Functional Skeletons for Parallel Coordination

John Darlington Yi-ke Guo Hing Wing To Jin Yang

Department of Computing
Imperial College
180 Queen's Gate, London SW7 2BZ, U.K.
E-mail: {jd, yg, hwt, jy}@doc.ic.ac.uk

Abstract. In this paper we propose a methodology for structured parallel programming using functional skeletons to compose and coordinate concurrent activities written in a standard imperative language. Skeletons are higher order functional forms with built-in parallel behaviour. We show how such forms can be used uniformly to abstract all aspects of a parallel program's behaviour including data partitioning, placement and re-arrangement (communication) as well as computation. Skeletons are naturally data parallel and are capable of expressing computation and co-ordination at a higher level of abstraction than other process oriented co-ordination notations. Examples of the application of this methodology are given and an implementation technique outlined.

1 Introduction

This paper proposes the use of skeletons as a coordination language for programming parallel architectures. The coordination language model, as proposed by Gelernter and Carriero, builds parallel programs out of two separate components, the *computation model* and the *coordination model* [4]. Applications written in this way have a two-tier structure. The coordination level abstracts all the relevant aspects of a program's parallel behaviour, whilst the computation level expresses sequential computation through procedures written in an imperative base language. Such a separation allows the task of parallel programming to focus on the parallel coordination of sequential components. This is in contrast to the low level parallel extensions to languages where both tasks must be programmed simultaneously in an unstructured way.

Although developing coordination languages has become a significant research topic for parallel programming, there is still no general purpose coordination language designed to meet the requirements of constructing verifiable, portable and structured parallel programs. In this paper, we propose an approach for parallel coordination using functional skeletons to abstract all essential aspects of parallelism including data distribution, communication and commonly used parallel computation structure. Applying skeletons to coordinate sequential components, we have developed a structured parallel programming framework, SPP(X), where parallel programs are constructed in a structured way. In the SPP framework, an application is constructed in two layers: a higher skeleton

coordination level and a lower base language level. Parallel programs are constructed by using a skeleton based coordination language (SCL) to coordinate fragments of sequential code written in a base language (BL). The fundamental compositional property of functional skeletons naturally supports modularity of such programs. Using skeletons as the uniform means of coordination and composition removes the need to work with the lower level details of computation such as port connection. The uniform mechanism of high level abstraction of parallel behaviour means that all analysis and optimisation required can be confined to the coordination level which, being functional and constructed from pre-defined units, is much more amenable to such analysis and manipulation than the base language components or other coordination mechanisms.

This paper is organised into the following sections. In section 2, a skeleton coordination language, SCL, for is introduced and an example is presented to show its programming style and expressive power. In section 3, a concrete SPP programming language is proposed by taking Fortran as the base language for specifying sequential computation. Related work is overviewed in section 4. We finally summarise our work in section 5.

2 SCL: A Structured Coordination Language

We introduce a structured coordination language SCL as a general purpose coordination language by describing its three components: configuration and configuration skeletons, elementary skeletons and computational skeletons.

2.1 Configuration and Configuration Skeletons

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. In this paper distributed arrays are used as our underlying parallel data structure, though this idea can be generalised to richer and higher level data structures. Each distributed array, called a parallel array, has the type `ParArray index` α where each element is of type α and each index is of type `index`. In this paper we use << ... >> to represent a `ParArray`. To take advantage of locality when manipulating such distributed data structures one of the most important issues is to coordinate the relative distribution of one data structure to that of another, i.e. data alignment. The importance of abstracting this *configuration* information in parallel programming has been recognised in other languages such as High Performance Fortran (HPF), where a set of compiler directives are proposed to specify parallel configurations [5]. In SCL, we abstract control over both distribution and alignment through a set of *configuration skeletons*.

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 to divide the

initial structure into nested components and an **align** function to form a collection of tuples representing co-located objects. This model, illustrated in Fig.1, clearly follows and generalises the data distribution directives of HPF. Applying

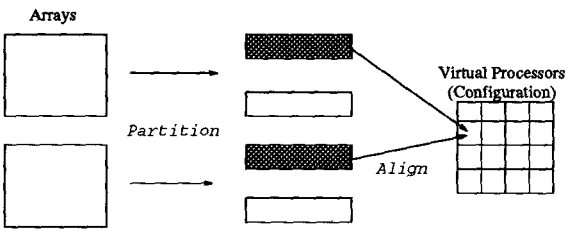

Fig. 1. Data Distribution Model.

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 o partition f A) (q o partition g B)
```

This skeleton 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**. **Partition** divides a sequential array into a parallel array of sequential subarrays:

```
partition :: Partition_pattern → SeqArray index α →
             ParArray index (SeqArray index α)
```

where **Partition_pattern** is a function of type ($\mathbf{index_S} \rightarrow \mathbf{index_P}$), where $\mathbf{index_S}$ is associated with the **SeqArray** and $\mathbf{index_P}$ addresses the **ParArray**. The type **SeqArray** is the ordinary sequential array type of our base language. Some commonly occurring partitioning functions are provided as built-in functions. For example, partitioning a `l` × `m` two-dimensional array using **row_block** we will get:

```
partition (row_block p) A = << ii := B | ii ← [1..p] >>
  where B = SeqArray (1:l/p, 1:n)
            [(i,j) := A (i+(ii-1)*l/p,j) | i←[1..l/p], j←[1..n]]
```

Other similar functions for two-dimensional arrays are **col_block**, **row_col_block**, **row_cyclic** and **col_cyclic**. The **align** operator:

```
align::ParArray index α → ParArray index β → ParArray index (α,β)
```

pairs corresponding subarrays in two distributed arrays together to form a new configuration which is an **ParArray** of tuples. Objects in each tuple of the configuration are regarded as being allocated on the same processor. A more general configuration skeleton can be defined as:

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

where `fl` is a list of distribution strategies for the corresponding data objects in the list `dl`.

Applying the `distribution` skeleton forms a configuration which is an array of tuples. Each element `i` of the configuration is a tuple of the form $(DA_1^i, \ldots, DA_n^i)$ where `n` is the number of arrays that have been distributed and DA_j^i represents the sub-array of the `j`th array allocated to the `i`th processor. As 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 $\langle DA_1, \ldots, DA_n \rangle$ where the DA_j stands for the distribution of the array 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`: $\langle DA_1, \ldots, DA_n \rangle$, a new configuration `C'`: $\langle DA'_1, \ldots, DA'_n \rangle$ 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 following skeleton `redistribution`:

```
redistribution [f1 ,..., fn] <DA1 ,..., DAn> = <f1 DA1 ,..., fn DAn>
```

SCL supports nested parallelism by allowing `ParArrays` as elements of a `ParArray` and by permitting a parallel operation to be applied to each of elements (`ParArrays`) in parallel. An element of a nested array corresponds to the concept of *group* in MPI [7]. The leaves of a nested array contain any valid sequential data structure of the base computing language. The following skeleton `gather` collects together a distributed array:

```
gather :: ParArray index (SeqArray index α) → SeqArray index α
```

Another pair of configuaration skeletons are `split` and `combine`:

```
split :: Partition_pattern → ParArray index α →
         ParArray index (ParArray index α)

combine :: ParArray index (ParArray index α) → ParArray index α
```

`split` divides a configuration into sub-configurations. `combine` is used to flatten a nested `ParArray`.

2.2 Elementary Skeletons: Parallel Arrays Operators

In the following we introduce functions, regarded as *elementary skeletons*, abstracting basic operations in the data parallel computation model.

The following familiar functions abstract essential data parallel computation patterns:

```
map :: (α → β) → ParArray index α → ParArray index β
map f << x0,...,xn >> = << f x0,...,f xn >>

imap :: (index → α → β) → ParArray index α → ParArray index β
imap f << x0,...,xn >> = << f 0 x0,...,f n xn >>

fold :: (α → α → α) → ParArray index α → α
fold (⊕) << x0,...,xn >> = x0 ⊕ ··· ⊕ xn
```

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

Data communications among parallel processors are expressed as the movement of elements in **ParArrays**. 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. Communication skeletons 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)] >>
```

For a $m \times n$ array, the following **rotate_row** operator express the data rotation of all rows:

```
rotate_row::(Int→Int)→ParArray (Int,Int) α→ParArray (Int,Int) α
rotate_row df A =
          <<(i,j) := A(i,(j+(df i)) mod n) | i←[1..m], j←[1..n]>>
```

where **df** is a function and **(df i)** indicates the distance of rotation for the **i**th row. An operator **rotate_col** for rotating columns can defined in the same way.

Broadcasting can be thought as a regular data-movement in which a data item is broadcast to all sites and aligned together with the local data. This skeleton is defined as:

```
brdcast :: α → ParArray index β → ParArray index (α,β)
brdcast a A = map (align_pair a) A
```

where **align_pair** groups a data item with the local data of a processor.

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). We model this 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. Obviously, the

`fetch` operation models only one to one communication. For the one dimensional definitions, two functions can be defined as:

```
send :: (Int → (SeqArray Int Int)) → ParArray Int α
          → ParArray Int (SeqArray Int α)
send f << x0,...,xn >> = << [xk | 0 in f k],...,[xk | n in f k] >>

fetch :: (Int → Int) → ParArray Int α → ParArray Int α
fetch f << x0,...,xn >> = << x(f 0),...,x(f n) >>
```

The above functions can be used to define more complex and powerful communication skeletons required for realistic problems.

2.3 Computational Skeletons: Abstracting Control Flow

A key to achieving proper coordination is to provide the programmer with the flexibility to organise multi-threaded control flow in a parallel environment. In SCL this flexibility 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. This structured approach of process coordination means that the behaviour of a parallel program is amenable to proper mathematical rigour and manipulation. Moreover, a fixed set of computational skeletons can be efficiently implemented across various architectures. In this subsection, we present a set of computational skeletons abstracting data parallel computation.

The `SPMD` skeleton, defined as follows, abstracts 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 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 synchronization 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 synchronization*.

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` defined as follows:

```
iterFor terminator iterSolve x
        = fst (iterUntil iterSolve' id con (x, 1))
          where
                iterSolve' (x, i) = (iterSolve i x, i+1)
                con (x, j) = j > terminator
```

2.4 Parallel Matrix Multiplication: A Case Study

To investigate the expressive power of SCL, in this subsection we define the coordination structure of two parallel matrix multiplication algorithms using SCL. The two following matrix multiplication algorithms are adapted from [9].

Row-Column-Oriented Parallel Matrix Multiplication: Consider the problem of multiplying matrices $A_{l \times m}$ and $B_{m \times n}$ and placing the result in $C_{l \times n}$ 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 2.

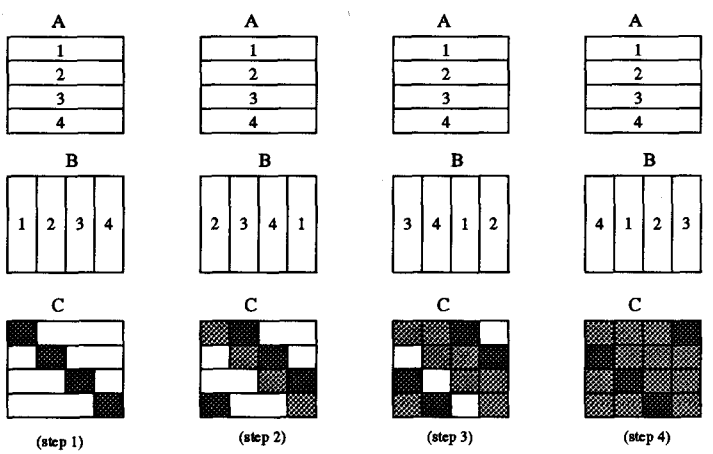

Fig. 2. Parallel matrix multiplication: row-column-oriented algorithm

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

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.

Block-Oriented Parallel Matrix Multiplication: This time we wish to multiply an `l` × `m` matrix `A` by an `m` × `n` matrix `B` on a `p` × `p` processor mesh with wraparound connections. Assume that `l`, `m` and `n` are integer multiples of `p` and `p` is an even power of 2. Initially both `A` and `B` are partitioned into mesh of blocks and each processor takes a `(l / p)` × `(m / p)` subsection of `A` and a `(m / p)` × `(n / p)` subsection of `B` (Fig.3(a)). The parallel algorithm staggers each block

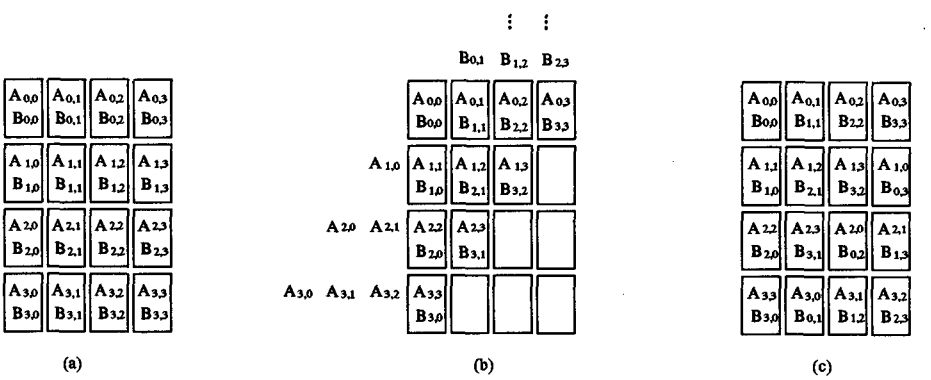

Fig. 3. Block-oriented algorithm: initial distribution

at row i of `A` to the left by i block column positions, and each block column i of

B upwards by i block row positions (Fig.3(b)) and the data is wrapped around (Fig.3(c)). The overall algorithm structure is also an SPMD computation iterated p times. At each step the local phase of the SPMD computation multiplies the pair of blocks located locally using a sequential matrix multiplication program and then the global phase moves the data: each processor passes its portion of A to its left neighbour and passes its portion of B to its north neighbour. The SCL code for this algorithm is shown below:

```
matrixMul :: Int → SeqArray index Float →
             SeqArray index Float → SeqArray index Float
matrixMul p A B = gather DC
    where
        C = SeqArray ((1,SIZE(A,1)), (1, SIZE(B,2))
             [ (i,j) := 0 | i ← [1..SIZE(A,1)], j ← [1..SIZE(B,2)] ]
        <DA, DB, DC> = iterFor p step dist
        fl = [((row_col_block p p), (rotate_row df1)),
             ((row_col_block p p), (rotate_col df1)),
             ((row_col_block p p), id)]
        dl = [A, B, C]
        dist = distribution fl dl
        df1 i = i (* to indicate the distance of rotation *)

step i <DA, DB, DC> =
    SPMD [(gf, SEQ_MM 0)] <DA, DB, DC>
    where
        newDist = [(rotate_row df2), (rotate_col df2), id]
        gf X = redistribution newDist <DA, DB, X>
        df2 i = 1 (* to indicate the distance of rotation *)
```

Abstraction. The above examples highlight an important feature of SCL. The parallel structure of a class of parallel algorithms for matrix multiplication can be abstracted and defined by the following SCL program:

```
Generic_matrixMul p distribustrategy redistribustrategy A B = gather DC
    where
        C = SeqArray ((1,SIZE(A,1)), (1, SIZE(B,2))
            [ (i,j) := 0 | i ← [1..SIZE(A,1)], j ← [1..SIZE(B,2)] ]
        dist = distribution distribustrategy [A, B, C]
        <DA, DB, DC> = iterFor p step dist

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

Thus, the row-column-oriented and the block-oriented parallel matrix multiplication program become instances of the generic parallel matrix multiplication code by instantiating the corresponding distribution and redistribution strategies. That is, the SCL code for generic parallel matrix multiplication defines an *algorithmic skeleton* for parallel matrix multiplication. This example shows how

an application oriented parallel computation structure could be systematically defined.

3 Fortran-S: Coordinating Fortran Programs with SCL

As an exercise in developing a concrete SPP language we are designing a language, Fortran-S, to act as a powerful front end for Fortran based parallel programming. Conceptually, the language is designed by instantiating the base language in the SPP scheme with Fortran. Thus, to write a parallel program in Fortran-S, SCL is used as a coordination language to define the parallel structure of the program. Local sequential computation for each processor is then programmed in Fortran.

The matrix multiplication examples (section 2) can be coded in Fortran-S by instantiating the sequential local procedure **SEQ_MM** with the following Fortran subroutine for matrix multiplication (Fortran 90 syntax is adopted):

```
SUBROUTINE SEQ_MM (IT, IDX, X, Y, Z)
  INTEGER, INTENT (IN) :: IT, IDX
  REAL, DIMENSION (:,:), INTENT (IN) :: X
  REAL, DIMENSION (:,:), INTENT (IN) :: Y
  REAL, DIMENSION (:,:), INTENT (INOUT) :: Z
  INTEGER :: I, J

  START = ((IT + IDX) * SIZE(Y,2)) MOD SIZE(Z,2)

  DO I = 1, SIZE(X,1)
    DO J = 1, SIZE(Y,2)
      DO K = 1, SIZE(Y,1)
        Z (I,J+START) = Z (I,J+START) + X (I,K) * Y (K,J)
      END DO
    END DO
  END DO
END SUBROUTINE SEQ_MM
```

The *argument intent* of the parameters identifies the intended use of the variables. Variables specified with **INTENT(IN)** must not be redefined by the procedure, whilst **INTENT(INOUT)** variables are expected to be redefined by the procedure, and variables specified with **INTENT(OUT)** pass information out of the procedure.

In Fortran-S, the basic data type for SCL programming is the **ParArray** which is regarded as the parallel data structure whilst the basic data types, including arrays of Fortran are the sequential data structures. Thus, Fortran subroutines handle only Fortran data objects.

Fortran-S can be implemented by transforming Fortran-S programs into conventional parallel Fortran programs, that is sequential Fortran augmented with message passing libraries. Due to the functional nature of SCL source level transformation can be applied to optimise the parallel behaviour of the program, including granularity adjustment, nested parallelism flattening, optimised data distribution and interprocessor communication [2].

Currently, we are building a prototype system based on Fortran 77 plus MPI [7] targeted at a Fujitsu AP1000 machine [6]. The matrix multiplication example has been translated to Fortran77 plus MPI on an AP1000. Due to the richness of information provided by the Fortran-S code, the performance data is very encouraging, as shown in Fig. 4 for an array size of `400 × 400`.

nprocs	time(sec)	speedup
1	521.41	1.0
4	133.22	3.9
8	66.11	7.9
10	52.73	9.9
16	32.88	15.9
20	26.55	19.6
25	21.46	24.3
50	12.31	42.3
80	8.95	58.2
100	7.44	70.1

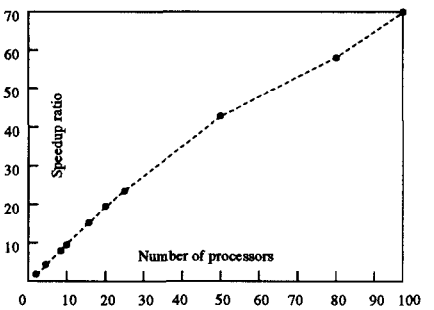

Fig. 4. Parallel matrix multiplication: speedup

4 Related work

Comparison with HPF. HPF supports data parallel programming by adding extensions to Fortran 90 including compiler directives for data distribution. Our work has been motivated by HPF. For example, there is a direct correspondence between the distribution specified by the HPF directives and the `distribution` skeleton. Configuration skeletons can be regarded as functional abstractions of HPF directives. Since SCL configuration skeletons are freely composable they are much more flexible than the fixed HPF directives. Moreover, the SCL operators are extendable and provide a more powerful means to express data distribution/re-distribution, alignment and movement.

Other coordination languages. One of the best known coordination languages is Linda, by Gelernter and Carriero [4]. As a coordination language, Linda abstracts MIMD parallel computation as an asynchronously executing group of processes that interact by means of an associative shared memory. Our work differs by extending the coordination language to describe all aspects of parallel coordination including partitioning and scheduling of parallel activities.

The coordination language PCN [3], promoted the concept of composing together modules by connecting together explicitly-declared communication ports. An interesting development of the PCN approach is the P^{3}L system [8]. Rather than using a set of primitive composition operators, a set of *parallel constructs* are used as program composition forms. Each parallel construct in P^{3}L abstracts a specific form of commonly used parallelism. This approach is based on the integration of the skeleton approach [1] and the PCN model. Such an integration,

however, is not smooth since the high level abstraction of parallel computation structure is compromised by the lower level process model.

5 Conclusion

In this paper we have proposed functional skeletons as a new mechanism for developing general purpose parallel coordination systems. The work stems from our original work on functional skeletons to capture re-occurring patterns of parallel computation. This has been extended so that control of all aspects of parallel computation can be now expressed using skeletons. Therefore, it provides an ideal means for coordinating parallel computation. In this paper we have presented a coordination language, SCL, and a parallel programming scheme, SPP(X), obtained by applying SCL to coordinate computation programmed in a base language, X.

This work present a significant synthesis of some major developments of designing parallel programming systems including the coordination approach, data parallel programming, skeleton-based higher lever construction of parallel applications and declarative parallel programming. It provides a promising solution to the engineering problems of developing a practical structured programming paradigm for constructing verifiable, reusable and portable parallel programs.

Acknowledgements

The second author is supported by the ESPRC funded project GR/H77545 and the fourth author is supported by a British Council grant.

References

1. J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In *Parallel Architectures And Languages, Europe: PARLE 93*. Springer-Verlag, 1993.
2. J. Darlington, Y. Guo, and H. W. To. Structured parallel programming: Theory meets practice. Technical report, Imperial College, 1995. unpublished.
3. Ian Foster, Robert Olson, and Steven Tuecke. Productive parallel programming: The PCN approach. *Scientific Programming*, 1(1), 1992.
4. David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.
5. High Performance Fortran Forum. *Draft High Performance Fortran Language Specification, version 1.0.* Available as technical report CRPC-TR92225, Rice University, January 1993.
6. 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.
7. Message Passing Interface Forum. *Draft Document for a Standard Message-Passing Interface.* Available from Oak Ridge National Laboratory, November 1993.
8. S. Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs.* PhD thesis, Università Delgi Studi Di Pisa, 1993.
9. Michael J. Quinn. *Parallel Computing: Theory and Practice.* McGraw-Hill, second edition, 1994.