AMPL: Design, Implementation, and Evaluation of A Multiprocessing Language

Roger B. Dannenberg

31 March 1981

Abstract

AMPL is an experimental high-level language for expressing parallel algorithms which involve many interdependent and cooperating tasks. AMPL is a strongly-typed language in which all inter-process communication takes place via message passing. The language has been implemented on the Cm* multiprocessor, and a number of programs have been written to perform numeric and symbolic computation. In this report, the design decisions relating to process communication primitives are discussed, and AMPL is compared to several other languages for parallel processing. The implementation of message passing, process creation, and parallel garbage collection are described. Measurements of several AMPL programs are used to study the effects of language design decisions upon program performance and algorithm design.

This research was partly supported by the National Science Foundation under grant SPI-7821198 and a fellowship from the Fannie and John Hertz Foundation.

The Cm* multiprocessor project was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

Table of Contents

List of Figures

List of Tables

1 Introduction

Motivation for the design of AMPL originated in the consideration of data-flow languages and experience with hardware design. Since the inception of digital electronics, hardware designers have designed interpreters, real-time control systems, and data-processing systems which utilize thousands of computing elements, all operating in parallel. Data-flow computers are an approach to achieving this level of parallelism in a programmable machine. Data-flow programs are normally restricted, in that only functional or "stateless" programs are allowed. The value of a "variable" cannot be changed since values are used to synchronize computation. In AMPL, networks of computing elements can be interconnected to perform computation like the nodes of data-flow machines, but like hardware elements, nodes can have state. Not only does this allow operations with side-effects, but storage can be distributed and maintained at the site of computation rather than in a "structure processor" [1] or other special storage unit as required by a data-flow machine.

AMPL is not designed for the implementation of systems which must survive hardware and software errors in order to provide reliable service over extended periods of time. As in early algorithmic languages for uniprocessors, little attention is paid to issues of exception-handling and error recovery. AMPL is machine-independent and can be implemented on a variety of multiprocessor and computer network structures. Since AMPL is an experimental language, it has intentionally been kept simple and small. A richer syntax and additional features would be expected of a production language.

1.1 Language features

The language is based on the use of message passing for both communication and synchronization. Shared memory is not available. The language provides for the dynamic creation of processes, the ability to pass references to processes through messages, and garbage collection of processes. AMPL is a strongly-typed language, and borrows heavily from Modula [37] and Pascal [36], using a similar syntax and also restricting the use of dynamic structures to simplify storage allocation [35]. Its parallel processing facilities are descendents of CSP [19], although substantial changes have been made, for example, to allow dynamic process creation.

A short example of an AMPL program is given in Figure 1. This example is presented here only to give the reader an idea of what the language is like; details of the language will be explained in Section 4. The program adds 10 pairs of numbers and prints the results. Two processes are used in addition to an output process predefined by the language. One process generates numbers to be added, and formats the output. The other process actually performs the additions, returning results to the formatting process. Figure 2 illustrates the run-time structure and communication paths of this program.

```
type refportinteger = refport integer;

module main; {this process created automatically}
port
    SumPort: integer;
var
    sum, i: integer;
    adder: refmod AdderMod;
begin    {add some numbers using an instance of AdderMod}
    adder := create(AdderMod, Self.sumport);   {parameter tells    }
    i := 1;                                    {where to send results}
    while i < 10 do
        send i to adder.APort;
        send 100 to adder.BPort;
        send i to WrInt;             {print i}
        send ' + 100 = ' to WrStr;   {print a string}
        accept SumPort(sum);         {get sum of i + 100}
        send sum to WrInt;           {print sum}
        send 1 to WrLn;              {print 1 newline}
        i := i + 1
        end;
    end;


{the following module adds two integers and sends the sum to result:}

module AdderMod(result: refportinteger);
port
    APort: integer(1);    {operands arrive one at a time}
    BPort: integer(1);    { so buffer size is set to 1 }
var
    a, b: integer;
begin
    while true do
        accept APort(a);        {get operands... }
        accept BPort(b);
        send a + b to result;   { ...return result}
        end
    end
```

Figure 1: A simple AMPL program which adds 10 pairs of numbers and prints the results.

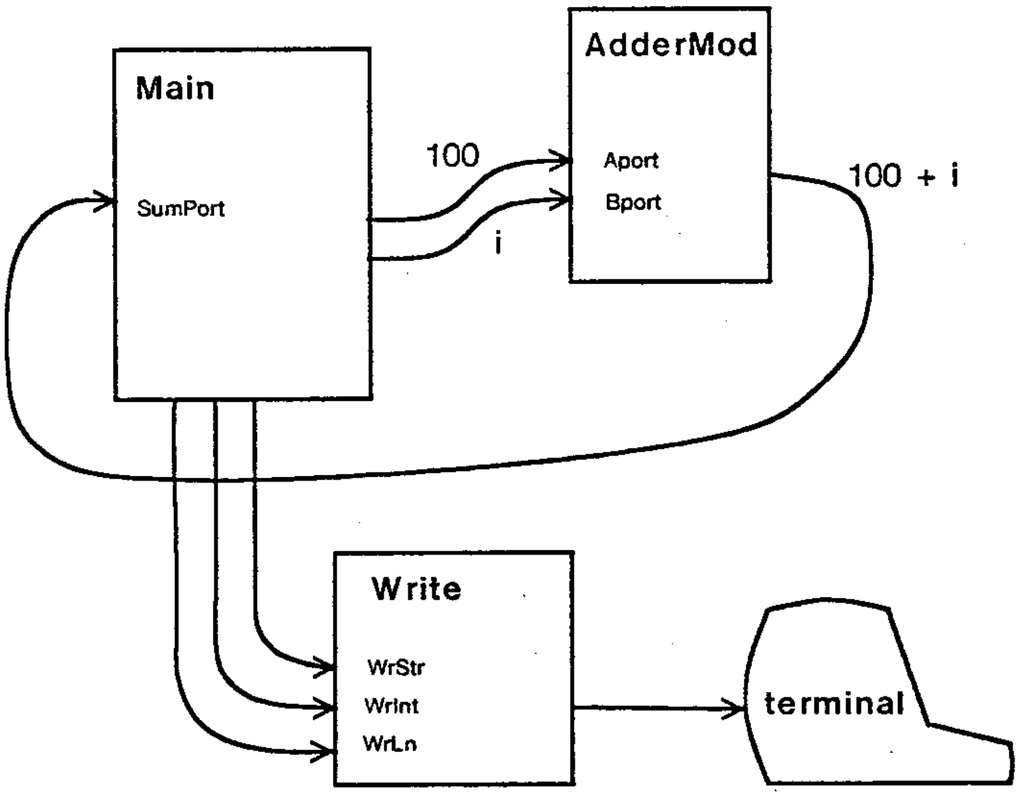


Figure 2: Run-time structure of the program presented in Figure 1.

1.2 Implementation

AMPL has been implemented on the Cm* multiprocessor [33] at Carnegie-Mellon University. The compiler is written in Bliss-36 and runs on a DECsystem20 computer. The compiler translates AMPL source programs into Bliss-11. After compiling the Bliss-11 code, the program is linked with the AMPL run-time system (also written in Bliss-11) and loaded under the Medusa operating system on Cm*. The implementation is complete except for a few features which are not interesting from a research standpoint; for example, enumerated types are not implemented.

1.3 Organization

A discussion of some issues in message passing is presented in Section 2. In this section, the semantics of various proposed and existing languages are explained and compared to the design of message passing in AMPL. Although communication mechanisms involve the most important design decisions, a number of other aspects of the design are important. These remaining issues are discussed in Section 3. In Section 4, the syntax and semantics of AMPL are presented. This section also points out the minor differences between the AMPL design, and its implementation on Cm*. The run-time system is as large as the compiler and considerably more complex, owing to the parallelism it supports and the functionality provided. Section 5 describes the design and implementation of the AMPL run-time system. The compiler is fairly straightforward except for the fact that it generates Bliss-11 rather than object code. A short discussion of the compiler appears in Section 5.7. A number of programs have been written in AMPL to exercise the implementation and to collect performance data. The objectives, instrumentation, experiments, and results are presented in Section 6.

2 Message-passing schemes

Many proposals have been made for language constructs which provide synchronization and communication in parallel programs [2, 5, 7, 8, 12, 15, 18, 19, 30]. Message-passing is an attractive basis for a parallel language because it combines the two notions of synchronization and communication. Since some sort of explicit synchronization is required whenever interprocess communication takes place, it is difficult to overlook a transaction that requires synchronization. If messages are buffered automatically, it is easy to construct programs which are synchronized by data-flow rather than control-flow.

The design space for a message-passing system is large. Some of the issues addressed in designing AMPL are stated below. Further discussions can be found in the papers referenced above. Comparative studies by Bloom [4] and Gentleman [14] are also relevant. Bloom develops a methodology for comparing synchronization schemes and applies it to monitors, serializers, and path expressions. Gentleman discusses many issues of message passing, with an emphasis on message passing in the THOTH operating system.

2.1 Ports

In AMPL, messages are sent from a sender to a port which belongs to a receiver. Only one receiver can receive messages from a given port, and that right cannot be transferred. This decision eliminates a level of indirection (mapping from ports to receivers). A port with "transferable ownership" can be simulated with an AMPL process.

A design with output ports as well as input ports was also considered. Output ports could be dynamically *connected* to input ports, and send statements would specify only the output port. This approach has the advantage that a structure of interconnected processes can be constructed without including knowledge of the structure in the process declarations. Instead, the interconnection can be performed by a process external to the structure.

This strategy has some nice properties, but after writing some small programs, it was found that ports would be reconnected frequently, leading to confusing programs. In practice, parameters and variables can be used to simulate output ports in many cases.

2.2 Synchronization

The expression of synchronization constraints is an important concern. In AMPL, as with many message-passing systems, all synchronization is accomplished by sending and receiving messages. Bloom [4] categorizes synchronization constraints as either exclusion or priority constraints, and notes that these constraints are expressed in terms of several types of information, for example, the operation requested and the local state of the receiver of the message. Another view of synchronization is in terms of sorting.

2.2.1 Sorting

Synchronization implies that some messages will be delayed while others are accepted; that is, messages to a process are acted upon according to some ordering. Determining the order in which messages are received is a form of sorting. In AMPL, two mechanisms are available for sorting messages at the receiver. First, messages are sent to ports associated with the receiving process. Normally, a separate port is associated with each operation so that messages will arrive sorted by the requested operation. Second, messages are queued in chronological order. While this may not seem like sorting at all, it is actually quite a useful property. It simplifies the task of fairly servicing requests. Furthermore, if messages are received in the order in which they are *sent*, the sender can sometimes avoid waiting for replies, thus increasing parallelism and decreasing message traffic. We will have more to say about this in Section 2.3.

2.2.2 Selecting messages

Sorting messages into ports does not fully specify the order in which messages are received; additional constraints may be applied. Selection of a port from which to receive a message can be accomplished in two ways. Conventional control-flow statements (if-then-else, while-do, etc.) can be used to select a port and an **accept** can then be performed on that port. An **accept** will delay the process if no message is present.

Now consider the case where the programmer wants to accept the first message to arrive at either of several ports. One solution is to poll the status of the ports, but this can be very inefficient. The problem is solved with a nondeterministic **select** statement. This statement lists a number of *alternatives*, each of which specifies one or more ports and enabling conditions which must be met in order to accept a message from that port. The enabling conditions only need to be reevaluated when a message is received (the time at which conditions may change). Otherwise, the process is suspended.

There are a few more details pertaining to **select** statements in AMPL. Since AMPL has no shared variables, enabling conditions can only change when a message arrives. The enabling conditions may involve tests of whether a port is empty (no messages waiting) or ready (at least one message waiting). There is no primitive corresponding to the COUNT attribute of Ada which would tell how many messages are waiting in a queue. Therefore, conditions are only reevaluated when a message arrives at an empty port. Further optimization can be performed to minimize the number of reevaluations by marking ports on which the enabling conditions depend. The process reevaluates conditions only when a new message arrives at one of the marked ports, and the port is empty. (These implementation concerns are in one sense irrelevant to language design; however, performance can be greatly affected by small changes in the definition of the language. We feel it is important to assess the performance cost of various design decisions.)

In AMPL, **select** statements are fair. For a given **select** statement, preference is given to the least recently selected alternative whenever several alternatives are enabled. This decision was made so that we would have the opportunity to study its effect on the implementation and program behavior.

Another common form of synchronization arises when a program must wait for a number of operands to become available. Both **select** and **accept** statements allow the programmer to specify a list of ports from which to accept messages, rather than just one. This is useful in writing data-flow programs where multiple arguments to a function arrive as messages from several sources. In the case of **accept** statements, this notation is essentially equivalent to a sequence of accept statements. In the case of **select** statements, however, if several ports are listed in an alternative, then the alternative is not enabled until all ports have waiting messages.

2.2.3 Sorting and selection in other languages

Monitors [18] provide priority queues which allow sorting based on values other than time. Priority queues are not easily simulated in languages without them, but no priority mechanism is included in AMPL because the additional language constructs and implementation effort did not seem to justify a feature that would rarely be used.

IPC [30, 31] allows messages to be "previewed" before receiving them. The AMPL programmer can always accept the message into a holding variable for examination before performing further operations. The preview mechanism is more useful in IPC where type information is sometimes desired before the data is received, but in AMPL messages are strongly typed (discussed below) so this information is already known.

Ada [12] provides mechanisms similar to those of AMPL, but the select statement of Ada evaluates enabling conditions only once at the beginning of the statement's execution. This is certainly more efficient for some applications, but is not as general as the reevaluation scheme of AMPL. Is the added power worthwhile? None of the examples coded in AMPL for performance measurements required this generality. On the other hand, efficient evaluate-once code could be generated for all of our examples as an optimization. Determining that this optimization is possible is a simple task for the compiler. Therefore, while the AMPL scheme may be used only rarely, it can be provided at little or no cost in performance.

Ada does not define the choice of alternatives in select statements when several alternatives are enabled. CSP defines the choice as random or nondeterministic. AMPL's "fair" policy is more difficult to implement, but provides more safety to the programmer. The policy of all three languages can always be modified by placing more constraints in the enabling conditions of each alternative.

In Ada, it is difficult to wait for and receive a collection of arguments for a function from different sources. The Preliminary Ada Reference Manual [20] states "To wait for several events to have *happened* merely requires a sequence of accept statements." Unfortunately, this strategy cannot be used to accept arguments in the order in which they become available, and the processes supplying the arguments will not resume execution without delay. Since Ada does not provide buffering, this sort of synchronization is awkward. *MOD [8] provides buffered ports, but there is no equivalent of a *select* construct to allow conditional waiting on multiple ports as in Ada and AMPL.

2.3 Send semantics

Another set of issues relates to the question of buffering messages. AMPL buffers messages so that a sender need not block waiting on the receiver. The primary motivation for buffering is to increase parallelism. Even if a reply is required, the sender can often do useful work before getting the reply. Another reason for the inclusion of buffering is that it is used frequently. While buffering could be implemented at the language level, it would not be as efficient as an implementation at the system level.

AMPL **send** statements are defined so that a message is delivered to a port before the sender continues. Delivery of a message means putting the message in a chronologically ordered queue. The sender does *not* wait for the receiver to remove the message from the queue. There are two reasons for this delivery wait. First, since messages are ordered by *arrival* times, this wait is necessary to order messages by *send* times. The following property is obtained:

If a send to a port completes before another begins, the first message sent will be the first one received.

In the case where the sends are from the same process, more efficient protocols can maintain a chronological ordering, but consider the case illustrated in Figure 3. Suppose process A wants to deliver a message to port P after which Process B will deliver a message, and these messages must arrive in order. Suppose further that **send** statements do not wait for delivery. Now, after Process A sends to port P (message 1 in the figure), A instructs Process B to send a message by sending message 2. B responds by sending message 3. But wait! Message 1 may not have arrived at Port P since Process A never waited for its delivery. In AMPL, Process A cannot proceed until message 1 is delivered, so no race condition can arise. This sort of synchronization is used only occasionally in AMPL programs, but failure to wait for delivery could result in unreliable and very mysterious program behavior. Waiting for delivery also facilitates flow-control. If the buffer at the receiver is full, the message can be discarded since the sender is suspended with a copy of the message. A complete description of the message-passing implementation is given in Section 5.4. This section also discusses an optimization to avoid delivery waiting while still providing the same synchronization and flow-control capabilities. In any case, a process can have only one undelivered message outstanding, so this decision can potentially reduce parallelism by blocking processes unnecessarily.

Ada and CSP do not have any visible buffering of messages, so the synchronization and flow-control problems do not arise. With IPC, reply messages would be required to avoid the synchronization problem described above. Flow-control is provided by a more elaborate mechanism designed for greater efficiency in a network environment.

2.4 Naming

Designs vary greatly in the manner in which sources and destination for messages are specified. At one extreme is CSP in which both sources and destinations are statically defined in the source code. At the other extreme are designs like ITP [34] which is an attempt to allow the programmer great freedom in specifying (or leaving unspecified) both sources and destinations of messages. AMPL takes an intermediate position. Since processes can be dynamically created, names must also be created dynamically. Initially, only the process name is created and available to the creator. Port

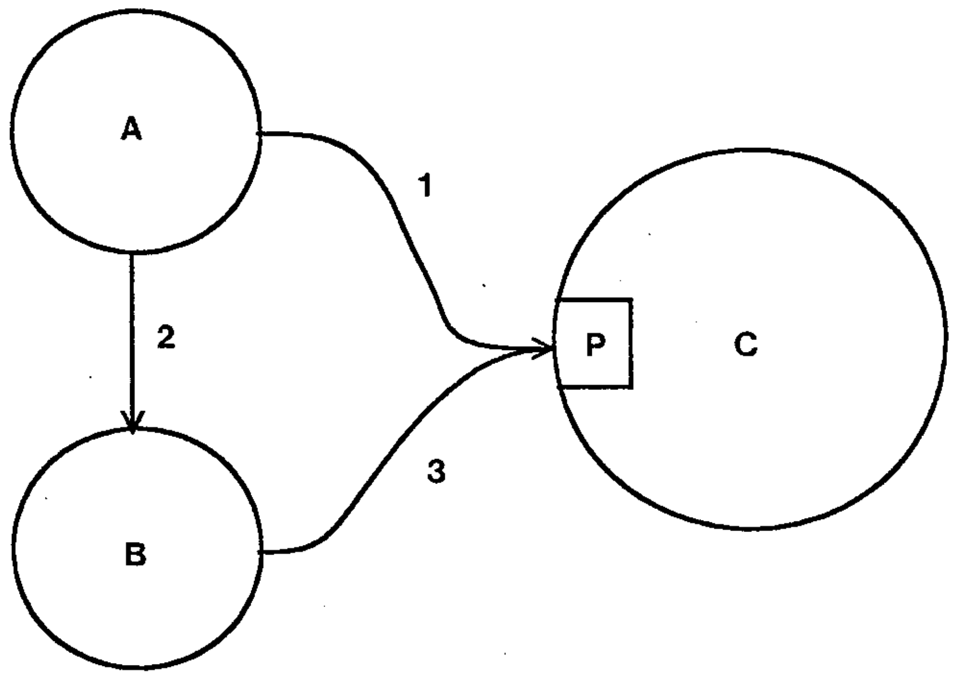


Figure 3: Two processes sending ordered messages to a single port.

names may be generated from process names as follows. To reference port p of process A, the programmer writes A.p. Process names and port names are full-fledged types and may be assigned to other variables, transmitted in messages, etc. The implementation and representation of port and process names are discussed in Section 5.3; the interaction of naming with the implementation of abstract types is discussed in Section 3.3.

A receiver names only the port in an **accept** statement; there is no choice of sender. On the other hand, the sender must know the name of the receiver. Port names may be thought of as capabilities to perform *send* operations on port objects. It was desired to keep AMPL simple while permitting the dynamic creation of computing structures or networks. The simplicity requirement is met by having only one mechanism for generating names, and one way to specify the sender and receiver of a message. Broadcasting messages, receiving on the basis of a senders identity, or receiving an array of messages from an array of ports are not primitive operations in AMPL.

Ada illustrates a slightly more elaborate naming scheme. Processes (Ada tasks) can be statically declared or dynamically created. The statically declared tasks can be named directly, whereas dynamically created tasks are named by using access variables. Ada also provides arrays of entries (an Ada entry is analogous to an AMPL port.) In preliminary Ada, and in CSP, all processes are created statically.

Manipulating names in AMPL is like the use of pointer variables in several ways. Pointers are similarly general, allowing the creation of arbitrary structures. Pointers are also difficult to use, and are typically used only when other primitive structures (e.g. arrays and records) are inadequate. Usually, the programmer uses pointers to create a regular structure such as a linked list, a tree, or a graph with special properties, but current programming languages do not have adequate means of expressing these structures. The best that can be done is to give the programmer a set of general primitives so that the desired structure can be constructed. Many AMPL programs also have simple regular structures, but we do not know how to concisely express these structures in a general way. Therefore, AMPL programs are free to pass and copy names without restraint. In practice, this over-generality has resulted in almost no programming errors; however, we must note that only small programs with well thought-out structures have been implemented.

2.5 Invoking operations

A common use of a message is to invoke a function to be performed by the receiver. This can be likened to a monitor call or an operation on an abstract data type. Several decisions can be made to simplify this use of message passing. First, a process can have many ports, where each port corresponds to a particular operation. Thus, the operation does not need to be decoded by examining the message's content. In AMPL, ports are associated with instances of processes. That is, when a process is created, a set of ports are also created for the process. Only that process can receive messages from these ports, but any process with a reference to a port can send to it.

Often, messages are used to invoke operations which produce a result. Several languages have built-in mechanisms for returning results to the sender of the original message. There are many ways to facilitate the programming of reply messages, and it was not clear which if any to choose when AMPL was designed. It was decided to provide the necessary primitives, and not to provide built-in facilities for sending replies. As a result, programs are not biased toward a particular construct, and programs can be analyzed for common sequences of primitives that might indicate desirable properties of a built-in reply facility. The ability to send a port reference in a message is used frequently to specify a reply port for the result.

In IPC, every message contains a standard field for specifying a reply port. IPC has not yet been integrated with any languages to the extent that a single construct can invoke an operation and obtain a result; however, functions can be written to hide the necessary operations from the programmer.

An example of a language with built-in reply mechanisms is *MOD. In *MOD, the same syntax is used to invoke procedures, create processes, and send messages. All of these actions can optionally return a value, in which case the invoker waits for the results. A limitation of *MOD is that in cases where a result is required, in response to a message, the result must be generated and returned from within the same syntactic construct that receives the message. The receiver cannot, for example, store the message away and come back to it later, or request another process to return a result.

In Ada, a *rendevous* may return a result by modifying *var* parameters. This does not allow entry calls to be made directly from within expressions, but a function can be defined that performs the entry call and returns a result.

Distributed Processes (DP) [15] bases all process communication on a construct similar to Ada's entry call. The caller is always suspended until the call completes. CSP has no mechanisms for sending a result, although naming in communication constructs is symmetric and static, so there is never a problem of determining where to send a reply.

A reply facility would greatly enhance the readability of some AMPL programs. Section 5.4 illustrates how replies could lead to greater efficiency at run-time by reducing the number of messages.

3 Other language design issues

3.1 Types and storage management

AMPL is a strongly-typed language. In particular, ports are typed, and messages must have types that match those of their destination ports. Safe garbage collection in AMPL is made possible by strong typing. Typing of messages sometimes interacts undesirably with the sorting mechanisms described in Section 2.2.1. To illustrate the problem, refer to Figure 3 and supppose that message 3 sent by process B has a different type from that of message 1, but process C, the receiver, must receive both messages in order. For example, message 1 may be the last data message of a stream and message 2 may be an end-of-stream message. Since the messages have different types, two ports must be used, say P1 and P2 (see Figure 4). With the mechanisms presented so far, there is no way to guarantee that messages are received in order unless extra synchronization messages are sent. The problem is that messages are sorted first by ports and then by arrival time. We want to receive messages from P1 and P2 sorted first by arrival time and then by ports. This alternate sorting order can be indicated in port declarations. The syntax will be explained in Section 4.6. This technique is illustrated in Section 6.9. An alternative solution is to provide union-types, so that the data message (message 1) and end-of-stream message (message 2) could be sent to the same port. In a larger language, this would probably be the best choice, since union-types have other uses and are therefore more general.

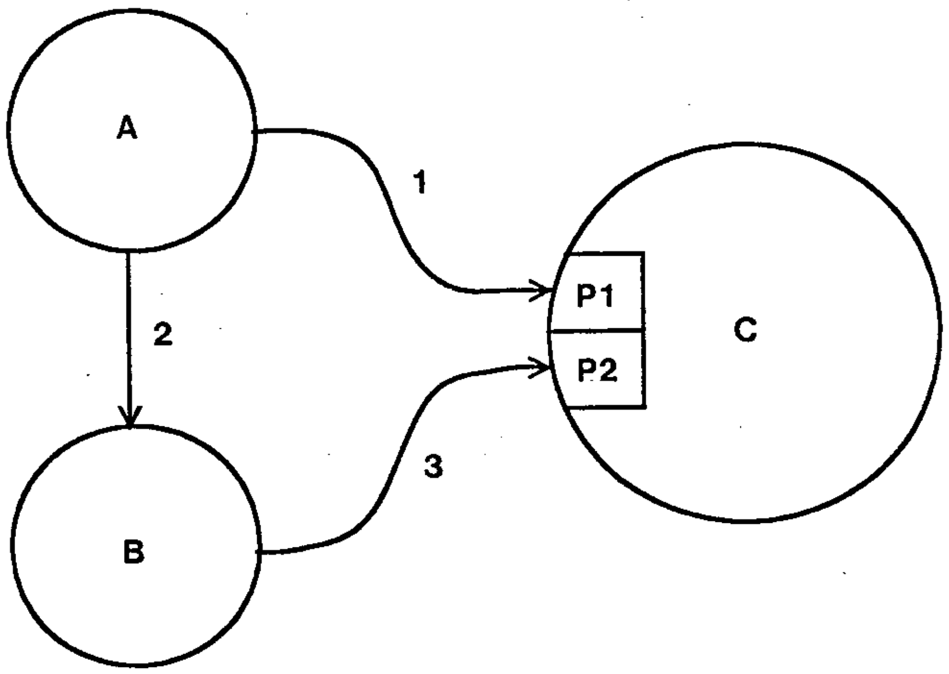


Figure 4: Synchronization of messages arriving at two ports.

Several aspects of typing in the AMPL design simplify the storage-allocation problem. The typing system is restricted so that the size of all objects are known by the compiler. AMPL does not provide procedures (although procedures may be simulated with other mechanisms). Storage allocation is discussed further in Section 5.7.1. The storage-allocation strategies in turn affect the design of garbage collection.

The absence of procedures allows the maximum stack size of a process to be determined by the compiler. The maximum size for a given process is bounded and depends only on static properties of the code. Heap storage is required only for processes, since variables cannot be allocated dynamically and there are no "pointer" types in AMPL.

A more elaborate and flexible set of types as illustrated by modern programming languages (e.g. Ada) is desirable for many applications. The typing system of AMPL was chosen primarily to keep the implementation simple; it provides enough power to allow interesting parallel programs to be written without overly complicating the implementation.

An example of an implementation which has devoted much more attention to the storage-allocation problem is Mesa [25]. Although Mesa is implemented on a uniprocessor, Mesa programs can spawn processes and coroutines which require efficient allocation of heap objects. In fact, all procedure activation records are allocated from heap storage; thus, stacks can be of arbitrary size, and storage is only allocated as needed.

3.2 Creating processes

Processes are created dynamically in AMPL. A built-in function, *create*, is called to create a process and pass parameters to it. The value of the function is a reference to the created process. As discussed in Section 2.5, AMPL has no built-in mechanisms for obtaining replies, so a functional syntax was chosen for the *create* primitive.

An alternative syntax, which is more consistent with AMPL message-passing facilities would be to use a language-defined port called *create*. Messages would be sent to the *create* port to specify the process to be created, the parameters for the process, and a *reply* port to which a reference to the created process would be sent. This alternative approach would allow the creator to continue executing in parallel with the creation of the new process, but the overall time to create a process would be slightly longer.

3.3 Encapsulation

Data abstraction is an important element in many current languages. Ordinarily, abstract type representations are hidden or encapsulated through the use of scope rules. Access to the type is then provided through procedures which are declared to be visible to users of the type. AMPL processes may be viewed as implementations of abstract types. The variables in a process are "hidden" objects since thay are not shared. On the other hand, the ports belonging to a process can be referenced by other processes. Abstract operations are invoked by messages to the corresponding port. The representation of the type is encapsulated by the process.

Abstract types which use many processes can also be constructed. The simplest way to do this is to define a top-level "interface process" to serve as an interface between users of the type and the type implementation. The interface process, when created, in turn creates *representation* processes which together form the representation of the type. If the interface process does not distribute references to the representation processes, then these processes will be hidden from the users of the abstract type. Thus, encapsulation is performed operationally rather than through the use of special scope rules.

In more elaborate implementations of abstract types, it becomes necessary to control access on a more detailed level. The operations visible to a user can be restricted by providing the user with names of only a subset of the ports of a process. (While possession of a process name implies the ability to name all ports, possession of a port reference does *not* allow the owner to name the corresponding process.) Figure 5 illustrates this concept. The abstract type is a data-base, implemented with many interconnected processes. A single *manager* process is originally created to instantiate the data-base and grant access to it. The manager has only one port, used to gain access to the data-base. Users who want to use the data-base send a request to the manager, who responds by creating a *connection* process and returning references to a subset of the connection process's ports. Messages from the users to these ports cause reads and writes on the data-base. Notice that the user has no access to the data-base except through his connection process, and only ports made available by the manager are accessible. The manager is not a bottle-neck for data-base accesses, since messages after the initial one are sent to connection processes.

3.4 Processor allocation

AMPL has no facility for specifying physical process locations. Processor allocation is performed by the run-time system. Because processes do not share memory and interact via messages, it is particularly easy to move processes from one processor to another. Thus, allocation can be varied dynamically. Where there are large numbers of processors, it is difficult to know how and where to locate processes, especially in programs which create processes dynamically. The philosophy behind the design of AMPL is that programs should specify a high degree of parallelism, with far more processes than processors. This helps insure that the available parallelism will be utilized even though the allocation of processors is less than optimal. The absence of shared memory means that the set of variables used by a process is always known, so it is easy to arrange to keep variables physically near the processor which accesses them. Very little research has been done relating to processor allocation in this sort of system. The AMPL implementation has hooks to allow process migration, but none is currently performed.

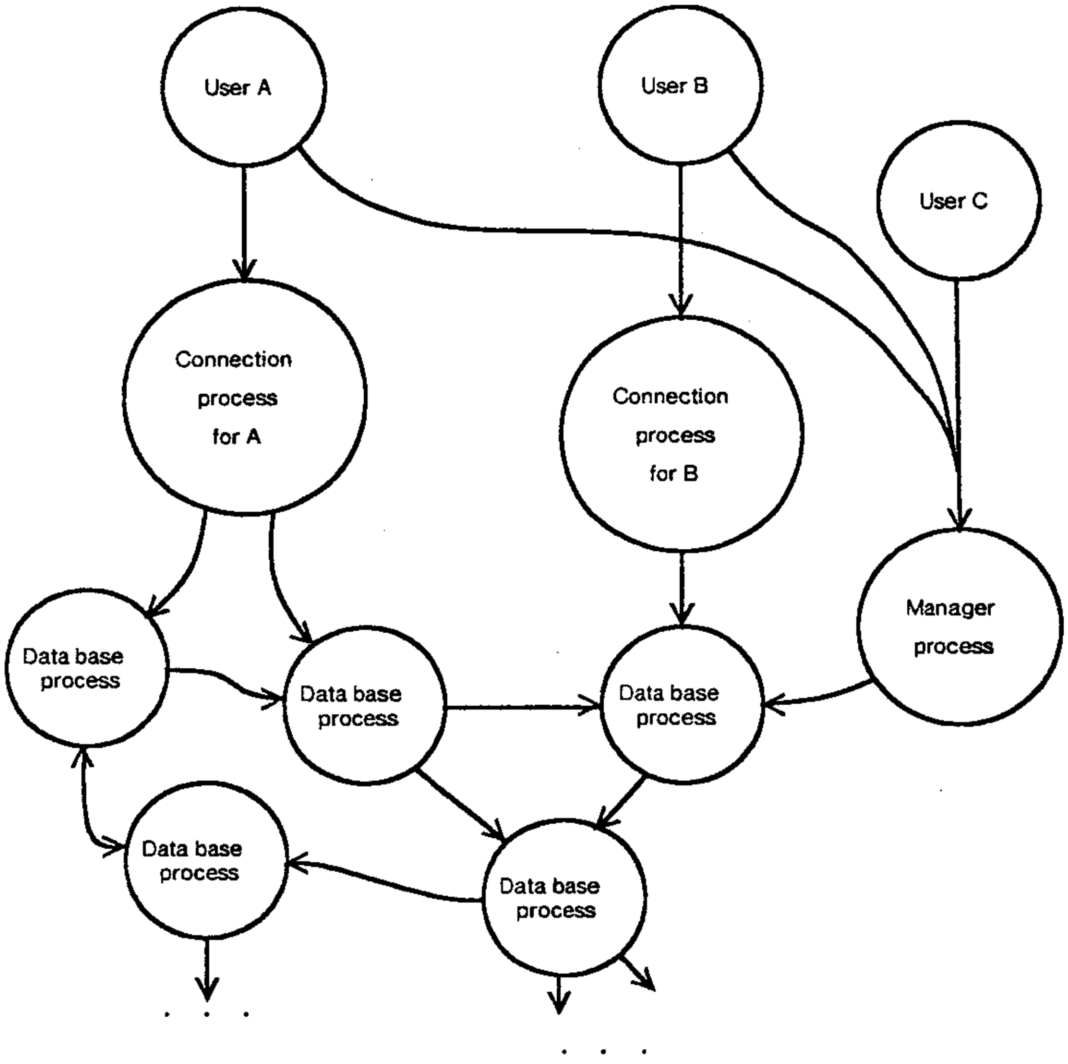


Figure 5: Process structure for a parallel data-base implementation.

4 Language definition

In this section, the syntax of AMPL is defined with a BNF-like notation. The semantics are defined informally. The syntax definition is based on the machine-readable description which is fed to an automatic parser generator (FEG) [13]. Metasymbols used in the description are:

{ } < > ::= ? * ; , |

Nonterminals are denoted by an arbitrary number of characters enclosed in angle brackets, e.g. <x>. The meta-language constructs used are:

{ <x> | <y> } . . .	either <x> or <y>
{ <x> }?	an optional <x>
{<x> }+	one or more occurrences of <x>
{ <x> }*	zero or more occurrences of <x>
{ <x> }; + ...	one or more occurrences of <x>, separated by semicolons
{ <x> };*	zero or more occurrences of <x>, separated by semicolons
{ <x> }, + ...	one or more occurrences of <x>, separated by commas
{ <x> },*	zero or more occurrences of <x>, separated by commas

Double quotes are used to indicate that a meta-language symbol is to be treated as a language symbol, e.g. "}". The remainder of the meta-language should be obvious to anyone familiar with BNF.

Throughout this section, a number of details about the language and its implementation are printed in small text like this. The reader may wish to skip these details.

4.1 Lexical rules

```
<letter> ::= A | B | C | ... | Z | a | b | c | ... | z

<digit> ::= 0 | 1 | 2 | 3 | ... | 9

<stringchar> ::= " " | ! | """ | # | $ | % | & |  (  |  )  | * |
                  +  | , |  -  | . | / | : | ; | "<" | ">" | ? |
                  @  | [ |  \  | ] | ↑ | _ | ' | "{" | "|" | "}" |
                  ~  | <digit> | <letter>

<identifier> ::= <letter> { <letter> | <digit> }*

<integer> ::= { <digit> }+

<string> ::= ' { <stringchar> | '' }* '
```

The lexical rules are similar to those of Pascal [36], except real numbers are not allowed. A stringchar is a blank or any printing ASCII character other than the single quote character. To simplify the implementation, the language has no facility for constructing strings with non-printing characters (other than blanks). Within a string, a single quote is indicated by two adjacent single quote characters. Comments are arbitrary strings enclosed in braces or the symbols "(*" and "*)". The following are reserved words in AMPL:

accept	**and**	**array**
begin	**const**	**div**
do	**else**	**end**
if	**mod**	**module**
not	**of**	**or**
port	**record**	**refmod**
refport	**select**	**send**
then	**to**	**type**
var	**when**	**while**

In this report, keywords will always be printed in **boldface.**

4.1.1 Examples

X
Master
Abc123xyz
10
4096
{ this is a comment }
(* this is a comment *)

4.2 Programs

```
<program> ::= { <constant declarations> }?
              { <type declarations> }?
              <module declarations>

<module declarations> ::= { <module definition> };+
```

A program declares global constants and types, and a collection of modules. Notice that it is not possible to declare variables at this level. A module is actually a special type, and serves as a template or a type declaration for a *process.* Since modules are not processes, there must be a way of instantiating them. All processes must be explicitly created at run-time with two exceptions: (1) system defined modules for I/O are created automatically, and (2) an instance of the program-defined module MAIN is created automatically.

A program must have a module called main. Omission of this module will not result in an error message from the compiler, but will cause the linker to complain.

Figure 1 on page 2 is a short example of a complete program.

4.3 Modules

The module declaration is the primary unit of program structure in AMPL. It defines constants, types, ports, and variables associated with the module. A process, or module instantiation, contains a collection of ports, storage for variables, a stack, and buffers for messages delivered to the ports.

```
<module definition> ::= module <identifier>
                            { <parameter list> }? ;
                              <block>
```

A module definition includes an identifier, a parameter list, and a block. The identifier is a name for the module and has a global scope. This identifier is used as an argument to the *create* function and is also used in **refmod** type declarations. The optional parameter list implicitly declares variables. These variables are initialized to actual values passed by the *create* function call to be described later. The block is defined as follows:

```
<block> ::= { <constant declarations> }?
            { <type declarations> }?
            { <port declarations> }?
            { <variable declarations> }?
            begin { { <statement> }? };* end
```

Figure 1 has several examples of module definitions.

4.4 Constants

Constants are defined as in Pascal:

```
<constant declarations> ::= const { <constant definition> }+

<constant definition> ::= <identifier> = <constant> ;

<constant> ::= { { + | - }? { <identifier> | <integer> } | <string> }
```

The scope of constant identifiers is global if declared outside of a module, and local to the module if declared inside the module. Forward references in constant definitions are allowed but are of limited use since a constant cannot be an expression.

4.4.1 Examples

```
const
   Nslaves = 4;
   text = 'Hi there';
   ArraySize = Nslaves;
```

4.5 Types

AMPL has a limited number of types. As in Modula, there are no types which must be dynamically allocated within module instances. Unlike Modula and Pascal, types need not be declared in any particular order. Forward references and recursive types are handled properly.

```
    <type declarations> ::= type { <type definition> }+

    <type definition> ::= <identifier> = <type> ;

    <type> ::= { <simple type> | <array type> |
                 <record type> | <ref type> }
```

Types are equivalent if they are structurally the same. For arrays, the number of components must be identical and the component types must be equivalent. For records, the types of corresponding fields must be equivalent and the number of fields must match. Subrange types are all equivalent to type *Integer*, but the implementor may assume that the value of a variable of a subrange type is bounded by the constants specified in the type declaration. This assumption may be checked at run-time. Reference types are equivalent if the referenced types are equivalent. No two modules are considered equivalent; this is the only case where structural equivalence is not applied.

In the current AMPL compiler, structural equivalence is not implemented. It was decided that checking cyclic structures was not worth the effort, given the experimental nature of the project. Straightforward name equivalence was rejected because it requires more types to be declared by the user. Instead, a "relaxed" name equivalence is used. The rules for equivalence are carefully defined to handle most cases of structurally equivalent types without the need to handle cyclic types. The author does not recommend this scheme for any language; however, rules are included here for users of the current compiler. Types are equivalent if

1. The types have the same name.
2. The types are array types and have the same size, and the component types are equivalent.
3. The types are either both refmod or both refport types, and the referenced types are equivalent.
4. The types are subrange types. (Integers are considered a subrange type.)

A type declaration of the form "Type X = Y" where Y is a type identifier does *not* create a new type, that is, X and Y are equivalent.

4.5.1 Simple types

```
    <simple type> ::= { <constant> .. <constant> | <identifier> }
```

Integer and *Boolean* are predefined types. *True* and *False* are predefined constants of type *Boolean*.

Currently, subrange types are treated as integers and no range checks are performed at run-time except for array indexes.

4.5.2 Structure types

```
<array type> ::= array [ <simple type> ] of <type>

<record type> ::= record <field list> end

<field list> ::= { <field> };+

<field> ::= <identifier list> : <type>

<identifier list> ::= { <identifier> },+
```

Arrays are single dimensional, but multi-dimensional arrays can be simulated easily using arrays of arrays.

In the current implementation, record field identifiers must be unique across all record field identifiers in the same scope. The compiler distinguishes the names internally, but uses the field names to generate Bliss-11 macros in the generated output. If two records have identical field names, the Bliss-11 compiler will complain that two macros have the same name.

4.5.3 Reference types

```
<ref type> ::= { refport | refmod } <type>
```

A reference type can only refer to a module or a port, and variables of this type are initialized to a special constant *nil*. Refmod values are only created by the function *create*, to be described in Section 5.5. A refmod value is essentially a pointer to a process. A refport value is created by applying a port name as a "field selector" to a refmod variable. For example, if M is a module with port p, and V is a variable of type **refmod** M, then V.p is an expression yielding a **refport** type. (The exact type will be **refport** U, where U is the type of port p.)

For refmod types, the type specified must be the identifier of a module.

4.5.4 Examples

```
type
   RefMaster = refmod Master;
   RefIntPort = refport integer;
   SlavesType = array [1 .. Nslaves] of refmod Slave;
   AnswerType = record N, M: integer;
                 Who: refmod Slave
                 end;
```

4.6 Ports

```
<port declarations> ::= port { <port definition> }+

<port definition> ::= { <port specification> ; |
                        ( { <port specification> };+ ) ; }

<port specification> ::=
                  <identifier> : <type> { ( <constant> ) }?
```

Port declarations define ports through which messages are received. All identifiers in port specifications are known globally so that, given a reference to a module, all ports of that module can also be referenced. All other identifiers declared inside a module are local to that module. The first form of port definition (a port specification followed by a semicolon) declares a port and optionally gives a FIFO message queue length. The constant in parentheses specifies the number of messages which will be buffered before senders are blocked. A reasonable default value is computed if the constant is omitted.

The second form is a list of port specifications in parentheses. Messages sent to any of the listed ports will always be received in the order of their arrival times. All messages in the list logically share a single queue. (The implementor is free to use separate queues and timestamps, etc.) The size of the queue is the maximum of any constants given in the port specifications. If no constants are specified, a reasonable default value is computed. The use of this second form of specification is explained in Section 3.1.

Declaration of a port within a module indicates that the port is to be instantiated whenever the module itself is instantiated. Therefore, if the module is instantiated twice, then separate ports are created for each process. Processes can only accept messages from their own ports. Any process with a reference to a port can send messages to it.

4.6.1 Examples

port
  P: integer(3);
  NamePort: **refmod** *Slave;*
  (data: integer (Nslaves); EndOfStream: boolean);

4.7 Variable declarations

```
<variable declarations> ::= var { <variable definition> }+

<variable definition> ::= <identifier list> : <type> ;
```

Variable declarations are similar to those of Pascal. Like ports, variables are instantiated whenever the module itself is instantiated. The variables are accessible only by the process which contains them. No variables can be shared by multiple processes.

4.7.1 Examples

```
var
  names: SlavesType;
  I, J: integer;
  owner: refmod Master;
```

4.8 Variable references

```
<variable> ::= { <identifier> | <variable> [ <expression> ] |
                 <variable> . <identifier> }
```

The syntax for referencing a variable is like that of Pascal. The field selection (dot) notation also serves to reference a port within a module as explained in Section 4.5.3. Every process has an implicitly declared variable called *self* which is initialized with a reference to the process itself. In a process which is an instantiation of module M, the type of *self* is **refmod** M. If module M declares a port p, then self.p refers to the name of port p in that process.

4.8.1 Examples

```
X
X.Who
A[I]
A[I][J][message.row]
self.ResultPort
```

4.9 Expressions

AMPL uses a standard expression syntax. The operators, listed in order of increasing precedence (with equal precedence operators on the same line) are:

```
               or
               and
               not
      <  <=  >=  =  >  <>
              +  -
          *  /  div  mod
          -  (unary minus)
```

Operators are defined as in Pascal, except comparison operators (< <= >= = > <>) are not defined for *Boolean* arguments. Also, AMPL does not support floating point numbers. No operators are defined on structures. The syntax for expressions is:

```
<expression> ::= { <conjunction> | <expression> or <conjunction> }

<conjunction> ::= { <negation> | <conjunction> and <negation> }

<negation> ::= { <comparison> | not <comparison> }

<comparison> ::= { <sum> | <sum> <compare op> <sum> }

<compare op> ::=  "<"  |  "<="  |  ">="  |  =  |  ">"  |  "<>"

<sum> ::= { <product> | <sum> <add op> <product> }

<add op> ::= { + | - }

<product> ::= { <factor> | <product> <mult op> <factor> }

<mult op> ::=  { * | div | mod }

<factor> ::= { <term> | - <term> }

<term> ::=  { <integer> | <string> | <identifier> <actual list> |
                    ( <expression> ) | <variable> }
```

The expression syntax contains a production for function calls. Functions cannot be defined by the user, but two system defined functions, *ready* and *create*, are available. These will be described below.

4.9.1 Examples

*3 * (-X + R.count)*
X + 3 > A[I] **or** *ready(inport)*

4.10 Conditional statement

```
<if statement> ::= if <expression> then { { <statement> }? };+
                   { else { { <statement> }? };+ }? end
```

The expression must be of type *Boolean*. Notice that the statement is terminated by the symbol **end**. Statement lists may follow the symbols **then** and **else**. Either or both statement lists may be empty. The entire **else** clause is optional.

4.10.1 Examples

```
if I = MaxCol then A[I] := B end
if flag then X := 1
  else if flag2 then X := 2; Y := 3 end
  end
```

4.11 Iteration statement

```
<while statement> ::= while <expression>
                      do { { <statement> }? };+ end
```

The expression must be of type *Boolean*. The iteration statement is also followed by the symbol **end**, to mark the end of the statement list following **do**.

4.11.1 Example

```
while I < len do
 A[I] := 0; I := I + 1 end
```

4.12 Send statement

```
<send statement> ::= send <expression> to <expression>
```

The send statement is used to send messages to ports. The first expression is evaluated to yield a value which becomes the content of the message. The second expression is evaluated to yield a port reference. If the first expression is of type T, then the second expression must be of type **refport** T. The message is sent to the port, and the sending process suspends until the message is delivered to the receiver, i.e., the message is copied into the destination port's message queue. This guarantees that if a message is sent to a port after the completion of a **send** to the same port, the messages will be received in the same order they are sent. (See also Sections 4.6, 2.3, and 5.4). It is an error to send to a null port descriptor or to a process which has halted. In the event that the destination port buffer is full, the sender remains suspended until the message can be delivered.

4.12.1 Examples

```
send I + 1 to B.IntPort
send A[I] to X
send pat to self.back
```

4.13 Accept statement

```
<accept statement> ::= accept <accept list>

<accept list> ::= { <accept item> },+

<accept item> ::= <identifier> ( <variable> )
```

An accept statement is the simplest statement that receives a message or set of messages. The identifiers in accept items must be port identifiers declared in the same module. Messages can only be received by the process in which the port is declared. For the case with one accept item, a message is removed from the indicated port's message queue and is assigned to the indicated variable. If the queue is empty, the module instance suspends until a message arrives.

In the case where more than one accept item is present in the accept list, each port identifier must be distinct. The process suspends until a message is present at each port's message queue. Then, one message is removed from each queue and assigned to each variable in the order accept items are listed. An accept statement with a list of accept items is identical to a sequence of accept statements with one accept item each, except with a sequence of accept statements, some messages may be accepted before all messages are present.

4.13.1 Examples

```
accept P(x)             {take message from P and store in x}
accept Index(i), Value(A[i])] {use two messages to update an array}
accept Index(i); accept Value(A[i])]
  {similar to previous example, but Index message may be accepted }
  { before Value message is available}
```

4.14 Select statement

```
<select statement> ::= select { <alternative> };+ end

<alternative> ::= { when <expression> }?
                        accept <accept list>
                        then { { <statement> }? };+ end
```

The select statement is a generalized non-deterministic form of the accept statement. It allows a process to act upon any of several message arrivals and to place constraints on which message is accepted next. One and only one of the select alternatives is executed. An alternative is said to be *enabled* when its conditional expression is true and a message is waiting in the queue for each port specified in the accept list. Again, ports in the accept list must be distinct.

If no alternatives are enabled, the process suspends until an alternative is enabled. If one alternative is enabled, messages are accepted as in the simple accept statement, and the corresponding list of statements is then executed. If more than one alternative is enabled, the one least recently executed is selected. This provides a fair choice in the sense that, if a given clause in a given alternative is enabled repeatedly when the select is executed, then the alternative will eventually be chosen. If no enabled alternative is the least recently executed, i.e. none have been executed at all, then the choice is made arbitrarily.

There is no restriction on the **when** expressions. However, since no variables are shared, the value of any expression cannot change unless a message arrives at an empty message queue.

The function *ready* takes a port identifier (not a port reference) as its argument and returns *true* if the port has a non-empty message queue. Otherwise *false* is returned. A process can only examine its own message queues. There is no direct way to determine the state of the ports of another process. The *ready* function is useful for indicating priority in select statements and in avoiding suspension when a message queue is empty.

4.14.1 Examples

The first select statement, when executed inside a loop, will implement mutual exclusion on a shared piece of data. The second statement grants requests to read or write to shared data. Writers have priority, and multiple readers are allowed:

```
select
  accept read(reader) then
    send data to reader end;
  accept write(data) then end
  end

select
  when ReadCount = 0 accept WriteRequest(writer) then
    send OK to writer;
    accept WriteDone(writer)
    end;
  when not ready(WriteRequest) accept ReadRequest(reader) then
    send OK to reader;
    ReadCount := ReadCount + 1
    end;
  accept ReadDone(reader) then
    ReadCount := ReadCount - 1
    end
  end
```

4.15 Call statement

```
<call statement> ::= <identifier> { <actual list> }?

<actual list> ::= ( { <expression> },+ )
```

The only built-in AMPL procedure is *create*, which takes a module name as its first parameter, and module actual parameters in following parameter positions. The types of the actuals must match the types of the corresponding module formal parameters. The values of the actuals are assigned to formal parameters when the module is created. Otherwise, the formal parameters are regarded as ordinary variables. The process executing the create is suspended until the creation is complete. The created process executes the statement list in the block given in the module definition. The process terminates after the last statement of the block is executed. A process also terminates if it is suspended and no other process has a reference to it. Such a process will never be resumed. *Create* may be called as a function in which case a refmod value is returned.

In the current implementation calls to create may not be nested.

The programmer cannot define procedures in AMPL, but procedure calls can be simulated by creating a module instance and waiting for a reply message containing results.

4.15.1 Examples

mref := create(M, 1, y)
create(M, 1, y)

5 Implementation

AMPL is implemented on the Cm* multiprocessor. Although AMPL exploits many features of Cm*, the language was not designed specifically for this machine. Cm* consists of about 50 computer modules in five clusters. Each computer module consists of an LSI-11 processor, a local memory, and a switch to couple the processor to its local memory. The switch can be programmed to direct certain memory access requests to a high-speed mapping processor called a Kmap. The switch also allows the Kmap to access the local memory. A Kmap is associated with each cluster, and Kmaps communicate with each other over high-speed links. The resulting configuration allows processors to access any memory location in the machine, but access times vary by an order of magnitude depending on whether access is a local, an intracluster, or an intercluster memory reference.

The Kmaps contain high-speed horizontally microprogrammed processors which could best be used as message processors in an AMPL implementation. Unfortunately, the programming effort required would be very large, and the new microcode would be too large to coexist with either of the available operating systems. For these reasons, it was decided to implement AMPL on an existing operating system, Medusa [28].

While Cm* hardware is almost ideal for AMPL, the process and message-passing abstractions provided by Medusa are poorly matched to our requirements. As a consequence, the AMPL run-time system implements its own processes, ports, scheduling, and storage allocation. Medusa processes exist in separate address spaces and the smallest grain of protection provided by hardware is a 4096-byte page. The smallest process in Medusa is at least this large. At most 16 processes can be created in any computer module and designers of Medusa intended that processes be rather static entities. It was decided to use Bliss-11 coroutines to implement AMPL processes. This allows processes to share an address space, and allows the run-time system to choose a process representation that is optimized for AMPL processes. Since several AMPL processes can share a single address space, there is no lower bound on process size due to hardware restrictions.

5.1 Ports

Medusa has a fairly elaborate message-passing system. At first sight, Medusa seems like an ideal base for AMPL because it emphasizes the use of message-based communication over shared memory. Upon closer inspection, however, it is found that Medusa *pipes* are only remotely similar to AMPL ports. First, pipe creation is a time-consuming operation due to protection, addressing, and memory management problems handled by the operating system. One way to get around this problem is to pre-allocate a large number of pipes in a commonly accessible place (Medusa's *shared descriptor list*) and assign pipes to processes dynamically. Pipes would be reused rather than destroyed. The shared descriptor-list allows a maximum of only 512 pipes, which would be too small for many programs. Another problem with pipes is that if a pipe is full when a message is sent, the sender is blocked. Assuming that AMPL processes are implemented as coroutines, what we want to happen is the suspension of one coroutine, not the suspension of the Medusa process which is responsible for executing many coroutines (AMPL processes). For these reasons, Medusa messages cannot be used directly as AMPL messages.

On the other hand, Medusa pipes are used extensively as a base for the AMPL message-passing facility and for communication in general. Messages are particularly convenient for invoking remote operations in a distributed run-time system.

In spite of the overall cleanliness of Medusa pipes and messages for implementing a distributed program, the use of global shared memory is used for some synchronization tasks. Medusa provides a few memory operations like atomic increment, decrement, lock, and unlock, which execute much faster than the message operations.

5.2 Process pairs

An ideal implementation of AMPL would use at least two types of processors. One would be optimized to perform message-passing and resource-allocation functions, and the other would be designed to efficiently execute AMPL programs. Earlier, it was mentioned that the Kmaps could perform the communication functions. In our implementation, computer modules are used in pairs. One processor in each pair is called the *communication processor*, or CP, and serves to create processes, deliver messages, and perform garbage collection. The other processor is called the *application processor*, or AP, and serves to actually execute AMPL programs. The CP and AP interact closely and share the memory used for process frames. The run-time system is composed of many logically identical CP/AP pairs (see Figure 6). These pairs could be physically implemented on one computer module, but there are at least three reasons for not doing so. First, it is desirable to service incoming messages promptly to avoid letting a pipe become full. A full pipe might block some other process and perhaps lead to deadlock. One solution is to use interrupts to notify the receiver immediately, but Medusa does not provide a means of interrupting a process when a message arrives. Secondly, while the total memory of Cm* is large, individual computer modules have a limited amount of memory; typically about 40K bytes are available for programs and data. Separating the CP and AP allows the code for each to reside on separate machines. Third, using two physical processors can provide a large degree of additional parallelism, since message passing operations overlap other computations. The decision to use processors in pairs rather than, say, one communication processor for every two application processors was somewhat arbitrary. It turned out that some programs saturate the CP and others saturate the AP, so the right mixture is defined to a large extent by the nature of the AMPL program. Dynamic optimization of processor assignment would greatly complicate the implementation.

While there is very close interaction and a high degree of communication within a CP/AP pair, the interaction of one pair with another is much more limited. Communication between pairs is almost entirely conducted via messages. While not always the most efficient organization, this approach has the advantage of reducing complexity to a manageable level. If each CP were free to access the process frames of each AP, many complex interactions could arise. Consider the following scenario: Several CP's are delivering messages to a process. Another CP is simultaneously scanning the process frame for references needed for garbage collection. A fourth CP is rescheduling the process for execution after delivering a message, and a fifth CP is attempting to relocate the process to another AP. The possibility for errors is large, and tracking down the cause of a failure by recreating the preceding sequence of events may not be possible.

Now consider the situation where CP/AP pairs interact via messages. Each of the operations listed above is requested by sending messages to the CP associated with the process. The CP receives one request at a time, performs the operation, and moves on to the next request. Each operation is non-interruptable, and at most only needs to synchronize with the AP. It may seem that a great deal of parallelism is sacrificed. Actually, each of the other CP's is freed to perform their own local tasks in parallel. A further advantage of this message-based approach is locality. The CP must perform a large number of memory operations on a process frame to deliver a message. Intracluster memory accesses are about three times faster than intercluster accesses. By not splitting CP/AP pairs across clusters, it is guaranteed that the memory operations will be fast. A second point of efficiency is the elimination of locking and synchronization that would be required if process frames could be accessed by many communication processors.

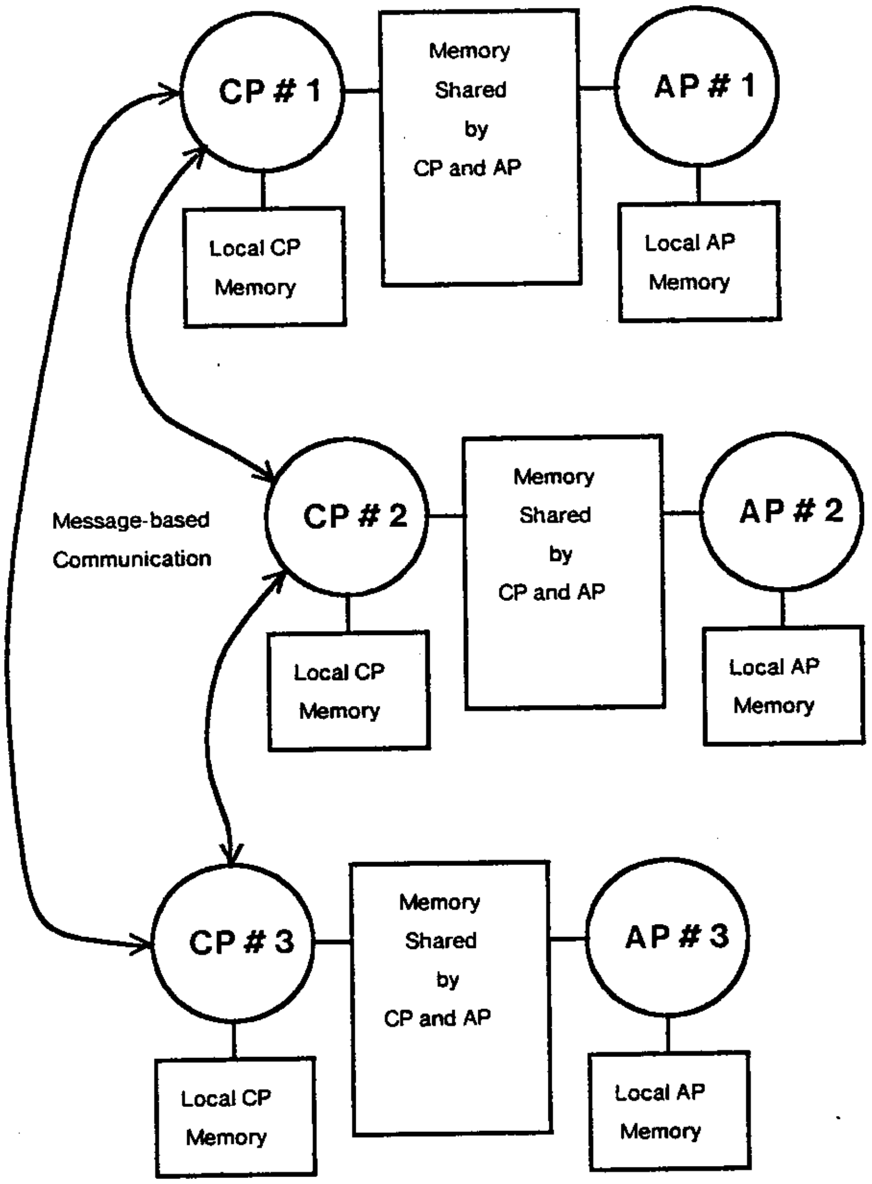


Figure 6: Basic structure of the run-time system.

Message passing introduces some overhead. More data is moved and copied to pack, send, receive, and unpack messages. Messages also take substantial amounts of Kmap processing time. Because there are so many opposing factors, it is unclear how a shared memory approach would compare to the present implementation, but it would certainly be less understandable.

5.3 Name space

A process or port *name* is defined as a reference to a process or port. A process is represented by an entry in a *process descriptor table* (see Figure 7). A process name is therefore a reference to a process descriptor table entry. Names of ports and processes are represented with two words as shown in Figure 8. The first word is an index into a process descriptor table. The second word contains a CP/AP pair index and an optional port number.

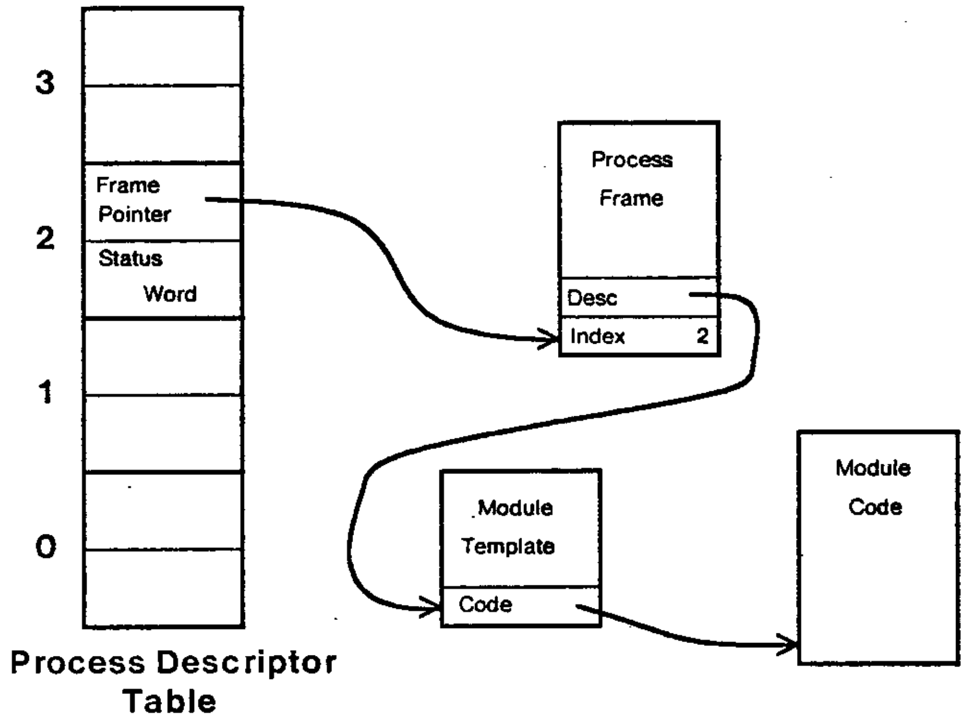


Figure 7: The process descriptor table.

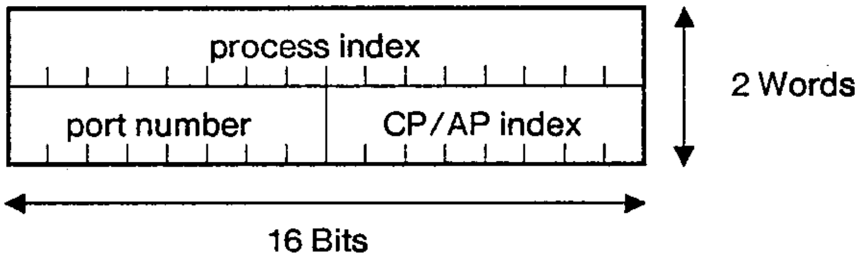


Figure 8: Representation of a process or port name.

An interesting property of the representation is that, given a process name and a port index, a port reference can be immediately constructed. The port index is used to simply "fill in" the port index field of the process name. This is how port selection is implemented. Port indexes are assigned by the compiler.

To map a name to a process, the CP/AP *pair index* is used to locate the physical processor. The *process index* is used to index a descriptor table located in memory shared by the pair. This table provides the address of the process frame and a number of status bits. If the name is a port reference, a *port index* is specified. The port index selects a port within the process frame. The name space is larger than the anticipated maximum number of processes. When a process terminates, the process frame is reclaimed immediately, but the descriptor table entry is retained. The status in the table is marked *halted* so that any future attempts to deliver messages to the process can be detected. Garbage collection is used to reclaim names.

All processes on a given CP/AP pair share a single address space. Subscripts must be checked to prevent inadvertent memory accesses which might corrupt run-time system structures or other processes. The shared address space makes context switching more efficient and allows processes to be smaller than the smallest hardware-protected memory segment (4K bytes).

Provision is made to relocate any process. There is no way to find and change all references to a process, since no back-pointers are maintained from the process frame to process names. Instead of immediately correcting the references, a forwarding address technique is used. To move a process, a new name is created for the new location. The original descriptor table is modified to hold a "forwarding address", i.e. the new process name, and the state is set to *forward*. Whenever a message arrives at the old process location, the sender is informed of the address change, and the message is forwarded. (Messages all contain the name of the sender.) The sender uses the address change notice to update the outdated reference. Eventually, if all references to the old name are updated, no more will exist, and the garbage collector will recover the old process descriptor table entry for reuse. The current implementation has all the necessary data structures, but does not actually move processes.

The process descriptor table is very useful for debugging and garbage collection, since it provides a fixed location in which all names are defined.

5.4 Implementation of send statements

A number of steps are performed to implement **send** statements. Execution begins when an AP reaches code for a **send** in a process. Execution of the **send** involves the use of several Medusa messages and several processors. At the completion of the **send**, the AMPL message must reside in the message queue of the destination port, and the AP resumes execution of the next AMPL statement.

The AP which initiates the **send** constructs a Medusa message in a *send buffer* allocated by the compiler. The Medusa message contains the following:

1. A function code (SEND).

2. The destination port name.

3. The name of the sender.

4. The actual AMPL message.

All Medusa messages sent by the AMPL run-time system have, as the first word, a function code which identifies the type of the message. These function codes will be written in SMALL CAPITALS. A Medusa message with the function code (for example) SEND will also be referred to as a SEND message or simply as a SEND.

The AP directly sends this message to the CP indicated by the destination port name. At this point, the AP suspends the current process. The next process on the ready-to-run queue is executed while the message is delivered. The time to perform this context switch is much less than the time required to deliver the message.

When the CP receives a message, it first reads the function code and then calls a handler to perform the requested function. Figure 9 outlines the structure or the CP program. In this case, a *receive handler* is called. The receive handler reads the process index from the destination port name and locates the process frame. Each process frame contains storage for message queues. The frame also contains the address of a *module template* which is a storage map used to locate message queues and variables in the process frame (see Figure 7). Once the CP finds its way to the appropriate queue, three cases arise:

```
initialize data structures
loop
        if message in high-priority pipe then
                receive message
                case message function code of
                        SEND:   call receive handler
                        REPLY:  call reply handler
                        CREATE: call create handler
                        .
                        .
                        .
                        end case
        elif message in low-priority pipe then
                receive message
                case message function code of
                        GCMARK:         call gcmark handler
                        GCSCAN:         call gcscan handler
                        .
                        .
                        .
                        end case
                end if
        end loop
```

Figure 9: The structure of the CP program.

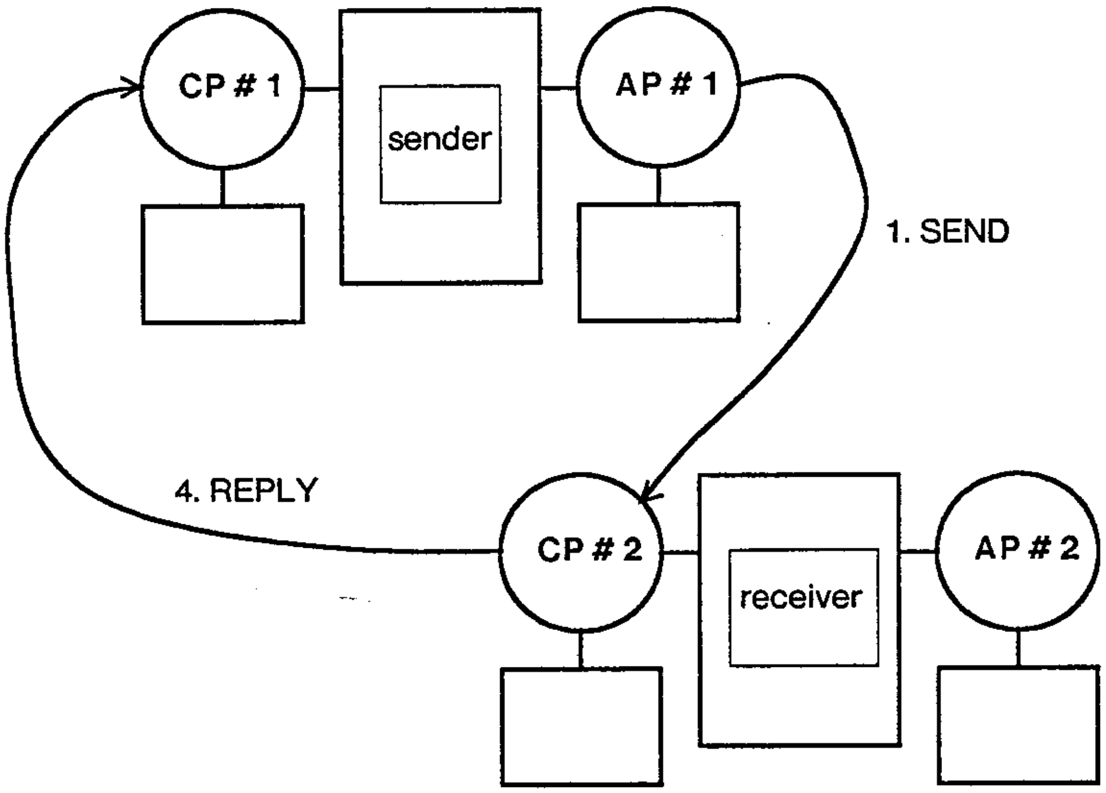
1. The queue is empty. The message is copied into the queue. If the receiving process is suspended, it is moved to the ready-to-run queue, because the process may be waiting for the message.

2. The queue is neither empty nor full. The message is copied into the queue and no rescheduling is attempted.

3. The queue is full. The name of the sender is placed on a special *waiting* list associated with the buffer, and the message is discarded.

In the first two cases, the message is delivered, so the sender must be rescheduled. The receive handler sends a Medusa message with the function code REPLY to the CP associated with the sender. (See Figure 10.) This completes the processing at the receiver's end. When the sender's CP receives the REPLY message, a *reply handler* is called. This routine simply reschedules the sending process by placing it on the AP's ready-to-run queue. Finally, the AP removes the process from the queue and resumes execution.

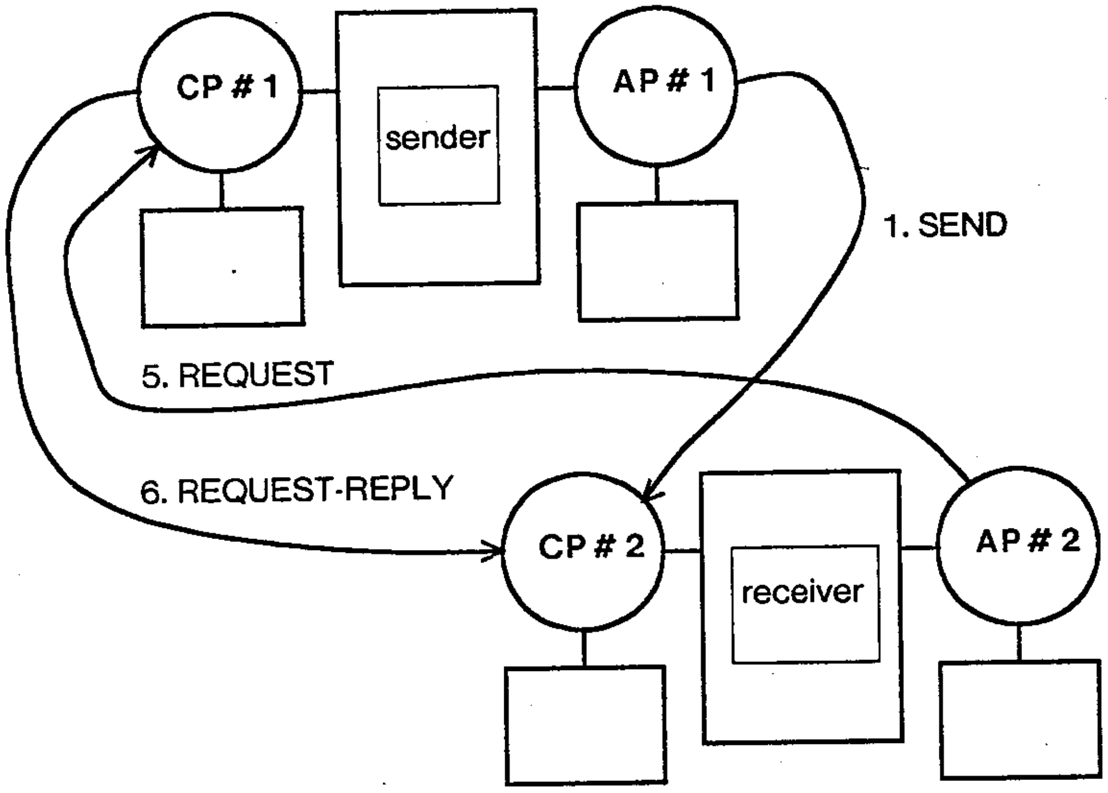
In the third case (the message queue is full), no REPLY message is returned, and the sending process remains suspended. (See Figure 11.) The name of the suspended process is saved on a waiting list. Whenever an AMPL message is removed from a queue, the waiting list is checked to see



1. AP # 1 sends send message to CP # 2
2. AP # 1 suspends sender
3. CP # 2 copies message into receiver's message queue
4. CP # 2 sends reply to CP # 1
5. CP # 1 reschedules sender
6. AP # 1 resumes execution of sender

Figure 10: Sending a message to a non-full port.

if any senders are suspended because of a previous attempt to send an AMPL message. If the waiting list is not empty, a REQUEST message is sent to the CP associated with the name at the head of the waiting list. When the CP receives the REQUEST, it finds the suspended process and locates its send buffer, which still contains a copy of the original SEND message. The function code is changed to REQUEST-REPLY and the (Medusa) message is again sent to the destination CP. This time, it is known that there is room in the destination port, so the suspended process is rescheduled without waiting for a reply. When the destination CP receives the REQUEST-REPLY message, the correct message queue is again located and the message is placed in the queue. As before, if the queue changes from empty to non-empty and the process is suspended, it is rescheduled.



1. AP # 1 sends send message to CP # 2
2. AP # 1 suspends sender
3. CP # 2 puts sender's name in receiving port's waiting list
4. receiver accepts a message, making room for a new one
5. AP # 2 sends request to CP # 1
6. CP # 1 resends sender's original message
7. CP # 1 reschedules sender
8. CP # 2 delivers message
9. AP # 1 resumes execution of sender

Figure 11: Sending a message to a port which is initially full.

Messages are delivered in order, even when senders are blocked waiting for a queue to become non-full. The actual procedure used by the receive handler is a little more complicated than described. The waiting list is a FIFO queue of process names. A message can be placed directly into the message queue only if the waiting list is empty. Otherwise, even if space is available for the message, the sender's name is added to the waiting list and the message is discarded. This prevents a message from being placed ahead of one sent previously, and preserves space for messages that have been requested. A name is removed from the waiting list only after the REQUEST-REPLY arrives to fill the reserved slot in the queue.

An optimization of this implementation is possible. A semaphore in each process frame is used to indicate that an AMPL **send** or **create** operation is in progress from that process. The semaphore is initialized to one. A *P* operation is performed on the semaphore by the process at the beginning of each **send** or **create** statement, and a *V* operation is performed by the CP when the operation completes. Also, a *P* and *V* operation surround each **accept** and **select** statement. The process is not suspended after initiating a send or create operation. The sender can continue executing up to the beginning of the next message operation. The language definition says that the message must be delivered before the sender can continue. The optimization does not obey this rule, but the language semantics are identical. How could a programmer detect that this optimization has been made? One way to find out if a process continues after a **send** is to try sending it a message or to have it send a second message. Both of these possibilities are prevented by the semaphore. The only other way to detect the optimization is to write a program that depends on real computation time. The language definition makes no guarantees of relative computation speeds or scheduling policy. The process will appear as if, after the send or create operation is complete, a number of statements are executed instantaneously. What actually happens cannot be observed.

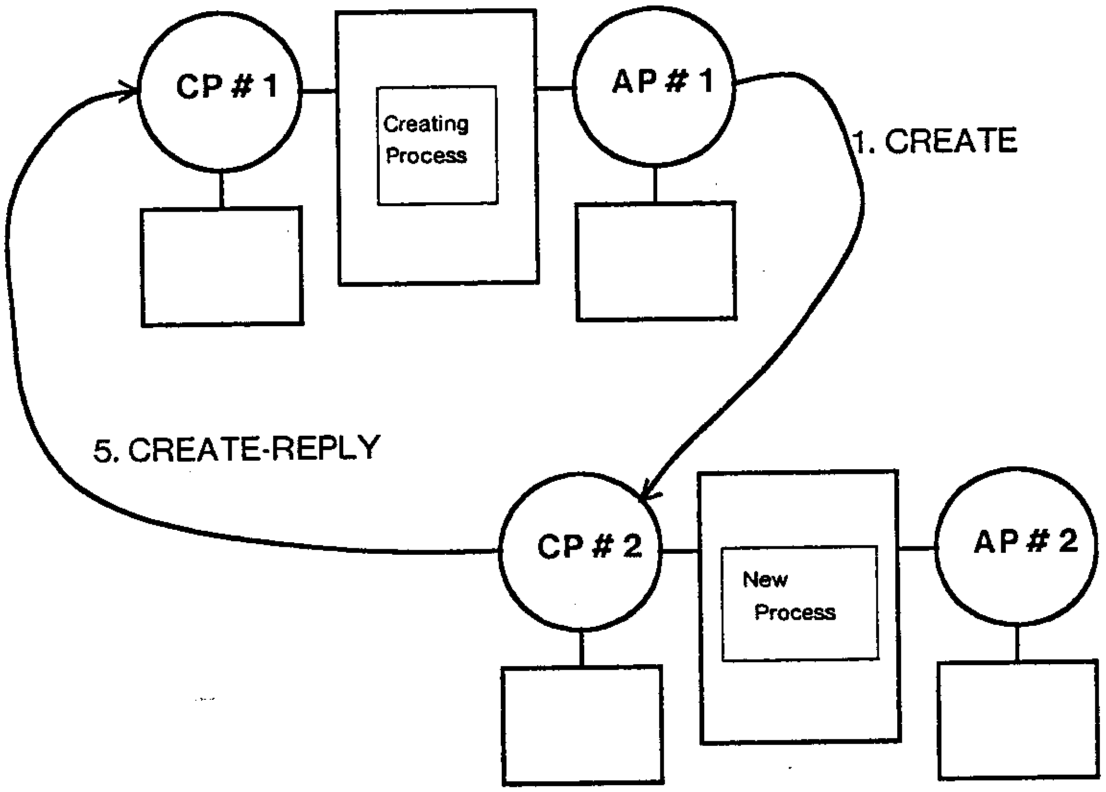
Another optimization could be made if AMPL were extended to support replies from messages. Rather than sending a REPLY when a message is placed in a queue, the REPLY would be sent after the message was received, and would contain a return value. This would require two or four Medusa messages, depending on whether the initial sender blocks waiting for a non-full queue. The current language and implementation requires either four or six Medusa messages to perform the equivalent task.

5.5 Creating a process

Like **sends**, an AMPL **create** can result in a number of Medusa messages, and involve several communication processors. The AP initiates the create operation by constructing a CREATE message in its send buffer. The message contains the name of the module to be instantiated (the name is represented by a module template address). The message also contains

1. the name of the current process (the sender).
2. An offset within the current process frame indicating where to return the new process name.
3. A counter initialized to zero to be used by the communication processors.
4. Values of actual parameters specified in the **create** call.

The AP selects a destination CP for the (Medusa) message, and sends it. The process suspends while the create operation is performed.

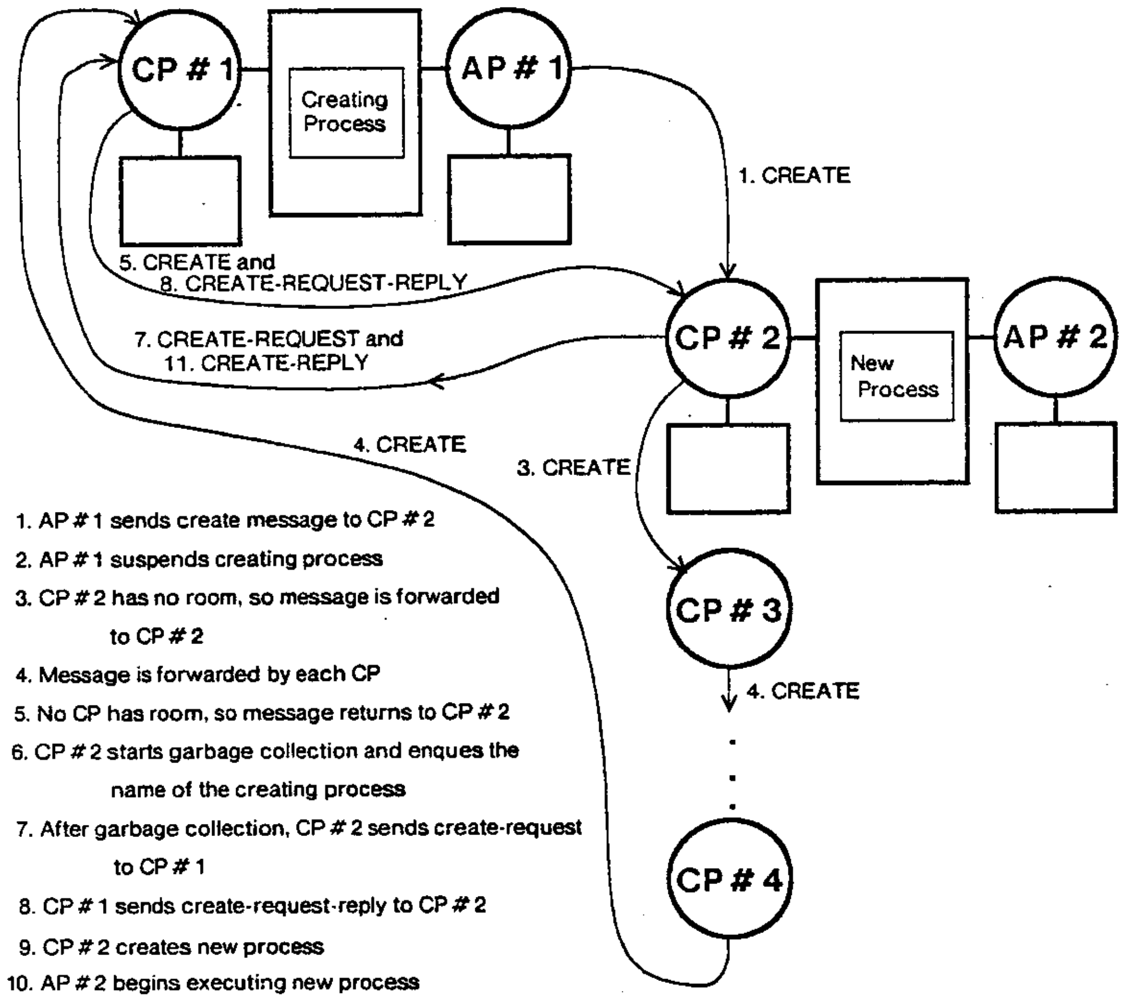


1. AP # 1 sends create message to CP # 2
2. AP # 1 suspends creating process
3. CP # 2 creates and schedules new process
4. AP # 2 begins executing new process
5. CP # 2 sends create-reply message to CP # 1
6. CP # 1 reschedules creating process
7. AP # 1 resumes execution of creating process

Figure 12: Creating a process where space is immediately available.

When the destination CP receives the message, it calls its *create handler*. Let us assume that there is a free slot in the process descriptor table and enough free memory for the process frame. (See Figure 12.) The CP uses its local copy of the module template to initialize a process frame, including the stack, message queues, and variables. The actual parameters are copied from the message to the formal parameter locations in the frame. The created process is scheduled for execution, and a CREATE-REPLY message is returned to the CP of the process which sent the CREATE message. The CREATE-REPLY contains the name of the new process, the offset specified in the CREATE message, and the name of the sender of the CREATE message.

When a CP receives a CREATE-REPLY, it locates the process which sent the CREATE message. The CP writes the name of the created process into process frame at the specified offset and reschedules the process. The process resumes execution at the next AMPL statement.



1. AP # 1 sends create message to CP # 2
2. AP # 1 suspends creating process
3. CP # 2 has no room, so message is forwarded to CP # 2
4. Message is forwarded by each CP
5. No CP has room, so message returns to CP # 2
6. CP # 2 starts garbage collection and enques the name of the creating process
7. After garbage collection, CP # 2 sends create-request to CP # 1
8. CP # 1 sends create-request-reply to CP # 2
9. CP # 2 creates new process
10. AP # 2 begins executing new process
11. CP # 2 sends create-reply message to CP # 1
12. CP # 1 reschedules creating process
13. AP # 1 resumes execution of creating process

Figure 13: Creating a process where space is not immediately available.

If the destination CP does not have room for the process frame, or does not have room in its descriptor table, garbage collection is started, and the CP forwards the CREATE message to another CP. (See Figure 13.) Process creation then proceeds as before. The CP's are ordered cyclically for the purpose of forwarding CREATE messages, so if no CP has room for the process, the message returns to the original destination CP. The CREATE message has a counter which is incremented each time the message is forwarded, so a CP can detect if the message has been forwarded by every CP. When this happens, the name of the process which originally sent the message is placed on a waiting list and the message is discarded. This waiting list is similar to the waiting list for saving the names of processes blocked by full message queues (see Section 5.4).

When garbage collection completes, each CP checks its waiting list to see if any processes are blocked waiting for a process creation. If so, a CREATE-REQUEST message is sent to request that the original create attempt be retried. This works very much like the REQUEST message used in implementing AMPL **sends**. The CP that receives the request simply locates the send buffer with the original message, changes the function code to CREATE-REQUEST-REPLY, and resends the message. If no CP has room to create a process on the second attempt, program execution is aborted and a message of explanation is written on the user's terminal. Otherwise, the CREATE-REQUEST-REPLY is treated just like a create message. A new process is created and a CREATE-REPLY is sent to the CP for the process executing the **create** statement.

This strategy effectively slows down program execution, when necessary, to give more processor time to the parallel garbage collector. As memory is exhausted on CP's, more and more processes are suspended waiting for create operations. This self-regulation is actually observable in practice and allows programs to run to completion, which would otherwise outrun the garbage collector and exhaust the available memory. It is possible that a program could be terminated when, after one more garbage collection cycles, another attempt at creation would succeed. This has not happened.

Whenever a process terminates, the AP sends the process name to its CP in an UNCREATE message. The CP responds by marking the process descriptor as *Halted*, and returns the process frame to free storage.

5.6 Garbage collection

In this section, the parallel garbage collector used in the AMPL run-time system is described. To our knowledge, there is only one other distributed parallel garbage collector implementation in existence [6]; it is used in the StarOS operating system [21]. Other descriptions of parallel garbage collectors exist [10, 24], but these deal with only one garbage collection process. For this reason, a rather detailed discussion of the garbage collector is included. Significant differences between the AMPL and StarOS garbage collectors will be presented.

All garbage collectors are based on the following simple "mark-scan" algorithm[1]

1. Mark the set of objects known not to be garbage (called the *root objects*).

2. Mark all objects that can be reached by following pointers from the marked objects. Repeat this step until no more unmarked objects can be found.

3. Reclaim the storage used by any object not marked; these objects are garbage because there is no way to reach them by following pointers from the set of root objects. The remaining non-garbage items will be referred to below as *reachable* objects.

The AMPL garbage collector consists of four phases, two of which run in parallel, as illustrated by Figure 14. The algorithm still corresponds to the classic mark-scan algorithm, and the extra phases are for synchronization only.

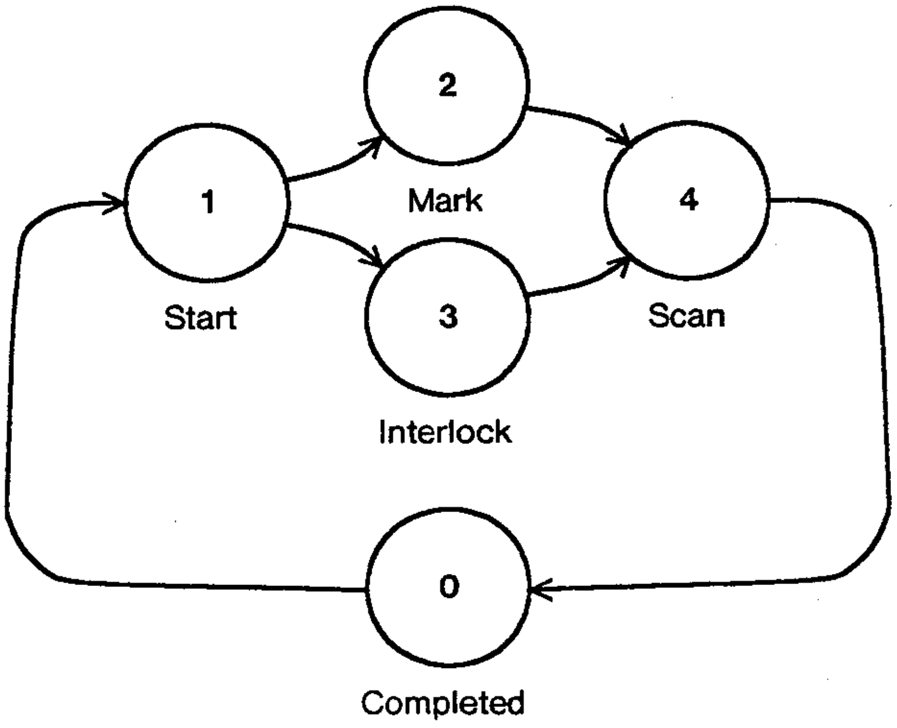


Figure 14: Garbage collection phases.

Garbage collection is performed by the communication processors. Recall the structure of the CP (Figure 9). Each CP has two Medusa pipes; one is for high-priority messages like SEND messages, and

[1]Even the semi-space copying algorithm [3] fits this description. Rather than setting a "mark bit," objects are marked by moving them to another block of memory, thus an address bit is used as the mark bit. [17]

the other is for low-priority garbage-collection messages. The CP continually polls its two pipes for messages. The low-priority pipe is examined only if the high-priority pipe is empty. Every message received has a function code which is used to specify a handler for that message. Each handler performs a small task, sometimes sending new messages. Garbage collection runs as a background task since garbage collection messages are sent to the low-priority pipe. This organization effectively multiplexes the CP to perform many tasks while only using one process. Since CP's interact almost exclusively through messages, a large amount of overhead for synchronization within each message handler is avoided. For globally synchronizing garbage collection phases, global counters and Medusa-provided atomic increment and decrement operations are also used. These operations are much faster than Medusa messages, but their use is purely an optimization of a message-based implementation.

5.6.1 What to collect

The only dynamically allocated objects in AMPL are processes and messages. Messages take up storage only until they are delivered, so garbage collection is only concerned with reclaiming process names (descriptor table entries) and process frames. The initial (root) set of processes known not to be garbage is composed of the following:

1. The processes being executed by AP's.
2. Processes in the ready-to-run queues.
3. Processes whose names are on the create waiting queue of some CP. (These processes are blocked waiting to create a process.)
4. Process names in messages waiting to be delivered.

Any process reachable from this set is marked; the rest are garbage. If a name is identified as garbage, and the name refers to a process frame, then the process frame is not only suspended, but it can never be rescheduled (no other process can send it a message). Therefore, the frame is returned to the free storage pool.

5.6.2 Phase 1

Any CP can start garbage collection. To do so, a memory location global to all CP's is locked with a Medusa "test and set" operation to prevent two CP's from starting garbage collection simultaneously. Then, GCSTART messages are sent to each CP. The CP which starts garbage collection is called the *garbage collection master.*

When a CP receives a GCSTART message, it finds all the root processes, and sends GCMARK messages to the CP associated with each name. Many of these messages will be sent from each CP back to itself. This is a simplification which could be optimized. The function of GCMARK messages is given below. The CP then sets a local state variable to *GCstarted*, and returns a GCSTARTREPLY message to the garbage collection master, who is identified by a field in the GCSTART message.

There are some root names which cannot be accessed by any CP when garbage collection is started. These are names which are in Medusa pipes. Consider a process suspended waiting for a REPLY message. The reply could have the only reference to this process, so if garbage collection completed before this message arrived, the suspended process would be erroneously collected. To prevent this, the garbage collection master sends an INTERLOCK message to each high-priority port after a GCSTARTREPLY message is received from each CP. Receipt of the INTERLOCK message is described below.

5.6.3 Phase 2

The second phase is concerned mainly with receiving and handling GCMARK messages. Each GCMARK message contains one process name. The handler for these messages starts by setting the mark bit for that name in the process descriptor table. If the name was previously unmarked, and there is a process frame for the name (the process has not terminated), then the frame is scanned using the module descriptor to locate all process and port names. For each non-null name found, a GCMARK message containing the name is sent to the CP indicated in the name. (Recall that each name specifies a process descriptor table index and a CP/AP pair index.)

While a frame is being scanned, a lock is set on the frame to prevent the AP from copying any process or port names. Otherwise, the garbage collector might miss a reference as it is being moved, or it might read a name in a corrupt state, i.e. the first word of one name and the second word of another. In STAROS, objects are not locked as they are scanned. Instead, the operating system must be called to copy a name, and the system notifies the garbage collector. The corresponding action in AMPL would either be to require the CP to copy references, or to have the AP inform the CP whenever a name is copied. In the latter case, locks would still be needed to prevent the CP from reading names in a corrupt state. The CP would release the lock between the reading of each name. The trade-offs are summarized below. For the method implemented,

1. The CP performs only one lock operation per process scanned.

2. The AP may be blocked for the time it takes to scan the frame for names.

For the alternate scheme,

1. The CP performs one lock operation for each name scanned.
2. The AP is at most only blocked for the time it takes to read one name.
3. The AP must make a test and possibly send a message to the CP whenever a name is moved.

While marking is taking place, processes can be sending messages and creating new processes. The following invariant is maintained:

> *If a process is marked by the garbage collector, then a* GCMARK *message has been sent for each name contained in that process frame.*

To maintain this property, special precautions must be taken when sending messages and creating processes. Every SEND, REQUEST-REPLY, CREATE, and CREATE-REPLY message has a tag which is used to mark the message if the sender is marked. If a message is marked, then it is known that a GCMARK message has been sent for each name in the message, because the names in the message are copied from variables in a marked process frame.

When an AMPL message is delivered to a port, the tag is checked. If it is marked, then no action is taken. If the message is unmarked, and the receiving process is marked, then a GCMARK message is sent for every name in the message before the message is placed in the receiver's message queue. This must be done to maintain the invariant without unmarking process frames.

In the case of create messages, the created process is marked to prevent its garbage collection. If the create message tag is unmarked, then a GCMARK message is sent for each name supplied in the actual parameter list.

Phase 2 completes when there are no more GCMARK messages, and all reachable processes have been marked. Determining when this condition is reached is not simple because messages are sent and received asynchronously by all CP's. Even if all pipes are empty, there could be a message in transit. Completion is detected by maintaining a global counter which is incremented before each GCMARK message is sent. After a GCMARK message is received and the process frame is scanned, the global counter is decremented. The counter (initially zero) returns to zero when all GCMARK messages have been processed.

In StarOS, garbage collectors rely on real-time constraints to determine when the mark phase is complete. When it *appears* that marking is done, the garbage collectors wait for a predetermined time to make sure no new names are discovered. Since marking is assisted by Kmaps, processing is fast, and the time delay is short. A third method could be based on sending replies to each GCMARK message. A fourth approach is discussed in Appendix I.

The mark phase is guaranteed to terminate in the following sense. Assume that unlimited memory and name space are available, so that processing is never held up because space is unavailable. Garbage collection messages are handled at a finite rate in each CP, but the high-priority messages, i.e. non-garbage collection messages which may contain names, can be handled at an arbitrarily fast, but finite rate. The mark phase will terminate in a finite time. The proof is based on the invariant stated on page 44, the fact that created processes are marked, and the fact that processes are never unmarked during the mark phase. From these three facts, we can deduce the following:

1. The number of unmarked, but reachable processes is finite. We assume there are a finite number of processes to begin with. Since all created processes are marked, and no processes change from the marked to the unmarked state, the number of unmarked processes does not grow during the mark phase.

2. The number of unmarked but reachable processes is non-negative. This is obvious.

3. The number of unmarked reachable processes is monotonically decreasing. The number can never increase because no processes are unmarked during the mark phase. To see that the number will decrease, consider the set of reachable unmarked processes. There must be at least one reference from a marked process to one of these unmarked processes. By the invariant, a GCMARK message has been sent with that reference. The queues of GCMARK messages can get arbitrarily large, depending on the relative amount of processing time devoted to garbage collection, but the queues are always finite. Since garbage collection messages are handled at a finite rate, this message will eventually be received, the process will be marked, and the number of unmarked processes will decrease.

Using the above three observations, we see that the number of unmarked, but reachable processes must go to zero. Eventually, all reachable process frames become marked and no more GCMARK messages are sent, so the queues become empty. At this point the mark phase terminates.

In StarOS, a different technique is used to prevent the garbage collector from missing a reference to a reachable object. Whenever a new reference is made, the reference is sent to the garbage collector so that the referenced object will be marked. If assignment of references occurs arbitrarily more frequently than garbage collection operations, then the marking phase will not terminate. (This can occur even when the total number of names and objects is constant.) This property is not a problem in practice for StarOS.

5.6.4 Phase 3

Recall that in phase 1, an interlock message was sent to the high priority pipe of each CP. Phase 3 is the period between the end of Phase 1 and the time the interlock messages are all received. Phase 3 overlaps Phase 2, as indicated in Figure 14.

Some rather intricate synchronization involving another global counter is used to detect when both phases 2 and 3 have completed in all CP's. The StarOS analogue of phase 3 is the time delay used to check that all marking is complete. In StarOS, the time during which a name can be hidden (in Kmap registers) is bounded by a short interval, so simply waiting is preferable to the use of interlock messages.

5.6.5 Phase 4

The CP detecting the end of Phases 2 and 3 in all CP's sends a GCSCAN message to each CP. When this message is received, the process descriptor table is scanned by each CP and unmarked processes are collected. This is currently all performed in one continuous operation, but the CP could easily alternate between checking the high-priority queue and scanning a few more table entries. As each CP completes its scan, it increments yet another global counter. When this counter reaches the number of CP's, the "garbage collection in progress" lock is reset to allow another garbage collection to be started. The period between garbage collections is referred to as phase 0.

5.6.6 An optimization

An extra status bit is used in the process descriptor table entry for each process to indicate when a GCMARK message for that process has been sent. The bit can only be set when a GCMARK message is sent from a CP to itself because, otherwise, a CP would have to access the process descriptor table of some other CP. Before a GCMARK message is sent from a CP to itself, this status bit and the mark bit are examined. If either is set, the message is redundant, so it is not sent.

In StarOS, a special mark is used to indicate that an object has no references outside the cluster. For global garbage collections, there must be at least one garbage collector per cluster. These objects can be collected without synchronizing with other garbage collection processes, since if no local references are found, the object is known to be garbage. This optimization has not been included in the AMPL garbage collector since processes are created on remote CP/AP pairs to achieve more parallelism, resulting in many references between pairs.

5.7 The Compiler

The AMPL parser was implemented by an automatic parser-generator called FEG [13]. The code generator is written in Bliss-36 using support tools that interface cleanly with the FEG-produced parser and allow the parse-tree to be conveniently accessed.

The code-generator outputs Bliss-11 code. There are several advantages to this approach.

Bliss-11 code is easier to read than assembly or machine code. In fact, much of the compiler was debugged by reading the output. A second advantage is that Bliss-11 is easier to generate than assembly code. In particular, the Bliss-11 compiler takes care of register allocation and control structures. The Bliss-11 macro facility is used not only to make code more readable, but also to act as a post-processor for the AMPL compiler. Finally, the Bliss-11 compiler is an excellent optimizer. It probably results in better code than a simple assembly-code generator would produce. The fact that Bliss-11 is a rather low-level language is an advantage here, since there are few assumptions in Bliss-11 which interfere with the efficient translation of AMPL programs.

5.7.1 Storage allocation

All storage within process frames is statically allocated by the compiler. Process frames consist of several sections. A system-information section provides the process name, a pointer to its module descriptor, the frame size, etc. The message queue section contains storage for each message queue. Queues are fixed-sized linear lists. For queues which are shared by several ports (see Section 3.1) the queue-element size is as big as the largest message. A variable section contains storage for the formal parameters and variables. A send buffer is allocated which is large enough for any message sent by the module. Finally, there is a fixed-sized stack. Because of restrictions imposed by Medusa, the stack and send buffer cannot cross a 4096-byte page boundary. The compiler allocates storage to meet this constraint. Knuth's buddy algorithm [23] is used by the memory manager so that process frames are alligned on the proper boundaries. The storage layout is completely defined by the compiler and encoded into a *module template* which is used by the CP to find message queues and to locate names during garbage collection. The template also points to the code for the module as illustrated in Figure 7.

Because local accesses are much faster than nonlocal accesses, code is always kept local to the processor which executes it. Each AP is provided with a copy of its portion of the run-time system, the compiled AMPL code, and the module templates. Since code is available at each AP, AMPL processes run with equal efficiency on any AP. Code and templates are shared when several instantiations of a module exist at a single CP/AP pair.

A standard part of the run-time system is a program which will describe an AMPL process, showing its name, frame address, buffer locations and contents, etc. This program can be invoked from the Medusa debugger. Enough information is present so that a complete symbolic debugger could be added to the AMPL system if desired.

6 Performance measurements

A number of small programs have been coded in AMPL and executed on Cm*, with the goal of learning how the language influences algorithm design and affects performance. The measurements reveal run-time characteristics of AMPL programs. In some cases, the data suggests optimizations of the implementation, and extensions or modifications to the language.

Comparing the performance of AMPL programs to other programs implemented on Cm* is difficult. We would like to know what are the consequences of design decisions in AMPL, whereas what we measure is often strongly influenced by one particular implementation. For example, in [22], two versions of the program PDE are compared. One version executed the problem in 7 seconds using 38 processors. Another "improved and optimized" version accomplished the same task in 7 seconds using only 7 processors. Both of these programs were written in Bliss-11 for Cm*. Programs written in different languages should demonstrate even wider variations in performance.

In the case of the PDE program, the faster version demonstrated relatively less speedup as a function of the number of processors. In fact, the real execution time began to *increase* when more than 22 processors were used. The slower program continued to run faster as more processors were added. This paradox is resolved by the consideration of contention. When many processes share data, the time devoted to synchronization increases with the number of processors. If synchronization and communication costs dominate the total cost of computation, then speedup due to parallelism will be slight or even negative.

Some of the AMPL programs to be described exhibit a negligible communication cost, and the total execution time is determined almost entirely by the AP processing time. Other programs are dominated by the communication cost, and AP's are idle much of the time. Changing the relative execution speeds of the AP and CP by optimizing the run-time system or increasing the quality of the code generator could radically change some of our results. For this reason, measures of program performance are sought which are independent of relative execution speed. One such measure is the ratio of information passed in messages to the information stored in or fetched from local variables. A low ratio indicates that most of the execution time is spent accessing local memory and performing local computation. A high ratio indicates that most of the execution time is spent communicating with other processes, and little local computation is performed.

6.1 Instrumentation

A flexible system for monitoring execution has been implemented. The code for the run-time system contains macro calls for every important event. For example, the routine which performs an AMPL **send** statement includes the following macro call:

```
AMPLsend(.len, .buff);
```

where ".len" is the length of the message and ".buff" is the send-buffer address. The compiler also generates a few macro calls for instrumentation.

Instrumentation macros are defined in an *include* file which can be altered according to what measurements are desired. For example, the above call to AMPLsend is normally defined to generate the following code:

```
(InSndPB[0] = (.len);
 $KAdd(InSndPB);
 $KIncWord(AMPLsendNum));
```

The first two lines add the message length to a sum, using an indivisible addition operation provided by Medusa microcode. The third line indivisibly increments a counter. The whole operation executes in about 100us, equivalent to about 13 LSI-11 instruction times.

The use of macro calls simplifies modifications to the instrumentation. For example, if we were interested in the sources or destinations of messages, the AMPLsend macro could be rewritten to collect the extra information. Instrumentation overhead can be eliminated by redefining all instrumentation macros to the empty string.

A dedicated processor is used to gather this data at run-time, so as to provide minimal interference with the run-time system. One advantage of this approach is that values like the number of bytes sent in messages can be accumulated using single-precision arithmetic without danger of overflow. The instrumentation process periodically clears these accumulators and transfers their contents to a central, extended-precision accumulator. (The fact that instrumentation variables are shared by several processes explains why indivisible operations are used in the above example.) Figure 15 illustrates the instrumentation organization. Another use of the instrumentation process is to take samples of the state of the run-time system. AP and CP utilization are computed by sampling flags which tell when the AP and CP are idle. Using a separate process to collect data also allows each AP and CP to avoid time-consuming inter-cluster memory updates and reduces the code required for the AP and CP processors.

A few general comments about our measurements must be made. First, execution times of AMPL

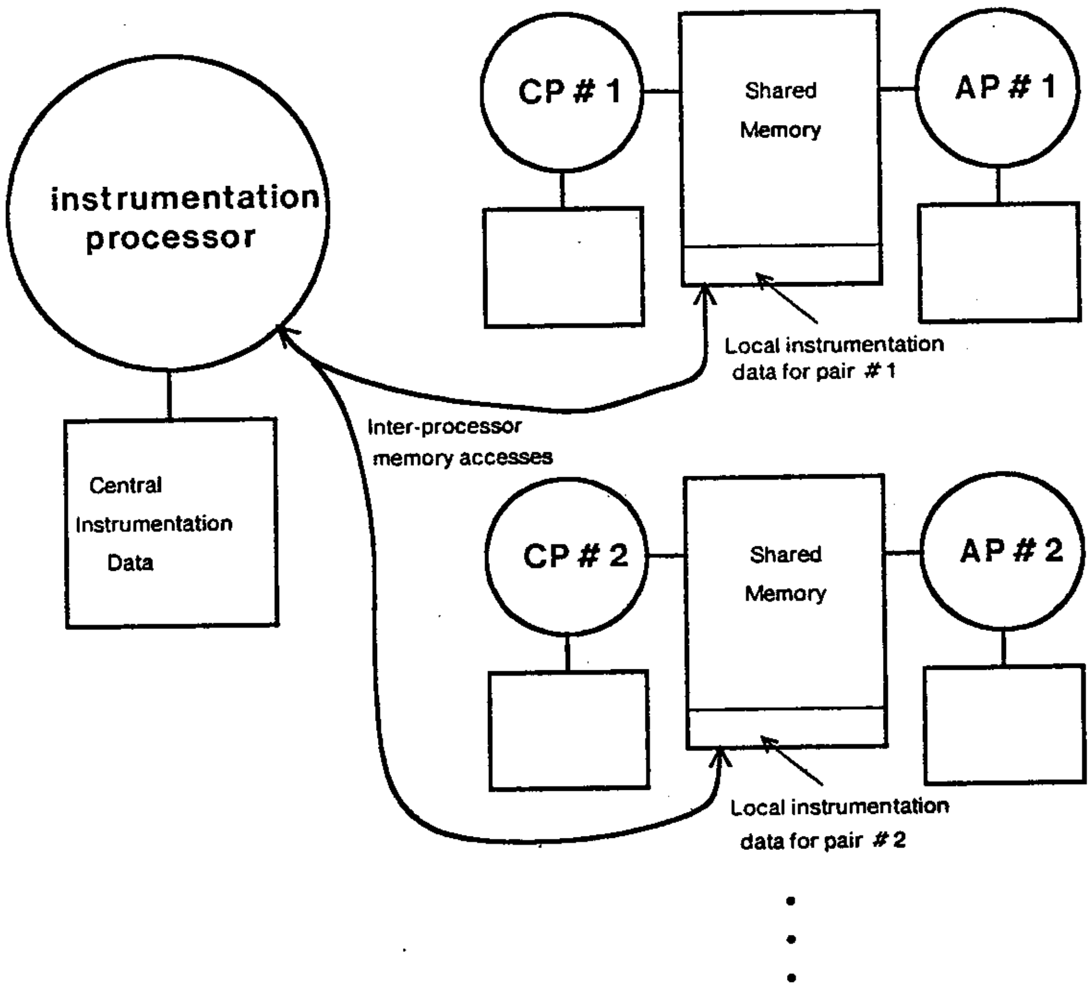

instrumentation processor
Central Instrumentation Data
CP # 1
Shared Memory
AP # 1
Local instrumentation data for pair # 1
Inter-processor memory accesses
CP # 2
Shared Memory
AP # 2
Local instrumentation data for pair # 2

Figure 15: Instrumentation of the AMPL run-time system.

programs vary slightly from one run to the next. The execution speeds of the computer modules on Cm* vary. Many objects, including Medusa pipes and some memory pages are dynamically allocated. The choice of object locations can also affect timings. Special efforts have been made to specify locations where placement is critical, but all measurements given in the following sections should only be considered accurate to within 5 percent. Consequently, most of the measurements are rounded to two decimal places in this report. Secondly, the run-time system was modified so that the user is optionally queried whenever a process location is chosen. In this way, the user can override the default locations chosen by the run-time system. This facility was used in a few of the tests to achieve the desired configuration. The location query can be enabled or disabled by using the linker to set a flag. Unless otherwise indicated, all measurements are made with all bounds checking and debugging facilities enabled. The debugging facilities slow the run-time system by approximately 10 percent.

6.2 Static measurements

The run-time system has a total of 3424 lines of code of which 28% are comments. This does not include any of the Medusa operating system, or the rather large file that contains macros and definitions used to interface Bliss-11 programs to Medusa. The compiler has 3364 lines of code. Of this, 3058 lines are in the code generator (15% are comments), and 306 lines define the syntax for the parser generator.

The code sizes for the run-time system are listed in Table 1. Typically, 16K bytes are allocated by each AP for the process frame heap. About 15K bytes are available for compiled AMPL programs, limiting program size to about 350 lines of code. Since little attention was paid to conserving memory, there is considerable room for improvement in this area. Given the small address space of Cm* processors, a method of distributing code as well as processes would be necessary to run large programs.

Description of Code	**Size (bytes)**
memory manager	792
garbage collector	2646
debugging and display routines	2638
input/output routines	1408
Bliss-11 run-time routines	240
other AP routines and initialization	2830
other CP routines and initialization	5336
AMPL input/output modules	4334
Total	20224

Table 1: Run-time system code size.

6.3 Send operations

The program SNDS is used to measure the time to **send** and **accept** a minimum length AMPL message. The program was run on a single CP/AP pair. (The same code is executed regardless of the destination of a message.) The program is written so that all sends are to an empty port and all accepts are from a full port, so this represents the fastest time to deliver a message. Section III.1 has a listing of SNDS. The time to send 10000 messages is 96 seconds, so each message takes 9.6 ms, or about 1200 LSI-11 instruction times.

Some simple analysis was performed to see where the time is spent. Table 2 provides a breakdown of the total time in terms of instruction counts. Instruction counts are used rather than instruction *times* to simplify the analysis. An average instruction time of 8 microseconds is assumed. Actual instruction times are a function of the type of instruction, the source addressing mode, the destination addressing mode, and the location of all memory references (local, intra-cluster, or inter-cluster). Where Medusa operations are invoked, the equivalent number of LSI-11 instructions is taken from [22] or [28]. The instruction counts indicate that the computation time is spread fairly uniformly among the logical subtasks required to send a message. There is little overlap between the CP and AP in this program. The measured time is therefore somewhat misleading, since if two processes each sent a message, the CP and AP execution would overlap. Although the time to deliver a message would increase, the measured throughput would be significantly greater. Furthermore, if multiple CP/AP pairs are used, even more parallelism is obtained.

		Instructions	Percent of Total
AP instructions for **send** statement:			
evaluate arguments and call send routine		8	1%
build SEND message		108	11
send Medusa message		62	6
context swap		68	7
	subtotal	246	25
CP instructions to handle SEND message:			
poll pipes for SEND message		119	12
SEND handler		180	18
send REPLY message		84	9
reschedule receiver		40	4
	subtotal	423	43
AP instructions for **accept** statement:	*subtotal*	123	12
CP instructions to handle REPLY message:			
poll pipes for REPLY message		119	12
REPLY handler		36	4
reschedule sending process		40	4
	subtotal	195	20
	total	987	100%
	total estimated time	7.9 ms	

Table 2: Time to **send** and **accept** a message.

The total estimated execution time is faster than the measured time. Several factors have not been included in the estimate. First, non-local memory references have not been considered. The AMPL process frame is local to the AP. All references from the CP to the process frame take about 10us, so each reference costs approximately one extra instruction time. Second, our estimate of 8us per instruction may be incorrect, particularly since our implementation uses a large amount of indirect addressing to locate variables and port structures in the process frame. Third, estimates are based on inspection of the code, not on instruction traces. In particular, it is assumed that when a message arrives at the CP, the CP is at a random point in its cycle of polling the high- and low-priority pipes. A complete cycle (two conditional receives) takes just over 1ms.

Table 3 contains an alternate breakdown to the total execution time. As indicated, Kmap (microcode) operations take a significant amount of the total execution time. Besides using Medusa message operations, the Kmap operations include microcoded block-move operations to build the SEND message, to copy the AMPL message value into the destination message queue, and to copy from the queue into the variable specified by the **accept** statement.

	Instructions	Percent of Total
save registers and coroutine call	84	9%
debugging	90	9
Kmap operations	366	37
other	447	45
total	987	100%
total estimated time	6.9 ms	

Table 3: Alternate analysis of the execution of a **send** statement.

6.4 Create operations

The program CREA can be used to create an arbitrary number of "empty" processes which contain no parameters, variables, ports, or statements. This program was run on a single CP/AP pair; it creates and destroys 400 processes in 8 seconds, or one process in about 20 ms. The process descriptor table is large enough that no garbage collection is invoked during these create operations. In this time the AP is idle 87% of the time. The CP is never idle.

Tables 4 and 5 provide an analysis of timings analogous to those given for **send** statements.

Most of the time (58%) is taken by the CP to allocate, initialize, and schedule the new process frame. The CP executes 420 instructions[2] to fill the frame's stack area with zeroes, which takes 18%

[2]Since the time taken by the inner loop which fills zeroes is critical, instruction timings from [9] were used rather than instruction counts. The equivalent count of 420 is obtained by dividing the total time by 8us. This is the same formula used to derive equivalent instruction counts for microcoded operations.

AP instructions for **create**:		Instructions	Percent of Total
evaluate arguments and call create routine		8	0.3%
build CREATE message		67	2.9
send Medusa message		62	2.7
context swap		68	2.9
	subtotal	205	8.9
CP instructions to handle CREATE message:			
poll pipes for CREATE message		119	5.2
allocate process frame and table entry		317	13.7
initialize process frame		802	34.7
schedule new process		40	1.7
send CREATE-REPLY message		62	2.7
	subtotal	1340	58.0
CP instructions to handle CREATE-REPLY message:			
poll pipes for CREATE-REPLY message		119	5.2
CREATE-REPLY handler		31	1.3
reschedule creating process		40	1.7
	subtotal	190	8.2
AP instructions to run created process:			
build UNCREATE message and context swap		117	5.1
send UNCREATE message		62	2.7
	subtotal	179	7.7
CP instructions to handle UNCREATE message:			
poll pipes for UNCREATE message		119	5.2
UNCREATE handler		22	1.0
deallocate process frame		255	11.0
	subtotal	396	7.1
	total	2310	100%
	total estimated time	18 ms	

Table 4: Analysis of process creation and termination.

		Instructions	Percent of Total
save registers and coroutine calls		185	8%
debugging		104	5
Kmap operations		510	22
other		1511	65
	total	2310	100%
	total estimated time		18 ms

Table 5: Alternate analysis of process creation and termination.

of the total time listed in the tables. This initialization is performed only to facilitate debugging and could be eliminated. A significant amount of time (24%) is spent allocating and deallocating process frames. This is an estimate based on the assumption that a memory block is split in two one time for each allocation, and two blocks are reunited one time for each deallocation. The memory manager execution time could probably be cut in half without much effort by writing more efficient memory management code.

Obtaining further improvements in the run-time system would be more difficult. The measured time for creation and termination of a process (20ms) is somewhat longer than the estimated time (18ms). The factors mentioned in Section 6.3 are applicable here. The process creation and termination time compares favorably with the message send and accept time. Note that we have included time to terminate a process and reclaim the process frame in the preceding measurements. The estimated time just to create a process is about 14ms.

6.5 Garbage collection

The program GC creates an arbitrary number of processes. This program is identical to CREA except the created processes each declare a port and a variable and attempt to execute an **accept** statement. No message is ever sent to the created processes, so all of them remain suspended. GC is listed in Section III.2. The process frames fill the available memory causing the garbage collector to be invoked. On a single CP/AP pair, the execution time for 2000 create operations is 53 seconds, or about 27 ms per create. This includes time for garbage collection, which is in progress 61 percent of the time. A total of 1138 GCMARK messages are sent to accomplish 95 garbage collection cycles. The AP is idle 84% of the time, and the CP is idle 8% of the time.

6.6 PDES

The PDES program uses the finite difference method to solve Laplace's partial differential equation with specified boundary conditions (Dirichlet's problem) [22].

Two versions of this algorithm were implemented. The first is PDES, which uses synchronous communication. (See Figure 16.) The grid is divided into slices, and a slave process is created to compute each slice. The slave processes exchange edges after each iteration of the relaxation algorithm. Synchronization is maintained only by waiting on edges, so neighboring slaves are only synchronized to within one iteration. In a program with N slaves, if the first slave is computing iteration I, then the last slave could be computing any iteration from I - (N - 1) to I + (N - 1). To detect convergence, each slave sends a status message to a master process after each iteration.

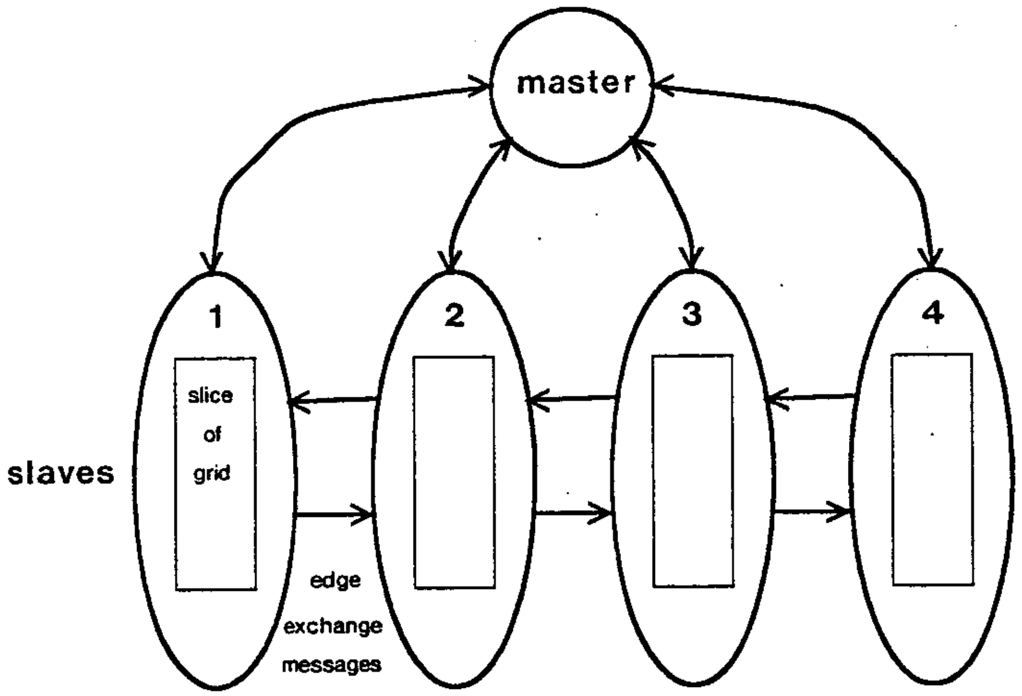


Figure 16: Structure of the PDES program.

The program parameters were varied in two ways. First, the program was executed with varying numbers of slaves on a single CP/AP pair. This allows us to measure the overhead incurred by dividing the grid across communicating processes. Secondly, the program was executed with slaves on one, two, and four CP/AP pairs to measure the relative speedup obtainable by using more processors. Figure 17 (unshaded bars) shows the real execution times and total CPU times for various parameterizations of the PDES program. All tests used a grid size of 34 by 34. The measurements do not include process creation and data initialization times. The overhead added by increasing the number of slaves is slight, so increasing the number of processors provides almost linear speedup.

An extra CP/AP pair was used for the master process, which creates the slaves and detects when the grid has converged. Keeping the master in a separate processor pair allowed us to easily measure the processing time taken by the master. In the worst case (with four slaves) the master uses only 3 percent of the total CPU time.

The number of iterations required to reach convergence increases as the number of slaves increases, because the rate of convergence depends on the order in which values of the grid are

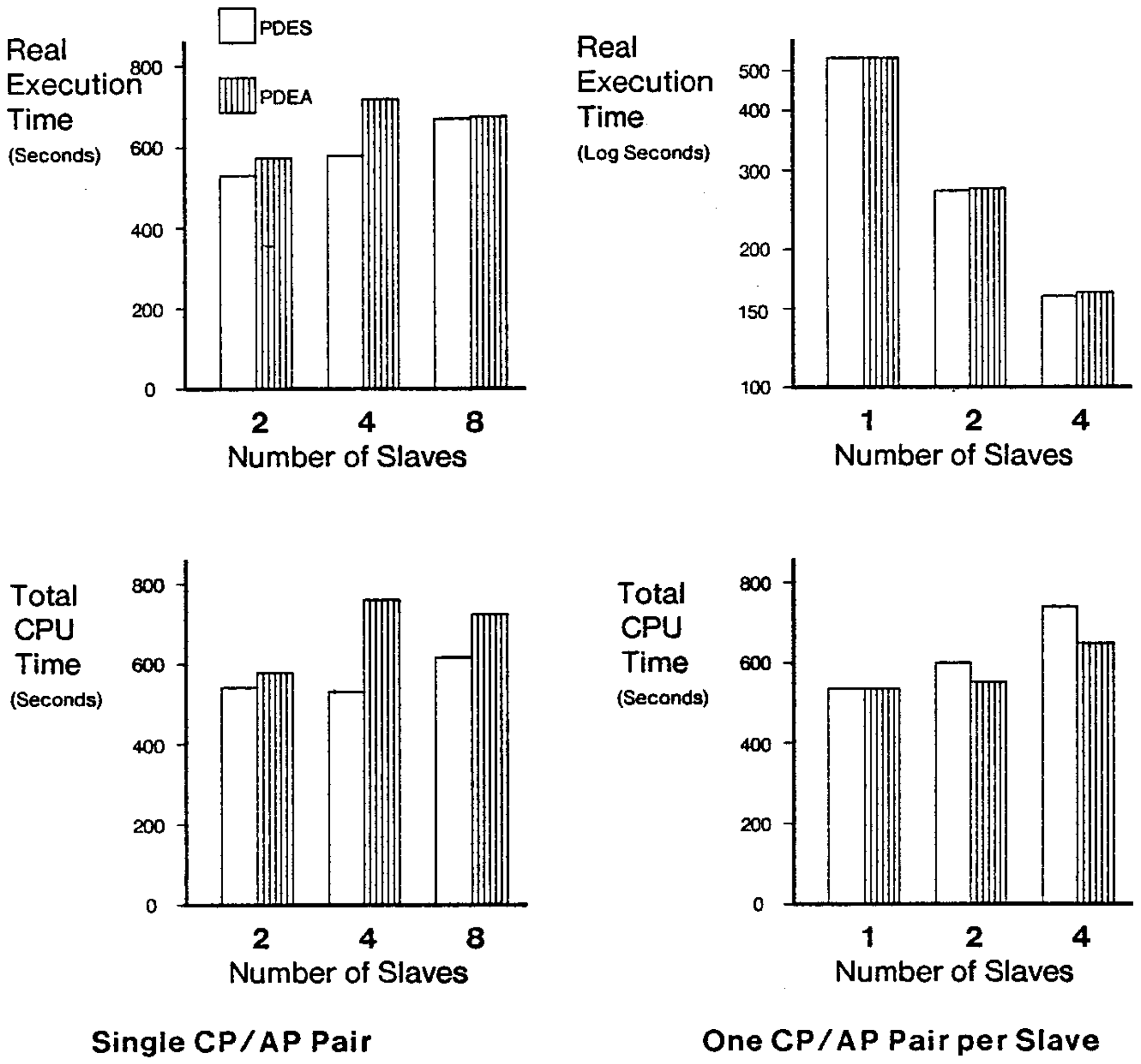


Figure 17: Measurements of the PDES and PDEA programs.

updated. The number of iterations required as a function of the number of slaves is illustrated in Figure 18.

The number of bytes accessed within each slave can be compared to the number of bytes transferred in messages. If we count only accesses to the grid data, each iteration reads five two-byte values[3] and writes one for each interior grid point, for a total of

[3]Four neighbors are averaged to compute the next value of each point. The current grid point data is read to test for convergence.

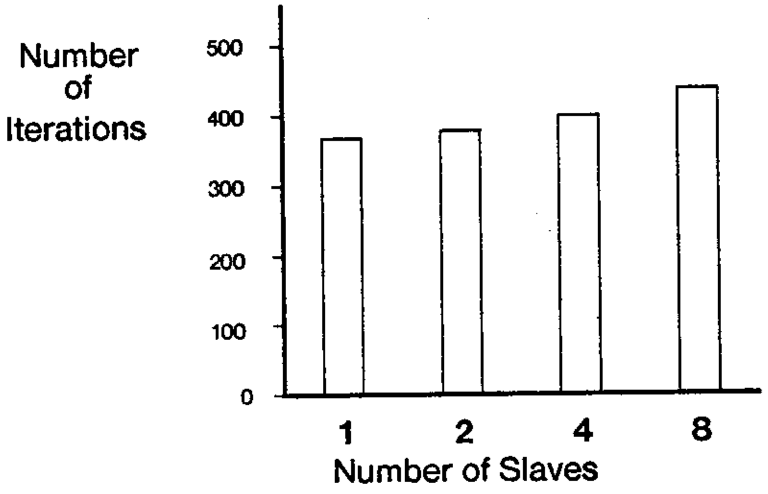


Figure 18: Number of iterations taken by the PDES program.

```
2 x (5 + 1) x 32 x 32 = 12288 bytes
```

per iteration. The total number of message bytes is summarized in Table 6 as a percentage of these local accesses. The overhead decreases as a function of the grid size.

number of slaves	message overhead
2	1.2%
4	3.4
8	8.0

Table 6: Relative overhead for messages in PDES.

To measure the effect of code quality on our results, the inner loop of the slave program was manually optimized by modifying the Bliss-11 code produced by the AMPL compiler. Subscript range checking was removed, and strength reduction was performed to eliminate multiplications in array accesses. Both of these optimizations could be performed by a modern optimizing compiler. The modified program was the version with two slaves and a single CP/AP pair. The execution time dropped from 531 to 75 seconds, a factor of 7 speed improvement. A small amount of time could also be saved by eliminating debugging statements, error detection code, and instrumentation, all of which are outside the inner loop for this particular program.

Other PDE programs have been implemented on Cm* using shared memory rather than messages to communicate between slave processes. A version which runs on Medusa takes 134 seconds to solve the problem for a 34 by 34 grid size using 1 slave and 89 seconds using 2 slaves. These figures indicate that shared memory does not provide a tremendous advantage over the message-based

AMPL programs for the PDE problem. It is difficult to make a more definitive statement since execution speed depends so strongly on code quality.

The Medusa version of the PDE program is asynchronous. It uses a separate processor for each slice, and there is no synchronization between processors. The grid memory is shared, so when a processor writes a new edge value, the value is immediately available to the processor's neighbor. In [22] it is shown how asynchronous PDE programs execute faster than unsynchronized ones. It is interesting to note that even the "asynchronous" algorithms rely on synchronization at the hardware level to prevent simultaneous reads and writes to memory words. If values were larger than the memory word size, say two-word floating-point numbers, the asynchronous algorithms would risk the possibility of reading corrupt data due to interleaved reads and writes of multiple-word values. On the other hand, synchronous algorithms and algorithms which do not used shared memory do not depend directly upon hardware-provided synchronization.

6.7 PDEA

The PDEA program solves the same problem as the PDES program, but in this version, the slaves run asynchronously. Each slave checks its ports for new edge values before each iteration. If an edge is present, the slave updates its grid. If no new edges are present, the slave performs an iteration using old values rather than waiting. Determining when the entire grid has converged is more difficult in the asynchronous version. The method used to determine convergence is discussed in in Appendix I.

6.7.1 Discussion

As shown in Figure 17 (shaded bars), the behavior of the PDEA program is more erratic than that of PDES. The rate of convergence seems particularly sensitive to variations in the process configuration. When the four-slave version is run on a single processor, an average of 522 iterations are required. When run on separate processors, the average number of iterations drops to 440. The explanation for this behavior is simple. A FIFO ready-to-run queue is used to schedule processes. A process is selected from the queue only when the current process suspends after sending a message, initiating a create operation, or attempting to accept a message from an empty buffer. In the PDEA program, the slaves on either edge of the grid have only one neighbor, and send only one message after each iteration. The slaves in the interior of the grid each have two neighbors and send two messages after each iteration. The interior slaves are therefore suspended twice as often as the two edge slaves and execute about half as many iterations if all process reside on a single CP/AP pair. The effect is maximal with four slaves, since half are edge and half are interior slaves. The maximum and minimum average number of iterations for differing numbers of slaves are shown in Figure 19.

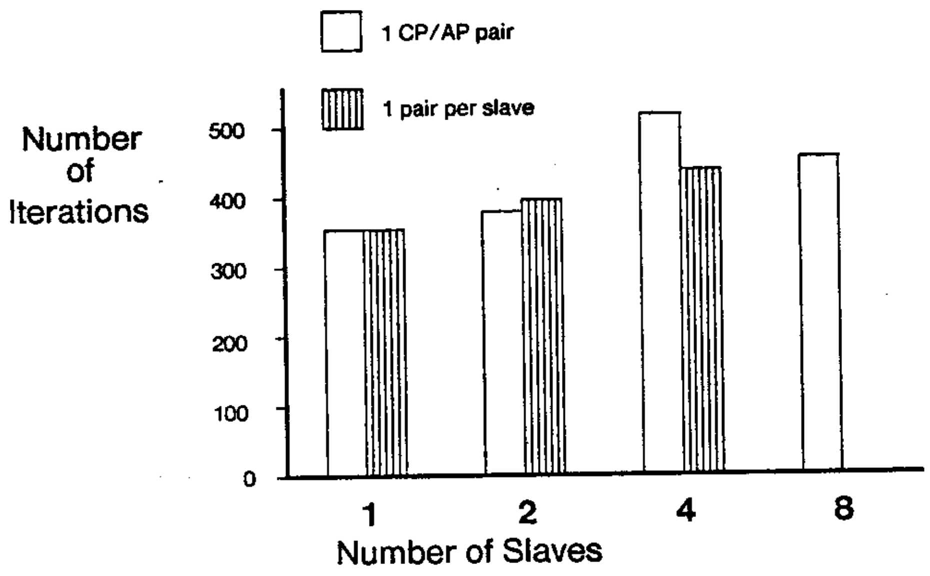


Figure 19: Number of iterations taken by the PDEA program.

In comparison with the PDES algorithm, the PDEA algorithm is slower. Apparently, the extra synchronization required by the PDES algorithm pays off by reducing the number of iterations required to achieve convergence.

6.8 Matrix multiplication

A parallel matrix multiplication algorithm was implemented in AMPL. (See Figure 20.) Two processes initially contain the two matrices to be multiplied. These matrices are divided into submatrices. On command from a master process, two submatrices are sent to a slave process which multiplies them and sends the product to a result process. Here, the submatrix products are summed to form the complete product matrix. When a slave is ready to perform a multiplication, it sends its name to the master. The port for these names is shown explicitly in the figure. The master uses these names to assign work to slaves, thus work assignments are nondeterministic and depend upon the relative execution speeds of the slave processes. The number of slaves can be altered by changing a constant and recompiling.

As with the PDE programs, several versions of the matrix multiply program were measured. In one set of tests, the number of slaves was varied while using a single CP/AP pair. In a second set of tests, the number of CP/AP pairs was chosen so that each slave ran on a separate processor.

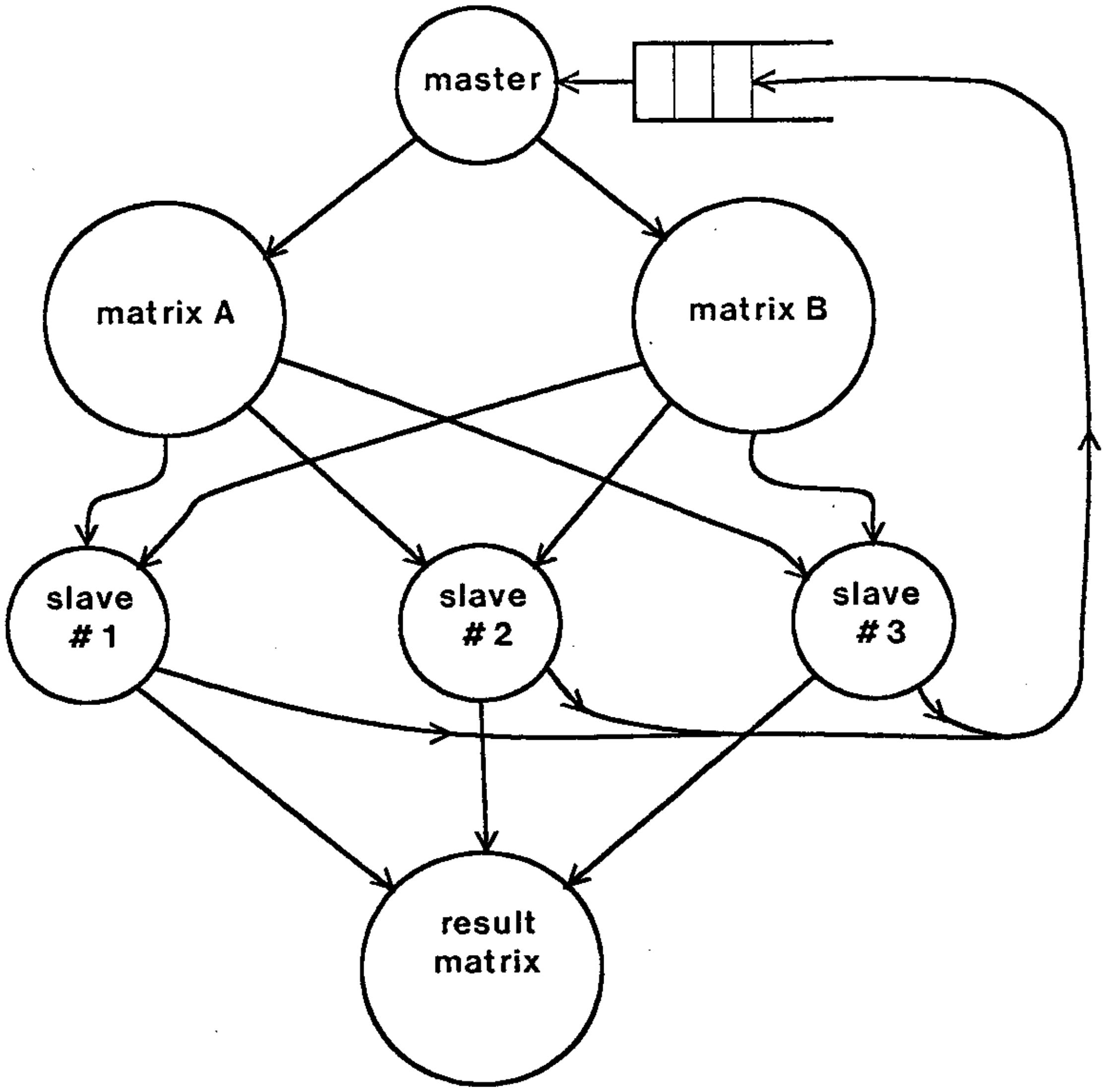


Figure 20: Structure of the MATR matrix multiplication program.

Measurements were started after all processes were created, and measurements ended after the product process produced a complete product.

Figure 21 shows real execution time and CPU time for varying numbers of slaves and processor pairs. Each test multiplied two 40 by 40 element integer matrices. The matrices were divided into 16 submatrices to be multiplied by the slaves. In tests with multiple CP/AP pairs, additional processors were used for the other processes, including the master process (scheduler), and result process (submatrix adder). The master and result processes take about 10s of AP and CP processor time. An additional 2.3s is taken by the two processes which supply the data to be multiplied. The result

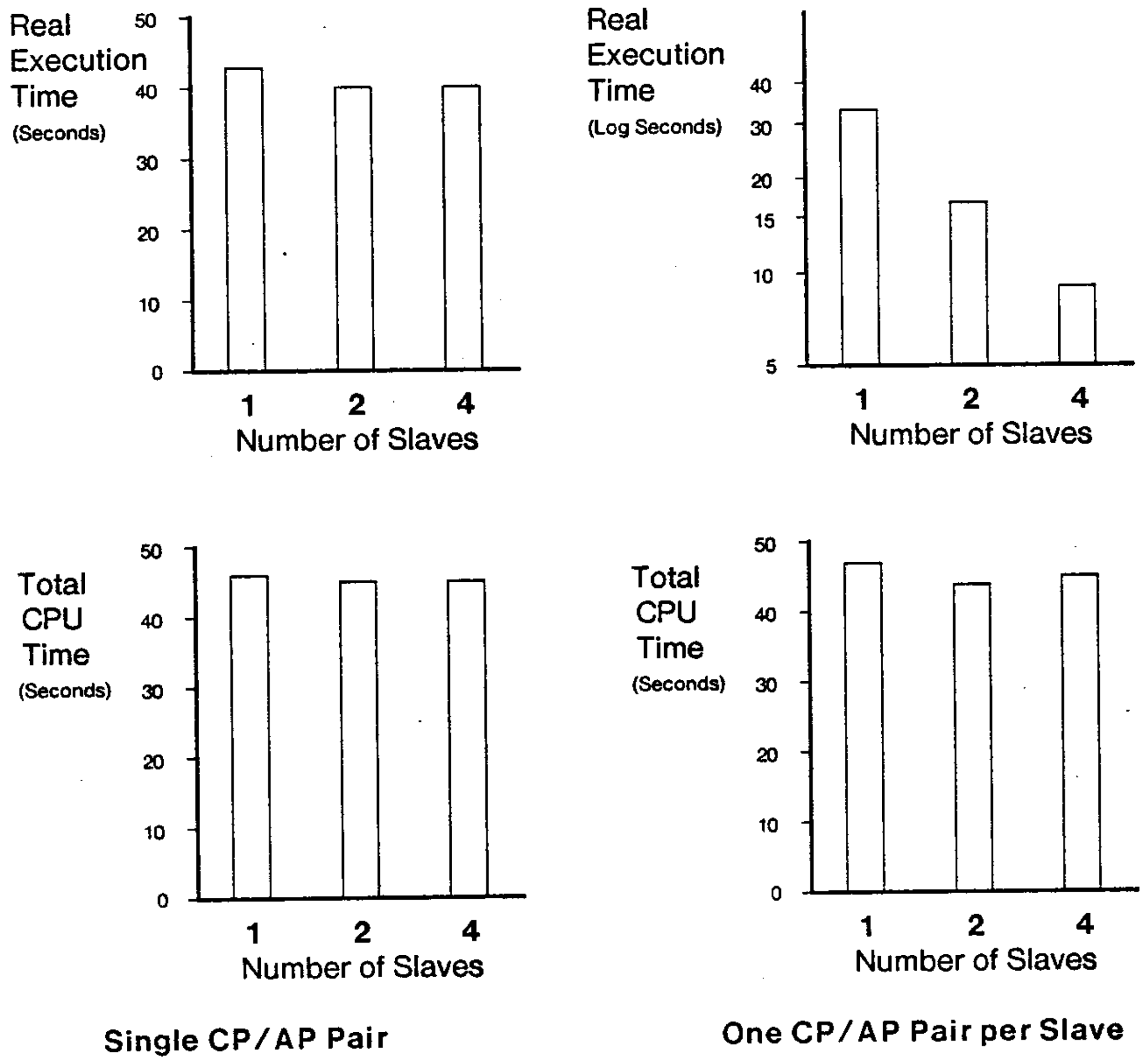


Figure 21: Measurements of the MATR matrix multiply program.

process execution time would determine the total execution time if more slave processes were added; however, the sums performed by the result process could be performed in parallel using a modification of our algorithm. Buffering in the result process allows the slaves to run without blocking, even with high utilization of the CP/AP pair executing the result process.

The message overhead for the matrix multiply algorithm can be computed easily. Each element in the product matrix is the product of two vectors of length 40. Each vector element is a two-byte integer, so 160 bytes are fetched and 2 are stored for each product matrix element. The total number of accesses is thus

162 x 40 x 40 = 259200 bytes.

The total number of message bytes is 39955, independent of the number of slaves. Data sent in messages is 15% of the theoretical minimum number of memory accesses. This number is a function of the matrix size and the number of submatrices. In our implementation of the matrix multiplication, slightly more local accesses are performed because vectors are not multiplied all at once, and many more accesses are made for array indexes. A more thorough analysis is presented in Appendix II.

6.9 Telegraph problem

This problem is taken from [27]. Telegrams are represented as words of text separated by blanks and terminated by the word "ZZZZ". The problem is to format telegrams for output by removing extra blanks and justifying the text. Details can be found in [27].

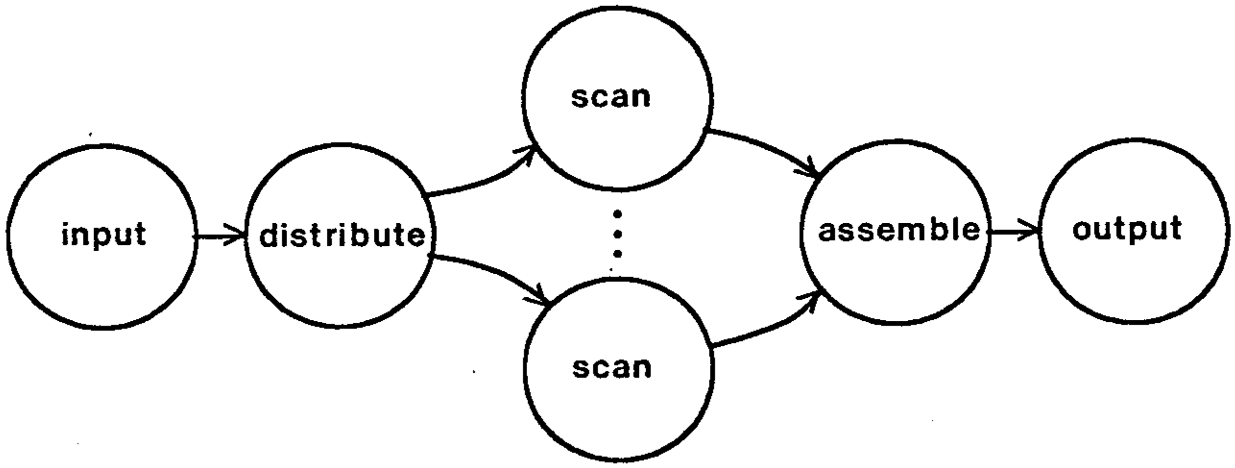


Figure 22: Structure of the TELE program.

A "pipeline" approach was used to solve the problem. (See Figure 22). Each line of input is sent to a *scanner* process which separates the text into words. Any number of scanners may be used. Scanners send the words and character counts on demand to an *assembler* process which fills output lines and sends them to an output process. In order to avoid interaction with physical I/O devices, an input process serves as a source of input text, and an output process serves as a "sink" for output text. As with other programs, measurements were begun after processes were created. We expected the addition of scanner processes to impose no noticeable increase in execution time on a single CP/AP pair. Figure 23 indicates that execution time increases slightly as more scanners are used.

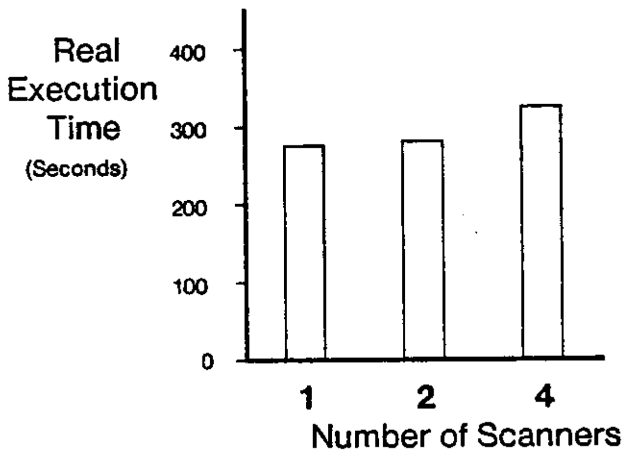


Figure 23: Execution times for the TELE program, using one CP/AP pair.

An initial set of experiments using separate CP/AP pairs for each process showed only a 19 percent speedup. It was found that the scanner was not scanning while the assembler formatted output because of inadequate buffering between the two processes. The number of messages buffered by the assembler process was increased so that the scanner could begin scanning the next line while the assembler formatted words of the current input line. This led to a speedup of 54 percent over the single CP/AP pair version.

We hoped to get speedup by using multiple scanners, since scanners do more work than other processes. Actual measurements show that this speedup is not realized. This leads to the hypothesis that the assembler process is a bottleneck. Measurements show that the utilizations of the CP and AP which serve the assembler process are not nearly one, although it could be that the CP and AP do not overlap their computations. If this were the case, the utilization of the AP could be low because it spends a significant amount of time waiting on the CP and vice versa.

6.10 Polynomial multiplication

A polynomial multiplication algorithm was adapted from an Algol 68 program [16]. Polynomials are represented as trees of processes. A pointer to a tree node is just a module name. The program creates and destroys many processes, and most of the computation time is spent by the CP's. The program is large and inelegant compared to the Algol 68 version. AMPL does not lend itself well to the functional style of programming required for this algorithm. The inclusion of procedures and a better syntax for invoking result-returning operations would greatly simplify the task of writing programs of this type. As an illustration, Figure 24 contains a fragment of the AMPL program. The module in the figure, "add", is used to add two polynomials. An "add" module is created for each addition. The addition algorithm is recursive, so the module creates new "add" modules if necessary.

The equivalent procedure in Algol 68 is illustrated in Figure 25. An extension to Algol 68 allows this procedure to execute with essentially the same degree of parallelism (see [16] for details). Many aspects of Algol 68 combine to account for the conciseness of this procedure:

1. The ability to define functions, e.g. the function "atoms".
2. User-defined infix and prefix operators, e.g. " + " and "even".
3. Expressions for structured values, used here to construct a value of type "poly".
4. The extension to provide parallelism, which is well-suited to this applicative program.

The first three factors could also be incorporated into an extended AMPL.

The AMPL polynomial arithmetic program creates 1340 processes and sends 2800 messages. When executed on a single CP/AP pair, the AP is idle 63 percent of the time and the CP is essentially never idle. Garbage collection is in progress 87 percent of the time. Versions of POLY for multiple CP/AP pairs cannot be run at the time of this writing due to a combination of hardware and software problems. Figure 26 summarizes the measurements obtained for the single CP/AP pair version.

7 Conclusions

AMPL demonstrates how message passing can be used to express interprocess synchronization and communication in a high-level language. The expressive power of the primitives in AMPL allowed us to develop programs with more complex control structures than previous Cm* application programs. While many synchronization errors were found in the run-time system as it was tested, the synchronization in all of the AMPL programs was correct from the start (which is not to say there were no other bugs!) We attribute the high level of correctness to several factors. First, the absence of shared memory forces the programmer to consider synchronization whenever processes interact. Second, type checking helps insure that data is interpreted correctly and that interfaces between processes are correctly implemented. Finally, since AMPL has built-in mechanisms for process creation and message passing, the programmer can avoid the implementation of tedious operating-system interfaces and concentrate on the problem at hand. In addition, the compiler can apply some checking to these operations.

```
module add(x,y: PolyType; sum: RefPolyType);
port
     OddPort:  PolyType(2);
     EvenPort: PolyType(2);
var
     poly1, poly2, new, zero: PolyType;
begin
     zero.atom := 0;
     if x.atom = 0 then send y to sum
     else if y.atom = 0 then send x to sum
     else if x.atom <> NotAtom and y.atom <> NotAtom then
          new.atom := x.atom + y.atom;
          send new to sum
     else
          if x.atom = NotAtom then
               send Self.EvenPort to x.ref.even;
               send Self.OddPort  to x.ref.odd;
          else
               send x   to Self.EvenPort;
               send zero to Self.OddPort;  end;
          if y.atom = NotAtom then
               send Self.EvenPort to y.ref.even;
               send Self.OddPort  to y.ref.odd;
          else
               send y   to Self.EvenPort;
               send zero to Self.OddPort;  end;

          accept OddPort(poly1); accept OddPort(poly2);
          create(add, poly1, poly2, Self.OddPort);

          accept EvenPort(poly1); accept EvenPort(poly2);
          create(add, poly1, poly2, Self.EvenPort);

          accept OddPort(poly1); accept EvenPort(poly2);
          if poly1.atom = 0 and poly2.atom <> NotAtom then
               send poly2 to sum
          else
               new.atom := NotAtom;
               new.ref  := create(PolyMod, poly1, poly2);
               send new to sum
               end
          end
          end
          end
     end;
```

Figure 24: A fragment of the AMPL polynomial arithmetic program.

Building AMPL on an existing operating system gave us some insight into what features an

```
op + = (poly x, y) sexp:
    if zerop x then y
    elif zerop y then x
    elif atoms (x, y) then new const (atom of x + atom of y)
    else new poly (odd x + odd y, even x + even y)
    fi;
```

Figure 25: Equivalent fragment of extended Algol 68 program.

Number of **send** statements executed	2800
Total bytes in SEND messages	15168
Number of CREATE statements executed	1340
Total bytes in **create** messages	25600
AP processor utilization	37%
CP processor utilization	99.67%
Number of **accept** statements executed	1980
Number of **select** statements executed	820
Garbage collection in progress	87%
Real execution time (seconds)	68

Figure 26: Measurements of the POLY program on a single CP/AP pair.

operating system should provide to support similar languages. Two approaches can be taken. In one approach, the operating system provides a basic set of abstractions without hiding the underlying physical machine from the user. An example of such an abstraction is a process. The power of the physical processor should not be restricted by the process abstraction [29]. If the user decides to implement his own abstractions, he will have the power and efficiency of the physical machine at his disposal. The lack of interrupts in Medusa is an example of a violation of this principle. If interrupts were available, we could have implemented the CP and AP together on a single processor. Without interrupts, polling is necessary to detect when work is available for the CP. Whenever the low-level machine is hidden, there is a danger of providing the "wrong" abstraction.

A second approach is to provide everything the user might need. In our case, this would require a number of abstractions not commonly found in operating systems. First, we would want to separately manipulate protected address spaces and processes to avoid the expense of allocating and deallocating a protected address space when processes are created and destroyed. The process descriptors should be accessible to the user so that a garbage collector could be implemented. Alternatively, the system could provide its own garbage collection. Small, dynamically created processes are necessary for an efficient AMPL implementation. There must be a flexible message-passing system with an arbitrary number of ports per process and the ability to selectively wait on sets of ports. Finally, this must all be as efficient as can be achieved by writing customized run-time routines, or the language implementor will probably reimplement the necessary facilities to achieve better performance.

A pattern or style of programming was observed in the test programs. Nearly all of our AMPL modules can be viewed as implementations of abstract types whose operations are invoked by sending a message to a port. If we were to design another language for parallel processing, we would attempt to use this as a basis for the language rather than message passing. The invocation of an abstract operation would normally be implemented by sending a message, but the actual construction of the message and its receipt would be hidden from the programmer. The compiler might then be able to detect special cases where simple mechanisms could be substituted for the full-blown message-passing scheme of AMPL. Section 2.5 gives an example of an optimization that a compiler might select automatically.

The potential for parallelism increases as we decompose a problem into more and finer operations. At some point, the overhead of invoking an operation becomes the an important factor. A computer architecture and run-time system might be developed to execute languages efficiently. In AMPL, messages are constructed by processes and delivered to a CP. There, the message header must be interpreted and the destination port located. When the message is finally accepted by the destination process, it is again interpreted, this time by user-written code, and some appropriate operation is performed. More work is needed to discover how this sequence of operations can be optimized. Work by Spector [32] and Nelson [26] gives encouraging evidence that we are just beginning to learn how to efficiently use computer networks and that large performance improvements can be obtained once critical functions are identified.

Although we had no problems in constructing programs with correct synchronization, none of our larger test programs were free of errors. Debugging a parallel AMPL program is considerably more difficult than debugging a sequential one. Since no memory is shared, it is difficult for any one process to determine much about the global state of the program. For example, when debugging the PDE program, we wanted to print snapshots of the grid to help locate a bug. Unfortunately, the grid is spread across several slaves, so the cooperation of several processes would be required to access it. New ports would have to be added to each slave to request access to the grid. Another problem is caused by the parallelism. For example, it is sometimes helpful to trace program execution by printing values of variables at run-time. Because of parallelism in AMPL, if several processes begin printing at the same time, output become hopelessly mixed up. Ordinarily, a collection of related debugging information must be packaged into a single message and sent to a printer process which formats and prints the data in a readable fashion. Obviously, more debugging aids are necessary. A symbolic debugger which could access all processes and monitor all messages would be a great help. There is also a need for methodologies which enable programmers to transform sequential programs into parallel ones. Most of the test programs we wrote are essentially sequential programs with a few simple modifications to subdivide and distribute the program. It would be nice to debug the sequential parts of the programs on sequential machines and to be able to take existing sequential algorithms and transform them to parallel programs.

The problems of storage allocation have been avoided to a large extent in the design of AMPL. An efficient implementation of a production language would require further investigation of the storage allocation problem. In particular, there may be special cases where modules can be represented with little memory overhead. In AMPL, message queues are allocated statically within process frames. Dynamic allocation might conserve memory and allow the **accept** statement to be implemented by changing pointers rather than by copying the message. Storage allocation is another area where a reduced overhead may allow finer grain processes and a higher degree of parallelism. Methods for efficiently allocating process frames are described in [25]. Using microcode support, process and procedure frames can be allocated from a heap with a very small overhead.

The problem of code location has been avoided in our AMPL implementation. Every processor has a complete copy of the compiled AMPL program. Memory could be used more efficiently if code could be placed in processors only as needed. Further efficiency is obtained when processes are located so that the number of copies of code is reduced. This savings must be balanced against the potential reduction in parallelism due to the choice of process locations.

Our experiments have indicated that a high degree of parallelism can be achieved in AMPL programs. The language supports the construction of parallel programs by providing high-level mechanisms (messages) for process synchronization and communication. By disallowing shared memory, we simplify the task of distributing processes and data while maintaining a low cost of communication. Our implementation presents a structure using separate processors for communication and program execution which provides further parallelism by overlapping communication and program execution.

8 Acknowledgements

My advisor, Peter Hibbard, helped me considerably to refine the design of AMPL and to keep the scale of this project to a reasonable level. Jarek Deminet, Pradeep Sindhu, John Ousterhout, and Rudy Nedved all answered innumerable questions about Medusa and helped track down bugs as the run-time system was designed and implemented. John Nestor, Aaron Wohl, and Jeff Baird helped me use the FEG system to construct the AMPL compiler. This work would not have been possible without the prior work of all those who designed and built CM*.

References

[1] W. B. Ackerman.
A structure processing facility for data flow computers.
In G. Jack Lipovski (editor), *Proceedings of the 1978 International Conference on Parallel Processing*, pages 166-172. IEEE Computer Society, August, 1978.

[2] Atkinson, R. and Hewitt, C.
Synchronization in Actor Systems.
In *4th Sigplan-Sigact Symposium on Principles of Programming Languages*, pages 267-280. January, 1977.

[3] Henry G. Baker, Jr.
List processing in real time on a serial computer.
Communications of the ACM 21(4):280-294, April, 1978.

[4] Toby Bloom.
Synchronization mechanisms for modular programming languages.
Technical Report MIT/LCS/TR-211, MIT, January, 1979.

[5] R. H. Campbell and A. N. Habermann.
Lecture Notes in Computer Science. Volume 16: *Path Expressions.*
Springer-Verlag, 1974.

[6] Robert Chansler.
personal communication.

[7] David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager.
Thoth, a portable real-time operating system.
Communications of the ACM 22(2):105-115, February, 1979.

[8] Robert P. Cook.
*MOD - A language for distributed programming.
IEEE Transactions on Software Engineering SE-6(6):563-571, November, 1980.

[9] Digital Equipment Corporation.
Microcomputer Handbook Series: *Microcomputer Processors.*
Digital Equipment Corporation, Maynard, Massachusetts, 1978.

[10] Edsger W. Dijkstra, et. al.
On-the-fly garbage collection: an exercise in cooperation.
Communications of the ACM 21(11):966-975, November, 1978.

[11] Edsger W. Dijkstra.
Finding the correctness proof of a concurrent program.
In *Program Construction*, pages 24-34. Springer Verlag, New York, 1979.

[12] *Reference Manual for the ADA programming language*
United States Department of Defense, 1980.

[13] John R. Nestor and Margaret A. Beard.
Front End Generator User's Guide
Carnegie-Mellon University, 1981.

[14] W. Morven Gentleman.
Message passing between sequential processes: the reply primitive and the administrator concept.
Software - Practice and Experience 11(5):435-466, May, 1981.

[15] P. Brinch Hansen.
Distributed processes: A concurrent programming concept.
Communications of the ACM 21(11):934-941, November, 1978.

[16] P. G. Hibbard, P. Knueven, B. W. Leverett.
Issues in the efficient implementation and use of multiprocessing in Algol 68.
In *Proceedings of the 5th Annual Conference on Implementation and Design of Algorithmic Languages*, pages 203-221. IRISA, May, 1977.

[17] Peter G. Hibbard.
personal communication.

[18] C. A. R. Hoare.
Monitors: An Operating System Structuring Concept.
Communications of the ACM 17(10):549-557, October, 1974.

[19] C. A. R. Hoare.
Communicating sequential processes.
Communications of the ACM 21(8):666-677, August, 1978.

[20] Honeywell, Inc. and Cii Honeywell Bull.
Reference manual for the ADA programming language.
SIGPLAN Notices 14, Part A(6), June, 1979.

[21] A. K. Jones, et. al.
StarOS, a multiprocessor operating system for the support of task forces.
In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 117-127. SIGOPS, 1979.

[22] Anita K. Jones, and Edward F. Gehringer, editors.
The Cm Multiprocessor Project: A Research Review.*
Technical Report CMU-CS-80-131, Carnegie - Mellon University, July, 1980.

[23] Donald E. Knuth.
The Art of Computer Programming.
Addison-Wesley, 1973.

[24] H. T. Kung and S. W. Song.
An efficient parallel garbage collection system and its correctness proof.
Technical Report, Carnegie-Mellon University, September, 1977.

[25] Butler W. Lampson and David D. Redell.
Experience with processes and monitors in Mesa.
Communications of the ACM 23(2):105-117, February, 1980.

[26] Bruce Jay Nelson.
Remote Procedure Call.
PhD thesis, Carnegie-Mellon University, May, 1981.

[27] R. E. Noonan.
Structured programming and formal specification.
IEEE Transactions on Software Engineering 1(4):421-425, December, 1975.

[28] John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu.
Medusa: an experiment in distributed operating system structure.
Communications of the ACM 23(2):92-105, February, 1980.

[29] David L. Parnas and Daniel P. Siewiorek.
Use of the concept of transparency in the design of hierarchically structured systems.
Communications of the ACM 18(7):401-408, July, 1975.

[30] Richard Rashid.
An Inter-Process Communication Facility for UNIX.
Technical Report, Carnegie - Mellon University, March, 1980.

[31] Richard Rashid, George Robertson.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 64-75. December, 1981.

[32] Alfred Z. Spector.
Performing Remote Operations Efficiently on a Local Computer Network.
Communications of the ACM 25(4), April, 1982.

[33] Richard J. Swan.
The switching structure and addressing architecture of an extensible multiprocessor, Cm.*
PhD thesis, Carnegie-Mellon University, August, 1978.

[34] Jan van den Bos, Rinus Plasmeijer, and Jan Stroet.
Process communication based on input specifications.
ACM Transactions on Programming Languages and Systems 3(3):224-250, July, 1981.

[35] N. Wirth.
Design and implementation of Modula.
Software - Practice and Experience 7(1):67-84, 1977.

[36] N. Wirth.
The programming language Pascal.
Acta Informatica 1:35-63, 1971.

[37] N. Wirth.
Modula: A programming language for modular multiprogramming.
Software-Practice and Experience 7(1):3-35, Jan, 1977.

I. Synchronization of relaxation algorithms

Consider a number of processes, each of which operates on a partition of a distributed data structure. The task of the processes is to transform the structure into one that satisfies some global constraint. The global constraint is satisfied when a local constraint is satisfied by each partition of the data structure, and all communication between processes has completed.

We have found two examples of this sort of parallel computation. One is the PDEA program. Here, the global constraint is specified by the finite difference equations. The global data structure is the grid, and slices of the grid form the partition. The local constraint is that the finite difference equation is satisfied by the local slice of the grid. The requirement that communication between processes completes insures that the difference equations are satisfied across slice boundaries.

Another example is the scan phase of the parallel garbage collector. Here, the system-wide heap memory is the global data structure, and the constraint to be satisfied is that all reachable objects are marked. The local constraint is that there are no pending requests to mark local objects, and a mark request has been sent for each reference in a locally marked object.

To illustrate the problem, we will first describe a simple but incorrect "solution":

> Whenever local constraints are satisfied, send a *done* message to a globally known process. When this process receives a *done* message from each of the cooperating processes, then the global constraint is satisfied.

The problem with this approach is that local constraints may become *not* satisfied due to the actions of some other process. Only after all local constraints are satisfied *and* all communication between processes has terminated can we conclude that local constraints will remain satisfied.

Now, suppose a *not done* message is sent to the globally known process whenever it is detected that a local constraint has become not satisfied. This leads to a race condition, where correct synchronization depends upon relative speeds of computation and communication. Dijkstra has proposed a solution to this problem [11]. In Dijkstra's solution, synchronization overhead is incurred every time processes communicate. The PDEA program performs the synchronization correctly and has the property that no synchronization messages are sent by a slave until the first time its local constraints are satisfied, i.e. the slice values have converged and no messages from other slaves have arrived. This algorithm is described below.

When a slave process operating on a slice of the grid detects local convergence, it sends its index and status (*converged* or *not converged*) to the master process, which is known to all the slaves. If a slave must perform further iterations after sending a *converged* status to the master, it sends a *not converged* status message before proceeding. The master keeps a count of how many slaves have converged. When this count reaches the number of slaves, the master asks all slices to confirm that they are still in the *converged* state.

If a slave's status is still *converged* when the *confirm request* arrives, the slave sends a *confirm reply* to the master. If the master receives replies from all slaves, global convergence has been achieved. On the other hand, if a slave changes status to *not converged*, then it will send a new status message to the master. Some care must be taken to assure that this status change will be detected and sent to the master before a reply can be made to a *confirm request*. Upon receiving a status message of *not converged*, the master decrements its counter, and ignores replies to the last *confirm request* message. Eventually, the counter will again reach the number of slaves. The master will send another *confirm request* message to each slave and await the replies. A unique key is used with each set of *confirm request* messages to distinguish replies. The structure of the slave program is given in Figure 1. The complete PDEA program is listed in Appendix III.3.

```
initialization of variables
converged := False
while true do
     if not converged and there are no messages then
          perform one iteration, setting converged
          if converged then
               send done to master
          else send new edges to neighbors
               end
     else
          select
               when no edge messages present accept confirm message do
                    if converged then send response to master end
                    end
               accept edge message do
                    if converged and new edge <> old edge then
                         send not done to master
                         converged := False
                         update old edge
                         end
                    end
               end
          end
     end
```

Figure 1: Structure of the slave program.

To show that this algorithm is correct, consider the sequence of events which must occur before the master detects global convergence. First, all slaves must achieve local convergence and send their status to the master. The master then sends requests to each slave. Then, each slave must send a reply to the master. Suppose there are N slaves and the master's counter contains N, but not all slaves have converged. It must be the case that a slave has not yet detected that a new message has caused it to become *not converged*.

The new message must have been sent before the last *converged* status message was sent to the master. AMPL **send** semantics guarantee that the new message is delivered before the last status message. Slaves give lowest priority to confirmation requests, so the new message is received before the confirmation request. Since the new message causes a status change, a status of *not converged* is sent to the master before the confirmation request is answered. The master receives status reports and confirmation replies in order, so the status of *not converged* will be received by the master before it can receive N confirmation replies. Therefore, the receipt of N replies indicates that the slaves have in fact converged, and all communication between slaves has completed.

Another way to view this algorithm is in terms of time delays. The slave processes send status reports to the master, but there is a time delay between the point at which status changes and the time at which the master is informed of the change. If all slaves report convergence to the master, and no reports to the contrary are received within the time required for the changes to be reported, then the master can safely assume that all the slaves have finished. To make sure that the master waits long enough, a set of *confirm requests* are sent to the slaves. The time taken for all slaves to reply to these requests is longer than the time required to detect and report a status change.

This is a complicated algorithm, but the savings can be large. The extra confirmation step allows cooperating processes to communicate without special synchronization until local convergence is obtained.

II. Matrix multiplication analysis

In the MATR program, the number of slaves was varied while the partitioning of the matrices into submatrices was held constant. Here, we consider the effect of changing the partition. Call the two matrices to be multiplied A and B. The matrices A and B are assumed to be N by N and are partitioned as shown in Figure 1. Matrix A has submatrices with dimensions N/q by N/r. Submatrices of B have dimensions N/r by N/s.

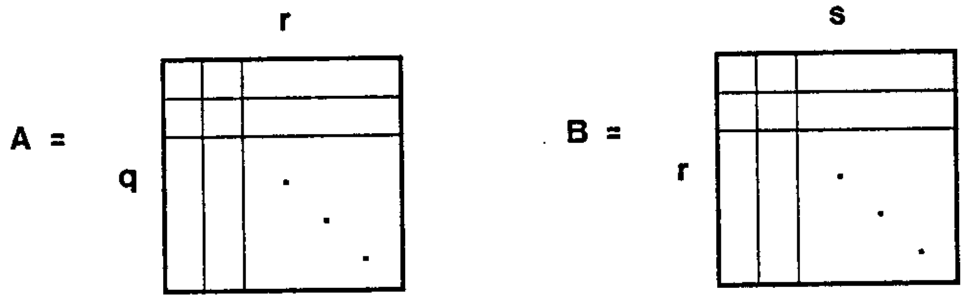


Figure 1: Two partitioned matrices.

The multiplication algorithm is:

> Send pairs of submatrices from A and B to a multiplication process. Send the submatrix products to a result process which performs submatrix additions to form the final product.

For example, let

$$A = \begin{matrix} S & T \\ U & V \end{matrix} \quad \text{and} \quad B = \begin{matrix} W & X \\ Y & Z, \end{matrix}$$

where S through Z are submatrices. Multiplication processes are used to compute the following subproducts:

$$SW,\ TY,\ UW,\ VY,\ SX,\ TZ,\ UX,\ VZ$$

These subproducts are combined by the result process to form:

$$\begin{matrix} SW + TY & SX + TZ \\ UW + VY & UX + VZ \end{matrix}$$

which is the product of A and B.

The *communication cost* is defined as the amount of data sent from A and B to slaves, plus the data sent from slaves to the result process. The unit of measure is the number of bits required to represent a single matrix element. Sparse matrices and variable length encodings of matrix elements are not considered.

The *potential parallelism* is determined by the number of submatrix multiplications. If the submatrices are small, the result process could become a bottleneck, limiting the effective parallelism. In this case, a more elaborate algorithm could be developed using multiple adder processes. this would not affect the communication cost or potential parallelism derived from our simpler model.

II.1 Potential parallelism

The number of subproduct multiplications, thus the potential parallelism, is qrs. In the special case where submatrices have dimensions 1 by 1 (corresponding to the conventional matrix multiplication algorithm), we have $q = N$, $r = N$, $s = N$, and $qrs = N^3$. The result process must perform only $O(rN^2)$ additions, while slaves perform a total of N^3 multiplications, so the relative amount of work performed by the result process is proportional to r, and inversely proportional to N.

II.2 Communication cost

The size of submatrices in A is N^2/qr. The size of submatrices in B is N^2/rs. The total number of matrix elements sent to the slaves is

$$pqr(N^2/qr + N^2/rs) = (s + q)N^2$$

. The size of submatrix products are N^2/qs so $qrs(N^2/qs)$ elements are sent from slaves to the result process. Therefore, the total communication cost is simply:

$$(q + r + s)N^2$$

III. Selected program listings

III.1 SNDS

```
module MAIN;
port
      answer: boolean;
      howmany: integer;
var
      b: boolean;
      i: integer;
begin
      while TRUE do
            send 'Enter number of sends: ' to WrStr;
            send self.howmany to RdInt;
            accept howmany(i);
            while i > 0 do
                  send TRUE to self.answer;
                  accept answer(b);
                  i := i - 1
                  end
            end
      end
```

III.2 GC

```
const
     CreateCmd  = 1;
     DestroyCmd = 2;

module main;
port
     CommandPort: integer;
var
     I: integer;
     Command: integer;
     NewMod:  refmod DummyMod;
begin
     while TRUE do
          send 'Command: (1) create (2) destroy ' to WrStr;
          send Self.CommandPort to RdInt;
          accept CommandPort(Command);
          if Command = CreateCmd then
               send 'How many creates:' to WrStr;
               send Self.CommandPort to RdInt;
               accept CommandPort(I);
               while I > 0 do
                         NewMod := create(DummyMod);
                         I := I - 1;
                         end;
               send 'created' to WrStr;
               end;
          if Command = DestroyCmd then
               NewMod := NIL;
               send 'destroyed' to WrStr;
               end;
          send 1 to WrLn;
          end
     end;

module DummyMod;
port IP: integer;
var I: integer;
begin
     accept IP(I)                    {indefinite wait}
     end
```

III.3 PDEA

```
{PDEA}

const   ColPerSlice = 6;
        NSlices = 8;
        NSlicesPlus1 = 9;
        Nrows = 34;
        UNITY = 1000;       {for fixed point simulation}

typeSliceType = array [1 .. ColPerSlice] of ColType;
        ColType   = array [1 .. Nrows] of Integer;
        ColRef   = refport ColType;
        NeighborsType = record Left,Right: ColRef end;
        ReportType = record From: integer;
                            Cycles: integer;
                            IsDone: boolean end;
        RefReportType = refport ReportType;
        RefInt = refport Integer;
        RefMain = refmod Main;
        KeyType = integer;

module SliceMod(myindex: integer; master: RefMain);
port
    StartUp: NeighborsType;
    LeftIn: ColType(2);
    RightIn: ColType(2);
    Confirm: KeyType;
var
    slice: SliceType;               {part of grid}
    column: ColType;                {receives data from LeftIn, RightIn}
    status: ReportType;             {sent to master after each iteration}
    nghbrs: NeighborsType;
    temp:  integer;                 {new value for grid}
    delta:  integer;          {change in grid value}
    change, converged: boolean;
    row,col: integer;
    key:   KeyType;
begin
    col := 1;         {initialization: edges get 1, center gets 0}
    while col <= ColPerSlice do
        slice[col][1] := UNITY;             {row 1}
        row := 2;                        {rows 2 thru Nrows-1}
        while row < Nrows do
            slice[col][row] := 0;
            row := row + 1
            end;
        slice[col][Nrows] := UNITY;         {row Nrows}
        col := col + 1;
```

```
     end;

     {special case for left slice}
if myindex = 1 then       {fill in left edge}
     row := 1;
     while row <= Nrows do
          slice[1][row] := UNITY;
          row := row + 1
          end
     end;

     {special case for right slice}
if myindex = Nslices then       {fill in right edge}
     row := 1;
     while row <= Nrows do
          slice[ColPerSlice][row] := UNITY;
          row := row + 1;
          end
     end;

accept StartUp(nghbrs);
status.From := myindex;
status.IsDone := FALSE;
status.cycles := 0;
converged := FALSE;      {force first iteration}
while TRUE do
     if not (converged or ready(LeftIn) or ready(RightIn) or ready(Confirm))
     then {perform one iteration}
          converged := TRUE;
          col := 2;
          while col < ColPerSlice do
               row := 2;
               while row < Nrows do
                    temp := (slice[col+1][row] + slice[col-1][row] +
                                   slice[col][row-1] + slice[col][row+1] + 2)/4;
                    delta := temp - slice[col][row];
                    slice[col][row] := temp;
                    if delta <> 0 then converged := FALSE end;
                    row := row + 1
                    end;
               col := col + 1
               end;
          status.Cycles := status.Cycles + 1;
          if converged then
               status.IsDone := converged;
               send status to master.report;
          else
          if myindex > 1 then
               send slice[2] to nghbrs.Left end;
               if myindex < Nslices then
```

```
                    send slice[ColPerSlice-1] to nghbrs.Right end;
                end;
        else
            select
                accept LeftIn(column)  then col := 1        end;
                accept RightIn(column) then col := ColPerSlice end;
                when not ready(LeftIn) and not ready(RightIn)
                accept Confirm(key) then
                    if converged then
                        send key to master.confirm
                        end;
                    col := 0
                    end {accept}
                end;    {select}
            if col <> 0 then        {new column}
                row := 1;
                change := FALSE;
                while row <= Nrows and not change do
                    if Slice[col][row] <> column[row] then change := TRUE end;
                    row := row + 1
                    end;
                if change then
                    if status.IsDone then
                        status.IsDone := FALSE;
                        send status to master.report
                        end;
                    Slice[col] := column;
                    converged := FALSE
                    end
                end
            end
        end {while loop}
    end;        {module}


module main;
port
    (Report: ReportType(2);
    Confirm: KeyType(2));
    IntPort: Integer(1);
var
    I: integer;
    converged: boolean;
    cycles: array [1 .. NSlices] of integer;
    slices: array [0 .. NSlicesPlus1] of refmod SliceMod;
    Ndone, Nconfirmed: integer;
    ConfirmKey: KeyType;
    nghbrs: NeighborsType;
    r: ReportType;
```

```
begin
     I := 1;
     while I <= NSlices do
          Slices[I] := create(SliceMod, I, Self);
          cycles[I] := 0;
          I := I + 1 end;

     send Self.IntPort to RdInt;      {delay start until terminal input}
     accept IntPort(I);

     I := 1;
     while I <= NSlices do
          nghbrs.Left := slices[I-1].RightIn;
          nghbrs.Right := slices[I + 1].LeftIn;
          send nghbrs to slices[I].StartUp;
          I := I + 1 end;

     converged := false;
     ConfirmKey := 0;
     while not converged do
          select
               accept Report(r) then
                    cycles[r.From] := r.cycles;
                    if r.IsDone then Ndone := Ndone + 1 else Ndone := Ndone - 1 end;
                    if Ndone = Nslices then
                         ConfirmKey := ConfirmKey + 1;
                         I := 1;
                         while I <= Nslices do
                              send ConfirmKey to Slices[I].Confirm;
                              I := I + 1 end
                    else Nconfirmed := 0 end;
                    end;
               accept Confirm(I) then
                    if I = ConfirmKey then Nconfirmed := Nconfirmed + 1 end;
                    if Nconfirmed = Nslices then converged := TRUE end
                    end
               end
          end;

     send 'PDE completed:' to WrStr;      send 2 to WrLn;
     send 'SLICE #    ITERATIONS' to WrStr; send 1 to WrLn;
     I := 1;
     while I <= Nslices do
          send I to WrInt;
          send '        ' to WrStr;
          send cycles[I] to WrInt;
          send 1 to WrLn;
          I := I + 1 end
     end
```