The ALPS Kernel

Prasad R. Vishnubhotla, Manas Mandal, John R. Mudd,
Andreas Mitschele-Thiel, Mahendra Ramachandran,
Chandrasekhar Gollamudi, Kalluri Eswar, Yow-Wei Yao,
Anita Kulshreshtha, Davender Babbar and Chien-Chiao Wu

Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue
Columbus, Ohio 43210-1277
email: alps@cis.ohio-state.edu

August 1989

The ALPS project is supported by the National Science Foundation under grant CCR-8810388.

Abstract

ALPS is a parallel programming environment with emphasis on the portability of application programs across parallel machines while ensuring their efficient execution. The programmer need not be concerned whether the machine is a shared memory multiprocessor or a processor network, or about the interconnection topology of the machine. To achieve both portability and efficiency, we use a new programming paradigm. Central to the ALPS paradigm are two concepts: *multiplexer channels* and *managers*. A multiplexer channel is a many-to-many communication medium shared by a group of processes. When a consumer wishes to receive a data item on a channel, it can nondeterministically be given any one of the values on the channel. Multiplexer channels are typically used as carriers of work-to-be-done, and the implementation can exploit the *data nondeterminism* to provide automatic work-load balancing by dynamically partitioning the messages on the channel among the consumers. ALPS supports object-oriented programming wherein objects can be shared by concurrent processes. The manager mechanism provides concurrency within a shared object. If there is a manager process for an object, each call to an object's procedure is intercepted by the manager. The manager can decide which of these calls should be allowed to progress concurrently and which of them should be queued up until a future time, thus implementing any necessary synchronization.

This document describes the ALPS kernel which provides a small set of primitives that support the ALPS paradigm and are callable from C programs. Initial versions of the kernel have been implemented on an Encore Multimax, a BBN Butterfly, a Transputer network, an Intel iPSC and a network of Sun workstations.

Key Phrases: Parallel Processing, Programming Paradigms, Operating Systems, Programming Environments, Machine Independence, Cooperating Processes, Object-Oriented Programming.

Contents

1 Introduction

ALPS is a parallel programming environment with emphasis on the portability of application programs across parallel machines while ensuring their efficient execution. The programmer need not be concerned whether the machine is a shared memory multiprocessor or a processor network, or about the interconnection topology of the machine. To achieve both portability and efficiency, we use a new programming paradigm which we proposed in [15]. This document describes the ALPS kernel which provides a small set of primitives that support the ALPS paradigm and are callable from C programs. Initial versions of the kernel have been implemented on an Encore Multimax, a BBN Butterfly, a Transputer network, an Intel iPSC and a network of Sun workstations.

Central to the ALPS paradigm are two concepts: *multiplexer channels* [16] and *managers* [17]. Multiplexer channels facilitate parallel programming using cooperating processes. Unlike a point-to-point channel, a multiplexer channel is shared among several processes that can send and receive messages on the channel. When a process wishes to receive a message on a channel, it may be given any one of the messages on the channel. This *data nondeterminism* can be exploited by the implementation by dynamically partitioning the messages on a multiplexer channel among the consumers. Multiplexer channels are typically used as carriers of work-to-be-done and the dynamic partitioning provides automatic work-load balancing. The messages on a multiplexer channel may contain either simple scalar values or addresses of *shared objects* that are stored at fixed locations.

ALPS supports object-oriented programming wherein objects can be shared by concurrent processes. An object type can be created by specifying the data memory requirement for the object, the procedures that define the operations on the object, and an optional manager procedure. When an object of a given object type is created, a new process is implicitly created and is made to execute the manager procedure. We refer to this process as the manager process for the object. Calls to the procedures of an object are implemented as *remote procedure calls* (RPC). If there is no manager for an object, a newly created process executes the called procedure right away. If there is a manager for the object, then each call to an object's procedure is intercepted by the manager. The manager can decide which of these calls should be allowed to progress concurrently and which of them should be queued up until a future time. Thus the manager mechanism provides concurrency within an object and implements the necessary synchronization. The manager can be programmed to optimize resource utilization by combining some of the user requests and also to monitor the use of the object.

1.1 ALPS in Perspective

Several approaches to parallel processing have been proposed and are being practiced. Restructuring compilers [11], which transform sequential code to parallel code, have been very popular for SIMD machines for several years. This approach is also beginning to emerge as a useful method for shared memory multiprocessors [12]. However, for use on MIMD machines, a more common approach has been to provide parallel programming support either through a programming language (e.g., CSP [9], Occam [10], Ada [2], SR [3], Argus [13]) or through an operating system interface (e.g., Mach [1], V-kernel [6], Sprite [14], Presto [5]) where the programmer expresses parallelism explicitly in the program rather than the parallelism being implicitly generated by a compiler. A problem with this approach is that it is not always possible to achieve both portability and efficiency for the programs because the mapping of the program components onto the machine's architecture tends to be a significant portion of the program development.

Dataflow [7,4] and Linda [8] are two other approaches to parallel programming. ALPS shares certain aspects with both of these approaches. It gains from their advantages and avoids their trouble spots. ALPS uses nondeterministic dataflow in the sense that a data item can flow to any one of a set of consumer processes. Such data nondeterminism, which is also inherent in the shared tuple spaces of Linda, allows the implementation to do efficient scheduling by replicating the worker processes and by load balancing the data among the worker processes. ALPS does not use pattern matching as Linda does. Naming a multiplexer channel is the only way to locate a data item and the receiving process may be given any one of the items on the channel. ALPS supports atomic receive operations which allow a consumer to receive two values from a single channel or from two different channels. These primitives have the flavor of wait-matching in data flow. Such atomicity is important in view of the data nondeterminism that is inherent to multiplexer channels. ALPS uses a distributed termination rule to allow consumers waiting on receive primitives to terminate smoothly. The object oriented programming support in ALPS not only makes it more general purpose but is also essential to develop some applications due to the nondeterministic nature of data flow.

2 Definitions and Terminology

Procedures. Unlike C functions which can be called without the kernel noticing it, procedures in ALPS are special entities which can be executed only with the kernel support. A procedure is made using the MakeProc primitive and supplying a C function which forms the body of the procedure. One way to execute a procedure is to invoke it using the Call primitive. This may be required either to execute code that is on a remote node (or in a different address space on the same node) or to execute code that has internal parallelism. A procedure call in ALPS is similar to the conventional sequential procedure call. However the implementation creates a new process to execute the called procedure. The calling process is delayed until the called procedure terminates and returns the results. Another way to execute a procedure in ALPS is to use it in the creation of parallel processes as described below.

I/O and Worker Processes. When an ALPS procedure is called, it could introduce internal parallelism by creating *i/o processes* and *worker processes* that execute in parallel. These i/o and worker processes communicate with each other only using shared multiplexer channels. Worker processes are more privileged than i/o processes when it comes to receiving messages on a channel. An i/o process is given a message on a channel only after all worker processes that have access to that channel have terminated. Like their names suggest, worker processes are meant to do the main work in the computation, and i/o processes are meant to interact with the outside environment. Two primitives MakeIO and MakeWorker are provided in order to create an i/o process or a worker process respectively. Either primitive supplies a procedure that should be executed by the process and also supplies a list of multiplexer channels that can be used by the process. The MakeIO primitive creates a single instance of the i/o process on the same node where it is executed. The MakeWorker primitive creates multiple instances of the worker process and executes them on the various nodes of the parallel machine as decided by the kernel. An optional replication count can be supplied to the MakeWorker primitive to indicate the exact number of instances of the worker process required overall on the parallel machine. Otherwise, the kernel will create an arbitrary number of instances of the worker process (at least one) and distribute them across the nodes. In either case, the programmer has no control in deciding which worker process executes on which node(s). A procedure that creates the i/o and worker processes is typically programmed to receive the final results from one of the i/o processes on a multiplexer channel and to return them as the results of the procedure call.

Creator Process. When a procedure is called using the Call primitive, the process that is created to execute the procedure is treated as an i/o process. This process is also given a special status called the *creator*. Only a creator process can create channels, i/o processes and worker processes; ordinary i/o processes and worker processes cannot. In other words, a callable procedure can execute the primitives MakeChan, MakeIO and MakeWorker but a procedure that is meant to be executed as an ordinary i/o process or as a worker process should not use these primitives. This rule is enforced not only to simplify implementation but also because it is felt that adherence to this structure results in clean programs. Notice that hierarchical parallelism is still possible because an i/o or worker process can call an ALPS procedure which in turn can create i/o and worker processes. The creator status is also given to the first user process, all manager processes, and all processes created using the Start primitive.

Multiplexer Channels. A multiplexer channel is a many-to-many communication medium which resembles a set-like structure and contains an unordered collection of messages. It provides communication between a group of producers and a group of consumers where both the producers and the consumers are anonymous. When a producer sends a message, it is deposited on the channel. When a consumer wishes to receive a data item on a channel, it can nondeterministically be given any one of the values on the channel.

A multiplexer channel constitutes a distributed data structure because the messages on a multiplexer channel can be arbitrarily partitioned and distributed by the kernel for implementation on distributed memory machines. The send and receive primitives executing on each processor can be implemented by referring only to the local memory of the processor most of the time. This directly contributes to efficient execution of programs.

Channel Access Rights. To implement process termination efficiently, the system should know for each channel and for each process that can use the channel whether the process is a sender, receiver or both sender and receiver for that channel. When an i/o or worker process is created, it is given a list of channels, and for each channel, it is also given its *access right* which is one of SENDER, RECEIVER, or SENDER_RECEIVER. Processes that have SENDER or SENDER_RECEIVER access to a channel are called the *potential senders* for the channel. Processes that have RECEIVER or SENDER_RECEIVER access to a channel are called the *potential receivers* for the channel. The creator of a channel is, by definition, a RECEIVER for the channel; *i.e.*, it cannot send on the channel.

Sending and Receiving Messages. Messages are sent on a multiplexer channel using the Send primitive. Only the processes with a SENDER or SENDER_RECEIVER access right to the channel can send messages on the channel. The Send operation is non-blocking in the sense that the sending process simply delivers the message to the underlying system and continues with its own execution. The implementation is expected to support unlimited buffering, possibly by moving the overflowing messages either to other processors or to a disk. Three forms of receive primitives are provided, namely, Receive, Receive21 and Receive12. Only processes with a RECEIVER or SENDER_RECEIVER access right to a channel can receive messages on the channel. The Receive primitive receives a single message from a channel. The Receive21 primitive receives two messages atomically from a single channel. The Receive12 primitive receives two messages atomically, one message on each of a pair of channels. In all the three cases, the receiving process may be given any one (or two) of the messages on the channel. Notice that the "21" in Receive21 stands for "2 messages on 1 channel" and the "12" in Receive12 stands for "1 message on each of 2 channels."

Dead Channel. A channel is defined to be *dead* if it is empty and all its potential senders have either terminated or are waiting to receive messages on the same channel using a Receive, Receive21 or Receive12.

Almost Dead Channel. A channel is defined to be *almost dead* if it has a single message in it and all its potential senders have either terminated or are waiting to receive messages on the same channel using a Receive21.

Distributed Termination. The primitives Receive, Receive21 and Receive12 are defined as value returning functions that return status information separately from any messages that may be returned as result parameters. The status can be either SUCCESS or ABORT. The value SUCCESS is returned when the receive primitive succeeds in receiving the desired messages. The value ABORT is returned when the system recognizes that the desired messages cannot be received now or at any time in the future; this can happen when the channel becomes either dead or almost dead. This information can be used to program smooth termination of the processes. SUCCESS and ABORT have the values 1 and 0 respectively.

Freezing a Channel. A channel is *frozen* to inform the system that no new i/o or worker processes will be given access to the channel. A channel cannot be considered dead or almost dead unless the channel is already frozen.

Value and Result Parameters. Parameters appear in the kernel primitives either as value parameters or as result parameters. Reference parameters are not supported. All primitives allow only one value parameter and it is up to the user to pack all his values together into one single value parameter before they are sent. Similarly, only one result parameter is allowed, and the user should have this parameter ready to receive all packed results as they are returned. As will be described in Section 3, a predefined structure *param* is provided with two fields *size* and *addr*. When a param variable p is passed as a value parameter, p.size number of bytes from the address p.addr are sent. Here p.size should have a value of zero or more. When a param variable p is supplied as a result parameter, p.size should specify the number of bytes available starting from p.addr for the result to be copied. If the result is more than size bytes long, it is truncated and only the first size number of bytes are copied. If the result is less than size bytes long, p.size is altered to give the actual size of the result. A macro *MakeParam(p, s, a)* is provided. Here *p* is the address of a param variable. The macro sets the size and addr fields of this param variable to the values *s* and *a* respectively. A param variable NULL_PARAM is predeclared and can be used as a dummy value parameter. NULL_PARAM can also be used to send and receive messages (signals) with no data in them.

Objects. ALPS provides support for shared objects. An object is an instance of an object type, which is like an abstract data type or a module. It has some internal data representation that is hidden from its users and defines some procedures that can be called by its users. An object type can be created using the primitive MakeObjType and supplying the size of internal data memory as well as the list of externally visible procedures that define the object type. An object of a particular object type can be created using the primitive MakeObject and supplying the object type. An object can have sub-objects, which are created by calling MakeObject from within the object. An externally visible procedure of an object can be called using the Call primitive and supplying it with the object descriptor and the procedure descriptor. A call to an object's procedure is implemented as a remote procedure call. An object variable NULL_OBJECT is predeclared. It represents a non-existing object, *i.e.*, a null object, and can be used as a dummy object parameter.

Manager. An object type can have an optional manager procedure. Instances of such an object type can be *shared* between concurrent processes. Synchronization within a shared object is implemented by programming the manager procedure appropriately. No other synchronization mechanism, such as semaphores, condition variables or barriers, is provided. When a shared object is created, a new process —called the manager process— is created and made to execute the manager procedure. When an externally visible procedure of a shared object is called, the procedure is not executed directly. Instead, control is given to the manager. In other words, the manager process intercepts all calls to the object's externally visible procedures. The manager expresses its readiness to intercept calls by executing the Accept primitive. Once a procedure call is accepted, it is up to the manager to actually execute the procedure. The manager may either execute it immediately or defer its execution until some synchronization conditions are met. When it is time to execute a procedure, the manager can execute the procedure either using the Call primitive or using the Start primitive. If the Call primitive is used, then the manager is blocked until the procedure terminates, meanwhile it cannot accept other calls. This method is useful to execute a procedure exclusively. If the Start primitive is used, then the process that executes the procedure proceeds asynchronously with the manager and the manager can continue with its own execution and perhaps accept other calls. When a procedure that was started using the Start primitive is ready to terminate, the manager process has to recognize its termination by executing the Await primitive which also receives the results from the terminating procedure. After a procedure is executed by the manager using either the Call primitive or the Start primitive, it is up to the manager to return the results to the caller by executing the Return primitive. A procedure variable called NULL_PROC is predeclared. It represents a non-existing procedure, *i.e.*, a null procedure, and can be used as a dummy manager procedure for an object.

Events. An object's manager is a server of *events*. It is usually programmed to select among several alternative events using the Select primitive. An event descriptor (see Section 3) has two fields, *tag* and *proc*. The tag describes the type of the event which can be either ACCEPT_CALL or AWAIT_TERMINATION. An ACCEPT_CALL event means that the manager is willing to accept a call to the procedure proc. An AWAIT_TERMINATION event means that the manager is willing to recognize the termination of one of the ongoing executions of the procedure proc. A macro *MakeEvent(e, t, p)* is provided. Here e is the address of an event variable. The macro sets the tag and proc fields of this event variable to the values t and p respectively.

3 Descriptors

We have seen that the ALPS kernel allows the application programmer to create and use several special entities such as multiplexer channels, procedures and objects. Since only the kernel knows the internal representation of these entities and how they should be manipulated, it is important that the kernel should maintain these entities hidden from the user both to relieve the programmer from unnecessary details and also to protect the entities from arbitrary modification. The only way a user can create or manipulate an entity is by executing an appropriate kernel primitive. This however requires that the user be able to uniquely identify each entity. Such unique identifiers are called *descriptors*.

For each category of descriptors, the kernel provides a descriptor type. These are *chan* for multiplexer channels, *proc* for ALPS procedures, *objtype* for object types, *object* for objects, and *callid* for return addresses. We refer to these as *system level descriptors* which means that the user need not be aware of their internal representation. The kernel also provides three *user level descriptors*, namely, *param* for parameters, *event* for events that can be recognized by a manager using the select primitive, and *objbase* which describes the memory allocated for an object and the procedures defined on it. The user level descriptors are provided to the users as C structures and are defined below.

```
typedef struct {
    int size;
    char *addr;
} param;

typedef struct {
    int tag;                /* tag:  (ACCEPT_CALL, AWAIT_TERMINATION) */
    proc proc;
} event;
```

```
typedef struct {
    int memsize;
    char *Mem;
    int psize;
    proc *ProcList;
} objbase;
```

Different system level descriptors are treated differently by the system. Descriptors of type proc or objtype are global entities which need not and cannot be passed around. Descriptors of type object are the only system level descriptors which can be passed both as ALPS procedure parameters and as message components. Descriptors of type chan and callid cannot be passed, either as procedure parameters or as message components. It is neither desirable nor necessary to pass user level descriptors within parameters or messages.

4 The Kernel Primitives

4.1 Multiplexer Channel Primitives

void **MakeChan**(C)
    chan *C;

MakeChan creates a new multiplexer channel and returns its descriptor in C. Variable size messages can be sent and received on a channel. Only a creator process can execute the MakeChan primitive. The process creating the channel gets RECEIVER access to the channel (this means that the creator process cannot send on the channel). Once a channel is created, the creator can give the channel to i/o and worker processes using the primitives MakeIO and MakeWorker. After all i/o and worker processes that use the channel are created, the creator of the channel executes the primitive FreezeChan indicating that no other processes will be given the channel. A channel is implicitly deleted when its creator terminates. At this time, all remaining messages on the channel are also deleted. Channels cannot be passed as message components or ALPS procedure parameters (the kernel does not recognize such channels).

void **FreezeChan**(C)
    chan C;

This primitive freezes the channel C and prevents the creation of any more i/o or worker processes that could access this channel. The system will not report a channel as dead or almost dead unless it is frozen. Only the process that creates the channel can freeze the channel.

void **Send**(C, Vparam)
    chan C;
    param Vparam;

The process executing this primitive should have either SENDER or SENDER_RECEIVER access to the channel C. The Send primitive deposits the message supplied in Vparam on the channel C and the invoking process continues execution immediately. The Send operation is non-blocking and the implementation should support unbounded buffering. It is an error to execute a Send operation on a channel that is either dead or almost dead.

```
int Receive(C, Rparam)
    chan C;
    param *Rparam;
```

The process that is executing this primitive should have either RECEIVER access or SENDER_RECEIVER access to the channel C. The Receive primitive receives a message on channel C and stores it in Rparam. If the channel C currently has no messages in it, the execution of this primitive is delayed until a message becomes available or until it is confirmed that the channel is dead. When a message is successfully received, the Receive primitive terminates by returning the value SUCCESS. If the channel C becomes dead, the Receive primitive terminates by returning the value ABORT. If the Receive primitive is executed by an i/o process, it is delayed until the channel C is frozen and all worker processes with RECEIVER or SENDER_RECEIVER access right to the channel C have terminated.

```
int Receive21(C, Rparam1, Rparam2)
    chan C;
    param *Rparam1, *Rparam2;
```

The process that is executing this primitive should have either RECEIVER access or SENDER_RECEIVER access to the channel C. The Receive21 primitive delays execution until there are at least two messages on channel C and then removes the two messages in a single step and delivers them in Rparam1 and Rparam2. When two messages are received from the channel, the Receive21 primitive successfully terminates by returning the value SUCCESS. If less than two messages are currently available, the primitive is delayed until two messages are available. If, however, the channel C becomes either dead or almost dead, the Receive21 primitive terminates by returning the value ABORT. If the Receive21 primitive is executed by an i/o process, it is delayed until the channel C is frozen and all worker processes with RECEIVER or SENDER_RECEIVER access right to the channel C have terminated.

```
int Receive12(C1, Rparam1, C2, Rparam2)
    chan C1, C2;
    param *Rparam1, *Rparam2;
```

The process that is executing this primitive should have either RECEIVER access or SENDER_RECEIVER access to both the channels C1 and C2. The Receive12 primitive delays execution until there is at least one message in each of the channels C1 and C2, and then removes the two messages in a single step and delivers them in Rparam1 and Rparam2 respectively. If a message from each channel is received, then the Receive12 primitive successfully terminates by returning the value SUCCESS. If at least one message on each channel is not currently available, the primitive is delayed until the messages are available. If, however, either one or both of the channels become dead, the Receive12 primitive terminates by returning the value ABORT. If the Receive12 primitive is executed by an i/o process, it is delayed until both the channels C1 and C2 are frozen and all worker processes with RECEIVER or SENDER_RECEIVER access right to either of the channels have terminated.

```
void MakeProc(Proc, Func)
    proc *Proc;
    void (*Func)();
```

This primitive creates an ALPS procedure with Func as its code body and returns its descriptor in Proc. This is the only way to tell the kernel to recognize a function Func as an ALPS procedure which can then be used in other ALPS primitives. A proc is a global entity (see Section 3), and should be defined as a global variable.

```
void MakeIO(Proc, csize, ChanList, ChanTypeList, Vparam)
    proc Proc;
    int csize;
    chan ChanList[];
    int ChanTypeList[];
    param Vparam;
```

This primitive creates an i/o process on the same node where it is invoked and makes the new i/o process execute the procedure Proc. The calling process continues immediately. ChanList is a list of channel descriptors and Vparam is a value parameter. Both are passed to Proc so that they can be used by the newly created process. ChanTypeList provides access rights for the channels in ChanList. csize gives the length of the arrays ChanList and ChanTypeList; it is also supplied to Proc. ChanTypeList is only for use by the kernel and is not supplied to Proc. Parameters are supplied to Proc in the order (csize, ChanList, Vparam). Only a creator process can execute the MakeIO primitive.

```
void MakeWorker(Proc, repcount, csize, ChanList, ChanTypeList, Vparam)
    proc Proc;
    int repcount, csize;
    chan ChanList[];
    int ChanTypeList[];
    param Vparam;
```

This primitive creates multiple instances of a worker process, distributes them across the nodes of the parallel machine and makes each instance execute the procedure Proc. The calling process continues immediately. The number of instances of the worker process is specified by repcount. If repcount is FULL_REPLICATION, then the system creates an arbitrary number of instances of the worker process. If repcount is other than FULL_REPLICATION, it should be a positive integer and the system creates exactly repcount number of instances of the worker process. As in the case of MakeIO, ChanList is a list of channel descriptors, Vparam is a value parameter which the newly created processes can use, ChanTypeList provides the access rights for the channels in ChanList, and csize is the length of the arrays ChanList and ChanTypeList. ChanList and Vparam are passed to Proc whereas ChanTypeList is only for kernel use. Parameters are supplied to Proc in the order (csize, ChanList, Vparam). Note that all instances of the worker process can access each channel in ChanList. Only a creator process can execute the MakeWorker primitive.

4.2 Object/Manager Primitives

```
void MakeObjType(ObjType, Memory, psize, ProcList, InitProc, Manager)
    objtype *ObjType;
    int (*Memory)();
    int psize;
    proc ProcList[], InitProc, Manager;
```

This primitive creates an object type and returns its descriptor in ObjType. Memory is a pointer to a user function which returns an integer value, and this is the number of bytes of memory that will be allocated for an object of this type. ProcList specifies a list of procedure descriptors that define the externally visible procedures of the object. psize is the length of ProcList and should be a positive integer. InitProc is the optional initialization procedure which is implicitly executed when the object is created. Manager is an optional procedure which implements the necessary concurrency and synchronization for the calls to the object. The null procedure NULL_PROC can be supplied in place of InitProc and Manager.

```
void MakeObject(Object, ObjType, InitParam)
    object *Object;
    objtype ObjType;
    param InitParam;
```

This primitive creates a new object of type ObjType and returns a descriptor for the created object in Object. The system will first evaluate the function Memory provided by the user in MakeObjType. InitParam will be implicitly passed as a parameter to Memory, and the function Memory should be programmed to receive it. The integer value returned by the function Memory specifies the memory requirement for the object - in bytes. Using this information, the kernel allocates the data memory for the object on one of the nodes. The allocation of the object is transparent to the user; it may be allocated either on the same node as the creating process or on a remote node. The InitProc supplied to the MakeObjType is now implicitly executed and is also supplied with the initialization parameter, InitParam.

If ObjType has a non-null manager procedure, then a process is implicitly created on the node where the object is allocated and is made to execute the manager procedure (this process is the manager process and has creator status). Such an object is shareable and the manager procedure can be programmed to implement the desired synchronization. If the manager procedure is null, *i.e.*, its value was NULL_PROC, then the object is not shareable.

The kernel maintains the location information of the object in the object descriptor. As described earlier, a predefined structure *objbase* is provided to the users. This structure has four fields: (i) *memsize*, the size of the data memory allocated for the object, (ii) *Mem*, the starting address of the data memory, (iii) *psize*, the number of externally visible procedures defined on the object, and (iv) *ProcList*, an array of procedure descriptors identifying the externally visible procedures. For each object, the kernel maintains its objbase structure, and supplies a pointer to this structure as an implicit first parameter to each procedure of the object including the initialization procedure and the manager procedure. Each externally visible procedure of an object should be programmed to accept parameters in the order (ObjBasePtr, Vparam, Rparam). The initialization procedure should accept the parameters (ObjBasePtr, Vparam) whereas the manager procedure needs to accept only the ObjBasePtr. The MakeObject primitive may be used within an object's procedure. This allows an object to have nested sub-objects.

```
void DeleteObject(Object)
    object Object;
```

This primitive deletes the Object. This involves releasing any data memory that was allocated for the object. All processes including the manager should be killed explicitly by the program before the object is deleted. Similarly all sub-objects should be explicitly deleted before the DeleteObject call is issued. Otherwise, the data memory of the object may be released but the processes and/or sub-objects may remain.

```
void Call(Object, Proc, Vparam, Rparam)
    object Object;
    proc Proc;
    param Vparam, *Rparam;
```

This primitive is used to call any ALPS procedure. If the called procedure is part of another object, Object is the object descriptor and Proc is a procedure descriptor which should be a member of ProcList for the object's type. The object may be on the same node as the caller or it may be on a remote node. In either case, the Call primitive is implemented as a remote procedure call. If the object is not shareable, *i.e.*, it has a null manager, the kernel creates a new process on the node where the object is allocated and makes that process execute the procedure Proc. This process gets the creator status. The calling process gets blocked until the called procedure terminates. When the called procedure terminates, the result value returned by the procedure is stored in Rparam. If object has a non-null manager, then the kernel sends a call request to the manager by supplying Object, Proc and Vparam (this request is received by the manager of Object using an Accept primitive). The manager may now use Start to start a new process for Proc and receive results from it using an Await, or it may perform some local computation to get the results. In either case, the calling process gets blocked until the manager executes a Return, when the results are stored in Rparam. Rparam should have the size and addr fields defined at the time the Call primitive is executed.

The Call primitive can also be used to call an independent ALPS procedure that is not within any object or to call an ALPS procedure defined in the same object as the caller. In either case NULL_OBJECT should be supplied as the first parameter to the Call primitive, and Proc should be the descriptor of the procedure being called. Such a procedure call is implemented by creating a new process to execute the procedure. This new process also has the status of creator. The same proc cannot be used both as an independent procedure and as an object's procedure.

```
void Accept(Proc, Callid, Rparam)
    proc Proc;
    callid *Callid;
    param *Rparam;
```

This primitive is meant to be executed by a manager process in order to accept a call to the procedure Proc defined in the parent object. The execution of the Accept primitive is delayed until there is a call to the procedure Proc. When a call arrives, the kernel generates a unique-id which describes the call, *i.e.*, the return address, and stores it in Callid. The Accept primitive also stores the value parameter supplied by the caller in Rparam. When the called procedure is actually executed, it is possible for the manager to supply a value parameter different from the one received in Rparam. Rparam should have the size and addr fields defined at the time the Accept primitive is executed.

```
void Start(Proc, Callid, Vparam)
    proc Proc;
    callid Callid;
    param Vparam;
```

This primitive is meant to be executed by the manager process in order to start the execution of Proc. A call to Proc should have been accepted earlier. The Start primitive creates a new process and makes it execute Proc asynchronously with the manager. The Callid supplied to the newly created process identifies the call accepted earlier, and it also becomes the unique-id of the execution of the Proc that is just started. The Vparam supplied to the started process could be the same value received in the Accept, or a different one. The new process is given the creator status.

```
void Await(Proc, Callid, Rparam)
    proc Proc;
    callid *Callid;
    param *Rparam;
```

This primitive is meant to be executed by the manager process in order to wait for the termination of an execution of Proc. The unique-id of the terminated procedure call is received in Callid, and is the same callid as the one that was passed to Proc when it was started. The results of the call are stored in Rparam. Rparam should have the size and addr fields defined at the time the Await primitive is executed.

```
void Return(Proc, Callid, Vparam)
    proc Proc;
    callid Callid;
    param Vparam;
```

This primitive is meant to be executed by the manager process in order to return from a call to the procedure Proc. Callid gives the unique-id generated when the call was received by the manager. Vparam gives the result parameter to be returned to the caller; this may be different from the actual result returned by the procedure.

```
int Select(esize, EventList)
    int esize;
    event EventList[];
```

This primitive waits for one of several events specified in EventList. An event is either a call to a procedure (recognized by an Accept) or the termination of an executing procedure (recognized by an Await). The length of EventList is specified in esize. If two or more events occur, the event with the smallest array index is selected. This facilitates programming of dynamic priorities in event selection. The index of the selected event is returned as the value of the Select primitive. It should be noted that the Select primitive is used only to select an event. The parameters supplied by the event should be received by a subsequent Accept or Await call on the corresponding procedure. Also note that in simple cases the Accept or Await primitives can be used without necessarily preceding them with a Select.

```
int Count(Event)
    event Event;
```

For any event, there can be more than one occurrence of the event waiting to be serviced. The Count primitive returns the number of pending occurrences of Event that are yet to be serviced by the manager. Typically, this primitive is used at the time of making an event list to be used in a subsequent Select primitive.

4.3 Initialization

```
void AlpsInit(InitFunc)
    void (*InitFunc)();
```

This is the first primitive the user program should call. The present version of the kernel is not a live operating system but is loaded along with the user program. The AlpsInit primitive is used by the kernel both to initialize the kernel data structures and to create the background processes that are required by the kernel. The kernel starts the user program at the function InitFunc by calling it in a fashion similar to an ALPS procedure call; *i.e.*, the process that executes the InitFunc is treated as an i/o process which also has the status of a creator.

5 Programming Tips

Since ALPS is a new non-traditional paradigm for parallel programming, it helps to follow a few guidelines to write correct and efficient programs.

- ALPS does not allow sharing between processes, except through system supported descriptors for channels, objects and parameters. Global variables are **not** allowed. However, some system descriptors like proc and objtype are defined to be global entities, and can be shared between processes. This is because of the lack of high-level language/compiler support. If a high-level language was available, the compiler would generate procs ad objtypes, and make sure they are accessed properly.

- Do not assign values to system descriptor variables. Only the kernel can do so and this happens when you call the appropriate "make" primitive. Once a channel or object is deleted, do not reuse the descriptor without doing another MakeChan or MakeObject on it.

- Remember to unpack params using the same format in which they were packed. And remember that parameter passing is by value/result only.

- It appears that a good way of programming internal parallelism within an ALPS procedure is to have the programmer think of the procedure as having an initial i/o phase, a pipeline of computational phases, and a final i/o phase. So an ALPS procedure could behave as a creator process which makes an INITIAL i/o process, a few worker processes, and a FINAL i/o process. The creator waits for a message from the FINAL i/o process telling it that the computation is done, before it terminates. (Look at the Dot-Product example in Appendix E.2 which illustrates the pipelining concept.)

- The C main program only calls AlpsInit, passing it InitFunc as a parameter. The actual main code of the user program should be put inside the function InitFunc, which is called by the kernel just like it would call an ALPS procedure. Since MakeProcs can be done only after an AlpsInit, a MakeProc on InitFunc is not required.

- Since all ALPS procedures should be registered with the kernel before using them, it is recommended that **all** MakeProcs be done in the function InitFunc itself.

- Termination of all user processes should be explicitly programmed. Make sure that all worker and i/o processes terminate smoothly. i/o processes terminate smoothly upon completion of code body. Worker processes are typically programmed in a loop, receiving values on channels, and processing them. The loops should terminate when the Receives return an ABORT value, and then the worker process should smoothly terminate.

- Make sure that the creator doesn't terminate until the computation is done. If the creator terminates too early, all channels created by it are deleted, and all messages on them are removed, thus terminating useful computation being performed by worker processes. Therefore the creator should wait on a channel on which the last i/o process sends a message signalling the end of the computation, in order to properly synchronize smooth termination.

- Always FreezeChan all channels that were created.

- Managers are typically programmed in a loop, which starts with a Select, followed by the manager either Accepting a Call, or Awaiting termination of a Started process. The manager has to be programmed to terminate, maybe by receiving a dummy Call which causes it to terminate its loop. The manager need not Start a new process for every Call it Accepts, and the user should remember to save the callids of these Calls to later Return results to the Calls.

The user should read [15] to get a feel for ALPS programming. A perusal of the example programs given in Appendix E will also help the user in writing ALPS program.

A Summary of the ALPS Kernel Primitives

A.1 Multiplexer Channel Primitives

```
void MakeChan(C)
    chan *C;

void FreezeChan(C)
    chan C;

void Send(C, Vparam)
    chan C;
    param Vparam;

int Receive(C, Rparam)
    chan C;
    param *Rparam;

int Receive21(C, Rparam1, Rparam2)
    chan C;
    param *Rparam1, *Rparam2;

int Receive12(C1, Rparam1, C2, Rparam2)
    chan C1, C2;
    param *Rparam1, *Rparam2;

void MakeProc(Proc, Func)
    proc *Proc;
    void (*Func)();

void MakeIO(Proc, csize, ChanList, ChanTypeList, Vparam)
    proc Proc;
    int csize;
    chan ChanList[];
    int ChanTypeList[];
    param Vparam;

void MakeWorker(Proc, repcount, csize, ChanList, ChanTypeList, Vparam)
    proc Proc;
    int repcount, csize;
    chan ChanList[];
    int ChanTypeList[];
    param Vparam;
```

A.2 Object/Manager Primitives

```
void MakeObjType(ObjType, Memory, psize, ProcList, InitProc, Manager)
    objtype *ObjType;
    int (*Memory)();
    int psize;
    proc ProcList[], InitProc, Manager;

void MakeObject(Object, ObjType, InitParam)
    object *Object;
    objtype ObjType;
    param InitParam;

void DeleteObject(Object)
    object Object;

void Call(Object, Proc, Vparam, Rparam)
    object Object;
    proc Proc;
    param Vparam, *Rparam;

void Accept(Proc, Callid, Rparam)
    proc Proc;
    callid *Callid;
    param *Rparam;

void Start(Proc, Callid, Vparam)
    proc Proc;
    callid Callid;
    param Vparam;

void Await(Proc, Callid, Rparam)
    proc Proc;
    callid *Callid;
    param *Rparam;

void Return(Proc, Callid, Vparam)
    proc Proc;
    callid Callid;
    param Vparam;
```

```
int Select(esize, EventList)
    int esize;
    event EventList[];

int Count(Event)
    event Event;
```

A.3 Initialization

```
void AlpsInit(InitFunc)
    void (*InitFunc)();
```

B List of ALPS Constants

Apart from the constants introduced thus far, the ALPS kernel also provides a constant called NUMBER_OF_PROCESSORS, which gives the number of physical processors expected to be available to the users. The complete lists are given below.

B.1 ALPS Literals

```
SENDER
RECEIVER
SENDER_RECEIVER
SUCCESS                    /* Value 1 */
ABORT                      /* Value 0 */
ACCEPT_CALL
AWAIT_TERMINATION
FULL_REPLICATION
NUMBER_OF_PROCESSORS
```

B.2 Predeclared ALPS Variables

```
NULL_PROC
NULL_PARAM
NULL_OBJECT
```

B.3 ALPS Macros

```
MakeParam(p, s, a)
MakeEvent(e, t, p)
```

C ALPS Error Checking

ALPS primitives try to keep error checking down to a minimum. However, some checks are performed, which either result in warnings or errors. No error values are returned. A warning results only in a message on the screen. An error not only causes an error message, but also causes the kernel to abort the user program as best as it can. The error message only gives the error detected, and the primitive during the execution of which the error occurred. It doesn't say which ALPS procedure was executing when the error occurred.

C.1 Warnings

- "Result parameter was truncated because it did not fit in the buffer provided." Generated by Receive, Receive21, Receive12, Call, Accept, and Await.

C.2 Errors

- "Not a creator." Generated by MakeChan, MakeIO, MakeWorker, and FreezeChan.
- "Invalid channel." Generated by FreezeChan, Send, Receive, Receive21, and Receive12.
- "SENDER or SENDER-RECEIVER access right required." Generated by Send.
- "RECEIVER or SENDER-RECEIVER access right required." Generated by Receive, Receive21, and Receive12.
- "Cannot send on a dead or almost dead channel." Generated by Send.
- "Invalid procedure." Generated by MakeIO, MakeWorker, Call, Accept, Start, Await, and Return.
- "Invalid channel in ChanList." Generated by MakeIO and MakeWorker.
- "Frozen channel in ChanList." Generated by MakeIO and MakeWorker.
- "Invalid procedure in ProcList." Generated by MakeObjType.
- "Invalid InitProc." Generated by MakeObjType.
- "Invalid Manager procedure." Generated by MakeObjType.
- "Invalid ObjType." Generated by MakeObject.
- "Invalid Object." Generated by DeleteObject and Call.
- "Called procedure not within specified object." Generated by Call.
- "Invalid Callid used." Generated by Accept, Start, Await and Return.
- "Event tag is not ACCEPT_CALL or AWAIT_TERMINATION." Generated by Select and Count.

D Input and Output

The ALPS kernel does not define Input/Output operations. However, each implementation supports the standard C I/O operations. For each supported I/O operation, the executing process gets blocked until the I/O operation is completed. When concurrent processes do I/O, the operations may be interleaved in some arbitrary fashion.

For this purpose, each implementation requires the user to use the ALPS C I/O library rather than the standard C I/O library. An ALPS header file "alpsio.h" is provided, and the user should use the declaration

```
#include "alpsio.h"
```

instead of the usual

```
#include <stdio.h>.
```

E Example Programs

E.1 Sum of Numbers

```
/*   /n/tribe/0/alps/examples/sum-of-numbers.c
     Manas Mandal. Aug 17, 1989.          */

/* This sample program computes the sum of integers read from the standard
   input. The Initial i/o process reads the values from the input, and sends
   them on a multiplexer channel. The Adder worker processes atomically
   receive two of these integers, add them, and put the result back on the
   same channel. When only one value is left, it is the sum of all the
   numbers, and the channel is almost dead. The Final i/o process now
   receives this value, and prints it out on the standard output.

   The SumOfNumbers procedure, which is an ALPS procedure with internal
   parallelism, creates two channels. The first channel is used by both the
   i/o processes and all worker processes. While the Initial i/o process can
   only send values on this channel, each Adder worker process can both
   receive and send values on this channel. The Final i/o process can only
   receive a value on this channel. The second channel is a synchronizing
   channel between the Final i/o process, which is a sender for this channel,
   and the SumOfNumbers process, which is a receiver. The Initial i/o process
   terminates after it sends all its messages. The Adder worker processes
   block on the Receive21 when there is only one value left. Now the system
   can recognize that the channel is almost dead, terminate all Adder worker
   processes, and give the single remaining message to the Final i/o process.
   After it finishes writing out the results to finish the computation, the
   Final process sends a synchronizing message on the second channel to the
   SumOfNumbers process, and terminates itself. The SumOfNumbers process, a
   creator process, now terminates, and all channels are implicitly deleted.
*/

#include "alps.h"
#include "alpsio.h"

void Initial(csize, ChanList, VParam)
int csize;
chan ChanList[];
param VParam;
{
     chan SumChan;
     int Num;
     param NumParam;
```

```
     SumChan = ChanList[0];
     MakeParam(&NumParam, sizeof(int), &Num);

     while (scanf("%d", &Num) != EOF)
     {
          Send(SumChan, NumParam);
     }
}

void Adder(csize, ChanList, Vparam)
int csize;
chan ChanList[];
param Vparam;
{
     chan SumChan;
     int Val1, Val2, Sum;
     param Val1Param, Val2Param, SumParam;

     SumChan = ChanList[0];
     MakeParam(&Val1Param, sizeof(int), &Val1);
     MakeParam(&Val2Param, sizeof(int), &Val2);
     MakeParam(&SumParam, sizeof(int), &Sum);

     while (Receive21(SumChan, &Val1Param, &Val2Param))
     {
          Sum = Val1 + Val2;
          Send(SumChan, SumParam);
     }
}

void Final(csize, ChanList, Vparam)
int csize;
chan ChanList[];
param Vparam;
{
     chan SumChan, SynchChan;
     int Sum;
     param SumParam;

     SumChan = ChanList[0];
     SynchChan = ChanList[1];
     MakeParam(&SumParam, sizeof(int), &Sum);
```

```
     Receive(SumChan, &SumParam);
     printf("The sum of the numbers is %d\n", Sum);

     Send(SynchChan, NULL_PARAM);
}

void SumOfNumbers()
{
     proc InitialProc, AdderProc, FinalProc;
     chan SumChan, SynchChan;
     chan c1[1], c2[2];
     int c1_type[1], c2_type[2];

     MakeProc(&InitialProc, Initial);
     MakeProc(&AdderProc, Adder);
     MakeProc(&FinalProc, Final);

     MakeChan(&SumChan);
     MakeChan(&SynchChan);

     c1[0] = SumChan;
     c1_type[0] = SENDER;
     MakeIO(InitialProc, 1, c1, c1_type, NULL_PARAM);

     c1[0] = SumChan;
     c1_type[0] = SENDER_RECEIVER;
     MakeWorker(AdderProc, FULL_REPLICATION, 1, c1, c1_type, NULL_PARAM);

     c2[0] = SumChan;
     c2[1] = SynchChan;
     c2_type[0] = RECEIVER;
     c2_type[1] = SENDER;
     MakeIO(FinalProc, 2, c2, c2_type, NULL_PARAM);

     FreezeChan(SumChan);
     FreezeChan(SynchChan);

     Receive(SynchChan, &NULL_PARAM);
}

main()
{
     AlpsInit(SumOfNumbers);
}
```

E.2 Dot Product

```
/*   /n/tribe/0/alps/examples/dot-prod.c
     Manas Mandal. Aug 17, 1989.   */

/* This sample program computes the dot-product of 2 vectors. This program is
   similar to the sum-of-numbers program, except that the i/o processes are
   given filenames as params, and there is a pipeline of Multiplier and Adder
   worker processes, all working in parallel. The Adder processes start the
   addition before all products are calculated, thus improving parallelism */

#include "alps.h"
#include "alpsio.h"

#define FileNameSize 20

typedef int valuetype;

typedef struct {
     valuetype a_val, b_val;
} pairtype;

void Initial(csize, ChanList, FileParam)
int csize;
chan ChanList[];
param FileParam;
{
     chan PairChan;
     char FileName[FileNameSize];
     pairtype Pair;
     param PairParam;
     FILE *fp;

     strncpy(FileName, (char *)FileParam.addr, FileNameSize);
     fp = fopen(FileName, "r");

     PairChan = ChanList[0];
     MakeParam(&PairParam, sizeof(pairtype), &Pair);

     while (fscanf(fp,"%d%d", &(Pair.a_val), &(Pair.b_val)) != EOF)
     {
          Send(PairChan, PairParam);
     }
}
```

```
void Multiplier(csize, ChanList, Vparam)
int csize;
chan ChanList[];
param Vparam;
{
     chan PairChan, SumChan;
     pairtype Pair;
     valuetype Product;
     param PairParam, ProductParam;

     PairChan = ChanList[0];
     SumChan = ChanList[1];
     MakeParam(&PairParam, sizeof(pairtype), &Pair);
     MakeParam(&ProductParam, sizeof(valuetype), &Product);

     while (Receive(PairChan, &PairParam))
     {
          Pair = *((pairtype *)(PairParam.addr));
          Product = Pair.a_val * Pair.b_val;
          Send(SumChan, ProductParam);
     }
}

void Adder(csize, ChanList, Vparam)
int csize;
chan ChanList[];
param Vparam;
{
     chan SumChan;
     valuetype Val1, Val2, Sum;
     param Val1Param, Val2Param, SumParam;

     SumChan = ChanList[0];
     MakeParam(&Val1Param, sizeof(valuetype), &Val1);
     MakeParam(&Val2Param, sizeof(valuetype), &Val2);
     MakeParam(&SumParam, sizeof(valuetype), &Sum);

     while ((Receive21(SumChan, &Val1Param, &Val2Param)) == SUCCESS)
     {
          Sum = Val1 + Val2;
          Send(SumChan, SumParam);
     }
}
```

```
void Final(csize, ChanList, FileParam)
int csize;
chan ChanList[];
param FileParam;
{
     chan SumChan, SynchChan;
     char FileName[FileNameSize];
     valuetype Sum;
     param SumParam;
     FILE *fp;

     strncpy(FileName, (char *)FileParam.addr, FileNameSize);
     fp = fopen(FileName, "w");

     SumChan = ChanList[0];
     SynchChan = ChanList[1];
     MakeParam(&SumParam, sizeof(valuetype), &Sum);

     Receive(SumChan, &SumParam);
     fprintf(fp, "The result of the dot product is %d\n", Sum);
     fclose(fp);

     Send(SynchChan, NULL_PARAM);
}

void DotProduct()
{
     proc InitialProc, MultiplierProc, AdderProc, FinalProc;
     chan PairChan, SumChan, SynchChan;
     chan c1[1], c2[2];
     int c1_type[1], c2_type[2];
     param FileParam;
     char InputFileName[FileNameSize], OutputFileName[FileNameSize];

     printf("Please enter Input filename: ");
     scanf("%s", InputFileName);
     printf("Please enter Output filename: ");
     scanf("%s", OutputFileName);
     printf("The dot product of the arrays in %s will be put in %s\n",
            InputFileName, OutputFileName);
     printf("Here we go ...");

     MakeProc(&InitialProc, Initial);
```

```
    MakeProc(&MultiplierProc, Multiplier);
    MakeProc(&AdderProc, Adder);
    MakeProc(&FinalProc, Final);

    MakeChan(&PairChan);
    MakeChan(&SumChan);
    MakeChan(&SynchChan);

    c1[0] = PairChan;
    c1_type[0] = SENDER;
    MakeParam(&FileParam, FileNameSize, InputFileName);
    MakeIO(InitialProc, 1, c1, c1_type, FileParam);

    c2[0] = PairChan;
    c2[1] = SumChan;
    c2_type[0] = RECEIVER;
    c2_type[1] = SENDER;
    MakeWorker(MultiplierProc, FULL_REPLICATION, 2, c2, c2_type, NULL_PARAM);

    c1[0] = SumChan;
    c1_type[0] = SENDER_RECEIVER;
    MakeWorker(AdderProc, FULL_REPLICATION, 1, c1, c1_type, NULL_PARAM);

    c2[0] = SumChan;
    c2[1] = SynchChan;
    c2_type[0] = RECEIVER;
    c2_type[1] = SENDER;
    MakeParam(&FileParam, FileNameSize, OutputFileName);
    MakeIO(FinalProc, 2, c2, c2_type, FileParam);

    FreezeChan(PairChan);
    FreezeChan(SumChan);
    FreezeChan(SynchChan);

    Receive(SynchChan, &NULL_PARAM);

    printf("... The result has been put in %s\n", OutputFileName);
    printf("\nHave a Nice Day\n");
}

main()
{
    AlpsInit(DotProduct);
}
```

E.3 Prime Number Generation

```
/*  /n/tribe/0/alps/examples/primegen.c

    Yee-Tim Lee. Aug 21, 1989.
    Mahendra Ramachandran. Aug 29, 1989.

    This program generates all prime numbers less than a given number N.
    In order to compute them, it needs the number of prime numbers between
    2 and sqrt(N) and also those prime numbers themselves.  It would then
    generate all prime numbers that are between sqrt(N) and N.

*/

#include "alps.h"
#include "alpsio.h"
#define MAX 10

void Input(csize, clist, N_param)
int csize;
chan clist[];
param N_param;
{
     param potential_prime_param;
     int i, N;
     chan to_sieve;

     to_sieve = clist[0];
     N = *((int *)(N_param.addr));
     MakeParam(&potential_prime_param, sizeof(int), &i);

     for (i=(int)sqrt(N); i<=N; i++)
     {
          Send(to_sieve, potential_prime_param);
     }
}

void Sieve(csize, clist, prime_i_param)
int csize;
chan clist[];
param prime_i_param;
{
     param potential_prime_param;
     int prime_i, potential_prime;
```

```
    chan in_chan;
    chan out_chan;

    prime_i = *((int *)(prime_i_param.addr));
    in_chan = clist[0];
    out_chan = clist[1];
    MakeParam(&potential_prime_param, sizeof(int), &potential_prime);

    while (Receive(in_chan, &potential_prime_param))
    {
         if ((potential_prime % prime_i) != 0)
         {
              Send(out_chan, potential_prime_param);
         }
    }
}

void Output(csize, clist, vparam)
int csize;
chan clist[];
param vparam;
{
    param prime_param;
    int prime;
    chan results_chan;
    chan signal_chan;

    results_chan = clist[0];
    signal_chan = clist[1];
    MakeParam(&prime_param, sizeof(int), &prime);

    while (Receive(results_chan, &prime_param))
    {
         printf("%d is a prime\n",prime);
    }

    Send(signal_chan, NULL_PARAM);
}

void PrimeGen()
{
    FILE *fp;
    char filename[10];
    int prime_i, N, num_input_primes, i;
```

```
param prime_i_param, N_param;
proc InProc, OutProc, SieveProc;
chan sieve_chans[MAX];
chan c1[1], c2[2];
chan signal_chan;
int  c1_type[1], c2_type[2];

MakeProc (&InProc, Input);
MakeProc(&OutProc, Output);
MakeProc(&SieveProc, Sieve);

MakeParam(&N_param, sizeof(int), &N);
MakeParam(&prime_i_param, sizeof(int), &prime_i);

puts("Up till what number N do you want primes generated?");
scanf("%d", &N);
puts("Type file having number of input primes and the actual primes");
gets(filename);
fp = fopen(filename, "r");
fscanf(fp, "%d", num_input_primes);
if (num_input_primes >= MAX)
{
     printf("Error- overflow the maximum number of channels \n");
}

for (i = 0; i <= num_input_primes; i++)
{
     MakeChan(&sieve_chans[i]);
}
MakeChan(&signal_chan);

c1[0] = sieve_chans[0];
c1_type[0] = SENDER;
MakeIO(InProc, 1, c1, c1_type, N_param);

c2_type[0] = RECEIVER;
c2_type[1] = SENDER;

for (i = 1; i <= num_input_primes; i++)
{
     fscanf(fp, "%d", &prime_i);
     c2[0] = sieve_chans[i-1];
     c2[1] = sieve_chans[i];
```

```
        MakeWorker(SieveProc, FULL_REPLICATION, 2, c2, c2_type,
                               prime_i_param);
    }

    c2[0] = sieve_chans[num_input_primes];
    c2[1] = signal_chan;
    MakeIO(OutProc, 2, c2, c2_type, NULL_PARAM);

    for (i = 0; i <= num_input_primes; i++)
    {
        FreezeChan(&sieve_chans[i]);
    }
    FreezeChan(&signal_chan);

    Receive(signal_chan, NULL_PARAM);
}


main()
{
    AlpsInit(PrimeGen);
}
```

E.4 Object Symbol Table

```
/* /n/tribe/0/alps/examples/symtable-slice.c
   /n/tribe/0/alps/examples/symtable.c

   Mahendra Ramachandran. Aug 29, 1989.

   The symbol table program consists of two object types, a symbol
   table object type which has a Manager so that its instances can be
   shared, and a symbol table slice object type whose instances cannot be
   shared. Both object types provide a C function, objtypename_makobjtyp,
   which initializes an objtype descriptor which can then be used by the
   user to create instances of that object type.  The procedure names of
   the exported functions as well as object type names are declared
   globally and are initialized by the objtypename_makobjtyp functions.

   To use the symbol table object type, you must first call the C function
   "symtab_makobjtyp()" which initializes the globally declared objtype
   descriptor "symtab_objtype". Using this objtype descriptor, you can
   call MakeObject to create an instance of the object type. When
   you call MakeObject, you should supply an initialization parameter
   which should specify the desired table size. This parameter will be
   used in two ways: (i) it will be used by the C function "symtab_memory"
   which is executed implicitly to compute the memory requirement for the
   object, and (ii) it will be supplied to the initialization function
   "symtab_initobject" which will be implicitly executed after memory
   allocation is completed.  The symbol table object is divided into a
   few symbol table slice objects. Procedures of the symbol table object
   are programmed such that they in turn call the procedures of
   appropriate symbol table slice objects.  The slice objects are created
   by the initialization function "symtab_initobject".

   Once everything is set up, you may make calls to the procedures
   "symtab_insert_proc" and "symtab_search_proc" with the item being
   inserted or searched for as the value parameter. The result parameter
   is an integer which is either FULL or NOTFULL for inserts and FOUND or
   NOTFOUND for searches.  Finally to kill the symbol table object, you
   should call "symtab_terminate_proc". Your program should have a typedef
   declaration which defines "datatype" to be the desired type of the
   symbol table entry item.
*/
```

```
/**************************************************************************/
/*                       Symbol Table Slice Object                        */
/**************************************************************************/

#define FULL 0
#define NOTFULL 1
#define FOUND 1
#define NOTFOUND 0

/* Global declaration: should be used after calling "slice_makobjtyp". */
proc slice_insert_proc, slice_search_proc;
objtype slice_objtype;

void slice_initobject(objbaseptr, init_param)
objbase *objbaseptr;
param init_param;
{
  int *currentindex, *slicesize;

  currentindex = (int *)objbaseptr->Mem;
  slicesize = (int *)(objbaseptr->Mem + sizeof(int));
  *currentindex = 0;
  *slicesize = *(int *)init_param.addr;
}

void slice_insert(objbaseptr, item, status)
objbase *objbaseptr;
param item, *status;
{
  int *currentindex, slicesize, slice_status;
  datatype *table;

  currentindex = (int *)objbaseptr->Mem;
  slicesize = *(int *)(objbaseptr->Mem + sizeof(int));
  table = (datatype *)(objbaseptr->Mem + 2 * sizeof(int));
  memcpy(&table[*currentindex], item.addr, item.size);
  (*currentindex)++;
  if (*currentindex == slicesize) slice_status = FULL;
  else slice_status = NOTFULL;
  MakeParam(status, sizeof(slice_status), &slice_status);
}

void slice_search(objbaseptr, item, status)
objbase *objbaseptr;
```

```
param item, *status;
{
  int *currentindex, slice_status, i;
  datatype *table;

  currentindex = (int *)objbaseptr->Mem;
  table = (datatype *)(objbaseptr->Mem + 2 * sizeof(int));
  slice_status = NOTFOUND;
  for(i = 0; i < *currentindex; i++)
    if (memcmp(table[i], item.addr, item.size) == 0)
    {
      slice_status = FOUND;
      break;
    }
  MakeParam(status, sizeof(int), &slice_status);
}

int slice_memory(init_param)
param init_param;
{
  int slicesize, mem_required;

  slicesize = *(int *)init_param.addr;
  mem_required = 2 * sizeof(int) + slicesize * sizeof(datatype);
  return(mem_required);
}

void slice_makobjtyp()
{
  proc proclist[2], initproc;

  MakeProc(&slice_insert_proc, slice_insert);
  proclist[0] = slice_insert_proc;
  MakeProc(&slice_search_proc, slice_search);
  proclist[1] = slice_search_proc;
  MakeProc(&initproc, slice_initobject);

  MakeObjType(&slice_objtype, slice_memory, 2, proclist, initproc, NULL_PROC);
}
```

```
/****************************************************************************/
/*                          Symbol Table Object                             */
/****************************************************************************/

/* Global declaration: should be used after calling "symtab_makobjtyp". */
proc symtab_insert_proc, symtab_search_proc, symtab_terminate_proc;
objtype symtab_objtype;

void symtab_initobject(objbaseptr, init_param)
objbase *objbaseptr;
param init_param;
{
  int tablesize, number_of_slices, slicesize, slice_mem_required;
  param slice_size_param;
  int *currentslice, *maxslice, i;
  object *slice_table;

  tablesize = *(int *)init_param.addr;
  number_of_slices = NUMBER_OF_PROCESSORS;
  if (tablesize % number_of_slices == 0)
      slicesize = tablesize / number_of_slices;
  else
      slicesize = tablesize / number_of_slices + 1;
  MakeParam(&slice_size_param, sizeof(int), &slicesize);

  currentslice = (int *)(objbaseptr->Mem);
  *currentslice = 0;
  maxslice = (int *)(objbaseptr->Mem + sizeof(int));
  *maxslice = number_of_slices;

  slice_table = (object *)(objbaseptr->Mem + 2 * sizeof(int));
  for (i=0; i < number_of_slices; i++)
    MakeObject(&slice_table[i], slice_objtype, slice_size_param);
}

void symtab_insert(objbaseptr, item, status)
objbase *objbaseptr;
param item, *status;
{
  param slice_status_param;
  int table_status, slice_status;
  int *currentslice, *maxslice;
  object *slice_table;
```

```
  MakeParam(status, sizeof(int), &table_status);
  MakeParam(&slice_status_param, sizeof(int), &slice_status);

  currentslice = (int *)(objbaseptr->Mem);
  maxslice = (int *)(objbaseptr->Mem + sizeof(int));
  slice_table = (object *)(objbaseptr->Mem + 2 * sizeof(int));

  if(*currentslice == *maxslice) table_status = FULL;
  else {
        table_status = NOTFULL;
        Call(slice_table[*currentslice], slice_insert_proc, item,
                                     &slice_status_param);
        if (slice_status == FULL) (*currentslice)++;
       }
}

void query(numchans, chanlist, vparam)
int numchans;
chan chanlist[];
param vparam;
{
  chan to_workers;
  int currentslice, i;
  object *slice_table;
  param  slice_obj_param;

  to_workers = chanlist[0];
  currentslice = *(int *)(vparam.addr);
  slice_table = (object *)(vparam.addr + 2 * sizeof(int));

  for(i = 0; i <= currentslice; i++)
    { MakeParam(&slice_obj_param, sizeof(object), &slice_table[i]);
      Send(to_workers, slice_obj_param);
    }
}

void locate(numchans, chanlist, data_param)
int numchans;
chan chanlist[];
param data_param;
{
  chan query_chan, results_chan;
  object slice;
  param slice_param, result_param;
```

```
  int result;

  query_chan = chanlist[0];
  results_chan = chanlist[1];

  MakeParam(&slice_param, sizeof(object), &slice);
  MakeParam(&result_param, sizeof(int), &result);

  while(Receive(query_chan, &slice_param) == SUCCESS)
    { Call(slice, slice_search_proc, data_param, &result_param);
      Send(results_chan, result_param);
    }
}

void symtab_search(objbaseptr, item, status)
objbase *objbaseptr;
param item, *status;
{
  proc queryproc, locateproc;
  chan query_to_locate, locate_to_creator;
  chan c1[1], c2[2];
  int c1_type[1], c2_type[2];
  param query_param;
  int currentslice, query_param_size, result;

  MakeProc(&queryproc,query);
  MakeProc(&locateproc,locate);

  MakeChan(&query_to_locate);
  MakeChan(&locate_to_creator);

  currentslice = *(int *)(objbaseptr->Mem);
  query_param_size = currentslice * sizeof(object) + 2 * sizeof(int);
  MakeParam(&query_param, query_param_size, objbaseptr->Mem);

  c1[0] = query_to_locate;
  c1_type[0] = SENDER;
  MakeIO(queryproc, 1, c1, c1_type, query_param);

  c2[0] = query_to_locate;
  c2_type[0] = RECEIVER;
  c2[1] = locate_to_creator;
  c2_type[1] = SENDER;
  MakeWorker(locateproc, FULL_REPLICATION, 2, c2, c2_type, item);
```

```
  FreezeChan(query_to_locate);
  FreezeChan(locate_to_creator);

  MakeParam(status, sizeof(int), &result);

  while(Receive(locate_to_creator, status) == SUCCESS)
    if (result == FOUND) return;
}

void symtab_terminate(objbaseptr, item, status)
objbase *objbaseptr;
param item, *status;
{ }

void Manager(objbaseptr)
objbase *objbaseptr;
{
  event events_list[3];
  callid caller_id;
  param data_param, result_param, tablestatus_param;
  datatype item;
  int result, tablestatus, event_selected, currentslice, i;
  object *slice_table;

  MakeEvent(&events_list[0], ACCEPT_CALL, symtab_insert_proc);
  MakeEvent(&events_list[1], ACCEPT_CALL, symtab_search_proc);
  MakeEvent(&events_list[2], ACCEPT_CALL, symtab_terminate_proc);

  MakeParam(&data_param, sizeof(int), &item);
  MakeParam(&result_param, sizeof(int), &result);
  MakeParam(&tablestatus_param, sizeof(int), &tablestatus);

  while (1)
    {
      event_selected = Select(3, events_list);
      if (event_selected == 0)
        {
          Accept(symtab_insert_proc, &caller_id, &data_param);
          Call(NULL_OBJECT, symtab_insert_proc, data_param,
                            &tablestatus_param);
          Return(symtab_insert_proc, caller_id, tablestatus_param);
        }
      else if (event_selected == 1)
```

```
        {
          Accept(symtab_search_proc, &caller_id, &data_param);
          Call(NULL_OBJECT, symtab_search_proc, data_param, &result_param);
          Return(symtab_search_proc, caller_id, result_param);
        }
      else
        {
          Accept(symtab_terminate_proc, &caller_id, NULL_PARAM);
          break;
        }
    }
  currentslice = *(int *)(objbaseptr->Mem);
  slice_table = (object *)(objbaseptr->Mem + 2 * sizeof(int));
  for (i = 0; i < currentslice; i++) DeleteObject(slice_table[i]);
}

int symtab_memory(init_param)
param init_param;
{
  int number_of_slices, mem_required;

  number_of_slices = NUMBER_OF_PROCESSORS;
  mem_required = 2 * sizeof(int) + number_of_slices * sizeof(object);
  return(mem_required);
}

void symtab_makobjtyp()
{
  proc proclist[3], initproc, manager_proc;

  slice_makobjtyp();

  MakeProc(&symtab_insert_proc, symtab_insert);
  proclist[0] = symtab_insert_proc;
  MakeProc(&symtab_search_proc, symtab_search);
  proclist[1] = symtab_search_proc;
  MakeProc(&symtab_terminate_proc, symtab_terminate);
  proclist[2] = symtab_terminate_proc;
  MakeProc(&initproc, symtab_initobject);
  MakeProc(&manager_proc, Manager);

  MakeObjType(&symtab_objtype, symtab_memory, 3, proclist,
                             initproc, manager_proc);
}
```

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. "Mach: A New Kernel Foundation for UNIX Development". In *Proceedings of the Summer 1986 USENIX Technical Conference Exhibition*, June 1986.

[2] *The Ada Programming Language Reference Manual.* United States Department of Defense, US Government Printing Office, February 1983. ANSI/MIL-STD-1815A-1983.

[3] Gregory R. Andrews. "Synchronizing Resources". *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, October 1981.

[4] Arvind, David E. Culler, Robert A. Ianucci, Vinod Kathail, Keshav Pingali, and Robert E. Thomas. "The Tagged Token Dataflow Architecture". Internal report, distributed in subject 6.83s, August 1983.

[5] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. "PRESTO: A System for Object-oriented Parallel Programming". *Software—Practice and Experience*, 18(8):713–732, August 1988.

[6] David R. Cheriton. "The V Kernel – A Software Base for Distributed Systems". *IEEE Software*, 1(2):19–42, April 1984.

[7] J. B. Dennis. "Data Flow Supercomputers". *COMPUTER*, 13(11):48–56, November 1980.

[8] D. Gelernter. "Generative Communication in Linda". *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[9] C. A. R. Hoare. "Communicating Sequential Processes". *Communications of the ACM*, 21(8):666–677, August 1978.

[10] Inmos. *OCCAM Programming Manual.* Prentice Hall International, 1984.

[11] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. "Dependence Graphs and Compiler Optimization". In *8th ACM Symposium on Principles of Programming Languages*, pages 207–218, ACM, 1981.

[12] David J. Kuck, Edward S. Davidson, Duncan H. Lawrie, and Ahmed H. Sameh. "Parallel Supercomputing Today and the Cedar Approach". *SCIENCE*, 967–974, February 1986.

[13] Barbara Liskov. "Distributed Programming in Argus". *Communications of the ACM*, 31(3):300–312, March 1988.

[14] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. "The Sprite Network Operating System". *COMPUTER*, 23–35, February 1988.

[15] Prasad R. Vishnubhotla. *"A Paradigm for Developing Portable and Efficient Parallel Programs for MIMD Architectures"*. Technical Report OSU-CISRC-6/89-TR 19, The Ohio State University Computer and Information Science Research Center, June 1989.

[16] Prasad R. Vishnubhotla. *"Multiplexer Channels: A Many-to-Many Communication Mechanism for Parallel Processing"*. Technical Report OSU-CISRC-8/88-TR 25, The Ohio State University Computer and Information Science Research Center, August 1988.

[17] Prasad R. Vishnubhotla. "Synchronization and Scheduling in ALPS Objects". In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 256–264, IEEE, June 1988.