Glenda Installation and Use

Benjamin R. Seyfarth
Jerry L. Bickham
Mangaiarkarasi Arumughum
University of Southern Mississippi

November 3, 1993

Abstract

This software is being developed to provide the capabilities of Linda [1], a group of functions capable of utilizing multiple processors which allows the user to more easily program in parallel. Named Glenda, its model was setup to closely parallel that of Linda - with a few exceptions. Glenda is built utilizing Parallel Virtual Machine (PVM 3.x) software system to provide the underlying communications. It supports most of the Linda operations with a few added capabilities to utilize PVM more fully.

1 Introduction

In this document we describe Linda, PVM and Glenda. We define the Glenda operations and illustrate their use with examples. We describe how to use the C Glenda preprocessor and how to prepare a Makefile to simplify using the preprocessor. We describe how to install the Glenda software and use it on a system. Finally we describe some of future plans for Glenda.

1.1 The Linda Model

As stated in "*How to Write Parallel Programs*" by Carriero and Gelernter, 1990, the Linda model is a parallel programming model which uses a global tuple space to coordinate the passing of messages between processes. It consists of only six functions, those being

out produces a tuple

in consumes a tuple

inp consumes a tuple if available

rd reads a tuple

rdp reads a tuple if available

eval executes a function in a subprocess producing a tuple

When using the Linda model and agenda parallelism, worker processes are all equally capable and retrieve tasks from an agenda until all work is done. One master process starts slave processes using the eval function. The master then sends tasks using some agenda. Each slave consists of a loop which retrieves the tasks from the agenda and sents their results back to the master process. When all the needed results have been received, the master process sends "poison pills" out to the slave processes that terminates them.

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

1.2 PVM - Parallel Virtual Machine

Developed by Oak Ridge National Laboratory, Emory University.

PVM is a collection of functions that, like Linda, allow the user to make use of a multiprocessor system. Its key features are:

- Can use a workstation network
- Can also use a multi-CPU system
- Architectures can be mixed on a virtual machine
- Most popular architectures are supported
- Asynchronous message passing model
- Messages consist of multiple components of eight different data types
- Messages are accessed by an integer message type
- Supports barriers and signals
- Supports Fortran and C
- Consists of about 53 functions for C and 36 for Fortran

1.3 Glenda

Glenda was developed to make use of both Linda's and PVM's positive attributes, while at the same time improving on their shortcomings. These were our primary goals for developing Glenda :

- Preserve Linda's global tuple space model
- Reduce the number of functions required to use PVM
- Maintain PVM's message passing efficiency whenever required
- Maintain PVM's portability
- Present Linda as an integral part of the host language.

1.3.1 Glenda Operations

These are the functions that make up Glenda.

- tid = gl_mytid()
- tid = gl_spawn (name)
- gl_out (name, ...)
- gl_in (name, ...)
- gl_inp (name, ...)
- gl_rd (name, ...)
- gl_rdp (name, ...)
- gl_outto (tid, name, ...)

- gl_into (name, ...)
- gl_exit ( )

There are a few noticeable differences from Linda. First of all, there is no eval function. Instead, we join Glenda by calling gl_mytid and use gl_spawn to start subprocesses. gl_mytid returns a PVM task id number and enrolls it into PVM. gl_spawn also returns a PVM task id number for the spawned process. Also, every tuple has a character string for its first component. The functions outto and into were added to make use of PVM's multicast capability. Outto can be used along with a PVM task identifier to send a tuple directly to a task. Into must then be used to retrieve a tuple sent using outto. To exit out of PVM and the tuple server, the function gl_exit must be used. Finally, structures, unions and typedefs are not supported by the glenda operations.

1.3.2 gl_out Examples

- `gl_out ( "data", i, k, value )`

  Value can be a scalar or an array. If it is an array the length is implicit.

- `gl_out ( "row", i, x:len )`

  Here the length to output for x is given. X could be an array or a pointer.

- `gl_out ( "column", j, x[j]:len )`

  Here we must specify the length to output. The preprocessor only keeps lengths for single dimension arrays.

1.3.3 gl_in and gl_inp Examples

The functions gl_in and gl_inp are used to input a tuple into tuple space. The function gl_in waits for a match before returning a tuple, while gl_inp returns a value of 1 if there was a matching tuple available and a 0 if a tuple was not available with no waiting involved. When a tuple is matched, that tuple is removed from tuple space.

- `gl_in ( "data", i, k, ?  value )`
- `gl_inp ( "data", i, k, ?  value )`

  These match on "data" i and k returning data for value. Value can be a scalar or an array. If it is an array the length is implicit.

- `gl_in ( "row", i, ?  x:len )`
- `gl_inp ( "row", i, ?  x:len )`

  These match on "row" and i returning data for the array x and returns the number of items as len.

- `gl_in ( "column", j, ?  x[j] )`
- `gl_inp ( "column", j, ?  x[j] )`

  Assuming that x is a two-dimensional array, these match on "column" and j, returning data into x[j]. The length is ignored.

1.3.4 gl_rd and gl_rdp Examples

The functions gl_rd and gl_rdp are used to read a tuple in tuple space. The function gl_rd waits for a match before returning a tuple, while gl_rdp returns a value of 1 if there was a matching tuple available and a 0 if a tuple was not available, without waiting. Unlike gl_in and gl_inp, gl_rd and gl_rdp only read a tuple; they do not remove the tuple from tuple space.

- `gl_rd ( "data", i, k, ? value )`
- `gl_rdp ( "data", i, k, ? value )`

  These match on "data" i and k returning data for value. Value can be a scalar or an array. If it is an array the length is implicit.

- `gl_rd ( "row", i, ? x:len )`
- `gl_rdp ( "row", i, ? x:len )`

  These match on "row" and i returning data for the array x and returns the number of items as len.

- `gl_rd ( "column", j, ? x[j] )`
- `gl_rdp ( "column", j, ? x[j] )`

  Assuming that x is a two-dimensional array, these match on "column" and j, returning data into x[j]. The length is ignored.

1.3.5 gl_outto and gl_into Examples

The functions gl_outto and gl_into make use of PVM 3.x's multicast capability. The function gl_outto outputs a tuple directly to a process using its PVM task id number, bypassing the tuple server. The function gl_into is used to receive a tuple sent using gl_outto.

- `gl_outto ( tid, "data", i, k, value )`
- `gl_into ( "data", i, k, ? value )`

  In this example, the PVM task id number tid is used to output a tuple containing "data" i, k, and value. Value can be a scalar or an array. The corresponding gl_into matches on "data" i, and k returning the data for value.

- `gl_outto ( tid : len, "data", i, k, value )`
- `gl_into ( "data", i, k, ? value )`

  This example is the same as the one above, but instead of sending the tuple to only one process, we are sending the tuple to all of the processes whose PVM task id numbers are in the array tid with length len. The length is only needed when tid is an array.

2 Installation

If you received the ".tar" version, to install Glenda, uudecode glenda.tar.Z.uue by typing

```
> uudecode glenda.tar.Z.uue
```

then decompress glenda.tar by typing

```
> uncompress glenda.tar.Z
```

and in the directory you wish to place Glenda type

```
> tar -xf glenda.tar
```

If you received the ".shar" version, you will have received about 6 email messages which should be saved into separate files. Each file should be edited to remove the header lines as instructed within the files. Then each file should be used as input for "sh" as in:

```
> sh glenda.shar.1
> sh glenda.shar.2
...
```

Either of the distribution methods will create glenda directories for you within the current directory. The directories created are as follows. The top level directory is named **glenda**. It contains the Glenda source code contains a Makefile and several subdirectories:

gcpp The C-Glenda preprocessor code

doc Documentation

examples Examples of gl code

include Include files for ts and examples

ts Tuple server code

To build Glenda you can use "make ARCH=SUN4" or whatever your architecture is. You might prefer to edit the Makefile in this directory and those in ts and examples to prevent compiling for the wrong architecture.

The Makefiles are designed expecting PVM_ROOT to be defined as the root directory for your PVM3.x software and expect to be able to write to the $PVM_ROOT/bin/$ARCH directory.

SGI machines will require the option "-lsun" when linking in order to link in the XDR routines. This must be changed in the ts/Makefile.

This version of the tuple server and support code is written for PVM 3.x and would need a little effort to connect to PVM 2.4. We have a file named pvmold.c which is nearly complete for providing PVM 3.x function calls using the PVM 2.4 library. The main missing piece is that the tuple server and gl_user.c get task information using pvm_tasks to determine which task is the tuple server. Pvmold.c does not have a version of pvm_tasks.

3 C Glenda Preprocessor

The C Glenda preprocessor converts the source file containing the glenda functions into a ".c" file, capable of being compiled by a c compiler. It is capable of detecting errors in the glenda syntax as well as detecting type errors. Usage is as follows:

```
cgpp filename.cg
```

The source code file must end in ".cg" in order to be valid. In order to make it easier to compile your programs which make use of glenda, this sample makefile will help. Make sure that you have a directory named "bin" in your home directory and that ARCH is set to the proper architecture type.

```
ARCH     =      RS6K
PVMBIN = $(PVM_ROOT)/bin/$(ARCH)
PVMINCLUDE = -I$(PVM_ROOT)/include -I../include
```

```
CC =        c89

CFLAGS = -g $(PVMINCLUDE)

PVMLIB  = $(PVM_ROOT)/lib/$(ARCH)
LIB = -L$(PVMLIB) -lpvm3 -lm

USER = ../gts/gluser.o

# This part converts a ``.cg'' file to a ``.o'' file.
# The default .SUFFIXES parameter had to be changed to
# accomplish this.
# The -mv command can be removed at your convienience.
.SUFFIXES:
.SUFFIXES: .o .cg .c .f .y .l .s

.cg.o:
        -cgpp $*.cg
        -$(CC) $(CFLAGS) -c $*.c
        -mv $*.c $*.x

# Place each master file and its corresponding slave file here.

all:    $(PVMBIN)/a $(PVMBIN)/b

$(PVMBIN)/a:        a.o
        $(CC) -o $(PVMBIN)/a a.o $(USER)  $(LIB)
        chmod go+rx $(PVMBIN)/a

$(PVMBIN)/b:        b.o
        $(CC) -o $(PVMBIN)/b b.o $(USER)  $(LIB)
        chmod go+rx $(PVMBIN)/b

clean:
        rm -f *.o
```

4 Tuple Server

Before invoking the tuple server, you must first invoke PVM 3.x. After invoking PVM with the proper configuration parameters, simply type "gts" at the prompt. The tuple server will automatically be placed in the background and will continually be trying to receive tuples. Now you are ready to start up your master process which was written using Glenda.

****Important****

Do not invoke the tuple server without invoking PVM first. Do not invoke your master process without invoking the tuple server first.

A typical sequence of commands is as follows:

```
> pvmd pvmhosts &
> gts
> master_filename
```

5 Sample Glenda Usage

This is an example of code using Glenda. The first file is "a.cg" The following file is the corresponding slave file, "b.cg". In this example, process a outputs an array to multiple b processes and waits for each process's response.

```
a.cg

#include <stdio.h>
#include <glenda.h>

main(argc,argv)
int argc;
char *argv[];
{
    int my_tid, a;
    int Size, N, i;
    int *Data;
    int j, Kids;
    int kid, step;

    my_tid = gl_mytid();

    if ( argc > 1 ) Size = atoi(argv[1]);
    else Size = 100000;

    if ( argc > 2 ) N = atoi(argv[2]);
    else N = 10;

    if ( argc > 3 ) Kids = atoi(argv[3]);
    else Kids = 10;

    gl_out ( "Size", Size );
    gl_out ( "N", N );

    for ( i = 0; i < Kids; i++ ) {
        gl_spawn ( "b" );
        gl_out ( "Kid", i );
    }

    Data = (int *) malloc ( Size * sizeof(int) );

    for ( j = 0; j < N; j++ ) {
        printf("Step %d of %d\n", j+1, N );
        for ( i = 0; i < Kids; i++ )
            gl_out ( "data", i, Data:Size );
        for ( i = 0; i < Kids; i++ ) {
            gl_in  ( "OK", ? kid, ? step );
            printf("Got OK from %d for step %d\n",kid,step+1);
        }
    }
```

```
    gl_in ( "Size", Size );
    gl_in ( "N", N );

    gl_exit();
}
```

b.cg

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

main(argc,argv)
int argc;
char *argv[];
{
        int my_tid, a;
int Size, N, i;
int *Data;
int k;

my_tid = gl_mytid();

gl_rd ( "Size", ? Size );
gl_rd ( "N", ? N );
gl_in ( "Kid", ? k );

fprintf(stderr,"Kid %d, Size %d, N %d\n",k,Size,N);

Data = (int *) malloc ( Size * sizeof(int) );

for ( i = 0; i < N; i++ ) {
gl_in ( "data", k, ? Data:Size );
gl_out ( "OK", k, i );
}

gl_exit();
}
```

These files, and other example files, are located in the directory **/glenda/examples/**. The file “mm.c” is a matrix multiplication program using PVM function calls. This file was acquired from the newsgroup comp.parallel.pvm and was created by Josef Fritscher, Technical University of Vienna. The files “mmgl.cg” and “mmgl_worker.cg” are glenda versions of “mm.c” and its worker program “mmworker.c”. The files “mmto.cg” and “mmto_worker.cg” are also glenda versions of “mm.c” and “mmworker.c”, but make use of gl_outto and gl_into.

6 Plans

These are a few things that are being or will soon be pursued.

- Multiple tuple severs to improve efficiency

- Improved hashing of tuples
- Faster data transfer
- Tuple visualization utility to aid in debugging
- Image display utility
- Tuple recovery ability in case of system crash
- A Fortran Glenda preprocessor
- Call back routines to handle tuples

7 How to obtain Glenda

To obtain a copy of the Glenda software, e-mail your request to `seyfarth@whale.st.usm.edu` and a copy will be sent to you as soon as possible. In the future, an anonymous ftp server may be set up to better facilitate distribution of the software.

8 Getting Help

Please send bug reports to `seyfarth@whale.st.usm.edu` and we will try to help. It would be helpful to describe your virtual machine configuration (hardware, PVM version), include a short segment of code illustrating the problem, and describe how it fails.

Good luck with the software.