Embodying Parallel Functional Skeletons: An Experimental Implementation on Top of MPI

Jocelyn Sérot

LASMEA URA-1793 CNRS, Campus des Cézeaux, F-63177 Aubière Cedex, France,
e-mail: Jocelyn.Serot@lasmea.univ-bpclermont.fr

Abstract. This paper aims at presenting an experimental but practical implementation of a skeleton-based parallel programming methodology based upon the integration of the MPI message-passing interface and a state-of-the-art ML compiler. The combination of a small number of higher-level communication abstractions and a SPMD style of programming has proven to provide a safe and fast way of designing parallel programs without loosing efficiency. The usefulness of the approach has been demonstrated by parallelising a complete image processing application.

1 Introduction

The concept of algorithmic skeleton was introduced by Cole [1] and Skillicorn [3] on the basis that many explicit parallel programs may actually be viewed as instances of a small number of generic patterns of parallel computation. Since then, they have been advocated as a pragmatical but efficient way of exploiting parallelism, especially in functional programs, within which they can be expressed in a very natural way as higher-order functions (HOFs). Yet, despite all the above arguments, it appears that very few practical implementations have effectively been built to support the initial claim. This fact may be attributed to a maybe under-estimated weakness of skeletons, already pointed out by Kesseler in [2]: up to now, they had to be re-implemented from scratch for every target architecture. The work described in this paper aims at tackling this problem by using a portable message-passing specification like MPI as an intermediate level of abstraction between skeletons definition and implementation.

2 The ML/MPI interface

Our goal is to coordinate the activity of ML programs running in SPMD mode using MPI communication routines wrapped into higher-level communication functions (the skeletons). Restricting our approach to SPMD mode pragmatically provides the easiest path for building and running distributed ML programs, since the replicated copies of the code can be generated using any existing sequential ML compiler. All that is required is a (foreign) interface allowing ML functions to make calls to a few MPI communication routines. We give here the specification of such a minimal interface, in Caml[1]:

- `type pid = int` is the type for process ids
- `type tag = int` is the type for message tags
- `type status = { src: pid; tag: tag }`
  is the type for received message status
- `val comm_size : unit -> int`
  returns the number of processes currently running
- `val comm_rank : unit -> pid`
  returns the current process id
- `val ssend : 'a -> pid -> tag -> unit`
  `ssend msg dst tag` makes a synchronous, blocking send of message `msg` to process with id `dst`, with tag `tag`
- `val recv : pid option -> tag option -> 'a * status`
  `recv (Some src) (Some tag)` waits (blocking) for a message with tag `tag` coming from process having pid `src`.
- `val run : (unit -> unit) -> unit`
  `run f` initialise MPI context, evaluates function f within this context and terminates MPI.

All these functions were implemented using only seven MPI calls and a few lines of of "stub-code" for exchanging arguments and results between the MPI C functions and their ML counterparts.

3 The skeletons

In the light of our existing knowledge of recurrent parallelisation schemes for image processing, three skeletons were specified and implemented: `scm` (split, compute and merge), `farm` (data farming) and `filt` (data filtering). Each skeleton is actually given two equivalent definitions: a purely *applicative* one, and an *operational* one. The applicative definition, written once in the functional base language using sequential HOFs, allows skeletal programs to be prototyped rapidly on sequential platforms. The operational definition uses the MPI calls described in the previous section and is intented to run as a coordinated set of processes on the actual parallel target platform.

SCM The `scm` skeleton encompasses most of data-parallel decomposition strategies. Its signature and applicative definition are:

```
val scm : ('a->'b list) -> ('b->'c) -> ('c list->'d) -> 'a -> 'd
let scm split f merge x = merge (map f (split x))
```

Its operational definition using MPI `ssend` and `recv` calls follows:

[1] Caml is a strict, polymorphic, higher-order, publicly available ML dialect.

```
let scm split f merge x =
  let np = Mpi.comm_size () and id = Mpi.comm_rank () in
  if id = 0 then (******* ROOT ***************************************)
    let zs = split x in                               (* Split data *)
    let n = List.length zs in                (* Check if enough procs *)
    if np < n then failwith "scm: not enough procs" else
    let x,xs = List.hd zs, List.tl zs in
    let _ = List.iter_index       (* Send n-1 data chunks to workers *)
      (fun i x -> Mpi.ssend x (i+1) 0) xs in
    let y = f x in                (* Having kept one for home work .. *)
    let ys = List.map_index                  (* Now, wait for workers *)
      (fun i x -> let y,_ = Mpi.recv (Some (i+1)) (Some 1) in y) xs
    in merge (y::ys)           (* Merge partial results and return  *)
  else (*************** WORKERS **************************************)
    let x, _ = Mpi.recv (Some 0) (Some 0) in        (* Wait for data *)
    let y = f x in                                  (* Do my own work *)
    let _ = Mpi.ssend y 0 1 in                   (* Send result back *)
    merge []                                          (* Local result *)
```

FARM and FILT From the operational point of view, the main characteristic of the previous `scm` skeleton is its very "static" behaviour. In particular, no provision is made for ensuring an even load-balancing between workers. This aspect is addressed by the `farm` skeleton, the goal of which is to apply a given function to a list a data using a "pool" of workers of any size, dynamically taking care of work-load distribution. From a functional point of view, `farm` is simply a `map`. Its parallel implementation involves some kind of dynamic control of the worker processes by the farmer, implemented using MPI *tagged* messages.

The `filt` skeleton is a variant of `farm` in which workers only return results satisfying a given predicate.

A note on polymorphism Because they actually hide (encapsulate) all atomic `sends` and `receives`, our definitions of skeletons also appeared to solve a "classical" problem related to message-passing based programming in ML: as long as programs use polymorphic `sends` and `recvs` (like those defined in section 2) they cannot be statically type-checked [2] and it's the programmer's responsibility to ensure that these atomic calls are used in a consistent manner. By contrast, the signature of each skeleton automatically *enforces* these type constraints.

4 A complete example

The skeletons quoted in section 3 were used for the parallelisation of an image segmentation process that aims at extracting polygonal approximations of significant contours from intensity images. The application, initially written in

[2] At the implementation level, polymorphism is handled using "flattening" and "deflattening" primitives to transform non-functional ML values into streams of bytes.

sequential ML, comprises three main stages: edge detection, edge tracking and edge approximation. Edge detection is performed using Canny-Deriche algorithm (smoothing using row- and column-oriented gaussian IIR filtering, gradient computation and extraction of local edge maxima). Edge tracking merges potential edge points into lines that are subsequently filtered according to two criteria: minimum length and existence of at least one point with value above a given hysteresis threshold. Polygonal approximation finally converts these lines into lists of vertices.

Figure 1 illustrates the sequencing of these stages along with the top level Caml program showing the instantiations of the skeletons that were used for their parallelisation. The first four stages are all instances of regular, local processing in which the value of one pixel in the result image depends only on pixels within a fixed neighbourhood and the work load is uniformly distributed across the image. Hence the use of the `scm` skeleton with geometric partitioning[3]. Despite the fact that it is intrinsically a non-local operation, edge tracking is also implemented using the `scm` skeleton: edges are tracked separately in every partition, the `merge_edges` function being responsible of merging those that have been artificially "broken" at partition boundaries. The last three stages (namely `min_length`, `hysteresis` and `polyg_approx`), because they process data of unpredictable size, make ideal candidates for the `farm` and `filt` skeletons.

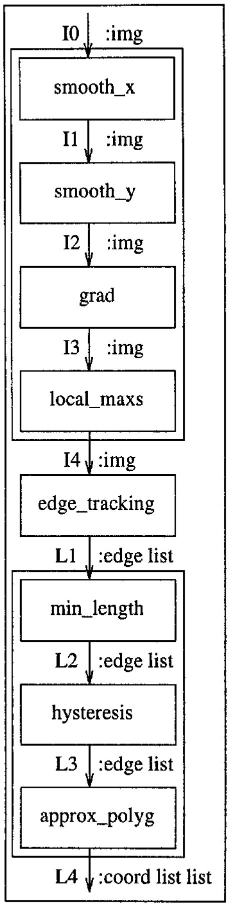
Getting a parallel version of the application actually boiled down to wrapping the eight calls `smooth_x` ... `polyg_approx` from the initial sequential Caml code into the corresponding skeleton HOFs and writing the `merge_edges` function. The resulting program has been tested using the MPICH implementation on a network of Sun4 workstations and showed correct behaviour and reasonable speedups.

5 Conclusion and further work

This work deliberately concentrated on two goals: ease of development and portability for functional skeletons. As a result many issues remain to be investigated. First, performance models for skeleton implementation are needed in order to help the programmer (or the compiler...) to select the most efficient skeleton for a given data profile and sequential function. Second, insofar as parallel applications are rarely written from scratch (and especially in ML !..), we definitely need a way of using existing sequential C code as skeletons arguments[4]. Finally, skeleton implementations could certainly be improved by using MPI *collective routines*, since parallel computers are likely to provide direct support for them in hardware and software.

[3] `row_block` (resp. `col_block`) and `block_row` (resp. `block_col`) are primitives of an `img` abstract data type (functor) handling the partitioning of images into sub-images (with various overlapping schemes) and their subsequent merging.

[4] Following in that the idea supported by Darlington *et al* in the SPP project [4].



```
let main () =
let f, s, t1, t2, lm, dm = get_params () in
let np, id = Mpi.comm_size, Mpi.comm_rank () in
let i0 = Img.read_file f in
let nr,nc = nb_rows i0, nb_cols i0 in
let i1 = scm
  (row_block np NoOverlap)
  (smooth_x s)
  (block_row NoOverlap) i0 in
let i2 = scm
  (col_block np NoOverlap)
  (smooth_y s)
  (block_col NoOverlap) i1 in
let i3 =
  scm (row_block np (Overlap 1))
  grad
  (block_row (Overlap 1)) i2 in
let i4 = scm
  (row_block np (Overlap 1))
  (local_maxs th1)
  (block_row (Overlap 1)) i3 in
let l1 = scm
  (row_block np NoOverlap)
  track_edges
  (merge_edges) i4 in
let l2 = filt (min_length lm) l1 in
let l3 = filt (hysteresis t2) l2 in
let l4 = farm (approx_polyg dm) l3 in
display_lines l4;
let _ = Mpi.run main
```

Fig. 1.

Acknowledgement This work was partially supported by EPCC, Edinburgh, under the TRACS program funded by EC TMR grant ERB-FMGE-CT95-0051.

References

1. M. Cole. *Algorithmic skeletons: structured management of parallel computations.* Pitman/MIT Press, 1989.
2. Marco Kesseler. Constructing skeletons in clean: The bare bones. In A. P. Wim Bohm and John T. Feo, editors, *High Performance Functional Computing*, pages 182–192, April 1995.
3. D. B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, December 1990.
4. J. Darlington, Y.K Guo, H.W. To, and J. Yang. Functional skeletons for parallel coordination. In *Proceedings of Europar*, 1995.