.17
$c_q/c_p = 2/3$	9.95, 10.9/1	9.95, 10.9/1	9.95, 10.9/1	9.95, 10.9/1

The cutting factor has little influence on the results and is mainly because the application scripts tested are not very sophisticated. We can exploit the cutting factor for objects that change the set of communication partners over time, and this is currently happening only with the master object of the salesman pattern and with objects of the tree pattern. We expect for cf to have a greater influence in real applications.

The last line of the table corresponds to allowing migration only if the load of the source processor is significantly higher than the load of the target processor, in which case very few migration occur. This is true even for ratios very close to 1, like 19/20 for instance. Therefore to exploit the benefits of migration we must allow the transfer of objects to processors with higher load and this in turn explains the speedup degradation.

We also tested migration using method (4) as the load dissemination policy. For instance, we obtained a speedup of 6.1 and an invocation message ratio of 1/2.28 for the configuration that corresponds to the second line, second column of the above table. A number of 10778 ping messages were needed which is 63 times smaller than the total number of invocation messages. The quality of the

results varies with the number of ping messages, with greater speedups being obtained when increasing the number of messages and vice-versa.

[SECOND SET]

7 Conclusions and Future Work

[CONCLUSIONS]

There is more than one direction in which our work could be extended.

First, the object model can be enriched by adding suport for passive objects, read-only objects, shared objects, stationary objects and hard-links. The key question is how can we exploit the specifics of each category of objects for better distribution policies.

Passive objects do not contribute to the load of the host processor. Read-only objects can be safely replicated where they are needed at an extra load for active objects or supplementary memory requirements for passive objects. Shared objects suport allows mutable objects be replicated as well, but additional system support is needed to maintain object state consistency. Stationary objects are hooked to a certain processor, and therefore migration is not an issue for this category. Hard links generate groups of objects that have to be on the same processor at all times.

Second, the current mechanisms could be optimized at least as far as concerning the efficient utilization of the network. For example, objects to be migrated could be delayed and then packed into the same message, if they have the same destination and more, if the destination processor is not underloaded. The same technique could be applied to remote invocation messages, but this time the requirement is that the source processor not be underloaded.

Third, the degree of detail with which the simulator models the real world is an important issue. The network model is quite simplistic and certain other components are modeled unrealistically. For instance, no mechanism for keeping track of object locations has been implemented; each processor knows at no effort the exact location of every object in the system.

Finally, the implementation of these techniques in a real system would provide us with a more accurate feedback as to the efficiency and the overhead incurred by applying them.