

Parallel Skeletons for Structured Composition

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 straightforward solution to the problems of compositional parallel programming by using skeletons as the uniform mechanism for structured composition. In our approach parallel programs are constructed by composing procedures in a conventional base language using a set of high-level, pre-defined, functional, parallel computational forms known as *skeletons*. The ability to compose skeletons provides us with the essential tools for building further and more complex application-oriented skeletons specifying important aspects of parallel computation. Compared with the process network based composition approach, such as PCN, the skeleton approach abstracts away the fine details of connecting communication ports to the higher level mechanism of making data distributions conform, thus avoiding the complexity of using lower level ports as the means of interaction. Thus, the framework provides a natural integration of the compositional programming approach with the data parallel programming paradigm.

1 Introduction

It has been recognised that effective parallel program development requires a more structured approach that separates the concerns of parallel computation from sequential computation. Such a separation allows the task of parallel programming to focus on the parallel coordination of sequential components [9]. This *coordination based parallel programming* approach is in contrast to the low level parallel extensions to sequential languages where both parallel and sequential computation tasks have to be programmed simultaneously in a unstructured way. Rather, with a powerful parallel coordination mechanism, parallel programs can be constructed systematically with the following compositional properties:

tional properties:

- **Reusability of sequential code:** Parallel programs can be developed by composing existing modules written in conventional languages.
- **Reusability of parallel code:** Complex parallel programs can be written by composing parallel modules.
- **Portability:** Parallel programs can be efficiently implemented on a wide range of parallel machines by specialised implementations of the compositional operators on target architectures.

Developing parallel coordination mechanisms with natural support for structured program composition has been one of the most important research areas in parallel programming. The most well known example is perhaps the Program Composition Notation (PCN) [8]. In PCN, composition is achieved through the plugging together of explicitly-declared communication ports. Thus the composition of two program modules may require the connection of several communication ports. A core set of three primitive composition operators: *parallel*, *sequential* and *choice composition* is used in PCN to specify the control flow of processes. More sophisticated combining forms, such as divide-and-conquer, can be defined and implemented using this notation and preserved as reusable templates. In PCN, logical variables are used as channel variables for communication synchronisation based on their single assignment feature. Based on the PCN, a class of parallel languages has been proposed such as Fortran-M [7], where Fortran is taken as the *base language* for sequential programming. Unfortunately, this clean methodology is still quite low level. Thus, high level compositions have to be programmed with lower level primitives concerned with process activations, interprocess communications and process mapping. This problem has been addressed by the recent development of the P³L language [4]. Rather than using a set of primitive composition operators, P³L uses a set of *parallel constructs* to program composition forms. Each parallel construct in P³L abstracts

a specific form of commonly used parallelism. For example, the `map` construct is used to compose programs to form data parallel computation. This approach is based on the integration of the skeleton approach [3, 5] and the PCN model. The computational structure of commonly used classes of algorithms are abstracted as skeletons and parallel programming becomes instantiating these skeletons. Such an integration, however, is not smooth since the high level abstraction of parallel computation is complicated by the lower level process model resulting in a system which presents difficulties for both programming and reasoning.

In this paper, we propose an alternative solution to composition which attempts to integrate both data and task parallelism and maintain portability. We propose the use of functional skeletons as a uniform approach to abstract all essential aspects of parallelism. Applying skeletons to coordinate sequential components provides a structured way of composition. Using skeletons as the means of composition removes the need to work with the fine details of port connection. Not only is this natural and high level, but it also enables reasoning about optimisations over the parallel aspects of a program through formal transformations.

This paper is organised into the following sections. In section 2, a structured composition language SCL is introduced by presenting three groups of SCL skeletons. Compositional parallel programming examples are presented in section 3 to illustrate the programming style and expressive power of the language. A transformation-based optimisation approach for parallel structures defined in SCL is outlined in section 4. Section 5 presents some performance figures from a hand worked implementation of one of the examples. We compare our approach with related work and summarise our work in section 6.

2 SCL: A Coordination Language for Structured Composition

We propose a structured coordination language (SCL) where parallel programs are constructed by composing procedures in a conventional base language using a set of high-level, pre-defined, functional, parallel computational forms known as *skeletons*. Application written in this way have a two-tier structure. The upper layer SCL language abstracts all the relevant aspects of a program’s parallel behaviour, including partitioning, data placement, data movement and control flow, whilst the lower level expresses sequential computation through procedures written in any sequential base language (BL). SCL skeletons can be freely composed and eventually instantiated with the base language components, but the base language cannot call any of the SCL primitives. This structure abstracts all parallel control

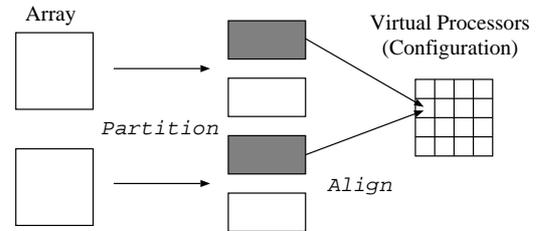


Figure 1: Data distribution model.

to the upper SCL level and facilitates optimisation by transformation rules applied safely to the SCL.

In this section, the SCL language is introduced by describing its three components: configuration skeletons, elementary skeletons and computational skeletons.

2.1 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, but 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 specify the relative distribution of one data structure to 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 used to specify parallel configurations [10]. 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 distributed 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 which 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 Fig. 1, clearly follows and generalises the data distribution directives of HPF. 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)

```

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** which are bulk data-movement functions specifying any initial data rearrangement that may be required. The **distribution** skeleton is defined by composing the functions **align** and **partition**. The skeleton **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 (**index_s** → **index_p**), where **index_s** is associated with the **SeqArray** and **index_p** addresses the **ParArray**. The type **SeqArray** is the ordinary sequential array type of our base language. Some commonly occurring partition strategy functions are provided as built-in functions. For example, partitioning a 1 × *m* two-dimensional array using **row_block** gives:

```

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

```

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 a tuple of the configuration are regarded as being allocated to the same processor. A more general configuration skeleton can be defined over lists of distribution strategies and arrays:

```

distribution [(f,p)] [d]
= p ◦ partition f d
distribution ((f,p):fl) (d:dl)
= align (p ◦ partition f d)
      (distribution fl 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, \dots, 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 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 $\langle DA_1, \dots, DA_n \rangle$ 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 is uniformly defined by applying bulk data movement operators to configurations. Given a configuration **C**: $\langle DA_1, \dots, DA_n \rangle$, a new configuration **C'**: $\langle DA'_1, \dots, DA'_n \rangle$ can be formed by applying f_j to the distributed structure DA_j where f_j is some bulk data movement operator 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 the elements of the **ParArrays** in parallel. An element of a nested array corresponds to the concept of a *group* in MPI [13]. The leaves of a nested array may contain any valid sequential data structure of the base programming language.

The following skeleton **gather** collects together a distributed array:

```

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

```

The functions **split** and **combine** provide another pair of configuration skeletons:

```

split :: Partition_pattern →
        ParArray index α →

        ParArray index (ParArray index α)

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

```

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

2.2 Elementary Skeletons

Next, we introduce several functions, known as *elementary skeletons* which abstract the basic operations of 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

scan :: (α → α → α) → ParArray index α
      → ParArray index α
scan (⊕) << x0, x1, ..., xn >>
  = << x0, x0 ⊕ x1, ..., 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` and the partial reduction operator `scan` abstract tree-based parallel reduction computation over arrays. The argument (\oplus) must be associative in both cases otherwise the result is undefined.

Data communication among parallel processors are expressed as the movement of elements in `ParArrays`. In SCL, a set of bulk data movement functions (called communication skeletons) are introduced as the data parallel counterpart of sequential loops which rearrange array elements. 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 an $m \times n$ array, the following `rotate_row` and `rotate_col` operators express the data rotation of all rows or columns, in which `df` is a function and `(df i)` indicates the distance of rotation for the *i*th row or column to be rotated.

```

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] >>

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

```

Broadcasting can be thought as a regular data-movement in which a data item is broadcast to all sites and is 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.

A variant of the `brdcast` operator is the function `applybrdcast`:

```

applybrdcast f i A = brdcast (f A(i)) A

```

this skeleton applies the function `f` locally to the data on the *i*th element and broadcasts the result.

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, or one to many communication. For one dimensional arrays, the two functions are 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  $\alpha$   $\rightarrow$ 
ParArray Int  $\alpha$ 
fetch f <<  $x_0, \dots, x_n$  >>
= << [ $x_k$  | f 0 = k], ..., [ $x_k$  | f n = k] >>

```

The above functions can be used to define more complex and powerful communication skeletons required by practical 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 `farm` skeleton, defined by the following functional specification, captures the simplest form of data parallelism.

```

farm :: ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow$   $\alpha \rightarrow$  ParArray index  $\beta$ 
       $\rightarrow$  ParArray index  $\gamma$ 
farm f env = map ( f env )

```

A function is applied to each object of a `ParArray`. The function also takes an environment which represents data which is common to all processes. Parallelism is achieved by utilising multiple processors to evaluate the jobs (i.e. “farming them out” to multiple processors). The `farm` skeleton can be generally defined based on the map operator of any underlying parallel data structure.

The `SPMD` skeleton, defined as follows, abstracts the features of SPMD (Single Program Multiple Data) computation:

```

SPMD [] = id
SPMD (gf, lf) : fs
= (SPMD fs)  $\circ$  (gf  $\circ$  (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, are sequential

program fragments in a base language. The *global operations* over the whole configuration are parallel operation 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*.

The `iterateUntil` 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 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 iSolve id con (x, 1))
  where
    iSolve (x, i) = (iterSolve i x, i+1)
    con (x, j) = j > terminator

```

3 Structured Composition in SCL

We illustrate the compositional features of SCL by programming two examples. The first is a parallel solver to solve a system of linear equations:

$$Ax = b$$

We use the Gauss-Jordan method with partial pivoting which is a simple variant of Gaussian elimination transforming the system of equations into an upper triangular form. The algorithm iteratively scans all the columns of the matrix. In a single iteration `i`, rows `i` through to `n` are searched for the row whose `i`'th column has the largest absolute value. The chosen row is then swapped with row `i` and used as the pivot row to annihilate elements. The algorithm can be parallelised by distributing the columns of the matrix and updating, in each iteration step, all the columns in parallel. The parallel structure of the program is expressed in SCL below. The matrix is partitioned column-wise simply by applying the partition operator. The main iteration is specified again by the computational skeleton `iterFor`.

```

gauss A p
= iterFor p elimPivot DA
  where

```

```

DA = partition [ column_block p ] [A]
elimPivot i x = map (UPDATE i)
              (applybrdcast PARTIAL-PIVOT i x)

```

where `PARTIAL-PIVOT` is the sequential procedure for selecting the pivot and `UPDATE` is the sequential procedure for updating based on the pivot. Parallel updating is specified by the computational skeleton `map` applying `UPDATE` to annihilate elements of all the columns in parallel.

The compositional power of SCL can be further illustrated by the following example of a parallel sorting algorithm suitable for implementation on hypercube multi-processor. The algorithm, known as hyperquicksort [15], has a non-trivial parallel control structure including data communication, barrier synchronisation and dynamic processor grouping (for nested parallelism). The following SCL program is a concise specification of these parallel behaviours, where `A` is the list of values to be sorted in a `d`-dimension hypercube multi-processor.

```

hypersort A d
  = let DA = map SEQ_QUICKSORT
        (partition (block 2^d) A)
        in gather (hsort d DA)

hsort 0 DA = DA
hsort d DA
  = let SubCubes
        = mergeAndDiv
          (exPart d (spreadPivot DA))
        in
        combine (map (hsort (d-1)) SubCubes)

spreadPivot
  = applybrdcast MIDVALUE 0

exPart d
  = SPMD [(fetchPartner, SPLIT d)]
  where
    fetchPartner <localData, toBeSent>
      = < localData recvData >
    recvData = fetch myPart toBeSent
    myPart i = xor(i, 2^(d-1))

mergeAndDiv
  = SPMD [(split (block 2), MERGE)]

```

The program is illustrated in Fig. 2 where the 32 values to be sorted are initially located on processor 0 (see Fig. 2 (a)). The first step distributes the list to be sorted evenly across 4 processors. After that the sequential quicksort code `SEQ_QUICKSORT`¹ is performed

¹All sequential code is omitted. This can be written, for example, in Fortran or C.

in parallel on each processor. The remaining computation is an iteration which merges locally sorted data of corresponding processors in subcubes. Each step of iteration is the composition of the following tasks:

1. pivot: broadcast the median value of the sequential array on node 0. The median value is computed by the sequential code `MIDVALUE` in node 0.
2. split: every processor uses the sequential code `SPLIT` which takes the broadcast median as a pivot value to split their local data into two portions.
3. exchange partner: For each processor p_i in an i -dimensional cube, the processors in the lower half of the hypercube exchange the upper portion of its sorted vector with the lower portion of the sorted vector in its partner in the upper half of the hypercube (see Figure 2 (d) and (f))
4. merge: each processor uses the sequential code `MERGE` to merge the portion it receives with the portion it has kept. Thus, at each step of iteration, an i hypercube is split into two sub-cubes where all value less than or equal to the pivot are in lower $(i - 1)$ -dimensional sub-hypercube, and all values greater than the pivot are in the upper $(i - 1)$ -dimensional sub-hypercube (see Figure 2 (e) and (g))

After 4 iterations, 32 values are sorted and are collected to processor 0.

The above examples show how parallel programs can be systematically constructed by composing sequential and parallel program fragments with respect to a user-defined parallel computation structure. Therefore, SCL provides a structured mechanism for composing programs based on the data parallel computation model.

4 Transformations for Optimisation

One of the advantages of the functional abstraction mechanism of SCL is that meaning preserving transformation techniques can be generally applied to optimise the parallelism specified uniformly in terms of skeletons. Thus, with such a high level functional specification of parallel behaviour, compile time optimisation can be systematically realised based on a class of transformation rules. We overview some basic transformation rules in this section.

Map Fusion: A simple but important transformation which reduces parallel overhead is the following *map fusion* law :

$$\text{map } f \circ \text{map } g = \text{map } (f \circ g)$$

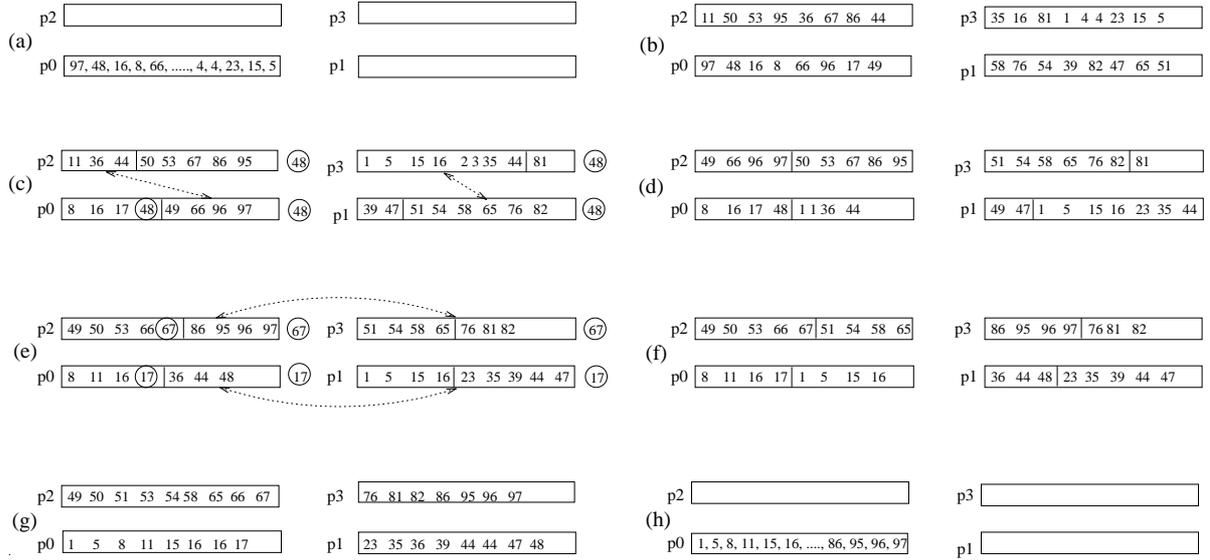


Figure 2: Illustration of the hyperquicksort algorithm on a 2-dim hypercube

To maintain the synchronous semantics of the system some form of barrier synchronisation must be performed between the two maps of the left-hand-side. This transformation reduces the need to perform a barrier synchronisation and provides for better load balancing. Since the map function abstracts parallel loops (e.g. the `forall` construct in HPPF), this transformation law is a functional abstraction of the *loop fusion* [11] technology of conventional compiler optimisation.

Map Distribution: The following transformation, called *map distribution*, is intended to increase the level of parallelism in a program:

$$\text{foldr1 } (f \circ g) = \text{fold } f \circ \text{map } g$$

Assume that f is associative and that g is applied to an element before f combines the accumulated result and the element. Clearly, the left-hand-side is not parallel as the combined function $(f \circ g)$ is not associative. However, by splitting the `foldr` into a `fold` and `map` the program becomes parallel. This transformation performs a very similar role to the well known *loop distribution* technique presented in [11].

Communication Algebra: As all data-movement patterns are described using functional operators, it is possible to develop a set of rules to optimise communication. For example the rules:

$$\begin{aligned} \text{send } f \circ \text{send } g &= \text{send } (f \circ g) \\ \text{fetch } f \circ \text{fetch } g &= \text{fetch } (g \circ f) \end{aligned}$$

enable communication steps to be removed by combining two communication steps into one. By taking these

transformation rules as algebraic laws, a powerful *communication algebra* is developed [6].

Flattening: Transformation can be applied to flatten nested data parallelism. Let

$$\text{sgf} = \text{gf}_0 \circ \text{map } \text{gf}_1 \circ (\text{split } P)$$

then the rule for flattening is:

$$\text{SPMD } [(\text{gf}_0, \text{SPMD } [(\text{gf}_1, \text{lf}_1)])] \circ (\text{split } P) = \text{SPMD } [(\text{sgf}, \text{lf})]$$

With this rule, nested SPMD computation can be transformed into a flat data parallel computation with a *segmented global function sgf*. Thus, the `sgf` provides the a similar functionality to the *Segmented Instructions* [1] used in the NESL language implementation. More transformation rules can be found in [6].

5 Experimental Results

To assess the performance of code resulting from this approach some initial experiments have been performed. The experiments were conducted on a Fujitsu AP1000 [12] using Fortran and MPI as the target code for our hand compilation. MPI was chosen to ensure future portability at the language level. The first step was the preliminary implementation of several elementary skeletons in a problem independent manner.

Hyperquicksort, as described in section 3, was chosen as a case study. As a SCL compiler is still under development, the compilation route was followed by hand. Firstly the recursive divide and conquer program was flattened into a linear iterative program by transformation. The resulting program is shown below:

no. procs.	runtime (secs)
1	30.55
2	15.16
4	8.49
8	4.43
16	2.88
32	2.17

Table 1: Performance of hyperquicksort.

```

hypersort A d
= let
  DA = map SEQ_QUICKSORT
      (partition (block 2^d) A)
  in iterfor d step DA
  where
  step i x = map MERGE
            (exPart d' (wpivot d' x))
  d' = d-i+1

wpivot d x
= align x pivots
  where
  pivots = SPMD
          [(fetch (mf d), MIDVALUE d)]
  mf d i = [ i / 2^d ] × 2^d

exPart d
= SPMD [(getpartner, SPLIT d)]
  where
  getpartner <localData, tobeSent>
    = align localData partnerData
  partnerData = fetch mypartner tobeSent
  mypartner i = xor(i, 2^(d-1))

```

The flattened SPMD program was then hand translated into intermediate code. The remaining tasks were to instantiate calls to the elementary skeleton templates with the sequential code and to link these skeletons together.

The resulting code was tested on an AP1000 using a vector of 131072 random numbers. Table 5 shows the total execution time in seconds as the number of processors is increased. A graph of the speedup is shown in Fig. 3. Note that linear speedup is not possible with this problem. Our achieved performance compares well with the best speedup available for this problem.

These encouraging performance results give us confidence that SCL skeletons can be efficiently implemented as libraries or macros defined over base languages and standard communication libraries. A prototype SCL compiler is currently under development.

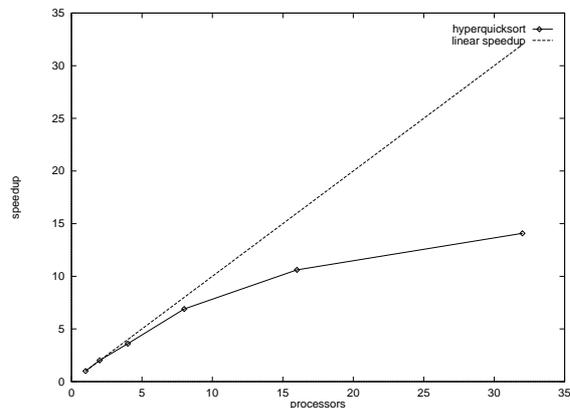


Figure 3: Speedup of sorting 131072 integers on AP1000

6 Related Work and Conclusion

In this paper we have proposed a structured composition framework based on the idea of using skeletons as a uniform mechanism for composition. The work is a natural integration of two major families of parallel programming: compositional languages and data parallel programming and skeleton based parallel programming. It stems from our original work on functional skeletons to capture re-occurring patterns of parallel computation and is based on systematical synthesis of a class of research activities on parallel programming including:

Pure functional languages: The side-effect free property of functional languages makes possible to specify the meaning of skeletons in a purely declarative way. SCL adopts the functional formalism to define the meaning of skeletons. Functional composition provides a natural model for synchronous composition within the SPMD computation paradigm. However, SCL is not a general functional programming system. By using only pre-defined skeletons to co-ordinating and compose sequential computation threads, the SCL has full control over granularity. This overcomes many of the efficiency problems of implicit parallel functional programming. Furthermore we abandon the use of a general functional computational model for sequential computation. Thus, we do not suffer from the inefficiency of general functional programming.

Data parallel languages: Data parallel languages exploit parallelism from basic aggregate data types [17]. Several examples of these for languages are the HPF standard [10], C* [16] and NESL [2]. We adopt data parallelism as our unit of composition. The main advantage our approach comes from the departure from the flat parallelism approach where parallel data structures cannot be nested. However, unlike NESL which

provides a similar level of parallelism, facilities are also provided for describing data distribution.

Compositional languages: As we discussed in the introduction, the main problem with process oriented compositional languages is the complexity of using ports as the means of interaction. This complexity increases when expressing programs containing multiple phases of data parallelism. This is a direct result of the loss of the knowledge that the collection of processes is a data structure. With SCL, we identify the idea of parallel configurations and parallel data structures. Thus, process interaction can be abstracted as collective communication operators between structures. Functional composition provides a uniform means of constructing configurations which provides a direct support for program composition within the data parallel paradigm. Moreover, parallel composition of concurrent tasks can be supported by applying a concurrent constraint programming model on the top of SCL layer. Thus, task parallelism is supported when it is needed.

Algorithmic Skeletons: Cole [3] first proposed a solution to the contradiction of elegant high level parallel programming and efficient low level parallel programming based on *algorithmic skeletons*. Our early work extended the concept of skeletons and exploited the clean relationship between skeletons and higher-order functions in a functional language [5]. The primary problem of the skeleton proposal was the lack of facilities for composing skeletons to define parallel structures. The major contribution of SCL to the skeleton based parallel programming is that we developed SCL as a functional abstraction language to define both skeletons and their composition forms. Thus, skeletons can be defined and composed to abstract various parallel computation structures.

The work by the Pisa group has also noticed the problems caused by the inability to compose skeletons [14]. However, the main focus of P³L is to connect together skeletons whose interfaces are single streams. The SCL approach attempts to capture a broader class of parallel algorithm. As such our approach tackles problems of data distribution as well as control distribution.

Future Work: As an exercise in developing a SCL 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 using Fortran as the base language. Thus, to write a parallel program in Fortran-S, we use SCL, which is the higher level of the Fortran-S language, to define the parallel structure of the program. Local sequential computation for each processor is then programmed in For-

tran. A prototype system based is under implementation targeted at a Fujitsu AP1000 machine [12].

Acknowledgements

We would like to thank our colleagues in the Advanced Languages and Architectures Section at Imperial College for their assistance and ideas. Special thanks are due to Moustafa Ghanem for his helpful comments on earlier drafts. The second author is supported by the ESPRC funded project GR/H77545 "Definitional Constraint Programming: a Foundation for Logically Correct Concurrent Systems".

References

- [1] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.
- [2] G. E. Blelloch. NESL: A nested data-parallel language (version 3.0). Technical Report CMU-CS-94 (Draft), Carnegie Mellon University, March 1994.
- [3] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman/MIT Press, 1989.
- [4] M. Danelutto, R. Di Meglio, S. Orlando, S. Palagatti, and M. Vanneschi. The P³L language: An introduction. Technical Report HPL-PSC-91-29, Hewlett-Packard Laboratories, Pisa Science Centre, December 1991.
- [5] J. Darlington, M. Ghanem, and H. W. To. Structured parallel programming. In *Programming Models for Massively Parallel Computers*. IEEE Computer Society Press, September 1993.
- [6] J. Darlington, Y. Guo, and H. W. To. Structured parallel programming: Theory meets practice. Technical report, Imperial College, 1995. unpublished.
- [7] I. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 25(1), 1995.
- [8] I. Foster, R. Olson, and S. Tuecke. Productive parallel programming: The PCN approach. *Scientific Programming*, 1(1), 1992.
- [9] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97-107, February 1992.
- [10] High Performance Fortran Forum. *Draft High Performance Fortran Language Specification, version 1.0*. Available as technical report CRPC-TR92225, Rice University, January 1993.

- [11] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.
- [12] H. Ishihata, T. Horie, S. Inano, T. Shimizu, S. Kato, and M. Ikesaka. Third generation message passing computer AP1000. In *International Symposium on Supercomputing*, pages 46–55, 1991.
- [13] Message Passing Interface Forum. *Draft Document for a Standard Message-Passing Interface*. Available from Oak Ridge National Laboratory, November 1993.
- [14] S. Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs*. PhD thesis, Università Delgi Studi Di Pisa, 1993.
- [15] M. J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, second edition, 1994.
- [16] J. R. Rose and G. L. Steele Jr. C*: An extended C language for data parallel programming. Technical Report PL 87-5, Thinking Machine Corporation, 1987.
- [17] J. M. Sipelstein and G. E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.