Structured parallel programming: a generic co-ordination model and a parallel Fortran

John Darlington *Imperial College, London, UK*

Abstract

We present a framework for parallel programming that attempts to reconcile the conflicting requirements of abstraction, efficiency and compatibility with established practice. The key to this approach is the adoption of a *co-ordination approach* where applications are constructed in two layers, the first layer specifying the parallel behaviour of the program and the second providing the sequential threads that make up the components of this computation. We develop a *Structured Co-ordination Language* that is capable of expressing all relevant aspects of parallel computation in a uniform notation. The co-ordination operators of this language are represented as *functional skeletons*, pre-defined building blocks with tailored implementations onto particular parallel machines.
In this first paper we present the general framework of Structured Parallel Programming and the lowest level co-ordination forms that provide control over the behaviour of an idealised distributed memory parallel machine. We present a particular instantiation of this framework to provide a Structured Parallel Fortran and discuss its implementation and optimisation.

Keywords: parallel programming; co-ordination languages; skeletons

1 Introduction: the requirements of parallel programming

The designers of languages or systems for parallel application construction face several requirements that, at first sight, appear to conflict. The purpose of such languages must be to enable users to build complex applications on parallel machines as easily as possible. This should imply some abstraction or simplification of the programmer's task. However, the only purpose of using parallel machines is to enable the programs to execute quickly. Given the current complex and varied nature of parallel machines this seems to imply the need for low-level control of execution patterns and resource allocation. Finally, there is a often overlooked but very real requirement that any approach must be compatible with conventional and accepted practice for both psychological and technical reasons. The software industry has proven extremely resistant to revolutionary solutions, as evidenced by the survival of Fortran, and any software solution *must* execute on a widely accepted or standardised bases to ensure economic portability.
In this paper we present an approach that attempts to square these various circles. The basis of this method is the use of a *co-ordination language* to organise the parallel execution of program components that are themselves expressed in a conventional sequential language and the realisation of this co-ordination language by a pre-defined set of primitives or *functional skeletons*.

2 SCL: a generic parallel co-ordination language

In [2], Gelernter and Carriero proposed the notion of co-ordination languages as the vehicle of expressing parallel behaviour. In this article, they wrote:

> We can build a complete programming model out of two separate pieces—the *computation model* and the *co-ordination model*. The computation model allows programmers to build a single computational

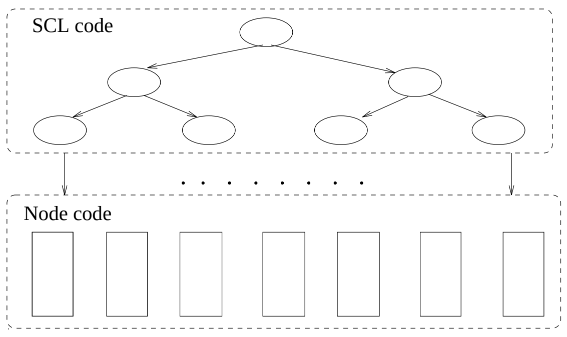

Figure 1: Two-tier structure of a SCL program.

> activity: a single-threaded, step-at-a-time computation. The co-ordination model is the glue that binds separate activities into an ensemble. An ordinary computation language (e.g. Fortran) embodies some computation model. A co-ordination language embodies a co-ordination model; it provides operations to *create* computational activities and to support *communication* among them.

We have adopted this approach to the problems of specifying and efficiently implementing parallel programs. In particular we have developed a Structured Co-ordination Language (SCL) that abstracts all the relevant aspects of a program's parallel behaviour. Programs written using SCL have a two-tier structure as illustrated in Figure 1. The upper layer is composed of compositions of SCL co-ordination forms, nested to any depth, that organise the parallel execution of the program while the lower level consists of procedures written in a conventional imperative language, such as Fortran or C, that express the sequential threads of this parallel computation. Note that there is a strict hierarchy, SCL co-ordination forms can be nested to any level but the imperative code cannot call an SCL form.

A second aspect of our approach is that SCL is not a general purpose language but is a set of pre-defined co-ordination forms or *skeletons* which are presented to the programmer as higher-order functions using a functional language syntax. The essence of a skeleton is that it provides a programmer with a pre-defined building block with which to construct his or her application. In our approach, because of the setting of co-ordination skeletons in a functional language, there is a clear separation between the *meaning* of a skeleton and its *parallel implementation*. Tailored, intelligent, implementations for skeletons can be provided onto particular target machines.

This breaking of the link between specifying what a program is intended to compute and how it will behave to achieve this is, we think, crucial to reconciling the requirements for abstraction and efficiency discussed earlier. SCL permits the specification and efficient implementation of higher-level, user-oriented co-ordination forms. In this paper we first present the lowest layer of SCL that provides a kernel on which the other more abstract layers can be efficiently implemented. This lower layer abstracts the operation required to efficiently program an idealised distributed memory parallel machine and provides a uniform notation to specify all relevant aspects of parallel behaviour: data partitioning and distribution, communication and multi-thread control.

2.1 SCL co-ordination primitives

SCL is a general purpose co-ordination language where all aspects of parallel co-ordination are specified as the composition of the following three classes of skeletons:

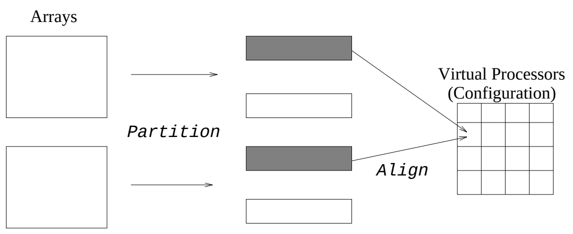

Figure 2: Data distribution model.

Configuration skeletons: co-ordinating data distribution The basic parallel computation model underlying SCL is the data parallel model. In SCL, data parallel computation is abstracted as a set of parallel operators over a distributed data structure, for example, a distributed array. A **configuration** models the logical division and distribution of data objects. Such a distribution has several components: the division of the original data structure into distributable components, the location of these components relative to each other and finally the allocation of these co-located components to processors. In SCL this process is specified by a `partition` function that divides the initial structure into nested components and an `align` function that forms a collection of tuples representing co-located objects. This model, illustrated in Figure 2, clearly follows and generalises the data distribution directives of HPF [3]. Applying this general idea to arrays, the following configuration skeleton `distribution` defines the configuration of two arrays `A` and `B`:

```
distribution (f,p) (g,q) A B =
      align (p ∘ partition f A) (q ∘ partition g B)
```

Throughout this paper we will use "curried" notation to define functions, thus `distribution` is a function defined with four arguments, the first two of which are pairs explicitly formed using the tupling notation, "`( , )`". The `distribution` skeleton therefore takes two functions pairs, `f` and `g` specify the required partitioning (or distribution) strategies of `A` and `B` respectively and `p` and `q` are bulk data-movement functions specifying any initial data re-arrangement that may be required. The `distribution` skeleton is defined by composing the functions `align` and `partition`.
A more general configuration skeleton can be defined as:

```
distribution [(f,p)] [d] = p ∘ partition f d
distribution (f,p):fl d:dl =
           align (p ∘ partition f d) (distribution fl dl)
```

where `fl` is a list of distribution strategies for the corresponding data objects in the list `dl` and "`:`" denotes the infix operator, known as "cons", that builds a list by adding an item to the front of the list.
Applying the `distribution` skeleton to an array forms a configuration which is an array of tuples. Each element `i` of the configuration is a tuple of the form $(\mathtt{D}A_1^i, \ldots, DA_n^i)$ where `n` is the number of arrays that have been distributed and $\mathtt{DA}_\mathtt{j}^\mathtt{i}$ represents the sub-array of the `j`th array allocated to the `i`th processor. As a short hand rather than writing a configuration as an array of tuples we can also regard it as a tuple of (distributed) arrays and write it as $\mathtt{<DA_1, \ldots, DA_n>}$ where the $\mathtt{DA_j}$ stands for the distribution of the array $\mathtt{A_j}$. In particular we can pattern match to this notation to extract a particular distributed array from the configuration.
Configuration skeletons are capable of abstracting not only the initial distribution of data structures but also their dynamic redistribution. Data redistribution can be uniformly defined by applying bulk data movement operators to configurations. Given a

configuration C: <$DA_1, \ldots, DA_n$>, a new configuration C′: <$DA'_1, \ldots, DA'_n$> can be formed by applying f_j to the distributed structure DA_j where f_j is some bulk data movement operator defined specifying collective communication. This behaviour can be abstracted by the skeleton `redistribution` as defined in [1]. SCL also supports nested parallelism by allowing distributed structures as elements of a distributed structure and by permitting a parallel operation to be applied to each of elements in parallel. An element of a nested array corresponds to the concept of *group* in MPI [6]. The leaves of a nested array contain any valid sequential data structure of the base computing language.

Elementary skeletons: parallel arrays operators In SCL, we use a set of second order functions as **`elementary skeletons`** to abstract essential data parallel computation and communication patterns. The basic functions specifying data parallelism include:

- `map` which abstracts the behaviour of broadcasting a parallel task to all the elements of an array.
- a variant of `map`, the function `imap` which takes into account the index of an element when mapping a function across an array.
- the reduction operator `fold` which abstracts tree-structured parallel reduction computation over arrays.

Data communication among parallel processors is expressed as the movement of elements in distributed data structures. In SCL a set of *bulk data-movement functions* are introduced as the data parallel counterpart of sequential loops and element assignments at the structure level. These elementary skeletons for communication can be generally divided into two classes: *regular* and *irregular*. The following `rotate` function is a typical example of regular data-movement.

```
rotate ::  Int → ParArray Int α → ParArray Int α
rotate k A = << i := A((i+k) mod SIZE(A)) | i ← [1..SIZE(A)] >>
```

Here the expression "`<< i := f i | i ← [1..k] >>`" is an "array comprehension" that denotes the array indexed from 1 to `k` whose `i`th element is `f i`.
For irregular data-movement the destination is a function of the current index. This definition introduces various communication modes. Multiple array elements may arrive at one index (i.e. many to one communication). This is modelled by accumulating a sequential vector of elements at each index in the new array. Since the underlying implementation is non-deterministic no ordering of the elements in the vector may be assumed. The index calculating function can specify either the destination of an element or the source of an element. Two functions, `send` and `fetch`, are provided to reflect this. Elementary skeletons can be used to define more complex and powerful communication skeletons required for realistic problems.

Computational skeletons: abstracting control flow In SCL the flexibility of organising multi-threaded control flow is provided by abstracting the commonly used parallel computational patterns as **computational skeletons**. The control structures of parallel processes can then be organised as the composition of computational skeletons. For example, in SCL, the `SPMD` skeleton, defined as follows, is used to abstract the features of SPMD (Single Program Multiple Data) computation:

```
SPMD []              = id
SPMD (gf, lf) :  fs  = (SPMD fs) ∘ (gf ∘ (imap lf ))
```

The skeleton takes a list of global-local operation pairs, which are applied over configurations of distributed data objects. The *local operations* are farmed to each

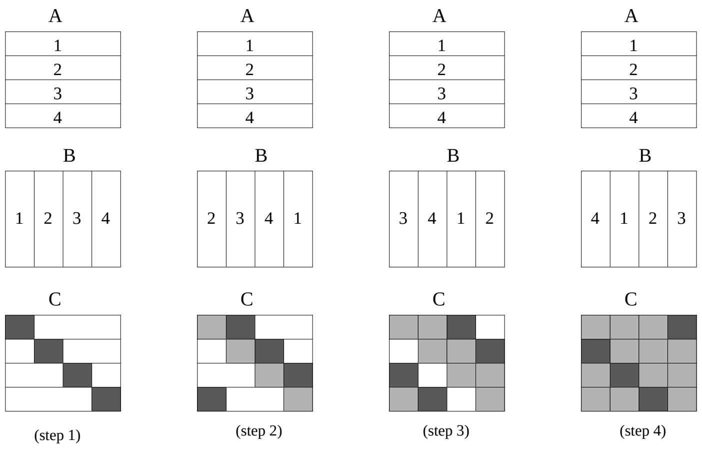

Figure 3: Parallel matrix multiplication: row-column-oriented algorithm.

processor and computed in parallel. Flat local operations, which contain no skeleton applications, can be regarded as *sequential*. The *global operations* over the whole configuration are parallel operations that require synchronisation and communication. Thus the composition of `gf` and `imap lf` abstracts a single stage of SPMD computation where the composition operator models the behaviour of *barrier synchronisation*. In SCL conventional control flow is also abstracted as computation skeletons. For example, the `iterUntil` skeleton, defined as follows, captures a common form of iteration. The condition `con` is checked before each iteration. The function `iterSolve` is applied at each iteration, while the function `finalSolve` is applied when the condition is satisfied.

```
iterUntil iterSolve finalSolve con x
          = if con x
            then finalSolve x
            else iterUntil iterSolve finalSolve con (iterSolve x)
```

Variants of `iterUntil` can be used. For example, when an iteration counter is used, an iteration can be captured by the skeleton `iterFor` using counter to control the iteration. By abstracting data distribution, communication and multi-thread control flow uniformly as basic skeletons, the SCL system supports the structured construction of parallel program by composing co-ordination skeletons using a set of well defined parallel data types.

2.2 Parallel Matrix Multiplication: A Case Study

To demonstrate the expressive power of SCL, we define the coordination structure of a matrix multiplication algorithm using SCL. The "row-column-oriented" matrix multiplication algorithm is adapted from [7].

Consider the problem of multiplying matrices $\mathtt{A}_{l\times m}$ and $\mathtt{B}_{m\times n}$ and placing the result in $\mathtt{C}_{l\times n}$ on `p` processors. Initially, `A` is divided into `p` groups of contiguous rows and `B` is divided into `p` groups of contiguous columns. Each processor starts with one segment of `A` and one segment of `B`. The overall algorithm structure is an SPMD computation iterated `p` times. At each step the local phase of the SPMD computation multiplies the segments of the two arrays located locally using a sequential matrix multiplication and then the global phase rotates the distribution `B` so that each processor passes its portion of `B` to its predecessor in the ring of processors. When the algorithm is complete each processor has computed a portion of the result array `C` corresponding to the rows of `A` that it holds. The computation is shown in the Figure 3.

The parallel structure of the algorithm is expressed in the following SCL program:

```
ParMM ::  Int → SeqArray index Float →
          SeqArray index Float → SeqArray index Float
ParMM p A B = gather DC
    where
        <DA, DB, DC> = iterFor p step dist
        dist = distribution fl dl
        fl = [(row_block p, id), (col_block p, id), (row_block p, id)]
        dl = [A, B, C]
        C = SeqArray ((1,SIZE(A,1)), (1, SIZE(B,2))
            [ (i,j) := 0 | i ← [1..SIZE(A,1)], j ← [1..SIZE(B,2)] ]

step i <DA, DB, DC> =
    SPMD [(gf, SEQ_MM i)] <DA, DB, DC>
    where
        newDist = [id, (rotate 1), id]
        gf X = redistribution newDist <DA, DB, X>
```

where `SEQ_MM` is a sequential procedure for matrix multiplication. Data distribution is specified by the `distribution` skeleton with the partition strategies of `[((row_block p), id), ((col_block p),id), ((row_block p),id)]` for `A`, `B` and `C` respectively. The data redistribution of `B` is performed by using the `rotate` operator which is encapsulated in the `redistribution` skeleton. The example shows that, by applying SCL skeletons, parallel co-ordination structure of the algorithm is precisely specified at a higher level.

3 Implementation

SCL is generic since the same co-ordination operations can be applied to sequential programs expressed in any conventional language. A particular structured parallel language is produced by applying SCL to a specific language. At Imperial College we have produced a Structured Parallel Fortran (SPF) by applying SCL to Fortran. As all parallel behaviour arises from the behaviour of known skeletons, the co-ordination primitives can be implemented by pre-defined libraries or code templates in the desired imperative language together with standard message passing libraries providing both efficiency and program portability. SPF has been implemented by transforming SPF programs into conventional parallel Fortran programs, that is sequential Fortran augmented with message passing libraries. A prototype system has been built based on Fortran 77 plus MPI [6] targeted at a Fujitsu AP1000 machine [4].
The expression of all parallel behaviour by the SCL layer enables parallel optimisation to be accomplished by program transformation on this layer, rather than by analysis of complex, imperative code. For example, an algebra of communication can be developed to optimising data-movement. Examples of these algebraic laws are:

$$\texttt{send f} \circ \texttt{send g} = \texttt{send (f} \circ \texttt{g)} \tag{1}$$

$$\texttt{fetch f} \circ \texttt{fetch g} = \texttt{fetch (g} \circ \texttt{f)} \tag{2}$$

$$\texttt{(rotate k)} \circ \texttt{(rotate j)} = \texttt{rotate (k + j)} \tag{3}$$

The use of such transformations can lead to considerable improvements in the cost of communication especially when communication can be completely removed. The algebraic axiomatisation of communication optimisation has been intensively studied in the context of developing an optimal compiler for conventional data parallel languages [5]. The commonly used approach is based on the analysis of index relations between two sides of an assignment. Since using SCL communications are explicitly specified in terms of a set of well defined communication operators, the index-based analysis can be systematically replaced by transformation rules abstracting the optimisation of communication behaviour.

4 Conclusion

The first, kernel, level of SCL and its realisation in SPF appears to meet at least the last two requirements of the three introduced earlier. Efficient programs can be written using SPF, optimised using transformation rules and compiled or expanded to code in conventional languages and communication libraries without any run-time overheads. Experiments have shown that SPF programs can achieve the same performance as hand written, low-level code. Furthermore as SCL is no more, or less, than a uniform design notation that can be applied to conventional languages and compiles to standard software platforms it is compatible with conventional approaches at both the psychological and technical levels.
The level of SCL shown so far is still fairly low level, albeit with some abstraction from current approaches. Crucially, however, the SCL style and notation permits *extensibility*. More abstract structures can be defined and efficiently implemented on the basis set out so far. In the concluding paper we show how this power can be exploited to define user-oriented structures to significantly simplify the task of parallel scientific programming.

Acknowledgements

The ideas presented in these two papers have been developed in collaborative projects within the Advanced Languages an Architectures Section and the Fujitsu Parallel Computing Research Centre at Imperial College. We are grateful to all our colleagues there but in particular to Peter Au, Moustafa Ghanem, Yi-ke Guo, Martin Kohler, Hing Wing To, Jin Yang and Wu Qian for their assistance. We thank Hing Wing To and Yi-ke Guo for their help in the preparation of these papers.
This work is supported by the UK Engineering and Physical Sciences Research Council and Fujitsu Laboratories who also support the Fujitsu Parallel Computing Research Centre. We are very grateful for their continued assistance.

References

1 J. Darlington, Y. Guo, H. W. To, and J. Yang. Functional skeletons for parallel coordination. In Seif Haridi, Khayri Ali, and Peter Magnussin, editors, *EURO-PAR'95 Parallel Processing*, pages 55–69. Springer-Verlag, August 1995.
2 David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.
3 High Performance Fortran Forum. *Draft High Performance Fortran Language Specification, version 1.0.* Available as technical report CRPC-TR92225, Rice University, January 1993.
4 Hiroaki Ishihata, Takeshi Horie, Satoshi Inano, Toshiyuki Shimizu, Sadayuki Kato, and Morio Ikesaka. Third generation message passing computer AP1000. In *International Symposium on Supercomputing*, pages 46–55, 1991.
5 Jingke Li and Marina Chen. Compiling communication efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–375, July 1991.

6 Message Passing Interface Forum. *Draft Document for a Standard Message-Passing Interface*. Available from Oak Ridge National Laboratory, November 1993.

7 Michael J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, second edition, 1994.