EC : A Language for Distributed Computing

Ashok K. Naik

Gautam Barua

Department of Computer Science and Engineering

IIT Kanpur - 208016, India

email - {naik, gb}@iitk.ernet.in

Abstract

Computation intensive programs can utilise idle workstations in a cluster by exploiting the parallelism inherent in the problem being solved. A programming language for distributed computing offers advantages like early detection of type mismatch in communication and offers structured mechanisms to specify possible overlap in communication and computation, and exception handling for catching run time errors. Success of a language depends on its ease of use, expressibility and efficient implementation of its constructs.

EC is a super set of C supporting process creation, a message passing mechanism, and exception handling. The pipelined communication constructs and multiple process instances help in expressing concurrency between computation and communication. Data driven activation of EC processes is used for scheduling.

EC has been implemented in a Sun-3 workstation cluster. An inter-node message passing mechanism has been built on top of the socket interface using the TCP protocol, and intra-node message passing is done by passing pointers to improve efficiency. However, **message_type** variables hide the implementation details, improve type safety and location transparency of a program.

Keywords

Distributed Languages, process, exception, pipelined communication

1 INTRODUCTION

With the decrease in the price of workstations along with their increase in computational power, workstation clusters are becoming an alternative to centralized computing platforms. But as the current generation operating systems allocate only one workstation to a user, most of the workstations are under-utilised. There have been attempts to distribute the load on a per task basis (Shivaratri *et al.*, 1992), but this does not reduce the computational time of large scientific and engineering applications unless the program is broken into smaller tasks. This requires considerable effort on the part of the programmer without any further support. Research in this area has been going on for quite sometime. The availability of fast, inexpensive systems, interconnected by fast networks, has given an added impetus to this area in recent areas. Commercial interest is being shown by a number of leading companies, such as Digital, HP-Convex, and IBM, in such clusters of workstations, as a low-cost alternative to vector machines. The typical commercial offering consists of a number of computer systems interconnected by a fast network such as FDDI. The software offering with such systems is the weak link, with the most common offering being a library of routines to help in the inter-node communication. What we have is a distributed system, with no sharing of memory, and with the need for a relatively coarse-grain level of parallelism to exploit the number of CPUs present.

Two main methods are being experimented with at various places. One is to use distributed shared memory to provide a shared memory model of multiprocessors to the users. The other is to provide primitives for explicit communication among processes in different processors, either by providing library functions or through distributed programming languages.

Some languages use sharing as an abstraction to hide issues related to communication. But, for efficient implementation, access to the lower level system software such as memory management unit, lower level communication primitives etc. is required, which is not possible in many cases. Also, the languages based on the shared memory paradigm provide support for synchronization, leaving the load distribution to the user and message traffic optimization is necessary to maximize the throughput. As problem specific optimization can not be done at runtime, performance depends upon the message volume generated by the heuristic used. This approach, being evolutionary in nature, may be suitable for using existing programs where portability rather than efficiency is of major concern. A variation of this is to use virtual distributed machines where a program includes definitions for data distribution mapping and program mapping. Fortran-D (Hiranandani *et al.*, 1991) uses data distribution mapping information to partition a sequential program and generate code for communication. DINO (Rosing *et al.*, 1990), PANDORE (Andre *et al.*, 1990), Spot (Socha, 1990) and Ensemble (Griswold *et al.*, 1990) also use these mapping information to generate a distributed program. Inefficiencies arise in dealing with

irregular data like sparse matrices (Lu *et al.*, 1991). As the issues like data partitioning, communication optimization etc. are in NP-Complete, a programming language should provide facilities to allow programmers to exploit problem specific features.

UNITY (Mishra, 1991) is an abstraction mechanisms for data parallel programs. A UNITY program is essentially a declaration of variables and a set of assignment statements. Program execution consists in selecting nondeterministically some assignment statements, executing it and repeating it forever. Program development is carried out in two basic steps: first a correct program is derived from specifications and this program is adapted to the target architecture by successive transformations of the original program in order to make the control structure explicit. UNITY uses abstract models for program development based on variant and invariant parts in the specification of a program. A naive implementation will result in inefficient programs. LINDA (Gelernter, 1991) is a language using an abstraction based on tuple space. All operations on tuple space are managed through predefined primitives. As access to the tuple space is based on key matching, all access to tuple space in LINDA programs are optimized by a compiler (Carrierio *et al.*, 1991). The languages based on these abstraction mechanisms rely too much on compiler technology to generate efficient codes, which we believe is not yet mature enough to do this.

Programming in distributed systems using explicit communication primitives is difficult because of issues like data distribution, correct use of communication primitives, buffering, hiding communication latency etc. Moreover, when implementation dependent features like buffering are not hidden from programs, it becomes nonportable. It is possible to build an abstraction at the user level to provide portability. For example, PVM (Beguelin *et al.*, 1993) builds a virtual parallel machine model on top of existing systems at the user level to provide the portability. But being a library based approach, it lacks features like compile time type checking and language support for latency hiding. So developing distributed memory programs using libraries is still difficult and error-prone. In a language based method on the other hand, a virtual machine model can be adopted and by porting the compiler, programs can be run on a variety of architectures. A language for distributed systems aims to be expressive enough to program parallelism, to provide scalability and reusability of modules, and compile time and run time checks to detect any incorrect use of communication primitives. Process is the lowest level of abstraction used for expressing parallelism and it fits quite naturally with a distributed memory computer. Because of this, it can be implemented efficiently. In this abstraction, processes must be created and 'connections' must be established according to topology requirements of an algorithm used. Communication primitives may be of synchronous or asynchronous type. But synchronous primitives with static process creation limits the concurrency of local processes (Liskov *et al.*, 1986). This being the oldest abstraction in use, there are many languages based on this paradigm. The most recent proposals include Hermese (Strom

et al., 1991) and Darwin/MP (Magee *et al.*, 1992). In the past, many programming languages based on the message passing paradigm have been proposed. Many of them, i.e. OCCAM (Burns *et al.*, 1988), CONIC (Magee *et al.*, 1989) etc. are meant to be for Transputer based machines. Some of the early languages such as SR (Andrew *et al.*, 1988), Argus (Liskov *et al.*, 1983) etc. were more concerned about transaction oriented or system programming applications. Most of these languages have a common heritage, the guarded commands of CSP. One of the motivations for this was proving program correctness. Another motivation was to provide overlap in computation and communication through the nondeterminism in the guarded commands. Languages such as BSP (Gehani, 1984), DP (Hansen, 1978) are extension of CSP. We believe, these languages do not aid the compiler by providing appropriate information suitable to exploit the data parallelism present in scientific and engineering programs. For example, as guarded commands can not proceed until full data is available, the overlap of computation with communication can not take place.

Object based and object oriented languages for distributed systems use objects as an abstraction mechanism (Wegner, 1991; Wegner, 1992) In this abstraction, an object is used as a structuring mechanism for complex data structures. Concurrency in an object model has three different forms ie, sequential, quasi-concurrent and concurrent. In the first case there is only one thread of execution. In the second form, an object acts as a monitor; so all threads are queued except one. In the last case, multiple threads may be active at the same time. To express concurrency at the calling side, asynchronous method invocation is used if the result is not required immediately; otherwise future-based synchronization is used.

Declarative languages are based on the functional paradigm in which the order of execution is not specified. Instead, a declarative program specifies what to do. Thus execution order of an operation in dataflow style is data driven, that is execution starts once all operands are available. To reduce synchronization overhead for scheduling an instruction, a sequence of instructions called a thread, forms a unit of data driven computation (Gao, 1993; Grit, 1990) . In process based functional programming languages, a *future* (Halstead, 1989) type function is used to provide asynchronous features and *continuation* based function calls (Hieb *et al.*, 1990) provide coroutine semantics to a function call. To provide concurrent access to shared objects, objects with state called a *domain* (Kessler *et al.*, 1989) which is similar to a monitor, is used.

All distributed applications are derived from problems that are inherently distributed in nature. For example, differential equations representing the physical system do not have action at a distance. The interaction is always with the immediate neighbours. This allows the problem to be solved using divide and conquer, quite naturally. This feature makes object based or process based approach natural candidates. Our approach to the design of a language for distributed systems is evolutionary rather than revolutionary. We

decided to use C as base language as it is popular and efficient and added the features that are orthogonal and easy to learn. So we have chosen the process abstraction as it is simple and easy to integrate with C. The proposed language, Extended C (EC), adds process and typed message variables to C and augments control structures by adding pipelines and process creation. Type checking of messages detect possible mistakes in communication. The process topology as required by the algorithm is specified at creation time.

2 BACKGROUND

Our experiments on numerical programs using DC (Subramaniam, 1991), a language for distributed computing, showed that performance depends very much on the amount of overlap in computation and communication. Sending large sized messages resulted in CPUs on remote nodes remaining idle, waiting for data transfer to be completed. So large sized messages had to be broken into smaller messages to improve the efficiency. But as this approach becomes cumbersome, some language support for specifying the concurrency in communication and computation was thought to be necessary.

From the user point of view, it was found that the point-to-point communication facility provided by the language was not convenient to use. Major portion of a typical program was for supporting communication, that is, connection set up and communication statements. It was felt that language level support for different communication patterns such as one-to-many and many-to-one type of communication will simplify programming.

3 THE LANGUAGE

3.1 Interprocess Communication

To allow a programmer to specify overlap in computation and communication, either asynchronous communication primitives or synchronous primitives with dynamic process creation has to be used. EC uses a hybrid of asynchronous and synchronous methods because synchronous versions are easy to understand and allow error handling to be specified as a part of the primitive. This form simplifies implementation of termination semantics of any exception raised during a receive. By specifying the computation to be performed while waiting for data, it allows the efficient use of CPU time. A similar construct has been proposed for RPC (Chang, 1989).

3.1.1 Message Variables

All communication is done through message variables. Message variables in EC are used as typed streams. A message variable is a tuple with two components: one component specifies the binding with the remote message variable, if any, and the other component specifies the buffer to be used for communication. No actual space is allocated for the

buffer. This is done separately. A message variable is a first class variable, so a message variable can be passed from one process to another. This allows the connection topology to be changed after process creation. This allows a child process to do the processing on behalf of its parent. This is very useful in client server applications where there are multiple specialized servers and a client does not know which server to use.

Type checking in communication, like type checking of other variables, ensures that both sender and receiver communicate with messages of the same data type and size. EC defines message variables which act as typed streams, and like structures in C, name equivalence form of type checking is used for message variables. A `message_type` declaration creates a new message type whose components specify the message composition. The syntax for a message type declaration is similar to a struct as shown below.

```
message_type msg1 {
    int a, b;
    struct my_struct y[100];
} x;
```

But unlike a struct type, pointers are not allowed inside it because size information is not always present with a pointer (a pointer of type char is used in many C programs to point to arrays), and without this, type checking is not possible. Dynamic arrays can not be used either, because unlike a struct variable, scope of temporaries needed for message variables is within communication primitives and their sizes may vary from one primitive to another with dynamic arrays present, so some other linguistic mechanism must be provided to support this. Compile time check alone will not suffice because the sender and receiver sides may be separately compiled and size information is available at the receiving end only after data has been received. Further, EC allows a message type to be parameterized, with parameters specifying the size of variable sized components. So a run time check at the receiving end must be done to ensure that the correct amount of data has been received. The following example shows a declaration of a parameterized message type.

```
message_type msg2 (m) {
    double x[m];
} z;
```

We will discuss the syntax and semantics of send and receive using parameterized messages later. EC extends parameterization to `message_types` nested within a `message_type` declaration so that the number of such components is not fixed. This is useful for sparse data distribution in dynamic processor configurations and allows efficient implementation

because it helps combine multiple communication primitives, thus saving run time overhead. Only one level of nesting is allowed, though. The following example illustrates the syntax.

```

message_type msg3 (p) {
    msg2 y[p];
} x;

```

message_type msg3 is composed of an array of **message_type** msg2, each of which refers to a vector of type double, and length m, which is a parameter of msg2. The number of msg2 elements, p, is a parameter of msg3. An instance of type msg3 has to now give values to both the parameters. This instance of the message variable is created either by a **receive** statement or by a C style assignment statement. For example, in the for loop shown below, variable x of type msg3 is used for communicating an upper triangular matrix of a square matrix a. As a pointer does not have any size information, and each component is of varying size, the assignment is type cast to type msg2 and the parameter of msg2 is used to specify the vector size.

```

for (i = 0; i < 100; i++)
    x.y[i] = (msg2(i)){&a[i][i]}

```

The final association of x with the parameter of msg3 is done in a **receive** statement, as for example,

```

receive x(100);

```

As the data area for message x is already associated inside the for loop, the above **receive** statement does not specify any data area. The parameter specified is checked against the previous association. (see Figure 1 for another form of the receive primitive).

In some applications, the message type to be communicated depends on the computation state at the sender side. A selective receive statement is usually used for this purpose. A selective receive is costly to implement because buffers for all alternatives must be specified, be enqueued for reception and be dequeued after a message has arrived. It is also error prone because the receiver side code must also be changed for any addition or deletion of message types at the sender side. So EC provides a **message_union** declarations similar to a union in C. This declares types of messages that are allowed in a selective receive statement using a message variable of message union type. A compile time check will detect if any message type is not allowed in the message union. Changes are limited to data structures, and existing code need not be changed. Also, this allows efficient implementation because a single message variable needs to be enqueued or dequeued and

a “lazy” form of message initialization avoids unnecessary overhead for message types that have not arrived. The declaration syntax for `message_union` is the same as that of `message_type`.

3.1.2 Communication Primitives

EC supports an asynchronous `send` (although blocking, if there is no buffer space available) and a blocking `receive` primitive for interprocess communication. A `send` operation is said to be completed after all its data has been accepted by the kernel. As data may be buffered by the kernel at both sides, send and receives are not synchronous. A `receive` operation may specify its own buffer or ask the run time system to allocate buffer for its data. The latter form is suitable when data may arrive before it is requested. This is because EC processes share a connection to the same remote node to avoid exhaustion of system resources like buffers and sockets. So data must be removed from the kernel buffer to allow other EC processes sharing the same socket to read. Without allocating buffers at the user level, potential overlap of communication and computation is reduced. As specified earlier, a message variable is bound with another message variable, and this is done at process creation time. This binding sets up a connection between two or more processes. A local variable is specified (the user level buffer), to hold data in (for a receive), or to send data from (for a send), in a `send` or `receive` primitive. Figure 1 illustrates the simple use of `send` and `receive` primitives.

In the example, a receive on variable `x` is done, and the result is placed in the local variable `a`. In the send statement, data in variable `c` is sent out on the message variable `z`. There is no provision for overlap in communication and computation in Figure 1. So if the process blocks in the `send` or the `receive` statement, the run time system schedules another ready thread, if any.

A program can also exploit parallelism within a thread with the help of software pipelining. A `pipeline` construct specifies a code block to be run in parallel with communication. EC provides both interlocked and noninterlocked pipelines at the user level. Figure 2 shows an interlocked pipeline in a EC program.

In Figure 2, the program segment receives a two dimensional array of dimension $m \times 100$ through message variable `mat1`, applies a function `f` to each element in each row, and sends the output via message variable `mat2`. Without any pipelining, the program would have to wait for the entire array to be received before beginning computation, and would have to wait for the computation to complete before starting the sending. The code segment in Figure 2 pipelines the I/O statements, `receive` and `send` with the code in the `compute` block. The `forall` statement is like a parallel do statement, except, because of the `barrier` statement, each branch is executed only if the barrier can be passed. In the example, the barrier, `mat1.b[j]` can be passed, if the j th component of `mat1` has been


```

message_type matrix {
    double p[10][10];
};

process matrix_multiply(x,y,z)
    in matrix x,y;
    out matrix z;
{
    double a[10][10], b[10][10], c[10][10];
    int i, j, k;

    receive x={a},y={b};
    for (i=0; i < 10; i++)
        for (j=0; j < 10; j++)
            for (k=0, c[i][j] = 0.0; k < 10; k++)
                c[i][j] += a[i][k] * b[k][j];
    send z = {c};
}

```

Figure 1: A simple matrix multiplication program

received. The j th component is the j th row of the input array. The code within the **forall** statement applies the function f to each element of the j th row, puts the result in the j th row of variable y , and, when the computation for a row is complete, it assigns the j th component of y to the j th component of variable $mat2$. This is a signal to the **send** statement, that the j th component of the output is ready for transmission, and this is then transmitted. So, as rows of input become available, computation on the row starts, and as a row of output is ready, it is sent. For machines with a single CPU, the **forall** is expanded to form another inner loop to avoid checking in every iteration.

The significance of declaring x and y inside the **pipeline** construct needs to be made clear. Statements within such a construct form a block, and the scope of these declarations are limited to this block. Further, since the run-time support has in all probability to allocate buffer space for the reception of messages (this has to be done, if data arrives before the **receive** statement can be executed), no space is allocated for these variables. Instead, they are aliased to the buffer space allocated by the run-time support routines. This helps reduce one copy from the run-time buffers to the variables themselves. Even though this is an implementation issue, and it should not be visible at the programming level, by providing such local variable declarations and encouraging programmers to use

```

message_type message_a {
    double arr[100];
};
message_type message_c(k) {
    message_a b[k];
}mat1, mat2;
int m, i, j;
:
pipeline {
    double x[m][100], y[m][100];

    receive mat1(m) = {x};
} with {
    forall j = 0 to m barrier mat1.b[j] {
        for (i = 0; (i < 100); i++)
            y[j][i] = f(x[j][i]);
        mat2.b[j] = y[j][0];
    };
    send mat2(m);
} /*end pipeline */

```

Figure 2: Use of *interlocked pipeline* construct

them, optimization by the compiler is possible. When the end of the **pipeline** statement is reached, space for these variables is released.

A pipeline without interlocking is achieved by not using the **barrier** construct. This form of pipeline is useful when there is no dependency between computation and communication.

3.1.3 Process Declaration and Creation

Process declaration syntax is similar to that of a function. The **create** keyword, followed by the name of a process and its arguments, is used to create a new process. The location of the new process is specified by the keyword **on node**, that follows this. Figure 3 shows an example. A **create** statement has a body that specifies the processes to be created. In Figure 3 it is a simple statement with a repeater, **forall**. The body of a **create** statement can be a list of process instances, each with process name and its

```

main()
{
    matrix x[10], y, z[10];
    double a[10][10][10], b[10][10], c[10][10][10];

    /* Generate matrices a and b */
    :
    /* associate data buffers */
    :
    y = b;
    /* partition data for 10 processors */
    for (i= 0; i < 10; i++) {
        x[i] = {a[i]};
        z[i] = {c[i]};
    };
    create forall i = 0 to 9
        matrix_multiply(x[i],y, z[i]) on node i;
}

```

Figure 3: Create child processes for matrix multiplication

arguments, followed by node number on which the process is created. To create multiple instances of the same process, a **forall** repeater construct is used. Node 0 corresponds to the local node in this case. All available workstations are identified by an integer assigned by the run time system. Because of the synchronous nature, a **create** statement terminates when all its child processes are terminated. A process may have message variables and exceptions in addition to standard C style arguments. The messages and exceptions in process arguments get bound at the time of process creation. A message variable passed as an argument, is bound to the corresponding message variable that is created along with the process creation. Henceforth, communication sent on this message variable becomes available in the newly created message variable in the child. As the same message variable can be used as an argument in a number of create statements, multicasting becomes available. If a message variable has a buffer component as well, the data in such a buffer is also implicitly sent to the newly created message variable to which it is bound. It therefore acts like normal parameter passing. The need for an extra, explicit communication statement, is done away with. The child process starts as soon as it receives all its parameters. But it does not wait for the data bound to the message variable

in the process argument to arrive. This allows pipelining of communication by using the pipeline construct. Figure 1 shows the syntax for process declaration. Figure 3 shows the syntax for process creation. Variables *x*, *y* and *z* are message types of type matrix (see Figure 1 for a declaration). In order to allow implicit sending of messages using these variables, on process creation, variables have to be associated with each message variable. The statements following the **for** loop and the one preceding it do this. Thus, the statement,

$$x[i] = \{a[i]\};$$

associates the variable *a[i]* with the message variable *x[i]*. Now, when the process *matrix_multiply* is created a few lines hence, the values in variable *a* are sent to the newly created process through the message variable *x* due to this association. Similarly, the contents of *b* are sent via the message variable *y*. Nothing is sent via *z* since it is declared as an out variable (Figure 1), even though it is associated with the variable *c*. Only a binding between *z* and the corresponding parameter in *matrix_multiply* is created. When a send occurs from the child process on this message variable, it will be received via *z* and thus placed into variable *c*. Even though the message variable *y* is being used in a number of **create** statements, no multicasting takes place at this time, as there is a separate send for each **create**. However, *y* is bound to a number of other message variables. So, if later, there is a send on *y*, then it will be multicast to all these variables in the child processes.

The process topology to be used depends upon the algorithm used. Figure 3 uses a tree topology because of the worker-coordinator paradigm used. In solving linear equations, a completely connected graph is required to implement pivoting. Similarly, FFT requires a hypercube topology. Different topologies for different problems is achieved by appropriate association of message variables in the argument.

3.2 Other Features

3.2.1 Process Synchronization

EC processes are normally nonpreemptive thus eliminating the need for any critical sections. But a new process may be invoked by a **create** statement or by raising an exception from a remote node. This requires some form of synchronization primitives. Also, sometimes it is required to define a process execution order for correct execution. A conditional synchronization facility is provided by **block** and **activate** primitives which take a process variable as argument. A critical section is implemented by using the **barrier** construct described earlier. By separating synchronous and asynchronous process activation, use of critical sections are further reduced.

3.2.2 Exceptions

Though operating systems like Unix support signals to implement limited interprocess communication and exception handling, we decided against using this as it is expensive and also unsuitable for multithreading environment supported by EC. Exceptions in EC serve two main purposes: handling error conditions, and modelling asynchronous activities such as load balancing. Another possible use is to implement a user level distributed shared memory scheme. EC supports both the termination model as well as the resumption model for its exceptions. The resumption model is included in the language because the termination model is not suitable for modelling asynchronous activities. The resumption model in EC provides a masking facility for recoverable asynchronous faults. User specifies a model of execution for the exception handler in an exception declaration. An exception declaration is similar to a function prototype of type **exception** with optional modifier **resumption_type** specifying type and, **in**, **out**, and **inout** specifying the direction of exception propagation from the block it is defined.

In EC, exceptions are raised by the **raise** statement and handlers are specified by **try** statements similar to those in C++. Apart from user initiated exceptions, the EC run-time system raises two exceptions **io_break** and **process_exited** to handle abnormal situations arising when an EC process terminates prematurely. In the case of **io_break**, the system rejects any data pending in queues and synchronizes the message streams appropriately.

4 IMPLEMENTATION

The implementation consists of a compiler front end and run-time system library providing an interface independent of the underlying Operating System or architecture. The compiler front end is a modification of the GNU C compiler, in which, grammar, typechecking and code generation for new features of EC is added. As with the GNU C compiler, the resulting compiler is CPU independent. The run-time system provides a library for light weight processes, and communication support. The current implementation is on a cluster of Sun workstations and uses the socket library interface.

4.1 Implementation of EC Process

EC does not use the light weight library interface in SunOS because language semantics demand nonpreemptive threads. So there is no need for critical sections in nonpreemptive EC processes. It also allows the compiler to optimize the context switching overhead. The language allows preemption of a process at the time of handling **resumption_type** exceptions. So a user writes code for critical sections only if a handler may concurrently modify sharable data structure. As the compiler knows a context switch may occur only at

`send`, `receive`, `barrier` or `create` primitives, it can determine the life time of registers and thus good register allocation algorithms using constructs as hints for determining the lifetime of register variables (Gupta *et al.*, 1989; Davidson *et al.*, 1991) can limit context switch overhead to saving and restoring of just the instruction pointer and frame pointer registers. A low context switching latency helps in programs involving frequent context switches such as distributed discrete event simulation. Another advantage of using nonpreemptive threads is that debugging is as easy as conventional single threaded applications because there is no need for a separate debug window for each thread. Other reasons for not using Sun threads are portability and also the fact that there is no direct support for creation of threads on remote nodes, which is essential for our applications. Also, EC implementation of threads use a single stack per processor which reduces memory requirements in contrast to preemptive threads such as Sun's light weight process library, and reduces the error due to stack overflow in threads with multiple stacks.

All EC processes are created by a stub residing on the run-time system. The compiler just generates calls to this stub with appropriate parameters to create a EC process either on the local or a remote node. Binding of message parameters of an EC process is also done by this stub procedure.

4.2 Communication Support

This part of the run time system is actually system dependent. Our implementation provides a communication interface on the top of the socket library of Unix. Any call through the compiler interface builds up the list of data to be sent or received and passes this information to an asynchronous communication routine which is driven by interrupts. Under Unix, SIGIO and SIGALRM signals are used to drives this routine. This allows the overlap of communication and computation.

The sending of data is regulated according to the capacity of the network. This is done by sending a fixed amount (which is 8KB in the current implementation) every time and then waiting for a SIGALRM signal to occur. When this occurs, another segment is sent. This way, the buffer requirements in the kernel are kept within limits as otherwise this adversely affects the data transfer rate.

The data reception is done through a handler routine for the SIGIO signal. If there is no receive pending, then buffer space is allocated, as explained in section 3.1.2, otherwise the buffer designated by the receive is used. As TCP/IP does not support multicasting, this is implemented through an intermediate layer which sends separate messages to all remote nodes. The composition of a number of message variables in one variable results in the use of the scatter-gather facility of the socket library. This reduces copying overhead. Reducing the number of copy operations is very important for performance as large volumes of data are transferred in numerical problems, and the implementation takes special

care on this.

All intra-node interprocess communications using message variables do not pass through the usual path. The run-time system passes these data through pointer manipulation whenever a construct does not explicitly specify that data be copied to a different area. The use of message variables ensures that type safety is not sacrificed for efficiency. Also, this makes a program location transparent.

The code generated by an EC compiler will be dependent not only on the CPU type, but also on the machine architecture and the communication software and hardware. Thus, if a node is a multiprocessor system, separate stacks for each thread will have to be present, to allow simultaneous execution. If the hardware supports multicasting, then this can be used. The flow control implemented on `sends` will be influenced by the buffer sizes available and the speed of the network. While an attempt has been made to make the code generated sensitive to a number of such parameters, the issues seem to be too complex and not understood clearly enough, to be able to make the compiler code generation phase portable. Manual tuning will be required.

5 CONCLUSION

EC, an extension of C for distributed programming in a network of computers, has been described. The language provides primitives for fast communication, and for overlap between computation and communication. The compiler generates code for a network of SUN-3 systems interconnected by an Ethernet, running SUNOS 4.1. Some example problems have been run on the system. These include, Matrix Multiplication, solving a set of linear equations and FFT. Preliminary results have shown that a speedup of 3.6 has been achieved on four nodes, with matrix multiplication. A number of other problems are proposed to be run and evaluated. The experience gained from this will be used to fine tune the implementation.

A question arises as to whether the language will be useful in the environment of newer systems that are faster, and that have faster communication links. This question can be answered fully only after gaining experience, but our assertion is that the basic assumptions behind the current design will continue to hold. The ratio between communication and computation speeds is essentially the same in the newer architectures using fast RISC CPUs and FDDI network links. So there is a need to provide overlap between the two activities. The grain of parallelism will remain the same, as software overheads remain relatively the same.

Programming using EC is clearly more difficult than writing a Fortran program. The comparison, however, should be made with other parallel programming languages. Even High Performance Fortran (Loveman, 1993) has many of the complexities that can be seen in EC. The current state-of-art in Distributed Programming is such that such com-

plexities seem difficult to remove. It is hoped that EC is a small step forward, allowing a programmer to write code reasonably easily without sacrificing on the performance. As in other language proposals, libraries of frequently used routines can be programmed using EC which can then be called from a Fortran or C program.

References

- [1] Andre, F., Pazat, J.L. and Thomas, H.(1990) Data Distribution in PANDORE. In *The Fifth Distributed Memory Computing Conference*, IEEE Computer Society Press.
- [2] Andrew, G.R., Olsson, R.A. , Coffin, M., Elshoff, I., Wilson, K., Purdin, T., and Townsend, G..(1988) An Overview of the SR Language and Implementation. *ACM Transactions on Programming Languages and Systems*, 1,51–86.
- [3] Beguelin, A., Dongarra, J., Geist, A. and Sunderam, V.(1993) Visualisation and Debugging in a Heterogenous Environment. *Computer*, 6,88–95.
- [4] Burns, A.(1988) Programming in OCCAM-2. Addison- Wesley Publishing Company.
- [5] Carriero, N. and Gelernter, D. (1991) A Foundation for Advanced Compile time Analysis of LINDA programs. in *Languages and Compiler for Parallel Computers(LNCS-589)* (ed. U. Banerjee et.al.), Springer Verlag.
- [6] Chang, C.(1989) REXDC: A Remote Execution Mechanism. in *SIGCOM'89 Symp. Communication Architecture and Protocols*, ACM Press.
- [7] Davidson, J.W. and Whalley, D.B.(1991) Methods for Saving and Restoring Register values across Function Calls. *Software Practice and Experience*, 2,149–165.
- [8] Gao, G.R.(1993) An Efficient Hybrid Dataflow Architecture Model. *Journal of Parallel and Distributed Computing*, 4, 293–307.
- [9] Gehani, N.H.(1984) Broadcasting Sequential Processes. *IEEE Transactions on Software Engineering*, 4,343–351.
- [10] Gelernter, D.(1991) Current Research on LINDA. in *Research Directions in High level Parallel Programming Languages (LNCS-574)*, (ed. J. P. Banatre and D. Le Metayer), Springer Verlag.
- [11] Griswold, W.G., Harrison G.A., Notkin, D., and Snyder, L.(1990) Scalable Abstraction for Parallel Programming. in *The Fifth Distributed Memory Computing Conference*, IEEE Computer Society Press.

- [12] Grit, D.H.(1990) A Distributed Memory Implementation of Sisal. in *The Fifth Distributed Memory Computing Conference*, IEEE Computer Society Press.
- [13] Gupta, R., Soffa, M.L. and Steel, T.(1989) Register Allocation via Clique Separator. in *The Proceedings of the ACM SIGPLAN'89 Conf. on PLDI*, ACM Press.
- [14] Halstead, R.H.(1989) New Ideas in Parallel Lisp: Language Design, Implementation and Programming Tools. in *Parallel Lisp: Language and Systems(LNCS-441)*, (ed. Robert H. Halstead Jr. and T. Ito) Springer Verlag.
- [15] Hansen, P.B.(1978) Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM*, 11,934-941.
- [16] Hieb, R., Dybvig, R.K. and Bruggeman, C.(1990) Representing Control in Presence of first class Continuation. in *The Proceedings of the ACM SIGPLAN'90 Conf. on PLDI*, ACM Press.
- [17] Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U. and C. W. Tseng.(1991) An Overview of Fortran D Programming System. in *Languages and Compiler for Parallel Computers(LNCS-589)* (ed. U. Banerjee et.al.), Springer Verlag.
- [18] Kessler, R.R. and Swanson, M.R.(1989) Concurrent Scheme. in *Parallel Lisp: Language and Systems(LNCS-441)*, (ed. Robert H. Halstead Jr. and T. Ito) Springer Verlag.
- [19] Liskov, B. and Scheifler, R.(1983) Guardians and Actions: Linguistic support for Robust Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 3,381-404.
- [20] Liskov, B., Herlihy, M. and Gilbert, L.(1986) Limitations of Synchronous Communication with Static Process Structure in a Language for Distributed Computing. in *The Thirteenth ACM Symp on POPL*, ACM Press.
- [21] Loveman, D.B.(1993) High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1,25-42.
- [22] Lu, L.C. and Chen, M.(1991) Parallelizing Loops with indirect array reference or Pointers. in *Languages and Compiler for Parallel Computers(LNCS-589)* (ed. U. Banerjee et.al.), Springer Verlag.
- [23] Magee, J., Kraumer, J. and Sloman, M.(1989) Constructing Distributed Programs in CONIC. *IEEE Transactions on Software Engineering*, 6,663-675.

- [24] Magee, J., Kraumer, J. and Dulay, N.(1992) Darwin/MP: An Environment for Parallel and Distributed Programming. in *Proceedings 1992 Hawai International Conference on System Sciences*, IEEE Computer Society Press.
- [25] Mishra, J. (1991) A Perspective in Parallel Program Design. in *Research Directions in High level Parallel Programming Languages (LNCS-574)*, (ed. J. P. Banatre and D. Le Metayer), Springer Verlag.
- [26] Rosing, M. and Weaver, R.P.(1990) Mapping Data to Processors in Distribution Memory Computations. in *The Fifth Distributed Memory Computing Conference*, IEEE Computer Society Press.
- [27] Shivaratri, N.G., Krueger, P. and Singhal, M.(1992) Load Distributing for Locally Distributed Systems. *Computer*, **12**,33-44.
- [28] Socha, D.G.(1990) An Approach to Compiling Single-Point Iterative Programs for Distributed Memory Computers. in *The Fifth Distributed Memory Computing Conference*, IEEE Computer Society Press.
- [29] Strom, R.E., Bacon, D.E., Goldberg, A.P., Lowery, A., Yellin, D.M. and Yemini, S. A.(1991) *Hermes : A Language for Distributed Computing*. Prentice Hall International, 1991.
- [30] Subramaniam, O.S.M.(1991) Enhancements to DC : A Distributed Programming Language. Master's thesis, Indian Institute of Technology, Kanpur, 1991.
- [31] Wegner, P.(1991) Design Issues in Object based Concurrency. In M. Tokoro, O. Nierstrasz, and Peter Wegner, editors, *Object based Computing (LNCS-612)*, Springer Verlag, 1991.
- [32] Wegner, P.(1992) Dimensions of Object Oriented Modeling. *Computer*, 21(10):12-19, 1992.

BIOGRAPHY: Dr. Barua is Head of Computer Science and Engineering Department and Computer Center. His areas of interest include Operation Systems and Distributed Computing.

Mr. Naik is a Ph.D. candidate in the department. His areas of interest include Compilers and Parallel Computing.