# A Software Environment for General Purpose MIMD Multiprocessors

Rajat Moona

## and V. Rajaraman

Supercomputer Education and Research Centre Indian Institute of Science Bangalore India.

Abstract: In this paper, we describe a software environment for general purpose MIMD machines based on packet switched message passing paradigms. The environment provides system calls for communication between various processing elements in the machine and system calls to get the configuration of the system. It assumes the presence of a Host processor through which facilities are provided for forking remote processes on various processing elements. The Host processor has all peripherals of the system and acts as an I/O processor of the system. During normal course of execution, it forks processes, collects results from various processing elements, sequences them and then produces output of the program.

#### 1 Introduction

A multiple instruction multiple data (MIMD) multiprocessor classification is given to the parallel computing machines which have multiple instruction streams over a set of interactive processing elements with shared resources. These multiple instruction streams operate on multiple data streams which are relatively independent of each other. Processing elements in a MIMD multiprocessor system operate asynchronously with separate controllers thus providing a flexible means of programming for general purpose applications. Various MIMD architectrure have been reported in the litrature. Most popular of these architectures are hypercube, mesh connected, ring connected, shared memory multiprocessors, processing elements connected in a circuit switched network.

Programming such a multiprocessor machine can be done in one of the two ways. In one case, a programmmer specifies his program in a programming language and the compiler running on parallel machine compiles this program, and create the task graph which can be loaded on various processing elements. Such a scheme requires enormous intelligence at the compiler as it has to identify parallelism in the program. Example of such approaches are the vectorizing compilers for languages available on today's supercomputers. In this catagory are also emerging new languages which are more natural for concurrent computing. Functional programming languages, data flow languages and logic programming languages are examples of these languages. However, majority of scientific applications lead to inefficient programming with such languages.

The other approach which has been widely adopted for the MIMD architecture is the one in which a programmer specifies parallelism in the program using constructs provided by a software environment. Such an approach is adopted in the CrOS III environment designed for a hyprecube [1].

A program for a multiprocessor system can be viewed as a collection of cooperative tasks which can be executed in parallel. These tasks communicate using the communication instructions implemented through the usage of shared resources. Such multitasking environments are available in UNIX operating system [2,3], Linda software environment [4] etc. We assume that one task is executed by only one processing element in the multiprocessor. A software environment can hide the underlying hardware and use this structure model for parallel computing.

In this paper, we describe a general purpose software environment (Figure 1) suitable for MIMD multiprocessor architectures. We assume a message passing paradigm between the processing elements and existence of local memory with each processing element. We also assume the model in which at one time only one task is executed by one processing element. A task can be ex-



Figure 1: Structure of a complete Software Environment

ecuted in parallel and asynchronously with other tasks. A host processor or an I/O processor is used for interaction with the programmer. The host loads the tasks into the memories of various processing elements. The host also provides inputs to the tasks, collects results, sequences and produces results at the output. We assume that a reliable method of communication is available between the processing elements. If communication is not reliable at the hardware level, a software layer can be built over the hardware which can make the communication reliable with message retransmissions in case of unreliable communications. Certain algorithms are specific to the architecture and work efficiently if coded with the hardware dependencies. In section 2, we will describe the system parameters which are specific to the architecture. We will also discuss how to model hardware dependencies in the software environment. In section 3, we will describe the part of the software environment which is specific to the hardware. In section 4, we describe ways to create and terminate tasks, and methods to achieve inter-task communication. In section 5, we present a sample program for numerical integration, written in C using this software environment. Section 6 concludes the paper.

## 2 Hardware Dependencies

Multiprocessor systems have a particular node connectivity which is suitable for a class of applications e.g. a point to point mesh connected multiprocessor may be suitable for problems where communication is restricted to only the nearest neighbours. Such an information is coded in the software environment using certain variables. There are subroutines available for the programmers which can be called with hardware specific information such as connectivity of the processing elements.

Any MIMD architecture can be parameterized using certain parameters specific to the architecture. A tree structure, for example, can be parameterized by depth and arity parameters. A hypercube structure is parametrized by its dimension and a mesh connected processor is parameterized by its dimension and number of processing elements in each dimension. In our software environement we have an array of system parameters call ParamArr. It codes all the hardware specific information of the architecture.

Other hardware specific information is coded in a parameter called mask. It codes the connectivity information of a processing element. This parameter may be obtained by a call to the software environment which takes the addresses of the processing elements as an input and returns the mask.

Using these hardware specific information, a programmer may program a multiprocessor machine independent of its configuration. We will now be describing the part of the software envionment which returns the system parameters.

We will be describing the subroutines available in the software environment from the C interface. Software environment does not however, impose any restriction on the programming languages.

### **3 System Parameters**

Subroutines in this section describe the interface for getting system parameters and address of a processing element.

# 3.1 sysparam Subroutine

sysparam routine returns the system parameters in an array ParamArr. Structure of the subroutine is as follows.

void sysparam(ParamArr);
int \*ParamArr;

#### 3.2 nodeid Subroutine

nodeid routine returns the address of the processing element on which this subroutine is called. Syntax of nodeid subroutine is given below.

int nodeid();

#### 3.3 get\_mask Subroutine

Subroutine get\_mask takes a list of processing elements' addresses as input and returns an integer mask specific to the architecture. This integer is used by process creation and communication calls. Syntax of routine get\_mask is given below. int get\_mask(n,list);
int n, \*list;

Having described these subroutines, we will now describe process creation, termination and the interprocess communication.

# 4 Process Management

In any MIMD system, there are two type of communications (1) data communication and (2) program communication. In the first type of communication, data is passed from the source processing element to the destination processing element and in the second type of communication, a program or a task is sent to the destination processing element by a source processing element or a host processor. Here again we have two models of execution of a program. In one case, there are multiple programs running on the multiple processing elements. In the other case a single program, but its different copies, run on multiple processing elements. In both cases, a program is sent to the destination processing elements by a host processor.

In the process creation part, we have the following subroutines.

#### 4.1 fork Subroutine

fork call is used to create processes on multiple processing elements. Model of execution in this case is "single program multiple data". Each newly created process has same code as of the old process. After new processes are created, execution of all processes proceeds concurrently from the same program location as in the source process. Syntax for this call is as follows.

int fork(mask);

int mask;

Mask is hardware dependent parameter which is used for establishing communication as described earlier. This call may also be used for creating more than one process at a time.

## 4.2 sendp Subroutine

This subroutine is used for other mode of communication where a remote task is totally independent of the source task. It is a "multiple program multiple data" mode of process creation. Syntax for this call is as follows.

void sendp(addr, prog, goaddr, loadaddr, len); int addr; char \*prog, \*goaddr, \*loadaddr; int len

A program is first loaded in the memory of the host processor at the address prog. It is then sent to the processing element whose address is given by addr. This processing element stores the task in its memory at the address loadaddr and starts execution from the address goaddr. Size of the task is given by len.

A task can be made "load only" type by making its goaddr=-1. This feature is required for certain applications like initial loading of the operating system. A task may be made completely relocatable if its loadaddr is specified as -1.

# 4.3 gbroadp Subroutine

This subroutine is used for creating multiple jobs each having the same program which need not be the same as the program in source processing element. Syntax for this call is as follows.

void gbroadp(prog, goaddr, loadaddr, len);
char \*prog, \*goaddr, \*loadaddr;
int len:

This subroutine creates as many tasks as there are processing elements in the system, with the parameters specified.

## 4.4 terminate Subroutine

A process may call procedure terminate which terminates the calling process. Syntax of this routine is given below.

### void terminate()

We now describe the inter-task communication routines which implement data communication among the processing elements in the MIMD architecture.

## 4.5 bread and ubread Subroutines

bread and ubread routines are used for reading a message from the processing element specified by one of its parameters. Syntax of these subroutines are given below.

void bread(mask, var, sizevar); int mask; char \*var; int sizevar; int ubread(mask, var, sizevar); int mask; char \*var; int sizevar:

These procedures read sizevar amount of data bytes from the processing element specified in variable mask in memory location whose address is given by variable var.

bread subroutine reads data in blocked mode of communication. Calling process is suspended till all requested data bytes are received. ubread subroutine reads data in unblocked mode of communication. If data bytes are not available then the control is returned back to the calling process with return value indicating the number of bytes that could not be read.

### 4.6 bwrite and ubwrite Subroutines

These subroutines are complementary to bread and bwrite. The syntax of these routines is the same as described earlier. These routines also allow a multicast of data messages in which one processing element can send data to more than one processing element. This information is coded in the variable mask.

### 4.7 sendd Subroutine

Subroutine sendd is used to send data message to a processing element whose address is specified in variable addr. Syntax of this call is as follows.

void sendd(addr, var, sizevar); int addr; char \*var: int sizevar:

sizevar amount of data bytes from the memory location whose address is given by variable var are sent to the processing element whose address is given by addr.

# 4.8 gbroadd Subroutine

Subroutine gbroadd is used to send data messages to all processing elements in the system. Syntax of this routine is as follows.

void gbroadd(var, sizevar); char \*var; int sizevar;

Using a set of the above routines, we will now be describing an application which performs the numerical integration of a function of one independent variable on an array of processing elements. We will be using a hardware dependent parameter p which gives the number of processing elements in the system.

## **5** Numerical Integration

For this problem, each processing element in the system integrates the function over a specific subrange of the integration



Figure 2: Partitioning of integration range for a function f

interval and sends results back to the host processor whose address is 0. Host processor receives integrated values from all other processing elements and adds them.

# 5.1 Mapping of the problem

If the function f is to be integrated over the range  $(\sigma, \sigma')$ , then this range is divided into p equal subranges. As there are p processing elements in the system, each processing element gets one such subrange for integration. The subrange  $(\mu_i, \mu'_i)$  for a processing element with nodeid=i can be found by the following formula.

$$\mu_i = \sigma + i * \frac{\sigma' - \sigma}{p}, \quad \forall i = 0 \dots p, \text{and},$$
$$\mu'_i = \mu_i + \frac{\sigma' - \sigma}{p}.$$

One such partitioning for an arbitrary chosen function f is shown in Figure 2 with reference to a 4 processing element machine.

We now describe the algorithms used for this problem.

5.2 Algorithm and Implementation

We now give the C code for implementing the above algorithm to run in parallel.

- (1)#include <math.h>
- (2)#include <stdio.h>
- (3)#include <comm.h>
- #define STEPS 60 (4)
- (5)
- (6) int peid, p, x
- (7)float min, max;  $/ * \mu, \mu' * /$
- (8)float gmax, gmin;  $/ * \sigma', \sigma * /$
- (9)float localrange;  $/ * \mu' - \mu * /$
- (10)
- (11) $main() \{$
- (12)int i, mask, k;
- (13)float integloc, readint;
- (14)(15)initbuf();
  - mask = (1 << (p-1)) 1;
- (16)fork(mask);
- (17)(18)
- peid = nodeid();
- (19)max\_min(peid);
- integloc = integ(STEPS); (20)
- (21)if (peid != 0) (22)
  - sendd(0, &integloc, sizeof(integloc));

```
(23)
          else {
(24)
```

(25)

(26)

(27)(28)

- for (k=0; k < p-1; k++) {
- bread(1 << k, &readint, sizeof(readint));</pre>
- integloc + = readint ;

printf("integral=%f",integloc);

- (29)} (30) terminate(); (31) } (32)(33) void max\_min(peid) (34) int peid; (35) { min=gmin + peid \* localrange; (36)(37) max=min + localrange; (38) } (39)(40) float integ(NUMSTEPS) (41) int NUMSTEPS; (42) { (43) int i: (44)float total, halfstep, step; (45)step=(max-min)/(float)NUMSTEPS; (46) (47)halfstep=0.5\*step; for(i=0, total=0.0; i<=NUMSTEPS-1; i++) {</pre> (48)(49) total + = step\*func(min+halfstep); (50) $\min += step;$ (51) } (52)return(total); } (53)(54)(55) float func(x) (56) float x; (57) { (58) return(exp(-x));(59) } (60) (61) void initbuf() (62) { (63) sysparam(&p);printf("Lower and Upper range : "); (64)
- (65) scanf("%f %f", &gmin, &gmax);
- (66) localrange=(gmax-gmin)/(float)p;

(67) }

## 6 Conclusion

In this paper, we described a general purpose software environment for a MIMD multiprocessor. This environment has been implemented on a broadcast cube architecture implemented at Indian Institute of Science Bangalore. The environment allows a programmer to exploit all features of the architecture but keeps its interface same across the architectures. A programmer may have to modify his program in terms of partitioning of the task but can use the same set of routines.

Acknowledgements : Authors would like to thank Mr. S. K. Nandy for his suggestions in improving the paper style, and correcting the manuscript.

# References

- Fox G. C., et. al. "Solving Problems on Concurrent Processors" vol. 1, Prentice Hall, New Jersey, 1988.
- [2] Rebecca Thomas, et. al. "Advanced Programmer's guide to UNIX system V", McGraw Hill, 1987.
- [3] Maurice J. Bach, "The design of the UNIX operating system", Prentice Hall, New Jersey, 1988.
- [4] Ahuja S., et. al. "Matching Language and Hardware for Parallel Computation in the Linda Machine", *IEEE Trans. Computers*, vol 37, no. 8, Aug 1988, pp. 921-929.