Skeleton Algorithms in Concurrent Clean

Viktória Zsók, Zoltán Horváth
Department of General Computer Science
Eötvös Loránd University, Budapest
H - 1117 Budapest, Pázmány Péter sétány 1/c
e-mail:zsv,hz@inf.elte.hu

Abstract

Skeletons can express very well the paradigms of the parallel programming. Functional skeletons are polymorphic higher order functions. The static part of a skeletal program is the description of the algorithm. The dynamic part can be described using strategies. Combining the skeletons with strategies an optimal efficiency can be obtained in the functional implementation of a parallel programming paradigm.

In the present paper parallel programming skeletal algorithms are investigated regarding the implementation in Concurrent Clean, efficiency, costs, abstraction levels. We have chosen four classical examples of parallel paradigm to illustrate the implementation and the use of the combination of the strategies with the skeletons.

1 Introduction

In the background of the functional skeletal programming is the functional skeleton theory given by J. Darlington [?]. The examples given in this article were first implemented in Haskell [?] and latter in Eden [?]. Nowadays there are significant efforts to standardise the parallel skeletons [?]. There are many advantages of the using of skeletons. The most important one is the reusability of the same algorithmic scheme for a whole class of problem, because they capture the common algorithmic form of the resembling problems. The design and the adequate implementation of the skeletons represent the counterpart work of the skeletal programming. There are disadvantages as well: the use of the low level language constructs, which decrease the portability of the program. It is difficult to transfer skeletons to another architecture because many of them are associated to a particular architecture. Another disadvantage is that the desired abstraction level of a program can be achieved very hard because it is difficult to abstract from the problem and thus decrease the generalisation of the problem.

Basically the skeletons are used to present in a very general form an algorithmic scheme. In the present paper they are used to present the principles of the parallel programming. The skeletons can be investigated regarding the design, the cost model of the implementation, the achieved abstraction level. On other interesting problem about skeletons is the transformation of a problem, which has to be applied in order to obtain a problem which can be solved with an already existing skeleton.

In functional languages they are higher order functions, they can be parameterised by functions, by types and by strategies. Depending on the goal we want to achieve, they can be combined with the strategies that are the most appropriate to the problem.

The implementation of the algorithms here are based on the lazy functional programming language Concurrent Clean. Concurrent Clean provides three type of language primitives for parallelism: annotations, used for parallel reducing of the expression; the sending and receiving channels for parallel communication inside one program and the thread for independent computations.

Evaluation strategies control the evaluation of the expressions. They have important role in the dynamic behaviour of a program. The specification of a strategy can be done separate from the program code, which makes the program code more clear. The parallelism provided by our programs is based on the strategies, on the possibilities offered by the strategies for parallel evaluation.

We have introduced in the [?] a module with evaluation strategy definitions for the lazy functional programming language Concurrent Clean. There are two essential strategy definitions the first one `par` for parallel evaluation and the second one `seq` for sequential evaluation. The composite strategies are based on these two. In order to achieve the parallel behaviour of a functional skeleton in Concurrent Clean the higher order functions are combined with the strategies. This combination will be the most exploited feature. A strategy is applied by the `'using'` function, which separates the static part from the dynamic part of the problem.

Our aim is to present some typical examples of parallel paradigms using functional skeletons combined with the basic strategies and the strategies derived from them.

2 Implementation of skeletons

2.1 Divide and conquer

The first example is the classical example of the divide and conquer paradigm which is recursively defined solution of a problem. The method consist in dividing the problem into sub-problems. The sub-problems will be solved independently after which the sub-result are combined in order to obtain the final result. The parallelism is introduced by solving simultaneously the sub-problems.

The Clean skeleton is the following:

```
dc t s d c str x
```

```
| t x = s x
| otherwise = cc (solv t s d c str (d x))
 where solv t s d c str xs = map (dc t s d c str) xs ‘using‘ str
```

The `t` parameter function solves the trivial case, the case obtained until which we continue the division of the problem into sub-problems. The `s` is the solution for the trivial case. The `d` function is defining the division into sub-problems. The `c` parameter combines the sub-results into a final one. The input is divided until we get the trivial case, for which we have solution. The `str` parameter is the strategy parameter and forces the evaluation to be done in parallel. Suppose there are the following functions: the `triv` function for the trivial case, the `solv` function for solving the trivial case, the `div` function for the division into sub-tasks, the `comb` function for the combination of the partial results into the final result. Using these functions the skeleton can be parameterised in the following:

```
Start = dc triv solv div comb (parlist rrnf) [3,9,7,6,2,8,5,4,1]
```

where the `parlist rrnf` will apply the elementary evaluation strategy `rrnf` to each element of the list in parallel and will determine the function to evaluate and to compute in parallel. The function can be defined for other linear data structures (eg. arrays), too.

In the presented algorithm we can observe that the number of the divisions executed by the skeleton on the initial data is equal to $\log_2 n$, where n is the length of the initial linear data.

Regarding this skeleton we can observe that the mechanism of the skeleton is the same of the associative operation. The function parameter of the skeleton must be an associative function and the final result can be obtained from the composition of the results of the sub-tasks.

Suppose we have a function f, an associative binary operator $\circ : H \times H$ and a sequence $x = \ll x_0, x_1, \ldots, x_n \gg$ of length $n \geq 3$. successively the following equality holds: $f \circ f(\ll x_1, x_2, x_3, \ldots, x_n \gg) = f(f(\ll x_1, x_2, x_3, \ldots, x_n \gg)) == f(\ll x_1, f_2, x_3, \ldots, x_n \gg)) == f(\ll x_1, x_2, f_3, \ldots, x_n \gg)) =$

Notation: $g_1 \circ g_2 = f(\ll g_1, g_2 \gg). We define the following auxiliary function g.$

$$g(x,i,l,k) = \begin{cases} f(\ll x_i, \ldots, x_i + l \gg) & \text{if } l > 1, \\ f(\ll x_i \gg) & \text{if } l1. \end{cases}$$

Lemma: Suppose we have the function f, an associative operator and the function g defined previously. The following equality holds:

$f(\ll g(x,i,l_1,k+1), g(x,i+l_1+1,l_2,k+1) \gg = g(x,i,l,k))$, where $l_1 + l_2 = l$

Proof: If $l_1 + l_2 > 1$ then $g(x,i,l_1,k) = f(\ll x_i, \ldots, x_i + l_1 \gg)$ in the same way we have $g(x,i+l_1,l_2,k) = f(\ll x_i + l_1 + 1, \ldots, x_i + l_2 \gg)$. Using the associative property of the function f the expression will be f$(\ll g(x,i,l_1,k+1), g(x,i+l_1,l_2,k+1) \gg = f(\ll x_i, \ldots, x_i + l_1, x_i + l_1 + 1, \ldots, x_i + l_2 \gg) = f(\ll x_i, \ldots, x_i + l_2 \gg) = g(x,i,l,k)$ because unifying two consecutive sequences of the level $k+1$ we get a sequence of the level k.

Based on this lemma we can say that if the function parameter of the Divide and Conquer skeleton has the associative property then the result can be obtained by the combination of the sub-results of the sub-tasks of the decomposition. Thus the initial problem can be solved by the recombination of the partial results of the division.

2.2 The farm

The farm skeleton is a simple form of the data parallelism. The skeleton is similar to the divide and conquer skeleton. It divides the input data into two parts, solves the two parts separately and than combines the results in the final result. The division of the tasks is done in one step. The skeleton is the following:

```
farm combine divide f x str = combine (map f (divide x) `using` str)
```

The skeleton requires a `combine` parameter for the combination of the two results, a `divide` function for division and a function `f` which solves the subproblems. Parallelism is included in the same way by the strategy parameter `str`. Comparing the two skeletons it can be seen than if the cost of the division of the input data into subgroups is too high, than choosing this farm skeleton is more optimal. In case of smaller input and smaller division costs the divide and conquer skeleton is appropriate.

This skeleton needs in the same way an associative function as parameter. Using the lemma of the previous subsection for the only one step of the division the final result can be obtained from the composition of the two partial results.

2.3 Pipe

This skeleton is very different from the others. The skeleton applies a sequence of functions to the elements of a data structure. By this method a linear parallelism is obtained. The `pipe` skeleton is:

```
pipe f fs x str = map (foldr (op) f fs) x `using` parlist str
```

The skeleton has the following elements as parameters: `f` the initial function, `fs` a sequence of functions, `x` a list (it can also be generalised to a linear data structure) and the strategy parameter `str`. The `op` operator can be for instance the function composition. The `parlist` strategy assures that the elements of the final result are computed in parallel.

The following lemma proves that the final result can be obtained using this pipe paradigm.

Suppose that the `fs` parameter is a sequence of functions: $fs = \ll f_0, f_1, \ldots, f_m \gg$. We apply the `op` operator to this sequence of functions using the standard function `foldr`.

Lemma: Let be $f^m = op f_0 (op f_1 (\ldots (op f f_m) \ldots). We use the notation$F = f^m. The following holds: $F(\ll x_1, x_2, \ldots, x_n \gg) = \ll F(x_1), F(x_2), \ldots, F(x_n) \gg$

Prof: By inductive method. For $i = 0$ we have $op f f_0(\ll x_1, x_2, \ldots, x_n \gg) = \ll (op f f_0)(x_1), (op f f_0)(x_2), \ldots, (op f f_0)(x_n) \gg$ because of the definition of the higher order function `map`, which applies its parameter function to each element of the input.

Suppose the property holds for $i = m$. This means that for $f^m = opf_0(opf_1(\ldots(opff_m)\ldots).\mathrm{F}(\ll x_1, x_2, \ldots, x_n \gg) = \ll F(x_1), F(x_2), \ldots, F(x_n) \gg$ is valid.

We have to prove that for $i = m{+}1$ it is valid too: $f^m{+}1 = opf_0(opf_1(\ldots(opff_m{+}1)\ldots) the equality \mathrm{f}^m{+}1(\ll x_1, x_2, \ldots, x_n \gg) = \ll f^m{+}1(x_1), f^m{+}1(x_2), \ldots, f^m{+}1(x_n) \gg$ holds.

For this $f^m + 1 = opf_0(opf_1(\ldots(opf_m(opff_m + 1)\ldots) = opf^m f_m + 1$.

But for $i = m$ we have $f^m(\ll x_1, x_2, \ldots, x_n \gg) = \ll f^m(x_1), f^m(x_2), \ldots, f^m(x_n) \gg$ then the $f^m{+}1$ can be computed in the following way: $f^m{+}1(\ll x_1, x_2, \ldots, x_n \gg) = \ll opf^m f_m{+}1(x_1), f^m f_m{+}1(x_2), \ldots, f^m f_m{+}1(x_n) \gg = \ll f^m{+}1(x_1), f^m{+}1(x_2), \ldots, f^m + 1(x_n) \gg$.

2.4 Reduce

This skeleton is a two level folding. The same decomposing idea is used by this skeleton. The input is divided into subgroups of data structures. The next step is to apply a parallel folding for each subgroup and in the end folding again the partial results.

```
reduce f init div dimension x = foldrr divx
   where
     foldrr = foldr f init
     divx = parMap rrnf foldrr (div dimension x)
```

The skeleton has as parameter the `f` operator and the initial value used at folding the subgroups. The requirement regarding the `f` is that it must be an associative operator. The `div` parameter is a division function and creates the subgroups of the size given in the `dimension` parameter. The divided data will be the `x` parameter. Parallelism is included by the `parMap` function, the parallel version of the `map` function introduced in [**?**].

This skeleton can be compared with the `pipe` skeleton in the same way the `farm` skeleton was compared with the `divide and conquer` skeleton. In the `pipe` skeleton there was applied only one time a folding operation applied on the function sequence, here there is multiple folding applied on the subgroups of the data. Comparing with the divide and conquer skeleton it can be observed a similarity, it is based on the same divide-solve-combine mechanism, but in this case on an other plan of action.

For this skeleton with the analogy of the pipe skeleton we can observe that dividing the input sequence into subparts and computing the results for this subparts and then using the folding operation separately for partial results gives the same result like folding the operation for the whole input sequence.

This is the next lemma, which will be demonstrated by induction.

Lemma:

3 Conclusions

The presented examples reinforce that the parallel paradigms can be very well expressed by skeletons, as was visible in other functional languages.

The parallelism in the lazy functional language Concurrent is achieved by the evaluation strategies. Many problems can be transformed to these basic skeletons, but also it is possible to design more complex skeletons using the basic ones. More general skeletons can be obtained similar to the elementwise processable functions using type parameterisation.

References

[1] Chandy, K.M., Misra, J.: *Parallel program design: a foundation*, Addison-Wesley, 1989.

[2] Cole, M.: Algorithmic Skeletons, In: Hammond, K., Michaelson, G., eds., *Research Directions in Parallel Functional Programming*, pp. 289-303, Springer-Verlag, 1999.

[3] Darlington, J., Field, A.J., Harrison, P.G., Kelly, P.H.J., Sharp, D.W.N., Wu, Q., While, R.L.: Parallel Programming Using Skeleton Functions, In: *Proc. PARLE '93 - Parallel Architectures and Languages Europe*, LNCS, Vol. 694, pp. 146-160, Springer-Verlag, 1993.

[4] Galán, L.A., Pareja, C., Peña, R.: Functional Skeletons Generate Process Topologies in Eden, In: *Int. Symp. on Programming Languages, Implementations Logics and Programs PLILP'96*, Aachen, Germany, LNCS, Vol. 1140, pages 289-303, Springer-Verlag, 1996.

[5] Klusik, U., Loogen, R., Steffen, P., Rubio, F.: Implementation Skeletons in Eden: Low-Effort Parallel Programming, In: Mohnen, M., Koopman, P., eds., *Implementation of Functional Languages, 12th International Workshop, IFL 2000*, Aachen, Germany, September 2000,

LNCS, Vol. 2011, pp. 71-88, Springer-Verlag, 2001.

[6] Hammond, K., Rebón Portillo, Á.J.:

HaskSkel: Algorithmic Skeletons in Haskell, In: Koopman, P. et al., eds., *Implementation of Functional Languages, 11th International Workshop, IFL'99*, Lochem, The Netherlands, September 1999, LNCS, Vol. 1868, pp. 181-198, Springer-Verlag, 2000.

[7] Horváth Z.: Parallel asynchronous computation of the values of an associative function, *Acta Cybernetica*, Vol. 12, No. 1, pp. 83-94, Szeged, 1995.

[8] Kesseler, M.H.G.: *The Implementation of Functional Languages on Parallel Machines with Distributed Memory*, PhD Thesis, Catholic University of Nijmegen, 1996.

[9] Okasaki, C.: *Edison User's Guide (Haskell version)*, Department of Computer Science, Columbia University, Online Document, May, 1999. http://www.cs.columbia.edu/~cdo/edison/.

[10] Serrarens, P.R.: Explicit Message Passing for Concurrent Clean, In: Hammond, K. et al., eds., *Implementation of Functional Languages, 10th International Workshop, IFL'98*, London, UK, September 1998, LNCS, Vol. 1595, pp. 229-245, Springer-Verlag, 1999.

[11] Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.J.: Algorithm + Strategy = Parallelism, *Journal of Functional Programming*, Vol. 8, No. 1, pp. 23-60, Cambridge University Press, January, 1998.

[12] Horváth Z., Zsók V., Serrarens P., Plasmeijer R.: Parallel Elementwise Processable Functions in Concurrent Clean, to appear in *Proceedings of the 5th International Conference on Applied Informatics*, Eger, Hungary, January 2001 and selected for *Computers Mathematics with Applications*, Elsevier.