HEWLETT
PACKARD
PISA SCIENCE CENTER
HEWLETT-PACKARD LABORATORIES

HPL–PSC–92–53

November 5, 1992

The P^3L programming environment

Bruno Bacci and Salvatore Orlando

The programming environment of P^3L is not very dissimilar from those of other high-level languages. An ordinary editor can be used to write the program, and the programmer has not to be aware of the underling architecture. The compiler and the run-time will take into account the specific characteristics of the target parallel system.

The `p3l` command can be used to compile, as well as to visualize graphical outputs concerning the program to be compiled. Using an option of this command, the parsing of the program takes place, and the logical structure of the program is shown. The same command can be used to compile the program for a specific architecture. The current version of the compiler produces a process network that can be statically mapped on a distributed memory, MIMD architecture, with a *mesh* interconnection network. A graphical output showing the mapping of the processes is also produced.

The output of the compiler is a set of communicating processes with some mapping directive. More precisely, the output produced by the P^3L compiler is a P^3M executable as described in [MS92a]. The P^3M executable may be executed by using either a functional emulator, or a simulator that produces performance profiling figures [MS92b].

Contents

1 Hardware platforms **3**

2 Installing the P^3L compiler **3**

3 Writing, compiling and running a simple P^3L program **3**

4 The logical structure of a P^3L program **7**

5 The mapping of a P^3L program **7**

6 The execution levels of the P^3M code **7**

6.1 Functional emulation (level 0) 7

6.2 Proteus simulation and performance evaluation 8

6.2.1 Profiling that does not distinguish between user code and P^3M mechanisms (level 12) 8

6.2.2 Profiling that distinguishes between user code and P^3M mechanisms, whose costs cannot be changed (level 22) 8

6.2.3 Profiling that distinguishes between user code and P^3M mechanisms, whose costs can be changed (level 32) 9

6.2.4 Giving fixed costs to user code portions 11

7 Changing the AM costs used by the compiler **11**

8 Debugging a P^3L program **12**

8.1 Identification of the processes mapped on a given processing node 12

8.2 Running `xdb` 13

References **14**

1 Hardware platforms

The platforms on which the current version of the P^3L environment has been tested are HP Apollo 9000/730 and 9000/750 computers running HP-UX 8.07 OS. It should also run correctly on other 9000/7X0 machine adopting the same OS.

These platforms are the same as those used to test the Proteus simulation tool [MS92b]. Anyway, if we are not interested in running Proteus, we can adopt other platforms on which the P^3L environment, i.e. the compiler and the simulator running the parallel code produced by the compiler, can run.

The P^3L compiler is made up of three parts:

- the *front-end*, written in Lex and Yacc [Mac92b]. Thus this part should be easily recompiled for running on other UNIX platforms;
- the *middle-end*, written in HP-Prolog II. Since this Prolog environment is only supported by HP, it should be verified whether other, more portable, Prolog environments are able to run the middle-end. The graphical outputs produced by the middle-end should not represent a problem, because these outputs are files visualized by the `XFIG 2.1.4`, a public domain application;
- the *back-end*, also written in HP-Prolog II. As for the middle-end, we can make the same comments on the portability of the Prolog code. The code produced by the back-end is a P^3M executable [MS92a].

As regard the P^3M executable produced by the back-end, it is possible to run this executable by using a functional emulator instead of the Proteus simulator tool. This functional emulator has been built on the C++ task package. It should be tested whether such a package could be used, after recompiling, on a different UNIX platform.

2 Installing the P^3L compiler

The P^3L compiler is composed of

- a parser (front-end), which corresponds to an executable file `p3l-parser`. This file must be placed in a directory included in the shell variable `path`.
- a set of directories containing the HP Prolog II programs of both the middle-end and the back-end, as well as the C++ files containing the templates of the system-provided processes.

  All these directories must be installed as sub-directories of a given `<path-name-dir>` directory. The environment variable `P3L_INSTALL` must be set as follows:

  ```
  setenv P3L_INSTALL <path-name-dir>
  ```

- the `p3l` command, used to invoke the compiler and the tools of the environment. This command is a Bourne Shell script, which must be placed in a directory included in the shell variable `path`. The command:

  ```
  p3l -help
  ```

  reports and explains the various options of this command.

3 Writing, compiling and running a simple P^3L program

The following is a simple P^3L program [Orl92].

```
//
// This P3L program is composed of a pipe.
// The pipe is composed of:
//       1) a sequential stage "init";
//       2) a pure-farm stage "f";
//       3) a sequential stage "fin".
// The farm has a sequential worker "work".
//



/////////////////////////////////////////////////////
// sequential stage "init"
/////////////////////////////////////////////////////
init in() out(int x[10], float y[10])
    ${
        if (fscanf(stdin, "%d %f", &x[0], &y[0]) == EOF)
            p3l_exit();
        for (int i=1; i<10; i++)
            fscanf(stdin, "%d %f", &x[i], &y[i]);
    }$
end


/////////////////////////////////////////////////////
// sequential stage "fin"
/////////////////////////////////////////////////////
fin in(int x[10], float y[10]) out()
    ${
        fprintf(stdout, "x:\t");
        for (int i=0; i<10; i++)
            fprintf(stdout, "%-6d ", x[i]);

        fprintf(stdout, "\ny:\t");
        for (i=0; i<10; i++)
            fprintf(stdout, "%-4.1f ", y[i]);
        fprintf(stdout, "\n\n");
    }$
end



/////////////////////////////////////////////////////
// sequential worker "work"
/////////////////////////////////////////////////////
work in(float y[10], int x[10]) out(int x1[10], float y1[10])
    f  module.C
end


/////////////////////////////////////////////////////
// farm "f"
/////////////////////////////////////////////////////
farm f in(int x[10], float y[10]) out(int x1[10], float y1[10])
  work in(y, x) out(x1, y1)
end farm


/////////////////////////////////////////////////////
// pipe "p"
/////////////////////////////////////////////////////
pipe p in() out()
```

```
    init in() out(int x[10], float y[10])

    f in (x,  y) out(int x1[10], float y1[10])

    fin in(x1,  y1) out()
end pipe



main
  p in() out()
end
```

The programmer can write this program using one of the editing program provided by the UNIX environment.

Suppose that the program has been written in a file `<p3l-program-file>` of the current directory. The P^3L compiler can be run by launching the command

```
p3l <p3l-program-file>
```

from the UNIX environment.

This command will run all the three parts of the compiler.

The first part, i.e. the front-end, performs the parsing of the P^3L program and of the C++ code included. If some syntax errors are discovered, these are signaled by self-explanatory messages.

If the first part has completed successfully, then the second part of the compiler, i.e. the middle-end, begins to run. The middle-end has to access the costs of the abstract machines (AM), namely P^3M, in terms of communications and sequential code execution. Since the automatic interaction with the costs of P^3M has not been yet included in the current prototype of the compiler, the various costs must directly be provided by the programmer.

By launching the compiler command above, no costs are supplied. So, the middle-end produces a *test* mapping of the program, with a fixed parallelism degree for each P^3L constructor. Fig. 1 shows the mapping produced. This picture is stored in an `XFIG` file, `graph_demo_<p3l-program-file>`, also stored in the current directory.

Looking at Fig. 1, we can see that the program is represented as a directed graph, where each circle represents a process, and each arrow a communication channel.

A *type list* is associated with each channel, and both a number and a label are associated with each circle.

The type list corresponds to types of the problem data (the implementation data are not shown by this figure) to be transmitted on that channel. The number corresponds to the specific node of the architecture (mesh) on which that process has to be mapped. The label associated with each circle refers to the process to be mapped onto that processing node. Notice that at most one process is mapped onto each node.

The label corresponding to each process is a prefix string concatenated with a numerical, system-provided, identifier. Looking at the prefix strings, we can distinguish on which processing nodes the user-specified processes `init`, `fin`, and `work` have been mapped.

Once the second part of the compiler has completed successfully, the third and last part, the back-end, begins to run. The P^3M executable is produced. The executable is composed of many files, all stored in the current directory.

At this point, the P^3M executable can be run. The program needs to read from `stdin`, so a file containing the input stream must be provided. In this case, each item of the input stream must be a set of 10 integer values and of 10 float values. For instance

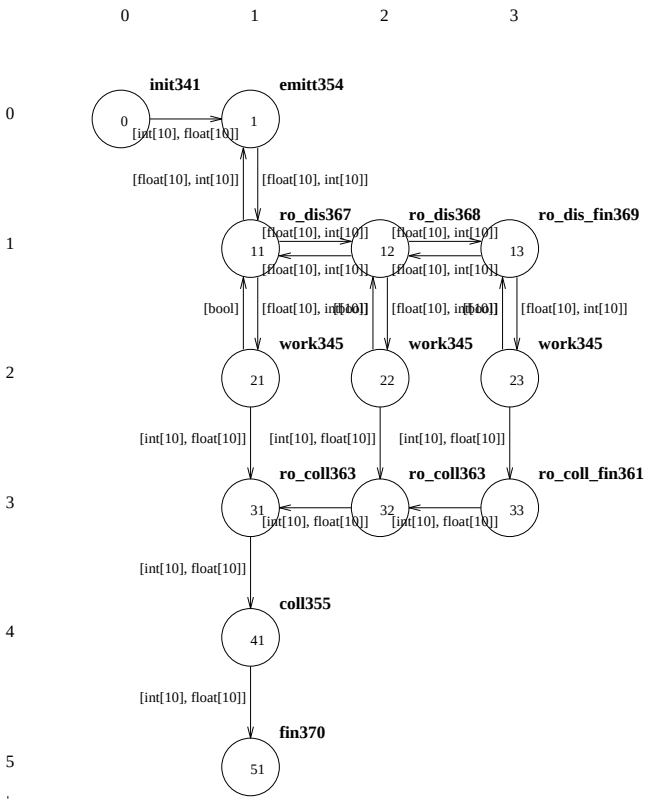

Figure 1: Mapping of a simple P^3L program

```
1     2     3     4     5     6     7     8     9     10
1.0   2.1   3.2   4.3   5.4   6.5   7.6   8.7   9.8   10.9

11    12    13    14    15    16    17    18    19    110
11.0  12.1  13.2  14.3  15.4  16.5  17.6  18.7  19.8  110.9
...
```

If the name of the file containing the input stream is `<stdin-file>`, then the following commands must be launched from the current directory [Mac92c]:

```
p3m -e 0 < <stdin-file>
```

This command will run the P^3M executable using the functional emulator (`-e 0`), and will produce the expected output of the P^3L program on `stdout`.

Alternatively, the following command can be launched

```
p3m -e 0 -ne
```

which produces an executable file `p3m.a.out` in the current directory. In fact, the option `-ne` means no execution.

Running `p3m.a.out`, starts the simulation. Therefore, the command:

```
p3m.a.out < <stdin-file>
```

will run the functional emulation of the P^3M executable, and the expected output of the P^3L program will be produced on `stdout`.

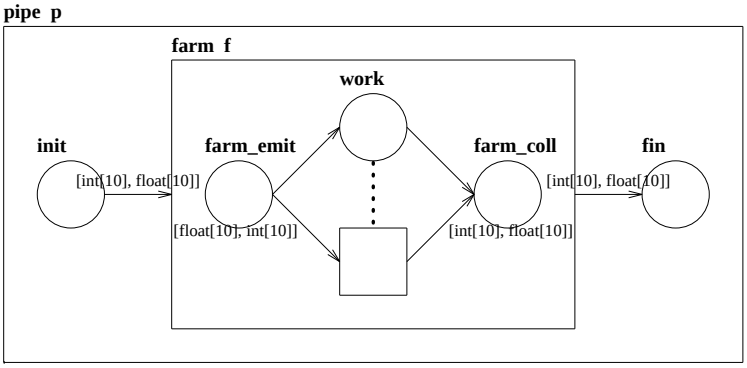

Figure 2: Logical structure of a simple P^3L program

4 The logical structure of a P^3L program

A P^3L program is hierarchically composed of many processes and parallel constructors. So, by only looking at the code, it can be difficult to distinguish this hierarchical composition.

Thus, the P^3L provides a tool, which produces a graphical `XFIG` object, stored in a file `logic_graph_demo_<p3l-program-file>` (also stored in the current directory).

The command for launching this tool is the following:

```
p3m -l <p3l-program-file>
```

This command causes the parsing of the P^3L program contained in `<p3l-program-file>`, and only if the parsing succeeds, it produces the graphical output.

If the `<p3l-program-file>` contains the simple program illustrated in the previous section, we obtain the graphical output shown in Fig. 2.

5 The mapping of a P^3L program

The graphical output produced by the middle-end shows the mapping of a network of processes. Not only user processes are shown, but also the ones provided by the P^3L compiler.

For instance, in Fig. 1 the processes `emit`, `ro_dis`, and `ro_dis_fin` are supplied by the compiler to implement the task distribution and the load balancing. The processes `coll`, `ro_coll`, and `ro_coll_fin` are instead supplied to implement the ordered collection of the results.

For each P^3L constructor several of these process structures can be generated by the compiler. The kinds of processes, their names, and their functionalities are deeply illustrated in [B+92].

6 The execution levels of the P^3M code

Several levels of execution are possible for the P^3M executables. We have shown only two options of `p3m`, the command for running the P^3M executable, i.e. `-e 0` and `-ne`.

All the possible options of this command can be consulted by launching the command:

```
p3m -doc
```

while a more detailed description can be found in [Mac92c].

6.1 Functional emulation (level 0)

The functional emulator (`p3m -e 0`) can be used during the development of the P^3L program for

- verifying the correct behavior of the program;
- debugging the program.

6.2 Proteus simulation and performance evaluation

The parallel profiling of a P^3L program running on a P^3M machine must be done by using Proteus [MS92b]. Currently, for verifying and evaluating the performance of a given run, we have to look at the graphical outputs produced by Proteus.

6.2.1 Profiling that does not distinguish between user code and P^3M mechanisms (level 12)

We have seen how the P^3M executable can be run using the functional emulator (`-e 0`). By only using another option, i.e. `-e 12`, the P^3L program will run on Proteus:

```
p3m -e 12 < <stdin-file>
```

Alternatively, the following command can be launched

```
p3m -e 12 -ne
```

which only produces an executable file `proteus` in the current directory. In fact, the option `-ne` means no execution. Running `proteus`, starts the proteus simulation. Therefore, the command:

```
proteus < <stdin-file>
```

will start the simulation, and will produce the expected output of the P^3L program.

This simulation performs the profiling of the various user processes allocated on the P^3M machine, as well as the system code of the machine. This code includes all the concurrent mechanisms of P^3M such as the loading of the program, the communication primitives, the interrupt handlers, etc.

The level 12 of simulation does not distinguish among these different kinds of code.

When the simulation stops, the graphical statistics of the simulation can be shown by means of the command:

```
stats
```

A window with several pull-down menus will appear. There are two Graphs menus. The former of the two menus contains an interesting option called Lifelines. It shows a diagram in which the *X axis* measures the time (expressed in clock cycles), while the numbers corresponding to the various processing nodes appear on the *Y axis*. These numbers are just those labeling the circles in the graph of Fig. 1.

The diagram shown by this option is called *Processor Lifelines*. For each processor, a horizontal black bar[1] is drawn whether the processor is busy during a given time interval. This kind of diagram is also called a *Gantt chart*.

Fig. 3 shows the diagram obtained from the profiling execution of a simple program, whose mapping is illustrated in Fig. 1. In this case the load is balanced among the three workers. The processors on which these workers are mapped are labeled with the numbers 21, 22, and 23. Notice the quite completely black bars corresponding to these three processors.

6.2.2 Profiling that distinguishes between user code and P^3M mechanisms, whose costs cannot be changed (level 22)

The `-e 22` option is used for running the P^3M executable on Proteus, taking into account the costs of the various P^3M mechanism. The simulation involved by this option distinguishes between the user code and the system code that implements the concurrent mechanisms of P^3M.

[1] On the screen this image appears in reverse mode, so the bars are white instead of black.

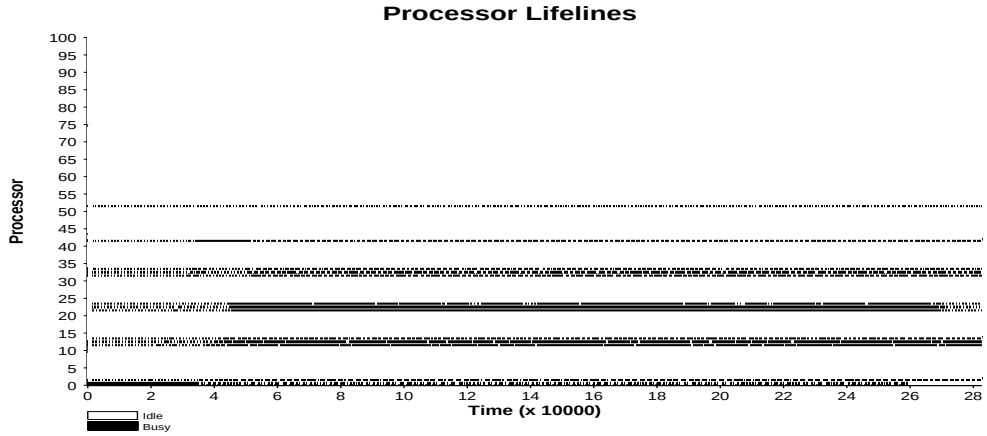

Figure 3: Lifelines of a simple P^3L program

The simulation done by the option `-e 22` is more accurate than the simulation of the option `-e 12`.

The commands

- to start the simulation with the new option, and
- to visualize the graphical statistics

are analogous of the commands discussed in the Sec. 6.2.1.

When this option is used, it is more interesting to consult another option of the pull-down menu Graphs of `stats`, i.e. the Modes option.

The diagram shown by the Modes option is similar to that shown by the *Processor Lifelines* option. The difference is that, in this case, different colors are used to distinguish between the execution of user-code, and other types of code.

Fig. 4 shows a *Processor execution modes* diagram. Both this diagram and that of Fig. 3 correspond to the the profiling execution of the same program. Because of the lack of colors, Fig. 4 does not print clearly in this document. Another reason is the large amount of tasks executed by the workers, so that the bars corresponding to the workers are composed of many short segments of different colors.

Anyway, by using the mouse, it is possible to draw a box on the windows to zoom in on of the figure. For instance, Fig. 5 shows a portion of the whole diagram of Fig. 4, obtained by zooming in on the window.

6.2.3 Profiling that distinguishes between user code and P^3M mechanisms, whose costs can be changed (level 32)

The last option that can be used for running the the P^3M executable on Proteus is `-e 32`. This option allows the user to change the costs of the mechanisms of P^3M.

Changes can be made to the part of the costs that are concerned with the processor execution of the various mechanisms (startup of communication mechanism, interrupt handlers, etc);

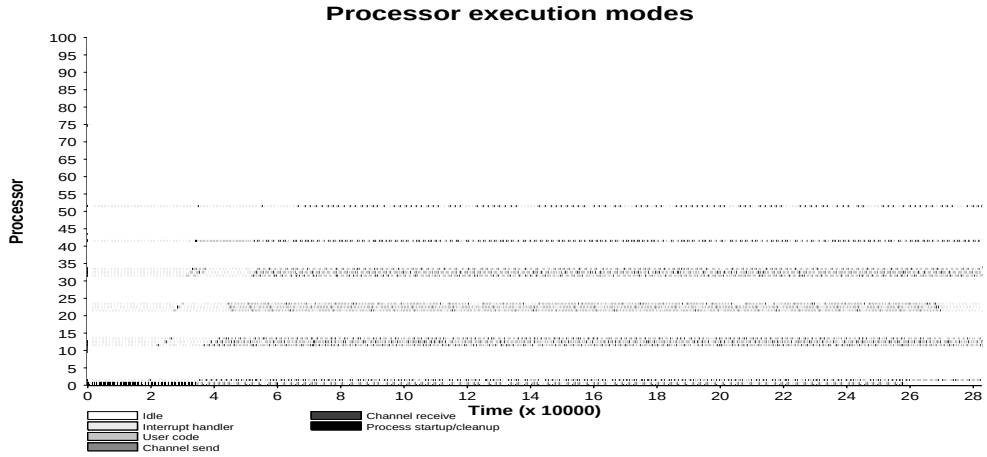

Figure 4: Modes bars of of a simple P^3L program

Figure 5: A portion of the Modes diagram

The other portions of these costs are concerned with the physical machine simulated by Proteus. For instance, the bandwidth of the communication links, the number of wires composing each link, etc. To change these parts of the costs, the Proteus manual should be consulted [BD91,MS92a].

The option `-e 32` of the command `p3m` runs the P^3M executable by using fixed costs for the various mechanisms. The values of these costs, given in terms of clock cycles, are included in a

default file `p3m-costs.C`. After running the P^3M executable with the `-e 32` option, this cost file is created in the current directory, and can be consulted and modified. For each mechanism, the file `p3m-costs.C` defines a *cost* as combination of several, fixed, elementary delays.

6.2.4 Giving fixed costs to user code portions

In many case, for instance for determining the performance of parallel programs whose actual code is not given, or for experimenting with particular structures of a parallel program, it is useful to give fixed costs to the user code.

The following is an example of P^3L sequential process that reads a pair of vectors, produces the same pair of vector, and takes `5000` clock cycles each time it is executed:

```
work in(int x[10], float y[10]) out(x, y)
    inc(timing.H)
    ${
         pass_time(5000);
    $}
end
```

It is worth noticing the `pass_time()` function used for this purpose. The prototype of this function must be declared in an include file by means of the following line:

```
extern "C" void pass_time(int time);
```

In the example above, this include file is `timing.H` (see [Orl92]).

7 Changing the AM costs used by the compiler

The compiler accesses the costs of the mechanisms exported by the target machine, which, in our case, is virtualized by an abstract machine (AM), namely P^3M.

The P^3L compiler makes several decisions on the basis of these costs. For instance, it decides the best parallelism degree to be exploited [B+92]. The costs of these mechanisms also depend on the specific program. Suppose that the compiler needs to know the cost of a send primitive. Usually, this cost is given by a formula such as the following

$$t_0 \ + \ t_1 \ nbytes$$

where t_0 is a constant (communication startup), while t_1 is a cost per byte. The factor $nbytes$ must be derived from the program, because it depends on the size of the specific message to be transmitted.

Until now, we have not yet implemented the automatic generation of these costs for each program to be compiled. Thus, these costs must be supplied by the programmer.

However, if the programmer launches the command

```
p3l <p3l-program-file>
```

a "test" mapping is produced, and no costs must be provided.

On the other hand, by using the following command

```
p3l -t <costs-file> <p3l-program-file>
```

the compiler accesses the costs included in the file `<costs-file>` of the current directory. If this file does not exist, then the `p3l` compiler asks the programmer several questions about the costs of the primitives employed for the implementation of the various P^3L constructors composing the specific program. All these cost are stored in the file `<costs-file>` first, and are subsequently used by the compiler to devise the optimal mapping for the program.

8 Debugging a P^3L program

The P^3L programs running on the *functional emulator* of P^3M can be easily debugged by using a standard UNIX debugging tool such as `xdb`.

Unfortunately, the programs produced by the P^3L compiler are already mapped on a specific P^3M machine, and are composed of a set of user- and system-provided processes. Looking at this process graph is not the best way for debugging a P^3L program. We expect that, in the near future, a simpler tool for debugging P^3L programs will be provided. This tool should be based on a sequential interpreter of the P^3L language, based on its functional semantics.

Anyway, before using `xdb` on the P^3M program produced by the P^3L compiler, it is necessary to compile the processes to be debugged with the option `-g`. To compile all the processes with this option, the environment variable `CXXOPTS` should be set to `-g`. The command `p3m -e 0` uses this variables to recompile all the processes with the option `-g`. Sometimes, if the command `p3m -e 0` has already been launched, the command `p3m -c` (or `p3m -Clean`) must be used before recompiling (the `p3m -doc` command describes in detail the usage of these options).

8.1 Identification of the processes mapped on a given processing node

The P^3L programmer is usually interested in debugging the portions of code that he has supplied. Looking at the Fig. 1, the processes specified by the P^3L programmers are those mapped on the nodes `0, 21, 22, 23, 51`. A C++ procedure call corresponds to each of these processes. This call is included in a C++ module, which, in turn, may use other modules. In the following, we will illustrate how the entry procedure (and also other information) corresponding to each of these processes can be found.

The `p3m-pgm` file (created by the command `p3m -e 0` in the current directory) can be used to know the process calls and the corresponding modules. This file contains the definitions of the process templates, the mapping of these processes, the definition of the channels, and the parameter passing to each process.

However, for our debugging purpose, we have interested in the `TEMPLATE` and `STATIC` sections of the `p3m-pgm` file. The following is the `TEMPLATE` section corresponding to the program shown in Sec. 3, and whose mapping is illustrated in Fig. 1.

```
TEMPLATES
code(0) = seq0   BLOCK(seq_no_input0.C  |   ./p3l__0__.C   |   )
stacksize(0) = 4000
code(1) = seq1   BLOCK(seq_synch1.C  |   module.C   |   )
stacksize(1) = 4000
code(2) = emit2   BLOCK(farm_emit2.C  |    |   )
stacksize(2) = 4000
code(3) = coll3   BLOCK(farm_coll3.C  |    |   )
stacksize(3) = 4000
code(4) = ro_dis4   BLOCK(ro_dis4.C  |    |   )
stacksize(4) = 4000
code(5) = ro_dis_fin5   BLOCK(ro_dis_fin5.C  |    |   )
stacksize(5) = 4000
code(6) = ro_coll_fin6   BLOCK(ro_coll_fin6.C  |    |   )
stacksize(6) = 4000
code(7) = ro_coll7   BLOCK(ro_coll7.C  |    |   )
stacksize(7) = 4000
code(8) = seq8   BLOCK(seq_no_output8.C  |   ./p3l__1__.C   |   )
stacksize(8) = 4000
```

Each `code(<templ-id>) ...` entry corresponds to a given process template, where `<templ-id>` is the numerical identifier of the template. Even if multiple instances of the same process template are mapped on distinct processing nodes, only one template entry appears in the `TEMPLATE` section. As an example, the process template that correspond to the worker of the farm is the

following:

```
code(1) = seq1 BLOCK(seq_synch1.C | module.C | )
```

where `seq1` is the call of the process.

The `BLOCK(<mod.C> | <other_mod.C> | <libs>)` structure is composed of three fields.

The first field `<mod.C>` corresponds to the file that contains the call of the process (`seq_synch1.C` in our example). The second field `<other_mod.C>` includes all the other files to be linked together to obtain the process (in our example, the only other file is `module.C`). The last field `<libs>` contains the libraries to be used to obtain the final process (in the example, this field is empty).

To distinguish which process template is mapped on each process, the `STATIC` section of the `p3m-pgm` file has to be used. The following is the `STATIC` section corresponding to the `TEMPLATE` section shown above.

```
STATIC
static_process(0, 51, 8)
static_process(0, 1, 2)
static_process(0, 13, 5)
static_process(0, 11, 4)
static_process(0, 12, 4)
static_process(0, 41, 3)
static_process(0, 33, 6)
static_process(0, 31, 7)
static_process(0, 32, 7)
static_process(0, 21, 1)
static_process(0, 22, 1)
static_process(0, 23, 1)
static_process(0, 0, 0)
```

Each entry of this section correspond to a processing node of the P^3M machine. In fact,

```
static_process(0, <proc-id>, <templ-id>)
```

shows which template process (identified by `<templ-id>`) has been mapped on a given processing node (identified by `<proc-id>`). Notice that the `<proc-id>`'s are the same numbers as those labeling the processes of Fig. 1.

In this example, the `farm f` has been mapped with three workers, where each worker corresponds to the same process template. In fact, we have three entries of the `STATIC` section that corresponds to the same process template, identified by `1`:

```
static_process(0, 21, 1)
static_process(0, 22, 1)
static_process(0, 23, 1)
```

8.2 Running `xdb`

The mapping graph (like that shown in Fig. 1), and the `STATIC` and the `TEMPLATE` sections of the `p3m-pgm` file can be used for identifying the call of a given process to be debugged.

However, to debug this process, the P^3L program must be recompiled with the `-g` option. A possible sequence of commands to obtain this is the following

```
setenv CXXOPTS -g
p3m -c
p3m -e 0 -ne
```

To run the `xdb` debugger by using a given file as `<stdin-file>`, the following command must be launched:

```
xdb -i <stdin-file> p3m.a.out
```

Suppose that the programmer wants to debug the worker of the `farm f` of our example. We have seen that the call of this process is `seq1`. Thus, a breakpoint must be set, by using the following `xdb` command:

```
b seq1
```

Notice that, in this example, we have three instances of the same worker process, so that the debugger will trace all the calls of these processes.

References

[B+92] B. Bacci et al. P^3L: a Structured High-level Parallel Language and its Structured Support. Technical Report HPL-PSC-92-DRAFT, Hewlett Packard Laboratories, Pisa Science Center (Italy), 1992. In progress.

[BD91] E.A. Brewer and C.N. Dellarocas. Proteus User Documentation Version 0.2. Technical report, Massachusetts Institute of Technology, October 1991.

[Mac92a] M. Mackey. P^3L interface to C++. Technical Report HPL-PSC-92-50, Hewlett Packard Laboratories, Pisa Science Center (Italy), 1992.

[Mac92b] M. Mackey. P^3L parser. Technical Report HPL-PSC-92-47, Hewlett Packard Laboratories, Pisa Science Center (Italy), 1992.

[Mac92c] M. Mackey. The `p3m` command. Technical Report HPL-PSC-92-48, Hewlett Packard Laboratories, Pisa Science Center (Italy), September 1992.

[MS92a] M. Mackey and T. Sullivan. P^3M machine interface definition. Technical Report HPL-PSC-92-43, Hewlett Packard Laboratories, Pisa Science Center (Italy), August 1992.

[MS92b] M. Mackey and T. Sullivan. Proteus user manual (PA-RISC). Technical Report HPL-PSC-92-44, Hewlett Packard Laboratories, Pisa Science Center (Italy), August 1992.

[Orl92] S. Orlando. The P^3L language manual (current implementation). Technical Report HPL-PSC-92-52, Hewlett Packard Laboratories, Pisa Science Center (Italy), 1992.