Strategic SPMD
(DRAFT)

Kevin Hammond

Division of Computer Science, University of St Andrews,
North Haugh, St Andrews, Fife, KY16 9SS, Scotland.
Email: kh@dcs.st-and.ac.uk

Abstract

This paper extends the existing concept of evaluation strategies, higher-order functions that can be used to separate behaviour from algorithm in parallel functional programs, to support the SPMD model of parallelism. Our model allows SPMD sub-programs to be composed with other parallel paradigms, or to be arbitrarily nested with other parallel paradigms. This represents a major step towards supporting the Group SPMD model, a generalisation of SPMD, in Glasgow Parallel Haskell. It also represents the first attempt of which we are aware to model SPMD programming in a system supporting implicit communication.

In order to ensure that the necessary barrier communications are modelled correctly at the high level we use the implementation concept synchronisation through logically shared graph reduction. We therefore ensure, in common with other work on evaluation strategies, that no new constructs need be added to the host functional language other than basic ones to introduce parallelism and enforce sequential execution. In order to ensure accurate modelling of the SPMD paradigm it is, however, convenient to introduce a primitive to control task placement.

Although there is, as yet, no efficient implementation of the SPMD strategy, we are confident that this could be constructed using an algorithmic skeleton, or similar implementation technique.

1 Introduction

This paper introduces a new "strategic" parallel function for Glasgow Parallel Haskell (GPH) that allows SPMD-style (Single Program Multiple Data) [SMTD+95] programming to be specified and used in a more-or-less implicitly parallel functional language[1]. SPMD calls can be freely composed or nested with other parallel strategies, whether or not these conform to the SPMD model. Of course, there is no guarantee that such arbitrary nesting will result in efficient parallel behaviour (unlike the sequential composition of SPMD strategies), but this facility does allow the Group SPMD [RaRü96] model (a more general model of parallelism than SPMD) to be supported directly in GPH. We hope that, ultimately, this will allow us to exploit the derivational style of program development favoured by O'Donnell and Rünger [ODRü97], in order to simplify the construction of complex, irregular parallel programs.

Task Creation

The two basic GPH annotations are `par` and `seq`. ``a `par` b`` *sparks* `a` (creates a *lazy future* [MKH91] that may evaluate `a`) and continues with the evaluation of `b`, which will normally contain a use of `a`. ``a `seq` b`` evaluates first *a* then *b* in turn. It is thus an exact equivalent of the normal sequential composition operator ";" in an imperative language. This represents the "evaluate-and-die" model of parallelism [PCS89]. A precise operational semantics for these primitives has been constructed by Hall et al. [HBT+98].

---

1 With apologies to Pascal Serrarens [Ser98]

For example, a parallel Fibonacci function can be described as follows. Note the use of `seq` to ensure that `n1` is evaluated before `n2` in the body of the definition, so allowing the opportunity to create a remote task to evaluate `n1` if it is sufficiently large.

```
pfib :: Int -> Int
pfib n
       | n <= 1     = 1
       | otherwise = n2 `par` (n1 `seq` n1+n2+1)
             where n1 = pfib (n-1)
                   n2 = pfib (n-2)
```

Evaluation Strategies

In principle, it should be possible to understand the functional semantics of an expression without considering its behaviour. In practice, annotated parallel functional programs tend to obscure the functional semantics in order to introduce appropriate parallel behaviour. Evaluation strategies [THL+98] allow this undesirable effect to be avoided by introducing the notion of a strategy, whose purpose is to specify behaviour independently of the result of a function. There is a broad analogy both with coordination languages [GeCa92 and with so-called algorithmic skeletons [Col89]. This is discussed in depth in an earlier paper on evaluation strategies [THL+98].

Evaluation strategies possess the highly desirable properties of being both simple and powerful — they can be directly composed or nested; they are user-extensible to cover new models of parallelism; they are type safe; and since they are written using the host functional language, they do not need a special operational semantics. Since a strategy is used only for its behavioural effect, its result type is uninteresting (except to indicate that the strategy has been used, and so ensure synchronisation among composed strategies) — we therefore use Haskell's "trivial type" `()`, which contains exactly one value (also written `()`).

```
type Strategy a = a -> ()
```

The simplest evaluation strategies control only the degree of evaluation

`r0` — no reduction

`rwhnf` — Reduce to Weak Head Normal Form

`rnf` — Reduce to full Normal Form

```
class NFData a where
    r0, rnf, rwhnf :: Strategy a

instance NFData a => [a] where
     rnf [] = ()
     rnf (x:xs) = rnf x `seq` rnf xs
```

These strategies are defined as instances of the type class `NFData`.

Evaluation strategies are attached to functional (algorithmic) code through the `using` function. This function has two parameters: the first is an expression that defines some value; the second is a strategy that can be applied to that value in order to induce some desired behavioural effect.

```
using :: a -> Strategy a -> a
x `using` s = s x `seq` x
```

For example, this can be used to define a hyper-strict version of `quicksort` that will not return any part of its result until its sublists have been fully sorted.

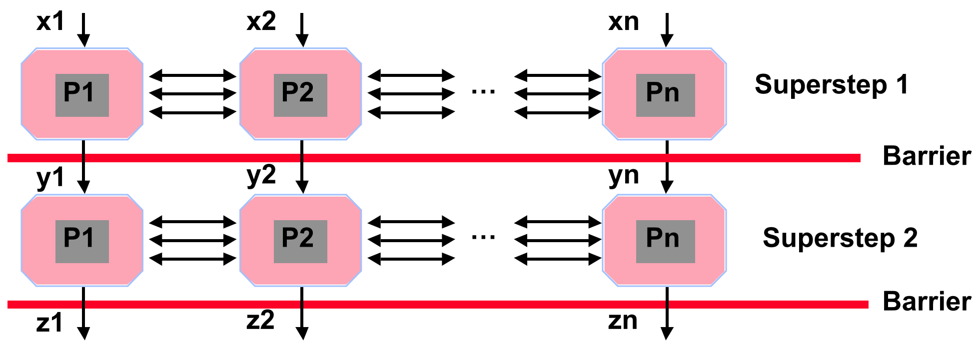

Figure 1: SPMD Programming

```
quicksort []   = []
quicksort [x]  = [x]
quicksort (x:xs)  = losort ++ (x:hisort) `using` rnf
       where losort = quicksort [y | y <- xs, y <  x]
             hisort = quicksort [y | y <- xs, y >= x]
```

2 SPMD Programming

The SPMD (Single Program Multiple Data) style of programming [SMTD+95] has become popular in the parallel programming community since it offers a way to decompose a parallel program in a simple structured way, whilst allowing relatively simple cost models of the resulting execution to be constructed. In the SPMD style, each processor has a copy of the program, and works on a subset of the input data. Each copy of the program operates in a number of supersteps, which combine computation and communication. Each superstep is separated by a barrier, after which all communication for that superstep is guaranteed to have terminated. This is shown diagrammatically in Figure 1.

The BSP (Bulk Synchronous Programming) model [SHM97] simplifies this model by requiring all communication to take place during the barrier steps. Thus in this model, execution of a parallel program on each processor is reduced to a sequence of interleaved computation and communication steps, some of which may be empty. The advantage is that a simple cost model can be constructed, with tight bounds on the actual execution cost. The cost is that excessive communications may be performed compared with the more flexible SPMD model.

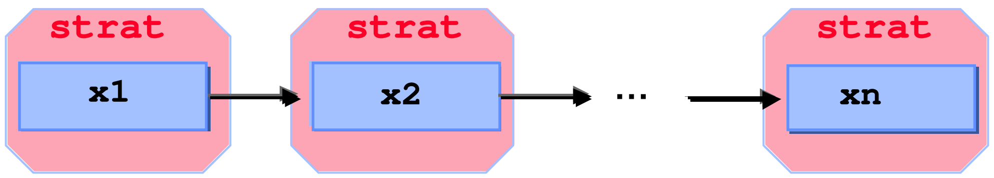

Figure 2: Data Parallel Strategy

Simple Data Parallel Evaluation Strategies

We have previously described a simple data parallel strategy whose purpose is to apply a strategy in parallel to all elements of a list, `parList` [THL+98]. The definition of this strategy for lists is repeated below. Its effect is as shown in Figure 2, where octagonal boxes represent parallel threads and `x1` etc. represent elements of the argument list.

```
parList :: Strategy a -> Strategy [a]

parList strat [] = ()
parList strat (x:xs) =
     strat x `par` (parList strat xs)
```

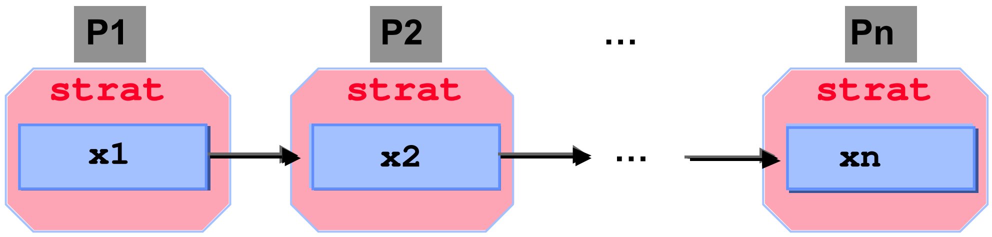

Figure 3: Placed Data Parallelism

This basic strategy can easily be extended to ensure placement of threads on particular processors, as shown in Figure 3, using the `placeList` strategy which is given below.

```
placeList :: Strategy a -> Strategy [a]

placeList strat l =
        placeList' l 1 where
          placeList' [] n =      ()
          placeList' (x:xs) n = parOn n (strat x)
                      (placeList' xs (n+1))
```

The definition of `placeList` uses a variant of `par`, `parOn`. `parOn n a b` is identical to `` a `par` b `` except that it is intended to spark `a` on processor `n`, consequently affecting the load balance. We deliberately leave the operational semantics of these processor identifiers vague (and avoid using topological operators such as the para-functional operators `left` and `right` [Hud86,MiHu95]) in order to allow alternative behaviours to be chosen by the runtime system if required, but our expectation is that these will normally map to independent processors in a good implementation of SPMD.

The definition of `parOn` is not a novel feature: para-functional programmers are accustomed to specifying processor locations using the `$on` annotation [Hud86], and Clean possesses a similar annotation amongst many others [Ser98]. The difference between those approaches and ours is simply one of emphasis: we are using these annotations to give additional hints to the dynamic runtime system, whereas in these other approaches, the annotations provide explicit control that must be adhered to. That is, we are working from a position that is implicitly parallel rather than one that is explicitly parallel [Ser98]. Our motivation for using hints rather than directives is that the latter unnecessarily restrict programs to a single topology, and prevent for example concurrent execution of different SPMD sub-programs on separate groups of processors when resources permit. We wish to allow these, and other possibilities in the system that we are constructing, while recognising that there is a special case where the parallel behaviour comprises only uses of SPMD-style program.

The SPMD Strategic Model

We define the SPMD model to be a variation of a conventional map function. This allows us to take arbitrary sizes of input. Like a normal map, each value in the result is a function of the corresponding value in the input. In the SPMD model we expect each of these results to be computed independently in a data-parallel fashion.

The SPMD strategic definition takes three arguments: an algorithmic function, a communications function (whose purpose is described below), and a list of values that the first argument is to be mapped over. It returns the list of results that is obtained by mapping the first function over the argument list, effectively the computation for a single superstep across all processors involved in the SPMD execution.

```
spmd    ::  (NFData b, NFData c) =>
        ([a] -> a -> b)          ->
        ([a] -> [b] -> c)        ->
        [a]                      ->
        [b]
```

Unlike a normal mapped function, the algorithmic function takes an additional argument which is the full list of inputs to the `spmd` function. This is needed to ensure access to all input data if needed (one variant of the SPMD model). The communications function is intended to expose all implicit communication in a superstep. It takes two arguments, the full list of inputs to the `spmd` strategy, and the list of results produced by the strategy. Two of the types (`b` and `c`) are overloaded to be any type that has the basic evaluation degree strategies (`r0, rwhnf, rnf`) defined for it.

The implementation can be described in strategic terms as follows. The most interesting part of the definition is the evaluation strategy that is applied to the result. This places each component of the computation across the available processors (ensuring that each component will be fully evaluated — by the `rnf` strategy argument to `placeList`), then forces the evaluation of the communications function, and finally forces the result to be produced. If the communications function is correctly defined, this will ensure that a barrier synchronisation occurs, since all values taking part in the communication must be fully evaluated before the SPMD implementation returns.

```
spmd f cf input =
        let result = map (f input) input
             comms  = cf input result
        in
              result `using` strat

        where strat res = placeList rnf res `seq`
                          rnf comms `seq`
                          rnf res
```

3 SPMD Examples

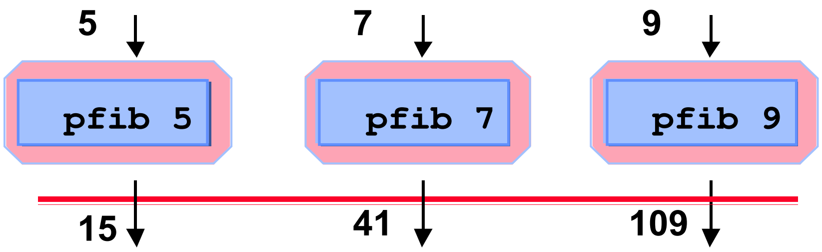

Figure 4: Simple SPMD Example

Simple SPMD Example

The first example simply distributes the function `pfib` across the data inputs as shown in Figure 4. The communication function is the trivial value, since nothing is communicated.

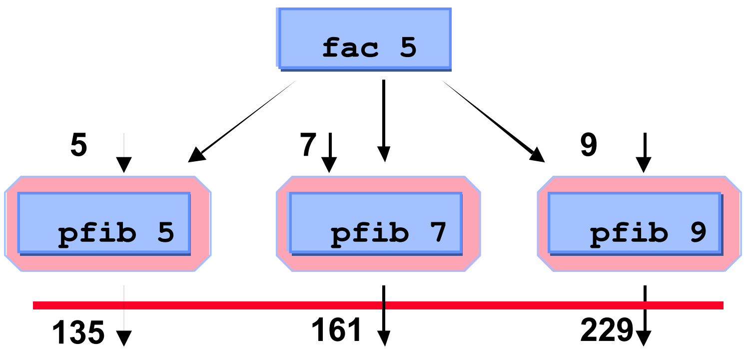

Figure 5: SPMD with Communicationn

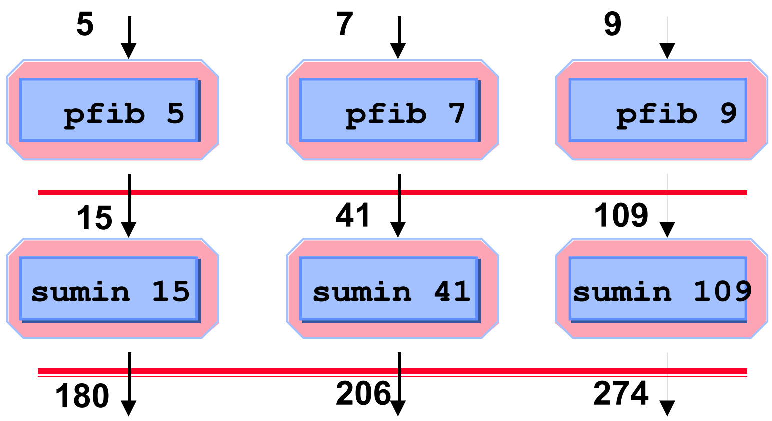

Figure 6: Multiple SPMD Supersteps

```
let     fib1 inputs v = pfib v
        nocomm inputs res = ()
in
        spmd fib1 nocomm [5,7,9]
```

Note that this example illustrates the ease with which paradigms can be nested, since `pfib` is itself defined as a function with parallel behaviour. It is not so easy to predict the effect of the combination of behaviours, however, since `pfib` has no task placement information.

SPMD with Communication

The second example introduces a shared constant that must be communicated among all processors as shown in Figure 5. The communication function defines this shared value, `const`.

```
let     const = fac 5
        fib2 inputs v = pfib v + const
        comm inputs res = const
in
        spmd fib2 comm [5,7,9]
```

Multiple Supersteps

The final example illustrates how a sequence of SPMD supersteps can be modelled. The program is written as a sequential pipeline using Haskell's $ operator (which is simply defined to be function application: `f $ x = f x`). The first superstep computes pfib for each input. The second superstep adds the sum of the results to this result. There is no implicit communication.

```
let     fib1 inputs v = pfib v
        sumin inputs v = v + sum inputs
        nocomm inputs res = ()
in
        spmd  sumin  nocomm           $
        spmd  fib1   nocomm           $
        [5,7,9]
```

4 Conclusions

We have defined a model of SPMD-style parallelism in the more-or-less implicitly parallel functional language GPH that supports nesting of arbitrary parallel paradigms, with a view to targetting the more general Group SPMD model of parallelism. This represents the first attempt that we are aware of to incorporate the SPMD style of programming in a language with implicit communication, though there have been previous (strict) functional language implementations that use explicit parallel control and communication, such as CamlFlight [FoCh95].

Although communication and synchronisation are normally implicit in GPH through logically shared graph variables, in the SPMD strategic implementation it is necessary to indicate communicated values in order to ensure full synchronisation between supersteps. While this is not a completely æsthetically pleasing solution, it does seem to be required with our present level of understanding of the SPMD model. Interestingly, omitting this communication function would not change the semantics of the program (provided all communicated values were needed), and might conceivably shorten execution time, but would not correspond directly to the SPMD model, and could potentially introduce parallel space leaks through delaying required computations.

While we have not yet mapped this definition to a real SPMD implementation, we believe that this should be possible by treating the definition as an algorithmic skeleton, and recognising the particular use of `rnf` to ensure barrier synchronisation. We therefore effectively propose to use skeletons as an implementation device in those situaton where they can be shown to be applicable, while also permitting more complex dynamic parallel structures to be defined in less regular cases.

Having constructed an efficient implementation of the SPMD strategic definition we intend to apply this to realistic applications, and as a target for the derivational system that has been proposed by O'Donnell and Rünger. Finally, it is an open question whether standard BSP/SPMD cost models hold for the system we have described. While we expect this to be the case, further work is needed to confirm this.

References

[Col89] Cole, M.I., *Algorithmic Skeletons*, Pitman/MIT monographs in Parallel and Distributed Computing, 1989.

[FoCh95] Foisy,C., and Chailloux, E., “Caml Flight: a Portable SPMD Extension of ML for Distributed Memory Multiprocessors”, *Proc. HPFC~'95 --- High Performance Functional Computing*, Denver, CO, 1995, pp. 83–96.

[GeCa92] Gelernter, D., and Carriero, N., “Coordination Languages and Their Significance”, *Comm. ACM*, **32**(2), Feb. 1992, pp. 97--107.

[HBT+98] Hall, J.G., Baker-Finch, C., Trinder, P.W. and King, D.J., “An Operational Semantics for Parallel Lazy Evaluation”, Submitted to *1998 Intl. Workshop on Implementing Functional Languages (IFL '98)*, UCL, London, Sept. 1998.

[Hud86] Hudak, P., “Para-Functional Programming”, *IEEE Computer*, **19**(8), 1986, pp. 60–71.

[MiHu95] Mirani, R. and Hudak, P., “First-Class Schedules and Virtual Maps”, *Proc. FPCA '9*}, 1995, pp. 91–99.

[MKH91] Mohr, E., Kranz, D.A., and Halstead, R.H., “Lazy Task Creation — a Technique for Managing the Granularity of Parallel Programs”, *IEEE Transactions on Parallel and Distributed Systems*, **2**(3), July 1991.

[ODRü97] O'Donnell, J.T. and Rünger, G., “A methodology for Deriving Parallel Programs with a Family of Parallel Abstract Machines”, *Proc. EuroPar~'97*, Springer LNCS 1300, August 1997, pp.~662--669.

[PCS89] S.L. Peyton Jones, C. Clack, and J. Salkild, “High-Performance Parallel Graph Reduction”, *Proc. PARLE '89 -- Conf. on Parallel Architectures and Languages Europe*, Springer-Verlag LNCS 365, pp. 193–206, 1989.

[Pey87] Peyton Jones, S.L. *The Implementation of Functional Programming Languages*, Prentice Hall International, 1987.

[PH+97] *Report on the Functional Language Haskell,* Version 1.4, Peterson J.W. and Hammond K. (Eds.), L. Augustsson, B. Boutel, F.W. Burton, J. Fairbairn, J.H. Fasel, A.D. Gordon, M.M. Guzmán, P. Hudak, R.J.M. Hughes, T. Johnsson, M.P. Jones, R. Kieburtz, R. Nikhil, W.D. Partain, S.L. Peyton Jones, and P.L. Wadler, 1997.

[RaRü96] Rauber, T. and Rünger, “Deriving Structured Parallel Implementations for Numerical Methods”, *Microprocessing and Microprogramming,* **41**, Special Issue on Parallel Systems Engineering, North Holland, 1996, pp.~589--608.

[Ser98] Serrarens, P., “More-or-less Explicit Parallelism in Clean”, Submitted to *1998 Intl. Workshop on Implementing Functional Languages (IFL '98)*, UCL, London, Sept. 1998.

[SHM97] Skillicorn, D.B., Hill, J.M.D., and McColl, W.F., “Questions and answers about BSP”, *Scientific Programming*, **6**(3), 1997, pp. 249–74.

[SMTD+95] Smirni, E., Merlo, A., Tessera, D., Haring, G., and Kotsis, G., “Modeling Speedup of SPMD Applications on the Intel Paragon: a Case Study”, Proc. HPCN~'95 --- High Performance Computing and Networks, Milan, Italy, 1995.

[THL+98] P.W. Trinder, K. Hammond, H.-W. Loidl and S.L. Peyton Jones, “Algorithm + Strategy = Parallelism”, *J. Functional Programming,* **8**(1):23—60, Jan. 1998.

[THM+96] Trinder, P.W., Hammond, K., Mattson, J.S. Jr., Partridge, A.S., and Peyton Jones, S.L., “GUM: A Portable Parallel Implementation of Haskell”, *(PLDI '96)*, pp. 79–88, Philadelphia, Pennsylvania, 1996.