

Par4All & PIPS programming rules

Corinne ANCOURT Béatrice CREUSILLET François IRIGOIN
Ronan KERYELL

—
HPC Project & Mines ParisTech

April 23, 2012

Contents

1	Introduction	1
2	C language	2
2.1	Program structure	2
2.2	Data Structures	2
2.3	Pointers	3
2.4	Arrays	3
2.4.1	Declarations	3
2.4.2	References to array elements	4
2.4.3	Declaration scope	5
3	Operators and standard libraries	5
4	Values of scalar variables	5
5	Fortran language	6
6	Directives (OpenMP...)	6
7	Using libraries	6
8	Conclusion	6

1 Introduction

Automatic parallelization is an intractable problem in the general case and it would be an illusion to rely on it in any case. Fortunately, well written programs can expose enough parallelism in a way automatic parallelizer can do a pretty good job. We advocate that to expose parallelism even before manual parallelization, programs should be cleaned enough from specific sequential processor optimizations.

Here we present some coding rules specific to the PAR4ALL automatic parallelizer that is based on the PIPS source-to-source language transformation workbench but we think that these coding rules can be useful with other tools as interesting guidelines too.

This document gives advices to write programs that are compliant with the current state of PIPS and that maximize the precision of analyses and thus help the tool in generating efficient code. As a preliminary clue, and not surprisingly, let us say that the more structured a program is, the easier it is to analyze.

```

1 // malloc wrapper with error testing
void * my_malloc(size_t size, int err_code,
                const char * restrict err_mess)
{
    void * res = malloc(size);
6    if (res == NULL) {
        // This IO is an issue since it prevent parallelization:
        fprintf(stderr, err_mess);
        /* This exit is transformed as an unstructured control graph, reducing
           parallelization opportunities... */
11    exit(err_code);
    }
    return(res);
}

16 /* wrapper provided to PIPS. Note that it is not this function that has to
    be used in the compiled code by the backend */
void * my_malloc(size_t size, int err_code,
                const char * restrict err_mess)
{
21    // No more IO or exit()!
    void * res = malloc(size);
    return(res);
}

```

Figure 1: Wrapping system calls.

2 C language

In the following we assume the reader is knowledgeable of the C99 programming language and more specifically the ISO/IEC 9899:TC3 WG14 norm¹.

2.1 Program structure

PIPS internal representation of a function body is a hierarchical control flow graph, which simply means that syntactic constructs fit into each other as Russian dolls. For instance, a `for` loop is made of a header, itself made of three expressions, and a body, which can itself include other loops, tests... Most analyses and transformations rely on the good properties of these constructs to generate better results. On the contrary, a poorly structured program (which contains `gotos`, `breaks`, multiple returns...) leads to poor results and may prevent further code transformations.

PIPS includes a code restructurer, but it has its own limitations. So a good programming practice is to avoid non-structured code. Sometimes however it may not be possible. In particular, this is the case when testing system calls return values. But, however important these tests are, they do not affect the program semantics when there is a correct execution, and they are sometimes implemented differently on the targeted architectures. So a good practice would be to wrap system calls in dedicated macros or functions which would receive a simpler and non-blocking definition for PIPS, different from the real implementation that can be more complex. Figure 1 gives an exemple for the `malloc()` function.

Another solution is to wrap error code between `#ifdef ERROR_CONTROL ... #endif` pairs to be able remove to remove such code on demand.

C99 offers the possibility of declaring data almost anywhere in the code. This feature is relatively new in PIPS and phases are unequally ready to accept this feature, which means that you may gather all data declarations at the beginning of a code block.

One of the main other restrictions is that PIPS does not handle recursive functions.

2.2 Data Structures

The C language provides several basic data structures, which can be combined to create more complicated ones. PIPS allows all combinations, but with some limitations in their usage.

¹The last draft can be found here: <http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf>

2.3 Pointers

Pointers allow to designate a single memory location using several names: this is called aliasing. However practical it may be, it may render subsequent program analyses more complicated. To generate safe results, the default behaviour of PIPS is to generate imprecise results in case of pointer dereferencing. If you know that you very safely use pointers, you can turn off this default behaviour, by setting the right properties (see the `pipsmake-rc` guide for more details). `PAR4ALL`, as a pragmatic tool, can do this right for you, if you specify the `--no-pointer-aliasing` command-line option.

A pointer analysis phase, called *Pointer Values*, is currently being integrated to PIPS to avoid these drawbacks².

But if you want to make the most of `PAR4ALL`, just follow the next rules:

- always use the same expression to refer to a memory location inside a function; for instance, avoid the following kind of code:

```
1     parts = my_syst->domain[i]->parts;
     parts[j] = ...;
```

since `my_syst->domain[i]->parts` is designated by two different expressions. This limitation does not hold for array index: `my_syst->domain[i]` and `my_syst->domain[j]` are rightly distinguished if $i \neq j$.

- do not assign two different memory locations to a pointer (that means that a pointer could be considered rather as single assignment variable);
- in particular, do not use pointer arithmetic;
- reserve the use of pointers for the sole dynamic allocation of arrays and to function parameter passing;
- avoid function pointers, they currently lead to imprecise results.
- do not use recursive data structures such as linked lists, trees...

Several of these limitations will be removed in a near future, so stay tuned!

For example avoid some code that use pointers to do some strength reduction on array accesses such as:

```
1  double a[N], b[N];
   double *s = a, *d = b;
   for(int i = 0; i < N; i++)
       *d++ = *s++;
```

but use a clearer version that reflect the original algorithm (or that can be seen as an inductive variable detection on the previous code):

```
1  double a[N], b[N];
   for(int i = 0; i < N; i++)
       b[i] = a[i];
```

which is detected as a parallel loop!

2.4 Arrays

2.4.1 Declarations

As array manipulations are often the source of massive parallelism, PIPS program transformations rely on powerful analyses (convex array region analyses) of array element flows over the whole program.

²Expert users can use it to check a function pointer usage. For example in PIPS by activating the `PRINT_CODE_POINTER_VALUES` and displaying the `PRINTED_FILE` resource.

```

1  /* In the following statement sequences and expression, all reference to a
   and b are made in a following way. So first the region with a[i-1] and
   a[i] is built, which is compact, and the a[i+1] reference is added,
   leading still to a compact region. */
   tmp = a[i-1] + a[i] + a[i+1];
6  b[i-1] = ...;
   b[i]   = ...;
   b[i+1] = ...;

```

Figure 2: Consecutive array accesses.

These analyses currently assume that all array usages conform to the declarations (no array bound check is performed). Note that you can have variable size arrays in C99 (as available in Fortran for decades), such as:

```

1  int n = f();
2  int m = g();
   double a[3*n][m+7];

```

That removes many older linearization old cases as seen later.

2.4.2 References to array elements

Since array reference are represented in PIPS by using integer polyhedron lattice, array references should be affine so that the parallelizer can prove iteration independence. For example $a[2*i-3+m][3*i-j+6*n]$ is an affine array reference but $a[2*i*j][m*n-i+j]$ is not (it is a polynomial of several variables).

This explain why you should not use array linearization to emulate access to multidimensional arrays as one-dimensional arrays, such as:

```

1  double a[n][m][1];
   double * p = a;
   for(int i = 0; i < n; i++)
     for(int j = 0; j < m; j++)
5    for(int k = 0; k < 1; k++)
       p[m*1*i + 1*j + k] = ...;

```

instead of the cleaner understandable version:

```

1  double a[n][m][1];
   for(int i = 0; i < n; i++)
     for(int j = 0; j < m; j++)
4    for(int k = 0; k < 1; k++)
       a[i][j][k] = ...;

```

In the first cluttered version we have a polynomial expression which can not be represented in the linear algebra framework of PIPS and this kind of loops can not be parallelized and communications on GPU cannot be generated.

The following version:

```

1  double a[n][m][1];
   double * p = a;
   for(int i = 0; i < n*m*1; i++)
     p[i] = ...;

```

can be parallelized in the OPENMP output but the communications cannot be generated in the GPU version yet. So avoid array linearization if possible.

In the future, we may implement transformations that delinearize this kind of array accesses.

To reduce their complexity, these analyses gather array elements in sets represented by convex polyhedra. This means that non-convex sets of array elements are approximated by sets that contain elements which do not belong to the actual set, and are thus imprecise. Of course, further analyses and transformations are more likely to succeed and produce efficient code if array region analyses are more precise. So, in case of successive accesses to array elements, it is recommended to group them as much as possible so that two consecutive accesses in the program flow can be represented by a convex set. As an example, you will prefer the version of figure 2 to the version of figure 3.

```

1  /* The array region access is first computed from a[i-1] and a[i+1] which
2     is non compact and then the a[i] element is added */
   tmp = a[i-1] + a[i+1] + a[i];
   /* The same for b */
   b[i]  = ...;
   b[i+1] = ...;
7  b[i-1] = ...;

```

Figure 3: Disjoint array accesses leading to imprecise array region analysis.

```

1  val1 = 0;
   val2 = 1;
3  for(i=0; i<n; i++)
   {
       ...
       val1 = val2;
       val2 = 1-val1;
8  }

```

Figure 4: Don't use this flip-flop coding style

2.4.3 Declaration scope

Even if PIPS is interprocedural and can deal with parameter passing through global variables, such global declarations should be avoided if possible and replaced by explicit argument passing in functions.

For example, the actual generation of GPU code may choke on some case of global arrays that are not well outlined and forgotten.

3 Operators and standard libraries

A huge effort has been made to integrate all standard libraries functions. However, depending on your operating system and/or your compiler version, some of your favorite ones may be missing. Don't hesitate to contact us on `pipsdev at cri dot ensmp dot fr` so that we can integrate them.

The effects of the cast operators on the analyses is currently under study: please use it sparingly.

4 Values of scalar variables

PIPS contains powerful semantics analysis of scalar variables values whose results are used by many other program analyses and transformations, such as dead code elimination, parallelization, array region analyses ...

To help these analyses, you should avoid embedding key scalar values involved in loop bounds, test conditions and array sizes and index expressions inside structured data types.

Also if you need flip-flop values inside a for loop, prefer the code of Listing 5 to the code of Listing 4.

```

1  for(i=0; i<n; i++)
2  {
       val1 = i%2;
       val2 = 1-val1;
       ...
   }

```

Figure 5: Prefer this flip-flop coding style

5 Fortran language

6 Directives (OpenMP...)

7 Using libraries

FFTW

Can be seen as a DSeL

8 Conclusion

PIPS/PAR4ALL is a powerful tool to generate code for a variety of architectures. To get the most of it, you should however follow some rules when building your application.

The latter should be as structured as possible, and use `for` loops like Fortran `do` loops preferably to `while` loops. Loop nest candidates for parallelization should not contain IOs or debug or error control code, or provide an easy way to switch them off.

Minimize the use of pointers as explained before, and don't use recursive data structures and function pointers. Then use your data as they are declared: in particular avoid array linearization and casting.

If some of these recommendations cannot be followed, then try to group the non-compliant code outside of loop nests which are good candidates for parallelization. This is a good practice, for instance, for heap allocations.

Even if these programming rules seem constraining, they are often considered as sound programming rules even for classical sequential programming. That means that (re)writing application to ease parallelization can be a good opportunity for code cleaning and modernization, independently of PIPS or PAR4ALL.