Init-time Shadow Width Computation through Compile-time Conventions (Report EMP CRI A-285)

Fabien Coelho (coelho@cri.ensmp.fr)

CRI, ENSMP, 35, rue Saint-Honoré,
F–77305 Fontainebleau cedex, France
Phone: +33 1 64 69 48 52

March 29, 1996

Abstract

In compiling stencil-like computations for distributed memory machines a standard technique is to allocate shadow edges to store remote elements from neighbor processors [1]. The efficient management of these edges by HPF compilers at subroutine boundaries is not obvious because the callers do not know the callees' requirements, thus it was proposed to allow their explicit declaration [2]. We present a technique to make this information available on entry in the program, ready to be used for heap and stack allocated data. It suits separate compilation of programs involving arbitrary recursions. The width computation is simply delayed at init-time and is based on simple compile-time conventions.

1 Problem

Extending the local declarations of distributed array so as to store remote elements is a standard technique for managing temporary data allocation of stencil-like computations. The management of these overlaps at subroutine boundaries is not obvious for non interpocedural compilers because of the following issues:

- the callees do not know about incoming array available overlaps.
- the callers do not know about the callees' requirements for overlaps.

The first problem is not a big issue. Indeed, the available overlap can be passed from the caller to the callee through conventional descriptors, and the callee program can be compiled to managed unknow but sufficient overlap declarations. However, if the overlap declaration is not sufficient, the callee must copy the parameter into a larger array in order to match the required overlap. It is desirable to avoid such useless runtime copies. In Adaptor[1], all arrays are heap allocated. When larger overlaps are required, the array is dynamically updated,

[1] Personnal communication from Thomas Brandes

and will remain with this larger allocation until its death. This limits the number of runtime copies required. However, they can be avoided.

2 Technique

We present a technique to make the callees' requirements available at runtime to the caller. This technique makes the following assumptions:

- explicit declaration interfaces about the callees are available to the caller.
- this interface describes the mapping of dummy arguments.
- the dummy arguments are passed by assumed shape.
- a subroutine can be compiled with a *unknown but sufficient* overlap for dummy arguments.
- the call graph may be arbitrary.
- data that may require overlap are heap or stack allocated (but not statically allocated at link time…).
- local overlaps required for each subroutine are compile-time constants.

Under these assumptions, simple compile time conventions allows to generate some initialization functions which will compute the overlaps at init-time. When the program is launched, these intialization functions are called to instantiate the needed overlaps for every array along the callgraph. Arbitrary recursion requires a fixed point computation, thus these initialization functions must be called till stability. They mimic the program callgraph and propagate the required overlaps from callees to callers. The technique is the following:

- compile-time
  - compile each routine with an *unkown but sufficient* overlap assumption about the dummy arguments.
  - global shadow width descriptors are created for the subroutine dummy arguments.
  - local shadow width descriptors are created for its local arrays.
  - an initialization function is created for these descriptors.
    * it calls the subroutine callees' associated initialization functions.
    * it uses its callees' exported descriptors to compute and update its own local and exported descriptors.
    * if exported descriptors are modified, then the computation is not yet stable and will have to be recomputed again.
    * care must be taken to manage recursion, that is not to run several time the same initializer for a given iteration.
- init-time
  - the main's initializer is called, and it will propagates the initialization to the whole callgraph.

- the processus is iterated till stability.
- the convergence of the iterative resolution is insured because:
  * each overlap is increasing.
  * the overlaps are bounded by the largest overlap required, which is a compile-time constant.

The idea of the technique is to propagate the possible actual to dummy argument bindings so as to compute the overlaps of all local variables, which will be available and large enough for all possible argument passing through the callgraph. The actual possible implementation described hereafter is obviously naive and not very efficient, and is just intended at showing the feasability of such a resolution at init-time and with separate compilation.

3 Example

For the example in [2] we would have (`$` is used as a special character to avoid any interaction with user's global identifiers):

- global initializer, to be called by `__main`.

```
int $_not_stable;
void $_initialize()
{
  int iter = 1;
  do {
    $_not_stable = 0;
    $_initialize_main(iter++);
  } while ($_not_stable);
}
```

- program `main` initializer:

```
static int $_main_a_1_lo; // LOwer overlap for A of MAIN, dim=1
static int $_main_a_1_up;
extern int $_sub_1_1_lo; // LOwer overlap of arg number 1 of SUB, dim=1
extern int $_sub_1_1_up;
void $_initialize_main(int iter)
{
  static int done = 0; // last iteration computed
  if (iter==done) return; // once per iteration!
  else done=iter;

  $_initialize_sub(iter); // CALLEES

  $_main_a_1_lo = $_sub_1_1_lo; // LOCALS
  $_main_a_1_up = $_sub_1_1_up; // max of requirements...
  // NO GLOBALS
}
```

- subroutine `sub` initializer:

```
int $_sub_1_1_lo = 0; // LOwer overlap for arg number 1 of SUB, dim=1
int $_sub_1_1_up = 0;
extern int $_not_stable;
void $_initialize_sub(int iter)
{
  int tmp;
  static int done = 0;
  if (iter==done) return; else done=iter;

  // NO CALLEES
  // NO LOCALS
  // GLOBAL DUMMY ARGUMENTS:
  tmp = 1;
  if (tmp!=$_sub_1_1_lo) {
    $_sub_1_1_lo = tmp;
    $_not_stable = 1;
  }

  tmp = 1;
  if (tmp!=$_sub_1_1_up) {
    $_sub_1_1_up = tmp;
    $_not_stable = 1;
  }
}
```

Note that a similar template is generated for every routine (here `main` and `sub`). The initialization routines do not require any interprocedural assumptions but the interface declarations which allow to reference the callees exported overlap descriptors. The iterative process is necessary to solve what is a fixed point computation over the callgraph, and the initial guard allows to manage arbitrary recursion and enforces that each initialization is run only once per iteration.

4 Algorithm

Here is the algorithm for the compiler generation of the required initialization functions, using the C language. When compiling routine r:

- for each distributed argument number i and dimension d that may be distributed and have an overlap generate the exported descriptors:

  `int $_`r`_`i`_`d`_lo = 0, $_`r`_`i`_`d`_up = 0;`

- for each argument number i and dimension d of callee c of r that may be distributed and require an overlap generates the imported descriptors:

  `extern int $_`c`_`i`_`d`_lo, $_`c`_`i`_`d`_up;`

- for each dimension d of local array of r named a that may be distributed and require an overlap generates the local descriptors:

  `static int $_`r`_`a`_`d`_lo, $_`r`_`a`_`d`_up;`

- generates the stability checker:

  `extern int $_not_stable`

- generates the initialization function:

  - generates the prolog of the function:

    ```
    void $_initialize_r(int iter)
    { int tmp; static int done = 0;
      if (iter==done) return; else done=iter;
    ```

  - for each callee c of r generates a call to its initializer:

    ```
      $_initialize_c(iter);
    ```

  - initialize each local array descriptor `$_`r`_`a`_`d`_`s (where a is the array name, d the dimension number, s is `lo` or `up`) as the maximum of the local overlap required l and the callee arguments a may be binded to in routine r.

  - initialize each global array descriptor `$_`r`_`i`_`d`_`s as above, and update the `$_not_stable` variable if it is modified:

    ```
      tmp = ...
      if (tmp!=$_r_i_d_s) {
        $_r_i_d_s = tmp, $_not_stable = 1; }
    ```

  - generates the initialization function postlog: `}`

- the following routine must be called before main for computing the overlap at init-time:

  ```
  int $_not_stable;
  void $_initialize()
  {
    int iter = 1;
    do {
      $_not_stable = 0;
      $_initialize_main(iter++);
    } while ($_not_stable);
  }
  ```

- stack or heap allocation of array a of r at runtime must use the overlap computed for each dimension, that is `$_`r`_`a`_`d`_`s.

- the caller must pass to the callees the available overlaps, which are insured to be greater or equal what is required.

5 Conclusion

We have presented a technique to compute at init-time the overlap required by callees, so as to avoid useless copies at runtime. It simply delays the interprocedural computation of this information when an interprocedural view of the program is possible in a separate compilation environment, that is at runtime! The technique suits heap and stack allocated data. It can be extended for:

- managing init-time known required overlaps (the required overlap was assumed to be a compile-time constant);
- managing dynamically allocated and pointed array overlaps, by computing at compile-time conservative assumptions about the possible aliasing of pointers and actual data.

Such a technique can also be used in the context of optimized library calls, if the initialization functions and argument descriptors are made available by these libraries.

References

[1] Hans Michael Gerndt and Hans Peter Zima. Optimizing Communication in Superb. In *CONPAR90*, pages 300–311, 1990.

[2] Carl D. Offner. *Shadow Width Declarations*. Digital Equipment Corp., January 1996. Proposal for HPF.