LaBRI

TITLE:

A neutral mechanism for the detection of deadlocks under PVM

AUTHOR(s): Serge CHAUMETTE

ADDRESS:

mail: Université Bordeaux I,
LaBRI,
Laboratoire Bordelais de Recherche en Informatique,
351, Cours de la Libération,
33405 TALENCE,
FRANCE

tel. : +33.56.84.69.04

fax: +33.56.84.66.69

e-mail: chaumett@labri.u-bordeaux.fr

A neutral mechanism for the detection of deadlocks under PVM

Serge Chaumette and Bruno Teplier

Serge.Chaumette@labri.u-bordeaux.fr

LaBRI, Laboratoire Bordelais de Recherche en Informatique, Université Bordeaux I, 351 Cours de la Libération, 33405 Talence, France.

Introduction

The framework of this paper it that of distributed applications developed using the model of message passing parallelism[1]. We focus on the problem of debugging such applications. In this paper we propose a new debugging mechanism to the PVM[2] users community. This mechanism makes it possible to collect and centralize information related to communication. It is neutral[9] in the sens of CCS[8] and weak bisimulation. This mechanism is based on our previous work on *grafts* and *spies*[5, 6]. The purpose of this paper is mainly practical; the theoretical aspects can be found in [9]. We illustrate this mechanism by building and using a neutral deadlock detection tool.

Grafts and spies

A *graft* is a process that can be inserted at run-time in a communication channel in order to be fed with and to possibly retransmit the data carried by this channel (see figure 1). Using this mechanism does not require the user to modify its sources since the necessary support is set-up at compile-time.

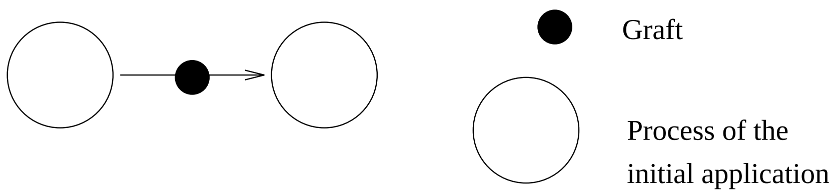

Figure 1: The basic mechanism of grafts

Grafts can be used for instance for visualizing the behavior of a distributed application or for testing its robustness – by inserting unexpected data into a communication channel –. Nevertheless, grafts are not a perfect tool for debugging purpose since they create a buffer effect that can modify the behavior of the initial application. Therefore we introduced the concept of a *spy*. A spy is a special form of graft that does not modify the content of messages before forwarding them. We have proven[4, 7] using CCS[8] that it is possible to build spies that do not modify, in the sens of weak bisimulation, the behavior of the initial program. These *neutral spies* can be implemented using a function that enable to wait for readyness of a receiver process. A neutral spy behaves as follows: wait for receiver to become ready; receive from sender; forward to receiver.

Grafts and spies have been implemented under Hélios[5] and under PVM[3, 6][1].

The scope of each spy is limited to the communication channel into which it has been inserted. It is therefore impossible for a spy to communicate with any other spy. Therefore we designed *C-spies* and *C-monitors* which we introduce in this paper.

C-spies and C-monitors

In order to be able to centralize the information collected by a set of neutral spies, we have extended our previous work. This led to the design of a new kind of neutral spy which we call a *C-spy*. A C-spy is not isolated to its own communication channel; it can communicate with the outside world. We also designed a process which we call a *C-monitor* to effectively centralize information. A C-monitor is allowed to receive a message from and to send a message to any C-spy in a non deterministic manner.

The key of the neutrality of this mechanism is that the combination of C-spies with a C-monitor is equivalent to a set of neutral spies, as previously defined. The neutrality of neutral spies directly leads to the neutrality of this new mechanism. We have proven this neutrality[9] using CCS and weak bisimulation.

[1]This work is supported by the Laboratory for High Performance Computing in the framework of a **LHPC**-C$^{\mathbf{3}}$ project

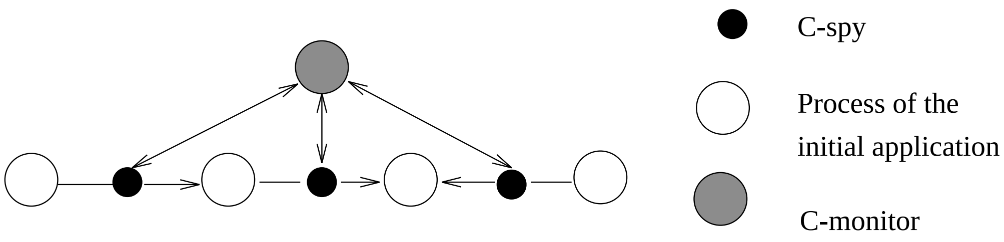

Figure 2: Communication of C-spies via a C-monitor

Neutral detection of deadlocks under PVM

In this section, we illustrate the use of C-spies and a C-monitor by implementing a neutral deadlock detection mechanism under PVM. We focus here on detecting deadlocks due to processes waiting for reception.

Being given the way C-spies can communicate with a C-monitor, the information required to implement a deadlock detection mechanism can easily be centralized into such a monitor. Each C-spy knows who are the sender and the receiver processes to which it is connected, and it is aware of the fact that the receiver is ready to receive. The form of C-spies allows them to forward this information to the C-monitor. Once a communication effectively happened, the involved C-spy signals it to the C-monitor. Then, the monitor is aware of pending communications.

It becomes easy to implement within this C-monitor an algorithm that detects a circuit of processes waiting for a message. When such a circuit is detected, it means that each process involved in it is blocked, waiting for a message from a process which is also blocked, and so on, what means that the application entered a deadlock.

— Source 1 : c-spy.c —

```
[...]
main(int argc, char **argv){

    int sender, receiver ,moniteur;

    pvm_mytid();

    _gs_setup(&argc, argv, &sender, &receiver);

    sscanf(argv[1],"%i", &moniteur);

    while (1){

        /* Wait for receiver to become */
        /*Ready to receive            */
        _gs_wait_for_receiver(receiver);

        /* Tell that to the C-monitor */
        msg_moniteur(moniteur, WAITING,
                     sender, receiver);

        /* Receive from the sender */
        _gs_mess_in(sender, receiver);

        /* Forward to the receiver */
        _gs_mess_out(sender, receiver);

        /* Tell that to the C-monitor */
        msg_moniteur(moniteur, DONE,
                     sender, receiver);
    }
}
[...]
```

— Source 1 : c-spy.c —

— Source 2 : c-monitor.c —

```
[...]
main(int argc, char **argv){

    int sender_tid, receiver_tid;
    int communication_status;

    pvm_mytid();

    while(1){

        /* Receive a message from any C-spy */
        pvm_recv(-1,-1);

        pvm_upkint(&sender_tid,1,1);
        pvm_upkint(&receiver_tid,1,1);
        pvm_upkint(&communication_status,1,1);

        /* Store state of communication */
        state_table[rank(sender_tid)]
                   [rank(receiver_tid)]
                       = communication_status;

        /* Is there a deadlock now ? */
        test_for_deadlock();
    }
}
[...]
```

— Source 2 : c-monitor.c —

Use example

To illustrate the use of this deadlock detection mechanism we have implemented a simple application that sets up a ring of processes called `site` (see source 3). A token is then injected into this ring. One of the processes of the ring called `init` (see source 4) measures the time it takes the token to make each of the turns. At the end `init` computes the mean time per turn.

— Source 3 : site.c —

```
[...]
main(int argc, char **argv){

    int tid[2];
    int token_flag=10;
    int loop;

    pvm_mytid();
    enter_ring(tid,argv);

    for (loop=0;loop<=NB_LOOPS;loop++){

        pvm_recv(tid[0],token_flag);
        pvm_initsend(PvmDataDefault);
        pvm_send(tid[1],token_flag);
    }
    pvm_exit();
}
[...]
```

— Source 3 : site.c —

— Source 4 : init.c —

```
[...]
main(int argc, char **argv){

    int tid[2];
    int loop;
    int token_flag=10;
    int duration[NB_LOOPS];
    double mean_time;

    pvm_mytid();
    enter_ring(tid,argv);

    for (loop=0;loop<=NB_LOOPS;loop++){

        duration[loop]=(int) time(NULL);

        /* Send token to the ring */
        pvm_initsend(PvmDataDefault);
        pvm_send(tid[1],token_flag);

        /* Receive token from the ring */
        pvm_recv(tid[0],token_flag);

        /* Compute duration of this loop */
        [...]
    }
    /* Compute mean duration of a loop */
    [...]
    pvm_exit();
}
[...]
```

— Source 4 : init.c —

We supplement this application with the support necessary to use C-spies and a C-monitor (see figure 3). This only requires recompiling the application and linking it with a library which we have developed.

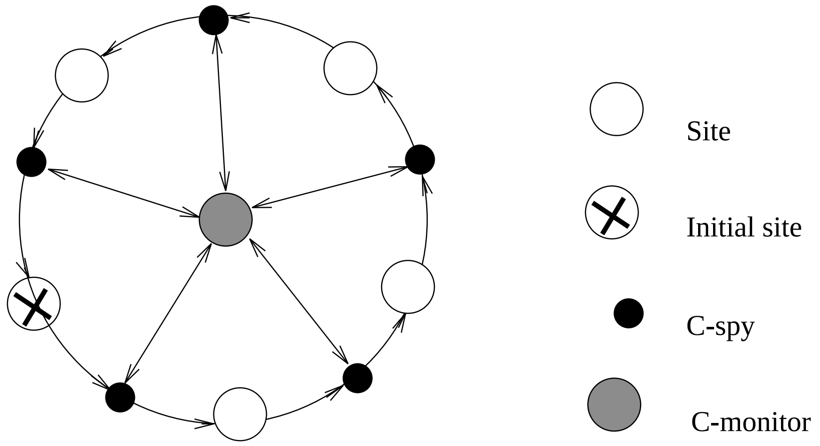

Figure 3: Detecting deadlocks in a ring using C-spies and a C-monitor

When running this application and spawning C-spies and the C-monitor (see source 5), it behaves exactly as if these processes did not exist till a deadlock is detected. The circuit of processes involved in this deadlock is then displayed.

This application runs up to a deadlock because when the last loop is entered by the process `init`, the fact of storing the current date into the `duration` table causes a violation of the boundary of this table and then the variable `token_flag` is overwritten[2]. This variable being used as a message tag for communication, the receiver

[2]This is due to the policy of allocation of local variables in the stack by the Gnu C compiler

will never be able to receive, hence to send, and so on.

— SOURCE 5 : Execution example —

```
$ pvm                                            |     pvm>
[...]                                            |     [2:t40003] Loop #0 took 1 second(s)
pvm> spawn -> init 4                             |     [2:t40003] Loop #1 took 2 second(s)
[...]                                            |     [2:t40003] Loop #2 took 1 second(s)
pvm> spawn -> c-monitor                          |     [2:t40003] Loop #3 took 1 second(s)
[...]                                            |     [...]
pvm> ps                                          |     [2:t40003] Loop #98 took 1 second(s)
           HOST      TID      FLAG COMMAND       |     [2:t40003] Loop #99 took 1 second(s)
           isis    40002     0x4/c pvmgs         |     [3:t40008] The following deadlock has been detected:
           isis    40003     0x4/c init          |     [3:t40008]
           isis    40004     0x4/c site          |     [3:t40008]  site_40007 ->init_40003 ->site_40004 ->
           isis    40005     0x4/c site          |                 site_40005 ->site_40006 ->site_40007
           isis    40006     0x4/c site          |     [...]
           isis    40007     0x4/c site          |     pvm> ps
           isis    40008     0x4/c c-monitor     |                HOST      TID      FLAG COMMAND
pvm> spawn -> c-spy 0x40003 0x40004 0x40008      |                isis    40002     0x4/c pvmgs
[...]                                            |                isis    40003     0x4/c init
pvm> spawn -> c-spy 0x40004 0x40005 0x40008      |                isis    40004     0x4/c site
[...]                                            |                isis    40005     0x4/c site
pvm> spawn -> c-spy 0x40005 0x40006 0x40008      |                isis    40006     0x4/c site
[...]                                            |                isis    40007     0x4/c site
pvm> spawn -> c-spy 0x40006 0x40007 0x40008      |                isis    40008     0x4/c c-monitor
[...]                                            |                isis    40009     0x4/c c-spy
pvm> spawn -> c-spy 0x40007 0x40003 0x40008      |                isis    4000a     0x4/c c-spy
[...]                                            |                isis    4000b     0x4/c c-spy
                                                 |                isis    4000c     0x4/c c-spy
                                                 |                isis    4000d     0x4/c c-spy
                                                 |     pvm>
```

— SOURCE 5 : Execution example —

Conclusion

In this paper we provide PVM users and more generally the message passing community with a new neutral mechanism to collect distributed information. We illustrate its use by building and using a neutral deadlock detection tool.

We believe that any support for programming distributed or even sequential applications must rely on a formal foundation that makes it possible to assert it really serves its purpose and it does not cause unexpected border effects. Therefore, we are using the neutral mechanisms we have already set up to design higher level tools. For instance, we are working on a system to perform neutral collection and visualization of communication related information.

References

[1] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surveys*, 21(3):261–322, 1989.

[2] A. Beguelin, J. Dongarra, G.A. Geist, R. Manchek, and V. Sunderam. A users's guide to PVM Parallel Virtual Machine. Technical Report TM–11826, Oak Ridge National Laboratory, August 1993.

[3] L Brunie, S. Chaumette, M. Cosnard, F. Desprez, L. Lefevre, M. Loi, M. Pouzandi, B. Tourancheau, and X. Vigouroux. The LHPC Programming Environment. In *Workshop on Environments and Tools for Parallel Scientific Computing*, 1994. To be published by SIAM.

[4] S. Chaumette. *Outils de développement et de mise au point pour la classe des machines parallèles à mémoire distribuée*. PhD thesis, Université Bordeaux I, 1992.

[5] S. Chaumette. Neutral debugging using grafts. *Information and Software Technology*, 36(7):465–470, 1994.

[6] S. Chaumette. Neutral observation of logical interprocess communication links in PVM applications. In *1994 PVM Users' Group Meeting, Oak Ridge, Tennessee*, May 1994. Also available on netlib.

[7] S. Chaumette. A step towards neutral debugging of distributed applications. In G.R. Joubert, D. Trystram, F.J. Peters, and D.J. Evans, editors, *Parallel Computing: Trends and Applications*, pages 273–280. Elsevier Science, 1994.

[8] R. Milner. A Calculus of Communicating Systems. *Lecture Notes in Comput. Sci.*, 92, 1980.

[9] B. Teplier. Technique de mise au point – communication inter espions. Mémoire de DEA, Université Bordeaux-I, 1994.