Parallel Programming by Transformation

Noel Winstanley

Department of Computing Science
University of Glasgow

Abstract. This paper presents a system to produce efficient implementations of parallel array-based algorithms from high-level specifications. It is structured as a transformation through a series of progressively more detailed representations. This allows the use of high-level programming features without losing the fine control of low-level languages. During the transformation process, parallel implementation decisions are introduced. Finally, a representation is reached which can be translated to C+MPI.

1 Introduction

The *Abstract Parallel Machine* (APM) methodology [OR97] is used to structure complex parallel algorithm derivations by defining parallel operations at an appropriate level of abstraction. An APM contains a set of computation sites which cooperate to implement a set of parallel operations. These operations are defined by recursive equations over the input, output and state of the sites. APM parallel operations are usually specified in the lazy functional language Haskell [PHA$^+$97].

The APMs are organised into a directed acyclic graph, where the child nodes are APMs which implement (at some lower level of abstraction) the operations provided by the the parent APM. Program derivation is by correctness-preserving transformation by equational reasoning within and between APMs in the hierarchy.

Note that the aim is not to parallelise arbitrary functional programs: rather a functional language is being used in a systematic way to model and transform imperative parallel algorithms.

Although the specifications are executable and informative, they omit many of the details required when producing a final imperative implementation. This is due to the use of high-level language features such as laziness and higher order functions, and a programming style based on garbage-collected lists rather than statically allocated arrays.

This paper presents a system to bridge the gap between the abstraction of an APM algorithm specification and the detail required for efficient implementation. We define a series of languages, where each one introduces more implementation details. The languages have APM definitions for their parallel constructs, but enforce a more imperative programming style than the arbitrary Haskell code used in higher-level specifications. This means that the languages fit into the

APM hierarchy, but are more amenable to translation to C+MPI, our target language.

Figure 1 illustrates the language sequence, where each language is represented as an oval. The first language in the sequence (*Sequential*) allows the expression of array-based algorithms; the following languages in turn: identify potential parallelism; introduce data distribution; and communication details, until at the end of the sequence a form is reached that can be translated to C+MPI.

Single Assignment C(SAC)[Sch94] – a functional variant of C with a system of extensible high-level array operations – is used as an intermediate stage in the translation to C. This simplifies the translation process, as SAC shares some of the features of the transformation languages. Furthermore, the SAC compiler is heavily optimising, producing code comparable with hand-written FORTRAN [Sch98].

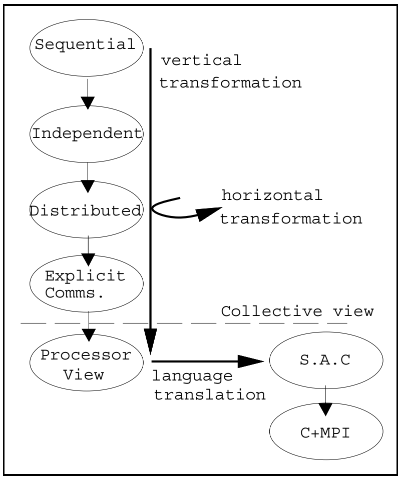

Fig. 1. Language sequence

In common with the APM methodology, derivation proceeds by transformation from one language to the next. The transformation is guided by a set of axioms. Some axioms equate constructs within a single language (*horizontal transformations*) and are used to introduce optimisations; others (*vertical transformations*) relate constructs in adjacent languages in the sequence: these introduce greater implementation detail. The series of transformations applied to a program can be used for correctness proofs and for retargetting – the transformations can be unrolled to a common point and then new machine-specific details introduced.

The languages are implemented in Haskell as libraries of combinators – i.e. as *Domain Specific Embedded Languages* [Hud98]. Doing this means that the benefits of the host language are inherited: strong typing of a rich type language; the infrastructure of compilers and tools; and a semantics that ensures the validity of equational reasoning. Importantly, the host language provides a common semantic base with which to relate constructs of the different transformation languages and more abstract APM specifications.

The next section introduces the first language in the transformation sequence. The following languages of the sequence are described in section 3, where the transformation process is informally illustrated with a small example.

2 The Sequential Language

The first language of the sequence forms a common core of algorithmic constructs to which later languages add constructs to express parallelism. The main datatype is the array, supported by a set of whole-array combinators such as those found in APL[Ive62] or SAC. User-defined structures and unions are also permitted. The update of variables is disallowed: this preserves referential transparency which simplifies the axiomatisation and use of transformations.

To ease the transformation process we must control language features that do not sit will with an imperative language, such as first-classness, laziness, higher-order functions and arbitrary recursion.

Monads [Wad92] are a standard technique for structuring computations in Haskell. The transformation languages can be said to be *monad-bound* – there is no way for the programmer to escape from the sequencing that the monad defines. This gives a fixed execution order and prevents computations relying on laziness for termination.

$$\begin{aligned}&\mathbf{for}\ (i = 0; i < B; i{+}{+})\\ &\qquad arr[i]\ =\ f(arr[i]);\end{aligned} \tag{1}$$

$$\begin{aligned}&\mathbf{seqVect}\ .\ \mathbf{for}\ (0,\ ({<}\ B),\ ({+}1))\\ &\qquad (\lambda i \rightarrow\ \mathbf{do}\ v\ \leftarrow\ arr\,!\,i\\ &\qquad\qquad\qquad f\ v)\end{aligned} \tag{2}$$

$$\mathbf{map}\ (\lambda(i, v) \rightarrow f\ v)\ arr \tag{3}$$

Fig. 2. Mapping an array

In Haskell all types are first-class – any type can be a parameter to another function or stored in a data structure. In the transformation languages, as with typical imperative languages, there is a discrimination between the type of procedures and the values that may be passed to or returned by them. A further subset of types may be stored in data structures. Extensible records [GJ96] and type classes [HHPW96] are used to encode these constraints into the type system of the host language.

In functional languages, control is described using recursion, either coded explicitly or packaged as higher-order functions. As both are problematic to translate, the transformation languages have iteration constructs similar to those found in imperative languages.

Compared to the large number of recursion combinators used by a functional language, imperative languages have a small set of iterative constructs. These are sufficient because of the use of mutable state – a for loop can be equivalent to a map, fold or scan depending on how variables are updated in the loop body.

As update is prohibited in the transformation languages, anything altered within a loop body must be explicitly propagated to the next iteration. This causes the set of loop constructs required to drastically expand. Although such explicit description of dependencies between loop iterations is useful for parallelisation, it is inconvenient for the programmer and makes a language unwieldy.

We circumvent this by expressing a loop construct as the composition of two or three combinators: one that generates a sequence of computations, possibly one which manipulates this sequence, and one that executes the sequence and return a result.

This is best illustrated by example: each blocks of code in Fig. 2 performs a map over an array. The C code (1) applies f to each element of arr. The equivalent code in our sequential language (2), uses the **for** loop generator to create a sequence of computations. The loop body is a lambda-abstraction over the loop index i. The loop executor **seqVect** executes a sequence of computations, and produces a vector of their results. Due to the functional nature of the language a new array must be created rather than updating the old array in place.

Since mapping over arrays is such a common operation, a whole array combinator **map** is provided. Equivalent code using this construct is given in (3).

Figure 3 gives another example of iteration constructs. The C code in (4) folds an array by summing the elements using an accumulating variable. A direct translation to our sequential language would produce (5). Here, the **forAccum** loop generator constructs a sequence of computations where the result of one computation is passed as a parameter to the next. The **final** loop combinator executes the sequence and returns the result of the final computation in it.

$$\begin{array}{l} sum\ =\ 0; \\ \mathbf{for}\ (i = 0; i < B; i{+}{+}) \\ \quad sum + = arr[i]; \end{array} \qquad (4)$$

$$\begin{array}{l} \mathbf{final} \\ \quad .\ \mathbf{forAccum}\ (0,\ (< B),\ (+1))\ 0 \\ \quad\quad (\lambda i\ sum \rightarrow sum\ +\ arr\ !\ i) \end{array} \qquad (5)$$

$$\begin{array}{l} \mathbf{foldSeq}\ (\lambda a\ b \rightarrow \mathbf{return}\ (a + b))\ 0 \\ \quad .\ \mathbf{for}\ (0, (< B), (+1)) \\ \quad\quad (\lambda i \rightarrow arr\ !\ i) \end{array} \qquad (6)$$

$$\mathbf{fold}\ (\lambda a\ b \rightarrow \mathbf{return}\ (a + b))\ 0\ arr \qquad (7)$$

Fig. 3. Folding an array

Alternatively, this could be expressed using the **foldSeq** loop executor (6) which combines the results of a sequence of computations using an auxiliary function. We can now substitute a **for** loop generator for the **forAccum**. This implementation had the advantage of removing the data dependency between one loop iteration and the next, introducing the possibility of parallelising the execution of loop bodies. As folding an array occurs frequently, a whole-array combinator is provided. The equivalent code for **fold** is shown in (7).

3 The Language Sequence

The language introduced in the previous section allows the description algorithms, but not their parallel behaviour.

Programs expressed in this language can be viewed as executing sequentially, or as specifying the behaviour of a parallel algorithm without committing to implementation details.

$$\begin{array}{l} a\ \leftarrow\ \mathbf{gen}\ bnd\ f \\ b\ \leftarrow\ \mathbf{fold}\ op\ v\ a \\ c\ \leftarrow\ \mathbf{map}\ g\ a \end{array}$$

Fig. 4. Sequential program

This section introduces the other languages in the transformation sequence. The program fragment in Fig. 4 will be used to illustrate how each language in the sequence introduces extra detail. The **gen** generates a new array of bounds bnd where each element is defined by f – a function from array index to element value. The array a produced is involved in two subsequent computations – a fold and a map. For conciseness, this example contains only array combinators. However, each languages has equivalent constructs for loop combinators.

During derivation, a program spends much time 'between' languages – where some code has already been vertically transformed into the next language while the remainder is still expressed in constructs of the previous language. It would be constricting to force the programmer to transform the entire program from one language to the next before the program had a defined semantics. To solve this problem, adjacent languages in the transformation sequence can be freely mixed within a program. As every transformation produces an executable intermediate program incremental development and testing is much easier.

3.1 Independent Computation

The second language in the sequence identifies potential parallelism: the machine model is an idealised parallel machine with infinite processors and no communication cost. The language introduces variants of the constructs provided by the original sequential language; these distribute computation over the idealised machine and perform implicit communication to return results to the main coordination program.

The computation performed by such a construct represents one *macro-step* in the SPMD programming methodology. Once all computations in a macro-step have completed, a result is returned to the the main program which redistributes it to the constructs which comprise the next macro-step.

As much of program as possible is transformed to use these macro-step constructs. In many algorithms there are sequential parts that cannot be parallelised; these are moved into a sequential macro-step containing one processor. The main program is now a description of the communication patterns between macro-steps, with all computation taking place on (unnamed) processors.

```
  a     ← indepGen bnd f
(b, c)  ← onProc (fold op v a)
             <|> indepMap g a
```

Fig. 5. Identifying parallelism

The code in Fig. 5 is equivalent to the previous program fragment but has been vertically transformed to the second language in the sequence by introducing parallel constructs.

The **gen** has been substituted by a **indepGen** construct. This produces an array where each element is computed independently on a separate processor. As there are no data dependencies between the fold and map, they may be computed in parallel – the <|> operator expresses this and returns a tuple of the results. The fold is sequential and so is placed on a single processor by using the **onProc** keyword, while the map can be parallelised using the **indepMap** construct. Therefore the above code has two macro-steps, with an implicit redistribution of a taking place so that the fold can be computed on a single processor.

3.2 Distributed Computation

In the previous language parallelism was introduced and the program partitioned into macro-steps. Transformation of the program to the third language maps the unbounded algorithm onto a machine with a finite number of processors.

This language has a set of parallel constructs similar to those in the previous language. However, most take an extra parameter: a data distribution that describes the distribution of the computation performed by the construct. These data distributions are an adaptation of *parameterised distribution functions* [RR95] which Rünger and Rauber use to describe the distribution of array elements amongst processors; they are capable of expressing irregular, block, cyclic and block-cyclic distributions.

Before any parallel computation can take place, the processor groups and topologies used must be defined. Figure 6 shows the block of code transformed to the third language, with processor group and virtual topology definitions added at the start of the program. The **getMainGroup** construct returns a sorted vector of the machine's processor identifiers. A new group (ag) is declared, containing all processors of the main group apart from processor 1 – which is used to compute the **fold**. A two-dimensional processor topology tp of dimensions (x, y) is then defined; this maps from two-dimensional coordinates to the processors identifiers in ag.

$$
\begin{array}{lll}
mg & \leftarrow & \mathbf{getMainGroup} \\
ag & \leftarrow & \mathbf{drop}\ 1\ mg \\
tp & \leftarrow & \mathbf{mkTopol}\ (x, y)\ ag \\
 & \vdots & \\
a & \leftarrow & \mathbf{distGen}\ (block\ tp)\ bnd\ f \\
(b, c) & \leftarrow & \mathbf{on}\ 1\ (\mathbf{fold}\ op\ v\ a) \\
 & & \quad <\!|\!>\ \mathbf{distMap}\ (block\ tp)\ g\ a
\end{array}
$$

Fig. 6. Adding data distribution

The array is now created using **distGen** and is distributed in a block-wise manner over tp. The **distMap** produces a new array with the same distribution as a. Meanwhile, the fold is performed on processor 1.

3.3 Explicit Communication

Due to scoping the results of one macro-step are available for use in later macro-steps. Therefore values resident on a processor may be accessed by a computation executing on a different processor. This non-local access means that a hidden communication is being performed. In the language implementations, array values maintain a record of the distribution of their elements. Operations that access the elements of distributed arrays check that the element is present on the local processor. If a non-local access is attempted, the program continues, but issues a warning.

The next stage of the transformation makes these hidden communications explicit by adding communication primitives to the program. Communications such as broadcast or scatter are represented as permutation functions, mapping from one data distribution to another. Applying a communication primitive to a distributed array returns a copy of the array that has the new distribution.

In Fig. 7 a **gather** collective communication has been added to the running example. This satisfies the non-local accesses of a on processor 1 by communicating to that processor all elements of the array resident on processors in ag. The other parameter to the communication, **selAll**, is a *selector* – a predicate on indexes that indicates which array elements to communicate.

$$\begin{array}{lcl} a & \leftarrow & \textbf{distGen}\ (\mathit{block}\ \mathit{tp})\ \mathit{bnd}\ f \\ a' & \leftarrow & \textbf{gather}\ 1\ \mathit{ag}\ \textbf{selAll}\ a \\ (b, c) & \leftarrow & \textbf{on}\ 1\ (\textbf{fold}\ \mathit{op}\ v\ a') \\ & & \quad <|>\ \textbf{distMap}\ (\mathit{block}\ \mathit{tp})\ g\ a \end{array}$$

Fig. 7. Explicit communication

In this case, the entire array is communicated. More subtle patterns can be expressed by using subsets of the group of processors the array is resident on or by using a different selector. There are a set of pre-defined selectors and logical connectives: using these, complex strides and halo communications can be expressed concisely.

3.4 Per-Processor View

Conventional sequential languages such as C present a *per-processor view* of the parallel machine; they describe the computation performed on each processor, but the behaviour of the entire machine is hard to ascertain. In contrast, the transformation languages presented so far give a *collective view* of the parallel machine – a program describes how the entire machine performs a computation. We believe that this has advantages over the per-processor view, as all parallelisation and distribution information is represented within the program.

The final stage in the derivation transforms the collective view program to a transformation language providing a per-processor view similar to C.

The transformation involves substituting the collective language constructs for equivalent processor-view constructs which test on the processor identifier so that only computations distributed to the current processor are executed. Likewise, the collective view communications primitives are replaced with their processor-view equivalents – i.e. MPI library calls.

Optimisations can now be performed, such as bundling similar communications together, reordering computations and partially evaluating parameters to constructs. The program is now in a form that can be straightforwardly translated to SAC+MPI which will then compile into a heavily optimised C+MPI program.

4 Conclusions & Further Work

In this paper we present a system for producing implementations from high-level parallel algorithm derivations by transformation through a series of languages.

The languages progressively introduce more implementation details until a form is reached that can be translated to C+MPI via SAC.

The languages produced are restrictive in some ways; due to their compilation method they must have an imperative flavour, but must still preserve referential transparency. Case studies in progress, taking APM specifications as their starting point, will show how troublesome this actually is.

Performing transformations by hand is tedious and error-prone. Some of the transformations are automatable: we plan to write tools to support these. However we do not aim to build a parallelising compiler – there is no requirement to automate every stage. Stages that require human insight will be supported by interactive tools.

Acknowledgements. I wish to thank my supervisor John O'Donnell, Joy Goodman, Richard Reid, Meurig Sage & Keith Sibson for helpful discussions and feedback.

References

[GJ96] Benedict R Gaster and Mark P Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science,Univerity of Nottingham, November 1996. `http://www.cd.nott.ac.uk/Department/Techreports/96-3.html`.

[HHPW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[Hud98] Paul Hudak. Modular domain specific languages and tools. In *Fifth International Conference on Software Reuse*, 1998.

[Ive62] K E Iverson. *A Programming Language*. Wiley, New York, 1962.

[OR97] John O'Donnell and Gudula Rünger. A methodology for deriving parallel programs with a family of abstract parallel machines. In *Third International EuroPar Conference*, pages 662–669, 1997.

[PHA$^+$97] John Peterson[editor], Kevin Hammond[editor], Lennart Augustsson, et al. Haskell 1.4, A non-strict, purely functional language. Report YALEU / DCS / RR-1106, Department of Computer Science, Yale University, April 1997.

[RR95] Thomas Rauber and Gudula Rünger. Parallel numerical algorithms with data distribution types. Technical Report 07-95, University of Saabrücken, 1995.

[Sch94] Sven-Bodo Scholz. Single Assignment C – functional programming using imperative style. In *IFL '94*. University of East Anglia, Norwich, UK, 1994.

[Sch98] Sven-Bodo Scholz. A case study: Effects of WITH-loop-folding on the NAS benchmark MG in SAC. In *IFL'98*. University College, London, UK., 1998.

[Wad92] Philip Wadler. The essence of functional programming (invited talk). In *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Albuquerque, New Mexico, January 19–22, 1992*, pages 1–14, New York, NY, USA, 1992. ACM Press.