The Amber System: Parallel Programming on a Network of Multiprocessors

Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska,
Henry M. Levy, and Richard J. Littlefield

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

This paper describes a programming system called *Amber* that permits a single application program to use a homogeneous network of computers in a uniform way, making the network appear to the application as an integrated multiprocessor. Amber is specifically designed for high performance in the case where each node in the network is a shared-memory multiprocessor.

Amber shows that support for loosely-coupled multiprocessing can be efficiently realized using an object-based programming model. Amber programs execute in a uniform network-wide object space, with memory coherence maintained at the object level. Careful data placement and consistency control are essential for reducing communication overhead in a loosely-coupled system. Amber programmers use object migration primitives to control the location of data and processing.

1 Introduction

Small-scale shared-memory multiprocessors are becoming widely available in implementations ranging from single-user workstations to mini-supercomputers. The proliferation of multiprocessors means that local area networks of these systems are likely to become common. This presents the opportunity to program a group of these machines to work together on a single application. For many applications, networks of small-scale multiprocessors will have greater performance potential than the fastest mainframes at significantly lower cost, and with greater flexibility.

Amber was designed to take advantage of this trend by supporting the development of parallel applications that use multiple machines in a network of shared-memory multiprocessors. Amber provides a set of programming facilities and abstractions that isolate the programmer from the low-level details of programming in this environment. The abstractions are intended to simplify communication, distribution, and parallelism, while supporting a dynamic program structure that can express and benefit from locality.

Amber is based on a model of computation in which a collection of mobile objects distributed among nodes in a network interact through location-independent invocation. Amber objects are passive fine-grained entities consisting of private data and a set of public operations that can be locally or remotely invoked. The active entities in the system are *thread* objects, which possess processor state and a runtime stack and can execute on a CPU. A typical application might contain many threads concurrently executing object operations on different processors in a node and on different nodes in the network. The threads in an Amber program execute in a flat network-wide shared object space. Object references can be transmitted across node boundaries and dereferenced on any node with consistent semantics, allowing programs to operate on distributed data structures in a uniform way.

Amber programs are written in an object-based subset of the C++ programming language [Stroustrup 86], supplemented with primitives for thread management and object mobility. The system is composed of a preprocessor to C++ and a runtime kernel which is linked with the user's program. Amber is implemented on the Topaz operating system for the DEC Firefly [Thacker et al. 88], a multiprocessor workstation based on VAX microprocessors. Applications have been executed on a group of eight Fireflies connected by a 10-megabit/second Ethernet.

This material is based on work supported by the National Science Foundation (Grants CCR-8611390, CCR-8619663, CCR-8700106, CCR-8907666,and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center, the External Research Program, and the Graduate Engineering Education Program).

1.1 Research Goals and Issues

Amber explores issues in parallel programming at both the system level and the application level. At the system level, Amber explores the viability of using a loosely-coupled network of small-scale multiprocessors as a large-scale machine. This raises issues in scheduling, virtual memory management, distribution, and coherency. At the application level, Amber explores how to structure an application to benefit from the target architecture. Here we wish to understand the programming primitives needed to express both locality and parallelism. Overall, Amber assesses the appropriateness of the object-oriented programming model for solving problems at both levels.

A major goal of the Amber project is to provide language-level support for concurrency and distribution using an existing programming language and operating system. This reduces the development cost of the system and makes it more attractive to programmers. We were able to achieve this goal using the facilities of the Topaz operating system and the C++ programming language. Topaz provides useful network services, support for threads, and remote procedure call [Birrell & Nelson 84]. The extensible class hierarchy of C++ enabled us to implement Amber without modifying the language or its compiler.

Amber's programming model assumes that the user will be aware of the network organization and will wish to take advantage of it to achieve maximum application speedup. This conflicts with the goal of network transparency. The tension between uniformity and performance is more pronounced in Amber than in other distributed systems because high performance for parallel applications is the primary purpose of the system. Good performance on a loosely-coupled multiprocessor demands careful attention to data placement in order to minimize remote references, which are three to four orders of magnitude more expensive than local ones. Our view is that data placement should be under the control of the program rather than the runtime system since the needs of different applications will vary widely. This compromises the uniformity of the programming model because it requires the programmer to deal explicitly with location. Amber attempts to strike a balance between uniformity and performance. The programming model is designed to isolate areas where the programmer must be concerned with location, while providing means to tune program organization for efficient execution.

1.2 Related Systems

Object-based models have frequently been used for distributed systems; examples include Hydra [Wulf 74], Clouds [Allchin & McKendry 83], Argus [Liskov 88], Eden [Almes et al. 85], and Cronus [Schantz et al. 86]. Systems such as Eden and Argus provide support for distributed objects at the programming language level using a special-purpose language. Experiments with this approach at the University of Washington led to the development of Emerald [Black et al. 87], a distributed programming language with support for fine-grained object mobility. Amber's distribution model and mobility primitives are derived from Emerald.

In contrast to these other systems, the goal of Amber is to execute a single application that performs a parallel computation, computes a result, and terminates. Amber does not support persistent objects, primitives for reliable distributed computing, or communication and cooperation between unrelated programs. In this respect Amber is related to object-based systems for concurrent programming on tightly-coupled machines. Amber is in fact a direct descendent of one such system, Presto [Bershad et al. 88a], a C++-based runtime package for building medium-grained parallel applications on shared-memory multiprocessors. Amber's thread model and synchronization model follow those of Presto.

Amber was developed to allow a network of machines to be treated as a loosely-coupled multiprocessor using a programming model based on distributed objects. Recent systems with similar goals include Sloop [Lucco 87] and Orca [Bal & Tanenbaum 88]. Amber differs from these systems in that its programming model and internal structure are designed to take advantage of shared-memory multiprocessors. Amber supports logical concurrency and true parallelism within each node as well as across nodes in the network. Concurrency within a mutable object is realized by placing the object on a single node and clustering all threads manipulating the object onto that node. This allows the consistency of the object's internal state to be efficiently managed by hardware-based synchronization primitives and memory coherence protocols. This organization of Amber programs into closely-cooperating clusters is similar to the task force structure in Medusa [Ousterhout et al. 80] and StarOS [Jones et al. 79], but in Amber this clustering is determined at run time and can change dynamically as the computation progresses.

Other researchers have investigated the use of a network as a loosely-coupled multiprocessor within the context of more traditional programming models. The Ivy system [Li 86] pioneered the use of network-wide shared virtual memory for this purpose. This approach allows distributed applications to be written using conventional programming techniques. Ivy maintains memory coherence by using virtual memory hardware to implement page ownership schemes analogous to hardware cache consistency protocols. One goal of Amber is to explore the relative merits of object-based versus shared-memory models for maintaining memory coherence in a parallel and distributed environment. This issue is discussed in Section 4.

2 The Programming Model

Amber provides the programmer with a set of predefined object classes for managing threads, synchronization, and distribution. Amber's abstractions are supplied by means of existing C++ language mechanisms such as subclasses and dynamic object creation. The programming model demonstrates that support for concurrency and distribution can be integrated into a class-hierarchical language to produce a uniform system with features that compose well.

The use of an object-oriented language provides other benefits. Object classes can hide not only the representation of objects but also the internal details of their execution, synchronization, and location. Other researchers have discovered similar benefits for implementing features such as persistence and recovery properties for objects [Herlihy & Wing 87]. Also, dynamically typed subclasses are a convenient vehicle for tailoring system behavior to meet the needs of a specific application. These ideas are made more concrete in the next few subsections, which present the details of Amber's programming model.

2.1 Threads

Amber's mechanisms for expressing concurrency are derived from the thread facilities provided by Presto. Presto was designed to make the use of threads inexpensive, allowing the programmer to efficiently manage more control streams than there are processors. In Amber, threads have the advantage of allowing the programmer to maximize throughput by overlapping computation with remote communication.

Like other objects, threads can be created dynamically using the C++ *new* operator. The basic operations on threads are *Start* and *Join*. The *Start* primitive starts a thread executing an operation on a specified object. *Join* blocks the caller until the specified thread terminates, returning the result from the operation specified in the *Start* call.

Threads provide explicit support for concurrency, in contrast to the implicit support provided by asynchronous object invocation mechanisms in languages such as Sloop. Amber invocations are synchronous, but threads can be used by either the invoking object or the invoked object to transparently provide asynchrony. An invoked object can exploit parallelism transparently to its invoker by creating and starting additional threads in response to an invocation. Alternatively, a thread can execute an asynchronous invocation by creating another thread to perform the invocation.

Amber's scheduler supports timeslicing and can be customized to use priority-based or adaptive policies tuned to the specific application. An application can install a custom scheduling discipline at runtime by replacing the system scheduler object with a similar object that supports the same interface but behaves differently [Bershad et al. 88b].

2.2 Synchronization Objects

Amber provides a flexible set of classes for controlling access to data shared by multiple threads. The system supports relinquishing and non-relinquishing locks, barrier synchronization, monitors and condition variables. Programmers can extend the class hierarchy to define custom mechanisms for concurrency control using these primitive synchronization objects. The intent is that programmers will select an appropriate concurrency control scheme for each user object and encapsulate the details of the synchronization within the class.

Amber's approach to synchronization differs from similar systems designed for networks of uniprocessors. Most of these systems support monitored objects and indivisible operations but no explicit lock primitives. We believe that fine-grained synchronization using lock primitives is desirable when the nodes in the network are multiprocessors. Fine-grained locking reduces contention and allows hardware-based spinlocks to be used to reduce latency when appropriate. Lock objects have additional advantages in a distributed environment because they are mobile and can be remotely invoked to enforce concurrency constraints involving multiple objects on different nodes.

2.3 Controlling Object Location

In Amber, threads invoking operations on an object move to the node where the object resides, so the division of computational load between the machines is determined by the locations of the program's data objects. Object location also has a significant effect on the network overhead incurred by the program. In general, interacting objects should be co-located in order to avoid the cost of a remote procedure call on each invocation. This must be balanced with the need to place objects so as to evenly distribute the computational load between machines.

Amber programmers take advantage of locality by using migration primitives to control object placement as the program executes. Objects can be moved even if they have active invocations: threads executing operations on a moving object are identified and moved with the object. Dynamic mobility is useful because some applications will need to reorganize object locations following different computational phases of a program, although static object placement is sufficient for many applications.

Amber's mobility primitives are modeled after mobility in the Emerald system [Jul et al. 88]. An Amber object can be moved with *MoveTo* and its location can be determined with *Locate*. Like Emerald, Amber provides other mobility primitives that are useful for improving program performance. The programmer can *Attach* an object to another object or *Unattach* an attached object. The attachment primitives allow a programmer to dynamically create structures of objects that move together and are always guaranteed to be co-located. Amber also supports replication of

readonly objects to reduce unnecessary communication overhead. Objects may be marked as *immutable* at runtime, indicating that they will never again be modified. Invoking *MoveTo* on an immutable object causes the object to be copied rather than moved. The attachment and immutability mechanisms in Amber are more dynamic than in Emerald, where they are specified at compile time.

Amber leaves all aspects of object location under the direct control of the program. Data objects never move unless the program explicitly moves them. Objects that are not explicitly designated as immutable are never replicated, eliminating the complexity associated with keeping multiple copies of a writable object consistent. This approach contrasts with other recent experiments with language-level support for distributed objects. The Orca language performs automatic object placement and replication of mutable objects, but provides no primitives for explicit object migration. Sloop includes advisory migration primitives, but the system may override the programmer's decisions under certain conditions. Amber attempts to provide a model of sharing and location that is uniform, predictable, and simple to implement. Our assumption is that the best policy for managing location is application-specific and is best left to the program or higher-level object placement software.

3 Implementation Issues

Amber programs execute as a set of cooperating Topaz tasks distributed across the network, with one task on each participating node. The tasks are created at program startup using Topaz facilities for creating remote processes. Each task is an execution of the same program image read from a distributed file system. Topaz supports multiple threads of control in a single task and fast remote procedure call between tasks [Schroeder & Burrows 89], facilities that are used to implement Amber thread scheduling, object migration, and internode object invocation.

The key implementation problem for Amber is the abstraction of a single network-wide object space with object mobility and transparent invocation of remote objects. The following subordinate issues must be addressed in order to implement this model:

- naming, creating, and destroying objects
- moving objects
- trapping nonlocal invocations
- finding remote objects
- migrating threads for remote invocations

Our goal was to implement Amber's shared object abstraction within the confines of C++, using techniques that perform little or no remote communication not directly requested by the program. We found that much of the implementation was straightforward if we used direct virtual addresses as the basis for object naming in the network, arranging each task's address space so that virtual addresses have the same meaning on all nodes. The next few subsections describe our implementation for Amber, with an emphasis on how this global virtual address space is managed and how it simplifies the implementation of the shared object space abstraction. Section 3.5 presents additional problems caused by intranode parallelism, and Section 3.6 discusses our use of the C++ language.

3.1 The Global Address Space

Object references and other pointers are frequently transmitted across the network in Amber. This happens when arguments to a remote invocation are passed by reference, or an object containing embedded pointers moves from one node to another. Also, any thread executing an operation on a moving object will move with the object and resume execution on the destination node, where it will continue to use addresses stored in its stack and registers. The transmitted addresses may be object references, program code addresses, pointers into static data such as string constants, or back links in the stack. It follows that all code and data items are visible to all nodes and may be referenced by any thread regardless of which node it is running on. The references must be resolvable on all nodes with uniform semantics.

One solution to this problem is to translate addresses whenever they cross node boundaries. This is the solution used in Emerald [Jul et al. 88]. Such a scheme permits each node to do independent memory management, which is useful for Emerald because it assumes a universe of long-lived objects created by multiple users. The problem with this approach is that it requires extensive compiler support to aid in the address translation. This is incompatible with our goal of using an existing widely-used language and compiler for Amber.

Amber avoids the need for address translation by ensuring that addresses retain their meaning when transmitted across node boundaries. The global virtual memory is implemented by arranging the virtual address space of each participating Topaz task identically. Program code and statically initialized program data are automatically replicated at the same addresses on all nodes because the tasks are activations of the same program image executed on homogeneous machines. All dynamic objects (including thread objects and their stacks) are assigned a distinct segment of the global address space when they are created, and each object occupies this same virtual address range on any node that it visits during its lifetime. The segment of virtual memory occupied by an object on one node is reserved for that object on all other nodes.

Amber's memory organization requires that nodes use disjoint regions of the address space for heap allocations of dynamic objects. The system must guar-

antee that two nodes do not attempt to allocate the same heap block, but it must do this without the expense of distributed agreement for each object allocation. Each node is assigned a private region of the virtual address space at startup time for its local heap allocations. Statically partitioning the entire address space in this way is limiting because objects are not allocated uniformly across nodes. For this reason, a large part of the address space is left unallocated at startup and is handed out later by an address space server as nodes exhaust their initial pool. The cost of extending the address space is not excessive because the regions are large enough (currently 1M bytes) that extensions are needed relatively rarely for applications that are moderate in their use of memory.

3.2 Handling Remote References

Each Amber object is referenced by a virtual address that is valid on any node, but the system must determine whether or not an object is local when it is invoked. To provide this information, each object has an *object descriptor* on every node that indicates whether or not the described object is locally resident. The descriptor may contain other information about the object, such as where to look for it if it is remote. Checks inserted by the preprocessor examine an object's local descriptor on each invocation. If the descriptor indicates that the object is remote, the invocation traps to the Amber kernel and is handled by a remote procedure call that moves the faulting thread to the node where the invoked object resides.

An Amber object is implemented as a record, the first part of which is its descriptor, and the remainder of which is its representation (the data local to the object). The virtual address of an object is therefore the address of its descriptor. When a new object is created it is allocated from the heap on a particular node. The descriptor for the object is initialized on that node to indicate that the object is resident so that it can be invoked. If a mutable object is moved, its descriptor is changed to indicate that it is not resident, and a forwarding address is inserted in the descriptor.

Objects and their descriptors are managed so that an uninitialized descriptor is detected and interpreted to mean that the object is remote. This eliminates the expense of initializing remote descriptors for a newly created object. An uninitialized descriptor is detected because unwritten pages of virtual memory are zero-filled by the Topaz operating system, and object descriptors are defined so that the resident flag is a one-valued bit. References to objects occupying heap blocks that were previously deallocated and reused are also handled correctly. This requires that the heap allocation algorithm be constrained so that heap blocks are never divided once they have been returned to the free pool.

3.3 Locating Mobile Objects

When the kernel handles a trap on an invocation of a remote object, it retrieves a forwarding address [Fowler 85] from the object's local descriptor. The forwarding address is left in an object's local descriptor when the object moves away from a node. The forwarding address may be out of date if the object moves frequently. In this case the object's location can be determined by following a chain of forwarding addresses, since the object leaves a new forwarding address on each node that it visits. It is costly to locate an object by following a forwarding chain, but this happens rarely because the object's last known location is cached on all nodes along the chain so that the object can be located quickly on subsequent references.

The situation is more complicated in the case of a trap on an object with an uninitialized descriptor, indicated by the presence of a null forwarding pointer. Each task has complete knowledge of the assignment of heap regions to nodes because a reference to the node that owns each heap region is obtained from the address space server when the region is first mapped by a task. This allows the system to use a heap object's virtual address to identify the object's *home node*, the node on which it was created. When a reference to an object with an uninitialized descriptor is detected, the kernel forwards the request to the object's home node. The home node can determine where the object resides by following the chain of forwarding addresses.

3.4 Object and Thread Migration

The global virtual address space simplifies object migration because it avoids the need to translate addresses stored in the moving object. Furthermore, there is no need to allocate space on the target node for the object since the address range that it will occupy is predetermined. Moving an object involves copying its contents from the source node to the destination node and updating its descriptors on both nodes. The implementation is complicated by the need to identify and move threads that are actively executing operations on the object. These *bound threads* must migrate with the object in order to preserve the consistency of the object's contents. This problem is discussed in Section 3.5.

Amber remote invocations are performed by simply moving the invoking thread to the remote node. In principle this is no more complicated than any other object move. The thread's control information and pieces of its stack are copied to the same address ranges on the remote node, the object descriptors are updated, and the thread is added to the scheduling queue on the remote node. Addresses in its processor registers and stack will continue to be valid on the destination node. In practice, thread migrations are handled slightly differently from migrations of other objects in order to optimize remote invocations made by the thread at the expense of invocations made on the thread object itself (e.g., a *Join* operation).

3.5 Object Mobility on Multiprocessors

Dynamic mobility is difficult to implement on multiprocessors because user threads may be attempting to manipulate a moving object concurrently with the mobility code running on another processor. This leads to a number of implementation concerns that would not arise on a uniprocessor. In uniprocessor object migration all threads on the node are implicitly suspended while mobility code in the kernel is in control of the single processor, making it a simple matter to determine which threads are bound to the moving object by examining their stacks. Furthermore, the system can preserve the atomicity of the invocation sequence by preempting threads only at safe points in their execution, avoiding a context switch between the residency check and the completion of the stack modifications indicating that an invocation is active.

On multiprocessor hardware the atomicity of descriptor checks can no longer be guaranteed. In a naive implementation a thread could check the descriptor and find that the object is local, but not actually complete the local invocation until after a move operation on the object has been initiated by another processor. This race condition will always exist if the descriptor is checked before the stack modifications associated with the invocation are made. An analogous race condition can occur on returns: a thread could check that an object is still resident before its return, only to have the object move after the check but before the actual control transfer. A related problem is that the set of active threads bound to a moving object is constantly changing while the mobility code is running.

One approach to solving these problems is to lock the invocation sequence and maintain a data structure that records which threads are currently executing within each object. This solution makes invocations expensive because of the need to synchronize and update the data structure. Another approach is to freeze all activity on the node during a move operation and examine the stacks of all local threads to determine which threads are bound to the moving object. This solution optimizes invocations but makes move operations complex and expensive. There are many gradations between these extremes.

Amber makes invocation-time residency checks at the start of each operation, after the invocation stack frame is pushed but before any user code is executed. Return-time checks are made immediately after the invocation frame that the thread is returning from has been popped. This guarantees that the executing thread can be identified as bound to the object before it actually checks the descriptor and enters the object. Threads already bound to a moving object are handled by an additional residency check that is made on each context switch into a preempted thread. Move operations in Amber preempt and reschedule all threads running on the source node, forcing them to make a residency check before they continue. The preemptions occur after the descriptor of the moving object has been marked as non-resident but before the object's contents have been copied to the remote node.

Local invocations are efficient with this scheme because they require no synchronization over the object descriptor, only a residency check consisting of a single VAX branch-on-bit-set instruction. Also, there is no need to halt all activity on a node during a move operation; at worst it will be necessary to briefly interrupt each processor. One problem is that some concurrency may be lost if the destination node is idle but the source node is busy, since suspended threads which are bound to the object will not move to the destination node until they are rescheduled on the source node. An added disadvantage is that the need to preempt all running threads causes the cost of mobility to increase as processors are added to a node. The assumptions behind these tradeoffs are (1) object moves are much less frequent than object invocations, and (2) improvements in processor speeds will make thread preemptions cheap relative to the network latency associated with a move operation.

3.6 Experience With C++

Our choice of C++ was partly motivated by its availability and its popularity with programmers. Another advantage of C++ is that it is efficiently implemented with a minimum of runtime support. Most other benefits of using C++ could have been obtained from any object-oriented programming language with an extensible class hierarchy and dynamic typing.

In our Amber prototype, object descriptors are allocated and managed by deriving all user classes from a single base class called *Object* whose private data items include the descriptor. The constructor and destructor functions for the *Object* class maintain the descriptor and ensure that object creation and deletion meet the requirements discussed in Section 3.3. Threads and synchronization objects are provided by introducing new subclasses of *Object*. The mobility primitives are operations on instances of class *Object*. Amber's distributed heap allocation is implemented by redefining the runtime library routines for the C++ operators *new* and *delete*.

One problem with C++ is that Amber's distribution model depends on the regularity of an object-oriented programming language. Amber assumes that a thread will never directly manipulate the internals of a remote object, since references to remote objects are recognized and trapped only on invocations. Furthermore, all data items that may be referenced remotely must be encapsulated in an object. The C++ language includes many performance features that circumvent constraints normally associated with an object-oriented programming model. Examples of such features are friends, public member elements, inline functions, unprotected structures, and the ability to include arbitrary C code in the program. These features can result in incorrect program behavior if they are used improperly in

a distributed environment. Nevertheless, they present opportunities to optimize interactions between objects that are known to reside on the same node.

There are a number of situations in which co-residency guarantees make it possible to use these features safely. Co-residency can be explicitly requested using Amber's attachment primitives. Also, C++ *member objects* (objects that are directly contained within some other object) always move with their containing object and are therefore co-resident with it. Co-residency guarantees can also be exploited to optimize invocations of functions in base classes or invocations of objects allocated from a thread's stack. Intelligent use of the performance features of C++ in situations where co-residency is assured can significantly improve program performance. For example, consider a multi-threaded object whose internal state is protected by a non-relinquishing lock. If the lock is a member object of the protected object then it can be safely acquired and released using fast inline function calls.

4 Comparison with Shared Virtual Memory

This section explores the relationships between page-oriented and object-oriented shared memory models. Both approaches offer uniformity relative to an RPC-based programming model, but they differ in other respects. The original motivation for an object-oriented memory in Amber was that objects are a natural unit for involving the programmer in data placement decisions. In this section we shall argue that the object is also a natural and efficient unit for maintaining coherence of the global address space, and that object-level coherence has a number of advantages over page-based coherence.

The memory organization of a loosely-coupled system is closely related to issues of consistency of the data shared by multiple nodes. At the hardware level each node can address only its private physical memory. Coherence of these private memories is difficult to maintain efficiently in a distributed environment. A similar problem is encountered by the designers of programming support for NUMA multiprocessors, where the varying costs of referencing different areas of memory motivate the use of caching, replication and data migration to improve program performance. NUMA programming systems such as PLATINUM [Cox & Fowler 89] make hidden data placement and replication decisions while presenting the programmer with a view of memory that is uniform and coherent at the byte level. This approach can work well for NUMA multiprocessors because the cost of a poor placement decision is typically not very high.

Similar shared memory models have been used to allow a network of machines to be programmed as a loosely-coupled multiprocessor. In Ivy, distributed processes execute in a global virtual address space with consistency of arbitrary bytes guaranteed across references from multiple nodes. Coherence of the shared memory is maintained by memory managers on each node, which use page faults to detect shared accesses and exchange coherency messages with other memory managers [Li & Hudak 86]. Remote references are handled by moving or copying the referenced page to the location of the faulting process. Distribution and load balancing are achieved by explicit process migration.

Amber represents an alternative vision of uniform and consistent memory in which the granularity of data coherence is the object rather than the individual byte. These systems present the programmer with a network-wide object name space, with consistency maintained by trapping invocations of remote objects. This memory model is uniform in the sense that it is unnecessary for the programmer to deal explicitly with the locations of objects when they are invoked, but it is more restrictive than the shared virtual memory approach because it requires adherence to an object-oriented programming discipline.

4.1 Function Shipping

A major difference between Amber and Ivy is that Amber takes a function-shipping rather than a data-shipping approach to coherence. Instead of attempting to maintain the consistency of mutable objects across references from multiple nodes, each object is placed on a single node where access to it is controlled through its operations. Function shipping is especially attractive when the nodes in the network are shared-memory multiprocessors because it clusters the threads referencing a given object onto the same node, where hardware-based synchronization and memory sharing can be used to their fullest performance advantage. The programmer of a data-shipping system such as Ivy can obtain the same advantages through an appropriate use of explicit process migration.

Distributed synchronization is simple and efficient in a function-shipping system. For example, Amber locks are objects which can be remotely invoked to synchronize threads executing on different nodes. References to a shared lock variable can cause a data-shipping system to thrash by repeatedly shuttling the page containing the lock variable between the nodes which are referencing it. Recent versions of Ivy have handled this problem by deviating from the data-shipping model and accessing shared lock variables with remote procedure calls.

For a certain class of programs the behavior of the function-shipping approach is more predictable than that of the data-shipping approach. It is easy to predict the communication overhead incurred by an Amber program that utilizes static object placement or that moves objects at well-defined points. A similar program for a data-shipping system can thrash when a memory page is repeatedly referenced by processes on different nodes. The Amber program can thrash when a thread repeatedly invokes the same remote object,

but this effect is less dependent on the orders of events and the timings of concurrent operations (except those involving explicit object moves). In such a program the location of an object can be determined from the program structure and is independent of which threads happen to be referencing the object at the moment.

4.2 Pages vs. Objects

The performance of a coherence policy is dependent upon the degree to which memory references made by the program are localized within the units used by the system to maintain coherence. In a distributed object system the granularity of coherence is the data object, a problem-oriented unit, whereas in shared memory systems it is the page, a unit that is dependent upon the hardware rather than the structure of the program.

The performance of a page-based coherency scheme may suffer if the sizes of data items do not match well with the page size. If a remote data item is larger than a page, an operation that accesses the item in its entirety will generate multiple page faults unless the process is explicitly moved to the location of the data item. In Amber, the thread moves to the location of the data item and the operation executes with a single network transaction. Alternatively, the Amber programmer can choose to migrate the object explicitly, making use of an efficient bulk transfer protocol.

If data items are smaller than a page, a page-based coherency scheme incurs unnecessary communication overhead when logically unrelated data items that happen to reside in the same page are referenced repeatedly by multiple nodes. The programmer for such a system must be aware of page sizes and boundaries to reduce this artificial sharing, just as programmers of current shared-memory multiprocessors need to be aware of cache line sizes in order to achieve the best performance. Page-based systems can reduce these problems by depending on the compiler to structure the data appropriately. This structuring comes for free in an object-based system.

Another argument for object-level coherence is based on a hypothesis that the memory reference patterns of object-oriented programs are more localized than similar programs using more traditional models. The body of an object operation can reference only the thread stack and the contents of the object itself, so an executing operation is likely to make a sequence of memory references local to the current object. In effect, there is knowledge implicit in the way the data area is divided into objects that can be exploited to make the coherence algorithm more efficient.

5 Cost of Amber Operations

The true test of Amber's performance is the behavior of applications built with the system. Section 6 describes a simple application and discusses its performance. It is also useful, though, to measure the cost of the primitives for concurrency and distribution. Table 1 presents some timings for basic Amber operations, as measured on Firefly workstations with four CVAX processors available for running user threads. The latency of these operations is highly sensitive to a number of factors, but the benchmarks that produced these timings attempt to measure the cost of the operations in the most common case. For example, the benchmarks assume that all moving objects and threads will fit in a network packet, and that the destinations are found by following a forwarding chain for one hop. These timings should be regarded as rough indications of the cost of the operations under light load conditions. Operations involving thread scheduling or network communication are more expensive on a heavily loaded system.

Operation	Latency (ms)
object create	0.18
local invoke/return	0.012
remote invoke/return	8.32
object move	12.43
thread start/join	1.33

Table 1: Latency of Amber Operations

We expect that the CPU cost of these operations will have less effect on program performance in the future. As processors get faster the CPU overhead of using any distributed system becomes less significant, and the performance of the system is dominated by network latency, which will remain roughly constant despite the advent of new high-throughput networks. The performance of a distributed system is best evaluated not by the cost of basic network operations, but by the degree to which the system prevents unnecessary network communication.

6 An Amber Application

This section presents the structure and performance of an Amber program that computes the steady-state temperature over the interior of a square plate given the temperatures around the plate's boundary. The behavior of this system is governed by Laplace's equation, which states that the value at each point is the average of the values of its neighbors. The algorithm used is Red/Black Successive Over-Relaxation (SOR), an iterative method that parallelizes well and is commonly used in practice [Ortega & Voigt 85]. This algorithm can be understood by analogy to a checkerboard. Each point of the problem grid corresponds to a square on the checkerboard. During each iteration, all of the black points are updated first, followed by all of the red points. After some number of iterations the computed values converge and the algorithm terminates. Black points have only red neighbors and vice versa, so each

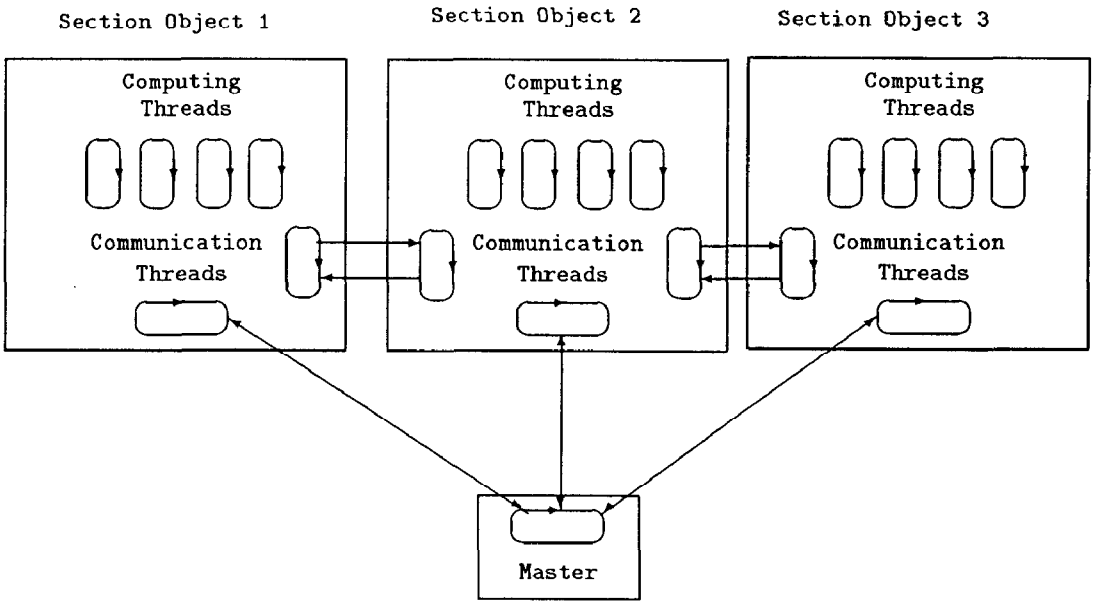

Figure 1: Structure of the Amber Red/Black SOR Implementation

of the update phases is highly parallelizable.

The algorithm is partitioned for loosely-coupled parallel execution by breaking the grid into sections and distributing the sections among the nodes. Some partitionings are clearly inefficient. For example, placing the entire grid in one object would result in unbalanced use of the available processing power. Placing each point in a separate object would involve excessive communication overhead. A more effective approach is to choose the partitioning so that one section object can be assigned to each node. This balances the load and allows the values for an entire edge of a section to be transferred in a single invocation.

The Amber SOR program has several sets of threads associated with each section object. One set of threads computes the values for the section's points in parallel on each iteration. Another set of threads is responsible for exchanging edge data with neighboring sections. The exchange of values for edge points of one color is overlapped with the computation for points of the other color. After each iteration the nodes synchronize at a barrier to determine if convergence has been reached. One additional thread per section is responsible for communicating with a single master thread regarding convergence. Figure 1 displays this structure for a decomposition with three sections.

The SOR algorithm is well-suited to a loosely-coupled multiprocessing model because the problem is regular and static, which makes it easy to choose a partitioning that balances the load evenly. The amount of computing required per section on each iteration depends only on the size of the section and is not affected by the data contained there. Nevertheless, SOR is a nontrivial algorithm which is typical of many iterative methods involving nearest-neighbor interactions. Performance measurements for the program are shown in Figures 2 and 3.

Figure 2 plots measured speedup of the SOR program as the number of nodes and the number of processors increases. For the purposes of this experiment, we selected a specific problem with a grid size of 122 by 842 points. Most of the partitionings were into eight section objects, except for the experiments involving three and six nodes, which were run with partitionings of six section objects. A significant amount of remote communication is required to solve this problem on multiple nodes. Each point in this figure represents the measured speedup for a particular experiment relative to a sequential C++ implementation used as the baseline case. Each point is labeled to indicate the number of Firefly nodes used, and the number of processors per node. For example, the point labeled "4Nx2P" corresponds to an experiment in which the eight sections of the grid were distributed among four Fireflies (two per Firefly) and two processors per Firefly were used (for a total of eight processors). A number of conclusions can be drawn from Figure 2:

- Good speedups are possible in this environment. The SOR program attains a speedup of 25 for the 8Nx4P case – eight Firefly workstations, each contributing four processors to the overall solution.

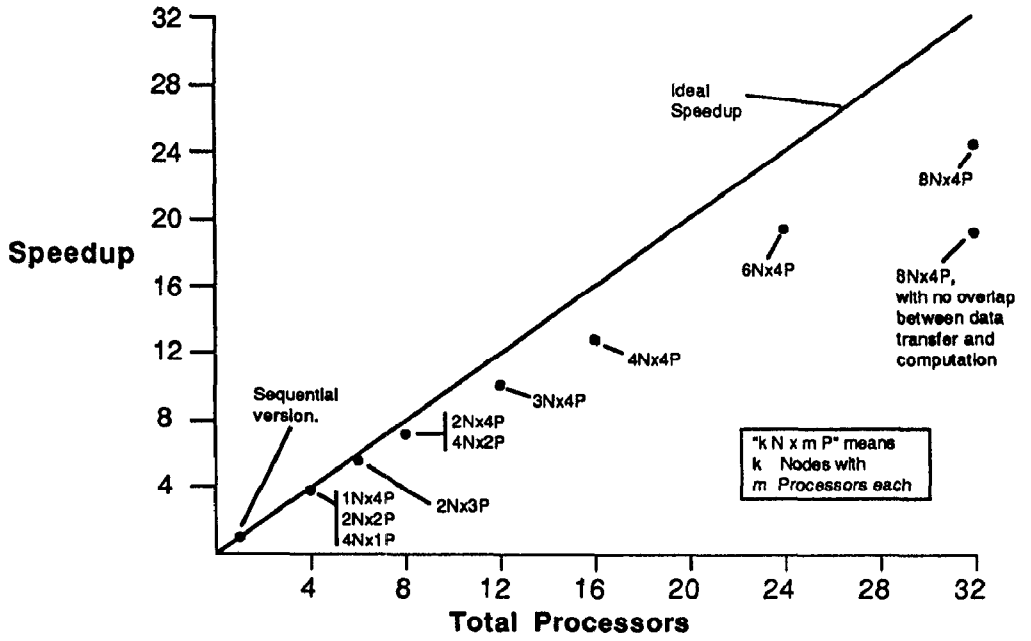

Figure 2: Measured Speedup for Amber Red/Black SOR Implementation

- Significant performance benefit comes from structuring the program so that transfers of edge data are overlapped with computation over the interiors of sections. This is demonstrated by the different performance of the two 8Nx4P cases. This shows the importance of overlapping communication and computation in a loosely-coupled environment.

- The overlapping of communication and computation makes it possible to keep all processors busy doing useful work even while communication is taking place. The performance of this application is not degraded significantly by the cost of remote communication. This is demonstrated by the speedup of the Amber version, which is close to the ideal speedup relative to the sequential version. Also, nearly identical speedups are achieved for all of the experiments involving a total of four processors (1Nx4P, 2Nx2P, 4Nx1P). Similar results were obtained from the experiments with eight processors (2Nx4P, 4Nx2P).

To be fair, the ratio of computation to communication for this program is a function of the grid size. Even if communication is highly efficient, for sufficiently small grids it will dominate computation and limit speedup. For sufficiently large grids computation will dominate and speedup will be good even if communication is relatively inefficient. Figure 3 shows the effect of varying the problem size for the particular configuration of four nodes with four processors each (4Nx4P in Figure 2). The horizontal axis in Figure 3 is the number of points in the grid. The vertical axis gives speedup relative to a sequential version of the program. The point marked "X" corresponds to the 122 by 842 grid used in Figure 2.

We were able to achieve good performance in our Amber SOR program for several reasons. A single network exchange is required to transfer an entire row or column of data between sections, regardless of how data happens to be laid out in the address space. Second, data transfers can be overlapped with computation by running the respective threads in parallel. This reduces the effect of network latency. Third, computation threads within a section can freely divide work among themselves, without danger of causing network activity.

We have not implemented this application under a system with a page-oriented distributed virtual memory, so it is impossible to make exact comparisons with such a system. Certainly a shared memory version under a system such as Ivy would have required less coding effort initially. The performance of the resulting program ultimately depends on how efficiently data can be shared between nodes. The methods for controlling sharing and communication using Amber, with its object-oriented distributed virtual memory, and using a system with a page-oriented distributed virtual memory, are quite different. Using a page-oriented system, the programmer would optimize data reference patterns by laying out data structures and partitioning the work so as to make each node reference different sections of the linear address space. If two nodes write-share the same block of addresses, the virtual memory system will thrash. It may not be obvious from the source code that this can happen. Also, the layout of the data in memory may incur the cost of multiple faults and multiple page transmission latencies to transfer edge data. With Amber the decomposition is addressed explicitly: the programmer has control over what data is transferred and when.

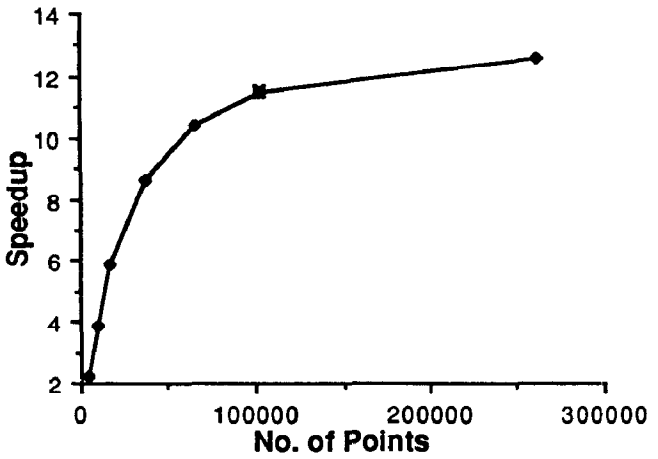

Figure 3: Effect of Varying SOR Problem Size (4Nx4P)

7 Summary

The Amber system permits a loosely-coupled network of multiprocessors to be viewed as an integrated system for executing a parallel application. This underlying hardware architecture is cost-effective for many parallel applications. Processors can be added to a computer system at small marginal cost, but packaging constraints limit the practical size of a single system. Therefore programmers will want to build parallel programs that cross machine boundaries.

With Amber we have shown that the distributed object model is useful for loosely-coupled multiprocessing as well as for distributed programming and distributed operating systems. Amber's object-oriented model strikes a balance between the ease of programming afforded by a page-oriented distributed virtual memory and the performance benefits of explicit management of location. We have achieved a simple and efficient implementation using an existing programming language and an existing operating system. Our application experience thus far indicates that the fundamental goal of Amber – to allow the power of a network of small-scale multiprocessors to be harnessed for a single parallel application – has been achieved.

8 Acknowledgements

Norman Hutchinson and Eric Jul were involved in early discussions of Amber's memory model. Guy Carpenter implemented several pieces of the Amber runtime system. Brian Bershad and Jan Sanislo provided numerous comments and helped with the operating system and hardware of the Firefly. Reid Brown, Tom Anderson, Jeff Bowden, and Ewan Tempero commented on early versions of this paper. Hugh Lauer assisted with final revisions. We would also like to thank the DEC Systems Research Center for providing the Firefly workstations and the Topaz operating system software.

References

[Allchin & McKendry 83] Allchin, J. and McKendry, M. Synchronization and recovery of actions. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 31–44, August 1983.

[Almes et al. 85] Almes, G. T., Black, A. P., Lazowska, E. D., and Noe, J. D. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[Bal & Tanenbaum 88] Bal, H. E. and Tanenbaum, A. S. Distributed programming with shared data. In *Proceedings of the International Conference on Computer Languages*, pages 82–91, October 1988.

[Bershad et al. 88a] Bershad, B. N., Lazowska, E. D., and Levy, H. M. Presto: A system for object-oriented parallel programming. *Software – Practice and Experience*, 18(8), August 1988.

[Bershad et al. 88b] Bershad, B. N., Lazowska, E. D., Levy, H. M., and Wagner, D. An open environment for building parallel programming systems. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming Environments, Applications, and Languages*, July 1988.

[Birrell & Nelson 84] Birrell, A. D. and Nelson, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Black et al. 87] Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[Cox & Fowler 89] Cox, A. L. and Fowler, R. J. The implementation of coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[Fowler 85] Fowler, R. J. *Decentralized Object Finding Using Forwarding Addresses.* PhD dissertation, University of Washington, December 1985. Department of Computer Science Technical Report 85-12-1.

[Herlihy & Wing 87] Herlihy, M. P. and Wing, J. M. Avalon: Language support for reliable distributed systems. In *IEEE Fault-Tolerant Computing Symposium Digest*, July 1987.

[Jones et al. 79] Jones, A. K., Chansler, R. J., Durham, I., Schwans, K., and Vegdahl, S. R. StarOS, a multiprocessor operating system for the support of task forces. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 117–127, December 1979.

[Jul et al. 88] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Li & Hudak 86] Li, K. and Hudak, P. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[Li 86] Li, K. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD dissertation, Yale University, September 1986. YALEU/DCS/RR-492.

[Liskov 88] Liskov, B. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Lucco 87] Lucco, S. E. Parallel programming in a virtual object space. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 26–34, October 1987.

[Ortega & Voigt 85] Ortega, J. and Voigt, R. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, pages 149–240, 1985.

[Ousterhout et al. 80] Ousterhout, J. K., Scelza, D. A., and Sindhu, P. S. Medusa: An experiment in distributed operating system structure. *Communications of the ACM*, 23(2):92–105, February 1980.

[Schantz et al. 86] Schantz, R. E., Thomas, R. H., and Bono, G. The architecture of the Cronus distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 250–259, May 1986.

[Schroeder & Burrows 89] Schroeder, M. D. and Burrows, M. Performance of Firefly RPC. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[Stroustrup 86] Stroustrup, B. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1986.

[Thacker et al. 88] Thacker, C. P., Stewart, L. C., and Satterthwaite, Jr., E. H. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Wulf 74] Wulf, W. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.