Coordinating Data Parallel SAC Programs with S-Net

Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko

University of Hertfordshire
{c.grelck, s.scholz, a.shafarenko}@herts.ac.uk

Abstract

We propose a two-layered approach for exploiting different forms of concurrency in complex systems: We specify computational components in our functional array language SAC*, which exploits data parallel properties of array processing code. The declarative stream processing language* S-NET *is used to orchestrate the collaborative behaviour of these components in a streaming network. We illustrate our approach by a hybrid implementation of a sudoku puzzle solver as a representative for more complex search problems.*

1 Introduction

Data parallel languages like SISAL [5], ZPL [3], or SAC [13, 10] are known to be particularly well suited for utilising implicit parallelism effectively. For certain applications, predominantly numerical applications on large homogeneous data structures, these languages can deliver parallel performance competitive if not superior to that of hand-coded FORTRAN programs [2, 4, 7, 9]. The beauty of this approach is that it is completely implicit and thus avoids all the usual pitfalls of concurrent programming such as deadlocks or race conditions.

However, even for numerical applications, other forms of concurrency are often required, which are more directly under the the programmer's control. The main challenge here is to achieve a level of explicit concurrency which is expressive enough to describe the concurrency needed yet sufficiently restrictive to guarantee an orderly system behaviour.

Streams in SISAL were to achieve exactly that. Unfortunately, the tight integration of the streams into the language gave rise to several drawbacks. The topology of streams is difficult to identify in a given program, which makes reasoning about its concurrent behaviour rather difficult. For the same reason, debugging requires a holistic approach. Besides these issues, there is an expressiveness concern: in order to stay in the side-effect free world of single assignments, impure features, which may be essential for managing concurrency, e.g. non-deterministic merge, cannot be specified.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

In this paper, we propose a two-layered approach for integrating stream processing (as a means for making concurrency explicit) with the fully implicit, data-parallel programming of SAC. In the outer layer, we introduce a novel declarative stream processing language, named S-NET [14, 8], which explicitly coordinates the asynchronous collaborative behaviour of SAC-encoded components. S-NET turns a SAC function into a stream processing component; then a set of network combinators allows us to connect various components into complex streaming networks.

This approach has several advantages: the strict separation between the two layers makes the explicit concurrency control imposed by the coordination layer easily identifiable. Although the individual SAC components are purely functional, the S-NET layer supports impure features such as non-deterministic merge. Debugging the concurrent behaviour becomes rather straightforward as all streams can be observed individually. Despite its impure features an orderly behaviour of the streaming network can be assured.

We illustrate the potential of our approach by a simple search problem: finding solutions to sudoku puzzles. While sudokus are simple enough to be explored in detail, they are computationally non-trivial as they require search over an imbalanced tree of theoretically up to 9^{81} possibilities.

The remainder of this paper is organised as follows. Section 2 introduces SAC; we sketch out a SAC-only implementation of the sudoku search problem in Section 3. In Section 4 we explain the design of S-NET and in Section 5 we illustrate our approach by discussing hybrid SAC/S-NET sudoku puzzle solvers. We discuss related work in Section 6 and finally present our conclusions.

2 Introducing SAC

Core SaC is a functional, side-effect free variant of C: we interpret assignment sequences as nested let-expressions, branches as conditional expressions and loops as syntactic sugar for tail-end recursive functions. The meaning of functional SaC code coincides with the state-based semantics of literally identical C code (cf. [13]). This language kernel is extended by n-dimensional state-less arrays: Any expression may evaluate to an array, and arrays may be passed between functions without restrictions. Arrays in SaC are neither explicitly allocated nor de-allocated. They exist as long as the associated data is needed, just like scalars in conventional languages.

Array types include arrays of fixed shape, e.g. `int[3,7]`, arrays of fixed rank, e.g. `int[.,.]` and arrays of any rank, e.g. `int[*]`. The latter include scalars which in SaC are considered rank-0 arrays with an empty shape vector. For convenience and equivalence with C we use `int` rather than the equivalent `int[]` as a type notation for scalars. SaC provides a small set of built-in array operations, basically primitives to retrieve data pertaining to the structure and contents of arrays, e.g. an array's rank (`dim(`*array*`)`) or its shape (`shape(`*array*`)`); a selection facility provides access to individual elements or entire subarrays: (*array*`[`*idx_vec*`]`).

Compound array operations are specified using WITH-loop expressions, SaC-specific array comprehensions:

`with { (` *lower_bound* `<=` *idx_vec* `<` *upper_bound*`) :` *expr*`;`
`}: genarray(` *shape*`,` *default*`)`

where *lower_bound* and *upper_bound* are expressions that must evaluate to integer vectors of equal length. They define a rectangular (generally multidimensional) index set. The identifier *idx_vec* represents elements of this set, similar to loop variables in FOR-loops. However, we deliberately do not define any order on these index sets. We call the specification of such an index set a *generator* and associate it with an arbitrary, potentially complex expression. By doing so we create a mapping between index vectors and values, in other words an array. As an example, consider the following WITH-loop

```
with { ([0,0] <= iv < [3,5]) : 42;
}: genarray( [3,5], 0)
```

that defines a 3×5 matrix with all elements uniformly set to 42. The scope of *idx_vec* is confined to the expression associated with the generator. It can be used to access the current index location. For example, the WITH-loop

```
with { ([0] <= iv < [5]) : iv[0];
}: genarray( [5], 0)
```

computes the vector `[0,1,2,3,4]`. Note that `iv` denotes a 1-element vector rather than a scalar. Therefore, we need to select the first (and only) element from `iv` to achieve the desired result. Actually, it is not the generator that defines the shape of the resulting array, but the first expression following the key word `genarray`. In our previous examples they always coincided with the upper bound vectors, but for example,

```
with { ([1] <= iv < [4]) : 42;
}: genarray( [5], 0)
```

computes the vector `[0,42,42,42,0]`. We still create a 5-element vector, but only the three inner elements are defined as 42 while all others are set to the *default value*, which is given by the second expression after the key word `genarray`. WITH-loops are not limited to have a single generator only. For example, the WITH-loop

```
with { ([1] <= iv < [4]) : 1;
       ([3] <= iv < [5]) : 2;
}: genarray( [6], 0)
```

defines the vector `[0,1,1,2,2,0]`. Whenever the index sets defined by the various generators are not pairwise disjoint, the order of the generators matters: in the example the array's value at index location `[3]`, which is covered by both generators is set to 2 rather than to 1. SaC actually features several variants of WITH-loops, e.g.

```
with { ([0] <= iv < [3]) : 3;
}: modarray( A)
```

Let us assume we have named the array defined by the previous WITH-loop `A`. The `modarray`-WITH-loop above then computes the vector `[3,3,3,2,2,0]`. More precisely, it computes a new array that has exactly the same shape as the existing array referred to by the expression following the key word `modarray`. The computation of those elements covered by one or more generators follows exactly the same pattern as in the case of `genarray`-WITH-loops. The remaining elements are defined by the values of the corresponding elements in the referred array.

One purpose of WITH-loops is to serve as an implementation vehicle for universally applicable array operations As a simple example, consider the definition of the vector concatenation operator `++`:

```
int[.] (++) (int[.] a, int[.] b)
{
 rshp = shape(a) + shape(b);
 res = with {([0]      <= iv < shape(a)) : a[iv];
             (shape(a) <= iv < rshp) : b[iv-shape(a)];
       }: genarray( rshp, 0)
 return( res);
}
```

We embed the WITH-loop within a function abstraction and use the built-in `shape` function to express generator boundaries in a symbolic way.

3 Solving Sudokus with SAC

Sudokus are played on a 9 by 9 board of numbers. Starting out from a board with several given numbers, the overall aim is to fill all empty positions with numbers so that the following conditions hold: (i) each row contains the numbers 1 to 9 exactly once, (ii) each column contains the numbers 1 to 9 exactly once, and (iii) each of the nine 3 by 3 sub-boards contains the numbers 1 to 9 exactly once. Although in general we may have an arbitrary number of solutions or no solution at all, all well-constructed sudokus have a unique solution.

In this section we develop a SaC program for solving sudokus. The central idea is to keep a 9 by 9 matrix of 9-element boolean vectors that represent the possible choices for each given position. We start out from an array containing `true` values only. Whenever we add a new number to the board, we eliminate all those options that are affected due to the 3 rules above, i.e., we set all corresponding positions in the same column, row and sub-matrix to `false`.

In SaC, this can be specified as a function:

```
 1  int[*], bool[*] addNumber( int i, int j, int k,
 2                             int[*] board, bool[*] opts)
 3  {
 4    board[i,j] = k;
 5    k = k-1; is = (i/3)*3; js = (j/3)*3;
 6    opts = with {
 7             ([i,j,0] <= iv <= [i,j,8]) : false;
 8             ([i,0,k] <= iv <= [i,8,k]) : false;
 9             ([0,j,k] <= iv <= [8,j,k]) : false;
10             ([is,js,k] <= iv <= [is+2,js+2,k]) : false;
11           } : modarray( opts);
12
13    return( board, opts);
14  }
```

which takes the following arguments: a position in the board specified by two integer parameters `i` and `j`, a number `k` to be placed at that position, a two-dimensional board `board` holding all numbers set so far, and a three-dimensional boolean array `opts` of options. As a result, `addNumber` returns modified versions of the board and the options which reflect the insertion of the number `k` at position `i,j`.

While the modification of the board requires only the manipulation of a single element of the board (cf. line 4), the modification of the options is expressed by a WITH-loop which spans over the lines 6 to 11. The generator in line 7 sets all options in position `i,j` to `false`, line 8 falsifies the option for the given number `k` in row `i`, and line 9 falsifies the option for the given number `k` in column `j`. Line 10 eliminates the option in the 3 by 3 sub-matrix where `i,j` is located in. Note here that the decrement of `k` is due to the fact that array indexing in SaC always starts with 0, whereas the numbers to be placed in the sudoku start with 1.

With this function at hand, after an initialisation phase which adds the pre-determined numbers, solving sudokus boils down to a search algorithm which successively adds numbers to all positions not yet filled until it eithers gets stuck or is completed. This can be specified as a function

```
int[*], bool[*] solve( int[*] board, bool[*] opts)
{
  if (! isStuck( board, opts)
      && ! isCompleted( board)) {
    i,j = findFirst( 0, board);
    mem_board = board;
    mem_opts = opts;
    for( k=1; (k<=9) && (!isCompleted( board)); k++) {
      if( mem_opts[i,j,k-1] ) {
        board, opts = addNumber( i, j, k,
                                 mem_board, mem_opts);
        board, opts = solve( board, opts);
      }
    }
  }
  return( board, opts);
}
```

It takes an actual board and an array of options as arguments and computes the first solution it finds or, if no solution exists, the board where the algorithm got stuck. At the core of this function we find a recursive call embedded into a FOR-loop which realises the back-tracking of the search. For each valid option at a given position `i,j`, we successively try to solve the given board until it is completed.

Since this, in the worst case, can lead to a 9-fold recursion for each of the numbers to be filled in, the choice of `i` and `j` directly affects the breadth of the search tree and, thus, has a vast impact on the runtime performance of the overall program. So far, we simply select the first occurrence of a zero in the board, i.e., the first empty field. In order to keep the potential need for back-tracking as small as possible, we replace the call to `findFirst` by a call of `findMinTrues( opts)` which selects a free position with a minimum number of options left.

If we want to parallelise this application[*], we can directly spot 2 potential sources for concurrency: `addNumber` and `findMinTrues` can be executed in a data-parallel fashion, and the recursive calls in `solve` can be done concurrently effectively transforming our depth-first search into a breadth-first search. While in SaC the former comes for free, i.e., it just requires multi-threaded code generation to be enabled, the latter cannot be expressed easily. Although, in principle, it is possible to use external libraries such as Pvm or Mpi this would be contrary to the declarative nature of SaC and would introduce all the well-known difficulties of handling and controlling concurrency explicitly.

We follow a different approach. We embed our

[*]It should be mentioned here that this algorithm leads to code that typically solves 9 by 9 sudokus in far less then a second. However, as sudokus can be played on any board of size $n^2 \times n^2$ parallelisation becomes essential for bigger puzzles.

SaC-program into the stream processing framework S-Net. It serves as a coordination layer on top of SaC and allows us to define which parts of the program we want to be executed concurrently in a declarative way.

4 Introducing S-Net

S-Net is a coordination language based on stream processing. It turns SaC functions into asynchronously executed, stateless stream-processing components, named *boxes*. Each box is connected to the rest of the network by two typed streams: an input stream and an output stream. Messages on these typed streams are organised as non-recursive records, i.e. label-value pairs. Labels are subdivided into *fields* and *tags*. Fields are associated with values from the SaC domain that are entirely opaque to S-Net; tags are associated with integer numbers that are accessible both on the S-Net and the SaC level. Tag labels are distinguished from field labels by angular brackets.

A box expects a record on its input stream to which it applies its associated SaC function (the box function). An S-Net box may yield multiple output records on the output stream in response to a single input record. Therefore, we cannot use the value of the function application as a result. Instead, the SaC function itself calls, potentially repeatedly, an interface function `snet_out` to produce a dynamic number of output records that are immediately sent to the output stream. As soon as the evaluation of the SaC function is complete, the S-Net box is ready to receive and process the next input record.

The functionality of a box is declared on the S-Net level by a *box signature*: a mapping from an input type to a disjunction of potential output types. For example,

```
  box foo (a,<b>) -> (c) | (c,d,<e>)
```

declares a box that expects records with a field labeled `a` and a tag labeled `b`. The box responds with an unspecified number of records that either have just a field `c` or fields `c` and `d` as well as tag `e`. The associated SaC function `foo` is supposed to be of arity two: the first argument may be of any array type while the second argument must be of type `int`. During its evaluation the function `foo` is supposed to call the interface function `snet_out` to send records to the output stream. For example,

```
  snet_out( 1, x);
```

yields a record according to the first output variant of the above type signature while

```
  snet_out( 2, x, y, 42);
```

yields a record according to the second output variant, as defined by the first argument to `snet_out`; the following arguments are used to construct the output record according to the box signature specification. In the latter case, for example, we construct a record with two fields `c` and `d` associated with the values referred to by the SaC variables `x` and `y`, respectively, and a tag `<e>` paired with the integer value 42. We use uniqueness typing in SaC to enforce a certain execution order on applications of `snet_out`.

The box signature naturally induces a *type signature.* Whereas a concrete sequence of fields and tags is essential for the proper specification of the box interface, we drop the ordering when reasoning about boxes in the S-Net domain and turn tuples of labels into sets of labels. The type signature of box `foo`, hence, is

```
  {a,<b>} -> {c} | {c,d,<e>}
```

We call the left hand side of this type mapping the *input type* and the right hand side the *output type.* To be precise, this type signature makes `foo` accept *any* input record that has *at least* field `a` and tag `<b>`, but may well contain further fields and tags. The formal foundation of this behaviour is *structural subtyping* on records: Any record type t_1 is a subtype of t_2 iff $t_2 \subseteq t_1$. This subtyping relationship extends nicely to multivariant types, e.g. the output type of box `foo`: A multivariant type x is a subtype of y if every variant $v \in x$ is a subtype of some variant $w \in y$. Again, the variant v is a subtype of w iff every label $\lambda \in v$ also appears in w.

Subtyping on input types of boxes raises the question what happens to the excess fields and tags. Subtyping relations would be satisfied if we simply ignored them. Instead, we retrieve excess fields and tags from incoming records and extend any output record produced in response to this very input record by these fields and tags, *unless* some label is already present in the output record, in which case the field or tag is discarded. We call this behaviour *flow inheritance.* Note that due to the presence of subtyping, flow inheritance is type-safe as it produces subtypes of the output type, which cannot violate type constraints. As an example let us assume the box `foo` receives a record {`a`,`<b>`,`d`}. While `a` and `<b>` are given as arguments to the associated box function, field `d` is kept by the runtime system. This is essential as the implementation of the box function is completely unaware of any potential excess fields and tags. The field `d` is attached to any output record of `foo` that follows the first output type variant; output records produced according to the second output type variant are left untouched as they already feature a field `d`.

Type inference algorithms developed for S-Net take full account of subtyping and flow inheritance, which can be dealt with statically. In conjunction record subtyping and flow inheritance prove to be indispensable when it comes to making boxes that were originally unaware of each other cooperate in a streaming network.

It is a distinguishing feature of S-Net that we do not explicitly introduce streams as objects. Instead, we use algebraic formulae to define connectivity in streaming networks. The restriction of boxes to a single input and a single output stream (SISO) is essential for this. S-Net provides 4 different network combinators: static serial and parallel composition of two networks and dynamic serial and parallel replication of a single network. These four combinators preserve the SISO property, i.e., any network, regardless of its complexity, can be used as an SISO component.

Let `A` and `B` denote two S-Net networks or boxes. Serial combination (`A..B`) constructs a new network where the output stream of `A` is directed to the input stream of `B`, and the input stream of `A` and the output stream of `B` become the input and output streams of the combined network, respectively. As a consequence, `A` and `B` operate in a pipeline mode.

Parallel combination (`(A||B)` constructs a network where all incoming records are either sent to `A` or to `B` and the resulting record streams are merged to form the overall output stream of the combined network. Each network is associated with a type signature. However, unlike box signatures they are inferred by the compiler. Network types control the flow of records in the case of parallel combination. Any incoming record is directed towards the subnetwork whose input type better matches the type of the record itself. If both branches in the streaming network match equally well, one is selected non-deterministically.

The parallel and serial combinators have their infinite counterparts: serial and parallel replicators for a single subnetwork. The serial replicator `A**(`*`type`*`)` constructs an infinite chain of replicas of `A` connected via serial combination. The chain is tapped before every replica to extract records that match the type specified as second operand. These records are merged into the overall output stream. The unfolding of the chain of networks is demand-driven.

The parallel replicator `A!!<`*`tag`*`>` also replicates network `A` infinitely far, but this time the replicas are connected in parallel. However, the best-match rule of the parallel combinator does not apply when it comes to choosing the proper replica branch for an incoming record because all replicas trivially have the same type, which would immediately lead to a non-deterministic choice. Instead, the parallel replicator comes with an additional tag specification. All incoming records must have the tag specified and the value of this tag decides to which replica a record is sent. Output records are non-deterministically merged into a single output stream, just as with the parallel combinator. While the actual number of replicas is adjusted by the runtime system on demand, it is guaranteed that any two records whose replication tags have the same (integer) value are sent to the same replica.

As pointed out before, the parallel combinator as well as the serial and parallel replicators merge the output streams of the subnetworks non-deterministically, i.e., any record produced proceeds as soon as possible. This behaviour makes it possible to write S-Net programs that adapt to the load distribution in a concurrent system. In case the order of the records in a stream is essential for the network, S-Net provides deterministic versions of all (but the serial) combinators: `|`,`*`,`!`, using single rather than double symbols.

In practice, we often see boxes that mostly or entirely serve housekeeping purposes, such as renaming, duplication or elimination of fields and tags or simple arithmetic operations on tag values. While all this can be easily accomplished using a SaC-implemented box, it is often more convenient to do this housekeeping on the S-Net level as it directly affects network construction. The construct we introduce for these purposes is called a *filter* and it looks as follows:

$$[\mathit{pattern} \to \mathit{record}_1; \mathit{record}_2; \ldots \mathit{record}_n]\,.$$

the type pattern on the left is a set of labels while each of the record specifiers on the right is a set of items of the following kinds:

- a field name occurring in the pattern: it is copied to the new record;
- $newfield = oldfield$, where *oldfield* occurs in the pattern: its value is associated with the label $newfield$ and included in the new record. The field $newfield$ may or may not occur in the pattern.
- *newtag* = *expression*, where the expression is composed from tag labels and arithmetic operators. Each tag label occurring in the expression must also occur in the pattern. The corresponding tag values are fetched, a new tag value is calculated according to the expression and the result is associated with the tag *newtag* in the new record. The initialisation of new tags is optional, tag values are set to zero by default.

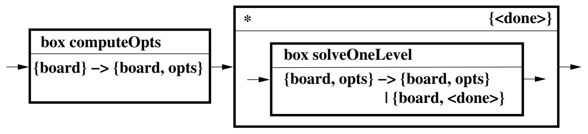
For example, the following filter consumes a record with fields a,b and the tag c and which creates two

new records: The first record has field a with the original value, field z with the same value and a tag $\langle t\rangle$ set to zero. The second record has fields b with the original value, a with the same value as b and the tag $\langle c\rangle$, whose value is incremented by 1:
`[{a,b,<c>} -> {a,z=a,<t>};`
`{b,a=b,<c>=<c>+1}]`

5 Solving Sudokus with SAC and S-Net

As we have discussed in Section 3, the pure SaC approach does not allow us to exploit concurrency between recursive calls to the sudoku solver. In this section we show how embedding the solver into an S-Net network not only enables us to execute these recursive calls asynchronously, but also to control the dynamic unfolding of processes/threads within the S-Net-layer itself.

Our first step is to shift the recursion from the SaC level to the level of S-Net. This can be achieved by transforming the recursive calls into S-Net-records and by embedding the function `solve` into an S-Net box which then serves as an argument to a serial replicator. Fig. 1 shows the



```
void solveOneLevel( int[*] board, bool[*] opts)
{
  if ( !isStuck( board, opts)
        && !isCompleted( board)) {
    i,j = findMinTrues( opts);
    mem_board = board;
    mem_opts = opts;
    for( k=1; (k<=9) && !isCompleted(board); k++) {
      if( mem_opts[i,j,k-1] ) {
        board, opts = addNumber( i, j, k,
                                 mem_board, mem_opts);
        if ( isCompleted( board)) {
          snet_out( 1, board, opts);
        } else {
          snet_out( 2, board, 0);
        }
      }
    }
  }
}
```

Figure 1. Embedding the solver in a network.

network and the modified version of the function `solve` named `solveOneLevel`. Instead of a recursive call `solveOneLevel` tries to place one further number at the selected position `i,j`. For each possible number at that position it outputs a record containing either the new board and its options or the final board and a tag `<done>`, which signals the completion of the puzzle.

For the purpose of illustration we use a graphical representation of the S-Net in Fig. 1. The SaC function `solveOneLevel` is represented as a box with type signature inscription. The box itself is embedded into a serial replicator with the termination pattern specified in the upper right corner. The replicator dynamically unfolds into a pipeline of `solveOneLevel` boxes. As soon as one of these boxes produces a record containing the tag `<done>`, that record leaves the conceptually infinite pipeline.

It should be noted here that in our sudoku example this unfolding cannot lead to pipelines longer than 81 replicas of the `solveOneLevel` box. This is due to the fact that each `solveOneLevel` box only emits a record if it can add a number to the board. Otherwise, it either emits no record at all (search is stuck) or it emits a record that contains `<done>` (solution is found). Left to the serial replicator we have another box named `computeOpts`, which takes the incoming board and realises the initialisation of the options arrays by repeatedly calling the function `addNumber` from Section 3.

If we assume that each box creates a separate process/thread the crucial question now is: to what extend do we exploit the possibility to concurrently examine different choices of the n^{th} number? If p is the number of pre-defined numbers, the n^{th} number is set by the $(n-p)^{th}$ replica of the `solveOneLevel` box. For each option k, it emits a record to the next replica. As a consequence, the $(n+1)^{th}$ number for each of these alternatives is placed sequentially. However, the placement of the $(n+2)^{th}$ number can happen concurrently with the placement of the $(n{+}1)^{th}$ number of the next alternative and so forth.

In order to be able to place the $(n+1)^{th}$ number concurrently we need to extend our network slightly. Effectively, we have to make sure that there are as many parallel replicas of the solveOneLevel box as we have options on each level, i.e., up to 9. This can be achieved by putting a parallel replicator around the solveOneLevel box within the serial replicator. Fig. 2 shows the modified S-Net. We

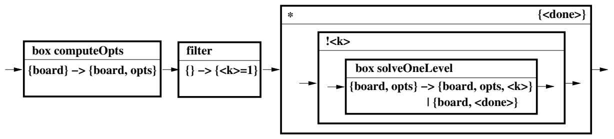


Figure 2. Refined S-Net with full unfolding.

use a new tag `<k>` for controlling the parallel splitting. Within the serial replicator, this tag can be conveniently generated by extending the output of the solveOneLevel box: whenever the board is not yet completed, we simply output the SAC-variable `k` along with the board and the options. Since `k` within each level represents the number that is being examined, this achieves the desired effect. However, we do not want to change our initialisation box `computeOpts`. Therefore, we need to insert a filter between the computeOpts box and the start operator which adds a tag `<k>` to each record. Note that the filter has the desired effect on records of the type {`board, opts`} although its fields do not occur in the filter. This is one of the benefits of the flow-inheritance of S-NET.

Another interesting feature of this network is that both replicators unfold dynamically. However, since the subsequent records with the same tag `<k>` are being processed by the same box, we know that on each stage no more than 9 replicas of the `solveOneLevel` box will be created as the value of `k` is always between 0 and 8. This guarantees a maximum of $9 \times 81 = 729$ of `solveOneLevel` boxes.

While 729 replicas of the `solveOneLevel` box might be acceptable, for bigger sudokus or in situations where we cannot derive proper upper limits for the unfoldings from the application itself, we usually want to control the unfolding of the replicators. This can be done by manipulating the control tags, in our case the tag `<k>` for the parallel unfolding and the tag `<done>` for the serial one. For example, we can control the number of parallel instances by a filter of the form

```
  {<k>} -> {<k>=<k>\%4}
```

which we put into the serial replicator in front of the parallel replicator. By using the modulo operation represented by the `%` symbol, we reduce all potential values for `<k>` to the range 0 to 3, which implicitly limits the parallel unfolding to a maximum of 4 instances.

In order to be able to control the unfolding of the serial replicator, we need to communicate the current level of unfolding, i.e., the number of numbers placed already, rather than a boolean flag indicating completion. After changing the tag `<done>` to a tag `<level>` that contains this information, we can use a more elaborate predicate for leaving the serial replicator such as {`<level>`} `| level > 40`. This leads to a situation where non-completed sudokus exit the serial replicator. Therefore, we need to link up yet another box which calls the full solver function from Section 3 resulting in the S-NET shown in Fig. 3.

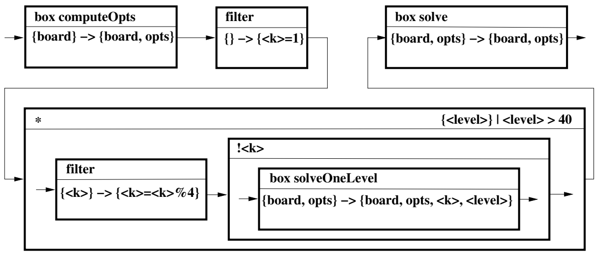


Figure 3. S-NET **with throttled unfolding.**

6 Related Work

The coordination aspect of the proposed stream processing language is related to a large body of work in so-called data-driven coordination, see [12]. Unlike most data-driven coordination languages, here we have a *complete* separation of coordination and computation. This is achieved by using SAC boxes as SISO stream transformers and treating the output purely declaratively.

The earliest related proposal, to our knowledge, is the coordination language HOPLa from the Utrecht University's Areadne project [6]. It is a Linda-like coordination language, which uses record subtyping (which they call "flexible records") in a manner similar to S-NET, but does not handle variants as we do, and has no concept of flow inheritance. Also, HOPLa has no static "wiring" and does not use type to establish a stream configuration.

Another early source to mention is the language SISAL [5], which pioneered high-performance functional array processing with stream communication. SISAL was not intended as a coordination language, though, and no attempt at the separation of communication and computation was made in it. Still it is important to acknowledge the stream variables of SISAL as an early example of task decomposition using streams.

Among more recent papers, we cite the work on the language Eden [11] as related to our effort, since it is based on the concept of stream communication. Here streams are lazy lists produced by processes defined in Haskell using a process abstraction and explicitly instantiated, which are coordinated using a functional-style coordination language. Also, like S-NET, Eden defines a connection topology for the processing entities; it however deploys the processes completely dynamically and even allows completely dynamic channels. Eden has no provision for subtyping and does not integrate topology with types.

Another recent advancement in coordination technology is the language Reo[1], whose focus is on streams but which concerns itself primarily with issues of channel and component mobility and which does not exploit static connectivity and type-theoretical tools for network analysis.

7 Conclusion

This paper demonstrates the advantages of a strict separation of concerns in using coordination as a vehicle for distributed parallel computing. The peculiarity of our approach is in the combination of an unmodified functional data-parallel language and a coordination language that cannot compute in the application domain. Put simply we use a clean computational language that cannot communicate and a clean coordination language that cannot compute but supports communication and nondeterminism. The combination in question has been exemplified by the array language SaC and the stream-based coordination language S-Net. The example application used was the search algorithm for solving sudoku puzzles, which offered an opportunity to demonstrate both data parallelism and a more general form of concurrency.

The main advantage of the separation mentioned above is the potential to reasoning about program units in isolation: while we may ignore the existence of a streaming network when reasoning about SaC components, we may likewise abstract from the computational properties of components when reasoning on the level of S-Net. The latter is helped by S-Net's highly developed type abstraction: the knowledge of a type signature is normally sufficient for reasoning about data paths and patterns of concurrency in an S-Net-encoded network; application data itself does not have to be analysed. In our example, we can control the dynamic unfolding exclusively by tag manipulations on the level of S-Net, without knowledge of communicated data fields. Furthermore, the S-Net layer provides us with features such as non-determinism which prove essential on the stream processing layer, but would immediately destroy invariant properties on the SaC level, which are essential for code optimisation. The typical downside of separating the coordination layer from the computation layer so much is the need to code in different languages and employ an intricate interface model between layers that are not aware of each other. It is our hope that interfacing on the basis of dual mapping: S-Net type signatures alongside SaC parameter tuples is convenient enough for the programmer, as would be using the housekeeping primitive `filter` in S-Net in order to adjust program units to any interface specifications that may be expected by S-Net networks.

References

[1] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, 2004.

[2] D. Cann. Retire Fortran? A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, 1992.

[3] B. Chamberlain, S.-E. Choi, E. Lewis, C. Lin, L. Snyder, and W. Weathersby. ZPL: A Machine Independent Programming Language for Parallel Computers. *IEEE Transactions on Software Engineering*, 26(3):197–211, 2000.

[4] B. Chamberlain, S. Deitz, and L. Snyder. A Comparative Study of the NAS MG Benchmark across Parallel Languages and Architectures. In *Proceedings of the ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC'00), Dallas, Texas, USA*. ACM Press and IEEE Computer Society Press, 2000.

[5] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the sisal language project. *J. Parallel Distrib. Comput.*, 10(4):349–366, 1990.

[6] G. Florijn, T. Bessamusca, and D. Greefhorst. Ariadne and hopla: Flexible coordination of collaborative processes. In *Proc. Coordination'96*. LNCS 1061, Springer-Verlag, 1996.

[7] C. Grelck. Implementing the NAS Benchmark MG in SAC. In *Proc. IPDPS'02*. IEEE Press, 2002.

[8] C. Grelck, S. Scholz, and A. Shafarenko. S-Net: A Typed Stream Processing Language. In *Proc. IFL'06*. Eötvös Loránd University, Budapest, 2006.

[9] C. Grelck and S.-B. Scholz. Towards an Efficient Functional Implementation of the NAS Benchmark FT. In *Proc. PaCT'03*, LNCS 2763. Springer-Verlag, 2003.

[10] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[11] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[12] G. A. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers*, volume 46. Academic Press, 1998.

[13] S.-B. Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[14] A. Shafarenko, S. Scholz, and C. Grelck. Streaming networks for coordinating data-parallel programs. In *Perspectives of System Informatics, (PSI'06)*, LNCS 4378. Springer-Verlag, 2006.