An Efficient Implementation Scheme of Concurrent Object-Oriented Languages on Stock Multicomputers

Kenjiro Taura Satoshi Matsuoka
Akinori Yonezawa
Department of Information Science, The University of Tokyo*

Abstract

Several novel techniques for efficient implementation of concurrent object-oriented languages on general purpose, stock multicomputers are presented. These techniques have been developed in implementing our concurrent object-oriented language ABCL on a Fujitsu Laboratory's experimental multicomputer AP1000 consisting of 512 SPARC chips. The proposed intra-node scheduling mechanism reduces the cost of local message passing. The cost of intra-node asynchronous message passing is about 20 SPARC instructions in the best case, including locality checking, dynamic method lookup, and scheduling. The minimum latency of asynchronous inter-node message passing is about 9μs, or about 120 instructions, employing the self-dispatching mechanism independently proposed by Eicken et al. A large scale benchmark which involves 9,000,000 message passings shows 440 times speedup on the 512 nodes system compared to the sequential version of the same algorithm. We rely on simple hardware support for message passing and use no specialized architectural supports for object-oriented computing. Thus, we are able to enjoy the benefits of future progress in standard processor technology. Our result shows that concurrent object-oriented languages can be implemented efficiently on conventional multicomputers.

*Physical mail address: 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan. Phone 03-3812-2111 (overseas +81-3-3812-2111) ex. 4108. E-mail: {tau,matsu,yonezawa}@is.s.u-tokyo.ac.jp

1 Introduction

Although a number of concurrent object-oriented programming languages (concurrent OOPL) have been proposed, relatively little attention has been paid to achieving high performance on *conventional* multicomputers such as CM-5, nCUBE/2, and AP1000[11].

Most work on high performance concurrent OOPLs has focused on combination of elaborate hardware and highly-tuned, specially tailored software [5, 14]. These software architectures (the compiler and the runtime system) exploit special features provided by the hardware in order to achieve the two key factors in high-performance concurrent OOPL implementation—efficient message passing between objects and efficient intra-node multithreading. The characteristics of the hardware features are: (1) Processors and the network are tightly connected—processors can send a packet to the network within a few machine cycles, and dispatching a task upon packet arrival takes only a few cycles as well. (2) The scheduling of threads is automatic and directly supported by hardware—the hardware manages the scheduling queue and the next runnable thread is scheduled automatically upon termination of the current thread. Some concurrent OOPLs on such fine-grain machines achieve impressive performance where the latency of message passing between objects on different nodes are 9μs[14], or about 200 machine cycles[5] for a request-reply cycle of method invocation.

The purpose of this paper is to demonstrate the techniques on conventional multicomputers that achieve comparable performance without the hardware facilities described above. Language implementation effort in this direction has been recently proposed; Culler et al. has proposed key features to efficiently manage fine-grain parallelism in a multicomputer setting in their work on TAM[4] and Active Message[13], using Id[8] as the primary target language. Although our work shares some characteristics with theirs, we propose several novel ideas which could be another essential step toward efficient parallel computing on conventional multicomputers.

To demonstrate this, we have developed a runtime environment for a concurrent OOPL called ABCL/onAP1000, which is based on the computation model ABCM[15]. The runtime environment features low-overhead intra-node scheduling and low-latency remote communication on conventional multicomputers which lack special hardware support such as message send instructions and hardware scheduling queues. The three key software technologies in our implementation of ABCL/onAP1000 are:

Integration of Stack-Based and Queue-Based Scheduling: Standard multiprocessing techniques on a single node employ a scheduling queue but suffers from the high overhead of software queue manipulation. Our scheduling mechanism avoids this overhead in many cases by employing efficient stack-based scheduling as much as possible.

Multiple Virtual Function Table: The *virtual function table* is a well-known technique for dynamic method dispatching in object-oriented languages. We propose techniques that extend its use in concurrent OOPLs with which several runtime checks in concurrent object execution can be avoided.

Hiding Latency of Remote Object Creation: The standard scheme for remote object creation requires inter-node communication, which causes unpredictable latency. Moreover, *split-phase allocation*, which works well on fine-grain machines, is not acceptable because of the high-overhead of context switching on conventional multicomputers. We present a technique which hides remote communication latency and avoids context switching in many cases.

The actual implementation of the language ABCL/onAP1000 was done on Fujitsu Laboratory's experimental multicomputer AP1000, which consists of 512 SPARC chips running at 25MHz, interconnected with a 25MB/s torus network. The preliminary measurements show the minimal cost of an asynchronous method invocation on a local object is 2.3μs, or about 20 instructions, including locality checking, method dispatching and scheduling. Also, the cost of an asynchronous message send between two objects on different nodes is 9μs in the best case. This is within a small factor of fine-grain machine performance[5, 14].

Our current prototype compiler generates C language source code. This decision was based on maintaining portability on various future machines.

2 Language Model

This section briefly describes our language, ABCL/onAP1000, which embodies characteristics common to most concurrent OOPL proposed so far[2]. A more comprehensive description of the language ABCL can be found in [15, 16].

2.1 Concurrent Objects and Message Passing

In our computation/programming model[15], computation is carried out by message transmissions among *concurrent objects*, which are units of concurrency and become active when they accept messages. More than one message transmission may take place in parallel and objects may become active simultaneously.

When an object *receives* a message, the message is placed in its *message queue*, and *accepted* for method invocation one-by-one (asynchronous buffered communication). Messages can contain *mail addresses* of concurrent objects as well as basic values such as numbers and booleans. When two messages are sent from the same sender to the same receiver, they arrive at the receiver in the same order as they were sent (*preservation of transmission order*). A sender can send a message to other concurrent objects only if it knows their mail addresses. Since the mail addresses of concurrent objects can be message arguments, the reference topology of concurrent objects is dynamic.

Each object has its own (autonomous) single thread of control, and its own encapsulated state variables (instance variables). The state of a concurrent object is also characterized by the *mode* of its execution; an object is in *dormant* mode if it has no messages to be processed, and it is in *active* mode if it is executing a method. Our model also facilitates *selective message reception* where an object waits for a certain set of messages within a method. Such objects are said to be in *waiting* mode.

2.2 Actions within A Method

A concurrent object can execute a sequence of the following five kinds of basic actions in its method:

1. *Message sends* to other concurrent objects.

   Messages could be *past type* (namely, "asynchronously send and no-wait", syntactically denoted by `[`*TargetObj* `<=` *Msg*`]`) or *now type* (namely, "asynchronously send and wait for a reply message", syntactically denoted by `[`*TargetObj* `<==` *Msg*`]`). The now-type is similar to procedure call, except that (1) the receiver of the request message may continue execution even after it has returned the reply message to the sender, and (2) the reply messages areo actually sent to another object, called *reply destination object* which resumes the original sender upon the reception of the reply message. The reply destination object may be passed to other objects, thus reply messages are not necessarily sent by the original receiver of the request message.

2. *Creation* of concurrent objects.

3. *Referencing and Updating* of the contents of its state variables.

4. *Waiting* for a specified set of messages (*Selective message reception*), including replies of now-type messages.

5. *Standard operations* (e.g., arithmetic operations) on values that are stored in its state variables and passed around by message transmissions.

2.3 Types and Objects

In this paper, we assume that types of variables and element types of containers (i.e., arrays and lists) are statically determined. Although a part of our proposed scheme is applicable regardless of the underlying type system, the overhead incurred by runtime tag handling in dynamic types may nullify our optimization. Objects are the only entities which could be referred to from remote nodes, while other data types, such as arrays and lists are private data of an object that are only referred to from the object.

2.4 Message Patterns

The syntax of a message is similar to that of Smalltalk, consisting of keywords and corresponding arguments. A message is distinguished from one another by its *pattern*, which is a combination of its keywords and its argument types. Unary messages (messages without arguments) are also supported. At compile time, a unique number is assigned to each message pattern.

2.5 Creation of Objects

Objects can be created dynamically on any node (processor). To provide the programmer with locality control, we provide two primitives, namely *local create* and *remote create*. In remote creation, the system determines where the object is created based on local information. The details are described in Section 5.2. We are currently developing a more sophisticated solution for issues on long-term locality management/load balancing.

3 Issues in Implementations of Concurrent OOPL

The major overhead in concurrent OOPL compared to sequential, single-node counterparts lies in: (1) overhead of intra-node communication/multiprocessing (e.g., scheduling queue manipulation) and (2) high cost of remote communication. Our proposed software architecture overcomes these problems.

A naive implementation of a message reception/method invocation for an object would require (1) allocation of an invocation frame, which holds local variables and message arguments of the method, (2) buffering a message into the frame, and (3) enqueueing the frame into the object message queue, and (4) enqueueing the object into the scheduling queue, if necessary (i.e., if the object is not already in the scheduling queue). The object enqueued in the scheduling queue will eventually get control. The above process is far more costly than the corresponding scheduling process for sequential OOPLs (i.e., method lookup + stack-based function call).

The remote communication overhead involves the start-up cost of message sends and the cost of dispatching appropriate tasks on the receiver node. This is complicated by the fact that the receiver node is not only required to service the messages between objects, but also must provide other services such as remote object creation, garbage collection, load monitoring, object migration, etc. On conventional multicomputers, these tasks are mostly done in software, thus consuming a significant amount of CPU time.

4 Intra-node Software Architecture of ABCL/onAP1000

As mentioned above, the naive method invocation schemes in concurrent OOPL are much more costly than that of sequential OOPLs. The source of inefficiency is heap-based (i.e., not stack-based) frame allocation/deallocation and queue manipulations (i.e., message queue and scheduling queue). In this naive scheme, frame allocation is done from the heap because scheduling of methods are not necessarily LIFO-based, that is, methods may be blocked in order to wait for messages and resumed upon the arrival of the message. The scheduling queue is also required for the same reason. In addition, since the processing of received messages may be postponed

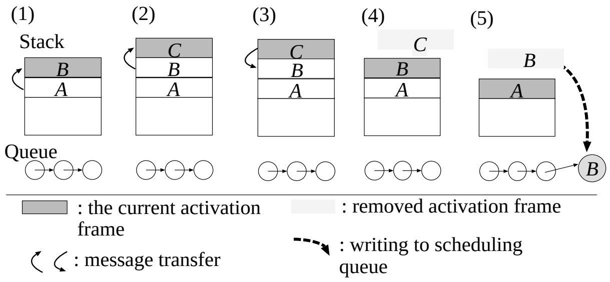

Figure 1: Intra-node scheduling strategy: (1) A sends a message to B. B starts execution immediately. (2) B sends a message to C. C starts execution immediately. (3) C sends the second message to B, and C continues execution because B is already active. (4) After C finished its execution, B executes the rest of the method. (5) When B finishes its method, B enqueues itself in the scheduling queue and will be scheduled later.

depending on the mode of the receiver object (e.g., messages to active objects cannot be processed immediately), each object must have its own message queue to which incoming messages are buffered, in addition to the global (node-wise) scheduling queue.

4.1 Intra-node Scheduling Strategy

Our key observation is that the full scheduling mechanism described above is not necessary in many cases. Instead, more efficient, stack-based scheduling similar to sequential language suffices. More specifically, in our scheduling strategy, if an object has no messages to be processed (i.e., *dormant*), its method is immediately invoked upon message reception without message buffering nor scheduling queue manipulation. In contrast, if it is already running (*active*), the message is buffered and the method is invoked later via a global scheduling queue.

To illustrate our scheduling strategy, consider three objects A, B and C, which execute the following sequence of actions on a single node (Figure 1). Assume that only A is initially active, the others are dormant, and each method is not blocked during its processing.

1. A sends a message to B.
2. B, in the method invoked by A, sends a message to C.
3. Then C, in the method invoked by B, sends a message to B.

In Step 1, B is invoked immediately because B is dormant, and the execution of A is temporarily suspended. Similarly, in Step 2, C starts its execution immediately, suspending the execution of B. On the other hand, in Step 3, since B is now active, the second message to B is buffered in its message queue and C continues execution. After C has finished its method, B resumes the execution of the rest of the method (Step 4 in Figure 1). At this point, there is a choice either to process the next message in the message queue of B, or resume A. We resume A in order to prevent B from monopolizing control; thus, B is enqueued into the global scheduling queue to be scheduled later (Step 5 in Figure 1). By this strategy, A would eventually get control even if B and C were to continue sending messages to each other.

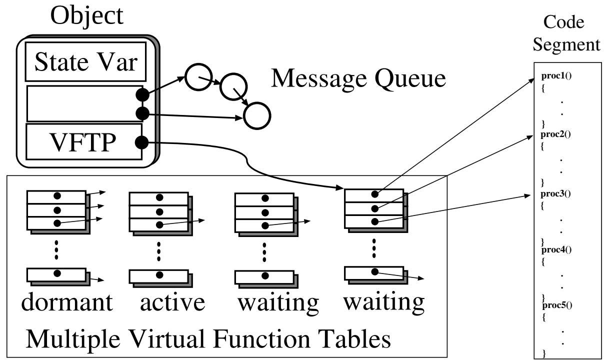

Figure 2: Components of an object.

4.2 Dynamic Method Dispatching via Multiple Virtual Tables

Although our scheduling strategy is now clear, the *mechanism* to implement it efficiently is not trivial. For example, it might seem that a runtime check is required on every intra-node message send in order to determine whether the receiver is dormant or not, making our scheme less attractive. Also, the scheduling mechanism must be *deadlock-free*, i.e., when the running object becomes *blocked* on the stack, other objects must be resumed. This and next sections describe an efficient realization of these strategies.

First, the representation of an object is an extension of that in sequential OOPLs. As in Figure 2, each object is composed of:

- A state variable box, which holds the state variables (i.e., instance variables) of the object. Each state variable is accessed with a fixed offset from the top of the object.
- A message queue, which is a list of received, unprocessed messages. Each item in the message queue is actually a heap-allocated frame which is allocated upon message reception. Each frame keeps its message arguments, local variables and other necessary fields for scheduling purpose.
- A *virtual function table pointer* (*VFTP*), which points to one of the *multiple virtual function tables*. A method of a class is compiled into a C function

and its address is placed in a virtual function table similar to that of sequential OOPLs (e.g., C++[12] and Eiffel[7]).

The key idea in our object representation is an extended use of virtual function tables: each class has *multiple* virtual function tables, each of which roughly corresponds to a mode (dormant, active, and waiting) of an object. When an object is in dormant mode, its VFTP points to the table that contains the method bodies, as is with sequential OOPLs. When the object becomes active, the VFTP is made to point to a virtual function table that holds tiny *queuing procedures* which simply allocates a frame, store the message into the frame and enqueues it to the message queue of the receiver object. Multiple virtual function tables liberate the sender object from the runtime checking of whether or not the receiver object is dormant, by incorporating it into the virtual function table look-up, which is already a necessary cost in OOPLs.

A message send begins with a runtime check to determine whether or not the receiver object is local by looking up the node ID of the object. (The locality check usually takes 3∼4 SPARC instructions.) If the receiver object is local, we simply look-up the virtual function table with the statically-determined index number of the message pattern and call the indexed procedure with the message as arguments. As described above, if the object is in dormant mode, the procedure will be the method body itself, which is executed immediately. Otherwise, if the object is in active mode, the procedure will be a queuing procedure, which stores the message into a message box and returns execution to the sender. Each queuing procedure is customized according to the interface of the corresponding method (i.e., the number of the arguments and the types of each argument).

For now-type messages, the mail address of the reply destination object is passed as well. After the invoked procedure returned, the sender checks its reply destination object to see if the reply has arrived; if the reply has not arrived the object becomes blocked. (The details are described in Section 4.3.) Notice that return from a procedure does not imply the arrival of the reply as with sequential procedural languages; rather, the reply message is explicitly sent as normal message passing to the reply destination object.

Each object checks its message queue at the completion of a method, and if there are any pending messages, enqueues itself in the global scheduling queue. Objects in the scheduling queue are invoked when the scheduling stack becomes empty.[1]

Selective message reception, described in Section 2, is also naturally realized by constructing a virtual function table for each selective message reception in the methods. The entries for acceptable patterns contain procedures which restore the context of the object, whereas the other entries contain queuing procedures. (This reflects the semantics of ABCL where unacceptable messages in selective message reception are buffered into the message queue. Other semantics, e.g., discarding unacceptable messages, can also be facilitated easily.)

In addition, we utilize the VFTP for two other purposes. Firstly, an object initializes its state variables lazily when it receives a message for the first time. A naive implementation would require an additional flag indicating initialization, which would result in checking the flag on every message send. We avoid this by building a virtual function table which holds pointers to an initialization routine which calls the method body after the state variable initialization. Secondly, using the multiple virtual function tables solves a problem in fast remote creation of objects. The problem is that, a message to an object created on a remote node may arrive before the virtual function of the object is properly initialized. The details of the problem and its solution is described in Section 5.2.

4.3 Combining the Stack and Scheduling Queue

Upon invoking a method of a dormant object, a frame is allocated on the *stack* thereby achieving fast frame allocation/deallocation. When such an invocation is blocked in the middle of the thread for the first time, it allocates another frame lazily on the heap, and saves its context (i.e., local variables) to the frame which will be kept until termination of the method. On the other hand, upon reception of a message by an active object, a frame is allocated on heap and the message is immediately stored there.

The object that has its activation frame on the top of the stack is physically executing. When the running object becomes blocked as a result of a selective message reception or waits for a reply of a now-type message send, it pops off its own stack frame and saves its context (i.e., instruction pointer and local variables) into the heap-allocated frame, thereby resuming other objects. At the same time, it also

[1] Our scheduling mechanism also allows invocation of objects in the scheduling queue even if the stack is not empty, but we do not employ this for the purpose of fair scheduling between objects. (See Step 5 in Figure 1)

switches the VFTP to point to the virtual function table whose entries of awaited messages have the context restoration routine and others have the queueing procedures. (More precisely speaking, in the case of now-type messages, since the reply is sent to the reply destination object rather than to the sender object, all the entries of the VFTP of the sender are the queueing procedure; the reply destination object actually resumes the sender on the arrival of the reply message.) When one of the awaited messages later arrives, the context is restored and the object becomes active again. We stress that object is not blocked as long as it finds an awaited message when it first checks its message queue (or, the reply destination object for a now-type message). In particular, for intra-node now-type message passing, it is usually the case that the reply will have already arrived when the sender checks the reply destination object because we employ stack-based scheduling; in such cases, stack unwinding does not occur.

Let us examine an example case where stack unwinding occurs and how our scheme schedules runnable objects (Figure 3). Consider object S sends now type message m to *active* object R, whose current activation frame is buried in the lower part of the stack. When sending m, S invokes the queueing procedure of m because R is active. After the queuing procedure returns, S checks the reply destination object to see if a reply has already returned. Since there are naturally no replies, S saves its context into a heap-allocated frame and resumes the object which activated S (i.e., Object O in the Figure 3). When control reaches R again and R finishes its current method, it checks its message queue to find that there are some messages, thus enqueueing itself to the scheduling queue (Step 3 in Figure 3). Eventually R processes m and sends a reply message to S, resuming the saved context of S.

Although most part of our scheduling is done via the stack, we also employ a global (node-wise) scheduling queue as well. The purposes of the scheduling queue are twofold:

1. To invoke buffered messages in the FIFO scheduling—Messages that were once buffered in a message queue are scheduled through the scheduling queue, as we have seen in Section 4.1.
2. To preempt objects—Preemption occurs when an object falls into a long internal loop or a deep recursion. When preemption occurs, the object stores its context into a heap-allocated frame and enqueues itself into the scheduling queue.

Each item of the queue consists of a pointer to the

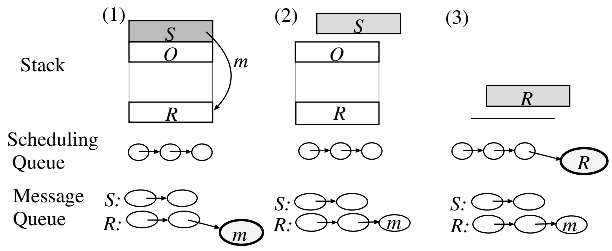

Figure 3: An example of stack unwinding. (1) S sends now type message m to R and m is enqueued. (2) S checks the reply destination object to find that no reply has arrived and saves its context into a heap-allocated frame. (3) When R gets control, it enqueues itself into the scheduling queue at the end of the method. m is eventually scheduled and the reply reaches S.

object which will be scheduled and a continuation address from which the object will restart execution. The system dequeues an item in the scheduling queue simply by transferring control to the continuation address contained in the item—the instructions starting from the continuation address perform the actual context restoration and activation of the scheduled object.

5 Inter-node Software Architecture of ABCL/onAP1000

We next describe the remote servicing aspects of our software architecture, such as message passing between objects on different nodes, and object creation on a remote node. Our software architecture assumes that the hardware (or message passing libraries) provides an interface to send and receive messages asynchronously. It is strongly desirable to be able to send/receive messages without OS kernel intervention. Supports for arbitrary length packets and preservation of message transmission order do simplify the implementation, although it is not a necessity. Message arrival may be notified by polling as in CM-5 or AP1000, or by interrupt as in nCUBE/2 or iPSC/2. Most multicomputers (including AP1000, our current target machine) support these facilities.

5.1 Efficient Remote Task Activation

As for underlying mechanisms of remote servicing aspects, we employed an "Active Message-like mechanism[13]",[2] which is a low overhead message dispatching mechanism for conventional multicomputers. The idea is that each kind of message attaches its own 'self-dispatching' message handler which is invoked immediately after the delivery of the message By giving a customized message handler for each kind

[2]We developed a technique similar to Active Message early 1992, and later found it in [13].

of remote messages, low overhead remote task dispatching is achieved.

Our compiler generates these specialized message handlers for many purposes such as remote message passing, remote object creation etc. Message handlers are classified into the following four categories:

1. Normal message transmission between objects.
2. Request for remote object creation.
3. Reply to remote memory allocation request.
4. Other services (load balancing, global garbage collection, etc.).

The handlers for Category 1 extract the pointer to the receiver object and arguments and schedule the receiver object in the manner described in Section 4.2. The compiler generates a specialized message handler for each message pattern (i.e., the number of arguments and the type of each message). Since the types of the arguments are statically determined, tags are no longer necessary and the arguments are processed efficiently by the specialized handler.

Categories 2 and 3 are used for remote object creation. The handlers for Category 2 create an object of a particular class at the address specified by the requester. The message handler returns the address of another memory chunk to the requester for its future use. The message handlers for Category 3 are provided as the handler address of this reply message. Our compiler generates a single (specialized) message handler of Category 2 for each class and one message handler of Category 3 for each chunk size. The details are described in the next section.

5.2 Remote Object Creation

In our software architecture a mail address of an object is uniformly represented as a pair ⟨ *processor number, a (real) pointer* ⟩. We employed this representation for maximum performance in local object access and to avoid the overhead of the export table management. One restriction is that in general it would prohibit the use of a simple copying/compacting garbage collector, as objects cannot be moved freely. We are now developing an algorithm whereby objects that are only referred to locally can be freely copied.

Under our representation scheme of mail addresses, object creation on a remote node requires a memory allocation on the target node in order to generate a remote mail address. Since the latency of remote communication is unpredictable, it is not acceptable to wait until a pointer to the allocated memory (chunk) is returned. *Split-phase allocation* is preferable on fine-grain machines where context switching to another thread effectively hide the latency[1]. On conventional multicomputers, however, split-phase allocation is undesirable because of high context-switching overhead. Rather, a scheme whereby the object can continue execution even in the presence of remote allocation request is preferable.

For this purpose, we employ a *prefetch scheme* where each node manages predelivered "stocks" of address of memory chunks on remote nodes, and the address for remote object allocation is obtained locally from the stock. Only when the stock is empty does context switching on remote object creation occur. The requested node later replies another chunk to replenish the stock. The stock is unlikely to become empty except when unusually frequent remote creations occur.

A typical sequence of remote object creation is as follows:

1. The requester node obtains a unique mail address locally from the stock.
2. A creation request message is sent to the node of the mail address.
3. The target node performs class-specific initialization (e.g., initialization of virtual function table) of the created object upon receipt of the creation message (message handler for Category 2).
4. The target node allocates a replacement chunk and returns its address to the requester node.
5. The requester replenishes the stock upon receipt of the address (message handler for Category 3).

An object on the requesting node can obtain the mail address of the created object in Step 1, rather than in Step 3, thereby effectively hiding the remote communication latency. Furthermore, this scheme avoids context switching (which, in contrast, is necessary for split-phase allocation) as long as some chunks remain in the stock.

There is one problem however: After the new mail address is locally obtained in Step 1, the request message sent in Step 2 must precede any messages to the object. Under our scheme, however, standard messages between objects could reach the uninitialized object when the obtained mail address is passed to other objects before the initialization message has reached the target node (Figure 4).

We again utilize the VFTP to solve this problem without adding any overhead to each message send. Each chunk is pre-initialized in the following way when its address is sent in Step 4: The message queue is empty and the VFTP is *temporarily* initialized to be the generic *fault function table*, whose entries are

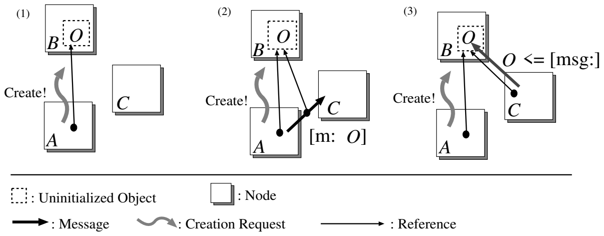

Figure 4: Initialization problem on remote object creation: (1) Sending creation request to B, an object on node A obtains the mail address of an uninitialized object O on node B from the stock. (2) The object on A sends a message which contains the mail address of O to some object on another node C. (3) It sends a message to O. This message may reach O before the actual initialization of O by the creation request.

	Time (μs)
Intra-node Message (to Dormant)	2.3
Intra-node Message (to Active)	9.6
Intra-node Creation	2.1
Latency of Inter-node Message	8.9

Table 1: Costs of basic operations.

entirely queuing procedures. This forces all messages to uninitialized objects to be enqueued. When the initialization message eventually arrives, the virtual function table is initialized to the proper one for the class of the object. In addition, the message queue of the object is checked for pending messages, and the first message is extracted and processed if it exists. This scheme works because the queuing procedures are generic for all objects, independent of their classes.

6 Preliminary Performance Measurement of ABCL/onAP1000

6.1 Efficiency of Basic Operations

Table 1 shows the cost of basic operations such as message passing or object creation. Our compiler generates C language code, which is in turn compiled with optimization level "-O4". Throughout all the measurements, Sun-OS 4.1.1 `cc` was used.

An intra-node past-type message send to a dormant object takes 2.3μs. This was measured by repeatedly invoking a null method with no arguments. Table 2 is the breakdown of this process. Further compile-time optimizations are possible under the following conditions: (1) Locality check can be eliminated for objects guaranteed to be local; for example, message passings which follow local object creation. (2) Switching of the VFTP is not necessary if the method does not send messages to other objects and

	Instructions
Check Locality	3
Lookup and Call	5
Switch VFTP to Active Mode	3
Execution of Method Body	–
Check Message Queue	3
Switch VFTP to Dormant Mode	3
Polling of Remote Message	5
Adjusting Stack Pointer and Return	3
Total	25

Table 2: Breakdown of intra-node message to dormant object.

is never blocked. (i.e., if the method is ensured to finish its execution without running other objects.) (3) Checking the message queue is not necessary if the object is not history sensitive. (4) Polling of remote message arrival is not always necessary, especially for small methods—we merely need to guarantee periodical polling of remote messages. Altogether, the overhead of an intra-node message to dormant objects varies from 8 (in this case, the overhead is truly comparable with virtual function call in C++) to 25 instructions.

The cost of an intra-node message to an active mode object includes: (1) a frame allocation, (2) storing the message into the frame, (3) enqueueing the frame into the message queue of the object, (4) enqueueing the object to the global scheduling queue (if the object is not already in the scheduling queue), and (5) re-scheduling of the object from the scheduling queue. The total time is approximately 10μs, which is over 4 times (or even more if we adopt the above optimizations) that of the dormant case. Our scheduling mechanism specially optimizes dormant case by stack-based scheduling, thereby reducing the total intra-node scheduling overhead.

Minimum inter-node latency, 8.9μs, is obtained by repeatedly transmitting one word past-type messages between two objects. There are no objects other than these two objects in the system and both are dormant upon message reception. This shows that conventional multicomputers with appropriate software technologies have much higher inter-node message passing capability than previously believed. In comparison to fine-grain architecture (Table 3), send and reply latency is approximately 18μs, or 450 cycles, which is only about twice of [5] or about 4 times of [14] when normalized to the same clock speed. The instruction counts includes message setup (in the script of the sender), polling, extracting of the message (in the message handler), system message buffer management and script invocation. The sender node

	Instruction Counts	Real Time (μs)	Cycles	Clock Rate (MHz)
ABCL/onAP1000	160	17.8	450	25
ABCL/onEM4[14]	100	9	110	12.5
CST (on J-Machine)[5]	110	4	220	50

Table 3: Comparison of send/reply latency.

	$N = 8$	$N = 13$
# of Solutions	92	73,712
# of Objects Creation	2,056	4,636,210
# of Messages	4,104	9,349,765
Total Memory Used (KB)	130	549,463
Elapsed Time on SS1+	84 ms	461,955 ms

Table 4: The scale of the N-queen program ($N = 8, 13$).

takes about 20 instructions to setup and send a message which is a total of 4 words including routing information, the mail address of the receiver object and the message argument. The receiver node consumes a total of about 50 instructions for polling, extracting the message and management of the system message buffer. The script invocation cost is the same as intra-node message send to dormant objects (approximately 10 instructions). The remaining portion of the latency is due to hardware, which is roughly 1.5μs each way. By (1) reducing this hardware latency and (2) more appropriate interface to network, our speed further approaches that of implementations on fine-grain machines. Furthermore, such implementations on conventional multicomputers can directly benefit from advances in single-processor technology.

6.2 Benchmark Statistics

To evaluate the proposed techniques on real applications, we measured the performance of the N-queen exhaustive search algorithm varying N. The small program ($N = 8$) is the one which terminates within 100 milliseconds on single CPU SPARC station 1+, and the large program ($N = 13$) involves more than 4,500,000 object creations and $9,000,000$ message passings. Table 4 shows the scale of the algorithm where $N = 8$ and 13. Table 4 also shows the elapsed time of the sequential version of basically the same algorithm running on a single SPARC station 1+, which has the same CPU as the node processor of AP1000. The sequential version is written in C++ and compiled by g++ with optimize option.

Figure 5 shows the speedup of the N-queen program relative to the sequential version. Since the sequential version of the algorithm uses run-time stack for the depth first search, it requires no heap memory and there is no need for termination detection, while our parallel version uses heap extensively for parallel

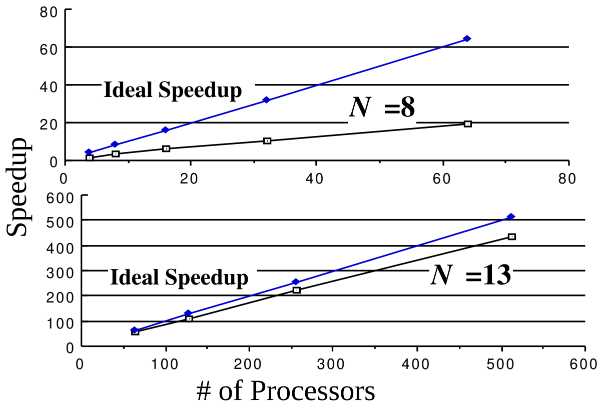

Figure 5: Speedup for N-queen problem ($N = 8, 13$).

search and acknowledgement message trace back the search tree for the termination detection. Nevertheless, approximately 20 times speedup is obtained for $N = 8$ on a 64 processors, and 440 times for $N = 13$ on 512 processors (approximately 85% utilization).

6.3 Effect of Stack-Based Scheduling

To demonstrate the effect of stack-based scheduling, we compare our performance to a more naive scheduling mechanism. The naive scheduling mechanism always buffers a message in the message queue of the receiver object and the object is scheduled through the scheduling queue.

Figure 6 shows the performance improvements in the N queen programs as N varies. In these programs, approximately 75% of local messages are sent to dormant mode objects. In general, we have observed approximately 30% speedup.

7 Related Work

Our work shares the goals of all attempts to make fine-grain computing feasible, not only for object-oriented concurrent languages but for other languages. Amongst them, our work is similar to Threaded Abstract Machine (TAM)[4] and Active Message[13] in aiming for high performance without elaborate hardware supports.

TAM is a multithreaded abstract machine whose instruction set, TL0, is intended to be mapped onto conventional instruction sets, so that TL0 can serve as an intermediate language on conventional multi-

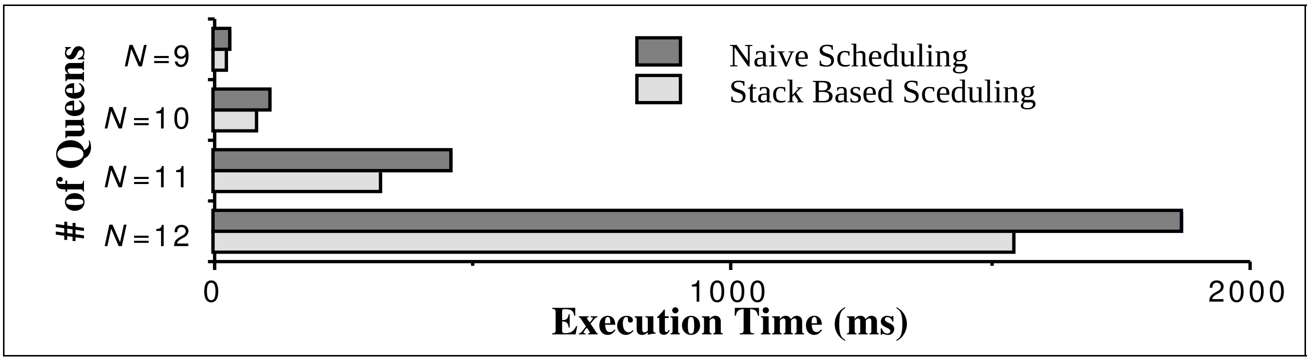

Figure 6: Effect of the stack scheduling.

computers for high-level parallel languages such as Id[8] and Multilisp[9]. The execution model of TAM is an extension of the hybrid dataflow model[6]: In addition to "threads" (instruction sequences which are exclusively executed), TAM recognizes a collection of logically related threads called the *code-block*. Under the TAM execution model, runnable threads within a code-block are ensured to be scheduled consecutively, thereby allowing inter-thread register allocation and inexpensive synchronization of threads. This unit of co-scheduling is called "quanta" in TAM.

8 Discussion

8.1 Stack-based Scheduling

Although TAM has shown an efficient dataflow synchronization and communication between threads *within* a code-block, its *code-block invocation* or *inter-code-block* communication (e.g., passing function arguments) mechanism has room for further improvement. That is, under the TAM execution mechanism, a code-block invocation (e.g., function call) always require frame allocation from heap, message buffering, and scheduling queue manipulation, while our implementation uses a stack as much as possible.

Although situations are different for our language and TAM, which is intended to support lenient[10] semantics where blocking in the middle of a function is more likely, we believe our stack-based scheduling mechanism is better in practical settings where small functions (or methods), which are not typically blocked, are extensively used.

The advantages of our stack-based scheduling are: (1) frames are cheaply allocated/deallocated as long as blocking does not occur, (2) messages are processed without buffering, (3) it is more likely that the result of a function (or method) has already arrived when the callee resumes the caller, because we run the callee (or the receiver object) first, and then resuming the caller (or sender object).

Of course, we do not claim that our stack-based scheduling is not unconditionally advantageous. It requires some registers to be saved before a function call, which is unnecessary under a TAM execution where a caller is never invoked until a callee is blocked. Nevertheless, the amount of registers to be saved will be typically small and the cost will be smaller than the cost of argument buffering and scheduling queue manipulation which is required in the TAM mechanism.

8.2 Method Inlining

Our implementation scheme relies on the fact that objects are always accessed through methods which are obtained from the virtual function table: The readers might point out that this would prohibit further optimization via inlining of method calls which is used extensively in sequential object-oriented language SELF[3] to achieve good performance. However, inlining is possible in the following way, as long as the class of a receiver object is known at compile time: let the class of the receiver be C, the virtual function table of class C for dormant mode be C_dormant_vft, and the method which should be invoked be C_method. The inlined code of the method call would be as follows:

```
if(receiver_node_id == my_cell_id){
    if(receiver_obj->vftp == C_dormant_vftp){
        inlined code of C\_method;
    } else {
        enqueue the message;
    }
} else {
    send the message to the receiver_node_id;
}.
```

This code still has additional checks for locality and the mode of the object. The full benefit of inlining is obtained by inferring such information and eliminating the checking code, but such inference is far more difficult than class inference in sequential OOPL settings, and is a subject of our future work.

9 Conclusion

We have proposed a software architecture for concurrent OOPLs on conventional multicomputers that can compete with implementations on special-purpose, fine-grain architectures. Our primary contribution is a novel intra-node scheduling mechanism which reduces the average cost of intra-node method invocation. Several novel techniques for avoiding many runtime checks have also been proposed. The preliminary measurements have shown that our performance compares favorably with the implementations on fine-grain machines. Restrictions of these mechanisms/techniques are small and they are widely applicable for various (even non-object-oriented) concurrent language implementations.

Acknowledgements

Our thought was improved through the discussion with Rishiyur S. Nikhil, who gave us many beneficial and thoughtful comments that go into details of our proposal. We thank to David E. Culler and Thorsten von Eicken for giving us many comments on our paper. The discussions with Bill Dally and Waldemar Horwat at our mutual visits were very fruitful. We would like to appreciate the offer of Fujitsu Laboratory in Kawasaki, Japan, which has provided us with the opportunity to use AP1000. Our special thanks go to Takeshi Horie for his helpful technical advices and access to the library sources of AP1000. We thank the members of the ABCL project group, especially Jeff MaAffer who substantially improved the presentation of our paper, and Masahiro Yasugi for daily technical discussion.

References

[1] Arvind and Robert A. Iannucci. Two fundamental issues in multiprocessing. In *4th International DFVLR Seminar on Foundations of Engineering Sciences*, volume 295 of *Lecture Notes in Computer Science*, pages 61–88, 1987.

[2] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[3] Craig Chambers and David Ungar. Making pure object-oriented language practical. In *OOPSLA*, pages 1–15, 1991.

[4] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *ASPLOS*, pages 166–175, 1991.

[5] Waldemar Horwat. Concurrent Smalltalk on the message-driven processor. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1989.

[6] Roberb A. Iannucci. Toward a dataflow/von neuman hybrid architecture. In *ISCA*, pages 131–140, 1988.

[7] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[8] R. S. Nikhil. Id (version 88.0) reference manual. Technical Report 284, MIT Laboratry for Computer Science, March 1988.

[9] Jr. Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *acm Transactions on Programming Languages and Systems*, 7(4):501–538, April 1985.

[10] Klaus Erik Schauser, David E. Culler, and Thorsten von Eicken. Compiler-controlled multithreading for lenient parallel languages. In *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 50–72, 1991.

[11] Toshiyuki Shimizu, Takeshi Horie, and Hiroaki Ishihata. Low-latency message communication support for the ap1000. In *ISCA*, pages 288–297, 1992.

[12] Bjarne Stroustrup. *The C++ Programming Language, second edition.* Addison Wasley, 1991.

[13] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *ISCA*, pages 256–266, 1992.

[14] Masahiro Yasugi, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/onEM4: A new software/hardware architecture for object-oriented concurrent computing on an extended dataflow supercomputer. In *ICS*, pages 93–103, 1992.

[15] Akinori Yonezawa. *ABCL: An Object-Oriented Concurrent System — Theory, Language, Programming, Implementation and Application.* The MIT Press, 1990.

[16] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA '86 Conference Proceedings*, pages 258–268, 1986.