Parallel Programming in a Virtual Object Space

Steven E. Lucco

AT&T Bell Laboratories
Holmdel, NJ 07733

Abstract

Sloop is a parallel language and environment that employs an object-oriented model for explicit parallel programming of MIMD multiprocessors. The Sloop runtime system transforms a network of processors into a *virtual object space*. A virtual object space contains a collection of objects that cooperate to solve a problem. Sloop encapsulates virtual object space semantics within the object type *domain*. This system-defined type provides an associative, asynchronous method by which one object gains access to another. It also provides an operation for specifying groups of objects that should, for efficiency, reside on the same physical processor, and supports exploitation of the topology of the underlying parallel machine. Domains also support the creation of *indivisible* objects, which provide implicit concurrency control. The encapsulation of these semantics within an object gives the programmer the power to construct an arbitrary hierarchy of virtual object spaces, facilitating debugging and program modularity. Sloop implementations are running on a bus-based multiprocessor, a hypercube multiprocessor, and on a heterogeneous network of workstations. The runtime system uses object relocation heuristics and coroutine scheduling to attain high performance.

1. Introduction

Sloop is a parallel language and environment that employs an object-oriented model for explicit parallel programming of MIMD multiprocessors. The object-oriented computational model provides a natural setting for the construction of parallel programs. By conceiving a problem solution in terms of a collection of cooperating objects, a programmer can partition a program into natural functional units, replicating these units according to the data parallelism inherent in the particular problem. Sloop exploits the power of this model by providing new semantics and facilities that address the special requirements of parallel systems.

The Sloop project has two major goals: to study methodologies for explicit parallel programming, and to apply the results of these studies to the design of a programming language and environment. Sloop evolved as a consequence of experiences in parallel programming (*e.g.* [LN87]) during a two year period.

The principal abstraction for the conceptual organization of Sloop programs is the *distributed object*. Distributed objects are a parallel extension of the *class* notion[GR83] with the following properties. First, a distributed object can be built from component objects located on different physical processors in a multiprocessor. Second, distributed objects are potentially accessible from any other object in the multiprocessor. Third, all of the component objects which constitute a distributed object can run operations simultaneously.

Sloop hides details of the underlying multiprocessor by providing an abstraction called a *virtual object space*. A virtual object space contains a collection of objects that cooperate to solve a problem. Sloop encapsulates virtual object space semantics within a predefined object type. This type, called *domain*, provides an operation for asynchronously creating and accessing objects. It also provides an operation for specifying groups of objects that should, for efficiency, reside on the same physical processor. However, the programmer can choose to construct distributed objects without explicitly mapping their components onto specific physical processors, in which case the Sloop runtime system performs mapping and load balancing functions. The encapsulation of these fundamental semantics within an object definition makes Sloop syntactically concise, since objects interact with domains exactly as they interact with objects of other types: through invoking operations.

Sloop has been running since January 1986 on two multiprocessors at AT&T Bell Laboratories: the S/Net, a bus-based architecture with up to 12 processors [Ah83], and an experimental 64 processor hypercube[De85]. In December 1986, a third version of the Sloop runtime system was completed for a heterogeneous collection of workstations connected by an ethernet.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commerical advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1987 ACM 0-89791-247-0/87/0010-0026 $1.50

The next section of this paper discusses an example that illustrates several benefits gained by organizing a parallel program in terms of distributed objects. Section three elaborates Sloop's computational model and describes how it facilitates the construction of distributed objects. Section four introduces mechanisms by which a programmer can speed the execution of a program by explicitly grouping heavily interacting objects, and by mapping objects to physical processors. Section five discusses the Sloop runtime system, focusing on automatic mechanisms used by the system to accelerate execution, especially for programs that do not make use of the explicit optimization techniques described in section four.

2. Distributed Objects

This section presents an example that shows how one can use distributed objects to organize a computation. Consider the following image processing application. We wish to determine a color for each of one million pixels (picture elements) in a bitmap (rectangular array) representing a graphics screen. The computation for each pixel can be carried out independently and takes an unpredictable amount of real time. To implement this application we must map the million pixels onto our n physical processors. Further, we need to devise a synchronization strategy for sending pixels to the graphics screen in the correct sequence. Note that this strategy interacts with the choice of mapping.

One naive solution is to divide the bitmap into n sections and assign each section to a processor. Whichever processor holds the first section of the bitmap will, upon completing its computation, output this section to the screen and send a message to the processor holding the next section, enabling that processor to output its information (when completed). However, this approach can waste large amounts of processor time. Since each pixel takes an unpredictable amount of time to compute, some processors will finish their portion of the task and sit idle while other processors compute more costly bitmap sections. As long as we rely on a static mapping of pixels to processors, the potential for this situation persists.

Thus, we need to find a method for dynamically varying the amount of computational power devoted to a subproblem. Refining our strategy, we may divide the bitmap into $1000n$ smaller sections. We then assign 1000 sections to each processor. If a processor finishes its computation, it searches for another processor that is still computing and takes half that processor's load. This simple optimization dramatically increases the communication and synchronization complexities of the implementation. Whereas the previous implementation used communication only to synchronize output, the new version requires additional communication to find heavily loaded processors and to move bitmap sections to idle processors. Further, since the new version dynamically remaps sections to processors, and since there are many more sections, correctly ordering these sections involves a complex protocol rather than a simple stream of messages.

In Sloop, we drastically simplify this scenario by building a distributed object. We organize our image processing implementation around a distributed ordered list of bitmap sections. Each processor repeatedly obtains a copy of the first uncomputed section from the list, computes it, and replaces it in the list. Whenever the next consecutive element in the list has been computed, the list object sends it to the graphics screen. If the list has no uncomputed section when a processor requests one, the list may instruct a currently executing processor to subdivide its bitmap section. This processor then inserts its new subsections into the list.

This Sloop formulation views correct ordering of bitmap sections not as a synchronization problem, but simply as a problem of maintaining the consistency of a list. The abstraction of the distributed list also eliminates the mapping problem: any number of processors participate in the computation. Further, processors can enter or leave the computation while it is in progress.

3. Programming in Sloop

3.1 A Virtual Object Space

Sloop provides a high-level computational model that makes the use of distributed objects straightforward and concise. From the programmer's perspective, the Sloop runtime system transforms a network of computers into a single virtual object space. In a running program, a collection of objects within the virtual space cooperate to solve a problem.

This model has several properties that facilitate distributed objects. First, objects compute and communicate in a uniform environment. The programmer need not pack and format data into a message in order to communicate information. Second, the programmer need not explicitly map objects to processors, although Sloop provides for explicit mapping when desired. Finally, one can build hierarchical structures without regard to crossing processor boundaries, because physically remote object references are supported transparently by the runtime system.

As in most object-oriented languages, a set of object types defines a Sloop program. Each type consists of structured private data, and operations for manipulating these data. Object types can include a constructor operation, which is invoked upon creation of each instance of that type. At any instant the *state* of the computation is the collection of objects in the

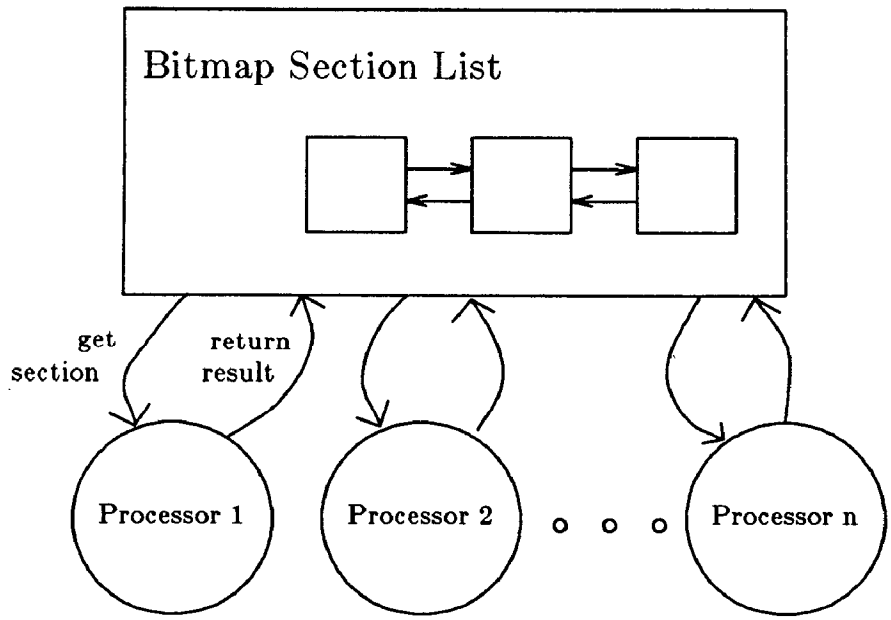

Figure 1--Distributed List Object for Image Processing Application

virtual space, their private data, and the set of operations invoked but not completed.

A Sloop program begins execution with a single object, called the *start object*. To run a Sloop program, the user designates a start object type, and provides arguments for the constructor of that type. The runtime system creates a start object of the designated type, and invokes the object's constructor with the given arguments. The start object in turn creates additional objects, each with the potential to access all other objects and to invoke operations on them. Thus, through a small collection of object definitions, the start object unfolds into a complex computation. This methodology produces programs that adapt easily to varying conditions. Because a computation begins with a single object, it can grow to fit different input sizes and amounts of available computational power.

Unlike processes in parallel languages such as [Ge85], Sloop objects become active only in response to an asynchronous operation invocation. In practice, this difference serves to reduce the amount of code devoted to synchronization in Sloop programs, and to increase modularity.

3.2 The Domain Object Type

Sloop represents a virtual object space through the object type *domain*. A domain is a system-defined object type that manages a collection of other objects. Three operations, *access, align,* and *copy*, define the functionality of domain objects. The *access* operation has two purposes: to create objects, and to provide references to existing objects. *Align* gives the programmer explicit control over the location of objects. *Copy* replicates the entire contents of one domain in another. These three operations provide the semantics of a virtual object space, encapsulated within the domain object.

Because a domain is an object, a program can create an arbitrary hierarchy of domains. Upon starting a new program, the Sloop runtime system creates a domain object called the *root domain*, within which the programmer can create objects, including other domains. The predefined identifier **root** is a reference to this initial domain object. Organizing a program in terms of several domains enhances the program's modularity and clarity. For example, a database may be implemented by a collection of read-only objects in a separate domain. The use of multiple domains also makes debugging easier, as one can encapsulate likely causes of error, such as semaphores and other objects used for synchronization.

One can use the domain copy operation to replicate a complex data structure. For example, a parallel compiler we developed in Sloop reads an input program and produces an abstract syntax tree. A second stage then uses the tree to produce machine code for the given program. We had the compiler construct the abstract syntax tree within one domain, and copy it to a new domain before modification. This gave us a representation of the program that could be checked for correctness, and compared to the final structure produced by the second stage.

3.3 Accessing and Creating Objects

Through the domain object type's *access* operation, Sloop provides an asynchronous, associative method for accessing and creating objects. An object p obtains a reference to object q by invoking *access* with the name of q, without interaction with other objects holding references to q. Specifically, The *access* operation takes as arguments an object's name, its type, and its mode, where the mode is either **read_only** or **read_write**. If an object with the given name exists within the invoked domain, *access* returns a reference to that object. Otherwise, *access* creates an object with the given name, type, and mode, and returns a reference to it.[1] Additional arguments may be passed through *access* to the new object's constructor. These semantics give the programmer the power to use distributed objects to completely decouple portions of a computation.

Suppose, for example, that we have a two stage computation. Corresponding to each stage is a compute object type that performs the stage's particular function. We connect the stages by a queue. The *access* operation allows the program to incrementally add compute objects to each stage. Each object begins its execution by invoking the *access* operation. The first such invocation creates the queue; later invocations return references to the queue. Stage one objects place their results on the queue, while stage two objects remove these results and complete the computation.

The *access* operation provides all the synchronization needed by multiple independent compute objects, distributed across a multiprocessor, to create and connect to a single queue object. Further, this queue object encapsulates all the synchronization needed to coordinate the execution of the compute objects at run time. In particular, the queue and compute objects contain no code devoted to management of semaphores, monitors, or other synchronization constructs. Instead, the queue uses the more concise mechanisms described in section 3.4.2.

One may decide to limit the access to an object x. To obtain this functionality, one can access x with a NULL name. In this case, the creator of x has the reference r returned by *access* at creation time. Other objects can only reference x if they have been given a copy of r. This usage is semantically similar to the *new* operator in Smalltalk[GR83], in that the returned value is the only reference to the new object.

1. If *access* matches a name, but the matching object does not agree in type or mode with the specified arguments, *access* generates a runtime error and returns <undefined>. Using an undefined reference also generates a runtime error.

3.4 Coordinating Object Interactions

Through the domain object type, Sloop facilitates asynchronous creation and manipulation of distributed objects. Combined with the object-oriented model's usefulness in representing a problem's solution in terms of natural building blocks, this facility gives a programmer the power to elegantly express parallelism. The following section describes Sloop's invocation and blocking mechanisms and illustrates how programs perform synchronization and preserve consistency within this framework.

3.4.1 Invocation

Object types have an attribute called their *exclusion level* which is either **re_entrant** or **indivisible**. The exclusion level specifies whether multiple operations may execute concurrently on an object.

The Sloop runtime system guarantees that only one operation at a time can execute on an **indivisible** object. This mutual exclusion property makes individual operations easy to define, as one can assume that the object is in a consistent state as long as each operation leaves it so. In addition, this property aids debugging in that only the sequence of operations, not interactions between concurrent operations, determines the behavior of an **indivisible** object. This property greatly simplifies the coordination of multiple objects.

Coordination of **re_entrant** objects can be much more complex, since one must consider interactions among operations. To ease this situation, Sloop restricts the concurrency of operations on re_entrant objects. Once invoked, a re_entrant operation will run until it explicitly surrenders control or invokes another operation.[2] If all operations on a re_entrant object leave the object's private data in a consistent state before blocking or invoking other operations, the private data cannot become corrupted. The comparison of this technique with the use of monitors[Br75] and other mutual exclusion mechanisms is a subject of our current research. In practice, most Sloop programs could not gain speed by using re_entrant objects, so the simpler indivisible objects are preferred. Analyzing 12 carefully written Sloop programs, we found that only three percent of the object types were specified re_entrant.

When one object invokes an operation on another object, it only waits for the operation to complete if the operation returns a value. Operations that do not return a value execute asynchronously with their

2. The Sloop debugger, discussed in section 3.5, can interrupt a re_entrant operation so that a user can check for pathological conditions.

invoking object. A sequence of operations invoked by object *b* on another object *a* will execute in order of invocation, possibly interleaved with operations invoked by other objects. With indivisible objects, all operations are performed first come, first served.

3.4.2 Blocking

Given that operations can execute completely asynchronously, the programmer needs a mechanism for coordination among objects. An operation that returns the first element from a queue, for example, must wait until the queue is non-empty. One can use the Sloop *await* operator to specify conditions that must be true before an invoked operation can proceed. Operations can contain statements of the form `await P`, where `P` is an expression involving the operation's arguments and the invoked object's private data. For example, a queue's *remove_first* operation can use a statement such as `await (head != NULL)`. Any *remove_first* operation invoked on a queue object will wait until this predicate `(head != NULL)` is true before executing. Operations invoked on the same object and held at the same await statement will complete in the order in which they executed await.

For indivisible objects, only the first statement in an operation definition can use the await operator. This restriction preserves the property that once an operation on an indivisible object begins execution, it will complete before any other operations execute on that object. **Re_entrant** objects can execute an await statement at any point. As noted above, these objects should ensure that operations leave the object's private data in a consistent state before surrendering control through an await statement. Because of the additional complexity, programs should use re_entrant objects only when necessary.

For efficiency, the Sloop compiler allows one to specify the operations that can cause an await predicate to change from false to true. For example, a queue's *remove_first* operation could use the await statement `await insert: (head != NULL)` to specify that the insert operation is the only operation that will make the queue non-empty.

3.5 Debugging

As programs become more difficult to conceptualize and coordinate, the strength of a language's support environment becomes more crucial to the success of the programming effort. This section gives a brief overview of Vision, a program animator and debugger for Sloop. Vision, which has been implemented on SUN workstations, adds several new features to sequential debugger technology, each of which targets a particular debugging activity. Vision provides both graphical and typed-command interfaces to these facilities. [Lu87] provides a more complete discussion of Vision, and its relationship to other parallel debugging systems.

A common first step in debugging object-oriented programs is to test the logic of each individual object type, running sequences of operations likely to occur in the actual program. Through Vision, the user can create objects, invoke operations on these objects, and examine their state. The user may also wish to investigate the interactions between objects in a running program. To facilitate this, Vision introduces a trapping mechanism called *object capture*. When Vision has captured an object, all operations invoked on the object pass through the debugger, enabling a user to examine and modify these operations. The programmer can specify program conditions, like breakpoint conditions in sequential systems, under which the debugger is to capture an object.

After an object is captured, the user controls the behavior of that object in the ongoing computation. As operations arrive the user can examine and modify arguments, pass the operation through to the object, just send a return value, or even destroy the operation altogether. When one suspects a particular object is malfunctioning, one can capture the object before starting the program, and supply its correct functionality interactively. One can also specify a "single-step" mode, in which the debugger captures objects as they are created.

Parallel programs are subject to a phenomenon called *probe effect* in which diagnostic tools, such as print statements or interaction with a debugger, modify the timing of a program in a way that affects its results. Further, programs sensitive to timing may contain intermittent faults that can not be reproduced deterministically. To help find timing bugs, Vision provides *deferred tracing*. The user can specify objects and operations for which execution traces are to be collected during program execution. After the program terminates or is stopped, the debugger can replay portions of the program's execution. The user interacts with the replayed computation as if it were actually running. Thus, one can run a program using the low probe effect trace collection until the program encounters its timing bug. Then one can examine the sequence of events that created this bug. Vision ensures that the interleaving of the replayed computation is identical to that of the original so that, in essence, one can deterministically reproduce a timing problem for detailed examination.

Exception handling is also crucial to finding timing bugs. On a multiprocessor that does not gracefully recover from exceptions such as address errors, the result is usually that the entire multiprocessor soon crashes. This is especially true on architectures that require all processors to act as intermediate stations in routing messages. In Sloop, if an operation encounters an exception, the runtime system stores information such as the invoked object's state, and optionally notifies the debugger. It then destroys the operation

and invokes an operation named cleanup (if defined for the object's type) with the name of the destroyed operation. Both the stored information and the recovery mechanism facilitate the discovery of timing errors.

Vision provides limited interpreter functionality. One can incrementally modify an operation definition, and interactively rerun a computation using the new definition.

Finally, Vision has an extensive program animation capability. Through the debugger, the user can set a display trace on an object. When the runtime system encounters the trace condition, it invokes a programmer-supplied display operation on the traced object. Display operations use device independent graphics commands to create a visual representation of an object on an area of the workstation screen allocated by the debugger. Such animation is very helpful in understanding programs and finding bugs. These graphics capabilities are also available outside of the debugging environment.

4. Specifying Object Locality

This section describes how a programmer can increase the efficiency of a Sloop program by exercising explicit control over the location of objects. For most current parallel architectures, the speed of local (intraprocessor) memory access exceeds the speed of of interprocessor communication. Because of this, operation invocations across processor boundaries will generally take longer than local invocations. To write efficient code, a programmer may need to identify groups of objects that interact heavily, and to specify that these groups should reside on the same processor. A programmer can often customize a distributed object so that it operates with a minimum of communication.

In addition, many parallel algorithms[3] are naturally expressed in terms of a particular topology. Finally, one often needs to tailor a load balancing technique to a particular implementation or to experiment with a variety of load balancing strategies. For these reasons, Sloop provides explicit methods for assigning objects to physical processors and for moving objects from one processor to another.

Recall that Sloop encapsulates virtual object space semantics within the object type *domain*. Sloop gives the programmer explicit control over object location through the domain type's *align* operation. This operation takes two objects as arguments. Upon its completion, both objects reside on the same processor. A third argument indicates whether this arrangement is permanent or temporary by specifying an *alignment mode*. The modes are **weak, strong,** and **replicated**.

Weakly aligned objects may subsequently become separated by future *align* invocations. Strongly aligned objects can not be separated. When an object moves to a new processor, all objects that are strongly aligned to this object move. Thus, one can construct groups of objects that act together in a tightly-coupled manner. **Read_only** objects with alignment mode **replicated** are stored locally in all processors, memory permitting, for efficient access.

One can also use the *align* operation to move objects to specific processors in a network. The Sloop library provides an object type called *marker* to support this. Before running a new program, the runtime system creates a marker object located on each processor in the physical network. By definition, these marker objects never move. The runtime system stores references to each of these markers in an array called `Nodes`. `Nodes[i]` refers to the marker object fixed to the *i*th physical node in the network. To move an object to a particular processor, one aligns the object with the marker corresponding to that processor.

A programmer can create additional marker objects and use them to project multiple layers of topology onto a network. For example, a binary tree maps simply to a hypercube. Several Sloop programs make use of an object type, *hcube_binary_map*, the constructor of which creates marker objects for a binary tree and aligns them with physical processor markers in the correct order for a hypercube. Other objects can use the tree markers without any reference to the underlying physical node markers.

A Sloop programmer can create a mobile environment for an object by the use of *restricted domains*. When creating a domain, one can pass an object reference called the *restriction object* as an argument to its constructor. This produces a domain which strongly aligns every object it creates to the specified restriction object. If the object moves to a different processor, the restricted domain moves with it. One could create an identical situation by explicitly strongly aligning all the necessary objects, but using a restricted domain is much more efficient.

The following example illustrates the use of alignment. Suppose we wish to solve a problem using a master/slave strategy, in which a master object produces subproblems, and *n* slave objects repeatedly request a subproblem from the master and solve it. Normally this pattern requires two message latencies per subproblem, one for the slave's request and one for the master's response. To reduce message traffic, the master and slaves can communicate through the abstraction of a set of subproblems.

This abstraction is implemented by two object types,

3. Particularly for numerical problems.

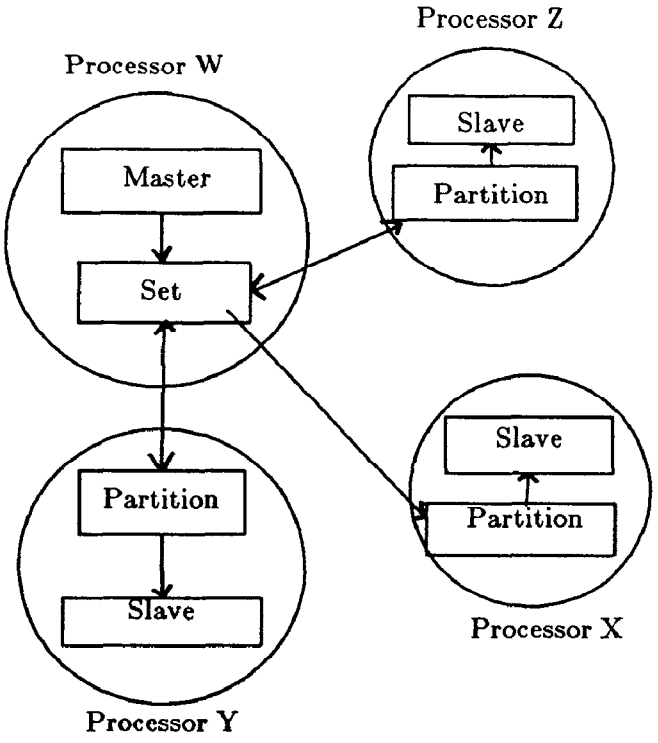

Figure 2--Master/Slave Computation Working Through a Distributed Set

named *set* and *partition*. The set object is aligned with the master object. Each slave object creates and aligns itself to a partition object. As partition objects are created, they notify the set object of their existence. The master inserts subproblems into the set object, which tries to allocate them evenly among all known partitions. Slaves fetch subproblems from their local partition. This data structure allows a master/slave implementation to run very efficiently, because subproblems flow from the master through the set to the partitions, where the slaves can access them directly. The Sloop library contains several such customized data structures, for acceleration of a variety of programming strategies.

5. *Implementation*

The Sloop compiler translates Sloop into C[KR78]. This technique allows code generated by the compiler to easily interface with the runtime system. Because C is available on many machines, compilation into C increases Sloop's portability. The compiler provides a simple inheritance mechanism; it is presently being modified to provide multiple inheritance and dynamic creation of types.

In addition, the compiler performs several optimizations. The most important of these, *return transfer*, is similar to tail recursion elimination. If a return statement in an operation *a* contains an expression invoking an operation *b*, the compiler attempts to transfer the responsibility to operation *b* for returning a value to the object that invoked *a*. This leaves the object on which *a* executed free to execute other operations without waiting for the return value from *b*.

To visualize this situation, imagine a list, each element of which is an object containing an integer. To determine if a given integer matches any of the integers contained in the list, one invokes a *match* operation on the first element. If the argument to this operation matches the integer in the first element, this element returns TRUE; otherwise, it returns the result of invoking *match* on the second element. Return transfer frees each list element as it invokes *match* on the next element. The element that eventually returns a value does so directly to the object invoking the original *match* operation.

The Sloop runtime system is written in the object-oriented language C++[St86]. Use of C++ aided in making the implementation portable: ninety-five percent of the code is common to the S/Net, hypercube, and distributed workstation implementations. The use of C++ also facilitated extension of the runtime system to allocate multiple root domain objects, allowing several disjoint computations to execute simultaneously. Presently, Sloop allows an arbitrary number of programs to share a multiprocessor, with a distributed hashing scheme mapping objects to processors. [Lu86] gives details of the hashing technique.

Three key features account for the high performance of the Sloop runtime system. First, the system uses coroutines rather than processes to run operations. Second, the communication protocol that implements interprocessor operation invocations was carefully tuned. Finally, the system heuristically moves objects around the network, attempting to place heavily interacting objects on the same processor.

On each processor, the system maintains a pool of coroutines, assigning operations to coroutines as the operations become executable. An operation's coroutine surrenders control to the system whenever it invokes some other operation. The system can switch from one coroutine to another in eight microseconds on a SUN 3/160 (which has a Motorola 68020 processor running at 16.67MHz), about the cost of a function call. The system does not assign an operation to a coroutine if the operation would block on its await statement. Instead it defers the operation, checking the await predicate when appropriate. The runtime system provides a hook into its scheduler to support real time applications. Through a system call, the user specifies a time interval, an object, and an operation. The system will then periodically invoke the operation on the object.

The runtime system may move objects around the network for several reasons. First, the system moves objects in response to explicit alignment. Second, the system moves objects to conserve memory on particular processors. Third, the system moves objects to balance computational load. If an object runs for a long time without invoking an operation, the system will move other objects away from it.[4] The system will, if necessary, override program-specified alignments to perform the latter two functions.

Finally, the system maintains statistics on a program's communication patterns and relocates unaligned objects to minimize overall message traffic. The most important such heuristic detects the use of an intermediate object for point-to-point communication. For example, two objects may communicate using a queue. Even though only these two objects invoke operations on the queue, the system may initially assign the two objects and the queue to three different processors. Upon discovering this situation, the runtime system aligns the queue (or other intermediate objects) with the object that sent the greatest number of messages to it. In practice, this heuristic works well for a variety of Sloop programs, especially those that do not use explicit alignment. For instance, it increased the speed of a distributed circuit simulator by 250% [Lu86].

One design consideration for the runtime system was to prepare for future architectures. Presently, over ninety percent of the cost of an interprocessor operation invocation is external to the Sloop runtime system, in the various multiprocessors' message passing mechanisms (software and hardware). Extrapolating from recent work on hardware routing[DS86] and object-based communication [Kr86], it seems likely that interprocessor operation invocation and object relocation will soon enjoy tremendous gains in speed. Such architectural improvements will decrease the need for explicit control of object placement, as the placements provided by runtime heuristics become sufficient for a larger proportion of programs. As message costs decrease, the fast context switching provided by Sloop's coroutines also grows in importance.

6. Conclusion

The Sloop project's goals are to study parallel programming methodology and to extend the object-oriented computational model to accommodate clear and efficient expression of parallelism. Our experiences indicate that organizing a program in terms of distributed objects can greatly simplify its construction and operation.

Using Sloop's associative, asynchronous access mechanism, objects located throughout a multiprocessor can interact without explicitly synchronizing. This facilitates the construction of Sloop programs that dynamically vary the amount of computational power devoted to a particular subtask. Sloop also provides a mechanism for specifying groups of objects that should, for efficiency, reside on the same physical processor, and facilities that permit exploitation of the topology of the underlying parallel machine. Sloop supports the creation of indivisible objects, which provide implicit concurrency control.

Through encapsulation of these virtual object space semantics within the *domain* object type, Sloop facilitates program debugging and modularity. In addition, the Sloop compiler and runtime system automatically accelerate program execution through the return transfer optimization, heuristic load balancing, and coroutine scheduling. Finally, the Sloop environment includes a debugger called Vision which introduces concepts such as object capture and deferred tracing to facilitate parallel debugging.

4. The newest version of Sloop experiments with more complex load balancing heuristics.

References

[LN87] S. Lucco and K. Nichols, "A Performance Analysis of Two Parallel Programming Methodologies in the Context of MOS Timing Simulation", *Proceedings of 1987 Spring Compcon*, IEEE Computer Society (February, 1987).

[GR83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983

[Ah83] S.R. Ahuja, "S/NET: A High Speed Interconnect for Multicomputers," *IEEE Journal on Selected Areas in Communications*, (November, 1983).

[De85] E. DeBenedictis, "Multiprocessor Programming with Distributed Variables," *Proceedings of SIAM Conference on Hypercube Multiprocessors*, (August, 1985).

[Ge85] D. Gelernter, N. Carriero, S. Chandran, and S. Chang, "Parallel Programming in Linda," *Proceedings International Conference on Parallel Processing*, (August, 1985).

[Br75] P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering 2*, (June, 1975), pp. 199-205.

[Lu87] S. Lucco, "Parallel Debugging with Vision," *in preparation*.

[KR78] B. Kernighan and D. Ritchie, *The C Programming Language*, Addison Wesley, Reading, MA, 1978.

[St86] B. Stroustrup, *The C++ Programming Language*, Addison Wesley, Reading, MA, 1986.

[Lu86] S. Lucco, "A Heuristic Linda Kernel for Hypercube Multiprocessors," *Proceedings of the Second Conference on Hypercube Multiprocessors*, SIAM (September, 1986).

[DS86] W. Dally and C. Seitz, "The Torus Routing Chip," *Distributed Computing 1* (1986), pp.87-196.

[Kr86] V. Krishnaswamy, S. Ahuja, D. Gelernter, and N. Carriero, "Progress Towards a Linda Machine," *Proceedings of the ICCD*, IEEE Computer Society and IEEE Circuits and Systems Society, (October, 1986), pp. 97-101.