How to Write Parallel Programs on the T3E Using Linda

Carlos Sosa
Chemistry Applications, Silicon Graphics, Inc./Cray Research, Eagan, MN 55121

cpsosa@cray.com
http://wwwapps.cray.com/~cpsosa/
Nicholas Carriero
Computer Science Department, Yale University, New Haven, CT 06510

carriero-nicholas@CS.YALE.EDU
http://www.sca.com

ABSTRACT:

Linda[1] is a model for parallel processing based on distributed data structures. Processors and data objects communicate by means of the *tuple space* (TS) or Linda memory. The interaction between the *tuple space* and the processors is carried out by four basic operations: **out**, **in**, **eval**, and **rd**. These operations add/remove/read data objects from/to the *tuple space*. **eval** on the other hand also forks processes to initiate parallel execution. In this presentation we'll discuss the use of Linda on the T3E.

KEYWORDS:

Linda, Tuple Space, Memory, Programming Model

Introduction

Since the early 80's there have been numerous parallel computer projects [1], such as, ICL DAP, Denelcor, Intel iPSC, NCUBE, Connection Machines, and many more. However, most applications have not migrated to parallel machines nor have they extensively taken advantage of clusters of many workstations. In part this is due to the fact that parallel programming is not easy, some algorithms currently being used by many applications in the scientific arena cannot be trivially subdivided in many independent tasks. Additionally, current compiler technology cannot fully and automatically parallelize large applications. In most cases, extensive manual parallelization is required to take advantage of a system with distributed memory on many processors.

Furthermore, a variety of parallel programming models have emerged in the last few years. This has introduced an extra dimension when a programmer is faced with the task of parallelizing a particular application. A particular parallel programming model is an attempt to allow users to excerpt control over the hardware. It provides a way for the user to distribute data and work among all processors available on a highly-parallel machine

Currently some of the most commonly used programming models on the CRAY T3E are: CRAFT, PVM, and MPI[2]. CRAFT, the Cray Fortran programming model was originally designed for the CRAY T3D, this model supports four programming methods: data-sharing, work-sharing, message-passing, and explicit shared-memory. On the other hand, PVM and MPI are message passing programming models.

An Alternative to the above mentioned programming models is Linda[3] which is a memory model. Linda's virtual memory (shared-addressable memory-like) on the CRAY T3E provides users with a simple and flexible programming environment. A model familiar to programmers on CRAY vector systems. Linda creates a virtual shared memory that is shared logically by all the processors on the CRAY T3E. In this paper we discuss how to write programs using Linda with emphasis on the CRAY T3E and briefly discuss its implementation on the CRAY T3E.

CRAY T3E Design Features

A major difference between traditional vector supercomputers an MPP machines is in the memory architecture[4]. Traditional vector supercomputers that use parallel vector processors (PVP) have one uniform shared-addressable memory among all the processors. For example, the CRAY T90 has 32 processors, each with very rapid access to central memory. Any processor can read any word in memory with the same time delay. On the other hand, MPP systems, such as the CRAY T3E used in this work, have distributed memory. Figure 1 illustrates an air-cooled CRAY T3E.

The CRAY T3E parallel computer system consists up to 2048 process elements (PEs) and 128 system/redundant PEs. Peak performance for the largest configuration is 1 TFLOPS. Each PE on this system is composed of a DEC Alpha 21164 EV5 RISC microprocessor. Memory size scales from 64 MBytes to 512 MBytes per PE (it can be extended to 2 GBytes). Local memory is divided into cached and non-cached. Cached data is distinguished by 0 in the uppermost bit (bit 39) of the physical address. A 1 in bit 39 identifies non-cached data. All remote loads/stores operations are carried out by external registers called E-registers. This is the only mechanism for accessing remote memory in T3E. The interconnect network on the Cray T3E is a 3D-torus with some partial planes allowed.

Figure 1. Air-cooled CRAY T3E

Process elements are built 4 to a printed circuit board. These 4 PEs have network connection to a globally-accessible I/O channel. Finally, it is important to mention that the EV5 has a 96 KBytes 3-way set associative secondary cache on chip[5] .

Parallel Programming Model

In general, we can consider three types of concurrency (parallelism): data level parallelism, task level parallelism, and functional level parallelism. In these three cases parallelism arises from data multiplicity, multi-tasking, and data pipelining, respectively. Corresponding to these types of parallelism, one might consider three parallel programming (*coordination*) paradigms: data parallel, message passing and shared-address location. These paradigms are not mutually exclusive, but they are clearly different from paradigms for computation.

In this context, Linda[6] is defined as a language-independent set of coordination operations that embraces the above parallel programming paradigms. Linda has been integrated with C and Fortran to define high-level parallel programming languages C-Linda and Fortran-Linda, respectively.

One of the key concepts in the Linda coordination model is the shared, content-addressed, ``virtual" memory *tuple space*. All the interprocess communication is carried out via operations on tuple space. In this model, the programmer never has to be concerned with or program explicit message passing constructs and never has to manage the relatively rigid, point-to-point process topology induced by message passing. In contrast, coordination in Linda is *uncoupled* and *anonymous*. The first means that the acts of sending (producing) and receiving (consuming) data are independent (akin to buffered message passing). The second means that process identities are unimportant and, in particular, there is no need to ``hard wire" them into the code. Conceptually, this amounts to the difference between trying to run a commodity trading pit with a tangled mass of telegraph sets (point-to-point) rather than yelling and listening (a trader lauches a bid into space, those interested in the bid pull it in). It also means data may live independent of a process, so shared variables are easy to support---unlike message passing which insists that data always be ``somewhere", and thus impose considerable overhead in the form of a process created to manage the variable that is to be shared.

Another important difference between Linda and systems like PVM and MPI is that Linda is implemented as a coordination *language* while the others are implemented as libraries. A language-level implementation provides better syntactic support and a richer semantic interface which simplifies coding---the examples illustrate this.

The data is moved from/to tuple space by using *tuples*. *Tuples* are defined as a sequence of different type of fields separated by a delimiter (comma) and enclosed in parentheses. Examples of *tuples* may be seen in Figure 2.

```
("task", 13.4, 7)
("evaluate", 6, hello(i))
("integer", i)
```

Figure 2. Examples of *tuples* using C-syntax.

In these three cases we have *tuples* with three and two fields. In all cases the first field is a string that can be used as a tag.

Linda Operations

Linda interacts with the *tuple space* using four basic operations. Three operations can be used to add/remove *tuples* from the *tuple space* and a fourth operation that is capable of creating new processes. These four operations are described in Table 1.

Table 1. Four Basic Linda Operations.

Operation	Description
out()	Defines or adds *tuples* in the *tuple space*
in()	Reads and deletes a *tuple* from *tuple space*
rd()	Reads a *tuple* from *tuple space*
eval()	Creates a new process and/or adds a *tuple* in the *tuple space*

In addition to these four operations Linda provides two variants of in and rd. inp and rdp, respectively[7]. These two operations are non-blocking forms of **in** and **rd** which return true or false if the request was successful or not.

CRAY T3E Linda Implementation

10 years of research at Yale and Scientific Computing have led to a general strategy for developing efficient Linda

implementations:

1. At module compile time, collect data about the Linda operations within a module.
2. At link time, analyze the pattern of Linda usage, as given by the data collected in 1. Use the results of this analysis to:
    1. Restrict the collection of *tuples* or *templates* (the ``tuple" produced by an **in()**/**rd()** operation) a given operation must consider by construction of a disjoint partitioning of the operations. An operation in one set cannot be influenced by the activity of an operation in another set.
    2. For each partition, choose a data structure for organizing the *tuples* and templates arising from operations in the partition. The data structure is chosen from a small list of standard structures (counting semaphores, queues, etc.).
3. Given a target architecture, make good use of its ``feature set" to implement Linda runtime services that ensure the data structures of item 2.2 are well supported. Note that assigning responsibility for managing *tuple space* to one processor is likely to create a bottleneck, so the the work of managing *tuple space* should be distributed across a number of processors.

Efficient Linda runtime support has been developed for a variety of platforms including shared-memory, distributed-memory, and LAN architectures. The T3D/T3E architecture, however, differs in significant ways from all of these. In essence the T3D/T3E can be viewed as an instance of an architecture somewhere between shared-memory and distributed-memory. It is like shared memory systems in that a data location can be referenced from any process, making it relatively easy for different processes to asynchronously access and update the data structures holding *tuple space*. Unfortunately, it is like distributed memory in that the address space is partitioned. This meant our traditional shared memory implementation---based on a model in which all addresses (local or shared) are in the same address space---wouldn't work. E.g., in the ``typical" SMP model, a data structure pointer is the same whether the pointer is to a local or shared address. On the T3D/E, a ``pointer" to an address on another node is completely different from a pointer to a local address. Driven by these considerations, we developed a new runtime loosely based on our shared memory approach but one that reflected and exploited T3D/E features. Two examples of the latter: 1) heavy use is made of the 64 bit swap operation to accomplish both synchronization and movement of important control data, 2) the ability to access *any* memory on *any* node is leveraged to provide "zero copy" data transfers when an **in()** precedes an **out()**.

Example

One of the first programs that any C programmer learns is the simple "Hello World" program. In this section we present the C-Linda version of "Hello World". It is interesting to note that even this simple example basically illustrates the use of all the Linda operations required to parallelize an application.

```
    #include <stdio.h>

    real_main (argc, argv)
    int    argc;
    char   *argv[];
{
    int nworker, j, hello(), sum, temp;

    if (argc != 2) {
    fprintf(stderr, "Usage: %s <workers> \n", *argv);
    }

    nworker = atoi(argv[1]);

    out("sum", 0);

           for ( j=0; j<nworker; j++ )
              eval("worker", j, hello(j));

           for ( j=0; j<nworker; j++ )
              in("done");
    in("sum", ? sum);

           printf("sum is %d\n", sum);



           for ( j=0; j<nworker; j++ ) {
              in("worker", j, ? temp );
              printf("%d^2 is %d\n", j, temp);

    }

           printf("hello_world is finished\n");

           lexit(0);

}

           hello(i)

    int     i;
{
    int     sum;

           printf("Hello, world from number %d\n", i);

    in("sum", ? sum);
    out("sum", sum + i);
    out("done");

           return(i*i);

}
```

Figure 3. C-Linda "Hello World" program.

This program requires as input the number of workers that will execute the `hello()` function. In the case of the CRAY T3E this number normally corresponds to the NPEs - 1. The first for loop creates workers that print "hello world", increment a global sum, define a *tuple* "done" in the *tuple space*, and return a product. The second for loop waits for a "done" *tuple* from each worker. Once the "done" *tuples* have been collected, the global sum is retrieved and printed. The last loop matches the *tuples* in *tuple space* created by the **eval** operation. This loop retrieves and prints in order the products left behind by the workers.

Summary

Linda provides a very simple and flexible parallel programming model. Its shared, content-addressed, ``virtual" memory *tuple space* eliminates the need for explicit message passing constructs. In the Linda programming model, three operations are used to add/remove *tuples* from/to the *tuple space*. In addition, a fourth operation creates new process (workers). All the communication is carried out via the Linda memory or virtual memory that is created at the software level. In this context, Linda programs often use the idea of having a master process. The master process creates workers and coordinates or does the accounting after each worker terminates its task.

Acknowledgments

The authors wish to thank the Corporate Computer Center at Silicon Graphics, Inc/Cray Research for providing time on the CRAYs T3D/E.

Footnotes

[1] Linda is a trademark of Scientific Computing Associates, Inc.

References

[1] Oliver A. McBryan, **Overview of Current Developments in Parallel Architectures**, in Parallel Supercomputing: Methods, Algorithms and Applications, Edited by G. F. Carey, John Wiley & Sons, NY, NY, 1989.
[2] Cray Research, **Cray Research MPP Software Guide**, SG-2508 1.1, Mendota Heights, MN, 1994.
[3] N. Carriero and D. Gelertner, **How to Write Parallel Programs: A Fisrt Course**, MIT Press, Cambridge, MA, 1990.
[4] R. W. Numrich, P. L. Springer, and J. C. Peterson, **Measurement of Communication Rates on the CRAY T3D Interprocessor Network**, Eds. W. Gentzsch and U. Harms, in High-Performance Computing and Networking, Springer-Verlag, Munich, Germany.
[5] S. Oberlin, R. Kessler, S. Scott, and G. C. Thorson, **CRAY T3E Architecture Overview**, Cray Research Manual, Chippewa Falls, MN, 1996.
[6] N. Carriero and D. Gelernter, **How to Write Parallel Programs: A Guide to the Perplexed**, ACM Computing Surveys, Vol. 21, 323(1989).
[7] Scientific Computing Associates, Inc., **Linda User's Guide and Reference Manual**, Scientific Computing Associates, New Haven, CT 06510.

Authors Biography

Carlos Sosa is a computational chemist in the chemistry applications group at Silicon graphics, Inc./Cray Research in Eagan. He is currently working with the Linda version of the electronic structure application *Gaussian* 94.

Carlos Sosa

http://wwwapps.cray.com/~cpsosa/

Nicholas Carriero is a research scientist at Yale University and Scientific Computing Associates. He is one of the main developers of Linda.

Nicholas Carriero

http://www.sca.com